Ohjelman yleisrakenne

Ei-tira-osion rakenteesta löytyy enemmän tietoa sen omasta dokumentoinnista[1]. Tira-osio on jaettu neljään pakettiin: **ai**, **algorithms**, **datastructure** ja **testing**.

Datastructure sisältää **minimikeon** (MinHeap), reitinhaun ja tekoälyn hyödyntämän **solmun** (Node) sekä yksinkertaisen **listan** (TacList) toteutukset. Algorithms sisältää kaksi abstraktia luokkaa, **reitinhaun** (PathFind) ja sen toiminnallisesti peliin liittävän GameUsagen.

Testingissä on **Performance-luokka**, joka sisältää yllättäen tehokkuustestaukseen liittyviä metodeja, sekä **AltPathFind**, joka on PriorityQueuea minimikeon asemasta hyödyntävä versio reitinhausta.

AI on ohjelman monimutkaisin osa, ja sisältää kaikkiaan kuusi luokkaa. Sen toiminnasta enemmän myöhemmin.

Tietorakenteiden toteutus

Tekniseltä toteutukseltaan sekä solmu, minimikeko, että lista ovat hyvin yksinkertaisia.

Minimikeko on käytännössä suoraan syksyn tietorakenteet ja algoritmit-kurssin luentomonisteesta javalle käännetty. Solmu puolestaan on pelin ruudukkokartalle tarkoitettu, sisältäen tiedot sen omasta sijainnista (x ja y - koordinaatit, x hämäävästi pystyakseli kartalla), ja etäisyydesta lähtösolmuun. Lisäksi solmulla on heapIndex-arvo, jota minimikeon etäisyydenpäivitys hyödyntää, sekä paino (moveCost), jota reitinhaku käyttää kaaripainon asemasta.

Lista on hyvin yksinkertainen dynaaminen taulukko, käytännössä hiukan hitaampi ja paljon rajoittuneempi versio javan ArrayListista.

En tee pseudokoodiesitystä ja siitä täsmällistä *O*-analyysiä listasta tai minimikeosta, koska ne olisivat täysin identtisiä tiran luentomonisteen kanssa. Ne myöskin toimivat käytännössä kuten pitäisi, tästä lisää testausdokumentissa.

Kuten testidokumentista huomaamme, lista toimii hieman hitaammin kuin javan ArrayList, mutta poistamisen aikavaativuus todellakin noin 10-kertaistuu listan koon 10-kertaistuessa. Tilavaativuus on $O(1)$ kaikille operaatioille paitsi jos listan kokoa joudutaan kasvattamaan, jolloin se on $O(n)$. Kasvatus

[1] https://github.com/ohinkkan/tira-labra/tree/master/dokumentointi

toimii samalla ‘tuplaa vanha koko’- periaatteella kuin käsittääkseni ArrayListikin, joten sitä tarvitaan harvoin.

Reitinhaun toteutus

Reitinhaku saa syötteenä kartan, jonka jokaisella ruudulla on tietty paino (moveCost), ja lähtöruudun. Lisäksi voidaan rajoittaa, kuinka suurta osaa kartasta lähtöruudun ympärillä reitinhaku tutkii.
Reitinhaku käyttää Dijkstran algorimia minimikeolla varustettuna. Sille ei kuitenkaan anneta maalisolmua, eikä se palauta polkuja, vain ruudukon johon on laskettu etäisyydet kaikkiin ruutuihin lähtösolmusta. Syynä tähän on se, ettei peli tarvitse muita ominaisuuksia.

Reitinhaun *O*-analyysi

Pseudokoodin esittämisestä olisi tässäkään tuskin hyötyä, samasta syystä kuin aiemmin. Reitinhaku tutkii neliön muotoista ruudukkokarttaa, joten solmujen määrä k on aina kartan ruutujen määrä eli kartan sivu s^2, ja kaarien määrä $4k - 4 * s$ eli noin $4k$, jos kartta on suuri.

Dijksran aikavaativuuden pitäisi olla $O((n+m)\,logn)$ missä n on solmujen ja m kaarien määrä, eli siis tässä tapauksessa $O((5s^2)\,log\,s^2)$. Jos sijoitamme vuorotellen arvot 10, 100 ja 1000 saamme ~2100, ~460000 ja ~69000000 eli suhteet 1 : 219 : 150. Testausdokumentista löytyvästä taulukosta taas näemme, että oman reitinhakuni vastaavat suhteet ovat 1 : 133 : 421, mitkä ovat selvästi ainakin samaa luokkaa kuin odotetut.

