Ohjelman yleisrakenne

Ohjelmasta löytyy 5 tärkeää pakettia ja 1 paketti performanssitestaukselle. Algorithms ja datastructures paketit sisältävät varsinaisen Tiralabra osuuden, application luokka sisältää ohjelmalle olennaista toiminnallisuutta ja koordinointia, UI taas nimensä mukaisesti käyttöliittymälle olennaisen toiminnallisuuden. Performance luokka ei ole normaalisti käytössä (ks. käyttöohje) ja se sisältää yksinkertaisen testaustoiminnallisuuden sekä ohjelman algoritmit toteutettuna Javan omilla tietorakenteilla vertailua varten.

Saavutetut aika- ja tilavaativuudet

Dijkstra ja A* ovat toteutettu minimikeolla siten, että aika- ja tilavaativuudet ovat näiden kohdalla seuraavat:

Aikavaativuudet:
1. Dijkstra: O((|E| + |V|) log |V|)
2. A*: O((|E| + |V|) log |V|) - A* aikavaatimus on sama kuin Dijkstralla, vaikka käytännössä nopeampi

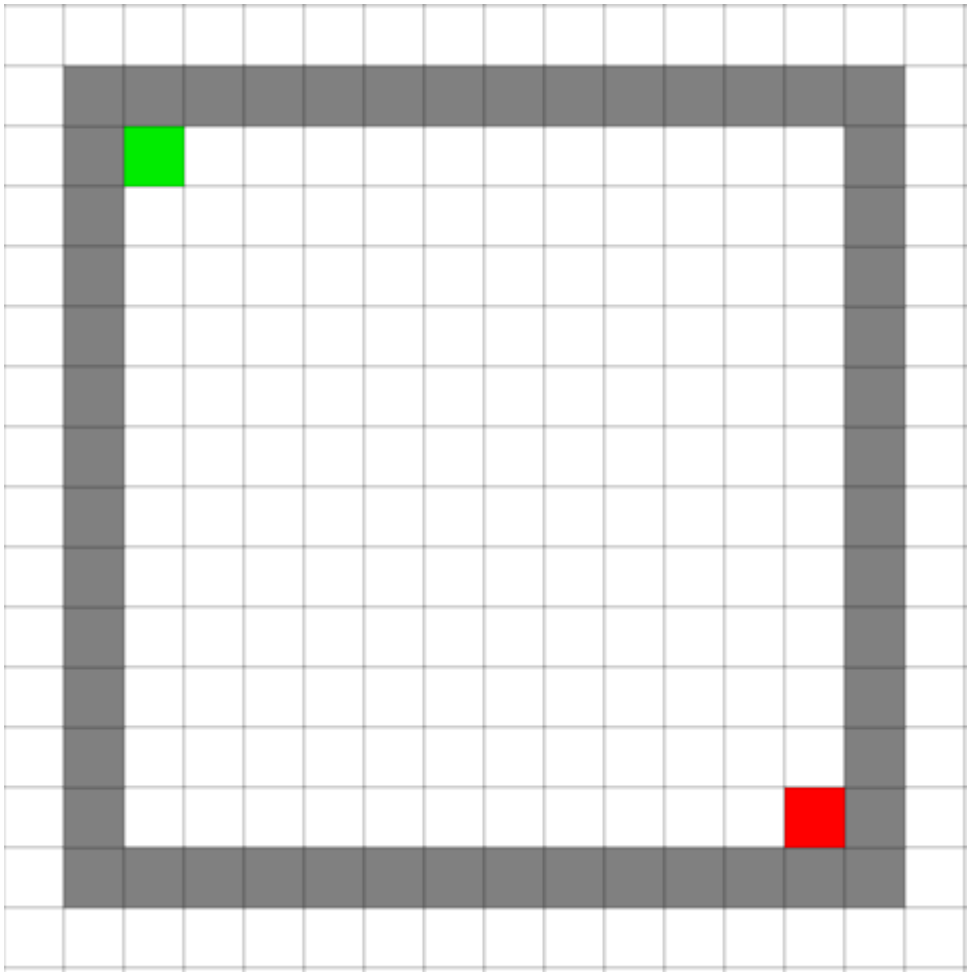
Tilavaativuudet:
1. Dijkstra: O(|V|)
2. A*: O(|V|)

Näissä V=sallitut ruudut (seiniä ei käydä läpi) ja E=sallittujen ruutujen väliset kaaret.

