Testausdokumentti

JUnit-testeillä testasin käyttöliittymä- ja rajapintaluokkia lukuunottamatta kaikkien luokkien perustoimintaa. Testien kattavuuden arvioinnissa auttoi pit-mutaatiotesti, joka antoi testien rivikattavuudeksi 98% ja mutaatiokattavuudeksi 93%.

Algoritmien testaus

Luokkaa SolvabilityDeterminer testasin etsimällä sellaisia 15- ja 8-pelejä, joista tiesi, ovatko ne mahdollisia tai mahdottomia ratkaista. Valitsin kolme ratkaisematonta peliä ja kolme sellaista, joihin on ratkaisu, ja katsoin osaako algoritmi määrittää niiiden ratkaistavuuden oikein.

Heuristisista funktioista testasin pääasiassa sen, antavatko ne sellaisen arvion jäljellä olevien siirtojen määrästä kuten niiden pitäisi. Valitsin muutaman satunnaisen 15- ja 8-pelin, laskin millaisen arvion pitäisi olla ja vertaisin sitä algoritmin antamaan. Lisäksi testasin, että kunkin algoritmin update()-funktio toimii eli että algoritmi antaa uuden arvion oikein yhden siirron jälkeen.

IDA*-algoritmista testasin, että algoritmi antaa siirtojen määrän oikein etsimällä pelejä, joiden minimisiirtojen määrä oli tiedossa. Haastavampaa oli tutkia, onko algoritmin antamat siirrot oikeita. Testasin sitä valitsemalla yksinkertaisen pelin, josta oikeat siirrot oli helppo nähdä. Kokeilin myös päätyykö peli tavoitetilaan, jos pelin laattoja siirretään algoritmin antamien siirtojen mukaisesti.

Tietorakenteiden testaus

Testasin List-tietorakenteen toimintaa. Testasin, että alkioiden lisääminen listaan onnistuu, listan tyhjetäminen onnistuu ja että lista antaa oikean alkion indeksin perusteella. Lisäksi testasin, toimiiko contains-metodi eli palauttaako metodi true, jos lista sisältää parametrina annetun olion.

Suorituskykytestaus

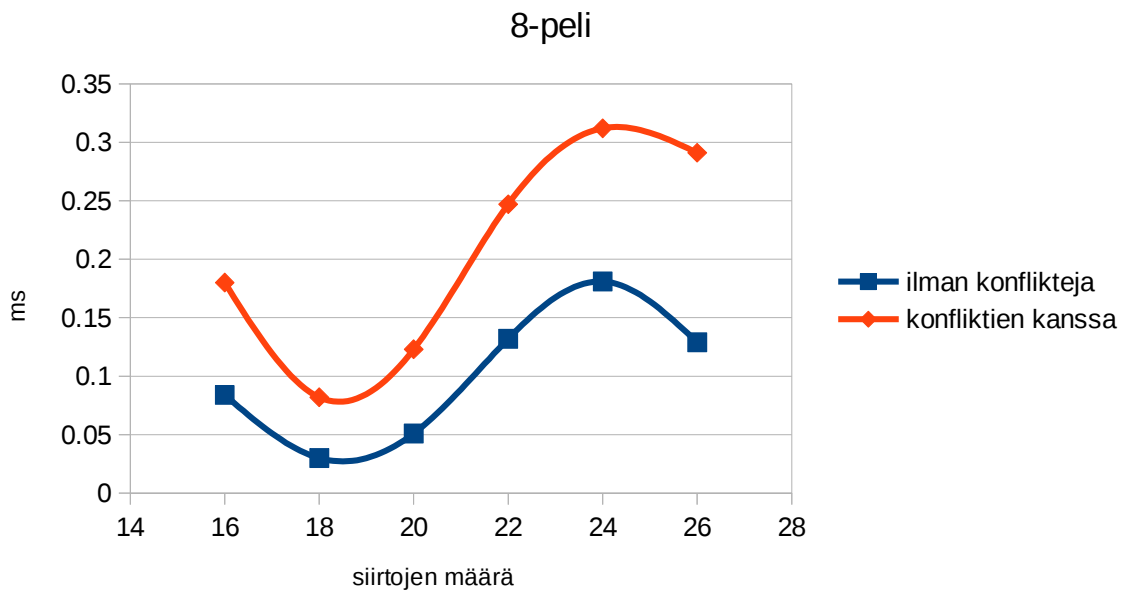
Testasin luokkien IDAStar, SolvabilityDeterminer ja List suorituskykyä. Testit löytyvät testihakemiston solver/logic/efficiencyTests hakemistosta.

