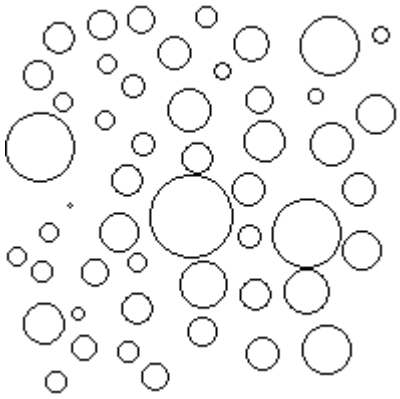
Testausdokumentti

Testaus

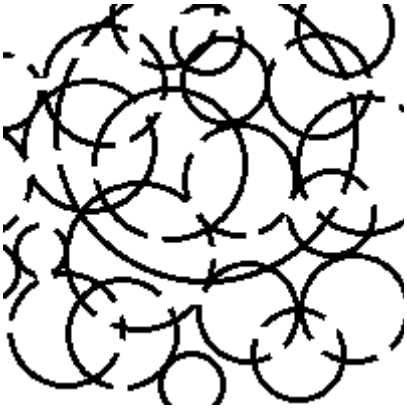
Testasin ohjelman eri algoritmien suoritusnopeuksia useilla syötekuvilla. Suoritin yhden algoritmin samalle syötteelle 10-300 kertaa syötteen koosta riippuen ja laskin keskiarvon siihen kuluneista ajoista. Toistin tämän joka algoritmille kaikilla syötekuvilla, ja lisäsin tulokset diagrammeihin.

Käytetyt syötteet

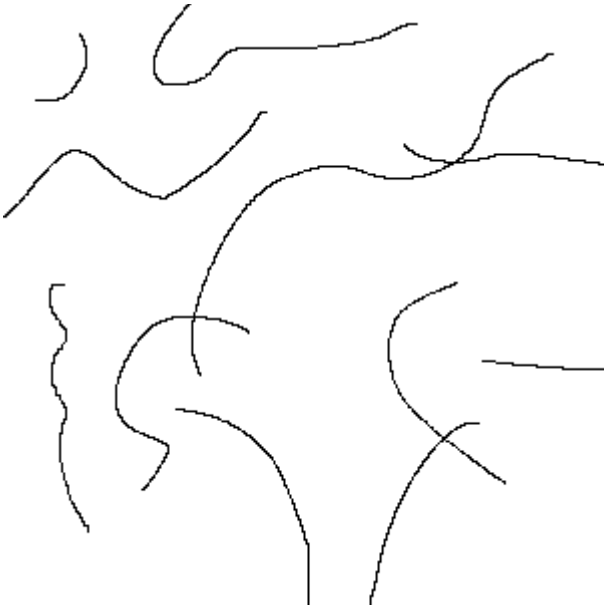
Syötteinä testeissä käytettiin seuraavia kuvia:



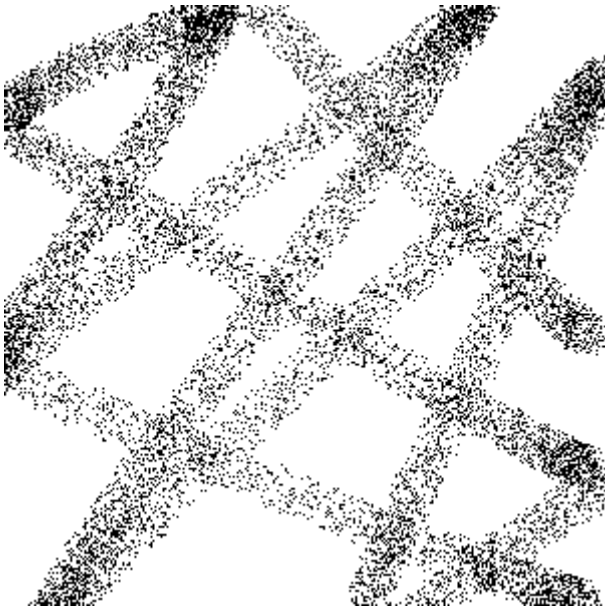
circles.png



circles2.png

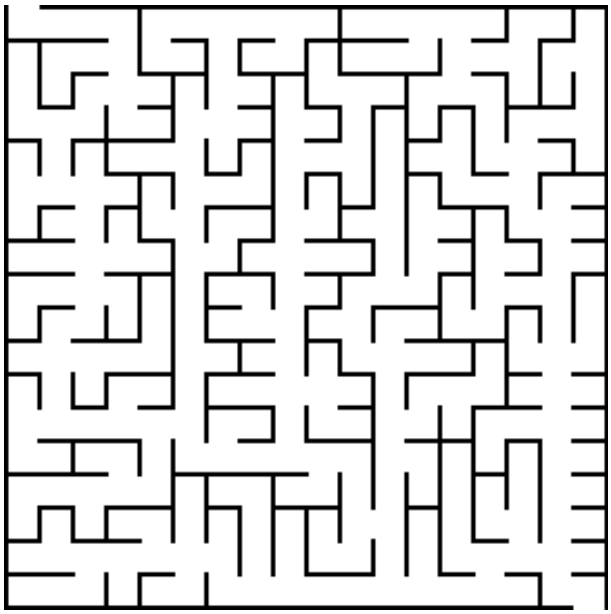


lines.png

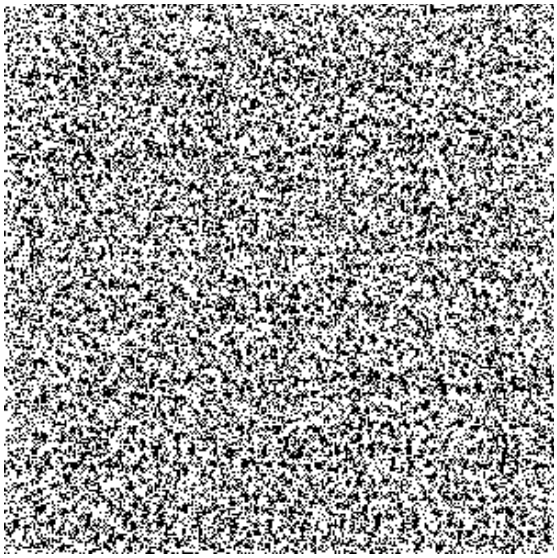


spray.png

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

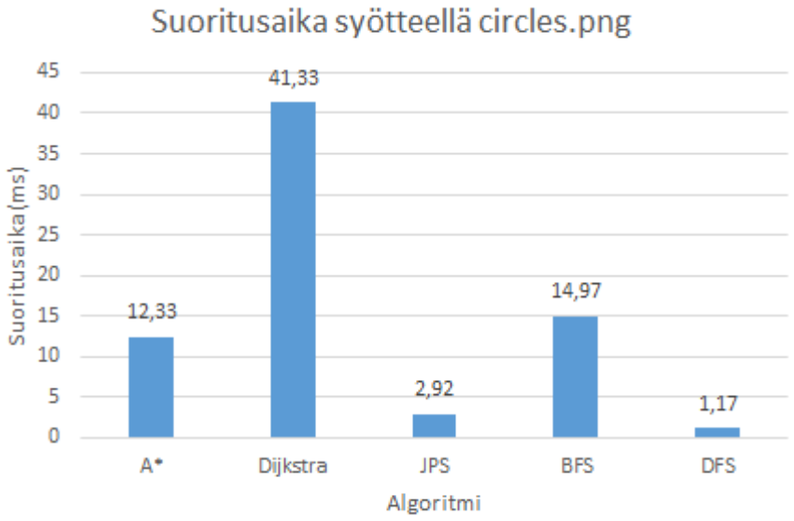
(suoritusaikojen kuvaajat ovat seuraavalla sivulla)

