Kauppamatkustajan ongelman ratkaiseminen Kruskalin algoritmin avulla

Henrik Nenonen

Pro gradu –tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Joulukuu 2019

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Joensuu
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Nenonen, Henrik Juhani: Kauppamatkustajan ongelman ratkaiseminen Kruskalin algoritmin avulla
Pro gradu –tutkielma, 44 s.
Pro gradu –tutkielman ohjaaja: FT Pasi Fränti
Joulukuu 2019

Pro gradu -tutkielman aiheena on avoimen kierroksen kauppamatkustajan ongelman ratkaiseminen Kruskalin algoritmin avulla. Ensiksi selvitetään mikä on kauppamatkustajan ongelma, kuinka kyseistä ongelmaa on yritetty ratkaista ja miksi ongelma on vaikea. Tämän jälkeen selvitetään mikä on pienin virittävä puu sekä mikä Kruskalin algoritmi on ja kuinka se toimii. Teorian jälkeen siirrytään tutkielman varsinaiseen asiaan, eli miten Kruskalin algoritmia muokkaamalla voidaan saada ratkaisu avoimen kierroksen kauppamatkustajan ongelmaan. Tutkielmassa esitetään Kruskal-TSP:ksi nimetty algoritmi. Sen rakennetta sekä eroa normaalin Kruskalin algoritmiin tutkitaan. Lisäksi algoritmin ongelmakohtia ja aikavaativuutta analysoidaan. Seuraavaksi tutkielmassa esitetään satunnaisuutta käyttävä versio Kruskal-TSP-algoritmista, jonka tarkoituksena on korjata ilmaantuneita ongelmakohtia. Sen jälkeen analysoidaan sekä perus- että satunnaistetulla Kruskal-TSP-algoritmilla saatuja tuloksia ja verrataan niitä optimaalisiin tuloksiin. Lopuksi sekä perus- että satunnaistettua Kruskal-TSP:tä verrataan muutamaan muuhun kauppamatkustajan ongelman ratkaisevaan algoritmiin.

Avainsanat: Kauppamatkustajan ongelma, Lyhin reitti, Kruskalin algoritmi, Pienin virittävä puu.

ACM-luokat (ACM Computing Classification System, 1998 version): F.2.2, G.2.2

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Joensuu
School of Computing
Computer Science

Nenonen, Henrik Juhani: Solving Travelling salesman problem with Kruskal’s algorithm
Master’s Thesis, 44 p.
Supervisor of the Master’s Thesis: PhD Pasi Fränti
December 2019

Subject of this Master’s thesis is solving the open loop travelling salesman problem with help of Kruskal’s algorithm. At first it is clarified what is travelling salesman problem, how it is tried to be solved and why the problem is so hard. After that is explained what minimum spanning tree is and what is Kruskal’s algorithm and how it works. After theory part focus shifts to main part of thesis, which is modifying Kruskal’s algorithm to solve open loop travelling salesman problem. In this thesis an algorithm named Kruskal-TSP is presented. It’s structure and difference to the normal Kruskal’s algorithm are examined. Also, the algorithm’s weaknesses and time complexity are analyzed. Next, in thesis is presented randomized version of Kruskal-TSP which is supposed to fix some of the appeared problems. This algorithm is also analyzed and both basic and randomized Kruskal-TSP results are compared to optimal results. Lastly both basic and randomized Kruskal-TSP are compared to some other TSP solving algorithms.

Keywords: Travelling salesman problem, Shortest path, Kruskal’s algorithm, Minimum spanning tree.

CR Categories (ACM Computing Classification System, 1998 version): F.2.2, G.2.2

Esipuhe

Tämä tutkielma on tehty valmiiksi Itä-Suomen yliopiston Tietojenkäsittelytieteen laitokselle syksyllä 2019. Haluan kiittää FT Pasi Fräntiä ohjauksesta ja kannustuksesta, sekä TkT Saku Kukkosta tutkielman tarkistamisesta ja avusta tutkielman hienosäätämisessä.

Lyhenneluettelo

ACM	Association for Computing Machinery; maailmanlaajuinen tietotekniikka-alan tieteellinen yhdistys
TSP	Travelling salesman problem; Kauppamatkustajan ongelma
MST	Minimum spanning tree; Pienin virittävä puu
ACO	Ant Colony Optimization
TS	Tabu Search

Sisällysluettelo

1 Johdanto 1

2 Mikä on kauppamatkustajan ongelma? 3

2.1 Termin synty 5
2.2 Miten kauppamatkustajan ongelmaa on yritetty ratkaista? 5
2.3 Kauppamatkustajan ongelman vaikeus 5
2.4 Kauppamatkustajan ongelma sovelletussa matematiikassa 6
2.5 Kauppamatkustajan ongelman käytännön sovelluksia 7

3 Pienin virittävä puu 8

3.1 Virittävä puu 8
3.2 Pienin virittävä puu 8
3.3 Historiaa 9
3.4 Kruskalin algoritmi 10
3.5 Pienimmän virittävän puun yhteys kauppamatkustajan ongelmaan . 11
3.6 MST-solmujen määrän yhteys TSP:n vaikeuden arviointiin 11

4 Kruskalin algoritmi avoimen kierroksen kauppamatkustajan ongelmaan 12

4.1 Algoritmin toiminta 12
4.2 Esimerkki algoritmin toiminnasta 13
4.3 Kruskal-TSP-algoritmin ero verrattuna Kruskalin algoritmiin 16
4.4 Esimerkkejä tapauksista, jossa algoritmi ei tuota hyvää ratkaisua ... 18
4.5 Esimerkki tapauksesta, jossa algoritmi toimii hyvin 21
4.6 Pseudokoodi ja aikavaativuus 22

5 Satunnaistettu Kruskal-TSP 27

5.1 Algoritmin toiminta ja pseudokoodi 27
5.2 Esimerkki satunnaistetun Kruskal-TSP-algoritmin toiminnasta 28

6 Kokeelliset tulokset 30

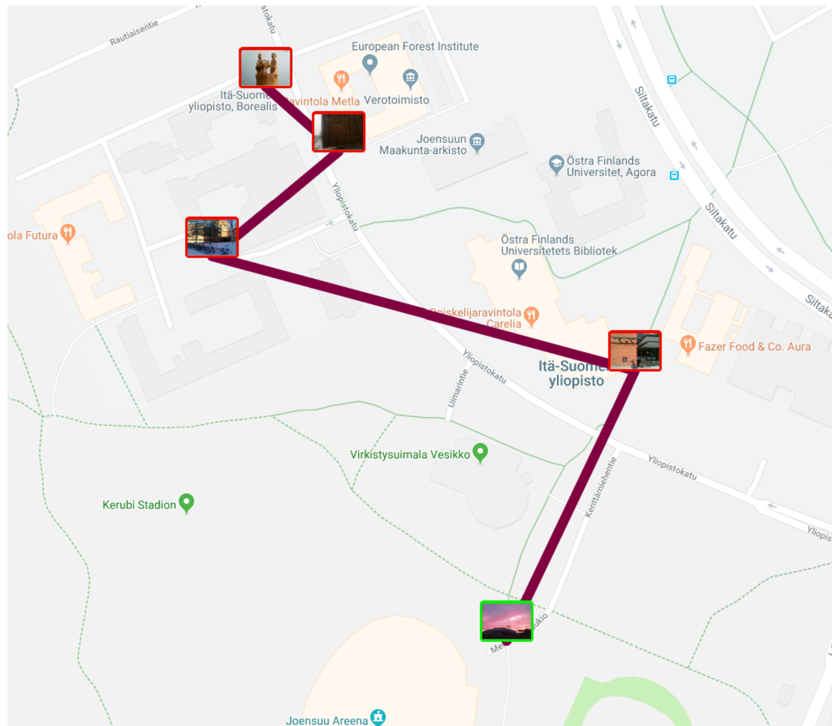
6.1 Testiasetelma 30
6.2 Esimerkkitulos O-Mopsi testidatasta 30
6.3 Kokonaistulokset 32
6.4 Kruskal-TSP verrattuna muihin algoritmeihin 39

7 Yhteenveto 41

8 Viitteet 42

1 Johdanto

Kauppamatkustajan ongelmassa on tarkoituksena löytää lyhin reitti, jolla päästään vierailemaan jokaisessa verkon solmussa tasan yhden kerran. Käytännön esimerkkinä on myyjä, joka haluaa löytää lyhimmän reitin kaupunkien läpikäymiseksi [14].



Kuva 1: Käytännön esimerkki kauppamatkustajan ongelmasta ja siihen tehdystä ratkaisusta.

Kauppamatkustajan ongelmalle on monia muita käytännön sovelluksia. Muutamia lisäesimerkkejä ovat ajoneuvojen reititys, piirilevyjen poraus ja varastosta tavaran kerääminen tehokkaasti. Ajoneuvojen reitityksessä on tarkoituksena löytää lyhin mahdollinen reitti, jolla voidaan käydä kaikissa vierailtavissa kohteissa. Postin jakelu on hyvä esimerkki tästä. [8]

Kiinnostavan kauppamatkustajan ongelmasta tekee se, että se on helppo määritellä sekä ymmärtää, mutta ongelmaan ei ole helppoa ratkaisua. Ongelman vaikeus aiheuttaa sen, että algoritmi joko toimii nopeasti ja antaa epäoptimaalisen tuloksen, tai vaihtoehtoisesti toimii hitaasti, mutta antaa optimaalisen tuloksen. Molempia on

mahdotonta saada. Ongelman ratkaisemiseen on kuitenkin kehitelty useampia erilaisia ratkaisuja. Tarkoituksena on yleensä saada lähes optimaalinen tulos nopeasti. [8]

Seuraavana on esimerkkejä *heuristisesta* lähestymistavasta, jolla on saatu hyviä tuloksia. Yksi tapa on lisätä kaaria yksi kerrallaan valmiin reitin muodostamiseksi. Ensin valitaan lyhin kaari. Tämän jälkeen valitaan lisää kaaria tulosverkkoon jonkun ehdon mukaan. Tämä ehto voi olla esimerkiksi se, että valitaan aina lyhin kaari, joka käy tulosverkon päähän. Toinen reitin muodostus tapa on ensin tehdä TSP-reitti millä tahansa tavalla ja sitten muokata sitä poistamalla huonoja kaaria ja yhdistämällä solmut järkevämmin. [14]

Avoimen kierroksen kauppamatkustajan ongelma liittyy pienimpään virittävään puuhun. Avoimen kierroksen kauppamatkustajan ongelmassa alku- ja loppupiste ei tarvitse olla sama. Tällöin voidaan ajatella, että kauppamatkustajan reitti on virittävä puu. Ei kuitenkaan välttämättä pienin virittävä puu.

Tässä tutkielmassa esiteltävän algoritmin ja sen eri versioiden pääidea on se, että muokataan pienimmän virittävän puun tuottavaa Kruskalin algoritmia sopivaksi. Algoritmia muokataan niin, että saadaan muodostettua avoimen kierroksen kauppamatkustajan ongelman ratkaiseva reitti. Tämä tehdään muutaman ehdon avulla, jotka vaikuttavat siihen, miten kaaria valitaan. Tarkoituksena on siis saada aikaan algoritmi, joka antaa lähes optimin tuloksen, kuluttamatta liikaa aikaa.

Seuraavassa luvussa käsitellään kauppamatkustajan ongelmaa, sen vaikeutta sekä käytännön sovelluksia. Kolmannessa luvussa käsitellään pienintä virittävää puuta, sekä tutkitaan Kruskalin algoritmia. Neljäs luku esittelee tutkielman pääasian eli muokatun Kruskalin algoritmin käytön avoimen kierroksen kauppamatkustajan ongelman ratkaisemiseen. Viidennessä luvussa esitellään satunnaistettu versio muokatusta Kruskalin algoritmista. Kuudes luku käsittelee tuloksia. Viimeisessä luvussa vedetään johtopäätökset.

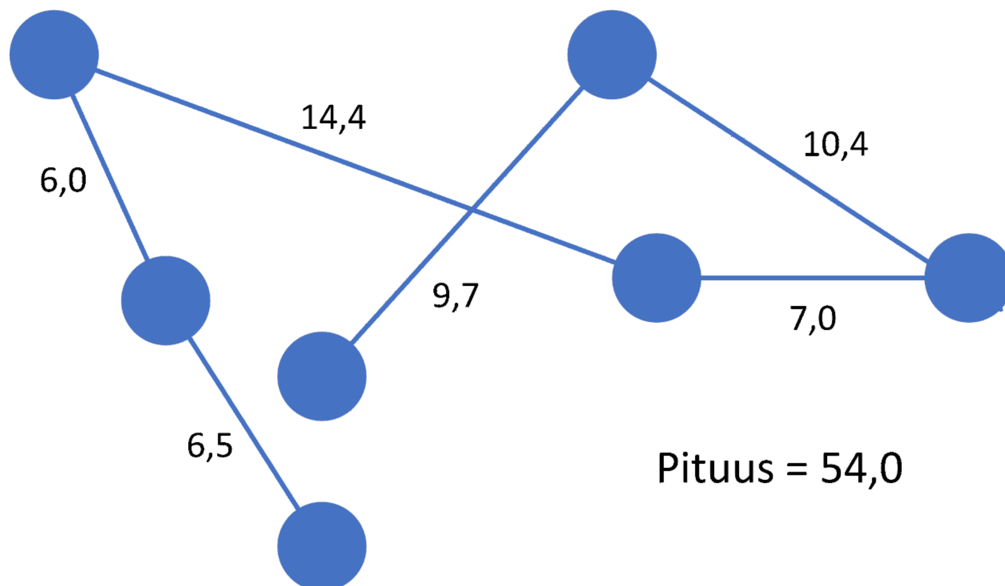
2 Mikä on kauppamatkustajan ongelma?

Kauppamatkustajan ongelmaa eli *travelling salesman problem* (TSP) pidetään yhtenä kuuluisimmista kombinatorisista optimointiongelmista [1]. Kyseisestä aiheesta on tehty huomattavan paljon kirjallisuutta. Suosiota voi selittää se, että ongelma on helppo määritellä, mutta laskennallisesti se on vaikea. Ratkaisusta olisi hyötyä monessa käytännön asiassa. [1]

Kauppamatkustajan ongelmassa tarkoituksena on saada selville lyhin reitti, niin että jokaisen *verkon* solmun lävitse kuljetaan [1]. Verkko koostuu äärellisestä määrästä *solmuja* ja *kaaria*. Kaaret yhdistävät solmuja toisiinsa. Kaaret eli solmujen väliset matkat ovat symmetrisiä, eli matka A:sta B:hen on yhtä pitkä kuin matka B:stä A:han [2]. Konkreettisesti ilmaistuna kauppamies haluaisi kulkea kaupunkien läpi mahdollisimman lyhyttä reittiä.

