Testausdokumentaatio

JUnit-testaus

Ohjelman JUnit-testaus sisältää ohjelman perustoimintojen oikean toiminnan tarkistamisen (ovet toimivat kuten pitääkin, tietorakenteet toimivat odotetusti, ne algoritmit joiden kuuluu löytää lyhin reitti todellakin löytävät lyhyimmän reitin jne.)

Algoritmien testaus

JUnit-testeillä testasin, että A*, Dijkstran algoritmi ja leveyshaku löytävät aina lyhyimmän reitin tilanteesta riippumatta, että ne reagoivat oikein tilanteessa, jossa reittiä ei löydy sekä sen, että kaikki käyttäytyvät järkevästi sokkelossa jossa on ovia. Testien pohjana käytin visualisointi päällä ajettuja yksinkertaisia sokkeloita, joista laskin manuaalisesti todellisen lyhimmän reitin ja optimaalisen käytöksen ovien tullessa vastaan. Testeistä on sekä versiot jotka sallivat vinottaiset liikkeet että versiot jotka eivät salli. Syvyyshaulle on koodissa läpimenevät testit, mutta niiden odottamat tulokset eivät perustu juuri mihinkään, sillä en pitänyt syvyyshakua erityisen tärkeänä.

A*:lle ja Dijkstralle on lisäksi erilliset testit, joissa testataan, ymmärtävätkö ne kaarten painot oikein (suo- ja jääruudut).

Tietorakenteiden testaus

Jokaiselta tietorakenteelta testasin JUnit-testeillä julkisten metodien oikean toiminnan, sekä vertasin niiden nopeutta vastaaviin Javan omiin tietorakenteisiin.

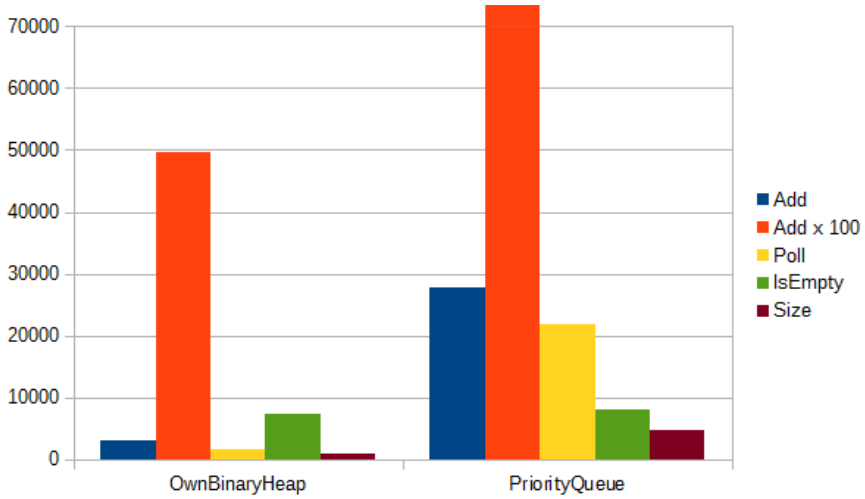
Tietorakenteiden nopeus verrattuna Javan vastaaviin

Kaikissa testeissä saadut ajat heittelivät noin 30% kumpaankin suuntaan, joten saadut nanosekuntiajat eivät ole täysin absoluuttisia. Nämä luvut edustavat silmämääräisesti tarkasteltuna keskimääräistä ajokertaa. Heittelystä huolimatta omien ja valmiiden tietorakenteiden käyttämät ajat olivat joka ajokerralla samoissa suhteissa toisiinsa, joten ainakin graafisista esityksistä saa jonkinlaista kuvaa ajoaikojen suhteista. Joissain tapauksissa tiettyihin operaatioihin kului huomattavasti pidempi aika kuin muihin, ja näiden palkkien on annettu jatkua kuvan ulkopuolelle matalampien palkkien hahmottamisen helpottamiseksi.

