COMP.CS.300 Tietorakenteet ja algoritmit 1, syksy 2021

Harjoitustyö 2: Connecting Towns

Viimeksi päivitetty 26.11.2021

Muutoshistoria

Alla ohjeeseen tehdyt merkittävät muutokset julkaisun jälkeen:

- 26.11. Korjattu esimerkin tulostuksissa all_roads:n laskemat teiden pituudet

Sisällys

Harjoitustyön aihe

Harjoitustyön toisessa vaiheessa ensimmäisen vaiheen ohjelmaa laajennetaan käsittelemään myös kaupunkien välisiä tieyhteyksiä 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 kakkosvaiheen dokumentissa esitellään vain *kakkosvaiheen uudet asiat*. Tarkoitus on kopioida ykkösvaiheen toteutus kakkosvaiheen pohjaksi ja jatkaa siitä. *Kaikki pääohjelman ykkösvaiheen ominaisuudet ja komennot ovat käytössä myös kakkosvaiheessa, vaikka niitä ei toisteta tässä dokumentissa.*

Käytännössä kakkosvaiheeseen siirtyminen kannattaa tehdä niin, että kopioi vaiheen 1 datastructures.hh-tiedostosta luokan private-osan ja mahdolliset muut lisäykset vaiheen 2 datastructures.hh-tiedostoon, ja vastaavasti kopioi vaiheen 1 datastructures.cc-tiedostosta kaikkien vaiheen 1 operaatioiden toteutukset vaiheen 2 datastructures.cc-tiedostoon (ja poistaa tuosta tiedostosta valmiina olevat tyhjät toteutukset vaiheen 1 operaatioille). Vaiheen 1

tehokkuuskommentteja ei tarvitse toistaa vaiheessa 2, ellei sitten jonkin vaiheen 1 operaation asymptoottinen tehokkuus muutu vaiheen 2 lisäysten takia.

HUOM! *Tässä harjoitustyössä arvostellaan ainoastaan työn uudet operaatiot (esim. verotukseen liittyviä operaatioita ja nimien aakkostusta ei arvostella enää, koska ne arvosteltiin jo aiemmissa töissä). Luonnollisesti kuitenkin uusien operaatioiden vaatimat perustoiminnot (kaupunkien lisäys, hakeminen yms.) ovat edelleen mukana arvostelussa, erityisesti ne osat joihin on täytynyt tehdä muutoksia vanhojen operaatioiden tukemiseksi.*

Terminologiaa

Menossa olevan englanninkielisen sisarkurssin vuoksi ohjelman käyttöliittymä ja rajapinta ovat englanniksi. Tässä selitys tärkeimmistä 2. vaiheen termeistä:

- **Road = tie.** Tie kulkee aina suoraan kahden kaupungin välillä, ja tietä voi kulkea kumpaan suuntaan tahansa. Tien *pituus* on sama kuin sen päätekaupunkien etäisyys, ts. $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ **pyöristetään alaspäin kokonaisluvuksi**.
- **Route = reitti.** Reitti koostuu jonosta teitä, jotka ”jatkuvat” niin että edellinen tie loppuu aina kaupunkiin, josta seuraava alkaa. Reitin *pituus* on sen sisältämien teiden pituuksien summa. Reitti ei voi palata takaisin samaa tietä, jota kaupunkiin on juuri saavuttu.
- **Cycle = silmukka.** Reitissä on silmukka, jos reittiä kulkemalla päädytään jossain vaiheessa takaisin sellaiseen kaupunkiin, jonka läpi reitti on jo kulkenut.

