Självständigt arbete på grundnivå

Independent degree project – first cycle

Datateknik
Computer engineering

Line Graph Utility
A software module for routing

Jonas Bergman

Mittuniversitetet
MID SWEDEN UNIVERSITY

MID SWEDEN UNIVERSITY
DSV

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
Supervisor: Börje Hansson, borje.hansson@miun.se
Author: Jonas Bergman, jobe0900@student.miun.se
Degree programme: Programvaruteknik, 180 credits
Main field of study: Software technology
Semester, year: Fall, 2015

Abstract

This project was about building a line graph utility, a software module that should read map data from a PostGIS database and transform that information into a line graph (edge based graph) that the calling software could use to perform routing decisions. This outer calling application is part of a project (by an anonymized company) for flexible public transportation, that is meant to manage and direct a fleet of vehicles to where the customers actually are, instead of idling at bus stops. The software module should take different kinds of restrictions and conditions into account when building the line graph, to reflect the actual traffic situation. That can be turn restrictions, traffic signs, inclination, or conditions such as temporary hindrances, time of day. Some are static, but others vary dynamically and the state is to be found in the database.

This study has found a set of tools that aids in the transformation of OpenStreetMap data into a PostGIS database; for building the topology of the map; querying the database; and data structures for representing the graph and line graph.

The result of the project is a piece of working software that can return a line graph as a Boost graph with some restrictions taken into account, but it has not yet implemented them all, and more specifically, it does not handle conditional restrictions yet. There remains a good deal of work to implement all that complex logic.

Contents

1 Introduction

The work presented in this thesis is about flexible routing of public transportation. The result of the work is a software module that loads map data and converts it into in-memory data structures that can be used for routing decisions by exposing an *API*, (Application Programming Interface). This module is part of a bigger transportation optimization system that is meant to enable flexible public transportation solutions.

The module will be used for finding efficient routes in a dynamic traffic environment, i.e. the complete solution must take turn restriction, traffic lights and road signs into account. The outline of how to do this is by loading map data from *OpenStreetMap*[1] into a database (*PostgreSQL*[2] extended with *PostGIS*[3]). Upon a request directed to the API, the module should build (with soft real-time requirements) a data structure suitable for passing to a routing algorithm.

1.1 Background and problem motivation

The company (*anonymized*) aims at developing a solution for managing *flexible* public transportation, meaning no more buses standing idle and empty at bus stops, waiting just in case another bus fills up. The buses can be directed to where they are needed, and part of the solution is finding the best routes and give directions to the drivers where they should go. The public does not need to wait at bus stops, but can ask for pick-up via a mobile app.

There can obviously be huge benefits from such a transportation system. Less vehicles are needed, and better utilization of the vehicles, which should be good both for the environment and the finances of the operation. The public should also benefit from having access to public transportation where needed, and not from fixed locations.

Central for such a system is efficient routing of the vehicles, with almost instant updates on restrictions made available to the drivers needing directions. This project is a small piece in that puzzle.

1.2 Overall aim

This project should result in a working software module, fulfilling the requirements set by the company. There is needed some preliminary studying of graph theory, data structures, and research into what theories and solutions that already might exist, and if so, if they can be adapted and used in this project.

1.3 Scope

The scope of this project is to create the routing data structures representing the map data, not the routing algorithms, although they might affect one another, such that the choice of algorithm might affect what data structures are suitable.

1.4 Detailed problem statement

The software in this project is a module, exposing a function. When the function is called, it should load map data from a database, which has previously been loaded with OpenStreetMap-data, and build a connected graph to be used for routing decisions, and the data structure is returned to the caller so it can be used for routing. The building of the graph should happen in *soft real-time* so that it reflects all known restrictions in the database. For example if one

[1] `http://www.openstreetmap.org`
[2] `http://www.postgresql.org`
[3] `http://postgis.net`

road gets temporarily closed it should be marked as such, and that should be represented in the graph.

The requirements from the company states that the graph should be represented as a *line graph*, which is a basic technique for representing available turns at junctions. The software module shall be implemented in C++, using the *Boost Graph Library*[4] for the data structures. The software should be developed using *Behavior Driven Development* (BDD) or *Test Driven Development* (TDD) as methodologies, and otherwise adhere to the company's coding standards.

1.5 Outline

- Chapter 2 will present some background on graph theory, and research in map routing, regarding both theoretical foundations and some available implementations.
- Chapter 3 shows the methods and tools used.
- Chapter 4 is about the design and implementation of the software module.
- Chapter 5 presents the results from testing the implementation.
- Chapter 6 will include some discussion and conclusions made during this project.

1.6 Contributions

The work presented in this report is the sole work of the author.

[4] `http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/index.html`

2 Related work

One of the first applications of *graph theory* was when *Leonhard Euler* considered the *Königsberg bridge problem*: Is there a way to walk over the seven bridges of Königsberg only once?

It is trivial to see that there is a close correlation between graphs and maps, as you can see in figure 2.1, with the roads and junctions in the map being lines and dots in the graph. A line is mostly called *edge* or *arc* and a connecting dot is called *vertex*, *node* or *point*; in this report it will mostly be *vertex* and *edge*, but to differentiate, another type of graphs called *line graph* will use the names *node* and *line* to distinguish. As one delves deeper into the theoretical material, one will find that there is good to know some *graph theory* and be familiar with some definitions.

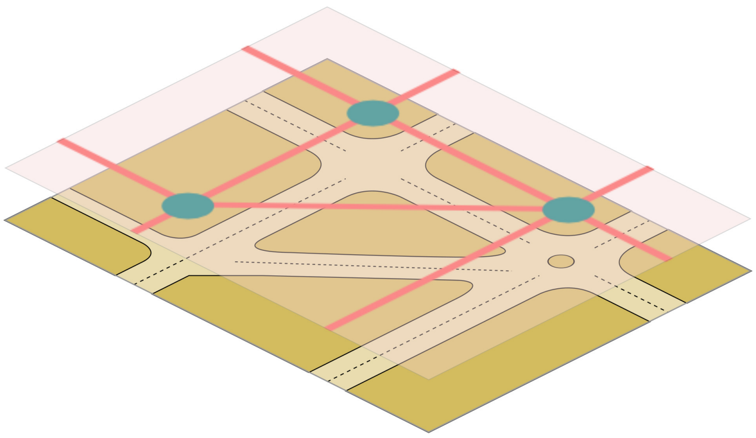
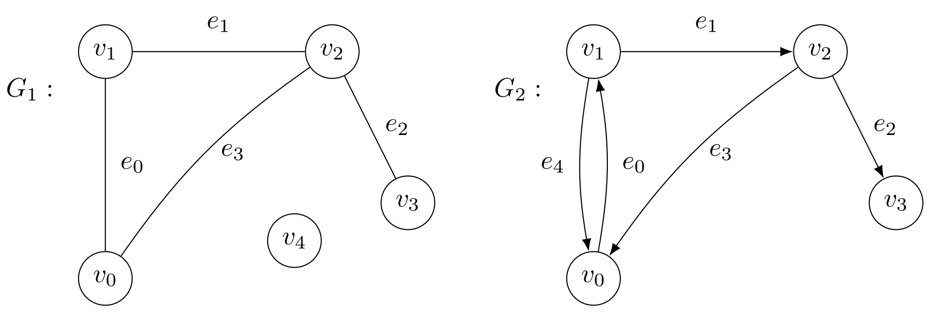


Figure 2.1: Graphs and road maps are a natural match.

2.1 Graph theory

There are good lecture notes such those from *Tampere University of Technology* [1] and *Univerity of Turku* [2] (both happen to be Finnish) and a good text book by *Reinhard Diestel* [3] are available to get into this subject. One does not need to understand all the concepts, but be familiar with some basic definitions and notations.

A *graph* is made up by *vertices* and *edges*, see figure 2.2.



(a) Undirected (b) Directed

Figure 2.2: A *graph* with *vertices* and *edges*.

So a graph G is a pair of sets, $G = (V, E)$ where $V = \{v_0, ...\}$ is the set of vertices and $E = \{e_0, ...\}$ is the set of edges. The edges can have their own labels as in the figure, or they can be denoted by the pair of vertices they connect: e_0 could be also named as (v_0, v_1) or v_0v_1. A graph can be *undirected* (figure 2.2a) if the edges have no sense of direction, or it can be *directed* (figure 2.2b) if the direction of travel along an edge matters; e_0 is distinct from e_4 because the have different directions although they connect the same nodes v_0 and v_1. A directed graph can also be called a *digraph*.

To decide how "big" a graph is, one can count the number of vertices, $|V|$, to get the *order* or *cardinality* of the graph. If one counts the number of edges, $|E|$ one gets the *size* of the graph. In figure 2.2, G_1 has order 5, and size 4; and G_2 has order 4 and size 5.

Edges are *adjacent* if they share a common vertex, and vertices are adjacent if they are connected by an edge, one can also say that v is *incident* with e. In figure 2.2a, v_0 and v_1 are adjacent but not v_1 and v_3, and e_0 and e_1 are adjacent but not e_0 and e_2.

The number of edges connecting to a vertex is called the *degree* of the vertex, $d(v)$. In figure 2.2a, $d(v_2) = 3$, $d(v_3) = 1$ and $d(v_4) = 0$. A vertex of degree 1 is called a *pendant* vertex, or *leaf*, and a vertex of degree 0 is called *isolated*. If all *components* of a graph are *connected*, then the graph is a connected graph. In figure 2.2 graph G_2 is connected, but graph G_1 is not because it has an isolated vertex as a component.

A graph is *planar* if it is possible to draw without edges crossing each other. It is *Eulerian* if one can travel over every edge in the graph only once (as in the *Königsberg bridge problem*). A graph is called *Hamiltonian* if one can visit every vertex in the graph only once (as in the *Travelling salesman problem*).

Travels in graphs can be called different names. Ruohonen [1] has the most general name *walk* for travel from vertex to vertex along edges. A walk is *open* if it ends on a different vertex than it started, or *closed* if it ends on the same vertex. If an edge is traversed only once, the walk is called a *trail*. If any vertex is visited only once then the trail is a *path*. If the walk is a path but with the start and ending vertices being the same, then the walk is a *circuit*.

One can partition a graph into *subgraphs* if on places a cut in a vertex (*cut vertex*) or over a set of edges (*cut set*). In figure 2.2a a cut vertex could be v_2 and a cut set could be $\{e_1, e_3\}$.

2.1.1 Graph representation

There are different ways of representing graphs. We have so far used

- Graph diagram
- Set definitions, $V(G) = \{v_0, v_1, v_2, v_3, ...\}, \quad E(G) = \{e_0, e_1, e_2, e_3, ...\}$

One can also use

- *Adjacency matrix*
- *Incidence matrix*
- *Adjacency list*

An *adjacency matrix* is a matrix that shows if vertices are adjacent or not. A value of 0 indicates that the vertices are not adjacent. For an *unweighted* graph, adjacency can be indicated with a 1, or if it is a *weighted graph*, it can be the value of the weight (e.g. edge length or cost). From figure 2.2a:

$$D = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{2.1}$$

An *incidence matrix* describes which vertices that are incident with which edges. In a directed graph it is *positive* if it is the *start* vertex of the edge, or *negative* if it is the *ending* vertex of the edge. From figure 2.2b:

$$A = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_0 & e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & -1 & -1 \\ -1 & 1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix} \end{array} \tag{2.2}$$

A *dense* graph has almost all vertices connected to each other, i.e. there are few 0s in the adjacency matrix. In a *sparse* graph there are a lot fewer edges than there could be, so the adjacency matrix has a lot of 0s. To be space efficient, especially in computing, it can therefore be better to represent a graph as an *adjacency list*, which simply lists for each vertex which other vertices it is adjacent to. No 0s needs to be included. From figure 2.2a:

$$v_0 : (v_1, v_2), \quad v_1 : (v_0, v_2), \quad v_2 : (v_0, v_1, v_3), \quad v_3 : (v_2) \tag{2.3}$$

2.2 Map routing

For graphs as those described above, there exists basic algorithms such as *Dijkstra* and *bidirectional search*, or more goal directed such as *A**, that tries to find the shortest path from vertex s (source) to vertex t (target). To do that, each edge needs to be associated with a length. That is, the *metric* is *distance*.

However, when it comes to map routing there can be other metrics that are more important than the shortest path. For example *time* (we want the shortest driving time); *road category* or *land use* (we don't want to route through a residential area with low speed limits, or avoid having to go by ferry); *turn cost* (turning slows driving down so prefer straight routes); *multimodal* (when going by public transport we want to minimize waiting and the number of exchanges); *via* (we want to travel via a specific road or city); and so on.

A really basic ingredient in map routing is of course also the fact that roads are directed, i.e. there can be one-way roads. It is also important to take into account that there can be turn restrictions, so that a turn is not allowed at a junction, although it looks like it on the map (and the graph). Even more complicating is the fact that different restrictions on roads might be permanent, or just temporary due to road work, accidents, etc, so there is a difference between *static routing*, where the metric costs are static, and *dynamic* routing where the costs fluctuate over time.

2.2.1 Overview

In an overview of route planning techniques from 2009 [4], it is stated that the starting point for a "horse race" in developing speed-up techniques started in 2005 (p.124), when continental sized road networks of Europe and USA were made publicly available. Before that, large map data had been proprietary and it was hard to compare different approaches. The last decade since then has seen a quick development in the area, so a new overview in a tech report in 2014 from researchers in German universities and Microsoft [5] stated that the previous report was now outdated. This last report is a great overview of route planning techniques from the basic *Dijkstra*, continuing to different families of techniques: *goal directed*, *separator based*, *hierarchical*, *bounded hop*. The report also describes combinations of different techniques and notes on *path retrieval* (getting a description of the shortest path, no just the cost), *dynamic networks* and *time dependence*.

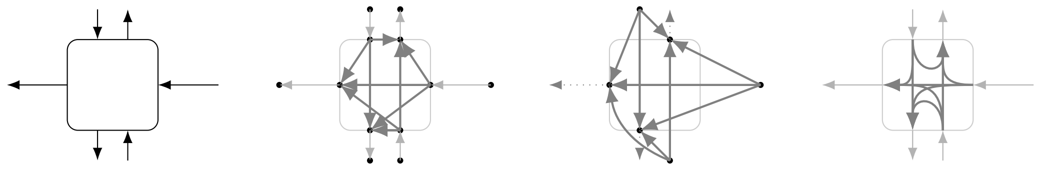
The motivation for the speed-up techniques is to enable "instant" route planning in large networks. The Dijkstra algorithm might need some seconds to complete a query, while one with some preprocessing might be able to perform a query in milli- or even microseconds. This is done by dividing the work into two distinct phases: the *preprocessing* phase, and the *query* phase. The preprocessing phase takes the original graph and performs transformations and builds new data structures. This is a process that can take a lot of time, from seconds to hours and even days depending on algorithm, and the data the size of the data structures might multiply several times. The gain is that the query phase executes almost instantly.

A lot of the research has been conducted on simple models without turn restrictions, so it is easy to compare the speed gain to Dijkstra's algorithm, and one have thought that adding turn costs or restriction on top will not be so hard. However, it turns out that most algorithms with large gains in speed are quite inflexible and have trouble to incorporate changing restrictions and metrics without the need for running the preprocessing phase again [6, p.2]. A more flexible way would be to have a separation of *topology*, i.e. how the graph "looks" with vertices and edges, from the *metrics*, i.e. the cost for travel in the graph.

Those techniques with preprocessing can be characterized as *offline* techniques, while techniques that perform all processing in the query phase can be called *online*. As said before, a lot of the research has been done on continental scale maps. But if one restricts one self to a metropolitan map with a graph of a smaller size, then perhaps the queries perform fast enough without preprocessing, or the preprocessing phase is so fast that can be run online?

2.2.2 Map representation

To have a real-world application that performs route planning, it also needs to seriously take turn restrictions into account, and to be more useful also be able to handle *turn costs*. There exists several techniques for that, see figure 2.3. The most straight-forward technique might be to introduce some new vertices so that edges have a *head* and a *tail* vertex, and turns are modeled by connecting head and tail vertices; this is called a *full-blown* representation (figure 2.3b). One of the most used representations is by converting the original *vertex-based graph* to an *edge-based graph* (also called *arc-based graph* or *line graph*). It can be viewed as connecting tails to tails, see figure 2.3c. These techniques introduces several new edges and vertices, inflating the space needed for the data structures. A more compact representation is keeping a table for each junction with the associated turn and turn costs, see figure 2.3d.



(a) None (b) Full (c) Edge-based (d) Turn table

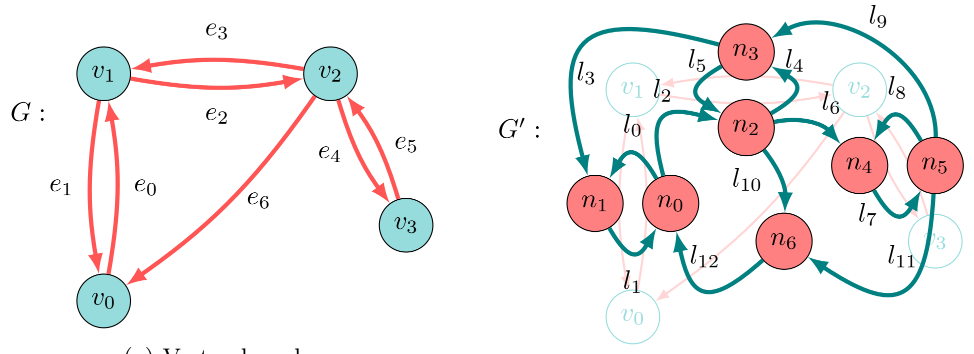
Figure 2.3: A closer look at a junction with two bidirectional and two unidirectional roads with different turn representations. *(After [6, p.8].)*

Edge-based graph

An *edge-based/arc-based/line* graph is a pretty straight forward transformation, where the edges of the original graph is turned into vertices in the transformed graph, and two vertices in the transformed graph is connected by an edge if a turn is allowed in the original graph. To make it simpler to distinguish beteween the vertex-based original graph and the new edge-based graph, we can call the new vertices *nodes*, and the new edges *lines*, i.e. "*road* = *node*", with nodes connected by lines, if a travel is allowed. This gives us a graph $G' = G_{edge-based} = (N, L)$ where N is the set of *nodes*, $N = \{n_0, n_1, ...\}, N = E$ and L is the set of *lines*, $L = \{l_0, l_1, ...\}$ connecting the nodes, see figure 2.4.