OwnBinaryHeap

Testissä täytin OwnBinaryHeapin ja PriorityQueuen ensin yksittäisellä ja sitten 100 solmulla, joilla on arvot välillä 0-299. Kumpikin tietorakenne käytti samaa komparaattoria solmujen vertailuun (NodeComparator). Valitsin solmumäärän 100 nimenomaan siksi, että tuolla arvolla molemmat tietorakenteet ylittävät oletuskapasiteettinsa (11) ja joutuvat kasvattamaan kokoaan. 11 on kuulemma jostain syystä optimaalinen oletuskapasiteetti jos etukäteen ei ole tietoa lisättävien alkioiden määrästä, en vain saanut selville, miksi.

Tietorakenne	OwnBinaryHeap	PriorityQueue
Add	2994ns	27797ns
Add x 100	49608ns	526015ns
Poll	1711ns	21810ns
IsEmpty	7271ns	8125ns
Size	855ns	4705ns

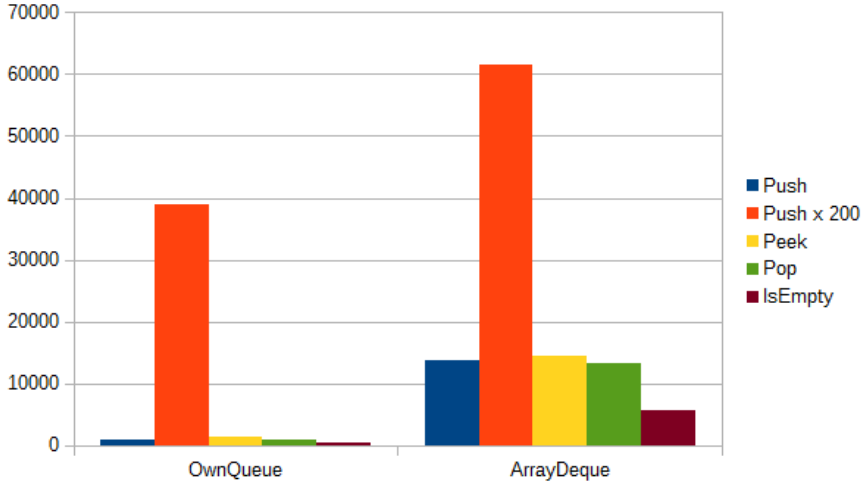


OwnBinaryHeap osoittautui PriorityQueuea huomattavasti nopeammaksi. Sen toimintaa kuitenkin rajoittaa pelkkien Node-tyyppisten olioiden salliminen, eikä se myöskään sisällä kaikkia PriorityQueuen ominaisuuksia, kuten `contains()` ja `remove()`.

OwnQueue

Testissä täytin OwnQueuen ja ArrayDequen ensin yhdellä ja sitten 200 solmulla ja otin eri operaatioihin käytetyt ajat ylös. Kummankin tietorakenteen oletuskapasiteetti on 16, joten kumpikin joutui laajentamaan taulukkoaan.

Tietorakenne	OwnQueue	ArrayDeque
Push	855ns	13685ns
Push x 200	38916ns	61582ns
Peek	1283ns	14540ns
Pop	855ns	13258ns
IsEmpty	428ns	5560ns



Jononi on huomattavasti nopeampi kuin Javan ArrayDeque. Toiminnallisuudeltaan se on kuitenkin sekä yksinkertaisempi että rajoittuneempi, kuten OwnBinaryHeapkin; se hyväksyy vain Node-tyyppisiä olioita.

OwnArrayList

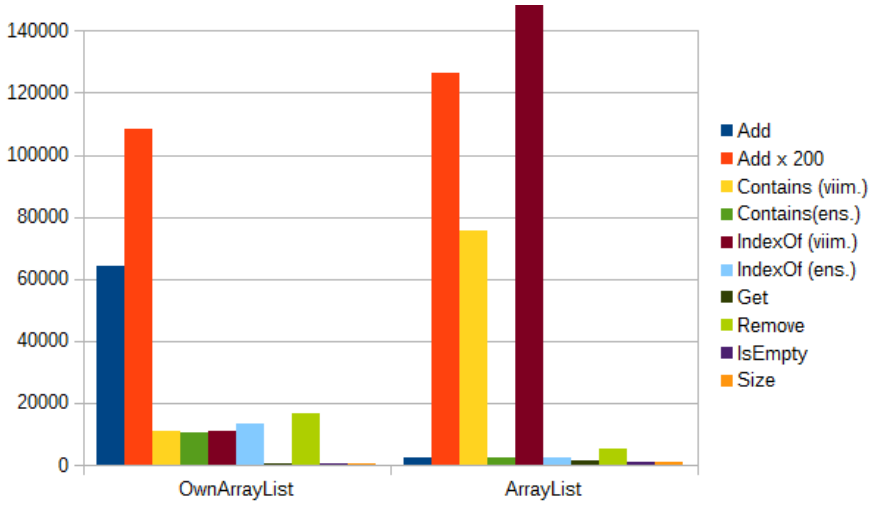
OwnArrayList on, toisin kuin muut tietorakenteeni, toiminnallisuudeltaan lähes samaa tasoa Javan oman ArrayListin kanssa, ja hyväksyy kaikentyyppisiä

