

Rakenne

Ohjelma on täysin kirjoitettu Java-kielellä ja se koostuu viidestä eri pääpakkauksesta, jotka ovat: io, käyttöliittymä, logiikka, main sekä util. Io-pakkauksessa sijaitsee Tulostaja-luokka, jonka tehtävänä on tulostaa kartta luettavaan muotoon tekstikäyttöliittymässä. Käyttöliittymässä sijaitsee Java Swing – GUI, joka toimii ohjelman graafisena käyttöliittymänä sekä Piirtaja, jonka tehtävänä on piirtää karttaan polkua. Logiikassa sijaitsevat omat tietorakenteet kuten ArrayList, PriorityQueue sekä Jono, mutta myös itse A*-algoritmin toteutus, Dijkstra, sekä MT-AA* (Moving Target – Adaptive A*). Reitinhakuun liittyvät luokat ovat samassa pakkauksessa. Muut luokat ovat reitinhaun tukena kuten Analysoija, jonka tehtävänä on analysoida annettu kartta, Heurestiikka joka laskee heurestisen arvon reitinhaku-algoritmillemme, Solmu joka toimii reitinhaku-algoritmin verkon solmuina sekä Enum Ympäristömuuttuja, jossa sijaitsevat globaalit muuttujat, kuten lokaali äärettömyys sekä solmujen painot. Mainissa sijaitsee ohjelman pääohjelma ja lopuksi util kansiossa ovat muun muassa karttakuvat .bmp -muodossa sekä Kuva ja Piste luokka, jotka auttavat sekä GUI:ssa, mutta myös reitinhaku-algoritmissa.

MT-AA* sekä ohjelman toiminta

MT-AA* on oikealta nimeltään Moving Target-Adaptive A* ja se pohjautuu hyvin vahvasti A*-algoritmiin ja heurestiikkaan. Nimen ”Adaptive”-osuus tulee siitä, että se käyttää hyödykseen Adaptive A*-algoritmin ideaa tallentaa heurestiset arvot perustuen peräkkäisten hakujen jälkeisiin etäisyysarvioihin. Tämän tallennuksen kaava on

$$h(s) := g(s_{\text{target}}) - g(s).$$

, jossa s tarkoittaa solmua ja g -tarkoittaa etäisyyttä lähtösolmusta. Se suoritetaan kaikille analysoiduille solmuille, jolloin heurestisia arvoja saadaan tarkennettua ja parannettua näin nopeuttaen seuraavaa samoihin kohtiin osuvaa hakua.

Moving Target taas tarkoittaa, että algoritmi pystyy reagoimaan tilanteisiin, joissa maali vaihtuu kesken reitinhakua. Maalin vaihtuessa MT-AA* päivittää jokaisen solmun heurestiset arvot kaavalla

$$h'(s) := \max(H(s, s'_{\text{target}}), h(s) - h(s'_{\text{target}}))$$

, jossa uusi heurestinen arvo on maksimi heurestista arvosta nykyisen solmun sekä uuden maalin välillä ja nykyisestä heurestisesta arvosta miinus uuden maalin entisestä heurestisesta arvosta.

Toteuttamani MT-AA* on etuliitteeltään *Eager*, joka tarkoittaa, että heurestiikka esilasketaan ennen reitinhakua, mutta algoritmista on olemassa myös hieman tehokkaampi ja monimutkaisempi *Lazy*-versio. Ero Eagerin ja Lazyn välillä on, että Eager käyttää esilaskentaa heurestiikassa ja Lazy laskee samalla, kun reitinhakua suoritetaan. Olen myös toteuttanut ohjelmassani MT-AA*-algoritmin lisäksi sen inspiraatiot eli A*-algoritmin ja A*:n inspiraation Dijkstran algoritmin.

Toteutusdokumentti
Kristian Wahlroos

Ohjelmani toiminta perustuu syötteenä saatuihin kuviin, joissa halutaan hakea reitti kahden pisteen A ja B välillä. Näiden pisteiden värit kertoo pisteen roolin kartassa jolloin esimerkiksi vihreä tarkoittaa lähtöä ja punainen maalia. Läpipääsemättömät alueet on merkattu mustalla ja sininen tarkoittaa, että alueen läpi voidaan kulkea, mutta se on hidasta. Kartat voivat olla muodoiltaan minkälaisia tahansa, mutta ohjelma odottaa löytävänsä maalin ja lähdön ennen reitinhakuja. Karttojen muoto oletetaan olevan .bmp-muotoa, koska niissä värit ovat määritelty hyvin RGB-arvoiksi väleille 0-255, jolloin kuvan analysointi kartaksi toimii, kuten tarkoitettu.