As one can see, the complexity and size of the graph grows in the transformation, what was $|V| = 4$ vertices and $|E| = 7$ edges became $|N| = 7$ nodes and $|L| = 13$ lines. The increase in size of the data structures is one drawback with this simple transformation, but on the positive side is the fact that one can apply ordinary algorithms such as Dijkstra to the edge-based graph just as easily as on the original graph. Another disadvantage might be that it lets the topology represent metrics, i.e. a turn restriction is hard coded into the topology, so how does one handle temporary restrictions?

Volker [7] has written a study on "Route Planning with Turn Costs" and uses edge-based



(a) Vertex-based

(b) Edge-based

Figure 2.4: Transformation of a graph G to an *edge-based* graph G'.

graphs as the foundation. He also introduces an *interface graph* as a link between a vertex-based and an edge-based graph. This new graph builds on an elaboration on what a turn and a junction actually is, with an incoming and an outgoing edge being adjacent on the junction vertex. One can thus see a turn T as either $T = (e_{in}, e_{out})$, i.e. an incoming and an outgoing edge, or as $T = (s, t, u)$ as going from vertex s to vertex u, *via* vertex t. This kind of finer look at what a turn is, is also used by others to build a more compact representation, see below, section 2.2.2.

Turn tables

One way of dealing with a more compact representation of turns and restrictions and costs associated with them is presented in "Efficient Routing in Road Networks with Turn Costs" [8], where they use a standard vertex-based graph (where roads are edges and junctions are vertices) on which they use speed-up techniques such as *Contraction Hierarchies*, but they associate each junction with a table that describes the costs of turns there, essentially describing all possible turns and restrictions. It turns out that a lot of junctions in a road map share the same characteristics and therefore can share the same *turn table*, so that on a map over Europe on average 18 vertices could share the same turn table. Thereby they managed to reduce preprocessing time by a factor 3.4 and space by a factor 2.4 with the same query times.

Another solution which uses the vertex-based graph with a turn table at the junctions is described in "Customizable Route Planning in Road Networks" [6, p.6]. Their solution uses a *separator-based* speed-up technique, which has been viewed as slower than hierarchical techniques, but they argue that this is the most flexible solution with a clean separation of *topology* and *metrics*, with two preprocessing stages before the query stage; one slow *metric-independent* that works on the topology, and a faster *metric customization* stage that can be run for each metric (takes about a second). This solution also uses the fact that many junctions share the same characteristics and therefore turn tables can be shared.

Bidirectional edges

So far we have thought of the directed original graph as having an edge for each direction of travel between two vertices, as with edges e_0 and e_1 in figure 2.4a. But this can be more compactly represented with one edge having a couple of flags indicating directions, meaning that for most roads we need not have two complete edge structures, but only one with two extra bits indicating the direction. This is however not something that can be done in an edge-based graph as the *lines* in it are not bidirectional.

2.3 Map data

One needs to have good map data as the source for building these data structures and apply smart algorithms on. With poor data it does not matter if one has smart algorithms. A said earlier, a race in route planning started with the public release of previously proprietary data. About the same time *OpenStreetMap* also began, which is an *crowd-sourced* project, meaning that anyone interested can be a cartographer and contribute with map data. Over the years the project has grown to an impressive size, and is used as the base for many applications. However, it turns out that the map data actually lacks a lot of turn restrictions. They might be hard to enter, and they are impossible to spot when comparing aerial photos with maps. Efentakis et al. states that for Athens with 277 thousand vertices only 214 restrictions were entered. They propose an automated remedy by comparing GPS-traces to the maps and deducing that turns seldom made are actually banned and could be marked as restricted in the map data [9].

All the same, a lot of high quality applications exists built on *OSM* (OpenStreetMap) data and this project aims at that to.

2.3.1 Projections

Generally speaking, the globe is spherical, but a map is flat. That means that one somehow needs to *project* the spherical data on a flat surface. There exists a lot of different *projections* that tries to do it best. To keep track of which projection one is working in, one can identify it by its *SRID – Spatial Reference System Identifier* that uniquely identifies which projection and which kind of coordinate system one works with.

2.3.2 Topology

The *topology* is about the relationship between objects in a map [10]. If one only thinks of a map as a collection of lines, it is hard to make something out of that information. It becomes useful when we understand the topology, that "this line is connected to that line at this point". The we have a relationship between the lines and can understand how to travel on the map.

Analyzing the topology also makes it possible to correct errors made while adding items to the map data, such as if two lines don't actually meet. Then there is a gap and there is no connection. When analyzing, one can opt to connect lines that are within a small distance of one another, thereby correcting mapping errors.

2.4 Available applications

Some of the research described earlier in this chapter is used actual working applications. For example *CRP (Customizable Route Planning)* [6] is used in Bing Maps, and *CH (Contraction Hierarchies)* are used in for example GraphHopper[1] and OSRM[2], and they are open source routing applications. So the source code is available so one can study how the data structures are implemented and how the algorithms work. There exists a lot of other solutions built on OSM as well[3], using Contraction Hierarchies or other speed-up techniques.

2.5 Memory or database

All research referred to so far has been about building data structures to be held *in memory* so algorithms can operate on them. But as we speak of *queries*, on might think that databases and query languages might be useful as well. There is some research, and a technique called *HLDB* is interesting [11], [12]. It is fast enough, and very flexible, permitting to query for *alternative routes* and *points of interest*.

[1]https://graphhopper.com/
[2]http://project-osrm.org/
[3]http://wiki.openstreetmap.org/wiki/Applications_of_OpenStreetMap

pgRouting is an open source database extension to *PostgreSQL*, often used for holding OpenStreetMap-data. It has a function called `pgr_trsp`[4] that looks for the shortest path with turn restrictions, so obviously standard relational databases can be part of a solution.

[4]http://docs.pgrouting.org/2.0/en/src/trsp/doc/index.html

3 Methodology

The overall methodology for the development is a combination a of *Behaviour Driven Development, BDD* and *Test Driven Development, TDD*, meaning all features of the module have either a *scenario* (BDD) or a *test case* (TDD) written.

As for tools, some are given in the specifications (see Appendix A), while others have been chosen during a selection process. Below is presented the tools chosen, and in some cases what alternatives that were also considered and tested. The categories are:

- Behaviour and Test Driven Development.
- Database.
  - Database and extensions.
  - Loading OpenStreetMap into database.
  - Build topology.
  - Examining map data.
  - Connecting to database from application.
- Reading configurations from `json`-file.
- Building graph.

3.1 Behaviour and Test Driven Development

Behaviour Driven Development tests usually have the structure: *Scenario* → *Given* → *When* → *Then*, written with words to describe the steps. An example in the *Gherkin* language is shown in listing 3.1.

```
Scenario: vectors can be sized and resized
    Given: A vector with some items
    When:  the size is increased
    Then:  the size and capacity change
```

Listing 3.1: Example of a BDD scenario in *Gherkin*.

So when developing BDD style one has to think through different scenarios and write them down, which can be helpful when thinking about what one tries to accomplish.

3.1.1 Tools, installation and usage

The testing library for this project is Catch[1], which is a small library for both BDD and TDD, where the BDD *"scenario"* corresponds to a TDD *"test-case"*, and *"given"*, *"when"*, *"then"* corresponds to *"section"*, meaning one can choose the development style one wishes. *Catch* was chosen because it is header only, and there is no need for complicated building of libraries and setting up paths; one can just include the header in the project and go.

Simply download the file `catch.hpp` and put it either in your project tree or in your path for includes.

Include the header in the source for your test, and get Catch to provide a `main`-method. See listing 3.2 for an example of how to implement the above stated "feature".

[1]`http://www.catch-lib.net`

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <vector>

SCENARIO ("Vectors can be sized and resized", "[vector]") {
    GIVEN ("A vector with some items") {
        std::vector<int> v(5);

        REQUIRE (v.size() == 5);
        REQUIRE (v.capacity() >= 5);

        WHEN ("the size is increased") {
            v.resize(10);

            THEN ("the size and capacity change") {
                REQUIRE (v.size() == 10);
                REQUIRE (v.capacity() >= 10);
            }
        }
    }
}
```

Listing 3.2: A basic BDD scenario with Catch

3.1.2 Alternatives

The BDD style of developing seems not to have caught on in c++ so much. There are a few libraries. Cucumber-Cpp[2] was investigated as it is an implementation for c++ of the Cucumber tool, which is widespread in many programming languages, so one could write the test for features in the ordinary `.feature`-files in the *Gherkin* language, that are common for writing features for tests. But I could not get Cucumber-Cpp to build correctly with CMake and the dependencies.

3.1.3 Remarks

It should not be a very difficult task to write a script that reads a `.feature`-file and outputs a template in c++, using the Catch syntax.

If one were to *not* go for BDD-style of testing, then one could go for TDD testing using Boost Test, if one would want to keep using Boost for most parts of the project.

3.2 Database

The database of choice, and in the requirements of the project, is PostgreSQL[3], with the extension PostGIS[4] which gives the database *spatial* and *geographic* capabilities, which are needed to simplify working with maps and such, for example when needing to measure distances in different projections. How to set up the database with users and passwords and such are not given in this report, but it is not so hard. When setting up databases one can interact via either the commandline or a *graphical user interface, GUI* such as *pgAdmin3*.

3.2.1 Tools, installation and usage

The tool set was given in the requirements, as mentioned before. On my Debian/Ubuntu system they can be installed as shown in listing 3.3.

[2] `https://github.com/cucumber/cucumber-cpp`
[3] `http://www.postgresql.org/`
[4] `http://postgis.net/`

```
$ sudo apt-get install postgresql postgresql-contrib-9.3 postgis postgresql-9.3-postgis-2.1 pgadmin3
↪  osm2pgsql
```

Listing 3.3: Installation of database tools

Listing 3.4 shows how to create a new database called `mikh_db` with a user "jonas" (that is already set up as a user with rights to create databases), and enabling the needed spatial extensions to work with map data.

```
$ createdb mikh_db -U jonas
$ psql -U jonas -d mikh_db -c "CREATE extension postgis;"
$ psql -U jonas -d mikh_db -c "CREATE extension postgis_topology;"
$ psql -U jonas -d mikh_db -c "CREATE extension hstore;"
$ psql -U jonas -d mikh_db -c "SET search_path=topology, public;"
```

Listing 3.4: Create database and enable spatial extensions.

3.2.2 Loading map data

To get the `.osm`-file, which is actually in `xml`, into the database one needs a conversion tool to parse the file and populate some tables with data.

Tools, installation and usage

There exists several tools for importing `OSM` data into a database. It was hard to know which one to pick and different options were tried, but the chosen tool is osm2pgsql[5]. It was installed in listing 3.3.

```
$ osm2pgsql -U jonas -d mikh_db -k -s mikhailovsk.osm
```

Listing 3.5: Usage of `osm2pgsql`.

Listing 3.5 reads an `.osm`-file in the current directory and populates the database `mikh_db`. The flags `-k` tells to use "hstore" for tags and, `-s` to make a "slim" conversion. Two different `.osm`-files have been provided for testing, "mikhailovsk.osm" and "partille.osm", hence the usage of "mikhailovsk.osm".

One might specify other flags as well. Among the options is to chose a different projection than the default `900913`. It is also possible to specify a `.style`-file which is a configuration over which tags to import. It is possible to use this file to decide which tags to import into the database and which tags to discard.

Alternatives

There exists a bunch of other tools that can convert OpenStreetMap files into database tables, such as *Osmosis, Imposm, osm2po, osm2pgrouting*, and others; all with different strengths and weaknesses, such as being good and free, but not open source.

3.2.3 Building topology

With the data in the database, it is time to build a topology of the map data, saying how the vertices and edges are connected, to make it possible to build a routable graph. A lot of the "nodes" in the `osm`-data are only useful for describing the geometry, while what is interesting when routing are the nodes that connects edges; that is the junctions at which roads meet. Therefore it is essential to analyze the data an build tables that contain information about the topology.

[5]http://wiki.openstreetmap.org/wiki/Osm2pgsql

One can have different thoughts of when to do this. It would be possible to do this at the preliminary step when loading the data into the database. That would be good if one was certain of that the topology is stable. If the network is more volatile, it would be better to build the topology on every query, to be certain that one always has the most up-to-date information. On the other hand; the topology for a road network should be stable, and temporary closures and other changing conditions will be better reflected in tags that can be queried when calculating costs for routing. That is the path taken in this project.

Tools, installation and usage

The choice for this project is PostGIS' topology extension. It is a part of PostGIS, which is already installed.

The osm-data from osm2pgsql has a table for all the lines in the map, called planet_osm_line, but in addition to roads it contains lines for railways, waterways, borders, buildings etc. So to build routing data we need to extract the lines only representing the roads, and put it in a new table. Listing 3.6 shows that.

```
$ psql -U jonas -d mikh_db -c "CREATE TABLE roads AS SELECT * FROM planet_osm_line WHERE highway IS
↪  NOT NULL;"
```

Listing 3.6: Creating a table with only roads in it.

Then one can build a topology of the roads, as shown in listing 3.7. The first line creates a new schema called roads_topo which will hold the topology data in the projection 900913 (the projection used when loading the database). The second line adds a column called topo_geom to the table roads in the public schema. The third line connects that column with the newly built corresponding topology in the roads_topo schema. The topology is built with a tolerance of 1.0 units. The unit for this projection is meters, so it means that if there are several nodes within 1.0 meters or a node within 1.0 meters from a road, they are joined. This can be essential to building a routable network. When running the validation tool in JOSM on the mikhailovsk.osm-file, it reported 16 suspect cases with nodes close but not connected, see figure 3.1.

