Valittujen tietorakenteiden ja tehtyjen ratkaisujen perustelu

Päädyin käyttämään harjoitustyön toisen vaiheen päätietorakenteina jälleen unordered_mappeja, jossa hakuavaimena oli joko tietyn väylän ID (WayID) tai kyseisen risteyksen sijainti (Coord) ja näitä hakuavaimia vastaavat arvot olivat structeja, joissa oli kunkin väylän ja risteyksen kannalta tarvittavia tietoja. C++:lla kirjoitettuna nämä tietorakenteet ja structit olivat siis seuraavanlaisia:

```
std::unordered_map<WayID,Way> ways_;
std::unordered_map<Coord,Node,CoordHash> nodes_;
```

```
struct Node
{
    Coord location;
    std::unordered_multimap<Coord,WayID,CoordHash> accesses;
    Status node_status;
    Distance steps_taken;
    Distance route_distance_so_far;
    Distance route_distance_estimate;
    Node* previous_node;
    Node* secondary_previous_node;
    WayID previous_way;
    WayID secondary_previous_way;
};
```

```
struct Way
{
    std::vector<Coord> coordinates;
    Distance distance;
};
```

Huom. Saatan tässä dokumentissa sekä koodin kommenteissa käyttää välillä termejä Node, Crossroad, Solmu ja Risteys ristiin keskenään **mutta viittaan näillä kaikilla samaan asiaan.**

Päädyin käyttämään kahta erillistä tietorakennetta, koska usea harjoitustyössä toteutetuista operaatioista otti parametrinaan väylän ID:n (tai muutoin hyötyi siitä että ID:llä voidaan hakea unordered_mapista nopeasti tätä vastaava väylä), jolloin oli toteutuksen tehokkuuden kannalta suotavaa, että tuolla ID:llä voidaan suoraan hakea unordered_mapissa tätä ID:tä vastaava väylä. Näitä operaatioita oli mm. all_ways, way_coords ja remove_way.

Koin myös kaksi tietorakennetta selkeämmäksi valinnaksi ratkaisun kokonaisuuden kannalta, sillä uskon että graafialgoritmien toteuttaminen oli suoraviivaisempaa näin. (Erityisesti A* ja Dijkstra koska ne huomioivat väylien painot eli pituudet). Kaksi tietorakennetta mahdollisti myös sen, että kunkin risteyksen structiin voitiin tallettaa tieto siitä, mihin risteyksiin tietystä risteyksestä on pääsy ja tätä naapurisuhdetta vastaava tieto siitä, mitä väylää pitkin sinne tulee kulkea. Risteyksen structissa oli siis unordered_multimap, jossa avaimena naapuriristeyksien koordinaatit ja näitä vastaavat arvot ovat väyliä, joita pitkin tulee kulkea päästäkseen hakuavaimen sijainnissa olevaan risteykseen. Tuo valittiin multimapiksi siksi, koska tällöin mahdollistetaan se, että kahden koordinaatin välillä voi olla useampi eri väylä, jotka ovat eri pituisia. Tällöin graafialgoritmeissa risteyksen naapureita läpikäydessä päästään kaikkiin tarvittaviin naapurien tietoihin käsiksi, koska jokaisella risteyksellä on structissaan tietona kaikkien naapurien koordinaatit ja näitä

vastaavien väylien ID:t. Tällöin naapurisolmujen tietoja voidaan hakea ja päivittää tehokkaasti, koska koordinaatit ja väyläID:t toimivat myös hakuavaiamina noihin keskustietorakenteisiin nodes_ ja ways_.

Luennolla käydystä harjoitustyön 1. osan perkauksesta opin, että voisi olla vielä tehokkaampaa käyttää osoittimia sen sijaan, että hakee tietoja unordered_mapista avaimilla. Vaikka unordered_mapista hakeminen onkin vakioaikaista, on se kuitenkin verrattuna osoittimiin hitaampaa, koska osoittimilla tietoa ei tarvitse hakea erikseen. Tein tässä kohtaa kuitenkin kompromissin jättää toteutuksen tuollaiseksi, enkä lähtenyt sotkemaan sinne enää lisää osoittimia, koska olin ehtinyt käyttää tässä kohtaa tähän harjoitustyöhön jo paljon aikaa.

