Masters in Computer Vision

Software Engineering Project

2013 - 2014

Oksana Hagen

Natalia Shepeleva

Emre Ozan Alkan

Klemen Istenič

January 2014

Contents

List of Figures

Chapter 1

Introduction

Every person in his life at least once leaves his home city for any reason, whether it is a business trip or just a vacation. Taking into account that modern technologies nowadays allow people to move from one point to another in a few hours, such trips becomes much easier to perform. Moreover it gives an opportunity to investigate almost every point on the globe.

That is why maps become indispensable attribute of any trip. There are two main categories of maps: paper and electronic maps. Classical paper maps can help in any situation but in comparison with electronic map they have one big disadvantage – they are not interactive. Electronic maps in its turn are also separated on "online" and "offline" maps. "Online" maps are interactive, fast and usually not limited to one city. Furthermore there is more than one "online" map, so user can use any on his taste.

But what to do if for any reason there is no internet connection? For this case "offline" maps can be used, especially if person appears in unknown city. And the main problem of any map that covers large area in this case can be absence of any important information like presence of grocery store or bakery or laundry for small cities. That is why implementation of maps for small areas and cities is very important, because it contains specific information about this particular region.

So the goal of our project was implementation of the map for city Le Creusot with basic set of data and functions like finding way between two points or showing main public facilities. It can be useful for any person whether it's a tourist, visitor on a business trip or a student. Especially we hope that this map can be useful for next *ViBOT/MsCV* promotions for simplifying their life in Le Creusot.

Chapter 2

Project management

In this chapter we want to present how our group organized and managed this project. Group members are:

- Ozan
- Oksana
- Klemen
- Natalia

As we all had a bit of experiences in software design, we knew how important a proper plan and research before the actual implementation is. Unfortunately, we were also aware that no matter how good the plan is, we will be forced to adjust it during the implementation. Either because of the things we overlooked while planning or because some assumptions we made were wrong. We decided to follow the *iterative and incremental development model*, which enabled us to adjust our plans after each of the implementation iterations. A schematic representation of the iterative and incremental development model can be seen on figure 2.1.

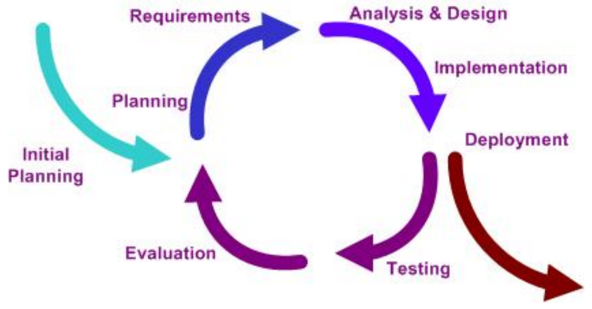

FIGURE 2.1: An iterative development model (image taken from [1])

2.1 Basic building blocks

On our first meeting we decided to split the project into four main parts to be able to fully manage the whole project at all times. This parts are:

- User interface
- Database and data structure
- Map representation
- Path algorithms

2.1.1 UI - User Interface

Part of the application that is responsible for user interaction with the application. Main goal is to make the interface as user-friendly as possible. In the beginning we made a rough sketch of how the interface should look (figure []), which has, as expected, changed during the process of later design and implementation. The UI is presented more in details in chapter [UI USER MAUNAL].

2.1.2 Database and data structure

Our database consists of two parts. The static part, having the information about the roads in Le Creusot (described in []) and the dynamic part with the information about the points of interest (POI). In chapter[] we address the facts we took into account while considering different database options and data structures.

2.1.3 Map representation

We separated map representation from other parts of user interface, because we think it is the most important of UI, so we will give special attention to it.

2.1.4 Path algorithms

In the different algorithms we adopted and developed, we do all the searching for paths, calculating distance and travel times and optimize the search results as much as possible. Our main goal is to make the algorithms work as quickly as possible, taking into account all the restrictions road networks has (oneway streets, footways etc.). We also want to make the algorithms as reusable as possible, to reduce the redundancy in development and minimize the possibilities of errors.

2.2 Softwares used for project management

Github, skype etc.

2.3 Meetings

some bullshit about that

Chapter 3

Initial planning

3.1 User requirement analysis

As mentioned in the previous chapter, we decided to use *the iterative and incremental development model*. Before starting the iterations we spend quite a lot of time on the initial planning, with special attention to user and general project requirement analysis. From the project's instructions we were able to identify the following major user requirements and construct use case diagram (figure 3.1):

- User should be able to enter start and end point either by:
  - mouse click;
  - specifying latitude and longitude;
  - selecting a point of interest.
- Find shortest path from point A to B (by foot or by car);
- Find the shortest way to all POIs of a certain category in a radius from point A (by foot or by car);
- Construct an itinerary from point A to B with visiting POIs in between (with max distance limit);
- View, edit and add POIs

