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

Harjoitustyö 2: Connecting Towns

Aapo Kärki
H292001

Koodissa kommentointi on tehty englanniksi oman selkeyden vuoksi, mutta tämä dokumentti suomeksi helppouden vuoksi. Pahoittelut, jos tästä tulee ongelmaa.

Toisen palautuksen funktioiden selitykset
Muutetut 1. osan funktiot

```
remove_town id bool remove_town(TownID id);
```

Tieyhteyksien myötä joudun myös poistamaan tieyhteydet, kun kaupunki katoaa. Tämä onnistuu vähän hitaasti, sillä tieyhteyksiä on useassa paikassa, mutta tämä on ensimmäisen osan funktio ja sen nopeudella ei ole väliä

```
clear_all void clear_all();
```

Ensimmäisen palautuksen clear_all funktio eroaa tästä muutamalla tavalla. Nyt tyhjennetään road_-tietorakenteet kaiken muun lisäksi. Ohjelma ei tästä oikeastaan hidastu alkuperäisestä. Yksi logaritminen operaatio lisää.

Muissa 1. osan funktioissa vain bugikorjauksia.

2. Osan tietorakenteet

Town* struct uutuudet

std::unordered_set<Town*> roads_ kaupungin tieyhteyksien tietorakenteena. Vektori tämän tietorakenteena toimisi yhtä hyvin ja vektorista iteroiminen olisi ”järkevämpää”. Toisaalta unordered_set yksinkertaistaa koodia suunnattomasti. nopeuttaa melkein jokaisen funktion alussa olevaa find-komentoa logaritmiseksi verrattuna vectorin lineaariseen find-komentoon.
Helpottaa teiden lisäämistä, sillä tien olemassaoloa ei tarvise tarkistaa. Sama poistamisessa. add_road ja remove_road ovat nopeampia.

Unordered_setin iteroiminen on kuitenkin aika lailla tismalleen yhtä tehokasta, kuin vectorin.

int group pitää kirjaa kaupungin ryhmästä/väristä/olotilasta. any_route, least_towns_route, road_cycle_route ja shortest_route tarvitsevat jonkin flagin, joka pitää yllä, onko kaupungissa käyty (harmaa), onko se valmis (musta) tai koskematon (valkoinen). Näitä edustavat luvut valkoinen = 0, harmaa = 1 ja musta = 2.

Tämän olisi voinut tehdä selkeämmin enum- rakenteella, mutta trim_road_network tarvitsee useampia ryhmiä, joita Kruskalin algoritmi voi tarkastella funktiossa. Tämän takia group on vain kokonaislukuna, jotta Kruskalin algoritmi voisi käyttää jo olemassaolevaa muuttujaa Town* structissa.

int d = std::numeric_limits<int>::max() ja
int de = std::numeric_limits<int>::max() ovat käytössä shortest_route funktioss, jossa täytyy vertailla kaupunkien välistä etäisyyttä toisistaan. Muuttujat ovat asetettu int:in maksimiluvuksi, sillä etäisyyksien täytyy olla aluksi äärettömiä, jotta A* -algoritmi toimii.

roadnetwork tietorakenteesta lisää trim_road_network function selityksessä

std::vector<TownID> pi = {} käytetään useassa algoritmissa ja siinä pidetään yllä ’jo-käydyn’ tiereitin kaupungit. Jokaisessa algoritmien kierrossa, tähän lisätään kyseisen kaupungin ”isäkaupungin” pi-vektori. Lopussa, kun esim päätekaupunki tai silmukka on löydetty. On funktiossa valmiina jo vektori, jossa on kaikki kaupungit, jotka käytiin läpi, jotta päästiin siihen.

2. osan funktiot

```
clear_roads void clear_roads();
```
Tyhjentään jokaisen kaupungin road_-tietorakenteet ja roadnetwork-rakenteen. Millään kaupungilla ei ole minkäänlaista tieyhteyttä muihin kaupunkeihin.
Tehokkuus: Average: O(n*k)
Worst: O(n^2)

```
all_roads std::vector<std::pair<TownID, TownID>> all_roads();
```
Funktiossa on kaksi for-looppia sisäkkäin, jossa insertataan sisimmässä loopissa kerran unordered_set tietorakenteeseen. **VAIKKA** funktiossa on 2 for looppia on toteutus yllättävän nopea. Kaupunkeja käydään läpi niin monta kertaa kuin on tieyhteyksiä. Jos jokaisesta kaupungista on suora tieyhteys jokaiseen kaupunkiin, niin silloin O(n^2). Inserttaaminen unordered_settiin on yleisesti O(1), mutta pahimmillaan O(n).
Tässä funktiossa unordered_set tulee hyötyyn, kun ei tarvitse huolehtia samojen teiden olevan kahta kertaa tietorakenteessa.
Lopussa tämä muutetaan vectoriksi. Operaation tehokkuus on O(k), jossa k on teiden määrä.
Tehokkuus: Worst case O(n^2*log(n)), jossa n = towns.size().
Average 2 O(k) = O(n*k*log(n)), jossa k vastaa teiden määrää

```
add_road ID1 ID2 bool add_road(TownID town1, TownID town2)
```

