

Määrittelydokumentti

Labyrintingeneroija ja –ratkoja

Tietorakenteet ja algoritmit harjoitustyö

Juri Kuronen

Kesä 2014

Sisältö

1. Johdanto.....	3
2. Käytetyt algoritmit ja tietorakenteet.....	4
2.1. Tietorakenteet.....	4
2.2. Labyrintin generointialgoritmit.....	5-6
2.3. Labyrintin ratkoja-algoritmit.....	7

1. Johdanto

Labyrintin generoimiseen ja ratkaisemiseen on olemassa lukuisia algoritmeja. Tässä työssä toteutan näistä yleisimpiä ja vertailen niiden suoritusajkoja. Luon myös käyttöliittymän, jonka avulla labyrintin generoiva algoritmi, sekä muita asetuksia, voidaan valita. Tämän jälkeen voidaan valita ratkaisemiseen käytettävä algoritmi, ja käyttäjälle esitetään visuaalisesti algoritmin tekemä ”työ”, eli esimerkiksi mitkä labyrintin soluista tuli tutkittua ratkaisevan reitin löytämiseksi.

Algoritmit toteutetaan tavallisille $N \times M$ ristikoille, missä N ja M voidaan valita. Tavallinen ristikko tarkoittaa nelikulmaista ristikkoa, jonka solut ovat neliöitä, joista reitit muihin soluihin jatkuvat joko suoraan tai tekevät kohtisuoran käännöksen. Lähtö- ja maalisolun asetetaan ristikon vastakkaisiin kulmiin. Lisäksi oletetaan, että mistä tahansa labyrintin pisteestä A on vain ja ainoastaan yksi reitti mihin tahansa pisteeseen B . Tämä tarkoittaa, että labyrintti on sykliton ja yhtenäinen, ja labyrintin kaaret muodostavat virittävän puun. Lisäksi oletetaan, että ratkaisevalla algoritmilla on tieto maalisolun sijainnista.^[1]

Kaikki algoritmien käyttämät tietorakenteet on toteutettu erikseen, eli mitään Javan valmiita tietorakenteita ei käytetä.

[1] Nämä ovat erittäin alustavia oletuksia. Työn edetessä oletuksia voitaneen muuttaa käyttäjän syötteellä.

2. Käytetyt algoritmit ja tietorakenteet

2.1. Tietorakenteet

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Lista. Lorem ipsum...

Pino. Lorem ipsum...

Joukko. Lorem ipsum...

Puu. Lorem ipsum...

2.2. Labyrintin generointialgoritmit

Käyttäjä valitsee labyrintin koon $N \times M$. Labyrintti alustetaan kaksiulotteisena $N \times M$ byte-arraynä, jossa ei ole vielä ollenkaan kaaria. Tämän jälkeen aloitetaan satunnaisesti luomaan polkuja lähtösolusta oletusten mukaisesti niin kauan, kunnes koko labyrintti on käyty läpi.

Rekursiivinen peruuttava haku. Käyttää pinoa ja laittaa aina käsitellyn solun pinon päällimmäiseksi. Lisäksi käytössä on $N \times M$ -kokoinen array soluista, joissa on vierailtu.

Joka solun kohdalla siirtyy satunnaiseen vierailemattomaan soluun niin kauan, kunnes tullaan soluun, jolla ei enää ole vierailemattomia naapureita. Tällöin algoritmi peruuttaa palaamalla pinon päällimmäiseen soluun niin kauan, kunnes löytyy solu, jolla on vierailemattomia naapureita. Jos tällaista solua ei löydy (eli pino on tyhjä) on labyrintti generoitu.

- Tilavaativuus $O(|V|)$. Pahimmassa tapauksessa pinon koko on koko labyrintin koko ja lisäksi arrayn koko on koko labyrintin koko.
- Aikavaativuus $O(|V|)$. Koko labyrintti käydään läpi, ja jokaisessa solussa vieraillaan keskimäärin 2 kertaa. Pinon operaatiot ovat $O(1)$.

Primin algoritmi. Ylläpitää listaa soluista, jotka voidaan seuraavaksi liittää labyrinttiin.^[2] Lisäksi on käytössä $N \times M$ -kokoinen array labyrintin osana olevista soluista.

Aluksi merkataan lähtösolu osaksi labyrinttiä ja kaikki lähtösolun viereiset solut, jotka eivät vielä ole osana labyrinttiä, listaan soluista, jotka voidaan seuraavaksi liittää labyrinttiin. Tämän jälkeen valitaan satunnaisesti yksi solu tästä listasta, liitetään se satunnaista reittiä kautta labyrinttiin, ja lisätään tämän solun naapurit listaan. Kun lista on tyhjä, on labyrintti generoitu.

