Masters in Computer Vision

Software Engineering Project

2013-2014

Ozan

Oksana

Klemen

Natalia

January 2014

Contents

CHAPTER 1

Introduction

As a project at Software Engineering class we had to develop an application that to some extent mimics well known Google Earth. The main idea is to give an user who is unfamiliar with Le Creusot the tool that enables him to search for shortest path between two points on the map, enables him to create an itinerary containing different types of points of interest. We had to do all this in C++ and in Matlab.

CHAPTER 2

Project management

In this chapter we want to present how our group organized and managed this project. As we all had a bit of software designing experiences, we knew how important a proper plan and research before the 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, because of the things we did not take into account while planning or because some things turned out to be different to what we assumed. Because of that 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 1.

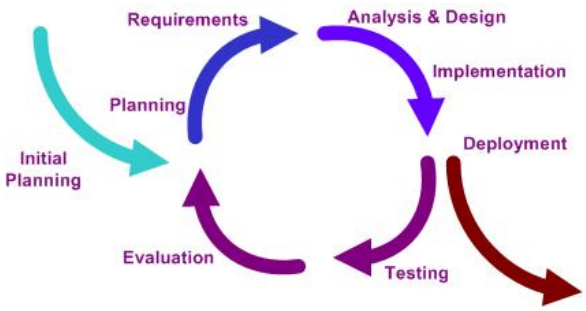
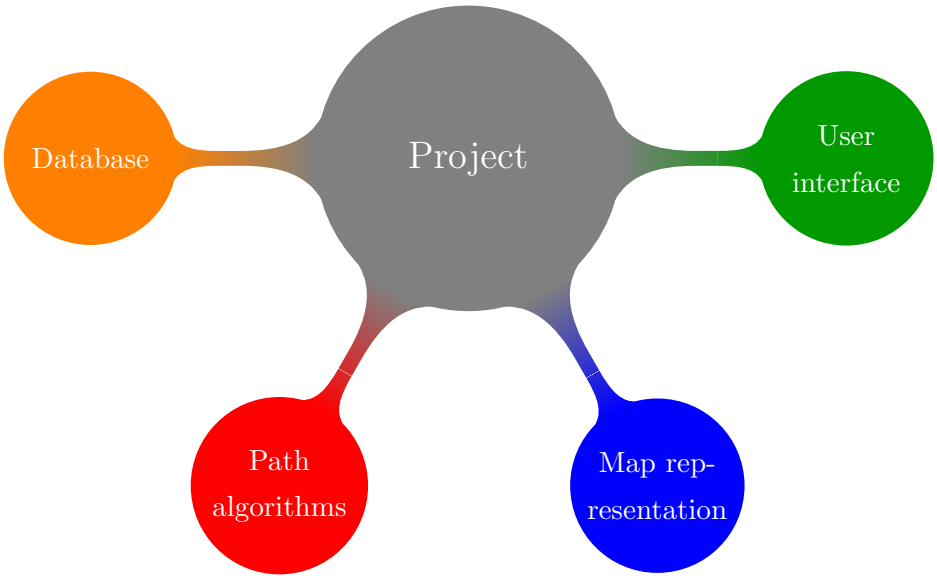


FIGURE 1. An iterative development model

1. Basic building blocks

To be able to fully manage the project at all times we decided to split the project into four main parts:

- User interface
- Database
- Map representation
- Path algorithms



1.1. UI - User Interface.

It is the 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 []), but we, as expected, changed it a little bit during the development. The UI is presented more in details in chapter [UI USER MAUNAL].

1.2. Database.

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. From the beginning we were deciding between using XML based and relational database. During the consideration we took into account the following facts:

- Most of the data is static - information about the roads in Le Creusot will not change;
- Dynamic data will change rarely - 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.

Because of all these reasons we decided to use XML based database. We have the information about the roads in one osm file, while the information about the points of interest is in xml file. We will describe both of them more in detail in later chapters.

1.3. Map representation.

TODO

1.4. Path algorithms.

This part consists of all calculations about paths, distances, travelling times and optimizations of the paths. 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. Softwares used for project management

Github, skype etc.

3. Meetings

some bullshit about that

CHAPTER 3

Initial planning

As already 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 requirements analysis. From the project's instructions we were able to identify the following major user requirements and construct use case diagram:

- 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 point B (by foot or by car)
- find all points of interest in a certain radius from point A (by foot or by car)
- construct an itinerary from point A to point B with points of interest in between (with max distance limit)
- view, edit and add points of interest (POI)

For better

1. Shortest path A -¿ B

2. title

!!!ADD image of the use case diagram

CHAPTER 4

Plan of implementation

After having the user requirements, we decided to make the global plan of implementation. Here we had to take into account, that we have to develop the application in C++ and Matlab. As show in the figure[] we have already splited 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 was to minimize the redundancies in the research and development stage and minimize 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 have to emphasis that this does not mean he/she developed that part by him/herself, not even that he is solely responsible for it. No matter the delegation, we all worked on all parts of the project, either in research, design or implementation stage. TODO: ADD IMAGE OF PARTS WITH DELEGATION

1. MAYBE TIME PLAN?

CHAPTER 5

Data

Our application uses two main types of information. *Static* map data describing the streets 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 POIs data, 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.

1. Map data

Acquisition of map data (street locations, street types, buildings etc.) can be extremely time consuming and can easily produce not reliable results. At the same time, there is a lot of different open source data available on-line. Together with the other groups we decided to use OpenStreetMap data by the OpenStreetMap Foundation[].

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.

