Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Meeting Scheduler

Bc. Lukáš Koc

Supervisor: Ing. Jan Baier

January 24, 2017

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

In Prague on January 24, 2017 . . . .

Citation of this thesis

Koc, Lukáš. *Meeting Scheduler.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Replace with comma-separated list of keywords in English.

Contents

List of Figures

Introduction

Motivation

The 21st century is often labelled as the century of data. Usage and gathering the data influence our lives on daily basis. Business, industry, fashion, society or transportation is transforming under the amount of data gathered and evaluated.

Alongside the growing of the amount of data collected, borders with European countries are loosening and it is possible for people to travel within Europe in order to discover new cultures or to find a job in the country of their interest. Most of the time it is an unknown location. Finding meeting point with the colleagues or friends, which might not know the location as well, might turn into difficult task.

Applying the knowledge of graph theory, geography and computer science, the goal of the thesis is to create application finding ideal meeting place for a group of people, saving time for everyone involved. Nowadays computers with multiprocessors or graphic cards are affordable for everyone, meaning paralelization of the computation is significant for every sophisticated application. Knowing that, the application should be designed to utilize possibilities of paralellization.

Geographic information system

Geographic information systems (GIS) are solving problems which are based on geospatial information. To achieve the goal special tools are being used such as visualization software, remote sensing and geography tools. Remote sensing tools gain information on specific objects or areas from a distance. Geography tools help to observe and research the environmental changes of the earth and its resources, its evolution of society and species. Visualization software, then, displays gathered data as 2D or 3D images [1].

With a drastic change of modern technologies and enormous amount of data a new science was born—geographic information science—which is focusing on geographic concepts, applications and systems. The new science opens doors to new problems and issues at a global scale, not easily imaginable a few years ago.

This thesis uses knowledge gathered through the course of time in geographic information science to map graph theory problems on real life data. The developed application belongs to the software category GIS software.

Problem description

Directed graph

A directed simple *graph* G is a pair (V, E), where V is a finite set of *vertices* and $E \subseteq V \times V$ are the *edges* of a graph G. The number of vertices $|V|$ is denoted by N and the number of edges $|E|$ is denoted by m throughout this thesis, . A *path* in G is a sequence of vertices $v_1, v_2 \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. A path with $v_1 = v_k$ is called a *cycle*. A graph (without multiple edges) can have up to n^2 edges.

Shortest path problem

Let $G = (V, E)$ be a directed graph whose edges are weighted by the function $f : E \to \mathbb{R}$. The length of a path is the sum of the weights of its edges. In this sense the weights can be reinterpreted as the edge lengths. A cycle whose edges sum to a negative value is *negative cycle.*

The shortest-path problem consists of finding a path of minimum length from a given source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if G does not contain negative cycles. Since our problem is based on real values (distance between two points), negative weights and cycles do not occur in the rest of the thesis. If there were negative weights, it is possible, using Johnson's algorithm [2], to convert the original weighting function $f : A \to \mathbb{R}$ to non-negative function $f' : A \to \mathbb{R}_0^+$ in time $O(n_m + n^2 \log n)$, which results in the same shortest paths.

Organisation of the Thesis

The thesis is structured as follows: First, the Chapter 1 describes the desired attributes of the data and finding possible data sources. The chapter then further explains how the data can be stored and processed in order to reduce the number of unnecessary nodes.

The next Chapter 2 focuses on the overall design of the application, beginning with the algorithms solving the SSSP problem. The main algorithm used for our application is Dijkstra algorithm. In order to increase the computation time, the possible paralellization of Dijkstra algorithm is further explained. The application also requires basic user interface.

The Chapter 3 describes the specific implementation of the application. More specifically how the application is implemented within server and client part, which frameworks were used and additional commentary of the code.

The results of the application are concluded in Chapter 4, where we compare the results based on the accuracy and time comparison between parallel and single threaded implementation.

Chapter 1

Data Analysis

1.1 Data fitness

As stated in Koc 2014 [3], proficient functioning of the application requires a fitting data source. The application needs to work with reliable and (preferably) daily updated data. The area of coverage should not be limiting the application, so that a high number of users would find it convenient. The data format should be unified in order to make manipulation and management effective. Choosing the correct data format also enables the application to combine different data sources.

To sum up the data are required to follow certain criteria:

- up-to-date
- verified
- human and computer readable
- easy-to-use and unified format
- freely available
- maintained, reliable

1.2 Possible sources

While searching for data, it was focused on sources providing free data of Europe, which is also available to the public. Further subsections describe sources which matched the criteria mentioned in section 1.1.

