UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

IMPLEMENTATION OF SELECTED ROUTING ALGORITHMS FOR NAVIGATION

BAKALÁRSKA PRÁCA

2012 Martin Kolínek

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

IMPLEMENTATION OF SELECTED ROUTING ALGORITHMS FOR NAVIGATION

BAKALÁRSKA PRÁCA

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky FMFI
Školitel:	RNDr. Richard Ostertág, PhD.

Bratislava 2012

Martin Kolínek

60900958

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Martin Kolínek
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Implementácia vybraných algoritmov pre plánovanie trás

Cieľ: Cieľom práce je implementácia vybraných algoritmov pre plánovanie trás. Okrem toho v práci bude implementované jednoduché zobrazovanie dát exportovaných z openstreetmap v XML formáte a výpočet dosiahnuteľných miest v danej vzdialenosti a danom čase. Implementácia má byť v jazyku Haskell alebo C++.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 06.10.2011

Dátum schválenia: 10.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

60900958

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Martin Kolínek
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Implementation of selected routing algorithms for navigation

Aim: Goal of the thesis is implementation of:
* selected routing algorithms for navigation
* application for simple viewing of exported data from openstreetmap in XML format
* algorithms for reachability in selected max time or max distance

Haskell or C++ language will be used for implementation.

Supervisor:	RNDr. Richard Ostertág, PhD.
Department:	FMFI.KI - Department of Computer Science
Assigned:	06.10.2011

Approved: 10.10.2011 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

....

Student Supervisor

Chcel by som sa poďakovat svojmu vedúcemu, rodine, spolužiakom, priateľom a všetkým ostatným, bez ktorých by táto práca nemohla vzniknúť.

Abstrakt

Táto práca sa zaoberá tvorbou niekoľkých programov, ktorých úlohou je demonštrovať algoritmy na plánovanie trasy pre účely navigácie. Ako mapové podklady používa dáta z projektu OpenStreetMap.

KĽÚČOVÉ SLOVÁ: navigácia, plánovanie trasy, OpenStreetMap

Abstract

This thesis describes creation of a suite of programs which demonstrate some algorithms for route planning in car navigation systems. It uses mapping data from the OpenStreetMap project.

KEYWORDS: navigation, route planning algorithms, OpenStreetMap

Contents

Introduction

Car navigation systems have become very popular in recent years. Their goal is to allow drivers to navigate in a possibly unknown area. Usually they consist of several components. First there are maps which are essential to be able to do anything since they are the only source of information relevant to navigation. Next, some way of determining current position of driver is necessary. While simple navigation suites use only GPS signal or manual input, the more sophisticated may use an accelerometer to determine position more precisely even when GPS signal is lost. Then there are components which provide input for the driver be it a speech synthesiser giving directions or a map renderer which simply displays the map with the route highlighted. But to give this information to the driver, we first need to obtain it. This is what route planning algorithms are for.

Given starting point, destination and a suitable map representation, route planning algorithms try to find the most efficient way from the start to the destination. The map is usually represented as a weighted graph with edges being roads and nodes their intersections.

Most of the mapping data is available under restrictive licenses which prohibit its use in route planning software. However there is OpenStreetMap which comes with a permissive license and its mapping data have more or less sufficient quality.

This thesis deals with creation of several programs which allow their user to create a map suitable for queries for shortest route. It also shows the implementation of two simple algorithms for routing and compares their performance. Another implemented feature is searching for places from a starting node within given cost.

In the first chapter the more advanced approaches to route planning are introduced. In the second chapter the technologies used in the project are described. The third chapter summarizes the principles behind the project and the fourth chapter describes implementation of these principles. The final fifth chapter presents the results.

Chapter 1

Current state of route planning solutions

Navigation software is popular and several implementations of routing exist. Many companies offer navigation software, be it in dedicated hardware or software applications which can be used on smartphones, tablets and other devices. There are also web based services which help with route planning.

Each of these solutions uses some route planning algorithm. Development in recent years was very active with several interesting results. Basic algorithm useful for these purposes is Dijkstra's algorithm for finding the shortest path in a graph without negative cycles[6]. While this algorithm has good worst case time bounds, these are not sufficient for graphs of large size like maps of whole countries or continents[6]. This leads to requirement of better algorithm. Without preprocessing the worst case scenario requires us to read all nodes and edges of a graph which is an $O(n + m)$ operation (with n nodes and m edges). Even if algorithm which achieved this bound existed, it would still be too slow for practical use in navigation.

1.1 Approaches

This means there is need for some form of preprocessing. Several approaches were studied during the last few years. In [6] an overview of accomplishments during this time can be found. In this section some of these approaches will be introduced.

In [7], three basic types of approaches are introduced.

Time independent . In time independent route planning the cost of moving from one point to another is constant.

Time dependent . In time dependent route planning the cost of moving from one point to another may depend on the time of departure from the starting point.

Stochastic . In stochastic route planning, the cost of moving from one point to another is random to some degree.

Most of the approaches introduced in this section achieve best results in the time independent scenario.

1.1.1 Highway Hierarchies

This approach is based on how many commercial solutions do route planning. They are using the fact that in longer queries, less important roads are used only at the start and at the end of the route[20]. This does not always lead to optimal solution, but the solution is usually satisfactory [1].

In highway hierarchies, roads that are used in a shortest path between some nodes that are not too close to each other, are considered important. This ensures that not important roads really do not need to be considered in longer queries. What 'not close to each other' means is an input parameter for this method.

It may seem that determining whether a road is on some shortest path would be computationally intensive. In [20] they show an algorithm which can be used and is quite fast.

Another important advantage of this approach is that the process of determining not important roads and removing them from searches for longer routes can be iterated, creating a hierarchy of more and more important roads.

This approach shows considerable speedups [6], however it does not seem to be used in any popular navigation solution.

1.1.2 Graph partitioning

Other possibility which gives reasonable results is partitioning the graph into multiple cells with some good properties and precomputing the shortest routes between each pair of boundary nodes within each such cell. This allows for a smaller graph which is composed of whole cells in which start and destination nodes belong, and other cells contracted into cliques of boundary nodes connected by edges with costs equal to costs of shortest paths between them. This graph is then used in a shortest path finding algorithm.

In [7] they consider using this approach for time-independent and time-dependent route planning, which is an improvement over most of other techniques where time dependent planning is not covered much. They also show that partitioning the graph correctly is not an easy task and present some solutions.

[1]otherwise these systems would not be used

While being older than other approaches, it still achieves good results. Bing maps use similar approach [13] described in [5]. Their implementation seems to be faster than implementation in [7] and is fast enough for practical usage.

1.1.3 A*/Landmarks/Triangle inequality

This approach, studied in [8] uses conventional approach of A* with an advanced heuristic to speed up the search. Simple heuristic usable for A* include euclidean distance. Heuristic used in this method is based on precomputed distances to several landmark nodes. Then the triangle inequality allows the difference between distances to given landmark from start and destination nodes to be used as an admissible heuristic i.e. let $dist(u, v)$ be distance between two nodes u and v, then let L be a landmark node, then $d_L(v) = dist(v, L)$ is distance of node v from landmark L. Triangle inequality ensures

$$\begin{aligned} d_L(u) &\leq d_L(v) + dist(u, v) \\ d_L(u) - d_L(v) &\leq dist(u, v). \end{aligned}$$

This shows that $d_L(u) - d_L(v)$ is an admissible heuristic for every landmark L. Taking maximum of these values for each landmark is a useful heuristic which can reduce the amount of examined nodes by a considerable factor [8]. On the other hand this is not enough for a serious improvemen. In [6] only speedup of factor 3 to 5 is observed.

1.1.4 Transit nodes

This approach is based on the observation that most long journeys pass through one of a limited set of nodes. Other key point is that when starting a long route from a node, first such node encountered is one of very small number of nodes [2]. The first set of nodes was called designated transit nodes and according to [2], there are roughly 10000 of them in the USA.

Precomputing the distances between transit nodes and the distances from each node to the first transit nodes encountered on longer route allows for faster queries by reducing search graph size. According to [6] this leads to enormous speedups of the route searching process.

Chapter 2

Used technologies

The choice of technologies used throughout the project was not clear from the beginning. Several options were considered and some proved to be better than others.

2.1 Programming language

Haskell as an advanced functional language with concise syntax and modern features like laziness, strong typing and type inference was the first choice. However some problems had to be overcome.

1. The libraries available are not as mature and proven as libraries for older and more mainstream languages. For example there were several options for database access. The first one we tried was HDBC, one of the most used database abstraction layers. It follows the usual design of database engine specific driver and an abstraction layer which allows users to issue sql commands and read results in a way independent of the driver used. The underlying database to be used was sqlite (more on that in 2.2). The sqlite HDBC driver used strings for storing all the values, converting between native types and string representation on every access to the database.

2. The Haskell language itself has its shortcomings. Laziness introduces new kind of issues not usually seen by programmers not used to using it and debugging them is not easy. One of such issues is running out of available memory by storing the steps to compute a value instead of the value itself. It is not easy to force haskell (or at least the most popular implementation, ghc) to evaluate such computations and thus reduce memory footprint.

3. Pure functionality, which is forced by Haskell, is also not easy to work with for programmers inexperienced with functional programming and may introduce unneeded verbosity. For example carrying some state around is often needed and though functional languages provide features which allow it in concise way (e.g. monads) but they are rather hard to use for people unfamiliar with them. This leads them to give functions additional parameters and return types which becomes overly verbose.

