COMP.CS.300 Tietorakenteet ja algoritmit 1, syksy 2022

Harjoitustyö 2: Rautatiet

Viimeksi päivitetty 17.11.2022

Muutoshistoria

Alla ohjeeseen tehdyt merkittävät muutokset julkaisun jälkeen:

- 17.11. Lisätty huomautus, että next_stations_from-operaatiossa asemat saa palauttaa mielivaltaisessa järjestyksessä.

Sisällys

Harjoitustyön aihe

Toisessa harjoitustyössä ensimmäisen harjoitustyön ohjelmaa laajennetaan käsittelemään kokonaisia junayhteyksiä ja tekemään niihin liittyviä reittihakuja. Osa harjoitustyön operaatioista on pakollisia, osa vapaaehtoisia (pakollinen = vaaditaan läpipääsyyn, vapaaehtoinen = ei-pakollinen, mutta silti osa arvostelua arvosanan kannalta).

Tässä kakkostyön dokumentissa esitellään vain kakkostyön uudet asiat. Tarkoitus on kopioida ykköstyön toteutus kakkostyön pohjaksi ja jatkaa siitä. Kaikki pääohjelman ykköstyön ominaisuudet ja komennot ovat käytössä myös kakkosessa, vaikka niitä ei toisteta tässä dokumentissa. Ykköstyössä arvosteltuja operaatioita ei kuitenkaan arvostella uudelleen, eivätkä ne vaikuta kakkostyön arvosteluun.

Terminologiaa

Tärkeimmät uudet termit

- **Train = juna.** (Kuvaa yhden junan kulkua lähtöasemalta pääteasemalle.) Junalla on yksilöivä *merkkijono-ID* ja lista asemia, joiden läpi juna kulkee, sekä lähtöajat ko. asemilta (ja saapumisaika pääteasemalle).
- **Route = reitti.** Reitti on jono asemia lähtöasemalta kohdeasemalle junayhteyksiä käyttäen. Reitin *pituus* lasketaan (pyöristysvirheiden minimoimiseksi) niin, että etäisyys kahden peräkkäisen aseman välillä ($\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$) **pyöristetään *ensin* alaspäin kokonaisluvuksi**, ja sitten nämä lasketaan yhteen.
- **Cycle = silmukka.** Reitissä on silmukka, jos reittiä kulkemalla päädytään jossain vaiheessa takaisin sellaiselle asemalle, jonka läpi reitti on jo kulkenut.

Huomaa erityisesti seuraavat asiat (myös kaikki harjoitustyön 1 huomioitavat asiat pätevät edelleen):

- *Tämän harjoitustyön uusissa operaatioissa* asymptoottiseen tehokkuuteen ei välttämättä pysty hirveästi vaikuttamaan, koska käytetyt algoritmit määräävät sen. Sen vuoksi harjoitustyössä algoritmien toteutukseen ja toiminnallisuuteen kiinnitetään enemmän huomiota kuin vain asymptoottiseen tehokkuuteen.
- Operaatioiden `route_least_stations`, `route_shortest_distance`, `route_with_cycle` ja `route_earliest_arrival` toteuttaminen ei ole pakollista läpipääsyn kannalta. Ne ovat kuitenkin osa arvostelua. **Vain pakolliset osat toteuttamalla vaiheen maksimiarvosana on 2.**
- Riittävän huonolla toteutuksella työ voidaan hylätä.

Ohjelman toiminta ja rakenne

Osa ohjelmasta tulee valmiina kurssin puolesta, osa toteutetaan itse.

Valmiit osat, jotka tarjotaan kurssin puolesta

Tiedostot *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui* (joihin **EI SAA TEHDÄ *MITÄÄN* MUUTOKSIA)**

Tiedosto *datastructures.hh*

- `class Datastructures`: Luokka, johon harjoitustyö kirjoitetaan. Luokasta annetaan valmiina sen julkinen rajapinta (johon **EI SAA TEHDÄ *MITÄÄN* MUUTOKSIA**, luonnollisesti luokkaan saa private-puolelle lisätä omia jäsenmuuttujia ja -funktioita).