olioita. Vertasin sitä ArrayListiin sekä solmuilla, liukuluvuilla että kokonaisluvuilla. ArrayListin aloituskapasiteetti on 10, joten annoin OwnArrayListille saman. Kaikissa tapauksessa käytin ensin yhtä ja sitten 200 lisättyä alkiota (välissä tyhjennys), jolloin lista joutui kasvattamaan itseään useaan kertaan. Kaikissa contains-testeissä etsitty alkio on taulukossa, joko ensimmäisenä tai viimeisenä. IndexOf-testeissä testataan ensimmäisen ja viimeisen alkion indeksin löytämistä. Get- ja Remove -testeissä on haettu ja poistettu 101. alkio.

Liukuluvuilla

Täytin kummankin tietorakenteen 200 satunnaisella liukuluvulla väliltä 0-1.

Tietorakenne	OwnArrayList	ArrayList
Add	64148ns	2138ns
Add x 200	108197ns	126158ns
Contains (viimeinen alkio)	11119ns	75695ns
Contains (ensimmäinen alkio)	10264ns	2566ns
IndexOf (viimeinen alkio)	11119ns	1910334ns
IndexOf (ensimmäinen alkio)	13258ns	2566ns
Get	428ns	1283ns
Remove	16678ns	5131ns
IsEmpty	427ns	855ns
Size	428ns	855ns



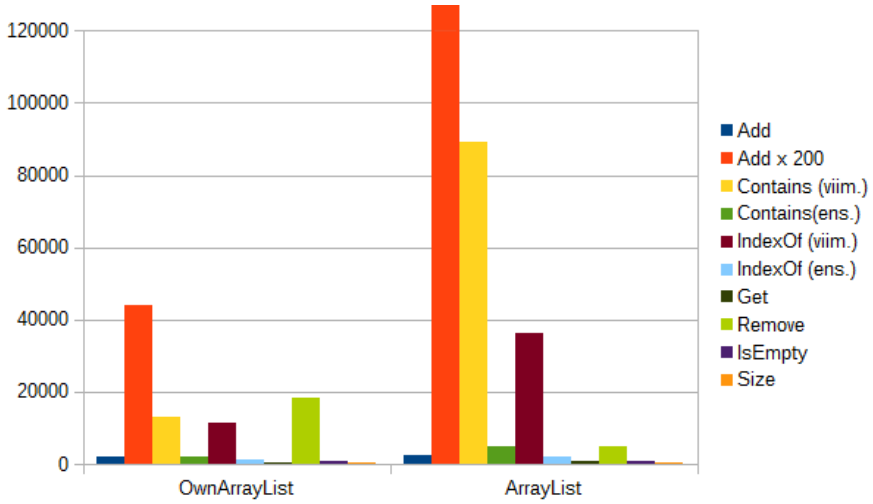
Liukuluvuilla Javan oma ArrayList toimi paljon nopeammin kuin OwnArrayList kaikissa tilanteissa, paitsi `isEmpty()` - ja `size()` -kyselyissä. ArrayListin `size()` on monimutkaisempi, kun taas kummassakin `isEmpty()` perustuu suoraan siihen, onko `size()` :n arvo `0` .

Kokonaisluvuilla

Täytin kummankin tietorakenteen 200 järjestyksessä olevalla kokonaisluvulla.