1.2.1 EuroGeographics

EuroGeographics is the membership association consisted of 60 organizations and 46 countries. It was created in year 2002, when the Comitée Européen des Responsables de la Cartographie Officielle (CERCO) and the Multi-purpose European Ground Related Information Network (MEGRIN) merged together. Its goal is to gather and collect spatial and infrastructural data of Europe [4].

EuroGeographics association provides the following products: EuroBoundaryMap, EuroGlobalMap, EuroRegionalMap and EuroDEM. EuroBoundary map mostly covers borders and administrative informations, EuroDEM map is commonly used for environmental change research or hydrologic modelling. EuroGlobalMap and EuroRegionalMap consists of many datasets: the administrative boundaries, the water network, the transport network etc. In order to download the data it is required to fill up the registration form.

EuroGeographics provides data in following formats

- Geodatabase
- Shapefile

EuroGeographics yield reliable source of data required by our application. Shapefile format can contain geographical data providing an option of additional attributes for the data. However, the registration form which needs to be filled almost every time in order to download the dataset makes the data processing cumbersome.

1.2.2 OpenStreetMaps

OpenStreetMaps (OSM) is a project officially supported by the OSM Foundation. OSM was created to build and provide open[1] geographical data available to everyone.

The OMS project was inspired by Wikipedia and is working exactly the same: Users are the ones contributing with their maps, gps measurements, aerial photographs etc. Since OSM creation in 2004, its community has significantly increased and the data are being updated daily. OSM provides data in their .osm format, which follows XML rules.

In course of time a lot of projects were created which work with OSM maps. Thanks to the team Mapzen and their Metro Extract project it is possible to download any major city data in additional two GIS data formats:

- Geojson
- Shapefile

[1]Open data means for any purpose as long as the OSM and it's contributors are credited.

OSM data are regularly maintained by the community. Documentation to the data is well written, so that any discrepancy within the dataset can be easily resolved. The OSM format requires additional parsing in order operate with the data since it is not official GIS format, but projects as Mapzen provides additional format variety to choose from.

1.2.3 EEA

The European environmental agency (EEA) is an agency of European Union providing information about the environment for the public. According to their official site [5] it currently consists of 33 member countries.

EEA offers various different datasets, maps and graphs about national designated areas, ecosystem types of Europe, water state and quality, national communications etc.

Depending on the type, these data are provided in the following formats:

- Excell table
- CSV
- Shapefile

Most of the datasets are displayed in interactive maps available on the EEA website.

After further investigation of the datasets provided by EEA, it seems they are not being updated regularly. In addition, most of the data sets are results of ecological studies or environmental researches (e.g. monitoring CO_2 or energy consumption).

1.2.4 European Observation Network for Territorial Development and Cohesion

The European Observation Network for Territorial Development and Cohesion (ESPON) 2013 Programme is mainly financed from European Regional Development Fund (ERDF) and its main goal is:

"Support policy development in relation to the aim of territorial cohesion and a harmonious development of the European territory . . . " [6]

Data are available as soon as users register and accept the Terms & Conditions. EPSON 2013 data are handled according to ISO 19115 scheme in two formats:

- XML
- Excel file

Unfortunately, the data provided by ESPON are having a same disadvantage as data provided by EEA in previous section: maintainability. Our application require updated data sets in order to accurately compute the optimal meeting place. Investigating the ESPON page further, the ESPON project was separated into two phases, first phase starting in July 2008 – February 2011 followed by second phase having a lifetime in span of February 2011 – December 2014.

As a data source the OpenStreetMaps were chosen since the data provided are regularly maintained by the community. In addition, the whole documentation in for of the wikipage contains all necessary information to learn the user how to fast start working with the data. As a format was chosen a Shapefile format, allowing the application to work with additional attributes if necessary (e.g. street names, type of road etc.).

1.3 Format

Geographical data exist in various formats depending on type and usage of the data. Data representing the elevation of mountains are better stored in different format whereas data representing the location of points of interest. Most of the existing formats typically fall into two main categories: vector format or raster format.

Both offer two different ways how to represent spatial data. However, the differences between vector and raster data types are equivalent to those in the graphic design world. The picture 1.1, which graphically explains how these two types process given data, serves for a better understanding.

Both representations carry various set of advantages and disadvantages. These will be described in the following subsections.

1.3.1 Raster representation

Raster type formats consist of equally sized cells arranged in rows and columns to construct the representation of space. Individual cells contain an attribute value and location coordinates. Together they create images of points, lines, areas, networks or surfaces.