- Tyyppimäärittely `TrainID` (koostuu merkeistä A-Z, a-z, 0-9 ja väliviiva -), jota käytetään junan yksilöivänä tunnisteena.
- Vakiot `NO_TRAIN`, jota käytetään joissain operaatioissa paluuarvona, jos operaatio epäonnistuu.

Tiedosto *datastructures.cc*

- Tänne luonnollisesti kirjoitetaan luokan operaatioiden toteutukset.

Graafisen käyttöliittymän käytöstä

QtCreatorilla käännettäessä harjoitustyön valmis koodi tarjoaa graafisen käyttöliittymän, jolla ohjelmaa voi testata ja ajaa valmiita testejä sekä visualisoida ohjelman toimintaa.

Huom! Käyttöliittymän graafinen esitys kysyy kaikki tiedot opiskelijoiden koodista! **Se ei siis ole "oikea" lopputulos vaan graafinen esitys siitä, mitä tietoja opiskelijoiden koodi antaa.** Jos paikkojen piirto on päällä, käyttöliittymä hakee kaikki asemat operaatiolla `all_stations()` ja kysyy asemien tiedot operaatioilla `get_...()`. Jos alueiden piirtäminen on päällä, ne kysytään operaatiolla `all_regions()`, ja alueen koordinaatit operaatiolla `get_region_coords()`. Jos junien piirto on päällä, ne kysytään joka aseman osalta operaatiolla `next_stations_from()`.

Ohjelman tuntemat komennot ja luokan julkinen rajapinta

Kun ohjelma käynnistetään, se jää odottamaan komentoja, jotka on selitetty alla. Komennot, joiden yhteydessä mainitaan jäsenfunktio, kutsuvat ko. Datastructure-luokan operaatioita, jotka siis opiskelijat toteuttavat. Osa komennoista on taas toteutettu kokonaan kurssin puolesta pääohjelmassa.

Jos ohjelmalle antaa komentoriviltä tiedoston parametriksi, se lukee komennot ko. tiedostosta ja lopettaa sen jälkeen.

Alla operaatiot on listattu siinä järjestyksessä, kun ne suositellaan toteutettavaksi (tietysti suunnittelu kannattaa tehdä kaikki operaatiot huomioon ottaen jo alun alkaen).

Komento **`Julkinen jäsenfunktio`** `(Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)`	**Selitys**
(Kaikki harjoitustyön 1 operaatiot ovat myös saatavilla.)	(Ja tekevät saman asian kuin harjoitustyössä 1.)
`clear_trains` `void clear_trains()`	Poistaa kaikki junat, mutta ei koske asemiin eikä alueisiin. *Tämä operaatio ei ole mukana tehokkuustesteissä.*

