Dokumentaatio

junyuan.fang@tuni.fi
H292042

Tietorakenteen valinta:

//phase 1////////////////////////////

```
//Place
struct Place{
    Name name_;
    PlaceType type_;
    Coord xy_;
};
unordered_map<PlaceID,Place> placeUnOrMap_;

//Area
struct Area{
AreaID area_ID_;
Name name_;
vector<Coord> coords_;
Area * parent_=nullptr;
vector<Area*> subArea_;
};
unordered_map<AreaID,Area> areaUnOrMap_;
```

Unordered_map sopii työmme, koska ”alkion poisto”, ”tietyn etsintä”, ”lisäysmuualle” on keskimäärin Θ(1) tai pahimmillaan O(n), jonka tapahtuva todennäköisyys on todella pieni.

//phase 2//////////////////////////////

```
struct besideInfo{
        besideInfo(){
            d=-1;
            ptr=nullptr;
        }

        Distance d;
        CoordData* ptr=nullptr;
    };

    struct CoordData {
        CoordData(){
            this->coord=NO_COORD;
            this->besideInfo={};
```

```
            this->colour=Colour::WHITE;
            this->from=nullptr;
            this->fromWay=NO_WAY;
            this->d=-1;
        }
        Coord coord=NO_COORD;
        unordered_map<WayID, besideInfo> besideInfo={};
        Colour colour=Colour::WHITE;
        CoordData* from=nullptr;
        WayID fromWay=NO_WAY;
        Distance d=-1;//infinity
        //d is to next's distance
    };
    struct PQueComCoorD_Ptr{
     bool operator ()(const CoordData* a, const CoordData* b){
        return a->d > b->d;    //just like reload <, so smaller one goes
out from the p_queue first
    }
unordered_map <WayID, vector<Coord>> wayIDUnordMap_;
unordered_map<Coord, CoordData*, CoordHash> coordUnordMap_;//only
crossroada are added
```

Koska työssä on käytetty prioriteettijonoa, yksi tapa on luoda struct, ja määrittää struct sisällä vertailuoperaation.

Hajautustaulu sopii meidän työhön. Kurssi puolella saatu CoordHash auttaa iso kokoisen coordUnordMap:n ”rehash”.

Toteutksessa käytetyt STL funtiot, ja niiden asymptoottinen notaatio

Unordered_map:

- find, keskimäärin Θ(1) tai pahimmillaan O(n)
- empty, O(1)
- erase, keskimäärin Θ(1) tai pahimmillaan O(n)
- size, O(1)
- clear, keskimäärin Θ(1) tai pahimmillaan O(n)
- insert, keskimäärin Θ(1) tai pahimmillaan O(n)

Vector:

- back, O(1)
- pushback, amortisoitu Θ(1) tai pahimmillaan O(n)
- size, O(1)
- clear, O(n)

sort(): O(nlog(n))

//phase 1///////////////////////////////

```
    // Estimate of performance: O(1)
    // Short rationale for estimate:Only used size() to return int. and its
comlexity is O(1)
    int place_count();

    // Estimate of performance: O(n)
    // Short rationale for estimate: because unordered_map's clear() is
O(n)
    void clear_all();

    // Estimate of performance: O(n)
    // Short rationale for estimate:for-loop dominate the time
    std::vector<PlaceID> all_places();

    // Estimate of performance: in average theta(1),worst O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find() and size(), and both are theta(1)
    bool add_place(PlaceID id, Name const& name, PlaceType type, Coord xy);

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find()
    std::pair<Name, PlaceType> get_place_name_type(PlaceID id);

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find()
    Coord get_place_coord(PlaceID id);

    // We recommend you implement the operations below only after
implementing the ones above

    // Estimate of performance: O(n log(n))
    // Short rationale for estimate: same as sort()'s complexity and
unordered_map's find() in average is theta(1)
    std::vector<PlaceID> places_alphabetically();

    // Estimate of performance: O(n log(n))
    // Short rationale for estimate:same as sort()'s complexity and
unordered_map's find() in average is theta(1)
    std::vector<PlaceID> places_coord_order();
```

```
    // Estimate of performance: O(n*m)/ O(n)
    // Short rationale for estimate:m is string's length, because strings
area shart so we assume m=constant
    std::vector<PlaceID> find_places_name(Name const& name);

    // Estimate of performance: O(n)
    // Short rationale for estimate: one for-loop
    std::vector<PlaceID> find_places_type(PlaceType type);

    // Estimate of performance:in average theta(1),Worst case: O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find(). Because we find it from datastructure then we change it
    bool change_place_name(PlaceID id, Name const& newname);

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find(). Because we find it from datastructure then we change it
    bool change_place_coord(PlaceID id, Coord newcoord);

    // We recommend you implement the operations below only after
implementing the ones above

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:complexity depends on unordered_map's
find() and insert()
    bool add_area(AreaID id, Name const& name, std::vector<Coord> coords);

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:no for loop, and unordered_map's find()
dominate the time
    Name get_area_name(AreaID id);

    // Estimate of performance: in average theta(1),Worst case: O(n)
    // Short rationale for estimate:no for loop, and unordered_map's find()
dominate the
    std::vector<Coord> get_area_coords(AreaID id);

    // Estimate of performance: O(n)
    // Short rationale for estimate:one for-loop for using push_back(), and
pushback() is armortized theta(1)
    std::vector<AreaID> all_areas();

    // Estimate of performance: O(n)
```

```
    // Short rationale for estimate:one for-loop for using push_back(), and
pushback() is armortized theta(1)
    bool add_subarea_to_area(AreaID id, AreaID parentid);

    // Estimate of performance: O(n)
    // Short rationale for estimate:while-loop increase the time complexity
    std::vector<AreaID> subarea_in_areas(AreaID id);

    // Non-compulsory operations

    // Estimate of performance: O(1)
    // Short rationale for estimate:there we only change the "flag"'s value
so is constant
    void creation_finished();

    // Estimate of performance: O(n)
    // Short rationale for estimate:Depth First Traversal is O(n + m),
where n is the number of nodes, and m is the number of edges.
    std::vector<AreaID> all_subareas_in_area(AreaID id);

    // Estimate of performance: O(n)
    // Short rationale for estimate: there was one for loop, and a sort()
function in the for-loop. Should be O(n*mlog(m)), but m <=3 and it's small
we assume it is a constant
    std::vector<PlaceID> places_closest_to(Coord xy, PlaceType type);

    // Estimate of performance: in average theta(1), O(n)
    // Short rationale for estimate:because we only used unordered_map's
erase()
    bool remove_place(PlaceID id);

    // Estimate of performance: O(n+m)
    AreaID common_area_of_subareas(AreaID id1, AreaID id2);
```

