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 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 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 valmiita tietorakenteita ei käytetä.

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

2. Työssä käytetyt tietorakenteet

Työssä on toteutettu kaikille tietotyypeille *lista*, *pino* sekä *jono*. Lisäksi aputietorakenteena on käytössä prioriteettikeko-, *puu*- ja *joukkotietorakenteet*. Tietorakenteiden toimintojen kuvaukset olettavat, että lukija tietää, kuinka tietorakenteiden tulisi käytännössä toimia. (Tietorakenteet ja algoritmit –kurssin tiedot.)

2.1. Lista

Tieto tallennettu järjestämättömänä *Object-arrayhun*. Lista tukee kaikkia tietotyyppejä. Listan operaatiot lukevat myös kahta kokonaislukua: *listan koko* ja *Object-arrayn (maksimi)koko*.

Listaan on toteutettu seuraavat toiminnot:

Toiminto	Kuvaus	Tila- ja aikavaativuus.
Konstruktori	Alustaa Object-arrayn. Tahdottaessa voi antaa Object-arrayn alkukoon. Listan kooksi asetetaan 0.	T & A: O(1).
add(item)	Asetetaan *item* Object-arrayn kohtaan *listan koko*. Lisätään listan kokoa yhdellä. Tämä operaatio asetti uuden *item*:in listan hännille. Jos listan koko saavutti Object-arrayn koon, luodaan uusi, kaksi kertaa isompi Object-array ja kopioidaan listan sisältö siihen.	T&A: Object-arrayn päivitys on harvinainen O(n)-operaatio, joten amortisoidusti tila- ja aikavaativuus O(1).
removeLast()	Jos lista ei ole tyhjä, vähentää listan kokoa yhdellä. Tämä operaatio poistaa listan viimeisimmän alkion. Viimeisin alkio myös palautetaan.	T & A: O(1).
removeByIndex(key)	Vaihtaa swap()-operaatiolla kohdan *key* ja listan viimeisimmän alkion paikan. Tämän jälkeen kutsuu *removeLast()*-operaatiota.	T & A: O(1).
removeByValue(item)	Etsii listasta *item*:in. Jos se löytyy, kutsuu *item*:in indeksillä *removeByIndex()*-operaatiota.	T: O(1). A: O(n), koko lista saatetaan joutua käydä läpi.
get(key)	Palauttaa listan *key* kohdassa olevan alkion.	T & A: O(1).
empty()	Tarkistaa onko listan koko 0 ja palauttaa vastaavan totuusarvon.	T & A: O(1).
size()	Palauttaa listan koon.	T & A: O(1).
join(List)	Yhdistää annetun listan tiedon tähän listaan kutsumalla add(item)-operaatiota annetun listan jokaiselle alkiolle.	T: Amortisoidusti O(1). (Object-arrayn päivitys.) A: O(n), sillä annettu lista on käytävä läpi kokonaan.
swap(index1, index2)	Vaihtaa annetun kahden alkion paikkaa listan Object-arrayssa.	T & A: O(1).
reverseList()	Muuttaa listan alkioiden järjestyksen päinvastaiseksi.	T: O(1). A: O(n), lista käytävä läpi.

2.2. Pino

Pinoon on tallennettu tieto *pinon koosta* sekä *päällimmäisestä pinoalkiosta*. *Pinoalkio* on aputietorakenne, jossa on tallennettuna pinoalkion *tieto* (mitä tahansa tietotyyppiä) sekä mahdollinen *seuraava pinoalkio*.

Pinoon on toteutettu seuraavat toiminnot:

Toiminto	Kuvaus	Tila- ja aikavaativuus.
Konstruktori	Asettaa pinon kooksi 0.	T & A: O(1).
push(data)	Luo uuden pinoalkion, jonka tiedoksi annetaan *data*. Asettaa pinoalkion uudeksi *päällimmäiseksi pinoalkioksi* ja kasvattaa pinon kokoa yhdellä.	T & A: O(1).
pop()	Jos pino ei ole tyhjä, palauttaa *pinon päällimmäisen alkion* ja vähentää pinon kokoa yhdellä.	T & A: O(1).
empty()	Tarkistaa onko pinon koko 0 ja palauttaa vastaavan totuusarvon.	T & A: O(1).
size()	Palauttaa pinon koon.	T & A: O(1).