Komento **`Julkinen jäsenfunktio`** `(Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)`	**Selitys**
`add_train ID Station1:Time1 CStation2:Time2... StationN:TimeN` `bool add_train(TrainID trainid, std::vector<std::pair<StationID, Time> stationtimes)`	Lisää tietorakenteeseen uuden junan annetulla uniikilla id:llä. Juna kulkee annettujen asemien läpi ja lähtee niiltä annettuina aikoina. Viimeisen aseman aika on saapumisaika pääteasemalle (jolla on merkitystä vain route_earliest_arrival-komennossa). Lähtöajat lisätään myös aseman tietoihin niin, että harjoitustyö 1:n `station_departures_after` näytää ne. Jos annetulla id:llä on jo juna tai jotain asemaa ei löydy, ei tehdä mitään ja palautetaan `false`, muuten palautetaan `true`.
`next_stations_from ID` `std::vector<StationID> next_stations_from(StationID id)`	Palauttaa asemat, jotka ovat heti seuraavia asemia annetun aseman läpi kulkevissa junayhteyksissä. Jos asemalta ei lähde junia, palautetaan tyhjä vektori. Jos id:tä vastaavaa asemaa ei löydy, palautetaan vektori, jonka ainoa alkio on NO_STATION. Pääohjelma järjestää tuloksen id:n mukaan, joten asemat voi palauttaa mielivaltaisessa järjestyksessä. (Pääohjelma kutsuu tätä eri paikoissa.)
`train_stations_from StationID TrainID` `std::vector<StationID> train_stations_from(StationID stationid, TrainID trainid)`	Palauttaa luettelon asemista, joiden läpi annettu juna kulkee lähdettyään annetulta asemalta. Jos id:illä ei löydy asemaa tai junaa, tai juna ei lähde annetulta asemalta, palautetaan vektori, jonka ainoa alkio on NO_STATION.
(Allaolevat kannattaa toteuttaa todennäköisesti vasta, kun ylläolevat on toteutettu.)	
`route_any StationID1 StationID2` `std::vector<std::pair<StationID, Distance>> route_any(StationID fromid, StationID toid)`	Palauttaa jonkin (mielivaltaisen) reitin annettujen asemien välillä. Palautetussa vektorissa on ensimmäisenä alkuasema ja matka 0. Sitten tulevat kaikki reitin varrella olevat asemat, ja kokonaismatka ko. asemaan saakka. Viimeisenä alkiona on kohdeasema ja reitin kokonaismatka. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaakumpaa id:tä ei ole, palautetaan vektori, jonka ainoa alkio on {NO_STATION, NO_DISTANCE}.
(Seuraavien operaatioiden toteuttaminen ei ole pakollista, mutta ne parantavat arvosanaa.)	

Komento **Julkinen jäsenfunktio** (Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)	**Selitys**
`route_least_stations StationID1 StationID2` `std::vector<std::pair<StationID, Distance>>` `route_least_stations(StationID fromid, StationID toid)`	Palauttaa sellaisen reitin annettujen asemien välillä, jossa on mahdollisimman vähän asemia. Jos useilla mahdollisilla reiteillä on yhtä monta asemaa, voi niistä palauttaa minkä tahansa. Palautetussa vektorissa on ensimmäisenä alkuasema ja matka 0. Sitten tulevat kaikki reitin varrella olevat asemat, ja kokonaismatka ko. asemaan saakka. Viimeisenä alkiona on kohdeasema ja reitin kokonaismatka. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaakumpaa id:tä ei ole, palautetaan vektori, jonka ainoa alkio on {NO_STATION, NO_DISTANCE}.
`route_with_cycle StationID` `std::vector<StationID>` `route_with_cycle(StationID fromid)`	Palauttaa annetulta asemalta lähtevän reitin, jossa on sykli, ts. reitti päätyy uudelleen jollekin reitillä jo oleva asemalle. Palautetussa vektorissa on ensimmäisenä alkuasema, sitten tulevat kaikki reitin varrella olevat asemat. Viimeisenä on syklin aiheuttava toiseen kertaan tuleva asema. Jos useilla mahdollisilla reiteillä on sykli, voi niistä palauttaa minkä tahansa. Jos syklistä reittiä ei löydy, palautetaan tyhjä vektori. Jos id:tä vastaavaa asemaa ei ole, palautetaan vektori, jonka ainoa alkio on NO_STATION.
`route_shortest_distance StationID1 StationID2` `std::vector<std::pair<StationID, Distance>>` `route_shortest_distance(StationID fromid, StationID toid)`	Palauttaa annettujen asemien välillä kokonaismatkaltaan mahdollisimman lyhyen reitin. Jos useilla reiteillä on sama kokonaismatka, voi niistä palauttaa minkä tahansa. Palautetussa vektorissa on ensimmäisenä alkuasema ja matka 0. Sitten tulevat kaikki reitin varrella olevat asemat, ja kokonaismatka ko. asemaan saakka. Viimeisenä alkiona on kohdeasema ja reitin kokonaismatka. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaakumpaa id:tä ei ole, palautetaan vektori, jonka ainoa alkio on {NO_STATION, NO_DISTANCE}.

