Tampereen yliopisto

Juuso Rantanen

KRUSKALIN JA PRIMIN ALGORITMIEN VERTAILU PIENIMMÄN VIRITTÄVÄN PUUN ETSIMISESSÄ

Informaatioteknologian ja viestinnän tiedekunta (ITC)
Kandidaatintyö
Huhtikuu 2019

TIIVISTELMÄ

Juuso Rantanen: Kruskalin ja Primin algoritmien vertailu pienimmän virittävän puun etsimisessä
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan koulutusohjelma
Huhtikuu 2019

Graafiteoria on matematiikan osa-alue, jota hyödynnetään laajasti useilla muilla tieteenaloilla. Virittävät puut ovat osa graafiteoriaa. Virittäviä puita käytetään usein suunnittelun apuna, sillä reaalimaailman asioita voidaan monesti mallintaa graafien avulla. Pienimpiä virittäviä puita etsitään graafeista algoritmien avulla ja niiden löytämiseen on tarjolla useita eri algoritmeja.

Tässä työssä tutkittiin ja vertailtiin kahta pienimmän virittävän puun etsimiseen käytettyä Kruskalin algoritmia ja Primin algoritmia. Ohjelmoitujen algoritmien tehokkuuksia sekä toteutuksia tutkittiin ja vertailtiin toisiinsa tarkoituksena selvittää niiden tehokkuutta pienintä virittävää puuta etsittäessä eri kokoisissa graafeissa.

Molempien algoritmitien asymptoottinen tehokkuus vastasi odotettua asymptoottista tehokkuutta. Lisäksi molemmat algoritmit olivat mahdollisia toteuttaa käyttämällä C++-ohjelmointikielen standardikirjastoa ja standardikirjaston sisältämiä tietorakenteita. Tehokkuuksia vertailtaessa Kruskalin algoritmi oli tehokkaampi kuin Primin algoritmi kaikissa tilanteissa. Lisäksi Kruskalin algoritmin optimaalinen toteuttaminen oli helpompaa kuin Primin, joka olisi vaatinut joko kolmannen osapuolen kirjastoja tai oman prioriteettijonon toteuttamista.

Avainsanat: algoritmi, Kruskal, Prim, pienin virittävä puu

SISÄLLYSLUETTELO

1. JOHDANTO ... 1
2. GRAAFI ... 2
2.1 Graafin rakenne ... 3
2.2 Esitystavat tietokoneella ... 4
3. PUURAKENTEET ... 6
3.1 Virittävät puut ... 7
3.2 Pienimmän virittävän puun algoritmit ... 8
3.2.1 Primin algoritmi ... 8
3.2.2 Kruskalin algoritmi ... 9
4. ALGORITMIEN VERTAILU ... 11
4.1 Primin testaaminen ... 11
4.1.1 Tulokset ... 13
4.2 Kruskalin testaaminen ... 14
4.2.1 Tulokset ... 15
4.3 Vertailu ... 16
5. YHTEENVETO ... 18
LÄHTEET ... 19

LIITE A: TESTITULOKSIEN TAULUKOT

LIITE B: VERTEX-DATATYYPPI

LIITE C: SATUNNAISGRAAFIN KASVATUSALGORITMI

LYHENTEET JA MERKINNÄT

Decrease-key	Prioriteettijonon ominaisuus, jossa alkion muuttuessa prioriteettijono päivitetään.
O	O-notaatio eli asymptoottinen yläraja
Θ	theta-notaatio eli asymptoottisesti ylhäältä ja alhaalta rajoitettu

1. JOHDANTO

Graafiteoria on matematiikan osa-alue, jota hyödynnetään kaikilla tieteenaloilla. Graafien avulla voidaan mallintaa lukuisia reaalimaailman asioita. Esimerkiksi tieverkosto on helppo mallintaa graafiksi ja graafialgoritmeja hyödyntäen voitaisiin tieverkkoa optimoida, etsiä lyhimpiä reittejä paikasta toiseen tai tutkia mistä kaupungista pääsee suoraan toiseen.

Pienimmät virittävät puut ja niiden etsimiseen käytetyt algoritmit ovat iso osa graafiteoriaa. Ne ovat tärkeä osa monia suunnitteluprosesseja, joissa tarvitaan yhteyksien optimointia ja niiden tutkimista. Tietoliikennetekniikka on yksi isoista insinöörialoista, jossa pienimpien virittävien puiden etsiminen on tärkeä osa tietoverkkojen optimointia.

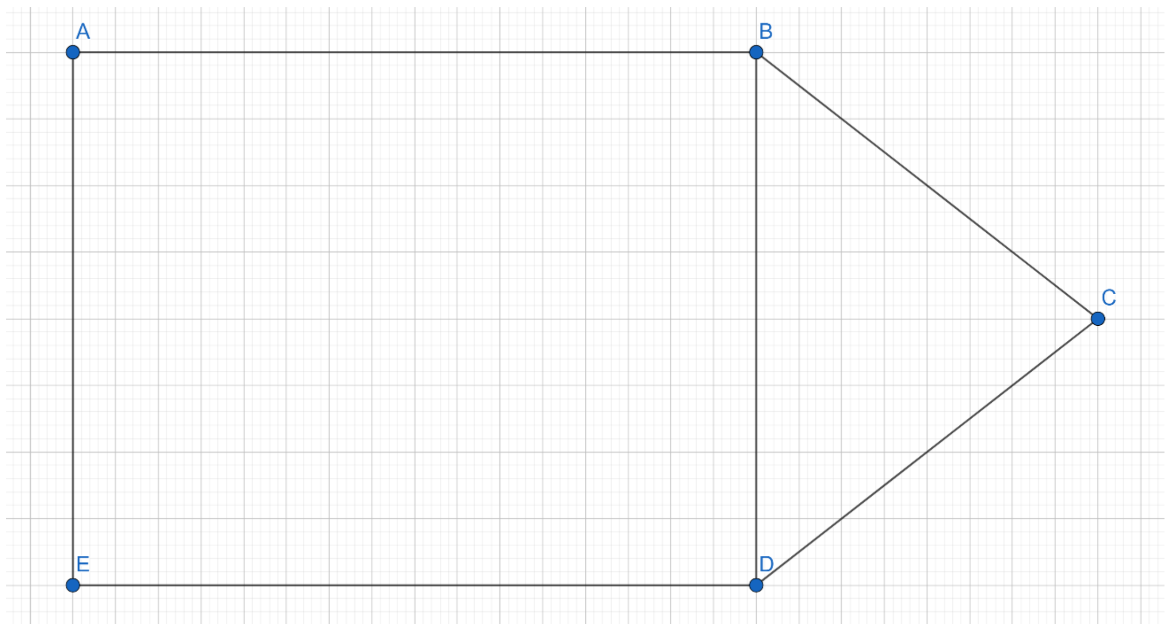
Työn tarkoituksena on tutkia ja vertailla kahta klassista pienimmän virittävän puun etsimiseen käytettyä algoritmia, jotka ovat Kruskalin ja Primin algoritmit. Tarkoituksena on tutkia pääsevätkö itse ohjelmoidut algoritmien toteutukset lähelle niille annettuja teoreettisia asymptoottisia tehokkuuksia. Lisäksi tarkoituksena on suorittaa vertailua algoritmien välillä.

Luvussa 2 käsitellään graafeja yleisesti ja tutustutaan niiden teoriaan ja esittämiseen tietokoneympäristössä. Seuraavaksi luvussa 3 käsitellään puurakenteita ja virittäviä puita. Lisäksi luvussa esitellään yleisesti puun kasvatusalgoritmit ja perehdytään tarkemmin Primin ja Kruskalin algoritmeihin. Neljännessä luvussa on Kruskalin ja Primin algoritmien tutkimista ja vertailua itse ohjelmoitujen algoritmien avulla saatujen tulosten perusteella. Viimeisessä luvussa 5 on yhteenveto saaduista tuloksista. Liitteessä A on taulukoita, jotka sisältävät algoritmien testauksessa saadut tulokset, liitteessä B on vertex-datatyypin toteutus ja liitteessä C satunnaisgraafin rakennusalgoritmin toteutus.