Osa harjoitustyössä toteutetuista operaatiosta etsii jonkinlaista reittiä kahden pisteen välillä tai silmukkaa graafista. Kirjoitin graafialgoritmit (DFS, BFS, Dijkstra, A*) omiksi olion sisäisen rajapinnan metodeikseen. Valinnat käytettävistä graafialgoritmeista oli näitä tarvitsevissa operaatioissa melko itsestäänselviä sen tiedon perusteella, mitä luennoilta olin oppinut. Route_shortest_distance hyödyntää A*-algoritmia, koska tämä operaatio halusi löytää painotetusta graafista lyhyimmän reitin kahden pisteen välillä. Tämä oli tehokkaampi valinta kuin Dijkstra, koska Dijkstra tekisi tämän operaation toteutuksen kannalta turhan paljon työtä kartoittamalla lyhyet reitit kaikkiin koordinaatteihin. BFS ja DFS ei puolestaan sopisi siksi, että Route_shortest_distance :ssa meitä kiinnostaa myös kaarien eli väylien painot eli pituudet, eikä ainoastaan graafissa otettavien askelien lukumäärä. Route_any hyödyntää DFS:ää, koska route_anyn tarkoitus oli löytää vain jokin reitti kahden pisteen välillä. BFS olisi ollut tässä turhan raskas, koska se olisi etsinyt aina lyhyimmän reitin. DFS oli siis tässä kohtaa tehokkaampi valinta, koska etsityn reitin pituudella ei ollut väliä. Route_least_crossroads taas hyödyntää BFS:ää, koska nyt niiden risteyksien lukumäärä, jotka tulevat tietyllä reitillä reittiä kuljettaessa vastaan, voidaan ajatella askelien lukumääräksi graafissa. Tällöin BFS soveltui tähän tarkoitukseen parhaiten, koska se löytää nimenomaan lyhyimmän reitin askelien perusteella, eikä välitä väylien absoluuttisista pituuksista. Route_with_cycle hyödyntää jälleen DFS:ää, mutta hieman eri muotoisena, koska nyt haluttiin löytää väylästä vain reitti, joka tekee silmukan. Tätä varten risteyksen structiin lisättiin tiedot `Node* secondary_previous_node` ja WayID secondary_previous_way; jotta silmukan löytyessä kyseisen silmukkaristeyksen tietoihin voitiin tallettaa tiedot kahdesta eri solmusta; siitä johon viimeksi tultiin silmukan löytyessä (secondary) sekä siitä, josta kyseiseen silmukkasolmuun tultiin alun perin, kun sen ei havaittu vielä olevan silmukka.

Graafialgoritmeja kutsuvat metodit hyödyntävät myös yhteisiä olion sisäisen rajapinnan metodeja calculate_distance(vector<Coord>), distance_between_nodes(Coord, Coord), restore_nodes() ja track_route(Coord), joiden nimet selittävät melko hyvin niiden toiminnan. Restore_nodes() resetoi graafin sellaiseen tilaan, jotta sille voidaan toteuttaa graafialgoritmi (kaikki solmut valkoiseksi, kuljetut matkat

nolliksi jne.) Track_route() ottaa puolestaan parametrina koordinaatin, ja jäljittää reitin, jota pitkin tähän kuljettiin graafialgoritmissa, ja lisäilee paikkatiedot vektoriin ja palauttaa sen muotoisen vektorin, jonka metodien haluttiinkin palauttavan. Näissä ei sinänsä ole mitään erikoista, mutta ajattelin tehdä ne erillisiksi metodeikseen välttääkseen toisteista koodia eri operaatioiden välillä.

