University of Essex

Final Report

# Digital Mapping Web Application: routing for multi-floor buildings

*Author:*
Bogdan Suvar

*Supervisor:*
Keith Primrose

April 27, 2011

# Abstract

Digital maps are vital for any environment, especially for a university campus with a complex numbering system. Furthermore, digital maps capable of rendering directions across multiple floors of a building are nonexistent, most commercial mapping systems rendering directions only for one level. In this project, a novel approach to render directions in a web-based format across different floors is developed by making use of Geographic Information System tools and cartographic libraries. The report describes the design and implementation of two algorithms that make use of existing cartographic libraries and applications for developing a web application.

In the report, the following are described in detail:

- GIS applications, libraries and their contribution to the project. OpenEV is an application for manipulating vector files and is used as a back-end user interface. Mapserver is an application that generates images from vector files for web publishing.

- the preprocessing and routing algorithms. If the locations on campus are represented as nodes that are connected through edges in a graph, the preprocessing algorithm builds the graph from geometry features in the vector files. On the graph, a bidirectional shortest-path algorithm reveals the shortest distance between any two nodes.

- projecting the maps on the web. The user can access the application in a web browser and can make two types of queries: location search and directions between two locations. The location search renders a map with the query result highlighted in a different colour. Directions between two locations will display a path from the source location to the destination. Should the source and destination be on different floors the path will be displayed for each floor between the source/destination and stair case or lift through which transition to another floor is made.

# Contents

# Listings

# List of Figures

# Part I

# Introduction and Background

# 1. Introduction

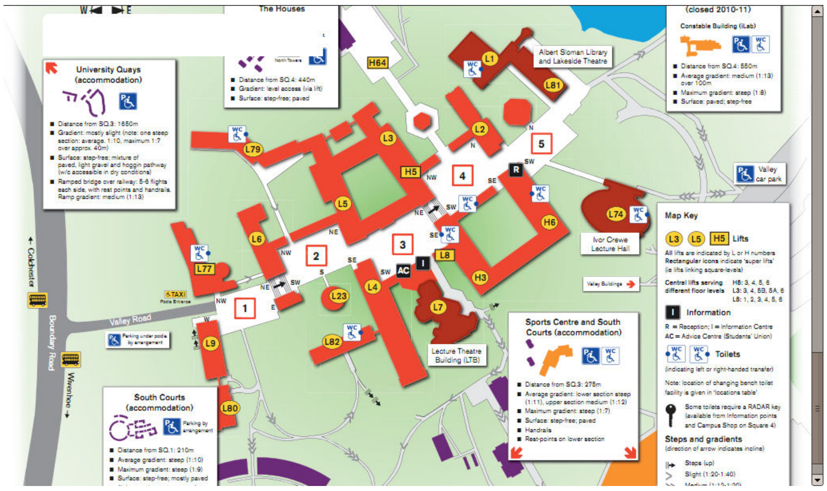
## 1.1 Motivation

Every year, hundreds of students, guests and academics arrive at the University of Essex to study, work or visit. Often, when they are required to go to a room such as 4S.6.28 confusion ensues. This confusion is caused by *static* maps which require the user to read guidelines and instructions in advance, increasing the difficulty of locating rooms quickly. The university room numbering system takes time to learn, and even after years of studying or working at the university, knowing the shortest path between two locations is difficult. This issue, however, is common to other campuses or large buildings with many floors.

### 1.1.1 Existing Systems

#### 1.1.1.1 Paper-Based

At the moment the university has very detailed information about the numbering system available in handbooks, leaflets and on its website. Details such as the closest entrance from the squares, square numbers and toilets that are accessible for people with disabilities are represented as shown in Figure 1.1. While the map is very detailed, with clearly marked buildings, squares and entrances, a user that wants to locate a room on the map in a matter of urgency would find this map useless. That represents the main issue with the current map guidelines and handouts. Another issue regarding the current maps resides in the fact that they are printed on paper and any further changes in the numbering system, such as a building extension or an interest point, cannot be propagated for the user to use the most up-to-date version. Furthermore, any changes on the map layout involves an update of the physical resources which leads to economic and environmental costs.



The Houses
University Quays (accommodation)
Albert Sloman Library and Lakeside Theatre
Constable Building (iLab)
Valley car park
Map Key
Lifts
Information
Toilets
Steps and gradients
Ivor Crewe Lecture Hall
Sports Centre and South Courts (accommodation)
Lecture Theatre Building (LTB)
South Courts (accommodation)
Valley Road
Boundary Road
Colchester
Wivenhoe

Figure 1.1: Current Campus Access Map

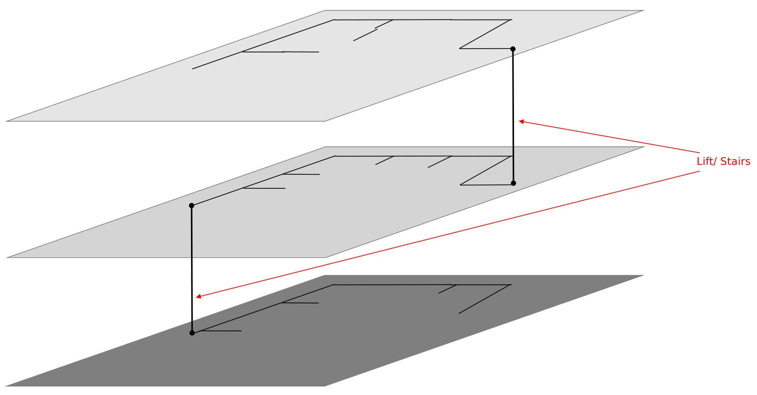


Figure 1.2: 3-Dimensional Graph

#### 1.1.1.2 Digital Mapping

Maps were reinvented in the middle of the last decade through web-based applications such as Google Maps, Yahoo Maps or Mapquest. This map revolution allowed a more intuitive, usable and scalable approach to represent maps and directions by leveraging on web technologies, such as AJAX, in order to build Rich Internet Applications. In building an electronic version of the current map for the University of Essex campus, as a scalable Rich Internet Application, I opted for using similar concepts and tools required for building the commercial digital mapping applications. Therefore, as further described in Chapter 2, the tools, programming languages and libraries are common in building Geographic Information Systems. Most of today's digital mapping application only renders directions or query results for one and not multiple levels. There is no system that displays directions or query results across multiple levels such as for an office building with many floors.

### 1.1.2 A Novel Approach

The application's architecture and design is new to digital mapping systems by providing the capability of not only identifying a point of interest across different floors, but also rendering directions on different floors. The project makes use of Geographic Information System third-party tools for defining paths and rooms, but also for web publishing. The innovation resides in growing a 3-dimensional graph from geometry features used in a routing algorithm that identifies the shortest-path between two locations, as shown in Figure 1.2.

The code-base developed in the project can be re-used in other Geographic Information Systems as it is built on the most popular standard GIS libraries.

## 1.2 Aims

The list below represents the aims of the project that can also be interpreted as a collection of functional and non-functional requirements.

**Scalability**

Scalability represents one of the most important aims for this project. The application needs to provide a back-end user-interface that allows further changes to be made with ease by a non-technical person. The algorithms must be designed in such a way that no editing of the source-code is necessary for adding further locations, interest points and routes on campus. The application needs to be independent from the current architecture of the University of Essex campus. It is important to avoid tight coupling to the current campus architecture and design a decoupled implementation that can be reused in other environments.

**Directions**

The map must provide directions between two locations on campus, even across different floors. This poses a significant challenge for the project in order to achieve the best usable product. There are currently no other digital mapping systems that render directions across different levels (i.e. Google Maps offers directions on a single plan).

**Portability**

The application must run regardless of the users' operating system. To achieve this, the application is built in a web-based environment which only requires a Javascript enabled browser to run. Another advantage of web-based implementation is device independence, meaning that the application can run on various devices that support a Javascript enabled browser.

**Open-Source**

All the components required to build the application, including libraries, third-party software and development environments must be open-source. This also allows the software to be independent from licenses or other commercial costs, and also share with the open-source community ideas and concepts that can be further reused.

**Speed**

Research shows [1, 2] that, online, speed plays a vital role in the user experience with psychological effects. The slower the loading time for a web page, the more likely the end-user will be frustrated, dissatisfied and perceive the website as less attractive [3]. Therefore, it is important to deliver results fast.

## 1.3 Achievements

The application is built in an entirely open-source environment and using only open-source software components and libraries. The application was developed by using Eric Integrated Development Environment although other alternatives were available. Other similar choices were made over the Geographic Information System tools used for the implementation: OpenEV for analyzing and manipulating geometry features in vector layers or Mapserver for projecting the vector layers in a web-based format. The following chapters will describe in detail the implementation advantages and pitfalls of using open-source alternatives.

For the project I had to familiarize myself with tools and concepts as well as different Application Program Interfaces and programming languages that were not taught as part of my degree. Therefore, although this represented a steep learning curve, it allowed me to broaden my horizons and identify patterns between Python and other languages I had to develop in during my degree, such as Java or C.

The project also provided an opportunity to learn how commercial digital mapping systems are built by having to familiarize myself with GIS tools used by the industry. It also provided an opportunity to develop my research skills, having to understand different and new concepts in order to develop the algorithms. By using three different APIs, a constant rate of learning had to be maintained across the length of the project. The major gains are related to quickly being able to adapt to different documentations, method signatures, patterns and locate the needed information quickly.

For the routing engine, while Dijkstra's [4] shortest-path algorithm is faster than other classic algorithms [5], novel implementations that have been developed are at least 50% faster [6]. The application will make use *Bidirectional Dijkstra* [7]for searching between two nodes in a graph, which is faster than the classic, single-source Dijkstra search. Faster alternatives can be developed as extensions to the current implementation, as described in Section 7.3.

The *Large-Scale Software Systems* module, taught by Professor Riccardo Poli provided the knowledge required to make use of Subversion as a versioning system for the project, but also employ *Test Driven Development* techniques for software development.

The *Business Information Systems* taught by Iain Langdon also allowed to gain further knowledge about creating diagrams for systems.

Fast query results were achieved through the use of *Dracones* wrapper of the Mapscript library, further described in Chapter 5.

The application is deployed on a LAMP architecture using only open-source software. The reports were written with open-source software such as LaTeX, BibTex, Dia (for diagrams) and Open-Office for the poster.

## 1.4 Report Structure

The report is divided in three main parts:

- Part I offers an introduction and background. It contains two chapters: Chapter 1 provides the introduction and motivation; Chapter 2 offers background information about the GIS applications and concepts.
- Part II describes the implementation details and consists of three chapters: Chapter 3 contains the implementation of the preprocessing algorithm; Chapter 4 details the implementation of the search algorithm and Chapter 5 discusses the projection details.
- Part III provides an overview over the evaluation, testing and the conclusions. It contains two chapters: Chapter 6 details the testing and usability evaluation of the product and Chapter 7 offers conclusions for the project and the software developed.

# 2. Geographic Information System

## 2.1 Introduction

There are two ways of building digital maps: by designing an entirely new architecture and concepts or by making use of existing Geospatial/Geographic Information System cartographic standards and specification. The terms GIS, Geographic or Geospatial Information System are used interchangeably across this report. For this project, the second alternative was chosen, which has both advantages and drawbacks. It is advantageous because, by following the cartographic standards for developing digital maps, there is a well-formed community from which the concepts can be quickly learned[8, 9]. A drawback represented the constant learning rate required for using the software, understanding the concepts and develop software using the various GIS libraries.

In order to adhere to the aim of using only open-source tools for the entire project, all the Geographic Information System software products are open-source. This raised numerous issues in setting-up the development environment as well as other technical challenges. Some of the tools were also not actively developed or widely used in the GIS community which required significant technical investigation in identifying appropriate solutions.

The two Geographic Information System software applications used in the project are:

- **OpenEV**
- **Mapserver**

## 2.2 Conceptual Model

Figure 2.1 displays an overview of the system components. Components whose title are in bold are developed specifically for this project and constitute the routing engine of the application. The components interact with each other without being tightly coupled. For instance, OpenEV can be replaced with other, commercial, alternatives for analysing and editing vector files; Mapserver can be replaced with other alternatives that publishes vector files on the web. The software developed in this project is, thus, not reliant on the software components being developed using standard cartographic libraries.

The *Preprocess* component consists of the algorithm that converts geometry features (or shapes) drawn in OpenEV into a graph. This graph is used for running a *shortest-path* algorithm between two nodes. The coordinates resulted from the algorithm are used for creating new geometry features that will be rendered by Mapserver.

## 2.3 OpenEV

OpenEV is an open-source application used for analyzing and manipulating geospatial data[10]. It is built in Python using various GIS libraries and also supports developer extensions. One of its main advantages is that it is cross-platform, being capable of running on Windows, Linux or Sun Solaris operating systems. Another advantage is that it supports a wide range of raster file formats for input, and can handle large file sizes (gigabytes) gracefully. Raster images can be inserted as layers. On top of a raster layer, vector layers can be defined.

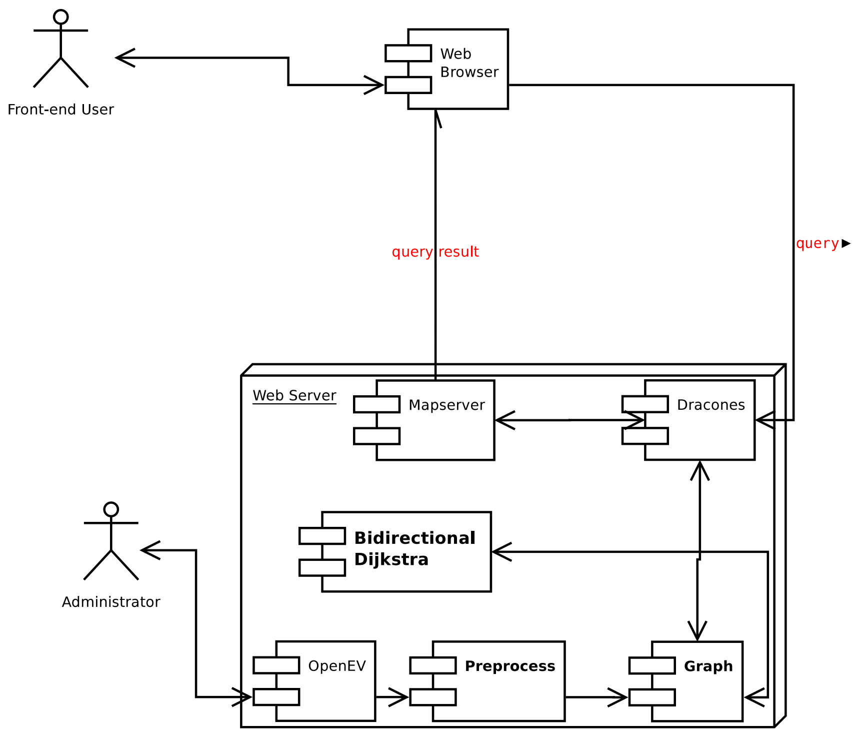


Figure 2.1: System Conceptual Model

Upon the definition of a vector layer, various geometrical shapes can be drawn, including points, polylines or polygons. The layer's background is transparent which allows geometry shapes to be modelled after a raster image, such as a floor plan. Shapes and geometry features are used interchangeably across the document and represent points, polylines or polygons.

