Toteutusdokumentti / Labyrintti Anu N. 12.1.2014

Kuten mainitsin *Dokumentointi*-kansiosta löytyvässä määrittelydokumentissa, valitsin TiRa-harjoitustyöni aiheeksi labyrintin. Päätin lähteä toteuttamaan työssäni ensin A*-hakua reitinhakualgoritmiksi, sekä minimikeko-tietorakennetta A*-haun käyttämäksi prioriteettijonoksi. Tämän jälkeen päätin vielä yrittää Jump Point Search -algoritmin [4] toteuttamista.

1. A*-haku ja hakua tukevat tietorakenteet

A*-hakua varten tarkastelin Helsingin yliopiston *Tietorakenteet ja algoritmit* [1]-kurssin sekä *Johdatus Tekoälyyn* [2] kurssin pseudokoodeja, mutta päädyin kuitenkin lopulta aloittaa toteuttamaan A*-hakua Wikipediasta [3] löytyvään pseudokoodiin pohjaten:

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the
                          // start node
    came_from := the empty map    // The map of navigated nodes

    g_score[start] := 0    // Cost from start along best known path

    // Estimated total cost from start to goal through y
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from, current_node)
    if current_node in came_from
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

Source: http://en.wikipedia.org/wiki/A*_search_algorithm

Artikkelissa mainittu aikavaativuus A*-algoritmille on pahimmillaan eksponentiaalinen ja parhaimmillaan polynominen, kun etsintä-avaruus muodostaa puun, maaleja on yksi ja heuristiikkafunktio h(x) noudattaa seuraavaa ehtoa:

$$| h(x) - h^*(x) | = O(\log h^*(x))$$ lähde: [3]

A* -haku on kuin Dijkstran algoritmi, mutta sen lisäksi, että pidetään kirjaa solmujen etäisyydestä lähtösolmuun, käytetään A* -haussa siis heuristiikkaa arvioimaan lyhintä reittiä solmuista maaliin. Arviota voidaan myös parantaa, jos myöhemmin löytyy lyhyempi reitti, joka kulkee aikaisemmin arvioidun solmun kautta. A* löytää optimaalisen reitin, jos heuristiikka ei koskaan yliarvioi etäisyyttä. Itse käytin Labyrintti-ohjelman toteutuksessa heuristiikkana Manhattan-etäisyyttä, jossa lasketaan yhteen kahden eri solmun x- ja y-koordinaattien erotuksen itseisarvot. Esimerkiksi euklidinen ”linnuntie” -etäisyys olisi ollut toinen vaihtoehto. [1, 2].

A*-haku löytyy Labyrintti-ohjelmassani pakkauksesta ”labyrintti.algot”, luokasta ”Astar”. Kuten mainittu, toteutin omana prioriteettijonon toteutuksena minimikeon. Toteutukseni pohjautuu Tietorakenteet ja algoritmit -kurssin kurssimonisteessa käsiteltyyn maksimikeon pseudokoodiin, jonka pohjalta olen siis muokannut minimikeko-toteutuksen [1]. Oma kekototeutukseni löytyy Labyrintti-ohjelmassa pakkauksesta ”labyrintti.tietorakenteet”, luokasta ”Keko”. Omaan kekototeutukseeni pohjautuvan A*-totetuksen suorituskyvyn ja oikeellisuuden tarkastelemiseksi mahdollistan ”Astar” -luokassa A*-haun (ja siis erityisesti prioriteettijonon) toteuttamisen myös Javan PriorityQueuella.

Yllä kuvatusta pseudokoodista poiketen toteutin polun rakentamisen työssäni itse toteutetun Pino-tietorakenteen avulla, joka löytyy pakkauksesta labyrintti.tietorakenteet, luokasta ”Pino”. Tämäkin toteutus pohjautuu Tietorakenteet ja algoritmit -kurssin materiaaliin [1].

Keskeinen tietorakenne toteutuksen kannalta on myös Solmu-olio (ks. pakkaus ”labyrintti.tietorakenteet” -> ”Solmu”), joka vastaa siis yhtä labyrintin ’ruutua’. Solmu-oliossa pidetään kirjaa mm. sen koordinaateista labyrintissa, matkasta lähtösolmuun, kokonaiskustannuksesta (eli lähtösolmusta solmuun+arvio solmusta maalisolmuun),

edellisestä solmusta lyhimmällä polulla, tiedosta, onko solmussa käyty, sekä siitä, onko solmu estesolmu, johon ei voi kulkea.

