Tietorakenteiden harjoitustyö – Määrittelydokumentti

Ratkaistava ongelma

Harjoitustyössä on tarkoituksena rakentaa ohjelma, joka selvittää annetusta maantieteellisten paikkojen muodostamasta verkosta kahden paikan välisen lyhimmän reitin. Ohjelma etsii lyhimmän polun käyttäen kahta eri algoritmiä. Tuloksena näytetään lyhin polku askel askeleelta, polun kokonaispituus sekä kummankin algoritmin suoritusajat.

Maantieteellisistä paikoista muodostuu painotettu suuntaamaton verkko siten, että jokainen paikka on verkon solmu, naapuripaikat ovat sen vierussolmuja ja välimatka kuhunkin naapuriin on solmun ja vierussolun välisen kaaren paino – siten kaarien painot ovat ei-negatiivisia.

Ohjelman syötteet

Ohjelman syötteenä ovat tiedot tästä suuntaamattomasta verkosta sekä lähtö- ja maalipaikka, joiden välinen lyhin reitti halutaan selvittää. Suuntaamattoman verkon tiedot annetaan datatiedostona, jossa kullakin rivillä on aina yhden paikan nimi ja koordinaatit (esim ”Helsinki;60 10 15/24 56 15”) sekä lista ao. paikan naapuripaikoista ja välimatkoista näihin (esim Helsingin osalta naapurilista olisi ”Espoo 16;Vantaa 15;Sipoo 29”.) A*-algoritmia varten ohjelma laskee paikkakuntien välisiä linnuntie-etäisyyksiä niiden koordinaattien perusteella.

Ohjelmaa ajettaessa käyttäjän on annettava käytettävän datatiedoston polku sekä lähtö- ja maalipaikkojen nimet.

Valitut algoritmit

Lyhyimmän polun etsintä toteutetaan ohjelmassa sekä Dijkstra- että A*-algoritmeilla. Nämä molemmat pystyvät selvittämään painotetun verkon kahden pisteen välisen lyhimmän polun kun kaarien painot ovat ei-negatiivisia. Dijkstran algoritmi etsii lyhintä polkua tasapuolisesti joka suuntaan lähtösolmusta ottamatta huomioon, millä suunnalla kohdesolmu sijaitsee. A*-algoritmi puolestaan voidaan mieltää Dijkstran algoritmin laajennukseksi, jossa suositaan polkuja, jotka ovat kohdesolmun suunnalla. Tätä varten A*-algoritmin on laskettava myös etäisyysarviot kustakin verkon solmusta kohdesolmuun.

Ohjelma esittää tuloksissa molempien algoritmien käyttämät suoritusajat; ajettaessa ohjelmaa erikokoisilla syötteillä voidaan seurata miten syötteen koko vaikuttaa suoritusaikaan. Algoritmit tarvitsevat apunaan minimikeko- sekä pino-tietorakenteita, jotka myös toteutetaan harjoitustyössä.

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) $\notin$ 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.

Lähteet

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