Harjoitustyössä harjoitellaan valmiiden tietorakenteiden ja algoritmien tehokasta käyttöä (STL), mutta siinä harjoitellaan myös omien algoritmien tehokasta toteuttamista ja niiden tehokkuuden arvioimista (kannattaa tietysti suosia STL:ää omien algoritmien/tietorakenteiden sijaan silloin, kun se on tehokkuuden kannalta järkevää). Toisin sanoen arvostelussa otetaan huomioon valintojen asymptoottinen tehokkuus, mutta sen lisäksi myös ohjelman yleinen tehokkuus (= järkevät ja tehokkaat toteutusratkaisut). ”Mikro-optimoinnista” (tyyliin kirjoitanko ”a = a+b;” vai ”a += b;” tai kääntäjän optimointivipujen säätäminen) ei saa pisteitä.

Tavoitteena on tehdä mahdollisimman tehokas toteutus, kun oletetaan että kaikki ohjelman tuntemat komennot ovat suunnilleen yhtä yleisiä (ellei komentotaulukossa toisin mainita). Usein tehokkuuden kannalta joutuu tekemään joissain tilanteissa kompromisseja. Tällöin arvostelua helpottaa, jos kyseiset kompromissit on dokumentoitu työn osana palautettuun **dokumenttitiedostoon.** (Muistakaa kirjoittaa ja palauttaa myös dokumentti koodin lisäksi!)

Huomaa erityisesti seuraavat asiat (myös kaikki vaiheen 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.

- **Osana ohjelman palautusta tiedostoon datastructures.hh on jokaisen operaation oheen laitettu kommentti, johon lisätään oma arvio kunkin toteutetun operaation asymptoottisesta tehokkuudesta lyhyiden perusteluiden kera.**
- **Osana ohjelman palautusta palautetaan git:ssä myös dokumentti (samassa hakemistossa/kansiossa kuin lähdekoodi), jossa perustellaan toteutuksessa käytetyt tietorakenteet ja ratkaisut tehokkuuden kannalta. Hyväksyttäviä dokumentin formaatteja ovat puhdas teksti (readme.txt), markdown (readme.md) ja Pdf (readme.pdf).**
- Operaatioiden `remove_road`, `least_towns_route`, `shortest_route`, `road_cycle_route` ja `trim_road_network` 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.

Graafisen käyttöliittymän käytöstä

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 kaupungit operaatiolla `all_towns()` ja kysyy kaupunkien tiedot operaatioilla `get_...()`. Jos vasallisuhteiden piirtäminen on päällä, ne kysytään operaatiolla `get_town_vassals()`. Jos teiden piirto on päällä, ne kysytään operaatioilla `get_roads_from()`.

Harjoitustyönä toteutettavat osat

Tiedostot *datastructures.hh* ja *datastructures.cc*

- `class Datastructures`: Luokan julkisen rajapinnan jäsenfunktiot tulee toteuttaa. Luokkaan saa lisätä omia määrittelyitä (jäsenmuuttujat, uudet jäsenfunktiot yms.)
- **Tiedostoon *datastructures.hh* kirjoitetaan jokaisen toteutetun operaation yläpuolelle kommentteihin oma arvio ko. operaation toteutuksen asymptoottisesti tehokkuudesta ja lyhyt perustelu arviolle.**

Lisäksi harjoitustyönä toteutetaan alussa mainittu dokumentti readme.pdf.

Huom! Omassa koodissa ei ole tarpeen tehdä ohjelman varsinaiseen toimintaan liittyviä tulostuksia, koska pääohjelma hoitaa ne. Mahdolliset Debug-tulostukset kannattaa tehdä cerr-virtaan (tai `qDebug`:lla, jos käytät Qt:ta), jotta ne eivät sotke testejä.

