Pathfinder: Interpretation of GPS data

Irving Antunes de Cerqueira Luz

Kgs. Lyngby 2005
IMM-THESIS-2005-93

Pathfinder: Interpretation of GPS data

Irving Antunes de Cerqueira Luz

Kgs. Lyngby 2005

IMM-THESIS-2005-93

ISSN -X

PREFACE

This thesis is the final work of my International MSc studies in Computer Systems Engineering in the department of Informatics and Mathematical Modeling (IMM), at the Technical University of Denmark (DTU). This thesis has been developed under the Division for Computer Science and Engineering (CSE) and Professor Paul Fischer has supervised this project.

During the development of my thesis I had the opportunity to finish my studies doing something of my interest. I developed a system applying mathematical theory, which always has been one of my favorite areas. I combined system development and mathematical theory to some of my ideas to solve a challenging problem.

I must thank my supervisor Paul Fischer, for his motivating guidance during this thesis, and the Coordinator of the International Master of Science in Computer Systems Engineering Flemming Stassen, for his guidance during all my international studies in Denmark. Some other course teachers I would like to thank, for their help and inspiration during courses of my study program, are: Jens Thyge Kristensen, Tom Østerby, Robin Sharp, Anne E. Haxthausen, Jørgen Fischer Nilsson and Michael R. Hansen.

I would especially like to thank my parents Anisio and Adelina, for the emotional and financial support during my international studies, my Danish sister Ingrid Vangkilde, for her support during my entire stay in Denmark, and my fiancee Elma Carvalho, for her emotional help during hard moments of these studies.

Kongens Lyngby, 31 January 2005

Irving Antunes de Cerqueira Luz

ABSTRACT

Global Positioning System, usually called GPS, is a satellite navigation system used for determining the precise location of an object and providing a highly accurate time reference almost anywhere on Earth. A GPS receiver is an electronic device attached to something that listens to multiple satellites and uses their information signals to determine and display the receiver's location, speed, altitude and heading. A GPS receiver decodes time signal transmissions and calculates its position by triangulation.

This thesis will show problems that GPS systems face due to lack of accuracy during interpretation of GPS data and ideas for solving these problems. This lack of accuracy may vary from system to system depending of the monitored global area, the number of accessible satellites and other factors.

In the domain of this thesis, the GPS information is going to be transmitted from a monitored mobile body to a data storing machine and converted into files to be used later. This data is going to be read by a pathfinder system whose aim is to make this information become as clean and undestandable as possible.

In this project, computational geometry concepts are going to be exploited to solve mathematical problems faced during the pathfinder development. Many ideas to solve the problems are going to be analyzed by studying their advantages and disadvantages.

Keywords: Global Positioning System (GPS), GPS receiver, triangulation, GPS data, pathfinder, computational geometry.

TABLE OF CONTENTS

1 INTRODUCTION

This project is about developing a system to construct a graphical map from a given input set of GPS motion data representing trajectories of monitored bodies. The satellite data might come from a GPS receiver attached to a vehicle, an animal or any other kind of mobile body. The road map then shows, on a user graphical interface, the streets or other kind of trails traced by such body.

The project has as objective to introduce GPS solution developments. Computational geometry [CG1] is helpful to solve these problems as they have a great geometrical aspect because of the essential data to work with (GPS data) that are information about geographical positions at fixed time intervals.

1.1 Problem description

The approach in this project is to create a graphical map from a set of given GPS-data. This information data can be disturbed by atmospheric effects and reflections by buildings or other great objects. Because of these disturbances and lack of complete accuracy, error margins have to be taken in account.

The problem with building a concise map is that we may have several lines on the map trying to represent the same path or road. Because error margins are assumed on the input data, we may have several shuffled lines. These shuffled lines would cause a confusing map and a lot of unnecessary data to manage.

Computational geometry [CG2] is a very useful tool that will be used during the solution of this problem, but processes involving computational geometry may involve too many data and heavy mathematical calculations. For instance, several points could be needed to represent a simple curved line and the more accurate you want the line to be the more points you need to represent it.

Calculations like point-to-point distances, line intersections, point-to-segment shortest distances, average points, etc, seem to be solved in few and simple mathematical operations, but sometimes they are not so simple. However, they should be very practical operations because they have to be calculated several times during the pathfinder program.

The simplest equation to find the shortest spherical distances (great circles distances [GCD]) between two points on the globe is a complex formula that involves many trigonometric functions. Now try to imagine several basic operations like this being used very frequently. Hence, one of the aims here is to look for short operations and to avoid the long ones when they are not really necessary.

Processing time is very susceptible to the used computational geometry. This means that a small code change may result in great time differences. Even though the approach in

this project is not to find optimal methods for solving the geometrical problems, I have tried to find solutions with acceptable processing time for a reasonable set of data.

As you can notice, the more data used to define paths, the more accurate they get. On the other hand, the more data used to define paths, the slower the program will get. Therefore, a balance between accuracy and computability is needed to have enough accurate data but an acceptable processing time

1.2 Project Requirements

The focus of this project is on the specification and theoretical solution of the problem. Relevant concepts have to be found and specified, e.g., path, crossing, branching, path fragment, etc. After that, more complex concepts have to be defined, e.g., path average, fragment similarity, clean path, etc.

Some methods using prior concepts need to be defined for the theoretical solutions, as finding average for similar paths, matching similar fragments of a path to another, and other relevant methods.

Conditions for the input data have to be specified to make it accessible for the defined methods. The input data has to be preprocessed to meet these conditions. The solution has to be implemented in Java language and the graphics should be implemented using the Swing library.

2 DOMAIN ANALYSIS

This chapter has the purpose of providing a better understanding of the problems that are going to be solved in this project. This is done by identifying its domain of use and all the basic terms used in a mathematical domain and finding relations among these terms.

2.1 Domain of use

The main use of the pathfinder system being developed in this project is to track mobile objects. These mobile objects can be classified according to some characteristics that are essential for the system development. For instance, it is not the same to develop a system for monitoring a car driving on the streets of a city or for monitoring an airplane performing intercontinental flies.

Therefore, these mobile objects can be classified according to their average speed, the scope of their trajectories, the kind of trajectories they use to perform and the required system accuracy for tracking them.

Commercial airplanes: The trajectories that they perform use to be quite linear, making it easier to be defined, and the required system accuracy for locating them usually isn't very high. For instance, one kilometer of distance does not represent much for airplane trajectories. On the other hand, their scope is frequently the entire globe and their normal average speeds are beyond 500 Km/h. This means that few seconds of monitoring failure might be a considerable data loss.

Maritime ships: Their trajectories are also quite linear and the average speed is considerable but not as fast as airplanes. This makes it easier to monitor ships than airplanes because the required accuracy is not very high either. The scope can also be global.

Trains: Train Trajectories are not so linear but we still could consider them linear trajectories with respect to trajectories of other mobile bodies. The required accuracy begins to get greater as well as the scope begins to be smaller with respect to prior examples. The speed may vary according to the type of train.

Animals: Of course it depends of the animal we want to monitor, but most of them may have very curved trajectories. Their scopes don't use to be (relatively) very large as well as their speeds don't use to be very high. The required accuracy may vary depending of the animal speed, size and other factors.

Cars: This is the domain used in this project. Trajectories of cars are not so curved like animal trajectories can be; anyway they can be very curved sometimes. They use to require more positioning accuracy than any other example mentioned before, especially because of the kind of traffic they face. Their scopes and average speeds are quite variable, but not excessive.

2.2 Definitions of basic terms

A common way of specifying terms is using a list of all the relevant terms to understand the domain. Some terms have strong relations among them and sometimes it is crucial to understand one concept before understanding another one. Most of the terms at this domain have to be defined mathematically, what sometimes makes the concept large or difficult to explain in few lines as in a simple term dictionary.

Other terms of this domain have similar concepts to the popular language, but here they have some constraints that popularly are not taken into account. Sometimes it can get confusing if we try to associate these terms to the frequent concepts we have in ordinary languages.

It was preferred to offer a more expansive explanation of each basic term instead of having just a term dictionary because these terms need to be well understood. Some terms will have more than one definition due to different constraints they may have. Here we try to name terms according to their constraints to avoid confusions.

2.2.1 Pathfinder program

This is the main program developed for a system to construct a graphical map from a given input set of GPS motion data representing trajectories of monitored bodies.

2.2.2 GPS path file

GPS path files are files containing input data given to the pathfinder program as the most basic data. Each of the given files represents a trajectory traced by one mobile body that is being monitored. For being considered *GPS path files* in this project, these data and files have to follow conditions defined here.

GPS path files have to be stored in a determined folder and directory to be read during the program execution. Each line of these files has to contain GPS-data about one specific geographic position, the time this information was obtained and its data precision. It makes no sense for a file to contain no lines because they try to represent the trajectory of a mobile body and if there is no trajectory then no file is needed. Each file line should have information similar to the line below:

100;0.9736992056677011;0.21888889734254205;69.2;OK

The first number in the line indicates the *time* (in seconds since midnight) when this data has been obtained; The second number indicates the mobile body position *latitude* on the Earth globe (latitude>0 at the north, latitude<0 at the south); The third number indicates the mobile body position *longitude* on the Earth globe (longitude>0 at the east,

longitude<0 at the west); The fourth number indicates the mobile body *altitude* above a reference geoid; The last information is a string that indicates this *data precision*. If the string contains 'OK' it means this data is trustful otherwise the precision may not be good and the data might not be accurate at all.

2.2.3 Points

A *point* is a set of geometrical information about a unique position on the globe. Unlike GPS-data, it will not contain other non-geometrical information as time and precision. In this context, it will have the same concept as in the mathematical context, but even in mathematics there are different point definitions depending of the kind of coordinate system that is used to identify the point. There exist systems using planar coordinates, polar coordinates, spherical coordinates, x-dimensional coordinates, etc. Here we are going to mention three kinds of points that are the most common ones for this domain. Each one has its advantages and disadvantages.

- Geographical point: This kind of point is the most accurate one to represent exact information about a position over the planet globe. The input data contained in *GPS path files* are geographical points. Each of these points has a latitude angle, a longitude angle and an altitude. In spite that this kind of points gives the most accurate positions on the globe, its coordinate system is the most complex to work with geometrically.

- Spherical point: It is simpler to work in a spherical coordinate system than to work in a geographical coordinate system because spherical points only have latitude and longitude, discarding system complexities due to altitude. On the other hand, only positions on the surface of the globe can be represented by these points.

- Planar point: The great advantage of a planar coordinate system is that it is much simpler to work in it geometrically talking. A planar point just has an x-axis coordinate and a y-axis coordinate, both representing linear distances instead of distance angles. The only disadvantage for this coordinate system is the loss of accuracy when trying to represent geographical positions from the real world, but this loss of accuracy may vary depending of the scope area and the pathfinder system itself.

A special classification for points is an endpoint. An end point is a point at some extremity of a continuous line, and if such line has an origin and a destination, we can be more specific classifying endpoints again as the line startpoint and the line endpoint respectively. This point classification is going to be used for the introduction of following terms.

Points will be represented by the lowercase letter $\boldsymbol{p}$ and identified by a subscript $\boldsymbol{i}$: p_i. The shortest distance between two points p_a and p_b will be represented by the expression $d(p_a, p_b)$. If the points are planar, this distance will be a Euclidean distance, and if the points are spherical, this distance will be an arc distance. Geographical point distances are

more ambiguous because they are a combination of Euclidean and arc distances due to the altitude.

2.2.4 Segments

A *segment* is a set of two *endpoints* (p_i and p_j) and every point on the shortest trajectory between them. All the points of a segment are planar points and the trajectory along its endpoints is a straight line.

An *arc segment* is also a set of two *endpoints* (p_i and p_j) and every point on the shortest trajectory between them, but the points of an arc segment are spherical points and consequently the trajectory along its endpoints is a curved line on a spherical surface.

Both segments and arc segments represent the approximated trajectory a mobile body has trailed from a position (point) to another. Segments and arc segments will be represented by the lowercase letter ***s*** and identified by a pair of subscripts ***i*** and ***j*** that specifies the segment endpoints: s_{ij} . To shorten future notations, the term 'segment' is going to be used to reference both *segments* and *arc segments*. The term 'arc segment' will be used only when necessary.

The *shortest distance* between a point p_a and a segment s_{bc} will be represented by the expression $d(p_a , s_{bc})$ or $d(s_{bc} , p_a)$, and the *shortest distance* between two segments s_{ab} and s_{cd} will be represented by the expression $d(s_{ab} , s_{cd})$.

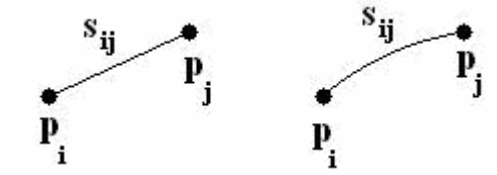

Figure 1. A segment (left) and an arc segment (right).

2.2.5 Vectors

A *vector* is a segment with a defined direction and an *arc vector* is an arc segment with a defined *direction*. Both, vectors and arc vectors, will be represented by the lowercase letter ***v*** and identified by a pair of subscripts ***i*** and ***j*** that specifies the vector segment and direction: v_{ij} . To shorten future notations, the expression 'vector' is going to be used to reference both *vectors* and *arc vectors*. The term 'arc vector' will be used only if necessary.

The subscripts ***i*** and ***j*** determine that the vector direction is from the *startpoint* p_i to the *endpoint* p_j. The segment s_{ij} and s_{ji} are the same, while the vectors v_{ij} and v_{ji} are not because they have different directions. The *shortest distance* between a point p_a and a

vector v_{bc} will be represented by the expression $d(p_a , v_{bc})$ or $d(v_{bc} , p_a)$, and the *shortest distance* between two vectors v_{ab} and v_{cd} will be represented by the expression $d(v_{ab} , v_{cd})$.

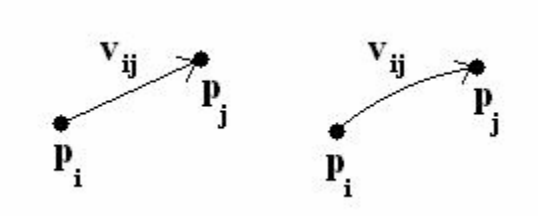

Figure 2. A vector (left) and an arc vector (right).

2.2.6 Paths

In this context, a *path* is a set of segments that represents an approximate trajectory of some monitored mobile body. These segment endpoints are going to be called *path vertices* and they are going to be used to define the *path*. An ordered list of *path vertices* (to represent a path) is going to be called *path vertex list*.

A *path* will be represented by the uppercase letter ***P*** and identified by a subscript ***x***: P_x . Each *GPS path file* is going to be converted into a *path* for making it easier to work with the input data.

The notation for a path vertex will be very similar to point notations; however, its subscript will not be just an identifier but an index number. Moreover, vertices will also have superscripts indicating the path they belong to. For instance, p_k^x represents a vertex that belongs to the path ***x*** and it is the ***k***-th element from its *path vertex list*, while p_k represents an ordinary point that not necessarily belongs to a path, and the subscript ***k*** only is a simple identifier.

Similar notations are going to be used for segments (or vectors) that belong to a specific path; therefore their endpoints must be consecutive vertices from such path. For instance, s_i^x represents a segment from the ***i***-th vertex of the path ***x*** to its consecutive vertex. Once the segment endpoints must be consecutive vertices, we don't need two subscript indexes (like for ordinary segments) but just the first endpoint index. Exactly the same notation concept is applied for vectors in case of oriented paths to be defined soon.

- Formal definitions for a path:

A *path* P_x is a sequence of segments $s_0^x , s_1^x , \ldots , s_{n-1}^x$ defined by an ordered list of *path vertices* $p_0^x , p_1^x , \ldots , p_n^x$, where p_i^x represents the *(i+1)*-th vertex from the *path vertex list*, and s_i^x represents the segment defined by the consecutive path vertices p_i^x and p_{i+1}^x .

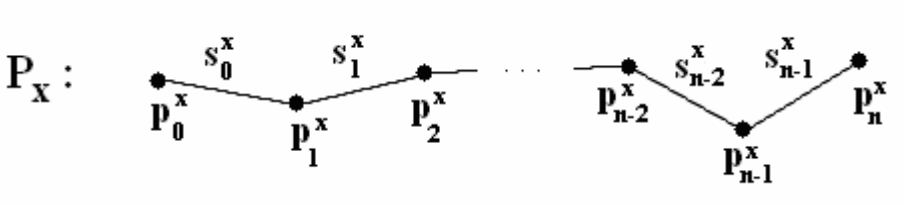

Figure 3. A path defined by its vertices and built by its segments.

A path may be defined in different ways depending on if it has an orientation or not. Sometimes it is interesting for the path to have an orientation while sometimes its orientation is completely useless.

An *oriented path* P_x is a sequence of vectors v_0^x , v_1^x , … , v_{n-1}^x, also defined by the same list p_0^x , p_1^x , … , p_n^x , where p_i^x represents the *(i+1)*-th vertex from this *path vertex list* , and v_i^x represents the vector defined by the consecutive vertices p_i^x and p_{i+1}^x.

A point is said to belong to a path P_x (oriented or not) when it belongs to a segment defined by consecutive vertices of P_x.

When a path has an *orientation***,** its first vertex p_0^x is called *path origin point* or *path origin vertex*, and its last vertex p_n^x is called *path destination point* or *path destination vertex*. In an *oriented path*, every point from the path belongs to a vector that indicates the direction along the path that should be followed from this point to reach the *path destination point*. Both *origin* and *destination points* belong to the *path vertex list*.

The path $P_{rev\,x}$ represents the path P_x with its opposite orientation. It will be called P_x's *reversed path* and will be determined by P_x's *vertex list* but in its opposite sequence.

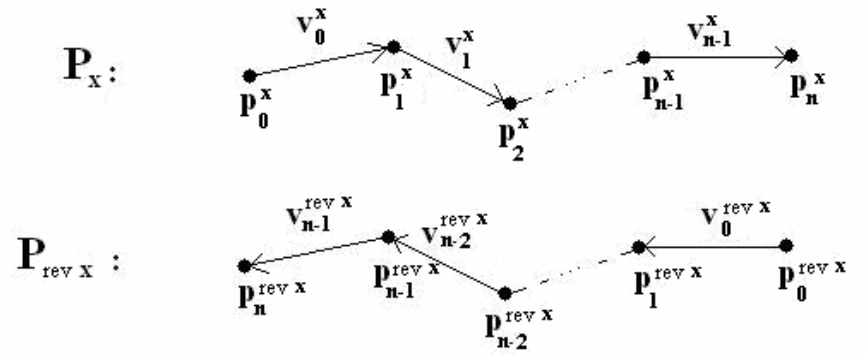

Figure 4. An oriented path built by its vectors (above) and its reversed path (below).

As we can notice, both oriented and non-oriented paths are represented by the same data structure: an ordered list of points (path vertex list). Therefore, the concept of path orientation is abstract[1]. One can decide if the link of two consecutive vertices will be

[1] Observation: The same path can be oriented and disoriented according to the convenience on a situation.

segments or vectors, although in some future definitions this abstract concept is important.

2.2.7 Intersections and loops

An *intersection* of two different paths P_x and P_y is said to occur when two segments of these different paths (s_i^x and s_j^y) intersect, that is when these two segments have at least one point that belongs to both of them. These points are called *intersection points*.

A path P_x is said to be ***self-intersecting*** if it contains at least two different egments, s_i^x and s_j^x, and either:

- s_i^x and s_j^x are not consecutive and at least one of their points belongs to both segments, or
- s_i^x and s_j^x are consecutive and more than one of their point belongs to both segments (at least one point besides their linking vertex p_j^x).

These common points from a *self-intersecting path* are called *self-intersection points*, and they are not necessarily vertices of their paths (figure 5: left example). Analogously, *intersection points* of different paths are not necessarily vertices of them.

A *loop* is said to occur in a path P_x when it is *self-intersecting* and the only *self-intersection points* of the path are its *origin* and *destination points* (figure 5: right example).

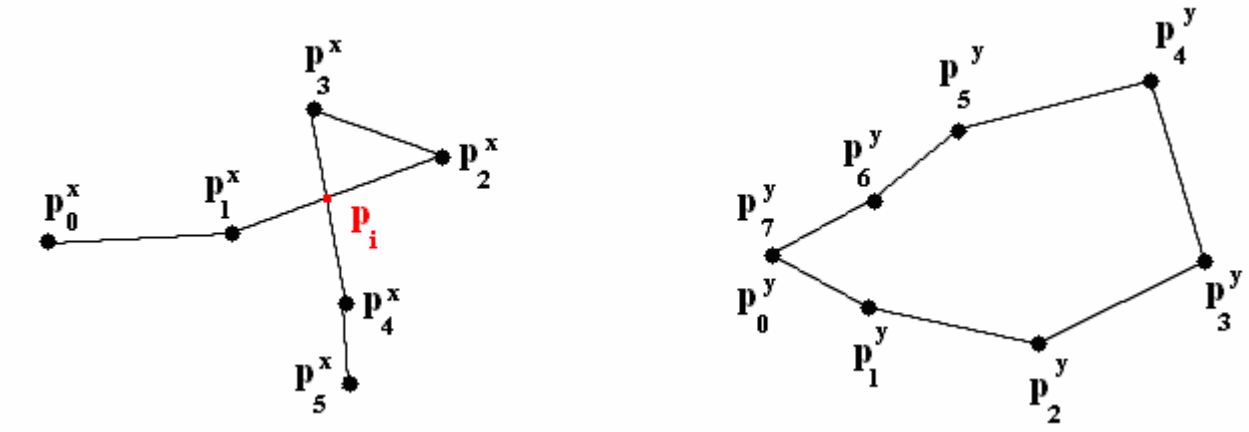

Figure 5. A self-intersecting path with its intersection point p_i (left) and a looping path (right).

2.2.8 Point, segment and vector sequences

A vector v_i^x is said to be *before* another vector v_j^x (from the same path P_x) when $\boldsymbol{i < j}$. Analogously, v_j^x is said to be *after* v_i^x.

Segments s_i^x and s_j^x or vectors v_i^x and v_j^x (from the same path P_x) are said to be *consecutive segments* or *consecutive vectors* if their indexes are consecutive numbers, as well as they are said to be the same segments or vectors if their indexes are the same ($\boldsymbol{i = j}$).

A point p_a is said to be *before* another point p_b when both of them belong to the same oriented[2] path P_x, p_a belongs to a vector v_i^x, p_b belongs to a vector v_j^x, and either:

- v_i^x is before v_j^x, or
- Both points are on the same vector ($\boldsymbol{i = j}$) but $d(p_a, p_i^x) < d(p_b, p_i^x)$.

Analogously, the point p_b is said to be *after* p_a.

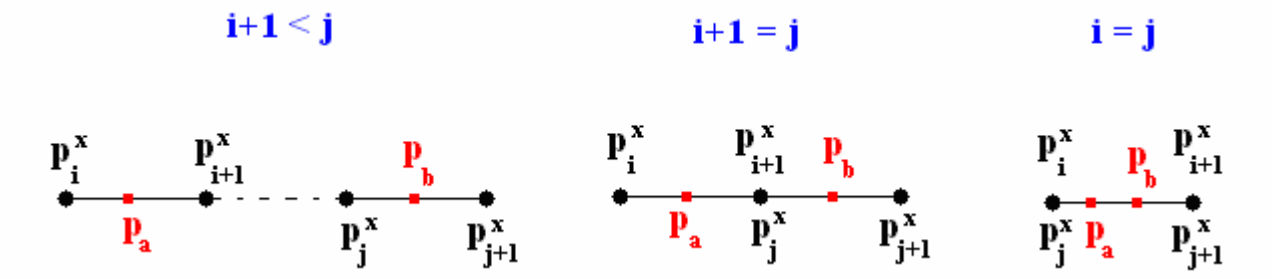

Figure 6. Examples of a point p_a before another point p_b from the same path P_x when they are in separate segments (left), in consecutive segments (middle), or in the same segment (right). At the same time, the point p_b is after p_a.

A point p_b is said to be *between* two other points p_a and p_c when they belong to the same oriented path P_x, and p_b is *after* p_a but *before* p_c[3].

---

[2] Observation: Even a non-oriented path has an ordered *vertex list*, and a point from this path may also be said to be *before* or *after* another one with respect to its list. To check if a point is *before* or *after* another one from a non-oriented path, just orient the path (considering the first vertex in the list as origin point, the last vertex as destination point, and vectors instead of segments) and follow the same concepts.

[3] Observation: One point can be after and before itself in a self-intersecting or looping path, as well as a set of points can be both after and before a unique self-intersection point. In the left example of the figure 5, the vertices p_2^x and p_3^x are after and before the intersection point p_i of the self-intersecting path P_x. In the looping path P_y at the right example of the same figure, all the points are after and before p_0^y and p_7^y, including themselves.

2.2.9 Path fragments

A *path fragment* is a subset of consecutive points from a determined path P_x, and is represented by an ordered list of vertices called *fragment vertex list*. This list will contain the following elements:

- A point p_o from the path P_x (not necessarily a path vertex), which will be the first element of the *fragment vertex list* and is going to be called the *fragment origin point*.

- A point p_d, not before p_o, in the path P_x (not necessarily a path vertex), which will be the last element of the *fragment vertex list* and is going to be called the *fragment destination point*[4].

- All P_x's vertices that are between p_o and p_d in the path P_x, which will be in the *fragment vertex list* in the same order that they were in P_x's *vertex list*.

A *path fragment* will be represented by the lowercase letter ***f*** and identified by subscripts and/or superscripts. Fragment notations may have few changes depending on which of its information is more important to manage, and then its subscript or superscript function will be different for each situation.

One subscript just gives a name for the fragment (e.g. the notation f_k is just identifies the fragment), but a pair of subscripts represent information about the *fragment origin* and *destination points* (e.g. the fragment f_{od} begins at the point p_o and ends at the point p_d).

One superscript represents just the path the fragment comes from (e.g. the fragment f^x is from the path P_x), but if it also contain a parenthesis with two indexes then it defines from which vector to which vector (from the path) the fragment goes (e.g. the fragment $f^{x(i,j)}$ goes from the vector v_i^x to the vector v_j^x).

We may use both subscripts and superscripts if we need full information about a fragment (e.g. the fragment $f_{od}^{x(i,j)}$ belongs to the path P_x and goes from the point p_o on the vector v_i^x to the point p_d on the vector v_j^x.

Path fragments have the same orientation as their original paths. This orientation can also be considered or not depending of the convenience. The path fragment $f^{rev\,x}$ will be called the *reversed fragment* of f^x and it will be represented by its same *fragment vertex list* but with an opposite sequence of vertices.

After fragments have been defined, they can be treated as independent paths once they also are represented by an ordered list of vertices.

[4] Note that the *fragment origin point* may be equal to the *fragment destination point* if the *path fragment* is a single point.

2.2.10 Path map

The *path map* of this system is a set of paths drawn on a two dimensional graphical surface, representing the trajectories of monitored bodies on a certain area. A path will be represented in the *path map* as a set of consecutive segment lines.

2.2.11 Map graph and nodes

A *Map graph* is a data structure used in this system to represent all paths created from previous obtained data and processes. The aim of a *map graph* is to find path relationships in a process for defining which paths are going to become part of the *path map*.

There exist two kinds of *nodes* to be defined here and they should not be confused. There are *nodes* that just represent paths in a *map graph*, and these nodes are going to be called *graph nodes*.

The other kind of nodes represents special points that link (two or more) different paths, i.e. concatenations, branchings, crossings, etc. The expressions of concatenations, branchings, and crossings are going to be defined later in the report. These nodes are going to be called *linking nodes* and they must have an especial treatment. *Linking nodes* should inform their positions, every path they link, besides other details.

2.3 Project aim

Many terms still have to be defined but they cannot be defined here yet because they are part of problems or solutions explained during the analysis design or because they need to be defined after some of these explanations.

For instance, the required definitions for branchings and crossings are quite ambiguous in some situations and difficult even for humans to define if they occur in a set of shuffled lines to be averaged, or not.

Actually, the aim of this pathfinder system is not just to obtain paths and throw them in a graphical map but to create a concise map containing comprehensible data to human eyes. Moreover, it has to be a computable system that discards unnecessary data that only would cause processing delays and system troubles.

Therefore, some processes are necessary to avoid unnecessary and confusing data. In this project we mention five necessary processes for reaching the aim of the pathfinder system: 1) Cleaning unnecessary points, 2) Detecting similarities between paths, 3) Averaging the similar ones, 4) Averaging similar path fragments, and 5) Stitching path extremes.

All of these processes are going to be explained further in this report, but not all of them have been deeply analyzed and implemented. The first three processes have been the most elaborated ones. Anyway, all these processes have been studied and mentioned further.

2.4 Relationship diagram

The following ER-model has as goal to give a graphical overview of the domain concepts and the relations among them. Every entity class and entity property in the following diagrams has been mentioned in the domain analysis.

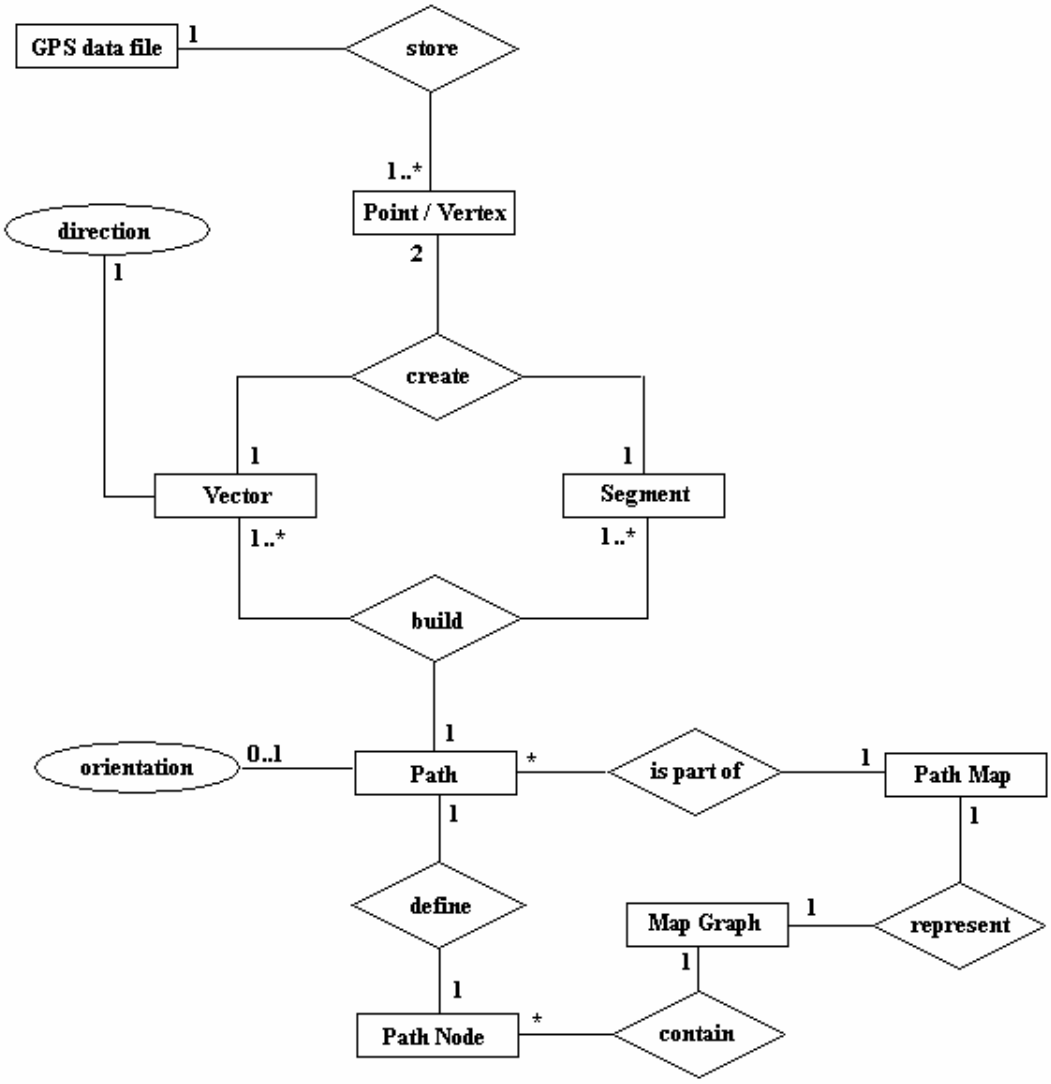

Figure 7. Relationship Diagram for the project.

3 SYSTEM ANALYSIS AND DESIGN

The single terms to understand the problem were already explained in the last chapter. In this chapter, the essential processes for reaching this project aim are going to be studied. More elaborated terms used for the solutions are also going to be defined here.

There are five fundamental processes for building a concise map from the given input data. The first one is the ***cleaning*** process and its goal is to discard unnecessary data facilitating future processes and saving processing time and computer memory. The second fundamental process is the ***similarity detection*** that finds pair of paths (or path fragments) that deserves to be replaced by just one average path. The third process is the ***path averaging*** process that finds the best path to replace two other similar paths with. This process is very important for having concise paths on the map. The fourth process of ***fragment averaging*** is a complement for the second and third processes but it works with path fragments instead of entire paths. The last procedure is called the ***stitching*** process and its goal is to connect some of the concise paths that deserve to have a permanent link. These fundamental procedures are going to be explained next.

3.1 Cleaning process

One of the objectives in this project is to manage inaccurate input information to create a concise map from it. This inaccuracy is liable to error margins that are going to be explained. It is tried to avoid confusing situations that make good data management difficult, e.g. several shuffled lines on the map that should represent the same road.

Another objective is to discard unnecessary data for avoiding excess of processing time due to useless extra calculations. To discuss about these two main objectives, we need to define some concepts before we use this process to reach them.

3.1.1 The EMD, PSD and PPW distances

The GPS-data, received as input, is not supposed to be very accurate information because of the satellite limitations and the disturbances that their signals may suffer. Normally, there is an error margin distance between a given GPS data position and the real geographic position it represents. This distance is relative to some factors, e.g. the number of used satellites. Hence, a constant will be used in the pathfinder program to represent such 'Error Margin Distance' ***EMD***.

Due to this error margin, two real equal trajectories would probably never be represented in exactly the same paths, but if they are quite close one to another they could be considered the same. There is a distance used to check if two paths are close enough to be considered the same one or not. This distance is also relative to some factors, e.g. the *EMD* distance and road standard widths. Therefore another constant will be used to

represent this 'path similarity distance' *PSD*. Two points are going to be called ***close enough*** if their distances are lower than or equal to the constant distance ***PSD***.

As it has been mentioned above, the PSD depends of how wide the used roads or paths can be. This width difference may be considerable if we are talking about boats on rivers, trains on rails, cars on motorways, airplanes on air lanes, bicycles on bike roads, animals on jungle tracks, etc. Therefore, we have to consider a 'path pattern width' ***PPW*** that our monitored object is going to use.

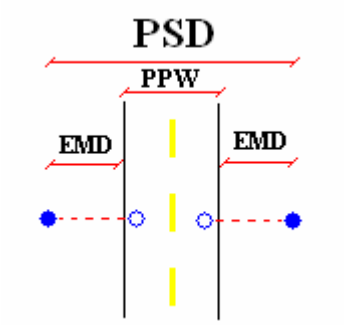

Figure 8. A road and the relationship between the constants EMD, PPW and PSD.

In the criterion used for this project, the best relation of these distances is the equation below (check the figure 8):

$$PSD = PPW + 2 \cdot EMD$$

This relation may be changed once PSD, PPW and EMD are independent constants defined in the program, but this relation ensures that two points from the same path are going to be considered as from the same path.

3.1.2 Vertex Fluctuation

Two consecutive geographical points sent by a GPS-receiver should be exactly the same if the monitored object was stopped during the time interval between these two data receptions. Due to an error margin, probably these two points would be very near but not the same. The points sent by the GPS-receiver are converted into path vertices. A *vertex fluctuation* happens when consecutive vertices actually are different just because of an error margin. These consecutive similar vertices are called ***fluctuation vertices***. In more formal definitions,

- Given a path P_x, two of its vertices p_i^x and p_j^x are *fluctuation vertices* if they are consecutive ($i = j \pm 1$), and the distance between such vertices is not bigger than the error margin distance, $d(p_i^x, p_j^x) \leq EMD$.

- A *vertex fluctuation* is said to occur in a path or path fragment where at least two of its vertices are *fluctuation vertices*.

- A *fluctuation sequence* is a set of vertices from the same path where every pair of consecutive vertices of this set causes a *vertex fluctuation* in the path.

- A *complete fluctuation sequence* is a *fluctuation sequence* where, given the last sequence vertex and its next vertex, they are not *fluctuation vertices*, as well as the first sequence vertex and its prior vertex.

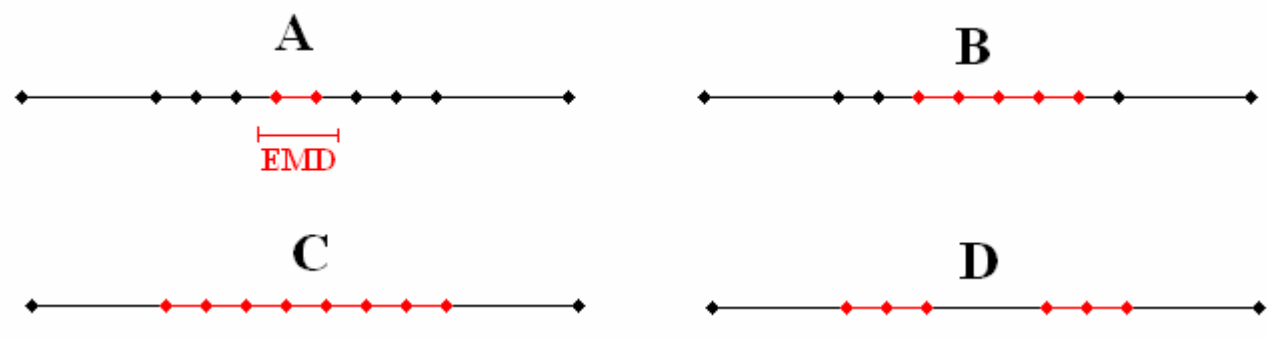

Figure 9. Examples are painted in red. Example A: two *fluctuation vertices* among many *vertex fluctuations*. Example B: a *fluctuation sequence* (that is not complete). Example C: a *complete fluctuation sequence*. Example D: two separated *complete fluctuation sequences*.

The shorter the data reception interval is the fewer possibilities of different trajectories exist along a short distance between two consecutive received points (especially if the distance is too short: lower than or equal to *EMD*). On the other hand, the longer the data reception interval is the less probable two consecutive non-fluctuating vertices are very near, but even if a coincidence existed we cannot predict what happened during a long data reception interval. So, Even if the monitored object were in (fast or slow) movement, two very near consecutive received points with a distance lower than or equal to *EMD* are "also going to be considered *fluctuation vertices*", supposing that nothing considerable happened along the trajectory between these points.

The altitude distance is not going to be considered here because the map is not 3D, so, the altitude is not going to help our 2D map to be more concise or more accurate; it would only raise difficulties and slow the calculations.

3.1.3 Self-similarity

Until now, we have talked about an averaging process and its aim of finding a path to substitute other two different but similar paths. We also mentioned the possibility of paths to be not similar but have similar fragments to be averaged. But we didn't even mention the possibility of the same path to have similar fragments.

Definitions for similarity are not so simple, and then we are not going to define *path similarity* nor *fragment similarity* in this subchapter for the *cleaning* process. We are going to let it be explained in the next subchapter for the *similarity detection* process.

The only thing we need to know here is that similarities may happen when paths are very close to each other and it seems that they represent the same trajectory. The same happens for path fragments, i.e. fragments from a same path may also be similar causing what we call *path* ***self-similarity***[5]. Once the term '*similarity*' is explained further we are going to be able to understand perfectly what *self-similarity* means.

An example of *self-similarity* is a situation where a mobile object passes by the same point during just one trajectory. It causes a path *self-similarity* that would be very short if it was just a crossing[6], or relatively long if a road was used more than once. Self-similarities may cause several problems to future processes and we may try to avoid them already in this cleaning process.

3.1.4 Cleaning types

A unique path may have *fluctuation vertices*, *self-similarities*, *self-intersections*, *loops*, etc, having many unnecessary data and troublesome path fragments. A cleaning process is the first fundamental process that is applied to already existing paths or to input *GPS data files* for transforming them in consistent paths without these useless confusions. These consistent paths are called *clean paths*.

There are different types of *cleanings* depending on each of the prior mentioned problems we want to avoid for *clean paths*. For instance, the *simple path cleaning* only cares about discarding completely unnecessary data.

- **A *clean path* does not have excess of unnecessary vertices after a *simple path cleaning*.**

The next type of cleaning discards all those vertices that do not increase much the path accuracy but decrease the processing speed. This *basic path cleaning* fills the following constraints (to be explained later):

- **A *clean path* does not have *non-OK vertices* after a *basic path cleaning*.**
- **A *clean path* does not have very distant consecutive vertices after a *basic path cleaning*.**
- **A *clean path* does not have *fluctuation vertices* after a *basic path cleaning*.**

A *general path cleaning* follows the same constraints than a *basic cleaning*, but one more constraint is added to this cleaning process. This constraint is not as easy to reach as the past ones because of possible confusions caused by '*immediate returns*' (that is also going to be explained later).

- A *clean path* does not have *non-OK vertices* after a *general path cleaning*.

[5] The term 'self-similarity' is going to be explained better in the subchapter of *total path cleaning* (3.1.4.4)
[6] The term 'crossing' mentioned here is the same used popular term, once this is an empiric example.

- A *clean path* does not have very distant consecutive vertices after a *general path cleaning*.
- A *clean path* does not have *fluctuation vertices* after a *general path cleaning*.
- **A *clean path* does not have *immediate returns* after a *general path cleaning*.**

A *total path cleaning* adds two more constraints to the *general cleaning*. These constraints may increase considerably the cleaning processing time, but at least we may use the same processing time to reach both additional constraints.

- A *clean path* does not have *non-OK vertices* after a *total path cleaning*.
- A *clean path* does not have very distant consecutive vertices after a *total path cleaning*.
- A *clean path* does not have *fluctuation vertices* after a *total path cleaning*.
- A *clean path* does not have *immediate returns* after a *total path cleaning*.
- **A *clean path* does not have *self-intersections* (or *loops*) after a *total path cleaning*.**
- **A *clean path* does not have *self-similarities* after a *total path cleaning*.**

Although the last constraints may be expensive in time, if detections for self-similarities and self-intersections are not achieved here, they will have to be achieved later; otherwise the map might be not concise. On the other hand, if we achieve these constraints here, we would simplify future fundamental processes and save time later.

Next, the reasons of each mentioned constraint of path cleaning procedures, and how they are achieved, are going to be explained.

3.1.4.1 Basic path cleaning

Here, each necessary constraint for the *basic path cleaning* is going to be explained. The *Simple path cleaning* should be explained first because it includes part of the *basic path cleaning* procedure, but due to practical reasons, *basic path cleaning* constraints are going to be explained first.

3.1.4.1.1 A clean path does not have non-OK vertices after a basic path cleaning.

Non-Ok points *or non-OK* vertices are those GPS-data whose last item (that indicates the data precision) does not contain the 'OK' substring. Hence, these data are not very trustful, so they do not increase the accuracy, but they include vertices in the path increasing calculations and decreasing the processing speed (the other vertices are called ***OK points*** or *OK vertices*).

At the beginning there were some ideas about what to do with these insecure signals, but we never know how wrong these data could be and they are undesired during time-expensive procedures.

Instead of finding what to do with these points, a *strict elimination* of the *non-OK* points is possible, splitting the path each time the elimination happened (due to the lack of these eliminated points without further vertex connections). Although we may have many split paths, we would ensure the elimination of false trajectories.

A *non-strict cleaning* of *non-OK* points would ignore them when transferring GPS data from a file to a path structure but without splitting the path, by linking the prior and next vertices of the ignored *non-OK* point.

A path segment can get very long due to consecutive eliminated *non-OK* points in a *non-strict* cleaning (where the path is not split), and we don't know what happened during this long trajectory. This problem will be discussed in the next constraint about distant consecutive vertices.

3.1.4.1.2 A clean path does not have very distant consecutive vertices after a basic path cleaning.

Too distant consecutive vertices may be found due to three reasons: data failure, excessive speed of the monitored object body, or discarded *non-OK* vertices. Data failures and excessive speed (higher than the expected one) don't use to be frequent situations and they are quite improbable. Discarded non-OK vertices are going to be more frequent and they might cause quite long path segments if these discarded vertices are consecutive ones in a *non-strict* cleaning.

A path full of long segments could be a very nonsense data when compared to the real path. We may assume a constant for a maximal segment length ***MSL***. We could use a maximal considerable speed, the error margin, and the signal reception time interval as parameters to consider this constant value. For instance, if the signal reception time interval is 10 seconds, the error margin is about 10 meters and the maximal considerable speed is of 200 Km/h, then our *MSL* constant should be about 575 meters:

$$speed \cdot interval + 2 \cdot error = \frac{200 \cdot 1000m}{3600\text{sec}} \cdot 10\text{sec} + 2 \cdot 10m \approx 575m$$

The *MSL* is just a constant defined initially and we may adapt it to the best value according to the system domain. It could even have an infinite value, if we want to consider any segment, independent of its length.

It is a good idea, though, to split a path into two, if there is a segment in it that is longer than a defined *MSL* value. The new paths will not lose any useful information from the original path file; moreover they will not use the very insecure information from the long segment.

A very long segment would probably not improve a path average, but would instead disturb a good one. We should discard the long segment and hope another similar

trajectory to happen with more *OK vertices* and consecutive short distances, resulting in a more accurate path.

However, non-OK vertices could happen everytime in a same trail because of certain given circumstances, for instance a long tunnel. It could be a good idea to represent some kind of link on the map for distant consecutive OK vertices (e.g. points with the same color may represent tunnel endpoints or other kind of endpoints). But it is not a good idea to represent such links with a straight segment possibly faking the trail between them. The possible fake is the reason why it was decided to split the paths when very long segments happens (figure 10).

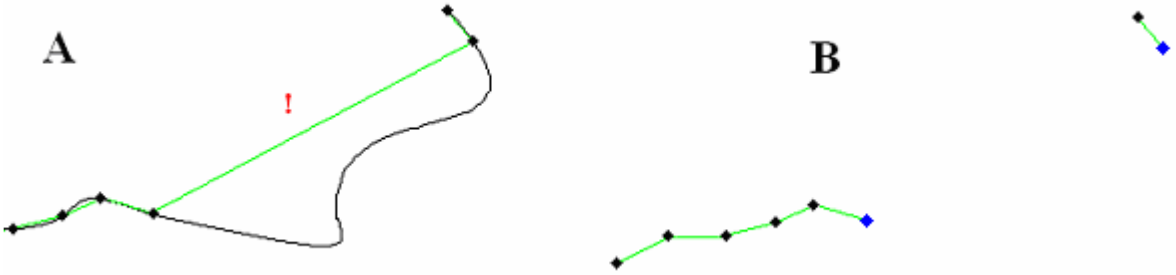

Figure 10. A trajectory where the black vertices are received data and the green segments are the supposed approximated trails between them. Example A: One long segment happened because of discarded consecutive non-OK vertices. Example B: The path was split in two, ignoring the long segment.

3.1.4.1.3 A clean path does not have fluctuation vertices after a basic path cleaning.

Fluctuation vertices do not help a map to be more concise, the map accuracy does not increase significantly due to them, and many times the map accuracy even decreases because of them. Moreover, they use to be several useless data that causes a considerable delay in many steps of the program.

It is always a good idea to eliminate *fluctuation vertices* from every path and to ignore input GPS data that cause these vertices. If two consecutive vertices have quite short distance between them, they are *fluctuation vertices* and probably try to represent the same position on the map.

Perhaps the very last supposition is not correct, because the proximity of consecutive vertices may be due to a coincidence in case that a significant trajectory happened during these near consecutive vertices (figure 11). But these very near vertices will not help us to predict what happened during this lack of information, and then they will not make the map much more accurate. So, (as it already has been said) it was decided to consider every pair of vertices whose distance between them is lower than the error margin distance *EMD* as *fluctuation vertices*.

The decrease of accuracy will not be significant if we discard *fluctuation vertices*, but later we may save time due to this operation. The question here is “how to eliminate the *fluctuation vertices*”. We should not just ignore all of them; otherwise we are going to lose important data. Each eliminated *fluctuation sequence* should be represented in the

clean path by one vertex. There is more than one kind of fluctuation cleanings. Next we explain some of them.

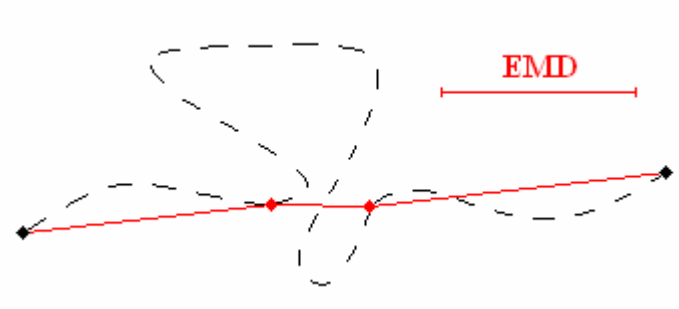

Figure 11. A real trajectory is represented by the black dotted line. The obtained GPS data are the black and red points, and they will be converted into vertices of a path (represented by the red straight lines) that tries to predict an approximated trajectory (obs: in this case the approximation is not very successful). There are two *fluctuation vertices* (red points) in this example and they don't help us to predict the real trajectory, therefore, if we replace them by just one average vertex, we are not going to lose significant information due to this replacement.

Fluctuation cleaning by average:

The first prupposed method for cleaning fluctuation vertices is: "If the distance between two consecutive vertices is too small (smaller than or equal to *EMD*) then an average point between them will be found to substitute both vertices by only one". But this method could not work in some situations. For example, if the mobile object traveled at a very slow speed (e.g. a car in a traffic jam) then all the vertices of a long path could be converted at the end to a unique average vertex! (example A in the figure 12). For making this possibility more improbable, I decided to work with *fluctuation sets* instead of working with pairs of *fluctuation vertices*.

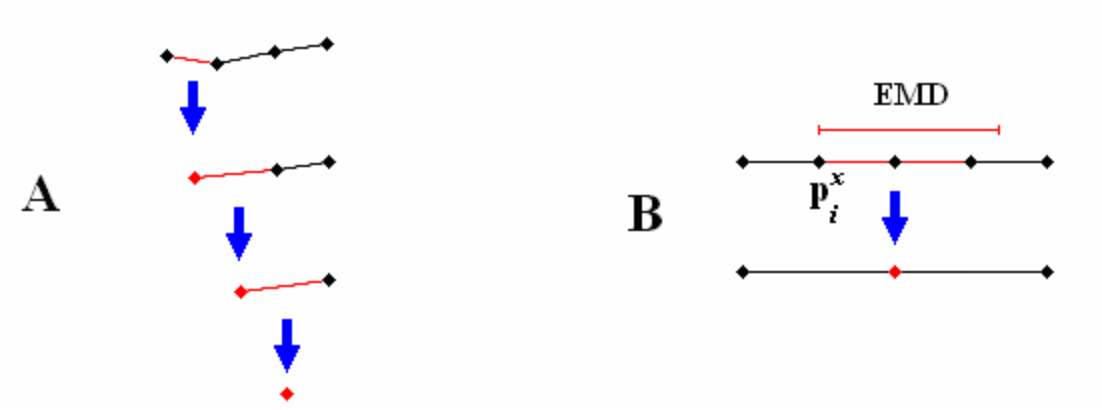

Figure 12. Example A: If we convert a pair of *fluctuation vertices* (linked by red lines) in an average vertex (represented by red points) and use this average vertex again to next averages, we may convert an entire path in a unique vertex. Example B: The *fluctuation set* from a vertex p_i^x (represented by points linked by red lines) is converted into an average vertex (represented by a red point).

The *fluctuation set* from a determined vertex p_i^x is a set of consecutive *fluctuation vertices*, composed by p_i^x itself and all the consecutive vertices after it whose distance

to p_i^x is lower than or equal to the error margin distance *EMD* (example B in the figure 12).

In other words, this set begins on a determined vertex and it includes all posterior consecutive vertices along the path whose distances to the given first vertex are not greater than *EMD* [7].

However, the new concept of *fluctuation set* does not avoid the possibility of converting the entire path in a single vertex during their substitutions (example A in the figure 13). It can still happen in traffic jam, for example.

To avoid this undesired possibility, we should not use average vertices to find the next *fluctuation vertices* or *fluctuation sets* to be averaged in this procedure, and then this problem would be solved. But if we don't use the average vertices for future fluctuation searches then the resulting path may still have *fluctuation vertices* at the end of this cleaning procedure (example B in the figure 13). So, the *vertex fluctuation* problem could still persist if we don't reuse average vertices, thus the main aim of this process would not be reached.

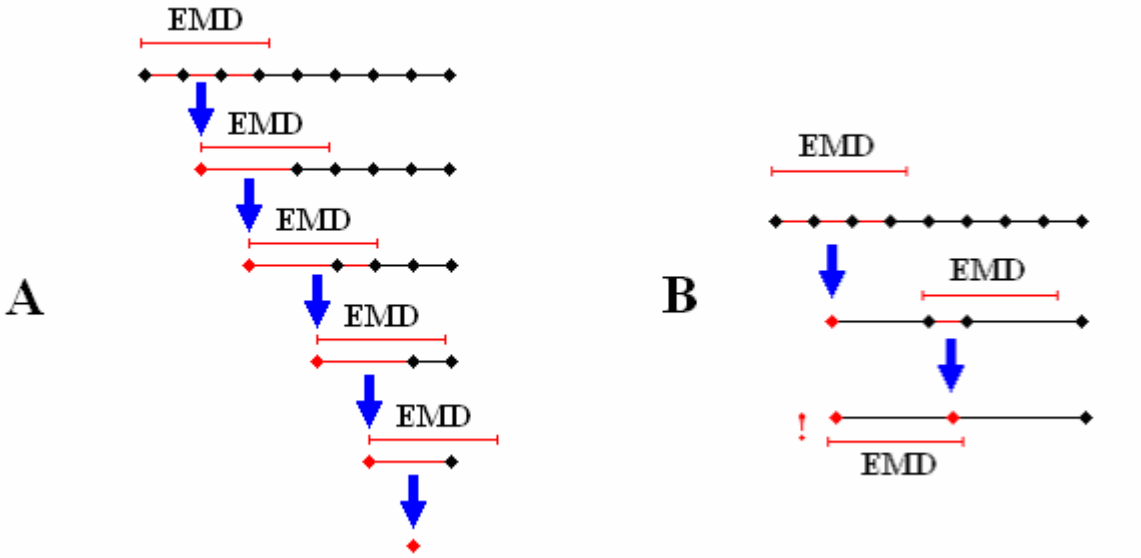

Figure 13. Example A: The path can still be converted to a single vertex even though *fluctuation sets* are used to find average vertices. Example B: The way for avoiding the problem on the example A is to use not the average vertex for the next *fluctuation set*, but then we may keep having *fluctuation vertices* in the resulting path, as we can notice in this example.

Therefore, if we don't reuse average vertices, the path would have to be checked again after it ends each time a new *vertex fluctuation* was found during the procedure (for guaranteeing the aim approach in this cleaning). The procedure might have to be repeated many times, but at least the prior undesired situation would not happen anymore.

[7] Observation: Even though the maximal distance among its vertices is the *EMD*, a *fluctuation set* can be long, e.g. a spiral path. A *fluctuation set* can also be unitary (have only one vertex) if no other vertex in the path is near to the determining vertex of the set.

Our question now is “how many times might we have to repeat this process? And, how expensive (in time) could it be if we have to repeat it many times for several paths?!”

If we are going to reuse average vertices, the *fluctuation sets* would be a very practical tool to diminish the number of necessary repetitions, because working with them would help to finish the *vertex fluctuations* much faster than working with pairs of *fluctuation vertices*. Even though *fluctuation sets* make several repetitions less probable, They don’t make them impossible.

In the next type of path cleaning, another possible solution (that also can be used for *basic path cleanings*) for discarding fluctuation vertices will be explained and its advantages and disadvantages will be discussed.

3.1.4.2 Simple path cleaning

The prior explained cleaning type is applied only for paths created from input data. The *simple path cleaning* can also be applied for already existing paths, e.g. paths resulting from further procedures. But this type of path cleaning has one constraint that extends the last constraint of the *basic path cleaning*.

However, the solution to be explained next for the only constraint of the *simple path cleanings* can also be used for the *basic path cleaning*, and is actually the suggested solution for the problems of fluctuation cleaning mentioned before.

3.1.4.2.1 A clean path does not have excess of unnecessary vertices after a simple path cleaning.

The most common vertices that cause excess of vertices in paths are *fluctuation vertices*, so we must begin this procedure also cleaning them.

Is there a procedure that can discard every consecutive *vertex fluctuation* without needing algorithm repetitions? The answer is ‘yes’, and here we have a suggestion for this *simple path cleaning* or *simple fluctuation cleaning*. Even though this procedure might lightly diminish the path accuracy, it would not diminish it significantly.

The *fluctuation sets* would be a useful tool again, but we don’t need to find their average vertices anymore. That means we will avoid problems due to averages. Only the first vertex of a *fluctuation set* will be used instead of its average vertex. The average calculation is not necessary anymore, and the first vertex from the *fluctuation set* is the only data from it that is necessary to be stored during this procedure. The other vertices from the set (if there are aby) are just garbage to be ignored.

Using only the first vertex (by ignoring the other fluctuating vertices from the set) instead of the using the average vertex for replacing the entire *fluctuation set* is the reason of the

possible accuracy loss, but we must analize if this simpler procedure would be efficient. How low an accuracy would be acceptable? How much would this change improve the processing time?

One thing is for sure: the distance between the first vertex and the average vertex from the same *fluctuation set* cannot be greater than *EMD* because no distance between vertices from a *fluctuation set* can be greater than that. Therefore, even though it is almost sure that the accuracy would diminish, it can not diminish significantly. Moreover, this accuracy loss would be inversely proportional to the error margin; hence, the lower the *EMD* distance is the more insignificant the accuracy loss is.

The great advantage of this algorithm is that we avoid unnecessary repetitions. Nevertheless, it is guaranteed that the discarding of every consecutive *fluctuation vertices* is achieved, because the distance from the first vertex of a *fluctuation set* to the first vertex of the next *fluctuation set* must never be lower than the *EMD* (figure 14).

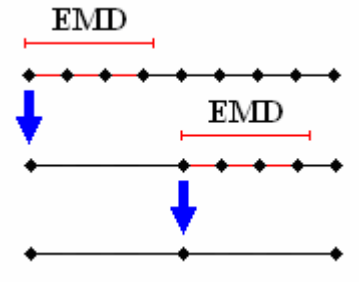

Figure 14. If the first vertex from each *fluctuation set* is used to replace it, *fluctuation vertices* will disappear without needing repetitions (this figure is not a good example for showing accuracy loss because the path keeps an absolute direction).

