

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

February 13, 2017

Acknowledgements

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

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on February 13, 2017

.....

Czech Technical University in Prague
Faculty of Information Technology
© 2017 Lukáš Koc. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

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

Introduction	1
Motivation	1
Geographic information system	1
Problem description	2
Organisation of the Thesis	3
1 Data Analysis	5
1.1 Data fitness	5
1.2 Data format	5
1.3 Possible sources	7
1.4 Storage	10
1.5 Preprocessing	14
2 Design of the application	19
2.1 The shortest path algorithms	19
2.2 Dijkstra algorithm	20
2.3 Parallelization of Dijkstra algorithm	22
2.4 User interface design	23
3 Realisation	25
3.1 Technology used	25
3.2 Server	26
3.3 Client	31
4 Results	35
4.1 Speed	36
4.2 Preciseness	36
Conclusion	37

Bibliography	39
A Acronyms	41
B GDAL/OGR diagrams	43
C Contents of enclosed CD	45

List of Figures

1.1	Realization of raster and vector data representation, obtained from SQL Server Rider [1]	6
1.2	Representation of a graph using the method of an adjacency list	10
1.3	Road on a map represented within the data	16
1.4	Example of the transit and connecting nodes.	17
2.1	Design of fully optimal user interface.	23
3.1	User interface of the application	32
4.1	Dataset containing the transportation nodes in the Prague region	35
4.2	Generalized version of the main Prague dataset	36
B.1	Inheritance diagram for OGRGeometry	43

Introduction

Motivation

The 21st century is often titled as the century of data. Usage and gathering data influences our lives on a daily basis. Industry, fashion, society, transportation etc. is transforming under the amount of data gathered and evaluated.

Alongside the growing amount of data collected, borders of 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 the location is unknown. Finding meeting points with colleagues or friends, which might have limited knowledge about the area too, might turn into a difficult task.

Applying the knowledge of graph theory, geography and computer science, the goal of the thesis is to create an 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. Therefore, paralellization of every sophisticated application is becoming significant. 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, evolution of society and species. Visualization software, then, displays gathered data as 2D or 3D images [2].

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

Within this chapter, first it is necessary to define terms from graph theory, since they are being used in scope of the whole thesis.

Graph theory definitions

Definition 1. A **graph** G is an ordered pair $G = (V, E)$. Non-empty set V contains vertices (or nodes) and a set E contains edges. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints. A graph (without multiple edges) can have up to n^2 edges.

The number of vertices $|V|$ is denoted by N and the number of edges $|E|$ is denoted by m throughout this thesis.

Definition 2. Two vertices are called **adjacent** if they share a common edge, in which case the common edge is said to join the two vertices. An edge and a vertex on that edge are called **incident**.

Definition 3. 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 .

Definition 4. A **walk** in graph is any sequence of adjacent distinct edges. A **path** in graph is a sequence of vertices v_1, v_2, \dots, 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**. The **length** of a path is the number of edges that it uses.

Definition 5. Let $G = (V, E)$ be a directed graph, whose edges are weighted by the function $f : E \rightarrow \mathbb{R}$, associating each edge with the real number called the **weight**. The length of a path is the sum of the weights of its edges. This graph is called **weighted graph**.

In this sense the weights can be reinterpreted as the edge lengths. A cycle whose edges sum to a negative value is a **negative cycle**.

Definition 6. *Given a graph G , the **distance** $d(x,y)$ between two vertices x and y is the length of the shortest path from x to y , considering all possible paths in G from x to y .*

Shortest path problem

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 [3], to convert the original weighting function $f : A \rightarrow \mathbb{R}$ to non-negative function $f' : A \rightarrow \mathbb{R}_0^+$ 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, further, explains how the data can be stored and processed in order to reduce the number of unnecessary nodes.

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 decrease the computation time, the possible parallelization of Dijkstra algorithm is further explained. The application also requires a basic user interface, which will be described in Chapter 2.

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, in which the results based on the accuracy and time comparison between parallel and single threaded implementation are compared.

Data Analysis

1.1 Data fitness

As stated in Koc 2014 [4], 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 Data 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 a different format compared to data representing the location of a point 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

1. DATA ANALYSIS

graphic design world. The Figure 1.1, which graphically explains how these two types process given data, serves for a better understanding.

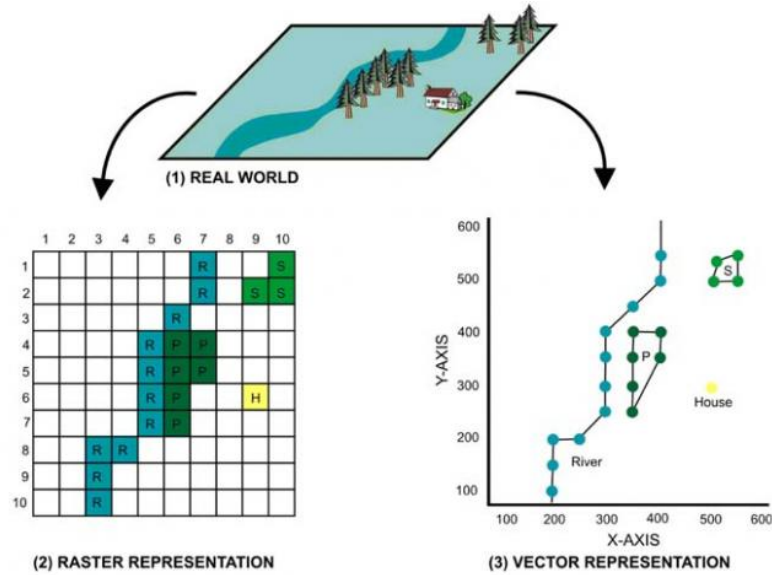


Figure 1.1: Realization of raster and vector data representation, obtained from SQL Server Rider [1]

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

1.2.1 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

Disadvantages

- Large amount of data

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

1.2.2 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

The idea of the Meeting Scheduler is to create the graph from the data provided. The graph is by the definition a topological space. Therefore, the vector representation makes creating the graph more straightforward. In addition, the raster representation is by its nature fitted to quickly provide graphical overview of the geographical data.