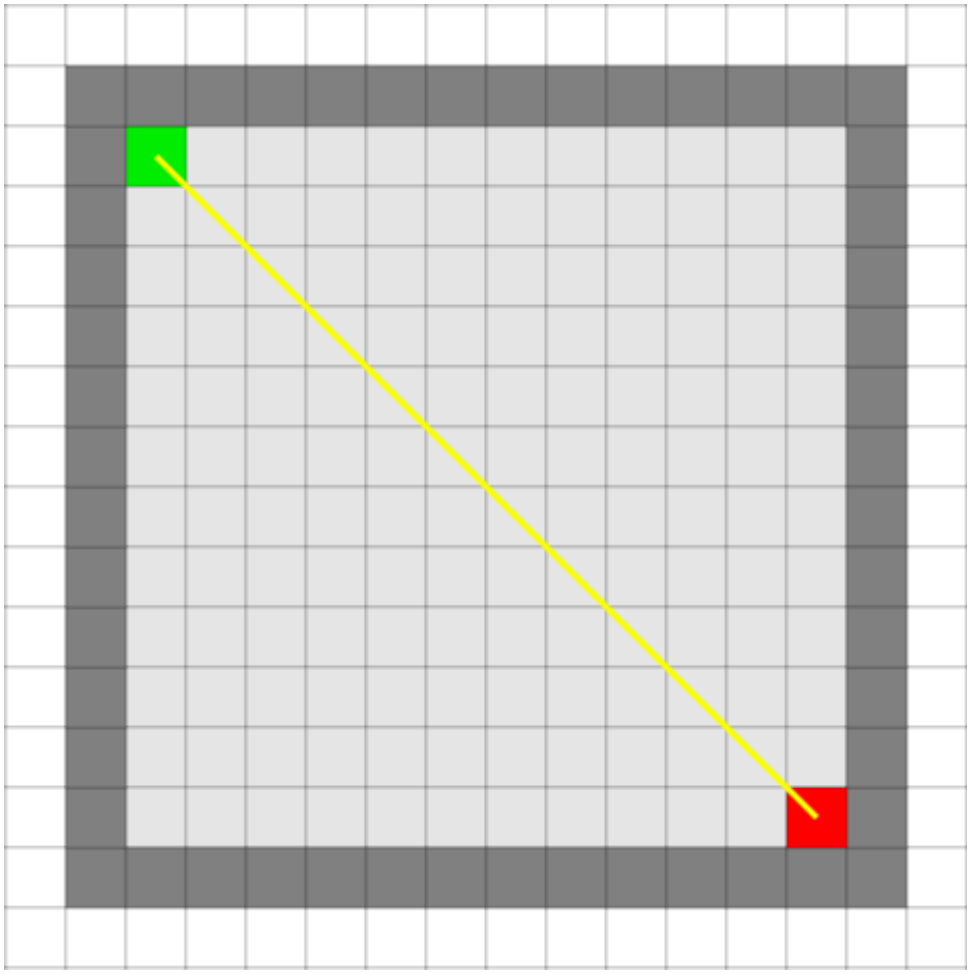
JPS analyysi

JPS algoritmin aika- ja tilavaativuus on hieman vaikeampi kysymys, eikä tätä aihetta kirjallisuudessa juuri olekaan tutkittu. Pahimman tapauksen aikavaativuus on helppo todeta m*n kokoisessa matriisissa, jossa lähtö ja maali ovat mahdollisimman kaukana toisistaan, eli vastakkaisissa kulmissa. Tällöin JPS joutuu rekursiivisesti käymään koko verkon läpi ja aikavaatimukseksi tulee O(n* m). Esimerkki JPS rekursiivisesta läpikäynnistä löytyy kohdasta fig.3. b, täältä: http://harablog.wordpress.com/2011/09/07/jump-point-search/

Seuraavassa demonstraatio äskeisestä päättelystä käytten PathFinding.js ohjelman rekursion visualisointia. Tummat ruudut ovat seiniä, reititys tapahtuu vihreästä punaiseen.