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 vaiheen 1 operaatiot ovat myös saatavilla.)	(Ja tekevät saman asian kuin vaiheessa 1.)
`clear_all` `void clear_all();`	Tyhjentää tietorakenteet eli poistaa kaikki kaupungit **ja tiet** (tämän jälkeen `town_count` palauttaa 0 **ja all_roads palauttaa tyhjän listan**).
`clear_roads` `void clear_roads()`	Poistaa kaikki tiet, mutta ei koske kaupunkeihin eikä vasallisuhteisiin. *Tämä operaatio ei ole mukana tehokkuustesteissä.*
`all_roads` `std::vector<std::pair<TownID, TownID>> all_roads()`	Palauttaa kaikki tietorakenteessa olevat tiet mielivaltaisessa järjestyksessä, ts. järjestyksellä ei ole väliä (pääohjelma järjestää ne id:n mukaan). Jokainen tie on listassa vain kerran, ja esitetään parina kaupunki-id:itä, jotka esitetään siinä järjestyksessä, että parin ensimmäinen id on pienempi kuin toinen id. *Tämä operaatio ei ole oletuksena mukana tehokkuustesteissä.*
`add_road` ***`ID1 ID2`*** `bool add_road(TownID town1, TownID town2)`	Lisää annettujen kaupunkien välille tien. Jos jompaa kumpaa kaupunkia ei löydy tai niiden välillä on jo tie, ei tehdä mitään ja palautetaan `false`, muuten `true`.
`roads_from` ***`ID`*** `std::vector<TownID> get_roads_from(TownID id)`	Palauttaa mielivaltaisessa järjestyksessä listan kaupungeista, joihin annetusta kaupungista menee suoraan tie. Jos annettua ID:tä vastaavaa kaupunkia ei löydy, palautetaan ainoana alkiona NO_TOWNID.

Komento **`Julkinen jäsenfunktio`** `(Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)`	**Selitys**
(Allaolevat kannattaa toteuttaa todennäköisesti vasta, kun ylläolevat on toteutettu.)	

`any_route ID1 ID2`
`std::vector<TownID>`
`any_route(TownID fromid, TownID toid)`
	Palauttaa jonkin (mielivaltaisen) reitin annettujen kaupunkien välillä. Palautetussa vektorissa on ensimmäisenä lähtökaupunki. Sitten tulevat kaikki reitin varrella olevat kaupungit, ja viimeisenä alkiona kohdekaupunki. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jommallakummalla id:llä ei löydy kaupunkia, palautetaan yksi alkio NO_TOWNID.

(Seuraavien operaatioiden toteuttaminen ei ole pakollista, mutta ne parantavat arvosanaa.)	
`remove_road ID1 ID2` `bool remove_road(TownID town1, TownID town2)`	Poistaa annettujen kaupunkien väliltä tien. Jos jompaakumpaa kaupunkia ei löydy tai niiden välillä ei ole tietä, ei tehdä mitään ja palautetaan `false`, muuten `true`.

`least_towns_route ID1 ID2`
`std::vector<TownID>`
`least_towns_route(TownID fromid, TownID toid)`
	Palauttaa sellaisen reitin annettujen kaupunkien välillä, jossa on mahdollisimman vähän kaupunkeja. Jos useilla reiteillä on yhtä monta kaupunkia, voi niistä palauttaa minkä tahansa. Palautetussa vektorissa on ensimmäisenä lähtökaupunki. Sitten tulevat kaikki reitin varrella olevat kaupungit ja viimeisenä alkiona kohdekaupunki. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jommallakummalla id:llä ei löydy kaupunkia, palautetaan yksi alkio {NO_TOWNID}.

`road_cycle_route ID`
`std::vector<TownID>`
`road_cycle_route(TownID startid)`
	Palauttaa annetusta kaupungista lähtevän reitin, jossa on sykli, ts. reitti päätyy uudelleen johonkin reitillä jo olevaan kaupunkiin. Palautetussa vektorissa on ensimmäisenä lähtökaupunki. Sitten tulevat kaikki reitin varrella olevat kaupungit ja viimeisenä on syklin aiheuttava toiseen kertaan tuleva kaupunki. Jos syklistä reittiä ei löydy, palautetaan tyhjä vektori. Jos id:llä ei löydy kaupunkia, palautetaan yksi alkio {NO_TOWNID}. Jos syklisiä reittejä on useita, palautetaan mikä tahansa niistä. Huom 1! Sykliksi ei lasketa sitä, että palataan päinvastaiseen suuntaan takaisin tietä, jota ollaan juuri kuljettu. Huom 2! Tulostuksen pitämiseksi yksikäsitteisenä *pääohjelma* tarvittaessa kääntää löytyneen syklin niin, että se alkaa pienemmällä kaupunki-id:llä.