For each layer a database schema can be defined in which columnar attributes for each geometry feature on the respective layer can be assigned. There is a limitation in OpenEV, at the moment, which does not support deleting attributes (columns) from the schema, but only allows them to be added. In Figure 2.2 a screen shot shows the usage of the *Tabular Shapes Grid* feature in OpenEV for defining the attributes for each polygon geometry feature.

### 2.3.1 Vector Files

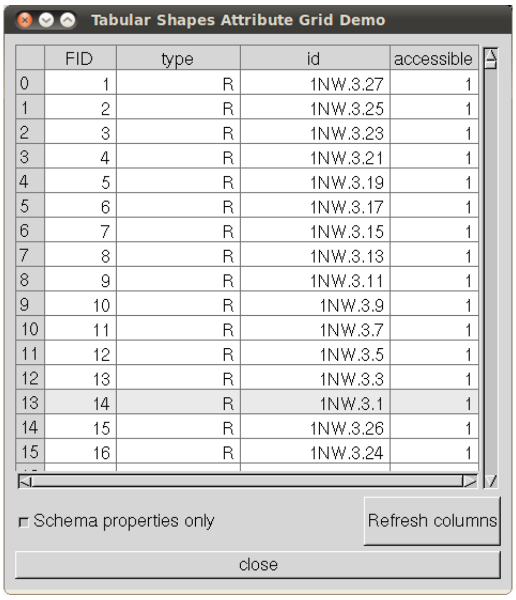
Each layer is saved as an ESRI shapefile having the extension *.shp. Shapefiles represent a simple way in which geometric features can be stored. However, the geometry features are of no use if they do not have attributes (i.e. id, type) assigned to them. Therefore, a separate file, of dBase format, stores the columnar attributes defined in the Tabular Shapes Grid for each shape defined on a layer. These attributes are necessary for the preprocessing algorithm (described in Chapter 3).

Another file that is also created every time a layer is saved is a shape index format, that contains a positional index of the geometry features which allows fast search forward or backward across the shapes. This will also be used for iterating through the features in the preprocessing algorithm, described in Chapter 3.

There is one, very important, limitation: each shapefile can only contain one geometry type. For instance, for a layer that contains polygons, it is possible to draw a polyline or a point on the same layer, the features being saved, but Mapserver can only render one geometry type per layer.

To summarize, there are three files created when saving each layer in OpenEV:

- **shp**: geometry shape features format
- **shx**: shapes index format



| | FID | type | id | accessible |
|---|---|---|---|---|
| 0 | 1 | R | 1NW.3.27 | 1 |
| 1 | 2 | R | 1NW.3.25 | 1 |
| 2 | 3 | R | 1NW.3.23 | 1 |
| 3 | 4 | R | 1NW.3.21 | 1 |
| 4 | 5 | R | 1NW.3.19 | 1 |
| 5 | 6 | R | 1NW.3.17 | 1 |
| 6 | 7 | R | 1NW.3.15 | 1 |
| 7 | 8 | R | 1NW.3.13 | 1 |
| 8 | 9 | R | 1NW.3.11 | 1 |
| 9 | 10 | R | 1NW.3.9 | 1 |
| 10 | 11 | R | 1NW.3.7 | 1 |
| 11 | 12 | R | 1NW.3.5 | 1 |
| 12 | 13 | R | 1NW.3.3 | 1 |
| 13 | 14 | R | 1NW.3.1 | 1 |
| 14 | 15 | R | 1NW.3.26 | 1 |
| 15 | 16 | R | 1NW.3.24 | 1 |

Figure 2.2: Tabular Shapes Grid in OpenEV.The *type*,*id* and *accessible* are attributes assigned by the user and specific to every shape

- **dbf**: dBase format containing the attributes of the geometry features

### 2.3.2 Representing the Floors

Given the limitation mentioned in the above section, for representing the floors, two types of vector layers are defined:

**Lines**

This layer represents the paths between rooms, interest points or various locations. This layer will be used by the preprocessing algorithm for converting the lines into edges of a graph. The order of this layer is higher than the polygons and raster(floor plan) layers. In an attempt to lessen the user input, there is no other input apart from drawing the lines and saving the layer with the level number as an integer; the weight and other details being assigned to them during preprocessing. There are few *very important* requirements for drawing these lines which are discussed in Section 3.2.1 of the following chapter.

**Polygons**

This layer represents the rooms, lecture halls, lifts or staircases as polygons. The order of this layer is higher than the raster layer. The polygons are drawn according to the floor plan (raster layer). For this layer, through the use of the Tabular Shapes Grid, attributes are assigned to the polygon shapes as follows:

**Type** as a string, to assign a type to the polygons as using the following acronyms:

- **R** for indicating a room such as an office, or other similar space
- **LTB** which stands for Lecture Theatre Building
- **LAB** to indicate a laboratory
- **FT,MT,DT** female, male and disabled (accessible for people with disability) toilet.

**id** also of type string, used to store the identification of that point of interest. Together with *type* it must uniquely represent a polygon. Thus, type and id fields form a *super-key*.

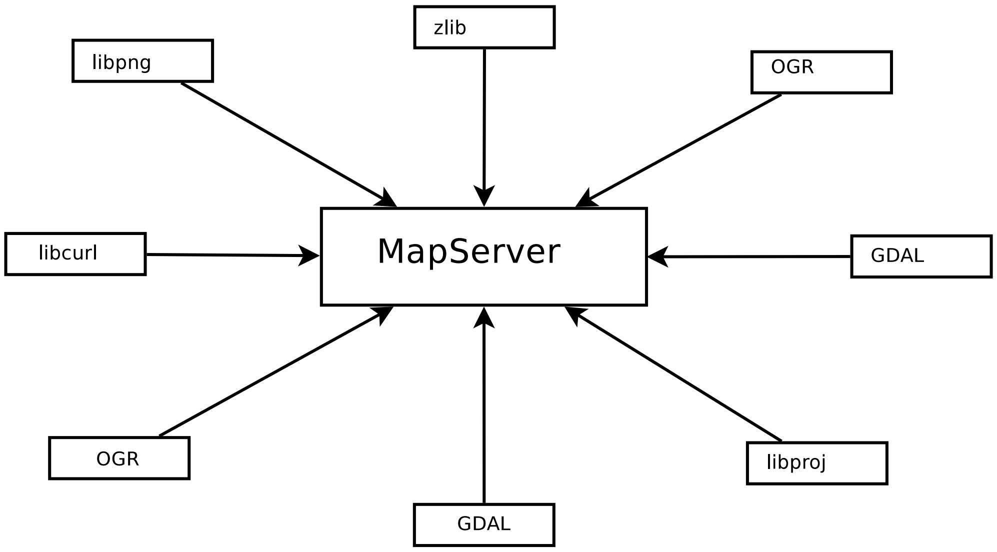


Figure 2.3: Mapserver application library dependencies

**accessible** as an integer, holds the value of 1 or 0. In the case of edges, 1 represents an accessible path (including lifts) and 0 otherwise. For rooms, represented as polygons, 1 represents the door opening indoors while 0 otherwise.

## 2.4 Mapserver

Mapserver is an open-source application for publishing spatial data on the web and usually runs on a web server, in our case, Apache. The application generates images based on the geometry features defined in the ESRI shapefiles that result from OpenEV. The most important file for Mapserver is the *mapfile*, with the file extension *.map. The mapfile, similar to an XML file, defines an entry for each of the layers defined in OpenEV. Furthermore, through its parameters the style of the image rendered, the file type of the image or the zooming levels can be controlled. The entire mapfile used in this project is listed in Appendix C.

The installation of Mapserver, on Ubuntu, presented a slight challenge because it required numerous libraries to be installed for the application to generate images, Figure 2.3 shows the libraries. The libraries had also version dependencies which further added to the technical complexity.

An unsuccessful attempt to convert the vector attributes from the shapefiles into a postGIS database (postgreSQL database server with GIS extensions) as recommended in Chapter 3 of Mitchell's book [11] resulted in the decision of using *shapefiles* instead. Should the attempt had succeeded, it would have provided a more decoupled design.

While the above solution would have also eliminated the single point of failure and allowed decoupling the presentation server from the data server, it would have added another layer of complexity for the back-end user. The back-end user would have been required to learn and familiarize herself with another set of commands for converting the shapefiles into the database and also define the connection properties(*username, password, server address*) inside each layer definition in the mapfile. For the reasons mentioned above, the implementation of a postGIS database was abandoned.

## 2.5 Programming Language

In developing the software for the project, two programming languages arose as potential candidates: Python and PHP. Analysing the documentation of Mapserver and Dracones as well as other facts about both languages resulted in the decision of using Python as the server-side development language

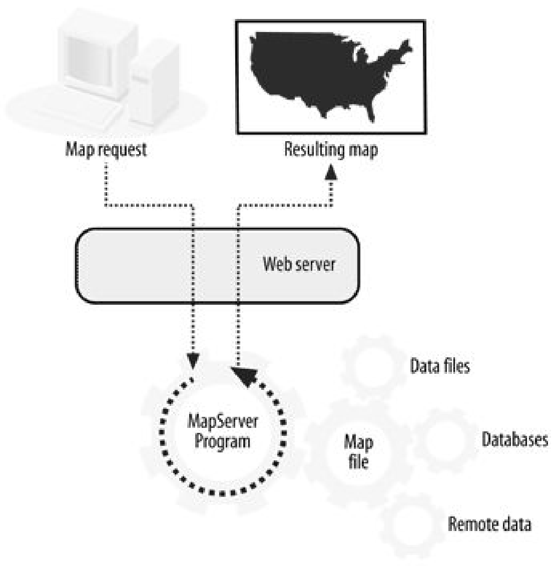


Figure 2.4: Mapserver structure

for this project. Further details about the reasons behind this decision can be found in Section 3.3 of the following chapter.

Although Python has not been taught as part of my degree, which required further resources to be allocated for learning the language, the benefits of using this programming language outweighs those of using PHP: its clear syntax, the built-in data-structures or the interactive interpreter are just a few.

For the client-side, Javascript is used to call the functions defined on the server-side. Although HTML5 would have provided a viable alternative, the current Dracones API only supports Javascript.

Part II

# Implementation

# 3. Preprocessing algorithm

## 3.1 Introduction

A graph data structure consists of a finite and mutable pairs of nodes, called *edges*. The use of the graph data structure in the routing algorithm is detailed in the next chapter. This chapter provides the implementation details of how the graph is populated from the OpenEV shapefiles.

Following graph theory, let us consider G = (V,E) where G is a graph built from a set of nodes, V, which are connected by a set of edges, E. Should the edges have a certain direction, the graph is called directed, otherwise, it is undirected. A search algorithm is used for identifying the path between two nodes in a graph. A directed graph allows the search algorithm to expand nodes only in a certain direction. The search algorithm expands the nodes based on the *weight* of each edge that connects a node to its neighboring nodes. The edges must contain non-negative weights for the algorithm to expand the nodes correctly and identify the shortest path between two nodes.

First, the graph needs to be populated. The sections below describe the details of growing an undirected weighted graph from line geometry features read from the shapefiles. Through code listings, the use of various libraries and data structures is detailed. By using the geometry features to grow the algorithm, the algorithm developed can be reused with any other application that use ESRI shapefiles format for manipulating geospatial data. Apart from re-usability, drawing the geometry features in OpenEV, or any other software that manipulates vector files, is more intuitive and easy for the back-end user. Therefore, OpenEV is considered the Graphical User Interface for the back-end user - the person who administrates the maps.

The search algorithm adopted in this project for identifying the shortest-path between two nodes is bidirectional Dijkstra and its implementation is explained in the next chapter. In this chapter the implementation of the preprocessing algorithm through the use of cartographic libraries is described.

## 3.2 Design

In OpenEV the user can draw various shapes, such as a line, polyline (a connected series of line segments), polygons or points. Each line has two coordinates representing the start and end of the line. Therefore, the pair of coordinates *x1* and *y1* can represent a node/vertex in the graph while an edge is formed by using the other pair of coordinates *x2* and *y2*, as shown in Figure 3.1.

The edges need to be connected for the search algorithm to successfully check each neighbour. Because each line or polyline, is an iterable collection of points with floating point accuracy, it is difficult for a user to draw lines that have their end/start points overlapping at the same exact coordinates of another start/end of a line. To address this issue, each point of a line or polyline is

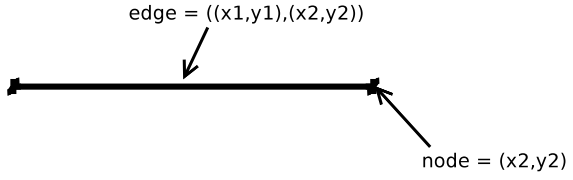


Figure 3.1: Representing a line as an edge

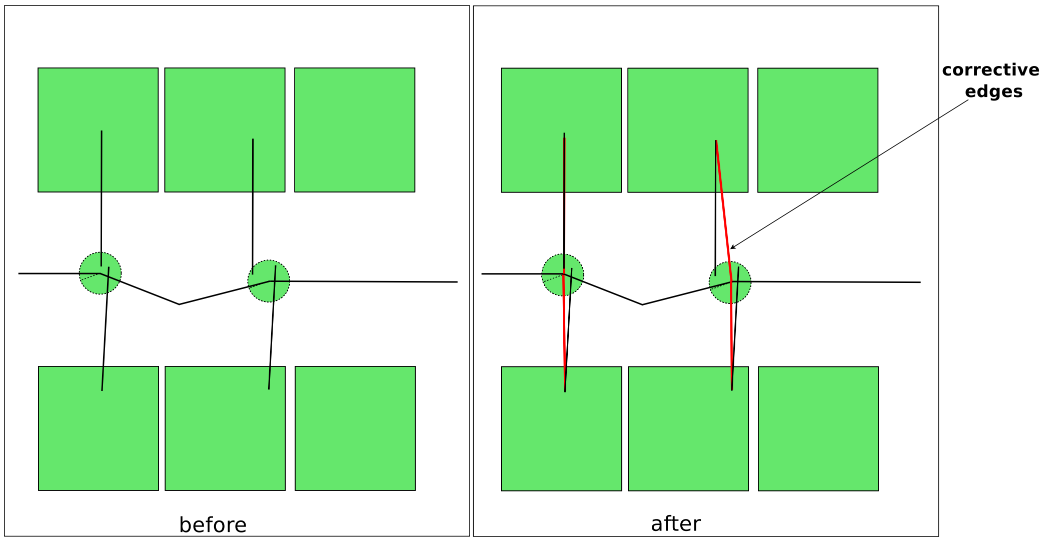


Figure 3.2: Using a radius to identify if points of a line geometry feature is close to other

checked to know whether they fall within a radius whose centre is the point of another line/polyline.