1.3 Possible sources

While searching for data, the main focus was not only on data in the vector type format, but also on sources providing free data of Europe available to the public. Further subsections describe sources which match the criteria mentioned in section 1.1.

1.3.1 EuroGeographics

EuroGeographics is the membership association consisting of 60 organizations and 46 countries. It was created in the year 2002, when the Comité 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 [5].

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 repeatedly in order to download the dataset would make the data processing to be a bottleneck.

1.3.2 OpenStreetMaps

OpenStreetMaps (OSM) is a project officially supported by the OSM Foundation. OSM was created to build and provide open¹ geographical data available to everyone.

The OSM project was inspired by Wikipedia and is working in the same manner: Users are contributing 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 .osm format, which follows XML rules.

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

- Geojson
- Shapefile

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 to operate with the data, since it is not official GIS format. However, projects as Mapzen provide also additional format varieties to choose from.

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

1.3.3 EEA

The European environmental agency (EEA) is an agency of the European Union providing information about the environment for the public. According to their official site [6] 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). Since the Meeting Scheduler application requires transportation datasets, the EEA does not provide fitting data.

1.3.4 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 ..." [7]

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

- XML
- Excel file

Unfortunately, the data provided by ESPON are having the same disadvantages as data provided by EEA in the previous section: maintainability. Our application requires 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 in order to analyse and deliver the goal. First phase starting in July 2008 until February 2011 followed by second phase having a lifetime span from February 2011 to December 2014.

As a data source the OpenStreetMaps were chosen, because the data provided are regularly maintained by the community. In addition, the documentation on OSM Wikipage contains all necessary information needed to learn the user how to immediately 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.).

1.4 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 lists are easy to implement and use. All vertices in a graph are mapped onto an array of pointers referencing to the first node of a linked list. In case a vertex does not have the adjacent vertices, its pointer is set to null. An 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.

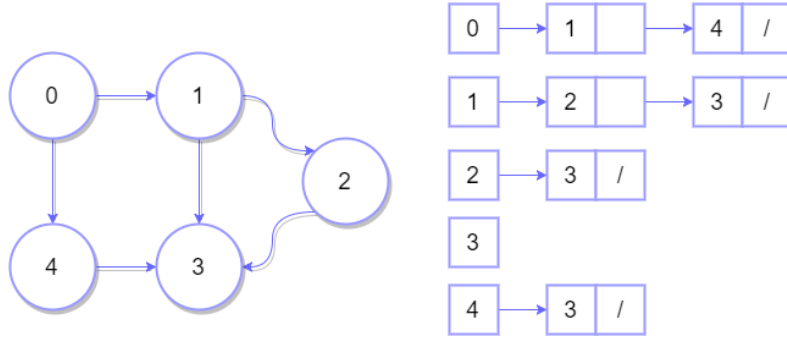


Figure 1.2: Representation of a graph using the method of an adjacency list

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, an 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 $\langle 0, \infty \rangle$, where 0

means two vertices are not adjacent and any non-zero value, meaning they are adjacent with an edge cost of that value[8]. 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 \mathbf{A} looks as followed:

$$\mathbf{A}(G) = \begin{matrix} & \begin{matrix} v_0 & v_1 & v_2 & v_3 & v_4 \end{matrix} \\ \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

Rows and columns represent vertices of a graph. In the case of matrix \mathbf{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 \mathbf{A} is directed, therefore the adjacency matrix is not symmetric.

1.4.3 Incidence matrix

Incidence matrix is very similar to the adjacency matrix, but instead of showing relations between vertices themselves it represents relations 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[8].

An interesting case is the incidence matrix for a directed graph. In that case the sign of the value within matrix \mathbf{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 \mathbf{M} looks as followed:

$$\mathbf{M}(G) = \begin{matrix} & \begin{matrix} e_0 & e_1 & e_2 & e_3 & e_4 & e_5 \end{matrix} \\ \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \end{matrix}$$

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 various 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

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

where E is the set of all edges and V the set of all vertices in the 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 the amount of outgoing edges \Rightarrow out-degree, which is

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

where E is the set of all edges, V is the set of all vertices in a graph and $\deg^-(v)$ function returns the 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 motivations 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 [9], the 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 the adjacency **A** matrix from section 1.4.2. Clearly this matrix contains less non-zero elements, therefore it is an example of a sparse matrix. Using the coordinate format, matrix **A** looks the following:

$$AA : \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$IR : \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 4 \end{bmatrix}$$

$$IC : \begin{bmatrix} 4 & 2 & 3 & 1 & 3 & 3 \end{bmatrix}$$

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 its 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 the fictional row begins on $n + 1$ position.

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

$$AA : \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$IR : \begin{bmatrix} 0 & 2 & 4 & 5 & 5 & 6 \end{bmatrix}$$

$$IC : \begin{bmatrix} 1 & 4 & 2 & 3 & 3 & 3 \end{bmatrix}$$

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 than 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:

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

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 1.3.2), 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 the distance between these two nodes is greater than zero.

The simplest solution to compute the 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 the distance between two real points, but in a two dimensional space. For computing the distance on the Earth, it is needed to use the great circle distances.

The great circle distance takes into account the curvature of the sphere, to provide more precise distance. Using great circle distance allow the application to provide decent approximation between every two nodes (obtained from the data-source) connected via one edge.

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

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

Currently, the Meeting Scheduler application can work with the selected data-source (Section 1.3). The graph is being represented as an adjacency list with weighted edges based on the real life distance. However, most of the world applications work with large amount of data. Therefore, in order to handle big amount of data, it is possible to use either clustering or node reduction, both described in following sections.