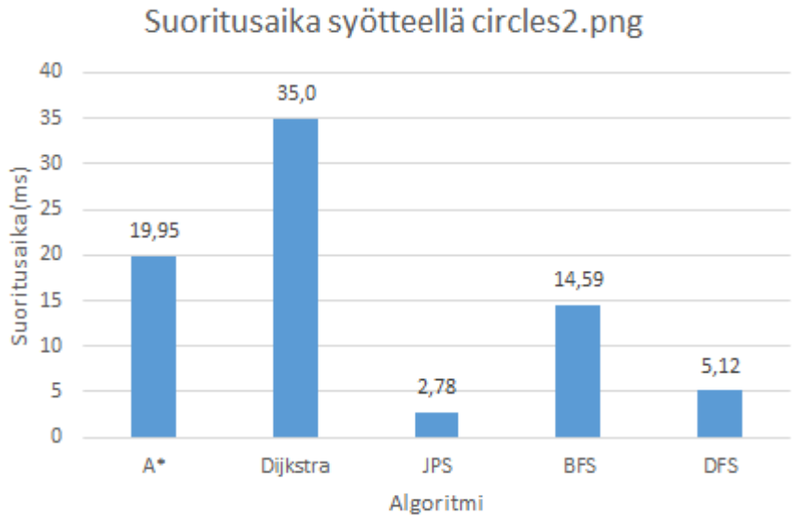
Kuvaajista näkee selvästi, että Dijkstran algoritmi suoriutuu jokaisesta syötteestä huonoiten. Se ei siis ole kovin hyvä valita verkoille, jotka ovat tasaisia ruudukoita ja joissa jokaisella solmulla on paljon naapureita. Sen lisäksi, että algoritmi ei käytä mitään heuristiikkaa suoritusnopeuden parantamiseksi, 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).

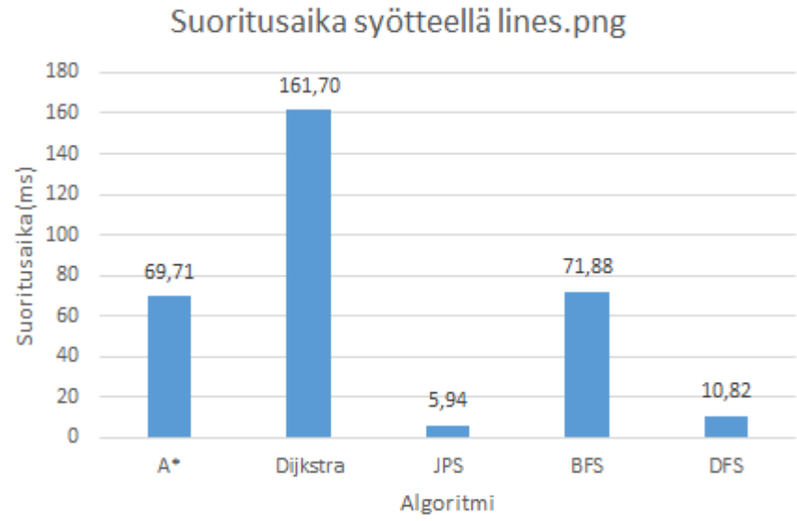
A* on selvästi Dijkstran algoritmia nopeampi kaikissa testatuissa tapauksissa. Sen suoritusnopeus on vain murto-osa tähän verrattuna, parhaimmillaan se laskee saman reitin käyttäen vain 2 % Dijkstran algoritmin käyttämästä ajasta. A* käyttääkin tehokasta heuristiikkaa päättääkseen, mitä solmuja kannattaa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