```
$ psql -U jonas -d mikh_db -c "SELECT topology.CreateTopology('roads_topo', 900913);"
$ psql -U jonas -d mikh_db -c "SELECT topology.AddTopoGeometryColumn('roads_topo', 'public',
↪  'roads', 'topo_geom', 'LINESTRING');"
$ psql -U jonas -d mikh_db -c "UPDATE roads SET topo_geom = topology.toTopoGeom(way, 'roads_topo',
↪  1, 1.0);"
```

Listing 3.7: Building a topology with PostGIS.

Alternatives

One might load the database, and build a topology with osm2pgrouting[6], and the PostgreSQL extension pgRouting[7]. That solution is pretty smooth, and might heal the topology with a tolerance, but it seems it only builds the topology and does not give access to tags and other information usable when calculating costs.

Another attempt, was to run a topology building SQL function (as in http://blog.loudhush.ro/2011/10/using-pgrouting-on-osm-database.html), and then run another function to remove all nodes without topological meaning. But that lead to the problem shown in figure 3.1, as there had been no "healing" of nodes first. One solution could of course to write antoher function for that, or to fix the .osm-file manually in JOSM before loading it into the database. But the solution with the PostGIS topology seems like a better way to go.

[6]http://pgrouting.org/docs/tools/osm2pgrouting.html
[7]http://pgrouting.org/index.html

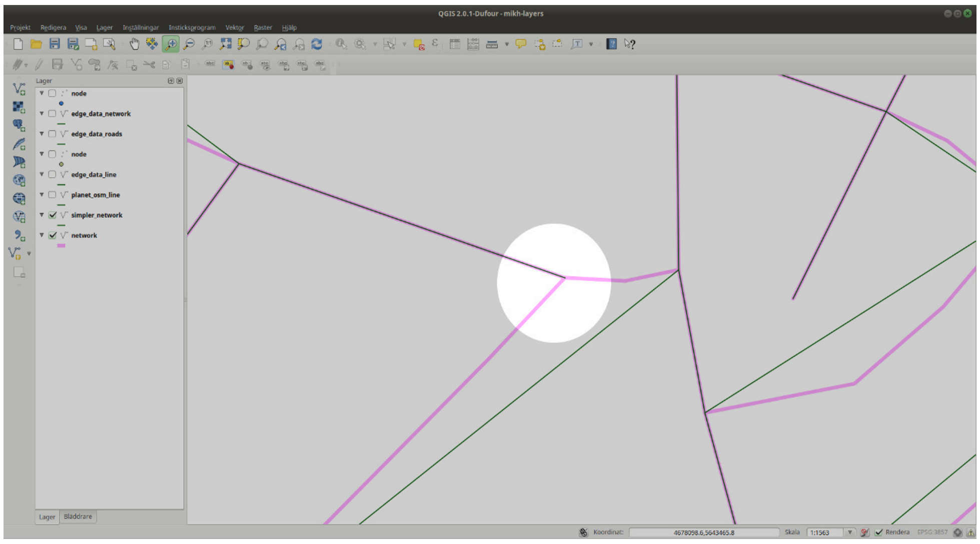


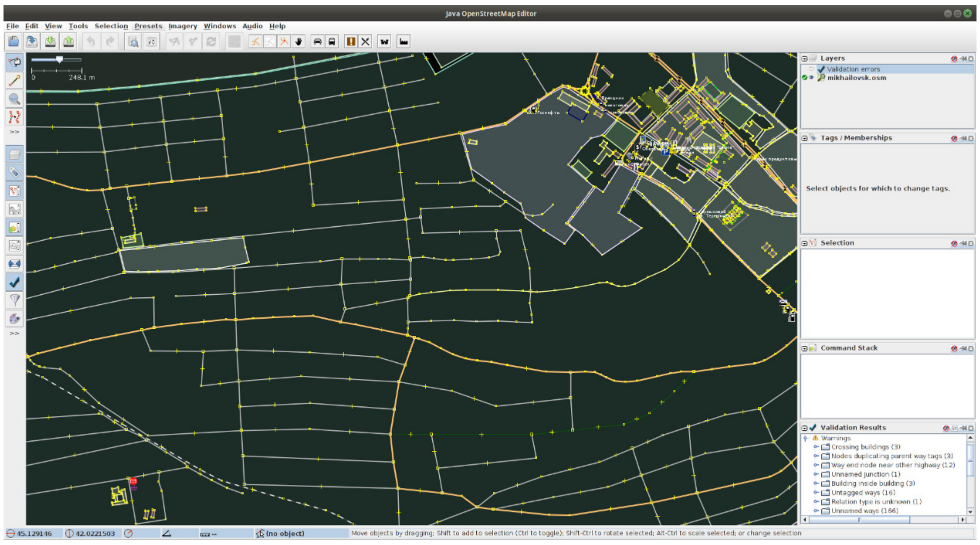
Figure 3.1: Error building topology with a node close but not connected.

3.2.4 Examining map data

Map data lends itself to visualization. And it is also useful to build a mental model of what one is working on, and to see the results.

JOSM

JOSM[8] is an editor for OpenStreetMap. It can open `.osm`-files and display them, inspect elements of the map, and it has tools for editing and validation, meaning one might be able to fix files that has problems. See figure 3.2.


Java OpenStreetMap Editor
File
Edit
View
Tools
Selection
Presets
Imagery
Windows
Audio
Help
248.1 m
Layers
Validation errors
mikhailovsk.osm
Tags / Memberships
Select objects for which to change tags.
Selection
Command Stack
Validation Results
45.129146
42.0221503
(no object)
Move objects by dragging; Shift to add to selection (Ctrl to toggle); Shift-Ctrl to rotate selected; Alt-Ctrl to scale selected; or change selection

Warnings
Crossing buildings (3)
Nodes duplicating parent way tags (3)
Way end node near other highway (12)
Unnamed junction (1)
Building inside building (3)
Untagged ways (16)
Relation type is unknown (1)
Unnamed ways (166)

Figure 3.2: JOSM editor with Mikhailovsk map.

[8]https://josm.openstreetmap.de/

QGIS

QGIS[9] is a tool that can load spatial data from databases and display, as well as load for example `.osm`-files. It makes it good to visualize for example query results or transformations you have made in the database. See figure 3.3 for an example with layers of PostGIS-data of "Mikhailovsk" on top of each other.

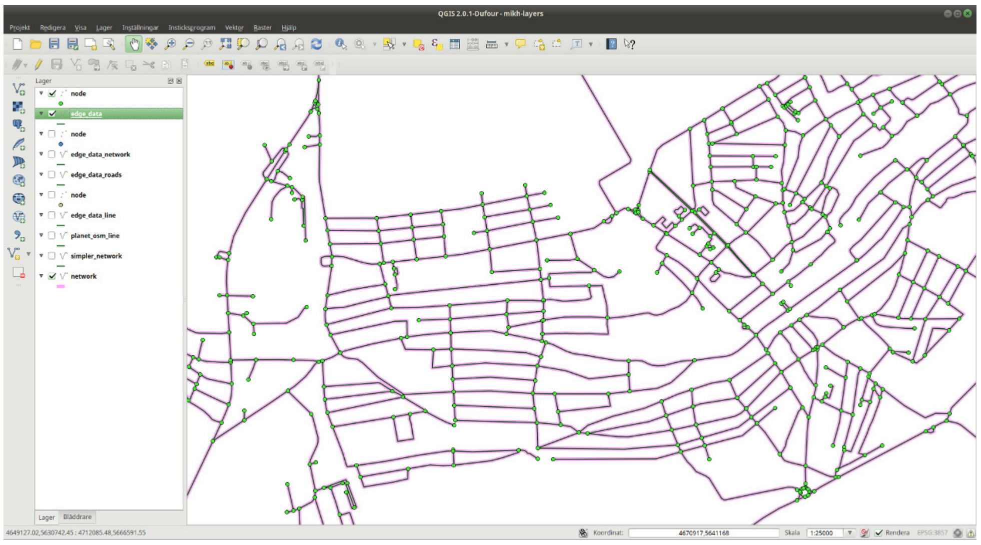


Figure 3.3: QGIS editor with Mikhailovsk map from PostGIS tables.

When loading data from PostGIS one might have to specify which projection to use for display. The default projection when loading `.osm`-files into the database using `osm2pgsql` is `SRID 900913`, and to display that correctly in QGIS one needs to use the projection `EPSG:3857`.

3.2.5 Connecting to database

After the module has read in the configuration, the next step is to connect to the database and perform some work on the map data before extracting relevant information.

The connection to the PostgreSQL database is handled by the library libpqxx[10], and while there exists a few alternatives, it is natural to go for the official alternative.

Tool, installation and usage

Installation of `libpqxx` is shown in listing 3.8.

```
$ sudo apt-get install libpqxx-4.0
```

Listing 3.8: Installing `libpqxx` on a Debian/Ubuntu system.

It is pretty straightforward to use: include the header, make connections and transactions. A snippet is shown in listing 3.9.

```
#include <pqxx/pqxx>
...
pqxx::connection conn("dbname=testdb user=tester password=tester hostaddr=127.0.0.1 port=5432");
```

Listing 3.9: Inlcude header and make a connection to the database.

When compiling, one must link with the libraries `pqxx` and `pq`, as shown in listing 3.10.

[9] http://www.qgis.org/
[10] http://pqxx.org/development/libpqxx/

```
$ g++ mytest.cpp -lpqxx -lpq -o mytest
```

Listing 3.10: Linking `libpqxx` at compile time.

3.3 Configuration

The module should be configured by a settings file, written as *json*. Settings can be related to the database such as host address, table names etc; or it can be configuration of costs for the routing such as speed limits, traffic lights, turn restrictions.

3.3.1 Tools, installation and usage

There is no meaning in writing a json-parser for this module as there exists lots of good libraries. The one chosen is Boost Property Tree[11], as the project uses other Boost libraries, and it simple enough to get started with.

As several Boost packages will be used in this project, it is just as good installing all of them (for a Debian/Ubuntu based system), see listing 3.11.

```
$ sudo apt-get install libboost-all-dev
```

Listing 3.11: Installation of Boost libraries.

An example to see how simple it is to parse a json-file is shown in listing 3.12.

```cpp
#include <string>
#include <iostream>
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>

void readJsonFile(const std::string& filename) {
    boost::property_tree::ptree pt;
    boost::property_tree::read_json(filename, pt);
    std::string host = pt.get<std::string>("host");
    int port = pt.get<int>("port");
    std::cout << "Host: " << host << ", port: " << port << std::endl;
}
```

Listing 3.12: Parsing a json-file.

3.3.2 Alternatives

One could go for a header-only solution here as well, such as jsoncons[12], which was also tested, but *Boost Property Tree* seemed nice and easy to get working if one already has the Boost libraries installed.

3.4 Build Graph

The requirements said that the *"Boost Graph Library (BGL)"* should be used for representing the graph and for returning the *line graph* structure for routing back to the calling application.

As discussed in section 2.1.1, the most space efficient way of representing a sparse graph is an *adjacency list*, and the *BGL* has such a data structure. Using template arguments one can configure what kind of data structures to use for the lists of *edges* and *vertices*, and the data structures to use for *edges* and *vertices*, and if the graph is *directed* or *undirected*.

If one has some properties of the edges and vertices that one wishes to keep in the graph (like the "cost" or some identifier of an edge), it is possible in several ways, either as *"interior"*

[11] http://www.boost.org/doc/libs/1_54_0/doc/html/property_tree.html
[12] https://github.com/danielaparker/jsoncons

or *"exterior"* properties, and `adjacency_lists` can use *interior* properties either as *"bundled properties"* or as *"property lists"*.

The *property lists* are external structures for some property that gets mapped to e.g. an edge in the graph.

The *bundled properties* are more intuitive, by using data structures as the *descriptors* of the *edges* and *vertices*, and with the properties as fields.

An example from the documentation for *bundled properties* [13] shows the difference clearly, in terms of how easy or hard it is to read or understand the code in the different approaches. See listing 3.13 showing the *bundled* approach and listing 3.14 showing the *property list* way.

```cpp
// Vertices = Cities
struct City
{
  string name;
  int population;
  vector<int> zipcodes;
};

// Edges = Highways
struct Highway
{
  string name;
  double miles;
  int speed_limit;
  int lanes;
  bool divided;
};

// Map using `City` as vertex descriptor and `Highway` as edge descriptor.
typedef boost::adjacency_list<
    boost::listS, boost::vecS, boost::bidirectionalS,
    City, Highway>
  Map;
```

Listing 3.13: Bundled properties in a graph.

```cpp
typedef boost::adjacency_list<
    boost::listS, boost::vecS, boost::bidirectionalS,
    // Vertex properties
    boost::property<boost::vertex_name_t, std::string,
    boost::property<population_t, int,
    boost::property<zipcodes_t, std::vector<int> > > >,
    // Edge properties
    boost::property<boost::edge_name_t, std::string,
    boost::property<boost::edge_weight_t, double,
    boost::property<edge_speed_limit_t, int,
    boost::property<edge_lanes_t, int,
    boost::property<edge_divided, bool> > > > > >
  Map;
```

Listing 3.14: Property lists in a graph.

In this project, the *bundled properties* were chosen for their ease of understanding and reading.

4 Implementation

As stated at the start of chapter 3, the software module called the *"Line Graph Utility", (LGU)*, that should be the outcome of this project, is sequential in nature. The complete specification is available in appendix A, but here is an outline of the main use case.

- The using application calls the LGU's `get_directed_line_graph()`.
- LGU queries the PostGIS database and builds a graph from the road network.
- LGU builds a directed line graph from the previous step (i.e. it converts nodes to edges and assigns weight to those edges based on road signs and other elements which are present in the nodes).
- `get_directed_line_graph()` returns a directed line graph structure which is based on a C Boost graph structure to the function caller.

This can be expanded to a series of steps. First comes a preliminary step, not actually part of the module, but essential during development and testing:

- Loading the map data into the database and build a topology.

The following steps are performed during development and usage of the tool:

- Load configurations from *json*-file.
- Get the relevant edges and vertices from the database; store the topology.
- Apply restrictions and costs on the topology.
- Build a graph structure from the topology, using *Boost Graph Library.*
- Transform the structure into a *line graph (edge-based/arc-based graph).*
- Return the line graph.

4.1 Design

The sequential nature of the module, with a few easily identifiable objects, lead to no big design process was deemed necessary. Taking an object oriented approach, it is easy from the above list to identify *configuration (and configuration reader); edges; vertices; database; topology; restrictions; costs; graph (and graph builder and transformer); line graph.* All can be packaged up in a *Line Graph Utility.* The design therefore evolved gradually without a master plan more specific than this.

Another reason for this, was that this project was a discovery into not really well understood territory, despite some introductory research. It was necessary to learn the tools and concepts as the project proceeded, so the design and implementation grew incrementally. The incremental goals set during development, was to be able to build a graph from the map data, later extended to being able to build a line graph from that, to finally being able to apply restrictions and costs to the graphs.

A decision that was made early on, was to try not to pass pointers around, but instead use references, to reduce the complexity of memory handling. That means that a lot of functions gets passed in a reference to an object to fill in, rather than return a pointer to a newly constructed object. All the same, some pointers could not be avoided and raw pointers were used in those cases.

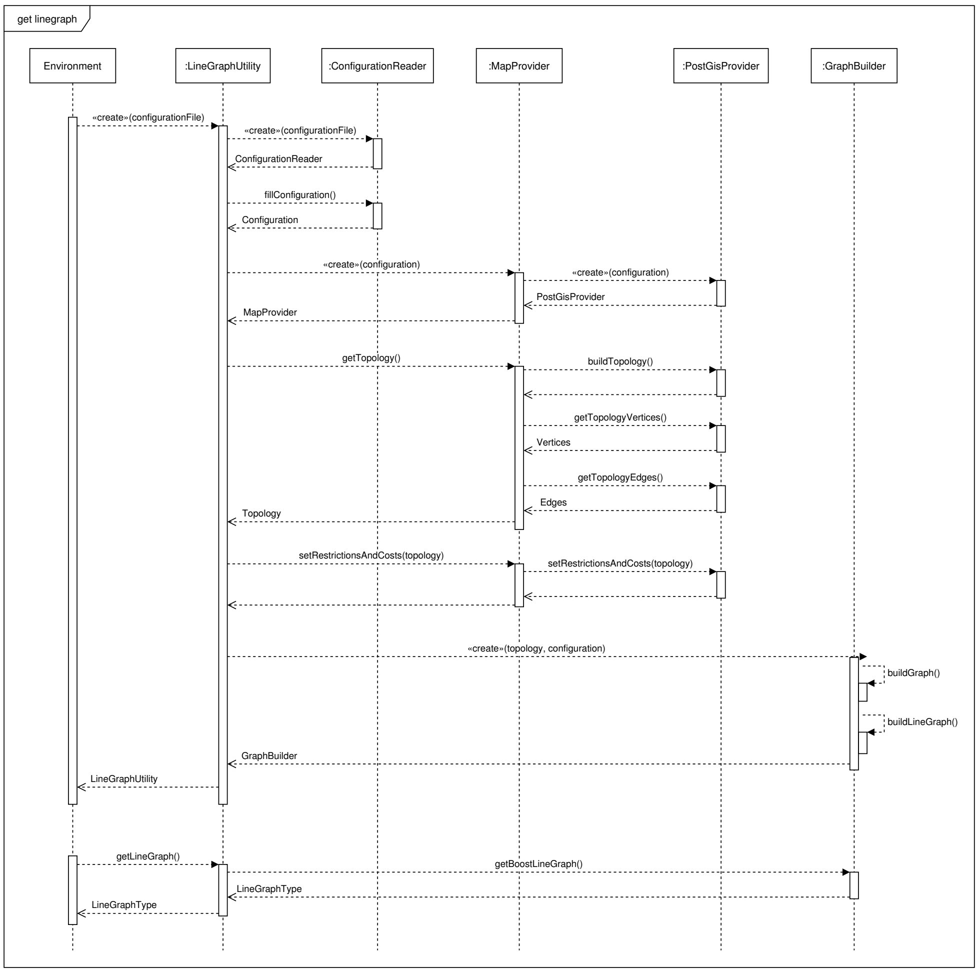


Figure 4.1: Sequence diagram of main use case to get a line graph.

4.1.1 Dynamic design

The sequence presented above has been refined into a design that can be shown in a sequence diagram, see figure 4.1 and appendix B.1.

The calling application *"Environment"* instantiates a `LineGraphUtility` object with the file name to a configuration file. The `LineGraphUtility` instantiates a `ConfigurationReader` that can be asked to fill in a `Configuration` object. The configuration contains among other things, a setting for which `MapProvider` to use. The idea is that one can read the *OpenStreetMap* data in several ways; for example parse the `.osm`-file, or use different databases or different tools to import `.osm`-files into the database. Hence the flexibility by using an abstract *map provider*. The only implementation in this project so far is the `PostGisProvider`, but others could be developed if it turns out there are better ways to access the map data.

So the actual work on retrieving the map data is performed by the `PostGisProvider`, that is fetching the `Topology` and applying *restrictions* and *costs* on the topology. The idea behind this separation is that the topology should be reasonably stable and constant, and that dynamic changes in the traffic, such as blocked roads, should be handled as restrictions and costs that are applied to the static topology. But it is also possible to perform an update on the topology if needed, for example if there has been built a new road. See figure 4.2 for a diagram of updating the restrictions and costs, and figure 4.3 for a diagram on updating the topology, (also in appendix B.2 and appendix B.3).

Back to figure 4.1, after having a restricted topology we instantiate a `GraphBuilder` object with the topology och configurations. This `GraphBuilder` builds a directed graph, and converts it to a *line graph*. If all went well the `LineGraphUtility` now is ready to serve the calling application a *line graph* any time it gets called.

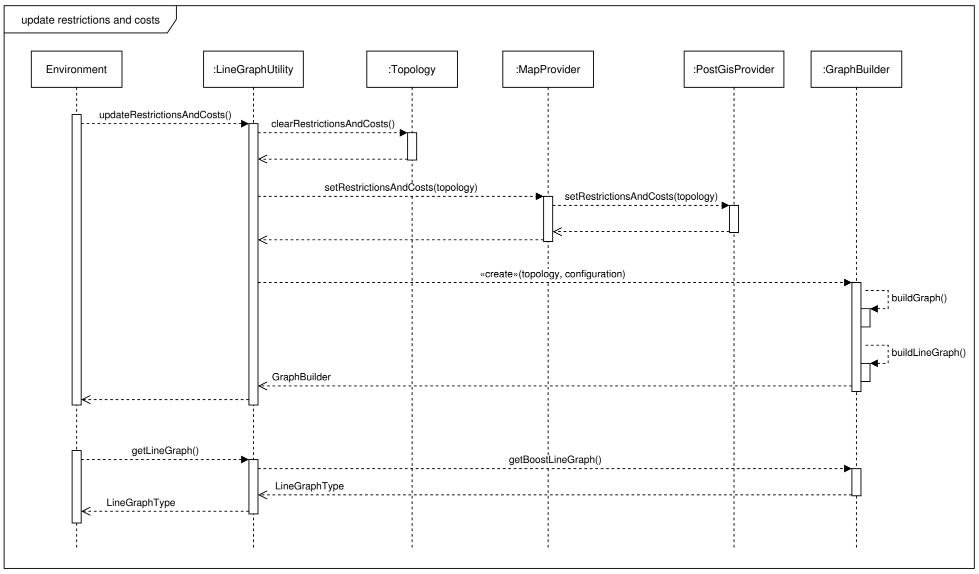

update restrictions and costs
Environment
:LineGraphUtility
:Topology
:MapProvider
:PostGisProvider
:GraphBuilder
updateRestrictionsAndCosts()
clearRestrictionsAndCosts()
setRestrictionsAndCosts(topology)
setRestrictionsAndCosts(topology)
«create»(topology, configuration)
buildGraph()
buildLineGraph()
GraphBuilder
getLineGraph()
getBoostLineGraph()
LineGraphType
LineGraphType

Figure 4.2: Sequence diagram of updating costs and restrictions on a topology.

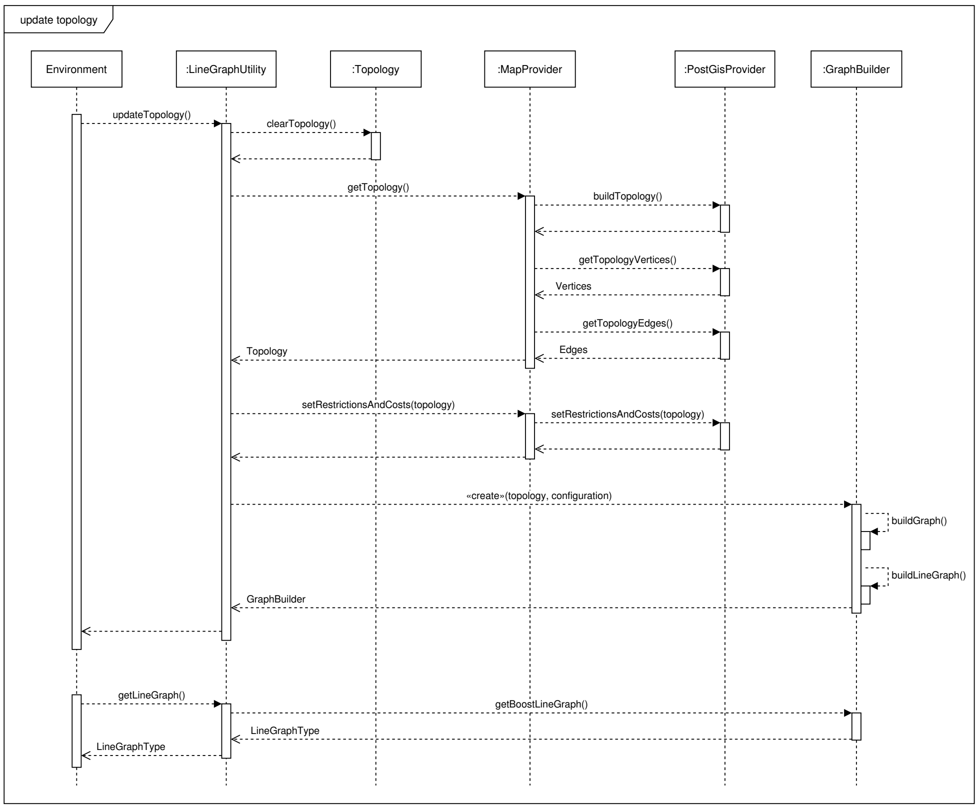


Figure 4.3: Sequence diagram of updating the topology.

4.1.2 Static design

A few classes were introduced in the sequence diagrams above, and a more complete view of the classes can be seen in the class diagram in figure 4.4 and in appendix B.4.

As can be seen, the application is divided in a few *"packages"*:

- `lgu`: The entry point to the `LineGraphUtility`.

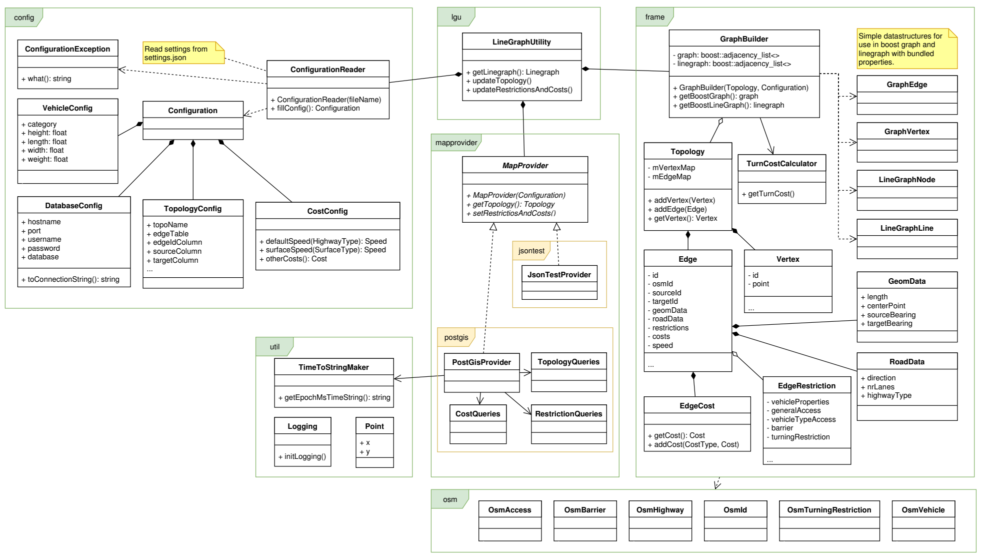


Figure 4.4: Class diagram of the *line graph utility*.

- `graph`: Classes related to graphs like `GraphBuilder`, `Topology`, `Edge`, `Vertex`.
- `mapprovider`: Classes related to providing map data.
- `config`: For handling configurations.
- `osm`: Helper classes for constants and concepts related to *OpenStreetMap* data.
- `util`: A few general helper classes.

This is an attempt to modularize the development process; to keep related classes in a specific area. It also makes navigating the code easier. Another attempt to make the packages coherent is that each package should have its specific *exception* class, that is the only public exception that gets thrown by the classes in the package. Other exception classes might be used internally but not exposed publicly.

4.2 Project structure

Apart from the packages above, there are directories in the project to support testing, setting up and documentation. This gives the project a basic directory structure as shown in listing 4.1.