Advantages

- Easy and ”cheap” to render
- Represent well both, discrete (urban areas, soil types) and continuous data (elevation)
- Grid natures provide suitability for mathematical modeling or quantitative analysis

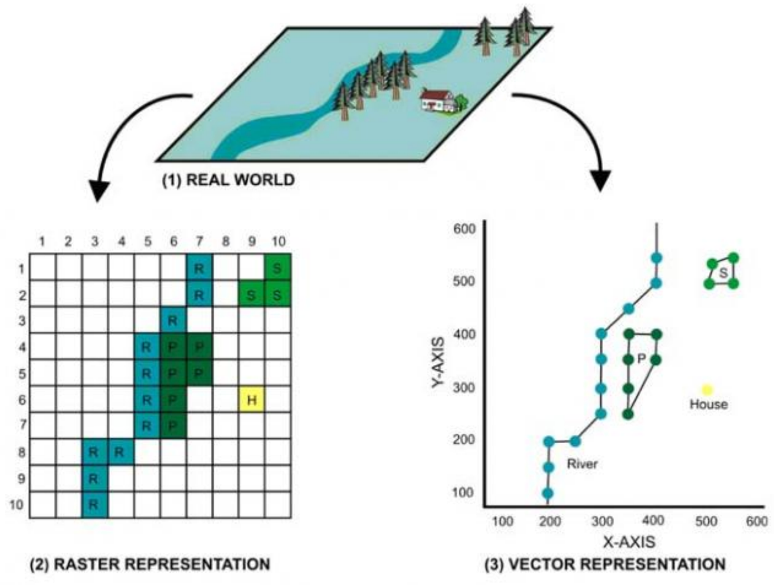

Figure 1.1: Realization of raster and vector data representation

Disadvantages

- Large amount of data
- Scaling required between layers
- Possible information loss due to generalization (static cell size)
- Difficult to establish network linkage

1.3.2 Vector representation

Vector type formats uses vertices as a basic unit. A vertex consists of x and y coordinates to determine its position. Using vertices, it is possible to create any shape to describe any object. One vertex creates a point, two can create a line etc. Objects created by vertices may contain additional attributes about the feature they represent.

Advantages

- Topology nature
- Compact data structure
- Easy to maintain
- Bigger analysis capability

Disadvantages

- For effective analysis, static topology needs to be created

- Every update requires rebuilding of topology
- Continuous data is not effectively represented

1.4 Storage

Since the data source and format question is resolved, the next step is to decide the representation of the graph in the memory. During the history of graph theory, three main representations are to choose from. In the following subsections a closer look on all three options will be taken.

1.4.1 Adjacency list

An adjacency list stores a graph as a list of vertices. Each vertex, then, contains an information about its adjacent vertices in form of a linked list. Adjacency list is easy to implement and use. All vertices in a graph are mapped onto the array of pointers referencing to a first node of a linked list. In case, a vertex does not have the adjacent vertices, its pointer is set to null. The example of an adjacency list for a simple graph can be found in Figure 1.2 and will be further used as example to also explain further representations.

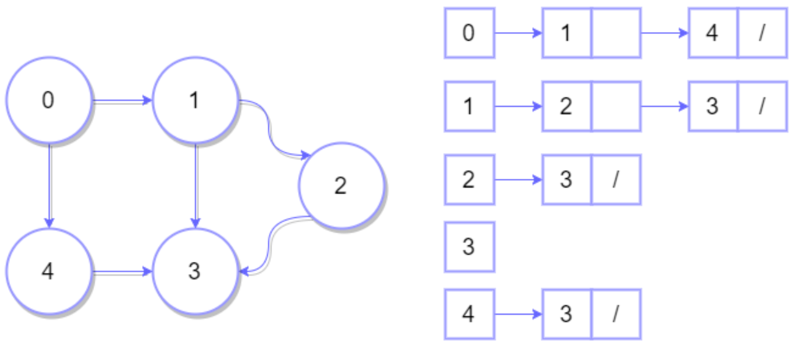

Figure 1.2: Representation of a graph using the method of an adjacency list

1.4.2 Adjacency matrix

An adjacency matrix is defined as matrix of a size $|V(G)| \times |V(G)|$, where $V(G)$ is a set of all vertices in graph G. Values within the matrix depend on the type of graph. Generally, adjacency matrix for unweighted graph is defined as a $\mathbf{A}(G) = [a_{ij}]$, where a_{ij} is the number of edges joining v_i and v_j. If the graph is weighted, the values are from the interval $\langle 0, \infty)$, where 0 means two vertices are not adjacent and any non-zero value means they are adjacent with an edge cost of that value[7]. Although, 1 edge at most must exist between every two vertices. For the graph G in the example used in

