Toteutusdokumentti Graphs
Tiralabra
14.6.2013
Elmeri Haapasalmi

Aiheena oli kahden reitinhakualgoritmin vertaaminen. Ensimmäinen, A-star, on yleisesti käytetty polunetsintä algoritmi ja on muunnos Djikstran algoritmista. Jump Point Search joka oli toinen vertailtava, on puolestaan muunnos A-star:ista. JPS:n ehtona on painoton 2d-ruudukko. Niinpä verkkoa esittää kaksiulotteinen kokonaislukutaulukko.

Olen toteuttanut näiden avuksi myös neljä tietorakennetta: minimikeon, solmun, jonon ja pinon. Solmua tarvitaan reitinhaussa esittämään verkon solmuja ja muodostamaan puurakenne algoritmin edetessä. Minimikekoa tarvitaan toteuttamaan prioriteettijono sekä A-starissa että JPS:ssä. Pinon tarvitsin reitin kääntämiseen ympäri maalisolmusta lähtösolmuun. Jonon otin käyttöön koska halusin jonkin dynaamisen joukon javan kiinteiden taulukoiden tilalle ja jono on yksi yksinkertaisimmista tietorakenteista toteuttaa.

Projektissa on myös mukana graafinen kuvaus algoritmeistä pakkauksessa gui mutta sitä ei pääohjelma enää aja ja se tarvittiin lähinnä debuggaukseen. Enää sillä on arvoa ainoastaan jatkokehitystä silmällä pitäen.

Pääohjelmassa otetaan keskiarvoaika käyttäjän antaman luvun verran. Jokaisella iteraatiolla keskiarvoajat tulostetaan standardi tulostusvirtaan.

O-analyysi

Pino:

double_up() on pahimmassa tapauksessa O(n) suhteessa pinon kokoon sillä se tuplaa käytetyn taulukon koon ja kopioi kaikki alkiot uuteen taulukkoon.

push() on vakioaikainen melkein aina paitsi tapauksessa jossa pino täyttyy ja se täytyy tuplata jolloin aika on sama kuin funktiolla double_up()

pop() on vakioaikainen sillä palautamme alkion pinon päältä.

is_empty() on vakioaikainen sillä palautamme yhden boolean vertailun arvon.

Jono:

enqueue() on vakioaikainen sillä laitamme yhden alkion taulukkoon.

dequeue() on vakioaikainen sillä palautamme alkion taulukosta.

is_empty() on vakioaikainen sillä palautamme yhden boolean vertailun.

Minimikeko:

pushdown() on O(log n) aikainen sillä se käy pahimmassa tapauksessa keon koko syvyydessään läpi.

double_up() on O(n) aikainen sillä taulukon koko tuplataan ja kaikki alkiot kopioidaan.

removemin() ei ainoastaan etsi pienintä vaan myös palauttaa ja poistaa sen joten sen pahin tapaus on O(log n) sillä metodia pushdown() joudutaan kutsumaan.

insert() on O(log n) sillä alkio asetetaan keon perälle ja jos se sattuu olemaan pienempi kuin mikään muu se nousee sieltä koko matkan keon läpi ensimmäiseksi.

remove_middle() on O(n + log(n)) aikainen siis lineaarinen sillä poistettava alkio joudutaan ensin etsimään keosta. Koko taulukon läpikäyminen vie pahimmassa tapauksessa O(n) ajan. Itse alkion palauttaminen ja pushdown() operaatio vievät vain O(log n)

is_empty() on vakio aikainen kuten edellisissäkin rakenteissa.

A-star:

Tarkastellaan A-staria hieman tarkemmin pseudokoodi esityksen avulla:

```
keko = vain aloitussolmu
suljettu joukko = tyhjä

while pienimmän f_arvon omaava keossa ei ole maalisolmu:
        nykyinen = solmu jolla on pienin arvo keosta
        lisätään nykyinen suljettuun joukkoon

        kaikille nykyisen naapureille:
                hinta = etäisyys alusta + arvio maalietäisyydestä
                jos naapuri on jo keossa ja hinta on pienempi kuin vanha arvo
                        poistetaan naapuri keosta, päivitetään arvo ja lisätään kekoon uudelleen
                jos naapuria ei ole vielä tavattu (ei keossa tai suljetussa joukossa)
                        asetetaan nykyinen naapurin edeltäjäksi
                        asetetaan sille hinta ja lisätään se kekoon
```

käydään läpi löydetty reitti maalisolmusta lähtösolmuun seuraten maali -> edeltäjä
asetetaan kyseinen solmu pinoon

palautetaan pino

A-starin nopeus riippuu heuristiikkafunktiosta ja avoimen joukon, tässä tapauksessa keon, toteutuksesta. Jos arvio maalietäisyydestä on 0, muuttuu A-star Djikstran algoritmiksi ja lyhimmän reitin löytäminen on taattu. Nyt pahin tapaus on tunnetusti O((kaaret + solmut)log(solmut)). Toisaalta jos heuristiikka (arvio maalietäisyydestä) on todella suuri suhteessa etäisyyteen lähdöstä, muuttuu A-star ahneeksi paras ensin -algoritmiksi, eikä optimipolkua välttämättä löydetä. Tällöin A-star on kuitenkin nopeimmillaan. Näiden kahden ääripään hidas/optimi ja nopea/ei-optimi välistä löytyy kolmas vaihtoehto. Jos heuristiikka-arvo on pienempi tai yhtäsuuri kuin hinta liikkua nykyisestä solmusta maalisolmuun, löytää A-star aina lyhimmän polun. Oikeanlaisen heuristiikan valitseminen on siis erittäin tärkeää hyvän tuloksen saamiseen. Itse olen käyttänyt Manhattan-heuristiikkaa joka tuskin on paras vaihtoehto ruudukossa jossa voi liikkua 8-suuntaan.