```
./
├── catchtest/
├── config/
├── doc/
├── graph/
├── lgu/
├── mapprovider/
├── osm/
├── preparation/
├── util/
└── README.md
```

Listing 4.1: First level directory structure of the project

Each directory has a `README.md` file, a textfile in `markdown` mark up, that explains the purpose of the directory. Each directory with code also should have a `catchtest` directory, where there are tests for the code in the directory/package.

4.2.1 `catchtest`

Most of the code developed in this project has been part of a test, and all *"packages"* have their own set of tests. *Catch* is a header-only framework, and that header resides in this root `catchtest` directory, and there is a source file that calls that header and functions as the entry point when testing, see appendix listing C.2. It also contains a sub-directory for configuration settings used during development.

In section 3.1.1 the *Catch* testing framework is introduced, and in listing 3.2 there is a small example of how to write a scenario. There one can see that the scenario is tagged with `"[vector]"`, and those tags can be used to determine which tests to run. If no tags are specified all tests are run, but if one specifies a tag, it only runs tests that matches that tag. One can also specify which tests *not* to run by prepending a tilde ('~') to the tag. See listing 4.2. If one wants to see the results of all tests and not only failed ones, one can add the flag `-s`. If running tests from inside and *integrated development environment, IDE* one can specify the arguments in a *"run configuration"* instead.

```
$ testapp ~[timing] -s
```

Listing 4.2: Running tests except those tagged with `[timing]`, showing all results.

4.2.2 `config`

See appendix listing C.3 for the contents of this package, whose purpose is handling configurations. The central part is a data structure `Configuration`, made up of data structures for *cost, database, topology* and *vehicle*. The configurations are filled in by the `ConfigurationReader` class which reads from a specified settings file.

The `CostConfig` are mainly concerned with keeping track of speeds for different categories of roads and surfaces. The types are specified in `OsmHighway` in the `osm` package, see section 4.2.7.

The `DataBaseConfig` is about connecting to the database.

The `TopologyConfig` is about which tables and columns in the database to use when getting the topology.

The `VehicleConfig` keeps characteristics about the vehicle we are routing through the map, such as weight, height, category (as specified in `OsmVehicle`, see section 4.2.7).

4.2.3 `doc`

Listing C.4 shows the contents of this package, that contains the documentation for the project. It has a directory for this report, and a directory for the UML diagrams.

The diagrams are not meant to be exact documentation, but rather give an idea of the concepts and the big picture, and therefore method names might be missing or spelled differently than in the actual code.

4.2.4 `graph`

The `graph` package (see appendix listing C.5) is really the central package, where the `Edge, Vertex, Topology` classes are, and the `GraphBuilder` resides. In addition there are the classes for restrictions and costs for edges (`EdgeCost, EdgeRestriction`); a helper class for calculating costs for turns (`TurnCostCalculator`); and a couple of essential simple types, `Cost` and `Speed` who are simply `typedefs`.

Edge

The `Edge` class represents an edge in the topology. So it keeps track of the *source* and *target* vertices, the original *OSM id* and of course its *id* in the topology. It also keeps track of properties of the underlying *road* (number of lanes; one-way; the road category), and its geometrical properties (length; centre point and *bearing* of the edge at the vertices to calculate turning angles).

Vertex

The `Vertex` class simply keeps the *id* it has in the topology and the *coordinates* of it, it does not keep track of an *OSM id* as it might not correspond to nodes in the *OSM* data, since the vertices were calculated when building the topology.

Topology

The `Topology` class is a collection of maps. One maps `{edgeId ↦ edge}`, that is it makes it possible to get to the topology's `Edge` object when one only has an *id* for it in the topology. The corresponding map exists for vertices, `{vertexId ↦ vertex}`. In addition there is a *multi-map* `{osmId ↦ edgeId}` that has an `osmId` as the key, that maps to several `edgeIds`. The reason for this is that an original road in the *OpenStreetMap* might be split into several edges when building the topology.

GraphBuilder

The `GraphBuilder` keeps a `Topology` and a `Configuration` as the base for building a *graph* and a *line graph*. The `GraphBuilder` header begins with defining a bunch of types, such as the data structures to be used for *edges* and *vertices* in the *Boost graph* (`GraphEdge`, `GraphVertex`). They keep track of the *id in the topology* and the corresponding *id in the graph*, so one can move from the one to the other. The `GraphBuilder` keeps a *map* for *vertex* such as `{topoVertexId ↦ graphVertexDescriptor}`, and a *multi-map* for *edges* such as `{topoEdgeId ↦ graphEdgeDescriptor}`. These maps make it possible to access the data in the data structures underlying the *descriptor*, e.g to get to the fields in the `GraphEdge` through the descriptor. The reason to use a *multi-map* for the edges are that the edge in the topology is undirected, but in the graph the edges are directed, so in the graph there should be a directed edge for each lane of the road, and thus for most roads there will be several *graph edges* for each *topology edge*.

The `GraphBuilder` also has the data structures that makes up the *nodes* and *lines* in the *Boost line graph*, and a map `{edgeId ↦ lineGraphNodeDescriptor}`, since that is the definition of a *line graph*: an edge turns into a node in the transformed graph.

There are also `typedefs` to make the code easier to work with, see listing 4.3.

```
typedef boost::adjacency_list
   < boost::listS, boost::vecS, boost::directedS,
    LineGraphNode, LineGraphLine >                             LineGraphType;
```

Listing 4.3: `typedef` a *line graph* to make the code more readable.

The operation when building the *graph* is that first all *vertices* in the *topology* gets added to the *Boost graph*, and then each *topology edge* gets examined to see if there are any restrictions that apply. If not, the correct number of edges (corresponding to the number of lanes) in each direction gets added to the graph.

When building the *line graph*, all the edges in the graph are added as *nodes* in the *Boost line graph*. To find which other nodes to connect to (= which travels are allowed), one has to look at all *out-going* edges from the *end vertex* of the edge, functioning as a *via-vertex*. That *turn* (or travel) is really in three parts: *source edge* → *via vertex* → *target edge*. The *OpenStreetMap* data also gives the option to specify turns as travel via another edge instead of via a vertex, but that is complicated, see discussion in 4.2.7. When the *adjacent* edges have been identified, they are one by one checked if they are part of any *turning restriction*. If the edge is not part of such a restriction, then a *line* (that is a *line graph edge*) is constructed from the *source node* to this *target edge/node* and added to the *line graph*.

EdgeCost

The `EdgeCost` is a class for keeping track of different costs for edges. It has tree types of costs: *travel time; barriers; other*. The *travel time* cost represents the time it takes to travel the edge, and is thus dependent on the *length, speed limit or road category* and *surface*. The *barrier type* is for costs that comes from slowdowns imposed by barriers such as *speed bumps, gates* and such. The *other* cost are for slowing down for *signs, traffic signals, zebra crossings* and the likes.

A note on stop signs: they are associated with a road. But it is generally only applicable in one direction. For example the stop sign only affects the incoming edge in a junction, not the reverse direction of the same road going out of the junction. Therefore one needs to look at the position of the stop signs and find out which junction it really belongs to, and then only apply the cost to the affected edge. This is not implemented yet, so at the moment edges in both directions of roads with signs have costs added, which is faulty behavior.

TurnCostCalculator

When calculating the *costs* or *weights* for a *line* in the *line graph* it is the cost for the *source node/edge* plus the cost for the *turn*. This `TurnCostCalculator` helps with that. The calculations for this has been re-factored out to its own class as one can imagine wanting to include different properties when calculating the cost. Thus it would make sense to make this an interface and add different implementing classes, but this project just has this one implementing class for now, and therefore skipped the interface.

The inspiration for the calculations made by this calculator comes from [7], but not all factors in that paper are included here.

It is obvious that it is more costly to make a sharp turn, as one needs to decelerate coming in to the turn, and accelerate going out of the turn. The sharper the turn, the slower one needs to go. The deceleration and acceleration characteristics are properties of the routed vehicle. Also if one is coming from a lower ranking road category and is turning into a higher category, one needs to give way, which is also a cost.

EdgeRestriction

Restrictions for edges/roads can be somewhat complicated. Some regulates *general access*[1] with values such as *yes, no*, but in addition much more arbitrary values such as *permissive; designated; discouraged; customer*. Then other restricts access depending on the *vehicle type* such as banning *cars* but allowing *buses*. Then again, the restriction can depend on the *vehicle properties* such as *weight* or *width*. In some cases, such as *sump buster barrier*[2], it can ban access for a car, but only impose a cost on a bus. A road can also be tagged as *disused*, which is clear, but it is not so clear what to do with a road marked as *no-exit*.

An edge might no have a restriction by itself, but be part of a *turning restriction relationship*, so that one can not turn from one edge to another, although traffic is allowed on both edges. In addition, the specifications (see appendix A) said that *conditional restrictions* should be respected, that is restrictions that only apply for example at a certain time, a certain day of the week, for vehicles with certain properties or of a certain category, see figure 4.5.

The *conditional restrictions* has not been implemented yet, and the whole class is marked by being developed incrementally while discovering how many separate and complex parts of *OpenStreetMap* represents some kinds of restrictions.

[1] http://wiki.openstreetmap.org/wiki/Key:access
[2] http://wiki.openstreetmap.org/wiki/Tag:barrier%3Dsump_buster

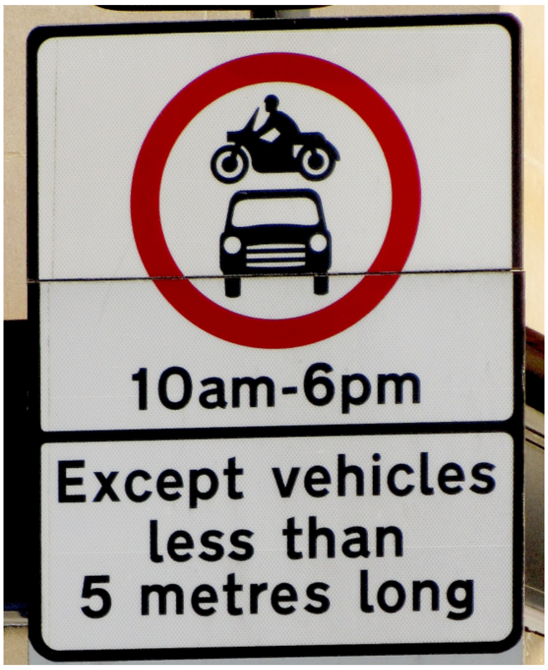


Figure 4.5: Conditional restrictions. [14]

4.2.5 `lgu`

The `lgu` package (see appendix listing C.6) is the *entry point* into the whole software module. The specification (appendix A) said that the module should be called from a function `get_directed_line_graph()`. This has not been written yet, so the entry point is by instantiating a `LineGraphUtility` object and call `get_line_graph()` on it, but it would be simple to write a wrapper to actually provide the specified function if needed.

The package is really only one class, `LineGraphUtility`, and how it works has been described in section 4.1.1.

4.2.6 `mapprovider`

This package (see appendix listing C.7), should contain sub-packages, as the `mapprovider` directory otherwise only contains an interface, `MapProvider`, and an exception class. The interface is the way to get map data from a source (such as a database) into the classes of the application.

There are two sub-packages in the project. One is `jsontest`, which in the initial phases of the project was used to load a small set of edges and vertices from a `json` file. It has been abandoned after loading from database was developed, but still hangs around.

The other sub-package is `postgis`, which is a map provider that uses a *PostGIS* database with the `postigs_topology` extension as the source for map data. This is where a lot of development has taken place during this project.

`postgis`

The `postgis` package uses the `libpqxx` to work with the *PostGIS* database. The `PostGisProvider` class gets passed in a `Topology` object to modify when asked for a *topology* or to set *restrictions and costs*. It also knows how to persist the *lines* and *nodes* of a *line graph* back to the database, which was desired functionality in the specification (see appendix A).

All the logic to work with the database and how to fill in the topology exists in this package. To make it more manageable, the `PostGisProvider` has four helper classes to actually perform the queries and handle the results from the database. They have names that describes their area of work: `CostQueries`; `LineGraphSaveQueries`; `RestrictionQueries`;

`TopologyQueries`. They are all *static* classes and cannot be instantiated, one can only call the methods statically.

Some remarks about those classes:

The `TopologyQueries` simply fetches the relevant data for vertices and edges. For the latter case, it also performs some calculations in the *SQL* query to calculate the geometric data.

The `LineGraphSaveQueries` creates a new schema and table and inserts some basic information about the *nodes* and *lines*.

The `RestrictionQueries` has to extract all the different information for *edge restrictions* (see section 4.2.4). It uses an inner class for *turning restrictions* to work with those queries and to extract `OsmTurningRestriction` data (see section 4.2.7) so those restrictions can be resolved. Turning restrictions are not really attributes of edges, but *relations* in the *OpenStreetMap*, and the `osm2pgsql` tool for importing *osm* data into a *PostGis* database does not really handle relations so they can be used straightforwardly[3]. Therefore some workarounds have been made: In the process of initializing the database on creation a `turning_restrictions` is created and a couple of custom *sql* functions are installed, that extract *turning restrictions* relations from the table `planet_osm_rels`, and parses what kind of restriction it is and the *osm ids* of the members (i.e. the edges and vertex involved). With those *ids* the involved *topology edges* are identified and stored as a string as the that is easier to make use of in the program than an array. The result are stored in the `turning_restrictions` table, and when running the `RestrictionQueries` for turning restrictions the topology ids are parsed and operation can continue.

4.2.7 `osm`

This package (see appendix listing C.8) deals with handling concepts and constants in *OpenStreetMap* data, such as enumerating the different categories of *accesses*[4], *barriers*[5], *highways*[6] and *vehicles*[7].

OsmTurningRestriction

In addition to those classes above, there is a class for dealing with the concept of *turning restrictions*, which are *relations* between *edges* and *vertices* in an *OpenStreetmap*. This class is an attempt to keep track of that information. In *OSM* a turning restriction is a relation of *(from $\rightarrow$ via $\rightarrow$ to)*. The 'via' part can be either a vertex (at a junction) or other edges, saying "travel from Here to There via roads This and That are not allowed". That kind of relationship is a lot trickier to represent, especially for this software module that only should build a *line graph* of the allowed turns, but has no routing information and thus cannot decide if a "via way" relation is allowed or not. It has therefore been disregarded in this project, and a routing application needs to decide that information some other way. The class `OsmTurningRestriction` has a field telling if it is a *via way* or a *via vertex* restriction.

4.2.8 `preparation`

Before anything else can be done, one needs to prepare the database. That means installing needed extensions to handle geometric and geographic data, and set up some tables and functions needed. Then one can add the map data to the database.

Appendix listing C.9 show the contents of this package. There is an `.sql` file for initializing extensions `postgis; postgis_topology; hstore` and installing functions for finding *turning restrictions*. And there is a `.sql` file to use when building the topology in advance. Then there is a file `LGU.style` which tells `osm2pgsql` which tags to create columns for in the tables, and which tags to ignore. Then there is also the original `.osm` files with map data for *Mikhailovsk* and *Partille*.

[3] http://wiki.openstreetmap.org/wiki/Osm2pgsql/schema
[4] http://wiki.openstreetmap.org/wiki/Key:access
[5] http://wiki.openstreetmap.org/wiki/Key:barrier
[6] http://wiki.openstreetmap.org/wiki/Key:highway
[7] http://wiki.openstreetmap.org/wiki/Key:vehicle

The way to prepare the database is shown in listing 4.4, which sets up a new database `mikh_db` for the *Mikhailovsk* map data.

