

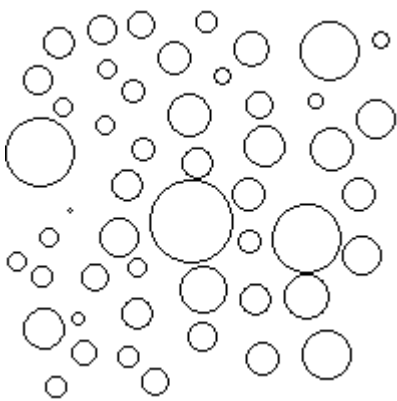
Testausdokumentti

Testaus

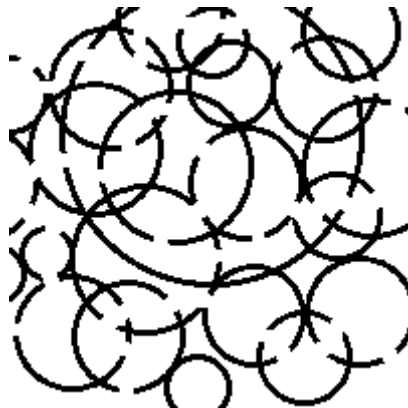
Testasin ohjelman eri algoritmien suoritusnopeuksia useilla syötekuvilla. Suoritin yhden algoritmin samalle syötteelle 10-300 kertaa syötteen koosta riippuen ja laskin keskiarvon siihen kuluneista ajoista. Toistin tämän joka algoritmille kaikilla syötekuvilla, ja lisäsin tulokset diagrammeihin.

Käytetyt syötteet

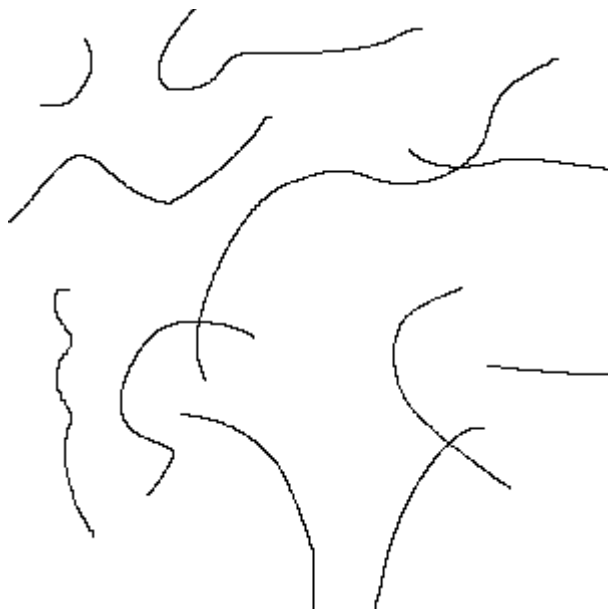
Syötteinä testeissä käytettiin seuraavia kuvia:



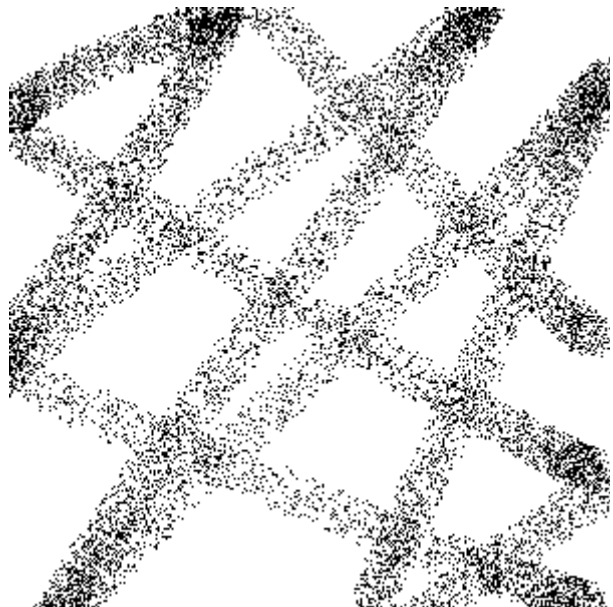
circles.png



circles2.png



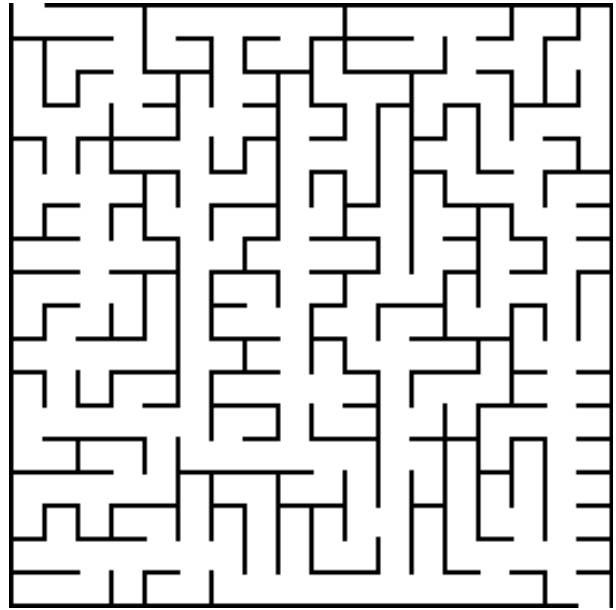
lines.png



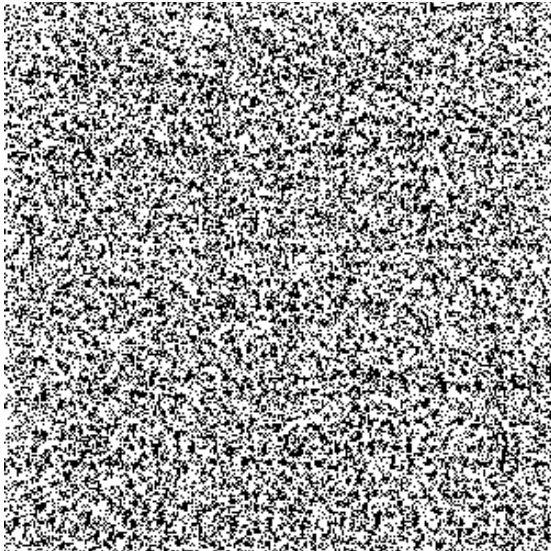
spary.png



horizontal.png



maze.png



noise.png

sekä kaksi muuta kuvaa, *no_obstacles.png* ja *unsolvable.png*, joista ensimmäinen on täysin valkoinen ja kooltaan 600x100 px, ja toinen on täysin valkoinen 1500x1500 px, mutta sen oikean alakulman pikseli on ympäröity mustalla, joten reitinetsintä ei löydä sitä.

Jokaisella syötekuvalla aloituspisteeksi määriteltiin vasen yläkulma ja lopetuspisteeksi oikea alakulma, paitsi kuvalle *horizontal.png*, jolla aloituspiste oli (250, 20) ja lopetuspiste (250, 480) sekä kuvalle *no_obstacles.png*, jolla aloituspiste oli (0, 50) ja lopetuspiste (599, 50).

Testien toistaminen

Testien suorittamiseen käytettiin seuraavaa komentoa:

```
java -jar Tiralabra.jar -i syöte -s lähtö -g maali -a ALGORIMI -c toistot --less
```

Eli esimerkiksi A*:n ajamiseen circles2.png -kuvalla 100 kertaa vasemmasta yläkulmasta oikeaan alakulmaan tulostaen vain kuluneet ajat onnistuu komennolla:

```
java -jar Tiralabra.jar -i circles2.png -s 0,0 -g 199,199 -a A_STAR -c 100 --less
```

Lisätietoja komentorivioptioiden käytöstä löytyy käyttöohjeesta.

Tulokset

(suoritusaikojen kuvaajat ovat seuraavalla sivulla)