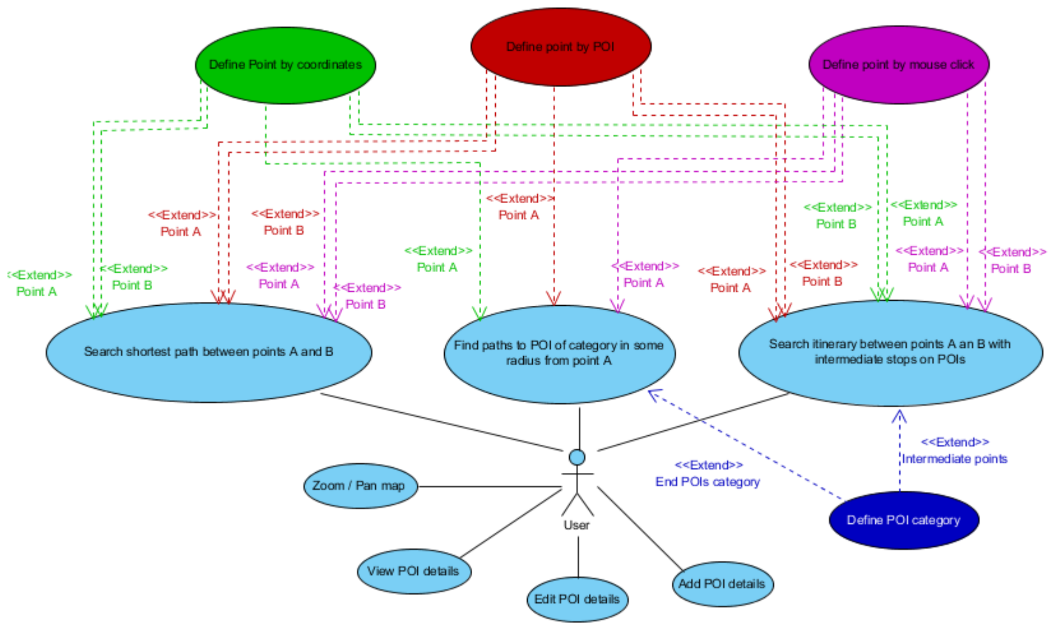
Define Point by coordinates
Define point by POI
Define point by mouse click
<<Extend>> Point A
<<Extend>> Point B
Search shortest path between points A and B
Find paths to POI of category in some radius from point A
Search itinerary between points A an B with intermediate stops on POIs
<<Extend>> Intermediate points
<<Extend>> End POIs category
Define POI category
User
Zoom / Pan map
View POI details
Edit POI details
Add POI details

FIGURE 3.1: Use case diagram

3.2 Plan of implementation

After having the user requirements, we decided to make the global plan of implementation. We took into account the fact that we have to develop the applications in **C++** and **Matlab**. As described in section 2.1, we have already split the project into four main parts. Because the user interface and map representations require completely different approaches in C++ and Matlab, we decided to split them into two parallel projects, one almost independent from another. On the other hand, we decided to use the same algorithms for both projects. The main reason is the minimization of the redundancies in the research and development stage and reducing the possibility of errors. This way we could have the same algorithms for both, differing only in pure implementation details.

Each of was delegated to oversee one of the parts (as shown in figure []). We really want to emphasis that this does not mean each of us solely did that part or that is responsible for it. No matter the delegation, we all worked on all parts of the project, either in research, design or implementation stage, so the delegation was just for preparing the material for the meetings.

3.3 MAYBE TIME PLAN?

Chapter 4

Data sources

Our application uses two main types of data. **Static** map data describing the streets and buildings and **dynamic** data about the different points of interest (POIs). The difference between static and dynamic data lies in the users ability to add and edit POI, while the map information is not meant to be changed by the user.

The acquisition of this data was not the essence of the project, so we were allowed to use any data source, with appropriate licence and collaborate with other groups.

4.1 Map data - OpenStreetMap

Acquisition of map data (roads, road types, buildings etc.) can be extremely time consuming and can easily produce unreliable results. At the same time, there are a lot of different open source data sources available on-line. Together with the other groups we decided to use OpenStreetMap data by the OpenStreetMap Foundation ([3]).

OpenStreetMap is an open source project providing free map data of the world. Data is published under the *Open Database License (ODbL) from Open Data Commons (ODC)*, which enables us to freely produce the works from the database, modify, transform and build upon the database.

4.1.1 Data format

OpenStreetMap data consists of four core elements:

- **Nodes** - basic point of location with longitude and latitude information. They can be used to mark a single point (POI) or in a list of nodes (way, relation);

- **Ways** - ordered list of nodes, representing a street or an area like a lake, forest etc.;
- **Relations** - ordered list of nodes and ways which can be in a relation (many different roads can be part of a long motorway);
- **Tags** - key-value pairs which are used to store information about different objects (nodes, ways, relations).

Figure 4.1 shows the difference between different core elements

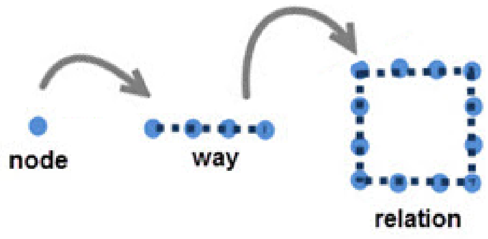

FIGURE 4.1: Representation of basic OSM elements (image taken from [2])

4.2 Points of Interest (POI)

One of the main requirements for our application is to enable user to search using different points of interest. For acquisition of this points, all of the groups again collaborated, each marking all the points in one part of Le Creusot. For each point we decided to acquire:

- name;
- location;
- address;
- photo.

In the end we also categorized all the points into 26 different categories.

Chapter 5

Database and data structure

5.1 Database

As described, our data consists of two parts. *Static* map data and *dynamic* points of interests data. For the beginning, we were deciding between using relational database and XML based database. With each of them having some benefits, we took the following facts intro account during the consideration:

- Most of the data is static - information about the roads in Le Creusot will not change;
- Dynamic data will rarely change - user will not often update information about the points of interest;
- Relatively small amount of data - Le Creusot is a small city, so there is not a lot of information about the roads. This gives us the opportunity to read all the information to memory at the beginning, reducing the latency caused by queries to the database;
- Easiness of install and mobility - using relational database, requires installation of different software (SQL server, connectors, etc.) on the clients computer, which we wanted to avoid.