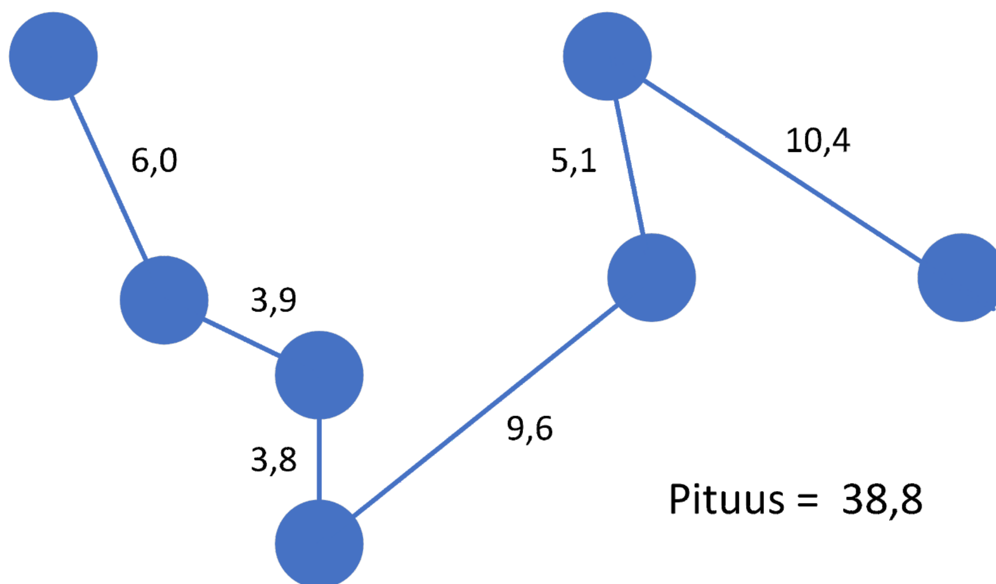
Ongelmasta on olemassa kaksi eri versiota. *Suljetun kierroksen (closed loop)* versiossa aloitus- ja päätepiste on oltava sama. Eli on päästävä samaan kaupunkiin mistä lähti matkaan. *Avoimen kierroksen (open loop)* versiossa loppupiste ei tarvitse olla sama.

Kuten kuvasta 2 näkyy, huonossa avoimen kierroksen TSP-reitissä solmujen järjestys on valittu niin, että tapahtuu turhaa matkustamista. Huonoimmassa tapauksessa reitissä on myös risteyskohtia. Nämä aiheuttavat sen, että reitin pituus ei ole optimaalinen.



Kuva 2: Huono avoimen kierroksen TSP-reitti

Kuvassa 3 näkyy hyvä avoimen kierroksen reitti. Siinä on huomattavasti vähemmän turhaa matkustamista. Tässä tapauksessa reitin kaarien yhteen laskettu pituus on optimaalinen.



Kuva 3: Hyvä avoimen kierroksen TSP-reitti

2.1 Termin synty

Ensimmäisen kerran Kauppamatkustajan ongelma -termiä on käytetty matemaattisissa piireissä 1930-luvun alkupuolella. 1940-luvulla ongelma tuli suosituksi, koska se oli hankala ja kiinnostava ongelma. Se myös liittyi muihin kombinatorisiin ongelmiin. Suosiota selitti myös se, että ongelman nimi muistutti klassista amerikkalaista henkilöä. Henkilö tarkoittaa tässä yhteydessä siis arkityyppiä, jolle on oma osastonsa amerikkalaisessa huumorissa. [6]

2.2 Miten kauppamatkustajan ongelmaa on yritetty ratkaista?

Ensimmäinen käännekohta ongelman ratkaisussa saatiin vuonna 1954 kun artikkeli ”Solution of a large-scale travelling-salesman problem” ilmestyi [11]. Kyseisessä tutkimuksessa saatiin selville, että lineaarisia ohjelmointialgoritmeja voidaan käyttää sijoitusongelman ratkaisemiseen [6]. Varsinainen TSP-ongelma ei kuitenkaan vielä ratkennut. Tutkimuksen tulokset kuitenkin antoivat toivoa, sillä sijoitusongelmaa voidaan kuvata hyvin samankaltaisena kuin kauppamatkustajan ongelma [6].

2.3 Kauppamatkustajan ongelman vaikeus

1960-luvun lopulla ymmärrettiin, että on olemassa helppoja ongelmia ja vaikeita ongelmia [6]. Helppoihin ongelmiin on löydetty algoritmi, joka toimii korkeintaan polynomiaalisessa ajassa suhteessa ongelman kokoon [6]. Vaikeissa ongelmissa sopivat algoritmit ovat luonteeltaan kaikki mahdolliset ratkaisuvaihtoehdot läpikäyviä [6]. Näytti siltä, että TSP-ongelmana on niin monimutkainen, että mikä tahansa ratkaisu vaatisi superpolynomiallisen ajan verrattuna ongelman kokoon eli solmujen määrään [6]. Eli ongelman ratkaisu vaatisi vähintään eksponentiaalisen ajan suhteessa solmujen määrään. Toisin sanoen, jos ongelmassa on syötteenä n solmua, niin aikavaativuus olisi vähintään 2^n

Kauppamatkustajan ongelma kuuluu NP-vaikeisiin ongelmiin. On hyvin epätodennäköistä, että kyseiselle ongelmalle on edes olemassa polynomiallista algoritmia [6].

NP-vaikeata ongelmaa voidaan yrittää lähestyä kahdella tapaa. Joko halutaan optimaalinen ratkaisu, jolloin saatetaan joutua kuluttamaan paljon aikaa. Toinen vaihtoehto on tinkiä optimaalisuudesta. Tällöin voidaan saada ratkaisu nopeasti, mutta ratkaisun laatu voi olla huono [6].

2.4 Kauppamatkustajan ongelma sovelletussa matematiikassa

Kauppamatkustajan ongelman tutkiminen on auttanut löytämään yleiskäytännöllisiä tekniikoita sovelletussa matematiikassa [7]. Se on auttanut kehittämään optimointitekniikan *mixed-integer programming* (MIP), joka sisältää lineaarisen optimointiongelman, mutta lisärajauksena on se, että jotkut muuttujista ovat kokonaislukuja. Tämä on auttanut MIP-optimointitekniikkaa soveltumaan ongelmiin, joissa on diskreettejä valintoja.

Kauppamatkustajan ongelma on myös *Branch and bound* -hakumetodin luonnin takana [7]. Branch and boundissa jokainen askel jakaa haettavan alueen kahteen tai useampaan osaongelmaan. Tarkoituksena on, että nämä osaongelmat ovat helpompia ratkaista kuin alkuperäinen ongelma. Bound vaiheessa taas poistetaan sellaiset osaongelmat, joiden hakeminen kokonaan on turhaa. Eli kyseisessä vaiheessa poistetaan sellaiset osaongelmat, joiden ratkaisu ei voi olla parempi kuin jo laskettu osaongelma.

Kauppamatkustajan ongelma on ollut mukana myös heurististen hakualgoritmien kehityksessä [7]. Heurististen algoritmien tarkoituksena on ratkaista ongelma nopeasti. Tarkoituksena olisi saada myös suhteellisen hyvä ratkaisu. Tuloksen laatu ei välttämättä ole kuitenkaan hyvä.

Kauppamatkustajan ongelmalla on ollut rooli DNA-laskennassa [7]. Vuonna 1994 Leonard Adleman aloitti kyseisen alan tutkimuksellaan, jossa ratkaistiin seitsemän

kaupungin kauppamatkustajan ongelma DNA-laskennalla [12]. Pieneen määrään DNA:ta pystytään tallentamaan paljon informaatiota. Tutkijoiden mukaan, tulevaisuudessa voi olla mahdollista käyttää molekulaarisia laitteita sellaisten ongelmien ratkaisussa, joita tavan tietokoneilla ei pystytä ratkaisemaan [7].

2.5 Kauppamatkustajan ongelman käytännön sovelluksia

Kauppamatkustajan ongelmalle on monia käytännön sovelluksia, sillä monet ongelmat voidaan muokata sopimaan kauppamatkustajan ongelmalla ratkaistavaksi [6].

Yksi selvä sovellus on logistiikka, josta esimerkkinä on *ajoneuvojen reititys*. Ongelmana on se, että miten kuluttaa vähiten polttoainetta, kun on olemassa monta ajoneuvoa ja monta asiakasta. On saatava selville millä ajoneuvolla on ajettava kenenkin asiakkaan luo ja missä järjestyksessä. Yleensä rajoitteina voivat olla ajoneuvojen kapasiteetti ja erilaiset aikarajat asiakkaiden tilauksille. [6]

Toinen sovellus on *piirilevyjen poraus*. Erilaisten piirilevyjen läpi tulee porata erikokoisia reikiä, jotta saadaan yhdistettyä eri kerroksien johtimet. Tässä on otettava huomioon, että poranterän vaihto vie aikaa. Eli samankokoiset reiät on syytä porata kerralla ja sitten vaihtaa terää. Ongelma voidaan kuvata kauppamatkustajan ongelmana. Reiät ovat solmuja ja kaaret solmujen välisiä etäisyyksiä Tarkoituksena on minimoida poran liikkumisaika. [8]

Kolmas sovellus on *lentokoneen turbiinien huolto*. Jokaisessa suihkulentokoneen turbiinissa on sisällä suuttimen ohjaussiivekkeitä. Näillä ohjaussiivekkeillä on jokaisella omat ominaisuutensa. Oikeilla säädöillä voidaan saada merkittäviä hyötyvaikutuksia, esimerkkinä tärinän pienentyminen ja polttoaineen kulutuksen vähentyminen. Ohjaussiivekkeiden säätäminen parhaimmalla tavalla voidaan kuvata kauppamatkustajan ongelmana. [8]

Muitakin sovelluksia on kuten kristallien rakenteen analysointi, tietokoneen johdotus, robotin liikkeiden ohjaus [8].

3 Pienin virittävä puu

Pienin virittävä puu eli *minimum spanning tree* (MST) on myös yksi tunnetuimmista kombinatoorisista optimointiongelmista [3]. Pienin virittävä puu on *virittävän puun* erityistapaus.

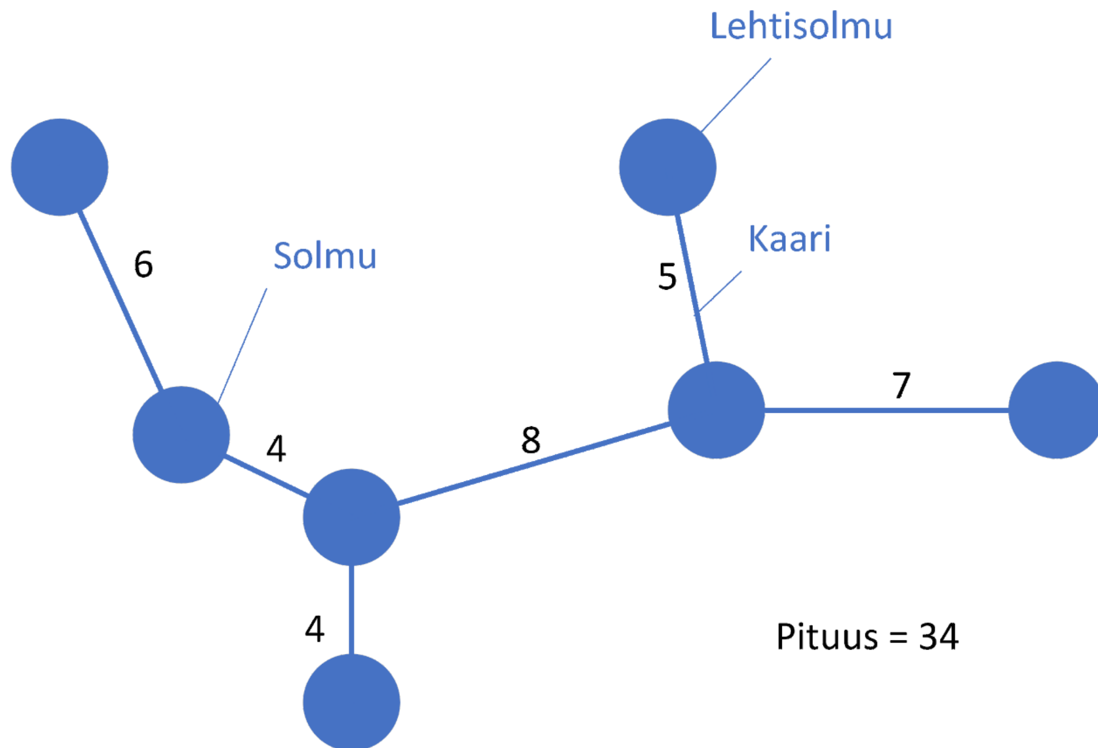
3.1 Virittävä puu

Virittävä puu on verkon osajoukko, jossa jokainen verkon solmu on liittyneenä tähän puuhun [5]. Käytettyjen kaarien yhteen laskettu pituus ei ole välttämättä minimi. Virittävässä puussa voi olla useita *lehtisolmuja* eli solmuja, joihin on yhdistyneenä korkeintaan yksi kaari. Virittävässä puussa ei voi olla *syklejä* eli kierroksia. Virittävä puu on siitä hyödyllinen, että sitä voidaan käyttää moneen asiaan. Virittävällä puulla voidaan luoda harvempi aliverkko, joka kuitenkin muistuttaa paljon alkuperäistä verkkoa. Virittävä puu on tärkeä suunniteltaessa tehokkaita reititysalgoritmeja. Lisäksi joihinkin laskennallisesti vaikeisiin ongelmiin voidaan saada lähellä optimia oleva ratkaisu. Virittäviä puita voidaan soveltaa myös dataverkkojen suunnittelussa [5].

3.2 Pienin virittävä puu

Pienimmän virittävän puun ero tavalliseen virittävään puuhun on se, että siinä käytettyjen kaarien yhteenlaskettu pituus on pienin mahdollinen [5]. Pienimmälle virittävälle puulle löytyy paljon käyttökohteita. Optimaalisen ratkaisun pystyy muodostamaan aina *ahneella (greedy)* algoritmilla kuten Kruskalin algoritmilla [4]. Kuitenkin on otettava huomioon, että ratkaisu ei välttämättä ole ainut mahdollinen. Verkosta saattaa pystyä muodostamaan useamman erilaisen virittävän puun, joiden

kaikkien painot ovat yhtä pieniä. Kuvassa 4 näkyy pienin virittävä puu. Siinä jokainen solmu on yhdistettynä ja kaarien pituudet ovat mahdollisimman pienet.