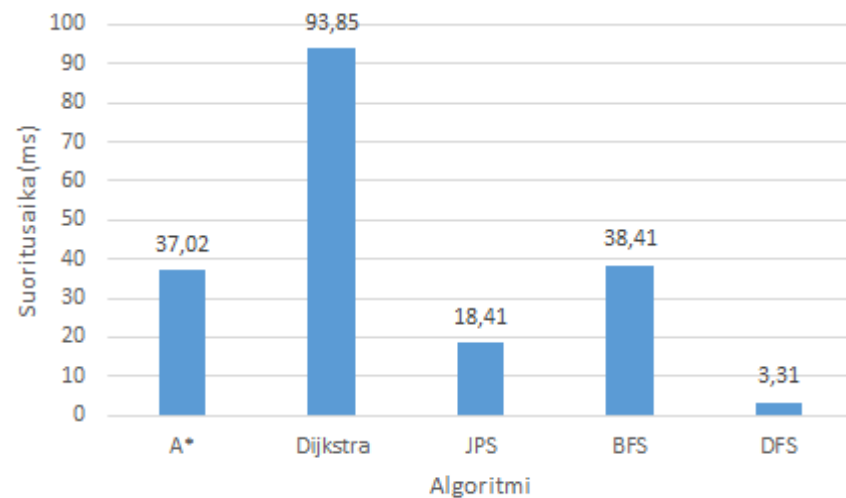
Kuvaajista näkee selvästi, että Dijkstran algoritmi suoriutuu jokaisesta syötteestä huonoiten. Se ei siis ole kovin hyvä valita verkoille, jotka ovat tasaisia ruudukkoita ja joissa jokaisella solmulla on paljon naapureita. Sen lisäksi, että algoritmi ei käytä mitään heuristiikkaa suoritusnopeuden parantamiseksi, se suorittaa jokaisella iteraatiolla update-key() -operaation keolle, joka sisältää verkon jokaisen solmun (pienenee kyllä suorituksen aikana). Update-key() -operaation aikavaativuus on $O(\log n)$.

A* on selvästi Dijkstran algoritmia nopeampi kaikissa testatuissa tapauksissa. Sen suoritusnopeus on vain murto-osa tähän verrattuna, parhaimmillaan se laskee saman reitin käyttäen vain 2 % Dijkstran algoritmien käyttämästä ajasta. A* käyttääkin tehokasta heuristiikkaa päättääkseen, mitä solmuja kannattaa tutkia, ja säilyttää keossa vain pienen määrän solmuja pitäen keko-operaatiot kevyinä.

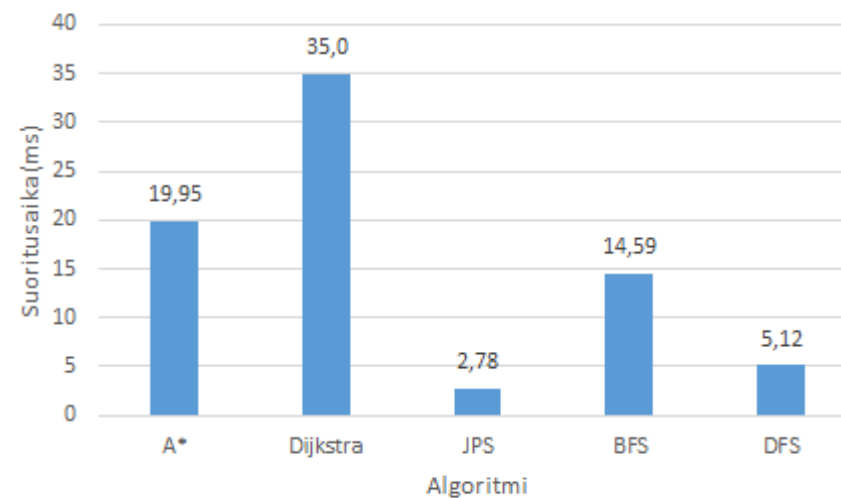
Jump point search on vielä tehokkaampi. Se peittoaa A*:n syötteestä riippuen jopa 2–50 kertaisella nopeudella. Vain yhdellä syötteellä, missä polku on täysin suora esteetön viiva, A* on marginaalisesti nopeampi. JPS:n tehokkuus perustuu ovelaan keinoon, missä suuri osa verkon solmuista eliminoidaan ja hypään niiden yli tutkimatta niitä.

Breadth-first search ja erityisesti depth-first search ovat molemmat melko nopeita, mutta kummatkaan eivät takaa palauttavansa lyhintä reittiä. BFS on optimaalinen ainoastaan, jos verkon jokainen kaari on yhtä pitkä, mutta tässä tapauksessa kaaria on sekä 1:n että $\sqrt{2}$:n mittaisia. Näiden algoritmien nopeus perustuu siihen, että ne molemmat käyttävät perustietorakenteenaan linkitettyä listaa ja sen vakioaikaisia operaatioita.