Figure 1.2, the adjacency matrix **A** looks like the following:

$$\mathbf{A}(G) = \begin{array}{c|ccccc} & v_0 & v_1 & v_2 & v_3 & v_4 \\ \hline v_0 & 0 & 1 & 0 & 0 & 1 \\ v_1 & 0 & 0 & 1 & 1 & 0 \\ v_2 & 0 & 0 & 0 & 1 & 0 \\ v_3 & 0 & 0 & 0 & 0 & 0 \\ v_4 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Rows and columns represent vertices of a graph. In the case of matrix **A**, first row and first column represent vertex 0, second row and column represent vertex 1 etc. The value in the third row and fourth column means that vertex 2 is adjacent with vertex 3. It is noticeable that the graph in example **A** is directed, therefore adjacency matrix is not symmetric.

1.4.3 Incidence matrix

Incidence matrix is very similar to adjacency matrix, but instead of showing relations between vertices themselves it represents relation between vertices and edges. Which means, the size of an incidence matrix is $|V(G)| \times |E(G)|$, whereas $V(G)$ is a set of all vertices and $E(G)$ is a set of all edges in graph G. The incidence matrix of graph G is then $\mathbf{M}(G) = [m_{ij}]$, where m_{ij} is the number of times (0, 1 or 2 in case of loop) that the vertex v_i and edge e_j are incident[7].

An interesting case is the incidence matrix for a directed graph. In that case the sign of the value within matrix **M** describes the orientation of the edge. Given the edge $e = (x, y)$, then, in the row of vertex x and the corresponding column for edge e, the value is positive. In the row of vertex y and the corresponding column for edge e, the value is negative. For the graph G in the example used in Figure 1.2, the incidence matrix **M** looks like the following:

$$\mathbf{M}(G) = \begin{array}{c|cccccc} & e_0 & e_1 & e_2 & e_3 & e_4 & e_5 \\ \hline v_0 & 1 & 0 & 1 & 0 & 0 & 0 \\ v_1 & -1 & 1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & -1 & 0 & 0 & 1 & 0 \\ v_3 & 0 & 0 & 0 & -1 & -1 & -1 \\ v_4 & 0 & 0 & -1 & 0 & 0 & 1 \end{array}$$

1.4.4 Sparse matrix

In mathematics matrices can be divided into two groups: sparse matrices and dense matrices. The definition might sound somehow vague, but sparse matrices are matrices containing huge amount of zero elements. A dense matrix is the exact opposite: containing very few zero elements. In previous subsections it is noticeable that each of the matrices (adjacency and incidence) consist of a lot of zero elements and only a few values are actually useful.

The amount of non-zero elements in the adjacency or incidence matrix depends purely on the degree of vertices in the graph. In both types of matrices, each row serves as a vertex and within non-zero values represent edges incident to the vertex. Depending on the graph is directed or undirected, the amount of non-zero elements in adjacency matrices will differ. Incidence matrices do not change their numbers, because they differ only in sign of the value.

For undirected graphs, the number of non-zero elements equals to

$$\sum_{v \in V} \mathrm{deg(v)} = 2|E|$$

where E is the set of all edges and V the set of all vertices in graph. The same principle applies to the directed graph of incidence matrices. For directed graphs of adjacency matrices, we can observe that the number of non-zero elements depend on amount of outgoing edges $\Rightarrow$ out-degree, which is

$$\sum_{v \in V} \mathrm{deg}^{-}\mathrm{(v)} = |E|$$

where E is the set of all edges, V is the set of all vertices in a graph and $deg^{-}(v)$ function returns number of outgoing edges from the vertex v.

The reason for mentioning sparse matrices in the first place is that there are functions and operations which could be done only with the sparse matrices, providing better memory usage. The main motivation for this section are the storage schemes in which sparse matrix could be stored. The usage of storage schemes enable all the advantages of the regular matrix representation with significantly less memory usage since only the non-zero elements are being stored. According to Yousef Saad [8], main 3 storage schemes will be discussed.

The coordinate format belongs to the simplest storage schemes of sparse matrices. The data structure consists of three arrays:

- an array containing all the (real or complex) values of the non-zero elements of the original matrix in any order
- an integer array containing their row indices
- an integer array containing their column indices

All three arrays are of length N, which is the number of non-zero elements.