2.3. Jono

Tieto tallennetaan Object-arrayhun. Jono tukee kaikkia tietotyyppejä. Kokonaislukuina on tallennettu jonon *pään* ja *hännän* kohdat, sekä *Object-arrayn (maksimi)koko.*

Jonoon on toteutettu seuraavat toiminnot:

Toiminto	Kuvaus	Tila- ja aikavaativuus.
Konstruktori	Alustaa Object-arrayn. Tahdottaessa voi antaa Object-arrayn alkukoon. *Pään* ja *hännän* kohdiksi 0.	T & A: O(1).
enqueue(item)	Asetetaan *item* Object-arrayn kohtaan *jonon häntä*. Siirretään *hännän indeksiä* yhdellä eteenpäin. Tämä operaatio asetti uuden *item*:in jonon hännille. Jos jonon koko saavutti Object-arrayn koon (*häntä* saavutti *pään*), luodaan uusi, kaksi kertaa isompi Object-array ja kopioidaan jonon sisältö siihen.	T&A: Object-arrayn päivitys on harvinainen O(n)-operaatio, joten amortisoidusti tila- ja aikavaativuus O(1).
dequeue()	Jos jono ei ole tyhjä, palauttaa jonon *päässä* olevan alkion. Siirretään *pään indeksiä* yhdellä eteenpäin.	T & A: O(1).
get(key)	Palauttaa *Object-arrayn* kohdan *key* alkion.	T & A: O(1).
empty()	Tarkistaa onko jono tyhjä ja palauttaa vastaavan totuusarvon.	T & A: O(1).
size()	Palauttaa jonon koon.	T & A: O(1).

2.4. Prioriteettikeko ja kekoalkio

A-hakualgoritmi* käyttää prioriteettikekoa, joten *kekoalkio* on suunniteltu A*-hakua varten. Kekoalkioon on tallennettuna *koordinaatti, kuljettu matka* ja *kustannusarvio*. Prioriteettikeko toteuttaa minimikeko-ominaisuuden, eli sen, että kunkin *vanhemman* arvo on pienempi kuin *lastensa* arvot. Kekoalkion arvo on *(kuljettu matka + kustannusarvio)*.

Keko toteutetaan arrayta käyttäen siten, että *kekoalkion kohdan 'parent'* lapset ovat *2 * parent + 1* ja *2 * parent + 2* kohdissa. Keon kohdan *'lapsi'* vanhempi on kohdassa *(lapsi + 1) / 2 + 1*.

Prioriteettikekoon on toteutettu seuraavat toiminnot:

Toiminto	Kuvaus	Tila- ja aikavaativuus.
Konstruktori(value)	Alustaa kekoalkio-arrayn. Tahdottaessa voi antaa kekoalkio-arrayn alkukoon. Keon kooksi asetetaan 0.	T & A: O(1).
insert(coordinate, distance, heuristic)	Lisää kekoon uuden kekoalkion. Alkio lisätään keon päähän, jonka jälkeen uutta alkiota vaihdellaan alkuun päin niin kauan, kunnes uudelle alkiolle löydetään paikka, mikä ei riko minimikekoehtoa. Tämän jälkeen päivitetään tarpeen tullen keon aputaulukon koko.	T: Amortisoidusti* O(1). A: Amortisoidusti* O(log n), sillä uusi kekoalkio täytyy tarpeen tullen siirtää keon alkuun. *: Aputaulukon päivitys on harvinainen O(n)-operaatio.
removeMin()	Poistaa ja palauttaa pienimmän kekoalkion. Poiston jälkeen keon viimeinen alkio siirretään alkuun, keon kokoa vähennetään yhdellä, ja alkukohdalle ajetaan heapify(), mikä korjaa minimikekoehdon.	T: O(1). A: O(log n), sillä heapify():n aikavaativuus on O(log n).
heapify(index)	Operaatio korjaa rekursiivisesti kekoehdon, jos se on rikki kohdassa index.	T: O(1). A: O(log n), sillä log n on keon korkeus.
get(key)	Palauttaa keon *key* kohdassa olevan alkion arvon.	T & A: O(1).
empty()	Tarkistaa onko keon koko 0 ja palauttaa vastaavan totuusarvon.	T & A: O(1).
size()	Palauttaa keon koon.	T & A: O(1).
swap(index1, index2)	Vaihtaa annetun kahden alkion paikkaa listan kekoalkio-arrayssa.	T & A: O(1).