A strict simple cleaning should not only care about *fluctuation vertices* but also about vertices that are not helpful for the path accuracy, so their absence in the path would not cause any consequence. The elimination of these verticess would avoid excess of vertices in paths even if these excessive vertices were not *fluctuation vertices*.

Those unhelpful vertices are detected by checking how much distances between vertices vary whether an intermediate vertex is removed. If no significant variance happens, definitely the vertex is unnecessary (left example at figure 15).

We must be aware that a single elimination of a vertex might not vary significantly the distance from its prior to its next vertex, but consecutive eliminations might do it discretely. So, when consecutive eliminations happen, we have to check not only the distances from the prior to the next vertex of an eliminated one, but the entire variance of the distance after the elimination sequence: from the prior vertex of the first eliminated vertex to the next vertex of the last consecutive eliminated vertex of the sequence (right example at figure 15). A program constant ***MSV*** (Minimal Significant Variance) will be used to determine the necessary distance variance for considering it significant.

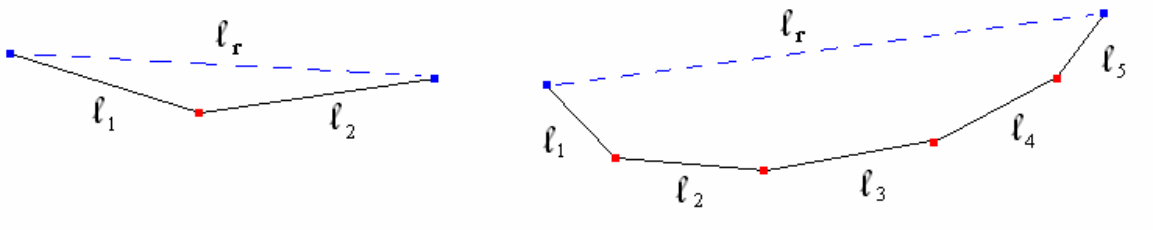

Figure 15. In this figure, the red vertices are being analyzed for possible eliminations, the blue vertices are prior or next vertices of the analyzed ones, blue dotted lines represent resulting distances whether the eliminations happen, and the black lines are path segments. The left example illustrates a possible single eliminations and the right example illustrates the possible elimination of a set of consecutive vertices.

We expect to have no excess of vertices after the simple path cleaning and it will save processing time for future processes and facilitate them.

3.1.4.3 General path cleaning

A *clean path* has to be free of *fluctuation vertices*, *non-OK vertices* and very large segments after a *basic path cleaning*. But *general path cleanings* do also eliminate *immediate returns* from the paths. The prior constraints do not need to be explained again but just the last one.

3.1.4.3.1 A clean path does not have immediate returns after a general path cleaning.

Initially, it is difficult to explain formally what an *immediate return* is, but it is not difficult to understand when an *immediate return* occurs. In the uppermost examples of the figure 16, we have trajectories with *immediate returns* (red lines) and these paths are not desired. The lowermost examples illustrate the equivalent desired trajectories that do not have *immediate returns* and may be represented by one or more paths (green lines).

In a not formal definition, *immediate returns* happen when similar consecutive fragments of the same path have very different directions. It may happen in cases as the next examples: A person in a car stopped it and went back some meters to check an interesting announcement; a vehicle turned back and drove through the same road; a car driving on an avenue made a U-turn; an animal made a very acute bend during its trajectory; etc.

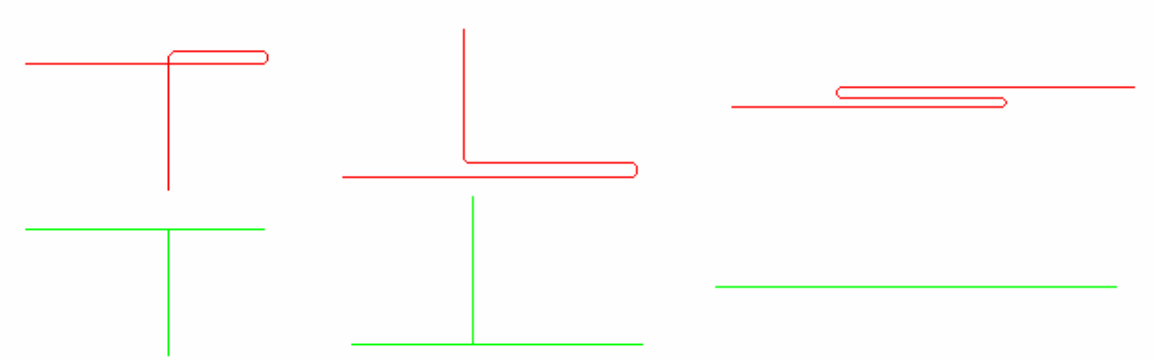

Figure 16. The uppermost examples illustrate trajectories with *immediate returns* and the lowermost examples illustrate equivalent trajectories without *immediate returns* (some of them represented by two paths).

One reason that makes *immediate returns* undesired is that they make *self-similarity* detection difficult to solve. For instance, we could say "we find a *self-similarity* on a path if two of its points are very close one to another, except if these points belong to the same or consecutive segments". But *immediate returns* make this condition insufficient because an *immediate return* can make a *self-similarity* to happen on consecutive segments. If we eliminate *immediate returns* from same paths, we can use this prior condition later for finding *self-similarities*.

Actually, an *immediate return* is undesired because it is a kind of *self-similarity*, and the averaging process for two paths gets very confusing due to *self-similarities* (we would never know which segments from *self-similarities* we should choose for the average).

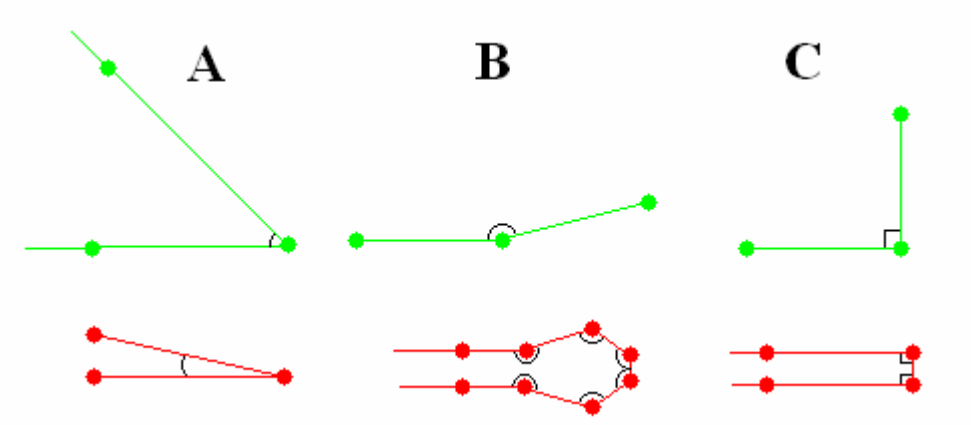

Figure 17. The angles formed by consecutive segments are not good references for determining if they form *immediate returns* or not. Above we have trajectories where every angle formed by consecutive segments is acute (examples A), obtuse (examples B) and right angles (examples C). For all examples we have trajectories we would like to accept (green lines) and trajectories we would not (red lines).

We could use the angle between consecutive segments for seeking *immediate returns* and try to eliminate them whether they happen, but angles are not enough to define the concept of *immediate return*. It is more complex and the figure 17 illustrates this.

The first idea was to detect *immediate returns* checking not only the angle formed by (two or more) consecutive segments but also the minimal distance from their endpoints to

the other segments. The first intent for this formal definition was: "an *immediate return* happens when the lower angle formed by two consecutive segments is lower than 90º and the minimal distance from their endpoints to the other segment is lower than or equal to the *PSD* distance[8]" (example A in the figure 18). This definition was not sufficient because *immediate returns* could keep occurring for 'soft' returns without acute angles between consecutive vertices (example B in the figure 18).

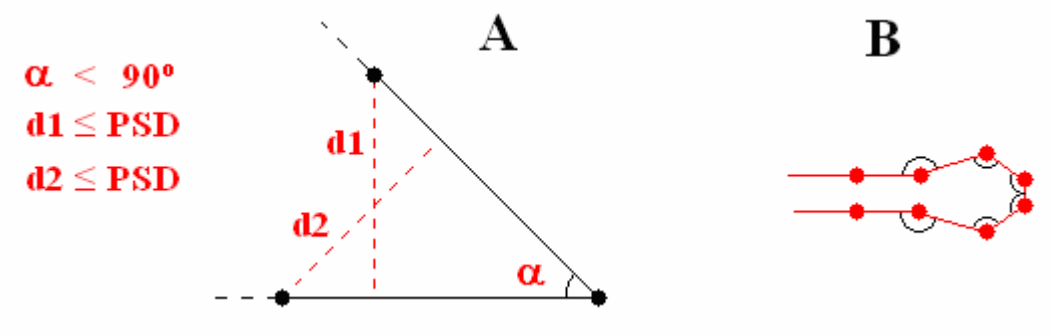

Figure 18. Example A: First idea of *immediate return* definition: The angle between consecutive segments should be acute and their endpoint vertices should be *close enough* to the other segment. Example B: The prior definition is not sufficient. This example illustrates a 'soft' return where the lower angles between all consecutive segments are obtuse but the *immediate return* keeps occurring.

Therefore, not always only two segments, but a set of consecutive segments have to be taken in account to check *immediate returns*, and this set of segments that has to be checked is going to be called *PSD range set of consecutive segments*, or simply *PSD range set*.

In a formal definition, the ***PSD range set*** of a given segment s_i^x (between the vertices p_i^x and p_{i+1}^x) is a set of consecutive segments before p_i^x whose last endpoints[9] are *close enough* to the vertex p_i^x (example A in the figure 19). It means, the *PSD range set* is the set of segments that are interesting to be compared with the segment s_i^x (red segment in the figure 19) for *immediate return* searching.

Now we can finally define *immediate returns*. An ***immediate return*** happens for a segment s_a^x when it forms an acute angle with some segment s_b^x from its *PSD range set* and the last endpoint of s_a^x or the first endpoint of s_b^x is *close enough* to the other segment.

To compare the angle of two non-consecutive segments, a segment has to be displaced to another making its first endpoint to coincide to the last endpoint of the other segment. The example B in the figure 19 illustrates how these angle comparisons are performed.

[8] Remark: The *PSD* distance defines if two points from different paths can belong to a same road tried to be represented by both paths. These points are close enough if the distance between them is lower than the *PSD* distance.

[9] Path orientations are going to be taken into account for the next definitions, but we are just going to consider segments with first and last endpoints (with respect to the path orientations) instead of vectors.

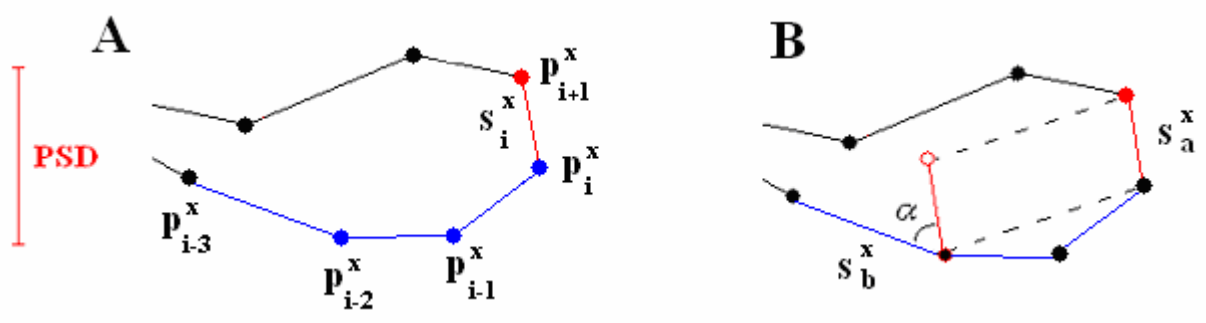

Figure 19. Example A: The *PSD range set* of the red segment s_i^x between the vertices p_i^x and p_{i+1}^x is the set of blue segments whose last endpoints (blue vertices) are close enough to this red segment. Example B: The angles formed by two non-consecutive segments (s_a^x *and* s_b^x) are checked by displacing the segment s_a^x and making its first endpoint to coincide to the last endpoint of the other segment s_b^x . They will form two angles, but we will just care about the acute angles for checking *immediate returns*, and only one of these two angles can be acute (whether there is an acute angle).

The questions now are "what should we do to eliminate *immediate returns* when they are found? Ignore vertices? Split paths?" Ignore a vertex in such situation may represent a quite considerable loss of accuracy, so it was thought to simply split the path in two. But later it was realized that the performance of this idea could be responsible of several path divisions, and some of these splits could even be unfair! For instance, a common turn on an ordinary corner could cause an *immediate return* but a path split wouldn't be desired in this situation (figure 20).

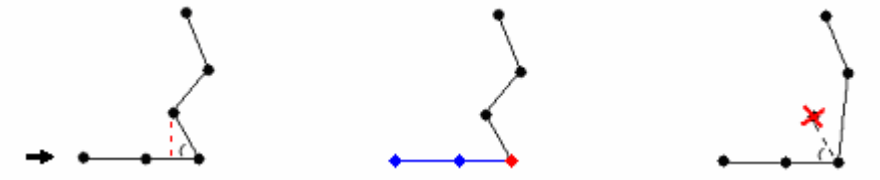

Figure 20. The leftmost example illustrates a possible trajectory of a car that turned on a corner occasioning a little *immediate return* due to an error margin. The middle example illustrates a path split in two ones (blue and black) due to the *immediate return*. The red point is the vertex where the *immediate return* is detected but actually the vertex responsible for it is the next vertex. The rightmost example illustrates that to ignore a vertex responsible for the *immediate return* could be much more interesting in this situation (but not always).

The chosen approach is to ignore those vertices that are responsible for *immediate returns* (explained in the middle example of the figure 20). But then we face another problem: we could lose considerable path accuracy! The figure 21 illustrates this possible loss.

So, sometimes it is better to discard vertices and sometimes it is better to split the path. The situations where we would desire to fix the path instead of splitting it happen only on 'natural' corner turnings (like in the figure 20). But a 'natural' corner turning is quite hard to define (due to the error margin) even if they are quite obvious for human eyes.

Figure 21. The leftmost example illustrates a possible trajectory of a car with a relatively small *immediate return* for taking another way to the left. The rightmost example illustrates the loss of accuracy that happens in this situation due to the removal of only one vertex that is responsible of an *immediate return*.

It is nice to fix an *immediate return* whether it is possible, instead of splitting the path. Many ideas were considered to solve this problem, but there were no simple solutions for fixing *immediate returns* that do not put in risk a significant accuracy loss.

Of course there are complex solutions for it, like checking angles and distances of many consecutive segments before and after any vertex elimination and giving a particular solution for each of the many possible situations, but these heavy solutions were not welcomed due to its complexity and possible time cost!

Many complex solutions were discarded because most of them were just partial solutions with many conditions with many comparisons that could slow the process significantly. The aim of *immediate return cleaning* is to solve problems here to make future processes easier and faster, but not to create problems already here making the program quite slow.

Instead of searching these complex solutions for the situation of 'natural' corner turnings (that are not returns) we could be less strict on *immediate return* checks. We were checking *immediate returns* for segments forming 'any' acute angle. Whereas we could check *immediate returns* only for segments forming an angle lower than a pattern angle (e.g. 60º, 45º, etc). It is a little risky because then *immediate returns* might happen without being detected and the more acute the pattern angle is the more risky it will be.

This angle (lower than 90º) that is taken as a pattern to check *immediate returns* is also going to be a program constant whose value can be previously defined and it is going to be called ***MAIR*** (*Maximal Angle for Immediate Returns*).

If the *MAIR* angle is reasonably close to 90º, it is quite difficult that a real return happens without being detected; and if the *immediate return* whose angle is greater than *MAIR* actually was not a return but an ordinary turned corner, we would be glad to ignore such *immediate return* and avoid unnecessary path splits. Anyway, mathematically the risk of not detecting real immediate returns will exist for any *MAIR* angle lower than 90º.

Nevertheless, to be totally strict is also a risk for memory management and the processing time due to the possibility of several path splits caused by many accidental *immediate return* that were not returns but simple corner turnings disturbed by the error margin.

3.1.4.4 Total path cleaning

This type of cleaning is the last one in this project and it has only two additional constraints that have not been explained before. They are going to be explained next.

3.1.4.4.1 A clean path does not have self-intersections after a total path cleaning.

Self-intersections are important information because they may be quite troublesome data to work with, especially when they form very small *loops* and we have to find path averages near to it. The prior *general path cleaning* (that avoids *immediate returns*) should discard every small loop (because they cause *immediate returns*) but not big loops. It is recommended to split the *self-intersecting* paths having two (clean) paths without *self-intersections* as result (figure 22). Then we can use them for further fundamental processes.

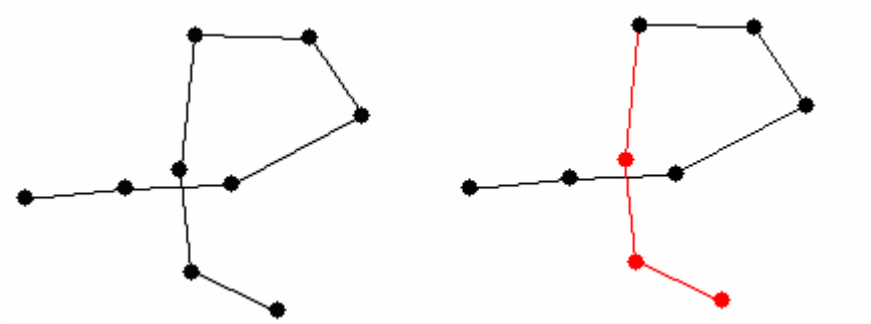

Figure 22. A *self-intersecting path* is illustrated in the left figure. In a *total cleaning*, this path should be split in two paths (right figure) without *self-intersections*.

However, a *self-intersecting* path must have a *self-similarity* near to its self-intersection, and we also want to clean paths from *self-similarities* in *total cleanings*. Therefore, instead of discussing about an effective procedure for avoiding only *self-intersections*, we are going to assume that they have *self-similarities* and they will be clean when their *self-similarities* are gone.

3.1.4.4.2 A clean path does not have self-similarities after a total path cleaning.

Later on, different paths are going to be checked for searching possible similarities between them, but one path may have similarities to itself (*self-similarities*), and it might complicate this further process. Then, we desire clean paths to be free of *self-similarities* and we are going to discard these inconveniences in this cleaning process.

It seems to be easy to find a simple definition for *self-similarity*, and the first definition that one uses to imagine is the following:

- "A *self-similarity* happens when two different segments from the same path have points that are *close enough (have a distance smaller than PSD between them*".

This definition is wrong, because any segment from any path has points that are *close enough* to points from its consecutive segment, and not every path has *self-similarities*. Then, another definition was quickly created:

- "A *self-similarity* happens when two segments from the same path, that are not consecutive segments, have points that are *close enough*".

The new definition is still wrong because of two reasons. First, two segments that are not immediately consecutives might still have points that are *close enough* without causing a *self-similarity* (figure 23). Second, remember that two consecutive segments may represent a return and they also should be considered *self-similarities* in this situation.

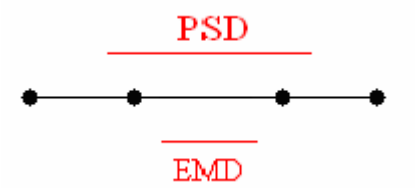

Figure 23. Two non-consecutive segments (the leftmost and the rightmost segments) can be *close enough* (having point distances lower than or equal to PSD) and cause no *self-similarity*. The minimal distance between consecutive vertices (clean of fluctuations) is EMD, and the distance PSD can be equal to 2 EMD + PPW.

Because of that, the need of the *PSD range set* definition arose, and it was used before in *general path cleanings*. The real reason of this set definition was the need of having a special segment set, where its segments can have points that are *close enough* to some point from another of its segment but without necessarily causing a *self-similarity*. That is the already defined *PSD range set*. Now we can define a *self-similarity*:

- "A ***self-similarity*** happens when a segment s_a^x has at least one point p_i^x that is *close enough* to some point from another segment s_b^x (from the same path) that does not belong to the *PSD range set* of the segment s_a^x".

However, special kinds of *self-similarities* may also happen in a *PSD range set*, for instance when very acute angles are formed by immediately consecutive segments. That is completely right, and this special kind of *self-similarities* in *PSD range sets* is what we have called *immediate returns*. Therefore, *immediate returns* are special kinds of *self-similarities* because they only happen in *PSD range sets*.

The *total path cleaning* will split the path if it is necessary for avoiding *self-similarities* in the same one, thus *self-intersections* on the same path will also be impossible. *Self-similarities* do not imply *immediate returns*, as *immediate returns* do not imply (other kinds of) *self-similarities*. This is one of the reasons why they are checked separately.

The approach of the *total path cleaning* procedure could be to find the average of all these segments that cause *self-similarity* and already replace them by their averages. But it was preferred just to split the path in two paths without *self-similarities* (nor *self-*

intersections) and let the average of similar segments to be done in a later averaging process.

3.1.5 Cleaning combinations

As it was said before, the *basic path cleaning* is only interesting for input data cleaning because the *simple path cleaning* has the only constraint from the *basic path cleaning* that is of interest for already constructed paths, even after modifications due to future processes.

However, the *general* and *total path cleanings* have additional constraints (that do not belong to the *basic path cleaning*) that could be usefull for already constructed paths. These additional constraints could be combined to the *simple path cleaning* for creating other types of cleaning for already constructed paths, but these types are not considered in this project.

3.2 Similarity Detection process

One of the main objectives of this project is to manage inaccurate information (liable to error margins) to create a concise map from it. It is tried to avoid confusing situations that make a good data management difficult, e.g. several shuffled lines on the map that should represent the same road.

Sometimes two paths or path fragments want to represent the same trajectory, and we would like to replace them by a unique substitute path or path fragment. For that, it is necessary to detect when these situations occur.

Here we are not going to care about path fragments yet because in this process we just want to detect and manage pairs of paths that are entirely similar (path similarity is going to be defined soon), preparing them for the next *path averaging* process. However, path fragments are going to be considered later in the *fragment averaging* process, and these fragments can be managed as if they were complete independent paths. But now, let us just to care about entire paths.

Resuming it, the goal of this process is to detect those paths that try to entirely represent the same trajectory and to manage them before the next *path averaging* process.

The next process of *path averaging* should be applied for those detected similar paths, for replacing them in a unique one. If we just work with clean paths, it is much easier to achieve similarity detections. If the *simple* or *basic cleaning* was applied, we are not going to have useless data to work with, and if the *general cleaning* and *total cleaning* were applied we shouldn't care about *immediate returns*, *self-similarities*, nor *self-intersections* anymore. Next we are going to define when paths are considered similar.

3.2.1 Path similarity

Path similarity is a condition where two different paths want to represent the same single trajectory and then they could be replaced by just one path. Many definitions of similarity are possible; some of them will be introduced is this subchapter and their advantages and/or disadvantages will be discussed. Their implementations may allow further concepts to be added.

There are two interesting similarity definitions that have been analyzed initially. In the first one, path orientations are not taken in account and we just care about path proximities. In the second definition, it is not enough one path to be completely close to the other one but the paths also have to keep similar orientations during all their trajectories.

3.2.1.1 Distance-similarity

The first definition for similar paths was: "if every vertex from both paths is near to some point from the other path, then these two paths are similar". This is the concept used for the created terms *distance-similarity* and *similar paths*.

Later on, it was found some situations where every point from a path is near to another path but we didn't want to consider them similar paths, because their trajectories were different. For instance, two mobile objects (one in each of these paths) beginning their trajectories at near path endpoints (at the same start time) would be in very distant positions after some time interval, even if they always had the same speed. A figure illustrating this situation can be found in the subchapter for *oriented distance-similarity* (figure 26).

Therefore, other concepts are necessary not just for checking the proximity of two paths but also for checking their approximated orientations during their complete trajectories. Then, the terms *oriented distance-similarity* and *similar oriented paths* were created, but they are going to be better defined later. Now we are going to define formally the simple *distance similarity* concept.

Distance-similarity:

Let say we have two different paths P_x and P_y. They are said to have ***distance-similarity*** if they fill the following conditions:

- Every vertex p_i^x is *close enough*[10] to a point p_a that belongs to P_y.
- Every vertex p_j^y is *close enough* to a point p_b that belongs to P_x.

Let us look at some examples to understand these conditions easily. The paths from the example A of the figure 24 are similar, while the paths from the example B of the same figure are not similar if we use the prior *distance-similarity* definition. The *PSD* distance (path similarity distance) is being taken in account to judge if each vertex is *close enough* to some point from the other path. In these figures, the red and green lines represent the smallest distance from path vertices to the other path. At the example B, the similarity fails because, even though all the vertices from the blue path are *close enough* to the black path, two vertices from the black path are not *close enough* to the blue path (the smallest distance is greater than the *PSD* distance).

3.2.1.2 Path similarity using polygon areas

It also was thought to use polygon areas to check similarity between two paths: "If the areas of all the polygons formed by path fragments between consecutive intersection

[10] Remarks: Two points are *close enough* if the distance between them is not greater than *PSD*, and a point belongs to a path when it belongs to a segment defined by consecutive vertices of the path.

points are considerably small, then the paths are similar". But this concept did not work always. In the figure 25, we can see a situation where such a polygon area could be small but the distance between some vertices and the other path could be quite considerable. Whereas the prior definition keeps working in this situation and would not accept the similarity.

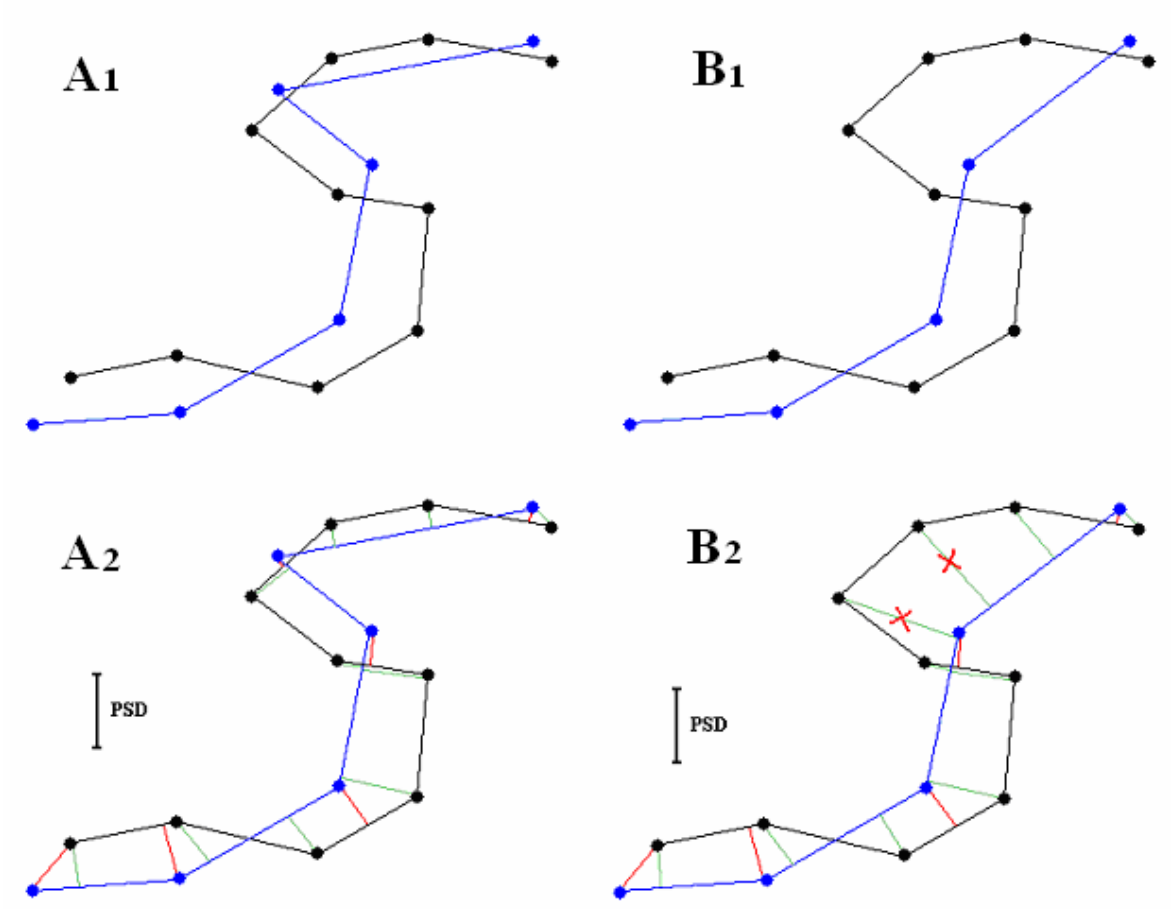

Figure 24. There is *distance-similarity* between the paths of the example A (left figures) but the *distance-similarity* fails in the example B (right figures). In the lowermost figures, red lines represent the smallest distance from vertices of the blue path to some point from the black path and green lines represent the smallest distances from vertices of the black path to some point from the blue path. The *PSD* distance is used to judge the similarity.

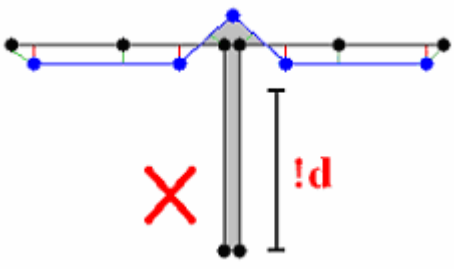

Figure 25. The similarity should be denied in this example, but the method using polygon areas doesn't work for this situation. Vertex-to-path distance is a better tool to judge similarity than polygon area is.

3.2.1.3 Oriented Distance-similarity

In the next figure 26, all the vertices from a path are *close enough* to the other path, so they have *distance-similarity*. However, the prior similarity definition may be not enough for judging similar trajectories in some situations. In this figure, for example, if two mobile objects begin the trajectory at near origin points and travel the same distance, they might be in very distant positions. Hence, we begin to understand the importance of path orientations for comparisons, and we need another concept for oriented similarity comparisons.

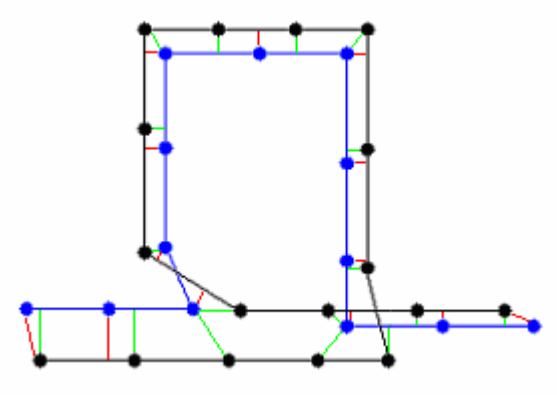

Figure 26. The similarity between the black and the blue paths should be denied in this example if we care about the path orientations and constant trajectory similarity, but the similarity is going to be accepted if we just use the simple *distance-similarity* concept.

To introduce the concept of *oriented distance-similarity*, we first have to introduce other concepts that were not necessary for the prior *distance-similarity* due to the definition simplicity.

3.2.1.3.1 Oriented distance-similarity by using approximation points:

For two paths to have *oriented distance-similarity* they have to follow the same *distance-similarity* conditions mentioned before, but one more condition is added for this new concept. Thus, it is necessary to introduce the new term for *'approximation fragments'*. When two paths are said to be similar, each of their vertices should have *approximation fragments* in the other path. Let us to define this new term in a formal way:

Given two paths P_x and P_y, the *approximation fragment* of a vertex p_i^x (from the path P_x) is every fragment f_a^y (from the other path P_y) where all its points are *close enough* to p_i^x and points immediately after or before f_a^y (whether they exist) aren't *close enough* to p_i^x (this means that the *approximation fragment* of a vertex should include every consecutive points that are *close enough* to its correspondent vertex). A simple example is illustrated in the figure 27.

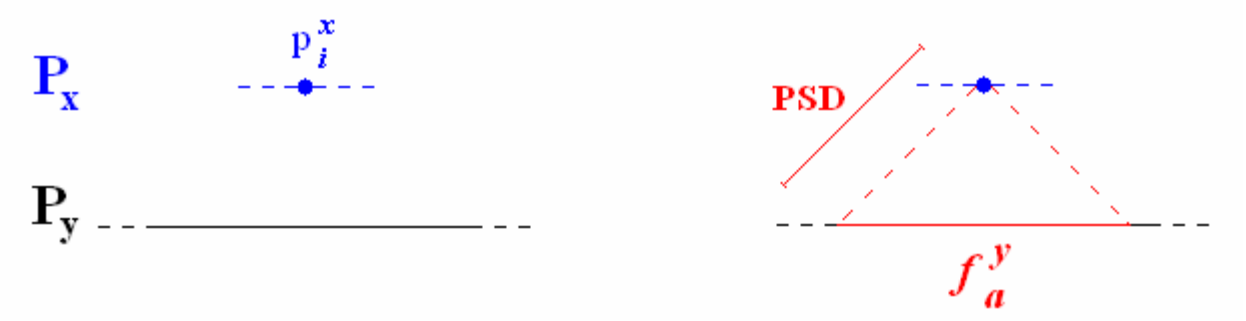

Figure 27. Finding the *approximation fragment* of a vertex p_i^x from the path P_x with respect to another path P_y (left figure): In the right figure, the red fragment is an *approximation fragment* of the only vertex in this figure. The points immediately after and before the fragment aren't *close enough* to the vertex.

If a vertex doesn't have any *approximation fragment* with respect to another path, then they aren't *similar paths* because it fails even the condition for simple *distance-similarity*. But in case that the condition for *distance-similarity* is accomplished, the vertex must have one and perhaps more than one *approximation fragment*, as we can observe in the figure 28, so we have to consider this possibility.

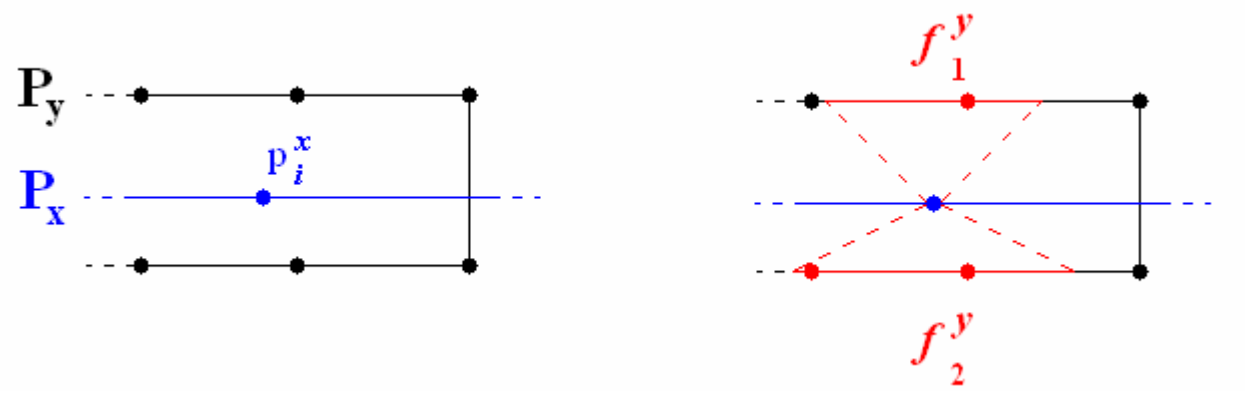

Figure 28. A vertex on a path might have more than one *approximation fragment* on the other path, even though *self-similarities* or *immediate returns* don't occur. This figure induces us to believe there *is self-similarity* in this example but it depends of the relativity between the distances on this figure and the *PSD* distance. The roads represented by the upper and the lower black path fragments might be parallel streets for instance.

The figure above only illustrates an example with two *approximation fragments*, but a vertex can have even more *approximation fragments* like these. However, only one of these *approximation fragments* is going to be chosen for each vertex when defining the concept of *oriented distance-similarity*. The chosen one is not necessarily the *approximation fragment* that is the closest one to an analyzed vertex, but the *first approximation fragment* (with respect to its path orientation) will be used to guarantee the similarity of the orientations.

The '*first approximation fragment*' means the one that hast points before any point from another *approximation fragment* from a same set. What we really want to find for each vertex is a unique point of comparison on the other path, and this point is going to belong

to the *first approximation fragment* of the vertex. This point of comparison will be called *prime approximation point*, but first we need to define the concept of *approximation point*.

An *approximation point ps* of a vertex p_i^x is a point from an *approximation fragment* f_a^y of p_i^x that has the closest distance to this vertex. Every *approximation fragment* has at least one *approximation point*.

Again, the same *approximation fragment* f_a^y might have more than one *approximation point*: these points from f_a^y have the same distance to the vertex p_i^x but these distances are equal and they are the closest distance from f_a^y one to p_i^x.

Again, we have to choose just one of the *approximation points* for each vertex, because just one of them is enough for similarity comparisons. We are going to choose the first *approximation point* (with respect to the path orientation) and this point is going to be called the *prime approximation point*. Formally talking, the *prime approximation point* of a vertex p_i^x is the first *approximation point* from its first *approximation fragment* of p_i^x with respect to the orientation of a reference path *Py*. It is illustrated in the figure 29.

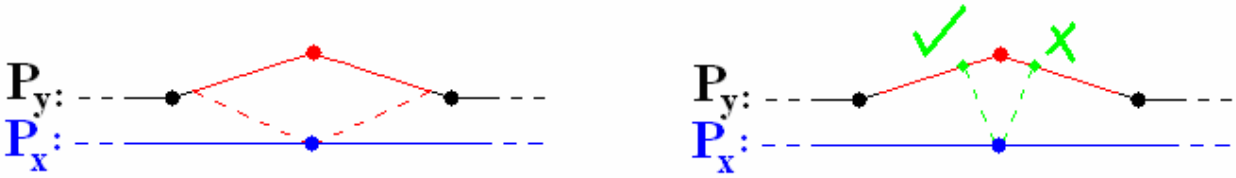

Figure 29. A vertex might have more than one *approximation point* from the same *approximation fragment*. In the left figure, the *approximation fragment* for the blue vertex is just defined, and in the right figure we can notice that the blue vertex has two points from this *approximation fragment* that have the closest distance to such vertex (two *approximation points*). The first of them (considering path orientations from left to right) is chosen to be the *prime approximation point* of the blue vertex (supposing that it has no prior *approximation fragment*).

Now that we have defined a *prime approximation point* and it is the only point for each vertex to be used for comparisons, we can finally define the conditions for *oriented distance-similarities*. However, this next definition got deprecated and the reason will be explained soon.

<u>*Deprecated oriented distance-similarity*</u>:

Let say we have two different paths $P_x = p_0^x, p_1^x, \cdots, p_m^x$ and $P_y = p_0^y, p_1^y, \cdots, p_n^y$. They are said to have *oriented distance-similarity* if they fill the following three conditions:

- *Px* and *Py* must have (simple) *distance-similarity*, thus every vertex from a path must have its *prime approximation point* on the other path.

- For all i, where i is the index of an existing vertex from the path Px, the *prime approximation point* of the vertex p_i^x is not before the *prime approximation point* of the prior vertex p_{i-1}^x.

- For all i, where i is the index of an existing vertex from the path Py, the *prime approximation point* of the vertex p_i^y is not before the *prime approximation point* of the prior vertex p_{i-1}^y.

In other words, every vertex from a path must be *close enough* to some point from another similar path, either they are oriented or not, but in an *oriented distance-similarity*, the comparison point for a vertex(*prime approximation point*) must not be before the comparison point for the prior vertex. It will guarantee a strict orientation similarity.

We can notice that it if we apply this *oriented distance-similarity* definition for the anterior example of the figure 26 (now illustrated in the figure 30 considering path orientations) the similarity will be denied, as we expected it to be because of different orientations in some considerable fragment. In a given moment of these trajectories, a vertex (from any of these paths) will have a *prime approximation point* that is before the *prime approximation point* of its prior vertex, so the *oriented distance-similarity* will fail.

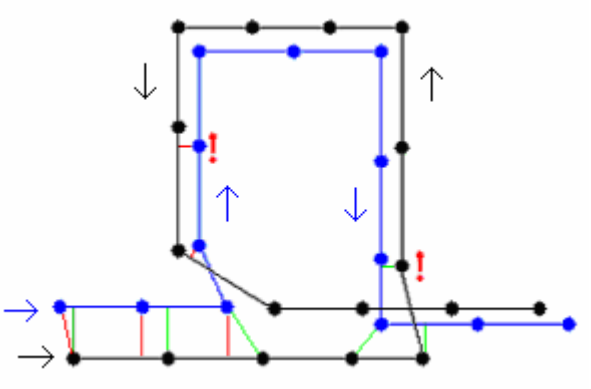

Figure 30. The same example that didn't fail for the simple *distance-similarity* fails for the *oriented distance-similarity*. When finding the *prime approximation point* of vertices, it does not follow every condition for *oriented distance-similarity*. The first failure (for each path) happens at a vertex marked with an exclamation sign.

3.2.1.3.2 Oriented distance-similarity by using approximation sets:

The prior concept of *oriented distance-similarity* by using *approximation points* is strong and quite hard to fail. On the other hand, this concept would be very hard to implement. Imagine that for each vertex of each path we would have to find its *first approximation fragment* for then calculating its *prime approximation point*, and only after that, the comparison for similarity would be applied.

Just the work for finding the first *approximation fragment* is sufficiently hard! Try to realize the needed calculations for determining the exact point where such fragment ends and begins. It would coast a lot.

Hence, it was thought to use 'sets of near segments' instead of *approximation fragments* because we don't need to calculate the limits of segments once they are already well defined by their *endpoints* (that are vertices on the path). Then we have to define new concepts for a simpler similarity definition, but these new concepts are very simple and all they can be defined in a single paragraph by using prior defined concepts.

'An *approximation segment* of a vertex' is a segment from another path that has points *close enough* to such vertex. We are going to define the '***approximation set*** of a vertex' as the first set of its consecutive *approximation segments* on another path. The vertex that has been analyzed for defining such set of segments is called the *generator vertex* of the *approximation set*. The next figure illustrates it.

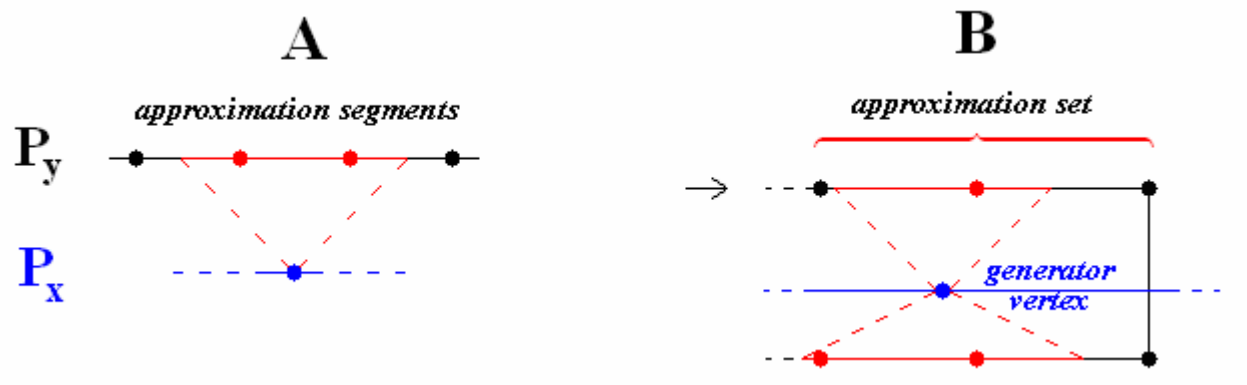

Figure 31. Any *approximation segment* encloses part of an approximation fragment for a given vertex. The example A shows to us that often the limit points of an *approximation fragment* (red line) are not necessarily well defined vertices, but in the example B we can see that *approximation sets* are always well defined by vertices because they are sets of *approximation segments*. In the example B we have more than one set (up and down) for one generator vertex, but again only the first of these sets will be chosen as its *approximation set*.

Another advantage during the implementation of this new similarity concept to be explained is that we don't need to check distances from vertices of a path *Px* to another path *Py* and then to check distances of vertices from *Py* to the path *Px*. Only one of these checks is necessary.

For checking complete similarity, we are going to compare the *approximation set* for every vertex (from only one path) with the *approximation set* for its next vertex. Next, we are going to mention the possible situations of similarity gap and explain how to ensure these situations won't occur (guaranteeing *oriented distance-similarity*) for each case:

1) An *approximation set* may have gaps of similarity among their segments (leftmost example in the figure 32) and a way for controlling that it doesn't occur is checking

every set vertex that links set segments. All these vertices must be *close enough* to the *generator vertex* of their *approximation set* for ensuring continuity of similarity

2) Whether two consecutive vertices are distant, their *approximation sets* on the other path may have gaps of similarity between them (rightmost example in the figure 32) and a way for controlling that it doesn't occur is checking every vertex between these two *approximation sets* (including their *endpoints*).

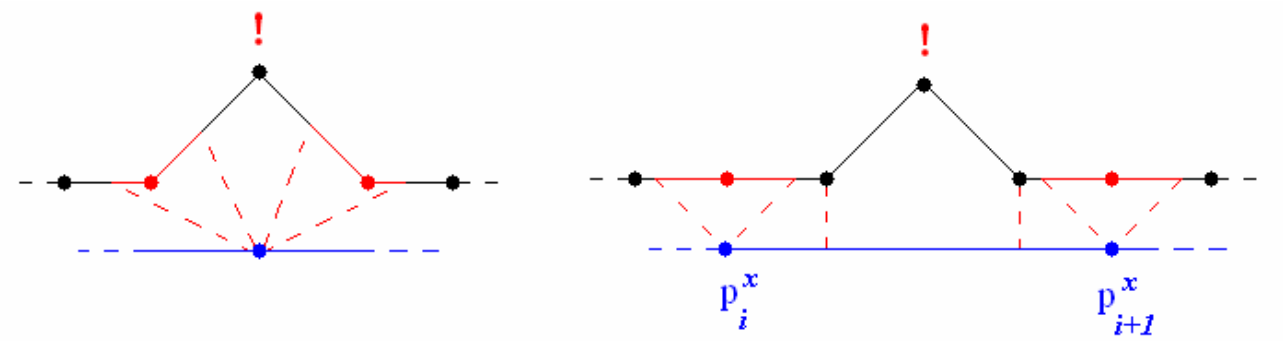

Figure 32. The leftmost example illustrates a generator vertex and all the consecutive segments of its unique *approximation set*, but this set has a gap of similarity due to a distant vertex. The rightmost example illustrates two consecutive *generator vertices* forming two *approximation sets* on the other path with a similarity gap between them, due to only one distant vertex.

3) Points from or before the first segment of the first *approximation set* of a path may have gaps of similarity. These gaps weren't checked here yet because they aren't in nor between *approximation sets* (figure 33). The same happens for possible gaps from or after the last segment of the last *approximation segment* of a path. The path endpoints must be checked to ensure that there is no similarity gap.

Figure 33. Similarity gap at the beginning and end of paths use to be very frequent, so, path endpoints have to be checked. One (right example) or more (left example) vertices from the same path extremity can fail. Only the vertices from one path (the blue one) would not fail because they have *approximation sets* on the other path.

4) We already checked every condition for *distance-similarity*, but nothing ensures yet that it is oriented. The way for ensuring similar orientation is controlling the order of the computed *approximation sets*. It is done by checking that any *approximation set* begins before the first endpoint of its next *approximation set*, as well as it ends before the last endpoint of its next *approximation set*[11].

[11] Remark: Each *generator vertex* has only one *approximation set* defined by its first set of consecutive *approximation segments*.

The new definition for *oriented distance-similarity* is going to be explained next, having its five conditions in the same order that they have been explained before.

Oriented distance-similarity:

Let say we have two different paths $P_x = p_0^x, p_1^x, \cdots, p_m^x$ and $P_y = p_0^y, p_1^y, \cdots, p_n^y$. They are said to have ***oriented distance-similarity*** if they fill the following conditions:

- Every vertex from *Px* must have one *approximation set* (with at least one *approximation segment* in each).

- Every vertex linking segments of any *approximation set* must be close enough to its *generator vertex*.

- For all *i*, where *i* is the index of an existing element from the path *Px* excepting the last one, if the last vertex p_ℓ^y from the *approximation s*et generated by a generator vertex p_i^x is before or the same first vertex p_f^y from the *approximation set* of the next generator vertex p_{i+1}^x, then all the vertices from *Py* between the vertices p_ℓ^y and p_f^y (including themselves) must be *close enough* to some point from the segment formed by the generator vertices p_i^x and p_{i+1}^x.

- All vertices from *Py* before the first *approximation set* endpoint and after the last *approximation set* endpoint for generator vertices from the path *Px* have to be *close enough* to some point from *Px*.

- For all *i*, where *i* is the index of an existing element from the path *Px* excepting the last one, the first vertex from the *approximation set* for a generator vertex p_i^x must be before the first segment from the next *approximation set* generated by p_{i+1}^x, and the last vertex from the *approximation set* for a generator vertex p_i^x must be before the last vertex from the next *approximation set* generated by p_{i+1}^x.

3.2.2 Detecting similarity

Now we would like to gather all those paths that have *oriented distance-similarity* between them. We want to create sets of *similar oriented paths* for replacing later each of these sets by a unique substitute path. It would save a lot of memory and future processing time, but a new question arises again.

“What if we have two paths that contain exactly the same trajectory but their origins/destinations is the only difference? Should they belong to the same set?”

The answer is simple: Yes, they should. As we have said before, orientation is something abstract that we can define according to our convenience. In the *similarity detection process* we don't care about origins and destinations and we might want to swap them.

Here we don't use orientations for knowing from where to where two mobile objects went, but we use orientations for checking if they used exactly the same real path that links these places, independently of the trajectory start or end. So, we can perfectly redefine which endpoint is going to be the origin and which endpoint is going to be the destination of a path for further comparisons. It is done by reversing the order of a *path vertex list*.

Explaining it in one paragraph: "If a path P_x from a set S (of *similar oriented paths*) has no *oriented distance-similarity* to a path P_y but it has to its reverse path $P_{rev\ y}$, we are going to include $P_{rev\ y}$ to the set S and we are going to substitute P_y by $P_{rev\ y}$ during all this comparison procedure and during the further *path averaging* processes for *similar oriented paths*".

The first thing that must be examined for checking if two paths are similar or not are their endpoints, because initially we expect similar paths to have similar endpoints.

3.2.2.1 Tolerance for path extremities

We would expect two mobile objects that went from the same place of origin to the same place of destination using the same trajectory to have *oriented distance-similarity*. But we have to consider some possible similarity denials at the beginning and/or end of these paths that should be *similar oriented paths*.

The first and very common of these similarity failures may happen due to a GPS-receiver delay for receiving its first satellite signals. Of course that this time can vary from system to system. Moreover, this is not the only factor that would influence in initial or final differences of two (paths that should be) *similar oriented paths*. Other factors might be small differences between the starting points (e.g. different car positions in a parking zone), the initial mobile object speed, the system error margin, etc.

Due to these possible factors, we must be tolerant with respect to path similarity during comparisons, but only at path extremities because we don't have reasons to tolerate similarity failures at the middle of path trajectories. An example of unequal extremities of *similar oriented paths* is illustrated in the figure 34, and it also shows the tolerance that should happen for their similarity comparisons.

Another program constant is going to be needed for defining how much length we can accept as tolerable path extremities, and it may vary from system to system as well as the factors that produce this problem. The used constant is called ***TLE*** (Tolerable Length for Extremities) and an exception is going to be considered during comparisons for finding *similar oriented paths*.

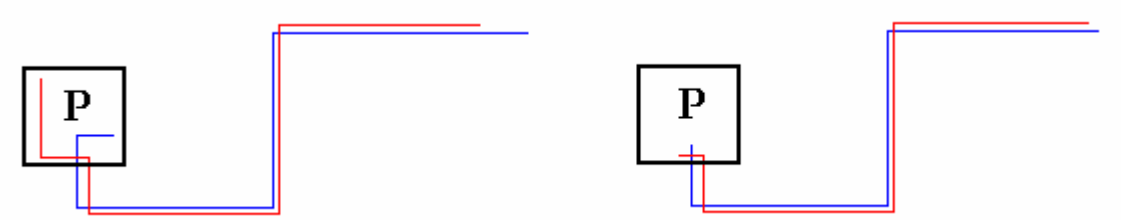

Figure 34. Two similar paths (red and blue lines) that begin in different places of the same parking zone. At the left example, they have different extremities but we would like them to be considered as *similar oriented paths*. At the right example, just the different extremities were tolerated making the *oriented similarity* possible.

Exception for path similarity comparisons:

When comparing the similarity between two pats P_x and P_y, if some point p_a from the path P_x is not *close enough* to the other path P_y but the length (along the path P_x) from p_a to some P_x endpoint is lower than ***TLE*** and p_a is before any other point that is *close enough* to P_y, we could consider that p_a does not exist 'for this similarity comparison' between P_x and P_y.

This exception solves the problem for unequal extremities of similar paths, but now there is a new problem for applying this exception. Let us to explain it using a new term *TLE range* as the set of initial segments and sub-segments of a path that are not after the point whose length (along the path) to its origin vertex is equal to ***TLE*** (as well as for the set of final segments and sub-segments of a path that are not before the point whose length to its destination vertex is equal to ***TLE***). We are going to explain this problem just for the initial extremities because then the explanation for final extremities can be induced.

It is easy to work with segments and vertices from a path but not with other path points. Let say that the last point p_ℓ from the *TLE range* for a path extremity is in the same segment than the first path similarity point p_i is. How would we know if p_i belongs to the *TLE range* or not?

There is only one way for knowing it: computing the points p_ℓ and p_i and comparing which point is before the other. If p_ℓ is before p_i then p_i does not belong to the range, otherwise it does belong. Sometimes this problem seems to be insignificant, but considering that the segment containing p_ℓ and p_i can be quite long then it becomes significant, so these points must be computed.

To compute p_ℓ is quite simple by measuring the *TLE* length from the path origin to some point of the respective segment containing p_ℓ. But… how to compute the first similarity point p_i in the segment? This is the problem.

Many ideas were analyzed for computing the first/final similarity points and all they had advantages and disadvantages. Again, we are going to explain the ideas for computing just the first similarity points in initial extremities, because solutions for computing last

similarity points in final extremities can be induced after it. The best ideas are going to be explained below.

- **The exact first similarity point:**

This is the most accurate solution but the hardest one to be reached. It is not impossible to find the exact path points where the first similarity happens with respect to another path, but we would need second order equations for calculating such points and the needed equations are not short at all. Therefore it was preferred to find approximated points instead of the exact first similarity points.

The points that should be computed are illustrated in the figure for the next solution, and we can have a basic notion of the second order equation that we would need for computing their positions (considering that the only information we have are the position of the segment vertices).

- **The first vertices from the first related segments:**

Related segments are pairs of segments of different paths that have at least one point from each segment that are *close enough* to each other. This solution just finds the first *related segment* of each path and considers the first vertex of each segment as the first similarity point for the correspondent path.

This is the easiest solution for determining some point to be taken as the first similarity point; on the other hand this is the only advantage of this idea, and not a good solution for many common situations. This approximation could be quite bad, especially when the considered segments are long. The next figure illustrates a possible bad approximation.

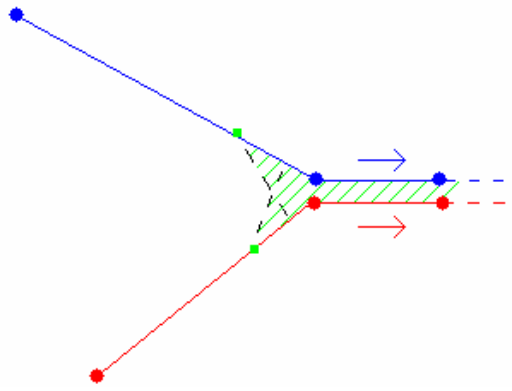

Figure 35. The green area represents a similarity area where every internal point is *close enough* to both paths. The green points represent the first similarity point for each path. We can notice that these green points might be quite distant from the first vertex of their segments. The leftmost segments of this figure are the first related segments.

- **The closest points from the first related segments:**

It was taken in account that it is not hard to find the closest points between two given segments, so the considered first similarity points for two paths could be the closest points between their first related segments.

For some situations this solution works perfectly, but for other situations it may cause a considerable accuracy loss as we can observe in the next explained figure.

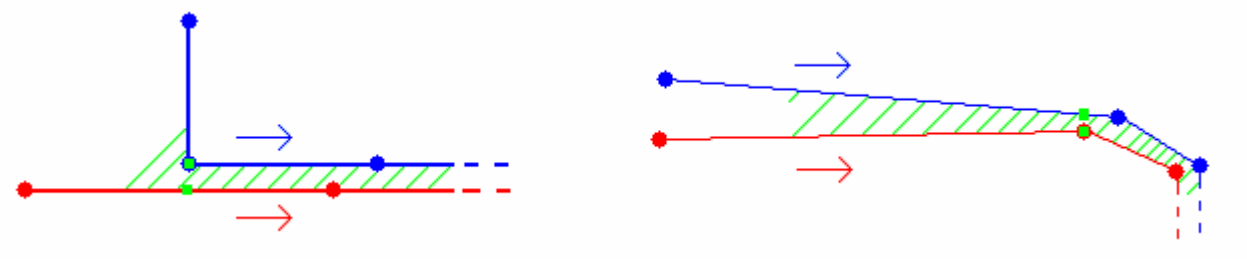

Figure 36 The green points here represent the closest points between the first related segments. Sometimes these green points are strategic for beginning an average process, but sometimes they can be quite distant to the real start of the similarity and a lot of necessary path averaging would be avoided.

- **Approximated first similarity point:**

This solution was the best one because it finds the balance between computability and accuracy. It doesn't try to use hard calculations for computing exact points and it does not try to guess the best approximation point (like previous ideas did).

For finding the first similarity point for a segment using this solution, we divide the segment into two sub-segments and choose one of its halves for continuing the procedure. The chosen half is the one containing the first similarity point and the other half is discarded. We continue this procedure until the remaining half is quite short, and then the distance between its middle and the first similarity point cannot be considerable.

Even if we didn't use hard calculations for it, we could compute a nice approximation for the wanted point. The only question is how to know which sub-segment half contains the first similarity point. It is not difficult to find out: If the middle of the sub-segment is not *close enough* to the other reference segment, it didn't reach the similarity area yet then the second half must contain the first similarity point, otherwise the first half already reached the similarity area then the first similarity point must be contained in the first sub-segment half. The first and second sub-segment halves are being considered with respect to its path orientation.