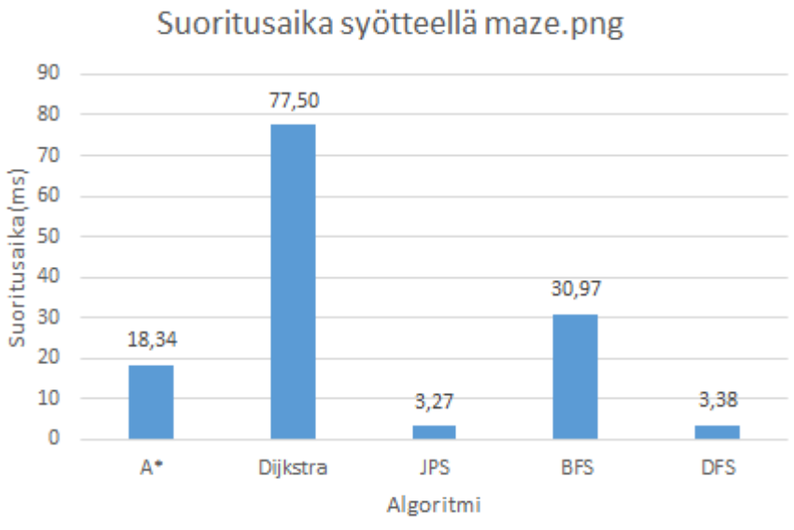
Jump point search on vielä tehokkaampi. Se peittoaa A*:n syötteestä riippuen jopa 2–50 kertaisella nopeudella. Vain yhdellä syötteellä, missä polku on täysin suora esteetön viiva, A* on marginaalisesti nopeampi. JPS:n tehokkuus perustuu ovelaan keinoon, missä suuri osa verkon solmuista eliminoidaan ja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

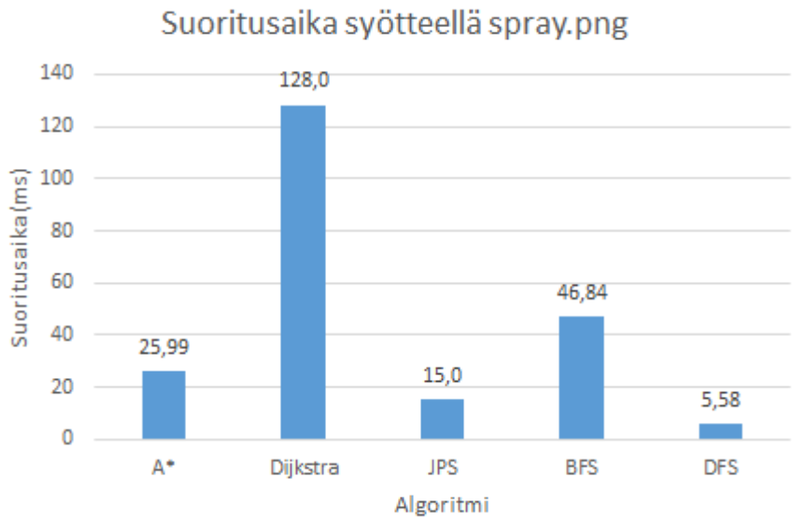
Breadth-first search ja ja erityisesti depth-first search ovat molemmat melko nopeita, mutta kummatkaan eivät takaa palauttavansa lyhintä reittiä. BFS on optimaalinen ainoastaan, jos verkon jokainen kaari on yhtä pitkä, mutta tässä tapauksessa kaaria on sekä 1:n että √2:n mittaisia. Näiden algoritmien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