If that condition is fulfilled, then the edge will be between the radius center and the penultimate point of the line whose last point is within the radius, as shown in Figure 3.2.

Because the lines are defined on a different layer than the polygons layer, in order to avoid entering details such as the type or id (i.e."4.S.28"), I use a computer-vision algorithm (further described in Section 3.6 below) to identify if a line's end (last/first point) falls within a polygon. If it does, the point coordinates along with the polygons attributes are saved together in the graph as a node.

The lifts and stairs are defined as polygons, and their *type* and *id* are used for connecting the point of the line that fell within these polygons. Section 3.8 describes the implementation.

### 3.2.1 Prerequisites

This section describes a few prerequisites that need to be fulfilled in order for a graph to be successfully built. A polyline contains multiple lines tightly connected to each other, each line contains a pair of coordinate points, *x,y and z* for the starting and ending points. For the purposes of this project, however, only the first two values are considered, $x$ and $y$, while $z$ can be used in the future for representing 3-dimensional shapes. The list of prerequisites is shown below:

- A line must have one end falling within the boundaries of a polygon, defined in the polygons layer. This is necessary in order not to assign any attributes to the lines, but to copy the attributes of the polygon. Because a line has two pair of coordinates, and each pair *x1* and *y1* is considered to be a node, the coordinates that are located within the area of a polygon will be assigned that polygons' attributes. Since the project handles multiple floor plans, this is valid only for polygons and lines of the same floor. The implementation is described in Section 3.6.

- It is imperative that the user chooses the points of a polyline to be at points of intersection with other lines such as in front of the doors on the floor plan. A line from the polygon representing the room and to the point of a polyline is drawn, as shown in Figure 3.2. The algorithm will consider each point of a polyline as a radius and check whether other start/end of lines fall within. The radius distance is defined in the *Config* file. The user must draw a line such that it falls in this radius.

- The level (floor) number is entered when the vector layer selected is saved in a shapefile. To avoid duplicates, different directories must be used for the shapefiles representing polygons and those representing lines. Therefore, both the polygons and lines shapefiles must have the same filename, representing the floor number as integers.

## 3.3 Python versus PHP

Having to chose a programming language for implementing the algorithm on the server-side, I did some research for identifying potential candidates. The research resulted in a comparison between PHP and Python. The conclusions are presented in the following list:

**Syntax**
The clean and friendly syntax of Python is incomparable with that of PHP. Thus, allowing more focus on performance and implementation instead of syntax errors.

**APIs**
Further described in Section 5.3.1, Dracones is a wrapper class of Mapscript library, extending its functionality so that a panning and zooming is done interactively on the front-end. Dracones has its API documented in both, Python and PHP. OGR is a wrapper class of the popular cartographic Geospatial Data Abstraction Library (GDAL) and has its documentation only in Python. NetworkX is a Python module that is used to create, manipulate and analyse the structure of complex networks.

**Exception Handling**
Input/Output operations are used when handling the shapefiles. It is important to gracefully handle any IO exceptions that might occur. Python's exception handling mechanism is similar to that of Java while PHP5 does have exception handling mechanism it lacks the equivalent of *finally*.

**Data Structures**
Python provides support natively for data structures such as tuples, lists or dictionaries that are used directly in the algorithms. Using these fundamental data structures, more complex ones such as a graph or a binary tree can be built. PHP5 has only arrays as its built-in data structure, which is a real limitation for algorithm design and implementation.

**Interactive Interpreter**
The Python interpreter allows methods and variables to be redefined interactively, skipping the long compilation cycle other languages have. This has proven to be of great use for both learning and understanding the language through experimentation, but also for debugging purposes, a feature that lacks in PHP. Another use of the interpreter is *testing*, test cases can be quickly built and executed.

**Application deployment**
Deploying a Python web application introduced challenges in configuring the Apache server, such as correctly defining the configuration directives to allow all the Python modules to share the same Virtual Environment of the Web Server Gateway Interface. PHP would have allowed the application to be deployed much easier since it is already installed on most servers.

While the deployment of the application and environment set-up would have been easier by using PHP instead of Python, the rest of the above list provided sufficient reasons for adopting Python.

## 3.4 Graph implementation

The implementation of the graph extends the graph class of the networkX module. The graph implementation has the following features:

**Dictionaries of dictionaries**

The nodes are stored in dictionaries, each node can also have its own *"attributes"* dictionary consisting of keys:value pairs containing data such as weight in the case of edges or location information (i.e. type, id, accessible) in the case of a node. This design permits assigning values to nodes and edges extracted from the .dbf files through the use of the OGR library. As explained in Section 2.3.1, the contents of the dbf file is the data entered in the Tabular Shapes Grid when the vector layer is defined. The design follows the representation Guido van Rossum in representing graphs [12], where the graphs are dictionaries that have very short access time (used for existence test) and allows easy access and removal of edges.

**Hashable nodes**

The Graph object stores the nodes in a Python dictionary. Dictionaries use the *hashing* mechanism to compute an integer from an arbitrary object and store it as a "key" in a hash table. This allows values to be constructed in constant time (O(1)), in the average case, with respect to the hash table size, but linear time considering the objects' size. Therefore, the node is a tuple containing two floating point values (geometry coordinates) retrieved while iterating through the points of a linestring geometry feature. Listing 3.1 exemplifies how an edge is added between two points of a linestring. The *for* loop on line 2 iterates through the number of points of the linestring. Given that a linestring always has at least two points, the conditional statement on line 3 is used to skip over the first one and, on line 9, the previous and the current points are added as nodes both in the adjacency dictionary and nodes dictionaries of the Graph object. Line 7 passes the two points as parameters to a method that computes the Euclidean distance, further described in Section 3.7.

```
1 if g.GetGeometryType() == LINESTRING :
2   for j in xrange(g.GetPointCount()):
3     if j > 0:
4       attributes[LEVEL] = lyr.GetName()
5       self.add_node(g.GetPoint(j-1),attributes)
6       self.add_node(g.GetPoint(j),attributes)
7       attributes[WEIGHT] = computeWeight(g.GetPoint(j-1), g.GetPoint(j))
8       attributes[ACCESSIBLE] = 1
9       self.add_edge(g.GetPoint(j-1), g.GetPoint(j), attributes)
```

Listing 3.1: Adding an edge from the coordinates of a linestring

## 3.5 Growing the graph

OGR is a Simple Features Library, open-source, written in C++ that provides read and write access to vector formats, among which the ESRI Shapefiles. The Python bindings for the libraries were generated through SWIG, its API being very similar to that of C++. Using the OGR library all the 3 files generated by OpenEV (shp, shx and dbf) are used for detecting the geometry type, iterating through the geometry features and extracting the attributes of each geometry feature assigned through the Tabular Shapes Grid.

In the *Preprocess* class, two methods are used for iterating through geometry features in a shapefile: one for the geometries of type linestring and another for those of polygon type. The method that iterates through the polygons (*loadPolygons*) is called first from the *Graph* class (see Class Diagram in Appendix D) with the path to the directory that contains the polygon vector files. Another call to the method that iterates through the lines (*loadLines*) is made afterwards, with the path to a different directory. The reference of each polygon geometry feature object is stored in a list that, in turn, is stored as a value in a dictionary with the key being the floor number. The list contains tuples of the geometry feature object reference and its *attributes*.

```
  for findex in xrange(lyr.GetFeatureCount()):
```

```
2      f = lyr.GetFeature(findex)
3      flddata = getfieldinfo(lyr, f, fields)
4      g = f.geometry()
5      attributes = dict(zip(fields, flddata))
6      attributes[LEVEL] = lyr.GetName()
7      if g.GetGeometryType() == POLYGON :
8        t = f, attributes #store the geometry feature along with its attributes
9        polygons.append(t)#store all the polygon geometry features in a list
10       if self.levelPolygons.has_key(lyr.GetName()):
11         self.levelPolygons[lyr.GetName()] = polygons
12        else:
13         tempDict = {lyr.GetName():polygons}
14         self.levelPolygons.update(tempDict)
15         tempDict.clear()
```

Listing 3.5 describes the implementation of the *loadpolygons* method. The first line will iterate through all the geometry features of the layer being loaded, the second is used to assign the feature object used for retrieving the attributes defined in the Tabular Shapes Grid for each feature (line 3). In the *attributes* dictionary, the shapefile file name (as it is saved in OpenEV) is used as the level number.

On line 8, a tuple is used to store the feature object along with the attributes dictionary of that shape. The tuple is saved in a list. The list is added to the *levelPolygons* dictionary, where each level has the floor number as a key and the list aforementioned as a value. For instance, if there are 5 levels, the *levelPolygons* dictionary will consist of 5 entries. Each entry having the key as the level number i.e. 1, 2, 3, etc., followed by a list of all the polygon geometry feature objects for that level.

Using the dictionary data structure allows for faster look-up (in the average case, O(1)) when the values are used for checking if a point of a line falls within a polygon (further detailed in Section 3.6). The search through the polygon features stored in the list is also faster by using a dictionary instead of a named tuple or a list.

The *loadLines* method iterates through the *LINESTRING* geometries of the shapefile using a *for* loop (Line 1 of Listing 3.2). At the end of the loop, each linestring geometry object is stored, along with its *attributes* dictionary, as a tuple, in a list. The list, thus, containing all the linestring geometry objects for a layer. Every new linestring geometry feature, has its points checked against the points of each linestring already stored in the list (line 9 and 13). The points of the linestring objects stored in the list are used as the centre of a radius in the *proximityCheck* method.

```
1    for findex in xrange(lyr.GetFeatureCount()):
2        ...
3        if g.GetGeometryType() == LINESTRING : #line or polyline
4            if self.lines: #list containing all the linestring geometry objects
5                for feature in self.lines:
6                    lineg = feature.geometry() #geometry object of the line from the list
7                    countPoints = g.GetPointCount()
8                    for x in xrange(lineg.GetPointCount()):
9                        if proximityCheck(lineg.GetPoint(x), g.GetPoint(0)) and lineg.
                             GetPoint(x) != g.GetPoint(0):
10                           attributes[WEIGHT] = computeWeight(lineg.GetPoint(x),g.GetPoint
                                 (1))
11                           attributes[ACCESSIBLE] = 1
12                           self.add_edge(lineg.GetPoint(x),g.GetPoint(1),attributes)
13                       if proximityCheck(lineg.GetPoint(x),g.GetPoint(countPoints-1)) and
                             lineg.GetPoint(x) != g.GetPoint(countPoints-1):
14               ...
15                         self.add_edge(lineg.GetPoint(x),g.GetPoint(countPoints-2),
                               attributes)
```

Listing 3.2: Checking whether the points of a line fall in the radius of another

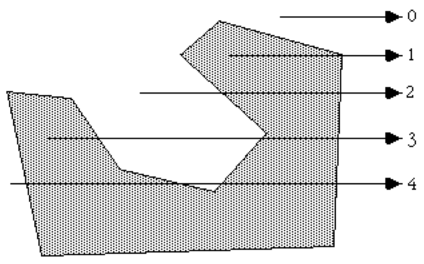


Figure 3.3: Ray Casting Example: counting the number of intersection of a line starting from the point that needs to be checked

After the *proximityCheck*, on lines 9 and 13, the weight of the "corrected" edge is calculated. Section 3.7 describes how this distance is calculated.

The value of the *accessible* attribute of the edge is being assigned the value 1 given the fact that the line represents a path through corridors where wheelchair access is possible. The only edge that has its accessible attribute equal to 0 is the edge representing the stairs.

## 3.6 Point within Polygon

Since the user assigns all the attributes, such as the room number (id) or type (R, LTB, LAB) to the polygons (using the Tabular Shapes Grid in OpenEV), but the lines/polylines defined in a separate layer (and separate shapefile) represent the edges in a graph. It is, thus, important to check whether the first or last point of a line is within a polygon in order to copy all the polygon's attributes to the nodes' attributes. This approach saves the user, when drawing the lines, from using the Tabular Shapes Grid to add attributes to each line. Therefore, the OpenEV user only has to draw the lines and save the level as the shapefile name.

A linestring is an iterable collection of points which allows us to check only if its first and the last point falls within a polygon. For this reason the polygons are loaded first and the lines second. The *loadLines* method checks whether the first or last point of each linestring of a layer falls within the polygons of the same layer.

Identifying whether a point falls within a polygon appears to be deceptively simple because visually, it is easy to identify and comprehend whether a point is in a polygon or not, checking this computationally becomes more complex. There are two methods of doing this: Ray Casting algorithm and OGR API.

The **Ray casting** algorithm [13] is an image order algorithm used in computer graphics. The concept behind *Ray Casting algorithm* is to draw an imaginary line from the point that needs to be checked, across the polygon. The number of intersections within the polygon boundaries is counted. If the count is odd, the point must be inside, otherwise it is outside. Refer to Figure 3.3 for a graphic representation of the algorithm.

The implementation of the algorithm is detailed in Appendix A and follows Bourke's original implementation [14] in C programming language.

The OGR API has a method, *Within*, which is called on a geometry object and takes as a parameter another geometry object. The method compares the references of the two objects and returns a boolean value.

```
if j == 0 or j == g.GetPointCount()-1:
  point = ogr.Geometry(ogr.wkbPoint)
  point.AddPoint(g.GetX(j), g.GetY(j))
  if self.levelPolygons.has_key(lyr.GetName()):
    for polygon in self.levelPolygons[lyr.GetName()]:#iterate through the polygons on
        the same level
      if point.Within(polygon[0].GetGeometryRef()):
```

Listing 3.3: OGR's Within method for point in polygon detection

A point object is created from coordinates of the first or last point of a linestring (line 2 and 3 of Listing 3.3). The *for* loop iterates through all the elements of the *levelPolygons* dictionary which was populated in the *loadPolygons* method. Each element is a tuple that has two elements: the polygon geometry feature object and a dictionary of attributes. Therefore, the *Within* method call, on Line 6, is applied to the newly created point object and the Geometry Reference of the first element of the tuple - the polygon geometry feature object.

```
attributes[TYPE] = polygon[1][TYPE]
```

Listing 3.4: Copy of data between the polygon and point dictionaries

Should the method return true, the tuple's second element, the dictionary, is accessed as shown in Listing 3.4 and its values are assigned to the attributes dictionary of the point that fell within the polygon.