These three were eventually the main reasons for dropping Haskell and moving to a more conventional language. C++ seemed like a good choice. With C++11 finally getting close to being standardized, many new features which make C++ a more friendly language have become more available in compilers and better documented.

This has led us to choose C++ as a primary language in this project. We tried to use many of the new features to ease implementation and increase code readability.

2.2 Database

In the beginning sqlite seemed like a good choice for its simple usage with basically no setup, availability for many different platforms including mobile devices and rather simple and well documented interface. However it didn't take long for its shortcomings to manifest themselves. With the amounts of data involved, advanced query optimizations simplify queries by a considerable factor. While sqlite can be really fast, such optimizations are not one of its strong points.

Since mapping data are involved, range queries were required and sqlite does not support them much. There exists an R* Tree module that should allow indexing for fast range queries. But this was not used since at the time this was discovered, PostgreSQL already seemed to be a better alternative.

PostgreSQL has more features than sqlite which allowed for easier implementation of storage backend. For example there is PostGIS, a spatial extension to PostgreSQL database, which really simplifies spatial queries i.e. queries involving positional data. PostGIS is also used in this project.

As for database access, several PostgreSQL specific and database engine independent libraries exist. The native PostgreSQL C API libpq was chosen because it is well documented and stable. Upon libpq a C++ abstraction layer was formed. This will be covered in section 4.1.

2.3 Libraries

We used several C++ libraries for various tasks throughout the project.

2.3.1 Graphics

Since we were working with mapping data, there was need for visual output which required a graphics library. Vector graphics is the main method for drawing maps. For this reason we required the ability to work with vector graphics easily. Other desired features included good documentation, simple usage, easy integration with code which didn't have it in mind and ideally small size (thus concentration on graphics and not too many other things). GTK+ meets most of these requirements but it is written in C programming language. However there is gtkmm, C++ bindings for GTK+.

GTK+ uses cairo for drawing, which allows for easy hardware acceleration [4]. Furthermore it has C++ bindings (cairomm). While the documentation is not very verbose it was sufficient to enable us to implement the simple drawing we needed.

2.3.2 Boost

Boost [3] is a set of well known C++ libraries which simplify many tasks and extend the functionality of C++ language. The components used in this project include regular expressions, program options, filesystem, range, tests and signals.

The most obvious use of Boost are tests which are written using Boost Test library. In our project are about 60 written and 180 more generated[1] tests. Boost Test allowed for simple definition of test cases and test suites and provided a runtime environment for running these test cases selectively.

Regular expressions are very useful for parsing simple data which is exactly how they are used in this project. Using Boost Regex library simplified regular expression usage and thus eased reading of some configuration files.

Most command line applications need command line parameters. To avoid hard to remember sequences of parameters, command line options are used. There are many libraries which provide this functionality e.g. getopt but Boost Program Options achieves the same thing and Boost was already used in this project. It provides nicely formatted help outputs and has simple interface which makes reading program options easy.

The Boost Range library is used for processing of larger amounts of data when the whole data might not fit into memory and lazy collections are used instead. Boost Range provides a functional approach to collection processing which simplifies some tasks.

Filesystem library was used to communicate with filesystem (creating and deleting some directories) in a cross platform way and signals are used in an event driven approach to XML file parsing.

2.3.3 CodeSynthesis XSD

Since OpenStreetMap map resources are used there is the need to read their data format. This is an XML with a fairly simple schema. Since this data can be huge and might not fit into memory it is impossible to use a DOM parser and thus a SAX parser is used. When the project started Expat was used for parsing.

Since we wanted to validate whether input XML was in the correct format XML schema validation requirement arose. This lead to discovering CodeSynthesis XSD. What it does is generate C++ code which parses and validates XML according to supplied XML schema. This greatly simplifies the XML parsing code and provides simple input error detection.

[1]testing SQL statements, more in 4.1

2.4 Other tools

Other support tools used in this project are CMake, Perl and Doxygen.

Building C++ applications is not an easy task, especially in different environments. There are tools which simplify this task such as autotools, scons or CMake. Because there was previous experience using CMake we leaned towards CMake. CMake allowed for automatic code generation during build, simple dependencies resolution and build configuration.

Doxygen being one of the most popular documentation tools is used for API documentation. Since most of the code is to be used later, having documentation is desirable.

Perl is a scripting language which is used for some code generation in this project.

Chapter 3

Principles

This chapter covers how this project works, listing concepts and approaches used.

3.1 Maps

Since route planning algorithm needs maps on which to work, we needed some mapping data to work with. There are several sources of mapping data with various restrictions on allowed usage. There are commercial maps which may be provided to public under some conditions. However for our purposes, maps with no restrictions on usage are more suitable.

Currently only one such mapping data exists – OpenStreetMap. Data from OpenStreetMap is available for any purpose under one condition, which is that modifications to this data is made available under a compatible license. [10]

3.1.1 OpenStreetMap data availability

OpenStreetMap data is available in rendered form on their web page [11]. This form is useless for us since we need data in a structured way which allows us to construct a graph on which to search for routes. Reading such data from a rendered map is nearly impossible.

Luckily OpenStreetMap provides exports of their data in a more structured format. It is possible to download data straight from their web page but this is limited to small regions. However there also exist periodical snapshots of large areas (cities, countries or continents) in the form of planet.osm [14]. There are many providers of this kind of exports with various selection of regions and frequencies with which data is exported.

3.1.2 OpenStreetMap data format

Data is exported in XML format with incomplete schema available in [1]. OpenStreetMap maps consist of three basic primitives – nodes, ways and relations. Each of these primitives has simple key/value attributes which define what given element models. This way of storing data seems sufficient for all mapping needs of OpenStreetMap.

Nodes represent points on the map. Each has a latitude and a longitude which define its real world position. Latitude and longitude are floating point numbers representing given coordinate in degrees using the WGS84 standard [9]. Nodes are used to model objects that do not have significant dimensions e.g. traffic signs, traffic lights, ATMs etc. Nodes are also used to define points of interest and define trajectory of ways.

Ways represent curves. Apart from attributes they have a sequence of nodes. These nodes define the trajectory of the way. Ways are used to model most of the map features as they can represent roads, boundaries, rivers etc. Ways are also used to model areas. These are represented by closed ways with attributes which suggest this way is an area.

Relations are used to model relationships between other elements. Each relation consists of several elements with each element having a role. This combined with attributes allows for representing things such as turn restrictions, multipolygons, bus routes etc.

3.2 Import

After retrieving data from OpenStreetMap in XML format, these data needs to be imported into some kind of storage which allows for more efficient retrieval. A relational database was used as such storage. The schema of this database was chosen to mimic the way OpenStreetMap stores mapping data. This means that nodes, ways and relations were stored, with nodes having position, ways referencing member nodes and relations having members. All of these elements also have attributes.

Having data stored in a relational database allowed for complicated queries with minimal effort. This later proved to be very valuable when constructing some structures to speed up some operations.

However using separate database means the data first needs to be transferred to this database. This led to creation of an `importer` program with the purpose to parse the XML document and copy appropriate data to appropriate tables. This importer program also removes artifacts such as incomplete ways from imported data. Such artifacts originate from the fact that when downloading XML from planet.osm these data are cropped by the boundaries of the region. Some ways and relations cross the boundary though and the elements they reference may be removed by this cropping.

3.3 Route searching

We were searching for the shortest route between the starting point and the destination point. Route searching is done using the basic Dijkstra algorithm and A* algorithm using great-circle distance as heuristic.

The great circle distance is the distance between two points on a sphere measured along the shortest path between the two points which goes along the surface of the sphere.

There was also a requirement for finding all roads reachable from a starting point within given cost. This problem was solved using a modification of Dijkstra's algorithm.

A graph is required for these algorithms. This graph is extracted from the mapping data in the database using the `roaddbcreate` program. This program extracts the relevant ways from the database, assigns a cost to each edge on this way and then stores these edges in the database in another table.

What these relevant ways are is dependent on how are we going to use the route. When in a car these are ways which model roads where cars may enter. Which ways these are is determined by way attributes. Attributes with key highway mark roads and paths. The value determines what kind of a road or path it is e.g. value primary means a rather important road and value residential means a small road in an urban area.

Since ways are composed of nodes, an edge can be created for each pair of nodes on a way. For every such edge a cost is computed based on position of its endpoints. This cost is currently computed as the great-circle distance between the endpoints. This means that when a shortest path on this graph is found it represents the shortest route.

Problem is that in real maps there are restrictions on which routes are valid. For example it is illegal to enter a one way street in wrong direction. Some turns are also illegal. We need to represent these restrictions in the graph we are going to use for route planning.

3.3.1 One way streets

One way streets are easier to handle as the created graph can have unidirectional edges and two edges are created for streets that are not one way.

The only remaining problem is to determine which streets are one way. This proved to be harder than expected for OpenStreetMap. Which ways are one way is determined by an attribute with key `oneway`. This attribute can have multiple values, some determine the way is one way others that it is not.

When a way is marked as one way a vehicle can pass the nodes which define the way only in the order in which they are referenced by the way. However there is an exception to this rule. When the value of the one way attribute is `-1` then the nodes may be only passed in the opposite order [12].

Also, there are attributes which when present in a way imply that the road is one way [12]. However these attributes are inconsistent [21]. Specifically there are instances where way with attribute `highway=motorway_link` without a `oneway` attribute is considered one way and also instances where it is considered two way. Currently the `roaddbcreate` program considers the roads marked `highway=motorway_link` a one way road.