2. GRAAFI

Graafeja kuvataan yleensä joukkoina pisteitä, jotka yhdistetään toisiinsa joukolla viivoja. Graafin sisältämiä pisteitä kutsutaan solmuiksi ja solmuja yhdistäviä viivoja kaariksi [1]. Esimerkki yksinkertaisesta graafista on sukupuu, jossa henkilöt ovat sukupuun solmuja ja lapset sekä vanhemmat yhdistävät viivat ovat graafin kaaria. Graafeja hyödynnetään usein siten, että niiden sisältämät solmut ja kaaret kuvaavat jotain reaalimaailman konseptia.

Matemaattisesti graafeja kuvataan yleensä kaavalla graafi G = (V, E), jossa V on graafin solmujen joukko ja E solmujen välisten kaarien joukko. Nämä joukot parina muodostavat graafin G. Yksinään solmujen ja kaarien joukkoa voidaan myös kuvata merkitsemällä niitä V(G) ja E(G). Lisäksi graafissa olevien solmujen määrä ilmoitetaan kirjaimella n. [1] Esimerkiksi graafi **kuvassa 1** sisältää viisi solmua eli n = 5 ja kuusi kaarta solmujen välillä. Graafi voidaan kuvata joukkoina V(G) = {A, B, C, D, E} ja E(G) = {AB, AE, BC, BD CD, DE}. Jos solmujen välillä on vain yksi kaari, voidaan kaari kuvata sen päätepisteiden avulla, mutta useampien kaarien yhdistäessä samat solmut voidaan kaarille antaa omat nimet [1].



Kuva 1. *Yksinkertainen graafi.*

Solmujen ollessa kytkettynä toisiinsa kaarella kutsutaan kyseisiä solmuja rinnakkaisiksi solmuiksi. Tosin jos solmuilla ei ole yhdistävää kaarta, kutsutaan niitä irtonaisiksi. Vastaavia nimityksiä käytetään myös kaarien kohdalla, jolloin kaksi kaarta ovat rinnakkaisia,

jos niillä on yhteinen solmu. Solmun rinnakkaisten solmujen määrä kertoo myös solmun kulman, jota merkataan d:llä. [2] Esimerkiksi solmulla, josta lähtee kolme kaarta, on asteluku d = 3.

Graafin kaarille voidaan asettaa ominaisuuksia, kuten hinta tai suunta. Kaaren hinta kuvaa kyseisen kaaren hintaa graafissa. Hintana voi olla mielivaltaisesti asetettu arvo, tai asetettu hinta voi kuvata jotain reaalimaailman arvoa, kuten matkaa tai aikaa. Kaaret voivat olla myös suunnattuja, jolloin toinen silmukoista asetetaan lähtöpisteeksi ja toinen päätepisteeksi. Suunnattuja kaaria voidaan kulkea vain lähtöpisteestä päätepisteeseen. [1]

Graafin sisällä tarvitsee usein kulkea solmusta toiseen, jolloin voidaan käyttää polkuja. Polku on kulku graafissa, mikä voi sisältää saman solmun tai kaaren vain yhden kerran, lukuun ottamatta polun päätepisteitä. Jos polun alku- ja loppusolmu on sama, on polku suljettu. Polulle voidaan myös laskea kokonaishinta laskemalla polun kaarien hinnat yhteen. [3]

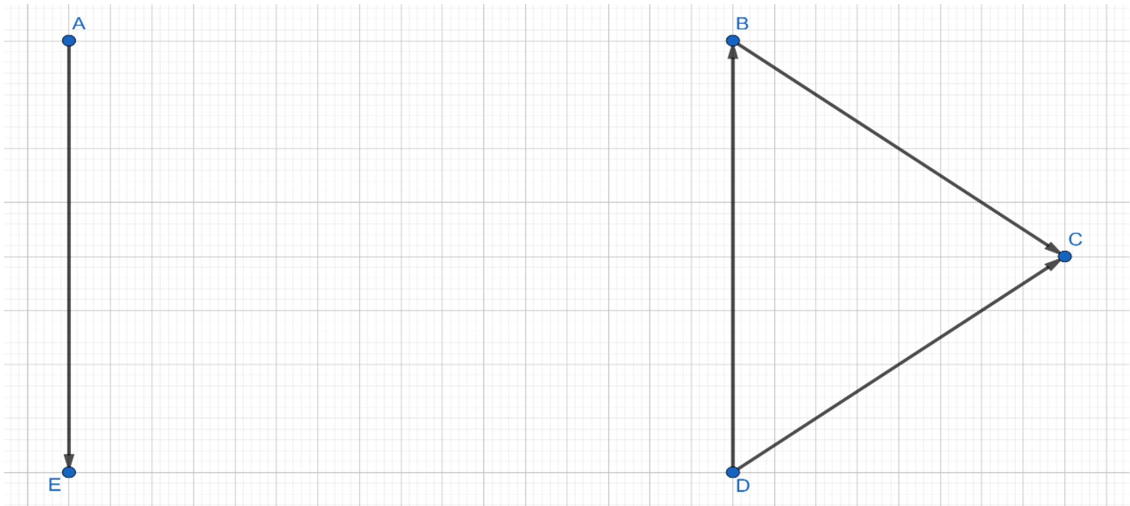
2.1 Graafin rakenne

Graafeja voidaan jakaa erilaisiin luokkiin graafien ominaisuuksien ja määrittelyiden mukaan. Yleisessä graafissa ei ole annettu mitään rajoitteita kaarille tai solmuille, mutta usein käyttökohteen mukaan on hyödyllisempää käyttää tiettyä tarkoituksenmukaisia graafiluokkia tai niiden yhdistelmää. [1]

Jos graafissa ei ole ollenkaan silmukoita eli kaaria, jotka yhdistävät solmun itseensä, on graafi joko yksinkertainen tai monigraafi. Yksinkertaisessa graafissa solmuparin välillä ei voi olla vierekkäisiä kaaria, vaan solmuparin voi yksinkertaisessa graafissa yhdistää vain yksi kaari. Kun graafissa on saman solmuparin välillä useampia kaaria, kutsutaan graafia monigraafiksi. [1]

Graafeja voidaan luokitella myös niiden komponenttien määrän mukaan. Graafin komponentteja voidaan merkitä joukolla c(G) [1]. Graafi, joka koostuu yhdestä ainoasta komponentista, on yhdistetty graafi. Yhdistetyssä graafissa jokainen solmu on yhteydessä kaikkiin muihin solmuihin vähintään yhdellä polulla. [2]

Jos graafi sisältää solmuja, joiden välille ei ole mahdollista löytää polkua, on graafi yhdistämätön. Yhdistämättömissä graafeissa on toisistaan irrallisia komponentteja, jotka eivät ole yhteydessä toisiinsa yhdelläkään polulla. Näin ollen ne koostuvat useista yhdistetyistä komponenteista.[2] Esimerkiksi **kuvan 2** graafi koostuu kahdesta komponentista, jotka ovat irrallaan toisistaan.