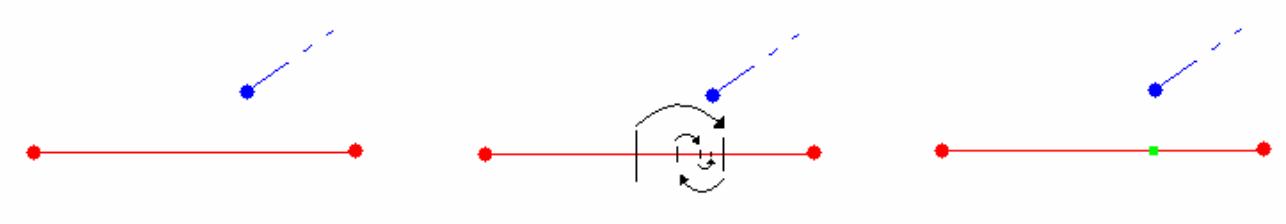

Figure 37. Left: In this example we would like to find the first similarity point for the red segment. Middle: The black transversal lines represent the middle of remaining sub-segments and the arrows represent the choice of a particular half, the other half is discarded. Right: Halves were chosen until the last sub-segment middle point (green point) is quite approximated to the first similarity point of the red segment.

Once we have defined how to compute the first and last similarity point for two paths, we can determine if they are *similar oriented paths* or not but tolerating possible initial and/or final undesired casualties happened in their extremities. The tolerated part of path extremities are just going to be ignored and they are not going to be part of future processes like the path averaging process.

3.2.3 Building the map graph

The process of similarity detection only cares about searching for similar paths, but it is used for managing these paths for further processes. A map graph is an excellent data structure for ordering the paths according to their similarities. The graph nodes would represent paths, and the edges (called 'links' in this project) would represent the similarities between them. The linked paths (or linked nodes) are going to be sets of paths that we would like to replace by a substitute or average path.

However, we may have situations where we have a set of shuffled similar paths to be replaced, but we don't know if we want to replace all them by just one average path because, even though every path is similar to another one in a node set, not all paths from the set have to be similar. We have this example in the upper left example at figure 38, where the light blue, green and yellow paths are similar; the black path is also similar to the yellow one, but not to the other paths. Then we should want an average path for the light blue, green and yellow lines and another average path for the black and yellow lines.

For solving this problem, we would have not only to apply the oriented similarity detection to every pair of paths and to link the nodes of similar ones, but we would have also to find node subsets where each node from the subset is linked to every other node from the same subset. Moreover, these subsets should be maximal subsets; this means they should not be subsets of another set where all nodes are linked [GM]. For instance, in the lower left figure, $S_i = \{P_1, P_2\}$ cannot be one of these subsets because it is a subset of $S_j = \{P_1, P_2, P_3\}$ where every node is linked too (then S_i is not maximal).

We could use a *'maximal clique algorithm'* to find these maximal sets. A clique in an indirect graph is a subset of vertices from such graph, each pair of which is connected by an edge.

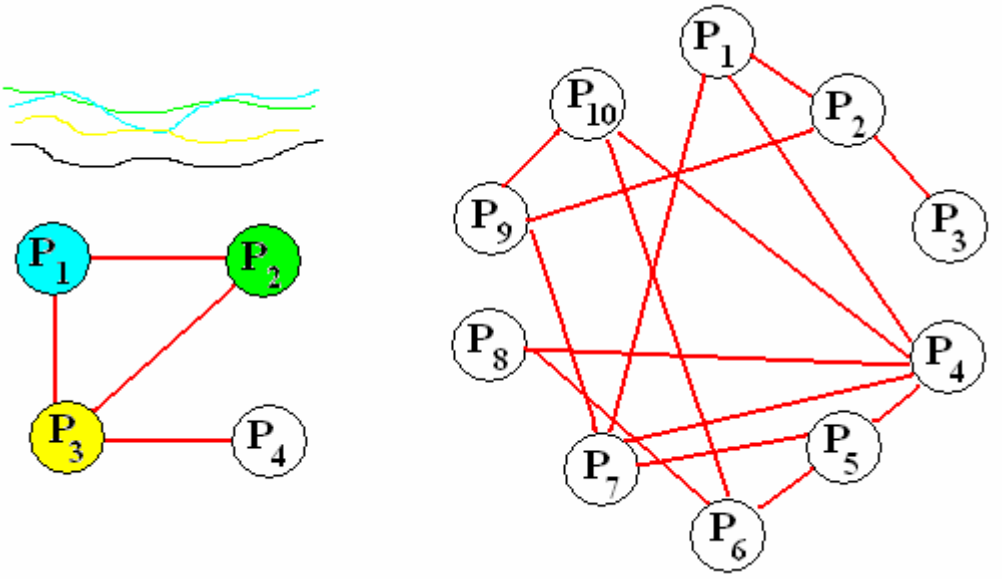

Figure 38. Left: A graph for four (upper) coloured paths. The black path P_4 only is similar to the yellow path P_3, thus this graph is not a clique. Right: A relatively small graph illustrating how complex to find *maximal cliques* can be.

But this is a hard problem to solve, especially because the *'maximal clique problem'* is a NP-complete problem (no polynomial time). Suppose we have a considerable quantity of paths and they create many graph subsets like the right example in the figure above. First we have to compare every pair of paths to form such sets, after we have to split these sets into maximal clique sets using the NP-complete *'maximal clique algorithm'*, for then finding their averages. How much time would it cost to us? The *'maximal clique problem'* is better explained in the chapter 36.5 of the book "Introduction to Algorithms" of Thomas Cormen et.at [IA].

Then it was decided to find the average path for each maximal set of linked nodes independently if they are cliques or not (this means that all nodes of a set have to be linked but not necessarily with direct links). Paths of a maximal set will be compared, pair by pair, and a pair of similar nodes (or paths) will be replaced by a new node for the average path of this pair. Therefore, the future *path averaging process* only needs to be applied for pairs of paths.

Probably it would be much more accurate to use maximal cliques, but it would cost us a lot of extra time and programming confusions. Remember that similarity will be applied not only for paths but also for path fragments! How many pair of similar path fragments could exist?

"To find a substitute path for a maximal set even if the set is not a clique" does not mean that "the similarity links are going to be ignored when choosing the sequence of path pairs to be averaged". We are never going to average paths of two nodes that are not linked, but we are going to replace each pair of linked nodes by a new substitute path node, always keeping those links to the new node that were referent to the prior substituted nodes (figure 39).

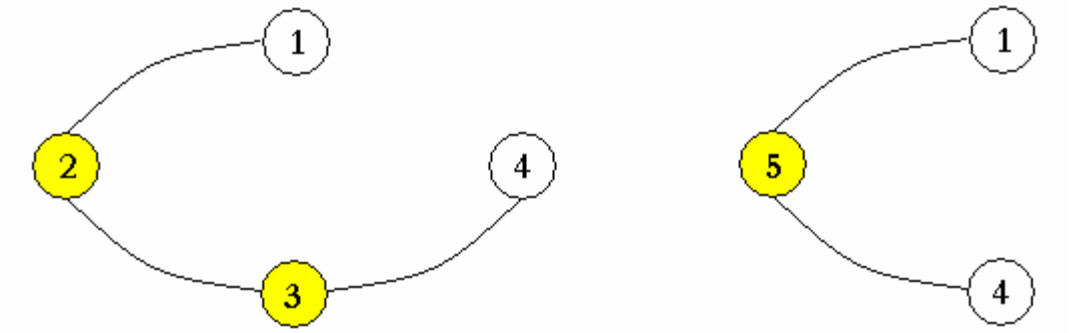

Figure 39. The yellow nodes 2 and 3 are replaced by a substitute node 5, but links from the node 5 to the nodes 1 and 4 exists due to the prior links from the substituted node 2 to the node 1 and from the substituted node 3 to the node 4.

But it may happen that a new substitute node is not similar to its linked nodes, because not all initial nodes (paths) were similar: the initial set was not a clique. In these controversial cases, a choice has to be done for substitutions of non-similar nodes: One node of the pair will remain as substitute and the other one will be discarded (because it makes no sence to average non-similar paths).

Path weights are going to be used for deciding these choices. It was decided to add *weight* variables to each path instance for knowing how much we can trust in it. The more *weight* a path has the more path averages it results from. This situation will be better explained and discussed in the next chapter (for situations where path averaging procedures fail and a substitute path has to be returned as result).

Another suggestion is to use weight for nodes in this procedure instead of path weights whether we don't want any interference from the path averaging process while building a map graph.

3.3 Path averaging process

This is the most elaborated process of this project and one of the most fundamental ones. Until now, we had processes for avoiding unnecessary data and for managing the system. The aim of this new process is to avoid map inconsistence. This is the part of the system where each pair of *similar paths* is going to be studied for computing a unique substitute path that better replace both of them.

Imagine situations where monitored mobile objects reuse several times roads that have already been trailed. Due to the error margin, we are going to have very similar but not equal paths for displaying the map. If we just include every received path to the map without replacing similarities, we might have a lot of unnecessary shuffled lines. Shuffled lines cause a confusing map and a lot of unnecessary data to manage.

To fight this undesired situation means to convert each pair of shuffled similar paths in a concise one by averaging them but always reaching a good accuracy. Converting two similar paths in a set of polygons makes much easier to obtain a path average from them, because polygons have well defined areas where to compute a resulting path in. But polygons to be used in this context have to be defined next because they have special properties:

A ***polygon*** is geometric figure surrounding an area limited by its boundaries. These boundaries are defined by two path fragments, where each fragment has at least one segment and both fragments have common origin and destination points[12] but no other common point.

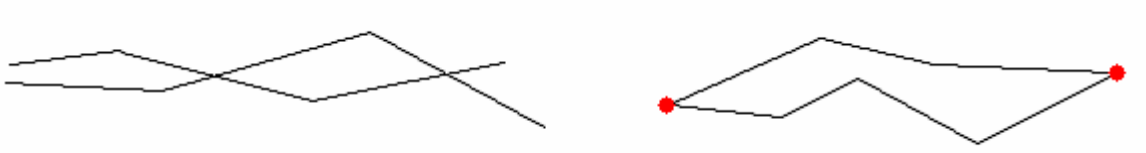

Figure 40. Left: Two similar paths forming one polygon. Actually, three polygons can be formed from this example. Right: Only one polygon where the red points are its origin and destination points. These points are the only common ones for the two fragments that form a polygon.

Normally (even similar) paths have no common origin or destination points at their beginning or end, but we could link their closer endpoints with a segment (for closing the polygon) and consider the middle segment point as the origin/destination for the polygon and as the common point for both fragments.

Similar paths may have many intersections, creating many polygons, thus we have to search for intersections along the paths. It might coast a considerable time whether the similar paths are long, and it might be very expensive for the program whether this search has to be done for several pairs of paths.

[12] Remark: We can decide which point is the origin and which point is the destination of each fragment according to our convenience. These points just have to be the first or last vertices from their fragments.

For instance, let suppose we have a monitored car traveling on international motorways during 5 hours, with a GPS-receiver saving information from a satellite each second. If there are no points to be discarded during a cleaning process, the path would have 18.000 vertices. If we use the simplest algorithm for detecting intersections among two paths like in this example, we would need about 18.000^2 =324.000.000 segment comparisons for searching common (intersection) points.

There are other algorithms that perform the same search in a better time. For instance, there is an algorithm for a procedure that performs this search in a *O(n log n)* time, where *n* is the number of path segments. This algorithm can be found in the chapter 35.2 of the book 'Introduction to Algorithms' of Thomas H. Cormen [IA]. The expended time for 18.000 * log 18.000 = 76.595 comparisons is much more reasonable than for 324.000.000 comparisons for the simplest algorithm. Anyway, there are some not solved problems in the book algorithm that could be necessary to and can be performed, e.g. dealing with vertical segments, finding points intersected by more than two segments.

Another suggestion for reducing the processing time, not just for intersections but also for similarity detections, is to perform a map grid for the pathfinder system. Only path segments belonging to the same grid cell can have intersections and only path segments belonging to the same or neighbor grid cells can be similar if cell sides are not smaller than the *PSD* distance. Hence, not every segment pair needs to be compared. On the other hand, an entire data structure would have to be built for the grid, where each grid cell would have to contain information about every segment or sub-segment it contains as well as each path segment will have to be linked to one or more grid cells.

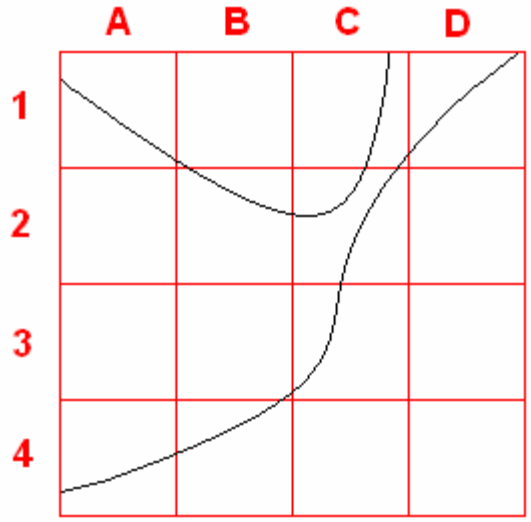

Figure 41. A map grid, with 4 rows and 4 columns, for helping the detection of path intersections or path similarities. Here the paths are quite accurate, but they use to be less accurate and divided in consecutive segments.

Nonetheless, we are not going to perform a map grid and we are going to use the simplest algorithm in this project, because its objective is not to perform optimal algorithms in time but to find out good solutions for the system requirements. Further attempts for optimal systems can be possible by replacing some program methods, like the method for

computing path intersections. We either will not care much about self-intersections here, once they should be solved during the cleaning process.

3.3.1 Polygon average

After the polygons (formed by two given paths) have been defined, we have to decide how to compute their averages. We are calling ***polygon average*** to a fragment that goes from the polygon origin to the polygon destination in a balanced way between both polygon fragments.

There are many different techniques for computing the average of a polygon. Actually we may have different polygon averages resulting from each technique. So, we have to consider the accuracy-computability relationship once more for choosing the technique to be used for polygon averages in this project. Each studied technique is going to be discussed next.

3.3.1.1 Polygon exact averaging

The first idea was to find a brilliant polygon average that would be the most centered line between its fragments, going from the common origin to the common destination, cutting their interior area in two equal sub-areas.

While this solution where being studied, a lot of problems arose. The main problem was how to compute centered segments for some polygon areas. It seems to be quite simple for some polygons like triangles or lozenges, but it may be quite complex for other weird polygons containing many disproportional side lengths and directions. Sometimes it could be hard even for humans to decide where the polygon center line should be.

Other non-mathematical decisions began to arise and then, some questions became important: Is it necessary to have a so accurate average path that divides the polygon interior in exactly equal areas using the most possible centered line? Does the result of these operations compensate their analysis complexity and the possible expensive processing time?

We have to take in account that we are working with data containing possible error margin and we also may have considerable intervals between data receptions, thus the original polygon itself don't use to be quite accurate. Due to error margin context, sometimes it might happen that exact fragment averages would get less accurate than an original fragment were. This means we could expend several time with complex mathematical procedures to have perfect average paths that actually are not very exact. Therefore, it was decided to think about another less complex technique for a good solution. So, the next studied techniques do not care about dividing the polygon in exact equal areas.

3.3.1.2 Polygon averaging by using triangulation

There is another technique where polygons would be divided in triangles, and then we could calculate the in-center (median intersection point) of all these triangles for defining the polygon average.

But, is it easy to find a rule for dividing a polygon in triangles? There are many possibilities for dividing them. The more sides a polygon has the more possible triangulations exist for this polygon, even if we avoid to connect points from same original polygon fragments and we do not use the origin nor destination point (figure 42).

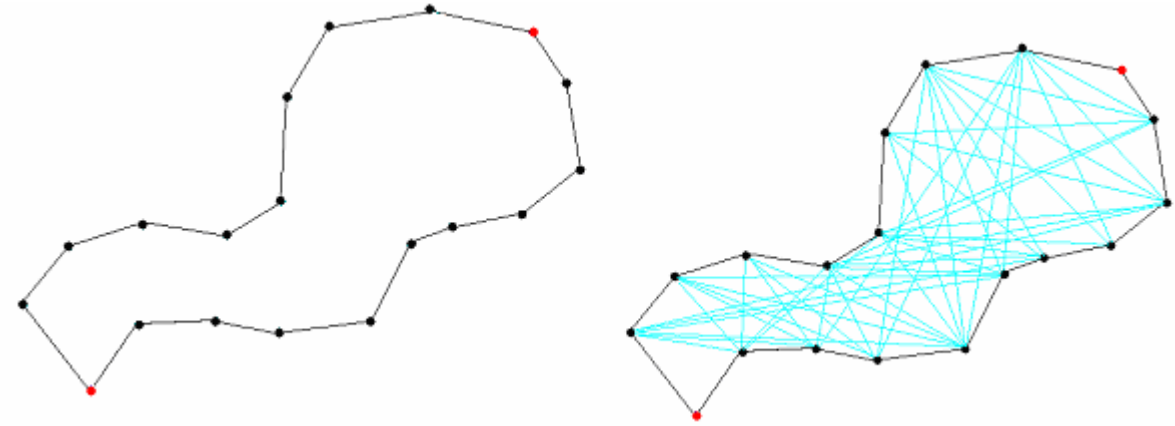

Figure 42. Left: A Polygon formed by two fragments that have common origin and destination (red) points. Right: Light blue lines show how many internal triangles may be formed for this polygon, even though they do not reach the origin nor destination point, each light blue line connects only vertices from different fragments, and they do not cross any boundary.

If we divide the polygon using different triangulations, the polygon average would change considerably. Therefore, it was necessary to choose one of many triangulation techniques.

One of the first found ones while searching for triangulation techniques was the Fast Industrial-Strength triangulation [FIST]. In the next figure we can observe the excellent job it can perform, but once again we discarded a good technique due to its complexity. Moreover, we don't believe to face so complex polygons in this project as the polygon illustrated in the figure.

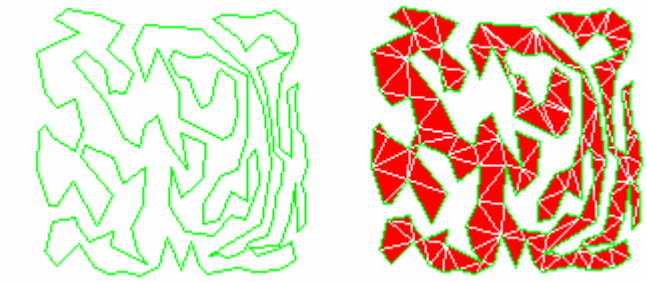

Figure 43. A complex polygon triangulated by using the Fast Industrial-Strength triangulation.

Another found technique was the Fast Polygon Triangulation based on Seidel's Algorithm [FPT], but normally the existing techniques do not follow an important constraint for our triangulation:

- All formed triangles should be consecutive from the polygon origin to the polygon destination point.

This constraint is going to be better explained later when applying the technique that does follow it.

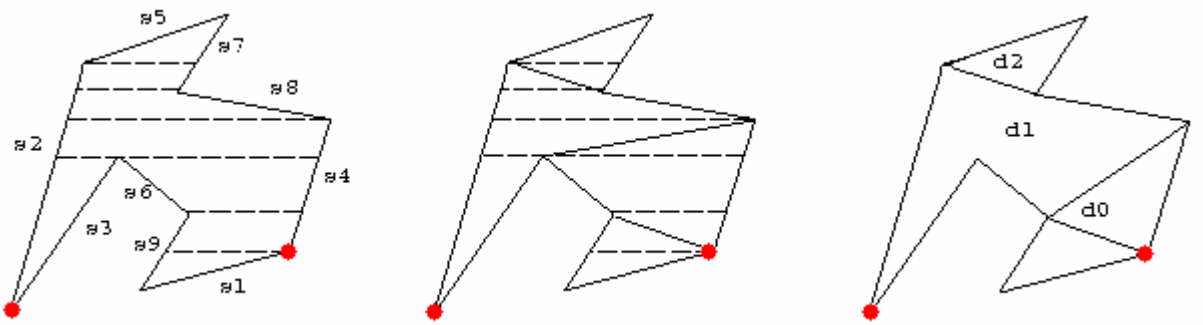

Figure 44. The Fast Polygon Triangulation technique for a polygon with red origin and destination points. This technique is incomplete and is not going to be explained, because we already can notice that it is not interesting. We can notice that it doesn't care about a sequence of consecutive triangles from the origin to the destination point.

Then, it was tried to build an own triangulation technique for this project instead of searching for existing ones.

3.3.1.3 Polygon averaging by using consecutive triangulation

This is our first created triangulation technique. Its procedure begins at the polygon origin point and ends at the polygon destination point, trying to ensure the constraint for all formed triangles to be consecutive. Then, we can compute the internal line of each triangle for forming the total polygon average.

In this procedure, trips are performed from the origin to the destination point along both polygon fragments, linking each vertex of a fragment to a vertex of the other one during their trips, excepting the origin and destination points that must not be linked to any vertex. The first link must happen for both next vertices of the origin point and the last link must happen for both prior vertices of the destination point.

All links during the trips are internal triangle sides of the polygon, and these links do not happen randomly. Every trip is performed by 'jumping' from vertex to (next consecutive) vertex in each fragment, and these jumps must not happen in both fragments at the same time. Each time a jump is done a link happens between the last reached vertices of both fragments.

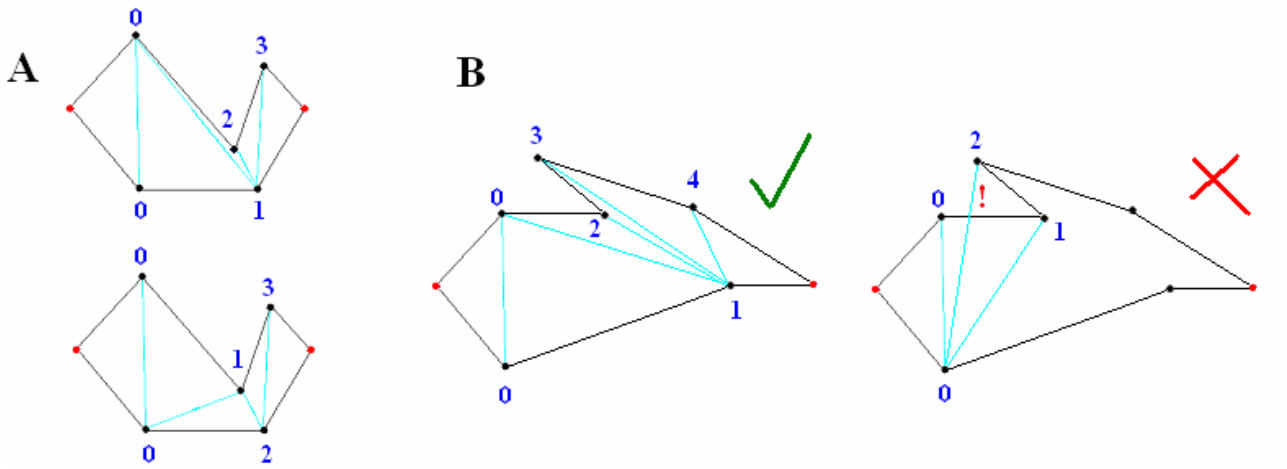

Figure 45. The example A shows that triangulations can become very different depending of the used (vertex to vertex) jumping order. Blue numbers represent these jumping orders. The example B shows that such order must not happen randomly, not just because we would like to obtain the best triangulation, but because triangles must be completely inner to the polygon and undesired triangles might happen accidentally (rightmost figure).

The only problem for this technique is to choose the next vertex to jump to. Sometimes it is not difficult to make a wrong choice and even basic triangulation constraints could be broken. The figure 45 illustrates these details.

The first imaginable solution is to retrocede in this procedure when wrong choices are detected for fixing them. However, this solution is quite troublesome and sometimes it might expend considerable time depending on the choices to fix. Therefore, another solution was planed and the next technique uses it.

3.3.1.4 Polygon averaging by using convex triangulation

This is the currently used technique for this project. Its result is almost as nice as a polygon exact average, but it is much simpler. Like the prior one, this technique also uses triangulation for dividing a polygon in triangles and it computes their inner lines for defining the polygon average.

These inner lines are just one segment for each triangle. Each of these triangle segments goes from the middle point of the internal triangle side to the middle point of the other internal triangle side (only the triangle side that is part of the polygon boundary must not be used). The inner line formed by these consecutive defined segments tends to form a nice average.

As we can see in the figure 46, the resulting polygon average use to be nicely distributed when using middle side points of well-chosen triangles. Moreover, it is very easy to find middle points of these triangle sides.

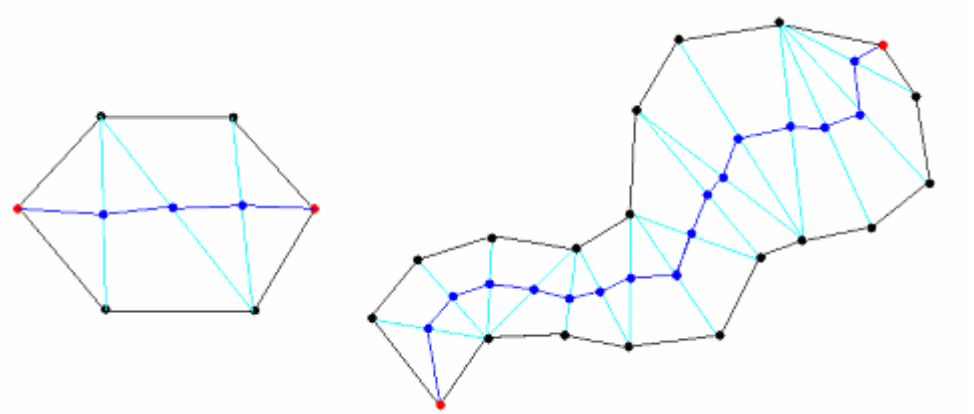

Figure 46. Fragment averages (blue lines) for a simple polygon (left) and a not very simple polygon (right) computed by using middle points of triangle sides (light blue lines). The red points represent the polygon origin and destination point.

The triangulation used here also have the same problems that the triangulation for the prior technique. The difference is that here we study better the vertices to be chosen instead of going back when wrong choices are detected. Then we first divide the polygon in convex sub-polygons and then triangulate them separately. Triangulation for convex polygons is definitely easier, thus the *convex triangulation* was the chosen technique to be used in the *path averaging process*.

Anyway we still may have problems using this technique. For instance, some polygons cannot be divided in consecutive convex sub-polygons. But these situations are not very frequent in this context, and all the problems we might have here do also represent problems for any prior technique.

Even when it was tried to solve every individual problem, new problems happened each time. There are infinite possible problems if we consider every possible weird path. But many of these situations are quite improbable to happen in the context of this project. Hence, it was decided to solve just those problems considered probable ones.

The possibility of other problems is not ignored, and the program tries to be prepared for dealing with any unexpected situation. The entire convex triangulation technique will be better explained during the complete path averaging process to be discussed next.

3.3.2 Path averaging by using convex triangulation

The *path averaging process* for this project is going to use the *convex triangulation* technique for computing the average for path pairs. We have been calling an *average path* to one that replaces other similar paths. The goal of this process is to compute an average path P_{avg} for two given original paths P_x and P_y. The path averaging process tries to keep the balanced shape for both original paths by following two conditions:

- P_{avg} contains all the intersection points of P_x and P_y.
- P_{avg} is always between P_x and P_y.

Resulting paths are not going to be cleaned during this process because the main aim here is to fight inaccuracy. Path cleanings can be performed later if it gets necessary for resulting paths from an averaging process.

The path averaging by using convex triangulation is divided in four procedures:

- Main procedure of the path average
- Polygon average procedure
- Sub-polygon definition procedure
- Convex sub-polygon average procedure

Each procedure is going to be explained next.

1) Main procedure outline:

The main procedure for this path averaging process simplifies it by dividing the tasks. It defines polygons formed by the parameter paths P_x and P_y (figure below). For that, it is necessary to define common origin and destination vertices v_o and v_d for both paths and to find their intersection points. Then we would have closed polygons, shaped by these two given paths and divided by intersection points.

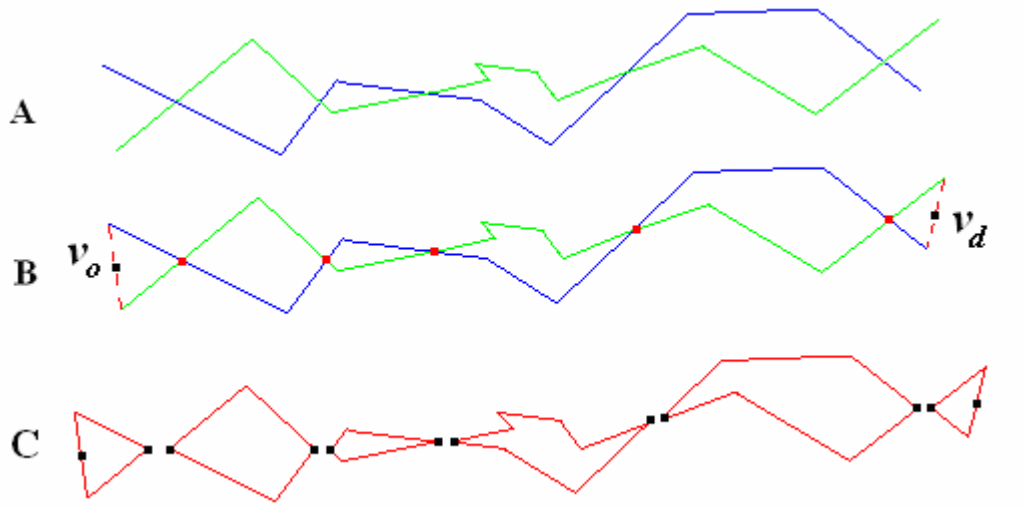

Figure 47. The upper Example A illustrates two similar paths to be averaged. The middle example B defines their intersection points (red) and a common origin and destination (v_o and v_d). The lower example C illustrates the resulting polygons defined by their common points.

The next procedure of the *path averaging process* computes the average of each individual defined polygon, and then it concatenates the resulting average fragments (in the same order) of their original polygons for having the final average path. The 'polygon average procedure' will be called each time it is wanted to average an individual polygon.

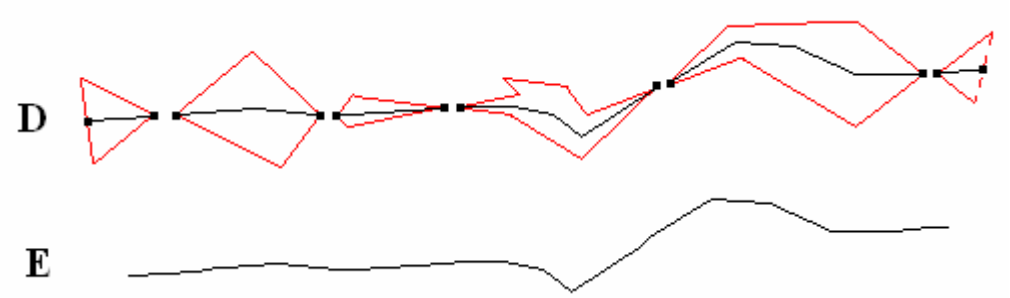

Figure 48. The upper example D illustrates the average of each polygon and the lower example E illustrates the resulting path average for the original given paths.

2) Polygon average procedure:

Here we only care about one individual polygon formed by two given path fragments whose origin and destination are common intersection points. This procedure has to compute an average fragment that goes from the origin point to the destination point along the inner area of the polygon (without crossing any of its boundaries).

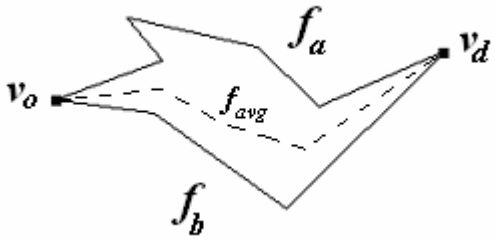

Figure 49. Ome polygon formed by two fragments (f_a and f_b, one from each initial path). They also have their common origin and destination point (v_o and v_d intersection points). An average fragment f_{avg} has to be computed as the polygon average fragment.

The tasks are going to be divided again in this procedure by splitting the given polygon in sequent convex sub-polygons. It is interesting to work with sub-polygons because it is possible to trace a straight line from any point to any other point of a sub-polygon without crossing any boundary, therefore, they are easy to triangulate. Then, sub-polygons can be divided again without problems. This polygon averaging process can result in better average fragments due to more interesting task divisions.

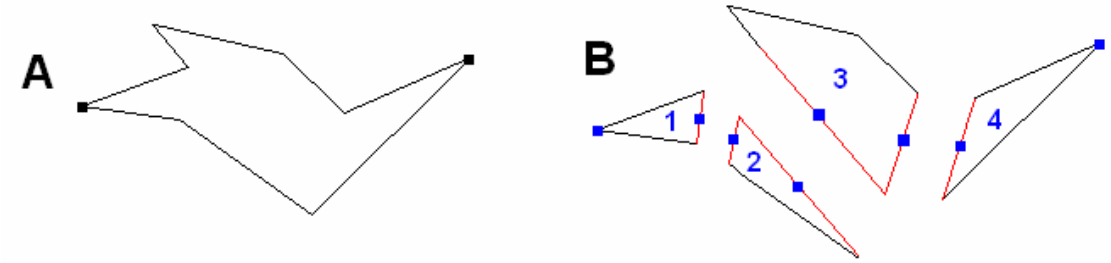

Figure 50. The example A illustrates a concave polygon to be split in convex sub-polygons. The example B illustrates the split performance following the constraints for this procedure. It results in four convex sub-polygons: the red lines are their common boundaries and the blue points are the middle points of the red lines. Later, these blue points are going to be considered the origins and destination points for the convex sub-polygons.

However, there are many possible strategies to divide a polygon in sub-polygons. It was chosen a strategy that reaches some constraints needed for a good polygon partition (figure above). These constraints and their reasons are listed below:

- Every sub-polygon must be convex: Convex sub-polygons are needed for the triangulation technique used in this process.
- Each sub-polygon must contain at least one vertex from each path to be averaged: The entire polygon average procedure must have influence from both sides (path fragments).
- Every vertex from both polygon fragments must be a vertex from some defined convex sub-polygon: All polygon vertices must be taken in account for the resulting polygon average.
- The sub-polygons must be consecutive along the original polygon: This means that every sub-polygon must have a common boundary to the prior and next one (if they exist), but not to any other sub-polygon. This is because a unique average line must go from the origin to the destination vertex without any branching.
- Sub-polygons should be as greater as they can, since they follow the prior constraints, and they must not overlap: Unnecessary polygon splits are avoided, as well as unnecessary repeating data due to overlapping sub-polygons.

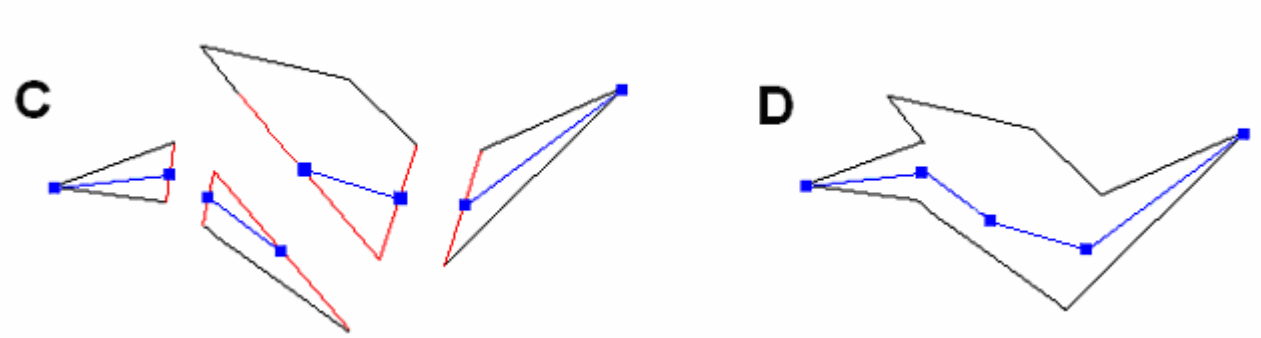

Figure 51. The left example C illustrates a hypothetical average for each convex sub-polygon and the right example D illustrates all them linked forming the polygon average.

This procedure calls a procedure to define its convex sub-polygons and another procedure to compute the average of each defined convex sub-polygon. The resulting fragments of the convex sub-polygons are concatenated as the polygon average fragments are for forming the resulting average path.

3) Sub-polygon definition procedure:

This procedure receives two convex fragments as parameters. Convex fragment (also called convex chain or *convex vertex list*) means a path fragment that would form a convex polygon whether we link its endpoints by a segment. In the next figure we can observe two (blue and green) *convex chains*.

Supposing that a segment links the *start points* of these chains and another segment links their *end points* (whether the points are not the same), we would have a polygon formed by two convex chains, but the formed polygon might be not convex yet. Both

convex chains form individually a convex polygon, but they must form a convex polygon 'together'. Therefore, the first step in this procedure is to check if both chains do form one convex polygon and fix the convex chains if they do not.

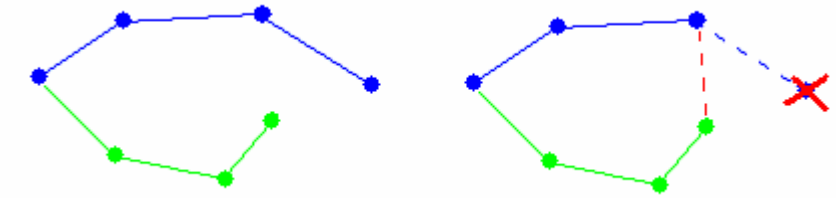

Figure 52. The left figure illustrates two convex chains with a common *start point* and different *end points*. The right figure illustrates a convex chain (blue) being fixed for forming a convex sub-polygon with the other (green) convex chain.

4) Convex sub-polygon average procedure:

This last procedure triangulates a convex sub-polygon (formed by two given convex chains) in order to obtain its average. The middle point of each internal triangle side computed for the sub-polygon is going to be a vertex for the sub-polygon average fragment.

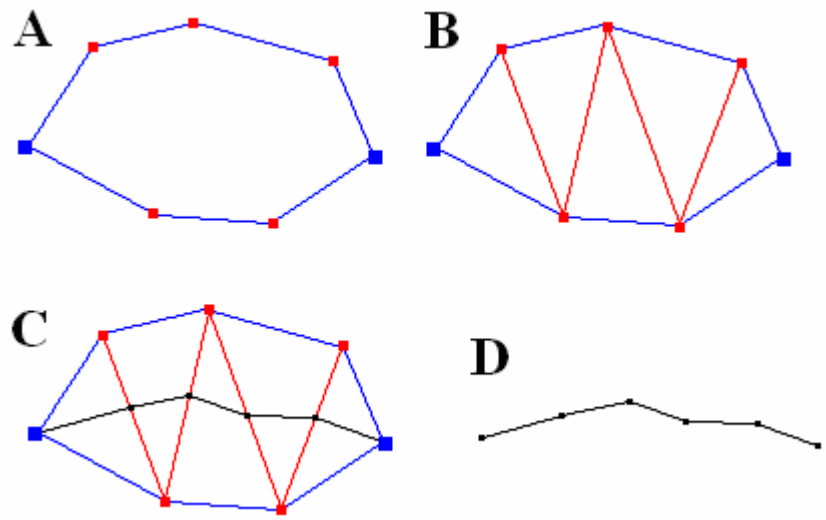

Figure 53. Example A: The convex sub-polygon to be averaged. The blue points are the origin and destination points. The red points are sub-polygon vertices. Example B: The triangulation of the sub-polygon. Example C: Middle (black) points of the internal triangle sides are used for the average. Example D: The resulting average fragment.

The algorithms for the four procedures of the complete *path averaging process* are going to be better explained with more details and discussed next.

3.3.2.1 Main procedure for the path averaging process

The algorithm of the first procedure for the *path averaging process* is going to be explained here. This is the main of four procedures for the entire process and it manages

the parameters to be passed to other procedures as well as the returned results for building the final path average.

At the beginning of this procedure, we have two similar paths P_x and P_y to work with, and we want to obtain the path resulting from their average. Actually, this manager procedure will not compute any average but it will call other procedures for doing it. This manager procedure will build the final average according to the returned results from other procedures.

Each step of this algorithm will be explained in details, always trying to offer an easy comprehension of it. The graphic of the figure 54 is an example of two paths that we wish to average and it would help to understand the algorithm.

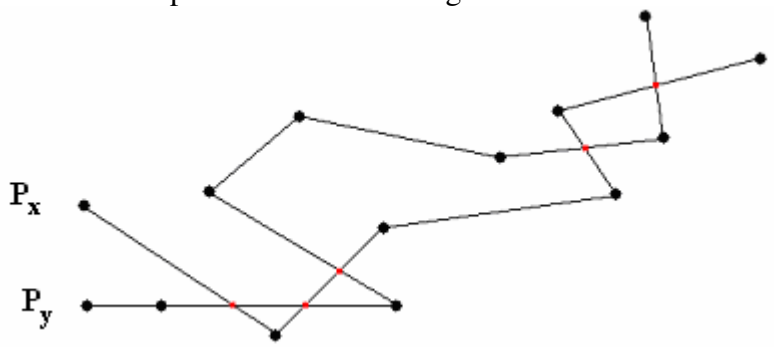

Figure 54. Two similar paths to be averaged: the path P_x and the path P_y. The black points are vertices of these paths and the smaller red points just represent the positions where both paths intersect: their intersection points.

Defining a polygon from two initial paths:

As we have said before, we expect to have two paths for this procedure then they need to be at least *unitary paths*. A path containing only one vertex is called a *unitary path* and this expression is going to be used further in this explanation.

The aim in this procedure is to divide the average job in parts, so we want to obtain simple polygons to be averaged individually. As we could notice in the last example figure, two paths don't use to start and/or end in exact equal positions (or points), thus normally we have to close the polygons at their extremities.

The initial path points, p_0^x and p_0^y, probably are not the same, as well as the destination points p_m^x and p_n^y. But we need both paths to have a common origin and destination points if we want every polygon to be closed.

We have to calculate a new common origin point p_o and add it at the beginning of both paths. The same happens to a new destination point p_d at the end of both paths. It is illustrated in the figure 55. The equations below are going to be used for creating the new origin and destination points.

$$p_o = \frac{p_0^x + p_0^y}{2} \qquad p_d = \frac{p_m^x + p_n^y}{2}$$

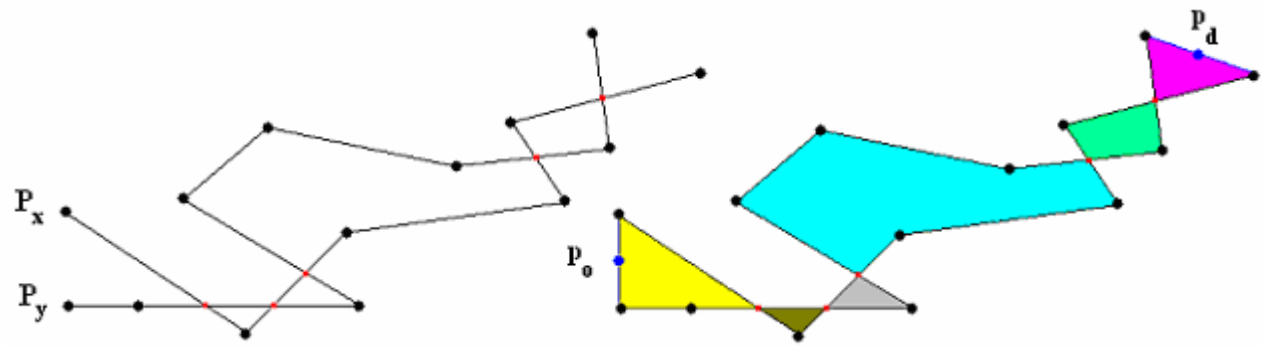

Figure 55. The (left) original paths and the (right colored) simple polygons they form. The (blue) common points were added to both paths for closing the first (yellow) and last (pink) simple polygons.

We also have to care about unexpected coincidences. If p_0^x and p_0^y, or p_m^x and p_n^y are the same points, then unnecessary points should not be added to the paths for closing already closed polygons. These 'repeated' points are going to be ignored.

The last and very improbable coincidences that may happen here is that the computed common point p_o and p_d might be equal (the same). In such situation, the average path is going to be a unique average point $p_o = p_d$, hence this point has to be immediately returned as the result because we don't have at least two intersection points that are necessary to continue this procedure.

Decomposing a polygon in simple polygons:

We want to average each simple polygon individually. For decomposing the original polygon in simple polygons, it is necessary to compute the intersection points of the paths P_x and P_y (red points in the prior figure), because these points define how we should fragment the original paths in order to split them correctly for obtaining each simple polygon.

When computing these points, we must consider that the intersection points may happen in a different order on each path, as it is illustrated in the figure 56. This undesired situation would cause troubles that make the *path averaging process* complex (e.g. some simple polygon might be inside another polygon and they cannot be averaged individually).

A pair of paths causing this situation will not be averaged but it will be treated as an 'average error' to be discussed in the last step of this procedure. It is possible to detect situations like this by checking the order that the intersection points happen in each path. This means that in an ordered search a new found intersection point should never be before the prior found intersection point in any path.

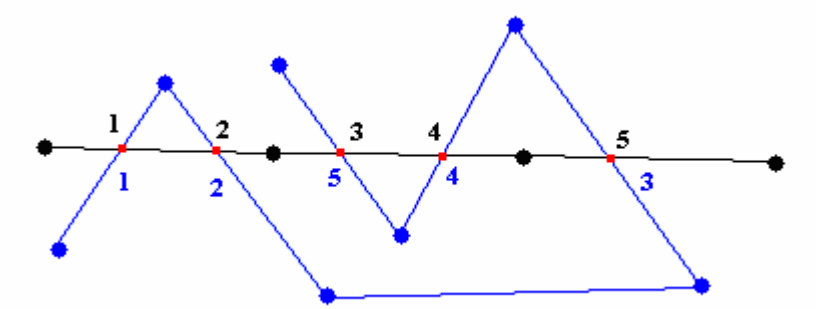

Figure 56. Blue vertices and segments of a path and black vertices and segments of another path. If we consider the (red) intersection points from left to right, they happen in a correct order for the black path but in a wrong order for the blue path. It causes a simple polygon inside another one.

Fortunately, this situation is not frequent at all because we expected *similar oriented paths*. This is the reason why we just consider these situations as errors instead of dealing with them.

After we have well defined intersection points and they follow a correct order for both paths, it is time to split both paths in fragments, where each fragment belongs to one and only one simple polygon.

Each simple polygon should be formed by two fragments: one from each path. Therefore, each path should have an ordered list of fragments, and the i-th fragment of a list must form a simple polygon together with the i-th fragment of the other list.

These two lists L_x and L_y, called ***fragment lists***, follow a sub-procedure that ensures two constraints:

- Each fragment list will contain every fragment (between intersection points) from it correspondent path (P_x or P_y).
- Each fragment list will be ordered in the same order as the fragments are in the path.

Averaging simple polygons:

In case that the *fragment lists* $L_x = f_0^x, f_1^x, ..., f_k^x$ and $L_y = f_0^y, f_1^y, ..., f_k^y$ are successfully created, we can define each existing simple polygon (that now we are going to refer to them just as polygons). The i-th fragments f_i^x and f_i^y of these lists build the i-th polygon formed by the paths P_x and P_y. Obviously, both paths are going to have the same number of fragments because the number of splitting *intersection points* is common for them.

Each pair of i-th fragments (f_i^x and f_i^y) forms a polygon that is sent to the next *polygon averaging procedure*. A returned fragment f_i^{avg} will be the average of these two sent fragments (or polygon). Each returned polygon average f_i^{avg} is added to the end of a path

that will be used to store the result of the whole averaging process[13]. This resulting path P_{avg} is going to be the *average path*.

Every polygon average is going to be added to the average path, except if some error happened during procedures called by this step.

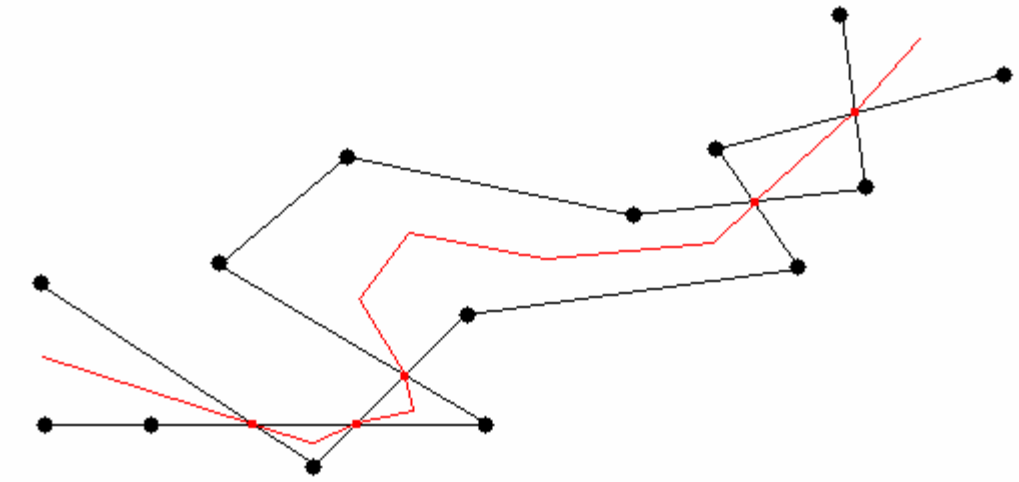

Figure 57. The resulting (red line) average for two (black) given paths.

Dealing with errors:

The average path P_{avg} should be returned as the result of this procedure, except if some error was detected during it. An error occurrence means that this procedure could not be completely performed due to some situation that it was not prepared to deal with.

Therefore, it makes no sense to return the average path P_{avg} if an error happened. What should be returned then? It is better to return back one of the given paths than to return nothing as result.

We know that the prior suggested return is not an average, but actually here we try to discard unnecessary paths and choose a substitute one. Hence, it is a good idea to return back the path that is better prepared for future averages instead of nothing. We would like to discard one of both given paths and return the other path that:

- Retains the most elaborated data.
- Would avoid more errors in the future

It was created a path member variable for the first item required above: the path ***weight***. This variable will inform us how many path averages it results from. Obviously, a path

[13] Observation: When a polygon average is added to the *average path* P_{avg}, the *start point* of this new average fragment f_i^{avg} is equal to the *end point* of the prior fragment f_{i-1}^{avg}, but it should not get repeated in the resulting path. These repetitions are going to be avoided.

will retain the more elaborated data the more times it has been averaged, and the tendency is a path to be closer to the real path whether its *weight* is higher. The path *weight* also helps us to obtain more accurate results from successful averages. This is going to be explained later in further procedures.

It was also created a path member variable for the second item required above: the ***counter-weight***. This variable will inform us how many times this path failed in average processes with other paths. Obviously, a path having a high *counter-weight* has some undesired situation that causes conflicts with other paths, and the tendency is a path to fail more times in the future whether its *counter-weight* is higher. Failed averaged processes would not increase path *weights* but their *counter-weights*.

The *total weight* of a path is the difference between its *weigh* and *counter-weight*. The *total weight* represents the resulting balance between its data accuracy and its conflictive factors. For instance, a path might result from many averaging processes but at the same time have conflicts that would cause failures in any future averaging process, so its *counter-weight* should overpass its *weight*, and a less elaborated path might replace it.

The *total weight* of given paths is what we will compare to determine which path should be returned back in case of average failures. The other path is just going to be ignored or discarded.

3.3.2.2 Polygon averaging procedure

The second procedure of the *path averaging process* is in charge of one simple polygon formed by two fragments f_i^x and f_i^y passed as parameters (and now we can drop their sub-index *i*). Here we have to split the polygon again in disjoint *convex sub-polygons* (colored sub-polygons in the figure 58), and we need to define new terms for the explanation of this procedure.

The aim of this procedure is to build an *average fragment* f^{avg} that lies completely inside the polygon built by the fragments f^x and f^y and goes from the polygon origin to the polygon destination point, replacing both original fragments.

Since now, the origin and destination points of the polygon are going to be considered as vertices, like the polygon vertices that they really are. But they keep being special ones: the ***origin vertex*** v_o where the *average fragment* f^{avg} is going to start, and the ***destination vertex*** v_d where the *average fragment* f^{avg} is gong to end[14].

[14] In the program code, the origin and destination vertices v_o and v_d keep being called as origin and destination points p_o and p_d. They are going to be called vertices just in the explanations for facilitating it.

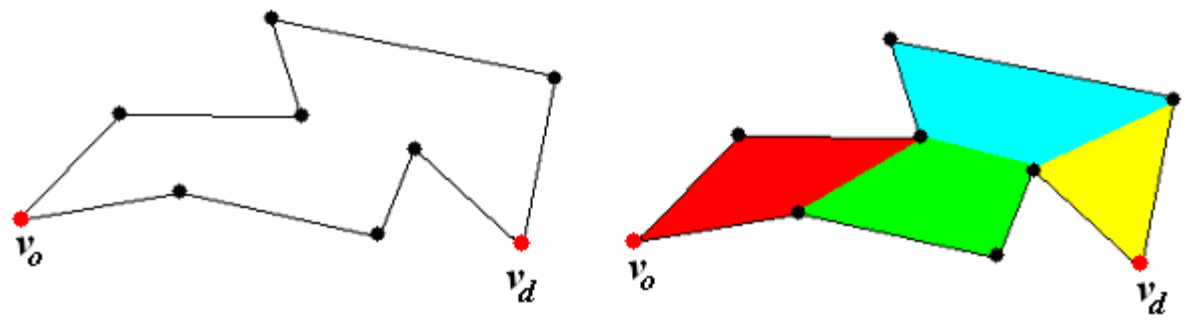

Figure 58. Left: A polygon formed by two given path fragments with red origin and destination vertices. Right: The same polygon split in disjoint convex sub-polygons. As we can notice, in a convex polygon every point from its boundary can reach any other point from the polygon with an internal straight line.

For splitting a polygon in disjoint convex sub-polygons, we are going to need reference vertices for both polygon fragments. These reference vertices will be called ***leader vertices*** lv^x and lv^y and they are used to mark the start limit of a convex sub-polygon. Their function is going to be better explained during the procedure algorithm.

There are also lists of consecutive vertices for each polygon fragment. These lists are going to be called ***convex lists*** and they use to contain the already mentioned *convex chains*. A ***convex list*** is a list of consecutive vertices (most of the times a *convex chain*) that forms no more than one simple convex polygon when its endpoints are linked (figure 59).

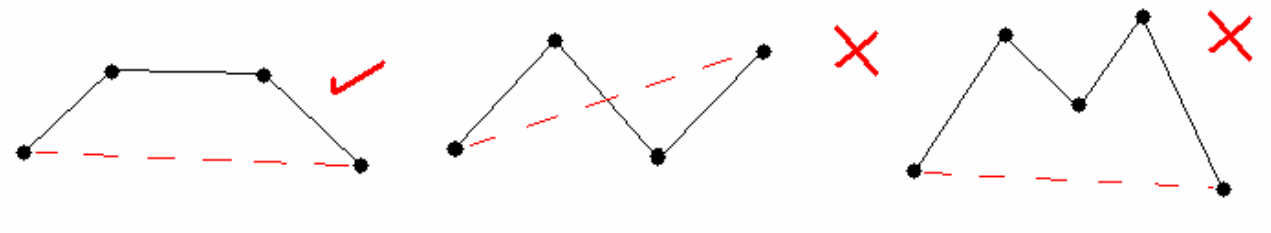

Figure 59. The right chain is a *convex list* because it forms one convex polygon when its endpoints are linked. The middle chain also forms (two) convex polygons but not just one. The right chain forms just one polygon but not a convex one. So, just the left chain is a *convex list* of vertices.

These special vertex lists for each correspondent polygon fragments are going to be the *convex lists* cl^x and cl^y. Their objective is not to form individual convex sub-polygons by themselves but to form one together. It is going to be better explained in the following steps of this procedure algorithm.

Treating initial situations:

At the beginning of this procedure, we automatically add the polygon origin vertex v_o to the *average fragment* f^{avg} as its first vertex. There is no doubt that this should be the start point for the polygon average.

If the destination vertex v_d is the next vertex of v_o in both fragments (there are no other polygon vertices: $f^x = f^y = \{v_o, v_d\}$), we don't have a polygon but just a segment formed

by these fragments. Both fragments are the same segment, and the average for just one segment is the segment itself. Then we just would have to add v_d to the end of f^{avg} and return it as result of this average.

Another possible situation that we should manage at the beginning of this procedure is the situation where not both but only one fragment (f^x or f^y) contains just the two vertices v_o and v_d, e.g. the polygon is a simple triangle.

This situation would cause problems for further procedures. We are going to understand them later, but now we can already observe one undesired condition in this example: We remember that internal triangle sides are needed for performing the *polygon average by using convex triangulation*. Do we have internal triangle sides in a simple three-sided polygon? Definitely we don't, but we can 'supply' it.

A new vertex can be added to the polygon without modifying it. This vertex v_{avg} is the average point of the fragment containing only two vertices (v_o and v_d), so v_{avg} will be in the middle of the segment defined by the origin and destination vertices (figure 60).

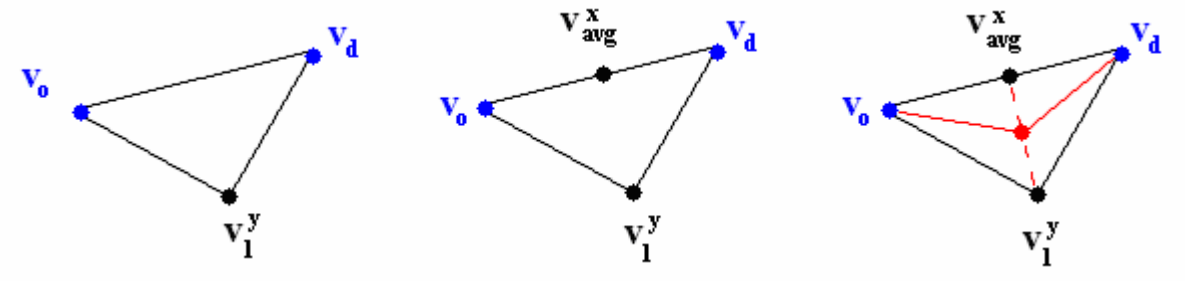

Figure 60. The left triangle is formed by two fragment f^x and f^y, where f^x contains only the polygon origin and destination (blue) vertices v_o and v_d. No nice average was possible, but (in the example at the middle) an average vertex v_{avg} was added to the fragment f^x between its two endpoints. A nicer (red) triangle average was possible for the right triangle by allowing the use of internal triangle sides for *convex triangulation*.

Without the supplied average vertex, the polygon average would be a straight line from the origin to the destination (blue) vertices, as we will be able to understand further, but this new average changes this situation permitting a much nicer polygon average.

Defining sub-polygons with convex lists:

We want to work with convex sub-polygons because they are much easier to average. Hence, we need to obtain them from the original polygons. The *convex lists* will help us to define them. But even if each *convex list* forms a convex sub-polygon individually, perhaps they don't form a convex sub-polygon together. We have to study the sub-polygons formed by *convex lists* and define new terms to be used by this step.

Let us begin by talking about *leader vertices* of sub-polygons. The *leader vertices* indicate where a new sub-polygon begins, and two casualties about the *leader vertices* are interesting (example B of the figure 61). First, at the beginning and only at the beginning of this *polygon averaging procedure* the *leader vertices* lv^x and lv^y are going to be the

same: the origin vertex v_o. Second, the *leader vertices* can never reach the destination vertex v_d, because no sub-polygon starts at the polygon destination.