IDAStar

8-peli

Tein IDAStarin suorituskykytestausta ensin pienillä 3 x 3 kokoisilla peleillä. Hain ohjelmallani kuusi esimerkkipeliä, joiden ratkaisemiseen vaadittavien siirtojen määrät olivat 16, 18, 20, 22, 24 ja 26. Testasin kauan IDAStarilla kesti kunkin pelin ratkaisemiseen. Testasin tätä kummallakin heurestiikalla, Manhattan-etäisyys-heurestiikalla ja Manhattan-etäisyys lineaaristen konfliktien kanssa-heurestiikalla. Kaikki testit toistettiin 1000 kertaa, joista laskettiin keskiarvo.

Siirtojen määrä	Ilman konflikteja (ms)	Konfliktien kanssa (ms)
16	0.084	0.18
18	0.03	0.082
20	0.051	0.123
22	0.132	0.247
24	0.181	0.312
26	0.129	0.291

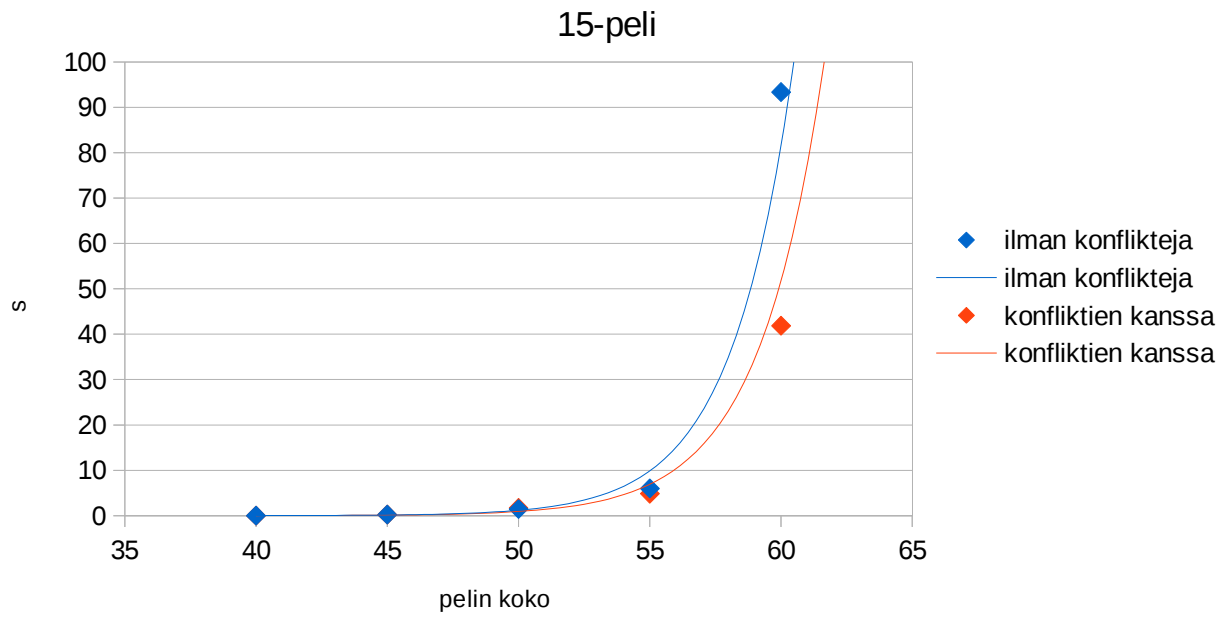


Algoritmi vaikuttaa pystyvän ratkaista kaikki 8-pelit nopeassa ajassa. Hieman yllättäen pelkkä Manhattan-etäisyys-heurestiikka toimi nopeammin kuin se, jossa oli laskettu lineaariset konfliktit mukaan. Vaikka konfliktien mukaan ottaminen parantaa funktion antaman arvion tarkkuutta, niiden laskemiseen kuluva aika on ilmeisesti sen verran suuri, että se ei hyödytä ainakaan näin pienillä peleillä.

15-peli

Tein samantyyppisiä testejä kuin 8-pelille myös 15-pelille. Valitsin viisi 4 x 4 kokoista esimerkkipeliä, joiden ratkaisemiseen kuluvien siirtojen määrät olivat 40, 45, 50, 55 ja 60. Testasin ratkaisuun kuluvaa aikaa kummallakin heurestiikalla. Koska ratkaisemiseen kuluva aika oli kohtalaisen suuri, kaikki testit toistettiin vain 10 kertaa, joista laskettiin keskiarvo.