1.5.2 Graph clustering

Clustering is a method to group multiple nodes based on similarity into one set – a cluster, as described in Schaeffer [10]. However in some of the clustering literature, a cluster in a graph is called a community [11].

Applying clustering multiple times on different groups of nodes, leaves the graph to be represented with these clusters instead of the nodes. If node in first cluster has an edge to another node within the second cluster, first and second cluster are connected with a node as well.

1.5.3 Node reduction

The Meeting scheduler transforms real world spacial data into a graph. Spacial data are widely used to map and describe the Earth. In order to be accurately describe any object of the surrounding it is necessary to record every little detail. Figure 1.3 is displayed as an example. A road from point A to point B is interpreted as a bended line between two nodes. But in the spacial data it is represented by a group of linked nodes to depict the curvature of the road.

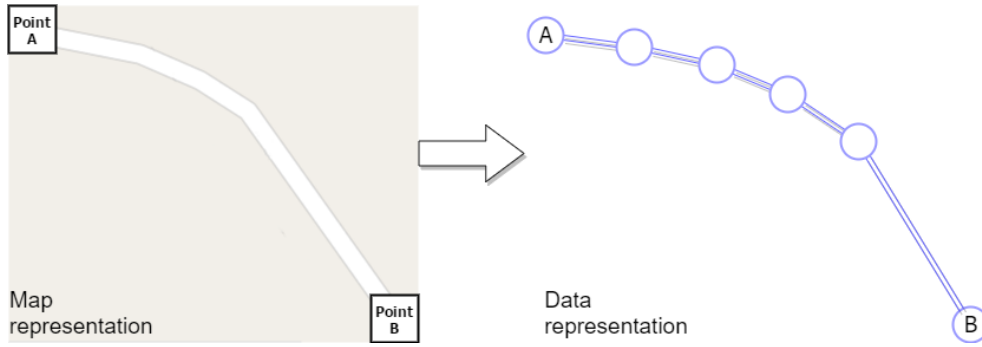


Figure 1.3: Road on a map represented within the data

For the purpose of finding meeting place, it is not required to have all the nodes forming the road saved in the memory. The road on Figure 1.3 could be represented only as pair of nodes A and B. That way the other 4 nodes representing the road would could be deleted and therefore saving memory.

The nodes legitimate for removing will be called *connecting* nodes. Within the scope of the thesis connecting node will be defined as following:

Definition 7. *Connecting node is a node having at most one outgoing edge and at most one incoming edge.*

Similar way it is possible to define transit node, which is the opposite of the connecting node.

Definition 8. *Transit node is a node having at least two outgoing edges or at least two incoming edges.*

Examples of connecting and transit nodes are depicted on Figure 1.4.

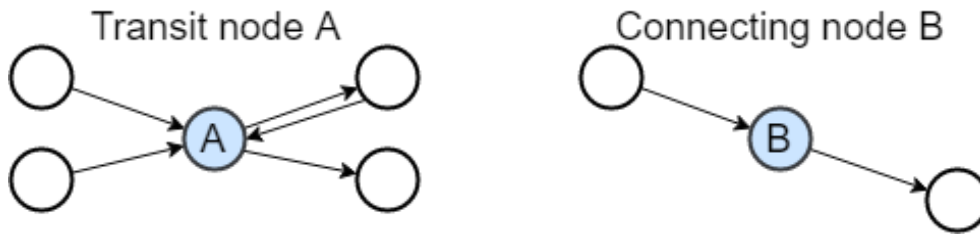


Figure 1.4: Example of the transit and connecting nodes.

Locating and removing connecting nodes from the data source significantly reduce the number of nodes within the graph. However, removing every connecting node would cause the loss of accuracy, because the meeting point would be only located on the intersections or junctions of the roads. It would not be possible to be able to meet in the middle of the street. Resolving the dilemma between memory and accuracy highly depends on the given data scenario. The possible solutions are either allowing the user to choose his own precision rate or leaving it up to the developer to decide how the data will be reduced. The solution to the problem used in this application will be further described in Realization Section 3.2.1.

Design of the application

2.1 The shortest path algorithms

Nowadays, for solving the shortest path problems exist 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 [12] and the Bellman-Ford algorithm [13, 14] 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 where 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 [15, 16] and Johnson's algorithm [3] 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 Dijkstra's algorithm on every node within the graph. For sparse graphs, Johnson's algorithm provides faster computation time than Floyd-Warshall algorithm [17].

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 positive 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 \mathbf{n} the function $\text{dist}(\mathbf{n})$ represents the distance from the starting node \mathbf{s} . For unreachable nodes the value returned by this function will be undefined. Next to the function dist , there is also the function $\text{prev}(\mathbf{n})$, which returns the node for the shortest path back to node \mathbf{s} .

```

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

```

Algorithm 1: Dijkstra algorithm

On the line number 6 in Algorithm 1, the choice of the node \mathbf{u} means choosing the \mathbf{u} with $\text{dist}(\mathbf{u}) \leq \text{dist}(\mathbf{w})$, in which \mathbf{w} is every other node having the state *open*.

2.2.2 Proof of correctness

In order to prove the algorithm will stop after a 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 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 that the set of closed nodes of a 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 steps executed. The theorem clearly is correct before and after the first step.

Let w be a node with a state set to closed in the last step. Let us consider a node v which is closed. If $v = w$, then the theorem is trivial. In the opposite case we will show, that there is the 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 the node with the lowest $\text{dist}(\mathbf{u})$ and dist of chosen nodes in each step represents a non-decreasing 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 the current step.

Due to the fact, that after the last step the set A contained 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 Algorithm 1, we can compute the time general complexity of Dijkstra algorithm. Let us assume we use the array in order to store the distances for all the N nodes. As proven in Theorem 1, the whole algorithm will execute at the most N steps, in each we are choosing the 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 the 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 the smallest one in time of $O(\log N)$ and adjusting the distances of the neighbours, which take $O(E \log n)$ 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 applications, we expect the graph having a form of a sparse graph, meaning for $M \ll N^2$ 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-Ford 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 for an 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 at the Meeting Scheduler application. This whole computation process leaves a lot of space for the performance optimization, especially the parallelization.