The *leader vertices* lv^x and lv^y define the ***first internal segment*** of a sub-polygon (example A of the figure 61). This segment is very important because it defines the border between two consecutive sub-polygons: where the prior sub-polygon ends and another starts. But at the polygon origin the leader vertices are equal, so there is no segment that they define. However, it is comprehensible that there is no *first internal segment* in this situation because there is no prior sub-polygon.

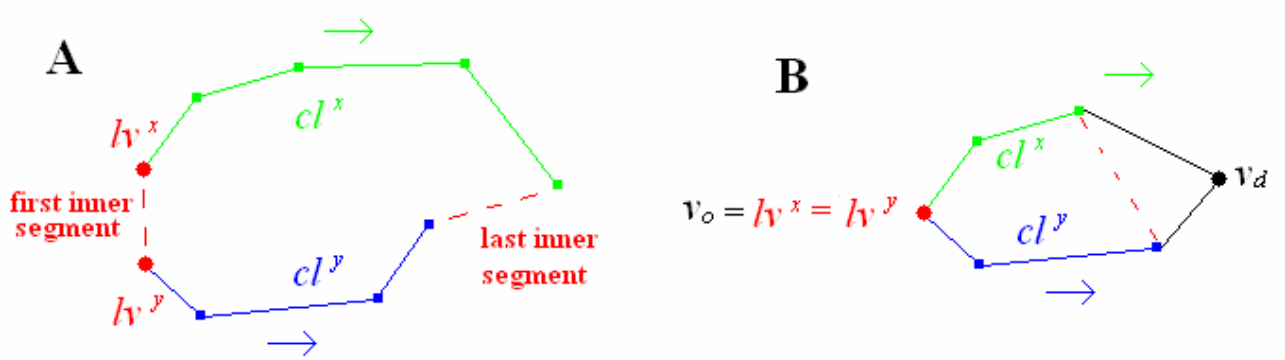

Figure 61. Example A: A sub-polygon formed by two convex lists (cl^x and cl^y) and all its parts: The leader vertices lv^x and lv^y are the first vertex of each correspondent convex list. They define the *first internal segment*. The *last internal segment* is defined by the last vertices of the convex lists. Example B: Here the *leader vertices* are the same for the first sub-polygon: the polygon origin vertex. The *convex lists* can never reach the polygon destination vertex.

Another 'internal' segment of a subpolygon is the ***last internal segment***. These *internal segments* are called 'internal' because they do not belong to the boundaries of the original polygon but they are into it, and they just limit sub-polygons. The *last internal segment* has a fundamental role later, when the *convex lists* are fixed for forming convex sub-polygons.

Until now it was not explained how the *convex lists* are built, and it is what we are going to do next. We want to build a *convex list* of consecutive vertices that begins in a determined point that will be its *leader vertex*, and we don't know yet where this list will end. This means that initially a *convex list* contains only its *leader vertex* and we are going to add more consecutive vertices to it.

Polygon vertices are going to be loaded to the *convex lists* separately, then we can explain this step for just one *convex list*, let say cl^x, and the same is going to be done for the other list cl^y.

A *convex list* cl^x must contain all the consecutive vertices of its fragment f^x, from a chosen *leader vertex* lv^x to the last vertex (after it) that forms a point or a segment or a *convex chain*. Initially, a *convex list* must contain the maximal number of consecutive vertices that it is allowed to: it must not form a concave figure or more than one closed figure. For that, every formed segment by consecutive vertices of the *convex list* must turn to the same side, not considering those segments that do not turn with respect to its

prior segment. The turning of the first *convex list* segment will be considered with respect to the *first internal segment*.

There are two exceptions for the *convex list* to be not maximal, and in the first case the *convex list* must contain just its *leader vertex*. The second case will be explained after. Let us try to explain the situation for the first case:

Two *convex lists* are created in the attempt to build a convex sub-polygon, even though some *convex list* has to be fixed later by removing its last vertex (example A in the figure 62).

But sometimes the first segment of a *convex list* and the *first internal segment* linked to it form an internal angle α that is greater than 180° (example B of the figure 62). Definitely this *convex list* cannot form a convex sub-polygon with any other *convex list*, thus it has to be fixed. The *first internal segment* has a final value that cannot be changed for fixings because it also belongs to a prior convex sub-polygon that did not offer any problem. Therefore the first segment of such *convex list* has to be discarded, and the only way for doing it is by removing every vertex from this *convex list* but its first vertex.

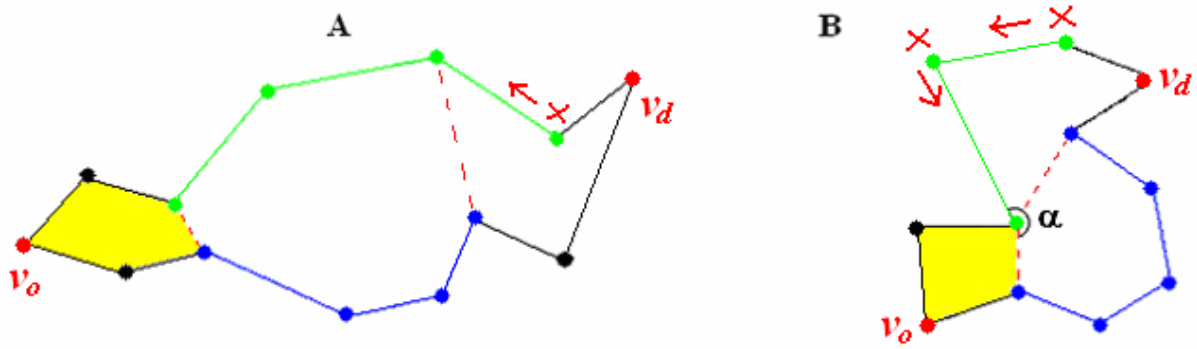

Figure 62. The yellow polygons represent prior convex sub-polygons that was already defined. The red dotted segments are *first internal segments* (leftmost) and *last internal segments* (rightmost) after the *convex lists* have been fixed. In the example A, it was necessary just to remove one last vertex from the (green) *convex list* to make the formed sub-polygon convex. But in the example B only the first green vertex could remain in its *convex list* for making the formed sub-polygon convex, because the first internal angle α of its *convex list* was greater than 180°.

As it has been said, after the *convex lists* cl^x and cl^y are defined in this step, sometimes they still have to be fixed because the sub-polygon that they form is not convex yet. As we can see in the example A of the figure 62, we have both *convex list* (green and blue) but the sub-polygon formed by them is not yet a convex sub-polygon. The green *convex list* has to be fixed for obtaining the convex sub-polygon we wish. After fixing it, the last green vertex does not belong to the *convex list* anymore.

In the example B of the figure 62, it is illustrated the only possible exception when building *convex lists*: Even before fixings, the built *convex list* will not be maximal due to this exception. In this example, the *first internal segment* and the first segment of the green *convex list* forms an internal angle α greater than 180°. But no convex sub-polygon may have an internal angle that is greater than 180°. So, even if there were a nice green *convex list* to be defined, only its *leader vertex* will belong to the fixed list in the situation

that its internal angle (defined by the *first internal segment* and the first *convex list* segment) were greater than 180°.

The question now is: how to check if this undesired angle happened? We shall have the first segment of a *convex list* and a *first internal segment* linked to it. They form two angles like every pair of segments. Both might be the internal one, so: how to check it?

There is an answer for this question but we are going to need the polygon segment that is prior to the *convex list*. The answer is in the next paragraph, and it is illustrated in the figure 63.

Let us consider the first segment of a *convex list* and a *first internal segment* linked to it. If the segment defined by their non-common endpoints intersects the prior segment to the convex list, then the first internal angle formed by this *convex list* is greater than 180º, otherwise it is not.

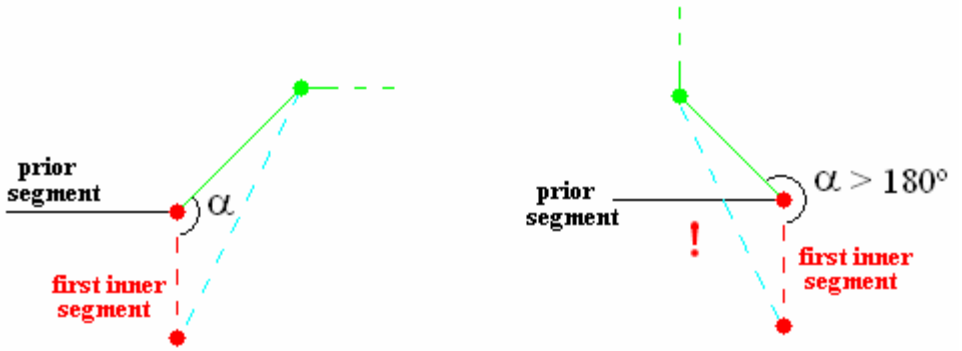

Figure 63. It is illustrated (in each example) the first segment of a (green) *convex list*, a (red dotted) *first internal segment* linked to it, and the (black) polygon segment that is prior to the *convex list*. The light blue dotted segment is defined by the non-common endpoints of the red and the green segments. If the light blue segment intersects the black segment (right example) then the first internal angle α is greater than 180º, otherwise (left example) it is not.

This exception will never happen for a very first convex list (or first sub-polygons) because there both leader vertices are the same *polygon origin vertex*, then there is no *first internal segment* for forming angles with.

The first exception for maximal *convex lists* has been explained and solved, but there is still a second exception for it: *Spiral lists* are also undesired. This is what the next explanation is about.

Actually, a *spiral list* is not a real exception because it is not a *convex chain*! But if we consider just the turnings of consecutive segments when constructing *convex lists*, we can fall in this mistake. If we link the endpoints of a spiral we are not going to have just one simple convex polygon. Therefore, even though all the consecutive segments of a *spiral list* turn to the same side, not all these segments are desired. Let us formally define a *spiral list*:

A *spiral list* is a list of vertices that forms at least three consecutive segments that always turn to the same side or do not turn with respect to the prior one, and if the first and last vertices of this list are linked then it forms a segment that intersects some of the

consecutive list segments or it forms a non-convex polygon (example A of the figure 64). The segment formed by linking these list endpoints is going to be called its *link segment*.

For avoiding *spiral lists*, we must control no 'spirality' during the construction of a *convex list* by checking one condition each time that a new vertex is added to it: "The turning from the last list segment to its *link segment* must turn to the same side than from its *link segment* to the first list segment" (examples B and C of the figure 64).

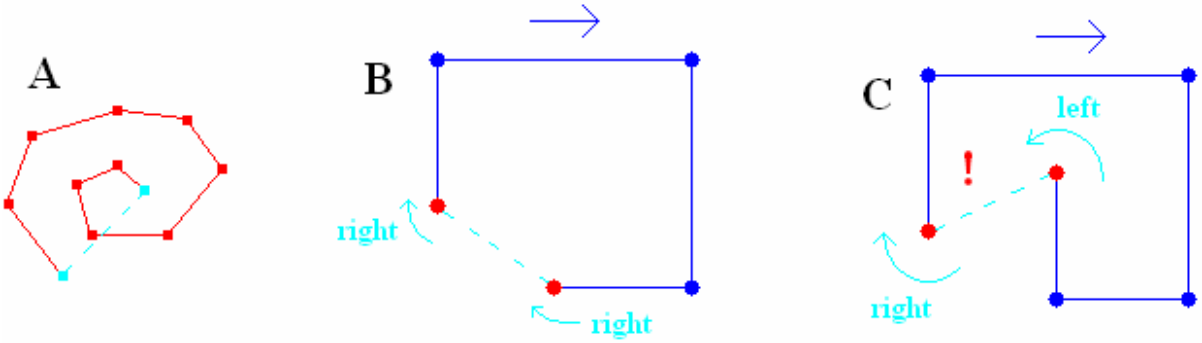

Figure 64. Example A: a *spiral list* with its endpoints linked by a *link segment*. Example B: the turning from the last segment of a vertex list to its *link segment* is the same than from its *link segment* to the first segment of the list, so it is a *convex list*. Example C: the turning from the last segment of a vertex list to its *link segment* is not the same than from its *link segment* to the first segment of the list, so it is a *spiral list*

Remembering the aim of this step again, the objective here is just to define *convex lists* but not to fix them yet. The fixings where mentioned just to explain two possible exceptions for not adding the maximal number of vertices to *convex lists*: 1) *spiral lists*, which really are not *convex lists*, and 2) first internal angles greater than 180º.

The last casualty we have to consider in this step is not an exception but a possible error: *Internal immediate returns*. We already defined *immediate returns* and tried to solve them during the *cleaning process*, but we agreed that sometimes they are quite difficult to be detected and we must be prepared for unexpected casualties.

Here we are going to work with the most undesired kind of *immediate returns*, and they happen when some next segment gets in a prior defined sub-polygon! This is why it is called ***'internal' immediate return*** (figure 65). We are not going to find solutions for it (once it is not responsibility of the *averaging process*) but we are just going to detect them and send an 'average failure alert' whether it is detected. This *averaging process* is not prepared for dealing with it.

However, just to detect it might be difficult. This detection is not difficult if a *first internal segment* is intersected. Such intersection should never happen in normal situations, thus it is obvious that something wrong happened.

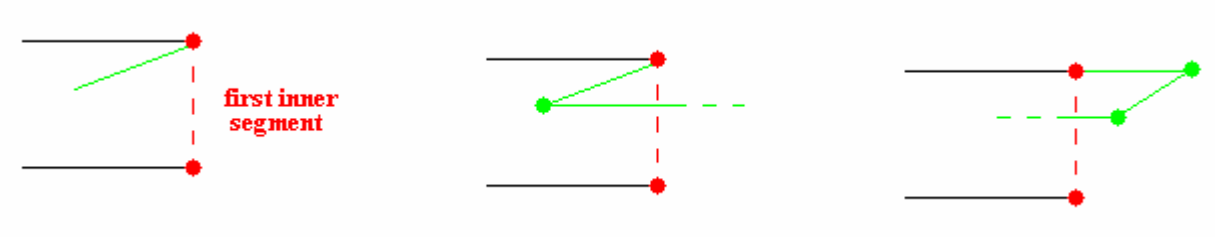

Figure 65. *Internal immediate return* occurrences. They are obvious when a *first internal segment* is intersected (middle and right examples), but they are quite difficult to be detected if the *cleaning process* already failed and no *first internal segment* is intersected (left example).

We are not expecting *internal immediate returns* to happen without intersection of a *first internal segment*, and this undesired intersection is the only thing we are going to check, because it is quite hard to detect *internal immediate returns* for other situation where the *first internal segment* is not intersected.

But then there is a little risk for detection failures of *internal immediate returns* as well as a detection failure could happen in the *cleaning process*.

Fortunately, these detection failures are not frequent and we suppose that, anyway, if an *internal immediate return* happened and it was not detected, some future problem would be detected later due to it or the happened mistake would not be considerable. For instance, a little error might happen in the end of a path, but we always expect weird situations at path extremities and a future stitching process might care about path extremities when linking them. If some path extremity is not linked, a little error in it would not be considerable.

Fixing convex lists for obtaining convex sub-polygons:

The prior step defined pairs of *convex lists* to form convex sub-polygons with, but perhaps this sub-polygon they form is not convex yet and some convex list has to be fixed. We are going to explain how to decide which vertices to discard from *convex lists* for obtaining a convex sub-polygon.

First we have to consider that if both *convex lists* contain only the *leader vertex* (lv^x and lv^y correspondently), a convex polygon cannot be created just by them and these list cannot be fixed. It is not permitted to remove elements from a unitary list (that contains only one vertex).

Three situations may cause both *convex lists* to have only the *leader vertex*. The first situation happens when both *convex lists* were reduced to unitary lists because of fixings or because both they form an angle that is greater than 180º with the *first internal segment* (figure 66).

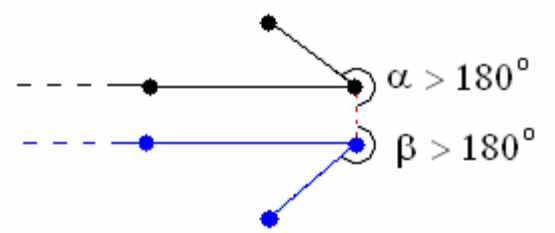

Figure 66. A first *internal segment* forming an angle grater than 180º with both next *convex lists*. In this situation both *convex lists* would be unitary, containing only the leader vertex. The procedure would not be able to continue.

If both *convex lists* are reduced to unitary lists, containing no segments, this means that we cannot form a next convex sub-polygon. This situation should not happen if the *cleaning* process avoided *immediate returns* and the path *similarity detection process* happened successfully. It is an unexpected failure that we cannot deal with, so this procedure should inform that this polygon is not able to be averaged by this process.

The second situation happens when the next vertex of both *leader vertices* is the destination vertex v_d, which can never be reached by them. In this case, the polygon averaging has been successful and only the destination vertex v_d needs to be added to the *average fragment* before returning it as the result of the polygon average.

The third and last possible situation happens when one *convex list* is unitary because the destination vertex v_d is the next vertex of its *leader vertex* but the other *convex list* is unitary due to fixings or the mentioned angle greater than 180º.

This conflictive situation can happen for polygons that still might be averaged. At the end of the polygon, a *convex list* cannot advance anymore and the other one formed an angle greater than 180º but nothing irremediable. In the attempt of allowing this procedure to continue normally, we are going to add an extra vertex to the *convex list* that could not advance anymore, between its *leader vertex* and the destination vertex v_d, like it was done for a particular situation at the beginning of this *polygon averaging procedure* (figure 60). Then we must go back to the prior step, redefine new *convex lists* and continue this procedure normally.

This vertex addition might weirdly happen many consecutive times, and it might be consequence of some prior non-detected error. So, this extra vertex addition should be available just a limited number of times, otherwise a procedure error is going to be considered.

Now let us suppose that we have a normal pair of *convex lists* cl^x and cl^y where at least one of them is not unitary, thus some sub-polygon is formed by them. If this sub-polygon is not convex yet we must make it convex by fixing some of its two *convex lists*.

So, it is time to fix *convex lists* (if necessary) for obtaining a convex sub-polygon and then to compute the average of the obtained sub-polygon. But this procedure is not going to do these jobs. It is going to call other procedures for doing it: The first called *sub-polygon definition procedure* is in charge of *convex list* fixings for obtaining the convex

sub-polygon, and the next called *convex sub-polygon averaging procedure* finds the sub-polygon average as its name implies.

For each *convex list*, the *sub-polygon definition procedure* is going to be called and, after it is performed, we will have two lists for a convex sub-polygon. Some vertices could be discarded from the *convex lists* when fixing them, but of course that these vertices are not discarded from the entire original polygon: they are discarded just from the *convex lists*. The next *convex lists* to be defined are going to reuse the vertices discarded from prior lists.

The *convex sub-polygon averaging procedure* is called for each pair of fixed *convex lists* (that forms a convex sub-polygon). The resulting average of such sub-polygon is going to be added to the polygon *average fragment* f^{avg}.

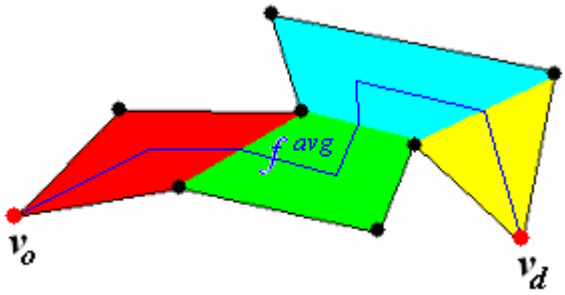

Figure 67. *Polygon Average fragment* resulting from the partial averages of each obtained convex sub-polygon.

After the called procedures, the *leader vertices* are set up with the value of the last vertex of each fixed *convex list*, and these lists are cleared for defining next convex sub-polygons. The steps for '*defining polygons with convex lists*' and '*fixing convex lists for obtaining convex sub-polygons*' are going to be repeated until the last convex sub-polygon has been defined and its average has been added to the polygon *average fragment* f^{avg}.

Remember that this procedure can be interrupted due to unexpected situations (errors) happened during called procedures or during this procedure itself. In such situation, this procedure just needs to inform somehow to the main procedure that an error happened.

3.3.2.3 Sub-polygon definition procedure.

The goal of this procedure is to control the convexity of one convex polygon defined by two given *convex lists* cl^x and cl^y, and to fix some or both *convex lists* whether necessary. Let begin with the first step.

Fixing possible initial exceptions:

The first situation we are going to analyze is simple and important: "If two *convex lists* turn to the same side, they cannot form a convex polygon". It is easy to observe in the next figure and this problem is also easy to be solved.

The only and easy way to avoid two *convex lists* turning to the same side is to convert one of these lists in a single segment, because a single segment does not turn to any side. The *convex lists* may keep being not convex, but the reason why it happens now is not because they turn to the same side. Even if they do not form a convex polygon yet, we will be able to fix them normally in the next step: *fixing convex lists*.

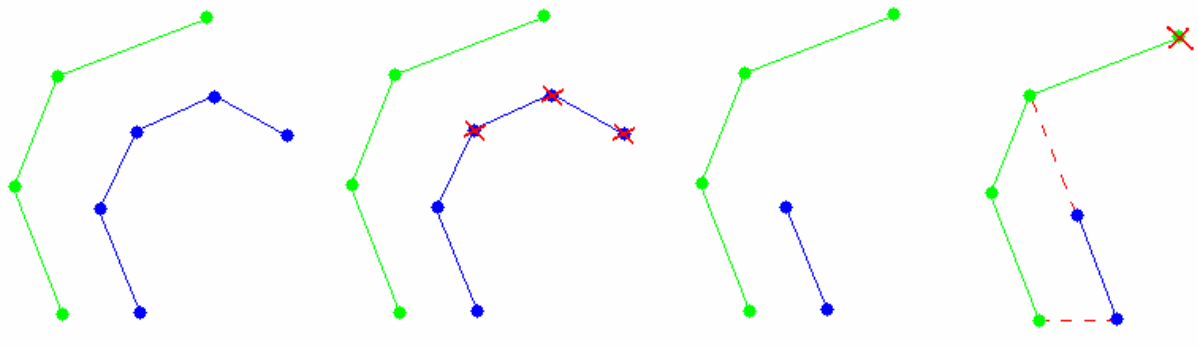

Figure 68. Two *convex lists* turning to the same side that obviously don't form a convex sub-polygon. The inner *convex list* is reduced to a single segment. It still doesn't form a convex polygon with the other *convex list*, but then the other list can be fixed for reaching the wanted convexity.

Due to practical reasons, not any list must be the one reduced to a segment. The inner of them both must be the reduced one. This means: "if they turn to the left then the leftmost list must be reduced to a segment, but if they turn to the right then the rightmost list must be reduced to a segment". This choice is done to avoid the definition of many little convex sub-polygons instead of defining few but bigger ones that are also convex.

Fixing convex lists:

If both *convex lists* do not form a convex sub-polygon, some internal angle of the formed sub-polygon has to be greater than 180°. At this moment, only four internal angles are able to be greater than 180° if the given lists are really *convex lists*. We also must remember that *convex lists* having the first internal angle greater than 180º do not reach this procedure. Then, actually only two internal angles may be greater than 180º, and these angles are formed by the last segment of each *convex list* and the *last internal segment* of the sub-polygon. The figure 69 is an example where these two angles are illustrated.

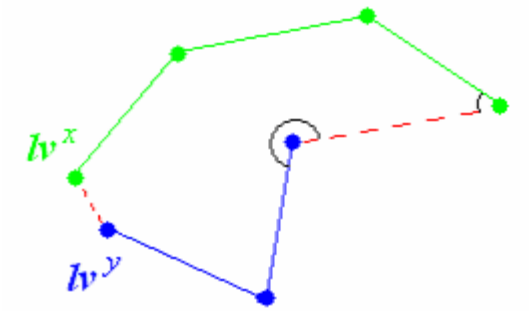

Figure 69. Internal angles of a sub-polygon that are possibly greater than 180º are formed by its *last internal segment* and the last segment of each *convex list*. Here, just one of these segments is greater than 180º but both might be so.

Each *convex list* is going to be analyzed with respect to the *last internal segment* for knowing if the other *convex list* must be reduced or not. If the angle formed by the last segment of a *convex list* and the *last internal segment* of the sub-polygon is greater than 180°, then this list must not be reduced but we must discard the last vertex from the other *convex list* until the sub-polygon formed by both lists gets convex (figure 70). The vertex eliminations are done one by one, and each time it happens this step has to begin again for checking if the sub-polygon got convex

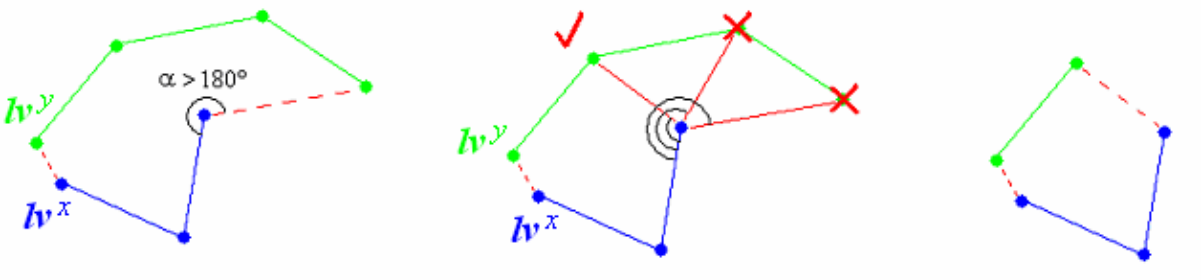

Figure 70. Left: Two *convex lists* starting by their *leader vertices* lv^x and lv^y where one of them forms an angle greater than 180º with the *last internal segment*. Middle: the last vertices from the other *convex list* are discarded one by one and the formed angle is reduced each time. Right: The formed angle is no greater than 180º anymore and the *convex list* is fixed: the sub-polygon got convex.

The last segment of a *convex list* and the *last internal segment* of the sub-polygon form two angles, and once again we cannot easily know which angle is the internal one. Therefore we are going to use turning sides instead of using angles. As we could see in the figure 70, the polygon turning side changes where the internal angle gets greater than 180º [PO].

Therefore, we need to compute the turning side of each convex list for checking if this is the same than the one from its last segment to the sub-polygon *last internal segment*. For computing each *convex list* turnings, we will need to get the first segment of each *convex list* and the sub-polygon *first internal segment*.

But perhaps we have no *first internal segment*: the *leader vertices* of cl^x and cl^y might be the common *polygon origin vertex* v_o, and common vertices form no segment. In this situation, we would have to use the first segment of both lists for computing their common turning side.

Unfortunately, one of both *convex lists* may have no segments (is a unitary list). First we are going to suppose that both lists are not unitary, and further we are going to explain the solution for unitary lists.

There is another problem that may happen when trying to compute the turning side for non-unitary *convex lists*. Perhaps the chosen segments for computing it do not turn to any side, and they would not inform us the main turning side of any *convex list*! A solution is to ignore the chosen first segment from each *convex list* and take the next segment (if some *convex list* has a next one) for trying it again. Of course that an error might happen

whether we convert both *convex lists* to unitary ones, but we expect this improbable situation not to happen and we are going to interrupt this procedure if it happens.

Once we already have the turning side of a *convex list*, we can check if it is equal to the turning side from the last segment of such list to the sub-polygon *last internal segment*. If both *convex lists* do so, no changes are necessary because they already form a convex sub-polygon. But if some *convex list* does not, then the other *convex list* must be reduced to fix the internal angle that is greater than 180º.

An error may happen whether both *convex lists* form an angle that is greater than 180º with the sub-polygon *last internal segment* (right example in the figure 71). These lists can be fixed normally but it would cause future *internal immediate returns*. It is recommended to react to this error as soon as it is detected.

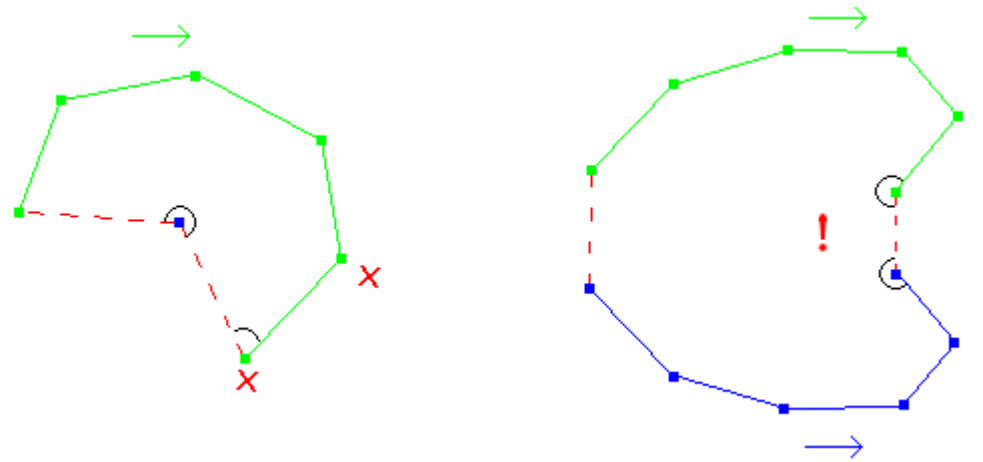

Figure 71. Left: One (blue) *convex list* is unitary, and then the (leftmost red dotted) *first internal segment* is used to be compared with the (rightmost red dotted) *last internal segment* for checking the reduction necessity of the other (green) *convex list*. The red crosses mean the vertices that must be discarded for obtaining a convex polygon. Right: Both *convex lists* form an angle greater than 180º with the *last internal segment*. It is an error that can be fixed but will persist later as an *internal immediate return*.

Now we have to analyze the situation of having one unitary *convex list*, remembering that having two unitary lists is not allowed here. This situation has some problems but it also has two advantages: 1) In this situation it is not possible that both lists need to be reduced, and 2) if some list is unitary and the sub-polygon is not convex we already know that the non-unitary list is the one that must be reduced. An example is illustrated in the left figure 71.

When dealing with unitary lists, we are going to check if two turning sides are the same. The first turning is from the last segment of the non-unitary *convex list* to the sub-polygon *last internal segment*. The second one uses to be from the *first internal segment* to the *last internal segment*. However, in weird but possible situations we might also have no *first internal segment* and it would have to be replaced by the first segment of the non-unitary *convex list*.

Then, the convexity may be checked also for sub-polygons formed by unitary *convex lists*, and the non-unitary list must be reduced if the two turning sides are different. If

some compared turning is null (have no side), we are going to consider that the turnings did not change.

If a *convex list* must be fixed, then its last vertex is discarded and this step is repeated to check sub-polygon convexity again. Once the sub-polygon formed by both lists is convex, this procedure is finished.

3.3.2.4 Convex polygon averaging procedure

This last procedure just has to compute the simple average of a convex sub-polygon formed by two *convex lists*.

Before beginning the sub-polygon averaging procedure, we have to define *turn vertices*. We are going to advance in each *convex list* from its first vertex to its last vertex, and each time we advance one vertex in some list we have to find a new average vertex for the last chosen vertices of each *convex list*. Hence, we need to know which vertex was the last chosen one for each list. They are going to be the *turn vertices* v_t^x and v_t^y (one for each list). They will represent the two vertices we are going to be working with.

Average initialization:

Initially, each *turn vertex* is going to be the first vertex of its correspondent *convex list*, and we have to check if these first vertices are not equal, i.e. they aren't the *polygon origin vertex* v_o.

If they are the same, we must advance to the next vertex of 'both' *convex lists*, because the origin vertex v_o has already been added to the polygon *average fragment* f^{avg} (and the average between the origin vertex v_o and any second vertex of a polygon is on its own boundary). We are going to compute the average for the present *turn vertices* and add it to an average fragment f^{sub} for the sub-polygon, excepting if an exception happened.

First, we had to check if some *convex lists* is unitary, having only the origin vertex v_o. In this case, not both *turn vertices* could advance. However, we can advance twice in the non-unitary *convex list* for solving this exception (figure 72). It is equivalent to advance just once in the list but compute no average vertex, ignoring the second vertex of the non-unitary *convex list* for avoiding an average vertex to be on the polygon boundary. The non-unitary *convex list* must have at least three vertices whether the *leader vertices* are equal and the other *convex list* is unitary.

In a second exception, the average vertex might be exactly equal to the prior vertex added to the polygon *average fragment* f^{avg}. This exception is possible to occur only if this is the first polygon of the whole *path averaging process*, due to the closure of the first polygon. In this exception, we do not add the average vertex to f^{sub} but we advance the

turn vertices in both *convex lists* again. There are no more exceptions in this procedure then we go to a next step.

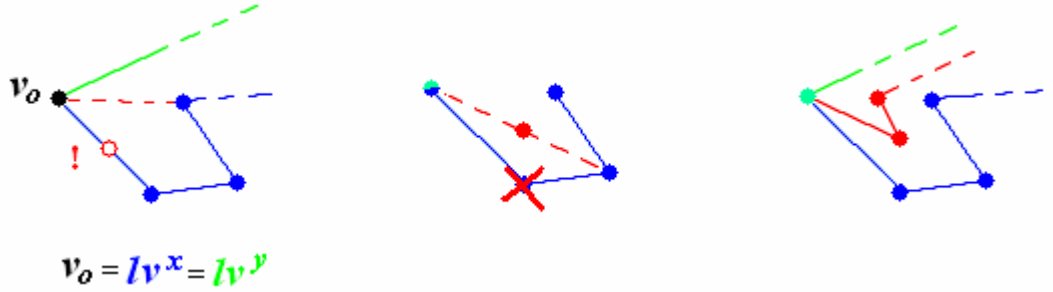

Figure 72. Left: the polygon *origin vertex* v_o is the *leader vertex* of two *convex lists*, and the green *convex list* had to be reduced to an unitary list containing only v_o. The first average vertex (red empty circle) would be on a boundary segment if the blue *convex list* advances to its second vertex. Middle: The blue *convex list* advances to the third vertex for computing the first average vertex, ignoring the second vertex. Right: The resulting averaged is quite acceptable.

Computing average vertices:

Here we will advance to the next *turn vertex*, compute a new average vertex for the convex sub-polygon and add it to the fragment f^{sub}. This step will be repeated until it cannot advance anymore in any *convex list* when both *turn vertices* reached the last vertex of their lists.

Now we just need to understand how to decide which *turn vertex* must advance in its *convex list* to compute a new average vertex. This entire step is illustrated in the figure 73.

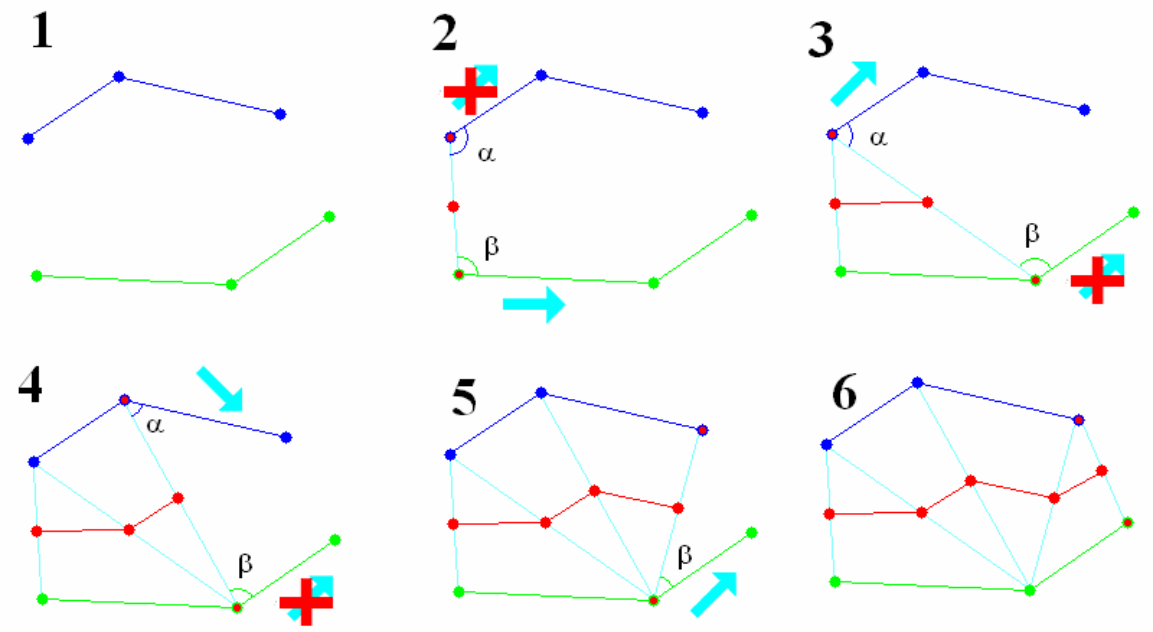

Figure 73. This is an example of the entire step for computing (red) average vertices for a convex sub-polygon formed by two (blue and green) *convex lists*. The (light blue) arrows represent a chosen segment to advance on for getting a new *turn vertex*, and the crossed arrows represent the discarded choice. The light blue segments are internal segments formed by *turn vertices* and the angles α and β will be used to choose the next vertex advancement. The red fragment is the resulting sub-polygon average fragment. In this example, it is assumed that every vertex has the same *weight*.

The *convex list* segment where this step must advance on for computing a new average vertex is the next segment to some present *turn vertex* v_t^x or v_t^y that forms the lower angle with the internal segment between them both.

Of course that if some *convex list* already has reached it last vertex, it has no next segments to form angles with, and then the other *convex list* is going to be the chosen one for advancing on.

Each average vertex would be the middle point of the segment that links the present *turn vertices* v_t^x and v_t^y if we didn't use *weights* or both *turn vertices* had the same *weight*. But it is not always the exact middle segment point because we use *weight*s that vary, and the position of the average vertex on this segment will depend of the *total weight* of each *turn vertex*. In the implementation of this project program, we just consider *path weights*, and then each *turn vertex* has the same *weight* than its path, of course.

Other possible suggestion is to give *weights* for each vertex individually, what would make possible to define different *weights* for disjoint fragments of the same path. However, in this project we just use *weights* and *counter-weights* for the whole path, and the two distances from an average vertex to its original vertices are inversely proportional to the *total weight* of the correspondent path of each original vertex, where *total weight* is the difference between *path weight* and *path counter-weight*.

3.3.3 Path cleaning after using convex triangulation

A path cleaning may be needed, after some types of averaging processes, for discarding unnecessary data that can be added to *average fragments* during the *path averaging process*.

The *path cleaning process* by using *convex triangulation* definitely needs a path cleaning for its resulting *average path*. The number of vertices for the *average path* is almost the sum of vertices from both original paths.

If one original path have *m* vertices, and the other original path have *n* vertices, and no *intersection points* coincide with path vertices, then we will have an *average path* containing $m + n - 1$ vertices.

Many of these vertices might be unnecessary data and a *simple cleaning* for discarding excessive points would be very welcomed.

3.4 Fragment averaging process and stitching process

Once we have replaced all paths that are completely similar, there are no unnecessary paths to manage anymore. Every present path should contain data that any further path average can replace. On the other hand, it doesn't imply that there are no more data to be replaced.

We still may have different paths that contain similar fragments. Therefore, we still may have shuffled lines on the map due to fragments that should be replaced by *averaged fragments*. *Fragment similarity* will be defined soon and we are going to discuss about the process in charge of finding out *similar fragments* from different paths to be replaced.

The *fragment averaging process* was very studied like the prior processes, but many of its problems were not solved in this project. Its development was chosen as a compromise between ease of implementation and program efficiency. During the implementation of this process, we have noticed the importance of the next *stitching process* that was discussed but not implemented in this project. The *stitching process* is important due to the need of its participation in the *fragment averaging process* for obtaining a nice resulting map. The *stitching process* and its importance are going to be discussed also in this chapter.

3.4.1 Fragment similarity

The same concepts used for *path similarity* are used for *fragment similarity* because paths and path fragments are abstractly the same. Path fragments also are defined by ordered list of vertices, thus they can be treated as independent paths. The same *similarity* definitions used for paths can be used for path fragments. Therefore, *similar fragments* are *similar paths* that belong to other paths.

The only difference between *similar fragments* and *similar paths* is that a fragment is 'part of' a path, and two fragment vertices are special: the *fragment origin* and *destination vertices*, v_o^f and v_d^f. These vertices are said to be special because perhaps they are points but not vertices from the path containing the fragment they belong to (i.e. they are not necessarily endpoints of path segments but they are considered as vertices with respect to their fragments: figure 74).

The special *origin* and *destination vertices* don't change the concepts defined before. We just need to treat the fragments as paths, and the definition for *fragment similarity* would be exactly the same.

A special kind of *fragment similarity* is the already explained *Self-similarity*. It is the situation where two fragments of the same path have *distance-similarity* (not necessarily oriented).

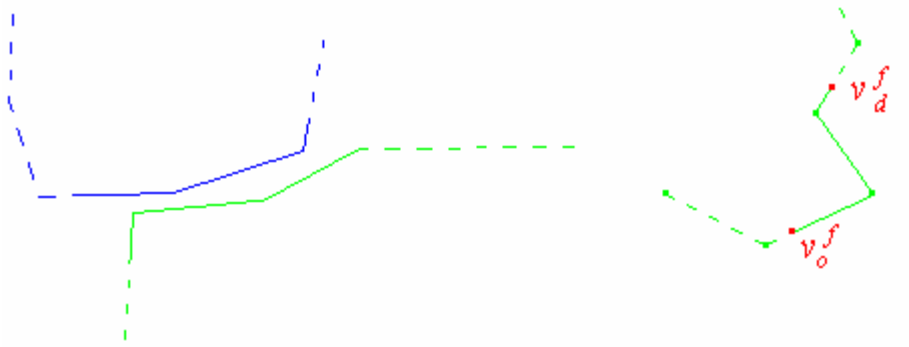

Figure 74. Left: The green and blue paths are not similar but they have (non-dotted) *similar fragments*. Right: A green path containing a (non-dotted) fragment whose (red) *origin* and *destination vertices* are not vertices of the path it belongs to.

3.4.2 Fragment Averaging process

At the beginning, it was thought that the *fragment averaging proces*s would be quite simple and fast to be implemented. We should have the *similarity detection process* and the *path averaging process* already designed, and actually we do have. The initial thought was: "Any path fragment can be treated as a path, so we can apply the same prior processes for detecting *fragment similarities* and for averaging *similar fragments*".

On one hand, it is true. We can define if two given fragments are similar by using the path *similarity detection process*. We also don't have any problem for computing the average of two given *similar fragments*.

On the other hand, it is false. An ordinary path can be split in infinite possible fragments combinations. It is not computable to check every possible pair of fragments that two paths may form for checking if these paths have some *similar fragment*! It is much more reasonable to check all possible pair of segments from two different paths.

However, it is much more troublesome to compute from where to where path fragment are similar than simply finding a couple of similar segments. Moreover, the problem is not just the complexity for defining the entire *similar fragments*. Even though we can average them as normal paths, what should we do with some resulting *average fragment*? And what should we do with paths whose fragments were averaged?

We cannot just discard paths after a fragment averaging like we did after a path averaging, because we don't want to lose data and a resulting *average fragment* does not substitute any entire path but just part of it.

First, we are going to discuss about a procedure for detecting *similar fragments* between two paths and defining from where the fragment similarity begins to where the fragment similarity ends. Later, we are going to discuss how to manage resulting *average fragments*.

3.4.2.1 Detecting fragment similarity

Now we have to find similarity between any part of two given paths P_x and P_y. There is only one computable way for checking if there is similarity between some pair of path fragments. Somehow, we have to compare every segment from P_x with every segment from P_y in a search for points that are *close enough*.

This step is a little similar to computing *intersection points*, but instead of searching for 'equal' segment points we look for 'similar' segment points. Once more, we have many possibilities for performing this search, e.g. constructing a grid map for comparing just near segments, creating an optimal search algorithm for comparing all segments, or using an algorithm that is not the most efficient but a simple one. The last example was chosen again because it is not tried to find an optimal solution but a first solution to the problem.

Even trying the simplest similarity search by comparing segments, troubles were found anyway. This procedure seems to be much easier than it really is. For instance, the fact that we don't have to care about extremity tolerance (like for path similarity detection) makes us to believe that it is going to be very simple. We are going to explain how the problems arose and how they were solved, or partially solved.

Initially, we want to find the first segment of the path P_x that has some point *close enough* to the path P_y. This is simple by using 'for' loops and checking each segment of P_x with every segment from P_y. Let us call these 'segments containing points that are *close enough*' as ***approximated segments***. The first time that we find (whether they exist) *approximated segments* by using 'for' loops, we can be sure that we have found the first segment from P_x (with respect to its orientation) that belongs to a *fragment similarity* between P_x and P_y. But did we find the first segment from P_y that belongs to this fragment similarity? The answer is yes if and only if path fragments had the same orientation. But if the fragment orientations are inverse, probably the answer is 'no' (figure 75).

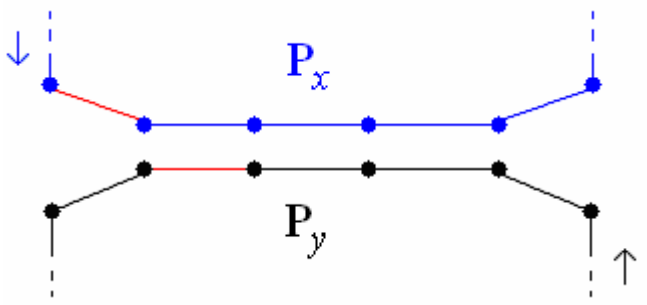

Figure 75. Fragments with inverse orientations do not help the *similarity detection*. Here we have two similar path fragments with inverse orientation. In this ordered search, every segment from the black path is compared with each segment from the blue path. Therefore, the first found *approximated segments* (red segments) are not going to be both the first segment of these similar fragments. In this example, the *approximated segment* of the black path is almost the last segment from this *similar fragment*.

If there is some *fragment similarity* between P_x and P_y, we want to obtain the two 'complete' fragments of this similarity, for instance the entire non-dotted fragments from the figure 75. We also wish their segments to be correctly ordered in cloned fragments, for working with them later. Hence, we need to answer the following questions:

- Is there some fragment similarity (or *close enough* segments) between P_x and P_y?
- If yes…
    - Which is the first *approximated segment* of each *similar fragment*?
    - Which is the last *approximated segment* of each *similar fragment*?

If we have the answer to these questions, we can know if there is *fragment similarity* and we can build a clone fragment for each *similar fragment* (whether they exist). These answers allow us to know the path of each (similar) fragment, the path segment where they begin and the path segment where they end. We can even apply methods for checking if these *similar fragments* have the same orientation or not.

When the first *approximation segments* are found, we can stop the 'for' loop and begin another type of *similarity search* using a 'while' loop. Now we only need to analyze segments prior to a first and next to a last *approximated segment*. If the segment prior to a first *approximated segment* is also approximated then it becomes the first one and this check is done again[15]. The same happens for last *approximated segments* and their next segments. Changing this search from a 'for' to a 'while' loop also tends to finish the process faster because is not necessary to check every pair of segments anymore.

When there are no more prior or next *approximated segments* to be found, we can build clone fragments with these segments following their original order. We also can check if the *similar fragments* are *inverse* ones, and we can equalize their *orientations* by changing the segment order of one of them whether they are *inverse fragments*. Here we want the average of these *similar fragments* to have a same orientation: the *reference orientation*. We are going to assume the first parameter path P_x as the reference path, and the *reference orientation* belongs to this path. This is just for having control of the resulting *average fragment* orientation.

After building the clone fragments of a *fragment similarity*, we could still keep searching for further *fragment similarities* between the paths. But instead it, this process just ends the *fragment similarity detection*, assuming that there are no more *fragment similarities*. This assumption may be false but if the resulting paths are checked again, new *fragment similarities* are able to be detected. Later, it is going to be explained how to manage resulting paths.

Even though we don't need to tolerate *path extremities* in this procedure, we must remember that we are working with segments. Perhaps the origin and destination points of fragments are not path vertices (or segment endpoints). Therefore we have to compute the first and last points where the *fragment similarity* starts and begins. The same procedure used for computing these points in *path similarity detections* is going to be used here, but here we have the advantage that we already know that these points must belong to the very first and very last segment of their correspondent fragments.

[15] There is no need to check segments before the first *approximated segment* of the (main reference) path P_x because we already know that it is the first segment of its fragment.

When having the cloned *similar fragments*, we can treat them as independent paths and we can use the *path averaging process* for computing an *average fragment* that would replace these two *similar fragments* in the map.

3.4.2.2 Averaging similar fragments

To average two defined *similar fragments* is as easy as to call the *path averaging process* passing the *similar fragments* to it as parameter paths. But the goal of this process is not only to compute the average of *similar fragments*, but also to figure out if they do deserve an average. We may have troublesome situations if we average any pair of *similar fragments*. We may even distortion nice paths due to it. For instance, imagine two paths containing a 'natural link' between them (figure 76), e.g. a path ends where another path begins. Of course that we are going to detect this kind of *fragment similarities*, but it doesn't deserve an average.

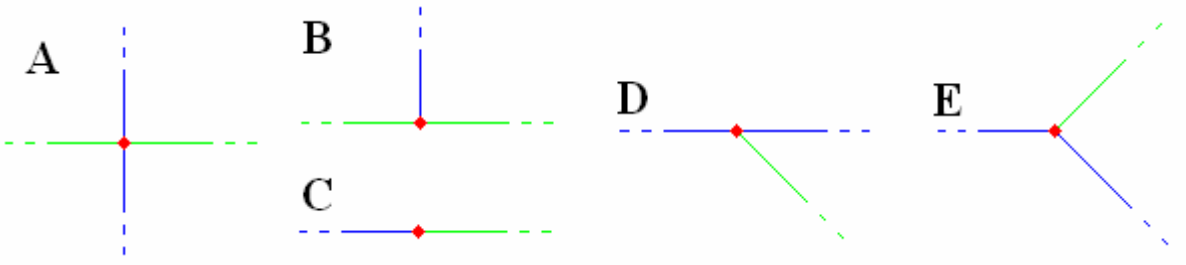

Figure 76. Natural links between two paths. Example A: a crossing. Example B: a branching. Example C: a road concatenation. Example D: a road fork. Example E: a road split.

In the next figure, we can observe that some averages are not necessary, and they also are unwished. This distortional type of fragment *averaging* generates quite short *average fragments*, and we can use this property to avoid them. A program constant ***MAFL*** (Minimal Average Fragment Length) is going to be used for determining the minimal length of an *average fragment* to be considered useful. Another program constant ***MPL*** (Minimal Path Length) is going to be used for limiting the shortest length that a path must have to be considered in the project processes.

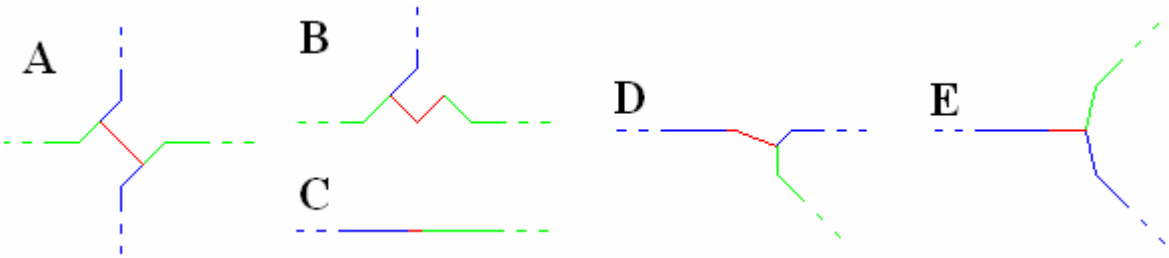

Figure 77. The same pairs of paths from the prior figure have been ordinarily averaged without deserving it. The example C suffered no consequences of it, but all the other examples suffered unnecessary distortions. The red fragments are *average fragments* that are common for the blue and green path.

Too short paths don't use to be important data and they can be conflictive in this process depending on the process management[16]. For instance, we are going to notice the conflict they can cause when we try to define a *crossing*. However, the constant *MPL* can be set up to zero making this process to accept any path.

3.4.2.3 Path concatenation, branching and crossing

There are three kinds of path links and their definition is a project requirement. It was not possible to define them before and it is still difficult to define path links due to special casualties.

Empirically talking, a *concatenation* is the link between two different paths where some endpoint of a path is equal to some endpoint of the other path. A branching is also a link between two different paths where some endpoint of a path also belongs to another path but is not its endpoint. A crossing is the last kind of link between two different paths where one segment of a path intersects one segment of the other path in one point.

These empirical definitions suppose we are dealing with non-shuffled paths without error margins, without self-intersections, with no large similarities, etc. Of course that these definitions are not always right. First, we cannot be very accurate in the definitions of path links if we work with paths containing error margin. Second, we have to use *path fragments* in these definitions instead of using entire paths, because even two *non-similar paths* can have similarity in some fragments.

Definitions for linking paths:

Let us consider two paths (that even might be the same one) having some *fragment similarity*. Then, let us consider a pair of *similar fragments*, where one fragment belongs to each path. These *similar fragments* must be 'complete'. This means that their prior and next (sub) segments cannot be part of this *fragment similarity*.

The considered '*complete similar fragments*' must be determinately short for the next definitions. The length for determining if these *similar fragments* are short enough can be modified. It could be determined by the constant *MAFL* or another constant created especially for that. We are going to use the term '*short enough*' for referring to this length during the definitions.

Considering the *complete similar fragments* again, they cause a ***path concatenation*** if:

- Both they are *short enough*,
- Each fragment contains a *path endpoint*, and
- These *path endpoints* are *close enough*.

[16] Too short paths might cause infinite loops in the implementation of this process. The constant *MPL* should not have a very small value for this implementation.

Considering the same *similar fragments*, they cause a ***branching*** if:

- Two *fragment endpoints* are *close enough* and they are not *path endpoints*.
- The fragments are not *short enough* or they have no intersections or they have more than one intersection[17].

One or two *branchings* may happen for a pair of *similar fragments*. They just happen near to extremities of *similar fragments*.

Considering the same *similar fragments* once more, they cause a ***crossing*** if:

- Both are *short enough*,
- They have no *path endpoint*, and
- They intersect just once.

If a crossing happens, it happens in the intersection point.

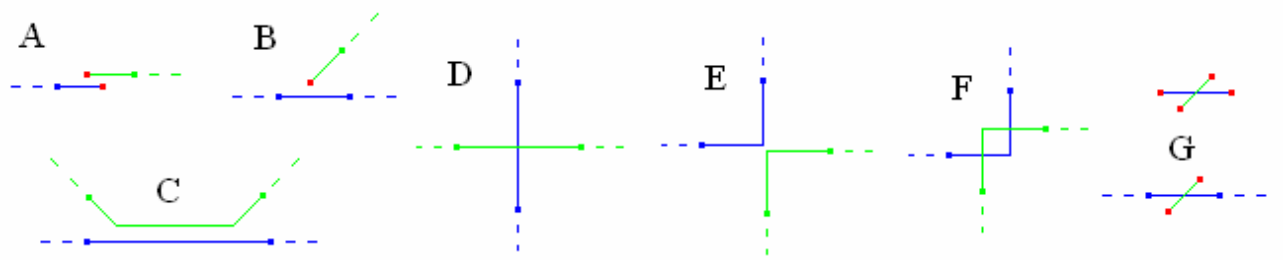

Figure 78. Example A: It is a *path concatenation* because one path begins approximately where another end (red points). Example B: It is a *branching* because one path begins or ends at the middle of another one. Example C: It is a pair of *branchings*, because no path begins nor ends but they got divided twice. Example D: It is a *crossing*, because no path begins nor ends and they have a unique *intersection point*. Example E: Popularly it would be considered a crossing but here it is a pair of *branchings* (like in the example C) because it has no intersection. Example F: Again, it is a pair of *branchings* because the intersection happens twice and the formed rectangle should be averaged. Example G: Definition conflicts due to very short paths.

Analyzing the definitions for path links, we can understand some conflicts that very short paths can create. The example G of the figure 78 illustrates two situations that we would consider as crossings, but perhaps these fragments are ordinary *similar fragments* that should be averaged. If we decide not to average crossings for avoiding unnecessary distortions, we might stop averaging several short paths that actually are ordinary *fragment similarities*. It is suggested to ignore very short paths by discarding them from the map.

After discarding very short paths, we still need to decide when *similar fragments* would result in distortions or in good averages. *Similar fragments* are passed as independent paths to the *path averaging process*, and then they are split in polygons like normal paths were.

We remember that normally the first and last simple polygons resulting from *similar paths* had to be closed by a segment. Actually, we just need to check simple polygons that initially are not closed, because every closed polygon deserves an *average fragment*. Every 'closed' polygon has intersection points as origin and destination vertices.

[17] Observation: A branching cannot be a crossing according to the definitions of this project.

Opened figures are not really polygons, but we are going to refer as 'opened polygons' to these figures that can form a simple polygon by linking their endpoints. Opened polygons (containing only one or no intersection points) might be a distortion if they are averaged. Then, we have to decide which polygon formed by similar fragments should be replaced.

Closed polygons must be averaged for sure, but opened polygons must be studied. It is a good suggestion to not average open polygon that is part of a crossing, because crossings always cause distortions when they are averaged. We shouldn't average opened polygons causing *path concatenations* as well, because this *fragment averaging process* don't manage concatenations. The *stitching process* should care about it.

If the first or last simple polygon formed by two *similar fragments* is an opened polygon and it causes a path *branching*, then only the closest segment to each fragment endpoint must not be averaged.

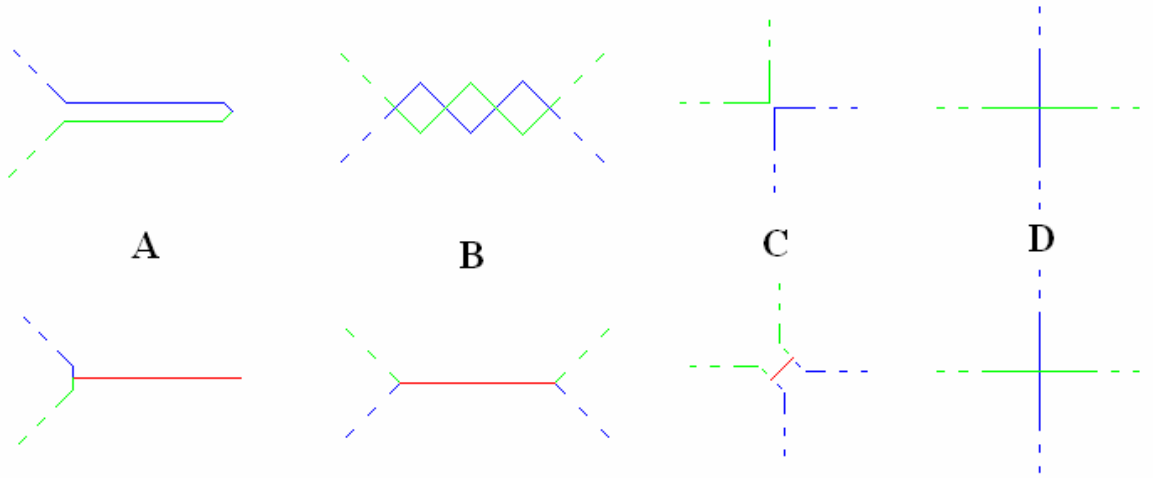

Figure 79. Example A: The *similar fragments* cause a branching, thus only the segments that contain fragment endpoints aren't averaged. Example B: All closed polygon are averaged. Example C: It is not a crossing but a pair of branchings, and then only some segments are not averaged like in the example A. Example D: It is a crossing and crossings are not averaged.

