Lab Course Algorithms on OSM Data: Tasks

FMI, Algorithms Department

April 5, 2024

This document contains the tasks for the Lab Course “Algorithms on OSM Data”. Each task contains a list of helpful resources. The resources are roughly ordered by their relevance/usefulness for the project.

Goal

The goal of the lab course is to build an application for ship routing on the ocean based on OpenStreetMap Data.

Guidelines

Programming Language

You can choose any programming language which is suitable for this kind of project. We recommend languages that are a) compiled and b) give you control over the memory layout. This means Go, Rust and C++ are some of the best choices. Java is also viable option but you may need to program, like it was C to have acceptable performance (Using arrays of primitives to store data). Javascript in the browser has strict memory limitations and can therefore **not** be used.

Usage of Libraries

In principal, you are free to use any libraries in your project. But it must be apparent in your code that you did the tasks yourself. To give some examples: It is fine to use a library for parsing and reading PBF files. It is not fine to use a library that implements Dijkstra’s algorithm. The reason for this is that the former is only a part and implementation detail of the task while the latter is the core of the task. Please ask on the forum if in doubt.

Doing Tasks

The tasks are meant to be completed as one incrementally growing project. But you are free to implement some parts as extra projects/executables/etc. Tasks 1-6 are necessary for getting the “Schein” (certificate to take part in final project) while the project as a whole will be graded including task 7 of the lab course. Tasks 1-6 can be done in teams of two.

Submissions

There will be two code submissions (via Ilias) and a presentation of the final project. In the code submissions you have to provide the code of your project as well as instruction how to compile and run the project. The instructions should work on Ubuntu 20.04 or via a container (e.g. Docker). The first submission will have to contain tasks 1-6 and the second one the whole project.

After the second submission there will be meetings for the presentations. In these you should present your project in 10 to 15 minutes, focusing on the speedup techniques used and their efficiency. You should present running times of the techniques as well as speedups. If applicable also report the change in PQ pops. A short demo of the project as well as any other interesting challenge you solved on the way is always appreciated in the presentations. Be prepared for questions about your project.

Tasks

Task 1: Understand OSM Data Structures

As a first step, we need to get to know how data is organized inside OSM. We are primarily interested in “nodes” and “ways”.

Resources

- The OSM wiki explains the core data structures as well as interpretations of individual tags.
- This youtube-playlist explains the OSM primitives and tags with lots of examples

Task 2: Extract Coastlines from a PBF File

Read in a PBF file and extract all coastlines contained in the file. You can start by just outputting them to console. To see your results, save them as GeoJson and use geojson.io. Merge touching coastline ways to enable further processing in the following tasks. It may make sense to store the coastlines in a format that is easy to read for future use.

Resources

- For testing, downloads of OSM data by region from geofabrik are useful (Antarctica is a good starting point)
- For the project, a pbf file containing only the coastlines can be found in the Ilias course. **You have to use this file for the project**.
- PBF Format
- Definition of the coastline tag
- Geojson format definition
- Geojson.io a GeoJson visualization site
- Geojson styling guide: adding CSS properties that are supported by geojson.io
- Detailed tagging information for coastlines

Task 3: Distinguish between Water and Land

Implement the point in polygon test to determine if a certain position is in the ocean (aka passable for ships). Be aware the latitude and longitude **are not** coordinates on a plane. **Use the spherical model of the earth** to do your calculations.

Resources

- Many calculations for lat long points
- Calculations with lat long but converted into vectors first
- Properties of Lines on a Sphere & Point in Polygon on Sphere
- Fast and Robust Point-in-Spherical-Polygon Tests Using Multilevel Spherical Grids (Uni network only)
- In-polygon test with spherical polygons (Uni network only)
- `https://en.wikipedia.org/wiki/Point_in_polygon`
- Video with 3D animation of spherical vs vector coordinates

Task 4: Graph Generation

Implement a graph representation which allows routing on the oceans corresponding to the input. Graph nodes should be distributed randomly over the water area. Use the point in polygon test from task 3 to determine whether you can generate a node in a certain position. Your node distribution should be roughly uniform on the surface. (Be careful: distributing nodes uniformly by picking spherical coordinates at random will not produce such a distribution, instead you will get a higher node density towards the poles and a lower density around the equator). This means that you should weight your latitude distribution according to the length of the circles of latitude, i.e. according to $\cos(\phi)$.

Each node selects its closest south-west, south-east, north-west, and north-east neighbor, and creates a bidirectional edge to those neighbors. Make sure you don't create duplicate edges when nodes select each other. You can use spatial data structures like a quadtree or a simple grid to quickly determine these 4 neighbors without having to look at all nodes. Edges should only be created to neighbors which are at most 30 kilometers away. You can use the haversine formula to approximate the distance between two spherical coordinate pairs. Edge lengths should be stored as integers to ensure reproducibility of routes. Save the graph as an adjacency array representation.