Komento **`Julkinen jäsenfunktio`** `(Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)`	**Selitys**
`route_earliest_arrival StationID1 StationID2 StartTime` `std::vector<std::pair<StationID, Time>>` `route_earliest_arrival(StationID fromid, StationID toid, Time starttime)`	**Huom! Tämä on haastavin ei-pakollinen operaatio!** *(Vinkki: yritä keksiä sopiva kustannusfunktio.)* Palauttaa annettujen asemien välillä reitin, jolla päästään perille mahdollisimman aikaisin. Jos useilla reiteillä on sama saapumisaika, voi niistä palauttaa minkä tahansa. Palautetussa vektorissa on ensimmäisenä alkuasema ja lähtöaika ko. asemalta. Sitten tulevat kaikki reitin varrella olevat asemat, ja lähtöajat niiltä. Viimeisenä alkiona on kohdeasema ja *saapumisaika* sille. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaakumpaa id:tä ei ole, palautetaan vektori, jonka ainoa alkio on {NO_STATION, NO_TIME}. *Huom! operaation ei tarvitse löytää reittejä, joissa vuorokausi vaihtuu kesken reitin.*
(Seuraavat komennot on toteutettu valmiiksi pääohjelmassa.)	**(Tässä mainitaan vain muutokset vaiheen 1 toiminnallisuuteen)**
random_trains *n* (pääohjelman toteuttama)	Lisää tietorakenteeseen (testausta varten) *n* kpl junia, jotka kulkevat satunnaisten asemien välillä. Huom! Arvot ovat tosiaan satunnaisia, eli saattavat olla kerrasta toiseen eri, eivätkä ne graafisesti muodosta järkevää ”karttaa”.
perftest all|compulsory|*cmd1*[;*cmd2*...] *timeout repeat n1*[;*n2*...] (pääohjelman toteuttama)	Ajaa ohjelmalle tehokkuustestit. Tyhjentää tietorakenteen ja lisää sinne *n1* kpl satunnaisia asemia, alueita ja junia. Sen jälkeen arpoo *repeat* kertaa satunnaisen komennon. Mittaa ja tulostaa sekä lisäämiseen että komentoihin menneen ajan. Sen jälkeen sama toistetaan *n2*:lle jne. Jos jonkin testikierroksen suoritusaika ylittää *timeout* sekuntia, keskeytetään testien ajaminen (tämä ei välttämättä ole mikään ongelma, vaan mielivaltainen aikaraja). Jos ensimmäinen parametri on *all*, arvotaan lisäyksen jälkeen kaikista komennoista, joita on ilmoitettu kutsuttavan usein. Jos se on *compulsory*, testataan vain komentoja, jotka on pakko toteuttaa. Jos parametri on lista komentoja, arvotaan komento näiden joukosta (tällöin kannattaa mukaan ottaa myös random_add, jotta lisäyksiä tulee myös testikierroksen aikana). Jos ohjelmaa ajaa graafisella käyttöliittymällä, "stop test" nappia painamalla testi keskeytetään (nappiin reagointi voi kestää hetken).

