Toteutusdokumentaatio

Yleisrakenne

Ohjelman käyttämät oheistiedostot on kuvattu tarkemmin käyttöohjeessa.

Ohjelman päätoiminnallisuus keskittyy **Pathfinding**- ja **AlgorithmRunner**-luokkiin. Pathfinding määrittää ajon parametrit sekä luo **MazeReaderin** ja AlgorithmRunnerin. MazeReader-luokka lataa tekstitiedostosta sokkelon, joka syötetään AlgorithmRunnerille. AlgorithmRunner ottaa myös tiedon käytettävästä polunetsintäalgoritmista, takaa-ajajan ja takaa-ajetun **MazeEntity**-instansseista sekä siitä, sallitaanko liikkuminen ruudukossa vinottain.

AlgorithmRunner parseroi sokkelosta verkon, jonka solmut se kytkee toisiinsa. Seinät eivät sisälly verkkoon lainkaan, ne jätetään siitä pois jo parserointivaiheessa. Näin polunetsintäalgoritmin ei tarvitse käyttää aikaa tarkistaakseen, onko ruutu kuljettava vai ei (paitsi ovien tapauksessa). Se, kytketäänkö vinottaiset ruudut toisiinsa, riippuu siitä, onko vinottain liikkuminen sallittu. Solmulla on siis 4 tai 8 naapuria.

Verkko tallennetaan kaksiulotteiseen taulukkoon **Node**-olioita. Nodella on tieto naapurisolmuistaan, tyypistään, sijainnistaan ruudukossa sekä algoritmien ja tietorakenteiden käyttämiä muuttujia.

AlgorithmRunner luo instanssin algoritmiluokasta, jonka se käskee ratkomaan sokkelon. Tällöin algoritmi etsii polun aloitussolmusta loppusolmuun, ja muodostaa reitin antamalla kulkemilleen solmuille parent-solmun. Kun maali löytyy, AlgorithmRunner kulkee polun lopusta alkuun parent-viitteitä pitkin ja generoi reitistä listan. Jos maalia ei löydy, saadaan tyhjä lista.

Pathfinding-luokka puolestaan liikuttaa takaa-ajajaa ja takaa-ajettua (jotka ovat MazeEntity-tyyppisiä olioita) sokkelossa sekä vastaa sokkelon muutoksista (ovien avautumiset ja sulkeutumiset), ja pyytää AlgorithmRunneria tarvittaessa laskemaan reitin takaa-ajajalle uudestaan. Takaa-ajaja käyttää liikkuessaan AlgorithmRunnerilta saatua solmulistaa, kun taas **TargetMover**-luokka liikuttaa takaa-ajettua tiedostosta ladattujen liikkeiden perusteella.

Algoritmit toteuttavat **Algorithm**-rajapinnan, joka sisältää AlgorithmRunnerin kutsuman **FindPath**(alkusolmu,loppusolmu)-metodin.

Position-luokka on sijaintitiedon esittämiseen. Sen instanssit sisältävät x ja y -koordinaatit.

OwnArrayList on ArrayList-tietorakenteen oma toteutus, **OwnBinaryHeap** binäärikeon ja **OwnQueue** jonon. Algoritmit käyttävät näitä tietorakenteita. **NodeComparator** puolestaan on komparaattori, jota OwnBinaryHeap käyttää solmujen vertailuun. Vertailutapa riippuu siitä, käytetäänkö kekoa A*:ssä vai Dijkstran algoritmissa.

ImageLoader, **Renderer** ja **Window** ovat käyttöliittymäluokkia eivätkä ohjelman toiminnan kannalta oleellisia. ImageLoader lataa visualisointiin käytetyt kuvat, Window luo ikkunan ja Renderer piirtää sokkelon sisältöineen ikkunaan.

Algoritmit

Aikavaativuuksissa n on solmujen määrä ja e kaarien.

A*

A*-algoritmin pitäisi olla suhteellisen nopea ja se löytää optimaalisimman reitin. Se toimii keon avulla. Jokaisella solmulla on kolme arvoa: g, f ja h. G on kuljettu matka lähtösolmusta tähän solmuun, h on arvioitu etäisyys maalisolmuun ja f on näiden summa. Keon avulla algoritmi pyrkii kulkemaan kevyintä reittiä pitkin. Tämä reitti on se, jonka f on pienin.

Algoritmi laskee h-arvon etäisyytenä suoraan linnuntietä nykyisestä solmusta maalisolmuun Pythagoraan lauseella.

Algoritmi ottaa huomioon kaarien painot: kyljet vastakkain olevien solmujen välinen etäisyys on 10, vinottaisten 14. Tämä on pyöristys kahden neliöjuuresta. Suoruuduilla on kaksinkertainen ja jääruuduilla puolikas paino.