Saavutetut aikavaativuudet

Algoritmin aikavaativuus on $O(|E| + |V| \cdot \log|V|)$ eli sama kuin Dijkstrassa. Tämä johtuu siitä, että algoritmin ydin ja pohja nojaa samaan ideaan, kuin Dijkstran algoritmi. Ainoa ero näiden kahden välillä kuitenkin on vakiokertoimet. A* ja MT-AA* käyvät solmuja paljon vähemmän kuin Dijkstra ja parhaassa tapauksessa vaadittava solmujen määrä on vain 5-10% Dijkstrasta. Tehokkuus perustuu pääosin heurestiikan tuomaan etuun, mutta myös MT-AA*:n ideologiaan tallentaa analysoitujen solmujen tietoja jokaisen haun jälkeen.

Suorituskyky ja O-analyysivertailu

Suorituskyvyltään algoritmi MT-AA* on huomattavasti nopeampi, kuin perinteinen Dijkstra peräkkäin ajettuna käytännössä millä tahansa kokoisella kartalla. Se pystyy myös käsittelemään kokoa $100 \cdot 1000$ olevia karttoja noin sekunnissa, joissa solmujen määrä nousee jopa 100 000 solmuun.

Kartan pohjapiirustuksella on ja ei ole väliä. Tämä tarkoittaa käytännössä sitä, että vaikeat kartat tuovat algoritmia lähelle Dijkstraa, mutta avoimet ja luonnolliset taas saattavat olla hyvinkin nopeita MT-AA* suoritukselle ja analysoinnille. Hyvässä kartassa operaatioiden määrä voi jäädä hyvinkin alhaiseksi heurestiikan takia. Onkin mielekkäämpää tarkastella keskimääräistä tapausta, joka Dijkstralla on

$$O(|E| + |V| \log |V|)$$

, mutta A*:lla ja MT-AA*:lla voidaan päästä polynomiseen aikaan, jos heurestin funktio täyttää seuraavan ehdon

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

Tämä ehto määrittelee täydellisen heurestiikan, jonka virhe ei kasva nopeammin, kuin h^* , joka palauttaa todellisen etäisyyden x-pisteestä maaliin.

Pahimmassa tapauksessa MT-AA*:n aloittaessa keskeltä huonetta ja maalin ollessa vaikeassa paikassa, voi algoritmi käydä yhtä paljon solmuja läpi kuin Dijkstra. Tämä kuitenkin tapahtuu erittäin harvoin ja on hyvin epätodennäköistä, että MT-AA*:n tehokkuus laskisi näin alas.

Useissa peräkkäin ajetuissa hauissa MT-AA* laskeekin entisestään A*:n vakiokerrointa jolloin uuteen hakuun tarvittavien analysoitavien solmujen määrä laskee ja algoritmi tehostuu jopa 50% A*:kin nopeammaksi peräkkäisillä hauilla.

Ainot hitaudet algoritmiin kuitenkin tuovat ohjelmassa tarkoituksella hidastettu MT-AA*, Javan oma metodi polun piirtämiselle sekä omien tietorakenteiden implementaatiot. Tarkoituksella hidastettu MT-AA* tarkoitus on tehdä mahdolliseksi sen, että käyttäjällä on aikaa kertoa uusi maali algoritmille klikkaamalla. Omien tietorakenteiden ajoittainen hitaus johtuu, ettei niitä ole optimoitu toisin kuin Javan omat tietorakenteet, jolloin muistinkäyttö ja tätä kautta tilavaativuus kasvavat. Jos tarkastelemme kaikkien toteutettujen algoritmien tilavaativuutta, niin voimme nähdä, että ylivoimesti raskain on MT-AA*. Syy johtuu sen tallentamista tuloksista ja samasta ongelmasta, joka A*-algoritmiakin vaivaa; heurestisten arvojen tallentamisesta ja sen tuomasta muistinkäytön kasvamisesta.

Ohjelma toimii hyvin syötteillä, joissa kartan koko on sellainen, että solmujen määrä jää noin 200 000. Sitä suuremmilla kartoilla voi ajoissa esiintyä hidastetta, koska muistinkäyttö kasvaa erittäin suureksi ja tuhansien solmujen piirtäminen on hidasta. Muistin kasvu johtuu siitä, että talletettu arvoja on paljon; kartta RGB-arvoina, kartta muutettuna RGB-arvoista vastaamaan verkon painoja, heurestiset arvot, oliot, etäisyysarvot, analysoidut solmut ja lopuksi lyhin polku. Osan näistä Javan roskienkerääjä pystyy hyvin keräämään pois, mut osa on aktiivisesti käytössä aina ajon alusta sen loppuun. Kaikista parhaimmat syötteet koon lisäksi ovat myös sellaiset syötteet, joissa kartta ei ole labyrinttimännen. Labyrintit ovat hyviä testaamaan erilaisia tehokkuuksia, mutta visuaalisesti sekä eroavaisuuksien näkemiseksi parhaimmat kentät ovat mahdollisimman realistiset kentät, joissa esiintyy myös isoja tyhjiä alueita. Näissä tyhjiissä alueissa tulevat MT-AA*:n sekä A*:n tehot hyvin esille. Syy miksi labyrintit eivät ole kovinkaan hyviä on se, että labyrinteissä saattaa olla paljon umpikujia, jotka tuovat lähelle maalia, mutta eivät vie oikeasti lähemmäksi maalia. MT-AA*, A*, eikä Dijkstra osaa varautua tällaisiin tilanteisiin johtuen siitä, että ne ovat niin sanottuja ahneita algoritmeja.