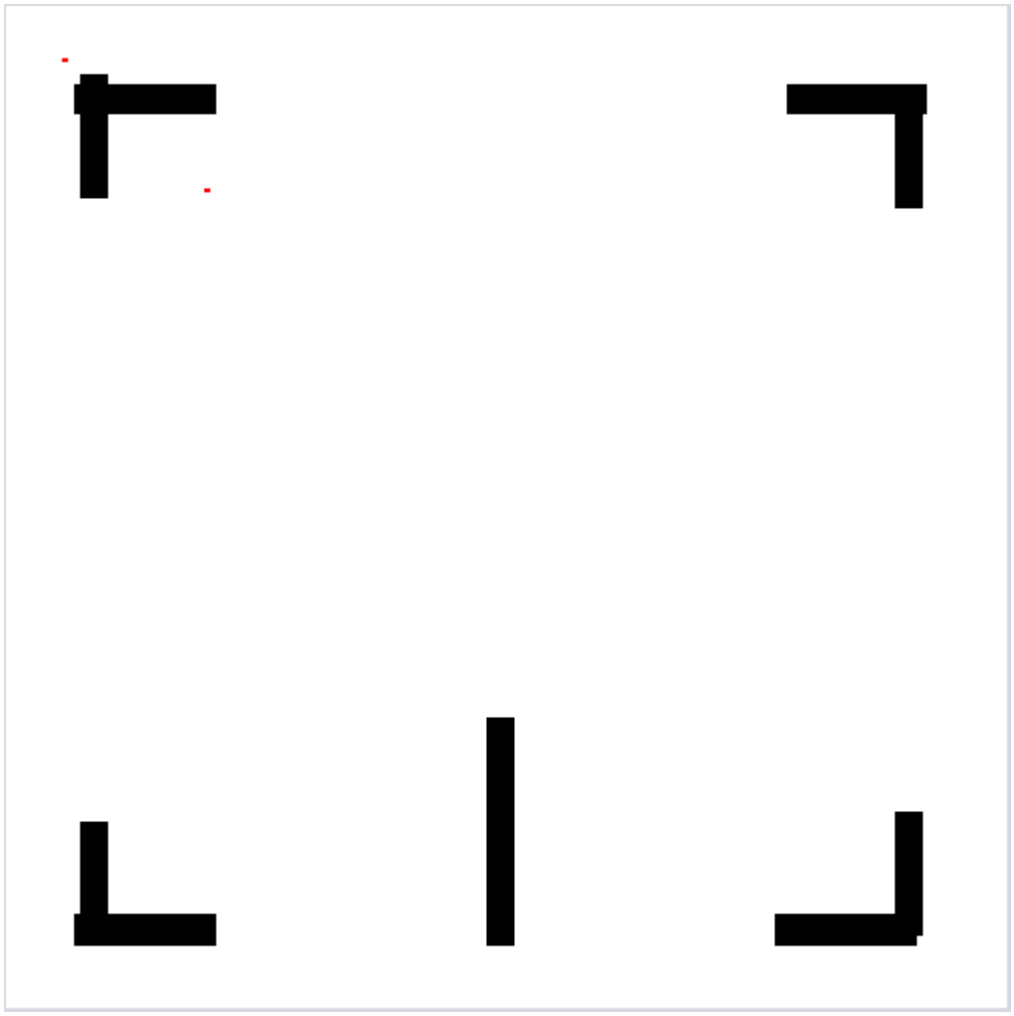
Seuraavassa näkyy rekursion läpikäymät ruudut harmaalla:



JPS algoritmin tilavaativuus on melko vaikea analysoida, mutta pahin tapaus on kiinni joko jump metodin rekursiopinosta tai minimikeon koosta, joka ei voi ylittää O(|V|) vaativuutta. Edellämainitussa esimerkissä jump metodin rekursiopino on pahimmillaan max(korkeus*sqrt(2)-1, leveys-1, korkeus-1), koska jump etenee diagonaalisesti, mutta joka eteemisellä etsii jump-pisteitä vaaka ja korkeussuunnassa.

Suorituskyky- ja O-analyysivertailu

Suorituskykyä on vertailtu eri algoritmien välillä sekä omien ja Javan tietorakenteiden välillä kahdella eri kartalla.
Small:



Large:

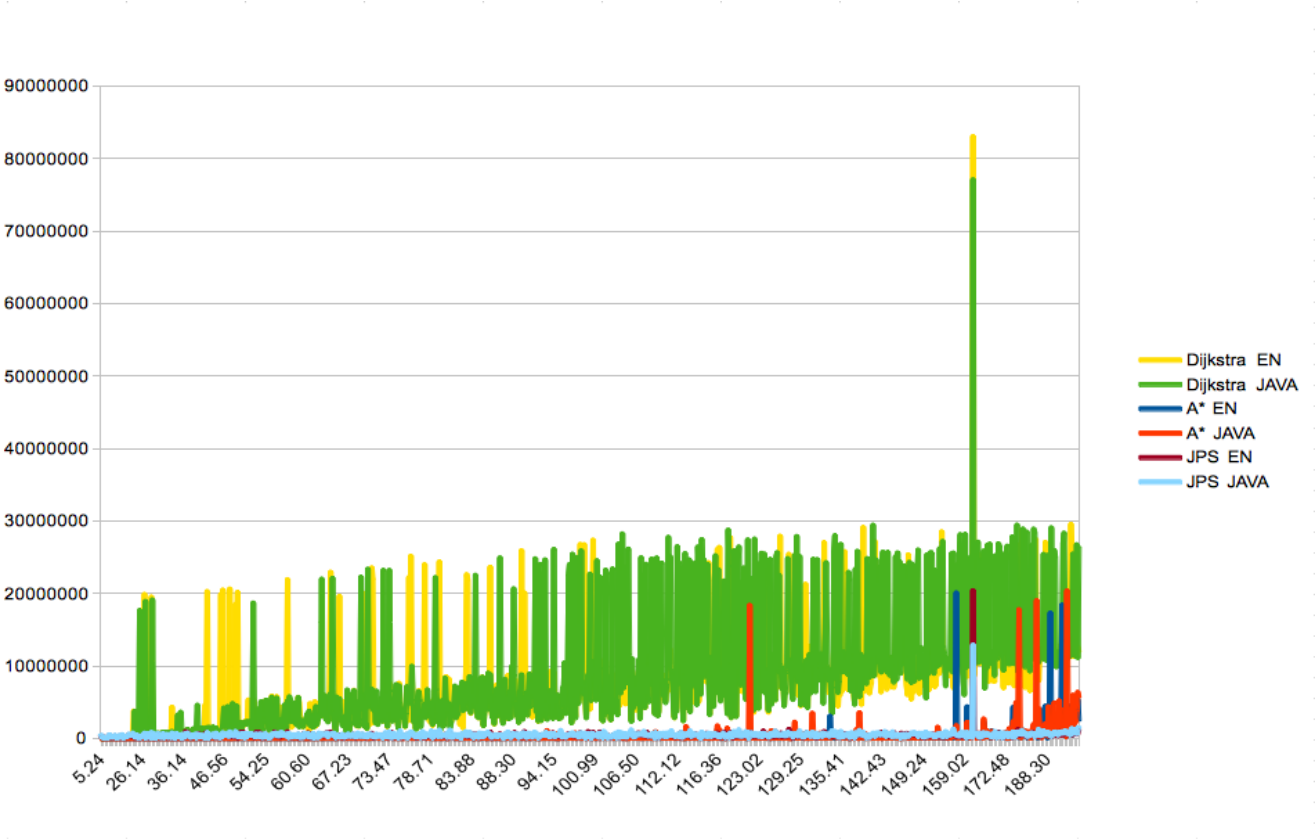
Seuraavissa tuloksissa EN tarkoittaa omilla tietorakenteillani toteuttamiani algoritmeja ja JAVA Javan omilla tietorakenteilla toteutettuja algoritmeja. Algoritmit ovat muuten täysin identtisiä ja mittaukset on pyritty tekemään mahdollisimman reilusti.