Kuva 4: Pienin virittävä puu

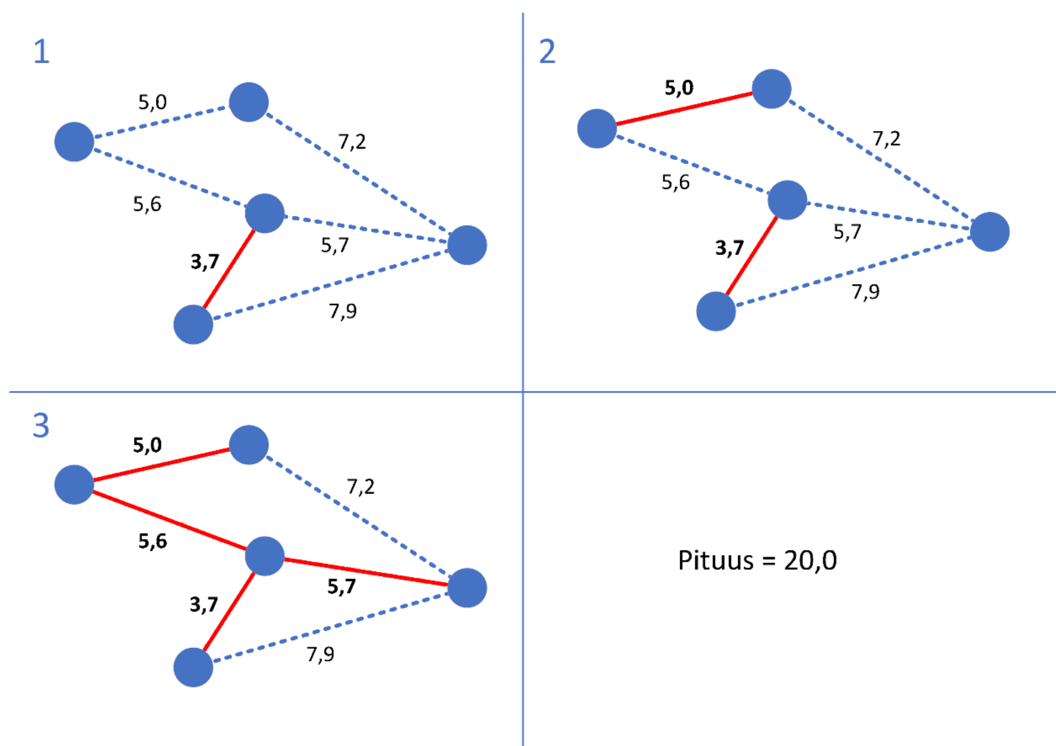
3.3 Historiaa

Ensimmäinen tunnettu algoritmi pienimmän virittävän puun rakentamiseen on vuodelta 1926 oleva *Borůvka:n algoritmi* [9], joka keksittiin huomattavasti ennen Kruskalin (1956) [4] ja Primin (1957) algoritmeja [10]. Borůvkan algoritmissa yksittäisessä vaiheessa jokainen solmu valitsee itsestään lähtevän lyhimmän kaaren lopputulokseen. Näin saadaan alijoukkoja, jotka sitten yhdistetään samalla tavalla lyhimmällä kaarella [5]. Kuitenkin sekä Primin että Kruskalin algoritmeja pidetään ensimmäisinä tehokkaina ratkaisuina pienimmän virittävän puun rakentamiseen [3]. Primin algoritmissa alussa valitaan satunnainen solmu. Sen jälkeen valitaan lyhin kaari, joka liittyy kyseiseen solmuun. Algoritmin jokaisessa vaiheessa valitaan lyhin kaari, joka liittyy lähimpään solmuun, joka ei ole jo lopputuloksessa [5]. Kruskalin algoritmia tutkitaan enemmän seuraavassa kappaleessa.

3.4 Kruskalin algoritmi

Kruskalin algoritmi on yksi tunnetuimmista ratkaisuista pienimmän virittävän puun rakentamiseen. Algoritmin toimintaa voidaan kuvata yksinkertaisesti niin, että valitaan aina lyhin kaari verkosta. Tässä on kuitenkin muutamia ehtoja. Syklejä ei saa syntyä jo valittujen kaarien kanssa [4]. Lisäksi kaari ei saa olla jo ennestään valittu [4]. Syklit voidaan tunnistaa osajoukkojen avulla. Mikäli lisättävän kaaren alku- ja loppusolmut kuuluvat samaan osajoukkoon, niin lisättävän kaaren lisääminen synnyttäisi syklin. Algoritmi ei tarvitse täydellistä verkkoa toimiakseen eli verkko voi olla monenlainen.

Kuten kuvasta 5 nähdään, ensimmäiseksi on valittu lyhin kaari. Tämä lisätään lopputulokseen. Seuraavaksi valitaan toiseksi lyhin kaari. Kaaren lisääminen on sallittua, sillä lisättävän kaaren alku- ja loppusolmut ovat eri osaverkoista. Lyhimmän kaaren valitsemista jatketaan, kunnes lopputulos on saatu. Alimpana kuvassa näkyy valmis pienin virittävä puu. Valmiissa lopputuloksessa kaikki verkon solmut sisältyvät puuhun ja syklejä ei ole.



Kuva 5: Kruskalin algoritmin vaiheet

3.5 Pienimmän virittävän puun yhteys kauppamatkustajan ongelmaan

Pienimmän virittävän puun avulla voidaan luoda ratkaisu kauppamatkustajan ongelmaan. Esimerkiksi Christofideksen algoritmi [21] toimii niin, että ensin luodaan pienin virittävä puu ja sen jälkeen puuta muokataan, jotta saadaan TSP-ratkaisu. Pienimmän virittävän puun luonnin jälkeen etsitään solmut, joista lähtee pariton määrä kaaria. Näistä solmuista tehdään oma aliverkko. Aliverkon lähimmät solmut yhdistetään toisiinsa pareiksi lyhimmillä mahdollisilla kaarilla. Sen jälkeen aliverkko yhdistetään pienimpään virittävään puuhun. Tämän jälkeen puusta muodostetaan *Eulerin polku*. Sen jälkeen ylimääräisi kaaria poistetaan ja lehtisolmuja yhdistetään, kunnes saadaan TSP-ratkaisu. [14]

3.6 MST-solmujen määrän yhteys TSP:n vaikeuden arviointiin

Pienimmän virittävän puun solmujen määrän yhteyttä kauppamatkustajan ongelmaan on tutkittu [16]. Tutkijoiden tarkoituksena oli arvioida TSP:n vaikeutta MST-solmukohtien määrän perusteella. Tämä tehtiin luomalla ensin pienin virittävä puu ja sen jälkeen laskemalla solmukohdat puusta. Solmukohtia ovat solmut, joista lähtee kolme tai useampi kaarta. Tutkijoiden hypoteesi oli, että mitä enemmän solmukohtia, sitä hankalampi TSP-reitti oli luoda. Lopputulokseksi saatiin, että MST-solmukohtien määrä korreloi erittäin hyvin optimaalisen TSP-reitin muodostamiseen tarvittavan ajan kanssa [16].

4 Kruskalin algoritmi avoimen kierroksen kauppamatkustajan ongelmaan

Kruskalin algoritmia voidaan soveltaa muutamalla muutoksella avoimen kierroksen kauppamatkustajan ongelmaan. Seuraavaksi esitetään yksityiskohtaisesti, miten tämä tapahtuu. Algoritmista käytetään nimeä Kruskal-TSP.

4.1 Algoritmin toiminta

Algoritmin ensimmäisessä vaiheessa luetaan syötetiedosto. Syötetiedosto on myöhemmissä esimerkeissä tekstitiedosto, joka sisältää jokaisen solmun koordinaatit. Tiedoston data luetaan matriisiin. Tämän jälkeen luodaan *täydellinen verkko* luetusta datasta. Täydellisessä verkossa on kaari jokaisesta solmusta jokaiseen toiseen solmuun. Täydellinen verkko luodaan käymällä läpi kaikki verkon solmut ja laskemalla etäisyydet verkon solmuista jokaiseen toiseen verkon solmuun. Etäisyyksien laskemiseen käytetään Haversine-funktiota $d = 2r \arcsin\left(\sqrt{sin^2\left(\frac{\varphi_1 + \varphi_2}{2}\right) \cos(\varphi_1) \cos(\varphi_2)\, sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$.

Kyseisessä kaavassa d on lopputulos eli kahden pisteen välinen matka pallolla. Kaavan r on maapallon säde, joka on 6356,752 Suomessa. φ_1 on pisteen 1 leveysaste radiaaneina ja φ_2 on pisteen 2 leveysaste radiaaneina. λ_1 on pisteen 1 pituusaste radiaaneina ja λ_2 on pisteen 2 pituusaste radiaaneina.

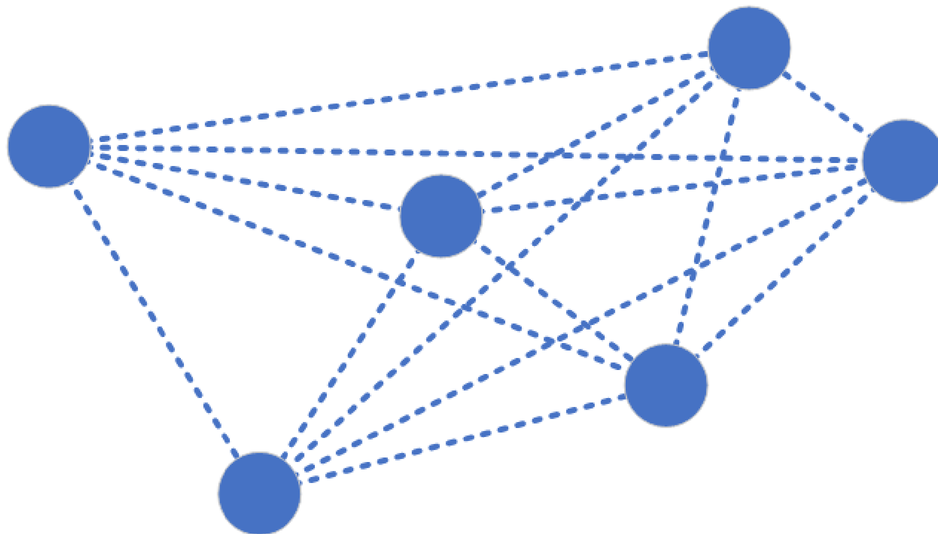
Seuraavassa vaiheessa täydellisen verkon kaaret järjestetään pienimmästä suurimpaan, sekä jokainen verkon solmu lisätään omaan osajoukkoonsa. Näiden valmistelujen jälkeen siirrytään varsinaiseen reitin luomiseen.

Aluksi valitaan lyhin kaari ja lisätään se tulosverkkoon sekä yhdistetään kyseisen kaaren solmut samaan osajoukkoon. Seuraavaksi valitaan toiseksi lyhin kaari. Jokaisen valitun kaaren kohdalla tarkistetaan aiheuttaako kyseinen kaari syklin eli kierroksen tai haarauman lopputulokseen. Mikäli näin tapahtuu, kaarta ei lisätä lopputulokseen vaan se ohitetaan. Syklin ja haaraumien tarkistus tapahtuu osajoukkoja

tarkastelemalla, sekä tarkistamalla, onko lisättävä kaari yhdistymässä verkon päätepisteisiin. Lyhimmän mahdollisen kaaren valitsemista jatketaan, kunnes TSP-reitti on valmis.

4.2 Esimerkki algoritmin toiminnasta

Kuten esimerkkikuvasta 6 näkyy, täydellisessä verkossa jokainen solmu on suoraan yhteydessä toiseen verkon solmuun. Tämä on tärkeää, jotta saadaan selville jokaisen kaaren pituus. Verkon luonnin jälkeen verkossa olevat kaaret järjestetään pienimmästä suurimpaan. Myöskin jokaiselle verkon solmulle luodaan oma osajoukko sen kaarista. Tämän jälkeen valitaan pienin kaari, ja tarkistetaan, voidaanko se lisätä lopputulokseen.



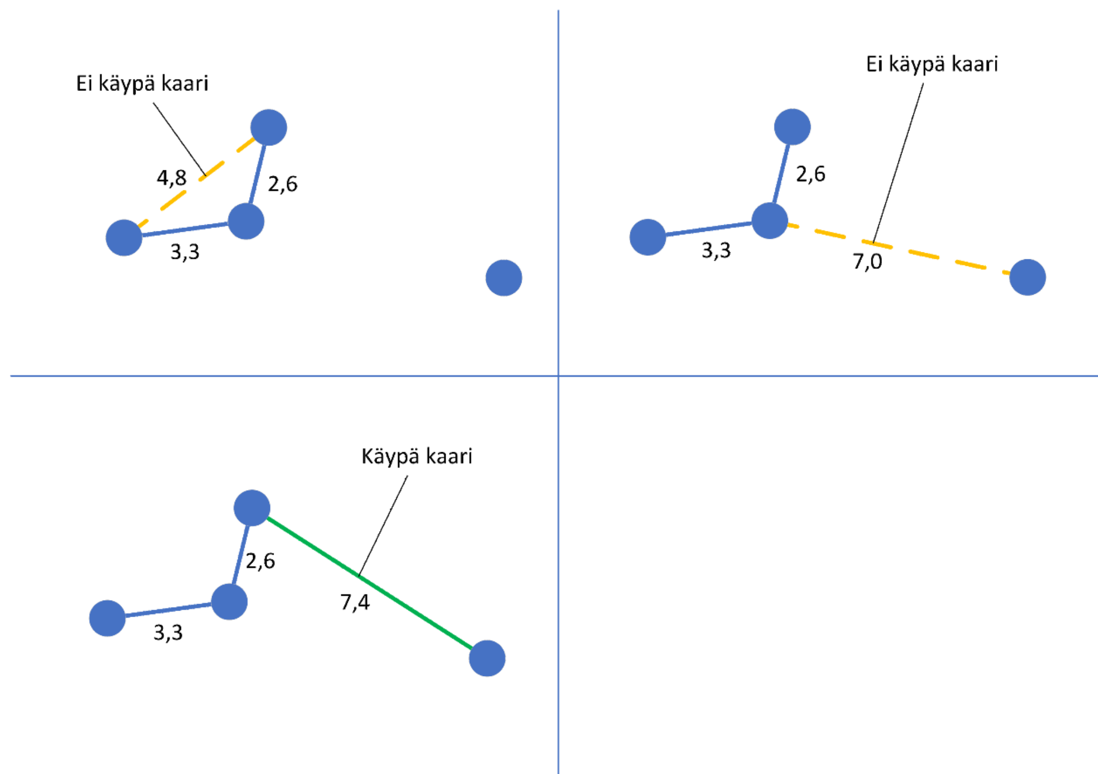
Kuva 6: Esimerkki täydellisestä verkosta

Kaarta lisättäessä on täytettävä kaksi lisäysehtoa:

1. Valittavan kaaren yhdistämät solmut eivät saa olla jo valmiiksi samassa osajoukossa
2. Valittava kaari saa yhdistää vain osajoukkojen päätepisteitä

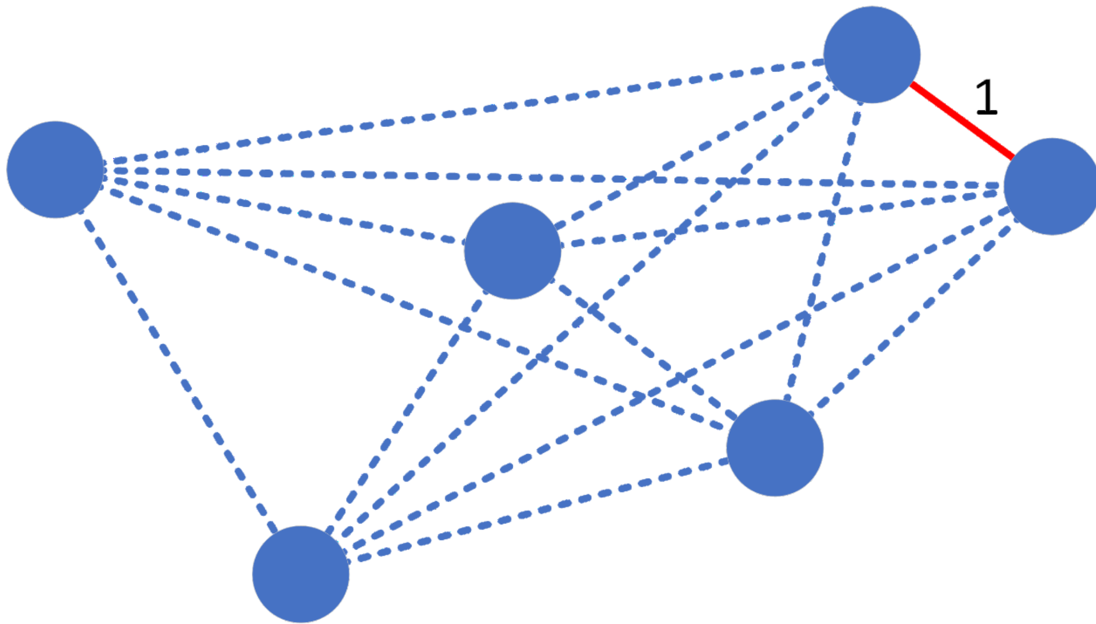
Kuvassa 7 on esimerkki lisäysehdoista. Kuvasta näkyy kuinka lyhimmän kaaren valitseminen aiheuttaisi syklin. Osajoukkoja tarkastelemalla saadaan kuitenkin selville, että kyseistä kaarta ei voida lisätä lopputulokseen.