Kuva 2. *Kahdesta komponentista koostuva suunnattu graafi.*

Graafeja voidaan luokitella myös kaarille annettujen ominaisuuksien perusteella. Esimerkiksi jos graafin kaikille kaarille on annettu suunta, kuten **kuvan 2** graafissa, on graafi suunnattu graafi. Graafit voivat olla myös suuntaamattomia tai osittain suunnattuja. [1]

2.2 Esitystavat tietokoneella

Graafeja käsitellään nykyisin pääasiassa tietokoneiden avulla, sillä ne sisältävät useimmiten niin paljon solmuja ja kaaria, että niiden käsittely piirtämällä tai matemaattisten joukkojen avulla on lähes mahdotonta. Tietokoneille graafit tallennetaan muistiin joko kytkentälistoina tai -matriiseina. [2]

Kytkentälistassa graafin jokaiselle solmulle tallennetaan lista sille rinnakkaisista solmuista. Kun nämä listat yhdistetään yhdeksi joukoksi, saadaan kokonainen kytkentälista. [2] Kytkentälistan voi toteuttaa luomalla listan listoja, jossa jokaisella ylemmän listan indeksillä oleva lista kuvastaa kyseisen solmun rinnakkaisia solmuja. Jos graafi on painotettu, voidaan rinnakkaisen solmun yhdistävän kaaren paino tallentaa parina solmun kanssa listaan. Olio-ohjelmointikielillä toinen mahdollinen tapa on toteuttaa solmuista oma luokka, joka sisältää kaikki kyseisen solmun tiedot kuten sen rinnakkaiset solmut. [4]

Kytkentämatriisissa graafista luodaan matriisi, jossa jokaiselle graafin solmulle on oma sarake ja rivi. Jos solmu V_i on rinnakkainen solmun V_j kanssa, asetetaan matriisin alkion a_{ij} arvoksi 1. Jos solmut eivät ole rinnakkaisia asetetaan niiden leikkauskohdan alkion arvoksi 0. [2] Painotetuissa graafeissa voidaan totuusarvon sijaan tallentaa kyseisen kaaren paino matriisin kohtaan, ja kaarille joita ei ole, voidaan alkion arvoksi tallentaa nolla tai ääretön riippuen mikä on käytännöllisintä. Kytkentämatriisi voidaan toteuttaa luomalla lista, joka sisältää listoja, joihin tallennetaan totuusarvomuuttuja. Tästä muodostuu matriisi, joka sisältää totuusarvomuuttujan tai kaaren hinnan solmujen leikkauskohdassa riippuen siitä ovatko kyseiset solmut rinnakkaiset. [4]

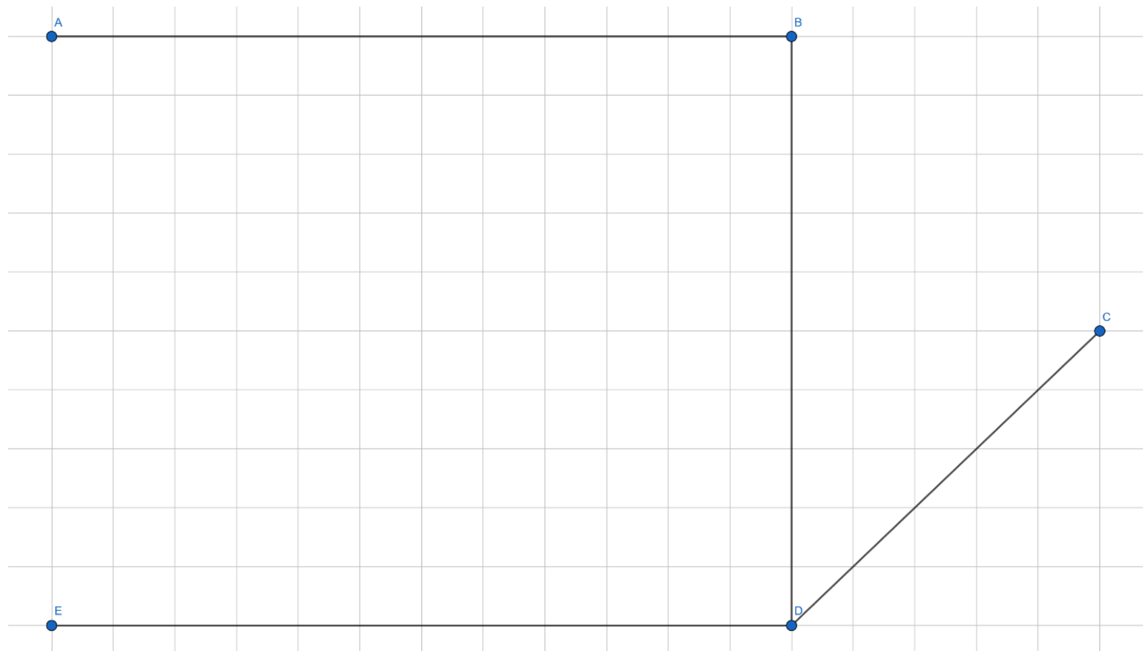
Esitystavan valintaan vaikuttavat käsiteltävien graafien ominaisuudet. Harvoissa graafeissa, joissa kaarien määrä on selkeästi pienempi kuin $E|V^2|$ käytetään yleensä kytkentälistoja, mitkä ovat esitystavaltaan tiiviimpiä. Kytkentälistojen asymptoottinen muistin kulutus on $\Theta(V + E)$, joten harvoissa graafeissa muistin kulutus ei kasva vielä kovin suureksi. Tiheissä graafeissa kytkentämatriisi on varsinkin muistin käytön kannalta selkeästi parempi, sillä sen muistin kulutus on aina $\Theta(V^2)$ riippumatta graafin kaarien määrästä. [4]

3. PUURAKENTEET

Puurakenteet ovat graafiluokka, jolla on omat ominaisuutensa ja käyttökohteensa kuten muillakin graafiluokilla. Puurakenteita käytetään paljon tietorakenteissa ja tiedon prosessoinnissa. Esimerkiksi **binääri-hakupuut** ovat paljon käytettyjä tietorakenteita ohjelmoinnissa. Puurakenteita käytetään myös paljon verkkojen suunnitteluun ja analysointiin ja niitä voidaan pitää yhtenä kulmakivenä optimaalisten verkkojen rakentamisessa. [1]

Puurakenne on yksinkertainen graafi, joka ei sisällä ainoatakaan piiriä eli suljettua polkua. Puut ovat myös aina yhdistettyjä graafeja, josta seuraa että jokaisesta solmusta on täsmälleen yksi polku sen muihin solmuihin. Viimeinen puurakenteelle uniikki ominaisuus on sen kaarien määrä, joka on aina E |n – 1| eli yksi vähemmän kuin sen solmujen määrä. Näistä ominaisuuksista seuraa myös se, että jokainen kaari on irrotusviiva eli kaaren poistaminen aiheuttaa graafin jakautumisen useampaan komponenttiin. Päinvastoin myös jokaista graafiin lisättyä kaarta kohden muodostuu siihen yksi piiri eli suljettu polku. [1]

Esimerkiksi **kuvan 3** graafissa on vain yksi mahdollinen polku solmusta toiseen ja se ei sisällä ainoatakaan suljettua polkua, joten se on puu. Kuvasta on myös helppo huomata, että graafiin lisättäessä kaari minkä tahansa kahden solmun välille syntyy siihen yksi piiri ja poistettaessa graafista kaari jakautuu graafi useampaan komponenttiin.



Kuva 3. Puurakenne.

Jos yhdistämätön graafi koostuu vain komponenteista, jotka ovat puurakenteita kutsutaan sitä silloin metsäksi [3]. Metsän jokainen komponentti sisältää siis kaikki puurakenteen ominaisuudet. Puiden ominaisuuksien perusteella voidaan todeta, että sen kaarien määrä on komponenttien määrä vähennettynä solmujen määrästä eli n – c(G) [1]. Lisäksi jos kaksi eri puukomponenttia yhdistetään toisiinsa kaarella säilyttää yhdistetty komponentti myös kaikki puun ominaisuudet.

3.1 Virittävät puut

Virittävä puu on yhdistetyn graafin aligraafi, joka on puurakenne mihin sisältyvät kaikki graafin solmut. Virittäviä puita kasvatetaan usein graafin solmujen ja kaarien ja varsinkin piirien ja irrotusviivojen prosessointia varten. [1] Esimerkiksi tietoliikennekytkimet laskevat virittävän puun niihin kytketystä verkosta ja sulkevat kytkimen portit, jotka eivät ole osana tätä virittävää puuta.