Jos käytetään kekoa prioriteettijonona, on tärkeä muistaa, että keosta alkion löytäminen vie O(n) aikaa keon koon suhteen. Kuten pseudokoodista huomaa, jos löydetty naapuri on jo keossa, täytyy kyseinen naapuri etsiä sieltä ja laittaa sinne uudelleen. Poistaminen ja laittaminen kekoon ovat O(log n) aikaisia mutta etsiminen O(n). Etsimisen saa kuitenkin vakioaikaiseksi pitämällä viitettä alkion sijaintiin (keossa) jossain muualla. Yksi kehityssuunta tälle työlle olisi tallettaa aina Node-olion sijainti keossa vaikka kyseiseen olioon. Näin A-staria saisi mahdollisesti nopeutettua sillä Noden päivittäminen muuttuisi O(n) -> O(log n).

Jump Point Search:

Jump Point Search eroaa A-starista oikeastaan vain kahden funktion verran: naapurien valinta ja jump-funktio.

Naapurien valinnassa osa naapureista voidaan jättää tarkastelematta. Ehdot joilla osa naapureista jätetään pois voidaan käydä läpi vakioajassa. Funktio on siis myös JPS:ässä vakioaikainen.

Jump-funktio on hieman monimutkaisempi ja se suoritetaan jokaiselle valitulle naapurille. Alla pseudokoodiesitys:

jump(nykyinen solmu, maalisolmu, suunta)
nykyinen solmu = askel suuntaan
jos nykyinen solmu on ulkona verkosta tai ei ole solmu
palauta null

```
jos nykyinen solmu on maalisolmu
        palauta nykyinen solmu
jos joku nykyisen solmun naapureista on pakotettu naapuri
        palauta nykyinen solmu
jos suunta on viisto
        // molemmille suunnan koordinaateille (x,y) tehdään rekursiivinen hyppy
        suunta = (x,0)
        jump(nykyinen solmu, maalisolmu, suunta)
        suunta = (0,y)
        jump(nykyinen solmu, maalisolmu, suunta)
        jos jompikumpi palauttaa muuta kuin null
                palauta nykyinen solmu
palauta jump(nykyinen solmu, maalisolmu, suunta)
```

Ainut ero A-star algoritmiin ison O:n notaatiossa on tämä jump-funktio.

Pahin tapaus on, jos hypätään kulmasta kohti vastakkaista kulmaa, siten että maalisolmu on vastakkaisessa kulmassa ja ruudukossa ei ole pakotettuja naapureita. Olkoon n neliöjuuri ruutujen määrästä. N on yhden sivun pituus sekä lävistäjän pituus. Jump funktion runko on vakioaikainen. Jokaisella askeleella (i = 1..n) kohti vastakkaista kulmaa tehdään siis kaksi kertaa n - i mittaista rekursiokutsua. Sivuttain ruudukossa liikkuvat rekursiot eivät enää hajaannu uudestaan joten kyseessä ei ole puurekursio! Funktio vierailee tasan kerran jokaisessa ruudussa siten aika on O(runko x (n+1)^2). Huomaa että funktio palauttaa null astuttuaan ulos ruudukosta. Siksi ruutujen määrä on oikeastaan (n+1)^2 eikä n^2.

Miten jump-funktio vaikuttaa A-starin pahimpaan tapaukseen en osaa arvioida. Muutenkin algoritmin tarkempi analyysi jätetään valistuneemmille lukijoille.

Käytännössä JPS näyttäisi olevan kaikissa tapauksissa A-staria nopeampi. Tarkempia tuloksia voi katsoa artikkelista: Harabor & Grastien, 2011. Oma nopeusvertailuni löytyy testausdokumentista.

Työtä voisi parantaa rakentamalla uskottavampia karttoja vertailtavaksi. Tällä hetkellä ruudukkoon arvotaan satunnaisia pisteitä esteiksi. Tämä tuskin kuvaa kovin hyvin algoritmin mahdollista käyttöä oikeissa tilanteissa. Muutenkin JPS:än ja A-starin huonoimmat ja parhaimmat tapaukset olisi mielenkiintoista kaivaa esiin. Voisi etsiä verkot joilla algoritmit suoriutuvat erittäin hyvin tai poikkeuksellisen huonosti.

Lähteet:

Harabor & Grastien, 2011: Online Graph Pruning for Pathfinding on Grid Maps
http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf

Daniel Harabor, Blog, 15.06.2013
http://harablog.wordpress.com/2011/09/07/jump-point-search/
Amit Patel, 15.06.2013
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html