Tietorakenne	OwnArrayList	ArrayList

Add	2138ns	2566ns
Add x 200	44049ns	649180ns
Contains (viimeinen alkio)	12829ns	89380ns
Contains (ensimmäinen alkio)	2138ns	4704ns
IndexOf (viimeinen alkio)	11546ns	36350ns
IndexOf (ensimmäinen alkio)	1283ns	2138ns
Get	428ns	855ns
Remove	18389ns	4705ns
IsEmpty	855ns	855ns
Size	427ns	428ns



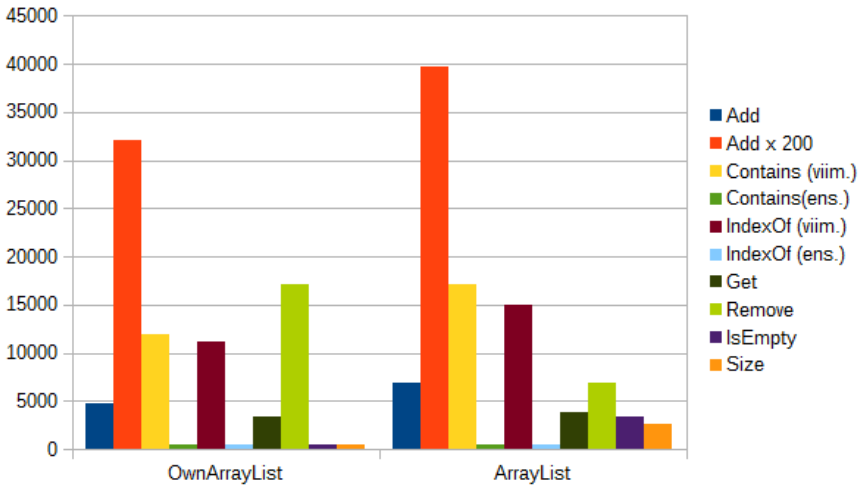
Kokonaisluvuilla OwnArrayList oli jälleen hitaampi kuin ArrayList, mutta ei läheskään yhtä pahasti kuin liukuluvuilla. Yksittäisessä lisäyksessä päästiin lähes samaan aikaan, ja `contains()` ja `indexOf()` jäivät vain puolta hitaammiksi.

Solmuilla

Täytin kummankin listan 200 solmulla.

Tietorakenne	OwnArrayList	ArrayList
Add	4705ns	6843ns
Add x 200	32074ns	39772ns
Contains (viimeinen alkio)	11975ns	17106ns
Contains (ensimmäinen alkio)	428ns	427ns
IndexOf (viimeinen alkio)	11119ns	14968ns
IndexOf (ensimmäinen alkio)	428ns	428ns

Get	3422ns	3849ns
Remove	17106ns	6842ns
IsEmpty	427ns	3421ns
Size	428ns	2566ns



Monimutkaisemmalla oliolla, tässä tapauksessa Nodella, OwnArrayList pääsi nopeudeltaan vieläkin lähemmäksi ArrayListia, ja aikaerot jäivät kaikissa operaatioissa `remove()` :a lukuunottamatta suhteessa paljon pienemmiksi kuin millään muilla alkiotyypeillä. `remove()` on siis ArrayListissa selvästi tehokkaampi kuin OwnArrayListissa. Kummassakin tietorakenteessa metodi kopioi vanhan taulukon uuteen taulukkoon lukuunottamatta pois pudotettavaa alkiota, mutta ArrayList käyttää `System.arraycopy` -metodia, joka on huomattavasti nopeampi kuin manuaalinen taulukon kopiointi.

Algoritmien manuaalinen testaus

Testauksessa ajetaan kaikki hakualgoritmit samanlaisessa verkossa. Nämä testit on ajettu käsin tulosten ohessa mainittujen parametrien avulla.

Testaus keskittyy algoritmien nopeuden ja käytetyn askelmäärän mittaamiseen. Yleiseen käyttöön optimaalisin algoritmi on nopea ja löytää parhaan reitin. Testeissä käytetään 105x105 ruudun sokkeloa (eli sen kokoista, että siihen kykeni vielä järkevästi manuaalisesti sijoittamaan erikoisruudut).

Tulos on keskiarvo kolmen ajamiskerran tuloksista, pyöristettynä millisekunteihin.