- Tilavaativuus $O(|V|)$. Listan koko on korkeintaan noin puolet labyrintin koosta, mutta arrayn koko on koko labyrintin koko.
- Aikavaativuus $O(|V|)$. Koko labyrintti käydään läpi. Jos listaan varataan tarpeeksi tilaa, listan operaatiot ovat kaikki aina $O(1)$. Ilman tilanvarausta, lisäys on kuitenkin amortisoidusti vakio-operaatio.

[2] Tämä on siis oikeastaan *modifioitu* Primin algoritmi. Tavallisesti Primin algoritmi ylläpitää listaa kaarista.

Kruskalin algoritmi. Jakaa aluksi joka solun omaksi joukoksi ja luo arrayn mahdollisista kaarista.

Joka suorituskerralla, arvo solu ja sen jälkeen arvo mahdollinen kaari.^[3] Jos tämä kaari yhdistäisi kaksi eri joukkoa, yhdistä nämä joukot ja luo labyrinttiin kaari tähän kohtaan. Kun kaikki solut kuuluvat samaan joukkoon, on labyrintti generoitu.

- Tilavaativuus $O(|V|)$. Joukkoja on labyrintin koon verran, sillä soluja on näin monta. Array kaarista on koko labyrintin kokoinen, ja joka solulla on enintään 4 kaarta, jotka voidaan tallentaa yhteen kokonaislukuun. Tämän arrayn tilavaativuus on siis $O(|cV|) = O(|V|)$, missä c on vakio.
- Aikavaativuus $O(|V| \log |V|)$. Pahimmassa tapauksessa kaikki kaaret on käytävä läpi, mutta koska kaarien lukumäärä on solujen lukumäärä kerrottuna vakiolla mille tahansa labyrintin koolle, on tämän aikavaativuus $O(|V|)$. Joukkojen yhdistely on keskimäärin $O(\log |V|)$. Alla on analyysiä joukkojen yhdistelystä.

Kun kaksi joukkoa yhdistetään, haetaan `getRoot()`-operaatiolla kahden joukon juuret, ja keskimäärin tämä on hyvin nopea $O(\log |V|)$ -operaatio. Koska `saveVertice()`-operaatio siivoilee solmuja ja kaaria koko ajan (ja kutsuu samalla `getRoot():ia`), ei ole mahdollista päästä (edes erittäin epätodennäköiseen) tilanteeseen, missä joukkojen yhdistelyn vaatimus olisi $O(1)$ tai $O(|V|)$.

Lisää tulossa.

[3] Tämäkin on siis oikeastaan *modifioitu* Kruskalin algoritmi.

2.3. Labyrintin ratkoja-algoritmit

Oikean käden sääntö. Tämä ratkoja aloittaa lähtösolusta ja lähtee etenemään labyrintissä "oikeaa kättä seinässä kiinni pitäen" niin kauan, kunnes ratkaisu löytyy.

- Tilavaativuus $O(1)$. Algoritmi tarvitsee tiedon vain nykyisestä koordinaatista.*
- Aikavaativuus $O(|V|)$. Algoritmi käy pahimassa tapauksessa koko labyrintin läpi.

* Vierailtujen solujen laskemiseksi tarvitsee version, jossa ylläpidetään visited-arraytä. Tällöin tilavaativuus on visited-arrayn koko, mikä on labyrintin koko $O(|V|)$. Molemmat versiot algoritmista on implementoitu koodiin.

Syvyysuuntainen haku. Etsii maalia syvyysuuntaisella haulla siten, että kunkin solun naapurit tallennetaan pinoon satunnaisessa järjetyksessä.

- Tilavaativuus $O(|V|)$. Syvyysshaun pino, joka on pahimassa tapauksessa koko labyrintin kokoinen.
- Aikavaativuus $O(|V|)$. Syvyysuuntainen haku vie tunnetusti $O(|E|)$ aikaa, mutta koska kaarien lukumäärä on solujen lukumäärä vakiolla kerrattuna, $O(|E|) = O(|cV|) = O(|V|)$.

Leveysuuntainen haku. Etsii maalia leveysuuntaisella haulla siten, että kunkin solun naapurit tallennetaan jonoon satunnaisessa järjetyksessä.

- Tilavaativuus $O(|V|)$. Leveyshaun jono ei voi paisua yhtä isoksi kuin syvyysshaun pino, ja pysyykin keskimäärin aika pienenä. Mutta tämäkin on (vaikkakin pienellä) vakiolla kerrottuna labyrintin koko, eli $O(|cV|) = O(|V|)$.
- Aikavaativuus $O(|V|)$. Leveysuuntainen haku vie tunnetusti $O(|E|)$ aikaa, mutta koska kaarien lukumäärä on solujen lukumäärä vakiolla kerrattuna, $O(|E|) = O(|cV|) = O(|V|)$.

Lisää tulossa.