TIE-20100 Harjoitustyö dokumentaatio
Vaihe 2

Yleiset ratkaisut

Reiteille (Route) on käytetty tietorakenteena structia, johon tallennettu seuraavat tiedot

```
RouteID id_
std::vector<Stop*> stops_
std::vector<std::vector<Time>> trips_
```

Alkuperäisiin **Stop** structeihin ei lisätty uusia tietoja kuten painoa, koska koodi haluttiin pitää täysin yhteensopivana ensimmäisen vaiheen kanssa. Reittien muodostamisessa solmut (**StopNode**) on toteutettu structeina, jossa seuraavat tiedot

```
Stop* stop_
Distance distance_
Distance weigth_
std::vector<std::pair<StopNode*, Route*>> destinations_
std::pair<StopNode*, Route*> previous_
int visited_
```

Graafin kaaret ovat tallennettu vektorissa destinations_. Parissa tallennetaan kohde, jonne päästään tästä pysäkistä sekä mitä reittiä pitkin sinne on päästään.

Kaikki reitit ja solmut tallennettiin hash-taulukkoihin std::unordered_map<RouteID, Route> ja std::unordered_map<StopID, StopNode>. Ohjelman toiminta perustuu hyvin pitkälti reittien ja solmujen etsimiseen käyttämällä niiden tunnistetta RouteID ja StopID. Ensiarvoisen tärkeää oli siis saada näiden haku mahdollisimman nopeaksi ja hash-taulukko tarjoaa nopean (käytännössä) vakioaikaisen haun riippumatta reittien ja solmujen määrästä. Reittejä on realistisessa tilanteessa hyvin vähän, joten vektorikin olisi täysin toimiva vaihtoehto lineaarisesta hausta huolimatta.

Funktioiden ratkaisut

Tehokkuuksissa std::unordered_map<> tehokkuudeksi on oletettu hauissa, lisäyksissä ja poistoissa Θ(1) vaikka hash-taulukon rakenteen takia on mahdollista, että tehokkuus on Θ(n) jos kaikki alkiot ovat samassa indeksissä. Funktioissa, joissa palautetaan vektori, on varattu vektorille valmiiksi oikea koko, jos se on ollut tiedossa, esimerkiksi **all_routes**() -funktiossa, jossa se on varmasti kaikkien reittien määrä.

Leveyteen ensin ja syvyyteen ensin -algoritmien tehokkuudeksi merkitään Θ(n), joka on tarkemmin laskettuna Θ(V+E), jossa V on solmujen määrä ja E kaarien määrä. Molemmat algoritmit pysähtyvät, kun määränpää löytyy. Kun reitti löytyy, solmujen tiedoista muodostetaan matka käyttämällä funktiota **make_journey**(), joka on lineaarinen. Tätä käytetään jokaisessa reitin luomisessa eikä listata siten enää alla erikseen. Lisäksi solmujen tallentamat tiedot kuten paino tyhjennetään joka kerta, joka on myös lineaarinen operaatio. Kolmas käytetty algoritmi oli Dijsktra ja sen kaksi eri tapausta käsitellään sitä käytettäessä.

