

Testausdokumentti

- *Mitä on testattu, miten tämä tehtiin
- *Minkälaisilla syötteillä testaus tehtiin (vertailupainotteisissa töissä tärkätä)
- *Miten testit voidaan toistaa
- *Ohjelman toiminnan empiirisen testauksen tulosten esittäminen graafisessa muodossa.
- *Testaus on ideaalitapauksessa suoritettava ohjelma. Tällöin testi on helposti toistettavissa, mikä helpottaa toteutuksen tekoa jo varhaisessa vaiheessa. On erittäin suositeltavaa käyttää testaukseen JUnitia.

Testit

- * Itse toteutettujen tietorakenteiden (tira-paketti) kaikki julkiset metodit on yksikkötestattu, lukuunottamatta Pari-luokkaa.
- * AStar-haulla on muutama testi (haku-paketti): haku ylipäänsä toimii ja eri laskimet reiteille toimivat.
- * App-luokassa (käynnistysluokka) on kattavasti debug-metodeja, joilla olen toteutusvaiheessa varmistanut luokkien toimivuutta.
- * Useissa luokissa on lisäksi mukana debug-metodeja, jotka keräävät ja tulostavat/palauttavat tietoa luokkien toiminnasta.
- * App-luokan testikäyttöliittymällä voi testata reitinhakua sekä pysäkkiverkossa että satunnaaisgeneroidussa verkossa. Reittihaku suoritetaan aina kymmenen kertaa, ja suorituksen jälkeen tulostetaan saatu reitti ja keskiarvo kuluneesta ajasta, pienin aika sekä suurin aika.
- * Gui-luokan testikäyttöliittymässä voi luoda uuden satunnaisen verkon sekä suorittaa hakuja pysäkki- ja satunnaisessa verkossa. Haun tulokset piirretään.

Testien tuloksia: suorituskykytestausta

Kuvissa kuljettu reitti on punaisella, käydyt solmut vaaleanpunaisella, alkusolmu keltaisella ja loppusolmu vihreällä.

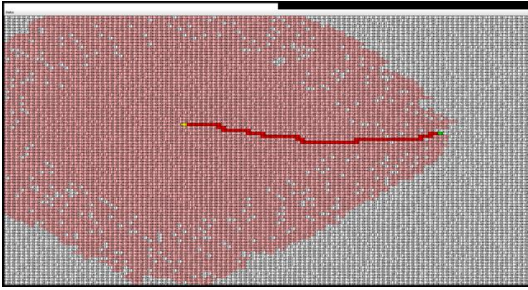
1. AStar-haku ja heuristiikka

Toistaminen:

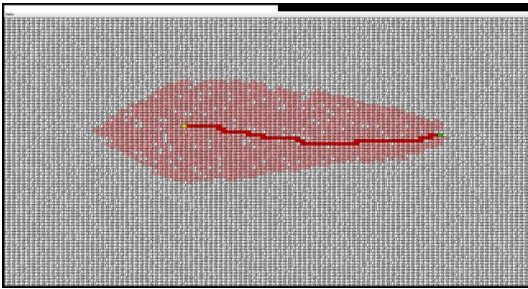
Asetetaan aluksi käyttöön uusi satunnainen verkko syötteellä 300:300:1:2:2:1. Valitaan sitten laskimeksi BFS, suoritetaan haku ja sitten valitaan heuristinen laskin ja suoritetaan haku uudestaan.

Solmujen käsittelyjärjestyksen määrää niiden hyvyys $g(n)$: reitin kustannus $f(\text{alku}, n)$ ja arvioitu jäljellä oleva kustannus $h(n)$.