2.3 Parallelization of Dijkstra algorithm

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

2.4 User interface design

The application requires the spacial information as input. Further, the user should be able to insert the positions of people in order to find a meeting point, 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 users.

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. 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. Example of fully optimal GUI is displayed on Figure 2.1.

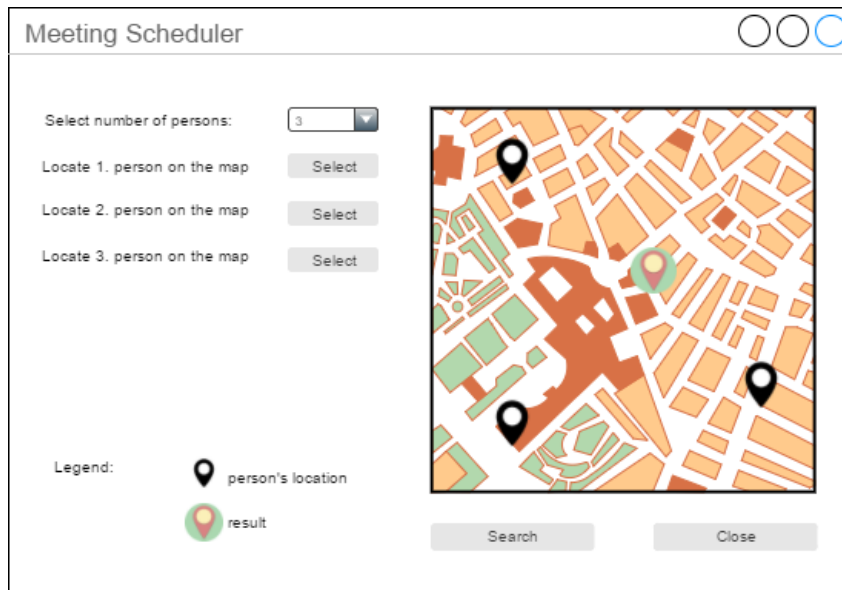


Figure 2.1: Design of fully optimal user interface.

The goal of the thesis is to create suboptimal user interface. First, the interface should provide the way to connect to the server, so the communication between server and client could be established. Further, there are several input boxes allowing the user to insert the position of the people. Last, the interface provides the user current status of the application through the logging service. Not only to deliver the result, but also to inform the user about the whole process, for example whether the data inserted are incorrect or the

2. DESIGN OF THE APPLICATION

connection has been lost. Creating suboptimal user interface is described in the next chapter in Section 3.3.

Realisation

3.1 Technology used

The goal of the thesis is to create complex desktop application with basic front-end to provide users proper control over the input data and the 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 the 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 (see Section 1.3), GDAL/ORG library provides optimal tools for reading and collecting information from our dataset.

3.2 Server

The Meeting Scheduler application is structured as a server-client model using TCP protocol. As defined in the Linux information project [18]: *“TCP uses error correction and data stream control techniques to ensure that packets to arrive at their intended destinations uncorrupted and in the correct sequence, thereby making the point-to-point connection virtually error-free.”*

This feature is expensive, when the amount of data is constant and large (e.g. file transfer). For data streaming is more appropriate user datagram protocol (UDP) [19]. However, the Meeting scheduler server and client communicate with using only small sized messages in form of coordinates, therefore the TCP provide optimal solution.

The *Server* class implementation in Listing 3.1 is straightforward and self-explanatory, providing methods to establish connection between client and server and allow further communication.

Listing 3.1: Server class

```
class Server
{
private:
    int sock, sockclient;
    struct sockaddr_in serv_addr, cli_addr;
    char buffer[512];
public:
    Server(int );
    int Listen();
    int ReadFromClient(char *);
    int WriteToClient(std::string );
    void CloseClientSocket();
    ~Server();
};
```

Server is responsible for providing client the results of their requests, more specifically meeting point based on input received. To achieve that, server should be having dataset available and preprocessed, so that computation can start without any delay. Hence, server is split into two parts: data processing module and computation module.

Data processing module is the part of the server ensuring the data are transformed in the corresponding directed graph. The graph will further be provided to another part of the server – computation module. Both modules are further described in the following sections.

3.2.1 Data processing module

Role of the data processing module is to transform gathered dataset into directed graph. The dataset is represented with the specific GIS file format, more specifically Shapefile format provided by team Mapzen (see Section 1.3.2). The dataset obtained has already been partially processed, so it contains only features describing transportation (e.g. road or rails). Data processing module is able to read the selected format and extract all the nodes from the transport points. These nodes represents vertices in the graph.

Afterwards, the edges are extracted from the relations of the nodes. In addition, each edge obtains the cost based on the great circle distance (see Section 1.5.1) between the points. Together with vertices the full graph is complete.

Based on the size of the graph, the amount of nodes can be reduced using node reduction or the structure of the graph can be modified by clustering. Both methods are described in Section 1.5. These changes to the graph obtained are shortening the computation time for the cost of accuracy.

3.2.1.1 Data-reader class

The application is reading data through the class called **DataReader**. The structure of the class is displayed in Listing 3.2. The constructor requires the string, which is representing the path to the Shapefile data-file. Within the constructor the GDAL library opens the file and set the pointer **GDALDataset * pDS** to the dataset within the input file.