Taking a closer look at adjacency **A** matrix from section 1.4.2. Clearly this matrix contains less non-zero elements, therefore it is an example of sparse matrix. Using coordinate format, matrix **A** looks the following:

$$AA: \begin{array}{|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 1 & 1 & 1 \\ \hline\end{array}$$

$$IR: \begin{array}{|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 0 & 1 & 4 \\ \hline\end{array}$$

$$IC: \begin{array}{|c|c|c|c|c|c|}\hline 4 & 2 & 3 & 1 & 3 & 3 \\ \hline\end{array}$$

Array AA stores values of non-zero elements, array IR stores the row index of the corresponding element and array IC stores the column index of the corresponding element. The memory needed for storing the matrix is now only $3N$ instead of the original N^2. The coordinate format excels with it's simplicity and flexibility.

If the elements inside array AA are listed by row, the array IR could be transformed to store instead only indices of the beginning of each row. The size of newly defined array IR is then $n+1$, where n is the number of rows in the original matrix. On the last position (+1), the number of non-zero elements within the original matrix is being written. It also may be represented as an address, where fictional row begins on $n+1$ position.

Array **A**, then, would be by this scheme described as the following:

$$AA: \begin{array}{|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 1 & 1 & 1 \\ \hline\end{array}$$

$$IR: \begin{array}{|c|c|c|c|c|c|}\hline 0 & 2 & 4 & 5 & 5 & 6 \\ \hline\end{array}$$

$$IC: \begin{array}{|c|c|c|c|c|c|}\hline 1 & 4 & 2 & 3 & 3 & 3 \\ \hline\end{array}$$

The transformation of the IR array and listing elements inside AA by row is called Compressed Sparse Row (CSR) format. In scientific computing CSR format is most commonly used for vector-matrix multiplication while having low memory usage. Through the years, Compressed Sparse Row format developed to a number of variations. For example storing columns instead of rows, a new scheme known as Compressed Sparse Column (CSC) format was created.

The last scheme, I would like to point out, is called the Ellpack-Itpack format, which is very popular on vector machines. The Ellpack-Itpack format stores matrices in two 2-dimensional arrays of the same size $n \times N_{mpr}$, where n is the number of rows of the original matrix and N_{mpr} represents the maximum of non-zero elements per row. The first array contains non-zero elements of the original matrix. If the number of non-zero elements is less then the N_{mpr}, the rest of the row is filled with zeros. The second array stores the information about the column in which the specific non-zero element is located. For each zero in the first array, any number can be added.

For the given matrix **EIF**:

$$\mathbf{EIF} = \begin{pmatrix} 4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 7 & 0 & 9 \\ 0 & 2 & 0 & 0 & 0 \\ 6 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 4 & 3 \end{pmatrix}$$

the Ellpack-Itpack format looks as followed:

$$\mathbf{AA} = \begin{pmatrix} 4 & 1 \\ 7 & 9 \\ 2 & 0 \\ 6 & 1 \\ 4 & 3 \end{pmatrix} \qquad \mathbf{IC} = \begin{pmatrix} 0 & 3 \\ 2 & 4 \\ 1 & 0 \\ 0 & 3 \\ 3 & 4 \end{pmatrix}$$

1.4.5 List and matrix comparison

Adjacency lists in their essence compactly represent existing edges. However, this comes at the cost of possible slow lookups of specific edges. In case of unordered list, the worst case of a lookup time for a specific edge can become O(n), since each list has length equal to the degree of a vertex. On the other hand, looking up the neighbours of a vertex becomes trivial, and for a sparse or small graph the cost of iterating through the adjacency lists might be negligible.

Adjacency matrices can use more space in order to provide constant lookup times. Since every possible entry exists, it is possible to check for the existence of an edge in constant time using indexes. However, the lookup time for a neighbour becomes O(n) since it is needed to check all possible neighbours.

Data used in the application produce sparse and/or large graphs, for which adjacency list representations suit better.

1.5 Preprocessing

Data are being read from Shapefile source obtained via OpenStreetMaps. After further analysis of the Shapefile provided by Mapzen (Section 1.2.2), each data entry is being represented as a line $\Rightarrow$ two nodes sharing an edge. At this point, the edge is missing its weight. Therefore, first step of preprocessing is to weight all the edges within the graph.

1.5.1 Great Circle Distance and Harvesine formulae

The weight of the edge should represent the price of getting from one node into the other. Our application is trying to find shortest path from multiple sources into one source, therefore the price should be based on the distance between source points and the final node, the shorter the better.

Each node has its specific coordinates $\Rightarrow$ latitude and longitude. This pair represents unique identifier for every node in the application. Let us have two nodes: node n having coordinates $lat1$ and $long1$, and node m having coordinates $lat2$ and $long2$. If $lat1 = lat2 \wedge long1 = long2$, then node $n = m$. If $n \neq m$, then at least one of the coordinates differs between nodes n and m meaning distance between these two nodes is greater than zero.