In the end we decided to use **XML based database**. We kept the two parts of the data split into separate XML files. We kept the OpenStreetMap data in the OSM file and created another XML file for POI data. This way, we could easily change either part of the data without compromising the other.

5.1.1 Disadvantage

Using XML based database has one big disadvantage, which we have to take into account. XML based database is saved into a file. This does not provide the ability to easily update POI data. Unfortunately, every time we update this data, we have to rewrite the whole file. This can in some occasions be the source of problems, as the writing to the file can be disrupted or cancelled, making the file unusable. However, we do not anticipate the user to update the data frequently, so we took this compromise.

5.2 Data structure

We have designed out class structure with the A* search algorithm and OpenStreetMap (OSM) file structure in mind. We followed the OSM structure, having a basic class called **Node**. In figure 5.1 we show the different types of classes we use with a small graphic representation for better understanding.

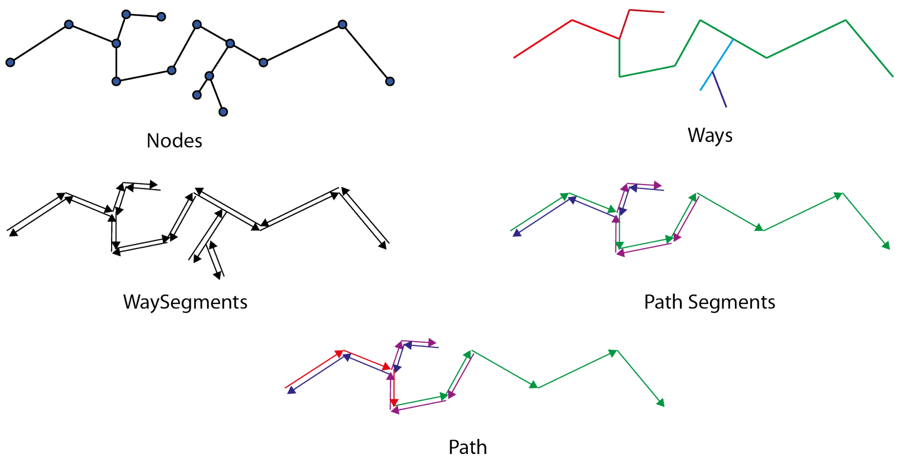

FIGURE 5.1: Different objects in our data structure

5.2.0.1 Node class

It is used to represent a single point on the map. This can either mark a point of interest (POI), or can be a part of a Relation such as *WaySegment, Building or Way*. It contains location information, some A* algorithm informations (weights, visited flags, etc.) and a vector of pointers of WaySegments to which we can traverse from this Node. This is important especially for A* algorithm, as it significantly reduce the time needed to find all possible and subsequently the correct next move.

5.2.0.2 Way class

Class implemented as in OSM file, to store the properties of each of the ways. It us usually used to represent whole streets, or part of the street that have the same properties. Each way is assigned a type (primary, secondary, etc.), direction (one-way, bidirectional) and access privileges (private or public).

5.2.0.3 Relation class

Relation is a base class from which we extend different relation classes (WaySegment, Building). It only contains a vector of pointers to Nodes that are in one specific relation and type of the relation.

5.2.0.4 WaySegment class

WaySegment is the main class for our path algorithms. WaySegment connects two nodes in a specific direction. This means that if the road between two nodes is bidirectional, we will create two WaySegment objects, one for each direction. We also have a pointer to an object Way, which belongs to the road traversing. This gives us the option of getting information about the road at any time.

5.2.0.5 Building class

Is a simple class containing all the nodes representing one building.

5.2.0.6 PathSegment class

It is a class that is used to store the result of a *shortest path search.* It contains all the pointers to WaySegment objects we need to traverse in order to get from point A to B.

5.2.0.7 Path class

Object that contains vector of PathSegment pointers. Path is the end result of any search. If the search is solely path from point A to B, the path is going to have only one PathSegment pointer. On the other hand, if we search for itinerary, Path will include multiple PathSegments, one for each pair of middle points.

5.2.0.8 Database class

Database is an main class for the whole database. It contains functions for correct parsing of XML files and also has containers *(std::map)* with pointers to each of the classes created (Node,Way,Bulding,WaySegment). We use this to quickly retrieve the classes based on their ID, and to be sure to delete all created objects in the destructor of the class. We also used a boost implementation of a *rtree* data structure to hold all the WaySegment objects to be able quickly do a spatial filtering. This is very useful in implementation of finding the close WaySegments.

In figure 5.2 we can see the whole class diagram of the database part of the application in C++.