Listing 3.2: DataReader class

```
class DataReader
{
private:
    GDALDataset * pDS;
    OGRLayer * pLayer;
    int featureCount;

public:
    DataReader(std::string );
    int InitLayer(int);
    NodePair GetNodesFromCurrentFeature() const;
    int GetLayerSize() const;
    double static DistanceBetweenTwoPoints(Node * , Node * );
    ~DataReader();
};
```

Having the dataset initialized, next step is to load the features within. Features are contained within the layers. Calling the method **InitLayer(int)**, the layer is initialised, setting the pointer **pLayer** to the first feature available. The argument of the function determines the index of the layer, if the dataset contains more layers. **OGRFeature** is a container holding a geometry and the attributes of the feature. Geometry types are displayed on the Figure B.1, obtained from official GDAL documentation [20].

After analysis of the dataset, the geometry of the road is represented as a **OGRLineString** class. Therefore, every feature contains two nodes connected together via an edge. Method **GetNodesFromCurrentFeature()** reads the first feature within the layer and returns a pair of nodes connected via edge in form of **NodePair** structure. **NodePair** structure contains two pointers to the **Node** class.

The last method within **DataReader** class to point out is static function **DistanceBetweenTwoPoints(Node*,Node*)**. The method takes two nodes as arguments and returns the great circle distance between them. It is static since it is not needed to be dependent on any object of the **DataReader** class.

3. REALISATION

Listing 3.3: Node class

```
class Node{
private:
    double lat;
    double lon;
    Node ** adjNodes;
    double * edgeW;
    int adjNodesCount;
    int maxAdjNodes;
public:
    Node( double, double);
    std::pair<double, double> GetCoords() const;
    double GetLat() const;
    double GetLon() const;
    int GetEdgesCount() const;
    Node ** GetEdges();
    std::pair<Node *, double> GetNode(int) const;
    int AddNeighbour(Node *, double);
    ~Node();
};
```

The class **Node** represents the vertex in the graph and its structure is shown on Listing 3.3. It stores the information about its position and weighted edges to other adjacent vertices. **Node** forms a foundation for the graph represented as **Graph** class, shown in Listing 3.4.

Listing 3.4: Graph class

```
class Graph
{
private:
    std::map<std::pair<double, double>, Node * > Nodes;
public:
    Graph();
    int AddNodePair(Node *, Node *);
    Node * GetNode(std::pair<double, double>);
    int ConstructGraph(DataReader *);
    int GetSize() const;
    Node * FindClosest(std::pair<double, double>);
    std::map<std::pair<double, double>, double * > CreateDistArray(int);
    static void RemoveDistArray(std::map<std::pair<double, double>, double * >
                                &);
    ~Graph();
};
```

Graph is storing nodes in the STL map structure, mapping the coordinates to a specific node. That causes each node to be stored once at the most. The class is constructed with **ConstructGraph(DataReader *)** using the **DataReader** as an argument. Within this method, the dataset is being read, every feature is being analysed and saved in form of the graph into **map<pair<double, double>, Node * > Nodes**.

Another noticeable method is **FindClosest(pair<double, double>)**. It is used to locate the closest node to the specific location. This method is required in order to locate the input of the user in the graph. Again, the method is using great circle distance to find the closest node to the input. On the other hand, it is needed to go through the whole graph and measure the distances for each node. This leaves the room for optimization of the application.

Except of the generic support methods, there are methods called **CreateDistArray(int)** and **RemoveDistArray(map<pair<double, double> > *)**. Distance array represents the structure, in which the results of the Dijkstra algorithm are stored.

It is a map, mapping the coordinates to the array of distance values. The size of the array is equal to the number of user inputs. Therefore, the distance array stores for every input the distance to every node in the graph. **RemoveDistArray(..)** method serves as a destructor for distance array.

The distance array is being used as a argument for the Dijkstra algorithm, further described in Section 3.2.2. After storing all the results for each input node, the last step is to go through all the vertices within array and select the one with the shortest distances to each input node.

3.2.2 Computation module

Computation module uses the graph constructed by the data processing module. Data processing is needed only at the start of the server or after update of the dataset. Thereafter, the server waits in the stand-by mode, expecting the client to send the input coordinates.

3.2.2.1 Algorithm

After receiving the input coordinates from the client, the application is ready to start the computation. The main computation algorithm is Dijkstra algorithm (see Section 2.2). Dijkstra algorithm is solving SSSP problem. The application is using Dijkstra algorithm for each of the input nodes, resulting having shortest distances to all nodes in the graph and choosing the most suitable one.

The version of sequential Dijkstra algorithm uses priority queue based on the distance to process every node in the graph. Priority queue affects the computation time of the algorithm based on the structure [21].

The priority queue needs to have operations:

ExtractMin to get vertex with minimum distance for all the vertices, whose shortest distance is not found yet

DecreaseKey to update the distance of adjacent vertices of currently extracted vertex; if the distance is smaller, update the information in the queue

The operations above are guaranteed by *Set* container in C++ STL. *Set* keeps all its keys in sorted order [22]. Hence, minimum distant vertex will always be at the beginning, allowing to extract the vertex with minimum distance (**ExtractMin**) and update distances of all other adjacent vertices accordingly (**DecreaseKey**) \Rightarrow if any vertex's distance become smaller, delete its previous entry and insert new updated entry.

The implementation of the sequential Dijkstra algorithm is located in Listing 3.5. The method has three arguments. First argument **g** represent the graph storing all the information about the nodes. Second argument **source**

3. REALISATION

is a pair of coordinates defining the input node. Third argument `dist` is a container, storing the distances from every input node to every other node in the graph. Last argument is an index of the input node.