Kuvassa 7 näkyy myös, kuinka kaaren lisäys sallitaan vain silloin kun lisättävä kaari liittyy osajoukkojen päätepisteisiin. Tämä on merkittävä ero normaaliin Kruskalin algoritmiin, sillä siinä sallitaan kaaren lisäys myös osajoukon keskelle. Tuloksena on, että toisinaan joudutaan valitsemaan pidempiä kaaria kuin normaalisti pienintä virittävää puuta muodostettaessa.



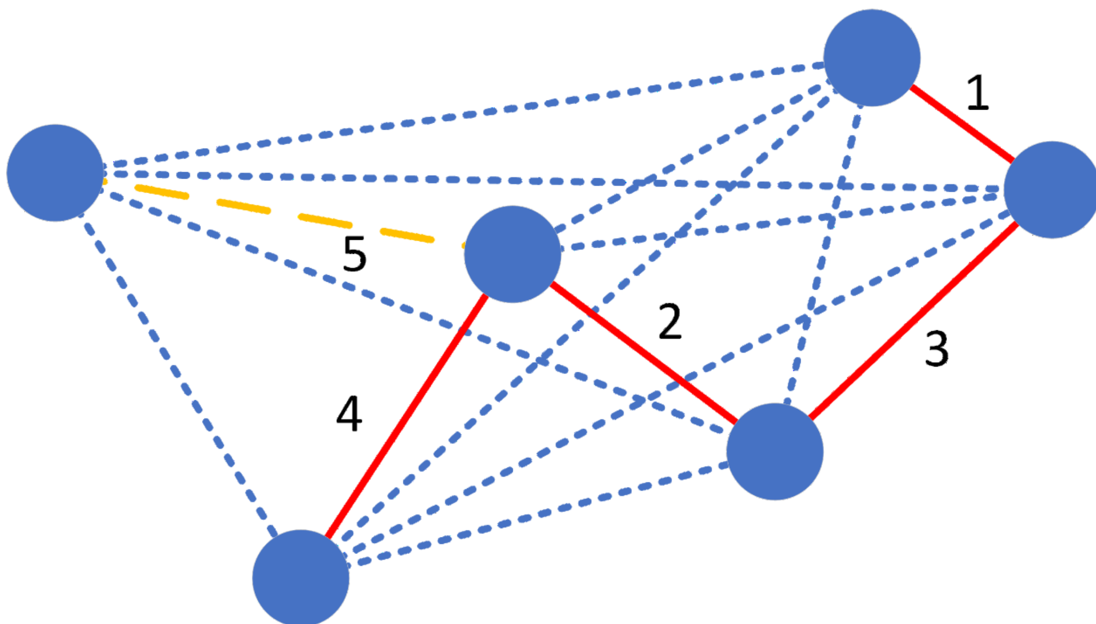
Kuva 7: Esimerkki syklistä, haaraumasta ja sallitusta kaaresta

Jos lisätty kaari ei aiheuta sykliä verkossa, eikä myöskään luo haaraumaa, niin kaari voidaan lisätä tulokseen. Ensimmäisen kaaren kohdalla näin ei voi käydä, kuten kuvassa 8, joten seuraavaksi siirrytään toiseksi pienimpään kaareen.



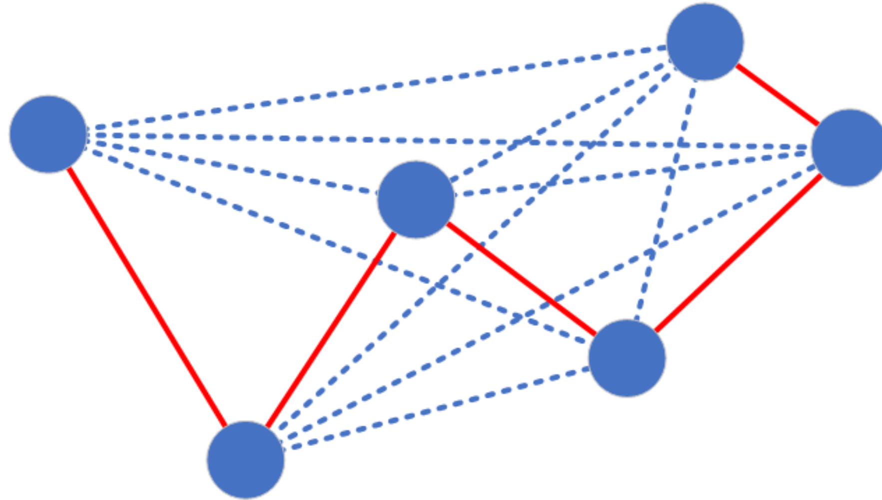
Kuva 8: Lyhimmän kaaren valitseminen

Kuvasta 9 näkyy, että myöhemmässä vaiheessa eli viidettä kaarta lisättäessä, algoritmi valitsisi kaaren, joka muodostaisi haarauman (kuvassa keltaisella pitkällä katkoviivalla). Haarauma kuitenkin tunnistetaan laskemalla, kuinka monta kaarta liittyy lisättävän kaaren alku- ja loppusolmuihin. Algoritmi tulee siihen tulokseen, että lisättävän kaaren toiseen solmuun on jo liittyneenä kaksi kaarta. Näin nyt valittu kaari täytyy ohittaa.



Kuva 9: Haarauma

Kun päästään loppuun asti, käydään läpi tulosverkko ja tallennetaan reitti sekä kokonaispituus tulostiedostoon. Reitistä etsitään lehtisolmu ja se asetetaan lähtösolmuksi. Sen jälkeen käydään läpi solmut ja lisätään ne kulkujärjestyksessä varsinaiseen lopputulokseen.



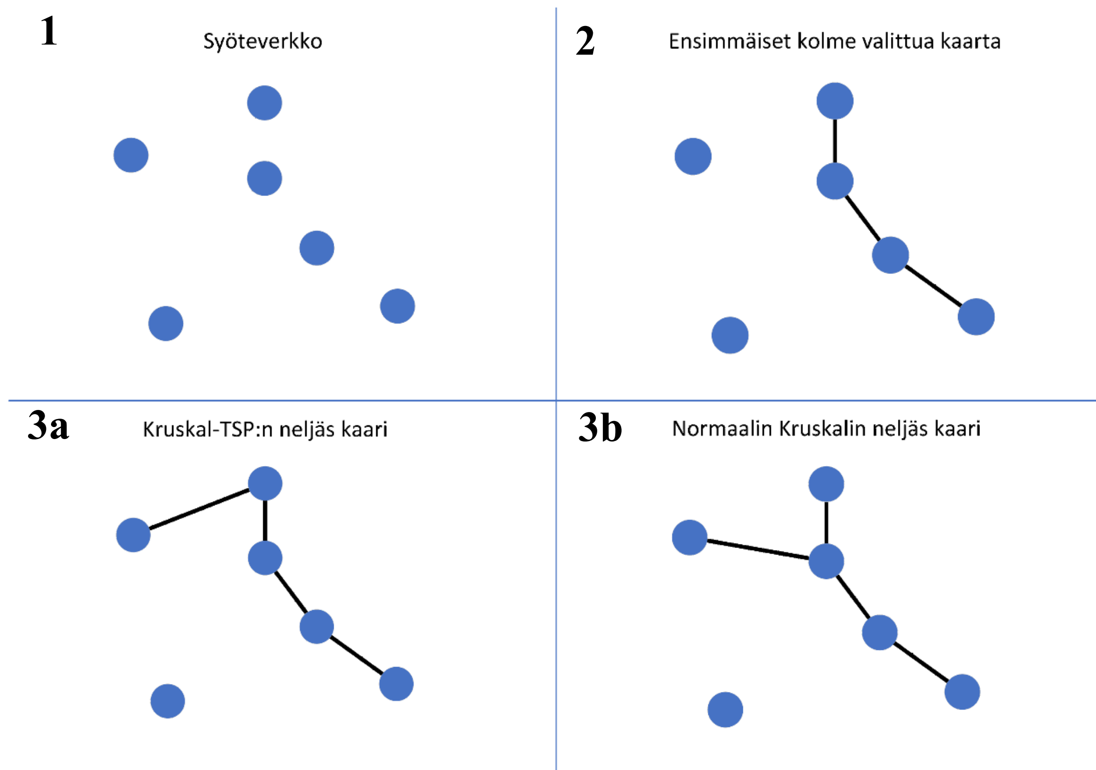
Kuva 10: Lopullinen reitti

4.3 Kruskal-TSP-algoritmin ero verrattuna Kruskalin algoritmiin

Kruskal-TSP-algoritmi muistuttaa hyvin paljon normaalia Kruskalin algoritmia. Merkittäviä eroja kuitenkin on. Seuraavassa kuvataan vaihe vaiheelta, kuinka kaarien valinta eroaa samanlaisella syöteverkolla. Tämän jälkeen vertaillaan, miten lopputulos eroaa algoritmien välillä ja kuinka lopputulosten kokonaispituudet eroavat.

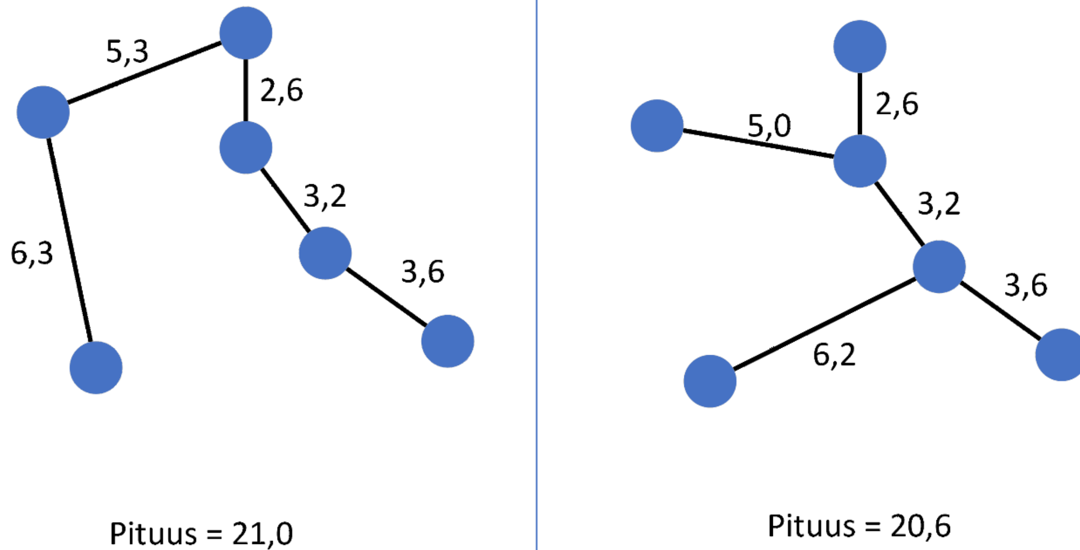
Kuten kuvan 11 yläosista näkyy, niin kummallakin algoritmilla valitaan aluksi samat kolme lyhintä kaarta. Samat kaaret pystytään valitsemaan Kruskal-TSP:ssä, sillä kaaret voidaan aina lisätä edellisen alipuun päihin, jolloin ei aiheudu haaraumia eikä syklejä.

Esimerkkiverkon neljäs kaari valitaan kuitenkin jo eri tavalla. Normaalissa Kruskalin algoritmissa valitaan yksinkertaisesti seuraavaksi lyhin kaari, joka ei aiheuta sykliä. Kruskal-TSP:ssä ei voida tehdä näin, sillä verkkoon syntyisi haarauma. Algoritmi valitsee siis lyhimmän kaaren, joka voidaan lisätä puun reunoihin.