Small

Keskiarvoja - nanosekunneissa

	EN	JAVA
Dijkstra	7148919	8962105
A*	291644	349945
JPS	464187	535496

Tulokset taulukossa, x: reitin pituus, y: aika nanosekunneissa



Hämmästyttävästi A* on itseasiassa JPS algoritmia nopeampi, vaikka sen suorituskyky alkaakin suuremmilla etäisyyksillä heittelemään enemmän. Vaikka JPS laskeekin vain pienen määrän hyppypisteitä, käy se rekursiivisesti läpi suuren määrän ruutuja. JPS algoritmin visualisointia onkin helppo väärinkäyttää, johon oikeastaan omakin ohjelmani syyllistyy, koska en visualisoi jump rekursiota.

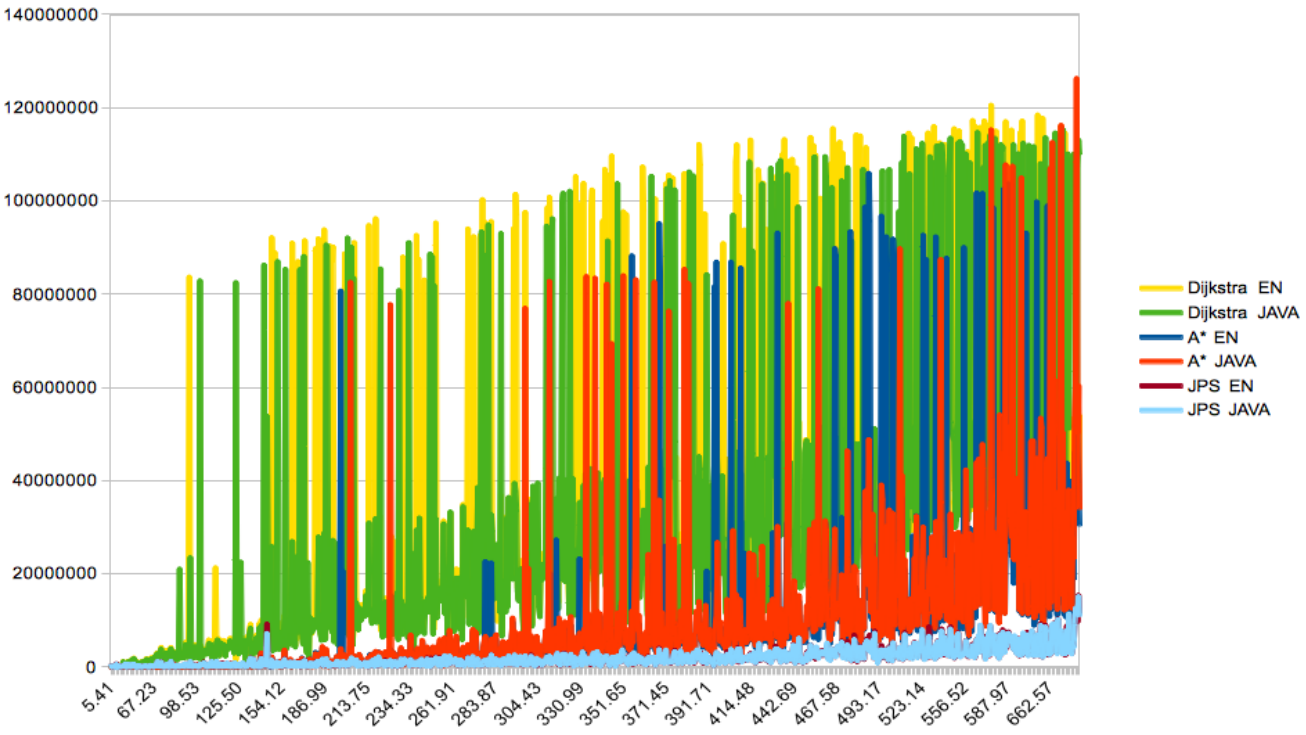
Mielenkiintoisesti myös omat toteutukseni ovat Javan tietorakenteita nopeampia. Tämä todennäköisesti johtuu siitä, että omia tietorakenteita ei ole suunniteltu kovin monipuolisiksi ja olen toteuttanut kekoon päivitysoperaation, kun taas Javan priorityqueue rakenteessa joudutaan heap-decrease-key operaatio toteuttamaan remove ja add operaatioilla.

