Tietorakenteiden harjoitustyö – Toteutusdokumentti

Harjoitustyönä on toteutettu Määrittelydokumentin mukainen ohjelma, joka selvittää annetusta maantieteellisten paikkojen muodostamasta verkosta kahden paikan välisen lyhimmän reitin. Polunetsintä suoritetaan sekä A*- että Dijkstra-algoritmeillä, sillä tarkoituksena on myös vertailla näiden algoritmien suoritusaikoja.

Ohjelman rakenne

Ohjelma koostuu 16 luokasta, kolmesta rajapinnasta ja kahdesta enumista. Luokat on jaettu paketteihin vastuualueidensa perusteella.

Käyttöliittymään liittyvät luokat (UserInputHandler, joka käsittelee käyttäjän antamat syötteet, sekä Messenger, joka vastaa käyttäjille näytettävien viestien ja kehotteiden esittämisestä) ovat paketissa *tiralabra.ui*.

Ohjelma lukee polunetsinnässä käytettävän verkon tiedot sisään määrämuotoisesta tiedostosta. Lisäksi ohjelma voi suorittaa määrämuotoisessa skriptitiedostossa listatut polunetsintäkomennot. Näiden tiedostojen lukemisessa ja tulkitsemisessa tarvittavat luokat ovat paketissa *tiralabra.datainput*. Tiedostojen käsittelystä vastaa DataFileHandler-luokka ja luetun informaation tulkitsemisesta olioiksi vastaavat PlaceGraphMapper- ja ScriptMapper-luokat. Ensiksi mainittu tulkitsee verkkotiedoston tietoja ja jälkimmäinen skriptitiedostoa. Paketissa on määritelty lisäksi kaksi rajapintaa, IDataMapper sekä IGraphMapper; näistä IGraphMapper perii IDataMapperin. IDataMapper määrittelee muutaman metodin, joita ylipäänsä tiedostoja tulkitsevan olion on toteutettava ja IGraphMapper lisäksi yhden metodin, jota verkkotiedostoa tulkitsevalta oliolta odotetaan.

Varsinainen polunetsintälogiikka sisältyy *tiralabra.search* –paketin luokkiin. PathSearcher-luokka vastaa algoritmien ajosta ja PathAlgorithm-luokka etsintäalgoritmin toteuttamisesta. PathAlgorithm sisältää sekä A*- että Dijkstra-algoritmin mukaiset polunetsintälogiikat.

Ohjelman tunnistamia tietokohteita vastaavat luokat ovat paketissa *tiralabra.domain*. Yksittäistä maantieteellistä paikkaa kuvaa PlaceNode; tällaisen naapureita kuvaa puolestaan NeighbourNode. Yhden algoritmin tuottamien polunetsintätulosten tallentamiseen käytetään PathSearchResult-olioita; kaikkien (kahden) algoritmin tulokset yhdestä polunetsintäpyynnöstä tallennetaan PathSearchResultSet-olioon. Skriptitiedoston sisältämistä komentoriveistä puolestaan muodostetaan Command-olioita. Paketissa määritellään myös INamedObject-rajapinta, jota toteuttavan luokan oliot voidaan tallentaa NamedArrayList-tietorakenteeseen. Tätä rajapintaa toteuttavat kaikki domain-luokat NeighbourNodea lukuun ottamatta.

Ohjelman hyödyntämät tietorakenteet, MinHeap (minimikeon toteutus), PathStack (paikoista koostuvan polun tallentamiseen sopiva pinorakenne) sekä NamedArrayList (dynaamisesti kasvava taulukko olioille, joita voi etsiä nimellä) ovat paketissa *tiralabra.datastructures*.

Ohjelman syötteet

Ohjelma saa syötteensä sekä komentoriviltä että tiedostoina. Komentoriviltä saadaan käyttäjän antamat komennot ja tiedostoista polunetsinnässä käytettävien verkkojen tiedot sekä - jos käyttäjä haluaa antaa useamman polunetsintäkomennon kerralla tai että polunetsinnät suoritetaan useampaan kertaan – polunetsintäkomentojen lista.