Actually, this step of the *fragment averaging process* is an optimization for *the path averaging process* because it is the best process where to implement it. But this step was not implemented in the *path averaging process* because it is not needed for working with entire *similar paths*. The problems for linking paths and their fragments were noticed during the implementation of a simple optional method for averaging fragments. Therefore, we are not working with the concepts of *concatenations*, *branchings* and *crossings* in the implemented program. The implementation of the *path averaging process* only suffered one little modification due to these definitions: It ignores very short *average paths* for avoiding disturbed *crossings*.

3.4.2.4 Managing averaged fragments

Sometimes the average of two *similar fragments* is null because it is an ignored result due to its too short length. However, in case that the resulting average fragment has a considerable size, what should we do with it? And, what should we do with the paths that contain the averaged original fragments? Definitely we have to discard those averaged original fragments from their paths because they are not useful anymore. They must be replaced.

Each original path can be split up to two shorter paths if we eliminate one of its middle fragments, thus the two original paths may result in up to five paths: two split fragments for each path and an *averaged fragment* (middle example in the figure 80).

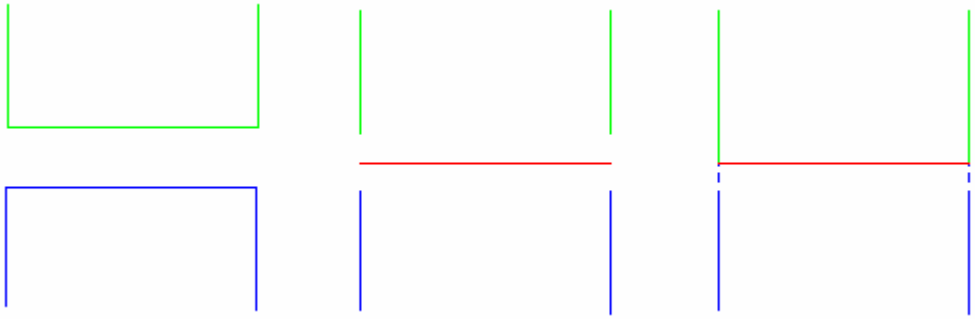

Figure 80. Left: Two paths (green and blue) have *similar fragments*. Middle: An (red) *average fragment* was computed for replacing the *similar fragments*. Right: The red *average fragment* replaces its original fragment from the green path, and the resulting blue split paths were linked to it.

Therefore, it was decided to replace the original fragment of a reference path P_x by the *average fragment*. Hence, we would have no more than three paths resulting from the two original ones.

Thus, now we have a path P_x' with a replaced fragment, and two paths P_y' and P_y'' split from a discarded original path. These paths P_y' and P_y'' have to be linked to the P_x', as we can observe again in the (right example in the figure 80).

3.4.3 Stitching process

The *stitching process* was analyzed but not implemented in this project. However, this process is very important for having a nice map. Actually, it avoids not only the inconsistencies caused by those path endpoints that are very near to each other but not linked.

Definitely, avoiding this inconsistency is not the only importance of this process. It was noticed when during the analysis of the *fragment averaging process*. There, we understood the importance of path links, and we also noticed that they must keep linked even after future procedures.

Paths resulting from a same fragment averaging can be correctly linked by adding the first and the last vertex of the *average fragment* (replaced in P_x') to the split paths P_y' or

P_y'' correspondently. But even though these points are added correctly creating a nice link, these links can suffer a distortion later. Let us suppose that the same fragment replaced in the path P_x' suffers a position change due to another fragment averaging (figure 81). The linked paths P_y' and P_y'' are not going to be displaced together with it! The same problem may happen to *path concatenations*. We need a stronger 'linking structure' if we don't want links to suffer distortions like in the last example.

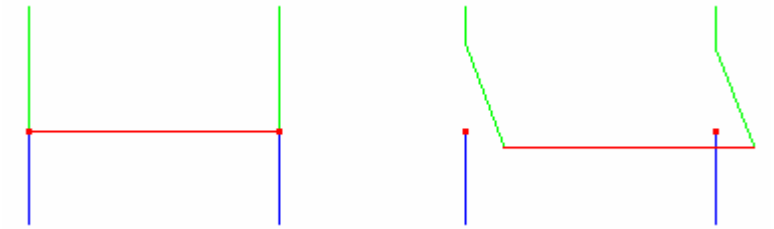

Figure 81. The green path containing a red replaced fragment was moved due to other averaging process with another (not illustrated) path fragment, but the linking vertices from the blue paths did not move together.

Then, the concept of the *linking nodes* appeared. These nodes, already defined in the domain analysis, are not real but abstract points in the map. A *linking node* must refer to two path vertices that it links, but not to the positions of these vertices because they might change. At the end of every process for building the map, we should add in the map a segment for each linking node.

This idea implies an array for storing *linking nodes*, and it also implies vertex identifications. We must be able to localize any referred vertex easily; therefore path identifications are also needed. It was thought to identify every vertex by its path ID and the vertex position in the correspondent *path vertex list*. But the problem is that "if we add or remove vertices from the path the vertex, IDs are going to change". The same problem would happen if we identify paths by their indices in a list for storing paths, and IDs must not change.

The *stitching process* should be used not only for linking paths but also for repairing some problematic complexities that are typical in path links, e.g. *self-intersections* and *immediate returns* caused by linking paths or fragments (figure 82).

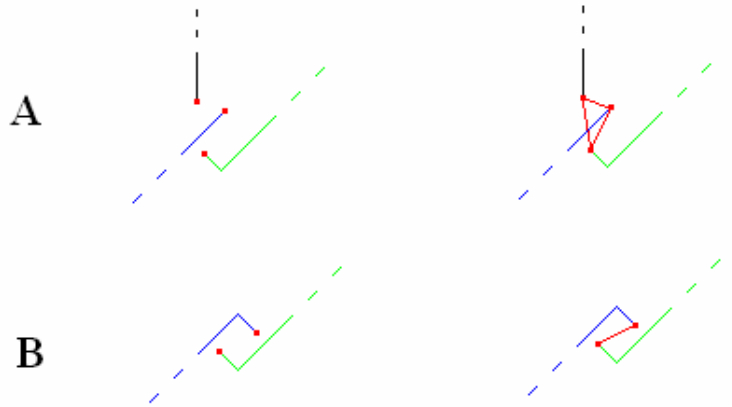

Figure 82. Example A: If more than two (red) path endpoints are very close, their individual links probably would not result in a concise link. Example B: Path links might result in not nice unions.

4 IMPLEMENTATION

Although the implemented codes were written for not using unnecessary processing time and avoiding other considerable problems, optimization is not the aim of them. Therefore, many of them probably can be improved.

Each coded class is going to be explained below, enhancing the priority classes and their most important methods. Ordinary methods will not be explained here but they can easily be understood by just checking their codes in the appendices, where all the codes are included.

4.1 The *Const* class

This is a class built only to gather important constant values that will decide the behavior of many program methods and the final map result to be displayed. Some of these constants are the distance for considering path similarities (PSD), the error margin distance that define vertex fluctuations (EMD), the tolerable lengths at extremes of similar paths (TLE), etc. All these important values are shown in the appendix A.

4.2 The *MListener*, *KListener*, *MapFrame*, *MapPanel* classes

These are classes and interfaces in charge of the Graphical User Interface issues. All they are going to have a short and simple explanation here.

MListener (whose code is added to the appendix I) is an interface for the mouse usage, but we are not going to use the mouse for displacing the map into the frame. We are going to use the keyboard for that. Therefore, the only objective of this interface is to allow the mouse to activate the focus for the keyboard on the frame used for displaying the map.

KListener (whose code is added to the appendix J) is the responsible interface for listening at the keyboard and displacing the map into the frame according to the pressed keys. These keys are going to be the number keys (containing arrows in the keyboard set for numeric keys) and the keys '+' and '-' (in the same set of keys) for zooming in and out the map correspondently.

The MapFrame class (whose code is added in the appendix H) just sets the frame for the map and creates its panel. The MapPanel class (whose code is also added in the appendix H) is the responsible for drawing the map according to its member called *pathList*. This member contains every path to be in the map. Changing the map is as easier as to change the paths contained in this array and repainting the panel. Other members of this class only define the position and size for the map to be displayed into the frame.

4.3 The *Point* and *Segment* classes

These classes includes methods for setting points and segments, mesuring them and computing relationships between them. They are the base for future implementations in this project. Other implementations depend of instances of these classes and of many methods provided by them. Even though we should work with positions on the Earth globe, the *Point* and *Segment* classes were limitedly implemented to work with planar coordinates, and the reason of that is going to be explained next.

4.3.1 Position Coordinates

The surface of the globe is divided into a spherical grid for the convenience of finding certain points. The grid consists of imaginary lines called latitude and longitude. Latitude is a series of circles running parallel to the equator and extending to both poles. Longitude is a series of lines drawn between the poles at regular intervals that pass perpendicularly through the equator. Everywhere on the globe, a particular latitude crosses a particular longitude and gives to us a pair of spherical coordinates indicating this position.

However, in this project we will work with planar coordinates instead of geographical or spherical coordinates [GoS] because of its geometric simplicity. The main aim of this project is not to work with a particular kind of coordinates but to find solutions for a pathfinder system, independently of the used coordinates.

The simplest type of coordinates is going to be used here. However, other coordinate types can be used without changing other implementations of this project. These *Point* and *Segment* classes are the responsible for determining which type of coordinate is going to be used. They provide tools to the project for computing distances, lengths, intersections, angles, orientations, etc, and the performance of these tools depend of the used coordinate system. Other classes of the *Pathfinder* project do not work directly with coordinates, but they work with *Point* and *Segment* instances and with the methods offered by these two classes.

Therefore, only methods of these classes have to be adapted to a new coordinate system if planar coordinates are going to be replaced. A program constant might be created to determine the type of coordinate to be used. *Point* and *Segment* classes might be coded for working with alternative different coordinate types, according to the last mentioned constant value.

We could insist on using spherical coordinates in this project but it would bring some problems. Even though an optimal processing time for the implemented methods is not an aim of this initial project, spherical coordinates would just slow down the processing speed of the program due to necessary complex formulas [ST] full of trigonometric functions for computing often used data like distances, lengths and intersections.

These formulas were replaced by simple planar formulas with a scope for relatively small areas on the globe (e.g. 10.000 Km^2). Anyway, this scope is quite considerable for an initial project and it allows us to convert spherical coordinates to planar coordinates. Then, we can use simple planar formulas in this project for computing those frequent needed data.

4.3.2 Dealing with geographical coordinates from income data

The income data we have for this program are files loaded with geographic coordinates, providing latitude, longitude and altitude for each point. The altitude is the minimal distance from an exact position to a reference geoid. Altitudes are just going to be ignored because we are going to work in a planar system. But latitudes and longitudes have to be transformed from the spherical coordinate system to *x* and *y* coordinates [Map].

Latitude is measured from -90 degrees (at the South Pole) to +90 degrees (at the North Pole), being zero at the Equator line. Longitude is measured from -180 degrees (at the West) to +180 degrees (at the East), being zero at the Greenwich Meridian. The reference lines for counting are the Equator, for latitude, and a line drawn through Greenwich in England, the prime meridian (or Greenwich meridian), for longitude. These are the zero lines that we are going to use as *x*-axis and *y*-axis correspondently in this project. The income data uses radians instead of degrees for measuring these angles: *π radians are equivalent to 180 degrees*.

For using planar coordinates, we have to work with distances but not angles of distance. We have to convert angles of latitude and longitude to distances on the globe. The Earth polar circumference (2π * Earth polar radius) is approximately 39940 Km. If we divide it in 360 degrees, we get 110,9 Km. This means that a degree of latitude is equivalent to about 111 kilometers on the *y*-axis. If we want to have the reference between 'latitude radians' and meters on the globe, we should divide the circumference by 2π radians instead of 360 degrees: ***1 radian of latitude is equivalent to about 6356,7 kilometers on the y-axis***.

For longitudinal lines, the conversions are a little different because they converge towards the poles, meaning that degrees of longitude vary according to the position on the Earth. At the Equator, one degree of longitude is practically the same length as one degree of latitude. At the north and south poles the distance between degrees of longitude is zero.

We have to calculate the coordinate value on the Equator for the given longitude and then we have to multiply it by the cosine of the 'reference latitude' for converting our calculated equatorial distance to the distance on the desired latitude. After it, we have converted he longitude angles in meters of distance on our *x*-axis. The following latitudes have being analyzed for being the 'reference latitude' on this project program:

- Denmark average latitude: 56°25’ or 0,98465656 radians
- Great Copenhagen average latitude: 55°40’ or 0,97156661 radians
- Lyngby approximated latitude: 55°46’ or 0,97331194 radians

It was decided to get the approximated latitude of Lyngby because most of the paths used for tests are near to it. But it is going to be just a constant called ‘ReferenceLatitude’ in the program, and it can be easily changed without affecting the remaining implementation. However, less than half degree of *reference latitude* is not going to change the distances significantly.

The Earth equatorial circumference (2π * Earth equatorial radius) is approximately 40075 Km (The Earth polar circumference and the Earth equatorial circumference differ a little bit [ER]). If we divide it in 360 degrees we get 111,3 km. So, a degree of longitude is also equivalent to about 111 kilometers on the Equator. If we want to have the reference between ‘longitude radians’ and meters on the Equator, we should divide the circumference by 2π radians instead of 360 degrees: ***1 radian of longitude on the Equator is equivalent to about 6378,1 kilometers on the x-axis***.

Now we have to convert this distance on the Equator to the distance on our chosen *reference latitude*. A degree of longitude is going to be equivalent to 111 * cos(55°46’) = 62,44 km on this latitude, so a longitude degree will represent 62,44 km on our *x*-axis. Similarly, 6878,1 * cos(55°46’) = 3588,09 km, so ***1radian of longitude on our chosen reference latitude will represent 3588,09 km on the x-axis***.

These calculated numbers are some of the program constants that are added in the appendix A (in meters instead of kilometers). The code of a simple method from the *Point* class for converting latitudes and longitudes to planar coordinates is called *sphericalToPlanar*. It can be found in the appendix B.

4.3.3 Distances

Three kinds of distances are needed in this project and their methods are implemented in the *Point* or *Segment* classes.

4.3.3.1 Point-to-point distance

We can use the very well known mathematical formula for calculating the Euclidean distance between two given planar points $p_1 = (x_1 , y_1)$ and $p_2 = (x_2 , y_2)$:

$$d(p_1 , p_2) = \sqrt{[x_1 - x_2]^2 + [y_1 - y_2]^2}$$

We have used this formula instead of using complex formulas [STC] for calculating the shortest distance between two spherical points.

4.3.3.2 Segment-to-point distance

Another useful distance is the minimal distance between a point and a segment. Planar coordinates make it much easier to be calculated because we just have to follow some simple planar geometric concepts. The formula for straight lines will be used for computing this distance:

$y = a \cdot x + b$, where a is the line slope and b is its vertical displacement.

Given two points from a line, $p_1 = (x_1 , y_1)$ and $p_2 = (x_2 , y_2)$, it is always possible to compute its slope and vertical displacement by using the following formulas:

$$a = \frac{y_1 - y_2}{x_1 - x_2} \qquad b = a \cdot x_1 - y_1 \text{ or } b = a \cdot x_2 - y_2$$

One possible situation should be taken in account. Vertical lines have infinite slopes and incomputable vertical displacements. There are simple solutions for it because, even if we cannot work with infinite values, vertical lines have another formula for this situation and it is even easier to work with this formula:

$x = c$, where c is the horizontal displacement constant [18].

Therefore, given both *endpoints* of a segment, we can define its segment line ℓ finding the equation constant values a and b, or c.

Given another point $p_3 = (x_3 , y_3)$, we can find the transversal line ℓ_t (of the line ℓ) that intersects the point p_3 (illustrated on the figure 83). We know that the slope of the transversal line is $a_t = -1 / a$ and we can calculate its vertical displacement b_t with the coordinates of the new point p_3: $y_3 = a_t \cdot x_3 + b_t \Leftrightarrow b_t = y_3 - a_t \cdot x_3 \Leftrightarrow b_t = y_3 + x_3 / a$.

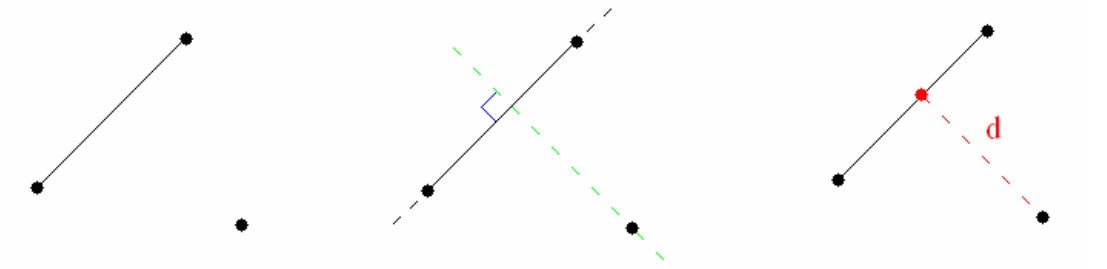

Figure 83. Left: a given segment and a given point for computing the distance between. Middle: It is defined a transversal line for the given segment line. The defined line intersects the given point. Right: The (red) point where the transversal lines intersect is the closest point of the segment line to the given point.

[18] Observation: All the methods that work with line equations check if they would be vertical. In such situation, they work with the line equation "$x = c$". If and only if the slope of a line is infinite, the slope of its transversal line is zero and vice-versa.

Once we have the transversal lines ℓ and ℓ_t, we can compute their unique intersection point $p_i = (x_i , y_i)$ by using formulas that result from the following equation calculations:

$$y_i = a \cdot x_i + b$$
$$- \quad y_i = a_t \cdot x_i + b_t$$

$$0 = (a - a_t)x_i + (b - b_t)$$

$$x_i = \frac{b - b_t}{a - a_t} = \frac{b - b_t}{a + 1/a}$$

$$y_i = a \cdot x_i + b \quad \text{or} \quad y_i = a_t \cdot x_i + b_t$$

Having the intersection point p_i, we can calculate the distance from this point to the given point p_3, what represents the closest distance from the given point p_3 to the segment line ℓ. It is not sure that this is the closest distance between the point and the segment yet! The figure 84 illustrates that if the intersection point p_i doesn't belong to the given segment, the closest segment point to p_3 is not p_i but one of the segment *endpoints*.

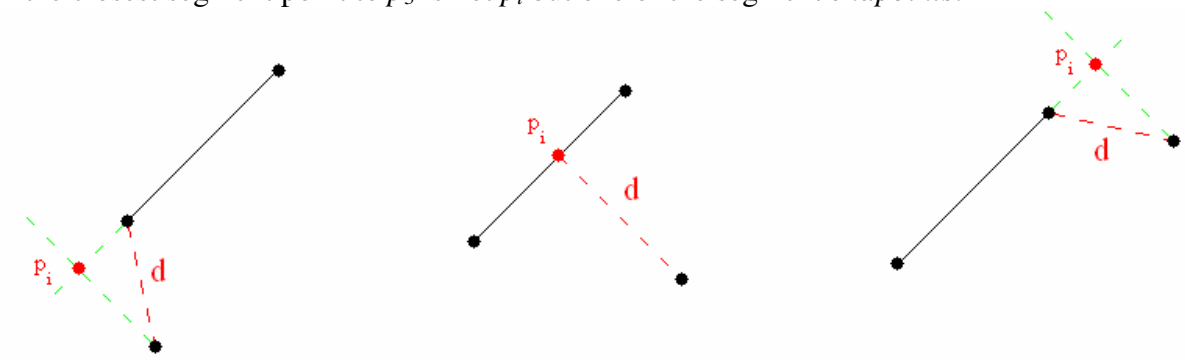

Figure 84: Not always the computed intersection point p_i belongs to the given segment, so the closest segment point to the given point is not p_i but a segment endpoint.

For avoiding this problem, we just check if the computed intersection point p_i belongs to the given segment. If it does so, we calculate and return the distance between p_i and the given point. Otherwise we compute the distances from the given point to both segment *endpoints* and return the shortest of them.

4.3.3.3 Segment-to-segment distance

Another useful distance that has its implemented method is the shortest distance between two segments. This distance is not difficult to be computed if we use other methods from the *Segment* class. We just have to realize that the minimal distance between two non-intersecting segments is the distance from an segment *endpoint* to the other segment (figure 85). We just have four *endpoints* for taking in account, so we only need to compare four distances and return the lowest of them. Intersecting segments are exceptions for these calculations, but it is obvious that the minimal distance of any two intersecting segments is zero.

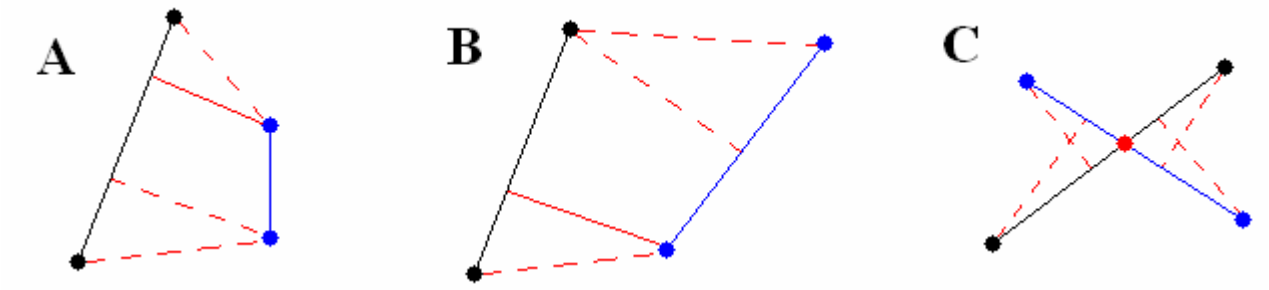

Figure 85. A pair of segments has four distances from some segment endpoint to the other segment. The lowest of them four is the minimal distance between the two segments (examples A and B), except if the segments intersect (example C). In intersecting situations, the minimal distance is zero.

The first step in this distance calculation is to check if the given segments intersect (method explained next). If they don't, each distance from an endpoint to the other segment is going to be calculated by using the method for computing minimal distances between segments and points. The shortest of them is going to be returned.

The method called *distance* for computing point-to-point distances belongs to the *Point* class and its code can be found in the appendix B. The methods also called *distance* for computing the segment-to-point and segment-to-segment distances belong to the *Segment* class and their codes can be found in the appendix C.

4.3.4 Intersections

The intersection point of two lines is easily computable in few steps, but a procedure for computing the intersection point of two segments needs more steps to be performed. However, it is not difficult just to check if an intersection happens between two segments.

4.3.4.1 Detection of intersections

Initially, we want a fast method that does not compute any intersection position but just checks if two segments do intersect. This method is frequently used, thus it is important to be as short and simple as it can [19].

[19] An algorithm for this method is explained in the chapter 35.1 of the book '*Introduction to Algorithms*' of Thomas Cormen et.at. [IA]

An initial and quick test called '*the boundary box*' is used for checking if it is possible for two segments to be intersecting or not. One segment is completely enclosed in a minimal 'box' (or rectangle). If the other segment is completely out of such 'box' the intersection is not possible.

Whether the two given segments do pass the boundary box test, we are going to use cross products in the next step. Due to cross products, we don't need to care about vertical lines and we can check in few steps if the intersection happens. One segment is going to be taken as the base for this test and its line (containing such segment) is going to be the 'judge' for it. Each endpoint of the other segment is going to be analyzed respect to the base segment line.

The cross product tells us in which side (left or right) of the base line each other endpoint is. If both endpoints of the other segment are at the same side of this line, the segments are not intersecting, but if they are at different sides or some of them are on this base line, definitely they are intersecting.

4.3.4.2 Positioning of segment intersection points

Often, we want to have the exact position where a segment intersection happened after we know it occurred. It is quite simple to calculate this position using the already mentioned line equations, but we have to deal with some possible exceptions. Now we are not just talking about vertical lines, whose problems were already solved. The new condition we have to consider is that segment intersections are not always just a point. The intersection of two segments could be another segment whether they are collinear.

Therefore, if we detect that there is intersection between two segments and we know that they are collinear (or have the same slope), a special intersection happened. This situation is not very frequent at all because a considerable coincidence is needed. However, a method for this possible coincidence was implemented.

A private method is going to be called when this not very common exception happens, and this method returns a resulting intersection segment (instead of an intersection point). If this exception does not happen, we just need to use simple mentioned calculations for computing the intersection point of two segments. A method is in charge of computing intersections and it will call the private method if necessary.

However, this method is in charge of computing intersections has also to return segment as result because of the possible exception. Actually, it is not a problem because we can return a 'unitary segment'. We call 'unitary segment' to a segment containing two equal endpoints, thus it really represents a unique point. Other methods have to check if this returned segment is unitary or not.

Whether the intersecting segments are collinear, two situations are possible. First, one of the segments could completely overlap the other one. Second, just part of them could overlap. Both situations are illustrated in the figure 86. Those situations where just two endpoints are equivalent or the complete segments are equivalent are going to be considered special but they also match the prior situations.

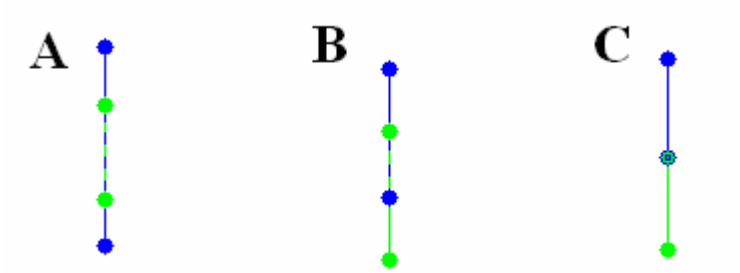

Figure 86. Two given collinear segments. In the example A a segment is completely overlapped by the other one. In the example B just part of them overlap. In the example C only one endpoint of each intersecting segment is equivalent to another: a special situation that matches the example B.

The private method for collinear exception is going to check all possibilities for collinear segments, taking in account that:

- Situations like the example A of the figure 86 occur independently of which given segment is the overlapped and which is the overlapping one.
- Situations like the example B of the figure 86 occur independently of which given segment is the upper and which is the lower one.
- If it happens that the segments are completely horizontal in situations like the example B, we are going to consider leftmost and rightmost segments instead of upper and lower segments.

A method called *checkIntersection* just inform if an intersection happened between two given segments. A method called *intersection* computes the position where two given segment intersect. If the given segments are collinear, the private method *collinearIntersection* is called. These methods belong to the *Segment* class and their codes can be found in the appendix C.

4.3.4.3 Angles, turnings and orientations

There are three properties of consecutive segments that give us notion of road directions. They are angles, turnings and orientations. They are going to be discussed below.

Angles:

The main reason for having a method in this project that computes angles is the need of detecting *immediate returns*. We must remember that not only consecutive segments may form *immediate returns*. Hence, not only the angles formed by consecutive segments need to be checked.

For computing the angle of non-consecutive segments, we are going to consider these segments as *vectors* containing start and end points. We are going to match vector start points for computing the angle they form. The computed angle can vary if we match a vector start point with a vector end point instead of matching both of their start points (Example A of the figure 87). Therefore, the orientation that we give to each segment, when converting them to vectors, is very important [20].

A pair of matched vectors forms two angles, and only one of these angles has to be returned. Only the smaller of them cares for *immediate returns*. Therefore, we decided this method to return the smaller angle, even though this method was coded not only for detecting *immediate returns*. It was also coded for other occasional needs.

So, we have to match vectors but there are different ways for doing it. How are we going to match the start points from separated vectors? Should we translate the position of one of them to the other's position?

No. This method does not translate vector positions by calculating new coordinates for their endpoints. Actually, this method doesn't match the vector start points. It works with them in their original positions because of efficiency reasons. Translations are not needed and the match happening here is completely abstract. We can compute the independent angle of each vector respect to the coordinate axes and then combine these computed angles to obtain the final one (Example B of the figure 87).

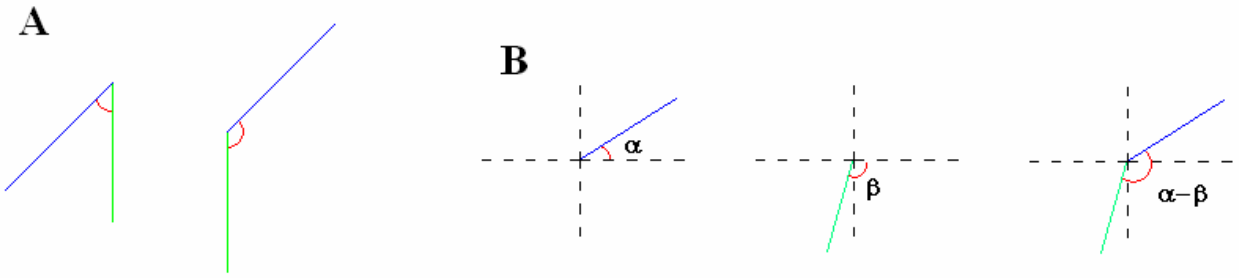

Figure 87. Example A: The same pair of segments/vectors matched in different ways can form different angles. Example B: The difference of independent vector angles results in the angle they form together.

Anyway, the direction given to each vector is still important. This method computes the slope of each vector line, what depends of the vector direction. Then, the vector angle can be obtained from slopes using inverse trigonometric function of arc tangent. Once again we have to care about vertical lines and their infinite slopes. However, we already know that the angle for an infinite slope is ± 90 degrees (or $\pm \pi/2$ radians).

The mathematical arc tangent function always can have two possible results: φ and $180+\varphi$. This means that obtained angles from the arc tangent of a line slope are possibly two because lines don't have directions like vectors have. It makes necessary to check if the default value ($-90º < \alpha < +90º$) from our obtained angle α is the correct one. Hence,

[20] Observation: The endpoint of a vector has to match the startpoint of another vector for immediate return checks, so a segment has to invert its path orientation before using this method. Which vector is inverted is not a decisive factor.

start and endpoints of each vector are checked for correcting the default obtained angles. An angle has to be corrected whether its vector direction goes from right to left. In case of correction, the vector angle has to be modified to the angle of the inversed segment direction (e.g. -135º instead of 45º, or 135º instead of -45º). A new range for possible angles will be the complete range $-180º < \alpha \leq +180º$ due to these corrections.

The last step of this method is to get the difference of the obtained angles of both vectors and return the minimal angle that these vectors form. The range of this angle difference is $0º \leq \alpha < 180º$, thus the resulting angle will always going to be minimal. If its complementary angle is wanted then it is quite easy to obtain: $360-\alpha$.

Orientations:

Sometimes it is also important to know if segments have the same orientation. This method was created to return a Boolean result indicating equal or different segment orientations. It is important to remark that '*orientation*' does not have the same meaning that '*direction*' has. Probably, different vectors have different directions, but at the same time, they can have the same orientation. The way that this method uses for obtaining its result gives to us an excellent notion of what orientation is.

This method uses the mathematical projection of a given vector on the line of another given vector for checking if they have the same orientations. This is the last method where vertical lines might cause problems.

We can check if the given vectors have the same orientation by examining the angle that one of these vectors forms with the projection of the other one. The figure 88 illustrates it. If such angle is zero, the vectors have the same orientation, but if this angle is 180º, the vectors have inverse orientation.

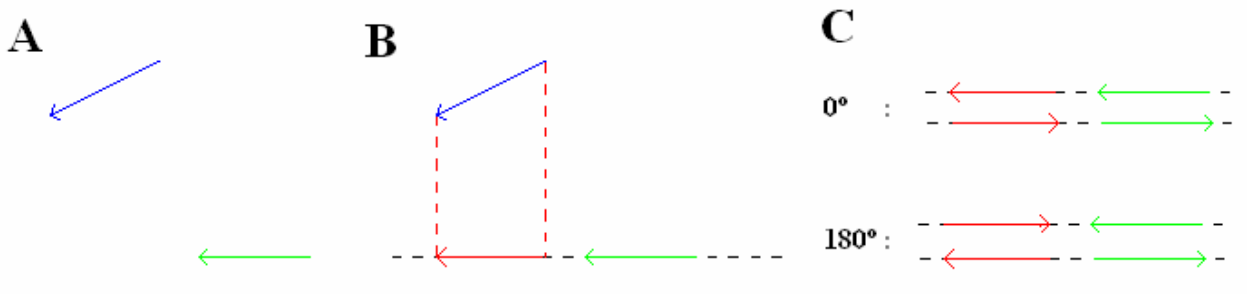

Figure 88. Example A: Two vectors for comparing their orientations. Example B: The projection of a vector over the line of the other vector. The red vector is the projection of the blue vector on the line of the green vector. Example C: Possibilities with respect to angles after projections. If the angle is zero the orientations are the same, otherwise they are inverse.

This method, as well as the prior ones, is not safe from exceptions. A vector could be transversal to the line of the other vector, and its projection would result in a single point without direction (nor angle). Even though it is controversial, it was decided to assume that in such case the vector orientations are the same. This controversial decision avoids

many problems, for instance, with unitary segments (with the same point as start and end).

Turnings:

Vector cross products are used here for the purpose of defining turning orientations. A positive cross product of two consecutive vectors v_1 x v_2 indicates that a left turning happens from v_1 to v_2, and a negative result indicates that a right turning happens from v_1 to v_2. Hence, only the sign of the resulting cross product interests for determining the turning orientation. Null signed results (zero) indicate that no turn happens. Comparing two turnings is as easier as defining them.

The method in this project program for computing angles is called *angle*. The method for checking orientations is called *sameOrientation*. The method for computing turnings is called *turnSign*, and the method for comparing two of them is called *sameTurn*. All of them belong to the *Segment* class and they can be found in the appendix C. There are other less important implemented methods from the *Point* and *Segment* classes that have not been discussed in this chapter. But they also can be found in the appendices B and C, they are short and can be understood with a simple view.

4.4 The *Node* class

This class is needed just for one reason, and this is the objective of its only method called *nodeAveraging* that can be found in the appendix D. Each instance of this class represents a node from the *map graph*. Each node represents a particular path obtained for this graph. The *map graph* is the only instance passed to this method as parameter. Its class member called *path* is loaded with the path it represents and its class member called *links* is loaded with the indices of all nodes from the *map graph* that must be linked to this one.

The final objective of the only method of this class is to reduce the *map graph* to a shorter and equivalent graph. To reduce the entire graph means to reduce each of its *complete connected sub-graphs*. Let us call *connected sub-graph* to a graph subset that has a, not necessarily direct, link from all to all of its nodes. These subsets are *complete* when there is no greater *connected sub-graph* including them.

The way for reducing each *complete connected sub-graph* is applying the method of this class to just one of the sub-graph nodes. This method is applied recursively to every node that the first one is linked to, caring to not be applied again to some node that has already been part of a reduction in this recursive procedure. This is the reason of the class member called *flag* that indicates if a node instance has already participated of some reduction process or not. The reduction of just two linked nodes means the average of the paths they represent.

Every node from the *map graph* is checked one by one, and this class method is applied to each one of these nodes that still doesn't have participated on a reduction process,

because such node is part of a new *complete connected sub-graph*. So, at the end of this check, the entire *map graph* will be reduced to only one node for each set of similar paths (or each *complete connected sub-graph*).

4.5 The *Path* class

The instances of this class represent the most important objects of this project. Therefore, it has more methods than any other class and they are essential methods. These methods have to be used for cleaning, detecting path similarity and averaging path instances.

The way how these essential methods work is not easily explained in short sentences. However, they already have been explained in the prior chapter of 'system analysis and design'. Therefore, Instead of explaining them in many paragraphs, it was preferred to write a pseudo-code for each method implemented in this class.

Actually, a textual explanation of these methods would be a repetition of the explanations from past chapters: perhaps a resumed explanation. Moreover, the pseudo-codes are also understandable and can offer a much better relationship between each part of the explanation and each part of the code. This means that a pseudo-code is much better related to each Java code line than a textual explanation would be.

4.5.1 Pseudo-codes

This subchapter is a comprehensive step. Once we have read the 'system analysis and design' chapter, we are ready to understand the following pseudo-codes, and if we read the pseudo-codes, we will be ready for understanding the Java codes. But reading and analyzing some of the next many consecutive pseudo-codes is just necessary if there is an interest or curiosity in some specific part of the program code, e.g. for improving part of the program, for optimizing some method, for correcting some detected error, etc. It is suggested to jump these pseudo-codes related to some part of the program of no interest.

If some pseudo-code is not understood, a simple review on a correspondent part of the system analysis and design should be sufficient for understanding it. Each pseudo-code will have a brief introduction, and there they will be related to methods and to prior correspondent sub-chapters. All their respective Java codes can be found in the appendix E for the *Path* class.

4.5.1.1 *Initial cleaning* pseudo-code

This first pseudo-code is for cleaning input data from a file and constructing path instances from it. The '*initial cleaning*' pseudo-code is related to the chapter 3.1 of *cleaning process*. It can be used for every type of cleaning mentioned in this chapter, excepting the *simple path cleaning* that will be represented by the next pseudo-code. The '*initial cleaning*' pseudo-code is related to the method called *cleaning* whose initial

parameters are: a file F_x that contains information about a particular path, and a flag indicating the type of wanted cleaning: 1 = *basic cleaning*, 2 = *general path cleaning*, 3 = *total path cleaning*.

1) Create necessary variables, including a path P_x to be loaded and a list L of resulting paths to be returned.

2) **While** there are (lines of) information to be read in the file F_x **do**:

   a) Read the information from the next file line and **if** the information is acceptable (is an *OK point*) **do**:

      i) Obtain the vertex p_i^x from the line information.

      ii) **If** p_i^x forms a very long segment [21] with its prior segment, split the file path by adding P_x to the list L, clearing P_x and adding p_i^x as its first element.

      iii) **Else**, **if** p_i^x does not fluctuate respect to its prior vertex and the solicited type of cleaning is *general* or *total* **do**:

         (1) Compare the segment formed by p_i^x with each segment of its *PSD range set* and **if** an *immediate return* occurs then split the file path by adding P_x to the list L, clearing P_x and adding p_i^x as its first element.

         (2) **If** the solicited type of cleaning is *total*, compare the segment formed by p_i^x with each prior segment from P_x, and **if** some *self-similarity* occurs then split the file path by adding P_x to the list L, clearing P_x and adding p_i^x as its first element.

         (3) Add p_i^x to P_x and update the variables that need to be updated.

3) Add the last built path P_x to L and **Return** the resulting path list L.

4.5.1.2 *Simple cleaning* pseudo-code

This pseudo-code is used for a fast cleaning of a path resulting from some averaging process. The '*simple cleaning*' pseudo-code also is related to the chapter 3.1 of *cleaning process*, but the only cleaning type that is related to it is the *simple path cleaning* (on the subchapter 3.1.4.2). The '*simple cleaning*' pseudo-code is related to the method called

[21] The first obtained segment is very long by default (because it doesn't exist any prior point to the first p_i^x), and then the path P_x has to be initialized instead of split.

simpleCleaning whose initial parameter is one path containing its respective vertex list: $P_x = p_0^x, p_1^x, ..., p_m^x$.

1) Create necessary variables, including a reference vertex p_{ref}^x (or reference point) that initially will be the first path vertex p_0^x.

2) **For** *i* from 0 to *m*-2, where *m* is the number of vertices from P_x , **do**:

   a) Compute the length of the segments s_i^x and s_{i+1}^x next to the vertex p_i^x.

   b) **If** some of these segments are too short, the vertex p_{i+1}^x is a *fluctuation vertex* of p_i^x, then, remove p_{i+1}^x from the path.

   c) **Else**, **do**:

      i) **If** p_i^x is not the reference vertex (what means that a fragment cleaning already started), add the lengths of s_i^x and s_{i+1}^x to a variable containing the total length l_f of the analyzed fragment. **Else** a new fragment begins to be cleaned, so make the total fragment length l_f equal to the length sum of s_i^x and s_{i+1}^x.

      ii) **If** the fragment length l_f is approximately equal to the distance between the fragment start point and the end point of the segment s_{i+1}^x, then the vertex p_{i+1}^x is not a necessary vertex once its absence practically doesn't change the fragment (even considering prior removals), then **do**:

         (1) **If** this is the first of some sequence of consecutive removals, then start a new fragment cleaning by making the new reference vertex p_{ref}^x equal to p_i^x.

         (2) Remove p_{i+1}^x from P_x.

4.5.1.3 *Path similarity* pseudo-code

This short pseudo-code is for detecting the similarity between two given paths. It is also going to check if these paths would be *similar oriented paths* whether one of them is inversed. However, this pseudo-code doesn't detect similarity by itself, but it calls another method (represented by the next pseudo-code) for doing it. Its only work is related to the management of the *similarity detection process*, and the introduction of the subchapter 3.2.2 is enough for understanding this management. The '*Path similarity*'

pseudo-code represents the method called *pathSimilarity* whose initial parameters are: two paths containing their respective vertex lists, $P_x = p_0^x, p_1^x, ..., p_m^x$ and $P_y = p_0^y, p_1^y, ..., p_n^y$, and a maximal length *tle* for path extremes that can be tolerated during path comparisons.

1) **Call** the method represented by the *'oriented similarity'* pseudo-code for computing an oriented average of the same given paths P_x and P_y.

2) **If** the result from the called method is a non-empty array (P_x and P_y are similar oriented paths) then **return** the same array as result.

3) **Else**, invert the path P_y and call the same method again for P_x and the inversed path P_y.

4) **Return** the resulting array: A non-empty resulting array indicates oriented similarity of P_x and the inverse P_y, and another empty resulting array indicates no similarity.

4.5.1.4 *Extremity similarity* pseudo-code

This pseudo-code checks if a path is similar to another one in their original orientations. This pseudo-code represents a method called by another one (represented by the prior pseudo-code). However, the similarity between two paths is still not checked here. Another further method (represented by the '*oriented similarity*' pseudo-code) is called for that. The '*Extremity similarity*' pseudo-code cares about eliminating those parts of tolerable path extremities that are not similar, and if no similarity is found in these extremities this method directly denies path similarity instead of calling another method.

The '*extremity similarity*' pseudo-code is related to the complete subchapter 3.2.2.1 of the *similarity detection process*. This pseudo-code represents the method called *isSimilar* whose initial parameters are: two paths containing their respective vertex lists, $P_x = p_0^x, p_1^x, ..., p_m^x$ and $P_y = p_0^y, p_1^y, ..., p_n^y$, and the maximal length *tle* for path extremities that can be tolerated in path comparisons.

1) **For** *i* from 0, while *i* is the index of a segment s_i^x that has some point belonging to the first tolerable path extremity [22] of P_x:

   a) **For** *j* from 0, while *j* is the index of a segment s_j^y that has some point belonging to the first tolerable extremity of P_y:

[22] Remark: the first/last tolerable extremity of a path are all its consecutive segments and sub-segments (from the first/last one) whose length sum do not exceed a defined maximal tolerable length, that in this algorithm is the given *tle* parameter.

i) **If** the segments s_i^x and s_j^y have the same orientation and they have some points that are *close enough* then finish both 'for' cycles. Two close segments, from different paths, were found in the initial extremities.

2) **If** two close segments s_i^x and s_j^y, from different paths, were found in the initial extremities, **do**:

a) **Call** the *'similarity limit'* pseudo-code, passing s_i^x and s_j^y as parameters, for computing the approximate points p_{init}^x and p_{init}^y from where the similarity begins.

b) Check if the computed points p_{init}^x and p_{init}^y belong to the first tolerable extremities of their correspondent paths. **If** some of them don't, **return** an empty array as result indicating no path similarity.

3) **Else**, **return** an empty array as result indicating no path similarity.

4) **For** *i* from 0, while *i* is the index of a segment s_i^x that has some point belonging to the last tolerable extremity of P_x:

a) **For** *j* from 0, while *j* is the index of a segment s_j^y that has some point belonging to the last tolerable extremity of P_y:

i) **If** the segments s_i^x and s_j^y have the same orientation and they have some points that are *close enough* then finish both 'for' cycles. Two close segments, from different paths, were found in the final extremities.

5) **If** close segments s_i^x and s_j^y, from different paths, were found in the final extremities:

a) **Call** the *'similarity limit'* pseudo-code, inversing and passing the segments s_i^x and s_j^y as parameters, for computing the approximate points p_{end}^x and p_{end}^y to where the similarity ends.

b) Check if the computed points p_{end}^x and p_{end}^y belong to the last tolerable extremities of their correspondent paths. **If** some of they don't, **return** an empty array as result indicating no path similarity.

c) Define a new path P_x' cloning only the part from P_x that are between p_{init}^x and p_{end}^x (they included). Also define a new path P_y' cloning only the part from P_y that are between p_{init}^y and p_{end}^y (they included). Here, the tolerable extremities are discarded.

6) **Else**, **return** an empty array as result indicating no path similarity.

7) **Call** the *'oriented similarity'* pseudo-code for checking if the cloned paths P_x' and P_y' are similar oriented paths by passing them as parameters. After it, **return** the resulting path.

4.5.1.5 *Similarity limit* pseudo-code

This short pseudo-code computes the point of a given segment where the similarity with another given segment begins. This pseudo-code represents a method called by another one (represented by the prior pseudo-code). The '*similarity limit*' pseudo-code is related to the search of the *approximated first similarity point* mentioned at the end of the subchapter 3.2.2.1 of the *similarity detection process*. This pseudo-code represents the method called *similarityLimit* whose initial parameters are: two given segments (from different paths to be compared) defined by their respective start and endpoints, $s_m^x = (p_s^x, p_e^x)$ and $s_n^y = (p_s^y, p_e^y)$, where the first defined segment s_m^x is the one where the similarity limit wants to be found.

1) Create and initialize necessary variables, including a segment to be a sub-segment s_{sub} of the segment s_m^x. Initially, s_{sub} is equal to s_m^x.

2) **While** s_{sub} is not shorter than an stipulated small distance (enough for saying that points separated by this distance might be considered the same), **do**:

   a) Update the subsegment s_{sub} by discarding its half whose endpoint is not close enough[23] to the other given segment s_n^y.

3) **Return** the middle point of the resulting subsegment s_{sub}.

4.5.1.6 *Oriented similarity* pseudo-code

This pseudo-code checks if a path is similar to another one in their original orientations. It represents a method called by another method (represented by the '*extremity similarity*' pseudo-code). The '*oriented similarity*' pseudo-code is related to the *oriented distance-similarity by using approximation sets* (subchapter 3.2.1.3.2 of the *similarity detection process*). This pseudo-code represents the method called *similarityDetection* whose initial parameters are: two paths containing their respective vertex lists, $P_x = p_0^x, p_1^x, ..., p_m^x$ and $P_y = p_0^y, p_1^y, ..., p_n^y$.

[23] Exception: Here, a '*close enough'* distance has to be a little smaller than PSD because we must ensure that the resulting approximate limit point will not be out of the similarity area.

1) Create necessary variables, including an empty *approximation set* A^y for the path P_y, another empty set A^y_{last} to be the last computed *approximation set*, a Boolean variable *flag* that indicates if still there are possible segments from P_y to be added to A^y, and a counter $j = 0$ where j will be the segment index for the path P_y.

2) **For** i from 0 to m, where m is the number of vertices from the path P_x , **do:**

   a) Set the *flag* for indicating that there are still possible segments to be added to the approximation set A^y**.**

   b) **While** the *flag* indicates that still there are possible segments to be added to the approximation set A^y and j is not greater than n-1, where n-1 is the number of segments from P_y, **do**:

      i) **If** the segment s^y_j is *close enough* to the vertex p^x_i and such segment doesn't cause similarity gap[24] in the *approximation set* A^y, add s^y_j to the end of A^y.

      ii) **Else**, s^y_j is not as approximation set, then **do**:

         (1) **If** the *approximation set* A^y for the vertex p^x_i is not empty (it has been defined), **do**:

            (a) Set *flag* for indicating that there are no more segments to be added to this approximation set A^y**.**

            (b) **If** there is a non-empty approximation set A^y_{last} before A^y, it is necessary to check if there is similarity gap between them, then **do**:

               (i) **If** the initial vertex of A^y is before the initial vertex of its prior *approximation set* A^y_{last} or **if** the final vertex of A^y is before the final vertex of A^y_{last} , **return** a value indicating failure.

               (ii) Check if there is some similarity gap between A^y and A^y_{last} by comparing each vertex (from P_y) between them with the segment from p^x_{i-1} to p^x_i . **If** a vertex is not *close enough* to this segment in some of these comparisons, **return** a value indicating similarity failure.

[24] If the start point of every segment (excepting the first added one) from an *approximation set* is *close enough* to its respective vertex, we may be sure that there is no 'similarity gap' for such set.

(c) Set A^y_{last} with the value of the approximation set A^y and empty A^y. After it, set *j* to zero for seeking a new approximation set for the next vertex of P_x.

(2) **Else**, **do**:

(a) **if** $j = n-1$ (there are no more segments to be checked) then a vertex do not have *approximation set*. Therefore, **return** a value indicating similarity failure.

(b) **Else**, increase j.

3) **Return** a value indicating that the parameter paths are completely similar.

4.5.1.7 *Path averaging* pseudo-code

This pseudo-code is for computing an average path for two given paths. However, this pseudo-code doesn't compute the average by itself, but it calls other methods for it. It uses the convex triangulation (subchapter 3.3.2) for averaging paths and it is more specifically related to the *main procedure for the path averaging process* of the subchapter 3.3.2.1. The '*Path averaging*' pseudo-code represents the method called *pathAveraging* whose initial parameters are: two paths containing their respective vertex lists, $P_x = p^x_0, p^x_1, ..., p^x_m$ and $P_y = p^y_0, p^y_1, ..., p^y_n$.

1) **If** some initial path is empty (so there is no average), **return** immediately an empty *path* as result.

2) Create necessary variables, including the origin and destination *points* (p_o , p_d) for the average of both initial paths, two growable arrays for being the fragment lists (L_x and L_y) of each given path, and an empty *path* P_{avg} to store the vertices of the average fragments.

3) It is necessary to load L_x and L_y with the elements from their respective given paths, but ensuring that they will have a common origin and destination point, thus they will form at least one polygon. These lists are going to be fragmented later. This step is divided in other three sub-steps:

a) Compute the average point p_o (new origin point) for the first vertex of each given path (p^x_0 and p^y_0) and add p_o at the beginning of both fragment lists L_x and L_y whether it is necessary. **If** p_o is equal to p^x_0 and/or p^y_0 , it is not necessary.

b) Copy all elements from the vertex lists of the given paths P_x and P_y to the correspondent fragment lists L_x and L_y.

c) Compute the average point p_d (new destination point) for the last vertex of each given path (p_m^x and p_n^y) and add p_d at the end of both fragment lists L_x and L_y whether it is necessary. **If** p_d is equal to p_m^x and/or p_n^y , it is not necessary.

4) **If** the computed origin and destination points are the same ($p_o = p_d$), the resulting average is a path with a unique vertex. Then, **return** a unitary average path as result and end the procedure.

5) **Call** the *'fragment lists'* pseudo-code for fragmenting the lists L_x and L_y, converting them in lists of path fragments ($L_x = f_0^x, f_1^x, ..., f_k^x$ and $L_y = f_0^y, f_1^y, ..., f_k^y$) divided by all intersection points (remembering that at least the origin points and destination points intersect). **If** some error result is returned from this method, jump to the step 7.

6) **For** *i* from 0 to *k***,** where *k* is the last element index of both fragment lists, **do**:

   a) **Call** the *'polygon average'* pseudo-code (from the class *Polygon*), passing a *polygon* formed by f_i^x and f_i^y (the correspondent fragments from each fragment list) as parameter.

   b) **If** an empty array is received from the called process (some error happens), finish the 'for' cycle. **Else** add the returned average fragment f_i^{avg} to the end of the average path P_{avg}.

7) **If** some error occurred before in some called procedure, **return** the given path with higher *total weight* as result, where the *total weight* of a path is the difference of its *weight* and its *counterweight* (in case that both paths have the same *total weight*, one of them is going to be returned by default).

8) No error happened before then **return** the average path P_{avg}.

4.5.1.8 *Fragment lists* pseudo-code

This pseudo-code is for fragmenting two paths according to the polygons that they form together. The '*fragment lists*' pseudo-code is called by the method represented by the prior pseudo-code, and it is also related to the subchapter 3.3.2.1 but more specifically related to *decomposing a polygon in simple polygons* (also in this subchapter). The '*fragment lists*' pseudo-code represents the method called *createFragmentLists* whose initial parameters are: two given growable arrays, initially containing the vertex list of a respective path, with similar origin points and similar destination points, $L_x = p_0^x, p_1^x, ..., p_m^x$ and $L_y = p_0^y, p_1^y, ..., p_n^y$ where $p_o = p_0^x = p_0^y$ and $p_d = p_m^x = p_n^y$.

1) Clone the data from the given arrays to other arrays (because the parameter arrays have to be modified and the initial data must not be lost).

2) **If** the origin or destination points of the arrays are different, **return** an error message (this method cannot continue).

3) Create necessary variables.

4) **For** *i* from 0 to *m*-1, where *m*-1 is the number of segments of a given list, **do**:

   a) **For** *j* from 0 to *n*-1, where *n*-1 is the number of segments of the other given list, **do**:

      i) Compare the segment s_i^x from a list to the segments s_j^y from the other list checking if they intersect.

      ii) **If** the checked segments do intersect, and their intersection is just a point[25], and it is a new found intersection (not equal to the prior one), then:

         (1) Confirm that the new computed intersection point is not before the prior computed intersection point in any of both given lists. **If** it is (so the order of intersections fails for some list) **return** a value indicating the failure.

         (2) Create a fragment *f* and add to it every vertex between the two last found intersections of the given fragment list L_x. Also include such intersections at the start and the end of *f* respectively, except if such intersections are equal to some other added vertex.

         (3) Add *f* to L_x, overwriting the prior list of vertices contained in L_x.

         (4) Repeat the steps (2) and (3) for the other list L_y.

5) **Return** a Boolean value indicating success.

[25] Observation: If the intersection of two segments s_a^x and s_b^y is also a segment, this intersecting segment is equal to s_a^x or s_b^y, let say s_b^y, and only two intersecting points are interesting: both endpoints of s_b^y. But its first endpoint was already found in the 'for' cycles when the prior segment to s_b^y was compared to s_a^x. As well as its last endpoint is going to be found again when the next segment to s_b^y is compared to s_a^x in the continuation of the 'for' cycles. So we can just ignore intersecting segments.

4.5.1.9 *Fragment averaging* pseudo-code

This pseudo-code is for averaging the first detectable fragment similarity between two given paths whether they have some fragment similarity. Actually, it calls another method for performing averages and it doesn't detect more than one fragment similarity (per time) between two paths, but this method can be applied more than once to the same pair of paths, if necessary, for ensuring every fragment similarity to be averaged. The '*fragment averaging*' pseudo-code is related to the *fragment averaging process* of the subchapter 4.3.2, and it represents the method called *fragmentAveraging* whose initial parameters are: two paths containing their respective vertex lists, $P_x = p_0^x, p_1^x, ..., p_m^x$ and $P_y = p_0^y, p_1^y, ..., p_n^y$, and a list of paths *L* where to add resulting paths from this pseudo-code.

1) **For** *i* from 0 to *m*-1, where *m*-1 is the number of segments from P_x

   a) **For** *j* from 0 to *n*-1, where *n*-1 is the number of segments from P_y :

      i) **If** the segment s_i^x is *close enough* to the segment s_j^y, **do**:

         (1) Create necessary variables for seeking segments (in respective paths) before and after s_i^x and s_j^y that are also part of the similar fragment.

         (2) **While** some prior/next[26] segment to the first/last found segment of its computed[27] fragment is similar to the first/last found segment of the other computed similar segment, then keep computing the similar fragments by seeking and adding prior/next similar segments.

         (3) Use the first/last endpoints of the first/last segments for constructing the similar fragments f^x and f^y, but considering that f^x and f^y must get the same orientation. Also exclude non-similar fragment extremities by replacing their first/last segment endpoints by the first/last similarity points. For defining these similarity points, it is necessary to **call** the *'similarity limit'* pseudo-code.

         (4) **Call** the *'path averaging'* pseudo-code for averaging f^x and f^y.

         (5) **If** the fragment average f_{avg} is a non-empty fragment, **do**:

            (a) Create a resulting path P_x' by replacing the initial fragment f^x by f_{avg} in the path P_x, and add P_x' to the list *L* of resulting paths.

[26] Observation: Actually, it is not necessary to check the prior segment to s_i^x because we already know that s_i^x is the first segment from its similar fragment.

[27] The initial computed fragments, for each path respectively, contain only s_i^x and s_j^y.

(b) Create a clone path P_{ant} of only the vertices from P_y that are before the defined fragment f^y (if there is some), and link P_{ant} to the resulting path P_x' by adding the correspondent endpoint of the average f_{avg} to the end of P_{ant}.

(c) Add P_{ant} to the list L of resulting paths **if** it is not a too short resulting path.

(d) Create a clone path P_{pos} of only the vertices from P_y that are after the defined fragment f^y (if there is some), and link P_{pos} to the resulting path P_x' by adding the correspondent endpoint of the average f_{avg} to the start of P_{pos}.

(e) Add P_{pos} to the list L of resulting paths **if** it is not a too short resulting path.

(f) **End** the 'for' cycles (ending this pseudo-code at the same way).

4.6 The *Polygon* class

The instances of this class represent basic objects for the performance of the most elaborated process of this project. This class offers the methods that make possible the *path averaging process*.

The way how these essential methods work is not easily explained in short sentences. Therefore, once more we are going to use pseudo-codes for explaining their performances.

4.6.1 Pseudo-codes

This subchapter is a comprehensive step like the last subchapter for pseudo-codes, and only the pseudo-codes of interest need to be analyzed.

Like in the subchapter for *Path* classes, each pseudo-code will have a brief introduction, and there they will be related to methods and to prior correspondent sub-chapters. All their respective Java codes can be found in the appendix F for the *Polygon* class.

4.6.1.1 *Polygon average* pseudo-code

The first pseudo-code of the *Polygon* class is for computing the average of a given polygon defined by two fragments. Actually, it split the given polygon in sub-polygons and calls another method for averaging sub-polygons. The '*polygon average*' pseudo-code is related to the *polygon average procedure* of the subchapter 3.3.2.2, and it

represents the method called *polygonAveraging* whose initial parameters are: a polygon defined by a fragment $f^x = p_0^x, p_1^x, ..., p_m^x$ from the *fragment list* of a path, a fragment $f^y = p_0^y, p_1^y, ..., p_n^y$ from the *fragment list* of another path, the origin and destination points of the polygon ($p_o = p_0^x = p_0^y$ and $p_d = p_m^x = p_n^y$), and the *weights* of each fragment.