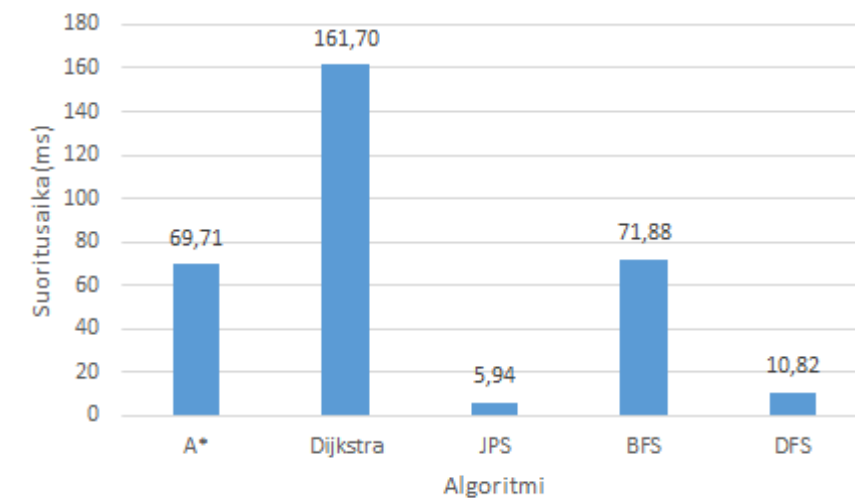
Suoritus aika syötteellä noise.png



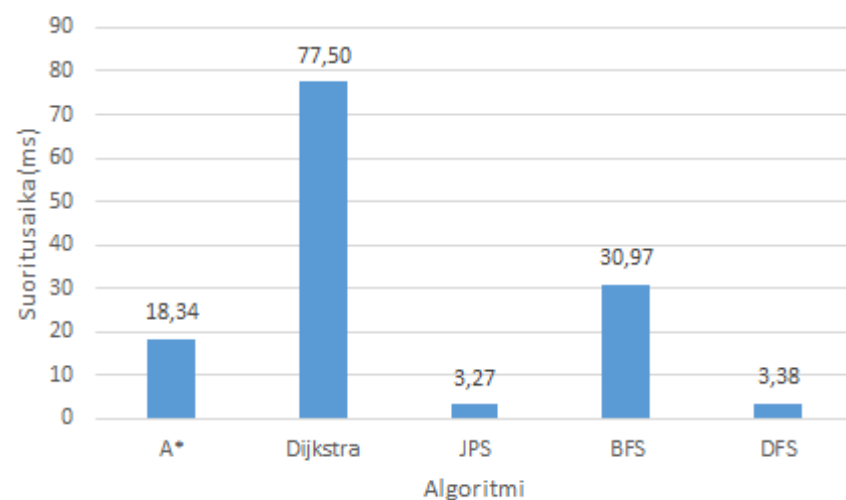
Suoritus aika syötteellä circles2.png



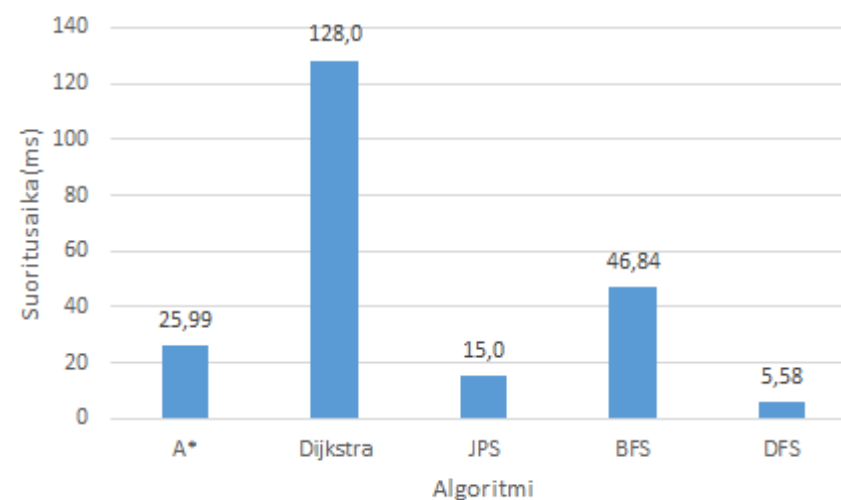
Suoritus aika syötteellä lines.png



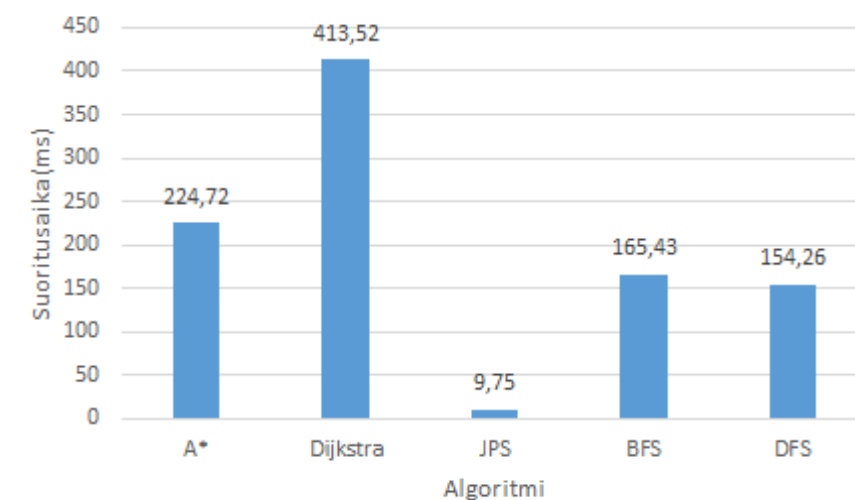
Suoritus aika syötteellä maze.png



