Testausdokumentti

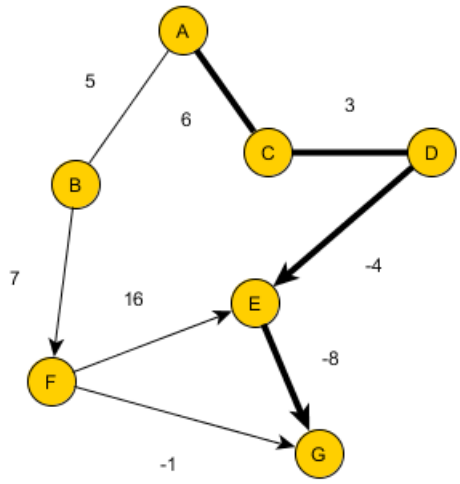
Junit testit

Lista junit testeistä:

- Tietorakenteet
    - HashtableTest.java. Hajautustaulun testit
        - Testaa kaikkia käyttötapauksia (poisto, lisääminen, hakeminen) sekä erikoistapauksia (olemassaolevan avaimen päälle lisääminen ja lisäämättömän avaimen haku)
    - HeapTest.java. Minimikeon testit
        - Testaa kaikkia käyttötapauksia (lisääminen, poistaminen, avaimen arvon alentaminen)
    - GraphTest.java. Verkon testit
        - Testaa kaikkia käyttötapauksia (solmujen yhdistäminen, solmujen erottaminen, kaaren painon vaihto, kaaren painon pyytäminen)
    - LinkedListTest.java. Linkitetynlistan testit
        - Testaa kaikkia käyttötapauksia (lisääminen, poistaminen)
    - QueueTest.java. Jonon testit
        - Testaa kaikkia käyttötapauksia (lisääminen, poistaminen)
    - StackTest.java. Pinon testit
        - Testaa kaikkia käyttötapauksia (lisääminen, poistaminen)
- Algoritmit
    - BFSTest.java. BFS algoritmin testit
    - DijkstraTest.java. Dijkstran algoritmin testit
    - CycleDetector.java. Syklintunnistus testit

Empiiriset testit

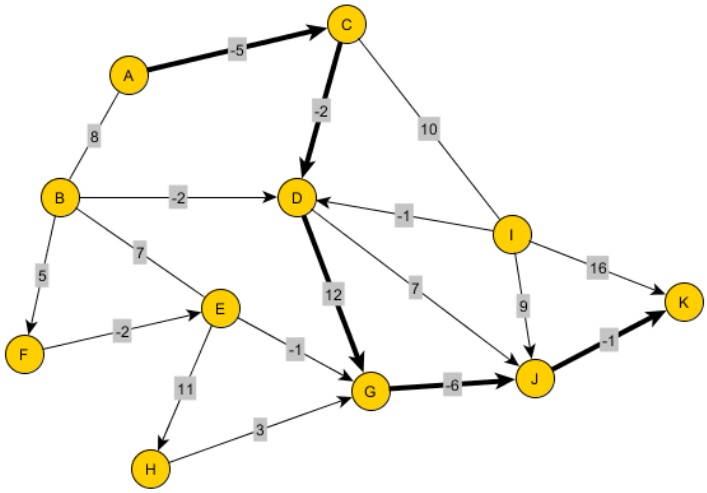
Bellman-Fordin algoritmia testattiin luomalla omia negatiivisilla painoilla varustettuja verkkoja ja tarkistamalla, että algoritmi antoi saman polun pituuden, kuin itse laskettu polun pituus. Virhetilanteita ei ilmennyt. Positiivisilla painoilla verkkoa tutkittiin ajamalla samalle verkolle sekä Dijkstra, että Bellman-Ford, jolloin molemmat löysivät yhtä pitkän polun lyhyimmäksi poluksi syötteenä annettujen kahden solmun välillä. Tämä vahvistaa algoritmin toimimista, koska toimintatavoiltaan ne ovat hyvin erilaisia, joten niistä tuskin löytyy samaa virhettä. Alla pari kuvaa, jotka esittävät Bellman-Fordin löytämät lyhyimmät polut negatiivisia painoja sisältävissä verkoissa.




```
Graph graph = new Graph();
Vertex a = new Vertex("A");
Vertex b = new Vertex("B");
Vertex c = new Vertex("C");
Vertex d = new Vertex("D");
Vertex e = new Vertex("E");
Vertex f = new Vertex("F");
Vertex g = new Vertex("G");
graph.connectBothWays(a,c,6);
graph.connectBothWays(a,b,5);
graph.connect(b,f,7);
graph.connectBothWays(c,d,3);
graph.connect(d,e,-4);
graph.connect(d,g,1);
graph.connect(e,g,-8);
graph.connect(f,e,16);
graph.connect(f,g,-1);
System.out.println(graph.shortestPath(a, g));
```

```
run:
Path length: -3
Path: C->D->E
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tässä kuvassa Bellman-Ford etsii lyhyimmän polun solmusta A solmuun G. Ohjelmakoodi on oikealla ja algoritmin antama polku tulostettuna alla. Lyhyin polku kulkee tummennettuja kaaria pitkin.