Aluksi etsitään unordered_mapista town1 ja town2, tallennetaan muuttujiin niitä vastaavat Town* structit, jotta voidaan viitata Town* sisältöön osoittimilla.
Samalla, jos jompikumpi saa arvon towns.end, sitä ei ole olemassa ja funktiota ei käydä läpi.

Insertataan molempien kaupunkien roads_tietorakenteeseen toinen kaupunki. Näin pidetään yllä mistä kaupungeista on tieyhteys toiseen. Inserttaus on yleisesti O(1), pahimmillaan O(n).

Lopuksi lisätään roadnetwork settiin kaupunkien välinen etäisyys ja molemmat kaupungit.

roadnetwork on käytössä vain ja ainoastaan **trim_road_network** funktiossa **Kruskalin** algoritmin takia. Roadnetworkin insert on logaritminen. **Tämä hidastaa add_road funktiota**, joka hidastaa perftestien lisäysosioita hiukan. N = 300000 lisäyksen kohdalla hidastuminen on noin **0.2 s**.
Jokainen funktio timeouttaa kuitenkin samalla N:n määrällä riippumatta onko tämä lisäys koodissa vai ei. **Tämä on kompromissi, mutta koen sen olevan oikeutettu.**

Tehokkuus: Worst case: 2O(n) + O(log(n)) = O(n+ log(n))
Average case: O(log(n))

```
roads_from ID std::vector<TownID> get_roads_from(TownID id)
```

Kerää kaikki suorat tieyhteydet yhdestä kaupungista.
Koska tiet Town* structeina ovat unordered_set tietorakenteessa, täytyy ne siirtää vectoriin id:inä yhden for loopin O(k) ja push_backin O(1) avulla.
Tehokkaimmillaan tieyhteydet olisivat id:inä vectorissa jo valmiiksi, mutta tällöin muu koodi vaikeutuisi ja tehokkuus saattaisi kärsitä.

Tehokkuus: O(k), jossa k kaupungin id:n tieyhteyksien määrä.

```
any_route ID1 ID2 std::vector<TownID> any_route(TownID fromid, TownID
toid)
```

Palauttaa jonkun reitin kahden kaupungin väliltä. Koodipohja on sama kuin least_towns_routen muokattu BFS algoritmi.

Testien perusteella std::list<Town*> q, toimii nopeiten, kun se imitoi first-in-first-out rakennetta eli jonoa. Listalla ja quella on tismalleen samat tehokkuudet tarvittavien toimintojen suhteen.
Jos muokataan q last-in-first-out rakenteeksi eli stackiksi, niin nopeus kärsii suunnattomasti perftesteissä ja any_route toimisi DFS:n tapaan.
Stack ja DFS -tyylinen algoritmi ovat hitaampia, sillä funktio voi tarkistaa yhden tien loppuun asti, vaikka haluttu kaupunki olisin alkukaupungin vieressä. BFS katsoo leveyden ensin, jolloin tätä ei voi tapahtua.

while-silmukassa käydään kaikki kaupungit läpi O(n), ja sen sisällä olevassa for_silmukassa käydään lopulta kaikki kaaret läpi O(k), eli worst case asymptoottinen tehokkuus on O(n+k). **Funktio loppuu, kun loppukaupunki on löydetty. Tämä nopeuttaa funktion tehokkuutta, mutta asymptoottinen tehokkuus on silti O(n+k).**

q -listan pop ja push komennot ovat O(1), samoin pi:n vectoriin push_back on O(1).