Virittävien puiden kasvattaminen tapahtuu algoritmien avulla. Puiden kasvattamiseen käytetyt algoritmit ovat ahneita algoritmeja, eli ne valitsevat joka askeleella sen hetkisen parhaan vaihtoehdon. Algoritmit perustuvat siihen, että virittävää puuta rakennetaan yksi kaari ja solmu kerrallaan valitsemalla puun kasvattamiseen parhaiten sopiva solmu. Lisättäväksi solmuksi valitaan joku kasvatettavan puun ulkopuolella oleva solmu, kunnes kasvatettava puu sisältää kaikki graafin solmut. [4]

Esimerkiksi pseudokoodisessa **algoritmissa 1** aluksi alustetaan kasvatettava puu T aloitus-solmuksi v. Tämän jälkeen alustetaan jono, johon lisätään aloitussolmun jälkeen kaikki puun rinnakkaiset tutkimattomat solmut. Tämän jälkeen algoritmia suoritetaan niin kauan, kunnes kaikki solmut on tutkittu. Solmujen tutkiminen tapahtuu käymällä silmukalla läpi kaikki tutkittavan solmun rinnakkaiset solmut ja tarkistamalla onko kyseisessä solmussa vierailtu jo aikaisemmin, kuten **algoritmin 1** rivillä 7. Jos solmu on vierailematon, asetetaan se vierailluksi ja lisätään se tutkittavien jonoon sekä kasvatettavaan puuhun. Kasvatettavaan puuhun lisätään myös kaari, joka yhdistää lisättävän solmun puuhun.

```
1 Alusta puu T solmuksi v                                          (1)
2 Alusta jono S tutkittavaksi solmuiksi
3 S.push(v)
4 while S != empty
5     tutkittava_solmu = S.pop
6     for(rinnakkainen; tutkittava_solmu->rinnakkaiset)
7           if(rinnakkainen != vierailtu)
8                 aseta rinnakkainen tutkituksi
9                 S.push(rinnakkainen)
10          T.push(rinnakkainen, yhdistävä_kaari)
```

Algoritmi 1. *Pseuodokoodinen puun kasvatusalgoritmi.*

Useasta eri komponentista koostuvalle graafille voidaan virittävän puun sijasta kasvattaa virittävä metsä. Virittävässä metsässä jokaiselle komponentille kasvatetaan oma virittävä puu ja näiden virittävien puiden joukko muodostaa virittävän metsän. [1]

Kasvatettuun puuhun voidaan vaikuttaa myös graafin kaarille annetuilla ominaisuuksilla. Painotetuissa graafeissa eli graafeissa, jossa kaarille on annettu hinta, voidaan kasvattaa virittävä puu siten että sen kokonaishinta on mahdollisimman pieni tai suuri. [2]

3.2 Pienimmän virittävän puun algoritmit

Pienimmät virittävät puut ovat virittäviä puita, joiden kokonaishinta on mahdollisimman pieni. Esimerkiksi sähköverkkoa voisi suunnitteluvaiheessa mallintaa graafina, jossa mahdolliset reitit, joille kaapelia voi vetää, ovat kaaria ja reittien risteyskohdat ja halutut päätepisteet solmuja. Tälle graafille voitaisiin laskea pienin virittävä puu, jonka hintana on kaapeleiden vetämisen hinta. Näin ollen virittävän puun perusteella voitaisiin rakentaa kokonaishinnaltaan halvin sähköverkko, joka kattaa kaikki halutut pisteet.

Pienimmän virittävän puun kasvattamiseen käytettävät algoritmit toimivat edellisessä kohdassa 3.1 mainitun yleisen virittävän puun kasvatukseen käytetyn algoritmin mukaan. Pienintä virittävää puuta rakennettaessa parhaana lisättävänä solmuna pidetään sitä, jonka lisääminen kasvattaa puun kokonaispainoa vähiten. Tunnettuja klassisia algoritmeja ovat Kruskalin ja Primin algoritmit, jotka ovat yleisiä algoritmeja pienimmän virittävän puun etsimiseen. [4]

3.2.1 Primin algoritmi

Primin algoritmi toimii vastaavasti kuin yleinen puun kasvatusalgoritmi. Algoritmissa aluksi kaikkien solmujen hinnaksi asetetaan ääretön ja niiden edeltävä solmu asetetaan tyhjäksi, kuten pseudokoodisen **algoritmin 2** riveillä 1 ja 2. Tämän jälkeen valitaan satunnainen solmu juureksi, jolle asetetaan hinnaksi nolla. Tämän jälkeen kaaria lisätään prioriteettijonoon käymällä läpi viimeksi puuhun lisätyn solmun kaikki rinnakkaiset solmut, kuten esimerkki **algoritmin 2** riveillä 6–11. Prioriteettijonosta valitaan kaari, joka on painoltaan pienin ja lisätään puuhun sen päässä oleva solmu. Koska jokainen lisättävä solmu on erillinen puusta, ei lisätty kaari voi muodostaa piiriä kasvatettavaan puuhun. Solmujen lisäämistä jatketaan, kunnes kaikki solmut ovat puussa ja lisätyt kaaret muodostavat pienimmän virittävän puun.[4]

```
1 for( v in graafi)                                          (2)
2     v.key = ∞, v.ed = null
3 Alusta jono S prioriteettijonoksi
```

```
4 S.key = 0
5 S.push(v)
6 while S != empty
7     tutkittava_solmu = S.pop MIN
8     for(rinnakkainen; tutkittava_solmu->rinnakkaiset)
9           if(rinnakkainen != vierailtu & rinnakkainen.key > paino)
10               S.push(rinnakkainen)
11               rinnakkainen.key = paino,
12               rinnakkainen.ed = tutkittava_solmu
13    aseta tutkittava_solmu tutkituksi
```

Algoritmi 2. *Pseudokoodinen Primin algoritmi.*

Primin algoritmin suoritusaikaan vaikuttaa prioriteettijonon toteutustapa. Esimerkki **algoritmin 2** rivit 1-3 suoritetaan solmujen määrän verran, niiden tehokkuus on O(V). **Algoritmin 2** rivien 6–11 for-silmukka suoritetaan graafin kaarien määrän verran tehokkuudella O(E). While-silmukka taas toistuu solmujen määrän verran ja siellä tapahtuva pienimmän arvon ottaminen jonosta riippuu sen toteutuksesta, kuin myös for-silmukassa tapahtuva uuden solmun lisääminen jonoon. Jos prioriteettijono toteutetaan vain listana, josta etsitään aina kevyin kaari, on algoritmin asymptoottinen tehokkuus $O(V^2)$, sillä jokaisella while-silmukan kierroksella jouduttaisiin etsimään oikea silmukka listasta, joka on tehokkuudeltaan O(V). Näin ollen kokonaistehokkuudeksi muodostuu $O(V^2)$. Yleisin tapa toteuttaa prioriteettijono on **binäärikeko**, jolla keon järjestyksen päivittäminen ja sieltä pienimmän alkion poistaminen tapahtuvat logaritmisessa ajassa. Näin ollen for-silmukka huonompana tapauksena määrittää kokonaistehokkuudeksi O(E log V). Nopeimmillaan Primin algoritmi suoriutuu, kun prioriteettijono toteutetaan **fibonacci-keon** avulla, jolloin tehokkuus laskee edelleen O(E + V log V) sillä keolla pienimmän ottaminen jonosta kestää logaritmisen ajan ja sinne lisääminen laskee vakioaikaiseksi. [4]

3.2.2 Kruskalin algoritmi

Kruskalin algoritmin suoritustapa eroaa perinteisistä puun kasvatusalgoritmeista, sillä siinä yhden puun sijasta algoritmi kasvattaa metsää, jossa puita yhdistetään toisiinsa, kunnes koko virittävä puu on löydetty. Algoritmin alussa graafin jokainen solmu asetetaan omaksi komponentikseen samoin kuin esimerkki **algoritmin 3** rivillä 3. Tämän jälkeen algoritmissa valitaan kaikista graafin kaarista hinnaltaan pienin kaari, joka yhdistää kaksi toisistaan irrallista komponenttia. Kun kaikki graafin kaaret on tutkittu muodostavat puuhun valitut kaaret pienimmän virittävän puun. [5]

```
1 Alustetaan puu T                                              (3)
2 for( v in graafi)
3     make-set(v)
4 järjestä kaaret hinnan mukaan
5 for(kaari in kaaret)
6     if(find-set(kaari.a) != find-set(kaari.b))
7           lisää kaari puuhun
8     Union(kaari.a, kaari.b)
```