Listing 3.5: Sequential Dijkstra algorithm

```
void MeetingScheduler::SequentialDijkstra(Graph * g ,
std::pair<double, double> source ,
std::map<std::pair<double, double>, double * > & dist ,
int input_index)
{
    // init priority queue based on the distance
    set< pair< double, PD > > pq;

    // insert first node to the queue
    pq.insert(make_pair( 0, source ));
    dist[source][input_index] = 0;

    while (!pq.empty()) {
        // ExtractMin from the queue
        pair< double, PD> tmp = *(pq.begin());
        pq.erase(pq.begin());
        PD u = tmp.second;
        Node * n = g->GetNode(u);

        // iterate through the adjacent nodes
        for (int i = 0; i < n->GetEdgesCount(); i++) {
            pair<Node*, double> node_edge = n->GetNode(i);
            double dist_temp = dist[u][input_index] + node_edge.second;
            if (dist[node_edge.first->GetCoords()][input_index] > dist_temp)
            {
                // removing already existing entry from the set;
                // for finalized nodes this step is unreachable
                if (dist[node_edge.first->GetCoords()][input_index] != DBLMAX)
                {
                    pq.erase(pq.find(make_pair(
                        dist[node_edge.first->GetCoords()][input_index],
                        node_edge.first->GetCoords())));
                }

                dist[node_edge.first->GetCoords()][input_index] = dist_temp;
                pq.insert(make_pair(
                    dist[node_edge.first->GetCoords()][input_index],
                    node_edge.first->GetCoords()));
            }
        }
    }
}
```

Time complexity depends of the structure used in *Set* container. According to SGI documentation [22], all the *Set* operations (e.g. insert, delete) take logarithmic time. Therefore, the solution in Listing 3.5 is $O(E \log V)$.

The method depicted in Listing 3.5 is called for each input node the server receives from the client. Result of each all is stored within one structure – `dist`. When Dijkstra algorithm finished computing the distances for every input, the next step is to find the node with the best combination of the distances.

3.2.2.2 Finding the result

As described in previous section, each node has the information about the shortest distance to all the input nodes. The optimal node needs to have the

shortest and uniform distances from every input node. Therefore, the summation and standard deviation is computed for every node.

The node with the shortest distances to every input node needs to have smallest summation of all the distances. The standard deviation of all the distances achieve the uniformity, therefore every person has it to the meeting place relatively equal (based on the value of the standard deviation).

The standard deviation (represented as a Greek letter σ) in statistics measure the amount of variation or dispersion of a set of data values [23]. The smaller the value of the standard deviation is, the more similar the values are. Choosing the result with the smallest standard deviation ensures the uniformity of the result. The formula used to compute discrete standard deviation for the node v look as follows:

$$\sigma_v = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

where N is the amount of input nodes, x_i is distance from the input at index i to the node v and \bar{x} is mean value of all the distances to the node v .

The optimal node needs to have the summation and standard deviation of all the distances to smallest. With growing summation of distances, the fitness of the node is less desired. The same applies to the standard deviation of the distances. Hence, to it is possible to put the summation and standard deviation into the relation. The fitness value of the node is represented as a multiplication of those two values. The node with the smallest fitness value is the optimal node.

3.2.2.3 Analysis of the fitness

In previous paragraph was defined the fitness value of the node as a multiplication of summation and standard deviation. Since the distance values are from $[0, \infty)$ interval, the summation of them can be from $[0, \infty)$ interval as well. If the summation of the distances is 0, then all the distances are also 0. That means all the input nodes are located in the same location. This edge case is already handled in the UI layer of the application, therefore the summation is from the interval $(0, \infty)$.

The standard deviation is in general (as well as the summation) from $[0, \infty)$. If standard deviation equals 0, it means all the values are the same. Ideally, the optimal node should have standard deviation of the distances equal to 0. However, this fact can completely override the contribution of the summation in the fitness value. Using real world data, it is possible that input nodes have equal distances to multiple nodes, but with different values ². In Figure is shown a simple example for two input nodes.

²with increasing number of input nodes the probability of having σ decreases

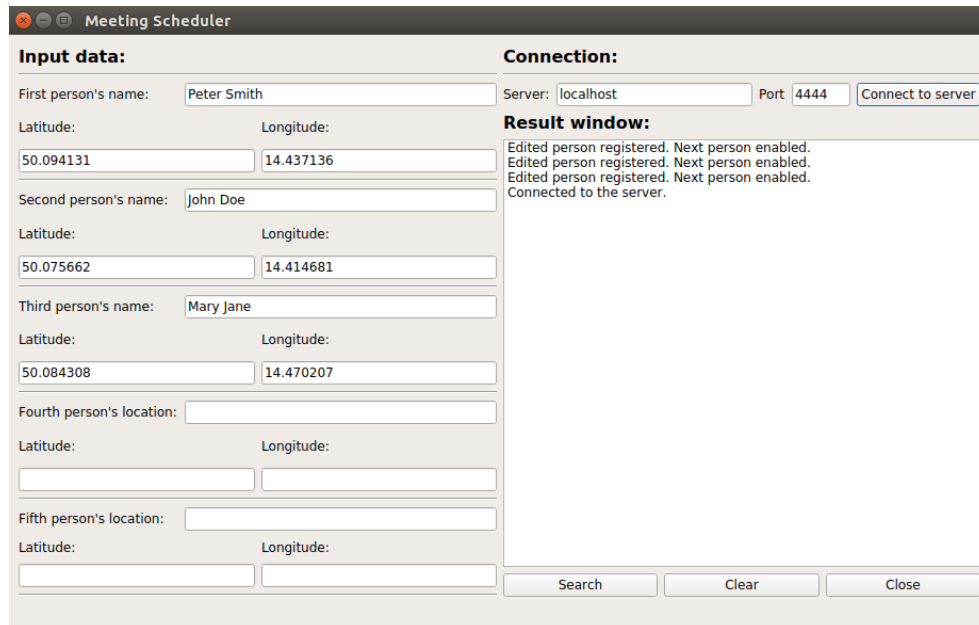
3. REALISATION

Therefore, there is a special handling of this case, where standard deviation equals zero. During the processing of all the nodes the summation of the currently optimal node is stored. If another node with same fitness is found, node with the smaller summation is chosen as the optimal node.

3.2.2.4 Parallelization

3.3 Client

Client provides basic GUI for the user. The user thought the client connects to the remote server and send the server locations of the people. The appearance of the current UI was created by Qt framework as mentioned in the beginning of the Section 3.1. According to Section 2.4, the final version of the user interface is presented in the Figure 3.1.