Algoritmini käyttää yhtä taulukkoa jossa on n solmua minkä lisäksi minimikeossa on niin ikään pahimmillaan n solmua, eli tilavaativuus on $O(2n) \approx O(n)$ kuten kuuluukin.

Tekoäly

Tekoäly käy siirtoaan miettiessään läpi kaikki yksikkönsä ja pelaa simuloidusti niiden kaikki mahdolliset eri toiminnot ja laskee niille arvot, simuloiden haluattaessa useita peräkkäisiä vuoroja MinMax-logiikkaan perustuen.

Tekoäly-algoritmin karkea pseudokoodi seuraavalla sivulla, mutta palautuksessa on mukana myös ylimääräinen verbaalisempi dokumentti tekoälyn kooditason toiminnasta ja rakenteesta- jota ei ehkä kannata lukea ellei todella kiinnosta...

Nyt miälettömän hiano esitys tekoälyalgoritmin toiminnan ideasta O-analyysillä höystettynä.

Olkoon toimintoja keskimäärin *k* kappaletta per yksikkö, molemmilla pelaajilla *h* yksikköä, ja tekoäly simuloi *n* peräkkäistä vuoroa.

```
pelaaVuorosiSenkinTekoÄly()
   turnsToSimulate = n    // kuinka monta rekursiota tehdään.
   parasToiminto = PSV(tekoäly, vihollinen, tämäKierros) // Pelaa Simuloitu
   parasToiminto.pelaaOikeasti()                                    Vuoro

PSV(Pelaaja pelaaja, Pelaaja vihollinen, Kierros kierros)
   parasToiminto = new toiminto
   parasToiminto.arvo = -∞
   for kaikille pelaajan yksiköille // h kertaa
         for kaikille yksikön mahdollisille toiminnoille // k toimintoa
               if laskeToiminnonArvo(toiminto).arvo > parasToiminto.arvo
                     parasToiminto = toiminto

   return parasToiminto

laskeToiminnonArvo(Toiminto toiminto)
   toiminto.teeSimuloidusti
   toiminto.arvo = moniMutkainenArvoLaskuJuttuJotaEnPseudokoodaa()
   turnsToSimulate - 1
   if turnsToSimulate > 0 // rekursiivinen kutsu
         if (kierros not lopussa)
               toiminto.arvo - PSV(vihollinen, pelaaja, kierros).arvo
         else toiminto.arvo - PSV(vihollinen, pelaaja, uusi kierros).arvo

   turnsToSimulate + 1
   toiminto.undo
   return toiminto
```

Eli aikavaativuus on $O((h*k)^n)$. Käytännössä *k* voi olla varsin suuri, (yleensä kymmeniä, usein satoja) eli useiden peräkkäisten vuorojen simulointi on todella raskasta vaikkei yksiköitä olisikaan kovin paljoa. Tilavaativuus sen sijaan on kohtuullinen, koska undon ansiosta rekursion tarvitsee luoda uusia olioita vain jos pelissä alkaisi kokonaan uusi kierros; muistiin jäävät siis ‘vain’ ylempien kerrosten suorittamattomat komennot, korkeintaan *n* kertaa, koska algoritmi toimii syvyyssuunnassa.

Puutteet ja parannusehdotukset

Tekoäly on onnettoman raskas, ja ValueLogicin laskenta-algoritmit ovat, hmm, approksimatiiviset. Toisaalta tekoäly siltikin pelaa ainakin osapuilleen fiksusti, eli jee. Oli miten oli, jonkinlainen karsintalogiikka ainakin liikkumisruutujen suhteen olisi varmasti paikallaan, eikä alfaBeetakaan ole poissuljettu ajatus.

Tällä hetkellä yksiköt liikkuvat oikeastaan teleporttaamalla, mutta koska peli toteuttaa kumminkin Dijkstran, olisi ruutu ruudulta-liikkuminenkin helpohko toteuttaa.

Itse peli on äärimmäisen kliininen ja peliteknisesti tympeä. Tämä oli tarkoituskin (tai pikemminkin, en tarkoituksella käyttänyt siihen aikaa) koska ulkoasun ja pelitekniikan säätäminen on sellaista jota viitsii tehdä sitten myöhemmin itsekin, ilman deadlineja sun muuta. Luultavasti.

Lähteet: Kurssimateriaali kursseilta ohjelmoinnin perus- ja jatkokurssi, ohjelmistotekniikan menetelmät, tietorakenteet ja algoritmit, ohjelmistotuotanto, johdatus tekoälyyn. Wikipedia (MinMax).