Heuristisen funktion $h(n)$ arvo vaikuttaa haun toimintaan. Jos $h(n)=0$, saadaan Dijkstra-haku (leveyssuuntaisen vastine):

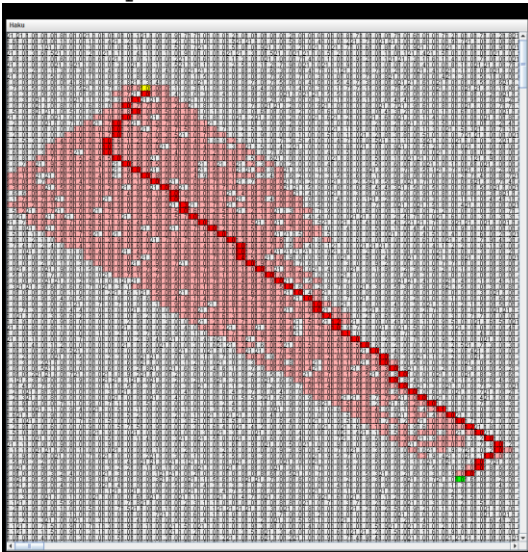


Sama haku suoritettuna siten, että $h(n) = |n.x - \text{maali}.x| + |n.y - \text{maali}.y|$:



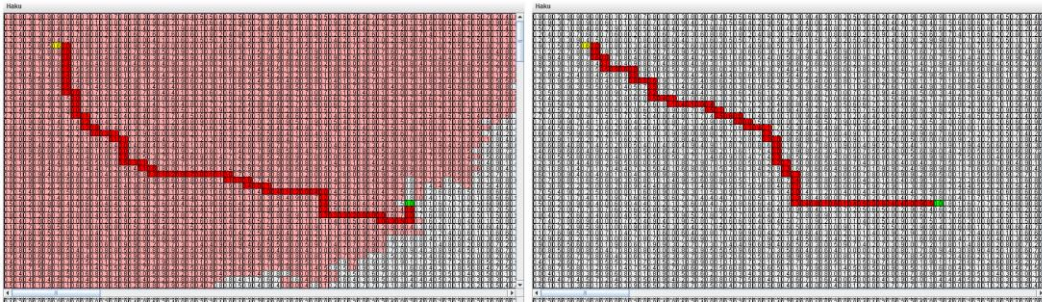
Solmuja ei ole siis käsitelty niin paljoa turhaan.

Mitä paremmin $h(n)$ vastaa oikeaa jäljellä olevaa kustannusta $f(n, \text{maali})$, sitä tehokkaammin haku toimii. Tätä voi kokeilla valitsemalla verkon siten, että minimiPaino on hyvin suuri. Jos valitaan kerroinPaino siten, että syntyvät verkon painot ovat arvoltaan lähellä yhtä, haku toimii tehokkaasti:



Jos heuristinen arvio yliarvioi jäljellä olevaa kustannusta, ei AStar-haku anna parasta mahdollista ratkaisua. Omassa toteutuksessani koko haku hidastuu ja saattaa kaatua tällöin (johtuu käsittelyjärjestyksen toteutustavasta: nykyisellään vaatimuksena on aikavaatimukselle $O(1)$ on, että $g(n)$ ovat kasvavia käsittelyjärjestyksessä edetessä). Tämän voi toistaa valitsemalla minimiPainon, joka on arvoltaan alle yksi. Myös löydetty reitti ei ole enää paras

mahdollinen:



Kuvassa sama haku käyttäen Dijkstra-hakua ja heuristiikkaa, joka yliarvioi jäljellä olevaa kustannusta.