Algoritmi 3. *Pseudokoodinen Kruskalin algoritmi.*

Kruskalin algoritmin tehokkuus perustuu tapaan, jolla irralliset puut toteutetaan. Yleisin tapa toteuttaa ne on käyttää **disjoint-set** rakennetta, joka käyttää puiden yhdistämistä arvon avulla ja polun tiivistämistä**. Find-set** etsii puukomponentista sen juuren ja polun tiivistämisellä myös jokaisen polun varrella olevan solmun juureksi asetetaan lopullinen juuri. Jos tutkittavat komponentit ovat erilliset yhdistetään ne arvon perusteella, yhdistämällä pienempiarvoinen komponentti isompaan **union-set**-ominaisuudella. Kaikkien irrallisten joukkojen operaatioiden kokonaistehokkuus on O(E log E). Kuitenkin koska yksinkertaisissa graafeissa kaarien määrä on aina $|E| < |V^2|$ on log(E) tehokkuudeltaan O(2 Log V), jolloin tehokkuudeksi saadaan O(E log V). [4]

4. ALGORITMIEN VERTAILU

Tutkimuksen tarkoituksena on vertailla itse toteutettuja Primin ja Kruskalin algoritmeja. Algoritmeja tutkitaan suorittamalla niitä satunnaisesti luoduissa graafeissa. Kaikki luodut testattavat graafit ovat yksinkertaisia, suuntaamattomia ja yhdistettyjä graafeja. Lisäksi testattavia graafeja on kahdenlaisia, sekä harvoja että tiiviitä. Harvoissa graafeissa solmujen välisten kaarien määrä on suhteellisen lähellä solmujen määrää, kun taas tiiviissä graafeissa kaarien määrä on mahdollisimman lähellä yksinkertaisen suuntaamattoman graafin kaarien maksimimäärää.

Molemmat algoritmit suoritetaan 20 kertaa samankokoista satunnaista graafia kohti, jonka jälkeen graafin kokoa kasvatetaan. Graafien kaarien painoarvot valitaan satunnaisesti, joukosta 1–|E|**.** Tarkoituksena on tutkia kasvaako algoritmien suoritusaika niiden teoreettisen asymptoottisen tehokkuuden perusteella, vai tapahtuuko niiden suoritusaikojen kasvussa poikkeuksia. Lisäksi tutkimuksessa vertaillaan algoritmeja niiden suoritusaikojen perusteella kuin myös niiden toteutuksien monimutkaisuuden kannalta.

Kaikki testaukseen käytetty ohjelmakoodi on toteutettu C++-ohjelmointikielellä käyttäen hyväksi sen standardikirjastoa. Ohjelmointi on tapahtunut käyttäen Qt-ohjelmointiympäristön versiota 5.12.0 ja kääntäjästä MinGW versiota 7.3.0 64-bit ilman optimointiasetuksia. Itse toteutuksessa graafin solmuille käytössä on itse ohjelmoitu struct-datatyyppi **Vertex** (liite B) ja graafille on toteutettu oma **Graph**-luokka. Satunnaiset graafit tuotetaan itse ohjelmoidulla algoritmilla (liite C), joka aluksi luo halutun määrän Vertex-datatyypin solmuja graafiin. Kun solmut on lisätty kasvatetaan satunnaisella puun kasvatusalgoritmilla graafin pohjaksi puurakenne, jotta voidaan varmistua siitä että graafi on yhtenäinen. Tämän jälkeen algoritmi lisää satunnaisten solmujen välille halutun määrän kaaria, tarkastaen että graafi pysyy yksinkertaisena.

Algoritmit ajetaan tietokoneella, jonka prosessorina on Inter Core i5-4690K. Siinä on neljä ydintä ja sen kellotaajuus on 3,50 GHz:ä. Muistia laitteesta löytyy 8Gt:ä ja sen nopeus on 1600 MHz. Käyttöjärjestelmänä tietokoneessa on 64-bit Windows 10 Home, jonka versio on 1809.

4.1 Primin testaaminen

Tutkimusta varten toteutettu Primin algoritmi käyttää prioriteettijonon toteutuksessa C++-ohjelmointikielen standardikirjaston prioriteettijonoa, joka rakentaa binäärikeon halutun tietorakenteen sisälle. Lisääminen ja pienimmän arvon poistaminen prioriteettijonoon on tehokkuudeltaan logaritminen O(log N), joten sen puolesta tietorakenne vastaa binäärikeolla toteutettua Primin algoritmia. Koska standardikirjaston jonon toteutus ei sisällä ominaisuutta arvojen päivittämiseen keon sisällä, oma toteutukseni lisää jokaisen

mahdollisen kaaren jonoon, jolloin binäärikeon operaatioiden suoritusaika kasvaa O(E log E). Tämä johtuu siitä että huonoimmassa tapauksessa samat solmut joudutaan lisäämään jonoon useamman kerran, kuitenkin enintään kaarien määrän verran. Algoritmin kokonaistehokkuus on kuitenkin O(E log V), saman periaatteen perusteella kuin kohdan 3.2.2 lopussa.

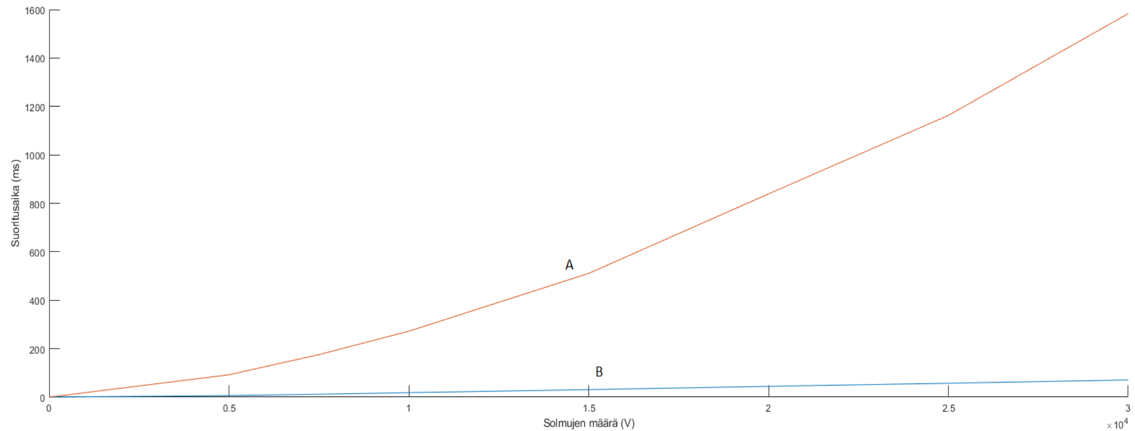
```
Graph Graph::PrimMst(int N)
{
  Graph tree; //Luodaan graafi johon puu rakennetaan

  // Luodaan prioriteettijono joka käyttää omaa vertausfunctiota.
  std::priority_queue<pair<int, Vertex*>, vector<pair<int, Vertex*>>,
decltype(&compare_edge_weight)> queue(&compare_edge_weight);
    Vertex* root = vertices_.at(0); //Valitaan aloitussolmu
    pair<int, Vertex*> vertex;

    root->distance = 0;
    queue.push(make_pair(root->distance ,root));

    //Käydään silmukassa läpi prioriteettijonoa
    while(!queue.empty()){
        vertex = queue.top();
        queue.pop();
        if(vertex.second->visited == true){continue;}

        tree.add_vertex(vertex.second);
        vertex.second->distance = vertex.first;

        // Lisätään jonosta otettu solmu puuhun.
         if(vertex.second->parent != vertex.second){
              tree.add_edge_mst(vertex.second->distance,
                                vertex.second,
                                vertex.second->parent);
         }

        if(tree.size() == N){break;}

        //käydään läpi kaikki tutkittavan solmun kaaret.
        for(auto edge: vertex.second->adjcacent_vertices){
            if(edge.first->visited == false &&
                    edge.first->distance > edge.second){
                edge.first->parent = vertex.second;

                queue.push(make_pair(edge.second, edge.first));
            }
        }
        vertex.second->visited = true;
    }
    return tree;
}
```

Ohjelma 1. *Primin algoritmin toteutus C++11-ohjelmointikielellä.*

Ohjelma 1:ssä näkyy toteutettu algoritmi. Algoritmin alussa riveillä 1–12 tehdään algoritmin suorittamista varten vaaditut alustukset. Alustamisen jälkeen käydään prioriteettijonoa läpi, joko niin kauan että jono on tyhjä tai kasvatettava puu sisältää kaikki solmut. Jonosta nostetut solmut lisätään puuhun ja yhdistetään kaarella vanhemmaksi asetettuun solmuun. Seuraavaksi riveillä 34–37 käydään läpi tutkittavan solmun rinnakkaiset solmut

