Distributed Systems seminar: Shortest-path problem

Heiki Pärn
Supervisor: Amnir Hadachi

Abstract

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its edges is minimized.

The problem is analogous to finding the shortest path between two intersections on a map where vertices correspond to intersections and edges correspond to road segments. The weight of an edge is the length of the corresponding road segment.

This is used in many applications. For example the driving directions in Google Maps[1] or Graphhopper[2].

Introduction

The program developed uses OpenStreetMap data to find the shortest (vehicle) path between two input coordinates. It is written in Java. To use the program you need to have a "map.osm" file in the same directory.

When running the program it first has to read in the map data. After this you can insert the coordinates of the starting and end points. The program calculates the shortest path between these points using the A* algorithm.
The repository is available at [3]. A compiled jar file of the program is available in the "downloads" section.

Getting the data

OpenStreetMap data is required to run the program. The easiest way to get it is by going to OpenStreetMap site http://www.openstreetmap.org and using the export feature. This works well for downloading a smaller amount of data.

To use a larger map (for example, the size of Tartu) it would be better to use the Overpass API. The URL to download all map data contained in a bounding box with bottom left coordinates minLat, minLon and top right coordinates maxLat, maxLon:

"http://overpass-api.de/api/map?bbox=minLon,minLat,maxLon,maxLat"

Here is an example URL that will download map file containing most of Tartu (35 MB).
www.overpass-api.de/api/map?bbox=26.6674,58.3373,26.7684,58.4089

The map file is in XML format. It consists of *nodes* and *ways*. A node is one of the core elements in the OpenStreetMap data model. It consists of a single point in space defined by its latitude, longitude and node id. Ways are relations between nodes. Example of a node:

```
<node id="8220937" lat="58.3790611" lon="26.7303632" version="6" timestamp="2011-03-19T12:36:44Z" changeset="7604312" uid="156900" user="k__"/>
```

The important parts of a node for this program are its "id", "lat" and "lon".

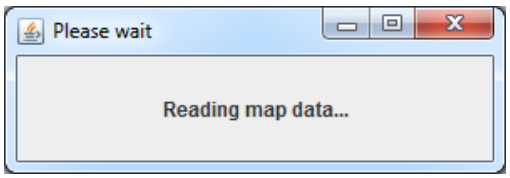
An example of a way:

```
<way id="33239457" version="5" timestamp="2014-08-23T19:58:51Z" changeset="24964005" uid="357111" user="enedaniel">
    <nd ref="3029578499"/>
    <nd ref="330041270"/>
    <nd ref="3029578545"/>
    <nd ref="377108084"/>
    <nd ref="3029578257"/>
    <nd ref="377110823"/>
    <nd ref="330042739"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="Lääne"/>
</way>
```

In a way the program first looks at the tags. If the way has the necessary tags (or doesn't have a tag such as "footway") then the program looks at the node ID's that are included in the way (such as `<nd ref="3029578499"/>`). Connections are added between the node objects that are referenced.

Running the program

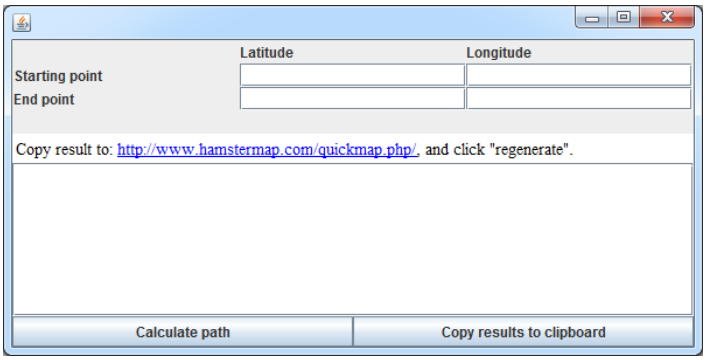
The program reads in all the relevant ways and nodes from the file. This means that the way must have *highway* tag and it must not have tags *track* or *footway*, since the scope of this project is only vehicle routing. Ways tagged with *<tag k="motor_vehicle" v="no"/>* are also ignored. If a street has *oneway* tag it is handled slightly differently from the usual two-way streets (the connections are only added one way). During this stage the program displays a loading window.