You can use geojson.io to visualize your grid graph: use 0-length edges instead of nodes for better performance, and perhaps only render a smaller version of your graph (with a couple thousand nodes). **Your submission for this task must contain a screenshot of your graph's nodes, or preferably of the nodes of a smaller version of your graph.**

Efficiency (The Java Caveat)

For an efficient implementation of the adjacency array, it is important to avoid double indirections. The edge array should not contain pointers/references. In languages **like Java** where you can access structured data only via references, this means you can **not** use an Edge class with **individual Edge objects** but you need to rely on **multiple arrays of primitives**. If you then access all these arrays at the same index, you get the all the information for one edge. So an efficient way to store a lot of small "objects" in Java could be realized similar to this example:

```
class Edge {
  private static int[] startNode;
  private static int[] destNode;
  private static int[] dist;

  public static int getStart(int i) { return startNode[i]; }
  public static int getDest(int i) { return destNode[i]; }
  public static int getDist(int i) { return dist[i]; }
}
```

Resources

- Algorithms and data structures: Adjacency array on p. 168

(Optional) Task 4.1: Loading and Saving Graphs to FMI File format

The fmi file format is a simple adaptable text format to represent graphs. Its basic structure looks like this:

```
#

5
9
0 49.00 10.00
1 49.01 10.01
2 49.02 10.02
3 49.03 10.03
4 49.04 10.04
0 1 9
0 2 8
0 4 7
2 0 6
2 1 5
2 4 4
3 2 3
4 1 2
4 3 1
```

Lines that begin with # are comments. After the comments follow two lines that tell the amounts of nodes and edges in the graph respectively. Then all nodes are listed. In each node line the id, latitude and longitude are separated by space. After the nodes the edges lines follow. Their structure is: source node id, target node id and distance. Please note that you can add arbitrary information to your lines if necessary.

Task 5: Dijkstra's Algorithm

Implement Dijkstra's Algorithm for shortest paths on your graph data structure. The implementation must use a heap data structure (e.g `std::priority_queue` from C++) and stop as soon as the target node ist reached. Your average Dijkstra run time on a graph with 4 million nodes should be about 400ms-600ms or less. Contact your supervisor if you are unsure about the running times of your Dijkstra implementation.

Resources

- Repository demonstrating efficient implementation of Dijkstra's Algorithm

Task 6: Interactive Routing Front End

Add a GUI to your project. It should be possible to set start and end nodes as well as visualizing the route. The calculated distance and number of queue pops should be displayed in your GUI. It should be possible to keep the selected start and end nodes for multiple queries, in particular for multiple queries with different routing algorithms/speed-up techniques (after Task 7).

Resources

- Leaflet an easy to use js framework for displaying maps
- Cesiumjs is a framework for 3D Geospatial visualizations
- OpenLayers a more involved js framework for displaying maps

Task 7: Speed Up the Routing

Implement a speedup technique/heuristic for your project. Measure the speedup for 1000 random queries (start and destination should be in water, not finding a route is an acceptable result) as well as the difference in nodes pulled out of the heap. Please report speed ups as well as average query times. Please use a graph with 4,000,000 nodes. If this is not feasible, e.g. because of time/memory limitations for the preprocessing, state your setup explicitly, in your presentation and README. Your algorithm must report the **same** distance and ideally path as before. Make sure the routing front end task requirements (Task 6) are still fulfilled!

For practice (and to get some additional baseline data to compare your speedup against), you may implement A* and bidirectional Dijkstra, but this is not necessary.

You have to choose at least one speedup technique to implement. Some options are listed below, and you'll likely find others in the literature – or you may come up with your own technique! If you choose a technique that is not listed below, please check with your advisor to see whether your choice would be suitable for the lab course.

- ALT: determines some landmarks, precomputes distances between them and all nodes, and uses these distances to guide the search using A*
- Arc Flags: partitions the graph and saves with each edge e the set of partitions into which a shortest path exists over e, queries can then disregard edges which do not lead into the target node's partition
- Jump Point Search: allows jumping horizontally/vertically through the graph (requires some adaptations for our edge costs)
- Sharc: Arc Flags with shortcuts
- Contraction Hierarchies: approximately order nodes by importance and introduce shortcuts such that each shortest path starting in s only visits nodes in ascending order of importance, then some middle node, and then only nodes in descending order of importance.[1]
- CRP: partition graph and compute an overlay graph which allows routing between partitions, a query then only has to consider the original graph in the source and target nodes' partitions

[1] We have some CH intro slides if you choose this topic, ask your advisor when the time comes.

- Transit Nodes: in road networks, long queries typically use a sparse subgraph (highway-like roads). We can precompute pairwise distances between selected transit nodes on this subgraph, and store for each node a few nearby transit nodes and distances to them. Long queries then become lookups.
- Hub Labels: for every node v, precompute distances to a few hubs around v such that for every shortest s-t-path, there is a node on the path which is a hub of both s and t. Queries then become lookups.

Find and look through the original papers of speedup techniques you're interested in to help you choose between them, and to get more details on your chosen speedup technique

This list is roughly ordered by complexity. As a rule of thumb: the more complex your speedup technique is, the less perfect your implementation and evaluation has to be in order to get a good grade. Note that most speedup techniques are designed for road networks, which have a very different structure than the graphs we work with in this lab course. So the speedups you will obtain will likely be less than the speedup reported in referenced papers.

Your focus should not be on obtaining the best speedup (this would be unfair against people who choose speedup techniques which are less suitable for our graphs), but rather on implementing a speedup technique well, optimizing your implementation, and evaluating your implementation. If your speedup technique is based on contraction hierarchies (but is not contraction hierarchies), you may not have to do the preprocessing yourself - ask your advisor for more info.