A*vis
A* algoritmin visualisointi-ohjelma

1. Aihemäärittely

Toteutetaan A* visualisoija.

käyttäjä voi käyttöliittymässä määrittää aloitus ja päätöspaikan (taulukossa). Tämän jälkeen käyttäjä voi vielä piirtää seiniä, joita algoritmi osaa kiertää.

Extra jos aikaa jää: Käyttäjä voi ladata kartan (taulukon) kuvatiedostosta ja määrittää siihen aloitus- ja lopetuspisteen.

Extra2 jos aikaa jää: Käyttäjä voi luoda kovuuksilla seiniä, joista algoritmi etsii vähiten "maksavan" reitin.

Aikavaativuus A*:lla tulee (s2013 luentomonisteen mukaan): O((|E|+|V|)log|V|)
Tilavaativuus tulee olemaan O(|V|)

lähteinä tulen käyttämään ainakin s2013 luentomonistetta:
http://www.cs.helsinki.fi/u/floreen/tira2013syksy/tira.pdf
Lisäksi wikipedian pseudokoodi A* toteutukseen http://en.wikipedia.org/wiki/A*_search_algorithm

Tietorakenteiden aikavaativuudet:
Minimikeko:

poll()	O(log n)
insert()	O(log n)

Hajautustaulu:

search()	O(1) (keskimäärin)
insert()	O(1) (keskimäärin)

Oma ArrayList:

insert()	O(1)
get()	O(1)

2. Toteutusdokumentti

2.1 ArrayList toteutus

Oletusarvoisesti luo 10-elementin kokoisen data-arrayn, johon dataa säilötään.
Lisäysoperaatio kutsuu ensin *ensureCapacity(size+1)* joka varmistaa, että data taulukossa on varmasti tarpeeksi tilaa lisättävälle elementille. Tämän jälkeen elementti lisätään data taulukon loppuun. Lisäys tapahtuu siis O(1) ajassa, sillä useimmiten tilaa on data taulukossa tarpeeksi.
Kun tila loppuun *ensureCapacity* käyttää javan *Arrays.copyOf* funktiota data taulukon laajentamista varten.
ArrayList luokka toteuttauttaa myös *contains(e), sublist(i), indexOf(e)* ja *iterator()* metodit.
indexOf metodin aikavaativuus on O(n), se käy läpi data taulukon alusta alkaen niin monta elementtiä kunnes löytää parametriksi annetun elementin ja palauttaa sen indeksin.
contains metodin vaativuus periytyy suoraan *indexOf* metodilta, sillä se on toteutettu katsomalla mitä *indexOf* palauttaa.

2.2 HashMap toteutus

HashMap luo DEFAULT_CAPACITY = 15000 kokoisen data taulukon johon se tallentaa kaiken datan.

HashMap on staattisen kokoinen, johon mahtuu DEFAULT_CAPACITY:n verran avain-arvo pareja. Törmäyksen sattuessa sijoitetaan uusi pari seuraavaan vapaaseen data taulukon riviin
locate(key) metodi etsii elementin data taulukosta O(1) ajassa, se ensin pyytää avaimen hashCode:a ja tämän jälkeen käy hashCode:sta alkaen data taulukon rivejä läpi. Mikäli yhtään yhteentörmäystä ei ole tapahtunut ei rivejä tarvitse käydä läpi kuin yksi kappale. Pahimmassa tapauksessa (jos data-taulu on täynnä niin locate voi viedä O(n) verran aikaa.

```
put(key,value):
          current_i = locate(key)
          if(current_i) /* päivitä olemassa oleva data */
          else
          /* etsi seuraava tyhjä paikka, alkaen hash(key) paikasta data
          taulukossa*/
```

put toimii ajassa O(1), sillä sen aikavaativuus on locate + seuraavan tyhjän paikan etsimiseen menevä aika, joka on keskimäärin vakioaikaista.
get toimii myös ajassa O(1), se kutsuu locate metodia ja mikäli locate palauttaa muun arvon kuin -1 se palauttaa data taulusta oikean arvon.
containsKey toimii myös ajassa O(1), se kutsuu locate metodia ja katsoo löytyikö indeksiä vai ei.