Komento **Julkinen jäsenfunktio** (Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)	**Selitys**

`shortest_route ID1 ID2`
`std::vector<TownID>`
`shortest_route(TownID fromid, TownID toid)`
	Palauttaa annettujen kaupunkien välillä kokonaismatkaltaan mahdollisimman lyhyen reitin. Jos useilla reiteillä on sama kokonaismatka, voi niistä palauttaa minkä tahansa. Palautetussa vektorissa on ensimmäisenä lähtökaupunki. Sitten tulevat kaikki reitin varrella olevat kaupungit ja Viimeisenä alkiona on kohdekaupunki. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jommallakummalla id:llä ei löydy kaupunkia, palautetaan yksi alkio {NO_TOWNID}.

`trim_road_network` `Distance trim_road_network()`	Karsii tiet (=poistaa niitä) niin, että jäljelle jätetään tiet, joiden yhteenlaskettu pituus on mahdollisimman pieni, mutta joilla edelleen löytyy reitti kaikkien sellaisten kaupunkien välillä, joiden välillä oli ennenkin reitti. Muut tiet poistetaan. Paluuarvona palautetaan tuloksena olevan tieverkoston kokonaispituus. Jos olemassa useita kokonaispituudeltaan yhtä lyhyitä tieverkostoja, mikä tahansa niistä kelpaa. **Huom! Tämä on haasteeksi tehty bonusoperaatio, jonka algoritmia ei suoraan löydy kurssin videoilta, vaikka algoritmi mainitaan muualla kurssin materiaalissa.**
(Seuraavat komennot on toteutettu valmiiksi pääohjelmassa.)	**(Tässä mainitaan vain muutokset vaiheen 1 toiminnallisuuteen)**
random_roads n (pääohjelman toteuttama)	Lisää kaupunkien välille maksimissaan n satunnaista tietä niin, että arvotut tiet eivät risteä keskenään (käyttöliittymän selkeyttämiseksi). Teitä saatetaan lisätä myös vähemmän. *Tehokkuustestissä tämä komento lisää aina maks. 10 tietä.*
random_road_network (pääohjelman toteuttama)	Lisää kaupunkien välille satunnaisen tieverkoston, jossa jokaisesta kaupungista pääsee toiseen ja tiet eivät risteä keskenään.

Komento **`Julkinen jäsenfunktio`** `(Komennoissa ei-pakolliset parametrit hakasuluissa ja vaihtoehdot erotettu |-merkillä.)`	**Selitys**
perftest all|compulsory|*cmd1*[;*cmd2*...] ***timeout repeat n1*[;*n2*...]** (pääohjelman toteuttama)	Ajaa ohjelmalle tehokkuustestit. Tyhjentää tietorakenteen ja lisää sinne *n1* kpl satunnaisia kaupunkeja **ja teitä** (ks. random_add). Sen jälkeen arpoo *n* 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 kaupunkeja ja teitä 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-data.txt*

```
# Adding towns
add_town Hki Helsinki (3,0) 3
add_town Tpe Tampere (2,2) 4
add_town Ol Oulu (3,7) 10
add_town Kuo Kuopio (6,3) 9
add_town Tku Turku (1,1) 2
# Adding crossroads as extra towns
add_town x1 xx (3,3) 6
add_town x2 xy (4,4) 8
# Adding roads
add_road Tpe x1
# add_road x1 x2
add_road x2 Ol
add_road Ol Kuo
```

```
add_road Tpe Kuo
add_road Hki Tpe
add_road Tpe Tku
```