The screenshot shows a Qt-based application window titled "Meeting Scheduler". The interface is divided into two main sections: "Input data:" on the left and "Connection:" on the right. The "Input data:" section contains five sets of input fields for person names and their locations (latitude and longitude). The first person's name is "Peter Smith" and their location is (50.094131, 14.437136). The second person's name is "John Doe" and their location is (50.075662, 14.414681). The third person's name is "Mary Jane" and their location is (50.084308, 14.470207). The fourth and fifth person's locations are currently empty. The "Connection:" section includes a "Server:" field set to "localhost", a "Port:" field set to "4444", and a "Connect to server" button. Below the "Connection:" section is a "Result window:" which displays a log of events: "Edited person registered. Next person enabled." (repeated three times) and "Connected to the server." At the bottom of the window are three buttons: "Search", "Clear", and "Close".

Figure 3.1: User interface of the application

The client is split into left and right section. Left section enables the user to enter the locations of the people. The edit blocks for other persons are uneditable, until every person before contains valid input. For example, after entering location and name of the first person, the fields for the second person become editable. After entering proper data into each person's fields, the user is notified in the *Result window* the next person is available.

The input required for the application is only set of latitudes and longitudes in order to find the meeting place. However, the user interface allows the user

to enter the name of the person. This enhances the general user flow, providing the user to associate the location of the coordinates with the name.

In the right section of the client the *Result window* and *Connection* is located. After every action user makes, the *Result window* updates according to the current state of the client or server.

Connection allows the user to connect to the server using server address and port. The server and client are connected via transmission control protocol (TCP) as explained in the beginning of the Section 3.2. After setting up the connection with the server it is possible to begin the algorithm by pressing *Search* button. Result is afterwards displayed in the *Result window*. Button *Clear* erases all the message in the *Result window*. At last, button *Close* naturally closes the application.

Implemented client is not far of fully optimal user interface mentioned in Section 2.4 and meets all the requirements of the sub-optimal version of the UI.

Results

For testing as the dataset was chosen the city Prague, capital of Czech republic. The dataset is available in two versions. First version consist of all the transport nodes available within the area. The dataset is displayed with QGIS Desktop software on the Figure 4.1.

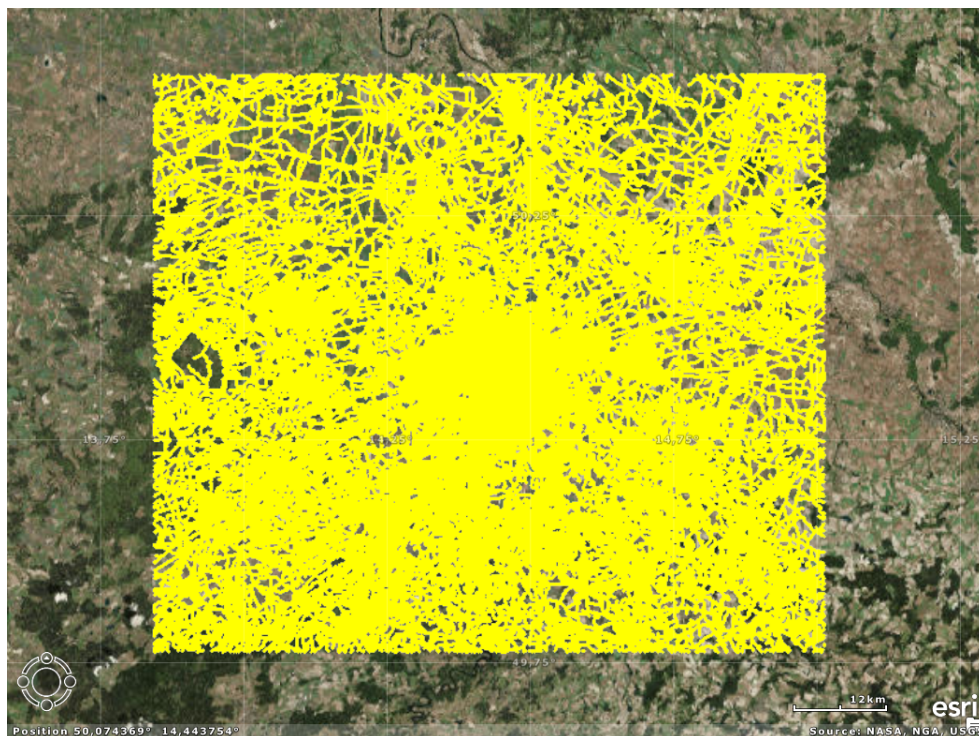


Figure 4.1: Dataset containing the transportation nodes in the Prague region

The second version has reduced the number of the nodes using the node

4. RESULTS

reduction technique (see Section 1.5.3) with tolerance 200 meters. After every 200 meters one connecting node is being removed. The second dataset is also displayed with QGIS Desktop software on the Figure 4.2.