Suoritettavat testit

- Algoritmien toiminta yksinkertaisessa sokkelossa (tässä käytetään 401x401-sokkeloa)
    - Sokkelotyypit: labyrintti ja avoin alue, jossa esteitä
    - Labyrinttityyppisen sokkelon läpi kulkee vain yksi oikea reitti
- Algoritmien toiminta sokkelossa, joka muuttuu
- Algoritmien toiminta kohderuudun liikkuessa
- Suurikokoinen yhdistelmäsokkelo, jossa esiintyy kaikkia edellämainittuja sekä lisäksi solmuja, joiden kaarilla on eri painot ("suo" ja "jää")
    - Eripainoisten kaarien sokkeloita ei näissä testeissä ole testattu erikseen, sillä ne eivät vaikuta merkittävästi ohjelman suoritusnopeuteen, ja muissa tilanteissa hyvä leveyshaku ei osaa ottaa painoja huomioon. Niiden toiminta on kuitenkin testattu JUnit-testeissä.

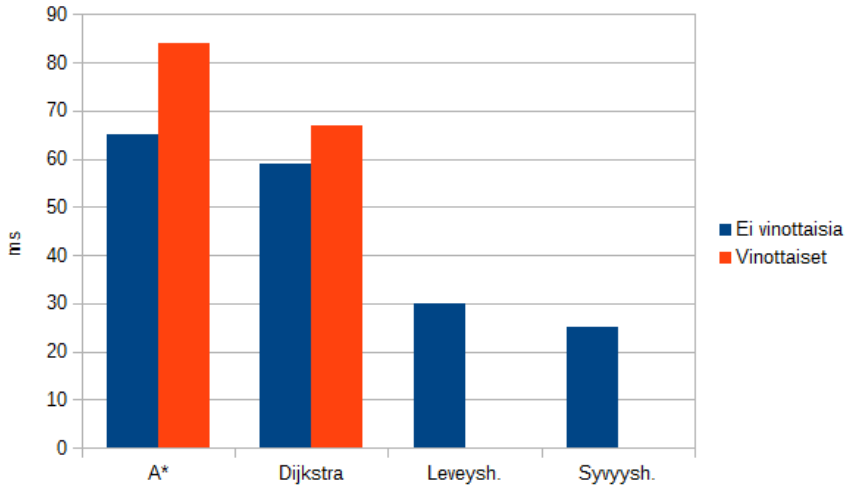
Kaikista testeistä ajetaan versiot, joissa vinottainen liike on sallittu ja kielletty. Vinottaisten liikkeiden tapauksessa käytetään vain A*:ä ja Dijkstran algoritmia, sillä vain ne osaavat ottaa huomioon vinottaisten liikkeiden hinnan.

Tulokset

Yksinkertainen sokkelo

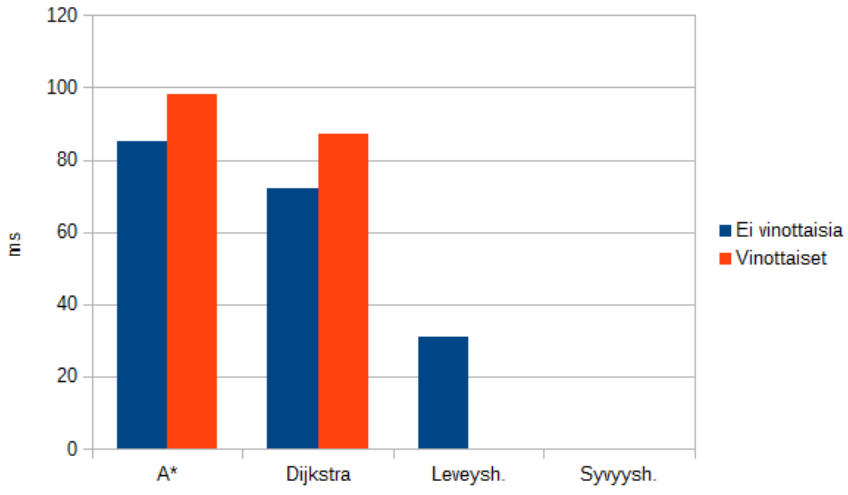
Labyrintti 401x401

Algoritmi	A*	Dijkstra	Leveyshaku	Syvyyshaku	A* (vinottaiset)	Dijkstra (vinottaiset)
Aika	65ms	59ms	30ms	25ms (noin joka toinen kerta stack overflow)	84ms	67ms
Askeleet	2877	2877	2877	7753	2029	2029
Ajoparametrit	`java -jar pathfinding.jar 401x401 false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401 false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401 false Breadth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401 false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401 true A* 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401 true Dijkstra 5 20 10 false false false 0`