Kuva 11: Kaarien valitseminen

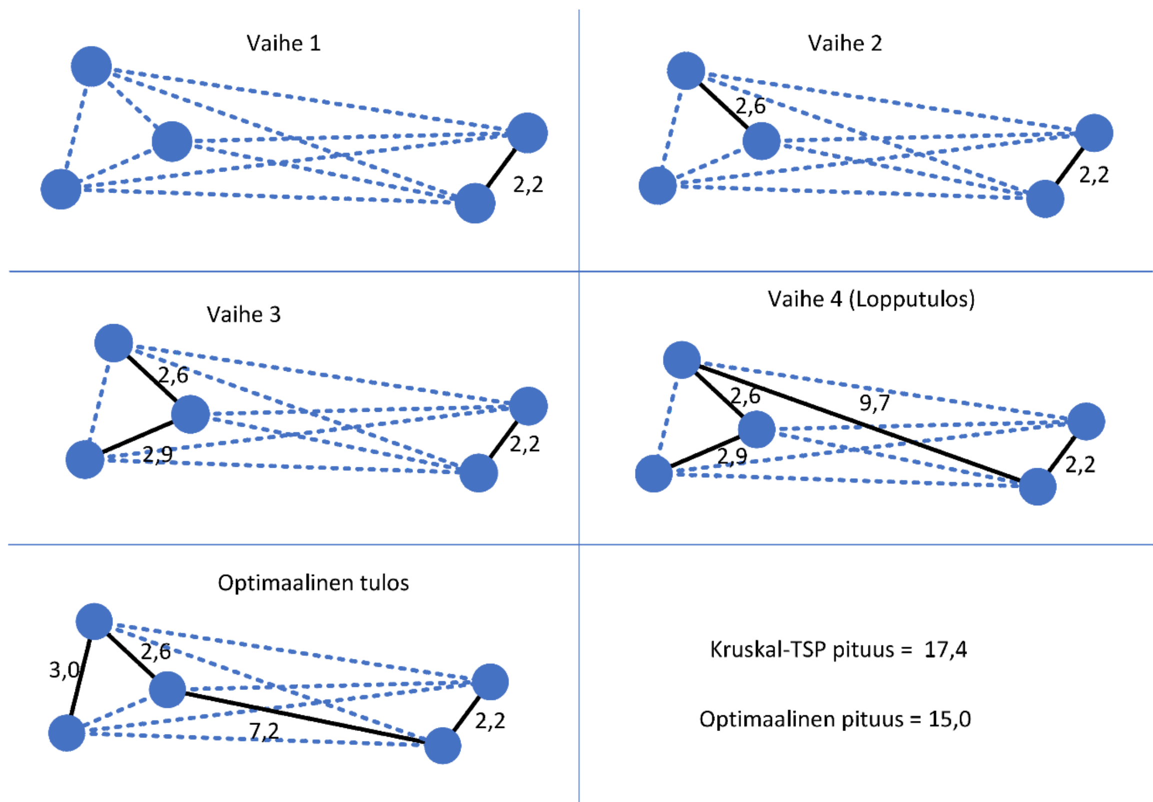
Kuvasta 12 näkyy, miten lopputulokset eroavat. Normaalissa Kruskalin algoritmissa valitaan viimeisessäkin vaiheessa lyhin kaari, joka ei aiheuta sykliä. Kruskal-TSP:ssä ei voida ottaa samaa kaarta, sillä myös se muodostaisi haarauman. Kruskal-TSP valitsee siis tässäkin vaiheessa lyhimmän kaaren, joka voidaan lisätä olemassa olevan puun päihin. Kruskal-TSP:n antaman tulosverkon kokonaispituudeksi tulee 21,0. Normaalin Kruskalin algoritmin antaman tulosverkon pituus on 20,6. Kruskal-TSP:n tulos on isompi, mutta ero on melko pieni eli vain 1,9 %.



Kuva 12: Kruskal-TSP:n antama tulos verrattuna normaalin Kruskalin algoritmin antamaan tulokseen

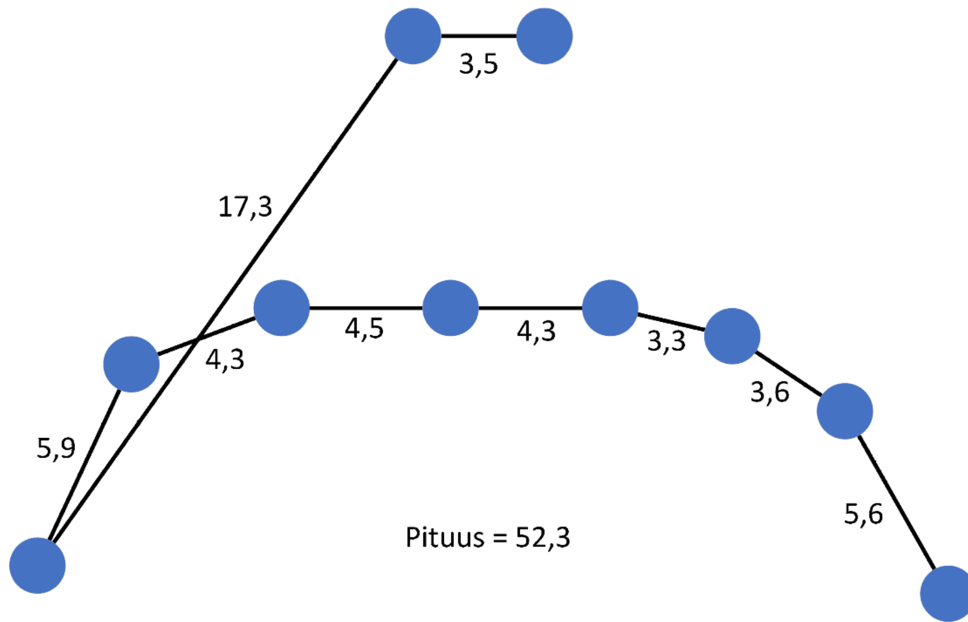
4.4 Esimerkkejä tapauksista, jossa algoritmi ei tuota hyvää ratkaisua

Algoritmilla on olemassa myös tapauksia, jolloin ei päästä hyvään lopputulokseen. kuvassa 13 olevalla verkolla algoritmi ei tuota optimaalista ratkaisua. Verkosta valitaan lyhin kaari ja lisätään se lopputulokseen. Seuraavaksi valitaan toiseksi lyhin kaari. Se ei aiheuta haaraumaa eikä sykliä, joten se voidaan lisätä lopputulokseen. Kolmanneksi lyhimmänkään kaaren valitseminen ei aiheuta sykliä tai haaraumaa, joten se voidaan lisätä lopputulokseen. Neljänneksi lyhin kaari aiheuttaisi syklin, joten se ohitetaan. Viidenneksi ja kuudenneksi lyhimmät kaaret aiheuttaisivat haarauman, joten ne ohitetaan. Lopulta päädytään lyhimpään kaareen, joka ei aiheuta haaraumaa tai sykliä. Kuten kuvasta 13 näkyy optimaalista ratkaisua ei saavuteta.



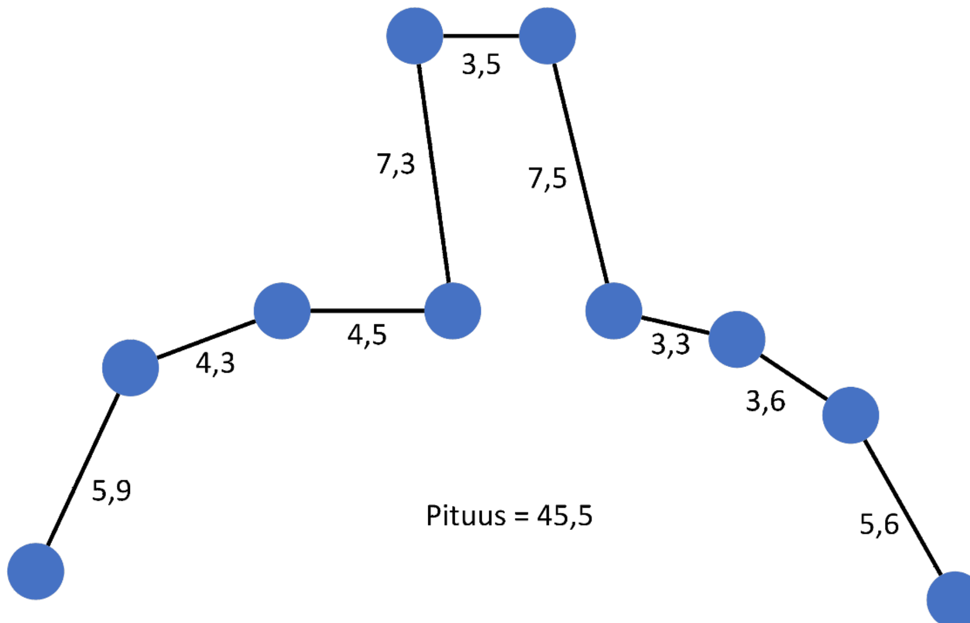
Kuva 13: Esimerkki huonosta lopputuloksesta verrattuna optimaaliseen lopputulokseen

Algoritmi voi tuottaa myös hyvin huonoja ratkaisuja. Seuraavana on esimerkki tästä tapauksesta. Kuvasta 14 näkyy, kuinka algoritmi valitsee aina lyhimpiä kaaria ahneesti, kunnes joudutaan valitsemaan yksi hyvin pitkä kaari. Tämä johtaa risteämiseen ja huonoon lopputulokseen.



Kuva 14: Kruskal-TSP:n tuottama ratkaisu

Kuvassa 15 olevassa optimaalisessa ratkaisussa ei ole risteämiä kuten kuvassa 14 eikä hyvin pitkiä kaaria. Kaarien yhteenlaskettu pituus on huomattavasti vähemmän. Huono tulos Kruskal-TSP:llä on noin 15 % pitempi kuin tämä optimaalinen.

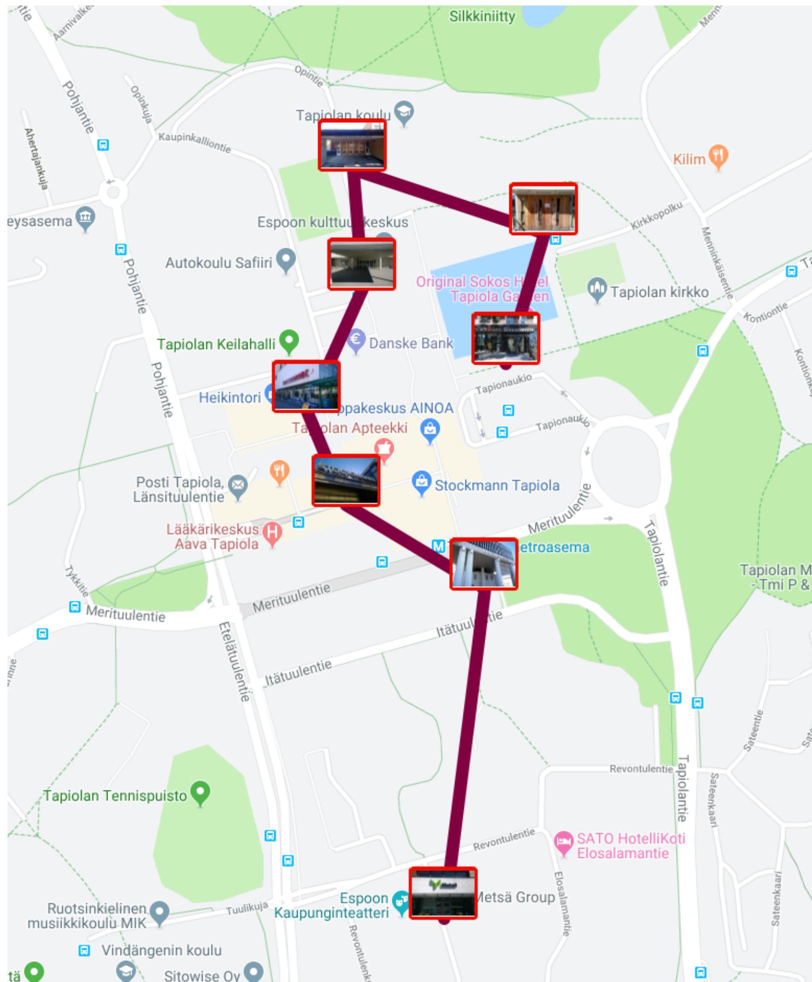


Kuva 15: Optimaalinen ratkaisu

Kuten esimerkkikuvista 14 ja 15 nähdään algoritmi ei tuota hyvää ratkaisua jokaisella verkolla. Algoritmi valitsee kaaria ahneesti lopulliseen reittiin. Tämä saattaa johtaa siihen, että valitaan lopputuloksen kannalta epäedullisia kaaria. Joissain vaiheissa olisi lopputuloksen kannalta edullisempaa valita hieman pidempi kaari, jotta voidaan valita seuraavaksi huomattavasti lyhyempi kaari. Tämä voitaisiin toteuttaa esimerkiksi satunnaisuuden avulla.

4.5 Esimerkki tapauksesta, jossa algoritmi toimii hyvin

Kruskal-TSP-algoritmi toimii kuitenkin usein hyvin. Seuraavana on tällainen esimerkkitapaus. Kuvassa 16 Kruskal-TSP löytää optimaalisen reitin kaikkien solmujen läpikäymiseksi. Optimaalisen reitin pituus on 1345,3 m ja Kruskal-TSP löytää kyseisen reitin. Optimaaliseen lopputulokseen päästään, vaikka kaaria valitaan ahneesti, sillä kaarien valinta ei aiheuta myöhemmin huonoa valintaa. Kuten kuvasta näkyy niin solmujen etäisyydet ja paikat sattuvat sopivasti niin, että ahneet valinnat eivät haittaa.



Kuva 16: Esimerkki Kruskal-TSP:n antamasta optimaalisesta reitistä

4.6 Pseudokoodi ja aikavaativuus

Seuraavaksi esitellään algoritmin tärkeimmät osat pseudokoodin muodossa sekä analysoidaan aikavaativuutta. Ensimmäisenä vuorossa on täydellisen verkon luonti.

Algoritmin alussa tarkistetaan syötematriisin koko sekä lasketaan sen avulla kaarien määrä. Tämän jälkeen luodaan verkko sekä alustetaan läpikäytävien solmujen määrä, aloitussolmu ja kaarten määrä nollaksi. Lisäksi luodaan lista solmujen yhteyksistä. Tästä päästään sisäkkäisiin silmukoihin, joissa lasketaan jokaisen solmun etäisyys jokaiseen toiseen solmuun. Yksittäisen kaaren alku- ja loppusolmu tallennetaan molemmin päin tekstinä *value1* ja *value2* -muuttujiin. Näiden avulla voidaan testata,

että kaarta ei olla lisätty jo aiemmin verkkoon. Mikäli kaarta ei olla jo lisätty niin kaari lisätään verkkoon ja sille merkitään alku- ja loppusolmu sekä pituus. Verkon luonnissa on kaksi sisäkkäistä silmukkaa, joten aikavaativuudesta tulee O(n^2). n on solmujen lukumäärä.

GenerateGraph O(n^2)

```
GenerateGraph(Matrix)

    Dimension <- size of Matrix
    Edges <- calculate from dimension //(dimension * (dimension-1))/2

    Tspgraph <- NewKruskal_tsp (dimension, edges)
    Nodestogothrough <- 0
    Startingnode <- 0
    Edgecount <- 0
    ArraylistOfConnections

    WHILE Nodestogothrough != Dimension
        Destinationnode <- 0
        WHILE Destinationnode != Dimension
            IF Destinationnode != Startingnode
            THEN
                Distance <- HaversineDistance(Startingnode,
                Destinationnode, Matrix)
                Value1 <- StringOfStartingnode + StringOfdestinationnode
                Value2 <- StringOfdestinationnode + StringOfStartingnode

                IF Distance != 0.0 AND ArrayListOfConnections !CONTAINS
                Value1 AND ArraylistOfConnections !CONTAINS Value2
                THEN
                    ADD value1 TO ArraylistOfConnections
                    ADD value2 TO ArraylistOfConnections

                    Tspgraph[Edgecount].source <- startingnode
                    Tspgraph[Edgecount].destination <- destinationnode
                    Tspgraph[Edgecount].weight <- distance

                    Edgecount++

                Destinationnode++

            Startingnode++

            Nodestogothrough++

    RETURN Graph
```

Seuraavana vuorossa on algoritmin pääosa eli reitin muodostaminen. Käytetyistä symboleista V on solmujen määrä verkossa ja e valittujen kaarien määrä. Pseudokoodin alussa luodaan taulukko, johon lisätään kaikki kaaret. Tämän jälkeen kaaret järjestetään lyhimmästä pisimpään. Sen jälkeen luodaan taulukko alijoukoille. Jokainen solmu lisätään omaan alijoukkoonsa. Tämän jälkeen alkaa pääsilmukka, jossa valitaan lyhin kaari taulukosta, sekä etsitään kaaren alku- ja loppusolmu. Tämän jälkeen alkaa toinen silmukka, jossa etsitään haaraumia. Sen jälkeen tarkistetaan, että voidaanko kaari lisätä lopputulokseen. Mikäli voidaan, niin kaari lisätään lopputulokseen ja valitun kaaren solmujen alijoukot yhdistetään. Lopulta valmis lopputulos palautetaan.