Julkiset jäsenfunktiot	Tehokkuus	Selitys
std::vector<RouteID> **all_routes** ()	Θ(n)	Jokainen reitti iteroidaan läpi ja lisätään vektoriin tehokkuudella Θ(1).
void **clear_all** ()	Θ(n)	std::unordered_map<>::**clear**() poistaa jokaisen alkion erikseen (?) ja tämä suoritetaan kaikille tiedoille.
bool **add_route** (RouteID id, std::vector<StopID> stops)	Θ(n)	std::unordered_map<>:sta voidaan vakioajassa löytää onko reittiä vielä olemassa, mutta koska jokaisen pysäkin olemassaolo pitää tarkistaa, funktio on lineaarinen pysäkkien määrässä.
std::vector<std::pair<RouteID, StopID> **routes_from** (StopID stopid)	Θ(n^2)	Jokaisen reitin pysäkit käydään läpi ja tarkistetaan, onko se haluttu pysäkki. Tarkalleen ottaen tehokkuus on reittien määrä kertaa keskimääräinen pysäkkien määrä reiteillä. Tehokkuutta olisi voinut parantaa lineaariseksi tallentamalla paitsi pysäkit reiteille, myös reitit pysäkeille. Tämä aiheuttaisi kuitenkin ehkä turhaa ylimääräistä päällekkäisyyttä ja ensimmäisen osan toimintaa jouduttaisiin myös muuttamaan.
std::vector<StopID> **route_stops** (RouteID id)	Θ(n)	Jokainen pysäkki iteroidaan läpi ja sen ID lisätään vektoriin tehokkuudella Θ(1).
void **clear_routes** ()	Θ(n)	std::unordered_map<>::**clear**() poistaa jokaisen alkion erikseen (?).
std::vector<std::tuple<StopID, RouteID, Distance>> **journey_any** (StopID fromstop, StopID tostop)	Θ(n)	Funktio käyttää leveyteen ensin -algoritmia.
std::vector<std::tuple<StopID, RouteID, Distance>> **journey_least_stops** (StopID fromstop, StopID tostop)	Θ(n)	Funktio käyttää leveyteen ensin -algoritmia.
std::vector<std::tuple<StopID, RouteID, Distance>> **journey_with_cycle** (StopID fromstop)	Θ(n)	Funktio käyttää syvyyteen ensin -algoritmia.
std::vector<std::tuple<StopID, RouteID, Distance>> **journey_shortest_distance** (StopID fromstop, StopID tostop)	Θ(nlogn)	Funktiossa käytetään Dijkstran algoritmia täydellisen graafin muodostamiseen. Toisin kuin edellä, kaikki solmut käydään läpi siis. Samoin kuin leveyteen ja syvyyteen ensin, algoritmi tutkii jokaisen solmun ja kaaren vain kertaalleen, mutta jokaisella kierroksella suoritetaan logn operaatio (luentokalvojen mukaan).
bool **add_trip** (RouteID routeid, const std::vector<Time> &stop_times)	Θ(1)	Bussien aikataulujen tallennus tehtiin reitteihin samaan formaattiin, joten vektorin kopiointi riittää eikä aikoja tarvitse yksitellen käydä läpi.
std::vector<std::pair<Time, Duration>> **route_times_from** (RouteID routeid, StopID stopid)	Θ(n^2)	Jokaisen pysäkin jokainen aika joudutaan käymään läpi. Sama optimointi olisi voitu

		suorittaa kuin **routes_from**() kohdassa kuvattiin.
std::vector<std::tuple<StopID, RouteID, Time>> **journey_earliest_arrival** (StopID fromstop, StopID tostop, Time starttime)	~Θ(nlogn)	Funktio käyttää Dijkstran algoritmia hieman poikkeavalla tavalla **journey_shortest_distance**() verrattuna. Oletuksena Dijkstran tarvitsee tutkia solmut vain kertaalleen, mutta aikataulujen takia pysäkkien välillä on monta kaarta eri painoarvoilla. Käytännössä tämä aiheuttaa ongelmia siten, että jotakin pidempää reittiä pitkin voidaan päästä nopeammin seuraavalle pysäkille. Koska pysäkille on päästy nopeammin, saatettaisiin siitä ehtiä eteenpäin aikaisempiin busseihin, joka taas aiheuttaa seuraavien pysäkkien uudelleen tarkistamisen. Uudelleentarkistaminen lisää suoritusaikaa huomattavasti ja turhaa tarkistamista on rajoitettu tarkistamalla aina, pääsisikö viereisille pysäkeille aiemmin. Lisäksi kun aikatauluja kelataan läpi, tarkistus lopetetaan, jos aika olisi jo myöhemmin kuin mitä kohdepysäkille on päästy.~ nlogn on hieman optimistinen kompleksisuus. Kompleksisuuden voisi tarkemmin arvioida olevan (kaaret*bussit+pysäkit)*log(pysäkit).