The simplest solution to compute distance between two points is using the Pythagorean theorem.

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where d is a distance between the nodes (x_1, y_1) and (x_2, y_2). If we map latitude on x-coordinate and longitude on y-coordinate, we would get distance between two real points, but in a two dimensional space. For computing the distance on the Earth it is needed to use great circle distance.

The great circle distance takes into account the curvature of the sphere, to provide more precise distance.

To compute the great circle distance on a sphere, harvesine formula is used:

$$\begin{aligned} Harvesineformula: \quad & a = \sin^2(\Delta\psi/2) + \cos\psi_1 \cdot \cos\psi_2 \cdot \sin^2(\Delta\lambda/2) \\ & c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{1-a}) \\ & d = R \cdot c \end{aligned}$$

Chapter 2

Design of the application

2.1 SSSP algorithms

For solving shortest path problems exist nowadays many algorithms. Most of them evolved from their predecessors. Each of them solves the problem with different parameters. The following list contains the essential algorithms solving the shortest path problem, which lay the foundation in graph theory science:

- Dijkstra's algorithm
- Bellman-Ford algorithm
- Floyd-Warshall algorithm
- Johnson's algorithm

The Dijkstra's algorithm [9] and the Bellman-Ford algorithm [10, 11] solve the single-source shortest path (SSSP) problem. SSSP problem can be defined as: Find the cost of the least cost path from v to $\forall w \in V$, given a directed graph $G = (V, E)$ with non-negative costs on each edge and a selected source node $v \in V$. The cost of a path is simply the sum of the costs on the edges traversed by the path. Dijkstra's algorithm is a greedy algorithm working with the graph were negative edges are not allowed. The Bellman-Ford algorithm is a non-greedy version of the Dijkstra's algorithm which allows to work with graphs having negative edges.

The Floyd-Warshall algorithm [12, 13] and Johnson's algorithm [2] solve the all-pair shortest path (APSP) problem. The Floyd-Warshall algorithm iterates all vertices v, in order to find a better path for every pair going through v in time $O(N^3)$. Johnson's algorithm, first, converts all the negative edges into positive ones and then, applies Dijsktra's algorithm on every node within the graph. For sparse graphs, Johnson's algorithm provides faster computation time than Floyd-Warshall algorithm [14].

2.2 Dijkstra algorithm

The algorithm was conceived by Edsger Wybe Dijkstra in 1956 and was officially published in 1959. Dijkstra's original idea was to find the shortest path between two nodes, however the course of time, among computer scientists, Dijkstra algorithm was accepted as an algorithm finding a path from one single node to all other nodes in the graph.

The Dijkstra algorithm was constructed to solve the applied problem: How to get from one point to another using the shortest path possible. The main criteria used to define the shortest path are either time or distance. Both quantities only have positive values. If these criteria are converted into graph theory, all the edges of the graph need to be also positive. The data available will only include possitive values, which are the nodes in the graph. Furthermore the graph will represent the geographical map. Therefore Dijkstra algorithm will be the choice of the problem solving algorithm.

2.2.1 Definition

There are many variations of Dijkstra algorithm. Here is a presented variation, in which nodes can be in three states: $fresh$, *open* and *closed* (Algorithm 1). For each node **n** the function `dist`(**n**) represents the distance from the starting node `s`. For unreachable nodes the value returned by this function will be undefined. Next to the function `dist`, there is also the function `prev`(**n**), which returns the node for the shortest path back to node `s`.

```
1  set the starting node s as open and dist(s) $\leftarrow$ 0;
2  for for each node n different from s do
3      set node n as fresh
4  end
5  while $\exists u$ with state open do
6      u $\leftarrow$ open node with minimal dist(u) ;
7      set u state as closed ;
8      foreach neighbour w of u do
9          if w is fresh or dist(w) $>$ dist(u) $+$ length(w, u) then
10             if w is fresh then  set w as open;
11             dist(w) $\leftarrow$ dist(u) $+$ length(w, u);
12             prev(w) $\leftarrow$ u
13         end
14     end
15 end
```

Algorithm 1: Dijkstra algorithm

On the line number **6** the choice of the node **u** means choosing the **u** with `dist`(**u**) $\leq$ `dist`(**w**), where **w** is every other node having the state *open*.

