Tira harjoitustyö

Verner Fortelius

Määrittely

Minkä ongelman ratkaiset ja millä tietorakenteella tai algoritmilla

Reitinhaku ruudukossa A*-algoritmilla.

Miten tehokkaasti toteutuksesi tulee ongelman ratkaisemaan (aika- ja tilavaativuudet)

O((|E|+|V|)log|V|)

Mitä lähdettä käytät. Mistä ohjaaja voi ottaa selvää tietorakenteestasi/algoritmistasi

http://en.wikipedia.org/wiki/A*_search_algorithm

http://www.policyalmanac.org/games/aStarTutorial.htm

http://www.briangrinstead.com/blog/astar-search-algorithm-in-javascript

http://www.cs.helsinki.fi/u/floreen/tira2012/tira.pdf

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Toteutusdokumentti

Ratkaiseeko toteutuksesi ongelman määrittelyssä esittämälläsi tehokkuudella

Olennaisimmat tietorakenteet tässä A*-toteutuksessa ovat hajautustaululla indeksoitu minimikeko ja hajautustauluja. Algoritmissa pitää usein selvittää onko tietty alkio olemassa, mutta esimerkiksi seuraavaa tai edeltävää alkiota ei koskaan tarvitse etsiä eikä järjestyksellä ole mitään merkitystä. Alkioita haetaan myös aina avaimen perusteella. Oikein toteutetussa hajautustaulussa lisäys-, poisto- ja etsi-operaatiot toimivat vakioajassa.

Algoritmi tarvitsee myös minimikeon jossa säilytetään löydetyt mutta ei vielä läpikäydyt solmut. Minimikeossa pienimmän avaimen poisto ja uuden avaimen lisäys toimii ajassa O(log n). Operaatiot jotka perustuvat avaimen etsimineen toimivat kuitenkin ajassa O(n). Tässä toteutuksessa minimikeossa on oma hajautustauluun perustuva indeksi jolloin etsiminen toimii vakioajassa.

Tehokkain tapa tallentaa ruudukko on kaksiulotteisessa taulukossa. A*-algoritmi itsessään toimii myös toisenlaisissa verkoissa mutta silloin varsinkin heuristiikkafunktio joka laskee optimaalisinta polkua, muuttuu olennaisesti. Myös verkon tallennustapa pitäisi silloin olla esimerkiksi vieruslista tai matriisi.

A*-algoritmin käytännön tehokkuus riippuu hyvin pitkälti heuristiikkafunktiosta. Mikäli heuristiikkafunktiota muutetaan niin että se aina palauttaa nollan, toimii A* samalla tavalla kuin Dijkstran algoritmi. Heuristiikkafunktio ei kuitenkaan saisi yliarvioida reittiä, koska tällöin löydetty reitti ei välttämättä ole lyhin.

Miksi näin on. Perustele pseudokoodia käyttäen

Ajantarve

Kaikki listat paitsi openset ovat toteutettu hajautustauluilla. Tästä syystä kaikki tarvittavat operaatiot jotka kohdistuvat näihin ovat vakioaikaisia eli O(1). Indeksoitu minimikeko on hitaampi. Metodit insert, decreaseKey ja deleteMin vievät aikaa O(log n). Koska minimikeon haku on toteutettu hajautustauluun perustuvalla indeksillä, on haun aikavaativuus O(1).

Verkko on tallennettu 2-ulotteiseen taulukkoon, joten solmuihin pääse käsiksi indeksillä vakioajassa. Myös heuristiikkafunktio perustuu tähän joten sekin on vakioaikainen.

Ulomman while-silmukkan ajokerrat riippuvat polun pituudesta. Pahimmassa tapauksessa reittiä ei löydy tai heuristiikkafunktio aliarvio reittiä jatkuvasti. Tällöin silmukka käy läpi kaikki solmut verkossa eli O(V). Jokaista solmua voi lisätä ja poistaa opensetistä korkeintaan kerran. Nämä operaatiot kestävät O(log |V|) joten yhteensä ajantarve on O(|V| log |V|).

Sisempi for-silmukka ajetaan niin monta kertaa kuin solmulla on naapureita eli O(|E|). Ruudukossa jokaisella solmulla on aina 3-8 naapuria. Kaikki operaatiot tässä osassa ovat kuitenkin vakioaikaisia.

Lopullinen ajantarve on siis O((|E| + |V|)log |V|). Tämä on itse asiassa sama kuin Dijkstran algoritmilla, koska heuristiikkafunktion hyödystä ei ole varmuutta.

Parhaimmassa tapauksessa reitti on suora viiva ilman esteitä, jolloin silmukka käy läpi ainoastaan reitillä olevat solmut. Käytännössä suorituskertojen määrä riippuu reitistä ja heuristiikkafunktiosta. Hakua voi nopeuttaa muuttamalla heuristiikkafunktiota niin että se yliarvioi etäisyyden maaliin. Tässä tapauksessa algoritmi ei kuitenkaan välttämättä löydä lyhintä reittiä.

Tilantarve

Algoritmi pitää koko ajan muistissa löydetyt ja käsitellyt solmut. Samalla tavalla kuin aikavaativuus myös tilavaativuus riippuu reitin pituudesta. Pahimmassa tapauksessa reittiä ei löydy jakaikki verkon solmut päätyvät muistiin. Parhaimmassa tapauksessa algoritmi käy läpi vain reitillä olevat solmut ja näiden naapurit. Aika- ja tilavaativuus ovat siis samat.