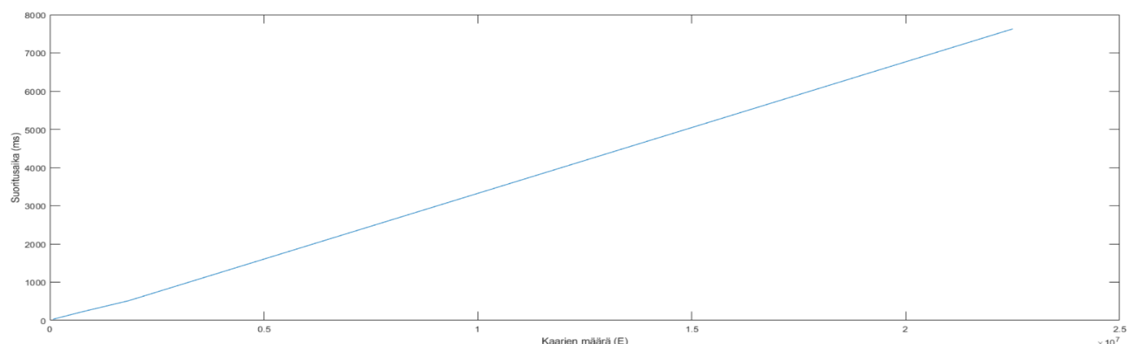
ja lisätään ne jonoon, jos niitä ei ole vielä tutkittu. Kun solmu on tutkittu, asetetaan se tutkituksi ohjelman rivillä 41.

4.1.1 Tulokset



Kuva 4. *Kuvaajassa x-akselilla on testatun graafin solmujen määrä V ja y-akselilla Primin algoritmin suoritusaika (ms). Käyrä A kuvaa algoritmin suoritusaikoja tiheällä graafilla, jossa kaarien määrä on sqrt(V) * V ja B harvalla graafilla, jossa kaarien määrä on 5 * V (liite A Taulukot 1 ja 3).*

Kuvan 4 kuvaajan perusteella Primin algoritmin suoritusaika kasvaa samalla nopeudella, kuin teoreettisen tehokkuuden perusteella voisi ennustaa. Käyrässä B, joka kuvaa Primin algoritmin suoritusaikoja harvassa graafissa, suoritusaika 5000 solmun pienimässä graafissa on 6,5 ms. Suurimassa graafissa, jonka solmujen määrä on 30 000, algoritmin suoritusaika on kasvanut 70,1 ms:iin. Tiheissä graafeissa suoritusaika sen sijaan kasvaa pienimmän graafin 92,2 ms:sta isoimman graafin 1,6 s:iin. Kuten **kuvan 4** kuvaajasta voi nähdä graafin tiheys vaikuttaa todella paljon suoritusaikoihin, kuten tehokkuudesta O(E log V) pystyy päättelemään, sillä O(lg V) operaatio suoritetaan |E| kertaa. **Kuvassa 5** myös näkyy, kuinka kaarien määrän kasvattaminen kasvattaa suoritusaikaa lineaarisesti graafissa, jonka solmujen määrä pysyy vakiona.



Kuva 5. *Kuvaajassa x-akselilla on kaarien määrä E ja y-akselilla suoritusaika (ms). Käyrä kuvaa kaarien määrän vaikutusta suoritusaikaan 15 000 solmun graafissa (liite A Taulukko 5).*

4.2 Kruskalin testaaminen

Kruskalin algoritmin toteutus on toteutettu hyödyntäen disjoint-set-tietorakennetta. Rakenne on toteutettu suoraan oman solmu-tietotyypin Vertex ominaisuuksiin. Tietotyyppi sisältää osoittimen puussa olevaan vanhempaan ja kokonaislukumuuttujan arvoa varten. Funktiot joukkojen löytämiseen ja yhdistämiseen on toteutettu graafiluokan jäsenfunktioina, mikä mahdollistaa toimintojen tapahtumisen suoraan rakennettavan puun sisällä. Kaarien järjestämiseen algoritmi käyttää standardikirjaston järjestysalgoritmia, jonka tehokkuus on O(N log N). Toteutuksen asymptoottinen tehokkuuden siis tulisi olla O(E log V).

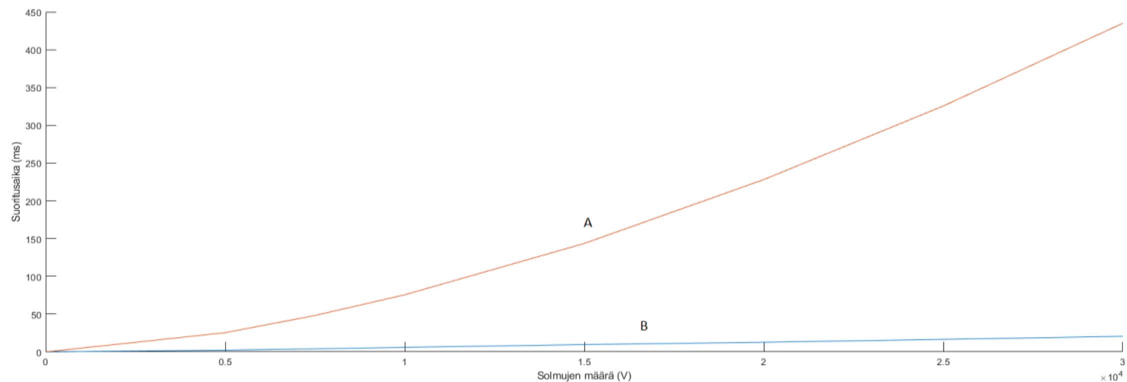
```
    Graph Graph::KruskalMst(int N)
 2  {
 
 4      Graph tree;
        Vertex* first;
 6      Vertex* second;
        Vertex* first_set;
 8      Vertex* second_set;
        int edge_counter = 0;
10
        //Alusta puun solmut erillisiksi puiksi
12      for(auto vertex: vertices_){
            tree.add_vertex(vertex);
14      }

16      sort(edges_.begin(), edges_.end());

18      for(auto edge: edges_){

20          first = std::get<1>(edge);
            second = std::get<2>(edge);
22
            first_set = tree.find_Set(first);
24          second_set = tree.find_Set(second);

26          if(first_set != second_set){
                tree.union_sets(first_set, second_set);
28
                tree.add_edge_mst(std::get<0>(edge), first, second);
30              ++edge_counter;

32          }
            if(edge_counter >= N-1){
34              return tree;
            }
36      }
    }
```

Ohjelma 2. *Kruskalin algoritmin toteutus C++11-ohjelmointikielellä.*

Ohjelmassa 2 muuttujien alustamisen jälkeen riveillä 12 ja 13 suoritetaan solmujen alustus metsäksi. Tämän jälkeen kaaret järjestetään painon mukaan pienimmästä suurimpaan. Tämän jälkeen riveillä 18–30 algoritmi tutkii, voiko kaaren lisätä puuhun aiheuttamatta

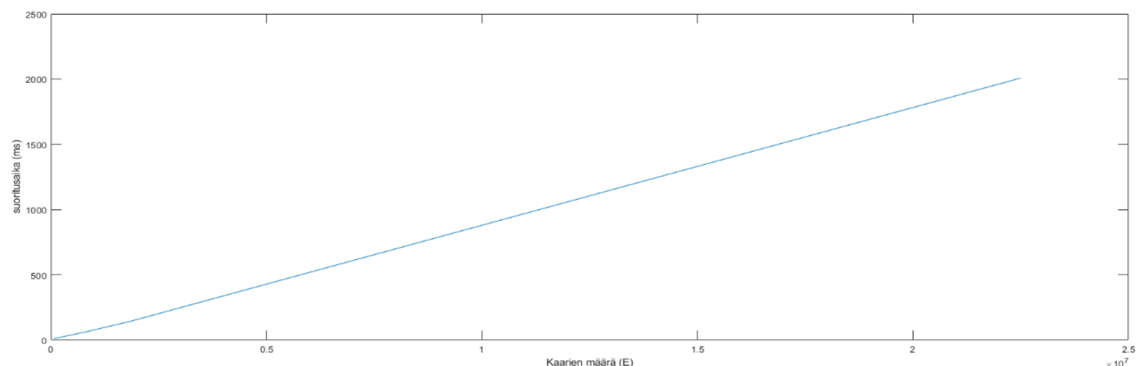
siihen piiriä. Viimeisenä silmukassa algoritmi tarkastaa onko puussa vaadittu määrä kaaria ja palauttaa valmiin puun tarkistuksen onnistuessa.