Aika 401x401-sokkelon ratkaisemiseen: 85ms - *Saadun reitin pituus:* 2029 - *Aikavaativuus:* O((e+n)*log(n))* - Saavutettu aikavaativuus: *O((e+n)*log(n))

Dijkstran algoritmi

Dijkstran algoritmin toteutus käyttää myös kekoa. Aluksi se merkitsee etäisyyden kaikkiin muihin solmuihin äärettömäksi. Tämän jälkeen se alkaa käydä verkkoa läpi kevyintä reittiä pitkin, ja laskee matkan varrella oleville solmuille etäisyydet alkusolmusta. Jos algoritmi törmää jo löydettyyn solmuun, se tarkistaa onko uusi reitti vanhaa kevyempi; jos on, se korjaa solmun etäisyysarvon.

Algoritmi käyttää samoja kaaripainoja kuin A*: 10 vierekkäisille ja 14 vinottaisille.

Aika 401x401-sokkelon ratkaisemiseen: 58ms - *Saadun reitin pituus:* 2029 - *Aikavaativuus:* O((e+n)*log(n))* - Saavutettu aikavaativuus: *O((e+n)*log(n))

Leveyshaku

Leveyshaku on Dijkstran algoritmia ja A*:ta yksinkertaisempi. Se on nopeampi, ja löytää lyhimmän reitin, mutta ei kuitenkaan ole täysin vertailukelpoinen näiden kanssa, sillä se ei huomioi kaarien painoja. Leveyshaku käy solmuja läpi jonon avulla niin, että se tutkii verkkoa joka suuntaan tasaisesti.

Aika 401x401-sokkelon ratkaisemiseen: 30ms - *Saadun reitin pituus:* 2029 - *Aikavaativuus:* O(n+e) - *Saavutettu aikavaativuus:* O(n+e)

Syvyyshaku

Syvyyshaku on anhe eli nopea algoritmi, mutta se löytää epäoptimaalisen reitin. Se lähtee kulkemaan yhteen suuntaan kunnes ei pääse enää eteenpäin, tämän jälkeen pakittaa ja jatkaa seuraavasta haarasta. Toteutus valitsee aina solmun naapurilistasta ensimmäisen; näin ollen se suosii oikealle kulkemista (oikea solmu on listassa ensimmäisenä).

Syvyyshaun löytämä reitti olisi oikeassa navigointikäytössä käyttökelvoton, mutta silti hyödyllinen, jos riittää tietää, onko reitti kahden solmun välillä olemassa.

Aika 401x401-sokkelon ratkaisemiseen: 26ms - *Saadun reitin pituus:* 7921 - *Aikavaativuus:* O(e) - *Saavutettu aikavaativuus:* O(e)

Ongelmat ja parannusehdotukset

Ohjelman suurin kompastuskivi tällä hetkellä on siinä, että algoritmit ja tietorakenteet tallentavat tietoja itse solmuihin, jolloin ne joudutaan resetoimaan aina kun algoritmi ajetaan uudestaan.

- Algoritmit tallentavat solmuihin esimerkiksi tietoja siitä, onko niissä käyty, etäisyyksiä maaliruutuun jne.
    - Tämä yksinkertaistaa ohjelman toimintaa, eikä algoritmissa tarvitse käyttää useaa hashsetiä.
    - Solmuun tallennettaisiin joka tapauksessa ainakin tieto sen vanhemmasta (jolloin polku voidaan luoda käymällä vanhemmat läpi), tätä ei kuitenkaan tarvitsisi resetoida sillä se päällekirjoitetaan algoritmia ajettaessa.
- Binäärikeko tallentaa solmuun solmun indeksin keossa.
    - Tämä mahdollistaa keolle nopean solmun indeksin haun pelkän solmutiedon perusteella. Katsoin tämän olevan sopiva uhraus, sillä vaikka aina polun uudelleenlaskennan yhteydessä joudutaan käymään koko verkko läpi ja resetoimaan jokainen solmu, on tähän käytetty aika kuitenkin vain noin 3% koko ohjelman suoritusajasta. Tämän järjestelmän korvaaminen algoritmeissa hashseteillä ja binäärikeossa keon läpikäynnillä on periaatteessa yksinkertaista.

Syvyyshakualgoritmi on rekursiivinen, joten suurilla sokkeloilla se kaatuu stack overflow -virheeseen.

Vielä monipuolisemman testauksen mahdollistaisivat erikseen eri oville säädettävät aukeamis- ja sulkeutumisvälit sekä säädettävä paino jokaiselle ruudulle. Niiden toteutus ei kuitenkaan ole testaustilanteisiini tarpeellista.