```
$ # 1. Create database
$ createdb mikh_db -U tester

$ # 2. Install extensions and functions
$ psql -U tester -d mikh_db -f init_osm2pgsql_postgis_topology.sql

$ # 3. Import OSM data
$ # Flags:  -s    Slim mode (add data to db, do not build all in memory)
$ #         -k    Keep tags in `hstore` if not in own column
$ #         -S    Style-file to use
$ osm2pgsql -U tester -d mikh_db -s -k -S LGU.style mikhailovsk.osm

$ # 4. Building topology (optional)
$ psql -U tester -d mikh_db -f build_postgis_topology.sql
```

Listing 4.4: Preparing a database with map data.

4.2.9 `util`

This package (see listing in appendix C.10) contains a few utility classes: one for *logging* (using *Boost logging*) to be used where needed in the application; one for a *coordinate point* and one for *producing strings from current timestamp* which is used when building temporary topologies.

4.3 Development environment

Development of the project and the coding has taken place in *Eclipse Luna 4.4.2*. The build system is the default in Eclipse on Linux, generating *makefiles*.

Settings:

- Compiler flags:
  - `std=c++11`
  - `DBOOST_LOG_DYN_LINK`
  - `O0`
  - `g3`
  - `Wall`
  - `c`
- Linker flags:
  - `lboost_log`
  - `lboost_log_setup`
  - `lboost_thread`
  - `lboost_system`
  - `lpthread`
  - `lpqxx`
  - `lpq`

The coding was to follow a *coding standard* (see appendix A.7) which regulates the naming scheme and the layout of the files.

As for working with the database the main tool has been *pgAdmin3*.

5 Results

The software module developed in this project does not fulfill all requirements (see appendix A), in that it does not handle *conditional restrictions* at all, and not all implemented restrictions are handled correctly, see section 5.1 below.

But the software does build a *line graph* that can be fetched and stored in database for inspection with visual tools, see section 5.2 below.

The project has so far been about implementing things and have not had any focus on performance, but some performance tests have been run, see section 5.3.

5.1 Specification fulfillment

Table 5.1 shows how much of the specification that has been fulfilled.

Table 5.1: Fulfillment of specification.

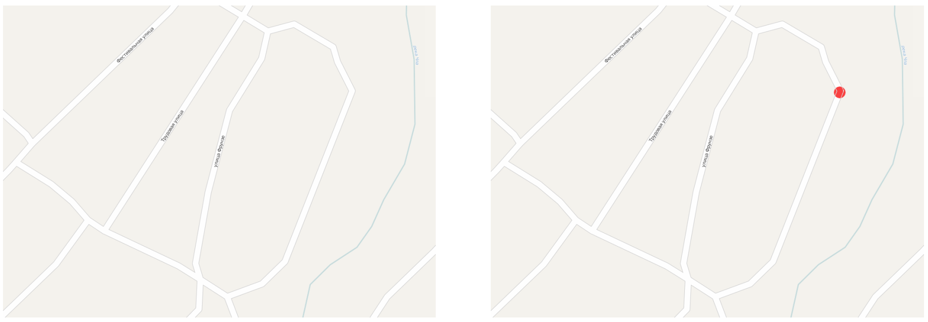
Section	Fulfills	Comment
1.2 Main use case		
1.2.1	X	As call to `LineGraphUtility::getLineGraph()`.
1.2.2	X	
1.2.3	X	
1.2.4	X	
1.3 Optional use case		
1.3.1	X	
1.3.2	X	
1.4 Functional requirements		
1.4.1		Lots of work remains to implement all restrictions.
1.4.2	X	
1.4.3	X	
1.4.4	X	Some small parts are hard coded.
1.5 Non-functional requirements		
1.5.1	X	Written in C++.
1.5.2	X	Did not find pgRouting really useable.
1.6 Testing requirements		
1.6	X	
1.7 Coding standard		
1.7	X	

5.2 Visual examination

Maps are easy to visualize, and a great number of tools exist to work with map data. Figure 5.1 shows a piece of a map over Mikhailovsk. In order to test if the handling of restrictions work, modified maps have been created. *JOSM*[1] is a tool for manipulating *OSM* data. In figure 5.1b it is indicated where a *bollard barrier* has been added in the middle of a road,

[1] https://josm.openstreetmap.de/

just to see if the restrictions work, and the new map is saved in its own `.osm` file, and a new database built for it.



(a) Original. (b) Modified with added barrier.

Figure 5.1: Map over part of Mikhailovsk. [15]

Using another tool, *QGIS*[2] can be used to load map data from for example a *PostGIS* database and viewed. In figure 5.2 the vertices and edges from the topology for that map has been layered on top of the image (with a slight misalignment). The topology is the same for both maps, it does not change with added barriers.

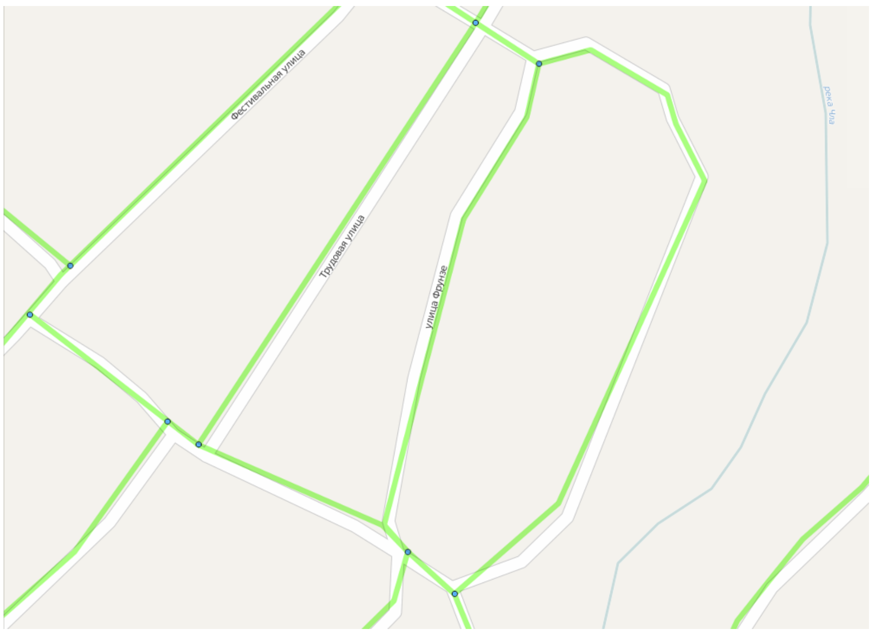


Figure 5.2: Edges (green) and vertices (blue) for the topology.

The interesting part is to see if the restriction has had any impact on the built *line graph*, see figure 5.3 for the original line graph, where the road is included in the line graph. It has a *node* in the middle and *lines* connecting to the adjacent *edges/nodes*.

Figure 5.4 shows the line graph after the restricting barrier has been added to the map. There one can see that the *edge* (road) has not been added as a *node* to the line graph, while all the other *lines* and *nodes* remain the same. This practically disables routing along that road.

[2]http://www.qgis.org

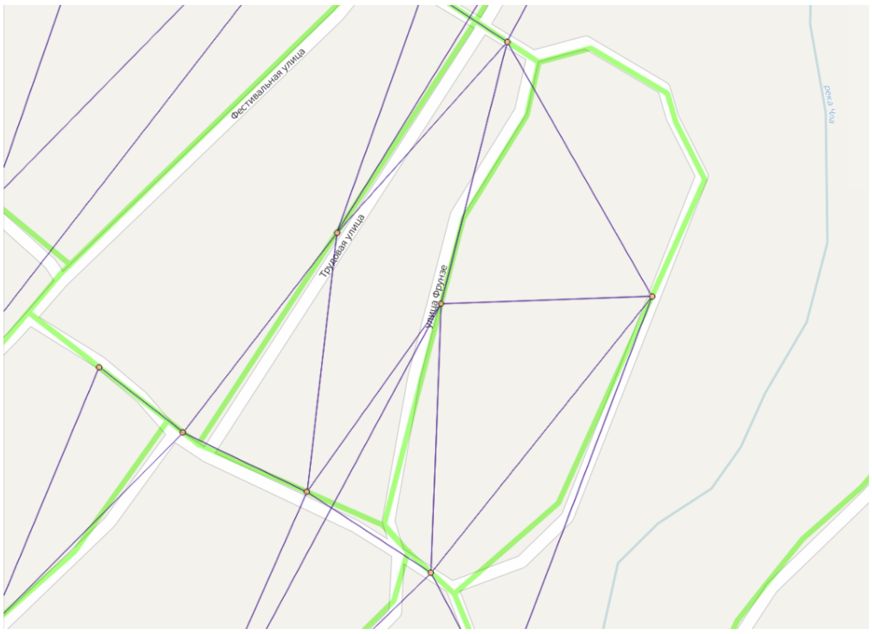


Figure 5.3: Original *line graph*. *Lines* in purple.

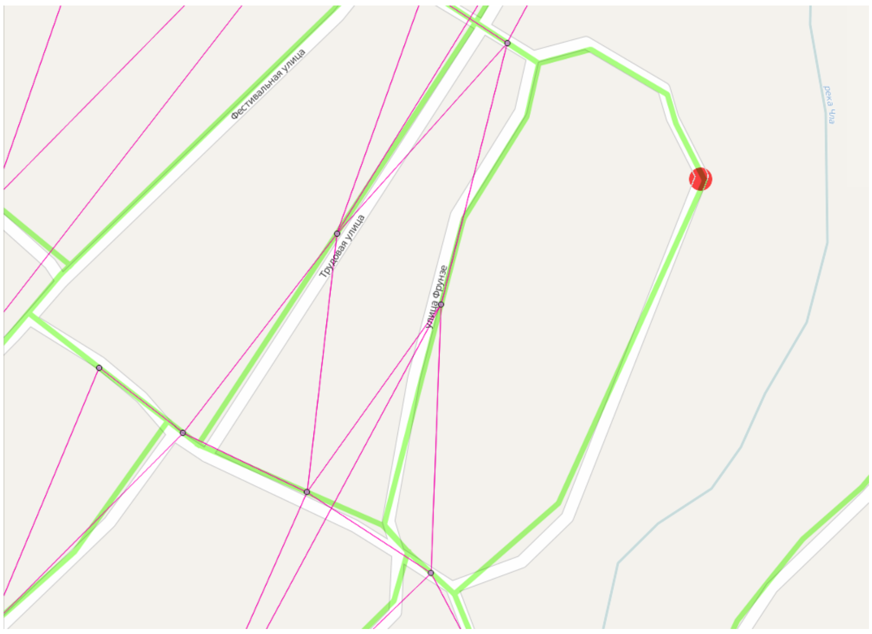


Figure 5.4: *Line graph* after added barrier. *Lines* in magenta.

5.3 Performance

There were *soft real time* requirements in the specification, but they were not specified more than that. But it is interesting so find out how long it takes to fetch a *line graph*, built on demand by the software module.

A few test cases were written in `LineGraphUtility_test.cc` that averages the number of *microseconds* it takes to instantiate a `LineGraphUtility` and fetch a *line graph*, over a given number of rounds.

The test runs on both a configuration with a pre-built topology, and a configuration that builds a temporary topology.

See table 5.2 for test results.

Table 5.2: Time in μs to fetch a *line graph*, with pre-built versus temporary topologies.

Test #		1	2	3	4	Sum
# of rounds		10	10	10	70	100
topology		avg (μs)	avg (μs)	avg (μs)	avg (μs)	avg (μs)
Mikhailovsk	normal	147859	149092	141782	133950	143171
	temporary	5026626	4924245	4917319	4875838	4936007
Partille	normal	180340	188405	179883	179978	182152
	temporary	10683194	10342420	10683873	10521535	10557756

The *size* (number of edges) and *order* (number of vertices) of the graphs are shown in table 5.3.

Table 5.3: Sizes of tested graphs

	Graph		Line graph	
	vertices	edges	nodes	lines
Mikhailovsk	654	1618	1618	4758
Partille	1645	2265	2265	5577

The results shows that, in order to meet *soft real time requirements* it is not possible to build temporary topologies at every instantiation of a `LineGraphUtility`, as the time increases dramatically. In the case of *Mikhailovsk*, the increase is from 0.14 s to 4.93 s, about 34 times as much. In the case of *Partille*, the increase is from 0.18 s to 10.55 s, nearly 58 times.

That fetching a line graph with a pre-built topology takes 0.15-0.2 s might fall within the requirements.

The test were conducted on a computer with 8 GB ram, processor Intel i7-4702MQ, running Linux Mint 17.1 with Linux kernel version 3.13.0-37-generic. The comiler flags were the same as for the rest of the project, i.e. no optimization.

6 Discussion

Presented below is my personal views of parts of the project and the outcome of it.

6.1 Research

6.1.1 Graph theory

Starting out on this project, I thought that one of the main obstacles would be no prior knowledge of *graph theory*, so I set out to allow for some time initially to get into the field. I am glad to have gained some fundamental knowledge of the area, but the time spent here could have been less.

6.1.2 Map routing

Reading about theory regarding map routing and graphs was really interesting, and a lot of research has been done in this area in later years. It initially gave me some ideas I thought I would like to try out, but once development got going, those theories vanished in favor of finding working solutions quickly.

6.1.3 Map data

OpenStreetMap is the source of map data for this project, and a lot of high quality projects. That puzzles me somewhat, because I have found it kind of messy. It is an *XML* application, but it has no official *schema*. That is, there is an informal consensus on which tags are good, but one can also make up ones own tags[1]. Another example of the messiness is the `maxspeed` tag, which can have the values *60; 50 mph; 10 knots*. That is, the default case is a unit of *km/h* and one can read the value as numeric. But one cannot be sure of that, because other units are allowed, and in that case one needs to parse the value as a string to find out which unit is used. It would surely have been better to let the unit be an attribute of the value, so one did not need to parse every value. In this project I decided to skip parsing, and just assume all values are *km/h*.

But, as said, a lot of good applications using *OSM* exists, see 2.4, so it is possible to work with. And it might also be unfair to say that *OpenStreetmap* is messy; it might be the case that it simply reflects the complex and difficult reality in the traffic, with lots of different rules and restrictions dependent on context and conditions.

6.1.4 Available applications

The fact that a lot of applications already existed, and some of them being *open source* and using *OpenStreetMap* as the the source for map data, made the direction of this project a little difficult. I proposed to the company that there are some good solutions out there that might just need some adaption to work, but they wanted their own thing. So the question for me was if I was to look at and copy features and concepts of those existing solutions anyway or just blindly go down my own path. In order to steer clear of issues with plagiarism I chose the latter, and that has surely impacted the project negatively. It would have been wiser to build upon the experience of others, developed through years.

6.2 Methodology

The main methodology for the project was supposed to be test driven (either BDD or TDD), but to be honest, most tests were written after the implementation of a feature, functioning

[1] `http://wiki.openstreetmap.org/wiki/Map_Features`

more as unit test, than driving the development. I don't think that has affected the outcome of the project negatively, it is more a matter which workflow feels best.

6.3 Design

Previously I have bee more into heavy design and modeling before starting coding, but in the last year I have tried to become more "agile", and start testing things out and be prepared to refactor and remodel when needed.

In this project perhaps it could have been good to design more, to have really thought through how the restrictions should work. On the other hand, a lot of the difficulties was discovered only when working on them, so it would be hard to have the full picture before. It is a balance in getting going and learning, and modeling before. What is clear, is that parts of the software as it stands now, should be re-modeled, specifically the *restrictions.*

6.4 Development

6.4.1 Coding standard

This was the first time I had to code to a standard. It was kind of awkward and unintuitive at first as it differs from my personal style, especially since having started to trying to practice *"Clean Code".* and have less comments and visual dividers in the file. But after a short time the style became pretty easy to use. I don't think I have followed the standard completely, but it was too lng to read and get into before beginning to code.

6.4.2 Memory management

I tried to avoid pointers and only use references, but that turned out to be clumsy, so at times I reverted to using raw pointers. Eventually, I found out that it would have been a lot better to use the smart pointers from C++11 (or even Boost), but I did not want to spend the time needed for learning how to use them and redo the memory management completely.

6.4.3 Tools

OSM conversion

I tested and looked at a number of tools for converting *OpenStreetMap* data to a *PostGIS* database, and the choice fell on *osm2pgsql.* I am not certain that it was the right choice, as it has is shortcomings when working with restrictions. Fortunately, the developed software module is flexible so one can write a new *MapProvider* if one decides to work with another tool, that uses a different approach.

Database

The *pqxx* library was easy and straight-forward to work with.

Boost

This was the first time for me to use *Boost.* I have only used small parts of the library: obviously the *graph* package, the *property_tree* for parsing *json* and the *logging* package. There are some tricky concepts, but also a lot of useful stuff. Getting into all the long names and templates takes some getting used to, but it was OK.

Catch test

I really enjoyed the *Catch* testing library; small and easy to use. It didn't play so nicely with *Eclipse CDT*, marking errors throughout in the editor, but good enough.

6.4.4 OpenStreetMap

Restrictions

Turning restrictions are *relations*, and *osm2pgsql* does not really handle relations, so a lot of parsing was needed. And in the case of *conditional restrictions* I have not found out how to work with them. *osm2pgsql* can be instructed to put tags into separate columns in the database, but with conditional restrictions the tags changes 'looks' and the only way to find them is by parsing the *hstore* column.

Also, the *restriction* class in the application is kind of messy. It could do with some remodeling, partly to clean up, and partly to make it more extensible to incorporate *conditions*. The *OSM* syntax for *conditional restrictions*[2] is shown in listing 6.1, and could work as a model for developing a more generic restrictions class.