2.5. Joukko (aputietorakenne)

Kruskalin algoritmi käyttää aputietorakenteena *joukkoalkiota*. Jokaisella *joukkoalkiolla* on tallennettuna *(final) id, (tähän tallennetun joukon juuren) joukkoalkio* sekä (*joukkoalkioon tallennettu) koko*.

Joukkoalkioihin on toteutettu seuraavat toiminnot:

Toiminto	Kuvaus	Tila- ja aikavaativuus.
Konstruktori(value)	Asettaa *id*:ksi annetun *value*:n. *Kooksi* asetetaan 1 ja juureksi *tämä joukkoalkio itse*.	T & A: O(1).
getRoot()	Hakee rekursiivisesti *tallennettua joukon juurta* kunnes saavutaan *koko joukon juureen*. ”Matkan varrella” päivittelee tallennetut juuret koko joukon juureksi.	T: O(1). A: O(log |V|).*
getId()	Hakee *koko joukon id*:n hakemalla koko joukon juuren *getRoot()*:lla.	T: O(1). A: O(log |V|).*
getNumOfElements()	Hakee *koko joukon koon* hakemalla koko joukon juuren getRoot():lla.	T: O(1). A: O(log |V|).*
joinTwoSets(set2)	Yhdistää kaksi joukkoa siten, että hakee kummankin joukon juuret getRoot():lla, laskee yhteen joukkojen koot ja lopuksi asettaa annetun joukon juureksi tämän joukon juuren.	T: O(1). A: O(log |V|).*

(*) Analyysia joukkojen yhdistämisestä ja *getRoot()*:ista: Kun kaksi joukkoa yhdistetään, haetaan getRoot()-operaatiolla kahden joukon juuret, ja keskimäärin tämä on hyvin nopea O(log |V|)-operaatio. Koska *Kruskalin algoritmin saveVertice()*-operaatio siivoilee solmuja ja kaaria koko ajan (ja kutsuu samalla *getRoot()*:ia), ei ole mahdollista päästä (edes erittäin epätodennäköiseen) tilanteeseen, missä joukkojen yhdistelyn vaativuus olisi O(|V|).

2.6. Puu (aputietorakenne)

Labyrintin ratkonta-algoritmien *findPath()*-operaatio käyttää aputietorakenteena *puualkiota*. Puualkioon on tallennettuna *vanhempi* sekä *(tämän, labyrintin jotain kohtaa kuvaavan, puualkion) koordinaatti (labyrintissä).* Konstruktori asettaa vanhemman ja koordinaatin.

Lisätietoa löytyy kohdasta 3.2. Labyrintin ratkonta-algoritmit.

3. Työssä käytetyt algoritmit

Työssä on käytetty lukuisia eri algoritmeja labyrintin generoimiseen ja ratkomiseen.

3.1. Labyrintin generointialgoritmit

Käyttäjä on valinnut labyrintin koon *N x M*. Labyrintti on alustettu kaksiulotteisena *N x M byte-arraynä*, jossa ei ole vielä ollenkaan kaaria. Generoivat algoritmit täyttävät byte-arrayn kohdat siten, että joka kohdassa on tieto mihin suuntiin tästä kohdasta on reitti.