2. Jump Point Search

A*-algoritmin lisäksi halusin vielä yrittää toteuttaa Jump Point Search -algoritmia, jonka **Daniel Harabor** ja **Alban Grastien** ovat esittäneet artikkelissa *Online Graph Pruning for Pathfinding on Grid Maps*. Algoritmin ajatuksena on parantaa A*-algoritmin tehokkuutta tarkastelemalla hakuavaruutta vain joidenkin solmujen suhteen, joita Harabor ja Grastien kutsuvat ”jump pointeiksi” eli ”hyppypisteiksi”. Näiden pisteiden väliset solmut ’hypätään yli’ tiettyjä ehtoja noudattaen, mikä parantaa parhaassa tapauksessa A*-algoritmin aikavaativuutta huomattavasti. Idea perustuu symmetristen polkujen läpikäymisen vähentämiseen - jos johonkin solmuun voidaan päästä optimaalisesti myös käsiteltävän solmun kautta, sitä ei tarvitse käsitellä nykyisen solmun naapurina. [4] Valitettavasti en ehdi tässä perehtyä tarkemmin JPS-algoritmin yksityiskohtiin, mutta artikkelin [4] lisäksi hyvä kuvaus löytyy esimerkiksi lähteestä [5].

3. Vertailu

Tarkoitus oli ehtiä työssäni vertailemaan vielä tarkemmin A* -ja JPS-algoritmien suoritusta ja oman kekototeutukseni suorituskykyä Javan PriorityQueueen. Löysin kuitenkin tässä viimeisten työpäivien aikana A*- ja kekototeutuksistani bugeja, jotka ehdin vasta aivan juuri saada korjattua, ja osin nämä jäivät vielä paikallistamattakin. Koska algoritmit eivät toimineet oikein ja toisiaan vastaavasti, eli palauttaneet samaa reittiä, en nähnyt kovin järkeväksi suorittaa vielä vertailuja, koska luvut eivät olisi kertoneet oikeita asioita.

Muutamat tässä viimeisessä hopussa pääohjelman (Main) kautta ajamani testit näyttäisivät kuitenkin siltä, että suunnilleen samoissa suoritusajoissa liikutaan eri algoritmeilla, joskin - kuten mainittu - algoritmien reittihaussa voi tällä hetkellä olla poikkeamia, ja toisaalta labyrintti, jossa testit on suoritettu, on erittäin pieni (7x7).

Tuloksia (ajat ovat nanosekunteina)