Large

Keskiarvoja - nanosekunneissa

	EN	JAVA
Dijkstra	32960587	31401687
A*	10388369	10852921
JPS	2015639	2009764

Tulokset taulukossa x: reitin pituus, y: aika nanosekunneissa



Suuremmilla syötteillä tulokset alkavat vaikuttaa oletettuja tuloksia: omat tietorakenteeni eivät ole enää Java nopeampi - vaikkakin hyvin pitkälti samalla viivalla - ja JPS on selvästi nopeampi kuin A*. Mielenkiintoista on myös huomata miten paljon vähemmän JPS suorituskyky heittelehtii kuin A* ja Dijkstra, joilla on välillä suuriakin piikkejä suoritusajassa.

Työn mahdolliset puutteet ja parannusehdotukset

Puutteet

Algoritmit eivät tue painotettuja verkkoja. A* ja sen variantit saisi toki helposti näitä tukemaan, mutta JPS osalta tämä on epävarmaa ainakin suorituskyvyn osalta.

A* toteutuksen pohjalta olisi voinut rakentaa lisääkin reititysalgoritmeja, esim. Best-First Search ja RSR.

Algoritmeja ja tietorakenteita voisi mahdollisesti optimoida vielä enemmän, mutta tämä vaatisi lisää tutkimustyötä sekä tietorakenteisiin ja reititysalgoritmeihin, joka ei tämän kurssin ajankäytön puitteissa minulta onnistunut.

Parannusehdotukset

Pienillä muokkauksilla työstä voisi rakentaa hyvän pohjan 2D grid reititysalgoritmikirjastoksi. Pohjana toimisi LosAlgoritmos luokka, jonka kautta voisi käyttää Algo-rajapinnan toteuttavia reititysalgoritmeja, joita voisi kirjastoon lisätä modulaarisesti.

Lähteet

Huom. lähteistä löytyy toteutuksia tämän työn algoritmeista, mutta en ole kuitenkaan käyttänyt Javalla kirjoitettuja lähteitä algoritmeja varten, joten kaikki koodi on omaa tuotosta.

JPS Blogipostaus ja linkki alkuperäiseen artikkeliin
http://harablog.wordpress.com/2011/09/07/jump-point-search/
http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/viewFile/3761/4007

JPS http://zerowidth.com/2013/05/05/jump-point-search-explained.html

A* heuristiikat http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

PathFinding.js - inspiraation lähde sekä GUI että lähdekoodin osalta
http://qiao.github.io/PathFinding.js/visual/

Jumper - LUA toteutus JPS algoritmista https://github.com/Yonaba/Jumper

Kartat http://www.movingai.com/benchmarks/