Lopuksi resetoidaan kaikki Town* olevat arvot, joita käytettiin vain tilapäisesti funktiossa. pi.clear() poistetaan luodut tieyhteydet joita on k määrä O(k). For loopissa, jossa käydään kaikki kaupunkit läpi O(n). Tehokkuus O(n+k)

Tehokkuus: O(n+k)

```
remove_road ID1 ID2bool remove_road(TownID town1, TownID town2)
```

Poistetaan tie. Kaupunkien yhteys toisista poistetaan ja samoin tie roadnetworkista.
Moni operaatioista on worst case O(n) ja yleisesti O(1). Yksi logaritminen find operaatio setille. set.erase on amortisoitu constant, jos poistetaan iteraattorin avulla eikä valuella.

Tehokkuus: Average case: O(log(n))
Worst case: O(n+ log(n)))

```
least_towns_route ID1 ID2 std::vector<TownID> least_towns_route(TownID
fromid, TownID toid)
```

Funktio on tismalleen sama kuin any_route. Koin tämän olevan nopein ratkaisu, minkä keksin, joten laitoin sen myös any_routeen. Funktio on muokattu BFS -algoritmi, joka tallentaa käydyn reitin kyseisen kaupungin pi-vektoriin ja kun haluttu päätekaupunki on löydetty, se palauttaa tähän mennessä saadun pi-networkin, jossa on least_town_route.

while-silmukassa käydään kaikki kaupungit läpi O(n), ja sen sisällä olevassa for_silmukassa käydään lopulta kaikki kaaret läpi O(k), eli worst case asymptoottinen tehokkuus on O(n+k). **Funktio loppuu, kun loppukaupunki on löydetty. Tämä nopeuttaa funktion tehokkuutta, mutta asymptoottinen tehokkuus on silti O(n+k).**

q -listan pop ja push komennot ovat O(1), samoin pi-vectorin push_back on O(1).

Lopuksi resetoidaan kaikki Town* olevat arvot, joita käytettiin vain tilapäisesti funktiossa. pi.clear() poistetaan luodut tieyhteydet joita on k määrä O(k). For loopissa, jossa käydään kaikki kaupunkit läpi O(n). Tehokkuus O(n+k)

Tehokkuus: O(n+k)

```
road_cycle_route ID std::vector<TownID> road_cycle_route(TownID startid)
```

Funktiossa hyödynnetään DFS;n ominaisuutta löytää silmukoita tieverkossa. Kun kyseisen kaupungin u, naapuri v on jo vierailtu (group = 1) tiedetään, että silmukka on löydetty. Lisätään siis palautettavaan vektoriin loput tarvittavat kaupungit ja uudestaan kaupunki v.

Asymptoottinen tehokkuus on sama kuin any_routen ja least_towns_routen O(n+k). While loop, jossa on for-loop. n = kaupunkien määrä ja k = tieyhteyksien määrä.

Tämä funktio on yleisesti valmis nopeammin kuin any_route ja least_towns_route, sillä tämä funktio päättyy, kun silmukka on löydetty. Se näkyy myös perftesteissä.

Tehokkuus: O(n+k)

```
shortest_route ID1 ID2 std::vector<TownID> shortest_route(TownID fromid, TownID toid)
```

Suoraan implementaatio kurssivideon A* -algoritmista. Relax-funktio ei ole erikseen erään toiminnallisuuden helpottamiseksi. Tehokkuus on aluksi sama kuin BFS ja DFS, jossa O(n+k), n= kaupunkien määrä ja k= teiden määrä, mutta nyt funktiossa lisätään prioriteettijonoon alkioita, jonka tehokkuus on O(log(n)). Tästä saadaan funktion tehokkuudeksi O((n+k)log(n)).

Algoritmi päättyy, kun toid löydetään, jolloin se päättyy kesken. Se käy pahimmassa tapauksessa koko tieverkon läpi.
A*: asymptoottinen tehokkuus on täsmälleen sama kuin Djikstran algoritmilla, mutta A* osoittaa oikean suunnan, toid:seen joka saattaa nopeuttaa algoritmia. Varsinkin tieverkko/kartta -malleissa tämä on järkevämpää, sillä yleensä lähimmät kaupungit yhdistyneet tiellä toisiinsa. A*:n linnuntiearviosta on siis paljon hyötyä.

Tehokkuus: O((n+k)log(n))

```
trim_road_network Distance trim_road_network()
```

https://www.youtube.com/watch?v=JZBQLXgSGfs&t=210s

Funktio on toteutettu Kruskalin minimum spanning tree algoritmilla, yllä olevan ohjevideon avulla.

Tämä funktio on kaikista raskain ja sen toiminta vahingoittaa add_road ja remove_road funktioita hiukan. Funktiossa käytetään roadnetwork -nimistä settiä, johon on tallennettuna pair<int, pait<Town*, Town*>>. Kaksi kaupunkia ja niiden välisen tien pituus. Set järjestää parit tien pituuden mukaan suuruusjärjestykseen.