Paikkatietojen muodostama verkko annetaan ohjelmalle aina tiedostona. Tiedosto on csv-muotoinen, käytettävä merkkikoodisto UTF-8 ja tiedoston yksi rivi edustaa aina yhtä maantieteellistä paikkaa. Rivi on muotoa ”*paikan nimi*;*leveysaste*;*pituusaste*;*naapuri1/etäisyys naapuriin 1;naapuri2/etäisyys naapuriin2”*. Paikan nimessä voi olla ääkkösiä, välilyöntejä ja tavuviivoja. Leveys- ja pituusaste ilmoitetaan desimaalimuodossa. Tämän jälkeen tulee lista kyseisen paikan naapuripaikoista, eli paikoista joihin siitä on suora yhteys. Listalla voi olla 0-n naapuripaikkaa. Jokaisesta naapuripaikasta ilmoitetaan naapuripaikan nimi sekä etäisyys sinne, kauttaviivalla erotettuna; etäisyys ilmoitetaan kilometreinä ja desimaalierottimena on piste. Esimerkiksi Helsingin osalta verkkotiedostossa on rivi ”Helsinki;60.171160;24.932580;Espoo/15.805679;Sipoo/29.133081;Vantaa/14.193181”.

Komentorivin mahdolliset syötteet ovat:

- ”?” : Ohjelma näyttää ohjeen.
- ”[*tiedostopolku*] [*lähtöpaikan nimi*] [*maalipaikan nimi*]” : Ohjelma hakee lyhimmän polun lähtöpaikasta maalipaikkaan verkossa, joka on määritelty tiedostopolku-parametrin viittaamassa tiedostossa; esimerkiksi ”data/suomi83.data Helsinki Järvenpää” hakee polun Helsingistä Järvenpäähän tiedoston suomi83.data kuvaamassa verkossa.
- ”+ [*lähtöpaikan nimi*] [*maalipaikan nimi*]” : Ohjelma hakee lyhimmän polun lähtöpaikasta maalipaikkaan käyttäen sitä verkkoa, joka on luettu viimeksi sisään; esimerkiksi ”+ Helsinki Tuusula”.
- ”* [*tiedostopolku*] [*toistokertojen lukumäärä*]” : Ohjelma lukee tiedostopolku-parametrin viittaaman skriptitiedoston ja suorittaa siinä listatut polunetsintäkomennot, toistaen ne toistokertojen lukumäärä –parametrin mukaisen määrän; esimerkiksi ”* test2.scr 100” toistaa test2.scr-tiedoston polunetsintäkomennot 100 kertaa.
- ”=” : Ohjelma tulostaa kaikki suoritusaikanaan saadut polunetsintäkomentojen tulokset komentoriville.
- ”q” : Päättää ohjelman suorituksen.

Skriptitiedosto on tekstitiedosto, jossa jokainen polunetsintäkomento on omalla rivillään. Komennot ovat samaa muotoa kuin komentoriviltäkin annettaessa.

Aika- ja tilavaativuudet

Dijkstran algoritmi

Dijkstran algoritmi on pseudokoodilla ilmaistuna seuraavanlainen:

```
Dijkstra(G, w, s)
   Initialize(G, s)
   S = ∅
   for ∀ v ∈ V
      heap-insert(H, v, distance[v])
```

```
while not empty(H)
   u = heap-del-min(H)
   S = S ∪ {u}
   for ∀ v ∈ Adj[u]
      Relax(u, v, w)
      heap-decrease-key(H, v, distance[v])
```

missä G = (V, E) on suunnattu verkko, V on verkon kaikkien solmujen joukko, E on verkon kaikkien kaarien joukko, s on lähtösolmu, H on minimikeko, S on niiden solmujen joukko, joiden lyhin etäisyys solmuun s on jo selvitetty, w(u, v) on kaaren (u, v) ∉ E paino, Adj[u] on solmun u vierussolmujen joukko, distance[v] on solmun v etäisyysarvio solmusta s ja path[v] on se joukon S solmu, joka edeltää solmua v lyhimmällä toistaiseksi tunnetulla polulla. Algoritmin käyttämät operaatiot `Initialize` ja `Relax` ovat:

```
Initialize(G, s)
   for ∀ v ∈ V
      distance[v] = ∞
      path[v] = NIL
   distance[s] = 0

Relax(u, v, w)
   if distance[v] > distance[u] + w(u, v)
      distance[v] = distance[u] + w(u, v)
      path[v] = u
```