2.3 MinimumHeap toteutus

MinimumHeap on toteutettu ArrayList jonona, jossa pidetään huolta, että jonon alussa on aina pienin mahdollinen arvo. Data tallennetaan Pari-muodossa ArrayListiin (heap), jossa parin ensimmäinen elementti on data ja toinen jonotusarvo kekoon.

```
1: add(d,i):
2:        p = new pair(d,i)
3:        heap.add(null)
4:        it = heap.size()-1
5:        while(it > 0 && heap.get(it/2).getSecond() > i):
6:                  heap.set(it,queue.get(it/2))
7:                  it = it/2
8:        heap.set(it,p)
```

add metodin rivit 2-4,8 ovat kaikki vakioaikaisia. 5-7 riveillä käydään läpi kekoa ja kuljetetaan keosta alaspäin elementtejä tehden tilaa uudelle elementille. Tässä it- muuttujaa jaetaan kokoajan kahdella, joten add toimii ajassa O(log n) lissä it oli alun perin kekolistan pituus -1 (n).

```
1: heapify(i):
2:        l = i*2
3:        r = l+1
4:        lrg = i
5:
6:        if( l < heap.size() && heap.get(l).getSecond() <
heap.get(lrg).getSecond()):
7:                  lrg = l;

8:        if( r < heap.size() && heap.get(r).getSecond() <
heap.get(lrg).getSecond()):
9:                  lrg = r;

10:       if(lrg != i):
11:                 swap(i,lrg)
12:                 heapify(lrg)
```

heapify toimii ajassa O(log n), sillä se kutsuu rekursiiviesti itseään, kuitenkin log n verran maksimissaan, sillä lrg on aina joko l tai r, jotka ovat aina kutsukerran parametri i*2. Heapifyn ideana on tarkistaa, voidaanko elementin i vasen tai oikea lapsi vaihtaa sen kanssa (ne ovat pienempiä kuin i:n elementin arvo itse).
poll metodi palauttaa heap taulukon ensimmäisen elementin (olettaen että siellä on pienin mahdollinen arvo). Tämän jälkeen se asettaa heap taulukon viimeisen elementin ensimmäiseksi ja kutsuu heapify (eli kuljettaa viimeistä elementtiä kunnes sille löytyy paikka), heapify on O(log n) joten poll toimii ajassa O(log n)
Lisäksi MinimumHeap toteuttaa metodit contains (ajassa O(n)), isEmpty (ajassa O(1)) ja swap (ajassa O(1))

2.4 A* (Astar) toteutus

Toteutus Java koodissa vastaa loogisesti samaa kuin alla esitetty pseudokoodi, mutta se sisältää historiatallennusta ja muita graafisteknisiä asioita, joilla ei ole A* kannalta merkitystä joten niitä ei tähän ole kirjattu.

```
1: Astar(G,heurastic):
2:          open = MinimumHeap()
3:          closed = ArrayList()
4:          cameFrom = HashMap()
5:          gScore = HashMap()
6:          for(Node n : G.allNodes()):
7:                    gScore.put(n,<INFINITY>)
8:          gScore.put(G.start, 0)
9:          open.add(start,0 + heuristic.estimate(G.start,G.goal))
10:         while(!open.isEmpty()):
11:                   Node current = open.poll()
12:                   if(current == G.goal):
13:                             return;
14:                   closed.add(current)
15:                   for(Node n : current.allNeighbors()):
16:                             if(closed.contains(n)) continue
17:                             int score = gScore.get(current) + n.getCost();
18:                             if(!open.contains(n) || pgscore < gScore.get(n)):
19:                                       cameFrom.put(n,current)
20:                                       gScore.put(n,pgscore)
21:                                       if(!open.contains(n)):
22:                                                 open.add(n,pgscore +
                                                   heuristic.estimate(n,G.goal))
```

riveillä 2-5 alustetaan tarvittavat apumuuttujat ajassa O(1)
riveillä 6-7 käydään läpi jokainen solmu verkossa G ajassa O(|V|)
riveillä 8-9 alustetaan ensimmäinen läpikäytävä solmu ajassa O(1)
riveillä 10-22 on itse looppi jossa käydään läpi open-solmuja niin kauan kun niitä on, pahimmassa tapauksessa solmuja tulee |V| kappaletta, joten looppi voidaan käydä läpi |V| kertaa
rivillä 11 haetaan MinimumHeapista pienin mahdollinen solmu, tämä vie aikaa O(log n) verran, jossa n on avoimien solmujen määrä, joka tässä toteutuksessa on huomattavasti |V|:tä pienempi lukumäärä, joten todetaan tämän rivin aikavaativuudeksi O(|V|)
riveillä 15-22 käydään läpi läpikäytävän solmun naapurit, tässä toteutuksessa naapureita voi olla 2-4 kpl joten looppi toimii vakioajassa.
rivillä 16 katsotaan onko solmu jo käyty läpi, ArrayList.contains toimi ajassa O(n)
rivillä 17 katsotaan uusi score, HashMap.get toimii ajassa O(1)
rivillä 18 HashMap.contains toimii ajassa O(n) ja HashMap.get ajassa O(1)
riveillä 19-20 HashMap.put metodikutsut toimivat ajassa O(1)
rivillä 21 ArrayList.contains toimii ajassa O(n)
rivillä 22 lisätään MinimumHeappiin uusi arvo, lisäys toimi ajassa O(log n)
Loopin riveillä 15-22 yhteisaika on siis O(4 * (n + 1 + n + 1 + 1+ n + log n)) joka tarkoittaa käytännössä O(n) aikaa, jossa n kuvaa kaikkien mahdollisten avoimien solmujen määrää, joka tässä toteutuksessa on korkeintaan neliön leikkaajassa olevien solmujen määrä, eli huomattavasti |V|:tä pienempi lukumäärä, joten todetaan tämän loopin aikavaativuudeksi O(1)
Loopin riveillä 10-22 yhteisaika on siis O(|V| * (log |V| + 1))josta tulee myös loppujen lopuksi koko toteutuksen aikavaativuus O(|V| * log |V|).