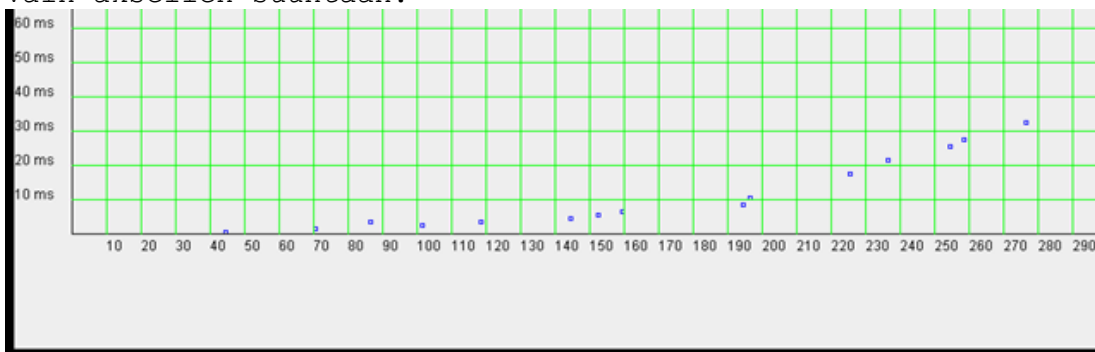
2. AStar-haku ja verkon tiheys

Toistaminen: syötteellä 300:300:1:2:2:1 luotu verkko ja syötteellä 300:300:1:2:2:0 luotu verkko. Suoritetaan näillä testejä-painikkeen mukaiset testit.

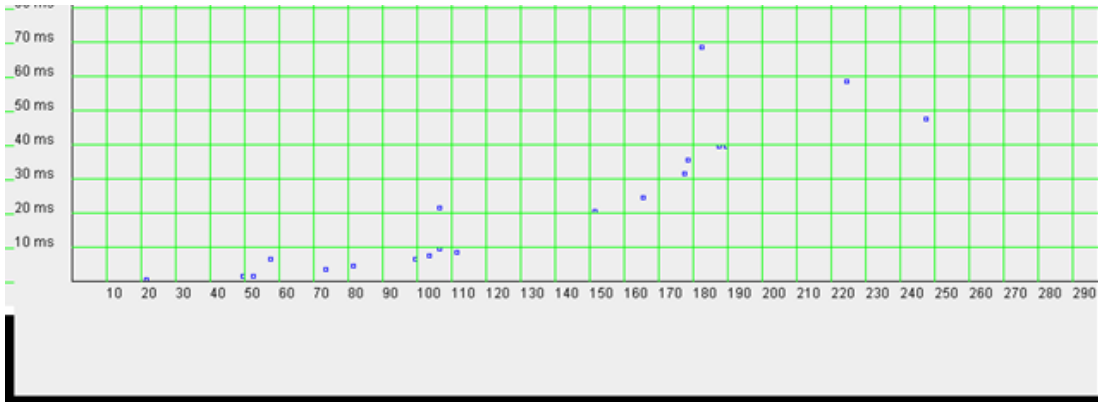
Oletusarvona verkossa voi kulkea vain koordinaattiakselien suuntaisesti. Valinnalla 300:300:1:2:2:0 luotu verkko sallii liikkumisen myös sivuittain.

Minimиаikavaatimus verkossa kulkemiseen on $O(k \cdot d)$, jossa k on naapureiden lukumäärä kullekin solmulle ja d on reitin pituus. Tavallisessa tilanteessa kaikkien käsiteltyjen solmujen n naapurit k lisätään käsittelyjärjestykseen: saadaan aikavaatimus $O(n \cdot k)$. Sivuttaisen liikkumisen sallivassa verkossa naapureita on 8, vain akselien suuntaisen liikkumisen sallivassa verkossa 4. Annetulla reitin pituudella d minimиаikavaatimuksien suhde on siis 2. Jos sivuttaissuuntainen kulkeminen sallitaan, lyhyin mahdollinen reitti on puolet siitä jos kulkeminen estetään. Pahimmillaan sivuttaisen liikkumisen sallivassa verkossa käsiteltyksi tulee paljon suurempi määrä solmuja kuin vain akselien suuntaan liikkuessa.