```
<restriction-type>[:<transportation mode>][:<direction>]:conditional
  = <restriction-value> @ <condition>[;<restriction-value> @ <condition>]
```

Listing 6.1: Syntax of conditional restrictions in OpenStreetMap.

6.5 Documentation

A lot of the time of the project has also been devoted to documentation and writing this report. I took the opportunity to learn how to write a report in LaTeX, using the excellent web service www.sharelatex.com. It has been a pleasure, and it feels really good not to depend on the shaky features with cross-referencing in word processors.

6.6 Results

As the project does not fulfill all requirements and did not finish on time, it was not all that successful. The reason for not meeting the specification is that I ran out of time, partly due to the specification was supplied more than two weeks late, and partly due to the complexities with handling restrictions.

It would be possible to continue development, most on finding good ways to handle conditional restrictions. From my horizon, I still think that the best solution would be to adapt an existing solution that has been developed and refined by many people through many years. Perhaps using *OSRM* together with a *PostGIS* database as demonstrated here: https://www.mapbox.com/blog/osrm-using-external-data/. But I do not have any overview of the greater project, and what it is trying to accomplish.

This project shows that there exists really good products, and that rolling ones own is not trivial. What first seemed like a straightforward sequential piece of software turned out to be tangled in complex handling of restrictions.

[2] http://wiki.openstreetmap.org/wiki/Conditional_restrictions

Bibliography

[1] Keijo Ruohonen. *Graph Theory*. Lecture notes. Tampere University of Technology, 2013. URL: http://math.tut.fi/~ruohonen/GT_English.pdf (visited on 2015-03-24).

[2] Tero Harju. *Graph Theory*. Lecture notes. University of Turku, Departement of Mathematics, 2012. URL: http://users.utu.fi/harju/graphtheory/graphtheory.pdf (visited on 2015-04-09).

[3] Reinhard Diestel. *Graph Theory*. 4th Electronic Edition 2010, Corrected Reprint 2012. 2012. URL: http://diestel-graph-theory.com/ (visited on 2015-03-25).

[4] Daniel Delling et al. "Engineering Route Planning Algorithms". English. In: *Algorithmics of Large and Complex Networks*. Ed. by Jürgen Lerner, Dorothea Wagner, and KatharinaA. Zweig. Vol. 5515. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 117–139. ISBN: 978-3-642-02093-3. DOI: 10.1007/978-3-642-02094-0_7. URL: http://i11www.iti.uni-karlsruhe.de/extra/publications/dssw-erpa-09.pdf (visited on 2015-04-12).

[5] Hannah Bast et al. *Route Planning in Transportation Networks*. Tech. rep. MSR-TR-2014-4. Jan. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=207102 (visited on 2015-04-11).

[6] Daniel Delling et al. "Customizable Route Planning in Road Networks". working paper, submitted for publication. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=198358 (visited on 2015-04-05).

[7] Lars Volker. *Route Planning in Road Networks with Turn Costs*. Student research project, Universität Karlsruhe (TH), supervised by P. Sanders and D. Schultes. Universität Karlsruhe, 2008. URL: http://algo2.iti.kit.edu/documents/routeplanning/volker_sa.pdf (visited on 2015-04-03).

[8] "Efficient Routing in Road Networks with Turn Costs". In:

[9] Alexandros Efentakis et al. "Crowdsourcing turning restrictions for OpenStreetMap." In: *EDBT/ICDT Workshops*. 2014, pp. 355–362. URL: http://ceur-ws.org/Vol-1133/paper-56.pdf (visited on 2015-04-14).

[10] QGIS. *Topology*. URL: https://docs.qgis.org/2.2/en/docs/gentle_gis_introduction/topology.html (visited on 2015-10-05).

[11] Ittai Abraham et al. "HLDB: Location-based services in databases". In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. ACM. 2012, pp. 339–348. URL: http://research-srv.microsoft.com/pubs/172429/hldb_GIS12.pdf (visited on 2015-04-14).

[12] Ittai Abraham et al. *HLDB: Location-Based Services in Databases*. Tech. rep. MSR-TR-2012-59. June 2012. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=166857 (visited on 2015-04-14).

[13] Doug Gregor. *Bundled Properties*. URL: http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/bundles.html (visited on 2015-10-08).

[14] "Achadwik". *File:Length and time restriction 2.jpg*. [License: Creative Commons Attribution-ShareAlike 2.0]. 2011. URL: http://wiki.openstreetmap.org/wiki/File:Length_and_time_restriction_2.jpg (visited on 2015-10-15).

[15] OpenStreetMap contributors ©. *OpenStreetMap*. [License: Creative Commons Attribution-ShareAlike 2.0]. 2015. URL: https://www.openstreetmap.org/#map=18/45.13903/42.03324 (visited on 2015-10-22).

A Specification

The complete specifications from the company.

A.1 General

Line Graph Utility, LGU is a software utility which can poll a PostGIS database for a road network and builds a directed line graph from that. The directed line graph is stored in memory and the call to `get_directed_line_graph()` returns a directed line graph stored in a C++ *Boost* graph structure. The directed line graph is built based on the time of the day, road signs, traffic lights and other conditions.

A.2 Main use case

A.2.1 Call `get_directed_line_graph()` from C++ code.

A.2.2 LGU queries the PostGIS database and builds a graph from the road network.

A.2.3 LGU builds a directed line graph from the previous step (i.e. it converts nodes to edges and assigns weight to those edges based on road signs and other elements which are present in the nodes).

A.2.4 `get_directed_line_graph()` returns a directed line graph structure which is based on a C Boost graph structure to the function caller.

A.3 Optional use case

A.3.1 All main use case steps.

A.3.2 Write the resulting directed line graph to a separate heterogeneous table in the PostGIS database so that the graph can be viewed in QGis.

A.4 Functional requirements

A.4.1 LGU should take into account the following elements when building a directed line graph and calculating a weight for each edge: road signs (including time scheduling for those), traffic lights, road type (OSM road types), time of the day, road marking (i.e. separate lanes should be treated as separate edges), crossing and roundabouts slowdown, slopes and downhills, one way streets, road conditions, 'closed road' attribute.

A.4.2 LGU should take into account restricted turns in the road network when building a directed line graph; i.e. it should not create edges between newly created nodes in a line graph.

A.4.3 LGU should only take road signs and other conditions which are already present in the PostGIS database, the database is the only source of data for LGU.

A.4.4 LGU should store all its settings in a `settings.json` file.

A.5 Non-functional requirements

A.5.1 LGU should be written in C/C++; or, Boost.Python can be used

A.5.2 LGU can re-use architecture and code from the pgRouting software, which has a very similar structure. Namely it can re-use the steps 1 and 2 of the pgRouting's source code:

- A C module that uses a query is passed in Postgresql in order to build a line graph.
- C++ modules that convert it into a boost graph, and launch the routing.
- Return a result into psql server (this step is not required)

A.6 Testing requirements

LGU should be tested with a road network map built from 2 `.osm` files, `partille.osm` and `mikhailovsk.osm`.

A.7 Coding standard

Not actually written down in this document, but noted in an earlier conversation was that the company uses a *coding standard* [1] that must be followed.

[1] http://www.possibility.com/Cpp/CppCodingStandard.html

B UML Diagrams

Sequence and class diagrams of the software module.

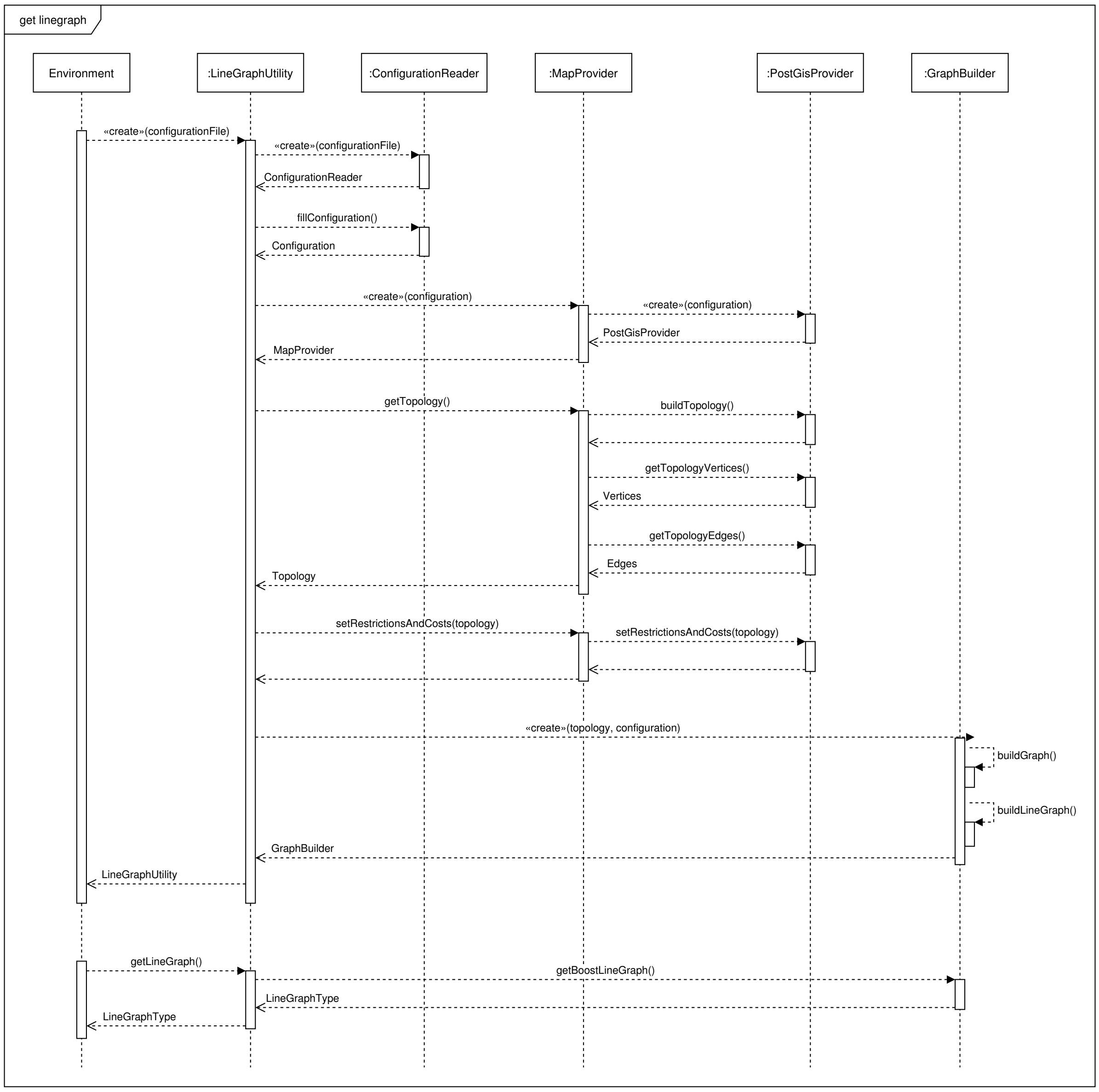


Figure B.1: Sequence diagram of main use case to get a line graph.

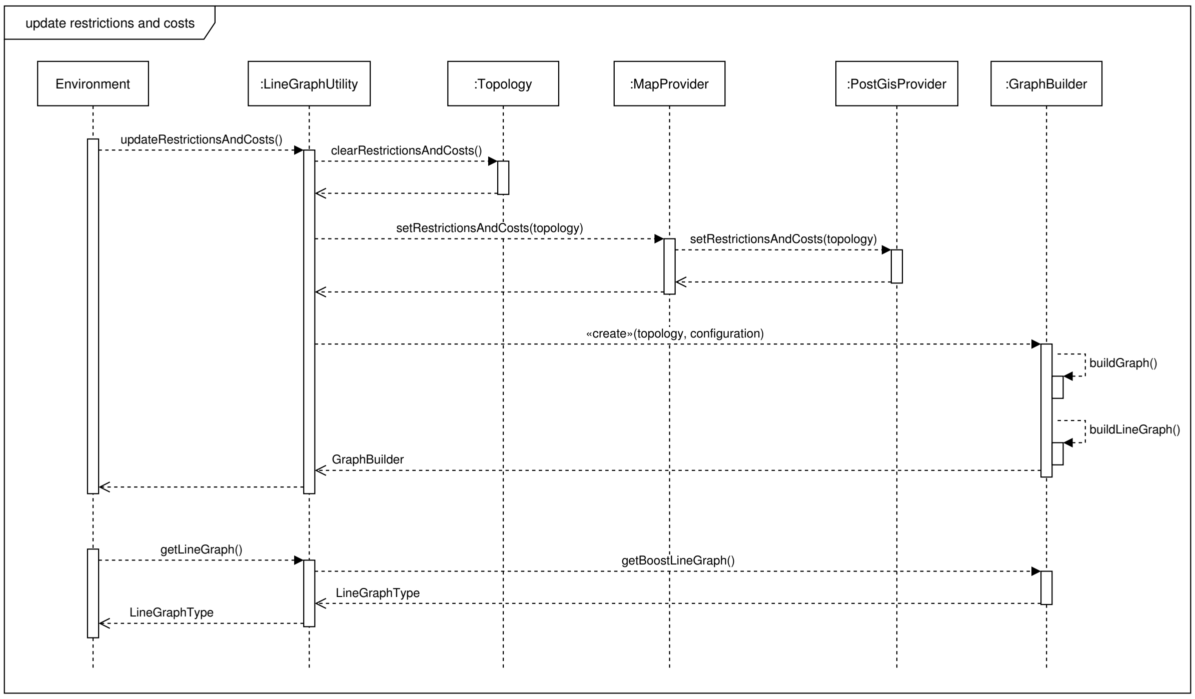

update restrictions and costs
Environment
:LineGraphUtility
:Topology
:MapProvider
:PostGisProvider
:GraphBuilder
updateRestrictionsAndCosts()
clearRestrictionsAndCosts()
setRestrictionsAndCosts(topology)
setRestrictionsAndCosts(topology)
«create»(topology, configuration)
buildGraph()
buildLineGraph()
GraphBuilder
getLineGraph()
getBoostLineGraph()
LineGraphType
LineGraphType

Figure B.2: Sequence diagram of updating costs and restrictions on a topology.

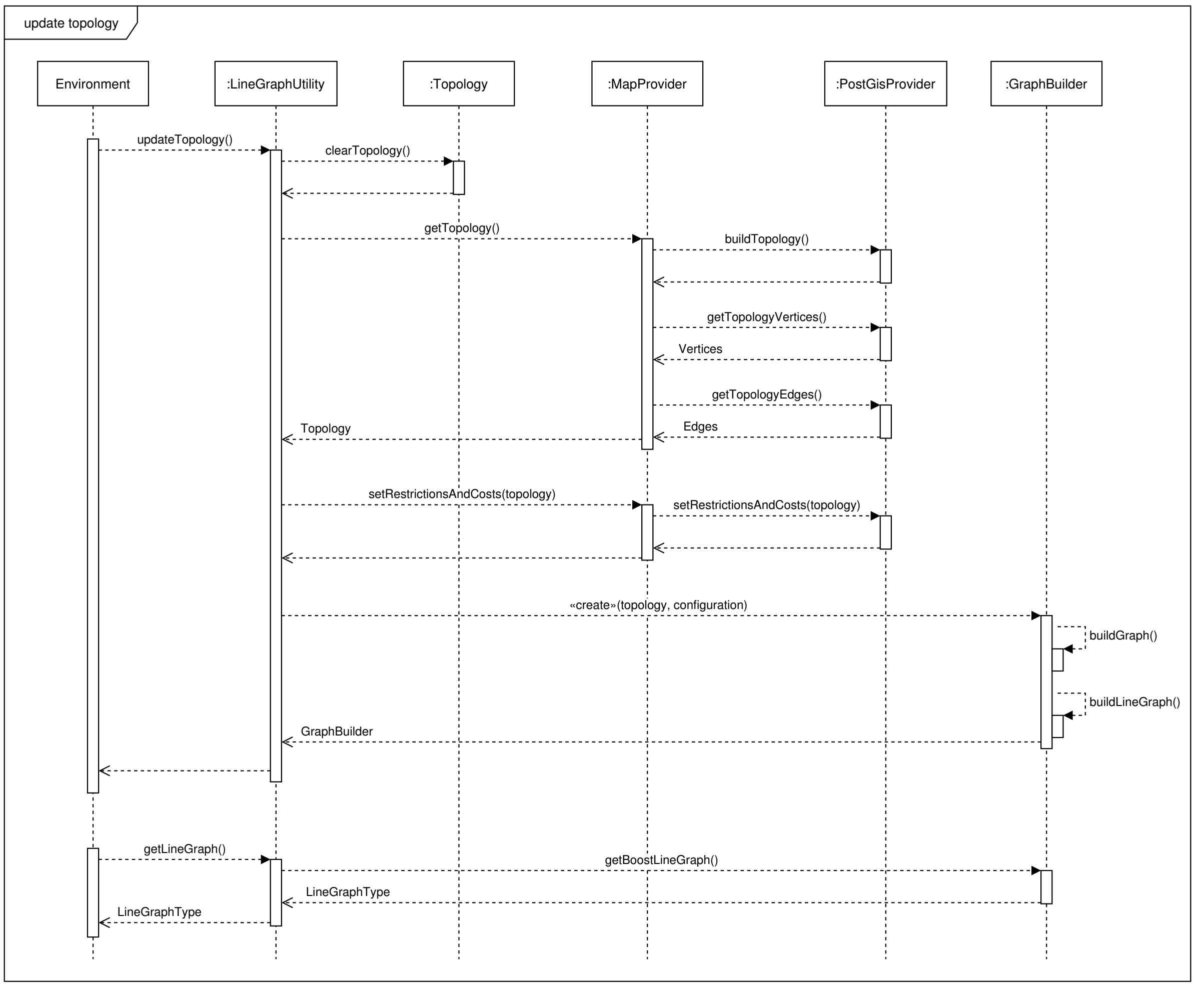


Figure B.3: Sequence diagram of updating the topology.

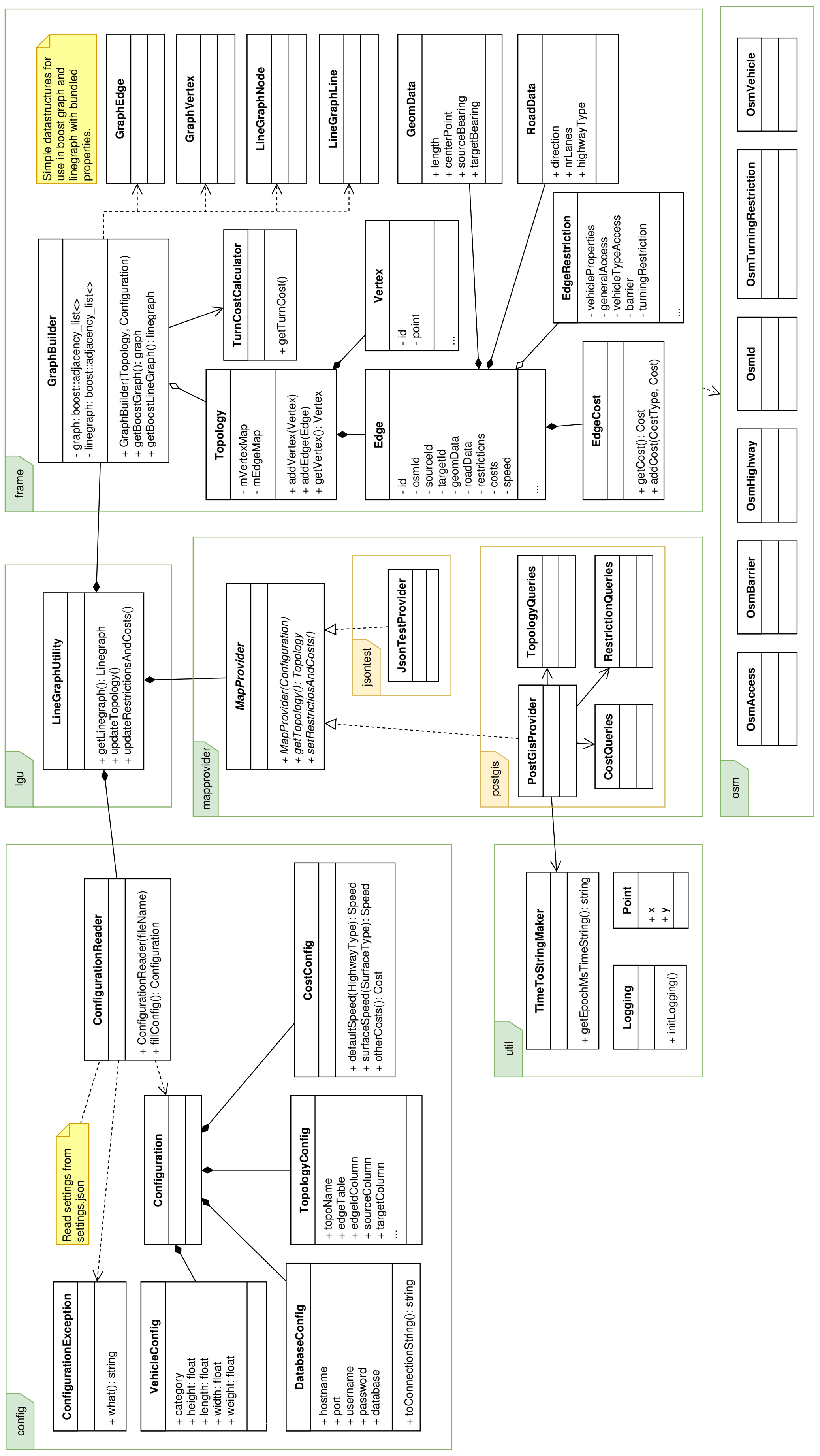


Figure B.4: Class diagram of the *line graph utility.*

C Directory listings

Contents of the directories in this project. The source code is available at https://bitbucket.org/jobe0900/exjobb/src.

C.1 root