Picture 1: "Loading window"

The time it takes to read map data depends hugely on the size of the map exported. For a small region it can be some seconds but for a map containing Tartu (the bounding box example given earlier) it took 33 minutes. This should be taken into consideration when choosing the data to test with.

After the data has been read in, it is possible to insert the latitude and longitude for a starting and end point.



Picture 1: Program UI

"Calculate path" button will calculate and display the set of coordinates for the shortest path. This is done by choosing 2 nodes: one closest to the starting point and the other to end point. A* algorithm is used to find the path between those two nodes.

There is currently no custom map overlay implementation to display the results. The easiest way to visualize it is by copying the resulting set of coordinates into an existing tool for displaying them. For example [4].

A* algorithm

The program uses A* algorithm to find the shortest path. It is a search algorithm that is widely used in path finding and graph traversal.

As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way.

If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached. [5]

The algorithm uses a heuristic which has to be *admissable*. This means that it must not overestimate the distance to the goal. In this program the heuristic used is distance from goal-node.

The distance between two coordinates is calculated using the *Haversine* formula. Haversine formula gives distances between two points on a sphere from their longitudes and latitudes. This[6] implementation was used in the program.

The A* algorithm implementation itself is based on the pseudocode from wikipedia[7] and it can be seen in aStar() method of ShortestPath.java in the project source.

Results

When comparing the results to Google Maps driving directions or Graphhopper, it seems to consistently work quite well.

While reading in data can take a considerable amount of time, the algorithm itself is quite fast. It takes around 300 milliseconds to find a path in Tartu, which is similar to time taken by existing routing services.

There are some occasional oddities in the calculated path but these are possibly the result of OpenStreetMap data insertion errors or incomplete exported data.

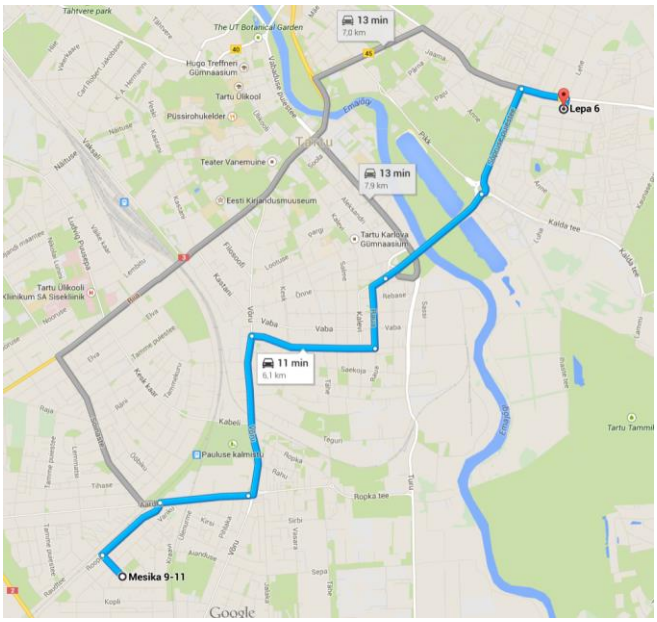
Sometimes the program gives a result that is actually shorter in length than Google Maps suggestion but when inserted into Google Maps the time estimate is longer. Google's algorithm is probably a lot more complex and takes into account more things than just the distance from goal.

The following pictures show an example result for shortest path between coordinates 58.350160, 26.705887 and 58.379696, 26.758984.