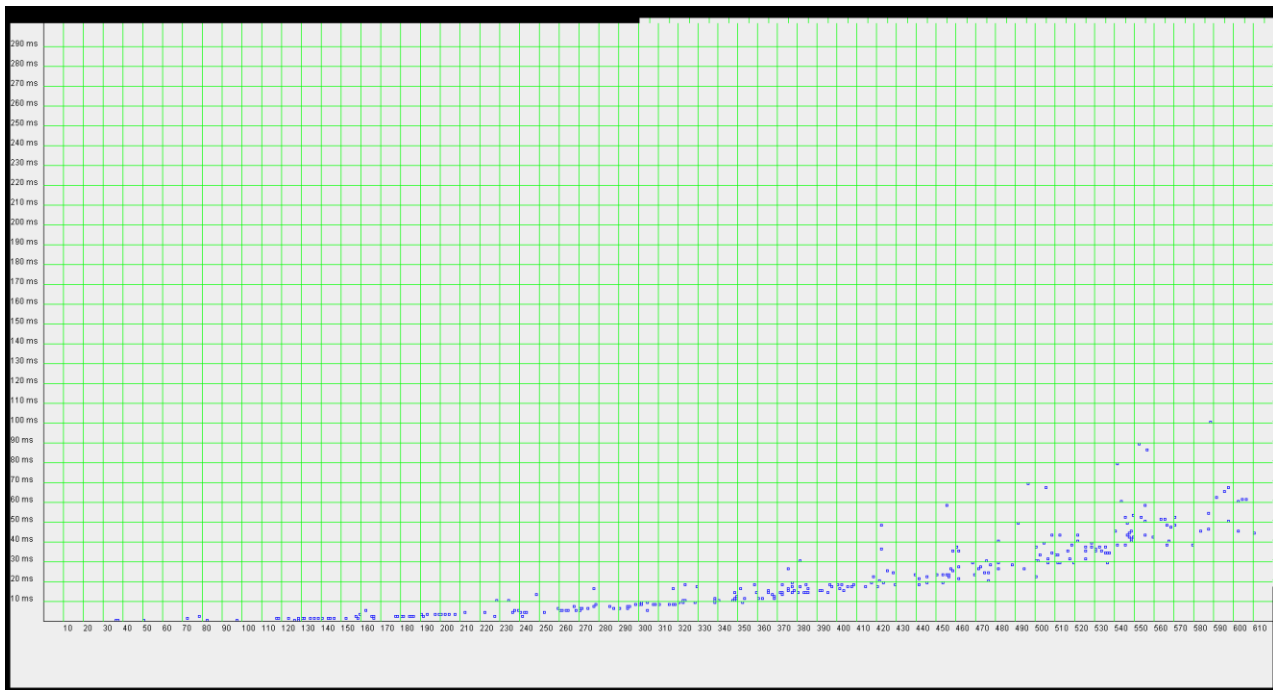
Vain akselien suuntaan:



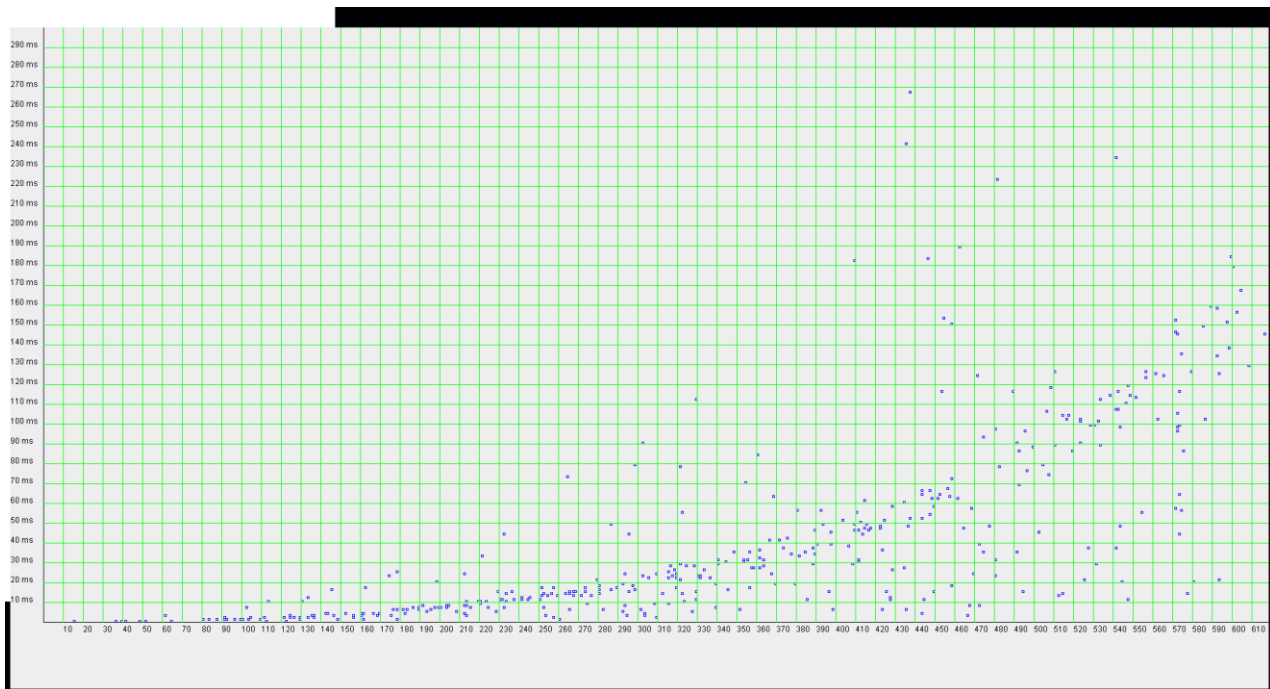
Myös sivusuunnassa:



900x900 verkossa 500 näytteellä ero korostuu entisestään. Vain akselien suuntaan:



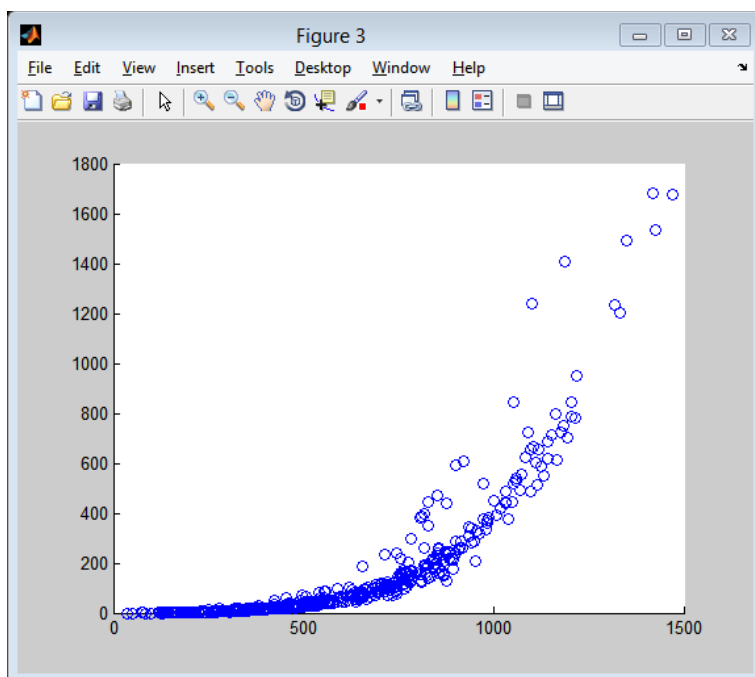
Pahimmillaan suoritusaika on eksponentiaalinen $O(k^d)$, tämä näkyy parhaiten kuvassa, jossa on sallittu liikkuminen myös sivusuunnassa:



3. Verkon ja polun koko

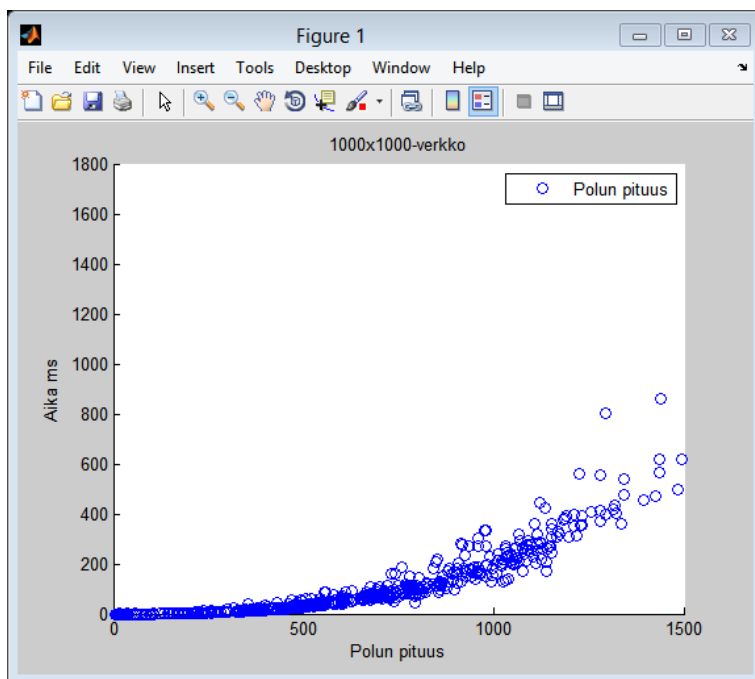
Toistaminen: luodaan syötteellä 1000:1000:1:2:2:1 verkko, jonka koko on 1000x1000. Nyt ohjelman piirtämään kuvaajaan ei mahdu kaikki pisteet. Myös kokonaissuoritus aika on melkoinen.

Suurilla verkoilla ja siten myös pitkillä poluilla aikavaatimus kasvaa. Kuvassa 500 hakua 900x900 -verkossa.



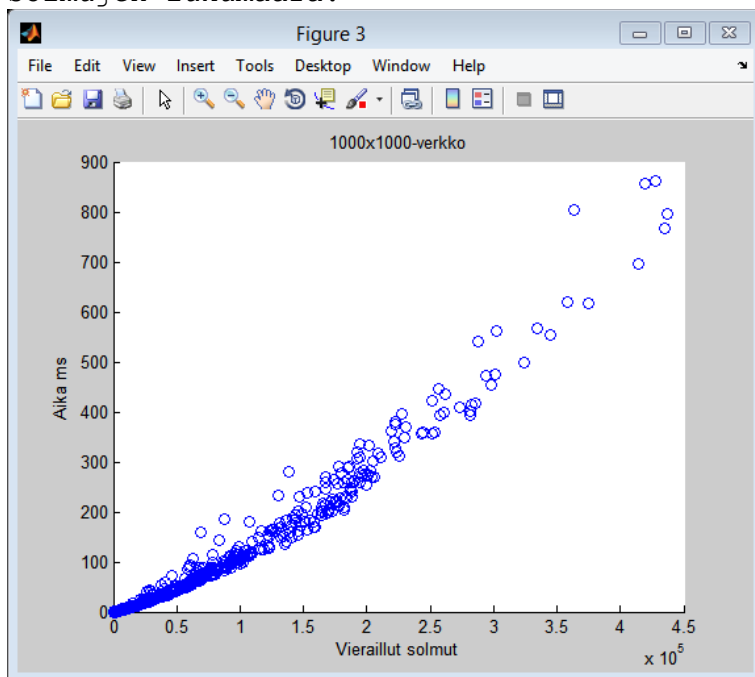
Muokattuani jo vierailtuja solmuja sisältävää hajautuslistaa siten, että sen

uudelleenhajautus toimii paremmin, sain 1000x1000 (parametrit 1000:1000:1:1:2:1) verkossa, jossa saa liikkua vain akselien suuntaan 500 haulla seuraavanlaiset tulokset:



Suoritusajat lyhenivät siis huomattavasti, varsinkin pitkillä poluilla.