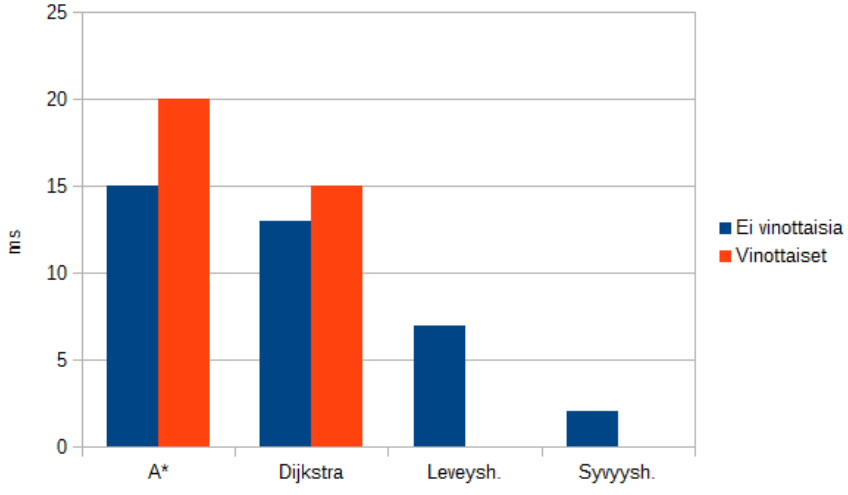
Avoin tila 401x401

Algoritmi	A*	Dijkstra	Leveyshaku	Syvyyshaku	A* (vinottaiset)	Dijkstra (vinottaiset)
Aika	85ms	72ms	31ms	Stack overflow liian suuren verkon vuoksi	98ms	87ms
Askeleet	797	797	797		468	468
Ajoparametrit	`java -jar pathfinding.jar 401x401_open false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401_open false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401_open false Breadth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401_open false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401_open true A* 5 20 10 false false false 0`	`java -jar pathfinding.jar 401x401_open true Dijkstra 5 20 10 false false false 0`



Labyrintti 105x105

Algoritmi	A*	Dijkstra	Leveyshaku	Syvyyshaku	A* (vinottaiset)	Dijkstra (vinottaiset)
Aika	15ms	13ms	7ms	2ms	20ms	15ms
Askeleet	265	265	265	2805	189	189
Ajoparametrit	`java -jar pathfinding.jar t105x105 false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105 false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105 false Breadth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105 false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105 true A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105 true Dijkstra 5 20 10 false false false 0`

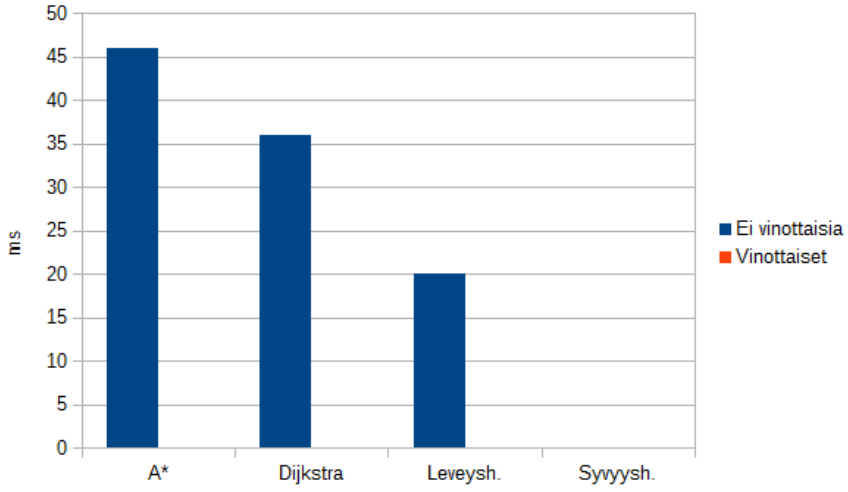


Tuloksista huomataan, että Dijkstran algoritmi toimii testatuissa sokkeloissa hieman AStaria nopeammin. A*, Dijkstran algoritmi ja leveyshaku tuottavat kaikki lyhimmän mahdollisen reitin. Kun kaarten painoja ei tarvitse ottaa huomioon, on leveyshaku algoritmeista paras. Syvyyshakua puolestaan ei voi käyttää varsinaisesti polunetsintään, sillä sen tuottama reitti on käyttökelvottoman pitkä. Syvyyshaku on kuitenkin merkittävästi nopeampi kuin muut, jos halutaan vain tarkistaa reitin olemassaolo.

