DATA STRUCTURE AND ALGORITHM – PHASE 2

PROGRAMMING ASSIGNMENT DOCUMENTATION

Name: Thuy Phuong Nhi Tran

Student no: H291937

Email: thuyphuongnhi.tran@tuni.fi

I. Datastructure explanations:

- Struct for stroring the information for the crossroads. The constructor will need the coordinate. There is unorder_map in this struct for storing the information of the "neighbor" crossroads that connects directly to this crossroad. Pair for storing the way that person goes to this crossroad from another possible crossroad.

```
struct Crossroad {
        Coord crossroad_coord;
        std::unordered_map<WayID, std::shared_ptr<Crossroad>> crossroad_way;
        bool visited = false;
        std::pair<WayID, std::shared_ptr<Crossroad>> pre_crossroad;
        Distance distance_for_dijkstra = 999999;

        //Constructor
        Crossroad(Coord xy_) {
            crossroad_coord = xy_;
        }
    };
```

- Struct for storing the information for the way. The constructor will need: distinguishing way ID, vector of multiple coordinates (where first and last elements are crossroads). The distance between the crossroads of the way will be calculated and store in variable way_distance with will not change

```
struct Way {
        WayID way_id;
        Coord_v way_coords;
        Distance way_distance = 0;
        std::pair<Crossroad_ptr, Crossroad_ptr> way_crossroads;
```

```
//Constructor

Way (WayID id_, Coord_v v_, std::pair<Crossroad_ptr, Crossroad_ptr> c_pair_){

    way_id = id_;

    way_coords = v_;

    way_crossroads = c_pair_;


    for (unsigned long i = 0; i < v_.size() - 1; i++) {

        Datastructures d;

        way_distance += static_cast<Distance>(floor(sqrt(d.distance_calculator(v_[i],
v_[i+1]))));

    }

}

};
```

- The container for storing all smart pointers to the Way is unorder_map with the unique key is the way ID. As the key of unordered_map needs to be unique and all method using Place ID to find a specific information, so WayID is suitable to be the key. Additionally, we do not need to the map to be in ascending order according to the Way ID and to reduce the time accessing to one element using Place ID, so unorder_map is suitable.:

```
std::unordered_map<WayID, Way_ptr> way_container;
```

- The container for storing all smart pointers to the Crosscroad is unorder_map with the unique key is the Coordinate after hashing. As the key of unordered_map needs to be unique and all method using Coordinate find a specific information, so Coordinate is suitable to be the key. Additionally, we do not need to the map to be in ascending order according to the Coordinate and to reduce the time accessing to one element using Coordinate, so unorder_map is suitable:

```
std::unordered_map<Coord, Crossroad_ptr, CoordHash> crossroad_container;
```

The idea in chosing this data structure is to create a weighted graph where the vertices are the crossroad and edges are the way with numeric value is the length of the way.

II. Asymptotic Performance of implementation

Function	Asymptotic performance	Brief explanation
all_ways();	O(n)	loop through the map to retrieve all keyswhich are

		WayID then push_back to the vector whose complexity is O(1)
add_way(WayID id, std::vector<Coord> coords)	O(1)	find and insert in unorder takes constant time.
ways_from(Coord xy)	O(n)	reserve in vector takes O(n), loop through all another crossroads of one crossroad takes O(n). Therefore, the total is O(n)
get_way_coords(WayID id)	O(1)	find element in unorder_map takes O(n)
clear_ways()	O(n)	using clear takes O(n), linear to the size
route_any(Coord fromxy, Coord toxy)	O(V+E)	Using BFS, where V is the vertices and E is the edges
remove_way(WayID id);	O(n)	using find takes O(1), using erase takes O(n)
route_least_crossroads(Coord fromxy, Coord toxy);	O(V+E)	Using BFS, where V is the vertices and E is the edges. The implementation is from route_any as both using BFS
route_with_cycle(Coord fromxy);	O(V+E)	Using BFS, where V is the vertices and E is the edges
route_shortest_distance(Coord fromxy, Coord toxy);	O((E+V)logV)	Using Dijkstra algorithm, where E is the edges and V is vertices
trim_ways();	O(|E|^2 + |E||V|log|V|)	using Kruscal algorithm, where E is the edges and V is the vertices

III. Result

- The implementation pass all the test cases in the given file.
- Performance result with my own computer: attachment in the perf-test-phase2.txt