TOTEUTUSDOKUMENTTI 14.6.2013

Fifteen Puzzle – tietorakenteiden harjoitustyö

@Tero Mäntylä

Mitä tein

Toteutin Fifteen Puzzle -palapeliin erilaisia ratkaisu algoritmeja, jotka etsivät satunnaisesti sekoitetulle palapelille pienimmän siirtosarjan, jolla sen voi ratkaista.
Viimeisimpänä tein A*-algoritmin pohjalta toimivan toteutuksen. Sitä varten koodasin itse minimikeon ja hajautustaulun. Hajautustaulussa tosin käytin Javan omaa Arrays-luokan deepHashCode()-metodia.
Tärkein työn kohde oli IDA*-algoritmin ideaan perustuva ratkaisija. Algoritmin nopean toiminnan kannalta tärkeässä osassa oli hyvän heuristiikan löytäminen ja sen koodaaminen toimimaan tehokkaati. Tätä osaa varten ei tarvinnut tehdä mitään tietorakenteet ja algoritmit kurssilla esiteltyä tietorakennetta.
Kurssin alussa iso osa työstä meni itse palapelin toiminnallisuuksien toteuttamiseen. Ne on tehty Puzzle-luokkaan.

Seuravaksi tarkastellaan toteutettuja tietorakenteita yksi kerrallaan.

Puzzle-luokka ja palapelin sekoittaja

Tämä tietorakenne muistaa viimeisimmän tilanteen - palojen paikat palapelissä erityisesti tyhjän paikan sijainnin ja viimeisimmän tehdyn siirron. Luokka tarjoaa myös toiminnalisuuden palojen hyväksyttäviin siirtoihin sekä palapelin sekoittamiseen satunnaiseen järjestykseen.
Palojen paikkatieto on talennettu kaksiulotteiseen byte-muuttuja tauluun. Koska palat pystytään kuvaamaan 16 numerolla, voisi tämän tiedon pakata 4 bittiin. Ja koko palapeli mahtuisi yhteen 64 bittiseen long-muuttujaan. Päätin kuitenkin yksinkertaisuuden vuoksi käyttää kuuttatoista 8 bittistä byte-muuttujaa, jotka vievät tilaa 128 bittiä. Eli muistitilan haaskaus on 100 prosentia. Muut tietorakenteen luokkamuuttujat ottavat tilaa 146 bittiä. Tällä ei IDA* toteutuksessa ole merkitystä, mutta A* kyllä tästä kärsii muisti intensiivisenä algoritmina.
Satunnaisen sekoituksen aikaan saamiseksi tein toteutuksen, jossa paloja vain siirrellään tuhat kertaa satunnaisella tavalla. Tämän sekoituksen heikkous on eri permutaatioiden saamisen eri suuret todennäköisyydet. Idean asteelle jäi toisen – paremman sekoittajan teko. Aika kun loppui kesken. Tässä sekoittaja arpoo palojen paikan palapelissä. Tämä tehdään vaihtamalla ensin viimeiselle paikalle jokin palapelin pala tai tyhjä. Seuraavaksi tehdään sama toiseksi viimeiselle paikalle jäljelle jääneille palapelin osille. Jatketaan kunnes kaikkille paikoille on arvottu uusi palapelin osa. Lopuksi täytyy vielä tarkistaa että permutaatio on ratkaistavissa oleva. Vain puolet kaikista permutaatioista on mahdollista ratkaista.

Minimikeko – MyMinHeap

Minimikeon toteutus on varsin kevyt. Siihen tuli vain toiminnallisuudet, joita A* tarvitsi eli kekoon lisäys ja pienimmän poisto pinosta. Algoritmi on suora kopio tietorakenteiden kurssi monisteesta. Ainoa erikoisuus on keon koon loppuessa siirretään keon viimeisen jäsenen paikaa osoittava muutuja sadas osan keon koosta taaksepäin. Näin keko ei koskaan käy ”pieneksi”. A* ei tässä toteutuksessa tarvitse niitä keon huonoimpia tapauksia lyhimmän siirtosarjan löytämiseen.

Hajautustaulu – MyHashSet
Kuten edellä niin tässäkin hajautustauluun on luotu vain A*:n tarvitsemat ominaisuudet eli hajautustauluun lisäys sekä tarkistus alkion sisältymisestä hajautustauluun. Hajautusfunktiona käytetään Javan omaa Arrays-luokan deepHashCode()-metodia. Jos törmäyksiä tapatuu niin hajautettavat alkio linkitetään listaksi. Hajautustaulun kooksi tuli kokeilujen jälkeen alkuluku 1500007, joka on mahdollisimman kaukana kakkosen potensseista 2^{20} ja 2^{21} .

A*-algoritmi
Tämä algoritmi on toteutettu luovasti seuraten tietorakenteet kurssin luentokalvoja. Suoraan kurssimateriaalin mukaan toteuttaminen ei ole mahdollista, koska pelkästään eri palapelin eri permutaatioiden muodosma verkko sisältää aivan liian paljon solmuja.
Oma algoritmini poimii aina keosta pienimän ”hinnan” sisältävän solmun. Hinta koostuu siirtojen määrästä lähtösolmuun ja heuristiikkan antamasta etäisyys arviosta ratkaisuun. Kun tämä solmu on otettu keosta lisätään se vierailtujen solmujen joukkoon, joka käytetään varmistamaan että emme mene enää uudestaan samaan solmuun. Tässä käytetään omaa hajautustaulua. Keosta poimitulle solmulle haetaan mahdolliset siirrot, jotka sitten laitetaan kekoon uuden hinnan kansa. Tätä jatketaan kunnes keosta nostetaan solmu joka on maali eli järjestyksessä.
Ratkaisuun johtava polku löydetään seuraten solmuihin tallennettua tieto edeltäneestä solmusta. Aika ja tila vaativuudet ovat samat kuin mitä A*:lle on tietorakenteiden kurssi monisteessa annettu.

IDA*-algoritmi
Tällä algoritmilla on selkeästi kaksi eri tasoa. Ulommalla tasolla tapahtuu laskentarajan kasvattaminen, jos ratkaisua ei löydy. Sisemmällä tasolla tehdään syvyyssuuntainen haku tätä laskentarajaa hyväksi käyttäen. Hakupuun tuottamattomia haaroja katkotaan heuristiikan avulla, joka arvioi sen hetkisessä tilanteen etäisyyden ratkaisuun. Jos tämän arvion ja jo tehtyjen siirtojen määrä on suurempi kuin laskentaraja niin haara voidaan katkaista. Ratkaisuun ollaan päästy kun etäisyys arvio on nolla.
Laskentarajan kasvatus tehdään kahdella, koska ennenkuin algoritmin suoritus alkaa selvitetään onko ratkaisuun tarvittavien siirtojen määrä parillinen vai pariton.
Etäisyyden arvioitiin on mahdollista käyttä erilaisia toteutuksia, jotka on käyty läpi testaus dokumentissa.
IDA* -algoritmin aikavaativuus on eksponentiaalinen ja tilavaativuus on vakio.

Lähteet

- tietorakenteet ja algoritmit luentokalvot
- http://en.wikipedia.org/wiki/15_puzzle
- http://en.wikipedia.org/wiki/IDA*
- http://kociemba.org/fifteen/fifteensolver.html
- http://juropollo.xe0.ru/stp_wd_translation_en.htm