Muuttuva sokkelo (ovet)

Algoritmi	A*	Dijkstra	Leveyshaku

Aika	46ms	36ms	20ms
Askeleet	302	302	302
Laskentakerrat	16	16	16
Ajoparametrit	`java -jar pathfinding.jar t105x105_doors false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_doors false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_doors false Breadth-first 5 20 10 false false false 0`



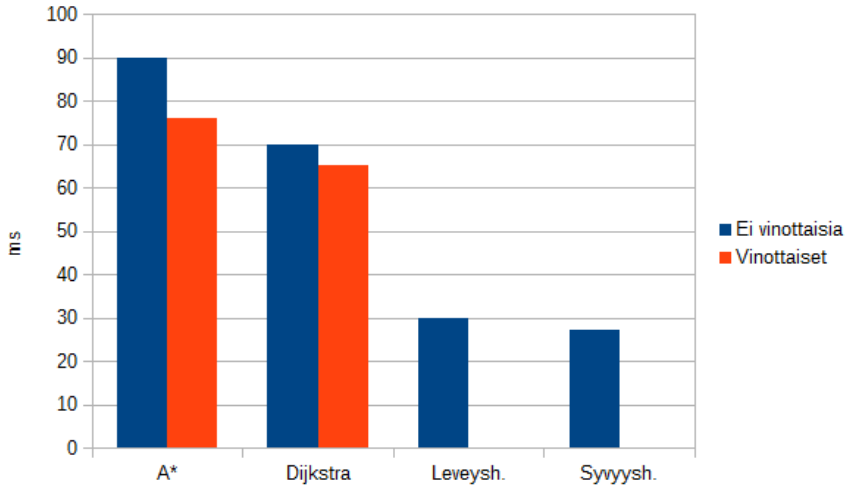
Syvyyshaku jäi loputtomaan looppiin. Etsijä ei ehtinyt ovelle ennen sen sulkeutumista ja se lähti etsimään uutta reittiä. Myös uusi reitti sulkeutui ennen kuin etsijä ehti ovelle, ja vanha reitti aukesi uudestaan. Tämän jälkeen etsijä yritti uudestaan vanhaa reittiä. Ajoparametrit: `java -jar pathfinding.jar t105x105_doors false Depth-first 5 20 10 false false false 0`.

Vinottaiset liikkeet saattoivat ohjelman samanlaiseen looppiin kuin syvyyshaku.

Muuttuvan sokkelon tilanteessa ei tapahtunut mitään erityisen yllättävää. Suoritusajoista kuitenkin huomaa, ettei aikaa kulunut kovin paljoa enempää kuin yksinkertaisessa polunetsinnässä, vaikka laskentakertoja oli 16-kertainen määrä. Syy tähän on siinä, että etsijän lähestyessä kohdettaan tarvittavan uuden polun pituus lyhenee. Pienemmässä sokkelossa reitti on eksponentiaalisesti nopeampi laskea.

Liikkuva kohde

Algoritmi	A*	Dijkstra	Leveyshaku	Syvyyshaku	A* (vinottaiset)	Dijkstra (vinottaiset)
Aika	90ms	70ms	30ms	27ms	76ms	65ms
Askeleet	299	299	299	3047	194	194
Laskentakerrat	60	60	60	67	39	39
Ajoparametrit	`java -jar pathfinding.jar t105x105_moving false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_moving false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_moving false Breadth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_moving false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_moving false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_moving true Dijkstra 5 20 10 false false false 0`