```
/
├── catchtest/
├── config/
├── doc/
├── graph/
├── lgu/
├── mapprovider/
├── osm/
├── preparation/
├── util/
└── README.md
```

Listing C.1: Directory structure in root of the project

C.1.1 README.md

```
LineGraphUtility (lgu)
======================
This software module uses OpenStreetMap data to fetch topology, restrictions and
↪  costs, and uses them to build a Graph, which is converted to a LineGraph.

## State of the software
The software module does not fulfill the specification yet.

#### Working features
- Building graph and linegraph respecting some **edge** restrictions:
    - Turning restrictions.
    - General access restrictions.
    - Vehicle type specific restrictions.
    - Vehicle property restrictions (weight, height...).
- Some restrictions on edges.
- Turning restrictions via a node, not via other roads.
- Costs.

#### NOT implemented features
- Inclination, different speed uphill or down hill.
- Conditional restrictions.
- Turning restrictions that are not one-to-one, but one-to-many.
- Turning restrictions via ways (not via nodes).
- Parsing units, i.e. assuming all dimensions are meters and weight in metric
↪  tons and speed in km/h.

### Organization
The code is organized in folders (kind of "packages") to keep it modularized. The
↪  packages are:

- **`catchtest`**: The main for the testing framework.
- **`config`**: For configuration related code.
- **`graph`**: For code that is related to Graphs.
- **`lgu`**: The main entry point into this software.
- **`mapprovider`**: The package for code providing access to map data.
- **`osm`**: Classes representing some concepts in OpenStreetMap data.
- **`preparation`**: osm-files and sql-files and instructions on how to set up
 ↪  database.
- **`uml`**: For uml documentation.
- **`util`**: A few utility classes.

Each folder should have its own `README.md` that describes what the contents and
 ↪  the purpose of that package is. Each package should also have their own tests
 ↪  in a `catchtest` folder, and preferably an *exception class*.

### Building
Right now  all development has been in Eclipse, so it is just a standard Eclipse
 ↪  project with the makefiles that Eclipse has set up in the `Debug` folder. The
 ↪  file `catchtest/catchmain.cc` provides the entry point for the software
 ↪  module during testing.

#### Libraries
There was only need for linking with `-lpqxx` and `-lpq` (for connecting to the
 ↪  database) until *Boost logging* was included, at which point it also became
 ↪  necessary to link with `-lboost_log -lboost_log_setup -lboost_thread
 ↪  -lboost_system -lpthread`.

#### Testing
As mentioned, testing is done with [Catch](https://github.com/philsquared/Catch).
 ↪  Tests can be written BDD-style, and it is header only. A few quirks: some of
 ↪  the macro-keywords, most notably `REQUIRE`, is reported as an error in the
 ↪  Eclipse editor, but one can ignore that.

### Style
I have tried to follow the style given in [C++ Coding
 ↪  Standard](http://www.possibility.com/Cpp/CppCodingStandard.html).

### Design
I have deliberately tried to avoid passing pointers around, and rather pass in
 ↪  references as IN-OUT parameters. The idea is that the central LGU class has
 ↪  stack variables of `Graph`, `Topology` that gets filled in, rather than
 ↪  obtained as pointers to objects on the heap. This is to try to reduce risks
 ↪  of complicated memory handling, while not have too much copying of large
 ↪  objects.

### Logging
Boost logging was the last feature added, and is so far only used in the `Graph`
 ↪  class. It needs to be compiled and linked with a lot of libraries:

    -lboost_log -lboost_log_setup -lboost_thread -lboost_system -lpthread

The log produced is `lgu.log` in the top level of the project.
```

C.2 catchtest

```
/
└─catchtest/
  ├─catch.hpp
  ├─catchmain.cc
  ├─README.md
  └─testsettings/
    ├─mikhailovsk-original.json
    ├─mikhailovsk-original-temp.json
    ├─partille-original-temp.json
    ├─partille-original-temp.json
    ├─. . . ............................................ (10 more .json files)
    └─restrictions/
      ├─mikhailovsk-barrier_block.json
      ├─partille-highway_traffic_signals.json
          └─. . . .................................... (17 more .json files)
```

Listing C.2: Files in /catchtest

C.2.1 catchtest/README.md

~~~
CATCH
=====
This project uses [Catch](https://github.com/philsquared/Catch) for testing. It
 ↪  allows for writing tests BDD-style.

It doesn’t play really nicely with Eclipse, as Eclipse’s editor marks `REQUIRE`
 ↪  as errors, so the project has a  lot of error markers throughout, without any
 ↪  real errors. But the Catch way of testing is nice, so it is worth is. And
 ↪  Eclipse flags a lot of errors for standard c++11 features as well...

When writing `SCENARIO`s or `TESTCASE`s one can tag those, which makes it easy to
 ↪  test small parts of the code. After building you can modify the Eclipse `Run
 ↪  Configuration` (or write on the command line) to only run those tests.

Example:

```cpp
SCENARIO ("Testing this module but not other", "[moduletag]")
{
    GIVEN ("a")
    {
        WHEN ("b")
        {
            THEN ("c")
            {
                REQUIRE (c)
            }
        }
    }
}
```

To specify which test to run, go to `Run` > `Run Configurations...`, select the
 ↪  `Arguments` tab and in `Program arguments` write the tag, e.g. `[moduletag]`,
 ↪  click `Apply` and `Run`.
~~~

```
A useful flag to add to the program arguments when running tests is `-s` to have
↪  print out of every step, else you only get the final report of how many
↪  scenarios have run.
```

C.3 config

```
/
└─config/
   ├─catchtest/
   │  └─ConfigurationReader_test.cc
   ├─Configuration.cc
   ├─Configuration.h
   ├─ConfigurationException.h
   ├─ConfigurationReader.cc
   ├─ConfigurationReader.h
   ├─CostConfig.h
   ├─DatabaseConfig.h
   ├─README.md
   ├─TopologyConfig.h
   └─VehicleConfig.h
```

Listing C.3: Files in /config

C.3.1 config/README.md

```
Configuration
=============

Configurations are set in the file `settings.json`. The different parts of the
↪  configuration is:

- Database
- Topology
- Vehicle
- Access
- Restrictions
- Costs

If one wishes to edit the setting, one can freely add and remove objects in
↪  _braces_ (`[` and `]`), for example if one wishes to edit which values for a
↪  tag inflicts a cost. But the strings in _curly braces_ (`{` and `}`) are keys
↪  that must exist, but the values for the keys can of course be edited.


Database
--------

Configuration for connecting to the database holding map data. The expected keys
↪  and values are:

- **"host"**:
    - *hostname* (e.g. `"localhost"`) or
    - *ip-address* (e.g. `"127.0.0.1"`).
- **"port"**:
    - *portnumber* (e.g. `5432`).
- **"username"**:
    - *username* (e.g. `"tester"`).
- **"password"**:
    - *password* (e.g. `"tester_pass"`).
- **"database"**:
    - *database name* (e.g. `"db_name"`).


Topology
--------

Configurations for building or reading topology from a database. Might have
↪  different meanings depending on which *MapProvider* are used. Topologies can
↪  be pre-built, or they can be generated each time, depending on the settings
↪  in `"build_topo"`. It is also possible to define a simple json test file, for
↪  testing simple topologies.

- **"provider"**:
    - *name* of *MapProvider*
        - `"postgis"` when using `postgis_topology` for building topologies.
        - `"pgrouting"` when using `pgrouting` for building topologies.
        - `"jsontest"` for simple json test topology.

- **"postgis"**:
    - **"topo_name"**:
        - *basename* for pre-built topologies (e.g. `"test"`), combined with
        ↪  `roads_prefix` and `topo_prefix` for actual names such as
        ↪  `"highways_test"` and `"topo_test"`.

    - **"roads_prefix"**:
        - *prefix* to add to `topo_name` (e.g. `"highways"`) for table of roads
        ↪  network, see above.

    - **"schema_prefix"**:
        - *prefix* to add to `topo_name` (e.g. `"topo"`) for schema with topology
        ↪  data when using `postgis_topology`, see above.

    - **"build"**:
        - **"temp_topo_name**:
            - `""` (*empty*) if not building temporary topologies.
            - `"epoch_ms"` for adding a string with the count of milliseconds
            ↪  since "Epoch" as the `topo_name`.

        - **"srid"**:
            - *number* identifying which projection to use.
                - `900913` for geometric metrical projection, unit meters.
                - `4326` for geographic spherical projection, unit degrees.

        - **"tolerance"**:
            - *snapping* of nodes in unit of projection when building topology,
            ↪  e.g. 1.0 for srid 900913, or 0.001 for srid 4326.

    - **"edge"**:

        - **"table"**:
            - *name* of the table with topology edges (e.g. `"edge_data"`), with
            ↪  column for id, source, target and geometry.
```

~~~
    - **"id_col"**:
      - *name* of the column in edge table with id of edges (e.g.
       ↪  `"edge_id"`).

    - **"source_col"**:
      - *name* of the column in edge table with vertex id of **source** of
       ↪  edge (e.g. `"start_node"`).

    - **"target_col"**:
      - *name* of the column in edge table with vertex id of **target** of
       ↪  edge (e.g. `"end_node"`).

    - **"geom_col"**:
      - *name* of the column in edge table with geometry of edge (e.g.
       ↪  `"geom"`).

  - **"vertex"**:
    - **"table"**:
      - *name* of the table with topology vertices (e.g. `"node"`), with
       ↪  column for id and geometry.

    - **"id_col"**:
      - *name* of the column in vertex table with id of vertices (e.g.
       ↪  `"node_id"`).

    - **"geom_col"**:
      - *name* of the column in vertex table with geometry of vertex (e.g.
       ↪  `"geom"`).

- **"pgrouting"**:
  - TODO.

- **"jsontest"**:
  - **"test_file"**:
    - `""` (*empty*) if not using *json-test provider*.
    - *filename* to a json test-file (e.g. `"test.json"`) looking like:

  ```json
  {
    "vertices": [
      [1,2,0],
      [2,2,1]
    ],
    "edges": [
      [1,1,2,0]
    ]
  }
  ```

    where each row in `vertices` are `[id,x,y]` and each row in `edges` are
     ↪  `[id, source vertex id, target vertex id, direction]`. Values for
     ↪  `direction` is `0 = BOTH`, `1 = FROM_TO`, `2 = TO_FROM`.


Vehicle
-------
~~~

```
Configuration about the vehicle to route through the topology. Information might
 ↪  be needed to take restrictions in account.

- **"category"**:
    - *name* of OSM category of the vehicle. [OSM
     ↪  Access](http://wiki.openstreetmap.org/wiki/Key:access). (E.g.
     ↪  "motorcar"). Definition of the category must state dimensions as below.
- **_"category_name"_**:
    - *height* of vehicle in meters.
    - *length* of vehicle in meters.
    - *width* of vehicle in meters.
    - *weight* of vehicle in tons.
    - *maxspeed* of vehicle in km/h.
    - *acceleration* is the time it takes from 0 to 100 km/h.
    - *deceleration* is the time it takes from 100 to 0 km/h.


Access
------

- **"allow"**:
    List of which values for tag `access` that permits access. Those values for
     ↪  `access` that are not listed here are considered to restrict access.


Restrictions
------------

- **"barriers"**:
    - List of which values for `barriers` that restricts access. Those values not
     ↪  listed are assumed to allow access.


Cost
----

Configuration relating to costs when routing through the graph.

