Tietorakenteiden harjoitustyö
Toteutus

Jukka Koskelin
013865465

Työn aihe: A* -reitinhakualgoritmia käyttävä ohjelma joka muodostaa kaksiulotteisen kartan ja etsii reitin kahden annetun pisteen välillä.

Ohjelman yleisrakenne

Ohjelma koostuu seuraavista luokista:

- Main, jossa “käyttöliittymä”.
- Kartta, joka sisältää kartan perustamiseen liittyvän logiikan, sekä polun etsinnän kutsun ja logiikan eri tietorakenteita käyttävien Laskentojen valitsemiseen
- KuvaLataaja, joka lataa BufferedImage -kuvan sekä piirtää siihen polun
- Noodi, joka vastaa yhtä kartan pistettä
- Keko, joka sisältää kekototeutuksen Noodeilla
- AvlPuu ja PuuSolmu, jotka sisältävät Avl-puun toteutuksen sekä Noodin kapseloivan PuuSolmun
- Pino, joka sisältää yksinkertaisen pinon toteutuksen Noodeille.
- Kolme Laskenta-luokkaa jotka sisältävät identtisen rakenteen kolmelle eri tietorakenteelle.
- Viisi kappaletta yksikkö- ja integraatiotestiluokkia.

Ohjelma toimii niin että ensin luodaan Kartta -instanssi joko annetun korkeuden ja leveyden mukaan, tai syötetään kuvatiedoston nimi. Kartta luo ja populoi itsensä Noodeilla, käyttäen tarvitaessa KuvaLataajaa. Tämän jälkeen Kartta-instanssin etsiPolku -metodia kutsutaan alku- ja loppupisteillä, sekä valitun tietorakenteen nimellä, Keon ollessa oletusarvo.

Kartta kutsuu valittua tietorakennetta käyttävää Laskenta-luokkaa, joka toteuttaa A-star -algoritmin ja palauttaa löydetyn polun. Jos Kartta muodostettiin kuvasta, Kartta pyytää vielä KuvaLataajaa piirtämään polun annetun kartta-tiedoston päälle.

Saavutetut aika- ja tilavaativuudet (m.m. O-analyysi pseudokoodista)

```
while (!valmis) {
    /** Haetaan läpikäymättömistä noodeista lyhimmän matka-arvion omaava */
    valittu = (T)lyhinMatkaLapikaymattomissa(); //Vakio-aikainen, haetaan keon pienin alkio
    /** Lisätään noodi läpikäytyihin ja poistetaan -käymättömistä */
    kaydytNoodit.add(valittu);  //Vakio
    kaymattomatNoodit.poistaPienin(); //Kekojärjestäminen, O(nlogn)


    /** Onko piste loppupiste? */
    if ((valittu.getxPositio() == loppuX)
        && (valittu.getyPositio() == loppuY)) {
```

```
            return laskePolku(noodit[alkuX][alkuY], valittu); //Silmukka, joka käy pahimmillaan läpi jokaisen noodin, |E|
        }

        // Käytään läpi kaikki viereiset Noodit
        List<T> viereisetNoodit = getViereinen(valittu);
        for (int i = 0; i < viereisetNoodit.size(); i++) { //Käydään läpi pahimmillaan jokaisen noodin jokainen viereinen
noodi, joita on aina neljä kappaletta. Eli… |E|x4? Eli |E|.
            T valittuViereinen = viereisetNoodit.get(i);
            if (!kaymattomatNoodit.sisaltaakoSaman(valittuViereinen)) {
                asetaKaymatonNoodi(kaymattomatNoodit, valittuViereinen, valittu, loppuX, loppuY);Vakio
            } else {
                /** Jos matka käsittelyssä olevan Noodin kautta on lyhyempi kuin Noodiin aiemmin tallennettu matka,
                 aseta käsittelyssä oleva Noodi edeltäjäksi ja päivitä matkaa */
                asetaUusiLyhinMatka(valittuViereinen, valittu); //Vakio
            }
        }
```

Voitanee siis todeta että pahimmassa tapauksessa aikavaativuus on hyvin lähellä Dijkstran algoritmin aikavaativuutta, joka on *O((|E| + |V|)log|V|)*, kun käytetään toteutukseen kekoa jonka operaatioiden aikavaativuus on *O(N log N).*

Avl-puun korkeus on *O(log N)*, eli periaatteessa senkin aikavaativuus algoritmin käytössä on *O(N log N)*. Testauksessahan suurilla kuvilla ja esteillä Avl-puu toimi lähes puolet hitaammin kuin Keko, joten voisi kuvitella ettei sen toteutus ole ihan täydellinen.

Tilavaativuuden puolesta varsinainen kartta sisältää taulukon jossa on alkioita kartan korkeus x pituus, läpikäytyjen noodien listassa on pahimmillaan alkioita taulukon koon verran, läpikäymättömien noodien listassa on kerrallaan neljä kappaletta noodeja, ja palautettavassa polussa on pahimmillaan noodeja jonkin verran vähemmän kuin koko taulussa (ainakaan ei heti tule mieleen reittiä joka kävisi läpi kaikki Noodit, ellei taulu ole todella pieni).

Suorituskyky- ja O-analyysivertailu (mikäli työ vertailupainotteinen)

Kts. testausdokumentin kohta “Suorituskykytestaus”.

Työn mahdolliset puutteet ja parannusehdotukset

Kts. testausdokumentin kohta “Puutteet”.
Parannusehdotuksista päälimmäisinä tulevat mieleen ohjelman rakenteen järkeistäminen niin että laskennan suorittavat luokat voidaan yleistää niin että logiikka toteutetaan vain kerran ja käytettävä tietorakenne parametroidaan sen käytettäväksi. Rajapintojahan sovellus ei käytä ollenkaan, niiden avulla ratkaistaisiin monta ongelmaa.

Yksikkötestit ovat nyt hieman tyhmissä namespaceissa, lähinnä että voivat testata tiettyjä protected -metodeja. Yksikkötestejä pitäisi olla yhtä paljon kuin toteutettuja luokkia, nyt on vähän

oiottu ja laitettu useamman luokan testejä samaan tiedostoon. Integraatio- ja suorituskykytestit pitäisi eriyttää omiin luokkiinsa. Yksikkötestien koodikattavuus on välttävä eivätkä ne juuri testaa erikoistapauksia. Lisäksi tajusin tätä kirjoittaessa että ne testaavat vain ja ainoastaan neliön muotoisia karttoja.