From the two implementations that check whether a point falls within a polygon, the second option - using the OGR API *Within* was chosen. Using the Ray Casting algorithm was faster in the test cases, but the OGR documentation does not provide sufficient information [15] for accessing all the coordinates of a polygon geometry feature required as parameters in the implementation of the Ray Casting algorithm.

Because of the exchange of attributes between the polygon and the point, stored as a node in the graph, it is now possible to retrieve the coordinates of a point by querying the type and id (i.e. "R", "4.S.28"). The *getCoord* method defined in the Graph class (see Listing F.1 in Appendix F) implements this logic.

## 3.7 Edge weight computation

In order to determine the shortest-path between two nodes, bidirectional Dijkstra algorithm is applied on a weighted graph. A weighted graph consists of its edges being assigned a non-negative weights. The weight can be represented as the Euclidean distance between two points and is calculated by using Pythagoras' theorem.

The Pythagoras' theorem is defined as follows:

$$a^2 + b^2 = c^2 \tag{3.1}$$

where $c$ represents the hypotenuse of a right triangle and $a$ and $b$ represents the length of the other two triangle sides, as showed in Figure 3.4. Thus, the distance between two points represents the hypotenuse - the line segment that connects them. Figure 3.4 displays two points, A(x1,y1) and B(x2,y2), in a 2 dimensional Euclidean space. The distance between A and B is given by:

$$d(A,B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.2}$$

Essentially Pythagoras' theorem can be implemented in two ways. First, by defining a method that implements Euclidean distance by taking two parameters,the pair of coordinates for the two points, then calculate the square root and return a floating point value.

The second, is through making further use of *OGR* library by using its API method *Distance*, that computes the distance between two geometry objects. But because the function is called with the coordinates of the point geometry object, this requires to create two geometry objects from the parameters, whenever a distance has to be computed.

In terms of speed, for testing the first approach *timeit* module proved to be useful to test the instantiation and access times of tuples. The coordinates are passed to the method as tuples, accessing

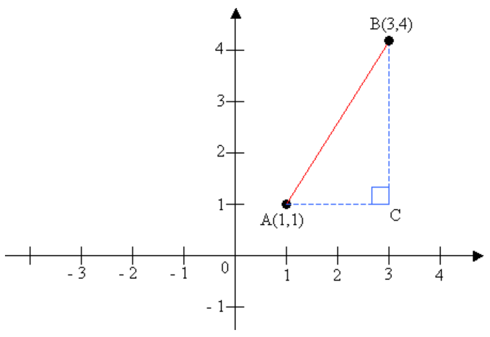


Figure 3.4: Calculating the distance between two points in a 2-dimensional Euclidean space

the tuples' indices. Accessing the tuples over 10000000 loops proved that the access time is 0.123 microseconds per loop.

This speed is caused by the fact that a tuple object is of immutable type, and the tuples are already instantiated in the *loadlines* method, the only expense being the access time required when calculating the square root. The second technique, of using the *OGR* library for creating two geometry objects, is inherently slower due to the fact that, in addition to the tuple access time needed for assigning it to the newly created geometry objects, there is also the cost of creating two geometry objects for every method call.

As detailed in Chapter 6 of *Learning Python* by Lutz [16], Python's garbage collector avoids the need to delete the objects created because once unreferenced the memory space is freed from the object space.

```
def distance(point1, point2):
    startpoint = ogr.Geometry(ogr.wkbPoint)
    endpoint = ogr.Geometry(ogr.wkbPoint)
    startpoint.AddPoint(point1[0], point1[1])
    endpoint.AddPoint(point2[0], point2[1])
    return startpoint.Distance(endpoint)
```

Listing 3.5: A method for calculating the distance between two points

To conclude, the first implementation method was chosen due to its speed. Creating two geometry objects from coordinates every time the distance needed to be calculated being more costly in terms of memory.

For implementing the algorithm in a Test Driven Development manner, I first wrote a class that inherits the *testCase* superclass of the *unittest* library and defined functions that use *assert* method to test the algorithms' method return values.

## 3.8 Representation of Lifts and Staircases

So far, the details of growing a graph from geometry features found in a layer have been described. But since a layer only represents one floor, it is important to connect the graph formed on each floor with the graph on the floor above and below.

Because each polygon is being assigned a type, an id, and an accessibility integer in the Tabular Shapes Grid, the type of the polygons identified as staircases will have the type "S" and an id assigned as an integer. It is very important that the polygon marked as staircase, over the same area of the floor plan, has the same id value with the polygons marked on the layer above or below.

The points that fall inside these polygons are saved along with their attributes dictionaries as tuples in a list. This list is used in the *Graph* class method, *loadgraph*, in which a nested loop checks each *"lift"* node against all the others and the node whose level is smaller or bigger by a value of 1 (meaning that the floor is either below or above) the coordinates of the two points are added as nodes of an edge. Listing 3.6 displays the *for* loop that iterates through a list whose elements are tuples containing the coordinates of the points that fell within polygons marked as lifts - type = "L". Because the two coordinates are not part of an Euclidean 2-dimensional space, calculating the distance as described in the section above is not possible then a static value is assigned (on line 4).

```
1  for lift in self.net.lifts:
2    for other_lift in self.net.lifts:
3      if (lift[1][ID] == other_lift[1][ID]) and abs(int(lift[1][LEVEL]) - int(
          other_lift[1][LEVEL])) == 1: #check the id first and then the level
4        attributes[WEIGHT] = LIFTSWEIGHT
5        attributes[LEVEL] = LIFTSID #assign "L" instead of an integer to the level
6        self.net.add_edge(lift[0], other_lift[0], attributes)
```

Listing 3.6: Iterating through the nodes marked as lifts

While the running time of the above code is O($n^2$) because of the linear search through all the elements of the list, it is important to remember that this operation is executed only once - when loading the graph into memory.

This chapter described the design and implementation details for growing a graph from geometry features.

# 4. Shortest-Path Algorithm

## 4.1 Introduction

Given a weighted undirected graph grown from geometry features located in the shapefiles, as described in the previous chapter, a search algorithm can be applied to find the shortest-path between two nodes. The algorithm implemented in this project is *Bidirectional Dijkstra*, whose implementation is described in the sections below.

Dijkstra's algorithm [4] is a single-source shortest-path algorithm and an application of dynamic programming. Its search expands in a sphere-like manner, identifying the shortest distance to any of the neighboring nodes on its way to the destination node.

Similarly, bidirectional Dijkstra, expands all the nodes in a sphere-like manner but from, both, directions, the source and the target.

The shortest-path may not be identified in the case of having *negative* edge weights, which does not apply in our case because the distance between two points is calculated through Pythagoras' theorem (see Section 3.7).

A second case in which the shortest path cannot be identified is when the nodes are not connected to the graph. This represented the main reason for connecting the linestring geometry features that fall within a radius, as described in Section 3.2.

The implementation of the algorithm follows the design and pseudo-code found in *Introduction to Algorithms*[17], *Python Algorithms*[18] and, David Eppstein's graph representation [19].

## 4.2 Design

Choosing the data structures to be used for the algorithm led to further investigations of the time complexity for various operations on data structures. Listing 4.3 displays the time complexity for every line/operation of the algorithm.

The dominant data structure in the algorithm is the dictionary. The table below consists of the Average Case times for dict objects, most of which are constant time, assuming that the hash function is sufficiently robust to avoid collisions.

| Operation | Average Case |
|---|---|
| Copy | O(n) |
| Get Item | O(1) |
| Set Item | O(1) |
| Delete Item | O(n) |

Compared with other data structures such as lists, dictionaries are faster.

## 4.3 Bidirectional Dijkstra Implementation

The algorithm function takes four parameters:

- a Python dictionary object representing the graph ($G$),
- source, a node under the form of a tuple containing point coordinates,

- target, a node under the form of a tuple containing point coordinates,
- weight, a string indicating the key in the edges attributes dictionary.

The algorithm function, defined separately and imported into the *Graph* class, is called in the method in Listing 4.1, which in turn is called from the *amissa* script that receives AJAX request from the client-side, further described in Chapter 5.

```
    def bdijkstra(self,srcQuery, dstQuery):
        stype,sid = self.getTypeAndId(srcQuery)
        dtype,did = self.getTypeAndId(dstQuery)
        src = self.getCoord(stype,sid)
        dst = self.getCoord(dtype,did)
        if src is not None and dst is not None:
            li = []
            length,path = bidirectionalDijkstra(self.net,src,dst)
            for i in range(len(path)):
                li.append(path[i])
            return li
```

Listing 4.1: Parsing a string to coordinates before running the search algorithm

The function is being passed two parameters, of string type, and in order to extract the coordinates of that string a parser method is called for both source and destination strings. *getTypeAndId* method makes use of regular expressions (see Listing E.1 in Appendix E) for checking the contents of the string to return a type and an id. This method is particularly useful for queries such as *"Square 1", "sq1", "LTB1" or "ltb 6"*.

Once *getTypeAndId* returns a type and id, they are passed to *getCoord* method that returns the coordinates tuple, its implementation is available in the *Graph* class listing in Appendix F.

In the sections below code listings are used to briefly describe the implementation of the algorithm, the entire implementation being available in Appendix *B*.

```
    dists =  [{},                {}]
    paths =  [{source:[source]}, {target:[target]}]
    border = [[],                []]
    seenNodesDists =   [{source:0},        {target:0} ]
    heapq.heappush(border[0], (0, source))
    heapq.heappush(border[1], (0, target))
    neighs = [G.neighbors_iter, G.neighbors_iter]
    finalpath = []
    direction = 1
```

Listing 4.2: Bidirectional Dijkstra data structure initialisation

The initialisation of the data structures used in the algorithm takes constant time, O(1).

Listing 4.2 shows the data structures used in the algorithm:

- *dists*: a list of two dictionaries, for searching from the source, and from the destination.
- *paths*: a list of dictionaries for each direction that stores the path from both, the source and the destination
- *border*: a heap, containing tuples under the form of (distance, nodes) of the nodes next to expand. The nodes are under the form of coordinates (x,y,x).
- *seeNodesDists*: a dictionary of distances to nodes already expanded, seen
- *finalpath*: variable to hold the shortest, discovered path
- *neighs*: contains a list of two references to the *neighbors_ iter* method of the Graph object. The iterator method returns the vertex's neighbours.

- *direction*: indicating the direction used for expanding the search. It can be either 1 or 0.

Using heapq module, on lines 5 and 6, the source and target values with the distance 0 are pushed into the heap while maintaining the heap invariant. Heaps are binary trees in which every parent node has a value equal or less than that of its children. The most used data structure in the algorithm is the dictionary. The access time for dictionary is smaller compared to that of a list or tuple: O(1) versus O(n).

A difference from the pseudo-code presented in *Introduction to Algorithms*[17] is that the pop return value does not return the largest (similar to max-heap) but the lowest value in the heap.

```
1       while border[0] and border[1]:
2           direction = 1-direction
3           (dist , v )= heapq.heappop(border[direction])        #O(log(len(border[direction])
                ))
4            if v in dists[direction]:                         #O(len(dists[direction]))
5                continue
6            dists[direction][v] = dist                    #O(len(dists[direction]))
7            if v in dists[1-direction]:                        #O(len(dists[1-direction]))
8                return finaldist ,finalpath
9            for w in neighs[direction](v):                #O(len(neighs[direction](v)))
10               if direction==0:
11                   minweight=G[v][w].get(weight ,1)          #O(1)
12                   vwLength = dists[direction][v] + minweight   #O(1)
13               else:
14                   minweight=G[w][v].get(weight ,1)
15                   vwLength = dists[direction][v] + minweight   #O(1)
16               if w not in seenNodesDists[direction] or vwLength < seenNodesDists[
                     direction][w]:#O(n) + O(1)
17                   seenNodesDists[direction][w] = vwLength
18                   heapq.heappush(border[direction], (vwLength,w))
19                   paths[direction][w] = paths[direction][v]+[w]
20                   if w in seenNodesDists[0] and w in seenNodesDists[1]:
21                       totaldist = seenNodesDists[0][w] + seenNodesDists[1][w]
22                       if finalpath == [] or finaldist > totaldist:
23                           finaldist = totaldist
24                           revpath = paths[1][w][:]
25                           revpath.reverse()
26                           finalpath = paths[0][w] + revpath[1:]
27      return False
```

Listing 4.3: Operations average running time for Bidirectional Dijkstra

Following the initialisation of the data structures, a while loop iterates through the border heap as long as they are not empty (the are nodes to be expanded). Listing 4.3 displays the algorithm implementation along with the running time for each operation on the right-hand side.

On line 3, the node with the smallest distance (the closest) is extracted from the border heap through the *heappop* operation. The *heappop* operation is logarithmic (O(log n)) because after it returns the root of the heap, in order to maintain the heap structure, the last item is moved to the root position and then compared with each of its children and swapped downwards until it is smaller than both its children. The same time, in the average case, is taken by the *heappush* operation.

First, it is important to check whether the node extracted has been found by checking whether it is in the dictionary of final distances on Line 4. Should *v* be in *dists*, the shortest path to *v* has already been found. If the vertex *v*, is also found in the dictionary of final distances of the opposite direction (on line 7) it means that the shortest path has been found and the final distance and path can be returned.

For every neighbour *w* of *v*, returned by the iterator method of the Graph object (on line 9), the weight of the *v->w* edge is retrieved on line 10, using a getter to retrieve the weight from the *attributes* dictionary. However, should the direction be set to forward, the edge is *w->v* and the weight is extracted accordingly. *vwLength* contains the shortest path to *w*, its value being the sum of the *v->w* or *w->v* edge weight and the distance to node *n*.

The condition statement which checks if vertex *w* has been seen already or if the current distance to *w* is shorter than the one already discovered. The following part of the algorithm is also called *relaxation*.

The relaxation procedure, starting at line 21, updates the cost of distance to *w* first, followed by the *heappush* operation for inserting the node into the border heap (line 23) and also update the path to include *w* (line 25).

Should node *w* be seen by the search in both directions (line 26), we check if this path is better than the path already discovered (line 28). If the conditional statement returns true the distances are updated accordingly (lines 29 -32).

# 5. Projection

## 5.1 Introduction

Projecting query results and directions across different levels on the web presented a challenging task from a User Experience (UX) point of view. There are two types of queries a user makes when using a digital map:

- single-location - for identifying the location of a single point on the map.
- directions between two locations - for displaying a route between two interest points on the map.