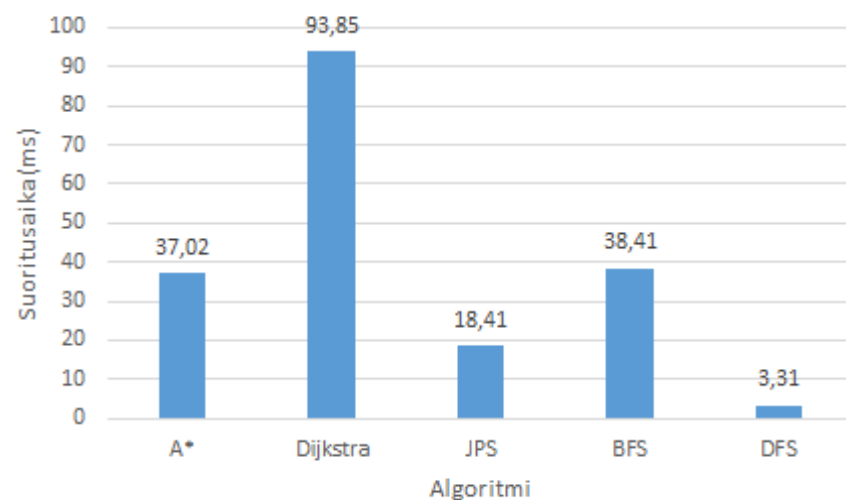
Suoritus aika syötteellä spray.png



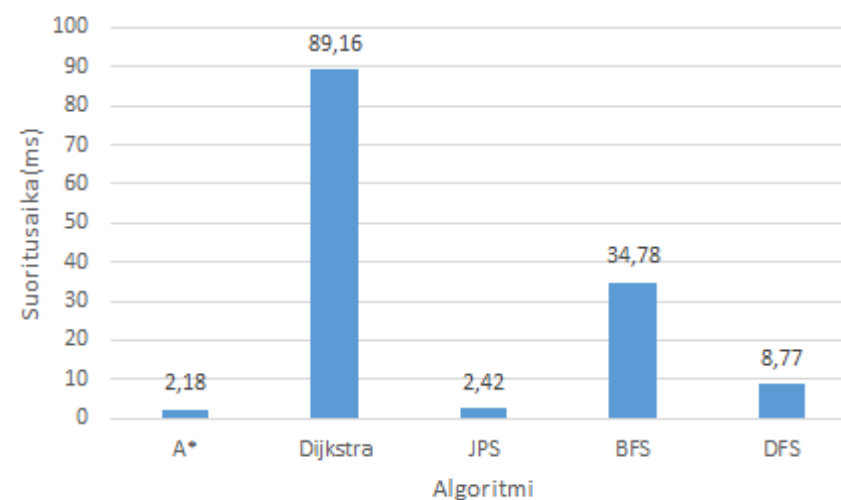
Suoritus aika syötteellä horizontal.png



Suoritus aika syötteellä noise.png



Suoritus aika syötteellä no_obstacles.png



Suoritus aika syötteellä unsolvable.png