Siirtojen määrä	Aika (s) ilman konflikteja	Aika(s) konfliktien kanssa
40	0.012	0.008
45	0.249	0.247
50	1.465	1.715
55	6.006	4.855
60	93.351	41.85



15-pelin ratkaisemiseen kuluva aika lähti hyvin jyrkästi nousemaan pelin koon kasvaessa. 60 siirtoa vaativan pelin ratkaisemiseen kuluva aika oli jo yli minuutin Manhattan-etäisyys-heurestiikalla. Paljon tuota isompia pelejä ohjelmallani ei luultavasti voi ratkaista järkevässä ajassa. 15-pelin ratkaisussa heurestiikka, jossa oli laskettu mukaan myös lineaariset konfliktit, toimi nopeammin varsinkin siirtojen määrän kasvaessa. 60 siirron peli ratkesi sen avulla jo puolet nopeammin.

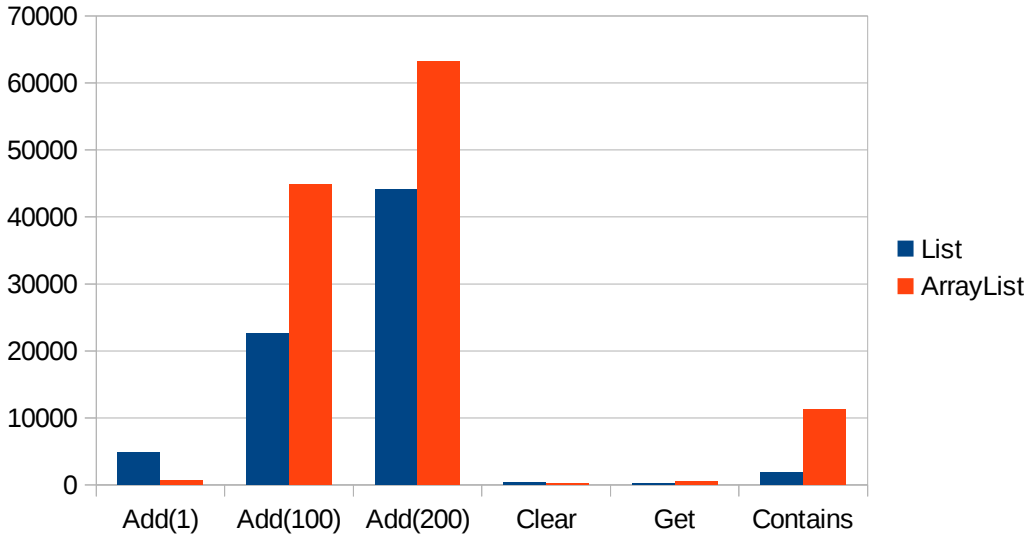
Ratkaistavuuden selvittämisen testaus

Testasin kauan luokaltani SolvabilityDeterminer kesti satunnaisten 15- ja 8-pelien ratkaisemiseen. Toistin testit 10 000 kertaa ja laskin keskiarvon. Satunnaisen 8-pelin ratkaistavuuden selvittämiseen kului keskimääräisesti 0.00275 ms ja 15-pelin 0.00367 ms. Algoritmi toimii nopeasti pienillä taulukoilla.

Tietorakenteiden vertaus Javan vastaaviin

Vertasin tekemääni List-tietorakennetta Javan valmiiseen ArrayList-tietorakenteeseen. Tein testit kokonaisluvuilla. Testasin yhden, 100 ja 200 järjestuksessä olevan luvun lisäystä ja lisäksi niiden poistoa. Testasin myös get- ja contains-metodeja, joissa hain satunnaista listoilta löytyvää lukua. Kaikki testit toistettiin 100 kertaa ja laskettiin niistä keskiarvot.

	List	ArrayList
Add(1)	4880ns	586ns
Add(100)	22579ns	44826ns
Add(200)	44071ns	63167ns
Clear	360ns	175ns
Get	253ns	523ns
Contains	1898ns	11233ns



Yksinkertainen toteutus lista-tietorakenteesta näyttää pärjäävän nopeudessa Javan valmiille ainakin näiden metodien osalta ja näin pienillä lukumäärillä. List-tietorakenteestani puuttuu kuitenkin aika paljon metodeja, mitä ArrayListillä on, kuten remove(), koska en tarvinnut niitä ohjelmassani.