Kruskal-TSP pääfunktiossa on kaksi sisäkkäistä while-silmukkaa, joten tästä saadaan jo aikavaativuus O(n^2). Lisäksi kaarien määrä suhteessa solmuihin on neliöllinen. Apufunktiokutsut eivät ole enemmän kuin O(n) ja ne ovat ulomman while-silmukan sisällä, joten pääfunktion aikavaativuudeksi tulee O(n^3)

KruskalTSP O(n^3)

```
    ArrayOfResult
    FOR i <- 0 TO V-1
        ArrayOfResult[i] <- NewEdge()
    SortEdges()
    ArrayOfSubsets

    FOR i <- 0 TO V-1
        ArrayOfSubsets[i] <- NewSubset()

    FOR v <- 0 TO v = V-1
        ArrayOfSubsets[v].parent <- v
        ArrayOfSubsets[v].rank <- 0
    e <- 0
    i <- 0
    LoopLength <- 0
    WHILE e < V -1

        NextEdge <- NewEdge()
        NextEdge <- Edges[i++]

        x <- Find(ArrayofSubsets, NextEdge.source)
        y <- Find(ArrayOfSubsets, NextEdge.destination)

        Samenodefound <- 0
        Samenodefound2 <- 0
        Loops <- 0
```

```
        WHILE Loops < LoopLength
            IF ArrayOfResult[Loops].source CONTAINS NextEdge.source
            THEN
                Samenodefound++
            IF ArrayOfResult[Loops].source CONTAINS NextEdge.destination
            THEN
                Samenodefound2++
            IF ArrayOfResult[Loops].destination CONTAINS NextEdge.source
            THEN
                Samenodefound++
            IF ArrayOfResult[Loops].destination CONTAINS
            NextEdge.destination
            THEN
                Samenodefound++

            Loops++

        IF (x != y) AND (samenodefound < 2) AND (samenodefound2 < 2)
            LoopLength++
            Result[e++] <- next_edge
            Union(subsets,x,y)

    RETURN Result
```

Apufunktiokutsussa Find etsitään rekursiivisesti mihin alijoukkoon solmu kuuluu. Rekursio aiheuttaa apufunktiolle O(n) aikavaativuuden.

Find O(n)

```
find(ArrayOfSubsets, nodenumber)

    IF (ArrayOfSubsets[nodenumber].parent != nodenumber) THEN

        ArrayOfSubsets[nodenumber].parent <- find(ArrayOfSubsets,
            ArrayOfSubsets[nodenumber].parent)

    RETURN ArrayOfSubsets[nodenumber].parent
```

Apufunktiokutsussa Union yhdistetään kaksi alijoukkoa toisiinsa. Find-funktion kutsuminen aiheuttaa O(n) aikavaativuuden.

Union O(n)

```
union(ArrayOfSubsets,nodenumber,nodenumber2)

xroot <- find(ArrayOfSubsets, nodenumber)
yroot <- find(ArrayOfSubsets, nodenumber2)

    IF (ArrayofSubsets[xroot].rank < ArrayOfSubsets[yroot].rank) THEN
```

```
        ArrayOfSubsets[xroot].parent <- yroot

    ELSE IF (ArrayOfSubsets[xroot].rank > ArrayOfSubsets[yroot].rank) THEN

        ArrayOfSubsets[yroot].parent <- xroot

    ELSE

        ArrayOfSubsets[yroot].parent <- xroot;
        ArrayOfSubsets[xroot].rank++
```

Algoritmi tiivistettynä:

```
GenerateGraph() O(n^2)
KruskalTSP() total O(n^3)
            SortEdges() O(n^2logn^2)
            While-loop (until enough edges in result) O(n^2)
                        Find() O(n)
                        Find() O(n)
                        While-loop (test for branching) O(n)
PathFinder() O(n^2)
DistanceCount() O(n)
```

Algoritmissa on kaksi vaihetta, jotka vaikuttavat eniten aikavaativuuteen. Ensimmäinen niistä on täydellisen verkon luominen. Täydellisen verkon luomisessa on kaksi sisäkkäistä silmukkaa, joka aiheuttaa itsessään $O(n^2)$ aikavaativuuden.

Toinen eniten aikavaativuuteen vaikuttava vaihe on itse TSP:n ratkaisu. Siinä tuloskaaritaulukon luominen sekä siihen kaarien alustaminen on aikavaativuudeltaan $O(n)$, kaarien järjestäminen pienimmästä suurimpaan $O(n^2 logn^2)$ ja alijoukkojen luonti $O(n)$ sekä sopivien kaarien valitseminen $O(n^3)$, koska kaaria on neliöllinen määrä verrattuna solmuihin. Tämä aiheuttaa tähän osaan $O(n^3)$ aikavaativuuden.

Lopuilla vaiheista ei ole juurikaan vaikutusta, sillä polun löytäminen on $O(n^2)$ ja pituuden laskeminen $O(n)$. Tästä voidaan laskea, että lopulliseksi aikavaativuudeksi jää $O(n^3)$.

5 Satunnaistettu Kruskal-TSP

Kruskal-TSP-algoritmia voidaan muokata lisää, jotta saadaan aikaan parempia tuloksia. Parempia tuloksia voidaan saada käyttämällä satunnaisuutta avuksi. Seuraavaksi esitetään, kuinka tämä tapahtuu.

5.1 Algoritmin toiminta ja pseudokoodi

Satunnaistetussa versiossa algoritmi ei valitse aina lyhintä mahdollista kaarta. Algoritmi valitsee kaaren satunnaisesti kolmesta lyhimmästä kaaresta. Tällöin on mahdollista saada aikaan parempia ratkaisuja kuin ilman satunnaisuutta. Ongelmana on kuitenkin se, että satunnaistaminen hyvin useasti tuottaa huonompia ratkaisuja. Tätä pystytään hieman korjaamaan toistamalla algoritmia useampaan kertaan ja ottamaan lopulliseksi ratkaisuksi paras lopputulos. Satunnaistetun version pseudokoodi on muuten samanlainen kuin perus-Kruskal-TSP:ssä, mutta se sisältää muutaman lisäyksen, jotka esitellään seuraavaksi.

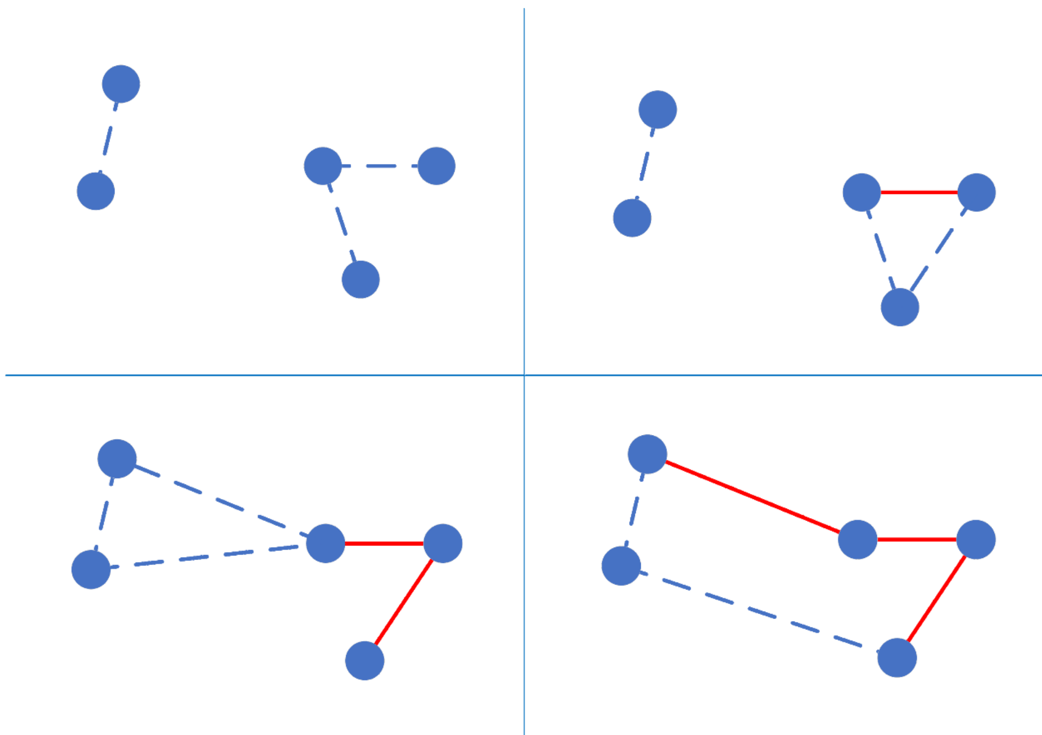
Randomized KruskalTSP

```
IF random = 1 THEN
   size <- length of Edges
   NewIndexesArray

IF random = 1 THEN
   NextEdge = PickRandomizedEdge(e)


PickRandomizedEdge(e)
    NewChosenIndexesArray
    NumbersToTake <- length of ChosenIndexesArray
    NumbersTaken <- 0
    j <- 0
    WHILE NumbersTaken < NumbersToTake
        IF e = V-2
        THEN
            NumbersToTake <- 1

        IF Indexes[j] = 0
        THEN
            ChosenIndexes[NumbersTaken] <- j
            NumbersTaken++
        j++
```

```
NewRandom()
Number <- Random.Integer(NumbersToTake)
ChosenIndex <- ChosenIndexes[Number]
NextEdge = Edges[ChosenIndex]
Indexes[ChosenIndex] <- 1

Return NextEdge
```

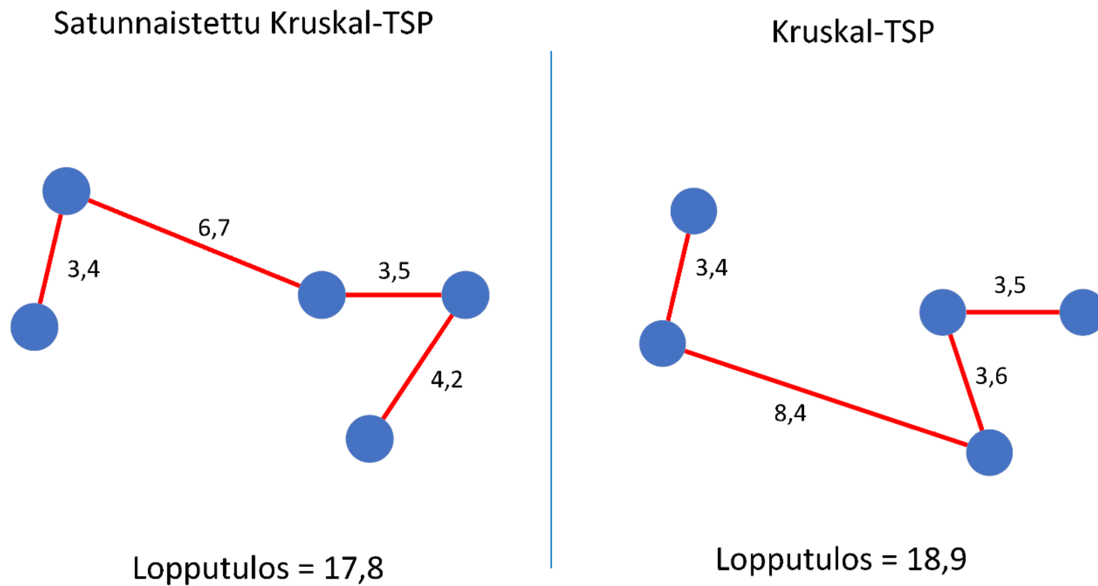
5.2 Esimerkki satunnaistetun Kruskal-TSP-algoritmin toiminnasta

Kuvasta 17 näkyy, että satunnaistetulla Kruskal-TSP-algoritmilla on mahdollista päästä parempaan lopputulokseen kuin perus-Kruskal-TSP:llä. Kuvasta näkyy kuinka satunnaistettu Kruskal-TSP valitsee aina satunnaisesti kolmesta lyhimmästä kaaresta (merkitty katkoviivalla). Kun yksi kaari on valittu kolmesta lyhimmästä, siirrytään seuraavaan vaiheeseen, jossa taas valitaan sattumanvaraisesti kaari kolmesta lyhimmästä. Samaa jatketaan, kunnes TSP-reitti on valmis. Kun valmiista reitistä puuttuu enää yksi kaari, voidaan valita suoraan lyhin kaari. Satunnaisuutta ei tarvita, sillä lyhimmän valitseminen takaa parhaimman lopputuloksen.



Kuva 17: Verkko, johon on merkittynä kolme lyhintä kaarta

Kuvassa 18 vasemmalla on mahdollinen lopputulos käyttäen satunnaisuutta. Tämä lopputulos on vain yksi mahdollisista. Tässä tapauksessa se on parempi kuin perus-Kruskal-TSP:n antama lopputulos. Jotta kyseiseen lopputulokseen päästään tarvitaan useampia algoritmin toistoja. Kuvan 10 oikealla puolella näkyy, että perus-Kruskal-TSP:n antama lopputulos on huonompi.