Koodin abstrakti versio

```
        openset.Insert(start);
n       while (openset not empty)
log n           current = openset.DeleteMin;
1               closedset.insert(current);
1               if (current is end)
                        return ReconstructPath

1               for each neighbour in current
1                       float tentative_gscore = g_score.get(current) + weight;
```

1	float tentative_fscore = tentative_gscore + Heuristic.GetDistance(neighbour, end, tolerance);
1	Float g_scoreneighbour = g_score.get(neighbour);
1	if (closedset.contains(neighbour) AND g_scoreneighbour <= tentative_gscore)
	continue;
	if (g_scoreneighbour == null OR tentative_gscore < g_scoreneighbour)
1	g_score.put(neighbour, tentative_gscore);
1	camefrom.put(neighbour, current);
1	if (!openset.contains(neighbour) AND !closedset.contains(neighbour))
log n	openset.Insert(neighbour);
	else
log n	openset.DecreaseKey(neighbour)

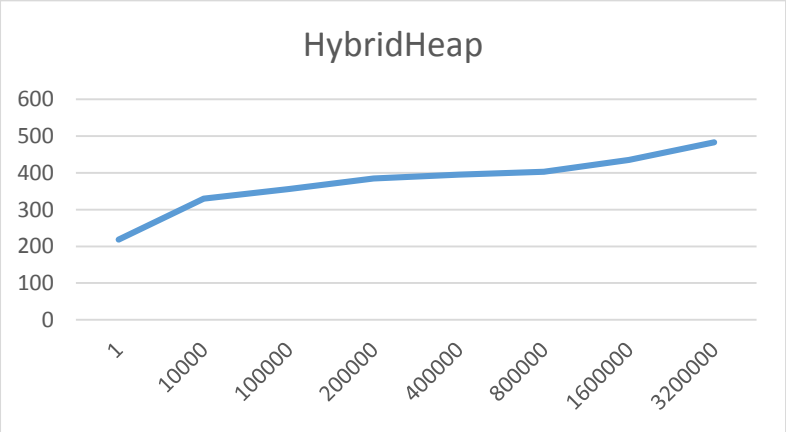
Testausdokumentti

Algoritmin kannalta kriittisimmät osat ovat testatut junit yksikkötesteillä. Testaustapana on ollut lähinnä black box testausta, joten vain public luokat ja metodit ovat testattuja.

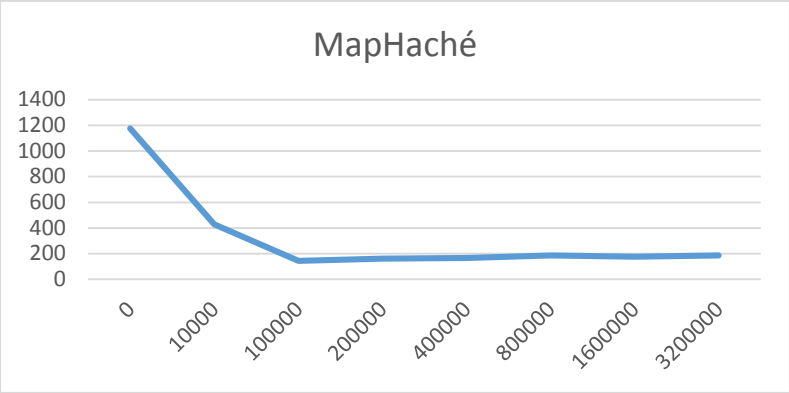
Testeillä löytyi muutamia bugeja. Esimerkiksi hashCode-metodi saattaa joskus palauttaa negatiivisen arvon jolloin tämä arvo mod x tulee olemaan negatiivinen. Tämä ei tietenkään toimi jos arvoa käytetään taulukon indeksinä.

Tietorakenteet testasin erikseen erilaisilla syötteillä ja tuloksien perusteella ne toimivat oletetussa ajassa.

HybridHeap insert suoritettiin 1000 kertaa ja mitattu aika nanosekunneissa. Ongelmana testaamisessa oli JVMn ja välimuistin jne. sulkeminen pois tuloksista. Tulos on kuitenkin selvästi parempi kuin O(n) ja huonompi kuin O(1).



Hajautustaulu toimii myös suurin piirtein odotetusti. Jälleen kerran ilmeisesti välimuisti ja mahdollisesti JVMn omat optimoinnit sotkevat tulosta. 1000 alkion lisääminen tyhjään tauluun kestää huomattavasti kauemmin kuin 1000 alkion lisääminen kun ensin on lisätty 100000.



Puutteita

Piirtäminen tapahtuu järjettömän hitaasti. Alkuperäinen bitmappi voisi olla tallennettu sellaisenaan eikä sitä tarvitsisi piirtää kokonaan uudestaan joka klikkauskerralla.

README

Miten ohjelmasi kääntyy

Riippuen hakemistosta

javac.exe -d <output directory> -s astar/.java*

Miten ohjelmasi ajetaan

java -jar "Astar.jar" tai *java astar/AstarGUI*

Tai sitten vain klikkaamalla jar-tiedostoa…

Ohjelma tarvitsee myös bitmapin joka toimii karttana. Tummempi väri tarkoittaa suurempaa painoa.

Klikkaamalla karttaa vasemmalla hiirinapilla, ohjelma etsii reitin uudestaan samasta aloitussolmusta. Klikkaamalla oikealla napilla ohjelma etsii reitin edellisen reitin lopusta.