1) Create the necessary variables, including a resulting average fragment f_{avg} already containing the origin point p_o, two empty *convex lists* cl^x and cl^y for the correspondent polygon fragments, and an index of the *leader vertex* of each convex list (lv^x and lv^y) that initially have the value 0 from the origin vertices.

2) Check some initial possible casualties. Do:

   a) If both origin vertices p_0^x and p_0^y have the destination vertex p_d as their next vertex, add p_d to the end of the *average fragment* f_{avg} and return it (actually, the polygon is a segment).

   b) **If** only one origin vertex, p_0^x or p_0^y, has the destination point p_d as its next vertex, find the average point p_{avg} of the origin p_o and destination p_d, then add the average point between them in the correspondent fragment as a new vertex ($f^x = p_o, p_{avg}^x, p_d$ or $f^y = p_o, p_{avg}^y, p_d$).

3) **Call** the *'convex list'* pseudo-code twice for defining each next *convex lists* cl^x and cl^y for each polygon fragment, passing both fragments (f^x and f^y), the index of both leader vertices (lv^x and lv^y) and the destination point p_d as parameters. **If** some resulting convex list is empty because of an occurred error, **return** an empty array as result.

4) **If** no leader vertex could advance (both *convex lists* cl^x and cl^y contain only one element), a special situation has to be handled, then **do**:

   a) **If** no *leader vertex* has the destination point as its next vertex, something went wrong in the average. **Return** an empty array as result.

   b) **If** the next vertex of both *leader vertices* lv^x and lv^y is the destination vertex p_d, add p_d to the average fragment f_{avg} and **return** it (the polygon average is complete).

   c) **If** only one *leader vertex lv* has the destination vertex p_d as its next vertex, compute the average vertex between *lv* and p_d, add it to the correspondent fragment between these vertices, clean both *convex lists* and **go back to the step 3** (due to flagged conditions and a **while** loop). A counter is used to **return** an empty array (indicating failure) **if** this situation repeats many consecutive times.

5) Here, the convex lists cl^x and cl^y contain the vertices of an internal polygon that we want to be convex, but perhaps it is not convex yet. Define the convex sub-polygon formed by these lists calling the *'convex sub-polygon'* pseudo-code, passing both lists as parameters.

6) **If** the called pseudo-code indicates failure or its results are not acceptable, **return** an empty array as result. Otherwise, update the *convex lists* cl^x and cl^y forming a convex sub-polygon.

7) Call the *'convex sub-polygon average'* pseudo-code passing the updated *convex lists* cl^x and cl^y as parameters. Add the resulting sub-polygon average fragment to the polygon average fragment f_{avg}.

8) The last vertex from each *convex list* becomes the *leader vertex* of its fragment. Empty the *convex lists* cl^x and cl^y and **go back to the step 3**.

4.6.1.2 *Convex List* pseudo-code

This pseudo-code is for creating a *convex list* for a given fragment (and its *leader vertex*) with respect to another given fragment (and its *leader vertex*). The '*convex list*' pseudo-code is also related to the *polygon average procedure* of the subchapter 3.3.2.2, but more specifically to *defining sub-polygons with convex lists*. This pseudo-code represents the method called *createConvexList* whose initial parameters are: two defining polygon fragment $f^x = p_0^x, p_1^x, ..., p_m^x$ and $f^y = p_0^y, p_1^y, ..., p_n^y$, an index for the *leader vertex* of each given fragment, and the polygon destination point p_d.

1) Create and initialize the necessary variables: a resulting *convex list* cl^x for the first given fragment f^x, the *first inner segment* s_i of the subpolygon that this expects to create, and a segment s_a^x to be analyzed and its prior segment s_p^x (both from the same *convex list* cl^x). These variables must have the initial values:

   a) Initially, cl^x contains just its *leader vertex*.

   b) The first inner segment s_i links both *leader vertices* indicated by the given indices.

   c) Initially, s_a^x is the segment formed by the *leader vertex* of cl^x and its next vertex.

   d) Normally, s_p^x is the segment from f^x prior to s_a^x, but **if** s_a^x do not have a prior segment in the fragment (so, the *leader vertex* of cl^x is the polygon origin point), assume s_i as the prior segment s_p^x.

2) **If** the leader vertex of cl^x is equal to the other leader vertex (so, both are the polygon origin point) or the segments s_a^x and s_i don't form an internal angle greater than 180º, **do**:

   a) Compute the side that cl^x turns to, create and update necessary variables for it.

   b) **While** the analyzed segment s_a^x doesn't reach the polygon destination point and it keep forming a convex list and it doesn't form an spiral with the prior analyzed segments, **do**:

      i) **If** an *inner immediate return* is detected, **return** an empty *convex list* indicating that it is not possible (for these processes) to compute such polygon average.

      ii) **If** the value for the turning variable (for the *convex list* cl^x) is zero and this new analyzed segment s_a^x do turn, keep its turning direction as the new value for the turning variable.

      iii) Add s_a^x to the *convex list* being created and update the segments s_a^x and s_p^x to their next segments.

3) **Return** the created *convex list* as result.

4.6.1.3 *Convex sub-polygon* pseudo-code

This pseudo-code is for defining convex sub-polygons by fixing two given *convex lists*. It calls another method for checking if some of these givens *convex lists* must be fixed. The '*convex sub-polygon*' pseudo-code is related to the *sub-polygon definition procedure* of the subchapter 3.3.2.3, and it represents the method called *toConvexSubpolygon* whose initial parameters are: two arrays containing convex lists $cl^x = p_0^x, p_1^x, ..., p_m^x$ and $cl^y = p_0^y, p_1^y, ..., p_n^y$ from the polygon fragments.

1) **If** both parameter *convex lists* turn to the same side, only the first segment of the internal *convex list* is going to be considered: remove all vertices of such list forming other segments. This situation is possible only **if** both *convex lists* have more than one segment. The turning from the first to the second segment of each *convex list* has to be analyzed.

2) The *'list reduction'* pseudo-code is going to be called for knowing if the first parameter *convex list* cl^x must be reduced to form a convex sub-polygon. **While** it must be reduced **do**:

a) **If** cl^x has more than one vertex, update the list removing its last vertex.

b) **Else**, no more elements must be removed from cl^x. Keep the unique element of cl^x and update the other list cl^y eliminating every vertex from it, except its first and last vertices.

3) The *'list reduction'* pseudo-code is going to be called for knowing if the second parameter *convex list* cl^y must be reduced to form a convex sub-polygon. **While** it must be reduced **do**:

a) **If** cl^y has more than one vertex, update the list removing its last vertex.

b) **Else**, no more elements must be removed from cl^y. Keep the unique element of cl^y and update the other list cl^x eliminating every vertex from it, except its first and last vertices.

4) **If** some failure happened during the called pseudo-code, **return** a value indicating such failure, **else return** a value indicating success.

4.6.1.4 *List reduction* pseudo-code

This pseudo-code is for checking and informing if a given *convex list* must be fixed for forming a convex sub-polygon with another given *convex list*. The '*list reduction*' pseudo-code is also related to the *sub-polygon definition procedure* of the subchapter 3.3.2.3, but more specifically to *fixing convex lists*. This pseudo-code represents the method called *mustBeReduced* whose initial parameters are: two arrays containing convex lists $cl^x = p_0^x, p_1^x, ..., p_m^x$ and $cl^y = p_0^y, p_1^y, ..., p_n^y$ from the polygon fragments.

1) **If** both given *convex lists* are unitary (only has one vertex), **return** a value indicating error.

2) Create the *last inner segment* s_i defined by the last vertices of both given *convex list*, and create also two segments to be analyzed (s_a^x and s_a^y, one for each *convex list*).

3) **If** some *convex list* is unitary (only has one vertex), **do**:

a) Define the analysis segment of the non-unitary *convex list*. The segment endpoints are the two last vertices of such list.

b) **If** the *convex lists* have not a common start point, define the analysis segment of the unitary *convex list* by equalizing it to the *first inner segment* (defined by the first vertex of both lists).

c) **Else**, there is no *first inner segment* then define the analysis segment of the unitary *convex list* by equalizing it to the first segment of the non-unitary *convex list*.

d) One *convex list* is unitary and must not be reduced, then **if** the angle formed by some analyzed segment (s_a^x or s_a^y) and the *last inner* segment s_i is greater than 180º, **return** a value indicating the non-unitary list that must be reduced.

4) **If** no *convex list* is unitary, **do**:

a) It is necessary to find out the turning direction for each *convex list*, then **do**:

i) **If** the start points of both *convex lists* are not common, check the turning direction of each *convex list* by analyzing the turning from the *first inner segment* to the first segment of the respective list.

ii) **Else**, there is no *first inner segment* then use the first segment from the other *convex list* (instead of the *first inner segment*) for the same analysis mentioned at the prior sub-step.

b) **If** some of these segments cause no turning, then ignore the first segment of both *convex lists* (if they have one) by removing the first vertex of each, and try to compute the turnings again by calling back this recursive pseudo-code and passing the altered *convex lists* as parameter. **Return** the same resulting value received from this recursive call.

c) Once the turning directions are defined, redefine both analysis segments s_a^x and s_a^y used before as auxiliary variables: define them by equalizing their first endpoint with the vertex before the last vertex of their correspondent *convex lists*, and equalizing their last endpoints with the last vertex of their correspondent *convex lists*.

d) It is necessary to check if the turning directions from s_a^x and s_a^y to the *last inner segment* s_i are the same than the turning directions of their respective *convex lists*, then **do**:

i) **If** the turning direction changes from s_a^x to s_i, **return** a value indicating that the other *convex list* cl^y must be reduced (and end this pseudo-code).

ii) **If** the turning direction changes from s_a^y to s_i, **return** a value indicating that the other *convex list* cl^x must be reduced (and end this pseudo-code).

5) **Return** a value indicating that no *convex list* must be reduced.

4.6.1.5 *Convex sub-polygon average* pseudo-code

This pseudo-code is for computing the average of a convex sub-polygon formed by two given *convex lists*. The '*convex sub-polygon average*' pseudo-code is related to the *convex sub-polygon averaging procedure* of the subchapter 3.3.2.4, and it represents the method called *convexSubpolygonAveraging* whose initial parameters are: two arrays containing *convex lists* $cl^x = p_0^x, p_1^x, ..., p_m^x$ and $cl^y = p_0^y, p_1^y, ..., p_n^y$ from the polygon fragments.

1) Create an array where to load the vertices of the average fragment f_{avg}^g, and create *turn vertices* (p_t^x and $p_{t'}^y$) for each given *convex list* cl^x and cl^y, initializing them with the value of the first vertex of their correspondent *convex lists* ($p_t^x = p_0^x$ and $p_{t'}^y = p_0^y$).

2) Compute the average of both *turn vertices* and add it to the *average fragment* f_{avg}.

3) **If** the *turn vertices* are equal ($p_t^x = p_{t'}^y = p_o$), then **do**:

   a) **If** some *convex list* is unitary, redefine the *turn vertex* of the other non-unitary list with the value of its next vertex (but do not compute nor add a new average).

   b) **Else**, redefine both *turn vertices* p_t^x and $p_{t'}^y$ with the value of the next vertex from their correspondent *convex lists* and compute the average vertex p_{avg} between them. **If** this average vertex p_{avg} is not approximately equal to the polygon origin point, add p_{avg} to the empty *average fragment* f_{avg}^g.

4) **While** some *turn vertex* has nor reached the end of its *convex list* yet **do**:

   a) **If** no *turn vertex* has reached the end of its *convex list* yet, it has to be decided which of them must advance next. Then, find out which segment next to a *turn vertex* forms the lower angle with the segment defined by both *turn vertices* (if both angles are equal, a segment is chosen by default). Advance the *turn vertex* that defines such segment by equalizing it to the value of its next vertex.

   b) **Else**, advance the *turn vertex* that didn't reach the end of its *convex list* yet (equalizing it to the value of its next vertex).

   c) Compute the average of the *turn vertices* and add it to the end of the *average fragment* f_{avg}.

5) **Return** the average fragment f_{avg} as result.

4.7 The *PathFinder* class

There is only one method for the *PathFinder* class, and this is the main method of the program. This method can be better explained textually, and only one small pseudo-code is going to be used during explanations. This pseudo-code is used just for part of this method.

Initially, all files contained in a particular folder are going to be read by the program, obtaining information from them for creating path instances. At the same time, the information is cleaned by a chosen cleaning method (referent to the *'initial cleaning'* pseudo-code). The folder name and directory position are defined in the construction of this class instance.

After all the initial path instances are created, they are compared one to another. A graph is created by constructing a node for each path instance and linking pairs of similar paths. The method represented by the *'path similarity'* pseudo-code is used for the similarity comparisons of this graph building process.

Once the graph is completely defined, it is time to find sets of linked nodes (of similar paths) that can be replaced by only one new average node (or average path). This process may reduce the number of paths considerably. The unique *Node* class method is used for these averages.

Even when there are no complete paths to be replaced anymore, we might still have many similar path fragments that should be replaced for obtaining a nicer resulting map from this main method. The sub-procedure being explained next is a little more complex, so a pseudo-code will be used to explain how all the similar fragments are computed. A list L for the present cleaned and averaged paths is the only variable defined before the next pseudo-code that is going to be used for it.

1) Create necessary variables, including an array P_r where to load every resulting path of this process, a pointer index p to some element of P_r, and a stack S where to load a particular path (for working with) and its resulting paths.

2) The first path from L is directly moved to P_r.

3) **While** L still have paths to be checked, **do**:

    a) Move the last path from L and add it to S.

    b) Make the pointer p to point to the last element (path) of P_r.

    c) **While** there is a path from P_r prior to p, **do**:

        i) **For** i from n to 0, where n is the initial number of elements (paths) from S, **do**:

(1) Compare the *i*-th path from *S* to the path from P_r pointed by *p*. **If** some fragment similarity is detected, average the fragments of this similarity, remove the *i*-th path from *S*, and add every resulting path to the top of *S*.

(2) **If** a fragment average happened in the prior sub-step (the size of *S* got bigger), remove the *i*-th path from *S*.

ii) **If** the path from P_r pointed by *p* was averaged with some path from *S*, remove such path from P_r.

iii) Decrease *p*.

d) Move all resulting paths from *S* to P_r.

4) Copy every resulting path from P_r to the empty list *L*.

After this pseudo-code, we expect the similar fragments to be averaged, giving to us a final list of paths ready to be added to a map. Displaying a path map is the last process of the main method of this program. Though, the final list *L* becomes a member of the panel that will display the map (the *MapPanel* class). A position in the map is chosen to be the map center, once this unlimited map can be displaced in the panel. Initially, at least part of the map paths wants to be seen in the panel.

5 TEST

In this chapter, the results of the implemented processes are going to be discussed individually. The general result of the project implementation was successful once it has shown the accomplishment of the most elaborated processes. The more elaborated a process was, the best results it demonstrated in this tests.

Even though it was acceptable to find some little failures in the first process of the pathfinder system, this *cleaning process* did not show any 'very rare' situation (e.g. weird *immediate returns*) that might cause such inconveniences, and then, no problem for this process was found during the tests discussed here.

In the second process, the *path similarity detection*, we did not expect any failure and no failure occurred. The results of this process were tested by checking the *map graph* before the *path averaging process* was performed. The *map graph* was the wished one, linking only those nodes of paths that were considered similar.

The most elaborated process was completely successful in these tests. The *path averaging process* did not show undesired results in any situation, even when these situations were made more difficult on purpose.

The fourth process for *fragment averaging* was not very elaborated in this project. Hence, even if it has shown some failures, these failures were accepted. We also should also consider that this not very elaborated process tried to supply the fifth process as well as it could.

The *stitching process* was not implemented and it has shown its necessity in these tests. The *fragment averaging process* tried to replace this fifth process but did not achieve it very well. The gap of the *stitching process* kept being notorious.

The figure 89 illustrates a map that results from 45 read files. The initial path *cleaning process* was applied on these input files and 44 paths resulted from them. We must remember that very short paths may be ignored and some paths may be split.

This figure, of the complete initial map, is not large enough too for showing the inconsistency of the map that we can have at the beginning of the pathfinder processes, even after the cleaning of each individual path. But it shows a map area enclosed in a red circle that is illustrated in the next figure 90.

The figure 90 zooms in the initial map illustrating one of the most conflictive regions of the initial map, and there we can observe several shuffled lines that we would like to fix by using average paths.

We can observe the success of this project when we compare these two first figures (of the initial map) with the next figures of the resulting map. But first, it is interesting to discuss about the value of the constants used in the pathfinder program.

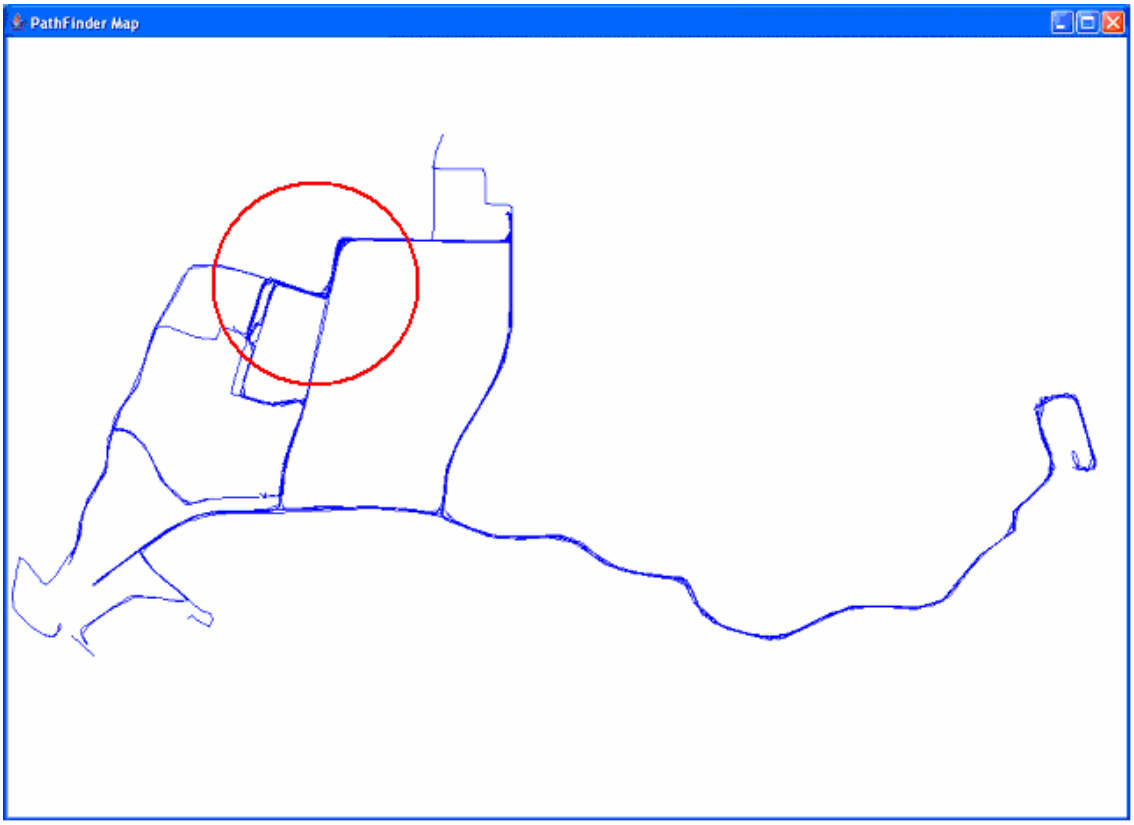

Figure 89: The complete initial map without any process performance. The red circle encloses part of the map illustrated in the next figure.

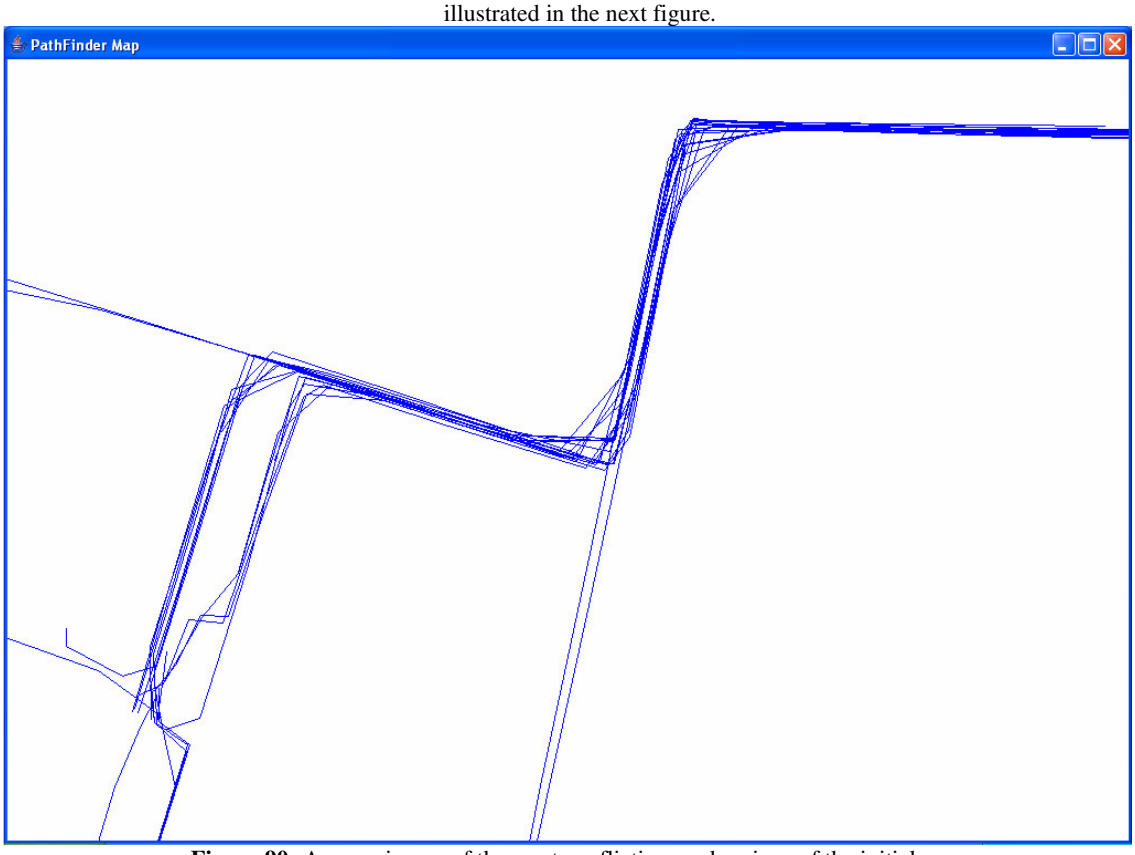

Figure 90: A zoom in one of the most conflictive road regions of the initial map.

The values that we set up for the program constants are fundamental for having a nice resulting map. Obtained results with wrong constant values don't use to be satisfactory. Therefore, we have to know the properties of the received input data and analyze the domain from where it comes.

For instance, we should not expect a nice result if we receive data with an error margin of 15 meters but we set up the constant *EMD* with the value 2, supposing that the maximal distance failure due to an error margin is up to 2 meters.

Another example of a bad set-up is when we monitor a fast mobile object running at 100 Km/h and the signal comes from the satellite each 10 seconds but we set up the constant *MSL* to 200, for instance. In such situation we do not accept segments that are longer than 200 meters but the segments are longer than 270 meters. It would be an exception if we have at least one segment in this resulting map!

There are similar failure examples for every wrongly set-up constant. Hence, it is not a good idea to randomly assign theses values; otherwise we should expect quite terrible results, but not due to system failure.

In these tests, the best values for the example's situation were:

- The error margin (*EMD*): 15 meters.
- The pattern path width (*PPW*): 10 meters.
- The maximal segment length (*MSL*): 350 meters.
- The minimal path length (*MPL*): 100 meters.
- The maximal tolerance length for path extremities (*TLE*): 100 meters.
- The *approximation* factor for point equalities: -6 (or 0,000.001 meters).
- The minimal significant variance when eliminating vertices (*MSV*): 5 meters.
- The distance for path similarities: $PSD = 2*EMD + PPW$.
- The minimal length for considering an average length (*MAFL*): $\sqrt{2} * PSD$.
- The minimal angle for not considering *immediate returns* (*MAIR*): $8\pi/18$ radians or 80º.
- The reference latitude angle (latitude of Lyngby): 0.973311946195504 radians.
- The Earth polar radius: 6356752.3142 meters.
- The Earth equatorial radius: 6378137.0 meters.

Using these constant values we achieved the map of the figure 91 as result of all processes of the pathfinder system. The conflictive map area enclosed in the red circle can be analyzed in the next figure 92. We can observe that it is much more concise than it was before. All the similar paths and fragments were averaged, thus we don't have shuffled lines anymore.

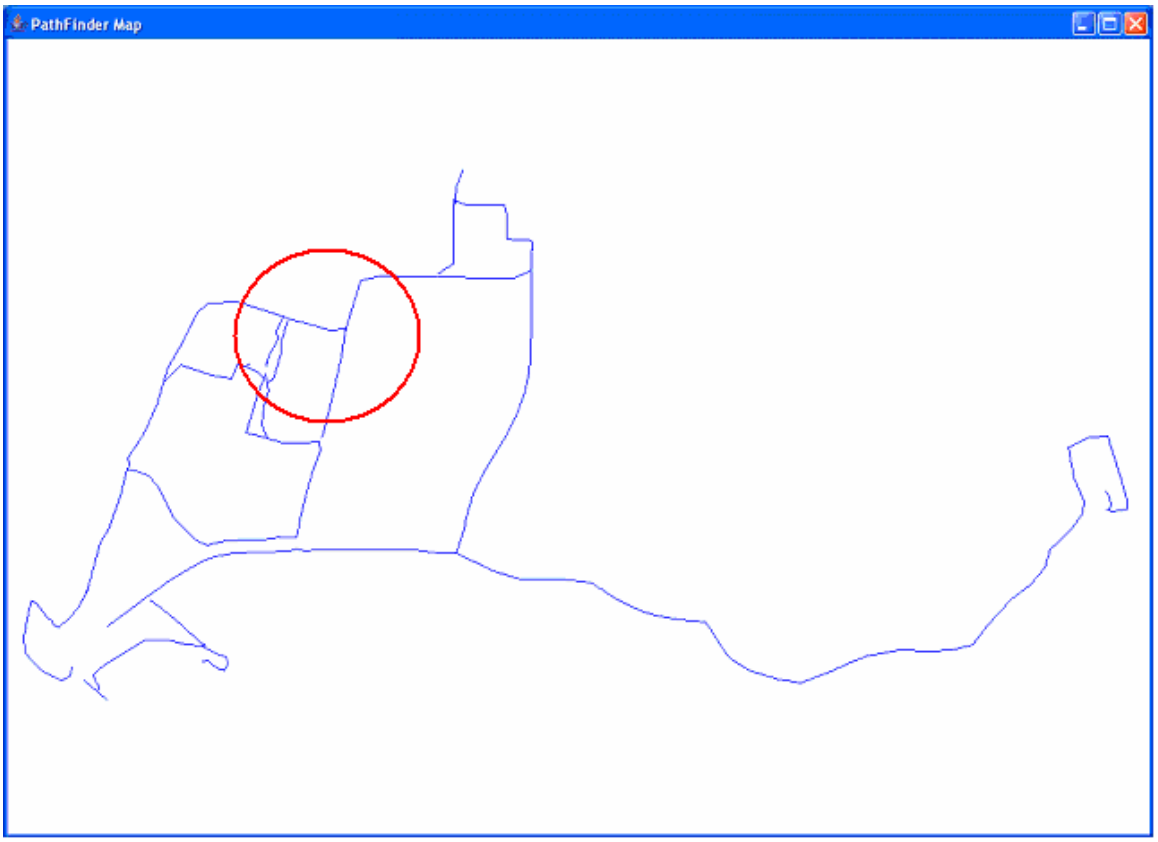

Figure 91: The complete resulting map. The red circle encloses part of the map illustrated in the next figure.

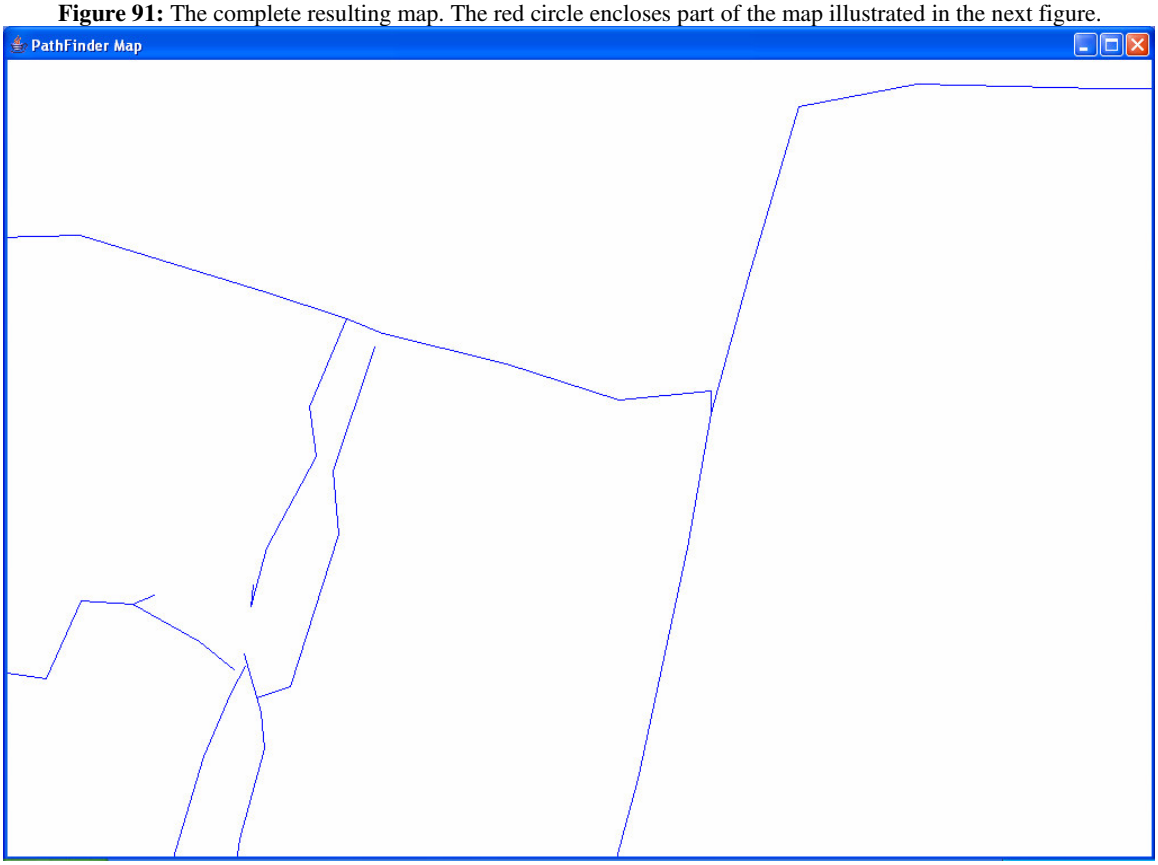

Figure 92: A zoom in one of the most conflictive road regions of the resulting map.

However, there are some failures that we can observe, like some little distortions, not linked path unions, and loss of very short path fragments. Anyway, these little failures will be explained later and they were not unexpected. We must remember that the fourth process was not strongly elaborated and the fifth process was not implemented. But we still can notice the accomplishment of the well elaborated processes of this project.

5.1 Testing the *cleaning process*

We have already seen examples of maps after the *cleaning process* for paths built from input data. Even though we haven't seen it in a detailed way in the figures before, the desired results demonstrate a good initial cleaning.

Actually, the initial *cleaning process* has been tested and checked but we are just going to show results of the *simple cleaning* performed after paths have been averaged. We are going to use red small circles for representing the points (or path vertices) in the map from where GPS data originated, and small green circles for positions where some path starts.

Even though we didn't illustrate any example of an initial cleaning, we can observe in the figure 93 that the excess unnecessary path vertices in a map can cause conflicts that are hard to manage. This figure illustrates a resulting map where no *simple cleaning* happened after any averaging process. We got a map with excess of unnecessary points everywhere, especially in the most averaged paths.

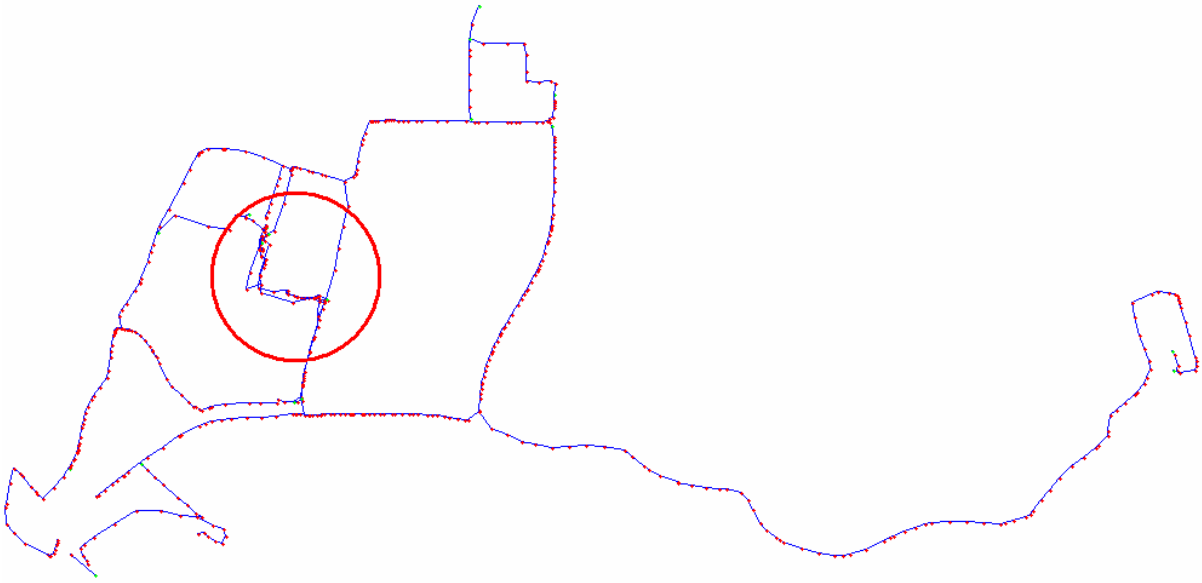

Figure 93: The resulting map after many averaging processes but without simple cleaning. The map region that is enclosed by the red circle shows conflicts caused by excess of vertices. However, this excess can be observed in any region of the map.

The more points a path has, the more conflictive it can be for averaging processes. The next figure 94 illustrates the same prior map but with performed simple cleanings after averaged processes. We can notice that it gets free of unnecessary points and the conflict

in the prior figure has disappeared. This means that cleaned paths have lower possibility of future conflicts. Hence, it shows the important role of the *cleaning process* for the pathfinder system. The figure 94 without excessive points also shows the achievement of the *simple cleaning* method.

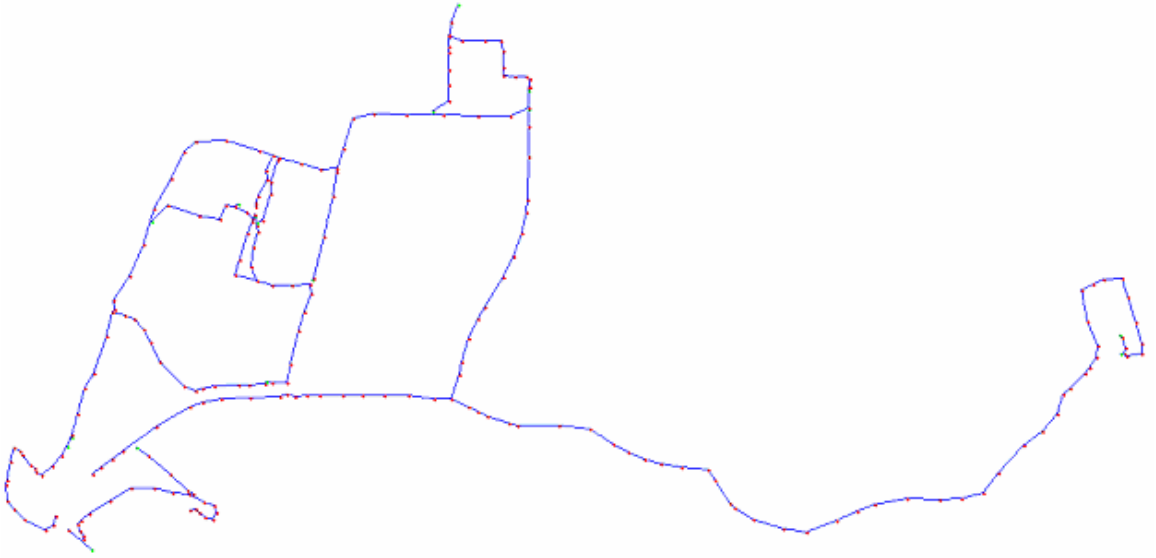

Figure 94: The resulting map after many averaging processes and simple cleanings.

5.2 Testing the *similarity detection process*

The great job performed by the *similarity detection process* was analyzed by checking the constructed map graph. In this test we have 44 paths generated from the original input data of the 45 cleaned GPS data files. Therefore, we have 44 original nodes representing each constructed path.

We obtained 21 nodes without any link, but also other seven linked sub-graphs. Five of the seven sub-graphs are complete cliques and two are not. Actually, these sub-graphs that are not cliques demonstrated the good result of the similarity detection.

The figure 95 illustrates one of these two sub-graphs and the paths that its nodes represent. The paths are not completely included to this figure because one set of path extremities does not have much variance amongst them, but other set of extremities does have considerable differences and they have been illustrated.

The same identification path numbers used in the program are used in this example. Some extremity differences make paths not similar but other extremities differences are tolerable. This demonstrates that the extremity tolerance does work.

Perhaps this example may seem to be wrong, but if we analyze it deeply we will notice that it is actually a success. We must remember that what is important here is not the straight distance between path endpoints but how long is the path extremity (of each path individually) that must be tolerated for making endpoints similar.

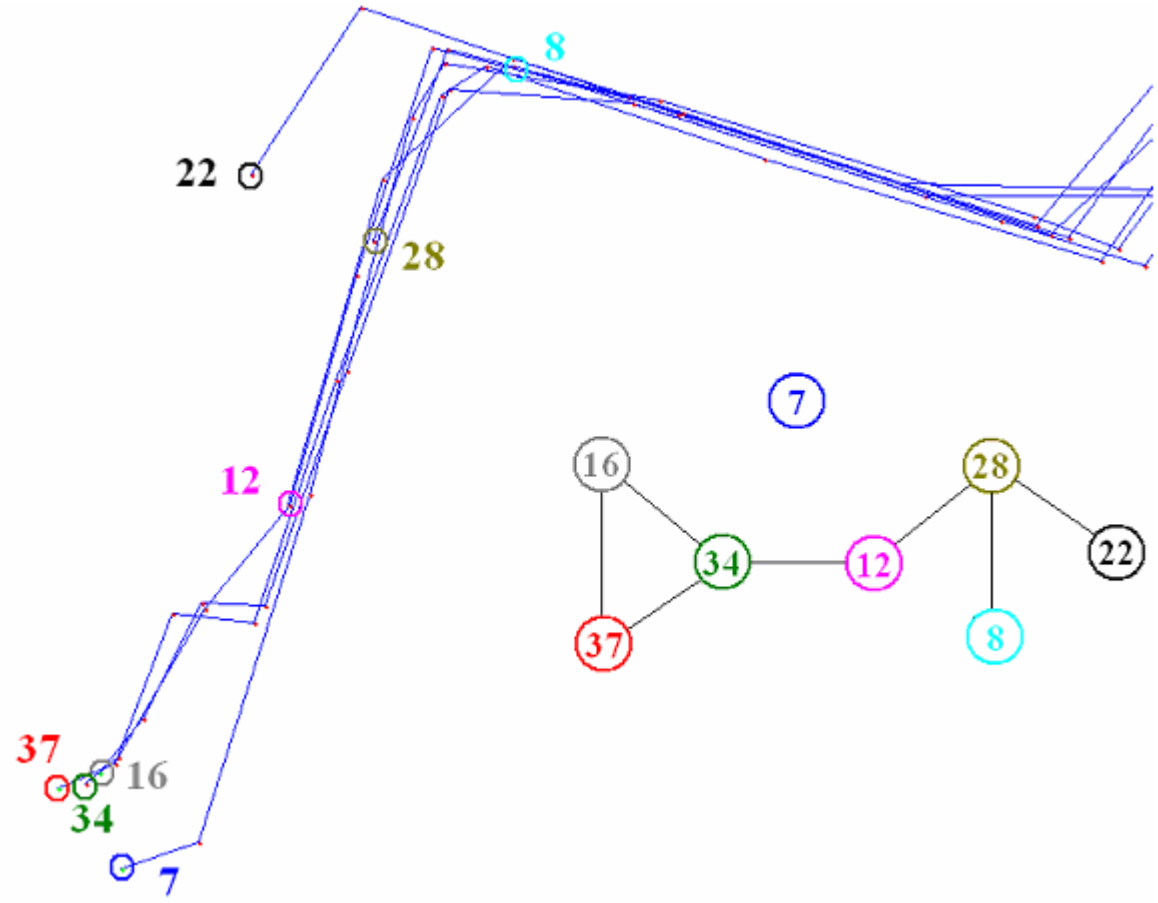

Figure 95: A sub-graph that is not a clique and a non-linked node. It is generated from the paths whose extremities are illustrated. The other path extremities are not included here, but the illustrated extremities are the rulers for the construction of these graph links. Little numbered circles on the paths represent the point where they start/end.

On the other hand, the average of some sub-graphs as this one, that is not a clique, might be the cause of little conflicts. However, the problem is neither the *similarity detection process* nor the *map graph* formed by it. The question is how to manage graphs that are not cliques once their management is a NP complete problem.

Another consequence of the *similarity detection process* is some losses of short map fragments due to the elimination of tolerable extremities when preparing *similar paths* for the *path averaging process*. Some fragment losses can be noticed in the figure 92. However, these consequences were expected.

5.3 Testing the *path averaging process*

This is the most elaborated process, thus we expected more from it than from any other process, and it didn't disappoint us. This process offered great results. No expected or unexpected failures happened due to this process. Only one pair of similar paths from the 44 original ones could not be averaged normally, and one of the original paths had to be returned as result. Anyway, further averages using this resulted path were possible and the averaging process was totally accomplished.

Of course we needed the help of the *simple cleaning* procedure for discarding some unnecessary points from each path resulting from an averaging, but it was easily possible for any resulting path.

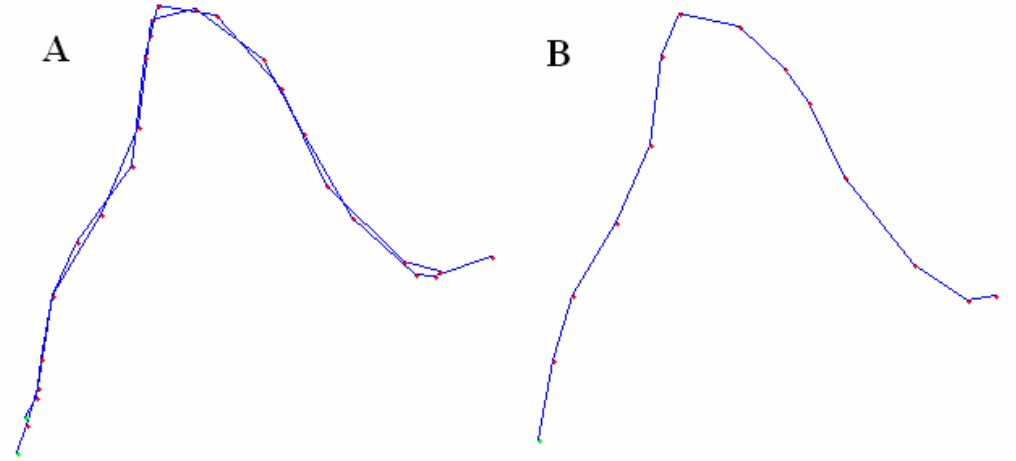

Figure 96: The example A is a pair of similar original paths. The example B is their resulting average after a *simple cleaning*.

Of course that the next *fragment averaging process* also shows the complete success of the *path averaging process* because fragments are considered as paths for being averaged. The figure 97 illustrates part of a map during the transition from this original map to a map without similar paths, and then to a map without similar fragments.

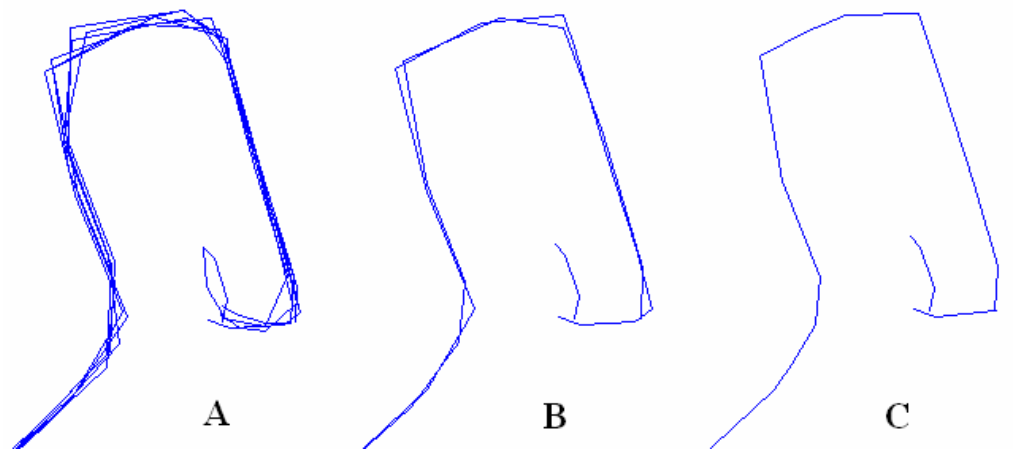

Figure 97: The example A illustrates part of the map before the *path averaging process*. The example B shows the same part of the map after the *path averaging process*. The example C illustrates this part of the map again, but after the *fragment averaging process*

5.4 Testing the *fragment averaging process*

Even though the *fragment averaging process* was not very elaborated, some good results can be observed from it, like the examples of figure 98. However, it is responsible of many unwished situations like, for instance, the little distortions observed in the figure 92, especially some distortions near to path unions.

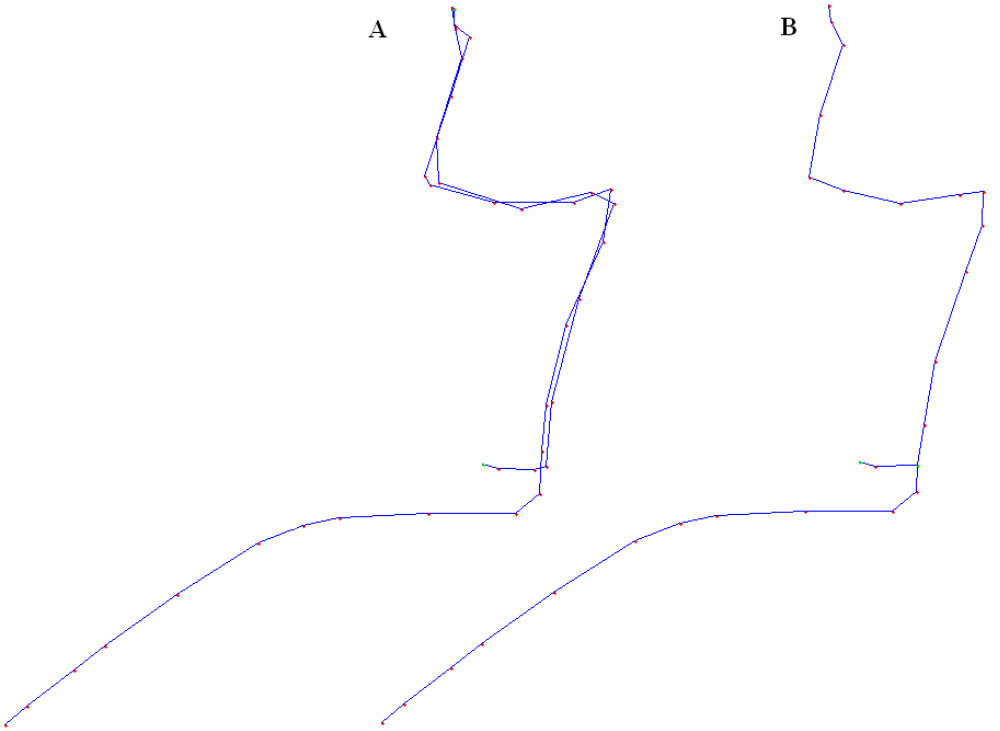

Figure 98: Three paths in the example B resulted from two paths in the example A. The original paths are not similar but they have a long fragment similarity. This similarity was well averaged and the unique link of these three resulting paths was a very good result.

These problems that happened many times near path unions (that are nor observed in the prior figure) are the evidence of that we need a *stitching process* in charge of solving link problems for path unions.

The *fragment averaging process* tried to supply the *stitching process* but it didn't achieve such process satisfactorily. Sometimes the *fragment averaging process* was able to solve little link problems, but many times it could not. Actually, sometimes it even made the problem worse. Therefore, it is suggested to elaborate a well analyzed *stitching process*.

Figure 99 demonstrates that we cannot blame the *cleaning process*, the *similarity detection process* nor the *path averaging process* for the unwished details in the last resulting map of this test.

In this figure, only the fragment averaging test (and the stitching process, of course) hasn't been performed yet. We can observe that, even if we still have many similar fragments to be averaged, all similar paths were averaged and their averages are quite accurate and clean. This figure does not show path distortions or bad path unions.

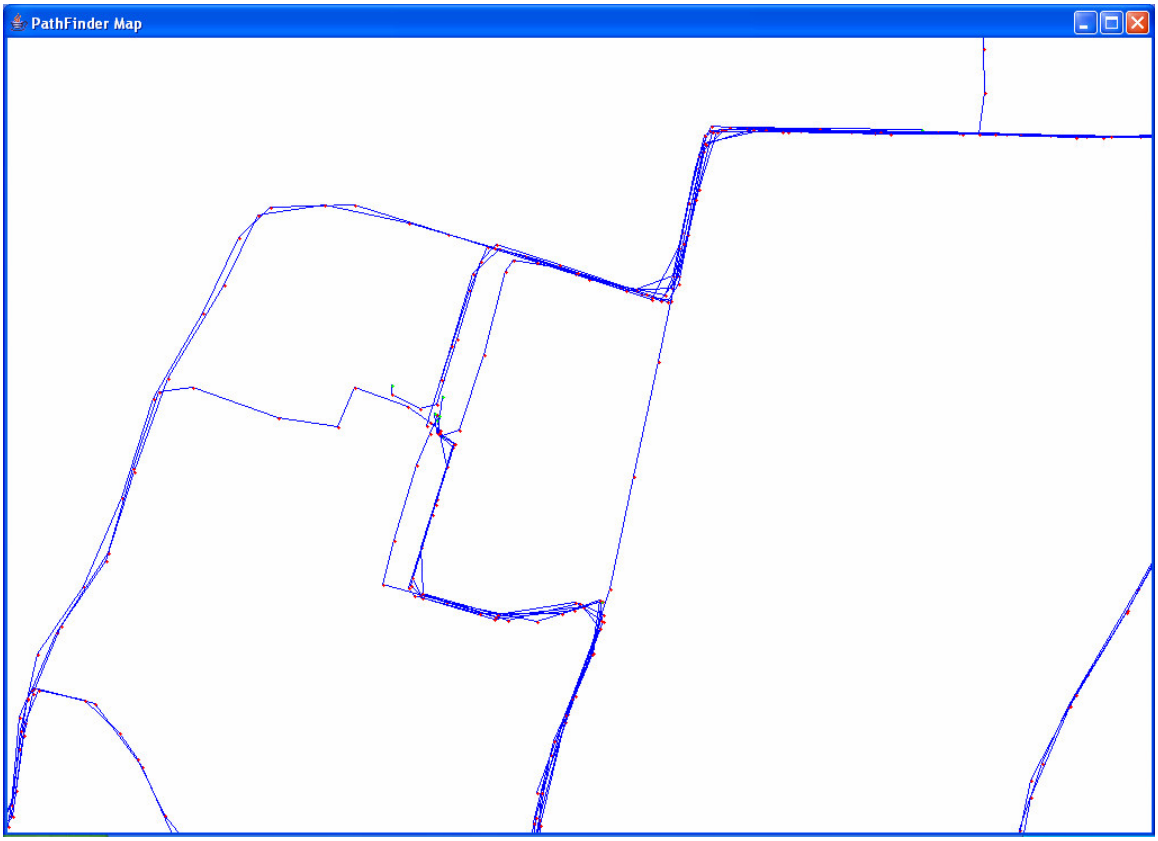

Figure 99: The most conflictive region of this map shows quite accurate results after the *path averaging process*, even though the map is not concise yet and a *fragment averaging process* is necessary.

Therefore, this test concludes that the more elaborated a process is the more successful it is. Every well-elaborated process reaches its aim. The process of fragment averaging, which is not very elaborated, needs improvement. The non-elaborated process of stitching is necessary for obtaining a map without any unwished detail.

6 CONCLUSION

The first decision in this project was to divide the performance of the studied pathfinder system in five processes: 1) cleaning process, 2) similarity detection process, 3) path averaging process, 4) fragment averaging process, and 5) stitching process. The obtained results of this project matches perfectly with the results expected for each process.

The three first processes, for eliminating unnecessary data from the map, finding similar paths on it and replacing these paths by single substitutes, were focused in this project because the system cannot have acceptable results without these being well developed. Theoretical solutions were elaborated. I've chosen some of these solutions and built algorithms based on them. Such algorithms were implemented and their results proved the expected success of the chosen theories.

The fourth process basically does the same that the two prior processes, but for path fragments instead of entire paths. This process was also discussed, but not deeply analyzed, because even though it has its particular problems, it is basically a combination of other processes.

The fifth process, for linking path unions, was not a strong issue in the project scope and was therefore not implemented. Moreover, algorithm implementations were not a project requirement, but a way for testing the result of chosen theoretical solutions.

Even though quite interesting solutions where created, the algorithms developed in this project can be optimized and the last process of stitching must also be developed for having a total successful result. The system optimization was not a requirement for this project either.

GPS systems are exposed to several external sources which introduce errors into a GPS position. Hence, any system interpreting GPS data has to be prepared for dealing with errors and trying to increase the accuracy of its data if possible. Data averaging is a good suggestion for minimizing errors from a group of similar data.

In more critical situations, the obtained input GPS data can be more accurate than the used for this project. For instance, it is possible to receive satellite signals in shorter time intervals and more satellites can be used for reducing inaccuracy.

Whereas in other situations, like the one offered for the project tests, some failures cannot be avoided, due to the lack of accuracy in the input data. Hence, some systems are made for dealing with possible failures and trying to minimize them instead of solving them. Sometimes it is a hard job because these assumed failures can cause hundreds of extra problems to deal with. Sometimes it is more efficient to discard troublesome data than trying to deal with it.

It is practically impossible to guess every problem that could arise when we are working with geometrical figures with no limit of vertices. I don't recommend the attempt of solving each of their possible problems. There is an infinity of possible problems in this area and some of them are very rare. Not all of these problems can be previously imagined and each time a problem is being solved, other problems may arise.

It is recommended to analyze the most frequent and the most important problems, and these ones should have well-elaborated solving algorithms. The possible but rare problems should not be priorities, as well as the not very significant ones. However, the system must be prepared for dealing with any possible exception. This means not to solve these exceptional situations, but to minimize considerably the failure that they cause or to discard their origins if this elimination will not represent a significant loss of information. For instance, it is recommended to discard one of two paths that could not be averaged instead of trying to solve every possible problem that could happen in the average of two similar paths.

I've noticed during this project that sometimes there are excellent theoretical concepts that are very strong, but they are not computable due to their complexity. The lack of functionality for these concepts is noticed only when trying to implement them. Many times strong mathematical concepts were developed for this project but they had to be modified for making them possible to be codified.

On the other hand, there are casual empiric concepts that have an enormous functionality. Therefore, only the implementation attempt of some theoretical concepts allows us to measure for sure their functionality. A constant challenge in the computational geometry is to find the balance between accuracy and efficiency.

Some good ideas have to be discarded due to the hard-problem algorithms that they require for their implementations. Sometimes, easy solutions can be found but, the simpler a good algorithm is, the more time it tends to take for dealing with thousands of data. Therefore, processing time is a critical issue for computational geometry.

Optimal algorithms are constantly required in the computational geometry due to the huge number of data that use to be necessary to deal with. Many procedures of this project are able to be optimized and some suggestions for their optimizations were in the chapters of this report.

7 REFERENCES

[CG1] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, Mark Overmars, *Computational Geometry: Algorithms and Applications* 2nd Edition, 2000

[CG2] *Computational Geometry* http://www-cgrl.cs.mcgill.ca/~godfried/teaching/cg-web.html Computational Geometry Laboratory, McGill University, Canada.

[ER] *Earth Radius* http://en.wikipedia.org/wiki/Earth_radius Wikipedia.

[FIST] *Fast Industrial-Strength Triangulation of Polygons [July 2004]* http://www.cosy.sbg.ac.at/~held/projects/triang/triang.html Universität Salzburg, Austria.

[FPT] Nikos Drakos, *Fast Polygon Triangulation based on Seidel's Algorithm [January 1994]* http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html Computer Based Learning Unit, University of Leeds.

[GCD] Rod Dinkins, *Formulae for Calculating Great Circle Heading and Distance Information [April 2003]* http://www.ac6v.com/greatcircle.htm

[GoS] John C. Polking, *The Geometry of the Sphere [January 2000]* http://math.rice.edu/~pcmi/sphere/ Rice University, Houston, Texas, USA.

[IA] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms,* MIT Press, *1996*

[GM] Sarah Price, *Graph Matching [2000]*
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARBLE/high/matching/graph.htm
Institute for Computer Based Learning, Heriot-Watt University, Scotland

[Map] *How to reap a map*
http://encarta.msn.com/encyclopedia_761577953/Map.html#631505884
Encarta World Atlas 2001.