Funktion aluksi kopioidaan nykyinen roadnetwork, poistetaan kaikki tiet ja tyhjennetään roadnetwork. Tämä mahdollistaa uusien (pelkästään optimaalien) teiden lisäämisen ohjelmaan.

Algoritmi käy läpi roadnetwork -setin kopiota, josta jokaisella loopilla lisätään add_road komennolla lyhin tie takaisin ohjelmaan. Tiet käydään läpi, jos kaupungit ovat uusia (group = 0), annetaan niille ryhmä muuttujan i mukaan. Jos myöhemmin tien kaupungeista toinen on jo ryhmässä, lisätään siihen ryhmään se toinen kaupunki, joka ei ole. Jos myöhemmin molemmat kaupungit ovat ryhmissä, lisätään uudempi ryhmä vanhempaan. Vanhemmassa ryhmässä on

todennäköisesti enemmän muutettavaa, joten se olisi hitaampaa. Tähän toteutukseen olisi järkevintä vertailla ryhmien kokoja, mutta kaikki yritykseni luoda sellaista olivat ultimaattisesti kuitenkin hitaampia, kuin nykyinen versio.

Ryhmiä vertaillaan, jotta vältytään silmukoilta ja turhilta lisäyksiltä.

Tätä vertailua ja looppia käydään läpi niin monta kertaa, kunnes i > n*(n-1)/2, joka on matemaattisesti pienin määrä yhteyksiä n määrän nodeja välillä, niin että kaikki yhdistyvät. Algoritmi loppuu ja jäljellä on trimmattu tieverkko.

Algoritmin tehokkuuskäyrä perftestien mukaan olisi N^2 mukainen tai vastaava. Algoritmin toiminnoista voidaan päätellä tehokkuukseksi kuitenkin

Tehokkuus: O(n*r*log(n)), jossa n= kaupungit, r = kaupungit eri ryhmissä ja log(n) = add_road

Tehokkuus hidastuu mitä pidemmälle päästään ja mitä enemmän r täyttyy. Lopulta hidastuminen muistuttaa **O(n^2*log(n)).**

Iso selitys roadnetworkista:

Toteutin roadnetworkin täyttämisen add_road-funktiossa, koska siitä aiheutuva add_roadin hidastuminen on melko pieni. Testasin roadnetworkin luomista pelkästään tässä funktiossa, mutta se hidasti perftestiä suuresti. **Koen, että setin aiheuttamat tehokkuudeltaan logaritmiset operaatiot remove_road ja add_road -funktioissa ovat järkevämpiä, kuin koko trim_roadnetworkin hidastuminen niiden takia.**

Ensimmäinen versioni oli setin sijaan prioriteettijonoa. Prioriteettijonon operaatiot top() ja pop() olisivat O(1). Toisaalta jouduin tekemään **custom prioriteettijonon**, sillä on tarvetta poistaa teitä prioriteettijonon keskeltä. Tämä operaatio ensin löytäisi poistettavan alkion, poistaisi sen, mutta sitten keko täytyisi koota uudelleen, jonka tehokkuus on 3*O(n). Tämä hidasti remove_road funktiota yli kolminkertaisesti.

Toinen versioni käyttää settiä, jota iteroin Kruskalissa pienemmästä tiestä suurempaan. Setin ja prioriteettijonon lisäysoperaatiot push() ja insert() ovat molemmat O(log(n)). Setistä voi toisaalta poistaa alkioita myös O(log(n)) tehokkuudella. Remove_road hidastuu nyt O(log(n)) verrattuna 3*O(n):ään.

Kaikki 2. osan perftestit etätyöpöydälllä linuxilla.

Funktiot ovat raskaita suurilla N määrillä, joten ohjelmaikkuna lagaa aika paljon. Se ei kuitenkaan kaadu.