3.3.2 Turn restrictions

Turn restrictions i.e. illegal turns are modeled as relations in OpenStreetMap. The relation has an attribute `type=restriction` and other attributes define what is restricted. For purposes of routing only two types of restriction need to be differentiated. They differ in whether attribute with key restriction has a value starting with `no` or starting with `only`.

The `only` restrictions model the fact that when going from a way which is the member of this relation with role `from` the only legal destination is the way with the role `to` and it is possible to pass only through elements which are members with role `via` [19].

The `no` restrictions model the fact that it is impossible to go from the `from` way to `to` way via the elements with role `via` [19].

The `via` member is usually only one node which is the common node of the `from` way and the `to` way. However there are cases when there is a way with role `via` or there are even multiple elements with the role `via`. One such example is a forbidden U turn through a railroad while turning left through it is possible (see figure 3.1). This can be modeled by making the way A in the figure the `from` member, the way C `to` member and way B the `via` member in a `no` restriction.

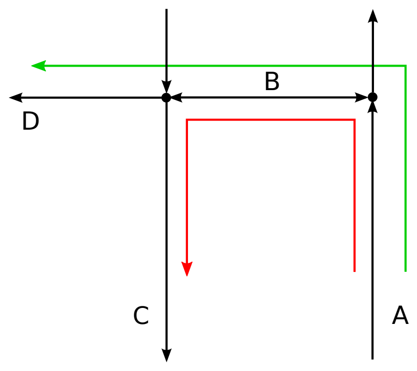


Figure 3.1: **U turn restriction**, it is legal to turn from A through B to D, but illegal to go from A through B to C

To make turn restrictions affect the graph used for route searching the nodes in this graph were split into several nodes. For each edge incident to a node a separate node was created. Usually all these nodes are connected but when there are turn restrictions some of these connections are removed.

This solves the problem with simple turn restrictions but turn restrictions where the `via` member is not a single node need to be approached differently.

One approach, proposed in [7] is to collapse the whole set of `via` elements to a single node and use the approach used for simple turn restrictions (see figure 3.2). Since this approach discards some elements the costs of the edges connecting the nodes corresponding to single node need to be adjusted to fix the modified costs.

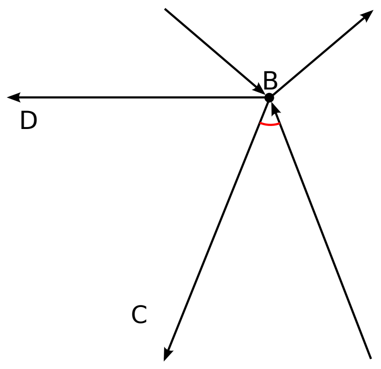


Figure 3.2: **Solution of the U turn restriction using collapsing**. The B edge is collapsed to single node. The turn marked red is forbidden.

Other approach is to create a copy of the `via` members, connect them to the original graph in suitable places and remove connection from the `from` way to original `via` members (see figure 3.3).

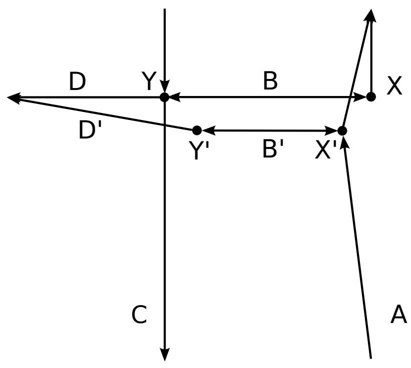


Figure 3.3: **Solution of the U turn restriction using duplication**. The B edge is duplicated with both X and Y nodes. All outgoing edges from X and Y are duplicated except for the forbidden turn. The A edge is redirected to the copy of the X node.

3.4 Reduction of number of nodes on a way

The graph for route finding can be rather large for bigger regions and so there is a need to somehow reduce its size. There are some nodes in the graph which do not have any other reason than to define the trajectory of a way. These nodes can be removed from the way keeping only the nodes in which more ways meet.

This is what program `wayreduction` is for. It selects ways from the database, determines which nodes on a way belong only to this one way and removes them, keeping only endpoints of the way and intersections. Its output is a copy of the table mapping nodes to ways with the unimportant nodes removed.

However removing these nodes changes the lengths of ways. This means that `roaddbcreate` program needs to be altered to cope with these changes in underlying data. For this reason it was changed to take two copies of the mapping of nodes to ways. One is the original mapping with all nodes and the other one is the filtered one. It then computes the cost using the original data but outputs a graph representing the filtered data.

This ensures that the shortest path in generated graph really corresponds to a real shortest route.

The problem with this approach is that having graph not containing all nodes in a map does not allow for queries with starting or destination point being one of the nodes filtered out. This can be solved for example by determining which nodes in the reduced graph are adjacent to the starting and ending point of the search and using these nodes as the starting and ending points. After this the distance between the real starting and ending point and the ones used for route planning needs to be added to the result.

The ways with reduced number of nodes are also used in drawing. When drawing a large region of the map there is no need for details on the ways. Using the filtered ways for drawing can improve its speed in these situations.

3.5 Visualization

After data had been imported into the database we needed a way to verify that the import process was successful. There was also need for visualization of the route searching functionality. For this reason the `displayer` application was created. Its purpose is to display the mapping data from the database.

This application displays ways representing some significant objects. Earlier these significant objects were specified in a configuration file and relevant ways were queried from the database as needed. This approach proved to be too slow so an alternative approach was tried.

Displaying was broken into two steps. In the first step the displayed elements are copied into a separate table and indexed for fast range queries. This is accomplished by the `edgecreate` program. It receives an XML configuration file which determines which elements to display and how to display them.

Data from this table is then displayed using the `displayer` application. It is a GTK+ application which displays elements from the database using cairo. It is also able to highlight routes and display information about elements.

For each zoom level different elements are displayed. This is required because for farther zoom levels all the information is too much to be displayed with reasonable speed.

Chapter 4

Implementation

In this chapter, implementation of different parts of the project will be covered. First the lower level libraries will be covered continuing with higher level programs and libraries.

4.1 Database abstraction layer

It was desirable to implement a database abstraction layer as libpq was used for database access. Because it is a C library, there is no automatic memory management and error reporting is done using error codes. All this can be solved using C++ features. Libpq's pointers were wrapped into objects with destructors that free the memory and check returned status codes and throw an exception when an error occurred. This wrapper is in the psql namespace.

4.1.1 `Database` class

Libpq's `PGconn*` represents a connection to the database engine. This is created by calling one of the functions `PQconnectdb` or `PQconnectStart`. Each of these takes a string describing the connection as a parameter. The difference between them is that the former initializes a synchronous connection and the latter starts connecting asynchronously. After finishing working with the connection it needs to be closed using the `PQfinish` function. This function also clears all the memory used by the `PGconn` structure [15].

This functionality is wrapped by the `Database` class. It's constructor takes the same string as the `PQconnect` function. It calls this function which initializes a `PGconn` pointer. It also allows for asynchronous connections when the constructor only starts initializing the connection and this process is finished when the instance is first used. The destructor calls the `PQfinish` function to free the resources used by the connection. This means there is no need to explicitly close the connection because when the `Database` object is destroyed the connection is automatically closed. To ease the implementation instances of the class are not copyable. However using the C++11 move semantics it can be moved, transferring ownership of the `PGconn` object.

The `Database` class is also responsible for transactions. The `begin_transaction` method starts a transaction on this connection, `commit_transaction` method commits a transaction and the method `rollback_transaction` rolls back current transaction. Savepoints in transactions are also supported using the `savepoint` and `rollback_to_savepoint` methods. These are implemented using normal statements which are sent to the database like `BEGIN TRANSACTION` etc.

Other responsibility of this class is to perform some common actions like changing the current schema or running the `ANALYSE` statement.

4.1.2 `Statement` class template

To execute a SQL statement libpq provides the functions `PQexec` and `PQexecParams`. The difference between them is that the latter allows for passing of parameters separately from SQL command text. These functions take the SQL command to execute, `PGconn` pointer representing a connection on which to execute the command and optionally the separate parameter values. They return a pointer to `PGresult` structure. Libpq then provides functions which extract information from this pointer. Such functions are for example `PQresultStatus` which returns the status of the command execution or `PQgetvalue` which returns a value of a column returned from the database [16].

Command execution functionality is wrapped by the `Statement` class template. This template is parametrized by three types, the `BindTypes`, `RetTypes` and `CopyTypes`. `BindTypes` represents the types of parameters for a command and `RetTypes` represents the types of columns returned from the database. The meaning of `CopyTypes` will be explained in following paragraphs. Each of these is a class template using new C++11 variadic templates to allow for arbitrary number of type parameters. This means that when creating a `Statement`, the user specifies the types of parameters and returned columns. This allows for a type safe access to statement parameters and returned rows.

What this means is that the types of the arguments to the `execute` method correspond to the declared types of parameters for the SQL command. Also the types of rows which can be extracted from the result set correspond to declared types of returned rows (see listing 4.1 for example).

The rows returned from the database are accessible by the `get_row` method of the `Statement`. This method takes the row number as a parameter. To find out how many rows were received from the database the `row_count` method is used.