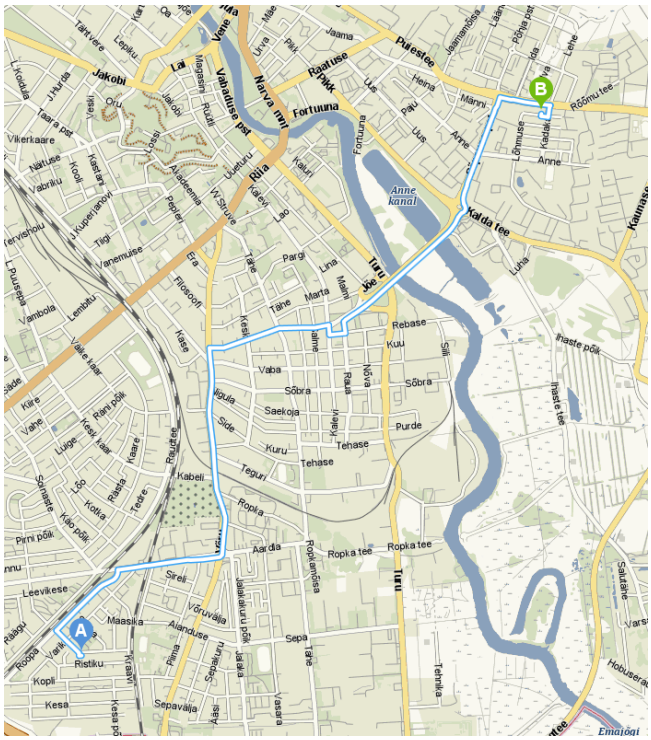
Picture 2 shows Google Maps driving directions output between these coordinates. Picture 3 shows a route suggested by Regio.

Picture 4 shows the program output shown on "hamstermap" site. The lines between points were added manually to make it easier to understand.

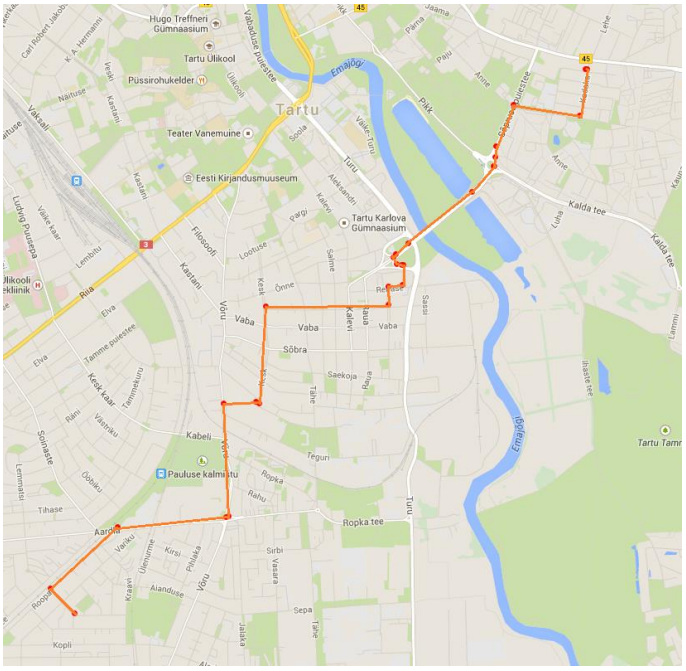
As you can see, all the routes are mostly similar with some slight differences. Most of the time the routes suggested by the program were similar to Google Maps routes. This shows that the algorithm should be working correctly.



Picture 2: Google Maps route



Picture 3: Regio[8] route



Picture 4: Project output with lines added manually

Future work

The main area of improvement would be reading in map data. This could perhaps be made faster by using an existing tool, such as Osmosis[9] for parsing the XML-file. Another possibility would be preprocessing the data which can make it much faster to read.

Currently you need to insert the coordinates for starting and end points. The result is also given as a set of coordinates. This could be improved by making a custom overlay for OpenStreetMap where the two input points could be selected on the map and where the route is also shown directly on the map.

Summary

The aim of this project was to develop a program for finding the shortest path using OpenStreetMap data. The program takes two coordinates as an input and displays a path between them as a set of coordinates.

A* algorithm was used for finding the path. The heuristic used for A* algorithm is the distance between nodes, which is calculated using the Haversine formula.

The results are promising but the main problem is that reading map data can take a lot of time. This is the main area that could be improved in the future.

References

[1]http://www.maps.google.com/

[2]https://graphhopper.com/maps/

[3]https://bitbucket.org/heiki112/ds_shortest_path

[4] http://www.hamstermap.com/quickmap.php

[5]http://waprogramming.com/download.php?download=50af7709377c22.88356189.pdf

[6]http://rosettacode.org/wiki/Haversine_formula#Java

[7]http://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode

[8]http://kaart.delfi.ee/

[9]http://wiki.openstreetmap.org/wiki/Osmosis