roads_from

```
      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000145382 ,  0.000244968 ,   0.00039035
     30 ,   0.00029096 ,  0.000286152 ,  0.000577112
    100 ,  0.000965562 ,  0.000238443 ,     0.001204
    300 ,   0.00311946 ,  0.000273604 ,   0.00339306
   1000 ,     0.012314 ,  0.000240887 ,    0.0125549
   3000 ,    0.0351655 ,  0.000250766 ,    0.0354163
  10000 ,      0.13525 ,  0.000309221 ,     0.135559
  30000 ,     0.425113 ,  0.000328506 ,     0.425442
 100000 ,      1.58851 ,  0.000322656 ,      1.58884
 300000 ,      5.22911 ,  0.000358933 ,      5.22947
1000000 ,      19.9225 ,  0.000409273 ,      19.9229
>
```

any_route random_add random_roads

```
      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000113354 ,   0.00200458 ,   0.00211793
     30 ,  0.000299064 ,   0.00236091 ,   0.00265998
    100 ,  0.000883146 ,   0.00293943 ,   0.00382258
    300 ,   0.00272744 ,   0.00654987 ,   0.00927731
   1000 ,    0.0100476 ,     0.018347 ,    0.0283945
   3000 ,     0.031815 ,    0.0728376 ,     0.104653
  10000 ,     0.120943 ,     0.436176 ,     0.557119
  30000 ,     0.416484 ,      1.57661 ,       1.9931
 100000 ,      1.56672 ,      3.41772 ,      4.98444
 300000 ,       5.1682 , Timeout!
>
```

least_towns_route random_add random_roads

```
      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000112888 ,    0.0647393 ,    0.0648521
     30 ,  0.000366378 ,    0.0724658 ,    0.0728322
    100 ,    0.0010784 ,    0.0873041 ,    0.0883825
    300 ,   0.00320467 ,     0.117146 ,     0.120351
   1000 ,    0.0102364 ,     0.258175 ,     0.268411
```

```
   3000 ,    0.0454571 ,     0.677981 ,     0.723438
  10000 ,     0.117363 ,      4.59043 ,      4.70779
  30000 ,     0.443395 ,      14.6391 ,      15.0825
 100000 ,      1.57711 , Timeout!
>

remove_road

      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,   0.00012254 ,   0.00291976 ,    0.0030423
     30 ,  0.000336869 ,   0.00307174 ,    0.0034086
    100 ,  0.000987798 ,   0.00325056 ,   0.00423835
    300 ,   0.00399301 ,    0.0034521 ,   0.00744511
   1000 ,    0.0110636 ,   0.00400956 ,    0.0150731
   3000 ,    0.0317426 ,   0.00477039 ,     0.036513
  10000 ,     0.134361 ,   0.00487154 ,     0.139233
  30000 ,     0.420095 ,   0.00577136 ,     0.425867
 100000 ,       1.5524 ,   0.00605804 ,      1.55846
 300000 ,      5.12256 ,   0.00699569 ,      5.12956
1000000 ,      19.5014 ,   0.00708621 ,      19.5085
>

road_cycle_route random_add random_roads

      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000103929 ,    0.0226477 ,    0.0227517
     30 ,  0.000350058 ,    0.0231105 ,    0.0234606
    100 ,  0.000930763 ,    0.0245957 ,    0.0255265
    300 ,   0.00283225 ,    0.0286677 ,       0.0315
   1000 ,   0.00969933 ,    0.0461705 ,    0.0558699
   3000 ,    0.0425581 ,     0.096608 ,     0.139166
  10000 ,     0.129943 ,     0.288869 ,     0.418812
  30000 ,     0.422801 ,       1.2281 ,       1.6509
 100000 ,      1.64073 ,      3.87993 ,      5.52066
 300000 ,      5.05377 ,      9.87942 ,      14.9332
1000000 ,      19.0819 , Timeout!
>

shortest_route random_add random_roads
```

```
      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000116736 ,     0.176538 ,     0.176654
     30 ,  0.000322556 ,     0.180321 ,     0.180644
    100 ,  0.000914987 ,     0.177863 ,     0.178778
    300 ,   0.00294071 ,     0.307222 ,     0.310163
   1000 ,   0.00994798 ,     0.743937 ,     0.753885
   3000 ,    0.0320859 ,      1.87952 ,      1.91161
  10000 ,     0.121567 ,      7.60714 ,       7.7287
  30000 ,     0.407056 , Timeout!
>

trim_road_network random_add random_roads

      N ,    add (sec) ,   cmds (sec) ,  total (sec)
     10 ,  0.000119032 ,   0.00445314 ,   0.00457217
     30 ,  0.000355375 ,   0.00693251 ,   0.00728789
    100 ,  0.000857177 ,    0.0213187 ,    0.0221759
    300 ,   0.00278217 ,    0.0608069 ,    0.0635891
   1000 ,   0.00951837 ,     0.316098 ,     0.325616
   3000 ,    0.0311038 ,      1.46311 ,      1.49422
  10000 ,      0.11742 ,      6.01329 ,      6.13071
  30000 ,     0.404372 , Timeout!
>
```