Puutteet ja parannusehdotukset

MT-AA* ei ole kaikista tehokkain algoritmi, koska se luottaa A*-algoritmiin. A* on osittain jopa ehkä hieman hidas algoritmi ja sitä paljon tehokkaampi on esimerkiksi JPS (Jump Point Search). Myöskin isoissa esim. 1000 * 1000 verkoissa ongelmaksi tulee se, että solmuja on liian turhan paljon jolloin analysoitavia solmuja voitaisiin ottaa mukaan vain esimerkiksi joka toinen, joka tehostaisi huomattavasti analysointia sekä verkon luontia. Kaikista tehokkain ratkaisu olisi kuitenkin päästä luopumaan *Eager*-tyylisestä ratkaisusta ja päätyä kohti *Lazy*-tyylistä ratkaisua, jossa arvot lasketaan vasta, kun niitä tarvitaan oikeasti.

Ajoittain ohjelma ei aina piirrä lyhintä polkua, joka johtuu synkronointiongelmista, kun muistinkäyttö on erittäin suuri. Tämä saattaa tapahtua esimerkiksi kartassa thesus.bmp.

Ohjelmani toimii muuten, kuten julkaisun *Speeding Up Moving Target Search* esittelemä algoritmi MT-AA*, paitsi maalin vaihtumisen tuomaa heurestisten arvojen päivittämistä tämän raportin alussa mainitulla kaavalla en saanut toimimaan kuten sen todennäköisesti oli tarkoitus toimia. Omassa ohjelmassani kävi niin, että jos kaavaa käytettiin maalin vaihtumiseen, niin se joskus toimi, mutta joskus haku rupesi hylkimään maalia eikä koskaan saavuttanut sitä. Tästä syystä toteutin hieman oman tavan ottaen mallia A*:sta, jolloin sain kaiken toimimaan. Suorituskyvyssä on siis hieman vajaavaisuutta puhtaaseen MT-AA*-algoritmiin.

Toteutusdokumentti
Kristian Wahlroos

Algoritmin aikavaativuutta voidaan parantaa käytännössä lisäämällä tilavaativuutta. Helpoiten tämä nähdään, kun ajetaan algoritmi antamalla lisää muistia esimerkiksi komennolla `java -jar -Xms6G`. Nyt ohjelma toimii huomattavasti paremmin, kun muistia on enemmän käytössä. Jos taas tarkastelemme algoritmia ihan algoritmiselta kantilta, niin aikavaativuuden kasvattaminen vaatisi käytännössä jonkilaisten ratkaisun sille, ettei heuristisia arvoja tarvitsisi käydä päivittämässä jokaiselle solmulle erikseen. Maalin vaihtumisessa voitaisiin esimerkiksi pitää yllä polkua nykyiseen solmuun asti ja noutamalla viereisten solmujen heuristiset arvot niin, että saataisiin tarpeeksi tarkat arvot, jotta haku jatkaisi oikeeseen suuntaan. Myös yksi ratkaisu olisi tallentaa heuristisista arvoista useita eri hetkiä, jolloin maalin liikkuesssa paikkaan, jossa heuristiikka on jo oltu laskettu kerran, voitaisiin hyödyntää tätä tietoa ja olla laskematta heuristiikkaa enää uudestaan. Toisaalta ehkä suurin pullonkaula on algoritmissa minimikeko ja verkon tallennus. Minikeon vaihtaminen esimerkiksi Fibonacci-kekoon nopeuttaisi huomattavasti algoritmin ytimen toimintaa.

Lähteet

- <http://idm-lab.org/bib/abstracts/Koen07e.html>
- <http://idm-lab.org/bib/abstracts/papers/aamas07.pdf>
- http://en.wikipedia.org/wiki/D*
- http://en.wikipedia.org/wiki/Incremental_heuristic_search
- http://en.wikipedia.org/wiki/A*_search_algorithm