- **"default_speed"**:
    - each road category has default speeds when none is specified. [OSM default
     ↪  speeds](http://wiki.openstreetmap.org/wiki/OSM_tags_for_routing/Maxspeed).
     ↪  Most roads have two speeds, `high` and `low`, which differentiate the
     ↪  speeds inside and outside of a town. `living_street` is always inside so
     ↪  only the low is important. `motorway` is really the `high` number, and
     ↪  the `low` number is the speed on the links (ramps). It is not trivial to
     ↪  find out if a road is inside or outside of that area, so for this
     ↪  application which is meant to be used for routing in urban areas (?), the
     ↪  `low` number is assumed for all cost calculations.
- **"surface"**:
    - each surface type is associated with a max speed in km/h over which one
     ↪  should not drive.
- **"barriers"**:
    - this is a list of which barriers causes a slow-down, and the number of
     ↪  seconds it is  probable it takes to pass.
- **"highway"**:
    - a list of values for the `highway` tag that can mean a time cost in seconds,
     ↪  such as zebra crossings, bus stops, stop or give way sign, and more.
- **"railway"**:
    - a list of values for the `railway` tag that can mean a time cost in seconds,
     ↪  such as a crossing between railway and highway (`level_crossing`).
- **"public_transport"**:
    - a list of values for the `public_transport` tag that can mean a time cost in
     ↪  seconds, such as a `stop_position` if one thinks it is suitable to slow
     ↪  down when passing a bus stop.
- **"traffic_calming"**:
    - a list of traffic calming objects and their time costs in seconds.
```

C.4 doc

```
/
└─ doc/
   ├─ report/
   │  ├─ jobe0900_report.zip ........................................ LaTeX original
   │  ├─ jobe0900_report.pdf
   │  └─ README.md
   ├─ uml/
   │  ├─ ............................................. .xml, .svg, .pdf diagram files
   │  └─ README.md
   └─ README.md
```

Listing C.4: Files in /doc

C.4.1 doc/README.md

```
Documentation
=============

This directory contains a directory for the project's report, and a directory for
 ↪  the UML diagrams.
```

C.4.2 doc/report/README.md

```
Report
======

The originals for the project's report. Written in
 ↪  [ShareLaTeX](https://www.sharelatex.com).
```

C.4.3 doc/uml/README.md

```
UML
===

Class and sequence diagrams to get an idea of the concepts, although not 100%
 ↪  accurate.

Diagrams are created in `draw.io`, exported as  `.svg` files, opened in
 ↪  `Inkscape` and saved as `.pdf` files (for usage in report).
```

C.5 graph

```
/
└─ graph/
   ├─ catchtest/
   │  ├─ EdgeCost_test.cc
   │  ├─ GraphBuilder_test.cc
   │  ├─ RestrictionsAndCosts_test.cc
   │  ├─ Topology_test.cc
   │  └─ TurnCostCalculator_test.cc
   ├─ Cost.h
   ├─ Edge.cc
   ├─ Edge.h
   ├─ EdgeCost.cc
   ├─ EdgeCost.h
   ├─ EdgeRestriction.cc
   ├─ EdgeRestriction.h
   ├─ GraphBuilder.cc
   ├─ GraphBuilder.h
   ├─ GraphException.h
   ├─ README.md
   ├─ RestrictionsException.h
   ├─ Speed.h
   ├─ Topology.cc
   ├─ Topology.h
   ├─ TopologyException.h
   ├─ TurnCostCalculator.cc
   ├─ TurnCostCalculator.h
   ├─ Vertex.cc
   └─ Vertex.h
```

Listing C.5: Files in /graph

C.5.1 graph/README.md

```
graph
=====

The `graph` package consists of classes for representing graphs.

## GraphBuilder
The GraphBuilder is responsible for buildign graphs and linegraphs. It takes a
↪  `Topology` and a `Configuration`and uses them for building a `graph` and
↪  `linegraph` based on [Boost adjacency
↪  lists](http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/adjacency_list.html).
↪  The `GraphBuilder` class holds several `maps` that connects the original
↪  Edges and Vertices to those used internally in the Boost graphs, so that it
↪  is possible to backtrack information about those elements. The internal Boost
↪  types keeps some properties
↪  ["bundled"](http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/bundles.html),
↪  instead of as "interior" properties.

The ordinary `graph` is a directed graph that connects the _**vertices**_ and
↪  _**edges**_ from the topology. The `linegraph` transforms that graph to an
↪  edge-based graph that turns the graph's edges into _**nodes**_ in the
↪  linegraph, and those edges are connected with _**lines**_.
```

```
### Topology
`Topology` is a class holding `Edges` and `Vertices` for the topology fetched
 ↪  from the `MapProvider`. It simply states which `Vertices` are connected by
 ↪  which `Edges`, without any costs or restrictions or directions. When created
 ↪  it validates that the `source` and `target` Vertices of the Edges actually
 ↪  exists in the topology.

### Edge
The Edge holds some relevant data from the topology. It has an `id`, and a field
 ↪  for which the original `osm_id` was before building the database topology. It
 ↪  also holds id to `source` Vertex and id to `target` Vertex, some data about
 ↪  the geometry and the "road" such as number of lanes, a structure for costs
 ↪  and optionally for restrictions.

### EdgeCost
The cost for travel among an edge is the number of seconds it takes. The base for
 ↪  this calculation is of course dependent on the length of the edge, and the
 ↪  speed. The speed an be set as an explicit `maxspeed` restriction, or by
 ↪  looking up the configuration for a `surface` if such is stated, else the
 ↪  speed is found by a look up for the default speed for the "highway type"
 ↪  (road category).

The travel time cost can than be modified by barriers, speed bumps, traffic
 ↪  lights ... on the edge (or points that can be applied on the edge).

### EdgeRestriction
The `EdgeRestriction` keeps track of restrictions that can be imposed on
an edge. Those restrictions are:

- **Vehicle properties**: weight, height, length, width, maxspeed.
- **General access**: [OSM wiki for
 ↪  access](http://wiki.openstreetmap.org/wiki/Key:access).
- **Vehicle type access**: as for General access, but specified for a category of
 ↪  vehicles, such as `motorcar` or `goods`.
- **Barrier**: if the edge is blocked with some kind of barrier.
- **Turning restrictions**: [OSM wiki for turn
 ↪  restrictions](http://wiki.openstreetmap.org/wiki/Relation:restriction).
- **Disused**: if the edge (road) is marked as no longer in use.
- **NoExit**: if the edge has no exit, it should not be used for building a
 ↪  linegraph.

(Turn restriction via other edges and not just via a vertex are difficult. At the
 ↪  time when converting the topology to a line graph it is impossible to have the
 ↪  relevant information. The solution is to set a flag on the Edge that there
 ↪  exist a VIA_WAY restriction that must be taken into account when routing, and
 ↪  the routing module must look up and make its own decisions somehow.)

### Vertex
The Vertex class is simple with just an `id` and a `point` location.

### TurnCostCalculator
This is a static helper class that assist in calculating the cost of making
 ↪  turns, when transforming the _graph_ into a _linegraph_. The cost for making
 ↪  turns are dependent on the angle of the turn, the category of the roads and
 ↪  the size of the vehicle. Other factors can be added.
```

```
### Exceptions
`GraphException` is the main public exception to be thrown from this package.
↪  `RestrictionsException` and `TopologyException` are thrown when building
↪  those classes, but not as exposed externally.
```

C.6 lgu

```
/
└─lgu/
   ├─catchtest/
   │  └─LineGraphUtility_test.cc
   ├─LineGraphUtility.cc
   ├─LineGraphUtility.h
   ├─LineGraphUtilityException.h
   └─README.md
```

Listing C.6: Files in /lgu

C.6.1 lgu/README.md

```
Line Graph Utility
==================

The main class in this utility.

Given a configuration file it picks a `MapProvider` and fetches a `Topology`,
↪  which is passed to the `GraphBuilder` along with the `Configuration`. The
↪  goal is to fetch a linegraph that is built according to the data found in the
↪  database and the configuration settings found in the configuration file.

A requirement for this utility was to be able to update data in the database
↪  which means this utility can also be requested to re-read the topology if
↪  there has been a change to them, or the restrictions and costs if there has
↪  been a change to them.
```

C.7 mapprovider

```
/
└─ mapprovider/
   ├─ catchtest/
   │  └─ MapProvider_test.cc
   ├─ jsontest/
   │  ├─ catchtest/
   │  │  ├─ JsonTestProvider_test.cc
   │  │  ├─ jsontest-settings.json
   │  │  └─ test-topology.json
   │  ├─ JsonTestProvider.cc
   │  ├─ JsonTestProvider.h
   │  └─ README.md
   ├─ postgis/
   │  ├─ catchtest/
   │  │  └─ PostGisProvider_test.cc
   │  ├─ CostQueries.cc
   │  ├─ CostQueries.h
   │  ├─ LineGraphSaveQueries.cc
   │  ├─ LineGraphSaveQueries.h
   │  ├─ PostGisProvider.cc
   │  ├─ PostGisProvider.h
   │  ├─ RestrictionQueries.cc
   │  ├─ RestrictionQueries.h
   │  ├─ TopologyQueries.cc
   │  └─ README.md
   ├─ MapProvider.h
   ├─ MapProviderException.h
   ├─ README.md
   └─ README.md
```

Listing C.7: Files in /mapprovider

C.7.1 mapprovider/README.md

```
MapProvider
===========

The `mapprovider` package exists to implement different classes that can provide
 ↪  access to OpenStreetMap data.

#### Background
There exists several solutions to import OpenStreetMap data into a database, and
 ↪  the different solutions all creates different schemas and tables. To keep the
 ↪  flexibility to change how we get the OSM data, the `mapprovider` exists to
 ↪  provide an abstract interface that providers must implement.

#### `jsontest`
The `JsonTestProvider` is a small map provider that was implemented to be able to
 ↪  read in a small set of well-known edges and vertices (such as the [pgRouting
 ↪  sample
 ↪  data](http://docs.pgrouting.org/dev/doc/src/developer/sampledata.html)), to
 ↪  be used under development of the `Graph` class.

#### `postgis`
```

```
The `PostGisProvider` exists for working with topologies built with the
↪  `postgis_topology` extension. This is the course taken during development of
↪  the `LineGraphUtility` so far. (Using `pgRouting` seemed to be useful only
↪  for building topology and not to get access to the other map data, and
↪  `osm2po` is not open source. But with the `MapProvider` interface it is
↪  possible to implement another if desirable.)

The `PostGisProvider` uses classes `TopologyQueries` and `RestrictionQueries` for
↪  querying the database.

### Exceptions
Each subpackage throws its own exception: `MapProviderException`, and
↪  `PostGisProviderException`.
```

C.7.2 mapprovider/postgis/README.md

```
PostGisProvider
===============

This is a concrete class that implements the MapProviders interface. It fetches
↪  map data from a PostGis database where the OSM data has been imported with
↪  `osm2pgsql`, and topology has been built with the `postgis_topology`
↪  extension, converts it to valid `Vertex` and `Edge` types and stores them in
↪  a `Topology`.

The handling of the queries are factored out in different static classes:
- `CostQueries` for handling costs.
- `RestrictionQueries` for handling restrictions.
- `TopologyQueries` for handling topology related stuff.
- `LineGraphSaveQueries` for persisting a LineGraph back to the database.
```

C.7.3 mapprovider/jsontest/README.md

~~~
JsonTest
========

This package exists for the sole purpose of testing in the initial stages of
↪  development, so that we easily could read in a small set of well-known edges
↪  and vertices.

But the test is now commented out, as it has not been updated to keep up with
↪  advances in the development.

Format of test file in json is:

```json
    {
        "vertices": [
            [1,2,0],
            [2,2,1]
        ],
        "edges": [
            [1,1,2,0]
        ]
    }
```
~~~

```
where each row in `vertices` are `[id,x,y]` and each row in `edges` are `[id,
↪  source vertex id, target vertex id, direction]`. Values for `direction` is
↪  `0 = BOTH`, `1 = FROM_TO`, `2 = TO_FROM`.
```

C.8 osm

```
/
└─ osm/
   ├─ catchtest/
   │  ├─ OsmAccess_test.cc
   │  ├─ OsmBarrier_test.cc
   │  ├─ OsmHighway_test.cc
   │  ├─ OsmTurningRestriction_test.cc
   │  └─ OsmVehicle_test.cc
   ├─ OsmAccess.cc
   ├─ OsmAccess.h
   ├─ OsmBarrier.cc
   ├─ OsmBarrier.h
   ├─ OsmException.h
   ├─ OsmHighway.cc
   ├─ OsmHighway.h
   ├─ OsmId.cc
   ├─ OsmId.h
   ├─ OsmTurningRestriction.cc
   ├─ OsmTurningRestriction.h
   ├─ OsmVehicle.cc
   ├─ OsmVehicle.h
   └─ README.md
```

Listing C.8: Files in /osm

C.8.1 osm/README.md

```
OSM
===

OpenStreetMap related classes and constants are placed in this package.


Relations
---------

There is no really easy way to get to relations if data has been imported with
↪  osm2pgsql. Best chance is to import in "slim mode" (with flag `-s`) and look
↪  through table `planet_osm_rel` and search the column `tags` for
↪  `restriction`. Then parse the `members` column for members of the relation
↪  and their roles.

The TurnRestriction class could be smarter with handling turn either via `nodes`
↪  or `ways` but it is not implemented yet.
```

C.9 preparation

```
/
└─preparation/
   ├─restrictions/
   │  └─. . . ..............................osm-files modified with added restrictions
   ├─build_postgis_topology.sql
   ├─init_osm2pgsql_postgis_topology.sql
   ├─LGU.style
   ├─mikhailovsk.osm
   ├─partille.osm
   └─README.md
```

Listing C.9: Files in /preparation

C.9.1 preparation/README.md

```
Preparing database
==================

The preparation differs depending on how you import the map data and build the
↪  topology. So far in this project import has been done with `osm2pgsql` and
↪  topology built with `postgis_topology`.


Preparation for `osm2pgsql` and `postgis_topology`
-----------------------------------------------------

To prepare the database for this software module when using `osm2pgsql` as
↪  importer of OpenStreetMap data into the database, we need to install
↪  extensions for:

- `postgis`
- `postgis_topology`
- `hstore`

and a couple of custom functions for finding turning restrictions:

- function `find_topo_edges_at_turning_restriction()`
- function `find_osm_turning_restrictions()`

The steps to follow are (assuming `mikhailovsk.osm` as source for OpenStreetMap
↪  data, and `tester` as a user with administrative rights in database, and
↪  `mikh_0530` as name of database):

### 1. Create database

    $ createdb mikh_0530 -U tester

### 2. Install extensions and functions

    $ psql -U tester -d mikh_0530 -f init_osm2pgsql_postgis_topology.sql

### 3. Import OSM data

    $ osm2pgsql -U tester -d mikh_0530 -s -k -S LGU.style mikhailovsk.osm
```

```
This uses the tool `osm2pgsql` to parse the osm-file into database tables.
The flags are

- `-s` slim, keeping extra tables.
- `-k` keeping tags in `hstore` if not in their own column.
- `-S` Style file, configuring which tags to have columns or not.

### 4. Building topology

This step is optional. It should be efficient and safe to build the topology once
 ↪  and for all after importing as differing conditions and temporary closures
 ↪  could be specified with costs and restrictions instead of via topology. But
 ↪  one can also configure the tool to build topology on each call, see the
 ↪  `configuration` package.

    $ psql -U tester -d mikh_0530 -f build_postgis_topology.sql

This step creates a table `public.highways_lgu` and adds a new schema called
 ↪  `topo_lgu` which contains tables for the topology.


Test databases
--------------
During testing different databases has been tested, they were created so:

#### `mikh_style`

    $ createdb mikh_style -U jonas
    $ psql -U jonas -d mikh_style -c "CREATE extension postgis;"
    $ psql -U jonas -d mikh_style -c "CREATE extension postgis_topology;"
    $ psql -U jonas -d mikh_style -c "CREATE extension hstore;"
    $ psql -U jonas -d mikh_style -c "SET search_path=topology,public
    $ osm2pgsql -U jonas -d mikh_style -s -k -S new.style mikhailovsk.osm
    $ psql -U jonas -d mikh_style -c "CREATE TABLE highways_test AS SELECT * FROM
     ↪  planet_osm_line WHERE highway IS NOT NULL;"
    $ psql -U jonas -d mikh_style -c "SELECT topology.CreateTopology('topo_test',
     ↪  900913);"
    $ psql -U jonas -d mikh_style -c "SELECT
     ↪  topology.AddTopoGeometryColumn('topo_test', 'public', 'highways_test',
     ↪  'topo_geom', 'LINESTRING');"
    $ psql -U jonas -d mikh_style -c "UPDATE highways_test SET topo_geom =
     ↪  topology.toTopoGeom(way, 'topo_test', 1, 1.0);"

#### `mikh_0522`

    $ createdb mikh_0522 -U jonas
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION postgis;"
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION postgis_topology;"
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION hstore;"
    $ osm2pgsql -U jonas -d mikh_0522 -s -k -S LGU.style mikhailovsk.osm
    $ psql -U jonas -d mikh_0522 -c "CREATE TABLE highways_test AS SELECT * FROM
     ↪  planet_osm_line WHERE highway IS NOT NULL;"
    $ psql -U jonas -d mikh_0522 -c "SELECT topology.CreateTopology('topo_test',
     ↪  900913);"
    $ psql -U jonas -d mikh_0522 -c "SELECT
     ↪  topology.AddTopoGeometryColumn('topo_test', 'public', 'highways_test',
     ↪  'topo_geom', 'LINESTRING');"
    $ psql -U jonas -d mikh_0522 -c "UPDATE highways_test SET topo_geom =
     ↪  topology.toTopoGeom(way, 'topo_test', 1, 1.0);"

#### `mikh_0530`
Described above.

#### `mikh_restr_0602`
As `mikh_0530` but using the modified osm-file `mikhailovsk-turnrestriction.osm`
 ↪  instead. That file has been modified with a turn restriction for testing
 ↪  purposes.

#### `mikh_restr_0617`
As `mikh_0602` but extra columns in `planet_osm_point` to get point restrictions.
```

C.10 util

```
/
└─util/
   ├─catchtest/..................................................empty directory
   ├─Logging.cc
   ├─Logging.h
   ├─Point.h
   ├─TimeToStringMaker.cc
   └─TimeToStringMaker.h
```

Listing C.10: Files in /util