3. Testausdokumentti

3.1 Unit testaus

Testaus on toteutettu jUnit-testeillä. Unit-testit kattavat kaikki kriittiset tapaukset tietorakenteissa sekä tarkistavat myös A* toimintalogiikan yksinkertaisten painotettujen verkkojen avulla.

jUnit raportti:

AStarTest testaa neljällä eri painotetulla verkolla, että algoritmi tuottaa oikean tuloksen.
Esimerkiksi testSimple1

- Solmut: A (0,start), B (1), C (2), D(0,end)
- Kaaret, A-B, A-C, B-D, C-D
- Lyhin polku löytyy A->B->D
- Toinen (ei lyhin polku) A->C->D

ArrayListTest testaa

- ArrayList operaatiot
  - isEmpty
  - subList
  - set
  - remove
  - add
- ArrayList ei voi täyttyä
- ArrayLististä ei voi poistaa/lisätä indeksin ulkopuolisia datoja
- ArrayListin rangeCheck toimii toikein

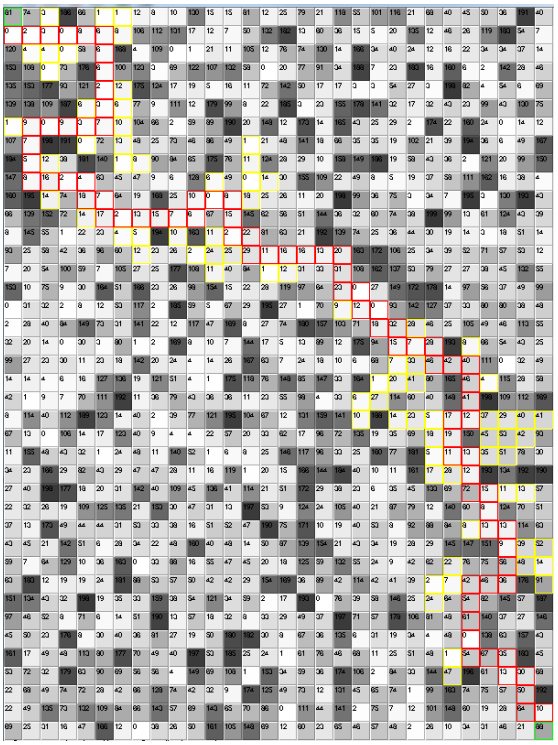
HashMapTest testaa

- HashMap operaatiot
  - put
  - get
- HashMap toimii oikein täyttyessään
- put samalla key:llä korvaa arvon, ei lisää uutta
- get metodi toimii oikein

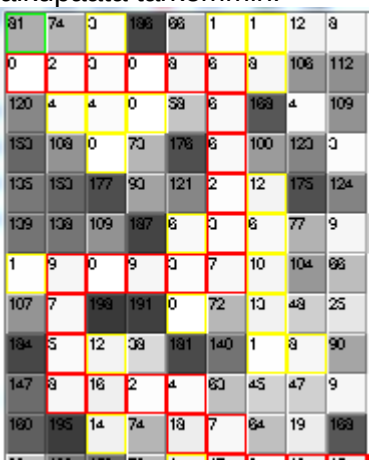
MinimumHeapTest testaa

- MinimumHeap operaatiot
  - add
  - poll
- Poll palauttaa aina varmasti pienimmän arvon