Algoritmin alustusosuus on aikavaativuudeltaan O(|V|). Algoritmin käyttämät keko-operaatiot `heap-insert`, `heap-del-min` ja `heap-decrease-key` ovat aikavaativuudeltaan luokkaa O(log n) kun keossa on n alkiota. `heap-insert`-operaatioita suoritetaan |V| kappaletta, joten tähän kuluu O(|V|log|V|). `while`-silmukassa `heap-del-min` -operaatiota kutsutaan kerran per solmu, joten aikaa menee tähän luokkaa O(|V|log|V|); lisäksi `Relax`-operaatio suoritetaan kerran per kaari - jolloin `heap-decrease-key`-operaatiota kutsutaan korkeintaan kerran – joten tältä osin aikavaativuus on O(|E|log|V|). Siten koko `while`-loopin ja samalla koko algoritmin aikavaativuus on luokkaa O((|V| + |E|)log|V|). Tilavaativuus puolestaan määräytyy keon tarvitseman koon perusteella: algoritmin alussa keko sisältää kaikki solmut, joten algoritmin tilavaativuus on O(|V|).

A-algoritmi*

A*-algoritmi on pseudokoodina seuraavanlainen:

```
Astar(G, w, a, b)
   Initialize-astar(G, a, b)
   S = ∅
   while (b ∉ S)
      valitse u ∈ V \ S, jolle startdistance[u]+enddistance[u]
                         on pienin
      S = S ∪ {u}
      for ∀ v ∈ Adj[u]
         Relax-astar(u, v, w)
```

missä a on lähtösolmu, b on kohdesolmu, `startdistance[u]` on solmun u etäisyys lähtösolmuun, `enddistance[u]` solmun u arvioitu etäisyys kohdesolmuun (joka arvioidaan `Initialize-astar-`

operaatiossa käyttäen heuristiikkafunktiota `EstimateDistance(u, b)`) ja operaatiot `Initialize-astar` ja `Relax-astar` ovat seuraavanlaiset:

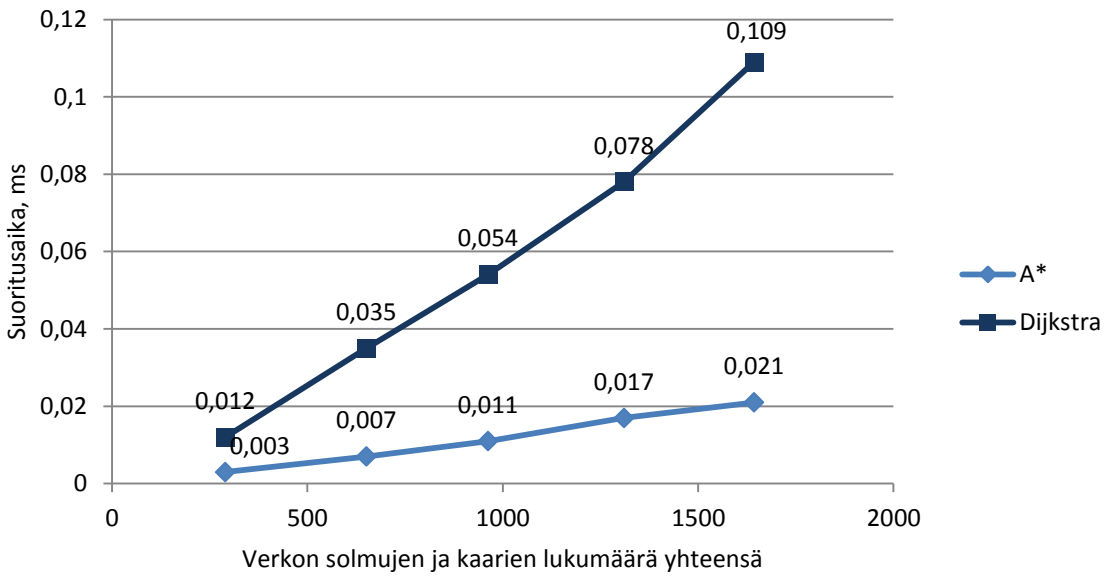
```
Initialize-astar(G, a, b)
   for ∀ v ∈ V
      startdistance[v] = ∞
      enddistance[v] = EstimateDistance(v, b)
      path[v] = NIL
      startdistance[a] = 0

Relax-astar(u, v, w)
   if startdistance[v] > startdistance[u] + w(u, v)
      startdistance [v] = startdistance [u] + w(u, v)
      path[v] = u
```