Operaation trim_ways() toteutus jäi minulla keskeneräiseksi, koska aikaa ei vain ollut enää enempää tähän harjoitustyöhön upotettavaksi. Avaan tässä kuitenkin ajatteluani tuon nykyisen toteutukseni taustalla. Lähtökohtana tässä minulla oli Dijkstran algoritmi, sillä koin sen soveltuvan parhaiten tähän tarkoitukseen luennoilla esitetyistä graafialgoritmeista, koska se kartoittaa lyhyimmät reitit koko graafin kaikkiin risteyksiin, eikä ainoastaan yksittäiseen risteykseen. Ajattelin tässä antaa Dijkstralle jonkin satunnaisen koordinaatin lähtötiedoksi (joka on risteys), ajaa Dijkstran sille, tämän jälkeen tarkistaa onko graafin kaikissa solmuissa käyty, ja jos ei ole, ajaa Dijsktra uudelleen siten, että kaikki risteykset tulee läpikäydyiksi. Dijsktralta jäi tavoittamatta ensimmäisellä suorituskerralla tiettyjä solmuja esimerkiksi kintulammitesteissä, koska tuossa kartassa oli erillisiä väyläverkostoja, joiden välillä ei oletusarvoisesti ollut yhteyttä (Tämän vuoksi myös Dijkstra ottaa parametrina minulla totuusarvon, jolla välitetään tieto siitä, halutaanko risteyksien tiedot resetoida vai ei) Tämän jälkeen kun kaikille väylille oltiin ajettu Dijkstra, pyrkimyksenäni oli ottaa jokaisen risteyksen tiedoista talteen sen väylän ID jota pitkin kyseiseen risteykseen oltiin tultu Dijkstran algoritmilla. (Koska tällöin nämä väyläthän ovat lyhyimpiä kahden pisteen välisiä väyliä). Kun tallessa oli ne väylät, jotka halutaan säästää trimmaukselta, voidaan kaikki muut väylät poistaa. Kun väylät on poistettu sekä ways_ta että risteyksien tiedoista, voidaan jäljellä olevien väylien pituus laskea yhteen ja palauttaa tämä arvo. Nykyistä toteutusta tehdessäni en kuitenkaan tullut ajatelleeksi, että Dijkstra kartoittaa lyhyimmät reitit nimenomaan sille annettuun lähtöpisteeseen, jolloin se ei tietenkään sovi ainakaan suoraa sellaisenaan tämän operaation trim_ways käytettäväksi. Uskoisin että tuon trim_waysin saisi toteutettua oikein nykyisestä toteutuksestani käymällä kaikkien risteyksien tiedoista naapurisolmut läpi, tarkastaa onko naapurisolmujen joukossa sellaisia, joilla on sama koordinaatti mutta niihin johtava väylä on eri, poistamalla duplikaattiväylistä ne, jotka ovat suurempia kuin lyhyin väylä ja poistamalla nämä väylät myös tuolta ways_ -tietorakenteesta. Tässä kohtaa totesin kuitenkin, että aikani ei enää riitä tämän toteuttamiseen ja testailemiseen, joten jätin tuon nykyisen toteutuksen sellaisekseen. (Sekä toteuttamani Dijkstran algoritmin jätin myös, vaikka sitä ei muut operaatiot kutsukkaan). Dijkstra itsessään mielestäni toimii kuten pitää, sillä testasin sitä myös route_shortest_distancen yhteydessä ja sain sillä vastaavia tuloksia kuin A*:lläkin.

Huom. Tuo trim_way kannattaa huomioida myös ohjelmaa testatessa. Itse en tätä aluksi tullut ajatelleeksi kintulammitestejä ajaessa ja luulin että muut reittiä hakevat operaatiot olisivat rikki, kunnes tajusin lopulta, että tuo nykyinen trim_way kuitenkin poistaa osan väylistä, joka voi vaikuttaa myös muiden väyläoperaatioiden myöhempään toimintaan, ellei väyliä lisää takaisin.