Harjoitustyö 2 – Niklas Nurminen

Aloitin harjoitustyön tarkastelemisen jälleenkerran lukemalla työohjeen ja miettimällä, mitä vaaditaan ja mikä olisi hyvä tietorakenne. Aika nopeasti kävi ilmi, että haluan toteuttaa graaphin, joka on suunnattu ja painotettu. Aluksi lähdin toteuttamaan ilman pointtereita, mutta nopeasti otin pointterit käyttöön. Aluksi yritin tehdä käyttäen normaaleja pointtereita, mutta muistin varaaminen tuotti hieman ongelmia, koska poistaminen ei ilmeisesti toiminut täysin. Tämän jälkeen toteutin koko ohjelman käyttäen fiksuja osoittimia.

Loin unordered_map:in jonka avaimena on *pysäkki-id* ja arvona fiksu osoitin *Node* nimeseen structiin. Node esittää ohjelmassa yhtä pysäkkiä, joten siltä löytyy myös *pysäkki-id:n* lisäksi koordinaatti muuttuja ja vectori, jossa on kyseisen pysäkin kaikki kaaret. Ne ovat myös fiksuilla pointtereilla toteutettu. Tein tämän siksi, että jos yhden pysäkin poistaa tulee kaarien poistua myös, koska tällöinhän kaarien lähtöpaikkaa ei ole olemassa. Yhdessä kaaressa on fiksuina osoittimina lähtöpaikka ja päämäärä, ne ovat fiksuilla osoittimilla siksi ettei tarvitse erikseen etsimis komentoja, koska voidaan suoraan löytää vaikka päämäärä pysäkin kaareet jne. Ohjelmassa on myös reitti tietorakenne tehty unordered_map:ssa jossa on avaimena on *reitti-id* ja arvona *deque* tietorakenne, jossa on *RouteStopInfo* struct sisällä. Tähän structiin tallennetaan pysäkin lähtöajat ja *pysäkki-id*, myös reitin järjestys pysyy kokoajan vakiona. Vaihdoin vectorin dequeen siksi, koska vector ei välttämättä pidä järjestystä samana!

Koodissa on hieman kommentoitu koodia, jotta siitä lukija ymmärtää jotain. Lyhyesti selitettynä *add_stop* tehdään *Node*, joka lisätään unordered_mappiin. *Add_route* funktiossa lisätään tietyille *Node*:lle niitä kaaria. Tässä vaiheessa käytännössä olemme rakentaneet toimivan tietorakenteen. Olen tehnyt myös pari apufunktiota kulkureittien löytämiseksi, jotta koodi pysyy jollain tavalla luettavana ja myös esimerkiksi journey_with_cycle_helper etsii rekursiivisesti syklin, joten se on tottakai oma funktionsa.

- *Journey_least_stops* on toteutettu käyttäen BFS algoritmia, eli breadth-first-search.
- *Journey_with_cycle* on toteutettu käyttäen DFS algoritmia rekursiivisesti.
- *Journey_shortest_distance* on toteutettu käyttäen Dijkstra algoritmia.

Koodia on hieman kommentoitu, mutta perjaatteessa koodi on melko helposti luettavaa. Vaikeinta varmasti koko harjoitustyössä oli saada ohjelma toimimaan osoittimilla, myös tietorakenteen fiksu toteutus on hankalaa, koska tällä hetkelläkään se ei ole järkevä.

Journey_earliest_arrival metodi jäi toteuttamatta kokonaan. Työhön annettiin hyvin aikaa, mutta muista kursseista ja työstä johtuen en ehtinyt tekemään metodia loppuun, koska tietorakennetta olisi tullut muuttaa hieman erilaiseksi. Olisin lähtenyt toteuttaa kyseistä metodia niin, että aluksi etsisin kaikki reitit jolla on aikoja. Tämän jälkeen minun pitäisi saada jokaisen *Node* pysäkkiajat ja tietyt kaaret jotka ovat näillä reiteillä. Tässä kohti minun tietorakenteeni ei enään ollut fiksu valinta, joten olisi ollut iso homma korjata kaikki. Kuitenkin jos tämän olisin tehnyt olisin seuraavaksi toteuttanut tämän melko samanlaila kuin *journey_shortest_distance*, mutta vain vertaillen aikaa, mikä kyseisen reitin lopuksi on.