1.1. Data format. OpenStreetMap data consists of four core elements:

- Nodes - are basic point of location. They consist of longitude and latitude information. They can be used as a single point (POI) or ia a list of nodes (way);
- Ways - ordered list of nodes, representing a street, an area (lake, forest etc.);
- Relations - ordered list of nodes and ways which can be in a relation (many roads can be included in a long motorway);
- Tags - key-value pairs which are used to store information about different objects (nodes, ways, relations)

Figure 1 shows the difference between them.

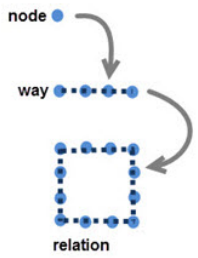


FIGURE 1. An iterative development model

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 aquisition of this points, all of the groups again collaborated, each marking all the points in 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.

3. 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 during the consideration into account the following facts:

- Most of the data is static - information about the roads in Le Creusot will not change;
- Dynamic data will change rarely - 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 od the data splitted into separate xml files. We kept the OpenStreetMap data in the osm files and created another XML file for POI data. This way, we could easily change th () This separation enables us to easily change either the either map data or the POI data.
IMAGE OF DB FILES
Using XML based database produced one big disadvantage, which we had to take into account. XML based database is saved in a single file, which 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.

CHAPTER 6

Data structure

We have designed out class structure with the A* algorithm for the shortest path and OpenStreetMap (OSM) file structure in mind. We followed the OSM structure, having a basic class called **Node**. In figure 1 we show the different types we use with a small graphic representation. **Node**

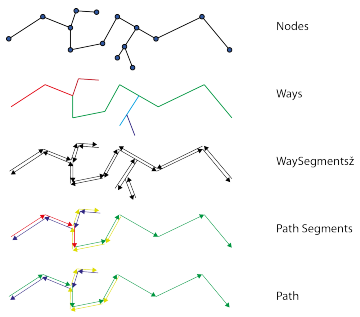


FIGURE 1. Different objects in our data structure

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 *way segment, building or way.* It contains location information, some A* algorithm informations (weights, visited, etc.) and a vector of Way Segments to which we can access from this Node. This is important expecially for A* algorithm, as it significantly reduce the time needed to find all possible and subsequently the correct next move.

Way

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 streets that have the same properties. Each way is assigned a type

(primary, secondary, etc.), direction (oneway, bidirectional) and access privilages (private or public).

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

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

Path Segment

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 point B.

Path

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 one PathSegment pointer. On the other hand, if we search for intinerary, Path will indluce multiple PathSegments, one for each pair of middle points.

Database

Database is an base class for the whole database. It contains functions for correct parsing of XML files and also contains containers *(std::map)* with pointers to each of the class created. 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 quickly find all the WaySegments in a neighborhood of a point. This is very useful in implementation of finding the closest waySegment.

In figure 2 we can see the whole class diagram of the database part of the application in C++.

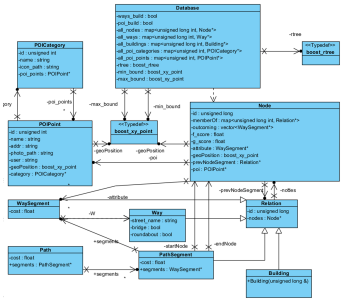


FIGURE 2. Class diagram

CHAPTER 7

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 oneway 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 (section[]) we were able to identify the three main user requests involving path algorithms:

- find shortest path from point A to point B;
- find shortest paths to all points of interest in a certain radius from point A (by foot or by car)
- construct an itinerary from point A to B with points of interest in between (with max distance limit)

From identified requests, 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 we need to find, but differing in how many unnecessary paths are also calculated. This subsequently mean longer calculation time, which we want to avoid. For this reasons we chose **A* path finding algorithm**.

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.

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 is what selected sets A* 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 matters and we have normal Dijkstra algorithm. Because we do not know distance from node to 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 to not found the shortest path. We decided to use air distance as a heuristic, as this guarantees that the actual distance will be equal or greater than that, so we never over-estimate it.

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 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. 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[]. It is worth mentioning that if the perpendicular projection of the point is outside of the interval between both nodes, it snaps to the border node. This can be seen on figure1 (left and right).

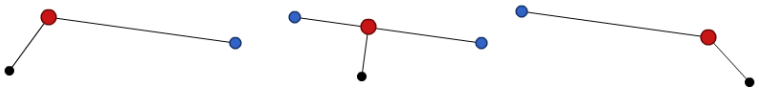


FIGURE 1. Snapping user point to border nodes (left,right) and snapping to perpendicular point where possible

With this information we create a new mock node which represents the start point of our search. 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.

2.1. Implementation in C++. Road snapping function requires an extremely time consuming operation of finding close segments. We could have actually calculated the perpendicular projection on every WaySegment in the map and then among them find the smallest, but this would mean ten 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 increasing it, until we find any intersecting WaySegments. This approach reduces the calculations of perpendicular projections from ten thousand to just a couple. The diagram of the proces is shown in the figure[]

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[matlab CITE] the code for calculation. The speed of the calculation turned out to be sufficient.

3. Shortest path A $->$ B search

4. Radius search

5. Bicycle search

CHAPTER 8

GUI

1. C++

1.1. UI.

1.2. OpenGL.

2. Matlab

2.1. UI.

2.2. library for maps.

CHAPTER 9

c++ vs matlab

Bibliography

[1] J. D. Bovey, M. M. Dodson, The Hausdorff dimension of systems of linear forms *Acta Arithmetica* (1986) 337-358.

[2] J. W. S. Cassels, *An Introduction to Diophantine Approximation*, Cambridge University Press, 1965.

[3]