For the first type of query, highlighting the polygon that matches the user query string is possible through Mapserver's query capabilities part of the Mapscript API. The challenge resides in rendering the results for the second type of queries: directions between two locations. While directions between two locations on the same level is common to all mapping applications, displaying the route across multiple levels is unique. Levels and floors are used interchangeable in this chapter.

This chapter details how Mapserver is used for projecting results for both types of queries on the web. The implementation details are divided in two main sections:

- **Server-Side** implementation
- **Client-Side** implementation

Dracones is the higher-level component for Mapscript whose implementation extends in both, client and server side.

## 5.2 Design

Figure 5.1 displays the layered architecture used in the project. Each floor has a number of different layers defined in OpenEV that are overlapping each other when rendered.

The top most layer is the *directions* layer which will render *line* geometry features created from the coordinates resulted from the search algorithm. Should the source and destination be on different floors, the path will be displayed between the source and a lift/staircase on the level of the source and continued from the list/staircase to the destination on the destination level.

The second layer, the rooms layer, contains only *polygon* shapes. This layer has also been used in the preprocessing algorithm.

The last two layers, *buildings* and *outdoors*, are rendered at the bottom, before of all the others. These layers serve as an outline of the buildings and the (outdoors) open-space areas such a the squares or a bridge between buildings.

The reason for separating the last two layers although they share the same type (polygons) is for **security, integrity and synchronization**. Different access levels may be required for users editing the vector layers. For instance, the *buildings* and *outdoors* layers are read-only and can only be edited by UNIX users with *sudo* privileges or system administrators. The file permissions can be changed using the *chmod* command for each of the shapefiles.

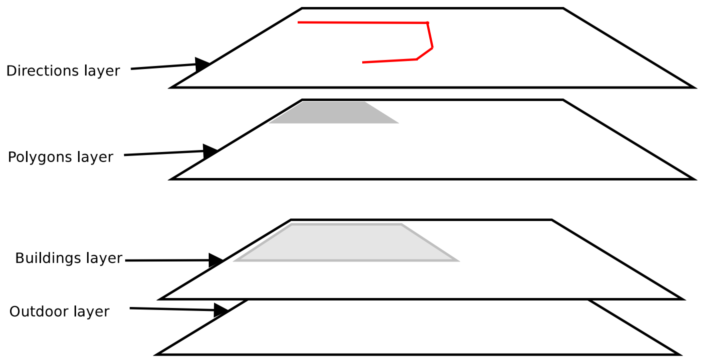


Figure 5.1: Layered architecture representing a single floor

Mapserver application aggregates the layers for generating an image. The resulting image is displayed in a browser on an HTML page in a *<div>* container. The controls defined for the front-end are:

- textboxes, for entering a string when searching for a point on a map, but also for entering two distinct location between which the route will be displayed.
- buttons, used for submitting the value in textboxes to the server-side for processing the request. Buttons are also used for switching between floors.

## 5.3 Server-Side

### 5.3.1 Dracones

Dracones is the main component at the server-side for dealing with the requests coming from the browser. It is divided in two parts: core and application.

The core part consists of the wrapper of the Mapscript library and a Javascript library. The Javascript library is used on the client-side in order to make calls asynchronously to the server. The wrapper functions are extended by a *web_interface* which is extended by the application script.

The Dracones application part, consists in a script file that extends the *web_interface* and any other developer-defined packages (as Graph in our case). The functions defined in this script (under the name of *amissa.py*) represent the gateway between the client-side HTML web page and the algorithms and Mapserver of the server-side.

The application script imports two modules: *web_interface* and *graph*. The first module is an interface of the wrapper through which the methods defined in the core part of Dracones can be accessed. Through the imported graph package, calls to the shortest-path algorithm as well as to other methods (i.e. parsing the string or retrieving the level of coordinates) are made.

Every function defined in the script follows the design shown in below:

```
def <function_name>(req):
    params, sess, dmap = beginDracones(req)
   #logic implemented here

    json_out = endDracones(dmap)
    return exitDracones(json_out)
```

Listing 5.1: Sample of Dracones application script function design

Each function takes a *req* object as a parameter, that is passed to the *beginDracones* method defined in core.py, in order to return three variables, *params* (CGI parameters),*sess*(current session) and *dmap* (the map object on which all the calls are made). *json_out* is the variable that stores anything that needs to be returned to the client, in this case, the modified map object.

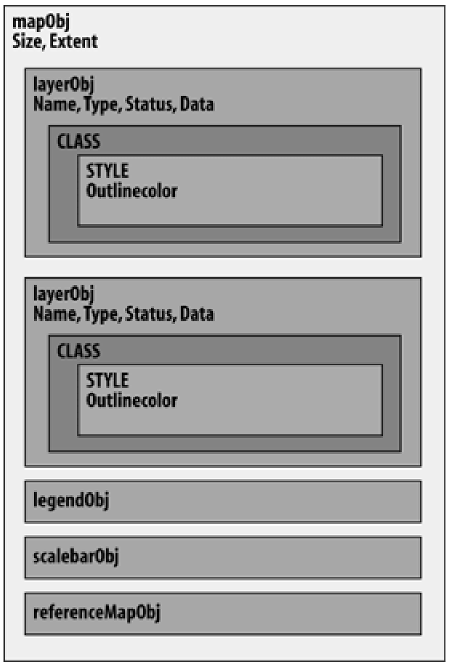


Figure 5.2: Mapscript API hierarchical object structure

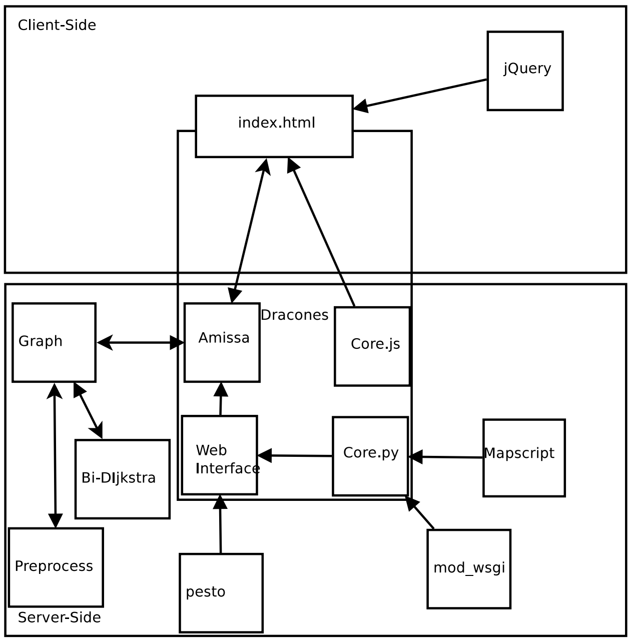


Figure 5.3: The Dracones environment between on the client and server

The directions were implemented by first defining a function in the script which takes as parameters two strings representing the source and destination, as shown in Listing 5.2.

```
 1 dispatcher.match('/directions', 'GET')
 2 @simple_tb_catcher
 3 def directions(req):
 4     params, sess, dmap = beginDracones(req)
 5     source = str(params.get('source'))
 6     dest = str(params.get('dest'))
 7     dmap.clearDLayer('2D','all')
 8     dmap.clearDLayer('3D','all')
 9     dmap.clearDLayer('4D','all')
10     dmap.clearDLayer('5D','all')
11     if source is not None and dest is not None:
12         points = graph.g.bdijkstra(source,dest)
13         level = graph.g.getLevel(source)
14         if level is '3' or level is '4' or level is '5': #show the outdoors for the
               selected layers
15             if dmap.dlayers[level+'O'].getStatus() == MS_OFF:
16                 dmap.dlayers[level+'O'].setStatus(MS_ON)
17         if level is None:
18             level = graph.g.getLevel(dest)
19         if level is not None:
20             dmap.dlayers[level+'B'].setStatus(MS_ON)
21             dmap.getDLayer(str(level)).queryByAttributes('id',dest,"{type} {id}")
22             dmap.dlayers[level].setStatus(MS_ON)
23             dmap.dlayers[level+'D'].setStatus(MS_ON)
24         if points is not None:
25             for c in xrange(len(points)-1):
26                 lv = graph.g.getCoordLevel((points[c][0],points[c][1],0.0))
27                 if lv is not None:
28                     dmap.dlayers[lv + 'D'].drawLine(points[c][0],points[c][1],points[c
                           +1][0],points[c+1][1],False)
29     json_out = endDracones(dmap)
30     return exitDracones(json_out)
```

Listing 5.2: Server-side execution of directions function

The first line matches the URL of the HTTP GET request to the function. On line 2 there is an exception handling method whose purpose is to catch any exception thrown inside the function and route them back to the core script for printing them in an HTML format.

The *params* object is used on lines 6 and 7 to extract the data sent from the client-side function call. The group of *clearDLayer* calls are used to clear the direction layers of any geometry features before creating new ones. Each floor has its own *"directions"* layer reserved for creating line geometries representing path. This is required for displaying level specific directions.

On line 12, a call to the search algorithm is made in which the source and destination strings are passed as parameters. The result of the call is being assigned to a list called *points*.

On line 13, the level of the souce is retrieved through a call to the *getLevel* method which return the level number for a given string. The implementation of the *getLevel* method is described in Appendix F.

On line 14, a conditional statement checks if the level number retrieved is one of the 3 levels on which there are squares. If the conditional statement yields true, the status of *outdoors* layer, marked by the letter "O" after the level number, is set to ON.

Should the source not be identified and the level be "None" an attempt to retrieve the level of the destination is made on line 18. Otherwise, the *buildings* layer status is set ON (line 20), the destination polygon and its layer is made visible (on line 21) and also the *directions* layer has its status set to ON.

On line 24, a for loop iterates through all the elements of the *points* list. The level for each of the coordinates is retrieved and assigned to the *lv* variable (on line 25).

on line 26, a line is drawn between two elements of the *points* list. The level on which that line is drawn is stored in the *lv* variable and is updated on every iteration.

The *queryByAttributes* is defined in the Mapscript API and takes a minimum of two parameters: the name of the attribute and its value. For instance, in the function described above, if the destination is "4S.6.28" that string is being as the value of the *id* attribute, and can be used to render only the polygons that share this value as their id.

*queryByAttributes* is the method used for providing results for single-location search queries.

### 5.3.2 Mapfile

The mapfile, listed entirely in Appendix C, is the most important file for Mapserver and, inherently, for projection. Its structure, being similar to that of XML, starts with a MAP tag containing parameters such as the NAME, STATUS, TYPE, DATA, but also the definition of each layer to be rendered through the LAYER tag. Some of mapfile's most used tags in the project are listed below:

- **TYPE** which defines the type of geometry for *each* layer. The rendering of the geometry types is dependent on the value specified for this parameter. This represents the reason for which only one type of geometries can be added per layer in OpenEV.
- **DATA** indicates the location of the shapefile for a specific layer. This parameter resides inside the LAYER tag.
- **STATUS** represents the default projection status of a vector layer, if set to OFF it will not be rendered. The DEFAULT or ON values will render the respective layer.
- **STYLE** which resides inside of the *CLASS* tag, that also resides inside the LAYER tag. It defines the styling characteristics of the layer's geometry features, such as *COLOR, SIZE, OUTLINECOLOR.* Using this tag, different colours are assigned to different layers: the buildings and outdoors layers have a different colour compared to the polygons and directions layer. The value assigned is based on the RGB colour module and requires three integers, as show in the sample listing below.

```
    CLASS
        STYLE
            COLOR 159 159 125
            OUTLINECOLOR 100 100 100
        END
    END
```

Listing 5.3: STYLE tag of a LAYER inside the mapfile

## 5.4 Client-Side

On the client-side, the map is loaded in a *<div>* asynchronously in an *HTML* page. The web page presented to the user, uses the jQuery and Dracones core.js libraries for two reasons:

- for passing and receiving the data inside the HTML elements. Using the Dracones core Javascript library a mapWidget object is instantiated and used for manipulating the mapfile through Dracones' core Mapscript wrapper. Furthermore, custom requests to functions defined in the application script (amissa.py) are made through asynchronous calls to the Dracones' core Javascript library.
- for scrolling and panning. Each map generated by Mapserver has an *extent* value, which controls the ares of the map that is visible to the user, similar to a porthole. The extent value, previously controlled based on the radio button control being checked and the mouse clicked on various points, is now controlled by the mouse action events such as scrolling.

Once a function has been defined in the script on the server-side, as discussed above. In order to call that function from the client-side, the Dracones core Javascript library is used to instantiate a map object on which function calls can be made.

Listing 5.4 reveals the instantiation of the map object, the *mapWidget*(part of Dracones core Javascript library), with some mandatory parameters: mapfile name(line 6), mid (Map Widget ID, in case of having more than one map instance per page)(line 5), the name of the application as it appears in the URL and in the Apache configuration(line 4) and the <div> element in which the images will be loaded. All the layers used must be initialised first regardless of their existing status in the mapfile.

```
1   jQuery(document).ready(function() {
2         var map = new dracones.MapWidget({
3             anchor_elem: 'map_div',
4             app_name: 'amissa',
5             mid: 'instance1',
6             map: 'amissa',
7             init_dlayers: ['2','3','4','2B','3B','4B','3O','4O','5O','3D','4D','5D']
8         });
```

Listing 5.4: Dracones Javascript MapWidget instantiation

After instantiating the MapWidget, the map variable is used to make request calls to the functions defined in the *amissa.py* script. The function calls are binded to events triggered by the HTML controls and can send function parameters, wrapped in a dictionary, or only execute one of the functions defined in the custom script.

```
1         jQuery('#directions').bind('click',function(){
2             map.setDLayers({off:['2','2B','3','3B','3O','4','4B','4O']});
3             map.customRequest({
4                 url:'/amissa/directions',
5                 data: {source:[document.getElementById("fromBox").value],dest:[
                      document.getElementById("toBox").value]}
6             });
7         });
```

Listing 5.5: Dracones Javascript Custom Request for retrieving directions

Listing 5.5, illustrates how a call for the directions function implemented on the server-side, is implemented on the client-side. The click event for the *directions* HTML control is binded to a function. On line 2, the status of the layers are set to be OFF in order not to display overlapping floors. Line 3 calls a *customRequest* in which the function defined in the amissa.py (on the server-side) is called by the URL. Thus, the URL first contains the name of the application, *amissa*, followed by the name of the function that needs to be called. Line 5 displays how the data, if any, is also passed to the function in the request: as a dictionary. The dictionary key is mentioned on the server-side to retrieve the value of that key:value pair.