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-in.txt" "example-out.txt"`

```
> clear_all
Cleared all towns
> clear_roads
All roads removed.
> read "example-data.txt"
** Commands from 'example-data.txt'
> # Adding towns
> add_town Hki Helsinki (3,0) 3
Helsinki: tax=3, pos=(3,0), id=Hki
> add_town Tpe Tampere (2,2) 4
Tampere: tax=4, pos=(2,2), id=Tpe
> add_town Ol Oulu (3,7) 10
Oulu: tax=10, pos=(3,7), id=Ol
> add_town Kuo Kuopio (6,3) 9
Kuopio: tax=9, pos=(6,3), id=Kuo
> add_town Tku Turku (1,1) 2
Turku: tax=2, pos=(1,1), id=Tku
> # Adding crossroads as extra towns
> add_town x1 xx (3,3) 6
xx: tax=6, pos=(3,3), id=x1
> add_town x2 xy (4,4) 8
xy: tax=8, pos=(4,4), id=x2
> # Adding roads
> add_road Tpe x1
Added road: Tampere <-> xx
> # add_road x1 x2
> add_road x2 Ol
Added road: xy <-> Oulu
> add_road Ol Kuo
Added road: Oulu <-> Kuopio
> add_road Tpe Kuo
Added road: Tampere <-> Kuopio
> add_road Hki Tpe
Added road: Helsinki <-> Tampere
> add_road Tpe Tku
Added road: Tampere <-> Turku
>
** End of commands from 'example-data.txt'
> all_roads
```

```
1: Hki <-> Tpe (2)
2: Kuo <-> Ol (5)
3: Kuo <-> Tpe (4)
4: Ol <-> x2 (3)
5: Tku <-> Tpe (1)
6: Tpe <-> x1 (1)
> roads_from Tpe
1. Helsinki: tax=3, pos=(3,0), id=Hki
2. Kuopio: tax=9, pos=(6,3), id=Kuo
3. Turku: tax=2, pos=(1,1), id=Tku
4. xx: tax=6, pos=(3,3), id=x1
> any_route Tku Hki
1. Turku
2. Tampere (distance 1)
3. Helsinki (distance 3)
> # Non-compulsory operations
> # First add a road to create more routes
> add_road x1 x2
Added road: xx <-> xy
> least_towns_route Hki Ol
1. Helsinki
2. Tampere (distance 2)
3. Kuopio (distance 6)
4. Oulu (distance 11)
> road_cycle_route Hki
1. Helsinki
2. Tampere (distance 2)
3. Kuopio (distance 6)
4. Oulu (distance 11)
5. xy (distance 14)
6. xx (distance 15)
7. Tampere (distance 16)
> shortest_route Hki Ol
1. Helsinki
2. Tampere (distance 2)
3. xx (distance 3)
4. xy (distance 4)
5. Oulu (distance 7)
> remove_road Tpe Hki
Removed road: Tampere <-> Helsinki
> roads_from Tpe
1. Kuopio: tax=9, pos=(6,3), id=Kuo
2. Turku: tax=2, pos=(1,1), id=Tku
3. xx: tax=6, pos=(3,3), id=x1
> all_roads
1: Kuo <-> Ol (5)
2: Kuo <-> Tpe (4)
3: Ol <-> x2 (3)
4: Tku <-> Tpe (1)
5: Tpe <-> x1 (1)
6: x1 <-> x2 (1)
> add_road Hki Tpe
Added road: Helsinki <-> Tampere
> trim_road_network
```

```
The remaining road network has total distance of 12
> all_roads
1: Hki <-> Tpe (2)
2: Kuo <-> Tpe (4)
3: Ol <-> x2 (3)
4: Tku <-> Tpe (1)
5: Tpe <-> x1 (1)
6: x1 <-> x2 (1)
```