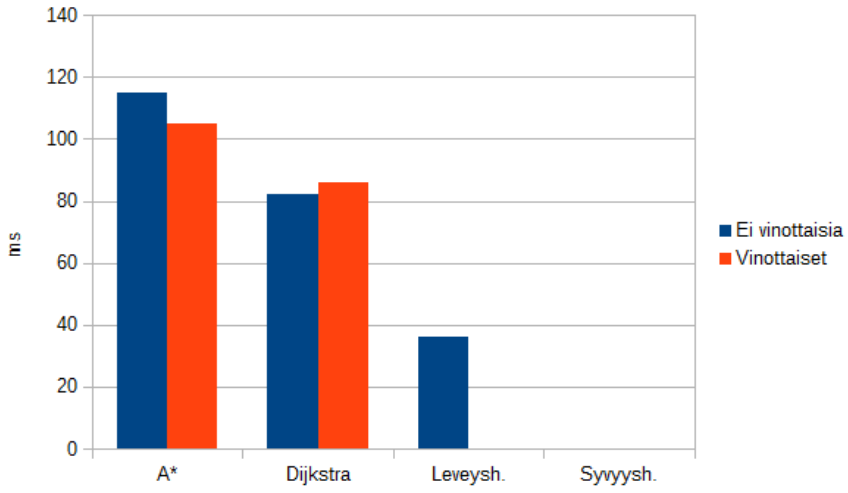
Tulokset ei-vinottaisilla liikkeillä vastaavat suurin piirtein edellisten testien perusteella odotettuja. Syvyyshaku teki kuitenkin hassun tempun liikkuvan kohteen kanssa; se käytti lähes saman ajan kuin leveyshaku, vaikka aiemmin se on ollut merkittävästi nopeampi. Huomattavasti huonomman reitin löytävä syvyyshaku joutui tekemään enemmän laskentakertoja, sillä etsijä ei pysynyt läheskään yhtä hyvin maalin perässä. Todennäköisesti syvyyshaulla olisi kuulunut kestää vielä paljon pidempään, mutta maaliruudulta loppuivat siirrot kesken 67 siirron jälkeen.

Vinottaisten liikkeiden tapauksessa huomataan päinvastainen ilmiö; vinottaisten liikkeiden avulla reitistä saadaan paljon lyhyempi, jolloin maaliruutu ei ehdi liikkua yhtä paljon ennen kuin se saadaan kiinni ja laskentakertoja tarvitaan paljon vähemmän. Tällöin ohjelman suoritus on huomattavasti nopeampi.

Käytännön peliympäristön simulaatio

Testataan siis sokkeloa, jossa esiintyvät kaikkien aiempien testien erityispiirteet yhtä aikaa, sekä ruutuja, joilla on vaihtelevat painot. Tämä tilanne muistuttaa eniten polunetsintäalgoritmin käytännön sovellutusta videopelissä.

Algoritmi	A*	Dijkstra	Leveyshaku	Syvyyshaku	A* (vinottaiset)	Dijkstra (vinottaiset)
Aika	115ms	82ms	36ms	Loputon looppi kuten aiemmin	105ms	86ms
Askeleet	333	333	352		267	267
Laskentakerrat	71	71	72		58	58
Epäonnistuneet laskentakerrat	5	5	5		5	5
Ajoparametrit	`java -jar pathfinding.jar t105x105_all false A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_all false Dijkstra 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_all false Breadth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_all false Depth-first 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_all true A* 5 20 10 false false false 0`	`java -jar pathfinding.jar t105x105_all true Dijkstra 5 20 10 false false false 0`



Näissäkin testeissä leveyshaku osoittautui nopeimmaksi, Dijkstra toiseksi nopeimmaksi ja A *hitaimmaksi. Leveyshaku kuitenkin jostain syystä päätyi kulkemaan pidempää reittiä kuin kaksi muuta. Tämä johtuu siitä, että se ei huomioi solmujen painoja, kun taas A* ja Dijkstra pyrkivät kulkemaan nopeampia jääruutuja pitkin - ja sinänsä siis kulkivat siirtomäärällisesti pidempää reittiä, vaikka reitin paino oli matalampi. Leveyshaku posotti suoraan suoruuduista läpi.

Epäonnistuneet laskentakerrat seurasivat viiden siirron tauosta, jonka aikana yksikään reitti ei ollut etsijälle auki.

Vinottaisilla liikkeillä kului suurinpiirtein sama aika kuin ei-vinottaisilla; lyhyemmästä reitistä seuraavat vähemmät laskentakerrat kompensoivat solmujen naapurien suurempaa määrää.