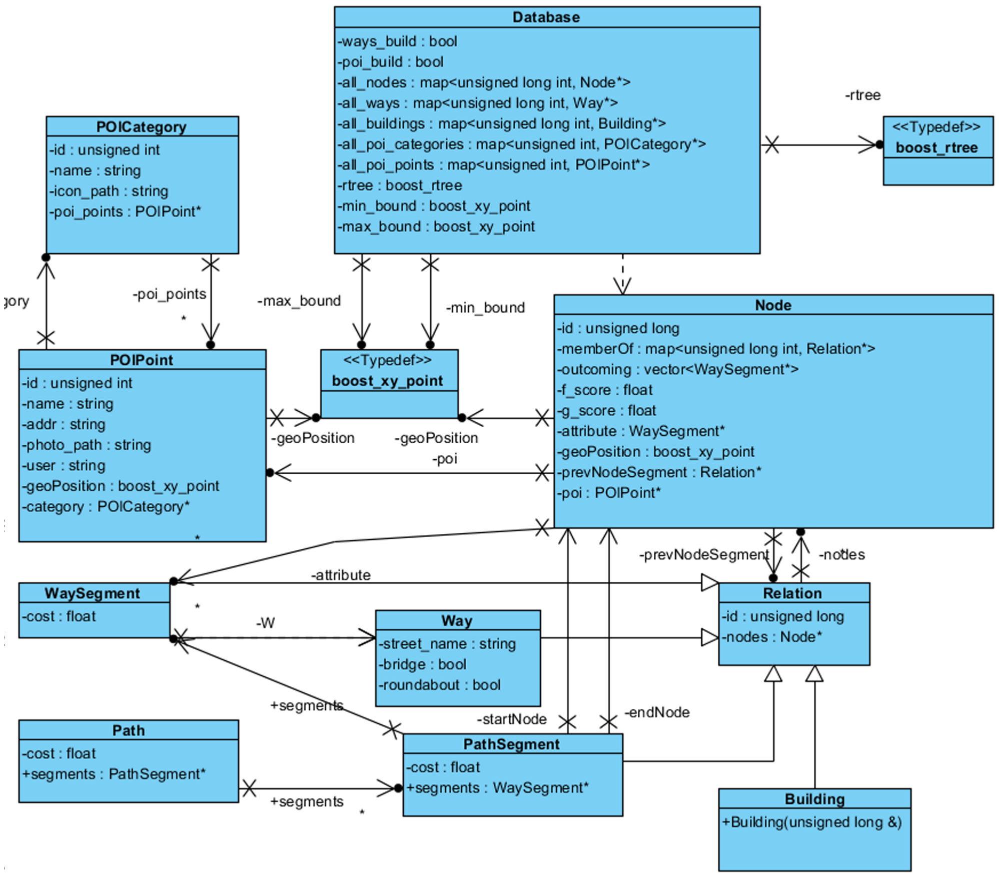

FIGURE 5.2: Class diagram

Chapter 6

Path algorithms

Finding the shortest path is one of the main problems in graph theory. It is a problem of finding a path between two nodes, where the sum of edge weights must be minimized. Graph can be undirected, directed or mixed.

The analogy with road maps can be clearly seen, where nodes correspond to intersections and road segments to edges on the graph. To properly represent one way streets we use directed graph. The weight of each edge can be interpreted as a length of each road segment.

There are numerous algorithms that are able to found the shortest path. After initial research we narrowed the possible algorithms to:

- Dijkstra's algorithm
- A* search algorithm
- Bellman - Ford algorithm
- Floyd - Warshall algorithm

From the user requirement analysis (segment **??**) we were able to identify the three main user requests involving path algorithms:

- Find shortest path from point A to B (by foot or by car);
- Find the shortest way to all POIs of a certain category in a radius from point A (by foot or by car);
- Construct an itinerary from point A to B with visiting POIs in between (with max distance limit);

From identified requirements, we can see that all of our path finding problems are actually problems of finding paths between points A and B. This are so-called **single-pair shortest path problems**. Analysing the algorithms listed above we determined that:

- Floyd - Warshall algorithm finds shortest paths between every pair of nodes in the graph;
- Dijkstra and Bellman - Ford algorithms find the shortest paths between source node and all other nodes on the graph (Bellman - Ford also permits negative weights);
- A* algorithm finds the shortest path between the source and target node.

All of the above algorithms encapsulate the result which we need, but differing in how many unnecessary paths are also calculated. This subsequently mean longer calculation times, which we want to avoid. For this reasons we chose **A* algorithm** for our path finding problems.

6.1 A*

A* algorithm is one of the most popular path finding algorithms. Its ability to combine the benefits of Dijkstra's algorithm (favouring nodes close to the start node) and Best-First-Search algorithms (favouring nodes close to the target node) makes it efficient and accurate.

6.1.1 Process

Algorithm works by traversing the graph node by node till it reaches the end node. At every step, node with the lowest cost ($f(x)$) is selected. Calculation of the cost and selection of the node is what sets A* algorithm apart from other greedy best-first search algorithms. Cost is calculated as a weighted sum of:

- $g(n)-$*exact cost* of the path from the starting point,
- $h(n)-$*heuristic estimated cost.*

Heuristics are used to control the A*'s behaviour. If $h(n) = 0$ only distance from the start node matters and we have normal Dijkstra algorithm. Because we do not know the real distance from observed node to the end target, as we haven't traversed it yet, we have to estimate it. It is important not to overestimate the distance, as this can cause the algorithm not to found the shortest path. We decided to use the air distance as a heuristic, as this guarantees that the actual distance will be equal or greater than that, so we will never over-estimate it.

6.2 Road snapping

As described in the previous section A* algorithm enables us to find the shortest path between two nodes in a graph. This enforces the restriction of searching only on the locations of the nodes. This is a huge limitation, as on some parts of the map users defined point can be located far away from the closest node. To overcome this restriction we implemented **a road snapping function**.

Road snapping function finds the segment on which the perpendicular projection of user's point is closest to the user's point itself. We only consider the segments which are appropriate for a specific mode of transport. Calculation of the perpendicular projection of a point on the segment is done using vectors and dot product between them. Algorithm used is described on website ([4]). It is worth mentioning that if the perpendicular projection of the point is outside of the interval between both nodes, it snaps to the closer border node. This can be seen on figure6.1 (left and right).

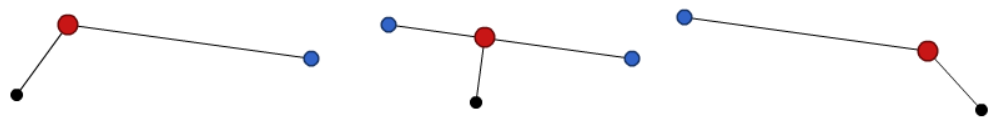

Figure 6.1: Snapping user point to border nodes (left,right) and snapping to perpendicular point where possible

With perpendicular projected point we create a *new mock node* which represents the start point of our search on the road. We connect this new node with the end nodes of the closest segment, making it possible to use it in A* algorithm. We repeat the same steps for the end point. After the search we remove all the nodes and edges we additional added, reverting back the graph to the same state as before the search.