```
Graph graph = new Graph();
Vertex a = new Vertex("A");
Vertex b = new Vertex("B");
Vertex c = new Vertex("C");
Vertex d = new Vertex("D");
Vertex e = new Vertex("E");
Vertex f = new Vertex("F");
Vertex g = new Vertex("G");
Vertex h = new Vertex("H");
Vertex i = new Vertex("I");
Vertex j = new Vertex("J");
Vertex k = new Vertex("K");
graph.connectBothWays(a,b,8);
graph.connectBothWays(b,e,7);
graph.connectBothWays(c,i,10);
graph.connect(a,c,-5);
graph.connect(c,d,-2);
graph.connect(b,d,-2);
graph.connect(b,f,5);
graph.connect(f,e,-2);
graph.connect(e,h,11);
graph.connect(e,g,-1);
graph.connect(h,g,3);
graph.connect(g,j,-6);
graph.connect(j,k,-1);
graph.connect(d,g,12);
graph.connect(d,j,7);
graph.connect(i,k,16);
graph.connect(i,d,-1);
System.out.println(graph.shortestPath(a, k));
```

```
run:
Path length: -2
Path: C->D->G->J
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tässä kuvassa on hieman suurempi verkko, jossa halutaan löytää lyhyin polku solmusta A solmuun K. Samoin tässä Bellman-Ford löytää lyhyimmän polun.

Flood fill algoritmia, testattiin visuaalisesti esimerkissä ”Pipes”. Ohjelma generoi kartan ja asettaa sinne seiniä (#) ja putkita (P). Tämän jälkeen se rikkoo putken kohdassa x,y ja vesi (@) leviää seinien rajaamaan huoneeseen. Algoritmi toimi moitteettomasti useiden suorituskertojen aikana (katso muutamat kuvankaappaukset alla).

```
PIPE IN 6,9 IS BROKEN!
---
################
#@@#@@@@#@@@#@@#
##@@@@@@@@@#@@##
###@@@@@@@@##@@@#
#@@@@@@#@@@@@@@@#
#@@@@#@@@@@@@@@##
#@@@@@@@#@#####@@#
#@@##@@@#..#.###
#@@@#@#@#.#....#
####.#@@#....###
################
```

```
PIPE IN 12,16 IS BROKEN!
---
####################
#....#.P..#@@#@@@@#
#.###.#.P#@@@@@@#@#
#P.P...PP#@##@##@@#
###P..#P#####@@@@#@#
##@#.###@@@#@@@@@@#
#@@@@@@#@#@@##@@@@#
#@@@#@@@@@@@@@@@@@#
#@@@#@@@@@@@@#@@###
#@@#@@@#@@#@@#@#@@#
#@@@@#@@@@@@#@@@#@#
#@@#@@##@#@#P#@@@@#
#@@@@###@#@##@#@@@#
#@#@@@##@@@@@@@#@@#
#@#@@#.#@@@@@@@@#@#
#@@#####@@@##@@@@@@#
###P.#.#@@@@@@#@#@#
####################
```

```
PIPE IN 6,15 IS BROKEN!
---
##########
##.##.#..#
#P..##..##
##.#.#..##
#..#P...##
##.P.P...#
#.#..P.P.#
#....PP.##
#..P.P...#
#...P..#.#
#...#..P##
#.....#.##
#P..P#..##
##..##.#@#
#P..#@##@#
#...#@@@@#
##########
```

A* algoritmia taas testattiin käyttöohjeessa mainitulla esimerkillä Adventurer. Ohjelma suoritettiin monta kertaa eri kokoisilla kartoilla ja sen valintoja analysoitiin. Toisaalta, koska Dijkstran toimiminen on jo todettu vahvaksi, olisi outoa, ettei A star vaikuttaisia toimivan myös. Algoritmin tekemissä valinnoissa ei ilmennyt virheitä (katso muutamat kuvankaappaukset alla).