Suunnat tallennetaan ja luetaan bitti-operaatioilla. Pohjoissuunta vastaa kohdassa 2^0 olevaa bittiä, itäsuunta 2^1, eteläsuunta 2^2 ja länsisuunta kohdassa 2^3 olevaa bittiä. Esimerkiksi 00000101 tarkoittaisi, että reitit suuntiin pohjoinen ja etelä ovat olemassa.

3.1.1. Rekursiivinen peruuttava haku

Käyttää pinoa ja laittaa aina käsitellyn 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).

3.1.2. 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 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. Listan operaatiot amortisoidusti O(1). (Itseasiassa käytännössä O(1), sillä listaan varataan aluksi tarpeeksi tilaa.)

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

3.1.3. 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 vie keskimäärin O(log |V|), ja joukkojen yhdistelyjä suoritetaan solujen lukumäärään verrannollinen määrä. (Analyysi joukkojen yhdistelystä löytyy kohdasta *2.4. Joukko (aputietorakenne)*.)

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

3.2. Labyrintin ratkonta-algoritmit

Työssä on toteutettu kolmen eri tyypin ratkonta-algoritmeja. ”Oikean käden sääntö” on yksinkertaisin ohjein varustettu ratkoja. ”Syvyyssuuntainen- ja leveyssuuntainen haku” ovat satunnaistettuja ratkojia. ”A*-haku” on älykäs ratkoja-algoritmi.

Lisäksi on toteutettu yleinen findPath()-algoritmi joka labyrintin ratkonta-algoritmin työn jälkeen kasaa löydetyn polun maaliin listaksi. findPath() etsii polun maaliin kulkemalla visited-arraytä pitkin. Koordinaatit on tallennettu TreeNodena pinoon siten, että pinossa on aina täsmällisesti juuri nyt kuljettu polku, sillä peruuttaessa visited-arrayssä pinosta poistuu peruutetut alkiot. Kun löydetään maali, pinon sisältö tyhjennetään polun listaan.

3.2.1. 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.

* 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.

3.2.2. 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|).

3.2.3. Leveyssuuntainen haku

Etsii maalia leveyssuuntaisella haulla siten, että kunkin solun naapurit tallennetaan jonoon satunnaisessa järjetyksessä.

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

3.2.4. A*-hakualgoritmi

Algoritmi alustaa visited-arrayn sekä prioriteettikeon, ja kutsuu rekursiivista findGoal()-metodia lähtösolun koordinaatille. Kullakin kutsulla asetetaan tutkittu koordinaatti vierailluksi ja päivitetään prioriteettikekoon solun naapurit.

Naapurit tallennetaan kekoalkioina, joissa on tietona *koordinaatti, kuljettu matka* ja *kustannusarvio*. Kustannusarvio on 1,1:llä skaalattu *manhattanin etäisyys* solusta maalisoluun. 1,1:n skaalauksella algoritmi käy ensin läpi soluja, joihin pääsemiseksi on kuljettu pitkä matka.

Kun käsittelyyn tulee maalikoordinaatti, algoritmi ratkaisi labyrintin.

- Tilavaativuus O(|V|). Visited-arrayn koko on koko labyrintin koko. Pahimmillaan prioriteettikeon koko on koko labyrintin koko.
- Aikavaativuus O(|V|). Pahimmillaan on käytävä kaikki labyrintin solut läpi.

4. Käyttöliittymä

Ohjelmalle on toteutettu yksinkertainen graafinen käyttöliittymä, jonka avulla käyttäjä voi valita labyrintin koon sekä labyrintin generoimiseen ja –ratkaisemiseen käytettävät algoritmit.

5. Lähteet

Think Labyrinth: Maze Algorithms. (http://www.astrolog.org/labyrnth/algrithm.htm)

Maze solving algorithm. (http://en.wikipedia.org/wiki/Maze_solving_algorithm)

Maze generation algorithm. (http://en.wikipedia.org/wiki/Maze_generation_algorithm)

Tietorakenteiden toteuttamisessa otettu mallia Javan vastaavista tietorakenteista. (http://docs.oracle.com/javase/7/docs/)