6.2.1 Implementation in C++

Road snapping function requires an extremely time consuming operation of finding close segments. One possible solution would be to actually calculate the perpendicular projections on every WaySegment in the database and then among them find the smallest.

This would mean thousands of unnecessary calculations. Instead we used rtree data structure containing all the WaySegments implemented in *Database class*. With this we were able to quickly find WaySegments whose bounding boxes intersect with our point. To compensate for areas, where roads are closely together, we implemented the search in small increasing steps. We start the with a small neighbourhood and gradually increase it, until we find any intersecting WaySegments. This approach reduces the calculations of perpendicular projections from thousand to just a couple. The diagram of the proces is shown in the figure 6.2

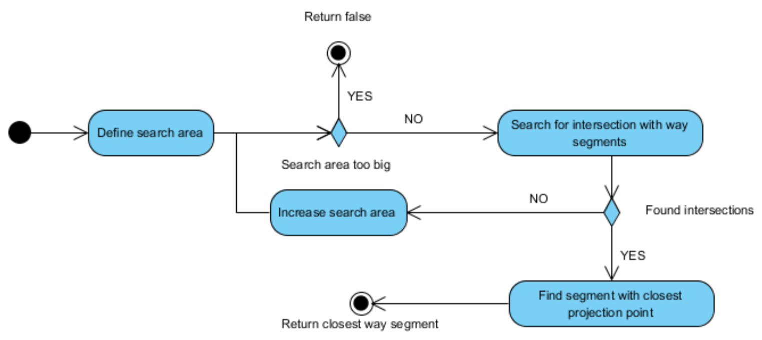
Return false
YES
NO
Define search area
Search area too big
Search for intersection with way segments
NO
Increase search area
Found intersections
YES
Find segment with closest projection point
Return closest way segment

FIGURE 6.2: Diagram of finding the closest WaySegment

6.2.2 Implementation in Matlab

Unfortunately Matlab does not have similar data structures capable of spatial filtering. To get the same end result we calculated the projection points on all the points and find the one with the minimal distance. To speed up the process of calculation we put the data in vectors and vectorize ([5]) the code for calculation. The speed of the calculation turned out to be sufficient.

6.3 Shortest path A $->$ B

This algorithm finds the shortest path between users defined points A and B on the roads that permit mode of transport selected by the user. With the usage of the road snapping functionality we can provide the user complete freedom in selecting the points anywhere on the map. The inputs to the algorithm are:

- Location of the start point
- Location of the end point
- Mode of transport

After finding the closest segments for start and end and adding mock nodes we can use A* algorithm to find the shortest path. The result of the algorithm is an object Path containing one PathSegment corresponding to WaySegments on the way from point A to B. If path does not exist we return an empty Path. The whole process is schematically shown on figure 6.3.

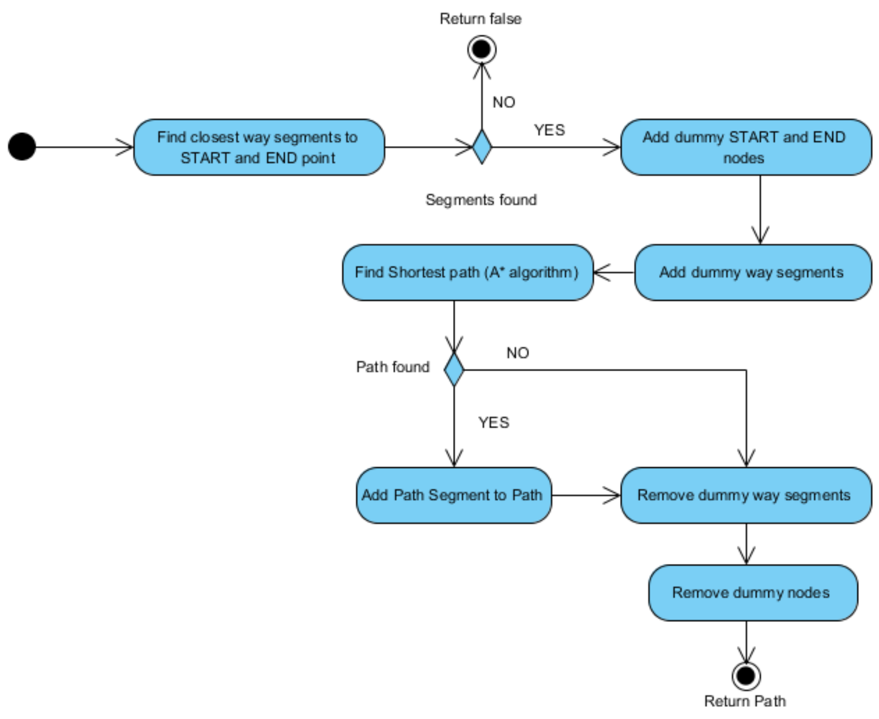
Return false
NO
YES
Find closest way segments to START and END point
Add dummy START and END nodes
Segments found
Find Shortest path (A* algorithm)
Add dummy way segments
Path found
NO
YES
Add Path Segment to Path
Remove dummy way segments
Remove dummy nodes
Return Path

FIGURE 6.3: Activity diagram of finding the shortest path between point A and B

6.3.1 Implementation

The implementation details for both applications (C++ and Matlab) have been already described in specific methods of A* and road snapping. Other things are just sequence of if sentences that do not need special explanation. Figures 6.4 and 6.5 shows the end result of the search as seen by the user.