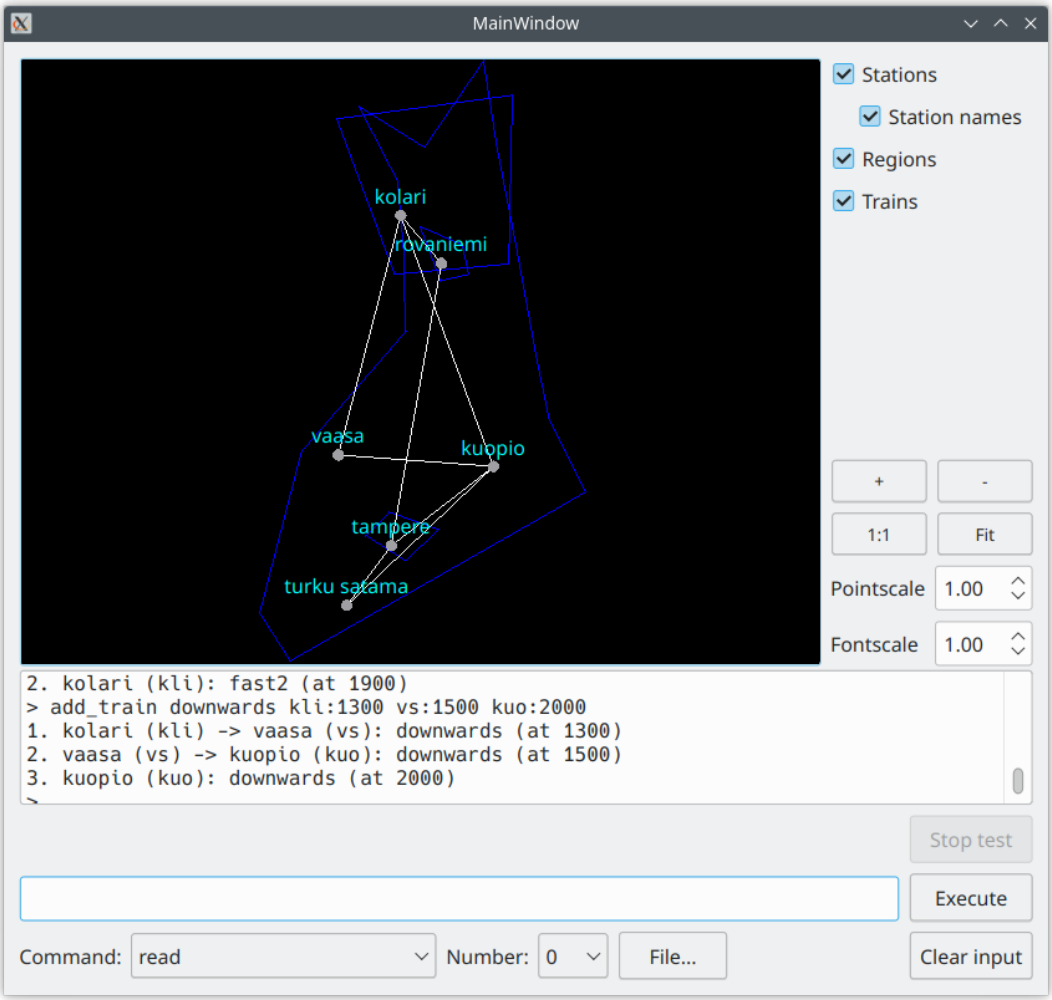
"Datatiedostot"

Kätevin tapa testata ohjelmaa on luoda "datatiedostoja", jotka add-komennolla lisäävät joukon paikkoja, alueita ja väyliä ohjelmaan. Tiedot voi sitten kätevästi lukea sisään tiedostosta read-komennolla ja sitten kokeilla muita komentoja ilman, että tiedot täytyisi joka kerta syöttää sisään käsin. Alla on esimerkit datatiedostoista, joka lisää väyliä sovellukseen:

- *example-trains.txt*

```
# Some imaginary example trains
add_train tukutuku tus:0800 tpe:0900 kuo:1000
add_train upwards tpe:0930 roi:1600 kli:2000
add_train fast1 tus:1000 kuo:1200
add_train fast2 kuo:1100 kli:1900
add_train downwards kli:1300 vs:1500 kuo:2000
```

Kuvakaappaus käyttöliittymästä



Yllä on kuvakaappaus käyttöliittymästä sen jälkeen, kun *example-stations.txt*, *example-regions.txt* ja *example-trains.txt* on luettu sisään.