2.2.2 Proof of correctness

In order to prove the algorithm will stop after finite number of step and its correctness, it is needed to define lemma about states in which nodes can be:

Lemma 1. *Nodes can only change state either from "fresh" to "open" or from "open" to "closed".*

Proof. The only time nodes can change state are on line **7** and **10** of the Algorithm 1. ♡

Theorem 1. *Dijkstra algorithm will stop computing after at the most N steps iterations of while cycle, where N is a number of nodes in graph.*

Proof. From the description of the algorithm and previous lemma it is clear the set of closed nodes of cycle will increase with each iteration by one and its size being between 0 and N. ♡

Theorem 2. *Let A be a set of closed nodes. The length already found path from v_0 to v is the length of the shortest path $v_0v_1 \dots v_kv$, where nodes $v_0, v_1, \dots, v_k$ are in set A.*

Proof. The proof is by induction on the number of step executed. The theorem clearly is correct before and after the first step.

Let w be a node with a state set to closed in last step. Let us consider a node v which is closed. If $v = w$, then the theorem is trivial. In opposite case case we will show, that there is a shortest path from v_0 to v through nodes in set A not containing the node w. Set L as a length of the path from v_0 to v through the nodes in A without w. Because in each step we choose node with the lowest `dist(u)` and `dist` of chosen nodes in each step represents nondecreasing sequence (weight of the edges are positive), then the length of the path from v_0 to w through nodes in A is at least D. Because we have chosen the D we know, there exists a path from v_0 to v through nodes in A which is not using node w.

Now let us consider a node v which is not closed. Let $v_0v_1 \dots v_kv$ be the shortest path from v_0 to v, where $\forall v_0, v_1, \dots v_kv \in A$. If $v_k = w$, then we changed the `dist` to the length of this path in current step. If $v_k \neq w$ then $v_0v_1 \dots v_k$ is the shortest path from v_0 to v_k through nodes in A and so we can assume that no nodes $v_0, v_1, \dots v_k$ is not w (according to last paragraph). Hence the length of the path was already set to the correct value before current step.

Due to the fact, that after last step the set A contains only the nodes, into which exists a path from node v_0, we have proven the correctness of Dijkstra algorithm. ♡

2.2.3 Time complexity

Now from the Algorithm 1 we can compute time general complexity of Dijkstra algorithm. Let us assume we use array in order to store the distances for all the N nodes. As proven in Theorem 1 the whole algorithm will execute at most N steps, in each we are choosing node from the set of *fresh* nodes having size $O(N)$. In each step we also need to check the number of nodes, which are being connected via edges outgoing from the currently checked node. Number of these checks in total is equal to at most $O(E)$, where E is a number of edges in the input graph. To sum up time complexity equals to $O(N^2 + M)$, i.e. $O(N^2)$ since E can be at the most N^2.

It is possible to improve this complexity by using heap instead of an array in order to store the distances. In the beginning the heap will contain N elements and in each step this number will be reduced by one: We find and delete smallest one in a time of $O(\log N)$ and adjusting the distances of the neighbours, which takes $O(E \log n)$ through all the edges. In total the time complexity of the algorithm is $O((N + M) \log N)$. As mentioned in Section 2.2 for real life problems we expect the graph having a form of a sparse graph, meaning for $M << N^2$ the heap version of the algorithm will provide much more better results.

Dijkstra algorithm is very similar to Bellman-Ford algorithm mentioned in subsection . The main difference between these algorithms is repetitive checking of the nodes. Once Dijkstra algorithm closes a node, it will never be checked again. Bellman-Fold algorithm goes through each of the nodes and recalculates the path in case negative edges exists in the graph. Because of this extra step, it is slower then Dijkstra, but can detect whether the graph is valid or not.

Dijkstra algorithm computes shortest paths from one single node to all of the others in the graph. However, the Meeting Scheduler application is searching optimal point given multiple input locations instead of single one. After applying Dijkstra algorithm on every input node, the graph instance now contains shortest paths for each input node. Therefore, every node in the graph holds an information about the distance to each of the input nodes. Within these nodes exists one node, having the optimal combination of shortest paths, which is the result the Meeting Scheduler application. This whole computation process leaves a lot of space for the performance optimization, especially parallelization.

2.3 Parallelization of Dijkstra algorithm

As mentioned in previous section, the Meeting Scheduler application applies Dijkstra algorithm on each of the input nodes. As a result every node stores the information about the shortest path to each of the input nodes.

2.4 User interface design

The application requires the spacial information as the input. Further, the user should be able to insert the positions of people in order to find a meeting, which is approximately of the same distance to all positions. For that, the user interface (UI) of the application should provide a way to register and store spacial information provided by the user.

Creating a fully satisfiable graphical user interface (GUI) is not main purpose of this thesis, so it was decided to create a simple GUI in which users can insert the geographical coordinates of participants. As a result, a single pair of coordinates of final destination will be received.