4.2.1 Tulokset



Kuva 6. *Kuvaajassa x-akselilla on testatun graafin solmujen määrä V ja y-akselilla Kruskalin algoritmin suoritusaika (ms). Käyrä A kuvaa algoritmin suoritusaikoja tiheällä graafilla, jossa kaarien määrä on sqrt(V) * V ja B harvalla graafilla, jossa kaarien määrä on 5 * V (liite A Taulukot 2 ja 4).*

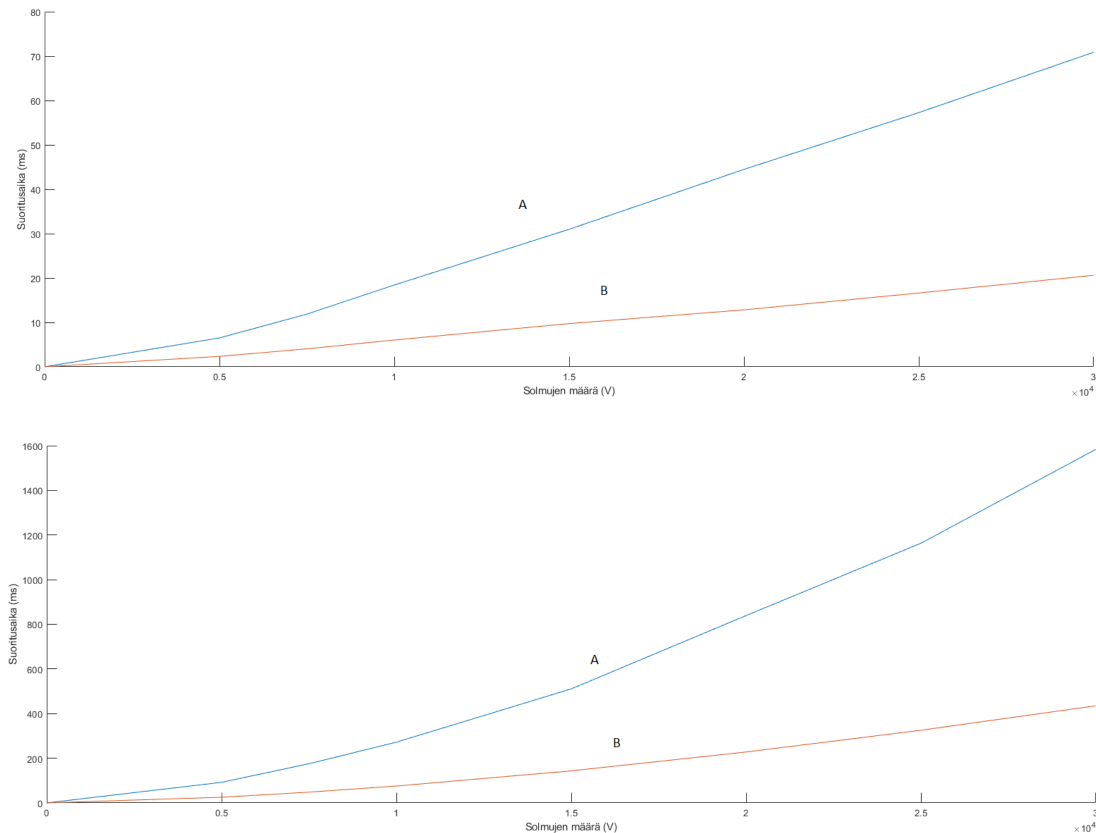
Kuvassa 6 näkyy kuinka Kruskalin algoritmin suoritusaika kasvaa graafin solmujen määrän kasvaessa. Sekä tiheässä että harvassa graafissa suoritusaika kasvaa odotetusti asymptoottisen tehokkuuden mukaisesti. Harvoissa graafeissa suoritus aika kasvaa 2,4 ms:sta 22,6 ms:iin ja tiheissä graafeissa 25,5 ms:sta 435,0 ms:iin. Kuvaajasta myös näkyy graafin tiheyden vaikutus pidempinä suoritusaikoina vastaavasti kuin Primin algoritmissa. Vastaavasti myös **kuvasta 7** voi todeta, että kaarien määrän lisääminen kasvattaa suoritusaikaa lineaarisesti solmuiltaan vakiokokoisessa graafissa.



Kuva 7. *Kuvaajassa x-akselilla on kaarien määrä E ja y-akselilla suoritusaika (ms). Käyrä kuvaa kaarien määrän vaikutusta suoritusaikaan 15 000 solmun graafissa (liite A Taulukko 6).*

4.3 Vertailu

Toteutetuista Kruskalin ja Primin algoritmeista oli Kruskalin algoritmi suoritusajoiltaan nopeampi sekä harvoissa että tiheissä graafeissa, kuten **kuvan 8** kuvaajien perusteella voi todeta. Kuitenkin kummankin tyyppisissä graafeissa tehokkuuksien suhde pysyi jotakuinkin samana, kuten **kuvan 8** kuvaajissa näkyy. Molemmissa testattavien graafien suoritusaika oli molemman tyyppisissä graafeissa Primin algoritmissa 3–3,5 kertainen millisekunneissa Kruskalin algoritmin suoritusaikaan nähden.



Kuva 8. *Kuvaajissa on Kruskalin ja Primin algoritmien keskinäiset suoritusajat, sekä harvassa (ylempi) että tiheässä (alempi) graafissa. X-akselilla on graafin solmujen määrä V ja y-akselilla suoritusaika (ms). Käyrä A kuvaa Primin algoritmia, kun taas käyrä B Kruskalin algoritmia.*

Tästä voidaan todeta, että toteutetuista algoritmeista Kruskalin algoritmin toteutus oli huomattavasti nopeampi, kuin Primin algoritmin. Varsinkin testattavia graafeja isommissa graafeissa, joissa suoritusajat kasvavat sekunteihin, on kolminkertainen ero suoritusajoissa merkittävä. Kuitenkin Ali Erkanin kirjoittamassa julkaisussa The Educational Insights and Opportunities Afforded by the Nuances of Prim's and Kruskal's MST Algorithms, ei vastaavalla tavalla toteutetuissa algoritmeissa ollut suoritusajoilla mitään eroa satunnaisissa testigraafeissa [6]. Erkanin julkaisussa testattiin myös Fibonacci-keolla to-

teutettua prioriteettijonoa Primin algoritmissa. Fibonacci-keolla toteutettu Primin algoritmi oli varsinkin tiheissä graafeissa selvästi tehokkaampi kuin Kruskalin algoritmi ja prioriteettijonolla toteutettu Primin algoritmi.

Saatujen tulosten perusteella voi todeta, että Primin algoritmin toteutuksessa on mahdollisesti joitain suoritusajan kannalta epäoptimaalisia ratkaisuja. Esimerkiksi Primin algoritmissa prioriteettijono vaatii, että solmujen tallennetaan kaaren painon kanssa parina , joka kasvattaa prioriteettijonon alkion lisäämis operaation suoritusaikaa. Lisäksi graafien toteutustapa on Kruskalin algoritmin toteutuksen kannalta tehokas, sillä algoritmin hyödyntämä **disjoint-set**-rakenne on toteutettuna suoraan solmujen käyttämän **Vertex**-datatyypin sisään, joka mahdollistaa rakenteen käyttämien operaatioiden toteuttamisen ilman ylimääräisiä tietorakenteita.