[PO] Paul Bourke, *Determining whether or not a polygon (2D) has its vertices ordered clockwise or counterclockwise [March 1998]*
http://astronomy.swin.edu.au/~pbourke/geometry/clockwise/
FIC Faculty, Swinburne University of Technology, Australia

[ST] *Spherical Trigonometry [1999]*
http://www.hps.cam.ac.uk/starry/sphertrig.html
University of Cambridge, United Kingdom

[STC] James Q. Jacobs, *Spherical Triangle Calculator [June 2000]*
http://www.jqjacobs.net/astro/arc_form.html

APPENDIX A – THE *CONST* CLASS

```
package path;

/*****************************************************************************
* This Const class contains the constant values used in this package.
*****************************************************************************/

public class Const
{
 /**
 * Defines how many digits to be considered when determining if two numbers are
 * approximatelly the same. (the '-' signal means after the numerical '.')
 */
 static final double approximation = -6;

 /**
 *  the radius of the equatorial line of the Earth
 */
 static final double EarthEquatorialRadius = 6378137.0;

 /**
 * the radius of the meridians of the Earth
 */
 static final double EarthPolarRadius = 6356752.3142;

 /**
 * The latitude that is going to be used as reference, supposing that we are just
 * going to act (reletively) near this latitude line.
 */
 static final double ReferenceLatitudeAngle = 0.973311946195504; //(lyngby latitude = 55°46')

 /**
 * Error Margin Distance: Defines the maximal distance for considering
 * vertex fluctuations.
 */
 static final double EMD = 15;

 /**
 * Pattern Path Width: Defines an acceptable width for paths.
 */
 static final double PPW = 10;

 /**
 * Path Similarity Distance: defines how distant the points of two paths can be
 * for still considering the complete paths similar.
 */
 static final double PSD = 2 * EMD + PPW;

 /**
 * Maximal Segment Distance: defines how long a path segment can be for
 * including it in a path.
 */
 static final double MSL = 350;
```

```
 /**
 * Minimal Angle for Immediate Return: defines the minimal angle necessary
 * for considering a immediate return to happen.
 */
 static final double MAIR = 8 * Math.PI / 18;

 /**
 * Tolerance Length for Extremes: defines the maximal path length that can
 * be tolerated at the path extremes when detecting path similarities.
 */
 static final double TLE = 100;

 /**
 * Minimal Average Fragment Length: defines the minimal length of the average
 * fragment of similar path fragments for being taken in consideration.
 */
 static final double MAFL = Math.sqrt(2) * PSD;

 /**
 * Minimal Path Length: defines the minimal length of a path for being taken in
 * consideration.
 */
 static final double MPL = 100;

 /**
 * Minimal Significant Variance: defines the acceptable loss of accuracy (in meters)
 * during clenings of consecutive vertices (used during simple cleanings).
 */
 static final double MSV = 5;
}//class
```

APPENDIX B – THE *POINT* CLASS

```
package path;

import java.util.Vector;

/*********************************************************************************
* This Point class is limited to planar coordinates and Euclidean distances.
* <p>
* This class includes methods for setting points, examining sets of them and
* finding their distances.
*********************************************************************************/

public class Point
{
 /**
 * The x-coordinate of the point.
 */
 public double x;
 /**
 * The y-coordinate of the point.
 */
 public double y;

 /*********************************************************************************
 * Constructs a point with coordinates 0 (zero).
 *********************************************************************************/

 public Point()
 {
  this.x = 0;
  this.y = 0;
 }

 /*********************************************************************************
 * Constructs a point so that it has a given x and y coordinates.
 *
 * @param x  the x coordinate for the point
 * @param y  the y coordinate for the point
 *********************************************************************************/

 public Point(double x, double y)
 {
  this.x = x;
  this.y = y;
 }

 /*********************************************************************************
 * Constructs a point with the same coordinates from a given point.
 *
 * @param p  the given point
 *********************************************************************************/

 public Point(Point p)
```

```
{
  this.x = p.x;
  this.y = p.y;
}

/**************************************************************************
* Creates a planar point from the given latitude and longitude of a
* spherical/geographical position.&bnps; Latitudes and longitudes are going to
* be considered in radians.
*
* @param latitude the latitude of the spherical/geographical point
* @param longitude the longitude of the spherical/geographical point
**************************************************************************/

public void sphericalToPlanar(double latitude, double longitude)
{
  this.x = longitude * Const.EarthEquatorialRadius *  Math.cos(Const.ReferenceLatitudeAngle);
  this.y = latitude * Const.EarthPolarRadius;
}

/**************************************************************************
* Makes this point to be equal to the given one.
* <p>
* OBS: this method makes values equal, but not pointers
* @param p  a point
**************************************************************************/

public void equalize(Point p)
{
  this.x = p.x;
  this.y = p.y;
}

/**************************************************************************
* Finds the distance from this point to a given point.
* <p>
* It uses the Euclidean formula for planar distances.
* @param p  the given point
* @return the point-to-point distance
**************************************************************************/

public double distance(Point p)
{
  if(p == null) return 0;
  return Math.sqrt((this.x-p.x)*(this.x-p.x)+(this.y-p.y)*(this.y-p.y));
}

/**************************************************************************
* Finds the index in a given vector of points where this point happens first
* <p>
* Compares the coordinates of each point of the given vector with the coordinates
* of this point.
* <p>
* Returns -1 if the point is not found.
*
* @param v  the given vector
```

```
 * @return the index of the first ocurrence
 ***************************************************************************/

 public int indexIn(Vector v)
 {
  for(int i=0;i<v.size();i++)
    if(this.x == ((Point)v.get(i)).x && this.y == ((Point)v.get(i)).y)
      return i; //if the first ocurrence is found
  return -1; //no ocurrence of this point was found
 }

/***************************************************************************
 * Checks if this point is equal or approximatelly equal to a given point.
 * <p>
 * Returns true if they are equal, false otherwise.
 * <p>
 * A given boolean flag determines if the equality should be approximated (the
 * flag is true) or exact (the flag is false). If it should be approximated then
 * an predetermined approximation constant is going to be taken in account.
 *
 * @param p  the given point
 * @param approximated  the flag for approximation or exactitude
 * @return true or false
 ***************************************************************************/

 public boolean equals(Point p, boolean approximated)
 {
  if(approximated)
  {//approximated equality
    if(Math.abs(this.x-p.x) >= Math.pow(10,Const.approximation)) return false;
    if(Math.abs(this.y-p.y) >= Math.pow(10,Const.approximation)) return false;
    return true;
  }
  else
  {//exact equality
    if(this.x != p.x || this.y != p.y) return false;
    return true;
  }
 }

/***************************************************************************
 * Computes the weighted average for this point and a given point.
 * <p>
 * The distance between the points is divided by the sum of both weights and
 * multiplyed by each weight, obtaining two distances. These distances are
 * proportional to their respective weights and their sum is equal to the original
 * distance. Therefore a weight-proportional average point is obtained.
 * <p>
 * This method also works for normal averages giving equal weights for each point.
 *
 * @param p  the given point
 * @param thisWeight  the weight for this point
 * @param pWeight  the weight for the given point
 * @return the average point according to the given weights
 ***************************************************************************/
```

```
 public Point weightedAverage(Point p,int thisWeight,int pWeight)
 {
  Point resultingPoint = new Point(0,0);

  resultingPoint.x = this.x + (p.x - this.x) * pWeight / (thisWeight + pWeight);
  resultingPoint.y = this.y + (p.y - this.y) * pWeight / (thisWeight + pWeight);

  return resultingPoint;
 }

}//class
```

APPENDIX C – THE *SEGMENT* CLASS

```
package path;

/*******************************************************************************
* This <code>Segment</code> class is limited to planar coordinates and Euclidean distances. 
* It represents segments of straight lines defined by a startpoint and an endpoint.
* <p>
* This class includes methods for setting segments, mesuring them and examining
* relationships between segments and other geometrical objects like points, lines
* and other segments
*******************************************************************************/

public class Segment
{
 /**
  * startpoint of the segment.
  */
 public Point p1;
 /**
  * endpoint of the segment.
  */
  public Point p2;

 /*******************************************************************************
 * Constructs a segment with points (0,0).
 *******************************************************************************/

 public Segment()
 {
   this.p1 = new Point();
   this.p2 = new Point();
 }

 /*******************************************************************************
 * Constructs a segment so that it beggins at a given startpoint and ends
 * at a given endpoint.
 *
 * @param point1 the given segment startpoint
 * @param point2 the given segment endpoint
 *******************************************************************************/

 public Segment(Point point1, Point point2)
 {
   this.p1 = new Point(point1.x, point1.y);
   this.p2 = new Point(point2.x, point2.y);
 }

 /*******************************************************************************
 * Constructs a segment so that it beggins at a startpoint defined by the given
 * x and y coordinates and ends at an endpoint also defined by given coordinates.
 *
 * @param x1 the x coordinate for the segment startpoint
 * @param y1 the y coordinate for the segment startpoint
```

```
* @param x2 the x coordinate for the segment endpoint
* @param y2 the y coordinate for the segment endpoint
*****************************************************************************/

public Segment(double x1, double y1, double x2, double y2)
{
  this.p1 = new Point(x1,y1);
  this.p2 = new Point(x2,y2);
}

/*****************************************************************************
* Constructs a segment with the same values of the given segment.
*
* @param s  the given segment
*****************************************************************************/

public Segment(Segment s)
{
  this.p1 = new Point();
  this.p1.equalize(s.p1);
  this.p2 = new Point();
  this.p2.equalize(s.p2);
}

/*****************************************************************************
* Sets this segment equalizing its start and endpoint with the same values
* of the start and endpoint of a given segment.
* <p>
* OBS: this method makes values equal, but not pointers
*
* @param s the given segment for setting this one like
*****************************************************************************/

public void equalize(Segment s)
{
  this.p1.equalize(s.p1);
  this.p2.equalize(s.p2);
}

/*****************************************************************************
* Sets this segment equalizing its start and endpoint with the same values
* of two given points.
* <p>
* OBS: this method makes values equal, but not pointers
*
* @param p1 the given point to set the segment startpoint like
* @param p2 the given point to set the segment endpoint like
*****************************************************************************/

public void equalize(Point p1, Point p2)
{
  this.p1.equalize(p1);
  this.p2.equalize(p2);
}

/*****************************************************************************
```

```
 * Checks if the given doubles (segment start/endpoints coordinates) are
 * approximately equal.
 * <p>
 * Returns true if they are approximately equal, false otherwise.
 * <p>
 * A predetermined approximation constant is taken in account in this method.
 *
 * @param d1  the first given double
 * @param d2  the second given double
 * @return true or false
 ****************************************************************************/

 private boolean equals(double d1,double d2)
 {
    if(Math.abs(d1-d2) < Math.pow(10,Const.approximation)) return true;
    return false;
 }

/****************************************************************************
 * Returns the length of this segment.
 * <p>
 * It uses the method <code>distance(Point p)</code> from the class {@link Point}
 * of this package for calculating the distance between the endpoint and the
 * startpoint of this segment.
 *
 * @return the segment length
 ****************************************************************************/

 public double length()
 {
  return this.p1.distance(this.p2);
 }

/****************************************************************************
 * Finds the minimal distance from this segment to a given point.
 * <p>
 * It finds the equation for the line of this segment and the equation
 * of a transversal line that passes on the given point. The distance between
 * the intersection point of this computed lines and the given point is the
 * minimal distance between the segment and the given point whether the
 * intersection point belongs to the segment.
 * <p>
 * If the intersection point is not part of the segment, the start or endpoint
 * of the segment has the closest distance to the given point.
 *
 * Cares are needed for the case of infinite slope of vertical lines but these
 * lines are even easier to work with.
 *
 * @param point  the given point
 * @return the minimal distance
 ****************************************************************************/

 public double distance(Point point)
 {
  //equation for the segment line: y = a x + b (where a=slope, b=displacement)
  double slope, displacement;
```

```
  //equation for the segment transversal line Y = -1/a X + c (where c=transversalDisplacement)
  double transversalDisplacement;
  //coordinates for the intersection point of the found transversal lines
  double intersectionX, intersectionY;

  if (equals(this.p1.x,this.p2.x))
  {//the segment line is vertical (slope = infinite)
   if (equals(this.p1.y,this.p2.y)) return point.distance(this.p1);
   intersectionX = this.p1.x;
   intersectionY = point.y;
  }
  else
  {//the segment line is not vertical
   slope = (this.p1.y - this.p2.y) / (this.p1.x - this.p2.x);
   displacement = this.p1.y - slope * this.p1.x;
   if (equals(slope,0)) intersectionX = point.x;  //the transversal line is vertical
   else
   {//the transversal line is not vertical
    transversalDisplacement = point.y + point.x / slope;
    intersectionX = (transversalDisplacement - displacement) / (slope + 1 / slope);
   }
   intersectionY = slope * intersectionX + displacement;
  }
  // now, the intersection point is already defined
  if(((intersectionX >= this.p1.x && intersectionX <= this.p2.x) ||
      (intersectionX >= this.p2.x && intersectionX <= this.p1.x)) &&
     ((intersectionY >= this.p1.y && intersectionY <= this.p2.y) ||
      (intersectionY >= this.p2.y && intersectionY <= this.p1.y)))
   return Math.sqrt(Math.pow(point.x - intersectionX, 2)
       + Math.pow(point.y - intersectionY, 2));//intersection in the given segment
  else
  {//the intersection point isn't in the given segment
   double distance1 = point.distance(this.p1);
   double distance2 = point.distance(this.p2);
   if (distance1 <= distance2) return distance1;//the nearest point in the segment is its point p1
   else return distance2;//the nearest point in the segment is its point p2
  }
 }

/**************************************************************************
 * Finds the minimal distance from this segment to another given segment.
 * <p>
 * In case the segments intersect, their minimal distance is zero. Otherwise the
 * minimal distance between two segments is the distance from some start or
 * endpoint to the other segment. So, the only 4 possible distances are
 * calculated and compared. The lowest distance is returned.
 * <p>
 * The method <code>distance(Point p)</code> of this class is used for that.
 *
 * @param s  the given segment
 * @return the minimal distance
 **************************************************************************/

 public double distance(Segment sgmt)
 {
  if(this.checkIntersection(sgmt)) return 0;
```

```
  //the distance from the startpoint from this segment to the other segment
  double distance1 = this.distance(sgmt.p1);
  //the distance from the endpoint from this segment to the other segment
  double distance2 = this.distance(sgmt.p2);
  //the distance from the startpoint from the other segment to this segment
  double distance3 = sgmt.distance(this.p1);
  //the distance from the endpoint from the other segment to this segment
  double distance4 = sgmt.distance(this.p2);

  //returns the lowest of the 4 found distances
  if(distance1 <= distance2 && distance1 <= distance3 && distance1 <= distance4)
     return distance1;
  if(distance2 <= distance3 && distance2 <= distance4) return distance2;
  if(distance3 <= distance4) return distance3;
  return distance4;
 }

 /*****************************************************************************
 * Check if there is intersection between this segment and a given segment,
 * returning true if there is intersection, false otherwise.
 * <p>
 * It uses the vector cross product for this aim. Vectors cross defines the side
 * a vector turns (right or left) to reach another vector. Remember that if two
 * segments are intersecting, the extremes of a vector should be at different
 * sides of the other vector line.
 * <p>
 * First, a quick possibility of intersection is checked using the bounding box
 * test.
 *
 * @param sgmt  the given segment
 * @return true or false
 *****************************************************************************/
 //Introduction to Algorithms, Thomas Cormen, chapter 35 page 889.
 public boolean checkIntersection(Segment sgmt)
 {
  //quick rejection...the bounding box test
  if(Math.max(this.p1.x,this.p2.x) >= Math.min(sgmt.p1.x,sgmt.p2.x) &&
     Math.max(sgmt.p1.x,sgmt.p2.x) >= Math.min(this.p1.x,this.p2.x) &&
     Math.max(this.p1.y,this.p2.y) >= Math.min(sgmt.p1.y,sgmt.p2.y) &&
     Math.max(sgmt.p1.y,sgmt.p2.y) >= Math.min(this.p1.y,this.p2.y))
    //the vector must turn to different sides (or not turn) if there is intersection
    if(((sgmt.p1.x-this.p1.x)*(this.p2.y-this.p1.y) -
        (this.p2.x-this.p1.x)*(sgmt.p1.y-this.p1.y)) *
       ((sgmt.p2.x-this.p1.x)*(this.p2.y-this.p1.y) -
        (this.p2.x-this.p1.x)*(sgmt.p2.y-this.p1.y)) <= 0 &&
      ((this.p1.x-sgmt.p1.x)*(sgmt.p2.y-sgmt.p1.y) -
        (sgmt.p2.x-sgmt.p1.x)*(this.p1.y-sgmt.p1.y)) *
       ((this.p2.x-sgmt.p1.x)*(sgmt.p2.y-sgmt.p1.y) -
        (sgmt.p2.x-sgmt.p1.x)*(this.p2.y-sgmt.p1.y)) <= 0) return true;
  return false;
 }

/*****************************************************************************
 * Finds the intersecting part of this segment and a given segment.&amp;nbsp; It
 * returns a segment containing such intersecting part.
```

```
* <p>
* If there is no intersection it returns a segment of null endpoints. If a
* unique point is the intersection part of the segments then it returns a
* segment with equal start and endpoint for representing a point of intersection.
* <p>
* The intersecting part might be a segment whether the segments are collinear,
* so the collinearity is checked and the method collinearIntersection(Segment)
* of this class is called whether they are collinear.
* <p>
* Actually, this method only finds the most common cases of point intersections
* and let the rare job of finding segment intersections to the method
* collinearIntersection(Segment) whether it is needed. Remember that parallell
* segments that are not collinear do not have intersection.
* <p>
* It finds point intersections by finding the ecuation y = a x + b of both
* segment lines and calculating the point that matches to both equations.
* <p>
* Cares are needed for the case of infinite slope of vertical lines but these
* lines are even easier to work with.
*
* @param sgmt  the given segment
* @return the segment containing the entire intersection
*****************************************************************************/

public Segment intersection(Segment sgmt)
{
 // return a null segment if there is no intersection
 if(!this.checkIntersection(sgmt)) return (new Segment(null,null));

 double intersectionX, intersectionY; //the intersection point coordinates
 if (equals(this.p1.x,this.p2.x))
 {//this segment is vertical
  //if the given segment is also vertical, hence the segements are collinear
  if(equals(sgmt.p1.x,sgmt.p2.x)) return this.collinearIntersection(sgmt);
  else
  {//the given segment is not vertical
   double sgmtSlope = (sgmt.p1.y - sgmt.p2.y) / (sgmt.p1.x - sgmt.p2.x);
   double sgmtDisplacement = sgmt.p1.y - sgmtSlope * sgmt.p1.x;
   intersectionX = this.p1.x;
   intersectionY = sgmtSlope * intersectionX + sgmtDisplacement;
  }
 }
 else
 {//this segment is not vertical
  double thisSlope = (this.p1.y - this.p2.y) / (this.p1.x - this.p2.x);
  double thisDisplacement = this.p1.y - thisSlope * this.p1.x;
  if(equals(sgmt.p1.x,sgmt.p2.x))
  {// the given segment is vertical
   intersectionX = sgmt.p1.x;
   intersectionY = thisSlope * intersectionX + thisDisplacement;
  }
  else
  {//no segment is vertical
   double sgmtSlope = (sgmt.p1.y - sgmt.p2.y) / (sgmt.p1.x - sgmt.p2.x);
   double sgmtDisplacement = sgmt.p1.y - sgmtSlope * sgmt.p1.x;
   //if the segments are parallel (equal slopes), so they are collinear
```

```
      if(equals(thisSlope,sgmtSlope)) return this.collinearIntersection(sgmt);
      intersectionX = (sgmtDisplacement - thisDisplacement) / (thisSlope - sgmtSlope);
      intersectionY = thisSlope * intersectionX + thisDisplacement;
    }
  } return (new Segment(intersectionX, intersectionY, intersectionX, intersectionY));
}

/*****************************************************************************
* Checks if this segment and a given collinear segment are intersecting. 
* Returns the segment containing the intersection.
* <p>
* It is checked if some part of both collinear segments matches,
* because even beeing collinear they might match nowhere having no
* intersection. A null segment is returned when no intersection is found.
* <p>
* In case there exist intersection, two of the four extremepoints (startpoint
* and endpoint) from these two segments are going to bound the intersecting
* fragment where they match. These two extremepoints are going to be found
* to define the intersection segment to be returned.
* <p>
* The x coordinates will be compared to check if a extreme point is in the
* other collinear segment. Except if the collinear segments are vertical and
* their x coordinates do not vary. In this case the y coordinates will be used.
*
* @param sgmt  the given segment
* @return the segment containing the entire intersection
*****************************************************************************/

private Segment collinearIntersection(Segment sgmt)
{
  if (equals(this.p1.x,this.p2.x))
  {// the collinear segments are vertical
    //if this segment is completely in the given segment sgmt
    if(((this.p1.y >= sgmt.p1.y && this.p1.y <= sgmt.p2.y) ||
        (this.p1.y <= sgmt.p1.y && this.p1.y >= sgmt.p2.y)) &&
       ((this.p2.y >= sgmt.p1.y && this.p2.y <= sgmt.p2.y) ||
        (this.p2.y <= sgmt.p1.y && this.p2.y >= sgmt.p2.y))) return(new Segment(this));
    //if the given segment s is completely in this segment
    if(((sgmt.p1.y >= this.p1.y && sgmt.p1.y <= this.p2.y) ||
        (sgmt.p1.y <= this.p1.y && sgmt.p1.y >= this.p2.y)) &&
       ((sgmt.p2.y >= this.p1.y && sgmt.p2.y <= this.p2.y) ||
        (sgmt.p2.y <= this.p1.y && sgmt.p2.y >= this.p2.y))) return(new Segment(sgmt));
    if(this.p1.y > sgmt.p1.y && this.p1.y > sgmt.p2.y)
    {//if the startpoint from this segment is above the entire given segment sgmt
     //(and the endpoint from this segment is into sgmt)
      if(sgmt.p1.y < this.p2.y) return(new Segment(this.p2, sgmt.p2));
      else return(new Segment(this.p2, sgmt.p1));
    }
    if(this.p2.y > sgmt.p1.y && this.p2.y > sgmt.p2.y)
    {//if the endpoint from this segment is above the entire given segment sgmt
     //(and the startpoint from this segment is into sgmt)
      if(sgmt.p1.y < this.p1.y) return(new Segment(this.p1, sgmt.p2));
      else return(new Segment(this.p1, sgmt.p1));
    }
    if(this.p1.y < sgmt.p1.y && this.p1.y < sgmt.p2.y)
    {//if the startpoint from this segment is below the entire given segment sgmt
```

```
    //(and the endpoint from this segment is into sgmt)
     if(sgmt.p1.y > this.p2.y) return(new Segment(this.p2, sgmt.p2));
     else return(new Segment(this.p2, sgmt.p1));
    }
    if(this.p2.y < sgmt.p1.y && this.p2.y < sgmt.p2.y)
    {//if the endpoint from this segment is below the entire given segment sgmt
     //(and the startpoint from this segment is into sgmt)
     if(sgmt.p1.y > this.p1.y) return(new Segment(this.p1, sgmt.p2));
     else return(new Segment(this.p1, sgmt.p1));
    }
   }
   else
   {//the collinear segments are not vertical
    //if this segment is completely in the given segment sgmt
    if(((this.p1.x >= sgmt.p1.x && this.p1.x <= sgmt.p2.x) ||
        (this.p1.x <= sgmt.p1.x && this.p1.x >= sgmt.p2.x)) &&
       ((this.p2.x >= sgmt.p1.x && this.p2.x <= sgmt.p2.x) ||
        (this.p2.x <= sgmt.p1.x && this.p2.x >= sgmt.p2.x))) return(new Segment(this));
    //if the given segment sgmt is completely in this segment
    if(((sgmt.p1.x >= this.p1.x && sgmt.p1.x <= this.p2.x) ||
        (sgmt.p1.x <= this.p1.x && sgmt.p1.x >= this.p2.x)) &&
       ((sgmt.p2.x >= this.p1.x && sgmt.p2.x <= this.p2.x) ||
        (sgmt.p2.x <= this.p1.x && sgmt.p2.x >= this.p2.x))) return(new Segment(sgmt));
    if(this.p1.x > sgmt.p1.x && this.p1.x > sgmt.p2.x)
    {//if the startpoint from this segment is to the right of the entire given
     //segment sgmt (and the endpoint from this segment is into sgmt)
     if(sgmt.p1.x < this.p2.x) return(new Segment(this.p2, sgmt.p2));
     else return(new Segment(this.p2, sgmt.p1));
    }
    if(this.p2.x > sgmt.p1.x && this.p2.x > sgmt.p2.x)
    {//if the endpoint from this segment is to the right of the entire given
     //segment sgmt (and the startpoint from this segment is into sgmt)
     if(sgmt.p1.x < this.p1.x) return(new Segment(this.p1, sgmt.p2));
     else return(new Segment(this.p1, sgmt.p1));
    }
    if(this.p1.x < sgmt.p1.x && this.p1.x < sgmt.p2.x)
    {//if the startpoint from this segment is to the left of the entire given
     //segment sgmt (and the endpoint from this segmentis into sgmt)
     if(sgmt.p1.x > this.p2.x) return(new Segment(this.p2, sgmt.p2));
     else return(new Segment(this.p2, sgmt.p1));
    }
    if(this.p2.x < sgmt.p1.x && this.p2.x < sgmt.p2.x)
    {//if the endpoint from this segment is to the left of the entire given
     //segment sgmt (and the startpoint from this segment is into sgmt)
     if(sgmt.p1.x > this.p1.x) return(new Segment(this.p1, sgmt.p2));
     else return(new Segment(this.p1, sgmt.p1));
    }
   }
   return null; //something wrong happened
  }

  /**************************************************************************
  * Finds the minimal angle (in radians) formed by this segment and a given segment,
  * as if the startpoint of both segments matches.
  * <p>
  * First it finds the independent angle of each segments (respect to a cartesian
```

```
 * system where the origin point is at the segment startpoint).
 * <p>
 * The found independent angles are combined to calculate the angle between them.
 * Two segments form two angles between them and the lowest one is returned.
 *
 * @param sgmt  the given segment
 * @return the minimal angle
 **************************************************************************/

public double angle(Segment sgmt)
{
  double slope, thisAngle, otherAngle, finalAngle;

  //computing the independent angle of this segment
  if(equals(this.p1.x,this.p2.x))
  {//this segment is vertical, hence its angle is a right angle
    if(equals(this.p1.y,this.p2.y)) return 0; //this segment is a point
    if(this.p2.y > this.p1.y) thisAngle = Math.PI / 2; //this angle is positive
    else thisAngle = Math.PI / -2; //this angle is negative
  }
  else
  {//this segment is not vertical
    //get the value of the slope of this segment
    slope = (this.p1.y - this.p2.y) / (this.p1.x - this.p2.x);
    thisAngle = Math.atan(slope);
    //this angle might be wrong yet because it might be > 90 or < -90 degrees)
  }
  //finding out if the angle of this segment is not between -90 and 90 degrees.
  //If it is not, correct thisAngle
  if(this.p2.x < this.p1.x)//180 >= thisAngle > 90 degrees or -180 < thisAngle < -90 degrees
    if(this.p2.y >= this.p1.y) thisAngle = thisAngle + Math.PI;//180 >= thisAngle > 90 degrees
    else thisAngle = thisAngle - Math.PI; // -180 < thisAngle < -90 degrees

  //computing the independent angle of the given segment: otherAngle
  if(equals(sgmt.p1.x,sgmt.p2.x))
  {//the given segment is vertical, hence its angle is a right angle
    if(equals(sgmt.p1.y,sgmt.p2.y)) return 0; //the given segment is a point
    if(sgmt.p2.y > sgmt.p1.y) otherAngle = Math.PI / 2; //the given angle is positive
    else otherAngle = Math.PI / -2; //the given angle is negative
  }
  else
  {//the given segment is not vertical
    //get the value of the slope of the given segment
    slope = (sgmt.p1.y - sgmt.p2.y) / (sgmt.p1.x - sgmt.p2.x);
    otherAngle = Math.atan(slope);
    //this angle might be wrong yet because it might be > 90 or < -90 degrees)
  }
  //finding out if the angle of the given segment is not between -90 and 90 degrees.
  //If it is not, correct the angle
  if(sgmt.p2.x < sgmt.p1.x)//180 >= otherAngle > 90 degrees or -180 < otherAngle < -90 degrees
    if(sgmt.p2.y >= sgmt.p1.y) otherAngle = otherAngle + Math.PI;//180 >= otherAngle > 90 degrees
    else otherAngle = otherAngle - Math.PI; // -180 < otherAngle < -90 degrees

  //computing the lowest angle and returning it
  finalAngle = (double)Math.round(Math.abs(thisAngle - otherAngle));
  if(finalAngle > Math.PI) return 2 * Math.PI - finalAngle;
```

```
  else return finalAngle;
}

/**************************************************************************
* Checks if this segment has the same orientation than a given segment,
* returning true if the segments have the same orientation and false if not.
* <p>
* It finds the projection of the given segment on the line of this segment, and
* then compares the angle between this segment and the projected segment. Such
* angle is 0 (zero) if the orientation is the same and 180 if it is inverse.
* <p>
* If the projection is a unique point, then this segment and the given segment
* are transversal. This method will assume they have similar orientation
* however it is controversial.
* <p>
* Cares are needed for the case of infinite slope of vertical lines but these
* lines are even easier to work with.
*
* @param sgmt  the given segment
* @return true or false
**************************************************************************/

public boolean sameOrientation(Segment sgmt)
{
  //segment representing the projection of the given segment
  Segment projection = new Segment();

  if(equals(this.p1.x,this.p2.x))
  {//this segment is vertical, hence the projection will also be vertical
    projection.p1.x = this.p1.x; projection.p2.x = this.p1.x;
    projection.p1.y = sgmt.p1.y; projection.p2.y = sgmt.p2.y;
  }
  else
  {//this segment is not vertical, hence the projection will not be vertical either
    //the slope for this segment line and for the projection line are the same
    double slope = (this.p1.y - this.p2.y) / (this.p1.x - this.p2.x);
    if(equals(slope,0))
    {//the segment is horizontal, hence the projection will also be horizontal
      projection.p1.y = this.p1.y; projection.p2.y = this.p1.y;
      projection.p1.x = sgmt.p1.x; projection.p2.x = sgmt.p2.x;
    }
    else
    {//the segment is not horizontal, hence the projection will not be horizontal either
      double displacement = this.p1.y - slope * this.p1.x;

      //the vertical displacement for the line that is transversal to the line of this segment
      //and passes on the startpoint of the given segment (necessary for the projection)
      double transversalDisplacement = sgmt.p1.y + (1/slope) * sgmt.p1.x;

      //the startpoint for the projection segment is defined
      projection.p1.x = (transversalDisplacement - displacement) / (slope + 1/slope);
      projection.p1.y = slope * projection.p1.x + displacement;

      //the vertical displacement for the line that is transversal to the line of this segment
      //and passes on the endpoint of the given segment (necessary for the projection)
      transversalDisplacement = sgmt.p2.y + (1/slope) * sgmt.p2.x;
```

```
      //the endpoint for the projection segment is defined
      projection.p2.x = (transversalDisplacement - displacement) / (slope + 1/slope);
      projection.p2.y = slope * projection.p2.x + displacement;
     }
    }
   if(projection.p1.equals(projection.p2,true)) return true; //the projection is a unique point
   if(equals(this.angle(projection),0)) return true;
   return false; //different orientations
  }

  /**************************************************************************
  * Returns a simple integer representing the side of the turn from this
  * segment to a given segment.&amp;nbsp; the value 0 (zero) means that no turning
  * happened (collinear segments), the value +1 means that a left turning happened,
  * and the value -1 means that a right turning happened.
  *
  * @param sgmt  the given segment
  * @return an integer indicating the turning direction
  **************************************************************************/

  public int turnSign(Segment sgmt)
  {
   double crossProduct = (sgmt.p2.x - this.p1.x) * (this.p2.y - this.p1.y) -
                         (this.p2.x - this.p1.x) * (sgmt.p2.y - this.p1.y);
   if(equals(0,crossProduct)) crossProduct = 0;
   if(crossProduct > 0) crossProduct = 1;
   if(crossProduct < 0) crossProduct = -1;

   return (int) crossProduct;
  }

  /**************************************************************************
  * Checks if the turning from this segment to a given segment is equal to a
  * given turn.&amp;nbsp; the value 0 (zero) means that no turning
  * happened (collinear segments), the value +1 means that a left turning happened,
  * and the value -1 means that a right turning happened.
  * <p>
  * It returns true if the turning directions are the same, or false otherwise.
  *
  * @param sgmt  the given segment
  * @return true or false
  **************************************************************************/

  public boolean sameTurn(Segment sgmt, int sign)
  {
   if(sign * this.turnSign(sgmt) < 0) return false;
   return true;
  }

}//class
```

APPENDIX D – THE *NODE* CLASS

```
package path;

import java.util.Vector;

/**************************************************************************
* This <code>Node</code> class represents nodes of a particular graph where the
* nodes themselves represents paths of a path set.
* <p>
* Not only the nodes are expressed in this class but also their links.
* <p>
* This class includes a method for averaging these nodes.
**************************************************************************/

public class Node
{
 /**
 * The path represented by this node in the graph.
 */
 Path path;
 /**
 * A list of indices (from a graph vector) pointing to nodes linked to this one.
 */
 Vector links;
 /**
 * A flag indicating if this node has already been checked or not during a graph check
 */
 boolean flag = false;

 /**************************************************************************
 * Constructs a node with no initial links for representing a given path.
 *
 * @param p  the given path.
 **************************************************************************/
 public Node(Path p)
 {
  links = new Vector(0);
  path = p;
 }

 /**************************************************************************
 * Computes the average of this node.
 * <p>
 * The average of a node is computed by averaging its path with the averaged
 * path of every linked node in a recursively way, taking care for not repeating
 * paths that already has been part of some other average.
 *
 * @param graph
 * @return the resulting path of the averaged node.
 **************************************************************************/

 public Path nodeAveraging(Vector graph)
 {
```

```
   Path nodeAverage = new Path();//the path resulting from some averaged linked node
   //vector for saving the similarity results of this node path and a linked node path after its average
   Vector similarPaths = new Vector(2);
   int linkIndex;//the index of a linked node from the graph

   this.flag = true;//indicates that this node is already part of an average process

   for(int i=0; i < this.links.size(); i++)
   {//for each linked node
    linkIndex = ((Integer)links.get(i)).intValue();

    if(((Node)graph.get(linkIndex)).flag != true)
    {//this node has not been averaged yet
     nodeAverage = ((Node)graph.get(linkIndex)).nodeAveraging(graph);
     similarPaths = this.path.pathSimilarity(nodeAverage,Const.TLE);
     if(similarPaths.size()!=0)
     {//there is similarity between this node path and the linked node path after its average
      this.path = ((Path)similarPaths.get(0)).pathAveraging((Path)similarPaths.get(1));
      this.path.weight++;
     }
     else if((this.path.weight-this.path.counterWeight) <
           (((Node)graph.get(linkIndex)).path.weight-((Node)graph.get(linkIndex)).path.counterWeight))
         {//there is no similarity and the total weight of the linked node path is heavier
           this.path = ((Node)graph.get(linkIndex)).path;
           this.path.counterWeight++;
         }//else if
    }//if
   }//for i
   return this.path;
  }//method
}//class
```

APPENDIX E – THE *PATH* CLASS

```
package path;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Vector;

/*************************************************************************
* This <code>Path</code> class represents paths that are a set of sequent vertex
* points imaginarely linked by straight segments.
* <p>
* Paths do not depend of the type of coordinates used by their points. The only
* needed change in case of a different coordinate type is to call another method
* in the 'cleaning(File,int)' method for adapting the file incomes instead of
* the 'sphericToPlanar(double,double)' method from the Point class.
* <p>
* Many methods are offered by this class for working with its instances, like
* cleanings, comparisons and averagings.
*************************************************************************/

public class Path
{
 /**
 * The vector containing all vertices of a path in an ordered way.
 */
 public Vector vertexList;

 /**
 * The number of averaged paths that this path results from
 */
 public int weight = 1;

 /**
 * The number of conflicts this path had in anterior averagings
 */
 public int counterWeight  = 0;

 /************************************************************************
 * Constructs a path initializing its ordered vertex list
 ************************************************************************/
 Path()
 {
  vertexList = new Vector(1);
 }

 /************************************************************************
 * Makes this path equal to a given one creating a clone of the vertex list and
 * copying its weight and counterweigth values.
 *
 * @param p  the given path
 ************************************************************************/
```

```
 public void equalize(Path p)
 {
  this.weight = p.weight;
  this.counterWeight = p.counterWeight;
  this.vertexList = (Vector)p.vertexList.clone();
 }

// ************************** CLEANING METHODS ******************************

 /**************************************************************************
 * Obtains a clean path from a file by a process of basic, general or total
 * cleaning during the file data reading.
 * <p>
 * A parameter is the file to be read and the other one indicates the type of
 * cleaning to be done. If its value is 1 (or some unexpected value) then a
 * basic cleaning will be done, if its value is 2 then a general cleaning will
 * be done, and if its value is 3 then a total cleaning will be done.
 * <p>
 * In a basic cleaning the file points that are not OK are just ignored.
 * Consecutive vertices causing too long segments provoke path partitions
 * generating more than one resulting path but any of them containing too long
 * segments. Basic cleanings do not allow fluctuation vertices either and simply
 * ignore new file points that causes fluctuation with the last accepted vertex.
 * <p>
 * General cleanings avoid immediate returns besides all basic cleaning steps.
 * If a new segment (formed by a point read from the file and the last
 * accepted vertex)forms an unaccepted angle with some prior segment close to
 * it, an immediate return happens and a path partition is necessary, where no
 * resulting path will contain both segments causing the immediate return.
 * <p>
 * Total cleanings avoid path self-similarities besides all general cleaning
 * steps. A new segment formed by the last read point from the file has to
 * compare its distance to all the already existing segments in the path being
 * built. If this distance is not long enough, this point will not be added to
 * this path but will be the origin of a new resulting path.
 *
 * @param file the file to obtain vertices from
 * @param type the type of cleaning to be done
 * @return a vector of paths resulting from this cleaning
 **************************************************************************/

 public Vector cleaning(File file, int type)
 {
  int i = 0, j = 0, k = 0;//ordinary counters
  String line = ""; //file lines
  double latitude = 0, longitude = 0;//spherical coordinates of a point from the file
  //a flag for indicating if the first point from the file is being analyzed
  boolean firstPoint = true;

  Vector pathList = new Vector(1); //vector of resulting paths
  Path path = new Path();//a partial resulting path
  double pathLength = 0;//the length of a particular path
  int pathLastIndex = 0;//the last index of a particular file

  Point analysisVertex = new Point();//a vertex just read from the file
  Point lastVertex = new Point();//the last accepted vertex during the cleaning
```

```
Segment analysisSegment = new Segment();//a segment being analyzed during the cleaning
Segment lastSegment = new Segment();//the just formed segment during the file reading

try
{//reading file lines one by one
  BufferedReader br = new BufferedReader(new FileReader(file));
  line = br.readLine();
  while(line!=null)
  {//while the file lines are not over
    line.trim();
    if(line.endsWith("OK"))
    {//if the new vertex is an OK point
      //convertion from file points to path vertices
      i = line.indexOf(";"); j = line.indexOf(";",i+1); k = line.indexOf(";",j+1);
      latitude = Double.parseDouble(line.substring(i+1,j));
      longitude = Double.parseDouble(line.substring(j+1,k));
      analysisVertex.sphericalToPlanar(latitude,longitude);//coordinate translation
      if(analysisVertex.distance(lastVertex) > Const.MSL)
      {//the last segment is too long
        if(firstPoint) firstPoint = false;
        else
        {//it is not the first read file point
          if(pathLength >= Const.MPL) //avoiding too short resulting paths
            {path.vertexList.trimToSize(); pathList.add(path);}
          path = new Path();
          pathLength = 0;
        }
        path.vertexList.add(new Point(analysisVertex));
        lastVertex.equalize(analysisVertex);
      }
      else if(analysisVertex.distance(lastVertex) >= Const.EMD)
      {//if the new vertex is not fluctuation vertex
        if(type > 1)
        {//the general cleaning is solicited
          k = 0;
          pathLastIndex = path.vertexList.size()-1;
          lastSegment.equalize(lastVertex, analysisVertex);
          while(pathLastIndex>k && lastVertex.distance(
                    (Point)path.vertexList.get(pathLastIndex-k))<=Const.PSD)
          {//there are segments from the PSD range set to be checked
            analysisSegment.equalize((Point)path.vertexList.get(pathLastIndex-k),
                              (Point)path.vertexList.get(pathLastIndex-k-1));
            if(lastSegment.angle(analysisSegment) < Const.MAIR && (analysisSegment.
                 distance(analysisVertex) <= Const.PSD || lastSegment.distance(
                      (Point)path.vertexList.get(pathLastIndex-k-1)) <= Const.PSD))
            {//an immediate return ocurred
              if(pathLength >= Const.MPL)
                   {path.vertexList.trimToSize(); pathList.add(path);}
              path = new Path();
              pathLength = 0;
              path.vertexList.add(new Point(lastVertex));
              k = pathLastIndex;
            }//if
            else k++;
          }//while
          if(type == 3)
```

```
          {//the total cleaning is solicited
            while(pathLastIndex > k)
            {//there are segments from the present path to be compared to the analysis segment
              analysisSegment.equalize((Point)path.vertexList.get(pathLastIndex-k),
                                       (Point)path.vertexList.get(pathLastIndex-k-1));
              if(lastSegment.distance(analysisSegment) <= Const.PSD)
              {//if a self-similarity (or self-intersection) ocurred
                if(pathLength >= Const.MPL)
                    {path.vertexList.trimToSize(); pathList.add(path);}
                path = new Path();
                pathLength = 0;
                path.vertexList.add(new Point(lastVertex));
                k = pathLastIndex;
              }//if
              else k++;
            }//while
          }//if (total cleaning)
        }//if (general cleaning)
        //updates before the next file data reading
        path.vertexList.add(new Point(analysisVertex));
        pathLength = pathLength + lastVertex.distance(analysisVertex);
        lastVertex.equalize(analysisVertex);
      }//if (fluctuation check)
    }//if (non-OK point check)
    line = br.readLine(); //read a new line
  }//while (file reading)
 }//try
 catch(IOException e)
 {//an error happened when trying to read the file
  System.err.println("IO Error");
  System.exit(-1);
 }
 //add the last found path to the list if there is some resulting enoughly long path
 if(!firstPoint && pathLength >= Const.MPL)
  {path.vertexList.trimToSize(); pathList.add(path);}
 return pathList;
}

/**************************************************************************
* Cleans a path from fluctuations and unnecessary points for avoiding excess
* of them.
* <p>
* This cleaning is made fragment by fragment where a vertex is selected to be
* the start vertex of a fragment to be analyzed. Next vertices of this fragment
* will be analyzed observing the consequences of this vertex withdrawal. Of
* course these consequences will not be significant in case of fluctuation vertex
* withdrawals, then the analyzed vertex is immediately ignored in such cases.
* <p>
* In other situations, if the summatory of original segment lengths from this
* fragment does not differ much (defined by the TDC constant) from the resulting
* fragment length after a new vertex withdrawal, then this vertex must be
* removed. Otherwise it must not be removed and will be the next start point
* for a new fragment to be analyzed.
*
* @param path the path to be cleaned
**************************************************************************/
```

```
 public void simpleCleaning(Path path)
 {
  //length of two consecutive segments and of the segment resulting from their cleaning
  double firstLength, secondLength, resultingLength = 0;
  //sum of lengths from original segments of a present analyzed fragment
  double lengthSum = 0;
  Point referencePoint = new Point();//start vertex of the fragment to be analyzed
  //flag that indicates if the analysis of the present fragment has already begun
  boolean fragmentCleaning = false;

  for(int i=0;i<path.vertexList.size()-2;i++)
  {//for each pair of consecutive segments
   firstLength = ((Point)path.vertexList.get(i)).distance((Point)path.vertexList.get(i+1));
   secondLength = ((Point)path.vertexList.get(i+1)).distance((Point)path.vertexList.get(i+2));
   if(firstLength <= Const.EMD || secondLength <= Const.EMD)
   {//there are fluctuation vertices
    path.vertexList.remove(i+1);
    i--;
   }
   else
   {//no fluctuation vertices
    if(fragmentCleaning)
    {//the cleaning of the present fragment already started
     resultingLength = referencePoint.distance((Point)path.vertexList.get(i+2));
     lengthSum = lengthSum + firstLength + secondLength;
    }
    else
    {//no cleaning of the present fragment has happened yet
     resultingLength = ((Point)path.vertexList.get(i)).distance((Point)path.vertexList.get(i+2));
     lengthSum = firstLength + secondLength;
    }
    if(lengthSum-resultingLength < Const.MSV)
    {//this fragment need to be cleaned
     if(!fragmentCleaning)
     {//the cleaning of the present fragment is going to start
      referencePoint.equalize((Point)path.vertexList.get(i));
      fragmentCleaning = true;//a new fragment analysis started
     }
     path.vertexList.remove(i+1);
     i--;
    }
    else fragmentCleaning = false;//a fragment analysis ended
   }//else
  }//for i
 }//method

// ********************** SIMILARITY DETECTION METHODS ************************

 /*****************************************************************************
 * Manages the similarity detection between this path and a given path.&bnps;
 * The complete similarity between the entire paths will be checked.
 * <p>
 * A vector is returned containing each path ready for an averaging procedure:
 * Having the same orientation and ignored different tolerable extremes (in
 * accord to a given tolerance length for extremes). If there is no similarity
```

```
* between these paths then the returned vector contain no element.
* <p>
* First, it looks for path similarity between these paths in their original
* orientation. If no similarity is detected then it looks again for similariy
* inverting the orientation of the given path.
* <p>
* The method <code>isSimilar(Path,double)</code> is called for the similarity
* comparison between two paths with the given orientations.
* <p>
* The returned vector contains no element or only two elements: the first
* element is the this path adapted to the averaging procedure and the second
* element is the given path adapted to it.
*
* @param path the given path
* @param tle  tolerance length for path extremes
* @return a vector with two path fragments ready to be averaged
*****************************************************************************/

public Vector pathSimilarity(Path path, double tle)
{
 Vector similarPaths = new Vector(2); //the vector to be returned

 similarPaths = this.isSimilar(path,tle);
 if(similarPaths.size() != 0) return similarPaths;//the paths are similar
 else
 {//there is no similarity for the paths in their original orientation
  //invert the vertex order of the given path
  Path inversePath = new Path();
  for(int i=path.vertexList.size()-1;i >= 0;i--)
       inversePath.vertexList.add(path.vertexList.get(i));
  //try the similarity again for the inverted given path
  similarPaths = this.isSimilar(inversePath,tle);
 }
 return similarPaths;
}

/*****************************************************************************
* Checks if this path and a given path are completely similar (excepting
* tolerable extremes) and returns a vector containing each path ready for an
* averaging procedure in case they are similar.&bnps; Otherwise it returns an
* empty vector.
* <p>
* First, it finds the first and last similar segment from this and the given
* path where to begin the similarity detection.
* <p>
* Then, it calls the method <code>similarityLimit(Segment,Segment)</code> for
* finding the first similar point of each obtained segment respect to the other
* one. The last similar points of the found last similar segments are going to
* be found in the same way but senting the inverted segments (to the mentioned
* method) as if they were intial instead of final.
* <p>
* Another method <code>similarityDetection(Path)</code> is called for checking
* similarity for the reamining part of the paths whether there is similarity in
* all the extremes (tolerated unsimilar extremes are excluded). This method
* works only with the original orientation of this path and the given path.
* <p>
```

```
 * Only the part of the paths that deserve to be averaged is going to be loaded
 * in the returned vector. The first vector element will contain part of this
 * path and the second vector element will contain part of the given path. If
 * the paths are not considered similar the returned vector will be empty. The
 * returned vector must not have more than 2 elements.
 *
 * @param path the given path
 * @param tle  tolerance length for path extremes
 * @return a vector with two path fragments ready to be averaged
 ********************************************************************************/

 private Vector isSimilar(Path path, double tle)
 {
  Vector similarPaths = new Vector(2);//vector containing 2 fragments to be averaged
  int i,j;// counters
  //The length sum of the segments at the extremes that are being tolerated
  double thisExtremeLength = 0, pathExtremeLength = 0;

  //The first similar segment of each path (where the similatity detection will begin)
  Segment thisInitialSegment = new Segment();
  Segment pathInitialSegment = new Segment();

  //********* FIRST PART: AT THE BEGGINING OF THE PATHS **********

  //SEARCHING THE FIRST SIMILAR SEGMENTS AT THE BEGGINING OF THE PATHS)
  //obs: only part of the segment needs to be into the tolerance area to be
  //considered part of it
  for(i=0; thisExtremeLength <= tle && i < this.vertexList.size()-1; i++)
  {//for each segment of this path that belongs to the initial tolerable extreme
   pathExtremeLength = 0;
   thisInitialSegment.equalize((Point)this.vertexList.get(i),
                        (Point)this.vertexList.get(i+1));
   thisExtremeLength = thisExtremeLength + thisInitialSegment.length();
   for(j=0; pathExtremeLength <= tle && j < path.vertexList.size()-1; j++)
   {//for each sgmt at the start of the given path that belongs to the toler. extrs.
    pathInitialSegment.equalize((Point)path.vertexList.get(j),
                         (Point)path.vertexList.get(j+1));
    pathExtremeLength = pathExtremeLength + pathInitialSegment.length();
    //if the first similar segments are found
    if(thisInitialSegment.distance(pathInitialSegment) <= Const.PSD &&
               thisInitialSegment.sameOrientation(pathInitialSegment))
      {i = this.vertexList.size();j = path.vertexList.size();}//end both 'for'
   }
  }

  //DEALING WITH THE SEARCH RESULT

  //The first similar point of each path (where the similatity detection will begin)
  Point thisFirstPoint = new Point(); Point pathFirstPoint = new Point();
  if(thisInitialSegment.distance(pathInitialSegment) <= Const.PSD &&
               thisInitialSegment.sameOrientation(pathInitialSegment))
  {//there are similar segments at the beggining of the paths (into a tolerance area)
   //FIND THE FIRST SIML. POINTS OF THE FOUND SGMTS, CHECK IF THEY ARE IN THE
TOLER. AREA
   //call another method to find the first point
   thisFirstPoint = similarityLimit(thisInitialSegment,pathInitialSegment);
```

```
    //if the first point of this path is out of the tolerance area
    if(thisExtremeLength-new Segment(thisFirstPoint,thisInitialSegment.p2).length()>tle)
        return similarPaths; //return an empty vector (there os no similarity)
    //call another method to find the first point
    pathFirstPoint.equalize(similarityLimit(pathInitialSegment,thisInitialSegment));
    //if the first point of the given path is out of the tolerance area
    if(pathExtremeLength-new Segment(pathFirstPoint,pathInitialSegment.p2).length()>tle)
        return similarPaths; //return an empty vector (there os no similarity)
  }
  else return similarPaths; //return an empty vector (no path similarity)

  //*************** SECOND PART: AT THE END OF THE PATHS *****************

  thisExtremeLength = 0; pathExtremeLength = 0;
  //The last similar segment of each path (where similatity detection will end)
  Segment thisFinalSegment = new Segment();
  Segment pathFinalSegment = new Segment();

  //FINDING THE FIRST SIMILAR SEGMENTS AT THE BEGGINING OF THE PATHS
  for(i=this.vertexList.size()-1; thisExtremeLength <= tle && i > 0; i--)
  {//for each segment of this path that belongs to the final tolerance extremes
    pathExtremeLength = 0;
    thisFinalSegment.equalize((Point)this.vertexList.get(i),
                        (Point)this.vertexList.get(i-1));
    thisExtremeLength = thisExtremeLength + thisFinalSegment.length();
    for(j=path.vertexList.size()-1; pathExtremeLength <= tle && j > 0 ; j--)
    {//for each sgmt at the end of the given path that belongs to the toler. extremes
      pathFinalSegment.equalize((Point)path.vertexList.get(j),
                          (Point)path.vertexList.get(j-1));
      pathExtremeLength = pathExtremeLength + pathFinalSegment.length();
      //if the last similar segments are found
      if(thisFinalSegment.distance(pathFinalSegment) <= Const.PSD &&
          thisFinalSegment.sameOrientation(pathFinalSegment))
            {i = 0; j = 0;}//end of both 'for' cycles
    }
  }

  //DEALING WITH THE SEARCH RESULT

  //The last similar point of each path (where the similatity detection will end)
  Point thisLastPoint = new Point(); Point pathLastPoint = new Point();
  //The resulting fragments of each path after excluding unsimilar tolerated extremes
  Path thisResultingPath = new Path(); Path pathResultingPath = new Path();

  if(thisFinalSegment.distance(pathFinalSegment) <= Const.PSD &&
                      thisFinalSegment.sameOrientation(pathFinalSegment))
  {//there are similar segments at the end of the paths (into a tolerance area)
    //FIND THE LAST SIML. POINTS OF THE FOUND SGMTS, CHECK IF THEY ARE IN THE
TOLER. AREA
    //call another method to find the last point
    thisLastPoint.equalize(similarityLimit(thisFinalSegment,pathFinalSegment));
    //if the last point of this path is out of the tolerance area
    if(thisExtremeLength-new Segment(thisLastPoint,thisFinalSegment.p2).length()>tle)
        return similarPaths;//return an empty vector (no path similarity)
    //else load the path part that deserves to be averaged to the the resulting fragment
    thisResultingPath.vertexList.add(new Point(thisFirstPoint));
```

```
    for(i=thisInitialSegment.p2.indexIn(this.vertexList);
              i != thisFinalSegment.p1.indexIn(this.vertexList) && i!=-1;i++)
       thisResultingPath.vertexList.add(new Point((Point)this.vertexList.get(i)));
    thisResultingPath.vertexList.add(new Point(thisLastPoint));
    //call another method to find the last point
    pathLastPoint.equalize(similarityLimit(pathFinalSegment,thisFinalSegment));
    //if the last point of the given path is out of the tolerance area
    if(pathExtremeLength-new Segment (pathLastPoint,pathFinalSegment.p2).length()>tle)
          return similarPaths;//return an empty vector (no path similarity)
    //else load the path part that deserves to be averaged to the the resulting fragment
    pathResultingPath.vertexList.add(new Point(pathFirstPoint));
    for(i=pathInitialSegment.p2.indexIn(path.vertexList);
              i != pathFinalSegment.p1.indexIn(path.vertexList) && i!=-1;i++)
       pathResultingPath.vertexList.add(new Point((Point)path.vertexList.get(i)));
    pathResultingPath.vertexList.add(new Point(pathLastPoint));
   }
   else return similarPaths; //return an empty vector (no path similarity)

   //********** LAST PART: RETURNING THE RESULTING PATHS ***************

   //CHECK SIMILARITY FOR THE RESULTING FRAGMENTS OF THE PATHS
   //(TOLERATED UNSIMILAR EXTREMES ALREADY EXCLUDED)
   if(thisResultingPath.similarityDetection(pathResultingPath))
   {//Similarity of both fragments accepted. Load the vector with the fragments
    thisResultingPath.vertexList.trimToSize();
    similarPaths.add(thisResultingPath);
    pathResultingPath.vertexList.trimToSize();
    similarPaths.add(pathResultingPath);
   }
   return similarPaths;
}

/*****************************************************************************
* Finds the point of a segment where the similarity begins respect to another
* segment.
* <p>
* This is a private method called by the method <code>isSimilar(Path,double)</code>.
* <p>
* The segment is divided in the middle and the segment half containing similarity
* is chosen to continue the procedure, until the remaining segment length is lower
* than a tenth of the error margin distance, having an acceptable distance
* between the real similarity limit point and some extremepoint of the remaining
* segment.
*
* @param analysisSegment  the main segment, where the first similar point wants
* to be found
* @param referenceSegment the segment referenced to for considering similarity
* @return the first similar point
*****************************************************************************/

private Point similarityLimit(Segment analysisSegment, Segment referenceSegment)
{
  Segment subSegment = new Segment(); //segment to be split
  Segment firstHalf = new Segment(); //first half of the sub-segment
  double accuracy = (Const.EMD/10);
  subSegment.equalize(analysisSegment);//initially the subsgmt = the analysis sgmt
```

```
  while(subSegment.length() > accuracy)
  {//the split continues (the subsegment is too big yet)
   firstHalf.equalize(subSegment.p1,subSegment.p1.weightedAverage(subSegment.p2,1,1));
   //if the first point of similarity is in the first half of the subsegment
   if(firstHalf.distance(referenceSegment.p1)<=Const.PSD-accuracy ||
      firstHalf.distance(referenceSegment.p2)<=Const.PSD-accuracy ||
      referenceSegment.distance(firstHalf.p1)<=Const.PSD-accuracy ||
      referenceSegment.distance(firstHalf.p2)<=Const.PSD-accuracy)
          subSegment.equalize(firstHalf); //eliminate subsegment second half
   //if the first point of similarity is in the second half of the subsegment
   else subSegment.p1.equalize(firstHalf.p2);//eliminate subsegment first half
  }
  return subSegment.p1.weightedAverage(subSegment.p2,1,1);
}

/*****************************************************************************
* Checks if this path and a given path are entirely similar (oriented similarity)
* .&bnps; Returns true if both paths are completely similar, false otherwise.
* <p>
* This is a private method called by the method <code>isSimilar(Path,double)</code>.
* The prior method already eliminated tolerable unsimilar extremes.
* <p>
* For being completely oriented similar paths, they have to fill the following
* conditions.
* <p>
* - Every vertex of this path should have a non-empty Approximation Set on the
* other path: For each vertex of this path, segments from the other path are
* going to be checked searching for the vertex Approximation Set. Not all
* segments need to be checked but all from the initial point of the last
* Approximation Set. Once an Approximation Set ends, the search does not need
* to continue.
* <p>
* - The initial vertex of an Approximation Set cannot be before the initial
* vertex of its prior Approximation Set: These two vertices are always going
* to be checked when a new Approximation set is defined.
* <p>
* - The final vertex of an Approximation set cannot be before the final vertex
* of its prior Approximation Set: These two vertices are always going to be
* checked when a new Approximation set is defined.
* <p>
* - No similarity gaps exists between Approximation Segments of the same set:
* The distance of a vertex and each vertex of its Approximation Set is going
* to be checked. If some of these distances are lower than PSD then the
* similarity fails.
* <p>
* - No similarity gaps exists between two Approximation Sets: The segment
* defined by an analyzed vertex and its prior vertex will be the reference to
* define if all vertices from the Approximation Set of an analyzed vertex are
* close enough to the other path, so no similarity gap happens between
* Approximation Sets of consecutive analyzed vertices. If some of these
* distances are lower than PSD then the similarity fails.
*
* @param path the given path
* @return true or false
*****************************************************************************/
```

```
private boolean similarityDetection(Path path)
{
 Point analysisVertex = new Point(); //the vertex that is being analysed
 Vector approximationSet = new Vector(1);//the approx. set of the present anal. vertex
 Vector lastApproximationSet = new Vector(1);//the approx. set of the last anal. vertex
 int i, j=0; //counters
 //the segment being compared to the present analysed vertex
 Segment analysisSegment = new Segment();
 //the segment between the analysed vertex and the last analysed vertex
 Segment comparisonSegment = new Segment();
 //flag indicating if there are still possible approx. sgmts to be found for the approx. set
 int flag;// 0 = yes, 1 = no
 for(i=0;i < this.vertexList.size();i++)
 {//for each vertex of this path
  analysisVertex.equalize((Point)this.vertexList.get(i));
  flag = 0; //let find approximation sets
  while(flag == 0 && j < path.vertexList.size()-1)
  {//for each segment of the given path
   analysisSegment.equalize((Point)path.vertexList.get(j),
                            (Point)path.vertexList.get(j+1));
   //Building the Approx. Set for the vertex and checking similarity constraints
   if(analysisSegment.distance(analysisVertex) <= Const.PSD &&
     (approximationSet.size()==0 ||
       analysisSegment.p1.distance(analysisVertex) <= Const.PSD))
   {//another approximation segment without similarity gaps is found
    //the approx. set is empty, add the first vertex of the segment
    if(approximationSet.size()==0) approximationSet.add(new Integer(j));
    approximationSet.add(new Integer(j+1));//add the second segment vertex
    j++; //jump to the next segment
   }
   else
   {//it is not an approximation segment
    if(approximationSet.size()!=0)
    {//the Approximation Set is completly defined
     flag = 1; //no more approximation segments to be added to the approx. set
     if(lastApproximationSet.size()!=0)
     {//this Approximation Set is not empty... let check similarity conditions
      if(((Integer)approximationSet.firstElement()).intValue() <
        ((Integer)lastApproximationSet.firstElement()).intValue())
       return false;//checking the first vertices of consecutive approx. sets
      if(((Integer)approximationSet.lastElement()).intValue() <
        ((Integer)lastApproximationSet.lastElement()).intValue())
       return false;//checking the last vertices of consecutive approx. sets
      comparisonSegment.equalize((Point)this.vertexList.get(i-1),analysisVertex);
      //checking path similarity between consecutive approx. sets
      for(int k=((Integer)lastApproximationSet.lastElement()).intValue();
          k <= ((Integer)approximationSet.firstElement()).intValue();k++)
       if(comparisonSegment.distance((Point)path.vertexList.get(k))>Const.PSD)
        return false;//a gap occured
     }//the Approximation set follow every similarity condition
     //save this Approximation Set as the last one and initialize a new empty one
     lastApproximationSet = approximationSet;
     approximationSet = new Vector(1);
     j=0; //jump to the next vertex
    }
    else if(j==path.vertexList.size()-2)//are there segments yet?
```

```
            return false; //some vertex has no approximation segment
          else j++;//jump to the next segment
       }//else
     }//while j
   }//for i
   return true; //the paths are completely similar
  }

// ***************************** AVERAGING METHODS ****************************

  /*****************************************************************************
  * Computes the average between this path an a given path.
  * <p>
  * Returns a path resulting from the average. Any initial path can be empty
  * otherwise an empty path is returned immediatly as result.
  * <p>
  * The endpoints for the average path are computed and added to the initial
  * paths, and in case they are equal the average result is going to be just a
  * point.
  * <p>
  * Then the paths are fragmented in accord to their intersection points
  * forming polygons (pairs of fragments). The polygons are averaged one by one.
  * If some error happens during a polygon average (or in the path fragmentation)
  * the path having the highest total weight is returned (if the total weights
  * are equal this path is returned by default).
  * <p>
  * The first and last polygons are eliminated whether they are very short. It
  * happens to avoid path distortions in short averagings as branchings and
  * crossings.
  *
  * @param path the given path
  * @return the averaged path
  *****************************************************************************/

  public Path pathAveraging(Path path)
  {
   //if some path is empty return a empty path as result
   if(path.vertexList.size()==0 || this.vertexList.size()==0) return new Path();

   //the endpoints for the resulting (averaged) path
   Point origin = new Point(); Point destination = new Point();
   //the fragment lists
   Vector thisFragmentList = new Vector(); Vector otherFragmentList = new Vector();
   Path averagePath = new Path(); //the resulting path of the average
   Vector averageFragment = new Vector();//average of each pair of fragment lists
   //vertex (from each path) to be analyzed
   Point thisAnalysisVertex = new Point(); Point otherAnalysisVertex = new Point();
   boolean error = false; //a flag for indicating if an error happened
   int i;//counter

   //computing the resulting origin vertex
   thisAnalysisVertex.equalize((Point)this.vertexList.get(0));
   otherAnalysisVertex.equalize((Point)path.vertexList.get(0));
   origin.equalize(thisAnalysisVertex.weightedAverage(
                           otherAnalysisVertex,this.weight,path.weight));
   if(!thisAnalysisVertex.equals(otherAnalysisVertex))
```

```
{//the origin vertices are not equal (the computed origin must be added)
 thisFragmentList.add(origin);
 otherFragmentList.add(origin);
}
//copying the paths to the fragment lists
for(i=0;i < this.vertexList.size();i++)
        thisFragmentList.add((Point)this.vertexList.get(i));
for(i=0;i < path.vertexList.size();i++)
        otherFragmentList.add((Point)path.vertexList.get(i));

//computing the resulting destination vertex
thisAnalysisVertex.equalize((Point)this.vertexList.lastElement());
otherAnalysisVertex.equalize((Point)path.vertexList.lastElement());
destination.equalize(thisAnalysisVertex.weightedAverage(
                        otherAnalysisVertex,this.weight,path.weight));
if(!thisAnalysisVertex.equals(otherAnalysisVertex))
{//destination vertices are not equal (the computed estination might be added)
 thisFragmentList.add(destination);
 otherFragmentList.add(destination);
}

//checking if the resulting path is not just a point
if(origin.equals(destination,true))
{//the origin and the destination are the same
 averagePath.vertexList.add(thisAnalysisVertex);
 averagePath.weight = this.weight + path.weight;
 averagePath.counterWeight = this.counterWeight + path.counterWeight;
 return averagePath;
}

//the path fragmentation
if(this.createFragmentLists(thisFragmentList,otherFragmentList))
{//the fragmentations has been successful.
 //avoiding path distortions in case of short similarities (e.g.crossings)
 if(thisFragmentList.size()>0)
  {
   if(((Point)((Vector)thisFragmentList.firstElement()).firstElement()).distance(
      (Point)((Vector)thisFragmentList.firstElement()).lastElement())
         < Const.MAFL)
            {thisFragmentList.remove(0);otherFragmentList.remove(0);}
  }
 if(thisFragmentList.size()>0)
 {
  if(((Point)((Vector)thisFragmentList.lastElement()).firstElement()).distance(
     (Point)((Vector)thisFragmentList.lastElement()).lastElement())
         < Const.MAFL)
   {
    thisFragmentList.remove(thisFragmentList.size()-1);
    otherFragmentList.remove(otherFragmentList.size()-1);
   }
 }

 //averaging pair of fragments
 int j; //counter
 for(i=0;i<thisFragmentList.size();i++)
 {//for each (pair of) fragment list
```

```
      Polygon polygon = new Polygon((Vector)thisFragmentList.get(i),
                  (Vector)otherFragmentList.get(i),this.weight,path.weight);
      averageFragment = polygon.polygonAveraging();
      if(averageFragment.size()==0)
        {error = true; i = thisFragmentList.size();}//error during polygon averaging
      else for(j=0;j<averageFragment.size()-1;j++)
              averagePath.vertexList.add((Point)averageFragment.get(j));
    }
   }
   else error = true; //error due to different order of intersection points

   //dealing with possible errors due to pairs of fragments that cannot be averaged
   if(error)
   {//an error happened
    //return the path with greater resulting weight as the result (ignore the other)
    if((this.weight - this.counterWeight) >= (path.weight - path.counterWeight))
      averagePath.equalize(this);
    if((this.weight - this.counterWeight) < (path.weight - path.counterWeight))
      averagePath.equalize(path);
    averagePath.counterWeight++;
    return averagePath;
   }

   //returning the resulting averaged path
   averagePath.weight = this.weight + path.weight;
   averagePath.counterWeight = this.counterWeight + path.counterWeight;
   simpleCleaning(averagePath);
   return averagePath;
  }

  /**************************************************************************
   * Converts two given paths to a pair of correspondent fragment lists.
   * <p>
   * It returns true in case that no error happens during this process or false
   * otherwise. The error occurs if the paths have different sequence of
   * intersection points
   * <p>
   * It finds each intersection points of both paths and (for each path) it adds
   * to a list the correspondent vertices between found pairs of consecutive
   * intersection points. Each time a intersection point is found, a list of
   * vertices (or fragment) is loaded in a vector and a new empty list is going
   * to be used for adding vertices into, creating then the fragment lists.
   * <p>
   * constraints: the order of intersections for each path will be checked and they
   * must be the same for both given paths. The paths also must have the same origin
   * and destination point. If the constraints are not accomplished, a null vector
   * will be returned indicating a failure.
   *
   * @param fragmentList1 the first given path to be fragmented
   * @param fragmentList2 the second given path to be fragmented
   * @return true or false
   *************************************************************************/

  private boolean createFragmentLists(Vector fragmentList1, Vector fragmentList2)
  {
   //clones of the original given paths (the fragment lists are going to be altered)
```

```
Vector path1 = (Vector)fragmentList1.clone();
Vector path2 = (Vector)fragmentList2.clone();
fragmentList1.clear(); fragmentList2.clear();
//the paths must have the same endpoints otherwise it indicating a failure
if(!((Point)path1.firstElement()).equals((Point)path2.firstElement()) ||
   !((Point)path1.lastElement()).equals((Point)path2.lastElement())) return false;

Vector fragment = new Vector();//the present fragment being buit
//Segments being analyzed to find for intersections
Segment analysisSegment1 = new Segment();
Segment analysisSegment2 = new Segment();
//Segment resulting from the intersection (inters. are not always simple points)
Segment intersectionSegment = new Segment();
//Point where the last intersection happened
Point lastIntersection = new Point();
lastIntersection.equalize((Point)path1.firstElement());
//segment index from each path where the last intersection happened
int lastIndex1 = 0; int lastIndex2 = 0;
int i, j, k;//counters

for(i=0; i < path1.size()-1; i++)
{//for each segment from the first given path
 analysisSegment1.equalize((Point)path1.get(i),(Point)path1.get(i+1));
 for(j=0; j < path2.size()-1; j++)
 {//for each segment from the second given path
  analysisSegment2.equalize((Point)path2.get(j),(Point)path2.get(j+1));
  if(analysisSegment1.checkIntersection(analysisSegment2))
  {//the analyzed segments do intersect
   intersectionSegment.equalize(analysisSegment1.intersection(analysisSegment2));
   if(intersectionSegment.p1.equals(intersectionSegment.p2,true))
   {//the intersection is just a point (ignore them if they are segments)
    if(!intersectionSegment.p1.equals(lastIntersection,true))
    {//this intersection is not equal to the last intersection (is a new one)
     //check that the order of intersections is equal for both paths
     if(i == lastIndex1 && analysisSegment1.p1.distance(intersectionSegment.p1)
        < analysisSegment1.p1.distance(lastIntersection))
           return false; //Error: unequal intersection sequences
     if(j < lastIndex2 || (j == lastIndex2 && analysisSegment2.p1.distance(
       intersectionSegment.p1)<analysisSegment2.p1.distance(lastIntersection)))
           return false; //Error: unequal intersection sequences

     //add to the fragments the vertices from the last intersection
     //point to the present one (without repeating points)
     fragment.add(new Point(lastIntersection));
     for(k = lastIndex1+1; k <= i ;k++) fragment.add((Point)path1.get(k));
     if(!((Point)fragment.lastElement()).equals(intersectionSegment.p1,true))
      fragment.add(new Point(intersectionSegment.p1));
     fragment.trimToSize();
     fragmentList1.add(fragment);
     fragment = new Vector();

     fragment.add(new Point(lastIntersection));
     for(k = lastIndex2+1; k <= j ;k++) fragment.add((Point)path2.get(k));
     if(!((Point)fragment.lastElement()).equals(intersectionSegment.p1,true))
      fragment.add(new Point(intersectionSegment.p1));
     fragment.trimToSize();
```

```
          fragmentList2.add(fragment);
          fragment = new Vector();
          lastIntersection.equalize(intersectionSegment.p1);
         }
        lastIndex1 = i; lastIndex2 = j;
       }//if
      }//if
    }//for j
  }//for i

  fragmentList1.trimToSize(); fragmentList2.trimToSize();
  return true;
}//method


/**************************************************************************
* Checks if there are similar fragments between this path and a given path
* .&bnps; Computes average fragments in case of similarity and returns a vector
* of resulting paths after this averaging.
* <p>
* Each segment of this path is checked to every segment of the given path until
* a segment similarity happens. Then, prior and next segments are going to be
* checked until the entire similar fragment is defined. Only the first fragment
* similarity is found by this method.
* <p>
* Unsimilar extremes of the similar fragments are eliminated by replacing the
* first/last fragment points by the first/last similar points. Then the
* fragment average is computed. Only reasonable averages are accepted to
* realize changes in the original paths.
* <p>
* When two similar fragments are averaged, the original fragment of this path
* is replaced by the averaged one, and the original fragment of the given path
* is eliminated from it (perhaps generating two smaller paths if the
* eliminated fragment were not at the start or end of the path). Therefore,
* one, two or three new paths are possible from this similarity. Very short
* resulting paths are not returned but just ignored. No paths are returned in
* case of no similarity.
* <p>
* If more than one new path results from a fragment average, they are redefined
* by linking them (equaling an endpoint of a path with some point from another)
* before they are returned. These links have been improvised and can be improved.
*
* @param path      the given path
* @param pathList  vector where the resulting paths are loaded to
**************************************************************************/

public void fragmentAveraging(Path path, Vector pathList)
{
  //segments to be analized from each path
  Segment thisAnalysisSegment = new Segment();
  Segment otherAnalysisSegment = new Segment();
  int i, j, k;//counters

  for(i=0; i < this.vertexList.size()-1; i++)
  {//for each segment from this path
    thisAnalysisSegment.equalize((Point)this.vertexList.get(i),
```

```
                    (Point)this.vertexList.get(i+1));
for(j=0; j < path.vertexList.size()-1 && i < this.vertexList.size()-1; j++)
{//for each segment from the given path
 otherAnalysisSegment.equalize((Point)path.vertexList.get(j),
                        (Point)path.vertexList.get(j+1));
 if(thisAnalysisSegment.distance(otherAnalysisSegment.p1) <= Const.PSD ||
    thisAnalysisSegment.distance(otherAnalysisSegment.p2) <= Const.PSD ||
    otherAnalysisSegment.distance(thisAnalysisSegment.p1) <= Const.PSD ||
    otherAnalysisSegment.distance(thisAnalysisSegment.p2) <= Const.PSD)
 {//a similarity happened for two segments
  //start and end indices for the similar fragment
  int thisStartIndex=i,thisEndIndex=i+1,otherStartIndex=j,otherEndIndex=j+1;
  //the initial and final segments of the present defined similar fragments
  Segment thisFinalSegment = new Segment(thisAnalysisSegment);
  Segment otherInitialSegment =
          new Segment((Point)path.vertexList.get(otherStartIndex),
                  (Point)path.vertexList.get(otherEndIndex));
  Segment otherFinalSegment =
          new Segment((Point)path.vertexList.get(otherStartIndex),
                  (Point)path.vertexList.get(otherEndIndex));
  //flag for indicating if the similar fragments have inverse orientation
  boolean inverseOrientation = false;
  //flag for indicating continuation of the definition of similar fragments
  boolean definingFragments = true;
  while(definingFragments)
  {//the similar fragments are not completely defined yet
   definingFragments = false;
   if(otherStartIndex > 0 &&
      thisFinalSegment.distance(otherInitialSegment.p1) < Const.PSD)
   {//recede the first index of the second path (inverted orientations)
    otherStartIndex--; definingFragments = true; inverseOrientation = true;
    otherInitialSegment.equalize((Point)path.vertexList.get(otherStartIndex),
                        (Point)path.vertexList.get(otherStartIndex+1));
   }
   if(otherEndIndex < path.vertexList.size()-1 &&
         thisFinalSegment.distance(otherFinalSegment.p2) < Const.PSD)
   {//advance the last index of the second path
    otherEndIndex++; definingFragments = true;
    otherFinalSegment.equalize((Point)path.vertexList.get(otherEndIndex-1),
                         (Point)path.vertexList.get(otherEndIndex));
   }
   if(thisEndIndex < this.vertexList.size()-1 &&
         (otherInitialSegment.distance(thisFinalSegment.p2) < Const.PSD ||
           otherFinalSegment.distance(thisFinalSegment.p2) < Const.PSD))
   {//advance the last index of the first (main) path
    thisEndIndex++; definingFragments = true;
    thisFinalSegment.equalize((Point)this.vertexList.get(thisEndIndex-1),
                        (Point)this.vertexList.get(thisEndIndex));
   }
  }//while

  //CREATING THE FRAGMENTS TO BE AVERAGED
  //similar fragments to be created and averaged
  Path thisFragment = new Path(); Path otherFragment = new Path();
  //defining the first similar point for the similar fragments
  Segment thisFirstSegment = new Segment(
```

```
                (Point)this.vertexList.get(thisStartIndex),
                (Point)this.vertexList.get(thisStartIndex+1));
Segment otherFirstSegment = new Segment();
if(!inverseOrientation)
   otherFirstSegment.equalize((Point)path.vertexList.get(otherStartIndex),
                    (Point)path.vertexList.get(otherStartIndex+1));
else otherFirstSegment.equalize((Point)path.vertexList.get(otherEndIndex),
                     (Point)path.vertexList.get(otherEndIndex-1));
thisFragment.vertexList.add(
      similarityLimit(thisFirstSegment,otherFirstSegment));
otherFragment.vertexList.add(
      similarityLimit(otherFirstSegment,thisFirstSegment));
//loading the fragments
for(k=thisStartIndex+1; k<thisEndIndex; k++)
         thisFragment.vertexList.add((Point)this.vertexList.get(k));
if(!inverseOrientation)
   for(k=otherStartIndex+1; k < otherEndIndex; k++)
       otherFragment.vertexList.add((Point)path.vertexList.get(k));
else for(k=otherEndIndex-1; k > otherStartIndex; k--)
         otherFragment.vertexList.add((Point)path.vertexList.get(k));
//defining the last similar point for the similar fragment
Segment thisLastSegment = new Segment(
                (Point)this.vertexList.get(thisEndIndex),
                (Point)this.vertexList.get(thisEndIndex-1));
Segment otherLastSegment = new Segment();
if(!inverseOrientation)
    otherLastSegment.equalize((Point)path.vertexList.get(otherEndIndex),
                (Point)path.vertexList.get(otherEndIndex-1));
else otherLastSegment.equalize((Point)path.vertexList.get(otherStartIndex),
               (Point)path.vertexList.get(otherStartIndex+1));
thisFragment.vertexList.add(
            similarityLimit(thisLastSegment,otherLastSegment));
otherFragment.vertexList.add(
            similarityLimit(otherLastSegment,thisLastSegment));

//averaging the similar fragments
Vector avg = new Vector();
avg = (thisFragment.pathAveraging(otherFragment)).vertexList;

if(avg.size() > 0)
{//there is an acceptable average
  Path averagedPath = new Path(); //resulting path after the averaging
  averagedPath.weight = this.weight;
  averagedPath.counterWeight = this.counterWeight;

  //replacing the averaged fragment in this path
  for(k=0; k <= thisStartIndex; k++)
       averagedPath.vertexList.add((Point)this.vertexList.get(k));
  for(k=0;k<avg.size();k++) averagedPath.vertexList.add(avg.get(k));
  for(k=thisEndIndex; k < this.vertexList.size(); k++)
       averagedPath.vertexList.add((Point)this.vertexList.get(k));
  pathList.add(averagedPath);

  //creating a new path anterior to the similar fragment to be excluded
  Path fragmentedPath = new Path();
  fragmentedPath.weight = path.weight;
```

```
        fragmentedPath.counterWeight = path.counterWeight;
        for(k=0; k <= otherStartIndex; k++)
              fragmentedPath.vertexList.add((Point)path.vertexList.get(k));
        //links the average and this new path
        if(!inverseOrientation) fragmentedPath.vertexList.add(avg.firstElement());
        else fragmentedPath.vertexList.add(avg.lastElement());
        //saving resulting paths but ignoring too short ones
        double length=0;
        for(k=0; length < Const.MPL && k < fragmentedPath.vertexList.size()-1; k++)
             length = length + ((Point)fragmentedPath.vertexList.get(k)).distance(
                          (Point)fragmentedPath.vertexList.get(k+1));
        if(length >= Const.MPL) pathList.add(fragmentedPath);

        //creating a new path posterior to the similar fragment to be excluded
        fragmentedPath = new Path();//
        //links the average and this new path
        if(!inverseOrientation) fragmentedPath.vertexList.add(avg.lastElement());
        else fragmentedPath.vertexList.add(avg.firstElement());
        for(k=otherEndIndex; k < path.vertexList.size(); k++)
              fragmentedPath.vertexList.add(path.vertexList.get(k));
        //saving resulting paths but ignoring too short ones
        length=0;
        for(k=0; length < Const.MPL && k < fragmentedPath.vertexList.size()-1; k++)
             length = length + ((Point)fragmentedPath.vertexList.get(k)).distance(
                          (Point)fragmentedPath.vertexList.get(k+1));
        if(length >= Const.MPL) pathList.add(fragmentedPath);

        i = this.vertexList.size();//finishing both 'for' cycles
       }//if (similarity is acceptable)
      }//if (similarity happened)
     }//for j
    }//for i
   }//method

}//class
```