Kuva 18: Mahdollinen valmis lopputulos satunnaistetulla Kruskal-TSP:llä (vasen) sekä perus-Kruskal-TSP:n antama lopputulos (oikea)

6 Kokeelliset tulokset

6.1 Testiasetelma

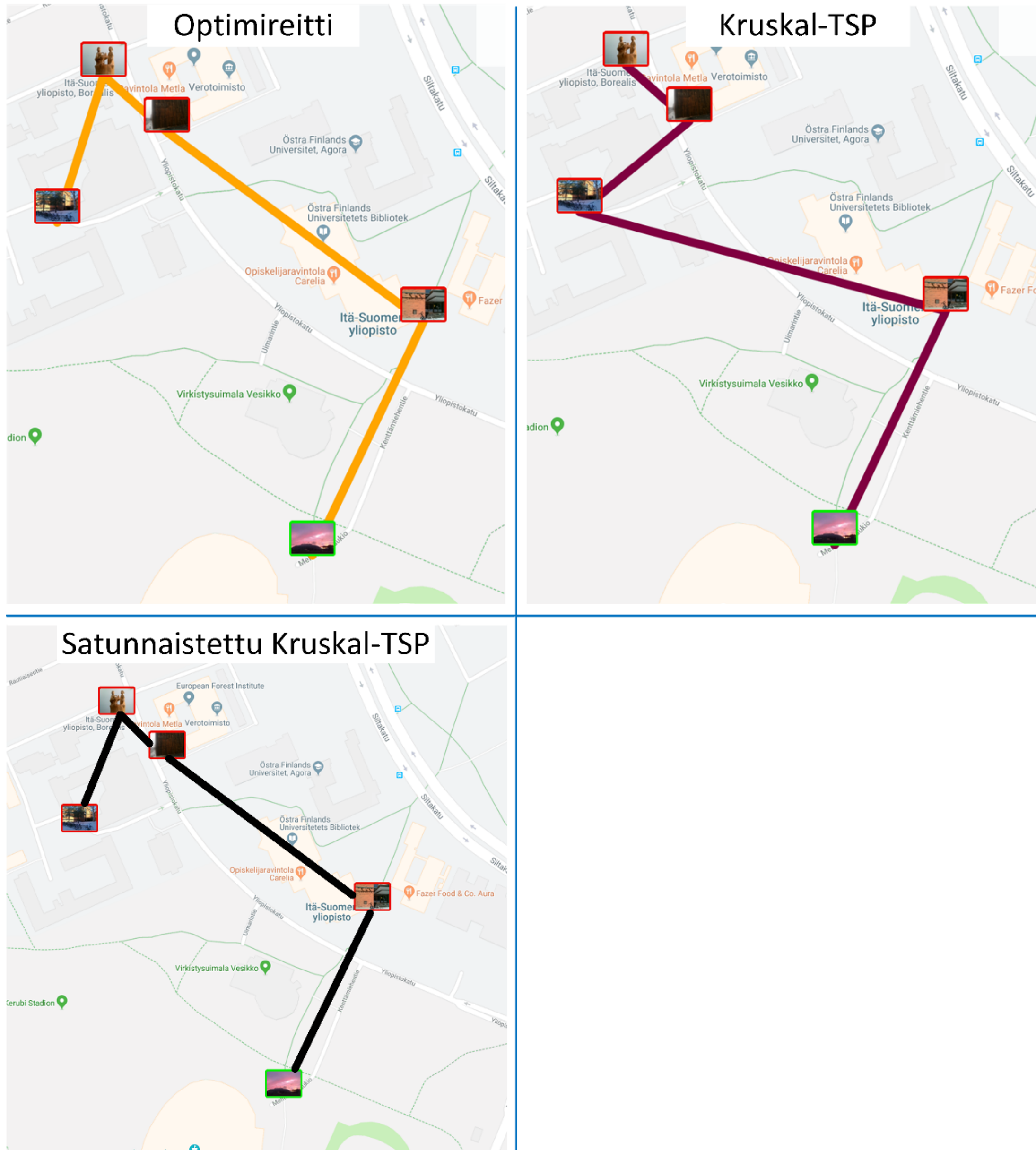
Kruskal-TSP-algoritmia testattiin myös käytännössä. Algoritmi toteutettiin Java-ohjelmointikielellä. Testaukseen käytetyt datajoukot olivat O-Mopsi [13], joka sisältää oikeita koordinaatteja ja Dots [15], joka sisältää xy-koordinaatteja (kaksiulotteinen karteesinen koordinaatisto). O-Mopsi on pieni datajoukko, sillä se sisältää 147 reittiä. Dots on huomattavasti isompi, sillä se sisältää 6447 reittiä. Dots-datajoukon ajamista varten Haversine-funktio vaihdettiin Euclidinean distance -funktioksi. Algoritmin toimivuutta ja suoritusaikaa mitattiin Windows 10 -tietokoneella, jossa on Intel core i7-8700k prosessori. Suoritusajan testauksessa ajot toistettiin 100 kertaa tuloksien luetettavuuden parantamiseksi. Saaduista tulosajoista valittiin aina mediaani varsinaiseen lopputulokseen. Sen lisäksi analysoitua aikavaativuutta verrattiin mitattuihin arvoihin. Lisäksi testattiin, kuinka paljon saadut etäisyystulokset erosivat verrattuna optimituloksiin.

Satunnaistettua Kruskal-TSP-algoritmia testattiin samalla tavalla kuin perus-Kruskal-TSP:tä. Satunnaistetussa Kruskal-TSP:ssä kaaria valittiin satunnaisesti kolmesta lyhimmästä. Jokainen datajoukko ajettiin 100 kertaa ja näistä tuloksista valittiin lyhin reitti. Satunnaistetun Kruskal-TSP:n datajoukkoina toimivat O-Mopsi ja Dots. Saatuja tuloksia verrattiin aiempiin perus-Kruskal-TSP:llä saatuihin tuloksiin. Lisäksi molempien algoritmien tuloksia verrattiin muutaman muun kauppamatkustajan ongelman ratkaisevan algoritmin antamiin tuloksiin.

6.2 Esimerkkitulos O-Mopsi testidatasta

Yksittäisestä O-Mopsi-esimerkistä voidaan nähdä kuinka satunnaistettu Kruskal-TSP voi antaa optimaalisen tuloksen, vaikka perus-Kruskal-TSP ei kyseiseen pysty. Kruskal-TSP:n antaman reitin pituus on 674 m. Satunnaistettu Kruskal-TSP antaa tulokseksi 637 m, joka on myös optimireitin pituus. Satunnaisuuden avulla päästään

eroon huonosta toisen kaaren valinnasta. Valitsemalla hieman pitempi kaari tässä vaiheessa mahdollistaa paremman kaaren valitsemisen seuraavaksi.



Kuva 19: Optimireitti, Kruskal-TSP:n ja satunnaistetun Kruskal-TSP:n antama tulos

6.3 Kokonaistulokset

Taulukko 1: Kruskal-TSP:llä ja satunnaistetulla Kruskal-TSP:llä saatuja tuloksia O-Mopsi ja Dots -datajoukoilla

Datajoukko	**O-Mopsi**		**Dots**	
Algoritmi	Kruskal-TSP	Satunnaistettu Kruskal-TSP	Kruskal-TSP	Satunnaistettu Kruskal-TSP
Maksimiero optimaaliseen	23,64 %	9,70 %	28,40 %	13,46 %
Keskimääräinen ero optimaaliseen	4,63 %	0,65 %	2,66 %	0,26 %
Optimireittien määrä prosentteina	36,73 %	74,15 %	56,93 %	87,82 %

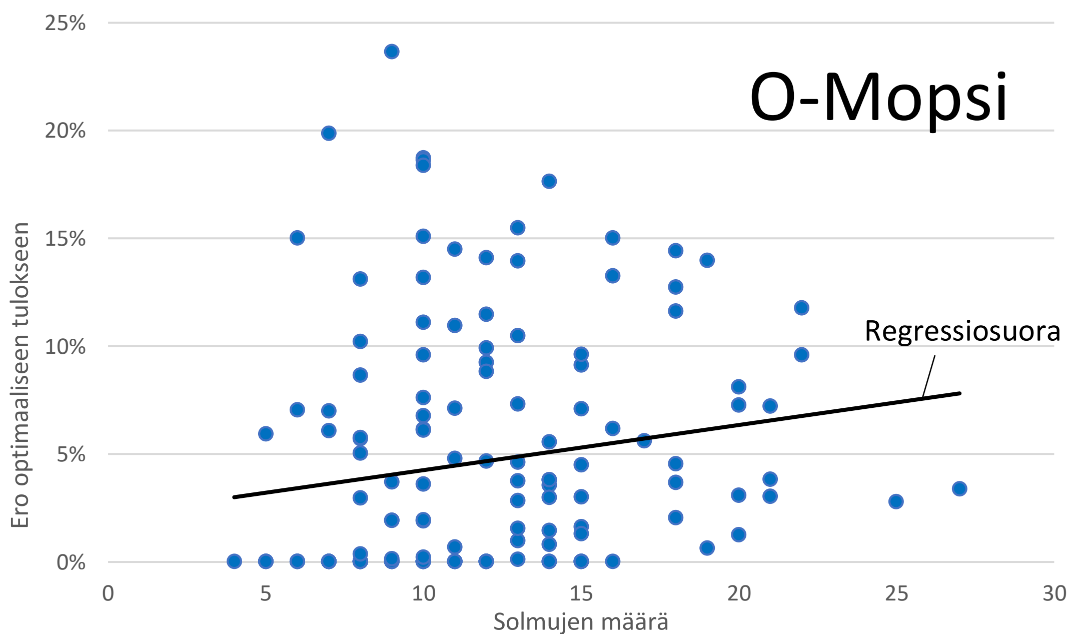
Kuten taulukosta 1 näkyy, niin tulokset satunnaistetulla Kruskal-TSP:llä ovat paljon parempia kuin perus-Kruskal-TSP:llä. O-Mopsi-datajoukolla huonoimman reitin pituuden ero optimaaliseen tulokseen tippui 23,64 %:sta 9,70 %:iin, mikä on huomattava parannus. Dots-datajoukolla huonoin tulos tippui huomattavasti 28,40 %:ta 13,46 %:iin.

Keskimääräinen ero optimaaliseen tulokseen laski myös selvästi. Perus-Kruskal-TSP:llä keskiarvo oli 4,63 % pitempi kuin optimaalinen tulos O-Mopsi-datajoukolla, satunnaistetulla Kruskal-TSP:llä päästiin 0,65 %, joka on jo lähellä optimaalista.

Optimaalisia reittejä saatiin perus-Kruskal-TSP:n avulla 36,73 % kaikista reiteistä O-Mopsi-datajoukossa. Satunnaistetulla Kruskal-TSP:llä vastaava tulos on 74,15 %, joka

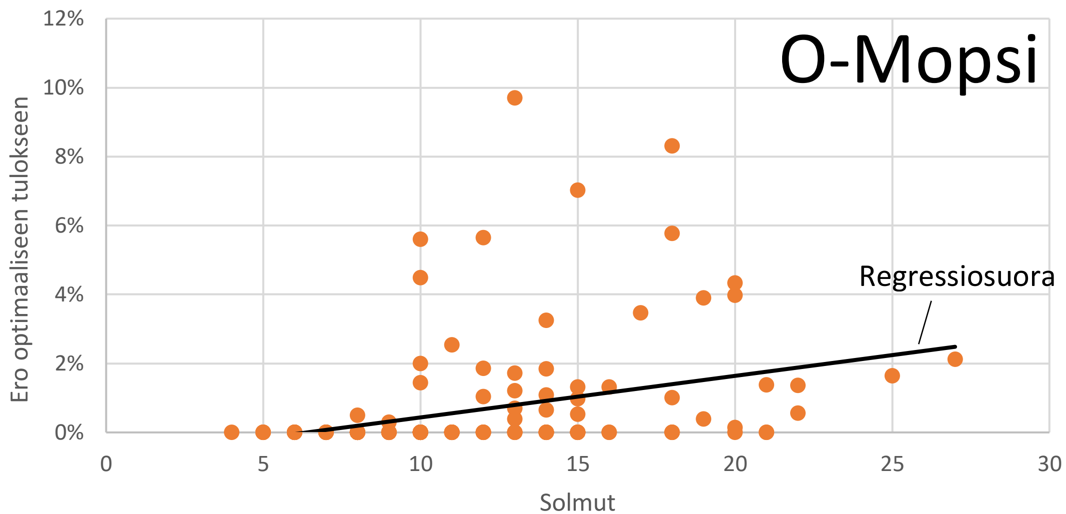
on huomattavasti parempi tulos. Myös Dots-datajoukolla päästiin samankaltaisiin tuloksiin kaikilla mittaustavoilla.

Kuvasta 20 näkyy perus-Kruskal-TSP:n antamat yksittäiset tulokset jokaiselle O-Mopsi-reitille. Eli Kruskal-TSP ajettiin jokaisella O-Mopsi-reitillä ja saatu tulos piirrettiin hieman läpinäkyvänä pallona kuvaan. Kuvasta näkyy, että 5–12 solmun reiteillä on paljon päällekkäisiä optimaalisia arvoja. Lisäksi mittausarvot ovat jakautuneet muodostamatta mitään selvää muotoa. Eli lopputuloksissa on hyvin suurta vaihtelua. Regressiosuorasta voidaan kuitenkin havaita, että solmumäärän kasvaessa keskimääräinen ero optimaaliseen tulokseen kasvaa hieman.



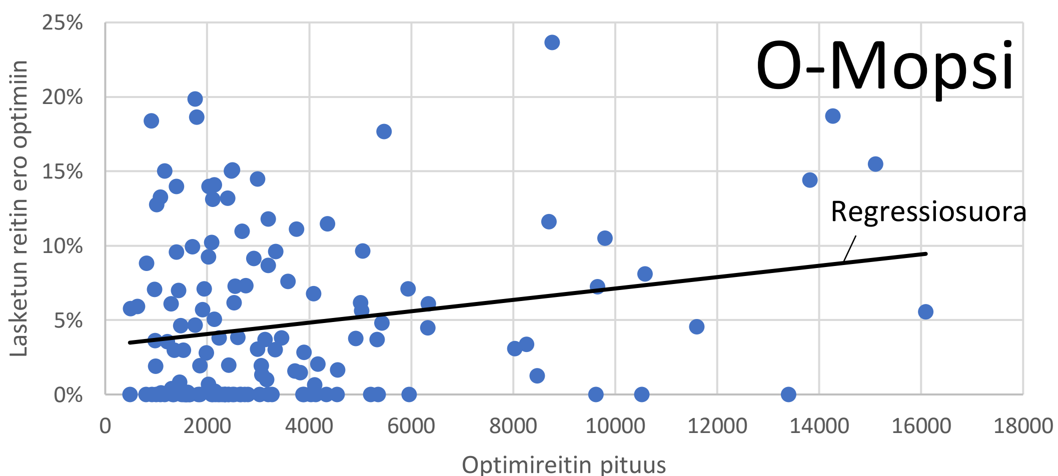
Kuva 20: Reittien ero optimaaliseen tulokseen perus-Kruskal-TSP:llä O-Mopsi-datajoukolla

Satunnaistetun Kruskal-TSP:n antamissa tuloksissa on vähemmän vaihtelua. Kuvasta 21 näkyy että 5–16 solmun reiteillä on paljon päällekkäisiä arvoja lähellä optimia. Yksittäisiä huonompia tuloksia on huomattavasti vähemmän kuin perus-Kruskal-TSP:n antamissa tuloksissa. Lisäksi nähdään, että huonoinkin tulos on kohtuullinen verrattuna perus-Kruskal-TSP:n antamiin tuloksiin. Regressiosuorasta nähdään, että myös satunnaistetulla Kruskal-TSP:llä ero optimaaliseen tulokseen kasvaa solmumäärän noustessa.