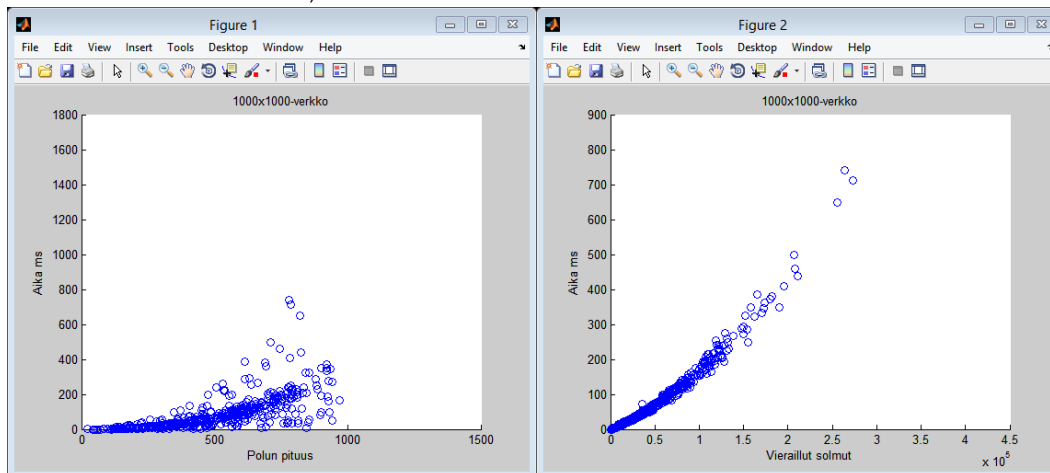
Enemminkin kuin polun pituus suoritusajan kanssa on yhteydessä vierailtujen solmujen lukumäärä:



Yhteys on varsin lineaarisen näköinen.

Vastaavat tulokset verkolle, jossa saa liikkua myös sivuttain (parametrit

1000:1000:1:1:2:0):

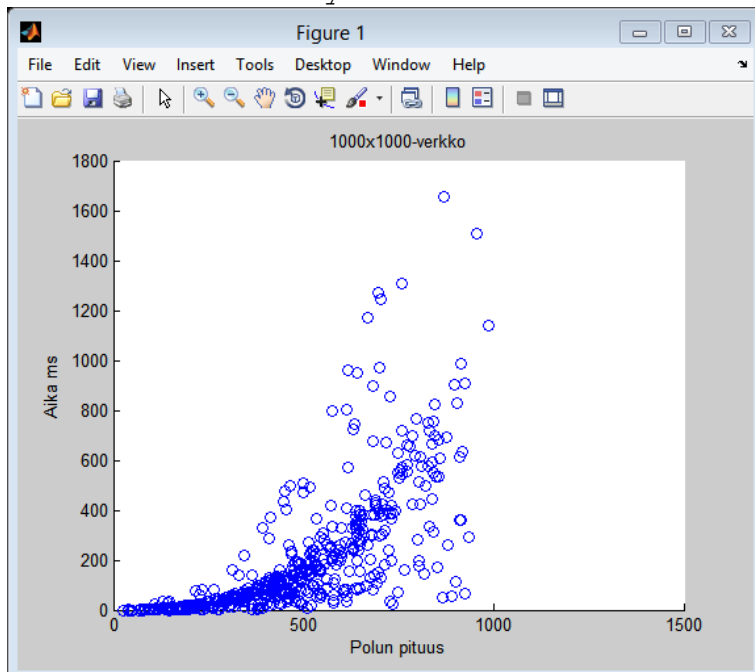


Nyt riippuvuus vierailtujen solmujen määrästä on suurempi.

4. Javan valmiit tietorakenteet

Kun testejä ajetaan AStar-olion etsiReitti-metodilla, on käytössä Javan omat tietorakenteet. Tätä voi testata vain vaihtamalla metodikutsun Gui-luokan aliluokan Tulokset metodissa testaa()-olevan rivin polku = aStar.etsiReittiOma(a, b) kohdalle. Metodi käyttää käsittelyjärjestyksessä PriorityQueue:a ja käsitellyissä solmuissa HashSet:iä.

1000x1000 verkosta, jossa voi liikkua sivuttain, 500 näytteen otos Javan tietorakenteita käyttäen:



Omat tietorakenteeni siis selvästi toimivat paremmin kuin Javan vastaavat.

Poikkeustilanteissa PriorityQueue on kuitenkin nopeampi: jos heuristinen arvio on ylisuuri, oma toteutukseni on lisäyksissä aikavaativuudeltaan $O(n)$

verrattuna PriorityQueue:n $O(\log n)$ -aikavaatimukseen.