This behaviour is sufficient for most of our use cases however there are situations where using PostgreSQL's `COPY FROM STDIN` command is very useful because it is faster than a series of `INSERT` statements. This is what the `CopyTypes` type parameter is for. It represents the types of columns to be copied into the database. When there are types associated with the `CopyTypes` parameter a call to execute leaves the `Statement` object and corresponding `Database` object in a copying state. After that calling the `copy_data` method of the `Statement` object allows for sending data to the database. Again the types of parameters of this method are the same as the types specified for `CopyTypes` for this `Statement`. Note that `CopyTypes` defaults to no types which means that this `Statement` does not represent a `COPY FROM STDIN` command. After copying data into database is done a call to the `end_copy` method returns the underlying connection from copying to normal state. Other way to return the connection to normal state is by destroying the `Statement` object which will call `end_copy` in its destructor.

In listing 4.1 the basic usage of `Database` and `Statement` is demonstrated.

Listing 4.1: Database abstraction layer basic usage

```
//Create a database connection
Database d(""); //the empty string is the connection string

//Create a statement for creating a table
Statement< BindTypes<>, RetTypes<> > st1(
    "CREATE TABLE tbl (a int, b int)", d);
//Execute this statement
st1.execute();

//Create a statement to copy some data into the table
Statement< BindTypes<>, RetTypes<>, CopyTypes<int, int> > st2(
    "COPY tbl(a,b) FROM STDIN", d);
//Execute it and start the coppying"
st2.execute();
//Copy some data to the table
st2.copy_data(1, 2);
st2.copy_data(2, 3);
st2.copy_data(3, 4);
st2.copy_data(2, 5);
//The copying process needs to be explicitly finished to allow
//for other usage of the connection.
st2.end_copy();

//Create a statement which selects some data from the table
Statement< BindTypes<int, int>, RetTypes<int, int> > st3(
    "SELECT a, b FROM tbl WHERE a>=$1 AND b<=$2", d);
//Execute the statement with parameters
st3.execute(2,4);
//Print the results
for(int i=0; i<st3.row_count(); ++i)
{
    //Retrieve a row from the statement object
    auto row = st3.get_row(i); //the type is std::tuple<int, int>
    std::cout<<std::get<0>(row)<<" "<<std::get<1>(row)<<std::endl;
}
```

Note that if wrong parameter types were specified on lines 16 or 28 a compile time error would occur. Also note that the return type of the `get_row` method is defined by the type parameters of the corresponding `Statement` template class as can be seen on line 33.

The `Statement` class template also has support for server side prepared statements using the `PQprepare` function of libpq. When constructing a statement the user can optionally provide a name and specify that the command should be a server side prepared statement. This way the constructor calls the `PQprepare` function and when the `execute` method is invoked it calls the `PQexecPrepared` function which executes a server side prepared statement. [16]

4.1.3 `Cursor` class template

When retrieving large amounts of data from the database there is a risk of running out of memory when the data set returned is too large. Cursors provide a mechanism to avoid this by returning the data in smaller chunks. Cursors could be facilitated using statements since they are handled using normal SQL commands. However since they are used in multiple places in this project they got their own abstraction in the form of `Cursor` class template.

Instances are constructed from existing instances of the `Statement` class. During construction the user also specifies the name of the cursor and the default number of rows to fetch. After constructing a `Cursor` object the user needs to call the `open` method. This results in a `DECLARE CURSOR` command being sent to the server. From this point the user can invoke the `fetch` method which optionally takes the number of rows to be received. If this number is not specified the default is used. After finished working with the cursor the `close` can be invoked to deallocate the server side cursor. Again the destructor calls this method so that all cursors are properly deallocated.

The strong point of cursors is that they can be encapsulated in iterators which allows for creation of lazy collections that can be used with the Boost Range library. For this reason the `CursorIterator` class template was created. It allows for iterating over the whole result set of an SQL command even if this result set would not fit into memory. Combined with Boost Range's `make_iterator_range` this allows for creation of a single pass range [1] which can be used to ease processing of large amounts of data.

4.1.4 SQL command collection

Since embedding SQL commands in C++ code is not very elegant a different approach was used. The SQL commands are stored in separate files and a perl script then converts these files to C++ source code to be used in the project.

For each SQL command a function taking a reference to a `Database` and returning a `Statement` is generated. In the file with SQL command, the type of resulting `Statement` is specified using a

[1]see the Boost Range concepts in [18]

SQL comment. To test that the assigned type corresponds to the provided type, command tests may also be specified in comments. Name can also be specified. When it isn't, the name of the file is used. This name prepended with `get_` is used for the generated function.

In listing 4.2 is an example of how such file might look like.

Listing 4.2: Example of a SQL file from which statements are generated

```
 1 --name create_test_table
 2 --type psql::BindTypes<>, psql::RetTypes<>
 3 --test-param
 4
 5 CREATE TABLE TestTable (A int primary key, B text, C bigint)
 6
 7 --name insert_test_table
 8 --type psql::BindTypes<int, std::string, int64_t>, psql::RetTypes<>
 9 --test-depend create_test_table
10 --test-param 10, "gda", 34422354323
11 INSERT INTO TestTable (A, B, C) VALUES ($1, $2, $3)
12
13 --name test_select_gt
14 --type psql::BindTypes<int>, psql::RetTypes<int, std::string, int64_t>
15 --test-depend create_test_table
16 --test-depend insert_test_table 10, "asdf", 20000000000
17 --test-param 9
18 --test-result 0
19
20 SELECT * FROM TestTable
21     WHERE A > $1
```

In this file three statements are defined. This creates three functions:

- `get_create_test_table`
- `get_insert_test_table`
- `get_test_select_gt`

For each of these functions a test will also be defined. For the first statement this test will consist only of a call of the `execute` method. For the second statement, since a dependency was defined on line 9, there will first be called the `get_create_test_table` function and the returned statement will be executed. On line 10 is shown how to specify parameters to the invocation of `execute` method. On line 16 parameters of a call to `execute` of a dependency are specified. Finally the comment on line 18 specifies that during the test the first returned row should be retrieved (the zero specifies index of the retrieved row). This throws an exception if the result types specified are wrong.

The process of generating C++ code from SQL files is incorporated into the build process and the resulting functions are defined in the `sqllib` namespace. This means that when the user needs to use a new SQL command in the program he writes the SQL command into a separate file with some additional comments and the build process creates a function which returns a `Statement` that can then be used in a type safe manner.

4.2 Database

The basic database schema is created by the `OsmDatabase` class in the `osmdb` namespace. This class has methods that create the tables, indexes and keys of the database. The used database schema is heavily inspired by the format in which OpenStreetMap presents it's data. The schema diagram can be seen in figure 4.1.

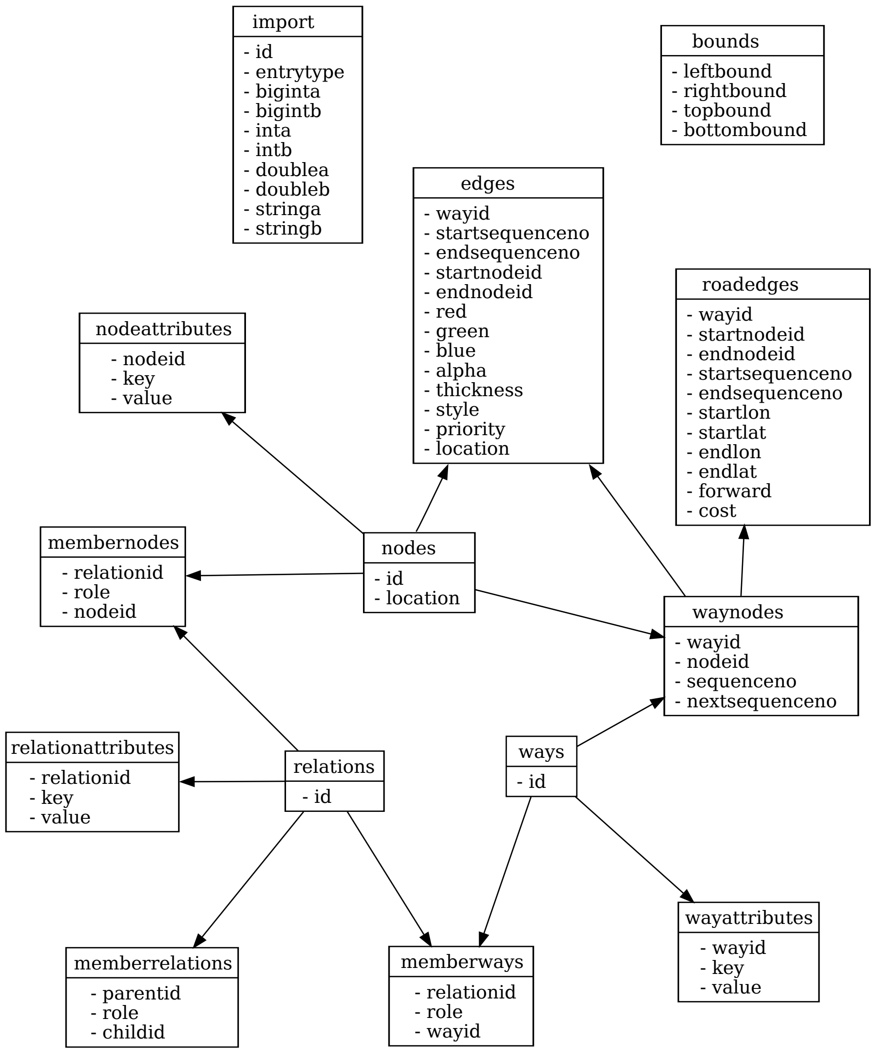


Figure 4.1: Used database schema

The schema consists of tables for nodes, ways and relations. Each of these has attributes which are stored in separate tables `NodeAttributes`, `WayAttributes` and `RelationAttributes`. Each node has an ID and a Location which has a PostGIS type for geographic position. Ways and relations both have only ID. The table `WayNodes` defines of which nodes a way is composed. Each Way has several rows in this table with NodeID column describing which node is member of this way and SequenceNo describing the position of the node on the way. NextSequenceNo is the position of the next node on this way. This is used when unimportant nodes are removed from ways to keep track of real positions of nodes on original way. To keep track of Relation members, three tables exist: NodeMembers, WayMembers and RelationMembers. Each of these keeps track which elements are members of which relations. Roles are also stored in these tables.