A fully optimal GUI would allow users to select a position through the displayed map of the dataset available. That way finding exact coordinates would be further transcribed into a marked point on the map and the general user-flow would be significantly improved, because the user would see a marked point on the map which would be the meeting point.

Chapter 3

Realisation

3.1 Frameworks used

The goal of the thesis is to create complex desktop application with basic front-end to provide user proper control over the input data and general overview over the application. In addition, the technology used should be cross-platform. In general, modern high-level programming languages prefer one platform over the other (C# Windows, ObjC iOS etc.).

In direction of keeping the application as simple as possible, the main computation part of the application is written in C++ language, which provides great computing performances on any platform while offering OOP principles in order to create more complex applications. In addition, C++ based back-end will make deployment on any server operating system effortless. Not to mention there are plenty open source libraries available making a realization of the whole project easily done.

One of the external tools used for the purpose of the application is Qt framework. Qt provides cross-platform tools to create basic GUI and is classified as FOSS computer software, therefore fitting the purpose of the application being open source. Usage of classes and functions of Qt framework is very straightforward while producing fully functional GUI as a front end for the application.

For reading the data files and following manipulation GDAL/ORG library was selected. GDAL is designed to read and write raster GIS formats. GDAL library is developed under Open Source Geospatial Foundation and released under the X/MIT license. As an addition to the GDAL is the ORG library which enables usage of simple features for vector formats. Together the whole GDAL/ORG library supports most of the GIS formats. Since the dataset of the application is in Shapefile format, GDAL/ORG library provides optimal tools for reading and collecting information from our dataset.

3.2 Server

3.2.1 Data processing part

3.2.1.1 Data

3.2.2 Computation part

3.2.2.1 Algorithm

3.2.2.2 Parallelization

3.3 Client

Chapter 4

Results

4.1 Speed

4.2 Preciseness

Conclusion

Bibliography

[1] Chen, J. X. Geographic Information Systems. *Computing in Science & Engineering*, Jan.–Feb. 2010: pp. 8–9, ISSN 1521-9615.

[2] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery*, volume 24, 1977: pp. 1–13.

[3] Koc, L. *Plánovač setkání*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[4] EuroGeographics. About Us. `http://www.eurogeographics.org/about`, [cit. 2016-09-04].

[5] EEA. Who we are. `http://www.eea.europa.eu/about-us`, [cit. 2016-09-06].

[6] ESPON. ESPON 2013 PROGRAMME. `http://www.espon.eu`, [cit. 2016-09-04].

[7] Bondy, J. A.; Murty, U. S. R. *Graph theory with applications*, chapter The incidence and adjacency matrices. New York: Elsevier Science Publishing Co., Inc., 1976, ISBN 0-444-19451-7, p. 7.

[8] Saad, Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2003.

[9] Dijkstra, E. W. *Numerische Mathematik 1*, chapter A Note on Two Problems in Connexion with Graphs. Mathematisch Centrum, Amsterdam, 1959, pp. 269–271.

[10] Bellman, R. E. On a Routing Problem. *The Quarterly of Applied Mathematics*, volume 16, 1958: pp. 87–90.

[11] Ford, L. R.; Fulkerson, D. R. *Flows in Networks.* Princeton, NJ: Princeton University Press, 1962.

[12] Warshall, S. A theorem on Boolean matrices. *Journal of the ACM*, volume 9, no. 1, January 1962: pp. 11–12.

[13] Floyd, R. W. Algorithm 97: Shortest Path. *Communications of the ACM*, volume 5, June 1956: p. 345.

[14] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition edition, 2001, 636–640 pp.

APPENDIX **A**

Acronyms

APSP All-pair shortest path

CERCO Comitée Européen des Responsables de la Cartographie Officielle

GUI Graphical user interface

DEM Digital Elevation Model

ERDF European Regional Development Fund

ESPON European Observation Network for Territorial Development and Cohesion

FOSS Free and Open-Source Software

GDAL Geospatial Data Abstraction Library

GIS Geographic information systems

MEGRIN Multi-purpose European Ground Related Information Network

OOP Object Oriented Programming

OSM OpenStreetMaps

SSSP Single-source shortest path

XML Extensible markup language

Appendix **B**

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
 exe ................................. the directory with executables
 src .................................the directory of source codes
   wbdcm .................................... implementation sources
   thesis .............the directory of LaTeX source codes of the thesis
 text ...................................the thesis text directory
   thesis.pdf........................the thesis text in PDF format
   thesis.ps...........................the thesis text in PS format
```