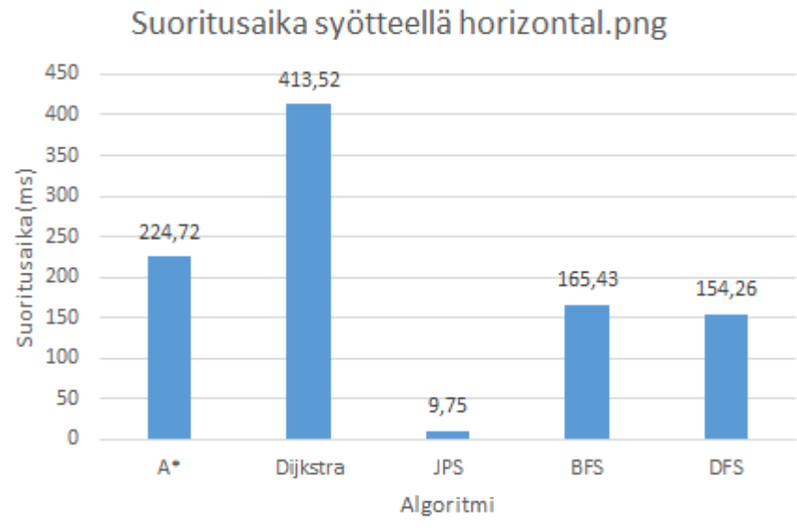

Suoritusaika syötteellä circles.png
Suoritusaika(ms)
45
40
35
30
25
20
15
10
5
0
12,33
41,33
2,92
14,97
1,17
A*
Dijkstra
JPS
BFS
DFS
Algoritmi

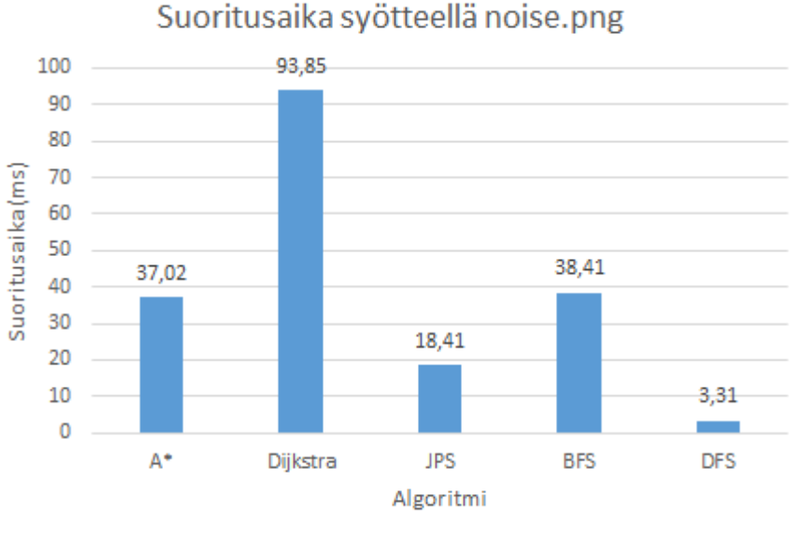

Suoritusaika syötteellä circles2.png
Suoritusaika(ms)
40
35
30
25
20
15
10
5
0
19,95
35,0
2,78
14,59
5,12
A*
Dijkstra
JPS
BFS
DFS
Algoritmi

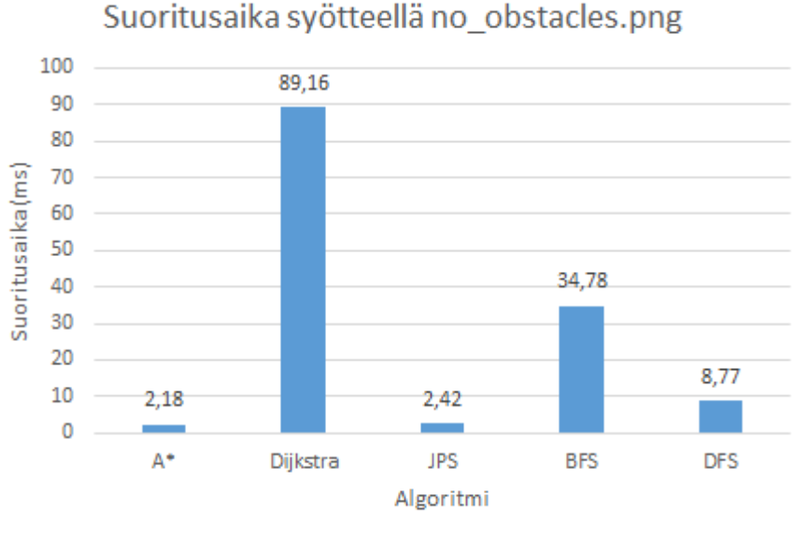

Suoritusaika syötteellä lines.png
Suoritusaika(ms)
180
160
140
120
100
80
60
40
20
0
69,71
161,70
5,94
71,88
10,82
A*
Dijkstra
JPS
BFS
DFS
Algoritmi