Esimerkki ohjelman toiminnasta

Alla on esimerkki ohjelman toiminnasta. Esimerkin syötteet löytyvät tiedostoista *example-compulsory-in.txt* ja *example-all-in.txt*, tulostukset tiedostoista *example-compulsory-out.txt* ja *example-all-out.txt.* Eli esimerkkiä voi käyttää pienenä testinä pakollisten toimintojen toimimisesta antamalla käyttöliittymästä komennon
`testread "example-compulsory-in.txt" "example-compulsory-out.txt"`

example-compulsory

```
> clear_all
Cleared all stations
> clear_trains
All trains removed.
> all_stations
No stations!
> read "example-stations.txt" silent
** Commands from 'example-stations.txt'
...(output discarded in silent mode)...
** End of commands from 'example-stations.txt'
> read "example-trains.txt"
** Commands from 'example-trains.txt'
> # Some imaginary example trains
> add_train tukutuku tus:0800 tpe:0900 kuo:1000
1. turku satama (tus) -> tampere (tpe): tukutuku (at 0800)
2. tampere (tpe) -> kuopio (kuo): tukutuku (at 0900)
3. kuopio (kuo): tukutuku (at 1000)
> add_train upwards tpe:0930 roi:1600 kli:2000
1. tampere (tpe) -> rovaniemi (roi): upwards (at 0930)
2. rovaniemi (roi) -> kolari (kli): upwards (at 1600)
3. kolari (kli): upwards (at 2000)
> add_train fast1 tus:1000 kuo:1200
1. turku satama (tus) -> kuopio (kuo): fast1 (at 1000)
2. kuopio (kuo): fast1 (at 1200)
> add_train fast2 kuo:1100 kli:1900
1. kuopio (kuo) -> kolari (kli): fast2 (at 1100)
2. kolari (kli): fast2 (at 1900)
> add_train downwards kli:1300 vs:1500 kuo:2000
1. kolari (kli) -> vaasa (vs): downwards (at 1300)
2. vaasa (vs) -> kuopio (kuo): downwards (at 1500)
3. kuopio (kuo): downwards (at 2000)
>
** End of commands from 'example-trains.txt'
> next_stations_from tpe
1. tampere (tpe) -> kuopio (kuo)
2. tampere (tpe) -> rovaniemi (roi)
> train_stations_from tpe upwards
1. tampere (tpe) -> rovaniemi (roi)
2. rovaniemi (roi) -> kolari (kli)
> route_any tus roi
1. turku satama (tus) -> tampere (tpe) (distance 0)
2. tampere (tpe) -> rovaniemi (roi) (distance 294)
```

```
3. rovaniemi (roi) (distance 1425)
```

example-all

```
> # First read in compulsory example
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
...
** End of commands from 'example-compulsory-in.txt'
> route_least_stations tus kli
1. turku satama (tus) -> kuopio (kuo) (distance 0)
2. kuopio (kuo) -> kolari (kli) (distance 797)
3. kolari (kli) (distance 1853)
> route_with_cycle kuo
1. kuopio (kuo) -> kolari (kli)
2. kolari (kli) -> vaasa (vs)
3. vaasa (vs) -> kuopio (kuo)
4. kuopio (kuo)
> route_shortest_distance tus kli
1. turku satama (tus) -> tampere (tpe) (distance 0)
2. tampere (tpe) -> rovaniemi (roi) (distance 294)
3. rovaniemi (roi) -> kolari (kli) (distance 1425)
4. kolari (kli) (distance 1673)
> route_earliest_arrival tus kli 0700
1. turku satama (tus) -> tampere (tpe) (at 0800)
2. tampere (tpe) -> kuopio (kuo) (at 0900)
3. kuopio (kuo) -> kolari (kli) (at 1100)
4. kolari (kli) (at 1900)
```