//phase 2//////////////////////////////

```
    // Phase 2 operations--------------------------------------------------
---

    // Estimate of performance: theta(n)
    // Short rationale for estimate: for-loop
    std::vector<WayID> all_ways();
```

```
    // Estimate of performance: O(n), on average case: theta(1)
    // Short rationale for estimate: no for or whileloops, both .insert()
and .find() are: O(n), on average case: theta(1).
    bool add_way(WayID id, std::vector<Coord> coords);

    // Estimate of performance: theta(n)
    // Short rationale for estimate: one for loop. Inside for loop's
function is .push_back, which takes O(1).
    // O(n)=O(n*1)
    std::vector<std::pair<WayID, Coord>> ways_from(Coord xy);

    // Estimate of performance: O(n), on average case: theta(1)
    // Short rationale for estimate:  .find() is used
    std::vector<Coord> get_way_coords(WayID id);

    // Estimate of performance: theta(n)
    // Short rationale for estimate: for loop
    void clear_ways();

    // Estimate of performance: O(V+E)
    // Short rationale for estimate: v describe while loop called
vertices, E describe for loop called edges
    // O(n) if coord is found from the last element. n>=V
    std::vector<std::tuple<Coord, WayID, Distance>> route_any(Coord fromxy,
Coord toxy);

    // Non-compulsory operations

    // Estimate of performance: O(n), on average case: theta(1)
    // Short rationale for estimate:  .find(), .erase()  are used
    bool remove_way(WayID id);

    // Estimate of performance: O(V+E)
    // Short rationale for estimate: v describe while loop called
vertices, E describe for loop called edges
    // O(n) if coord is found from the last element. n>=V
    std::vector<std::tuple<Coord, WayID, Distance>>
route_least_crossroads(Coord fromxy, Coord toxy);

    // Estimate of performance: O(V+E)
    // Short rationale for estimate: v describe while loop called
vertices, E describe for loop called edges
    // O(n) if id is found from the last element. n>=V
```

```
    std::vector<std::tuple<Coord, WayID>> route_with_cycle(Coord fromxy);

    // Estimate of performance: O(V+E)
    // Short rationale for estimate: v describle while loop called
vertices, E describle for loop called edges
    // O(n) if coord is found from the last element. n>=V
    // priority_queue is used O(log(n)), but O(n)>O(log(n))
    std::vector<std::tuple<Coord, WayID, Distance>>
route_shortest_distance(Coord fromxy, Coord toxy);

    // Estimate of performance:
    // Short rationale for estimate:
    Distance trim_ways();

Private:
// Estimate of performance: theta(n)
    // Short rationale for estimate: for loop
    void clearCoorDataMarks();
    // Estimate of performance: theta(n)
    // Short rationale for estimate: for loop
    Distance calWayDist(const WayID id);
    // Estimate of performance: O(n)
    // Short rationale for estimate: path is a line
    void printPath(vector<std::tuple<Coord, WayID, Distance> >& path, const
CoordData coordDataFrom, const CoordData coordDataTo);
    // Estimate of performance: O(n)
    // Short rationale for estimate: path is a line
    void printPath(vector<std::tuple<Coord, WayID> >& path, const CoordData
coordDataFrom, const CoordData coordDataTo);
```