In addition to these tables, there are two utility tables, the Bounds table and the Import table. The Bounds table stores the bounding rectangle of the data currently in database to ease finding the center of the displayed map. The Import table is the table into which the imported data is copied before being sanitized and moved to correct tables[2].

There are also other tables not present in the basic schema, their purpose is optimization and storage of precomputed information. One such table is the Edges table which stores the lines which should be drawn, other RoadEdges which stores edges of a graph used for routing. These tables are created by other classes in the osmdb namespace.

4.3 OpenStreetMap element classes

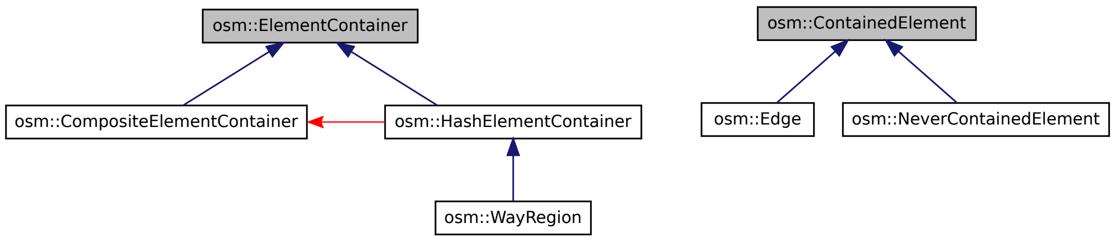
The namespace osm contains classes representing elements of OpenStreetMap, that is nodes, ways and relations. Then it contains class ContainedElement which represents an element which can be contained in an ElementContainer. This is used in graphical interface to allow for highlighting of elements drawn on the map.

The base class of OpenStreetMap elements is Element. It provides abstract functions for retrieving the type of the element and it's ID. The fill when implemented fills the properties of the element from the database using an instance of the osmdb::PropertiesSelection class. Other functions declared in this class are used for describing the element. The get_description should return a property tree describing the element. The get_highlighed when implemented returns an ElementContainer containing ContainedElements corresponding to this Element.

The ContainedElement has only two subclasses. One of them is the Edge class which represents a segment of a way and is used in graphical interface. The other one is NeverContainedElement which is used as a placeholder when an element it represents is not to be contained in any container.

The ElementContainer and ContainedElement classes use the visitor pattern to handle whether given element is in the container. That is the class ContainedElement declares an abstract method is_intersected accepting an argument of type ElementContainer and it declares abstract overloaded method intersects for each subclass of ContainedElement. When the is_intersected method is implemented it should invoke the appropriate overload of intersects method on its argument.

[2]this process will be described in section 4.4


osm::ElementContainer
osm::CompositeElementContainer
osm::HashElementContainer
osm::WayRegion
osm::ContainedElement
osm::Edge
osm::NeverContainedElement

Listing 4.3: `ElementContainer` methods

```
virtual bool intersects(Edge const& e);
virtual bool intersects(NeverContainedElement const& e);
```

Figure 4.2: Subclasses of `ElementContainer` and `ContainedElement`, red arrow represents membership, blue arrow represents inheritance, note that `CompositeElementContainer` may contain many instances of `HashElementContainer`.

Another requirement was to be able to combine several instances of `ElementContainer` to one new instance. For this reason the `CompositeElementContainer` class was created. It has a method which allows `ElementContainer` objects to be added to it. When it is queried whether a `ContainedElement` is a member, it checks whether any of its child `ElementContainer` objects contain it. However checking for this can take a long time when many `ElementContainer` children are present.

For this reason the `HashElementContainer` was created. It provides an additional method named `get_hash` which returns a number which can be used by `CompositeElementContainer` to narrow the set of its child element containers.

The only implementation of `HashElementContainer` is the class `WayRegion`. This class represents a part of a way. Hash returned by the `get_hash` method is the ID of the way. This means that when a `CompositeElementContainer` determines whether any of the `HashElementContainers` in it intersect an `Edge` it only has to check those that return the hash equal to the ID of the way the `Edge` is from.

Figure 4.2 summarizes the relation between `ContainedElement`, its subclasses, `ElementContainer` and its subclasses.

4.4 Import

The importer program uses the classes in the `osmxml` namespace to parse input XML document and inserts the data into the database using the `osmdb::ElementCopy` class. Then it removes artifacts from this data and moves it to appropriate tables using the class `osmdb::ImportTableProcessor`.

It also uses the `osmdb::OsmDatabase` class to create the database schema.

The `osmxml` namespace is composed mostly of classes generated using CodeSynthesis XSD from a simplified XML schema for OSM XML. This simplified schema is sufficient since the complete schema in [1] has support for information not used in basic mapping data exports. These use the SAX[3] approach when each encountered element dispatches an event which can be listened to by the user. This avoids loading the whole document into memory which cannot be done for very large documents.

The generated classes are wrapped in `XmlParser` class. After constructing an instance, the user can assign functions to handle all three element types. Then after one of the methods `parse_file`, `parse_memory` or `parse_stream` is invoked the assigned functions get called for each element present in the document.

The importer program connects these functions to function calls in class `osmdb::ElementCopy`. It contains methods for importing ways nodes and relations. Each of these takes the appropriate element as argument and uses the `COPY` command to put information about the element to the `Import` table in the database.

Imported data may not be completely consistent so they are processed to remove invalid artifacts. The `ImportTableProcessor` class in `osmdb` namespace does this processing and moves the data into appropriate tables in the database.

`ImportTableProcessor` performs various actions listed in enum `ImportTableAction`. These actions are:

1. Indexes that speed up additional processing of the `Import` table are created and the `ANALYZE` command is run to allow the planner to use them.
2. Elements already in the database which are also in the `Import` table are deleted from the database to be replaced by the new elements.
3. Element attributes which had their element removed by previous actions are deleted.
4. Relation members without parent relation are deleted.
5. Duplicit elements are deleted from the `Import` table.
6. Incomplete ways and relations are deleted from the `Import` table, that is relations with missing members and ways with missing nodes.
7. Attributes in the import table which had their corresponding elements removed by previous actions are deleted.
8. Attributes defined multiple times in the `Import` table are removed.
9. Elements and attributes are moved to appropriate tables in the database.
10. The `Import` table is cleared.

[3]Simple API for XML

To speed up the import the importer also allows for removing the foreign keys and indexes used in the database before starting the import process and recreating them after the import process is finished.

4.5 Reduction of nodes defining ways

The `wayreduction` program is supposed to be run after the import of data is finished. It's purpose is to reduce the number of nodes which define a way and thus reduce the size of the graph used for routing created from the map. This is accomplished by keeping only the important nodes on ways and removing the others.

The result of running this program on a database containing the map is creation of new schema where only the `WayNodes` table is defined. This table is then filled with only the important nodes.

When the search path for a connection is set[4] to this new schema and the original schema the client can access this combined schema in the same way it would access the original schema with the difference that the defined ways contain only the important nodes.

Which nodes are important depends on the arguments given to the `wayreduction` program. First it takes a list of attributes that need to be present in a way for it to be processed. There is also a distance limit. The important nodes are then the nodes which are intersections of multiple ways, the endpoints of a way or nodes that are farther from the last important node than the distance limit.

Five classes are used in this program:

- `osmdb::WayLister`,
- `wayred::WayNodeFilter`,
- `osmdb::OsmDatabase`,
- `osmdb::ElementCopy` and
- `osmdb::ImportTableProcessor`.

The latter three are used in the same way as in importer to create the `WayNodes` table and to insert the processed data.

The `WayLister` class is responsible for retrieving all the ways with given attributes and for each node on these ways the list of all ways they are a member of. This is accomplished by several database queries combined using functions on their result ranges. This allows for sequential processing of data which means there is no concern for running out of available memory.

`WayNodeFilter` processes each way and creates a copy with unimportant nodes removed. For each node contained in the input way it checks whether there are any other ways with required attributes

[4] via the `SET SEARCH_PATH` command, as described in [17]

that pass through it or it is far enough from last important node. From these nodes it constructs a new way which is then inserted into the database.

4.6 Road graph creation

The graph creation is made up of two steps. First, the `roaddbcreate` program computes the costs of edges on this graph and stores the graph in the database. Second, the classes `osmdb::RoadLister` and `roads::RoadNetwork` retrieve the data from the database and create the graph in memory.

The `roaddbcreate` program creates a table in the database named `RoadEdges`. This table contains each edge of the resulting graph information about its endpoints and its cost. The endpoints of an edge are nodes of the map and the stored information includes the ID of the node, its position within the way to which the edge belongs and its geographical location. All these information is stored directly in the `RoadEdges` table to avoid doing joins when retrieving this information.