Tästä on helppo havaita, että A*-algoritmi on huomattavan samankaltainen Dijkstra-algoritmin kanssa. A*-algoritmin käyttämä etäisyysarvio solusta *v* kohdesoluun lasketaan tässä sovelluksessa paikan koordinaattitietojen pohjalta haversini-kaavalla, jonka suoritus tapahtuu vakioajassa. Tällöin Dijkstra- ja A*-algoritmeilla on sama pahimman tapauksen aikavaativuus, $O\big((|E| + |V|)\log|V|\big)$, missä $|E|$ on verkon kaarien lukumäärä ja $|V|$ solmujen lukumäärä. Myös tilavaativuus O(|V|) on sama.

Algoritmien suorituskykyvertailu

Ohjelma seuraa algoritmien ajoaikaa ja näyttää sen polunetsintätulosten yhteydessä. Ajoajassa otetaan huomioon vain varsinainen etsinnän suorittaminen, ei esimerkiksi verkkotiedoston lukua tai verkon alustamista. Yksittäinen ajokerta ei anna luotettavaa kuvaa algoritmien vaatimasta ajasta; ajoajat ovat niin lyhyitä, että esimerkiksi koneen muut prosessit voivat vaikuttaa tulokseen. Suorituskykyvertailua varten paremman kuvan antaakin saman polunetsintätehtävän toistaminen ja näistä toistokerroista saatavien ajoaikojen keskiarvo. Alla olevassa kuvassa on esitetty A*- ja Dijkstra-algoritmien ajoajat testeissä käytetyillä verkoilla. Tarkempi selostus testauksesta ilmenee Testausdokumentin ”Suorituskykytestaus” –osiosta. Aineistoja on viisi erikokoista; x-akselilla esitetään aineiston sisältämien solmujen ja kaarien lukumäärien summa. Kuvion perusteella molemmilla algortimeilla suoritusaika näyttää kasvavan lineaarisesti suhteessa solmujen ja kaarien lukumäärän summaan.



Kuva 1: Ohjelman käyttämien algoritmien suoritusajat eri kokoisilla syötteillä

Ohjelman jatkokehitys

Ohjelma pitää nyt polunetsintöjen tulokset vain ajonaikaisessa muistissa; kehitettävänä toiminnallisuutena voisikin olla tulosten tallentaminen tiedostoksi. Ohjelma on nyt komentoriviltä käytettävä standalone-sovellus mutta se voitaisiin muuttaa palveluksi, jolle voisi rajapinnan kautta antaa polunetsintätehtäviä; näin se voitaisiin liittää osaksi esimerkiksi jonkinlaista reittiopasjärjestelmää – ohjelman käsittelemät paikathan voisivat olla esimerkiksi risteyksiä koordinaattitietoineen ja paikkojen välimatkat aitoja ajoetäisyyksiä.

Lähteet

Tietorakenteet ja algoritmit –kurssin luentokalvot (katsottu 31.7.2016)
https://www.cs.helsinki.fi/u/jkivinen/opetus/tira/k16/luennot.pdf

Rosetta Code –sivustolla esitetty Haversine-kaavan toteutustapa Javalla
https://rosettacode.org/wiki/Haversine_formula#Java