The above function call is made asynchronously to the *amissa* script, meaning that the result of the function will be returned in the <div> container specified upon the instantiation.

# Part III

# Conclusions and Evaluation

# 6. Testing and Evaluation

## 6.1 Test-Driven Development

Test-Driven Development [20], consists in writing failing test cases for the software that is to be produced first, and then proceed to development in order to pass these tests. The major advantage of this approach to software development consists in clearly defining the functionality of a method/class before it is being developed. Once the test are passed, the code is re-factored such that it meets the industry standards. Test cases were built for all the classes and methods developed (the search algorithm function, *Graph* and *Preprocess* classes), except for the Dracones script used for projection.

A sample unit test is shown in Listing 6.1. *assertEqual* method is used to call the *computeWeight* with two sample values, the last parameter being the result expected as return value from the *computeWeight* method. The *computeWeight* method computes the Euclidean distance between two points and is being implemented in the *Preprocess* class.

```
import unittest

class testGraph(unittest.TestCase):
  def setUp(self):
    net = Preprocess()

  def testComputeWeight(self):
    self.assertEqual(net.computeWeight((1.1111, 3.44444)(2.2222,4.33333)),3.14666195)
```

Listing 6.1: Unit Testing for Calculating the Euclidean Distance between two points

Before the unit test method is executed, a *setUP* method is used to load any objects, assign variables and prepare the environment for testing.

Another type of testing conducted has been the integration tests. After each change to the software, black-box testing of all the functionalities/features was conducted. Furthermore, the test cases also had to be passed.

## 6.2 Usability Evaluation

Usability evaluations were made across the software development process in order to identify further improvements that can be made and supply a product that meets the users' needs and expectations.

There are several usability evaluation types [21]:

- Expert evaluations
- Observations
- Interviews
- Enquiries
- Think-aloud
- Cognitive walk-through

- Usability laboratory tests

The project had planned usability evaluations after each milestone achieved. The first usability evaluation has been conducted in academic week 10, once the Mapserver, static, CGI application with sample maps has been developed. The usability evaluation consisted in an interview with 2 users which resulted in the following issues being identified:

- scrolling and zooming are not intuitive
- response time is too high

To address the above issues, Dracones component was implemented which does scrolling and panning interactively, but also returns results asynchronously - faster than the Mapserver CGI deployment.

For the second milestone, the deployment of Dracones, the usability evaluation was conducted in the academic week 24 on 3 users, under a Think Aloud Protocol (TAP)[22]. TAP consisted in the users saying whatever they are thinking, feeling or doing when performing a task. This allowed me to observe and understand the task-completion process.

The evaluation resulted in the following issues being discovered:

- floors numbers are not distinguishable
- directions across different floors are not displayed intuitively - for every floor.
- errors are rendered accordingly
- zooming "freezes"
- pop-ups required when hovering over a query result

The second issue, *directions across different floor are not displayed intuitively* led to creating tasks for developing a solution immediately, the directions now being displayed for every floor.

Both evaluations led to further improvements to be made to the application.

# 7. Conclusions

## 7.1 Project Management

### 7.1.1 Agile development

The project has been developed following an agile methodology. The *waterfall* development life cycle was not suitable for this project due to the constant learning rate of the various components which led to numerous changes to the design and specification even after the Software Specification had been written.

The agile software development methodology, namely XP (Extreme Programming), and its principles provided the project a more accelerated development pace and responsiveness to change. Agile software development prioritises people and interactions over tools and processes making the implementation more interactive and becoming more attached to the project outcomes without the need of focusing on the bureaucratic process of documentation.

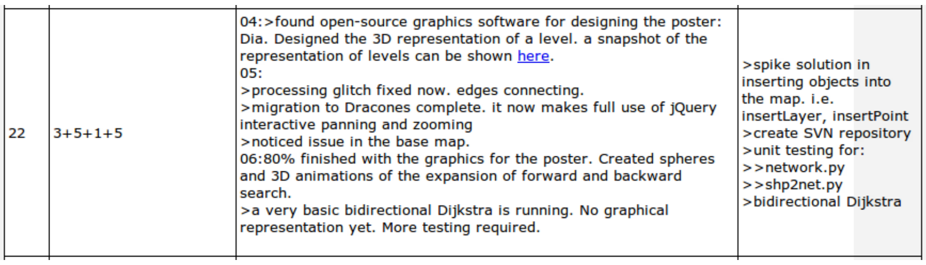
Such example would be the decision to alocate more resources in developing the preprocessing algorithm even if its design and implementation were not originally specified in the Software Requirements Specification. Another example would be the number of layers required to represent each floor which also has changed due to reasons mentioned in Chapter 5.

Changes in the project occurred frequently, reason for which XP proved to be the best choice for software development.

### 7.1.2 Time Management

While originally a Gantt chart has been developed, the agile practices focuses on estimated man-hours for implementing user stories, the chart loosing its value and purpose. User stories represent high-level requirements artifacts retrieved from a discussion with the customer. The user-stories were broken down into tasks which were listed in a *time log* section of the log-book on a weekly basis as a *TO-DO list*, prioritised accordingly. As shown in Figure 7.1, every (academic) week had a row in the time-log briefly listing achievements, the number of man-hours for implementing the achievements, and a list of tasks for the upcoming week or the remaining of current week. The achievements were also grouped by days of the month in some cases.

A man-hour is the amount of uninterrupted work performed to complete a task. Man-hours do not take in account breaks from work such as rest or eating. Keeping track of time spent on the project on a weekly basis allows an overview of hours spent and how I was affected by other modules. It also aids



| | | | |
|---|---|---|---|
| 22 | 3+5+1+5 | 04:>found open-source graphics software for designing the poster: Dia. Designed the 3D representation of a level. a snapshot of the representation of levels can be shown here.<br>05:<br>>processing glitch fixed now. edges connecting.<br>>migration to Dracones complete. it now makes full use of jQuery interactive panning and zooming<br>>noticed issue in the base map.<br>06:80% finished with the graphics for the poster. Created spheres and 3D animations of the expansion of forward and backward search.<br>>a very basic bidirectional Dijkstra is running. No graphical representation yet. More testing required. | >spike solution in inserting objects into the map. i.e. insertLayer, insertPoint<br>>create SVN repository<br>>unit testing for:<br>>>network.py<br>>>shp2net.py<br>>bidirectional Dijkstra |

Figure 7.1: A time log entry in the logbook for academic week 22

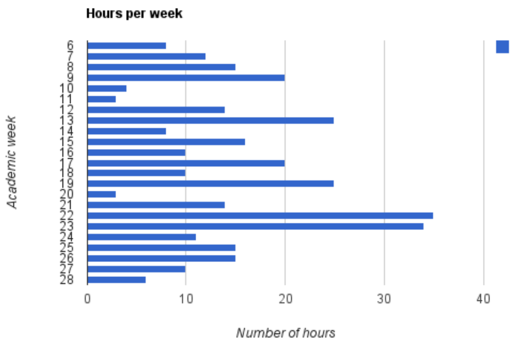


Figure 7.2: Number of man-hours per academic weeks

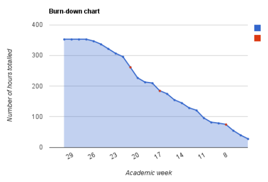


Figure 7.3: Burn-Down Chart

in identifying where delays occurred and try not to repeat them in order to improve. Unfortunately, the time-log records the number of hours starting with week 6, when the time-log was added in the log-book. The number of hours invested in the project before week 6 can be approximated to 20. Within this period the knowledge about Geographic Information Systems was consolidated for documenting the *initial report*.

Cumulating the number of hours recorded in the time-log results in plotting the burn-down chart in Figure 7.3. The *Burn-Down* shows the week over week effort in the project. Analysing the trend it can be noticed that there are weeks in which fewer than 5 hours were recorded (week 18), possibly due to overlapping with university coursework. The scheduled breaks (i.e. Christmas) are also visible (such as week 11) with no progress recorded.

## 7.2 Implementation

Throughout the project I had to familiarise myself with various APIs and cartographic standards. With the knowledge gained I developed the algorithms required to be able to use GIS software products together as a system. Therefore, the time dedicated developing the software was much less than the time required for learning the language, documentation of the various libraries, and the cartographic software products.

The architecture and design employed for this mapping application is unique in the open-source GIS community and the code-base developed (the preprocessing and search algorithms) can be successfully reused in other applications because it uses cartographic libraries used in the industry.

To deploy a Python web application, without any framework, on a LAMP architecture also required learning about Apache configuration and the 2-tier Python web architecture environment. I would certainly recommend a shorter route to get a Python application deployed: by making use of frameworks that are much easier to deploy, but provide *less* control.

Using Dracones for Mapserver provided the necessary library to create new geometry features from the coordinates of the shortest-path algorithm. The OGR library is the standard library used by cartographers to analyse and manipulate vector formats. I used this library for creating the

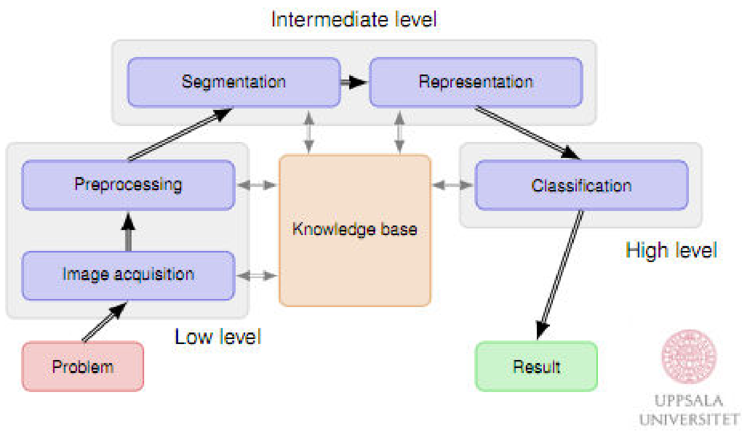


Figure 7.4: Steps used in Image Analysis

preprocessing algorithm for growing a graph from the vector layers. These vector layers (ESRI Shapefiles) are analysed or manipulated by OpenEV.

The *Algorithms and Data Structures* module taught in my second year, at Uppsala University, has been of great use for understanding graphs, data structures and Dijkstra. The *Business Information Systems* and Large-Scale Software Systems modules taught at University of Essex were also very useful for the project, particularly Large-Scale Software Systems from which I learned about Test-Driven Development and version control software with Subversion.

The *Information Systems* module taught by Keith Primrose during the first year provided the basis for developing applications on the web.

Numerous technical challenges have been encountered due to the use of only open-source software. However, the skills gained as well as the independence from other proprietary platforms provide an invaluable personal and professional advantage.

The project helped in developing my research skills, by observing concepts, tools and creatively identify solutions suitable for a problem. Given that the some of the concepts are not taught at University of Essex, the project relied on the ability of learning and locating resources individually. This experience providing more confidence to approach projects in which I have little or no background in the future.

## 7.3 Future Work

This section presents some ideas that can be further implemented in the future. Every implementation chapter has a sub-section. Therefore, for each chapter one or more extensions are presented.

### 7.3.1 Preprocessing

#### 7.3.1.1 Image Analysis

Image analysis is used for retrieving information from an image based on its characteristics. It is different from *image processing* which simply modifies the image, usually with the purpose of improving it. Future work on the project can make use of image analysis in order to create the geometry features with attributes retrieved from the image and, thus, eliminating the need to have a back-end user defining the shapes in OpenEV.

The main steps in image analysis are shown in Figure 7.4. This section will briefly discuss the potential application of image analysis to this project.

Using image analysis on the raster file that represents a floor plan, the need for drawing the lines in OpenEV can be eliminated. This can be achieved using Machine Learning techniques for pattern

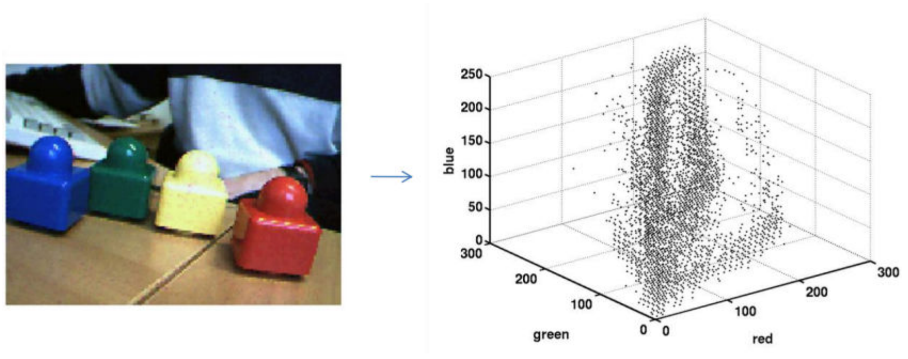


Figure 7.5: Sample 3D RGB Scatterplot - Feature space

recognition and classification. Gonzlaez and Woods book on *Digital image processing*[23] provides a thorough introduction, concept details and comparison as well as implementation strategies.

Classification consists of individual items (objects, patterns, image regions or pixels) that are grouped based on the similarity between the item and the description of the group. A *pattern* consist in the arrangement of descriptors, descriptors that are often called *features*. The most common pattern arrangement being a feature vector with $n$-dimensions. The patterns are placed in *classes* of objects that share the same properties.

The raster format file provided as a floor plan can be used for feature extraction and classification. Using the RGB colour model, the features of an image can be extracted in a 3 dimensional feature space. Figure 7.5 represents a sample of a 3D scatter-plot based on the RGB values of an image.

A similar feature space can be built from the images provided as floor maps. But choosing the right features can make the difference between a successful and unsuccessful classifier. Therefore, reducing the dimensionality space of features represents an essential step before any analysis of the data can be performed. The relevant information from the original dimensions must be preserved, however, according to some optimality criteria. Generally, for pattern recognition and classification problems methods such as Principal Component Analysis (PCA), Independent Component Analysis (ICA) or Fisher linear discriminate analysis are used in order to reduce the dimensionality of a problem.

Classification can be object-wise or pixel-wise. Object-wise classification uses shape, size, mean intensity, mean colour etc. to describe patterns while pixel-wise classification uses intensity, colour, texture or spectral information. Pixel-wise classification is the most suitable for the project as an extension because the floor plan is an image that can be splitted into a number of patterns (depending on the image size - i.e. 256 x 256 pixels = 256 x 256 patterns), a number of features (RGB - red, green and blue band) and a number of classes (room number, door, corridor, stairs, lifts, squares or building outlines).

