Määrittelydokumentti

Labyrintingeneroija ja –ratkoja

Tietorakenteet ja algoritmit harjoitustyö

Juri Kuronen

Kesä 2014

Sisältö

1. Johdanto

Labyrintin generoimiseen ja ratkaisemiseen on olemassa lukuisia algoritmeja. Tässä työssä toteutan näistä yleisimpiä ja vertailen niiden suoritusaikoja. 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 x 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 maalisolu asetetaan ristikon vastakkaisille sivuille. Lisäksi oletetaan, että mistä tahansa labyrintin pisteestä A on vain ja ainoastaan yksi reitti mihin tahansa pisteeseen B. Tämä tarkoittaa, että labyrintti on syklitön 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 muutella käyttäjän syötteellä.

2. Käytetyt algoritmit ja tietorakenteet

2.1. Tietorakenteet

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

Lista. Lorem ipsum...

Pino. Lorem ipsum...

Joukko (Kruskalin algoritmi). Lorem ipsum...

2.2. Labyrintin generointialgoritmit

Käyttäjä syöttää labyrintin koon N x M. Labyrintti alustetaan kaksiulotteisena N x 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 edellisen solun pinon päällimmäiseksi. Lisäksi käytössä on N x 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 x M -kokoinen array labyrintin osana olevista soluista.

Aluksi merkataan lähtösolu osaksi labyrinttiä ja kaikki lähtösolun viereiset solut, jotka eivät vielä ole osana labyrinttia, listaan soluista, jotka voidaan seuraavaksi liittää labyrinttiin. Tämän jälkeen valitaan satunnaisesti yksi solu 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|). Aluksi meillä on labyrintin koon verran joukkoja, sillä meillä on näin monta solua. 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|). Erittäin epätodennäköisessä tilanteessa joukko-operaatio getRoot() voisi viedä O(|V|) aikaa, mutta keskimäärin tämä tekee vain muutaman operaation O(log |V|) ja tekee samalla tulevista getRoot()-operaatioista nopeampia.

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 pahimmassa tapauksessa koko labyrintin läpi.

Syvyyssuuntainen haku. Etsii maalia syvyyssuuntaisella haulla siten, että kunkin solun naapurit tallennetaan pinoon satunnaisessa järjetyksessä.

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

Leveyssuuntainen haku. Lyhyt kuvaus, O(n):t.

Lisää tulossa. Varsinkin monimutkaisempia algoritmeja, jotka ratkaisevat labyrintin kuin labyrintin ilman helpottavia oletuksia.