The `osmdb::RoadLister` class retrieves this information and the `roads::RoadNetwork` creates the graph. The graph is created using instances of `roads::RoadNetworkNode`. These represent nodes of the graph. For each edge of the graph two instances are created, one for each endpoint. The ending endpoint is connected to starting points of all edges leading from the corresponding map node. This allows for modelling simple turn restrictions by simply removing the connection. The `roads::RoadNetworkNode` class contains a list of all neighbours and the costs required to traverse from this node to the neighbour.

4.7 Path finding

Path finding is done using the `pathfind::PathFinder` class. This class takes a road network and an instance of `pathfind::PathFindingAlgorithm` specialized to `roads::RoadNetworkNode const*` for parameters. The `roads::PathFindingAlgorithm` interface is a generic interface for path finding algorithms. It has a type parameter determining the type of the nodes of the graph on which it operates. It declares only one method, `find_path` which takes two collections of nodes as parameters. The first collection is the list of starting nodes and the second parameter lists the destination nodes. The method returns a list of nodes which form the shortest path from one of the starting points to one of the destination points.

The `find_way` method of `pathfind::PathFinder` class first finds which road graph nodes correspond to the starting and ending map node. Then it runs the `find_path` method of the algorithm class to obtain the list of road graph nodes which form the shortest path. After that the the map edges corresponding to the resulting path are found using the methods of `roads::RoadNetwork` class. It then stores these in an instance of `pathfind::Route` class which represents the shortest route found.

The `pathfind::AStarPathFinding` is the only class implementing the path finding algorithm interface. It is an implementation of the A* algorithm which takes the heuristic function as a parameter. This means that this class can also be used as the Dijkstra's algorithm when given heuristic which always returns 0.

This class is implemented using the `pathfind::AStar` class. It represents the state of the A* algorithm. It has the method `step` which does a step in the algorithm. It also has methods for inspecting different aspects of the state e.g. whether a shortest path to a node is already known. This allows for simple implementation of the `AStarPathFinding` class.

4.7.1 Area finding

Other requirement was finding the edges reachable from a starting point within given cost. This is done using the `pathfind::AreaFinder` class which is very similar to the `PathFinder` class. However instead of `PathFindingAlgorithm` it uses `pathfind::AreaAlgorithm` which is a very similar interface to `PathFindingAlgorithm` with the difference being that the area version takes a list of starting nodes and a maximum cost.

There is also a `pathfind::AStarAreaFinding` which implements the `AreaAlgorithm` interface. It uses the `AStar` class to determine which nodes are reachable from the starting nodes within the desired cost.

After determining which nodes on the road graph are reachable the `AreaFinder` class find the corresponding map edges from the `RoadNetwork` class. These are then contained in an object of class `Route` which can also be used for areas.

4.8 Display

To draw the map the `displayer` application is used. Its main class is `display::MapDrawingArea`. This class handles drawing of elements on the screen. It uses subclasses of `display::DisplayProvider` abstract base class to find out which elements to draw. It also uses classes in the `projection` namespace to handle the projection of geographic data on the screen.

Currently the projection used is the orthographic projection. This projection maps a point to the closest point on a plane tangent to the earth. This plane is determined by the currently displayed region.

This interface provides functions to retrieve elements within a rectangular region of the map. These elements are implementations of the `display::DisplayElement` interface. These are then projected on a plane and drawn. The `DisplayProvider` also has support for selection. It provides a function to retrieve elements near a point. These elements are then used to provide information about what was selected.

They are also used in `display::EdgeHighlighter` to highlight selected parts of the map. It uses the mechanisms of `osm::ContainedElement` and `osm::ElementContainer` to decide which elements need to be highlighted. This class is also used to highlight the computed shortest route or the edges reachable within maximum cost.

The window of `displayer` application consists of the drawing area, three input boxes, two buttons and a text pane. The input boxes are used to retrieve the starting and destination map node id for shortest route queries and the maximum cost used for reachability queries. The buttons are used to perform the queries. The text pane is used to provide textual information about results of queries and selected elements.

The most important `DisplayProvider` is the `osmdb::DisplayDB` which retrieves the display information about map edges from the database prepared by the `edgecreate` program.

This program creates an additional table named `Edges` which contains positions and properties of displayed edges. It fills this table based on a configuration file which defines display properties for edges which belong to ways with given combination of attributes. It accomplishes this by using the `osmdb::EdgeCreator` class, which for each combination of attributes retrieves matching ways and creates appropriate rows in the `Edges` table.

The contents of the `Edges` table need to depend on the zoom level for which they are used. For this reason, the `edgecreate` program takes the name of output schema as an argument. This allows for different levels of detail for different zoom levels. In combination with output from the `wayreduction` program this leads to rather fine grained control over what will be displayed for each zoom level.

The `edgecreate` program can also fill the `Edges` in such a way that using the `displayer` application with it creates a visual representation of the road graph used for route searching. This visual representation consists of edges with different colors. The shade of the color of an edge represents the relative cost of the edge with respect to its displayed length. This means that edges which have a high cost but are displayed short[5] have more saturated red color.

Because the contents of the `Edges` table are different for different zoom levels, the `displayer` applications uses a configuration file to determine which schema to use for each zoom level.

The last thing the displayer does is display the convex hull of the result area for the reachability queries. This is done using the `display::AreaBoundaryDisplayProvider` class and the function `geo::get_convex_hull`. The `get_convex_hull` function finds the convex hull using the Graham scan algorithm and `AreaBoundaryDisplayProvider` creates elements suitable for being drawn by `MapDrawingArea`.

[5]this may occur because of node filtering

Chapter 5

Results

This chapter summarizes results of this project.

5.1 Import

When running the `importer` program with a 2.7GB large Planet.osm export of Slovakia as input, it produces the following output.

Listing 5.1: Running the importer program

```
> ./bin/importer -i slovakia.osm -s svk -I -r -q -e -a
Dropping foreign keys and indexes
Starting import
Starting copy
Processed approximately 100000 elements
Processed approximately 200000 elements
...
Processed approximately 13500000 elements
Processed approximately 13600000 elements
Done copying
Created import primary key
Created import indexes
Done analyzing
Deleted 0 nodes which should be updated
Deleted 0 ways which should be updated
Deleted 0 relations which should be updated
Deleted 0 nodes which should be deleted
Deleted 0 ways which should be deleted
Deleted 0 relations which should be deleted
Deleted 0 orphan elements
Deleted 0 duplicit nodes in import
Deleted 0 duplicit ways in import
Deleted 0 duplicit relations in import
```

```
Imported 11885917 nodes
Imported 1783219 ways
Deleted 1983 incomplete ways in import
Deleted 680 incomplete relations in import
Deleted 192507 orphan elements in import
Deleted 0 duplicit attributes in import
Deleted 0 duplicit waynodes in import
Deleted 126 duplicit member elements in import
Imported 8461 relations
Imported 957900 node attributes
Imported 4920548 way attributes
Imported 29756 relation attributes
Imported 15385355 way nodes
Imported 3045 member nodes
Imported 59076 member ways
Imported 366 member relations
Done cleaning up
Updating metadata
Recreating indexes and keys
Running analyze
Success
```

The total time elapsed was approximately one and a half hour. This is a large improvement over the first implementation which did not drop keys and indexes before importing and used plain `INSERT` statements instead of `COPY`. Running that version of `importer` with the same input took three times as long.

5.2 Filtering out not needed nodes from ways

Running the `wayreduction` program several times to achieve different levels of detail produces the following output.

Listing 5.2: Running the wayreduction program several times

```
> ./bin/wayreduction -b svk -s svk_red6 -I share/map6.txt -l 0.7
Processing ways
Created import primary key
Created import indexes
Done analyzing
Imported 757858 way nodes
Done cleaning up
Success
> ./bin/wayreduction -b svk -s svk_red5 -l 1.5 -I share/map5.txt
...
Success
> ./bin/wayreduction -b svk -s svk_red -I share/way.txt
```

```
Processing ways
Created import primary key
Created import indexes
Done analyzing
Imported 341268 way nodes
Done cleaning up
Success
```

This action took a total of 72 minutes.

5.3 Creating the tables for display

After that the `edgecreate` program is run several times to create the `Edges` table for different zoom levels and for road graph.

Listing 5.3: Running the edgecreate program for different zoom levels

```
> ./bin/edgecreate -x share/toshow/to_show_edges/1.xml -i svk_red1,svk -o zoom1 -c
Creating tables
Inserting data
Creating keys and indexes
Done
> ./bin/edgecreate -x share/toshow/to_show_edges/2.xml -i svk_red2,svk -o zoom2 -c
...
Done
> ./bin/edgecreate -r svk_roads,svk -o zoom_roads -c
Creating tables
Inserting data
Creating keys and indexes
Done
```

The duration of this was 20 minutes.

5.4 Road graph creation

Finally the road graph is created using the `roaddbcreate` program.

Listing 5.4: Creating the road graph

```
> ./bin/roaddbcreate -f svk -r svk_red -o svk_roads -n -I share/way.txt
Creating table
Copying data
```

This was finished after 5 minutes.