6.4 Radius search

Radius search algorithm finds all the possible paths from user defined point A to all the points of interest (POIs) of a user specified category in some radius. The inputs to the algorithm are:

- Location of the start point

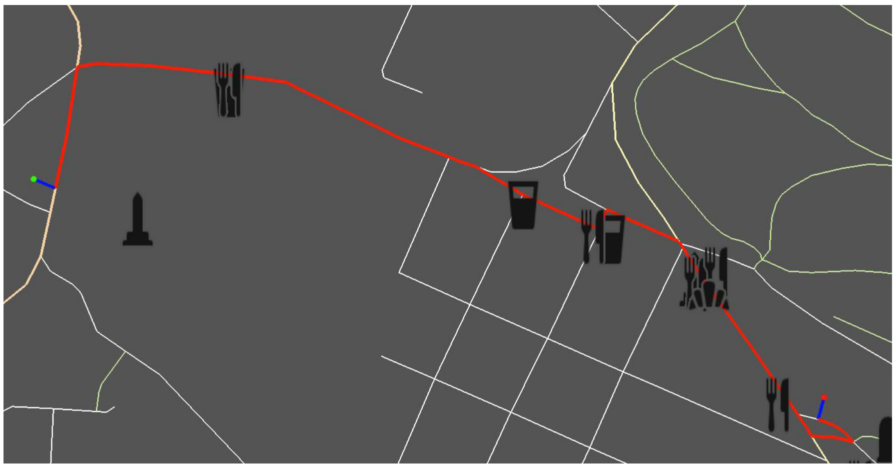

FIGURE 6.4: Result of the search for the shortest path as seen by the user in C++

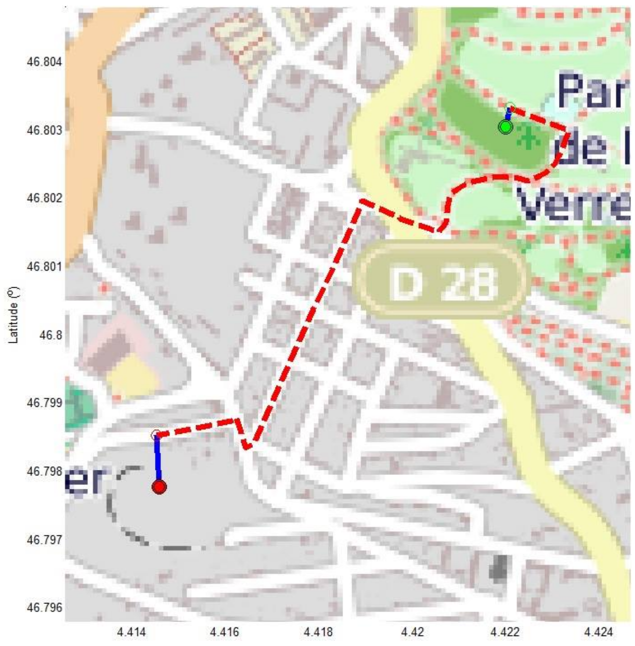

FIGURE 6.5: Result of the search for the shortest path as seen by the user in Matlab

- Category of POIs
- Max distance of the path
- Mode of transport

After finding the closest WaySegment to the start point, we go through all the POIs. For each of them we find the closest WaySegment and check the air distance between start and POI point. If the distance is bigger than the longest permitted path, we immediately discard the POI, because there is shorter way possible than that. Otherwise we try to find the shortest path using **Shortest path A $->$ B algorithm**. The final result of the algorithm is a bit different for both applications. In C++ we return the set of all possible paths ordered by the distance of the path, whereas in Matlab we only return the shortest path. Whole process is shown on figure 6.6.

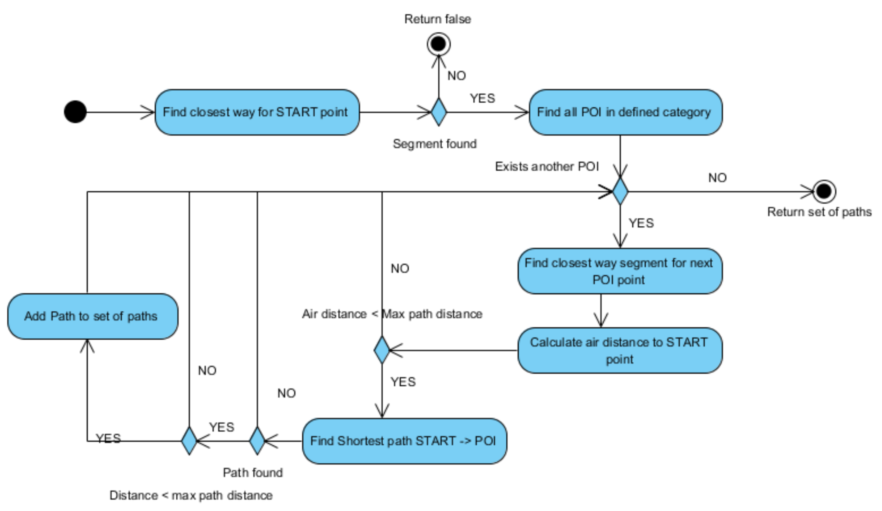
Return false
NO
YES
Find closest way for START point
Segment found
Find all POI in defined category
Exists another POI
NO
Return set of paths
YES
Find closest way segment for next POI point
NO
Add Path to set of paths
Air distance < Max path distance
Calculate air distance to START point
NO
YES
NO
YES
YES
Find Shortest path START -> POI
Path found
Distance < max path distance

FIGURE 6.6: Activity diagram of finding radius search between point A and POIs of specific categories

6.4.1 Implementation in C++

As mentioned, data structure used to store all found paths is *std::set*. We used that because our paths are unique and we want them ordered by distance. For this reason we implemented comparator that takes care of sorting the paths when they are inserted in the set. Figure 6.7 shows the end result of the search as seen by the user.