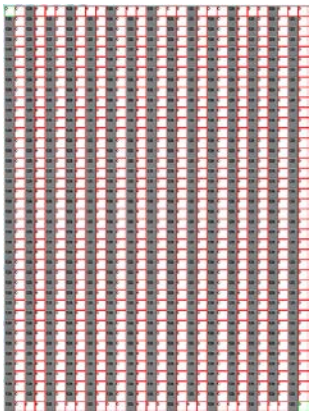
3.2 Yleinen testaus



Viereisestä kuvasta huomataan, että punainen (löydetty lyhin reitti) on lyhin reitti, keltainen väri indokoi tarkastettua solmua, joka ei kuitenkaan kuulu lyhimpään reittiin. Tarkastellaan kyseisestä ratkaisusta alkupäätä tarkemmin:

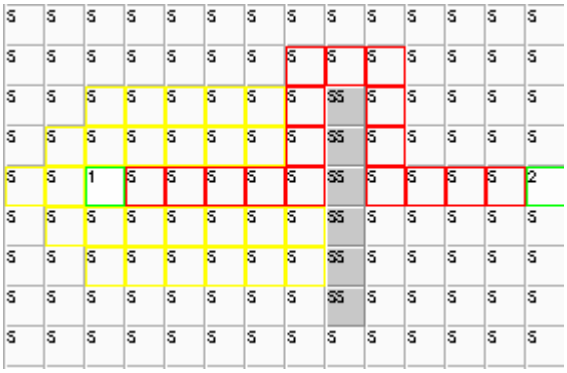


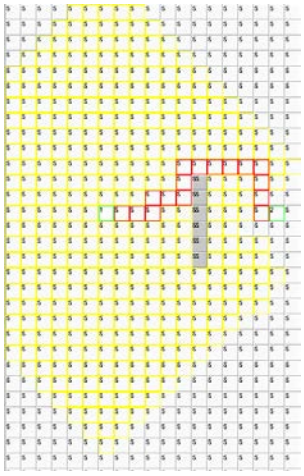
Huomataan, että selvästi punaisen reitin painojen kertymä on pienin mahdollinen kyseisellä reitillä. Kaikkien valittujen solmujen painot ovat < 20. Huomataan myös, että algoritmi on tutkinut myös muita mahdollisia reittejä, mutta todennut niiden kustannukseksi suuremman luvun.



Erikoistapaus on yksireittinen labyrintti, jossa on kuitenkin mahdollisuus maksua vastaan oikaista. A* algoritmin toteutus kuitenkin löytää ilmaisen reitin esimerkkilabyrintissä:

Viimeisenä testinä simppeli este maalin ja alkupisteen välissä:





Kaikki testit ovat käyttäneet suuntaavaa heurastiikkaa (painottaa läpikäynnissä läheisyysjärjestystä).Alimmassa esimerkissä jos otamme tämän heurastiikan pois huomaamme, että läpikäytävien solmujen määrä kasvaa huomattavasti (vasen kuva)

4. README

4.1 Kääntäminen

Kääntäminen tapahtuu NetBeans IDE:llä Run -> Build (F11)

4.2 Ajaminen

Ohjelma ajetaan komennolla
java -jar AStarVis.jar

tai

java -jar AStarVis.jar <kuvatiedosto>
Kuvatiedostossa pitää olla yksi
aloituspikseli r=0,g=255,b=0
lopetuspikseli r=255,g=0,b=0
Näitä pikseleitä EI saa löytyä enempää kuin 1kpl
Suositeltu kuvakoko on 30x40 px

4.3 Käyttäminen

Ohjelman pääikkuna ottaa vastaan erilaisia näppäin komentoja
R: luo uusi satunnainen verkko painoilla (A*vis ratkaisee ja animoi ratkaisun)
M: luo labyrintti (A*vis ratkaisee ja animoi ratkaisun)
C: avaa "A*vis Composer", jolla voit luoda omia verkkoja
D: Vaihda ratkaisuheurastiikaksi suuntaava funktio
N: Vaihda ratkaisuheurastiikaksi omia painoja katsova funktio

Kun vaihdat ratkaisuheurastiikkaa sinun täytyy joko A) painaa R, M tai muuttaa "composer":ssa verkkoasi

A*vis Composer ohje:
Painamalla hiiren vasenta näppäintä voit kasvattaa ruudun painoa 50:llä yksiköllä (voit myös maalata ruutuja)

painamalla hiiren oikeaa näppäintä jollekin ruudullee muuttuu se aloitusruuduksi (vihreä)

kun painat jollekin aloitusruudulle toisen kerran oikeaa näppäintä muuttuu se lopetusruuduksi.

HUOM: A* algoritmi ratkaisee vain yhdestä alkupisteestä yhteen päätöspisteeseen. Suurin x,y koordinaatillinen alku/lopetuspiste valitaan siis ratkaisuun mukaan.

Composerin saa tyhjennettyä sulkemalla ikkunan ja painamalla C uudestaan A*vis pääikkunassa (avaa uuden composerin)