Itse toteutuksia vertaillessa Kruskalin algoritmi on helpompi toteuttaa optimaalisesti verrattuna Primin algoritmiin. Primin algoritmin prioriteettijonon toteuttaminen algoritmin kannalta parhaalla tavalla vaatisi joko kolmannen osapuolen kirjastoa tai sen toteuttamista itse siten että **decrease-key**-ominaisuus on käytössä. Kruskalin algoritmi taas oli mahdollista toteuttaa optimaalisesti turvautumatta kolmansien osapuolien kirjastoihin ja käyttämällä samaa **Vertex**-tietotyypin sisällä olevaa **parent**-muuttujaa, jota myös Primin toteutus käyttää hyväksi ja **key**-numeroarvoa **disjoint-set** rakenteen toteuttamiseen.

5. YHTEENVETO

Työssä toteutettiin Kruskalin ja Primin algoritmit ja näitä toteutuksia testattiin, jotta pystyttiin vertaamaan vastasivatko toteutukset niiden teoreettisia tehokkuuksia ja ominaisuuksia. Algoritmeista testattiin myös kaarien määrän vaikutusta algoritmien suoritusaikoihin. Lisäksi algoritmeja vertailtiin keskenään niin suoritusaikojen kuin toteutuksien monimutkaisuuden ja onnistumisen näkökannalta.

Molemmat algoritmit vastasivat tehokkuuksiltaan odotettuja teoreettisia tehokkuuksia O(E log V). Molemmissa algoritmeissa myös kaarien määrä vaikutti odotetusti suoritusaikoihin. Vertailussa todettiin, että toteutuksista Kruskalin algoritmi oli kuitenkin Primin algoritmia tehokkaampi ja sen suoritusajat olivat pääsääntöisesti kolmasosa siitä mitä ne olivat Primin algoritmilla saman kokoisissa graafeissa.

Myös toteutuksia vertailtaessa Kruskalin algoritmi osoittautui paremmaksi, sillä sen optimaalinen toteuttaminen on merkittävästi helpompaa käyttämättä kolmannen osapuolen kirjastoja toteutuksessa. Primin algoritmi sen sijaan vaatii joko oman prioriteettijonon toteuttamista tai kolmannen osapuolen kirjaston hyödyntämistä, C++-ohjelmointikielen standardikirjaston prioriteettijonon sijaan, jotta toteutus olisi täysin optimaalinen.

Tämän tutkimuksen perusteella Kruskalin algoritmi on tehokkaampi kuin Primin algoritmi niin suoritusaikojen kuin toteutuksen yksinkertaisuuden osalta. Tämän perusteella varsinkin tilanteissa, joissa toteutuksen yksinkertaisuus on osatekijä algoritmin valinnassa on Kruskalin algoritmi parempi.

LÄHTEET

[1] J.L. Gross, J. Yellen, Graph theory and its applications, 2nd ed. Chapman & Hall/ CRC, Boca Raton (FL), 2006, 779 p.

[2] V.I. Voloshin, Introduction to Graph Theory, Nova Science Publishers, Incorporated, Hauppauge, 2009, 144 p.

[3] K. Ruohonen, Graafiteoria, Tampereen teknillinen yliopisto, 2013, 106 s. Viitattu 26.02.2019, Saatavissa: http://math.tut.fi/~ruohonen/GT.pdf

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, 2009, 132 p.

[5] J.B. Kruskal, On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, Proceedings of the American Mathematical Society, Vol. 7, Iss. 1, 1956, pp. 48–50.

[6] A. Erkan, The educational insights and opportunities afforded by the nuances of Prim's and Kruskal's MST algorithms, Proceedings of the 23rd Annual ACM Conference on innovation and technology in computer science education, ACM, 2018, pp. 129–134.

LIITE A: TESTITULOKSIEN TAULUKOT

Taulukko 1. **Primin algoritmin tulokset harvassa graafissa.**

Solmujen määrä N	Kaarien määrä 5*N	Suoritusaika (ms)
5 000	25000	6.5035
7 500	37500	11.8698
10 000	50000	18.4243
15 000	75000	30.9801
20 000	100000	44.4779
25 000	125000	57.2692
30 000	150000	70.8590

Taulukko 2. **Kruskalin algoritmin tulokset harvassa graafissa.**

Solmujen määrä N	Kaarien määrä 5*N	Suoritusaika (ms)
5 000	25 000	2.3513
7 500	37 500	4.0444
10 000	50 000	6.0360
15 000	75 000	9.7320
20 000	100 000	12.8291
25 000	125 000	16.6265
30 000	150 000	20.6182

Taulukko 3. **Primin algoritmin tulokset tiheämmässä graafissa.**

Solmujen määrä N	Kaarien määrä sqrt(N)*N	Suoritusaika (ms)
5 000	353553	92.2143
7 500	649519	175.4358
10 000	1000000	272.1352
15 000	1837117	510.7629
20 000	2828427	839.1180
25 000	3952847	1163.1327
30 000	5196152	1583.3889

Taulukko 4. **Kruskalin algoritmin tulokset tiheämmässä graafissa.**

Solmujen määrä N	Kaarien määrä sqrt(N)*N	Suoritusaika (ms)
5 000	353553	25.4866
7 500	649519	48.2313
10 000	1000000	75.6228
15 000	1837117	143.5500
20 000	2828427	228.1122
25 000	3952847	325.4946
30 000	5196152	435.0085

Taulukko 5. **Primin algoritmin tulokset 15 000 solmulla ja kaarien määrää kasvattamalla.**

Kaarien määrä	Suoritusaika (ms)
75 000	29.1551
150 000	52.5893
750 000	224.8672
1 837 117	514.7364
22 498 500	7626.4714

Taulukko 6. **Kruskalin algoritmin tulokset 15 000 solmulla ja kaarien määrää kasvattamalla.**

Kaarien määrä	Suoritusaika (ms)
75 000	9.7320
150 000	16.3664
750 000	58.1630
1 837 117	142.8690
22 498 500	2007.8002

LIITE B: VERTEX-DATATYYPPI

```
    struct Vertex{
2       std::map<Vertex*, int> adjcacent_vertices;
        std::map<Vertex*, int> adjcacent_vertices_mst;
4       bool visited;

6       int key;
        int distance;
8       Vertex* parent;

10      int degree;

12      Vertex()
        {
14          adjcacent_vertices = {};
            adjcacent_vertices_mst = {};
16          visited = false;
            key = 0;
18          distance = INF;
            parent = this;
20          degree = 0;
        }
22  };
```

Ohjelma 3. *Vertex-datatyyppi.*

LIITE C: SATUNNAISGRAAFIN KASVATUSALGORITMI

```
vector<Vertex*> all_vertices = vertices_;
    vector<Vertex*> visited_vertices;
    std::uniform_int_distribution<> all_random(0, N - 1);
    int random_index = all_random(gen);
    Vertex* root = vertices_.at(random_index);
    all_vertices.erase(all_vertices.begin() + random_index);
    visited_vertices.push_back(root);

    while(!all_vertices.empty())
    {
        std::uniform_int_distribution<> all_random(0, all_verti-
ces.size() - 1);
        random_index = all_random(gen);
        Vertex* new_vertex = all_vertices.at(random_index);
        all_vertices.erase(all_vertices.begin() + random_index);

        std::uniform_int_distribution<> visited_random(0, visited_verti-
ces.size() - 1);
        random_index = visited_random(gen);
        Vertex* neigbhor_vertex = visited_vertices.at(random_index);


        add_edge(weight(gen), new_vertex, neigbhor_vertex);
        edge_count++;

        visited_vertices.push_back(new_vertex);
    }

    while(edge_count < amount){
        random_index = rand() % vertices_.size();
        Vertex* vertex_a = vertices_.at(random_index);

        random_index = rand() % vertices_.size();
        Vertex* vertex_b = vertices_.at(random_index);

        if(vertex_a != vertex_b && vertex_a->adjcacent_verti-
ces.find(vertex_b) == vertex_a->adjcacent_vertices.end()){
            add_edge( weight(gen), vertex_a, vertex_b);
            edge_count++;
        }
        if(edges_.size() >= 0.5*(N*(N-1))){break;}
    }
```

Ohjelma 4. *Satunnaisen graafin kasvatusalgoritmi.*