6.4.2 Implementation in Matlab

In Matlab we implemented the algorithm using vectors and with the usage of vectorization. The calculations are not as efficient as in C++, but the usage of vectorization enables us to have results in sufficient amount of time.

6.5 Itinerary search

Itinerary search algorithm constructs a path, that connects users defined start and end point and go through points of interested of a user defined categories. Algorithm finds a path that goes through one of the POI of each of the categories. The number of the categories in the list (points we want to visit) is not limited by the algorithm, but we

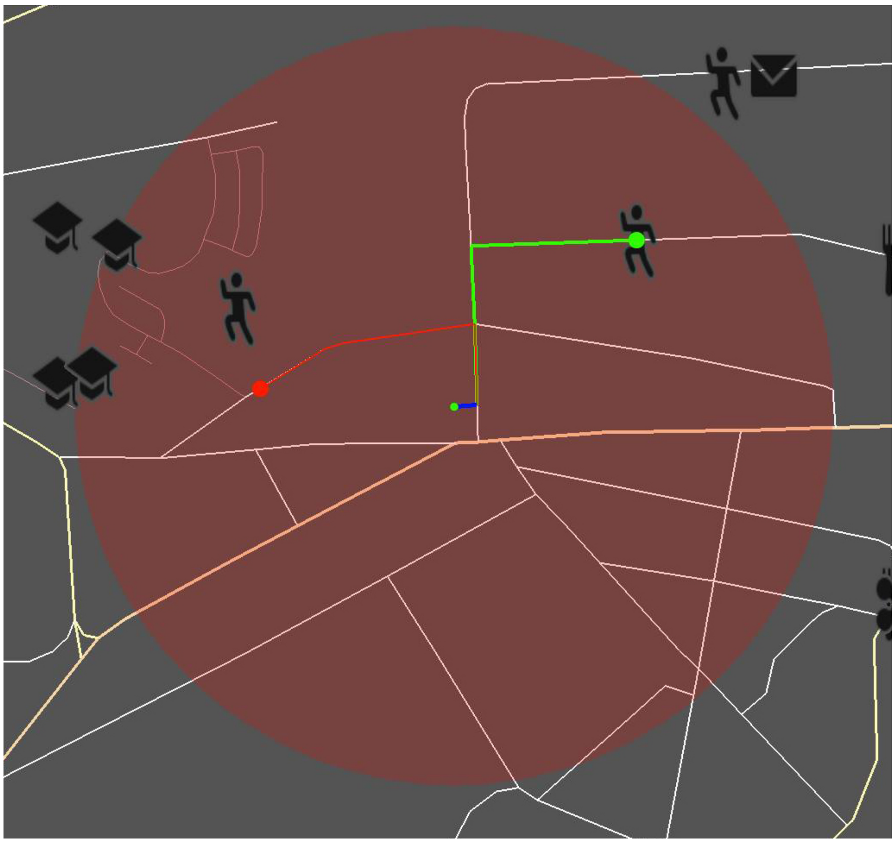

Figure 6.7: Result of the radius search as seen by the user in C++

have limited it in UI, because we have to be aware that the number of combinations increases significantly with every new category. Algorithm's inputs are:

- Location of the start point
- Location of the end point
- List of Categories of POIs we want to visit
- Max distance of the path
- Indication if the order of visited categories is important
- Mode of transport

Before the start of actual path finding, we construct all possible combinations of POIs on the way. As already mentioned, for each category in the list we want to visit just one POI. We add list of points for each of the combinations to a queue which is ordered by ascending order of the sum of air distances of points on the path.

Next phase is to find an actual path. We always look for the actual path of the list of points that have the smallest sum of air distance in the queue (top of the queue). If the air distance of the top element of the queue is bigger than the minimum distance found of the real path, we can stop the search, as we will not find smaller path then the one we already have. This way we do not keep looking for numerous paths which do not have any chance of being smaller, than the minimal path already found. Diagram in figure 6.8] shows the process of the search. The same as in previous algorithms, if no path is found we return an empty path.

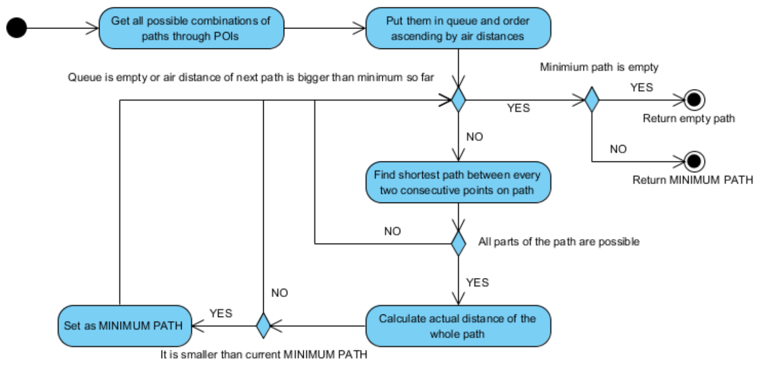
Get all possible combinations of paths through POIs
Put them in queue and order ascending by air distances
Queue is empty or air distance of next path is bigger than minimum so far
Minimium path is empty
YES
YES
Return empty path
NO
NO
Return MINIMUM PATH
Find shortest path between every two consecutive points on path
NO
All parts of the path are possible
YES
NO
YES
Set as MINIMUM PATH
Calculate actual distance of the whole path
It is smaller than current MINIMUM PATH

FIGURE 6.8: Activity diagram of finding shortest path from A to B through POIs of specific categories

6.5.1 Implementation in C++

Described algorithm was implemented in C++, using list data structure, to hold all the possible combinations of POIs and priority queue to sort the paths according the air distances. We used the selected data structures because they provided exactly what we needed. Priority queue because the order of our data is important and the smallest(largest) element goes out first. Figure 6.9 shows the end result of the search as seen by the user.