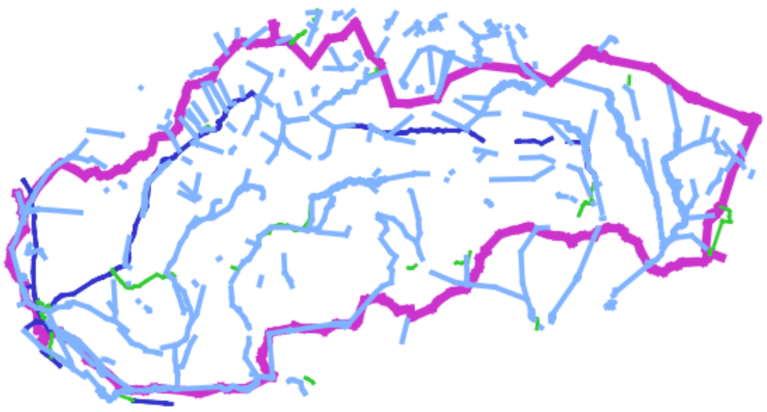


Figure 5.1: Drawing of Slovakia.

After this the database is ready to be viewed by `displayer` and queried for shortest routes or reachable areas.

5.5 Map drawing

In this section some drawings of the map are shown.

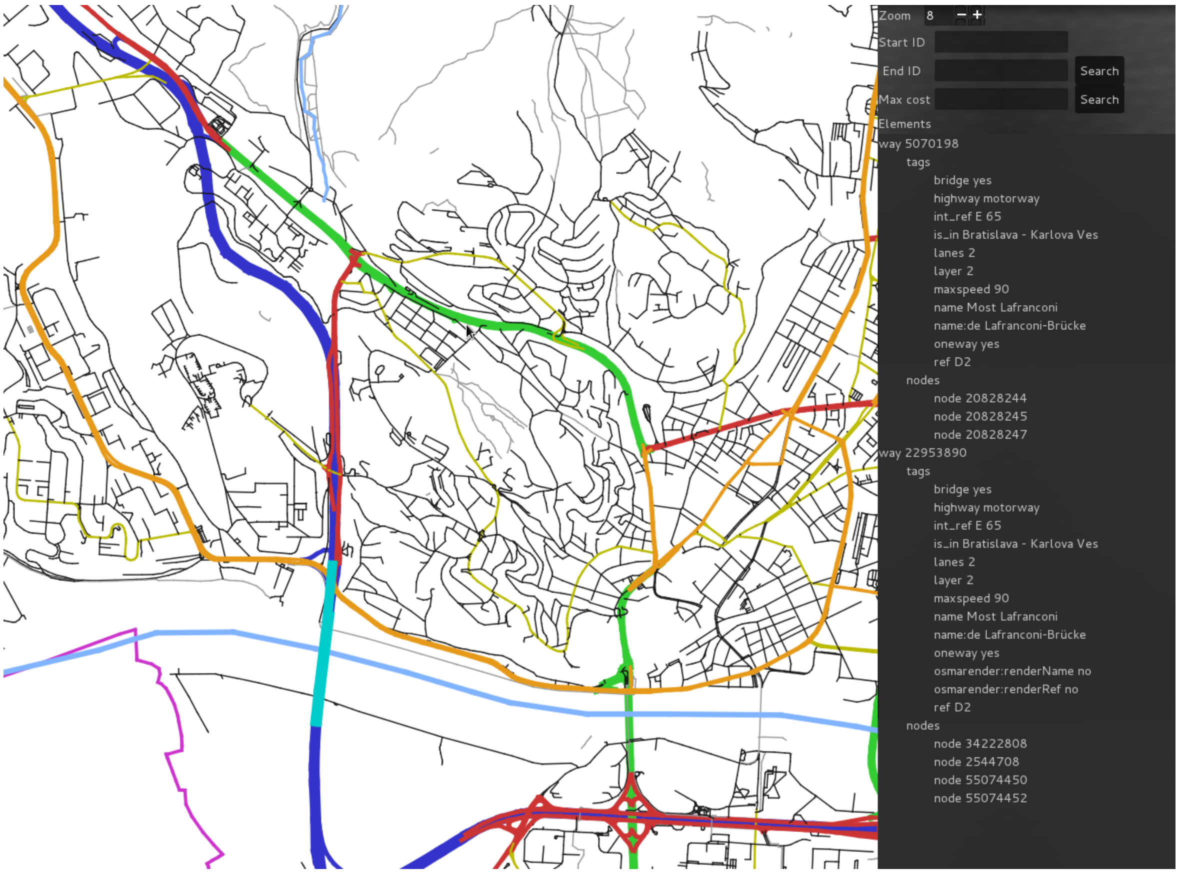
Figure 5.1 shows the drawing of whole Slovakia at the farthest zoom level. Only the boundary, rivers and the most important roads are drawn.

Figure 5.2 shows a part of Bratislava at closer zoom. It also demonstrates the selection mechanism, the bridge in the lower left is selected and a description of it is in the text pane.

Figure 5.3 demonstrates how road graph visualization works. The left half visualizes part of a road graph. The right half displays the same region using the normal map display. It can be seen that the edges which were shortened on the road graph have a red shade. It should also be noted that the arrows on the normal map display represent the order in which nodes belong to ways. These are displayed at the closest zoom levels. The arrows in the road graph representation visualize one way roads.

5.6 Route searching

The results of searching for a route from the Faculty of Mathematics, Physics and Informatics of Comenius University in Bratislava to some place in Trnava 5.4. Since only the roads already displayed



```
way 5070198
    tags
        bridge yes
        highway motorway
        int_ref E 65
        is_in Bratislava - Karlova Ves
        lanes 2
        layer 2
        maxspeed 90
        name Most Lafranconi
        name:de Lafranconi-Brücke
        oneway yes
        ref D2
    nodes
        node 20828244
        node 20828245
        node 20828247
way 22953890
    tags
        bridge yes
        highway motorway
        int_ref E 65
        is_in Bratislava - Karlova Ves
        lanes 2
        layer 2
        maxspeed 90
        name Most Lafranconi
        name:de Lafranconi-Brücke
        oneway yes
        osmarender:renderName no
        osmarender:renderRef no
        ref D2
    nodes
        node 34222808
        node 2544708
        node 55074450
        node 55074452
```

Figure 5.2: Part of Bratislava. Bridge in the lower left highlighted with cyan color is selected, and its properties are shown in the text pane.

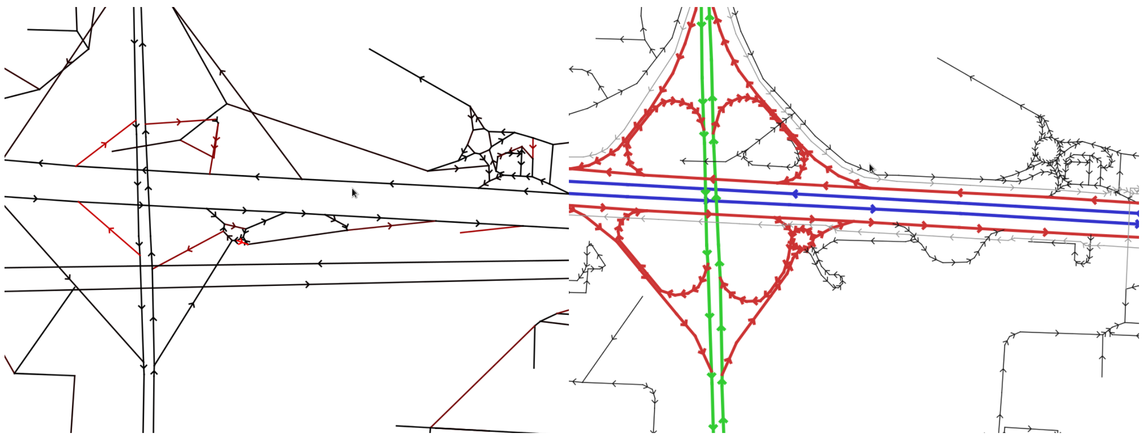
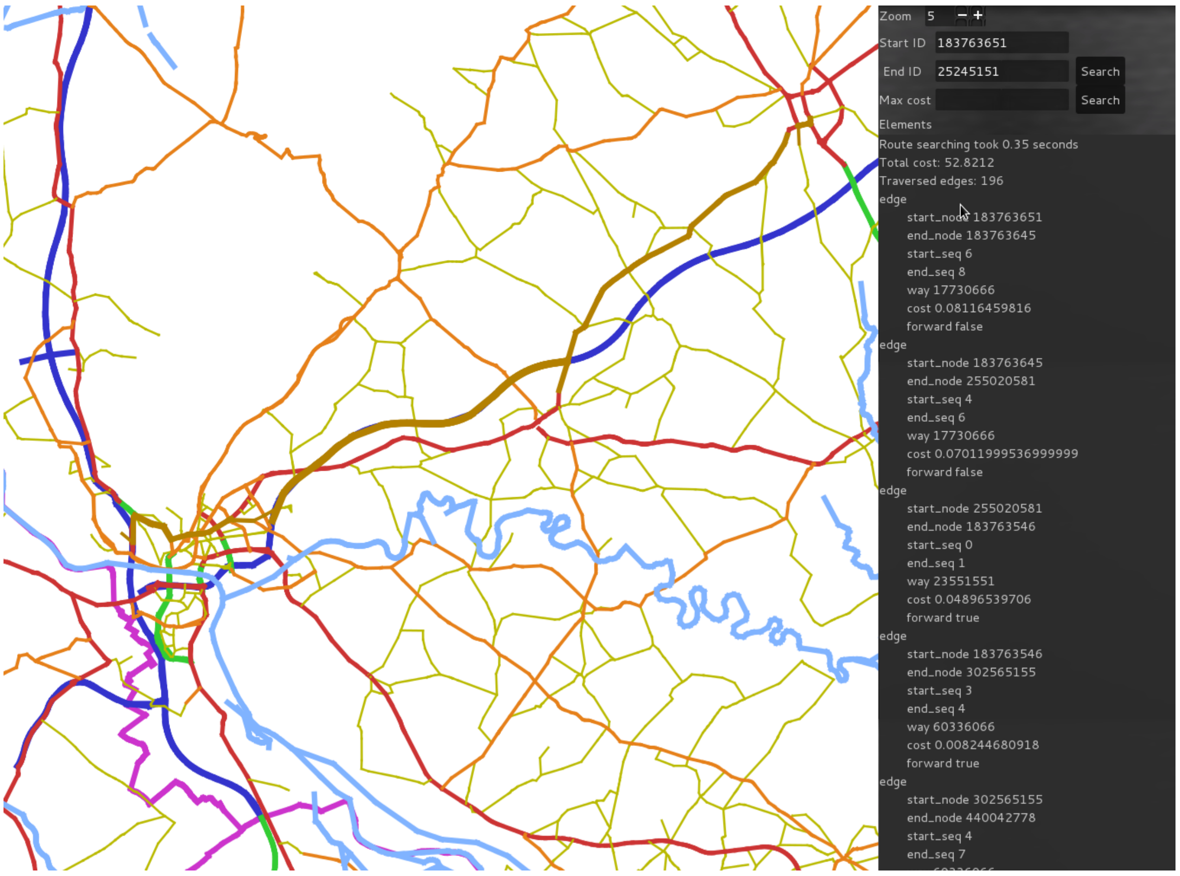