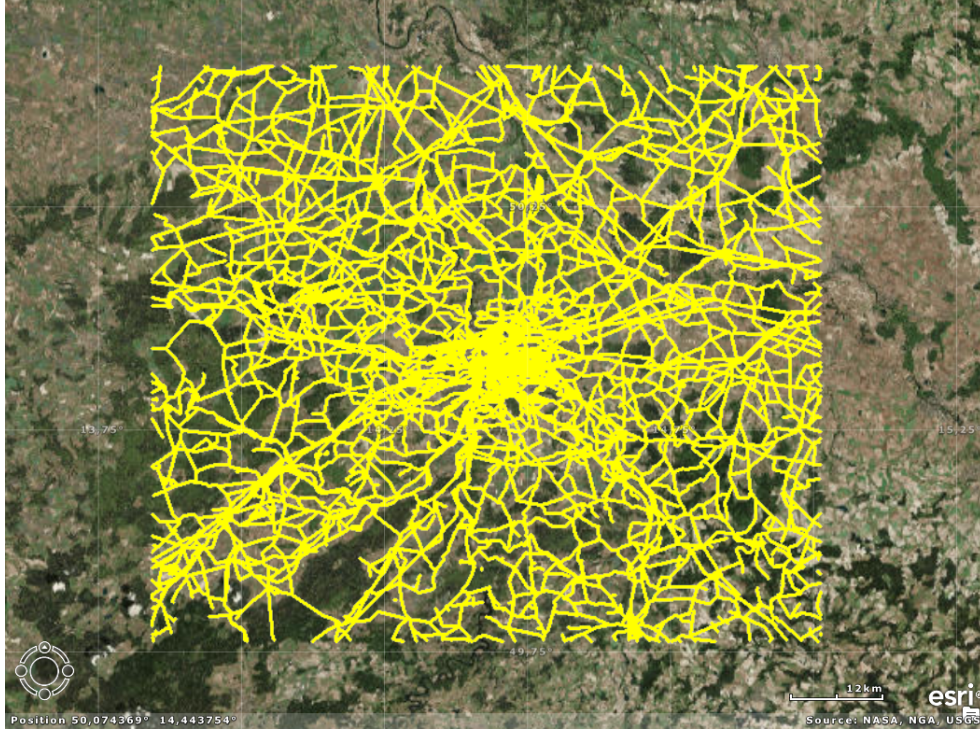


Figure 4.2: Generalized version of the main Prague dataset

The clustering method described in the Section 1.5.2 was not used in order to better measure the differences between sequential and parallel solution for the given problem.

4.1 Speed

4.2 Preciseness

Conclusion

Bibliography

- [1] Ayyappan, A. GeoSpatial – GIS File Formats. 2013, [accessed: 2016-08-26]. Available from: <https://sqlserverrider.files.wordpress.com/2013/10/raster-vector-gis-i4.jpg>
- [2] Chen, J. X. Geographic Information Systems. *Computing in Science & Engineering*, Jan.–Feb. 2010: pp. 8–9, ISSN 1521-9615.
- [3] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the Association for Computing Machinery*, volume 24, 1977: pp. 1–13.
- [4] Koc, L. *Plánovač setkání*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.
- [5] EuroGeographics. About Us. <http://www.eurogeographics.org/about>, [accessed: 2016-09-04].
- [6] EEA. Who we are. <http://www.eea.europa.eu/about-us>, [accessed: 2016-09-06].
- [7] ESPON. ESPON 2013 PROGRAMME. <http://www.espon.eu>, [accessed: 2016-09-04].
- [8] 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.
- [9] Saad, Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2003.
- [10] Schaeffer, S. E. Graph clustering. *Computer Science Review*, volume 1, no. 1, August 2007: pp. 27–64.

BIBLIOGRAPHY

- [11] Newman, M.; Grivan, M. Finding and evaluating community structure in networks. *Physical Review*, 2004.
- [12] Dijkstra, E. W. *Numerische Mathematik 1*, chapter A Note on Two Problems in Connexion with Graphs. Mathematisch Centrum, Amsterdam, 1959, pp. 269–271.
- [13] Bellman, R. E. On a Routing Problem. *The Quarterly of Applied Mathematics*, volume 16, 1958: pp. 87–90.
- [14] Ford, L. R.; Fulkerson, D. R. *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.
- [15] Warshall, S. A theorem on Boolean matrices. *Journal of the ACM*, volume 9, no. 1, January 1962: pp. 11–12.
- [16] Floyd, R. W. Algorithm 97: Shortest Path. *Communications of the ACM*, volume 5, June 1956: p. 345.
- [17] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition edition, 2001, 636–640 pp.
- [18] Project, T. L. I. TCP Definition. <http://www.linfo.org/tcp.html>, [accessed: 2016-12-11].
- [19] Kurose, J. F.; Ross, K. W. *Computer Networking: A Top-Down Approach*. Boston, MA: Pearson Education, fifth edition edition, 2010.
- [20] GDAL - Geospatial Data Abstraction Library. <http://www.gdal.org/>, accessed: 2016-11-30.
- [21] Chen, M.; Chowdhury, R. A.; et al. Priority Queues and Dijkstra’s Algorithm. Technical report, Austin, Texas: The University of Texas at Austin, October 2007.
- [22] SGI. Standard Template Library Programmer’s Guide. <http://www.sgi.com/tech/stl/1>, [accessed: 2017-1-1].
- [23] Bland, J. M.; Altman, D. G. Measurement error. *BJM: British Medical Journal*, volume 312, June 1996: p. 744.

Acronyms

APSP All-pair shortest path

CERCO Comité 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

TCP Transmission Client Protocol

XML Extensible markup language

GDAL/OGR diagrams

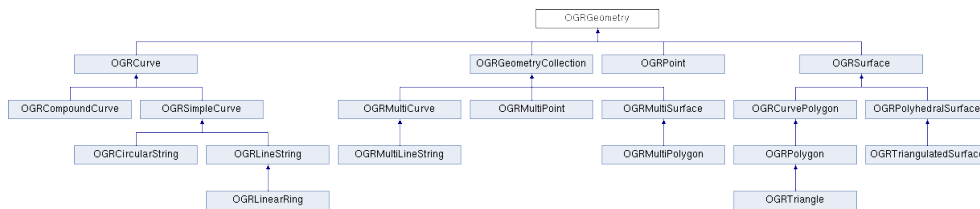


Figure B.1: Inheritance diagram for OGRGeometry

Contents of enclosed CD

readme.txt	the file with CD contents description
src	the directory of source codes
├── server	folder containing server sources
│ ├── graph	folder containing graph sources
│ ├── node	folder containing node sources used by graph
│ ├── data_reader	folder containing data reading sources
│ └── dijkstra	folder containing sources of Dijkstra algorithm
└── client	folder containing client 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