The features for each pattern (pixel) are calculated according to its colour in order to train a classifier. Any new samples are classified by the classifier. There are two approaches for classification:

**Supervised**
: This technique includes: box classifiers, Bayes classifiers (minimum distance, maximum likelihood) or neural networks (i.e. Multilayer Perceptron)

**Unsupervised**
: Clustering is one of the unsupervised methods that can be used: k-means or hierarchical clustering.

The supervised learning technique neural networks, can be used such that regions of the image can be selected for training, resulting in further images being classified according to the training set.

Another type of classifiers that can be considered are the Bayesian classifiers which are based on a *priori* knowledge of class probability, cost of errors and whose combination gives an optimum statistical classifier.

The above methods are supervised learning techniques, but other techniques such as reinforcement learning can also be used.

#### 7.3.1.2 Geometry features creation

Once the classification of image analysis is completed, the results can be used in creating geometry features with specific attributes depending on the classification results. Essentially, the use of OpenEV would be limited for correction purposes only.

However, while creating a geometry feature such as polygon with the coordinates resulted from the image analysis, defining the paths between different points is difficult. At the moment, the paths between two polygons are user-defined and it is up to the OpenEV user to judge how a path over one of the squares should drawn so that it is the shortest. The results of the image analysis can be used to define lines on the corridors simply by calculating the middle point between the boundaries of the polygons representing the rooms, and in front of each door to create two additional lines that connects the polygon to the polyline representing the corridor.

The complexity increases, however, when large open spaces such as the squares must have connecting lines between different points in order to compute the shortest-path between two points across the square. One solution would consist in creating a mesh of lines between all the exits and intersection points located at the square (open-space) boundaries.

### 7.3.2 Search algorithm

#### 7.3.2.1 Bidirectional A*

Given the fact that the search is started both from the destination and the source, the algorithm performs better single-source *Dijkstra*. However, even faster results can be achieved by employing heuristics in the bidirectional search [6].

The implementation of *bidirectional A** is very similar to that of bidirectional Dijkstra, the difference being the heuristic which estimates the distance to the target that adds up to the shortest distance to given node. The reason A* algorithm is faster and still maintains precision is due to its combination of both the distance to a vertex $v$ (g(n)) and the heuristic with the estimated cost to the target - f(n). Thus, each iteration of a loop will examine the vertex that has the lowest value for f(n):

$$f(n) = g(n) + h(n) \tag{7.1}$$

The value of h(n) would be assigned to the edge during preprocessing, but the search algorithm will also take a parameter for the heuristic. One heuristic used by *Ikeda et. al*[6] is the Euclidean distance which is already used in this project for the distance between two coordinates. A potential obstacle for bidirectional A* is the combination of bidirectional search and heuristics for searching in both directions. This issue has been addressed through dual feasibility, which divides the source and target heuristics and appends them to the length found to the current node. Therefore, the following formula is applied for modifying the original edge length (*l(u,v)*) with a dual feasible heuristic estimator that starts from the source (hence the *s* indicator, *h(s)*:

$$l'(u,v) = l(u,v) + h_s(v) - h_s(u) \tag{7.2}$$

But the above equation only includes the estimators for forward search, from the source, the equation adapted to include estimators for forward as well as backward search is:

$$l'(u,v) = l(u,v) + \frac{1}{2}(h_s(v) - h_s(u)) + \frac{1}{2}(h_t(u) - h_t(v)) \tag{7.3}$$

The modified length between the two vertices, $u$ and $v$, will always be positive due to dual feasibility of h(s) and h(t) - the heuristic evaluators of the search expansion from the start and target nodes.

#### 7.3.2.2 Accessible Routes

The current version of the algorithm deployed calculates the shortest-path between two points without evaluating the accessible binary value already existent in the attributes *dictionary* of both edges and nodes.

Extending the search algorithm function to include accessible routes would involve the following changes:

- passing an additional parameter in the data dictionary from the client-side function call.
- the search algorithm function will also take one additional parameter. The parameter will default to unaccessible (similar to how the weight parameter defaults to the *WEIGHT* string value). The *accessible* parameter will be used in a conditional statement when iterating over the neighbors of a node n. Therefore, only the edge whose *accessible* value match the parameter will be considered for expansion.

The *attributes* dictionary for nodes and edges already have an *accessible* attribute that can be used for implementing this extension.

### 7.3.3 Projection

#### 7.3.3.1 User Interface Refinements

One major improvement for the user interface would represent eliminating the two textbox controls required for directions and keep only one single textbox control and a search button. To achieve this, the client-side function calls would not require any change; the only change would be in the Graph class where the current *regular expression* function (see Listing E.1 in the Appendix) is used for parsing a string into a *type* and *id*.

The function will first search for the *to* word in the query string, remove the *"to"* keyword and recursively call the function with the string that occurred before and after the *"to"* keyword.

Another UI improvement can be applied to the mapfile by inserting a *raster* layer representing sattelite imagery of the campus surroundings. This layer would be rendered before all the other layers, providing an even more intuitive contextual navigation. This modification will, however, increase the size of the rendered image and also the response time of Mapserver which for every new generated map will also *re*-generate the sattelite raster image. Furthermore, providing a single sattelite view imagery layer is not sufficient for multiple zoom levels. Current digital mapping systems make use of tile-based layer rendering system and different zoom levels have distinct satellite imagery for each different values of the extent (zoom levels).

#### 7.3.3.2 Tile-based rendering

Mapserver generates an image for every query, or change of extent (zooming, panning). The more layers defined in the mapfile that have the status **on** and must be rendered, the more time it requires the Mapserver to generate an image for a given *extent*. Mapping application such as Yahoo! Maps or Google Maps use tile-based rendering for a given map extent which will load tiles of the map image that are ready for rendering instead of *"freezing"* with the previous rendered image while loading the new one.

The candidate application through which this can be achieved is *OpenLayers*. OpenLayers is a Javascript library that does not support mapfiles or shapefiles, but GML (Geography Markup Language) or KML (Keyhole Markup Language) which is also used by Google Maps. Shapefiles can be converted into the required formats by using the *ogr2ogr* command on UNIX. Both, OpenLayers and Mapserver are employing *caching* techniques such that the images for a given extent or query are stored in a temporary folder and if the same request is made at a later time, the already generated image is used. A sample of a map rendered through OpenLayers is displayed in Figure 7.6.

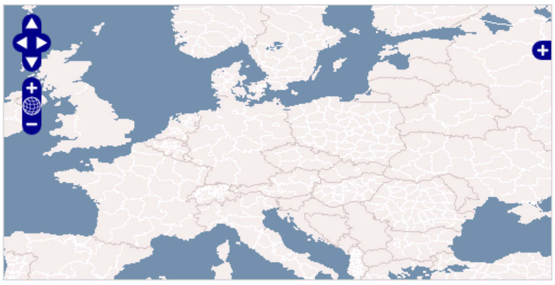


Figure 7.6: Screen Shot of OpenLayers

This extension would lead to a more responsive application that would also cache the tiles for saving bandwidth.

# Acknowledgements

Thanking the supervisor in a supervised written report is among the greatest clichés and as much as I would like not to end my report with a stereotype I really do have to express my gratitude towards my supervisor, *Keith Primrose*, for the guidance, the time management tips and the inspiration required to visualise problems from a different angle.

I would also like to thank Christian Jauvin of University of Montreal for the changes made to *Dracones* and Chris Garrard of GIS Laboratory at Utah State University whose lecture slides and answers to my questions helped in further understanding GIS.

Last, but not least I would like to thank Dr Yevgeniya Kovalchuk for some valuable advice in preparing the report.

# Appendices

# A. Ray Casting Algorithm

```python
def point_in_poly(x,y,poly):

    n = len(poly)
    inside = False

    p1x,p1y = poly[0]
    for i in range(n+1):
        p2x,p2y = poly[i % n]
        if y > min(p1y,p2y):
            if y <= max(p1y,p2y):
                if x <= max(p1x,p2x):
                    if p1y != p2y:
                        xints = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                    if p1x == p2x or x <= xints:
                        inside = not inside
        p1x,p1y = p2x,p2y

    return inside
```

# B. Bidirectional Dijkstra

```

import heapq
from config import *


def bidirectionalDijkstra(G, source, target, weight = WEIGHT):
    if source is None or target is None:
        raise ValueException("Bidirectional Dijkstra called with no source or target")

    if source == target: return 0, [source]

    dists =  [{},                  {}]
    paths =  [{source:[source]}, {target:[target]}]
    border = [[],                  []]
    seenNodesDists =   [{source:0},        {target:0} ]

    heapq.heappush(border[0], (0, source))
    heapq.heappush(border[1], (0, target))

    neighs = [G.neighbors_iter, G.neighbors_iter]

    finalpath = []
    direction = 1


    while border[0] and border[1]:
        direction = 1-direction
        (dist, v )= heapq.heappop(border[direction])
         if v in dists[direction]:
            continue
         dists[direction][v] = dist #equal to seen[direction][v]
         if v in dists[1-direction]:
            return finaldist,finalpath

         for w in neighs[direction](v):
             if direction==0: #forward
                 minweight=G[v][w].get(weight,1)
                 vwLength = dists[direction][v] + minweight #G[v][w].get(weight,1)
             else: #back, must remember to change v,w->w,v
                 minweight=G[w][v].get(weight,1)
                 vwLength = dists[direction][v] + minweight #G[w][v].get(weight,1)

             if w not in seenNodesDists[direction] or vwLength < seenNodesDists[
                 direction][w]:
                 seenNodesDists[direction][w] = vwLength
                 heapq.heappush(border[direction], (vwLength,w))
                 paths[direction][w] = paths[direction][v]+[w]
                 if w in seenNodesDists[0] and w in seenNodesDists[1]:
                     totaldist = seenNodesDists[0][w] + seenNodesDists[1][w]
                     if finalpath == [] or finaldist > totaldist:
                         finaldist = totaldist
                         revpath = paths[1][w][:]
```

```
                        revpath.reverse()
                        finalpath = paths[0][w] + revpath[1:]
    return False
```

Listing B.1: Bidirectional Dijkstra function implementation

# C. Mapfile

```
#define the map object
#

MAP
NAME "Amissa"
EXTENT -19000.0 -8300.0 19000 8300.0
SIZE 640 320
IMAGETYPE gif
SHAPEPATH "../../data/vector/"
WEB
    HEADER "/var/www/htdocs/fourth_web_header.html"
    FOOTER "/var/www/htdocs/fourth_web_footer.html"
    EMPTY "/fourth_empty.html"
    TEMPLATE "/var/www/am/amimap.html"
    IMAGEPATH "/var/www/am/tmp/"
    IMAGEURL "/am/tmp/"
END

QUERYMAP
    STATUS on
    STYLE hilite
    COLOR 255 255 0
    SIZE 640 320
END#queryMap



##########################################################################################
###################################OUTDOOR LAYERS#########################################

LAYER
     NAME "3O"#that is O from outdoors and not zero
     STATUS default
     TYPE polygon
     DATA "outdoors/3"
     CLASS
          STYLE
               COLOR 159 159 125
               OUTLINECOLOR 100 100 100
          END
     END
END


LAYER
     NAME "4O"#that is O from outdoors and not zero
     STATUS default
     TYPE polygon
     DATA "outdoors/4"
     CLASS
          STYLE
               COLOR 159 159 125
```

```
                OUTLINECOLOR 100 100 100
            END
        END
END


LAYER
        NAME "5O"#that is O from outdoors and not zero
        STATUS default
        TYPE polygon
        DATA "outdoors/5"
        CLASS
            STYLE
                COLOR 159 159 125
                OUTLINECOLOR 100 100 100
            END
        END
END


##########################################################################################
####################################BUILDING LAYERS#####################################

######################################LEVEL 2 - NO SQUARES###########################
LAYER
        NAME "2B"
        STATUS default
        TYPE polygon
        DATA "buildings/2"
      CLASS

          STYLE
              COLOR 199 179 175
                OUTLINECOLOR 105 105 105
          END
      END
END

#####################################LEVEL 3 - SQUARE 1#############################
LAYER
        NAME "3B"
        STATUS default
        TYPE polygon
        DATA "buildings/3"
      CLASS

          STYLE
              COLOR 199 179 175
                OUTLINECOLOR 105 105 105
          END
      END
END

######################################LEVEL 4 - SQUARE 2&3###########################
LAYER
        NAME "4B"
        STATUS default
        TYPE polygon
        DATA "buildings/4"
      CLASS

          STYLE
              COLOR 199 179 175
            OUTLINECOLOR 105 105 105
        END
    END
END

################################################################################################
##############################################ROOM LAYERS###################################
LAYER#2
    NAME "2"
    STATUS default
    TYPE polygon
    DATA "rooms/2"
    TRANSPARENCY 50
    FILTERITEM "id"

    TOLERANCE 1
    TOLERANCEUNITS meters

    CLASS

        TEMPLATE "rooms_query.html"
        STYLE
            COLOR 178 34 34
                OUTLINECOLOR 0 100 0
        END
    END

    METADATA
        qstring_validation_pattern '.'
    END

END#layer 2

LAYER#3
    NAME "3"
    STATUS default
    TYPE polygon
    DATA "rooms/3"
    FILTERITEM "id"

    TOLERANCE 1
    TOLERANCEUNITS meters

    CLASS
        TEMPLATE "rooms_query.html"
        STYLE
            COLOR 178 34 34
            OUTLINECOLOR 0 100 0
        END
    END

    METADATA
        qstring_validation_pattern '.'
    END

END#layer 3


LAYER#4
    NAME "4"
    STATUS default
    TYPE polygon
    DATA "rooms/4"
    FILTERITEM "id"

    TOLERANCE 1
    TOLERANCEUNITS meters

    CLASS
        TEMPLATE "rooms_query.html"
        STYLE
            COLOR 178 34 34
            OUTLINECOLOR 0 100 0
        END
    END

    METADATA
        qstring_validation_pattern '.'
    END

END#layer 4



##########################################################################################
####################################DIRECTION LAYERS####################################
#For each layer there is a direction line layer which is used to show directions for a specific level only.
LAYER
     NAME "1D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END

LAYER
     NAME "2D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END


LAYER
     NAME "3D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END


LAYER
     NAME "4D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END

LAYER
     NAME "5D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END

LAYER
     NAME "6D" #the letter following number 1 is D from data
     STATUS on
     TYPE line
    CLASS
        STYLE
                COLOR 255 0 0
        END
    END
END

END#map file end
```