APPENDIX F – THE *POLYGON* CLASS

```
package path;
import java.util.Vector;

/**************************************************************************
* This <code>Polygon</code> class represents geometrical figures formed by two
* lines (defined by lists of vertices) that have an equal origin and destination
* point but is expected to have no other intersection point.
* <p>
* Polygons do not depend of the type of coordinates used by their vertices.
* <p>
* The public method offered by this class is for averaging a polygon. Other
* private methods are subparts of this public method.
**************************************************************************/

public class Polygon
{
 /**
 * The list of vertices that define the polygon boundaries.
 */
 Vector vertexList1;
 Vector vertexList2;
 /**
 * The weight for each list of vertices.
 */
 int weight1;
 int weight2;
 /**
 * The origin point for both list of vertices (also considered the polygon
 * origin point).
 */
 Point originVertex;
 /**
 * The destination point for both list of vertices (also considered the polygon
 * destination point).
 */
 Point destinationVertex;

 /**************************************************************************
 * Constructs a polygon formed by two given lists of vertices and their weights.
 *
 * @param vlist1 the first given list of vertices
 * @param vlist2 the second given list of vertices
 * @param w1     the weight of the first given list of vertices
 * @param w2     the weight of the second given list of vertices
 **************************************************************************/

 public Polygon(Vector vlist1, Vector vlist2,int w1, int w2)
 {
  this.vertexList1 = (Vector)vlist1.clone();
  this.weight1 = w1;
  this.vertexList2 = (Vector)vlist2.clone();
  this.weight2 = w2;
```

```
  this.originVertex = new Point((Point)vlist1.get(0));
  this.destinationVertex = new Point((Point)vlist1.lastElement());
 }

 /**************************************************************************
 * Computes the average of this polygon by finding the average of its two paths
 * formed by its lists of vertices.&bnps; The average is done proportionaly
 * to the weight of each list.
 * <p>
 * The polygon averaging is done by defining sequent internal convex
 * subpolygons, averaging them and adding each average to a vector to be
 * returned at the end of this method.
 * <p>
 * Consecutive convex lists are computed for each list of vertices and special
 * situations are handled. The convex lists are computed by another method and
 * the prior function of this method is to manage the polygon averaging and to
 * deal with exceptions.
 * <p>
 * Convex subpolygons are defined from each pair of computed convex lists. The
 * subpolygons are also defined and averaged by other methods. An empty vector
 * is returned in case of average failure.
 *
 * @return a vector containing the resulting list of vertices after the average
 **************************************************************************/

 public Vector polygonAveraging()
 {
  Vector averageFragment = new Vector();//the resulting fragment of this average
  averageFragment.add(new Point(this.originVertex));//add the first average point
  Vector convexList1=new Vector(); Vector convexList2=new Vector();//convex lists
  //first indices of the next convex lists to be built (next leader vtx indices)
  int firstIndex1=0; int firstIndex2=0;
  int i, counter = 0; //counters
  boolean goBackAndTryAgain = false;//flag to jump some steps in a while loop
  Vector auxiliarVector = new Vector(2);//just for help

  //dealing with initial exceptions: at least 1 leader vertex cannot advance
  if(((Point)this.vertexList1.get(1)).equals(destinationVertex,false) &&
       ((Point)this.vertexList2.get(1)).equals(destinationVertex,false))
  {//both leader vertices have the destination point as the next vertex
   averageFragment.add(new Point(this.destinationVertex));
   return averageFragment;
  }
  if(((Point)this.vertexList1.get(1)).equals(destinationVertex,false))
  {//only the leader vertex 1 have the destination point as the next vertex
   this.vertexList1.remove(this.vertexList1.size()-1);//remove the last element
   this.vertexList1.add(originVertex.weightedAverage(destinationVertex,1,1));
   this.vertexList1.add(new Point(destinationVertex));
  }
  if(((Point)this.vertexList2.get(1)).equals(destinationVertex,false))
  {//only the leader vertex 2 have the destination point as the next vertex
   this.vertexList2.remove(this.vertexList2.size()-1);//remove the last element
   this.vertexList2.add(originVertex.weightedAverage(destinationVertex,1,1));
   this.vertexList2.add(new Point(destinationVertex));
  }
```

```
while(true)
{//until method results are returned
 //building a convex lists
 convexList1 = createConvexList(this.vertexList1,this.vertexList2,
                        firstIndex1,firstIndex2,destinationVertex);
 convexList2 = createConvexList(this.vertexList2,this.vertexList1,
                        firstIndex2,firstIndex1,destinationVertex);
 //handling an error: empty convex list
 if(convexList1.size()==0 || convexList2.size()==0) return new Vector();

 //handling special cases
 if(convexList1.size()==1 && convexList2.size()==1)
 {//both convex lists contain only one vertex and cannot advance
  if(!((Point)convexList1.get(0)).equals(
        (Point)this.vertexList1.get(this.vertexList1.size()-2),false) &&
     !((Point)convexList2.get(0)).equals(
        (Point)this.vertexList2.get(this.vertexList2.size()-2),false))
       return new Vector();//no leader vtx has the dest. point as next vtx
  if(((Point)convexList1.get(0)).equals(
        (Point)this.vertexList1.get(this.vertexList1.size()-2),false) &&
     ((Point)convexList2.get(0)).equals(
        (Point)this.vertexList2.get(this.vertexList2.size()-2),false))
  {//both leader vertices have the destination point as its next vertex
   averageFragment.add(new Point(this.destinationVertex));
   return averageFragment;
  }
  if(((Point)convexList1.get(0)).equals(
         (Point)this.vertexList1.get(this.vertexList1.size()-2),false))
  {//only the leader vertex 1 has the destination point as its next vertex
   this.vertexList1.remove(this.vertexList1.size()-1);//remove the last vtx
   this.vertexList1.add(((Point)convexList1.get(0)).
                          weightedAverage(destinationVertex,1,1));
   this.vertexList1.add(destinationVertex);
   if(counter++ > 10) return new Vector();//avoid infinite loops (error)
  }
  else
  {//only the leader vertex 2 has the destination point as its next vertex
   this.vertexList2.remove(this.vertexList2.size()-1);//remove the last vtx
   this.vertexList2.add(((Point)convexList2.get(0)).
                       weightedAverage(this.destinationVertex,1,1));
   this.vertexList2.add(new Point(this.destinationVertex));
   if(counter++ > 10) return new Vector();
  }
  goBackAndTryAgain=true;//jump the next steps and redefine a new convex list
 }

 if(!goBackAndTryAgain)
 {//define a convex subpolygon, find its average and add it to the vector
  if(!toConvexSubpolygon(convexList1,convexList2)) return new Vector();
  if(convexList1.size()==0 || convexList2.size()==0) return new Vector();
  auxiliarVector = convexSubpolygonAveraging(convexList1,convexList2);
  for(i=1;i < auxiliarVector.size();i++)
          averageFragment.add((Point)auxiliarVector.get(i));
 }//if
 //updating variables
 goBackAndTryAgain = false;
```

```
    firstIndex1 = firstIndex1 + convexList1.size()-1;
    firstIndex2 = firstIndex2 + convexList2.size()-1;
    convexList1.clear(); convexList2.clear();
   }//while (Go back to define a new convex list)
  }//method

  /**************************************************************************
  * Creates a convex list for a given list of vertices having as reference its
  * own leader vertex, another given list of vertices and its leader vertex.
  * <p>
  * The computed convex list is going to be as longer as possible but always
  * respecting the convexity, no spirality and other requirements respect to the
  * other given convex list (internal angle greater than 180).
  * <p>
  * A vector containing the resulting convex list is returned but it will be
  * empty in case of some occurred error.
  *
  * @param thisVertexList   the first given list of vertices
  * @param otherVertexList  the reference list of vertices
  * @param thisFirstIndex   the index for finding the leader vertex of the
  * given list to find the convex list from
  * @param otherFirstIndex  the index for finding the leader vertex of the
  * reference list
  * @return a vector with the resulting convex lists
  **************************************************************************/

  private Vector createConvexList(Vector thisVertexList, Vector otherVertexList,
                                  int thisFirstIndex, int otherFirstIndex, Point destination)
  {
   Vector convexList = new Vector();//the convex lists to be returned
   convexList.add(new Point((Point)thisVertexList.get(thisFirstIndex)));

   //the subpolygon boundary segment between the prior one and this one to be built
   Segment firstinternalSegment = new Segment((Point)otherVertexList.get(otherFirstIndex),
                                  (Point)thisVertexList.get(thisFirstIndex));
   //the next segment to be analized
   Segment analysisSegment = new Segment((Point)thisVertexList.get(thisFirstIndex),

(Point)thisVertexList.get(thisFirstIndex+1));
   Segment priorSegment; //the polygon segment before the one to be analized
   if(thisFirstIndex != 0)//the convex list does not begin in the polygon origin
    priorSegment = new Segment((Point)thisVertexList.get(thisFirstIndex-1),

(Point)thisVertexList.get(thisFirstIndex));
   else priorSegment = new Segment((Point)thisVertexList.get(thisFirstIndex),

(Point)thisVertexList.get(thisFirstIndex));

   if(priorSegment.p1.equals(priorSegment.p2,false) ||
        !new Segment(firstinternalSegment.p1, analysisSegment.p2).
              checkIntersection(priorSegment))
   {//ok: internal angle <= 180
    //integer for indicating which side the subpolygon is turning to
    int turnSign = firstinternalSegment.turnSign(analysisSegment);
    //the index of the last added vertex to the convex list
    int lastIndex = thisFirstIndex;
```

```
    //the segment linking the convex list endpoints
    Segment linkSegment =
       new Segment(analysisSegment.p2,(Point)thisVertexList.get(thisFirstIndex));
    priorSegment.equalize(firstinternalSegment);//for getting in the while

    while(!analysisSegment.p2.equals(destination,false) &&
            priorSegment.sameTurn(analysisSegment,turnSign) &&
                analysisSegment.sameTurn(linkSegment,turnSign))
    {//next segment accepted: more one element for the convex list
      if(!analysisSegment.p1.equals(firstinternalSegment.p2,false) &&
         analysisSegment.checkIntersection(firstinternalSegment))
              return new Vector(); //b) Error: internal immediate return
      if(turnSign == 0)
         turnSign=priorSegment.turnSign(analysisSegment);//avoid random turn side
      convexList.add(new Point(analysisSegment.p2));//add new vtx to this cvx list
      //update the variables
      lastIndex++;
      priorSegment.equalize(analysisSegment);
      //avoid unnexpected errors
      if(lastIndex>=thisVertexList.size()-1) return new Vector();
      analysisSegment.equalize(analysisSegment.p2,
                          (Point)thisVertexList.get(lastIndex+1));
      linkSegment.p1.equalize(analysisSegment.p2);
     }//while
   }//if

   convexList.trimToSize();
   return convexList;
}

 /**************************************************************************
 * Redefines the given convex lists to form a convex subpolygonand and returns
 * a flag indicating success or not in the subpolygon formation.
 * <p>
 * Only the part of each convex lists that forms a convex polygon is going to be
 * returned. The convex lists are reduced by eliminating its last points one by
 * one until they form a convex subpolygon.
 * <p>
 * Another method is called to know if the convex lists must be reduced or not.
 * This method just average the definition of the subpolygon and deal with
 * some possible problems that can happen.
 *
 * @param convexList1  the first given convex list to form the subpolygon
 * @param convexList2  the second given convex list to form the subpolygon
 * @return true or false (success or not)
 **************************************************************************/

 private boolean toConvexSubpolygon(Vector convexList1, Vector convexList2)
 {
   //checking and solving problem caused by convex lists turning to the same side
   if(convexList1.size()>2 && convexList2.size()>2)
   {//both lists have at least 2 segments (3 vertices)
     //first and second segment of each list
     Segment first1=new Segment((Point)convexList1.get(0),(Point)convexList1.get(1));
     Segment second1=new Segment((Point)convexList1.get(1),(Point)convexList1.get(2));
     Segment first2=new Segment((Point)convexList2.get(0),(Point)convexList2.get(1));
```

```
    Segment second2=new Segment((Point)convexList2.get(1),(Point)convexList2.get(2));
    //checking the convex lists turnings
    // -1 = turns to the right; +1 = turns to the left; 0 = do not turn
    int turnSide = first1.turnSign(second1);
    if(first2.sameTurn(second2,turnSide))
    {//both lists have turn to the same side
      //only the first segment from the most intern convex list interests
      if(first1.turnSign(first2) == turnSide) convexList2.setSize(2);
      else convexList1.setSize(2);
    }
  }

  while(mustBeReduced(convexList1,convexList2)==1)
  {//elements from the convex list 1 have to be removed for fixing the subpolygon
    if(convexList1.size() > 1) convexList1.remove(convexList1.size()-1);
    else
    {//elements cannot be removed from convex list 1 anymore
      //remove every element from the convex list 2 but it endpoints
      Point auxiliaryPoint = new Point((Point)convexList2.lastElement());
      convexList2.setSize(1);
      convexList2.add(auxiliaryPoint);
    }//else
  }//while

  while(mustBeReduced(convexList1,convexList2)==2)
  {//elements from the convex list 2 have to be removed for fixing the subpolygon
    if(convexList2.size() > 1) convexList2.remove(convexList2.size()-1);
    else
    {//elements cannot be removed from convex list 2 anymore
      //remove every element from the convex list 1 but it endpoints
      Point auxiliarPoint = new Point((Point)convexList1.lastElement());
      convexList1.setSize(1);
      convexList1.add(auxiliarPoint);
    }//else
  }//while

  if(mustBeReduced(convexList1,convexList2)==3) return false; //an error happened
  else return true; //no error...success
}

/**************************************************************************
* Checks if some of the given convex lists need to be reduced for them to form
* a convex subpolygon together.
* <p>
* It returns 0 if no convex list needs to be reduced for them to form a
* convex subpolygon. The value 1 is returned if the first convex list needs to
* be reduced and the number 2 is returned if the second is the one that needs
* it. The number 3 is returned in case of an error, then it is not known which
* convex list needs to be reduced.
* <p>
* It check such situation by checking the last angles formed by each convex
* list and an imaginary segment that links their last points. If the internal
* angle formed with a convex list is obtuse, the other convex list must be
* reduced. Special cares are needed for dealing with unitary convex lists.
*
* @param convexList1  the first convex list to be checked.
```

```
* @param convexList2  the second convex list to be checked.
* @return an integer indicating which convex list needs to be reduced
*****************************************************************************/

private int mustBeReduced(Vector convexList1, Vector convexList2)
{
 if(convexList1.size()==1 && convexList2.size()==1) return 3;

 //segments (from each list) to be checked
 Segment analysisSegment1=new Segment();Segment analysisSegment2=new Segment();
 //the boundary segment between this subpolygon and the next one
 Segment lastinternalSegment = new Segment((Point)convexList1.lastElement(),
                              (Point)convexList2.lastElement());

 if(convexList1.size()==1)
 {//the convex list 1 is unitary so must not be reduced
  analysisSegment2.equalize((Point)convexList2.lastElement(),
                       (Point)convexList2.get(convexList2.size()-2));
  if(!((Point)convexList1.firstElement()).equals(
                            (Point)convexList2.firstElement(),false))
      analysisSegment1.equalize((Point)convexList2.firstElement(),
                            (Point)convexList1.firstElement());
  else //both convex lists have the origin as their first point
    analysisSegment1.equalize((Point)convexList2.get(1),
                         (Point)convexList2.get(0));
  //if the polygon is not convex the convex List 2 must be reduced
  if(!analysisSegment1.sameTurn(lastinternalSegment,
                lastinternalSegment.turnSign(analysisSegment2))) return 2;
 }//if

 if (convexList2.size()==1)
 {//the convex list 2 is unitary so must not be reduced
  analysisSegment1.equalize((Point)convexList1.get(convexList1.size()-2),
                       (Point)convexList1.lastElement());
  if(!((Point)convexList1.firstElement()).equals(
                            (Point)convexList2.firstElement(),false))
      analysisSegment2.equalize((Point)convexList2.firstElement(),
                            (Point)convexList1.firstElement());
  else //both convex lists have the origin as their first point
    analysisSegment2.equalize((Point)convexList1.get(0),
                         (Point)convexList1.get(1));
  //if the polygon is not convex the convex List 1 must be reduced
  if(!analysisSegment1.sameTurn(lastinternalSegment,
                lastinternalSegment.turnSign(analysisSegment2))) return 1;
 }//if

 if(convexList1.size()!=1 && convexList2.size()!=1)
 {//no convex list is unitary
  //computing the side that the convex polygon will turn to
  analysisSegment1.equalize((Point)convexList1.get(0),(Point)convexList1.get(1));
  if(!((Point)convexList1.firstElement()).equals(
                            (Point)convexList2.firstElement(),true))
    analysisSegment2.equalize((Point)convexList2.firstElement(),
                         (Point)convexList1.firstElement());
  else //both convex lists have the origin as their first point
    analysisSegment2.equalize((Point)convexList2.get(1),
```

```
                    (Point)convexList2.get(0));
   int turnSide = analysisSegment2.turnSign(analysisSegment1);
   if(turnSide==0)//handling exceptions
   {//there is no turning...try this method again without the first vertices
    Vector subConvexList1= new Vector(); Vector subConvexList2= new Vector();
    subConvexList1 = (Vector)convexList1.clone();
    subConvexList2 = (Vector)convexList2.clone();
    if(convexList1.size()>1) subConvexList1.remove(0);
    if(convexList2.size()>1) subConvexList2.remove(0);
    return mustBeReduced(subConvexList1,subConvexList2);
   }
   //testing if some convex list causes concavity and need to be reduced
   analysisSegment1.equalize((Point)convexList1.get(convexList1.size()-2),
                    (Point)convexList1.lastElement());
   analysisSegment2.equalize((Point)convexList2.lastElement(),
                    (Point)convexList2.get(convexList2.size()-2));
   if(!analysisSegment1.sameTurn(lastinternalSegment,turnSide)) return 2;
   if(!lastinternalSegment.sameTurn(analysisSegment2,turnSide)) return 1;
  }//if

  return 0;
 }//method

 /**************************************************************************
 * Computes the average of a convex subpolygon formed by two given convex lists.
 * <p>
 * It returns a vector with a list of vertices resulting from the average of the
 * formed convex subpolygon.
 * <p>
 * Average points for the resulting vector are obtained from point averages of
 * pair of points (one from each convex list). These points are chosen by
 * advancing from the first to the last point from each convex list and these
 * averages are gotten in each advance. The choice of in each convex list to
 * advance is decided by formed angles by two segments: the segment between
 * the present points and the next segment of each list. The chosen convex list
 * to advance in is the one that forms a more acute angle (once they didn't
 * reach the end yet). When both lists reach the end, the average finishes.
 * <p>
 * This method also handles special situations.
 *
 * @param convexList1  the first list of vertices that forms a convex subpolygon
 * @param convexList2  the second list of vertices that forms a convex subpolygon
 * @return the vector containing the resulting vertex list
 **************************************************************************/

 private Vector convexSubpolygonAveraging(Vector convexList1, Vector convexList2)
 {
  Vector averageFragment = new Vector(1);//the resulting vector to be returned
  Point averageVertex = new Point();
  //the next segment of a particular convex list to be analyzed
  Segment analysisSegment = new Segment();
  //sgmt forming angles with analysis sgmts to define in which cvx list to advance
  Segment referenceSegment = new Segment();//sgmt between the last chosen vertices
  double angle1, angle2;//angles formed by next sgmts of each cvx list and ref. sgmt
  //last chosen vertices of each convex list
  Point turnVertex1 = new Point();
```

```
turnVertex1.equalize((Point)convexList1.get(0));
Point turnVertex2 = new Point();
turnVertex2.equalize((Point)convexList2.get(0));
//indices from the convex lists where to find the turn vertices
int turnIndex1 = 0; int turnIndex2 = 0;

//computing the average for the first vertices (subpolygon origin)
averageVertex.equalize(turnVertex1.
            weightedAverage(turnVertex2,this.weight1,this.weight2));
averageFragment.add(new Point(averageVertex));

//dealing with possible initial special situations
if(turnVertex1.equals(turnVertex2,false))
{//the first vertex of both convex lists is the polygon origin point
 if(convexList1.size() == 1 || convexList2.size() == 1)
 {//some convex list only have the origin point (is unitary)
  if(convexList1.size() == 1)
  {//the convex list 1 only have the origin point
   turnVertex2.equalize((Point)convexList2.get(1));
   turnIndex2 = 1;
  }
  else
  {//the convex list 2 only have the origin point
   turnVertex1.equalize((Point)convexList1.get(1));
   turnIndex1 = 1;
  }//else
 }//if
 else
 {//no convex list only has the origin point
  turnVertex1.equalize((Point)convexList1.get(1));
  turnVertex2.equalize((Point)convexList2.get(1));
  turnIndex1 = 1; turnIndex2 = 1;
  averageVertex.equalize(
     turnVertex1.weightedAverage(turnVertex2,this.weight1,this.weight2));
  if(!averageVertex.equals(originVertex,true))
            averageFragment.add(new Point(averageVertex));
 }//else
}//if

while(turnIndex1 < convexList1.size()-1 || turnIndex2 < convexList2.size()-1)
{//some convex list didn't reach its end yet...the average continues
 //advance with some vertex
 if(turnIndex1 < convexList1.size()-1 && turnIndex2 < convexList2.size()-1)
 {//no convex list reached its end yet
  referenceSegment.equalize(turnVertex1,turnVertex2);
  analysisSegment.equalize(turnVertex1,(Point)convexList1.get(turnIndex1+1));
  angle1 = referenceSegment.angle(analysisSegment);
  referenceSegment.equalize(turnVertex2,turnVertex1);
  analysisSegment.equalize(turnVertex2,(Point)convexList2.get(turnIndex2+1));
  angle2 = referenceSegment.angle(analysisSegment);
  if(angle1 < angle2)
  {//advance in the convex list 1
   turnIndex1++;
   turnVertex1.equalize((Point)convexList1.get(turnIndex1));
  }
  else
```

```
      {//advance in the convex list 2
       turnIndex2++;
       turnVertex2.equalize((Point)convexList2.get(turnIndex2));
      }
     }
     else
     {//some convex list already reached its end
      if(turnIndex1 == convexList1.size()-1)
      {//the convex list 1 already reached its end... advance in the cvx list 2
       turnIndex2++;
       turnVertex2.equalize((Point)convexList2.get(turnIndex2));
      }
      else
      {//the convex list 2 already reached its end... advance in the cvx list 1
       turnIndex1++;
       turnVertex1.equalize((Point)convexList1.get(turnIndex1));
      }
     }

     //computes the average for the present turn vertices and add it to the vector
     averageVertex.equalize(
          turnVertex1.weightedAverage(turnVertex2,this.weight1,this.weight2));
     averageFragment.add(new Point(averageVertex));
    }//while (the subpolygon average)

    return averageFragment;
   }//method

}//class
```

APPENDIX G – THE *PATHFINDER* CLASS

```
package path;

import java.io.*;
import java.util.*;

/**
* The PathFinder class displays a map resulting from the selection and modification
* of path files found in a determined file directory.
* <p>
* The paths contained in these files follow five main steps:
* <p>
* - CLEANING: Eliminates fluctuation vertices, avoids self-intersections and immediate
* returns in the same path. Paths are split whether necessary to avoid such inconvenients.
* <p>
* - SIMILARITY DETECTION: Creates a graph whose nodes represent the cleaned paths
* and the node links represent similarity between them. The applied similarity
* detection is just for whole paths (not for path fragments).
* <p>
* - PATH AVERAGING: Gets the similar paths according to the created graph and
* builds one averaged path for each set of linked nodes.
* <p>
* - FRAGMENT AVERAGING: Checks all pairs of resulting averaged paths for finding out
* fragment similarities. It Averages found similar fragments, replacing their paths by
* averaging results.
* <p>
* - GUI: Adds the final paths to a map and displays it.
*/

public class PathFinder
{
  /**
  * The map displayer.
  */
  MapFrame map;
  /**
  * The directory where to look for path files.
  */
  File pathDir;

 /*************************************************************************
 * Constructs an PathFinder instance creating a displayer for its resulting map.
 *************************************************************************/

  public PathFinder(String directory)
  {
   this.map = new MapFrame();
   this.pathDir = new File(directory);
  }

  /**
  * Main program for the PathFinder project.
  * @author Irving Antunes de Cerqueira Luz
```

```
 * @version 1.0
 *
 * @param args  arguments
 */

 public static void main(String[] args)
 {
  //initializing necessary variables
  PathFinder PF = new PathFinder("c://paths");//instance of this class
  Vector auxiliarList = new Vector();//only for help
  //list containing all cleaned paths (but not averaged yet)
  Vector allPaths = new Vector(1);
  //the chosen candidate paths for being displayed
  Vector pathList = new Vector(1);
  int i, j, k;//counters

// CLEANING...
// ********** LOOKING FOR AND CLEANING ORIGINAL PATHS **********

  String[] fileNames = PF.pathDir.list();//list of found path files
  Path p = new Path();//path instance to call the cleaning method
  double length = 0;//for computing path lengths

  for(i = 0; i < fileNames.length; i++)
  {//for all path files
   //get the paths resulting from this file path cleaning
   auxiliarList.clear();
   auxiliarList = p.cleaning(new File(PF.pathDir + "//"+ fileNames[i]),3);
   for(j=0;j<auxiliarList.size();j++)
   {//for each resulting path
    length = 0;//check if it deserves to be candidate for the map (enough length)
    for(k=0; k<((Path)auxiliarList.get(j)).vertexList.size()-1 &&
        length<Const.MPL; k++) length = length +
          ((Point)((Path)auxiliarList.get(j)).vertexList.get(k)).distance(
                (Point)((Path)auxiliarList.get(j)).vertexList.get(k+1));
    if(length>=Const.MPL) //if it deserves then add it
        allPaths.add(auxiliarList.elementAt(j));
   }//for j
  }//for i


// SIMILARITY DETECTION...
//********** CREATING A GRAPH ACCORDING TO SIMILAR PATHS **********

  Vector graph = new Vector(1);//a graph linking similar paths
  Node node;//graph nodes representing paths

  for(i=0; i<allPaths.size(); i++)
  {//for every cleaned path
   //create a node for the path and add it to the graph
   node = new Node((Path)allPaths.get(i));
   graph.add(node);
   for(j=0; j < i; j++)
   {//for every element (node) from the graph
    //check if there is similarity between this new node (path) and other node
    auxiliarList.clear();
```

```
      auxiliarList = ((Path)allPaths.get(i)).pathSimilarity(
                                  ((Path)allPaths.get(j)),Const.TLE);
      if(auxiliarList.size()!=0)
      {//a similarity was found
        //link the nodes
        ((Node)graph.get(i)).links.add(new Integer(j));
        ((Node)graph.get(j)).links.add(new Integer(i));
      }//if
     }//for j
    }//for i


//PATH AVERAGING...
// ************* FINDING THE AVERAGE FOR PAIRS OF SIMILAR PATHS ****************

  for(i=0; i < graph.size(); i++)
  {//for every element from the graph
    if(((Node)graph.get(i)).flag != true)//if it has not been part of some avg. yet
       pathList.add(((Node)graph.get(i)).nodeAveraging(graph));
  }


//FRAGMENT AVERAGING...
// ********** FINDING THE AVERAGE FOR PAIRS OF SIMILAR PATH FRAGMENTS **********

  //vector containing the paths that are ready to be displayed
  Vector averagedVector = new Vector();
  //contains candidate paths being analyzed for making part of the averagedStack
  Vector jobStack = new Vector();
  int index;//index of item from averagedStack being compared to a candidate path
  int priorSize; boolean averageFlag = false;

  averagedVector.add((Path)pathList.lastElement());
  pathList.remove(pathList.lastElement());
  while(pathList.size()>0)
  {
    jobStack.add((Path)pathList.lastElement());
    pathList.remove(pathList.lastElement());
    index = averagedVector.size()-1;//averagedVector index updated to its end
    while(index >= 0)
    {//revisar todo elemento de averagedVector
      priorSize = jobStack.size();
      for(i=jobStack.size()-1; i >= 0; i--)
      {//for each element of the jobStack
        ((Path)jobStack.get(i)).fragmentAveraging(
                              ((Path)averagedVector.get(index)),jobStack);
        if(priorSize != jobStack.size())
        {//a fragment average happened
          jobStack.remove(i);
          priorSize = jobStack.size();
          averageFlag = true;
        }
      }
      if(averageFlag) averagedVector.remove(index);
      averageFlag = false;
      index--;
```

```
    }
    for(k=0; k < jobStack.size(); k++) averagedVector.add(jobStack.get(k));
    jobStack.clear();
   }
  pathList = averagedVector; //new path list gets only averaged paths to be dispalyed


//GUI...
// ********************* DISPLAYING THE AVERAGED PATHS ************************

    if(pathList.size()>0)
    {//there are paths to be displayed
      //finding a point to be the map center
      int firstX=(int)((Point)((Path)pathList.get(0)).vertexList.firstElement()).x;
      int lastX=(int)((Point)((Path)pathList.get(0)).vertexList.lastElement()).x;
      int firstY=(int)((Point)((Path)pathList.get(0)).vertexList.firstElement()).y;
      int lastY=(int)((Point)((Path)pathList.get(0)).vertexList.lastElement()).y;
      if(firstX < lastX) PF.map.panel.leftRight = firstX-50;
      else PF.map.panel.leftRight = lastX-50;
      if(firstY > lastY) PF.map.panel.upDown = firstY+50;
      else PF.map.panel.upDown = lastY+50;
      //translating the selected paths to the map
      for(i=0;i<pathList.size();i++)
            PF.map.panel.pathList.add(pathList.elementAt(i));
    }
    PF.map.repaint();//displays the map
  }//method
```

APPENDIX H – THE *KMAPFRAME* AND THE *MAPPANEL* CLASSES

```
package path;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Vector;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**********************************************************************
* The class of the frame for displaying the map.
* <p>
* An extension of the {@link JFrame} class.
**********************************************************************/

public class MapFrame extends JFrame
{
 /**
  * The panel for the map
  */
 public MapPanel panel;

 /**
  * Constructs a Frame for the map with the screen size and a correct termination,
  * ending the program when this frame is closed.
  */

 public MapFrame()
 {
  JFrame frame = new JFrame();
  this.panel = new MapPanel(Color.white);
  //dimentioning the frame
  setSize(1025,740);
  Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  Dimension frameSize = frame.getSize();
  if (frameSize.height > screenSize.height) frameSize.height = screenSize.height;
  if (frameSize.width > screenSize.width) frameSize.width = screenSize.width;
  //correct termination
  addWindowListener(new WindowAdapter()
                {
                 public void windowClosing(WindowEvent e)
                 {
                  System.exit(0);
                 }
                });
```

```
   setTitle("PathFinder Map");
   //makes the frame/panel entirely visible and add listeners to it
   setLocation(0,0);
   this.getContentPane().add(panel,"Center");
   setVisible(true);
   JButton b = new JButton();
   this.panel.setFocusable(true);
   MListener mL = new MListener(this); KListener kL = new KListener(this);
   this.panel.addMouseListener(mL); this.panel.addKeyListener(kL);
   this.panel.setFocusTraversalKeysEnabled(false);
  }//method
}//class

/**********************************************************************
* The class of the panel where the map will be drawn.
* <p>
* An extension of the {@link JPanel} class
**********************************************************************/

class MapPanel extends JPanel
{
 /**
  * The vector containing all the paths to be drawn in the map panel.
  */
 public Vector pathList;

 public double upDown = 0;
 public double leftRight = 0;
 public double mapSize = 2; //The size of paths in this map panel

 /**********************************************************************
 * Constructs a map panel with a given background color.
 * <p>
 * Also initializes the vector to be loaded with paths for the map.
 *
 * @param color the map background color
 **********************************************************************/

 public MapPanel(Color color)
 {
  this.pathList = new Vector(1);
  setBackground(color);
 }

 /**********************************************************************
 * Redraws the panel.
 * <p>
 * Automatically called by the runtime system when the panel need to be redrawn.
 *
 * @param g  Object that connects the Java drawing commands to the actual drawing
 * mechanism of the current computer.
 **********************************************************************/

 public void paintComponent(Graphics g)
 {
  super.paintComponent(g);
```

```
  addLines(g);
 }

 /**************************************************************************
 * Draws the paths in the map panel.
 * <p>
 * It seeks all the vector of paths of this map panel, drawing
 * path by path, segment by segment.
 *
 * @param g  Object that connects the Java drawing commands to the actual drawing
 * mechanism of the current computer.
 **************************************************************************/

 public void addLines(Graphics g)
 {
  int i, j=0; //counters
  double x1 = 0, y1 = 0, x2 = 0, y2 = 0; //x and y coordinates for the start and
                                //endpoints of each path segment to be drawn
  g.setColor(Color.red);
  for(i=0; i < this.pathList.size(); i++)
  {//for each path of the vector containing all the paths to be drawn
   for(j=0; j+1 < ((Path)this.pathList.get(i)).vertexList.size() ;j++)
   {//for each segment (two consecutive points) of a path
    //defines the startpoint of the segment
    x1 = ((Point)((Path)this.pathList.get(i)).vertexList.get(j)).x;
    y1 = ((Point)((Path)this.pathList.get(i)).vertexList.get(j)).y;
    //defines the endpoint of the segment
    x2 = ((Point)((Path)this.pathList.get(i)).vertexList.get(j+1)).x;
    y2 = ((Point)((Path)this.pathList.get(i)).vertexList.get(j+1)).y;
    //reposition the segment to a pattern position
    x1 = (x1 - leftRight)/mapSize; y1 = (-y1 + upDown)/mapSize;
    x2 = (x2 - leftRight)/mapSize; y2 = (-y2 + upDown)/mapSize;
    //draw the segment line
    g.setColor(Color.blue);
    g.drawLine((int)x1,(int)+y1,(int)x2,(int)y2);
    //draw the segment startpoint
    //if(j==0) g.setColor(Color.green);//it is the first startpoint
    //else g.setColor(Color.red); //it is not the first startpoint
    //g.fillOval((int)x1,(int)y1,3,3);
   }//for j
   //draw the endpoint of the last segment (the last path point)
   //g.setColor(Color.red);
   //g.fillOval((int)x2,(int)y2,3,3);
  }//for i
 }//method
}//class
```

APPENDIX I – THE *MLISTENER* CLASS

```
package path;

import java.awt.event.ActionEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

/*****************************************************************************
* This class permits the key listener to listen the map window activating the
* focus on the class frame.
*****************************************************************************/

class MListener implements MouseListener
{
 /**
 * The frame where to listen the mouse
 */
 MapFrame mf;

 /**
 * Constructs a mouse listener on the given frame
 * @param frame
 */
 public MListener(MapFrame frame)
 {
  this.mf = frame;
 }

 /**
 * Invoked when the mouse button has been clicked (pressed and released) on a component.
 * <p>
 *
 * @param e  the MouseEvent object passed to the MouseListener
 */
 public void mouseClicked(MouseEvent e)
 {
  this.mf.panel.requestFocusInWindow();
 }

 /**
 * Invoked when a mouse button has been released on a component.
 *
 *  @param e  the MouseEvent object passed to the MouseListener
 */
 public void mouseReleased(MouseEvent e){}
 /**
 * Invoked when a mouse button has been pressed on a component.
 *
 *  @param e  the MouseEvent object passed to the MouseListener
 */
 public void mousePressed(MouseEvent e){}
 /**
 * Invoked when the mouse exits a component.
```

```
 *
 *  @param e  the MouseEvent object passed to the MouseListener
 */
 public void mouseExited(MouseEvent e){}
 /**
 * Invoked when the mouse enters a component.
 *
 *  @param e  the MouseEvent object passed to the MouseListener
 */
 public void mouseEntered(MouseEvent e){}
}//class
```

APPENDIX J – THE *KLISTENER* CLASS

```
package path;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

/*****************************************************************************
* This class permits the keyboard to be listened on the map window for displacing
* and zooming it when it is desired.
*****************************************************************************/

class KListener implements KeyListener
{
 /**
 * The frame where to listen the keyboard
 */
 MapFrame mf;

 /*****************************************************************************
 * Constructs a key listener on the given frame
 * @param frame
 *****************************************************************************/
 public KListener(MapFrame frame)
 {
  this.mf = frame;
 }

 /**
 * Invoked when the keys are typed.
 * <p>
 * '+': zoom in the map.
 * '-': zoom out the map.
 * '1': goes to the lower left direction in the map.
 * '2': goes down in the map.
 * '3': goes to the lower right direction in the map.
 * '4': goes to the left in the map.
 * '6': goes to the right in the map.
 * '7': goes to the upper left direction in the map.
 * '8': goes up in the map.
 * '9': goes to the upper right direction in the map.
 *
 * @param e  the KeyEvent object passed to the KeyListener
 */

 public void keyTyped(KeyEvent e)
 {
  char c = e.getKeyChar();
  if(c == '+')
  {
   if(this.mf.panel.mapSize > 0.21) this.mf.panel.mapSize = this.mf.panel.mapSize - 0.2;
  }
  if(c == '-') this.mf.panel.mapSize = this.mf.panel.mapSize + 0.2;
  if(c == '1') {this.mf.panel.leftRight = this.mf.panel.leftRight - 20;
```

```
            this.mf.panel.upDown = this.mf.panel.upDown - 20;}
  if(c == '2') this.mf.panel.upDown = this.mf.panel.upDown - 20;
  if(c == '3') {this.mf.panel.leftRight = this.mf.panel.leftRight + 20;
            this.mf.panel.upDown = this.mf.panel.upDown - 20;}
  if(c == '4') this.mf.panel.leftRight = this.mf.panel.leftRight - 20;
  if(c == '6') this.mf.panel.leftRight = this.mf.panel.leftRight + 20;
  if(c == '7') {this.mf.panel.leftRight = this.mf.panel.leftRight - 20;
            this.mf.panel.upDown = this.mf.panel.upDown + 20;}
  if(c == '8') this.mf.panel.upDown = this.mf.panel.upDown + 20;
  if(c == '9') {this.mf.panel.leftRight = this.mf.panel.leftRight + 20;
            this.mf.panel.upDown = this.mf.panel.upDown + 20;}

  this.mf.repaint();//repaint the map after it is displaced
 }//method

 /**
 * Invoked when the keys are pressed.
 *
 * @param e  the KeyEvent object passed to the KeyListener
 */
 public void keyPressed(KeyEvent e){}

 /**
 * Invoked when the keys are released.
 *
 * @param e  the KeyEvent object passed to the KeyListener
 */
 public void keyReleased(KeyEvent e){}
}//class
```