Figure 5.3: Road graph visualization in the left half, corresponding normal map in the right half.



```
Start ID  183763651
End ID  25245151
Max cost
Route searching took 0.35 seconds
Total cost: 52.8212
Traversed edges: 196
edge
    start_node 183763651
    end_node 183763645
    start_seq 6
    end_seq 8
    way 17730666
    cost 0.08116459816
    forward false
edge
    start_node 183763645
    end_node 255020581
    start_seq 4
    end_seq 6
    way 17730666
    cost 0.07011999536999999
    forward false
edge
    start_node 255020581
    end_node 183763546
    start_seq 0
    end_seq 1
    way 23551551
    cost 0.04896539706
    forward true
edge
    start_node 183763546
    end_node 302565155
    start_seq 3
    end_seq 4
    way 60336066
    cost 0.008244680918
    forward true
edge
    start_node 302565155
    end_node 440042778
    start_seq 4
    end_seq 7
```

Figure 5.4: Route from Faculty of Mathematics, Physics and Informatics of Comenius University in Bratislava to some place in Trnava is highlighted with light brown color, properties of the route are in the text pane.

at this zoom level are highlighted, there is a missing piece of the route close to the destination (the upper right).

5.6.1 Dijkstra's algorithm and A* comparison

A set of several routes was selected and both Dijkstra's algorithm and A* were tried on it. The time required for both of these algorithms was recorded. The results are in the table 5.1.

It can be seen that the Dijkstra's algorithm is slower in most cases. However for the longest query it is actually faster. The reason for this is that the resulting path spans the whole map so both A* and Dijkstra's algorithm probably had to inspect most of the nodes of the graph. And because the Dijkstra's algorithm does not need to compute the value of the heuristic it requires less time.

Route nr.	Length(km)	Number of edges	A* time (s)	Dijkstra time (s)
1	115.614	423	0.74	2.24
2	183.067	435	1.82	4.78
3	379.283	862	4.88	5.62
4	263.179	615	1.16	1.64
5	173.869	428	1.04	4.32
6	265.773	727	1.31	4.33
7	433.798	1163	5.56	5.51
8	75.602	196	0.13	0.54
9	48.134	121	0.05	0.11
10	173.766	411	0.70	1.84

Table 5.1: Results of the A* and Dijsktra's algorithm for 10 routes.

5.7 Reachability

Figure 5.5 demonstrates the result of a query for which edges are reachable from a the Faculty of Mathematics, Physics and Informatics of the Comenius University in Bratislava within 4 kilometers. It also shows the convex hull of the result.

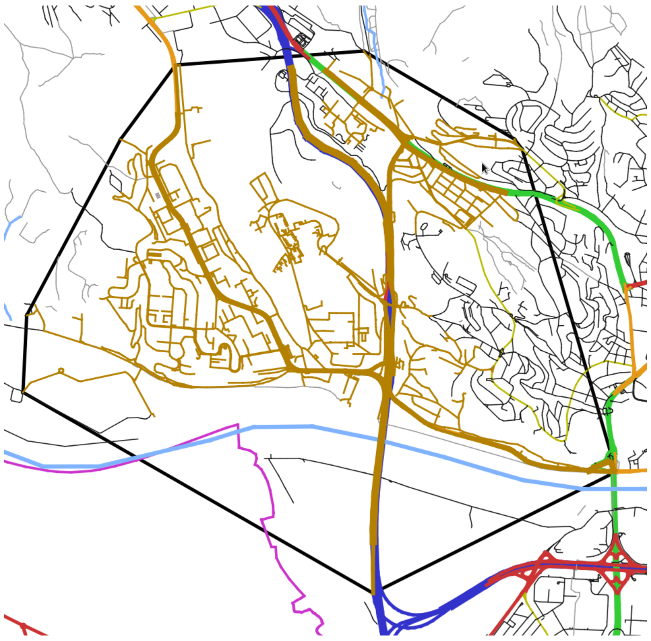


Figure 5.5: Roads within 4 kilometers from Faculty of Mathematics, Physics and Informatics of the Comenius University in Bratislava are highlighted with light brown color. Their convex hull is drawn with black color.

Conclusion

In this thesis, we presented the implementation of several programs for reading and processing of OpenStreetMap mapping data. We also presented simple algorithms for shortest route finding using the processed data.

We have seen that these simple approaches are able to find the shortest route. However the time needed is prohibitively large for longer routes. This means that advanced approaches are needed for real time route searching. But most of those introduced in the first chapter are less flexible than these simple approaches e.g they do not work well in the time dependent scenario. This means that there is need for some flexible and fast approaches.

Future work

The map data from OpenStreetMap provide much more information than this project currently uses. This information includes maximum speed, restrictions on types of vehicles which are allowed to enter roads etc. All this information should be used to provide more accurate results.

Also the result of this project is an application with rather large system requirements so reducing them should be a priority. This is required if the application was to be ported to any mobile device.

Bibliography

[1] *API v0.6/XSD*. 2012. URL: http://wiki.openstreetmap.org/wiki/API_v0.6/XSD.

[2] Holger Bast et al. *In Transit to Constant Time Shortest-Path Queries in Road Networks.*

[3] *Boost (C++ libraries)*. 2012. URL: http://en.wikipedia.org/wiki/Boost_%28C%2B%2B_libraries%29.

[4] *Cairo*. 2012. URL: http://www.cairographics.org/.

[5] Daniel Delling et al. "Customizable Route Planning". In: *Proceedings of the 10th International Symposium on Experimental Algorithms* (2011).

[6] Daniel Delling et al. "Engineering Route Planning Algorithms". In: *J. Lerner, D. Wagner, and K.A. Zweig (Eds.): Algorithmics, LNCS 5515* (2009), pp. 117–139.

[7] Ingrid C. M. Flinsenberg. "Route Planning Algorithms for Car Navigation". PhD thesis. Technische Universiteit Eindhoven, 2004.

[8] Andrew V. Goldberg and Chris Harrelson. *Computing the Shortest Path: A Search Meets Graph Theory.*

[9] *Node*. 2012. URL: http://wiki.openstreetmap.org/wiki/Node.

[10] *Open Database License*. 2012. URL: http://wiki.openstreetmap.org/wiki/Open_Database_License.

[11] *OpenStreetMap*. 2012. URL: http://www.openstreetmap.org/.

[12] *OSM tags for routing*. 2012. URL: http://wiki.openstreetmap.org/wiki/OSM_tags_for_routing.

[13] Chris Pendleton. *Bing Maps New Routing Engine*. 2012. URL: http://www.bing.com/community/site_blogs/b/maps/archive/2012/01/05/bing-maps-new-routing-engine.aspx.

[14] *Planet.osm*. 2012. URL: http://wiki.openstreetmap.org/wiki/Planet.osm.

[15] *PostgreSQL 9.1.3 Documentation, 31.1. Database Connection Control Functions*. 2012. URL: http://www.postgresql.org/docs/9.1/static/libpq-connect.html.

[16] *PostgreSQL 9.1.3 Documentation, 31.3. Command Execution Functions*. 2012. URL: http://www.postgresql.org/docs/9.1/static/libpq-exec.html.

[17] *PostgreSQL 9.1.3 Documentation, 5.7. Schemas*. 2012. URL: http://www.postgresql.org/docs/9.1/static/ddl-schemas.html.

[18] *Range Concepts.* 2012. URL: http://www.boost.org/doc/libs/1_49_0/libs/range/doc/html/range/concepts.html.

[19] *Relation:restriction.* 2012. URL: http://wiki.openstreetmap.org/wiki/Relation:restriction.

[20] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *G.S. Brodal and S. Leonardi (Eds.): ESA 2005, LNCS 3669* (2005), pp. 568–579.

[21] *Talk:OSM tags for routing.* 2011. URL: http://wiki.openstreetmap.org/wiki/Talk:OSM_tags_for_routing.