Programming assignment phase 2 (prg2): Retkikartta (Hiking map)

Max Gratschew 283272

Datastructure

For this project I decided to store ways as a pointer to a struct in an **unordered_map** with WayID as a key and the pointer to a struct of a way as a value. The way struct has members for id, coordinates for the way and way's length calculated by a helper function.

Also a Crossroad struct. Crossroad is created from way's start and end points. Each way creates two crossroads, if no way points to that crossroad yet. Crossroads are stored to an **unordered_map** as coordinates as a key and the pointer to a crossroad struct as a value. Crossroad struct has members for coordinates, a vector of ways that start or end to the crossroad and search related members as:

- color as if the crossroad has been handled in a search or not
- distance from the previous crossroad
- pointer to the previous crossroad (needed for bfs and dijkstra)
- vector of previous crossroads (needed for dfs to find a cycle)
- WayID as which way was used to reach the crossroad

Unordered_map was chosen because most of it's member functions are constant on size and a map structure was a very good choice for this phase.

Functions

Unordered_map was a good choice since functions all_ways, add_way, ways_from, get_way_coords, clear_ways and remove_way were implemented with unordered_map which made all of the previously mentioned either constant or linear in complexity while vector was present in some of those functions.

Route_any was implemented using depth first search. When it comes to finding any route if present, depth first search is the way to go. It's

complexity is only O(V+E) and it’s based on going through a path as long as you can, then backtracking back to some crossroad with more paths if no target was found.

Route_least_crossroads was implemented using breadth first search. I saw bfs as the only logical option for this, since bfs is implemented so one verticle’s direct neighbors at the present depth prior to moving over to the other depth levels. This is why bfs was the only logical choice for this meanwhile its time complexity is only O(V+E) too.

Route_shortest_distance was implemented with dijkstra’s algorithm using priority_queue. Priority_queue can be given a compare function which made it possible for the priority_queue to always have the shortest distance from starting node to be at the top. Using dijkstra’s algorithm with priority_queue was very logical to use here as it’s a known algorithm for searching the shortest path between two nodes when it comes to the weight of the curves. Also it’s time complexity is only O((V+E)lgV) which is pretty good for finding the shortest path.

I created seperate helper functions to implement these searches as you will see from the code and everything will become very clear.