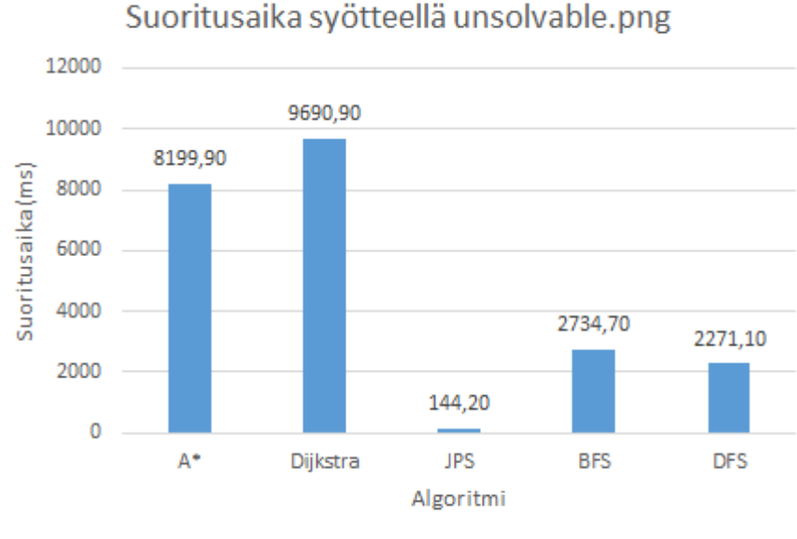

Suoritusaika syötteellä maze.png
Suoritusaika(ms)
90
80
70
60
50
40
30
20
10
0
18,34
77,50
3,27
30,97
3,38
A*
Dijkstra
JPS
BFS
DFS
Algoritmi


Suoritusaika syötteellä spray.png
Suoritusaika(ms)
140
120
100
80
60
40
20
0
25,99
128,0
15,0
46,84
5,58
A*
Dijkstra
JPS
BFS
DFS
Algoritmi


Suoritusaika syötteellä horizontal.png
Suoritusaika(ms)
450
400
350
300
250
200
150
100
50
0
224,72
413,52
9,75
165,43
154,26
A*
Dijkstra
JPS
BFS
DFS
Algoritmi


Suoritusaika syötteellä noise.png
Suoritusaika(ms)
100
90
80
70
60
50
40
30
20
10
0
37,02
93,85
18,41
38,41
3,31
A*
Dijkstra
JPS
BFS
DFS
Algoritmi


Suoritusaika syötteellä no_obstacles.png
Suoritusaika(ms)
100
90
80
70
60
50
40
30
20
10
0
2,18
89,16
2,42
34,78
8,77
A*
Dijkstra
JPS
BFS
DFS
Algoritmi


Suoritusaika syötteellä unsolvable.png
Suoritusaika(ms)
12000
10000
8000
6000
4000
2000
0
8199,90
9690,90
144,20
2734,70
2271,10
A*
Dijkstra
JPS
BFS
DFS
Algoritmi

Dijkstran algoritmin optimointi

Kuten toteutusdokumentissa mainitsin, käytin alun perin Dijkstran algoritmin toteutuksessa tavallista binäärikekoa. Myöhemmin muutin sen käyttämään indeksoitua kekoa, jossa hakuoperaatio toimii vakioajassa. Tämä laski algoritmin aikavaativuuden $O(|V|^2)$:sta $O(|V| \log |V|)$:hen. Alla on muutama suoritusaikadiagrammi aikaisemmasta versiosta. Ero on huomattava, erityisesti todella isolla 'huonoimman tapauksen' syötteessä (viimeinen diagrammi). Muiden algoritmien suoritusnopeudet saattavat vaihdella hieman edellisen sivun arvoista, sillä testaus suoritettiin eri aikoina, joten testausympäristö ei ollut välttämättä täysin identtinen.