Search omalla keolla 1000:
26821000

Search Javan PriorityQueuella 1000:
18376000

Search JPS:lla 1000:
33390000

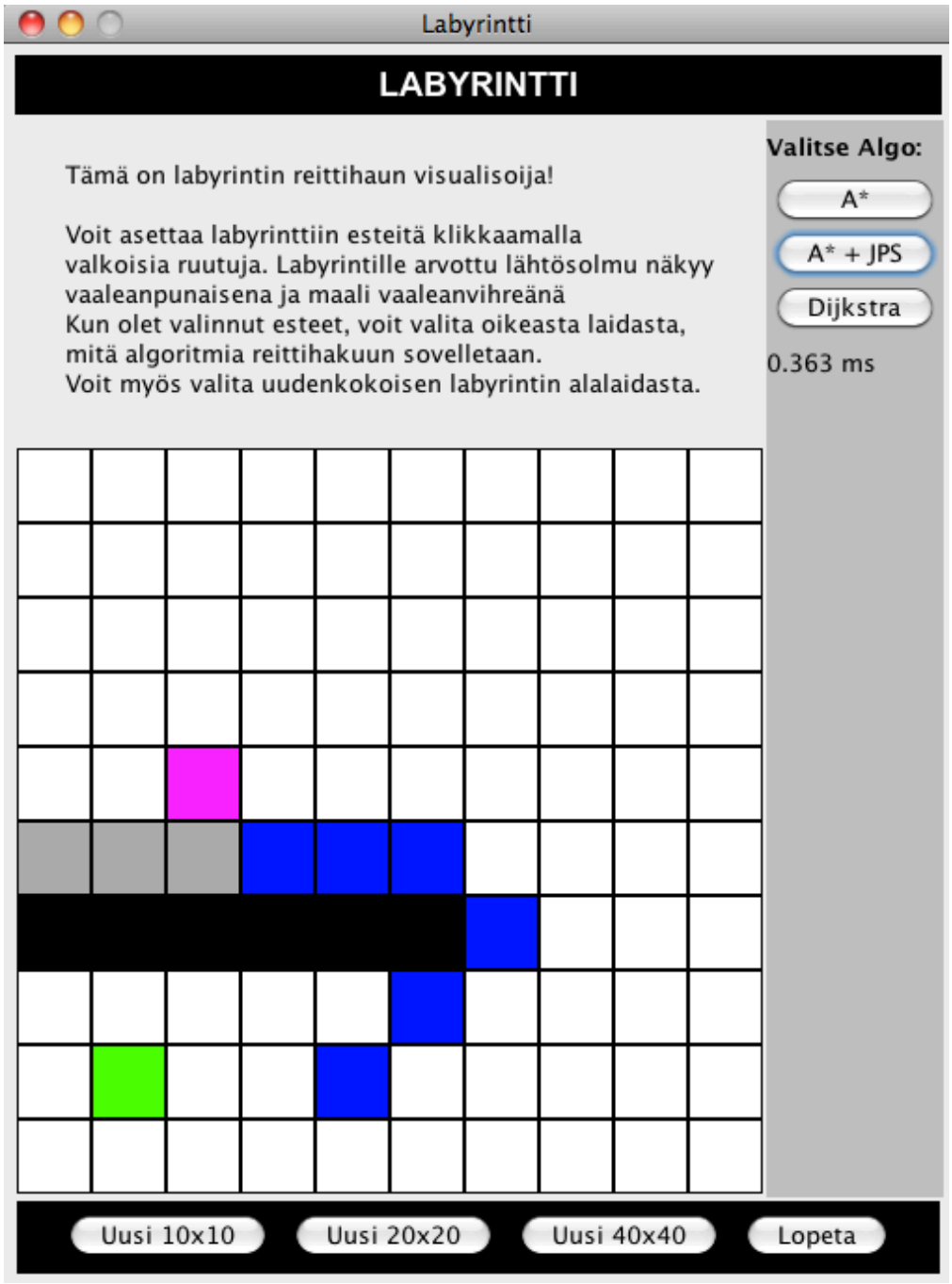
Search omalla keolla 10 000:
177695000

Search Javan PriorityQueuella 10 000:
223898000

Search JPS:lla 10 000:
276806000

4. Käyttöliittymä

Koska käyttöliittymän (kuva alla) toteuttamiseen jäi hyvin vähän aikaa, päätin käyttää toteutuksessa pohjana *Ohjelmoinnin harjoitustyöhön* rakentamaani käyttöliittymää (aiheena ko. harjoitustyössä minulla oli muistipeli). Käyttöliittymä on toteutettu GridBagLayoutilla, ja labyrintin ’ruudut’ ovat käytännössä JButtoneita, joita painamalla labyrintin vastaava solmu muuttuu esteeksi. Periaatteessa toteutus on ihan toimiva pienessä mittakaavassa, mutta labyrintin koon kasvaessa JButtonien valtava määrä alkaa käydä ilmeisesti muistille liian raskaaksi (tai sitten toteutus ei vain muuten ole tarpeeksi optimoitu, niin kuin ei varmasti vielä olekaan). Käyttöliittymään liittyvät luokat löytyvät pakkauksesta labyrintti.sovellus.



5. Bugeja / Parannettavaa

Kuten vertailu-kohdassa mainitsin, työ jäi minulta nyt jonkin verran kesken - löysin viime metreillä vielä suuria bugeja, kun pääsin visualisoimaan algoritmeja käyttöliittymän kautta. Osan bugeista sain vielä korjattua, mutta jostakin syystä, jota en enää ehtinyt paikantaa, haut laajentuvat etenkin lähtösolmun lähellä väärään suuntaan. JPS ei myöskään hypi kaikkien solmujen yli, joiden yli kuuluisi hyppiä.

Jatkossa hakuja pitäisi päästä tekemään ja testaamaan isommilla kartoilla. Jotenkin en tajunnut ajatella labyrintin käsitettä tarpeeksi isolla skaalalla. Olisi hyvä myös päästä vertailemaan edelleen erilaisia etäisyysheuristiikkoja jne. Työtä pystyisi muutenkin vielä algoritmien korjaamisen jälkeen laajentamaan moneen suuntaan, esim. kokeilemalla

Muutenkin työtä pystyy vielä algoritmien korjaamisen jälkeen laajentamaan moneen soveltaa reittihakua johonkin pelitilanteeseen (esim. Pacman).
Omaan työskentelyyn/ohjelmointiin sain tästä työstä paljon oppia, vaikka homma jäikin lopulta vähän kesken. Opin esimerkiksi, kuinka tärkeää on (=olisi ollut) työstää ohjelmaa todella pienissä palasissa, ja testata pienetkin metodit todella kattavasti ennen siirtymistä suurempiin kokonaisuuksiin. Myöhemmin virheiden metsästämisestä tulee entistä vaikeampaa. Opin myös vihdoinkin(!) käyttämään NetBeansin Debugging-työkaluja - en tiedä, miksi en ole aikaisemmin tajunnut niiden erinomaisuutta!

Lähteet:

[1] Tietorakenteet ja algoritmit -kurssin materiaali: http://www.cs.helsinki.fi/u/floreen/tira2013syksy/tira.pdf

[2] Johdatus tekoälyyn -kurssin materiaali: https://www.cs.helsinki.fi/webfm_send/1245

[3] Wikipedia: http://en.wikipedia.org/wiki/A*_search_algorithm

[4] Harabor, D. & Grastien, “A. *Online Graph Pruning for Pathfinding on Grid Maps”.* Available at: http://grastien.net/ban/articles/hg-aaai11.pdf

[5] http://zerowidth.com/2013/05/05/jump-point-search-explained.html