6.5.2 Implementation in Matlab

In Matlab we have implemented the algorithm with only one middle category. In the calculations we take advantage of Vectorization, which significantly increases the speed of the calculations. Figure 6.10 shows the end result of the search as seen by the user.

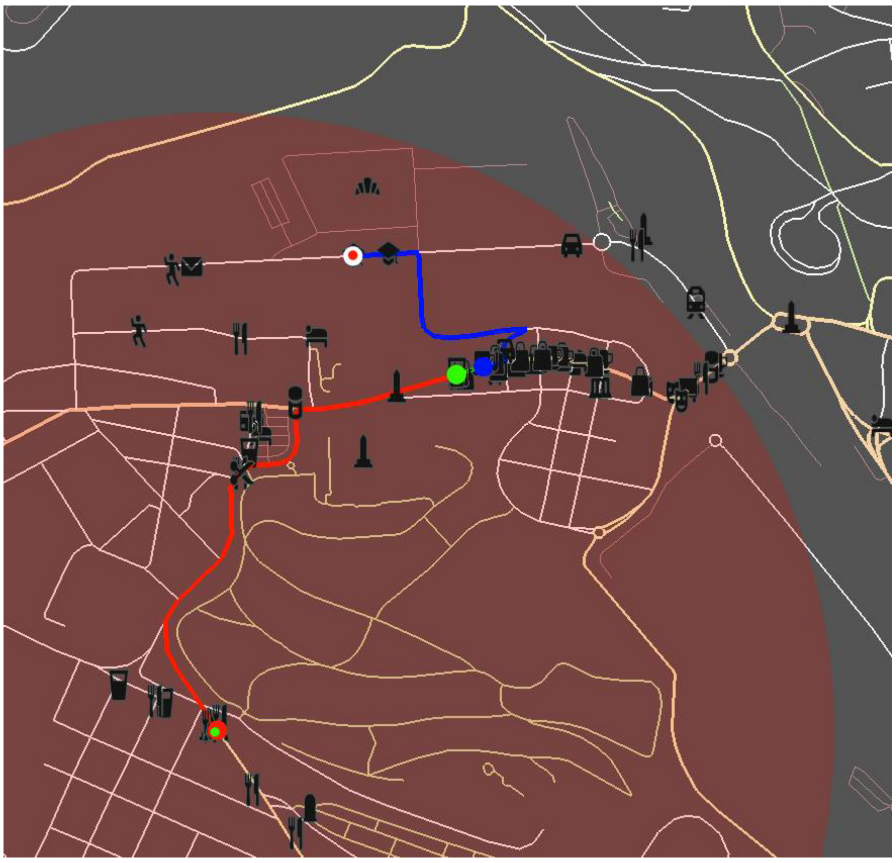

Figure 6.9: Result of the itinerary search as seen by the user in C++

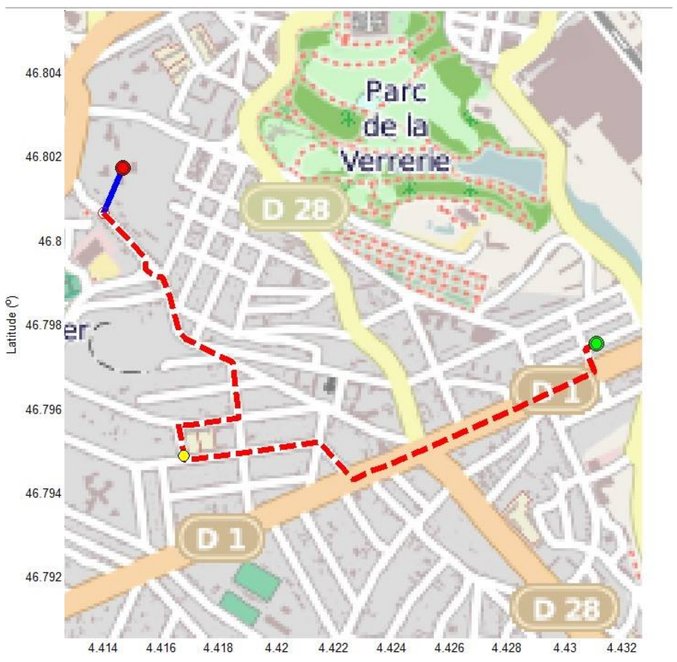

Figure 6.10: Result of the itinerary search as seen by the user in Matlab

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Iterative and incremental development. 2014. URL http://en.wikipedia.org/wiki/Iterative_and_incremental_development.

[2] Openstreetmap data overview. 2014. URL http://wiki.snowflakesoftware.com/display/GLDOC/OpenStreetMap+Data+Overview.

[3] Openstreetmap fundation. 2014. URL http://wiki.osmfoundation.org/wiki/Main_Page.

[4] Notmagi.me. 2014. URL http://notmagi.me/closest-point-on-line-aabb-and-obb-to-point/.

[5] Matlab - vectorization. 2014. URL http://www.mathworks.fr/fr/help/matlab/matlab_prog/vectorization.html.