Listing C.1: The mapfile used by Mapserver for the shapefiles defined in OpenEV

# D. Class Diagram

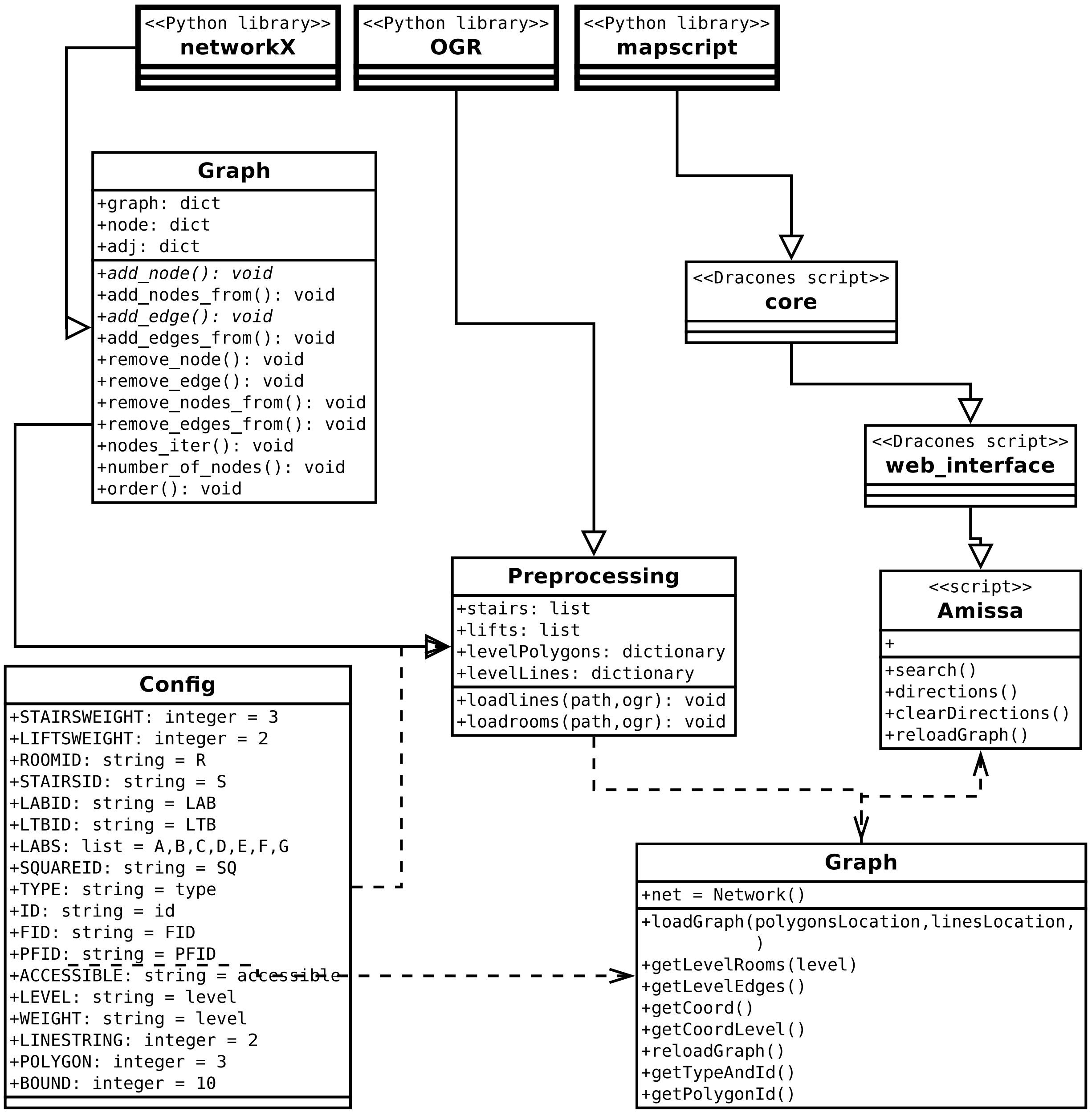


Figure D.1: Class Diagram of the all the classes and dependecies of the project

# E. Regular Expression

```
    def getTypeAndId(self ,query):
        """
        String parser for a query in order to identify its id and type.
        Particularly useful for queries like square 1, sq2, sq3, ltb 2, ltb3
        """
        string = str(query)
        if string[:3].upper() == LABID:#convert lowercase to upper case. First 3 letters
            try:
                if int(filter(lambda x: x.isdigit(),string)):
                    return LABID, (filter(lambda x: x.isdigit(),string)) # lambda function to parse and return the digits of the string
                elif string[3:] in LABS:
                    return LABID, string[3:]
            except ValueError: exit(255)
        if string[:3].upper() == LTBID: #check for the first 3 letters if they match the LTB acronym
            try:
                if int(filter(lambda x: x.isdigit(),string)):
                    return LTBID,(filter(lambda x: x.isdigit(),string))
            except ValueError: exit(255)
        if string[:2].upper() == SQUAREID:
            try:
                if int(filter(lambda x: x.isdigit(), string)):
                    return SQUAREID,(filter(lambda x: x.isdigit(),string))
            except ValueError: exit(255)
        else:
            return ROOMID, string.upper()
```

Listing E.1: Regular Expression function for parsing a string

# F. Graph class

```

from preprocess import *
from Config import *
import osgeo.ogr as ogr
import os
import networkx as nx
from bidijkstra import bidirectionalDijkstra

class Graph:
    """
    Graph object that stores all the nodes and edges into memory
    """
    net = Preprocess()

    def loadGraph(self,polygonsLocation, linesLocation):
        """
        The function loads only the shps from the path of two directories:
        one containing the shapefiles with polygon geometry features and the other
            line geometry features.
        parameters: polygonsLocation - location of shapefiles that contain polygon
            geometry features
                    linesLocation - location of the shapefiles containing line
                        geometry features

        """

        attributes = {}

        for room in os.listdir(polygonsLocation):
            if room.split(".")[1] == "shp":
                self.net.loadrooms(polygonsLocation+room, ogr)

        for line in os.listdir(linesLocation):
            if line.split(".")[1] == "shp":
                self.net.loadlines(linesLocation+line,ogr)
                del self.net.lines[:]#clear the lines features for the new level

        for lift in self.net.lifts: #add LIFT edges between floors
            for other_lift in self.net.lifts:
                if (lift[1][ID] == other_lift[1][ID]) and abs(int(lift[1][LEVEL]) -
                    int(other_lift[1][LEVEL])) == 1:#this only supports integers for
                    the moment.
                    attributes[WEIGHT] = LIFTSWEIGHT
                    attributes[LEVEL] = LIFTSID
                    self.net.add_edge(lift[0],other_lift[0],attributes)

        for stair in self.net.stairs: #adds STAIRS edge between floors
            for other_stair in self.net.stairs:
                if(stair[1][ID]==other_stair[1][ID]) and abs(int(stair[1][LEVEL]) -
                    int(other_stair[1][LEVEL])) == 1:
                    attributes[WEIGHT] = STAIRSWEIGHT
                    attributes[LEVEL] = STAIRSID
                    self.net.add_edge(stair[0], other_stair[0], attributes)

    def getLevelRooms(self, level):
        """
        Returns the  id's of the rooms on a certain level.
        parameter: level : string
        return: id : string - from the attributes dictionary of each node

        """
        for room in self.net.nodes():
            if (LEVEL and TYPE in self.net.node[room]) and self.net.node[room][LEVEL]
                == str(level) and self.net.node[room][TYPE] == ROOMID:
                return self.net.node[room][ID]

    def levelpaths(self, level):
        """
        Returns the edges for
        """
        for edge in self.net.edges():
            if LEVEL in self.net.edge[edge[0]][edge[1]] and self.net.edge[edge[0]][
                edge[1]][LEVEL] == str(level):
                return edge[0],edge[1]

    def getLevelEdges(self, lvl):
        """
        Returns all the Feature ID for the edges on a given level - useful for
            debugging
        parameters: level : string
        returns : edges : list
        """
        list = []
        for line in nx.generate_edgelist(self.net,data=[LEVEL, 'FID']):
            if line[len(line)-4] == str(lvl):
                list.append(line)
        return list

    def getCoord(self,type,id):
        """
        Returns the coordinates for a given type and id that can be found in the nodes
             attributes dictionary
        parameters: type : string
                    id : string
        returns: x,y,0.0
        """
        for node in self.net.nodes():
            if (ID in self.net.node[node] and TYPE in self.net.node[node]) and (self.
                net.node[node][ID] is not None and self.net.node[node][TYPE] is not
                None) and (self.net.node[node][ID] == str(id) and self.net.node[node][
                TYPE] == str(type)):
                return node[0],node[1],node[2]

    def bdijkstra(self,srcQuery, dstQuery):
        """
        Returns a list whose elements are tuples containing the coordinates of the
            shortest path between two coordinates
        parameters; source : string
                    destination : string
        return: path of shortest path : list of coordinates tuples
        """
        stype,sid = self.getTypeAndId(srcQuery)
        dtype,did = self.getTypeAndId(dstQuery)
        src = self.getCoord(stype,sid)
        dst = self.getCoord(dtype,did)
        if src is not None and dst is not None:
            li = []
            length,path = bidirectionalDijkstra(self.net,src,dst)
            for i in range(len(path)):
                li.append(path[i])
            return li

    def getLevel(self,query):
        """
        Returns the level for a string i.e.  1n1.4.1 or sq2
        parameter: query : string
        returns: level : integer
        """
        type, id = self.getTypeAndId(query)
        if type is not None and id is not None:
            return self.getCoordLevel(self.getCoord(type,id))

    def getCoordLevel(self,coordinates):
        """
        Returns the level of coordinates.
        parameter: coordinates tuple  i.e. (x,y,z)
        returns: level : integer
        """
        if coordinates is not None and LEVEL in self.net.node[coordinates]:
            return self.net.node[coordinates][LEVEL]

    def getCoordType(self,coordinates):
        """
        Returns the type of given coordinates that have the type stored in the
            attributes dictionary
        parameter: coordiantes : tuple
        return: type : string i.e. R, L, FM
        """
        if TYPE in self.net.node[coordinates]:
            return self.net.node[coordinates][TYPE]

    def reloadGraph(self):
        """
        Clears the current Preprocess object and calls the loadGraph
        """
        self.net.clear()
        self.loadGraph(POLYGONLOCATION,PATHSLOCATION)

    def getTypeAndId(self,query):
        """
        String parser for a query in order to identify its id and type.
        Particularly useful for queries like square 1, sq2, sq3, ltb 2, ltb3
        """
        string = str(query)
        if string[:3].upper() == LABID:#convert lowecase to upper case. First 3
            letters
            try:
                if int(filter(lambda x: x.isdigit(),string)):
                    return LABID, (filter(lambda x: x.isdigit(),string))
                elif string[3:] in LABS:
                    return LABID, string[3:]
            except ValueError: exit(255)
        if string[:3].upper() == LTBID:
            try:
                if int(filter(lambda x: x.isdigit(),string)):
                    return LTBID,(filter(lambda x: x.isdigit(),string))
            except ValueError: exit(255)
        if string[:2].upper() == SQUAREID:
            try:
                if int(filter(lambda x: x.isdigit(), string)):
                    return SQUAREID,(filter(lambda x: x.isdigit(),string))
            except ValueError: exit(255)
        else:
            return ROOMID, string.upper()

    def getPolygonId(self,coord):
        """
        Returns the Feature ID of the polygon given the coordinates of the point that
            is within its boundaries
        parameter: coordinates : tuple of floating points for x,y,z
        returns: integer
        """
        if coord is not None and PFID in self.net.node[coord]:
            return self.net.node[coord][PFID]

    def getPID(self, query):
        type, id = self.getTypeAndId(query)
        pid = self.getPolygonId(self.getCoord(type,id))
        if pid is not None:
            return pid


g = Graph()
Graph.loadGraph(g,POLYGONLOCATION,PATHSLOCATION)
```
Listing F.1: Graph class

# Bibliography

[1] A. Bouch, A. Kuchinsky, and N. Bhatti, “Quality is in the eye of the beholder: meeting users’ requirements for Internet quality of service,” in *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 2000, pp. 297–304.

[2] D. Farber, “Google’s Marissa Mayer: Speed Wins,” *CNET Between the lines*, 2006.

[3] Y. Xia Skadberg and J. Kimmel, “Visitors’ Flow experience while browsing a web site: its measurement, contributing factors and consequences,” *Computers in human behavior*, vol. 20, no. 3, pp. 403–422, 2004.

[4] E. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[5] P. Sanders and D. Schultes, “Engineering fast route planning algorithms,” in *WEA’07: Proceedings of the 6th international conference on Experimental algorithms.* Berlin, Heidelberg: Springer-Verlag, 2007, pp. 23–36.

[6] T. Ikeda, M. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh, “A fast algorithm for finding better routes by AI search techniques,” in *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994.* IEEE, 2002, pp. 291–296.

[7] I. Pohl, *Bi-directional search.* IBM TJ Watson Research Center, 1970.

[8] “Stack Exchange:Geographic Information Systems,” http://gis.stackexchange.com/.

[9] “OSGeo:Open Source Geospatial Foundation,” http://osgeo-org.1803224.n2.nabble.com/.

[10] “OpenEV software homepage,” http://openev.sourceforge.net/.

[11] T. Mitchell, *Web mapping illustrated.* O’Reilly Media, Inc., 2005.

[12] “Python Patterns: Implementing Graphs,” http://www.python.org/doc/essays/graphs.html.

[13] S. Roth, “Ray casting for modeling solids* 1,” *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109–144, 1982.

[14] P. Bourke, “Determining if a point lies on the interior of a polygon,” *Crawley, AU., November*, 1989.

[15] “OGR Geometry Class Reference,” http://geoinformatics.tkk.fi/doc/.

[16] M. Lutz, *Learning Python*, ser. Learning Python. O’Reilly, 2009. [Online]. Available: http://books.google.co.uk/books?id=1HxWGezDZcgC

[17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms.* MIT Press, 2009. [Online]. Available: http://books.google.com/books?id=VK9hPgAACAAJ

[18] M. Hetland, *Python Algorithms: Mastering Basic Algorithms in the Python Language*, ser. Apress Series. Apress, 2010. [Online]. Available: http://books.google.com/books?id=2l0AJv9J8Z0C

[19] "ActiveState Code»Recipes: Dijkstra's algorithm for shortest paths," http://code.activestate.com/recipes/119466/.

[20] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[21] D. Benyon, P. Turner, and S. Turner, *Designing interactive systems: people, activities, contexts, technologies.* Addison-Wesley, 2005. [Online]. Available: http://books.google.com/books?id=jt9QAAAAMAAJ

[22] C. Lewis, *Using the" thinking-aloud" method in cognitive interface design.* IBM TJ Watson Research Center, 1982.

[23] R. González and R. Woods, *Digital image processing.* Pearson/Prentice Hall, 2008. [Online]. Available: http://books.google.com/books?id=8uGOnjRGEzoC