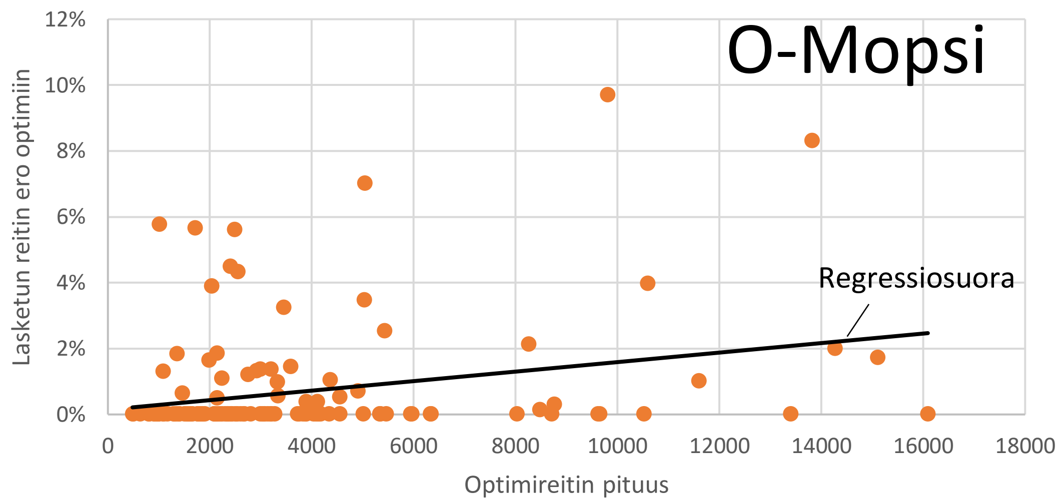
Kuva 21: Reittien ero optimaaliseen tulokseen satunnaistetulla Kruskal-TSP:llä O-Mopsi-datajoukolla

Kuvasta 22 näkyy, että lyhyitä reittejä on datassa paljon. Kuvaajasta näkyy myös, että vaikka optimireitti olisi lyhyt, niin Kruskal-TSP antaa useasti siltikin huonoja tuloksia. Tuloksissa näyttää olevan hyvin paljon vaihtelua reiteillä, joiden optimipituus on lähellä toisiaan. Kuvan regressiosuorasta näkyy, että keskimäärin optimireitin pituuden kasvaessa lasketun reitin ero optimaaliseen tulokseen kasvaa hieman.



Kuva 22: Lasketun reitin pituuden ero optimaaliseen tulokseen eri reittien pituuksilla O-Mopsi-datajoukolla perus-Kruskal-TSP:llä

Vastaavassa kuvassa 23 satunnaistetulla Kruskal-TSP:llä huomataan, että muoto on hyvin samanlainen, mutta ero optimaaliseen tulokseen on huomattavasti pienempi kaikilla reiteillä.



Kuva 23: Lasketun reitin pituuden ero optimaaliseen tulokseen eri reittien pituuksilla O-Mopsi-datajoukolla satunnaistetulla Kruskal-TSP:llä

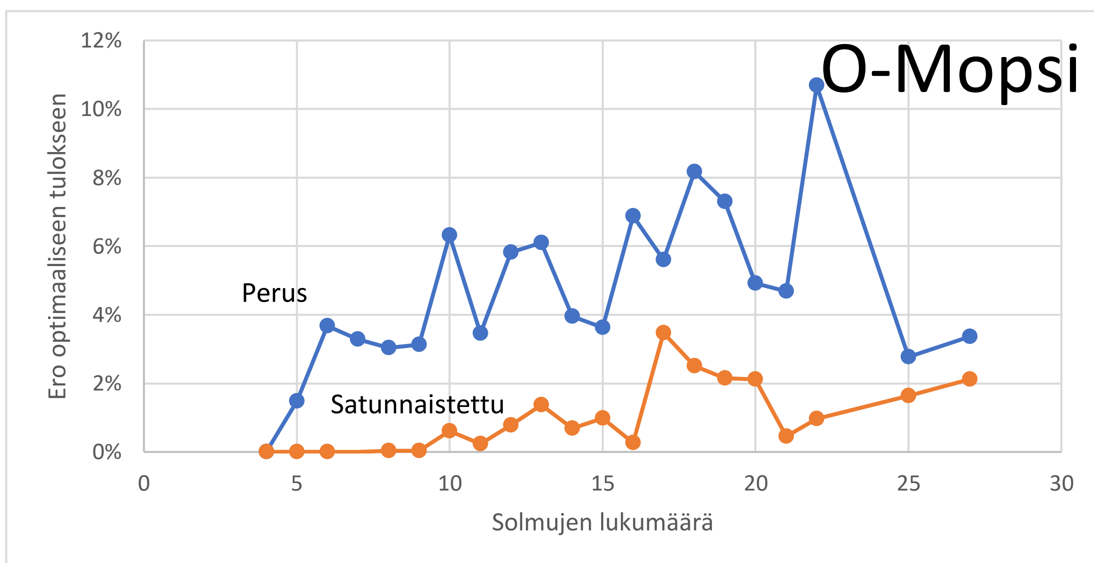
Taulukko 2: Kymmenen solmun reittien arvoja Kruskal-TSP:lle

Keskiarvo	Mediaani	Minimiarvo	Maksimiarvo
6,31 %	4,85 %	0 %	18,71 %

Taulukosta 2 näkyvät perus-Kruskal-TSP:n antamat tulokset 10 solmun reiteille O-Mopsi-datajoukolla. Kymmenen solmun reitit valittiin lisätutkintaan sen takia, että niitä oli kyseisessä datajoukossa eniten. Näin voidaan analysoida vaihtelua saman solmumäärän sisältävien reittien sisällä. Reittejä on 22 kappaletta. Tuloksissa on havaittavissa hyvin suurta vaihtelua. Tulokset vaihtelevat hyvin rajusti 0 % ja 20 % välillä. Tästä voidaan siis päätellä, että verkon muodolla on hyvin iso vaikutus lopputulokseen.

Kuvassa 24 näkyy perus-Kruskal-TSP:n antama keskimääräinen ero optimaaliseen tulokseen jokaiselle solmumäärälle. Eli jokaisen solmumäärän oma keskiarvo ero optimaaliseen tulokseen on piirretty taulukkoon. Tässä on myös suurta vaihtelua. Solmujen lukumäärällä ei näytä olevan suurta vaikutusta lopputuloksen huonouteen,

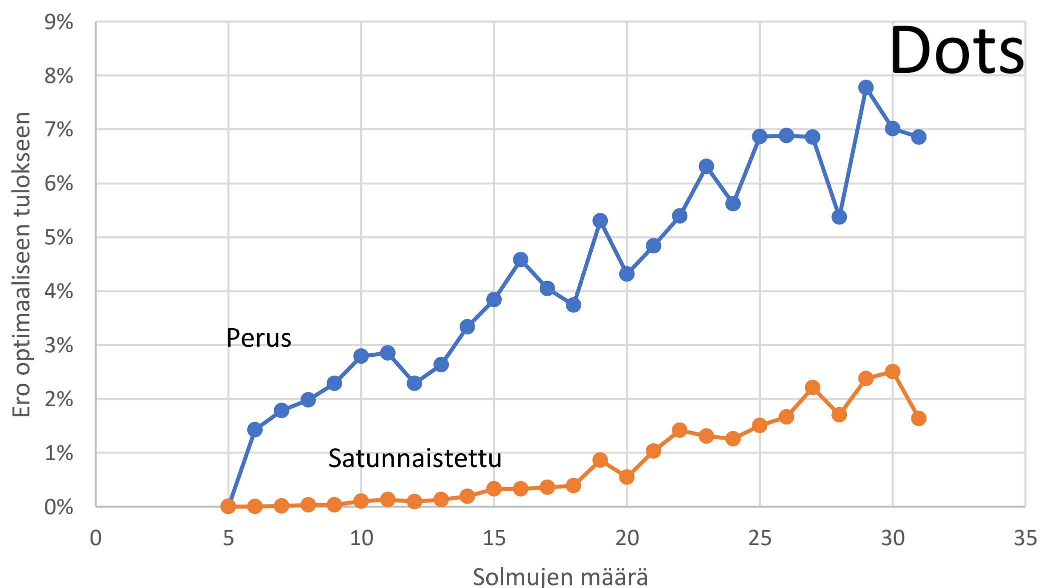
mutta isommilla solmumäärillä lopputulos näyttää eroavan optimaalista enemmän. Toisaalta vaihtelu on myös suurta. Tämä voi kuitenkin johtua siitä, että laskettuja reittejä kyseisillä solmumäärillä on vain muutama. Eli yksittäisen huonon tai hyvän reitin vaikutus on suuri. Korrelaatiokerrointa tarkastelemalla saadaan selville, että huolimatta suuresta vaihtelusta, korrelaatio on olemassa solmujen lukumäärän ja eron optimaaliseen tulokseen välillä. Korrelaatiokertoimen arvo 0,47 kertoo että korrelaatio on havaittavissa. Pahin keskiarvo on 10,68 % 22 solmun reiteillä. Keskimääräinen Kruskal-TSP:n muodostaman reitin pituus on O-Mopsi-datajoukkoa käyttäessä 4,63% pitempi kuin optimaalisen reitin.



Kuva 24: Jokaisen solmumäärän reittien keskimääräinen ero optimaaliseen tulokseen perus-Kruskal-TSP:llä ja satunnaistetulla Kruskal-TSP:llä O-Mopsi-datajoukolla

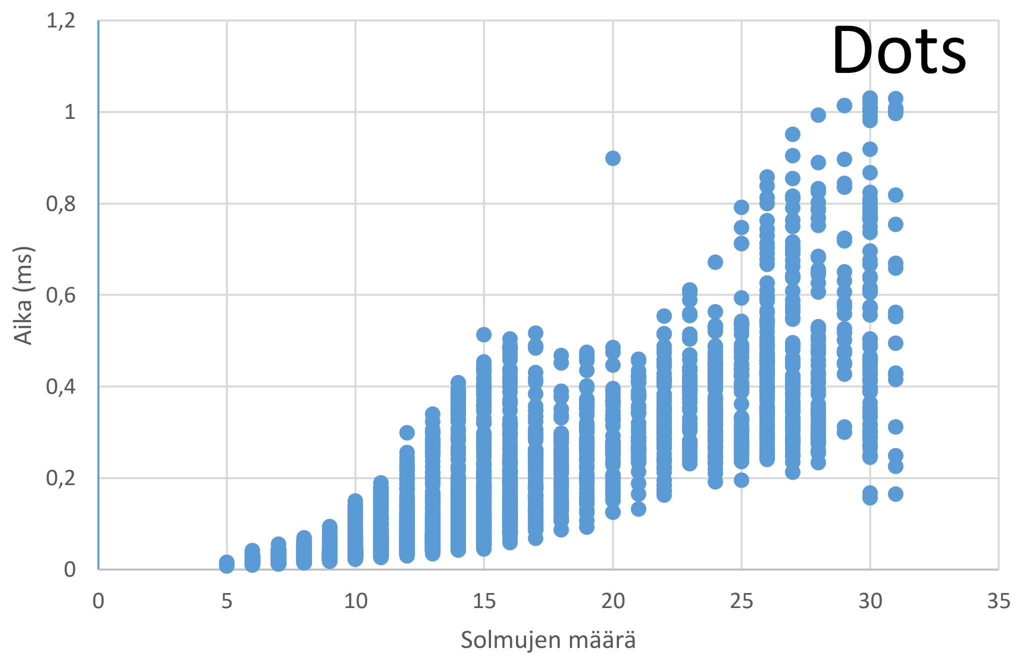
Kuvasta 24 näkyy myös satunnaistetun Kruskal-TSP:n antamat tulokset. Satunnaisuus paransi huomattavasti tuloksia O-Mopsi-datajoukkoa käyttäessä. Jokaisella solmumäärällä keskimääräinen tulos aleni huomattavasti verrattuna perus-Kruskal-TSP:hen. Erityisesti 4–9 solmun reiteillä päästiin optimaalisiin tuloksiin. Korrelaatiokerrointa tarkastelemalla saadaan selville, että korrelaatio kasvaa huomattavasti solmujen lukumäärän ja eron optimaaliseen tulokseen välillä satunnaisuutta käyttäessä. Korrelaatiokertoimen arvo 0,68 kuvaa jo selkeämpää korrelaatiota. Huonoin keskiarvo tippui 10,68 %:sta 3,47 %:iin.

Myöskin Dots-datajoukolla voidaan nähdä kuinka satunnaistettu Kruskal-TSP paransi tuloksia. Jokaisen solmumäärän keskimääräinen tulos aleni huomattavasti. Pahimmasta lähes 8 % erosta optimipituuteen päästiin alle 3 % eroon. Kuvasta 25 näkyy kuinka perus-Kruskal-TSP:llä ero optimiin alkaa nousta huomattavasti selkeämmin isommilla solmumäärillä, verrattuna satunnaistettuun Kruskal-TSP:hen. Satunnaistetulla versiolla ero optimiin nousee hyvin lievästi ja 18 solmuun asti ollaan hyvin lähellä optimia. Korrelaatiokerroin sekä perus- että satunnaistetulle Kruskal-TSP:lle on hyvin korkea. Perus-Kruskal-TSP:n korrelaatiokerroin on 0,96 ja satunnaistetun 0,93. Dots-datajoukolla solmujen määrä korreloi lähes täysin lopputuloksen huonouteen.



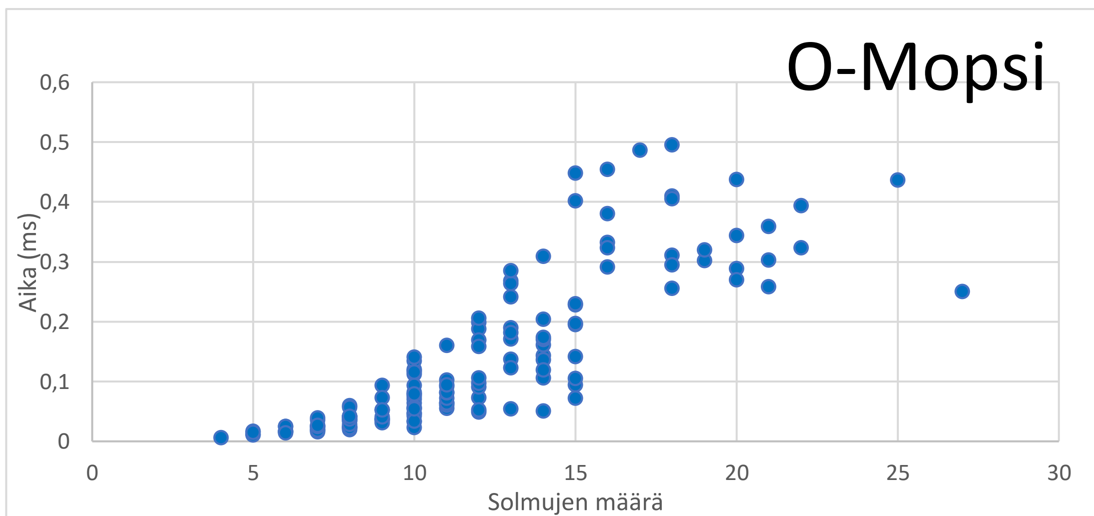
Kuva 25: Jokaisen solmumäärän reittien keskimääräinen ero optimaaliseen tulokseen perus-Kruskal-TSP:llä ja satunnaistetulla Kruskal-TSP:llä Dots-datajoukolla

Perus-Kruskal-TSP:n suoritusaikaa testattiin myös käytännössä. Kuvassa 26 näkyvät isommalla eli Dots-datajoukolla luotujen reittien tekemiseen kulunut aika millisekunteina. Aikaisemmin analysoin algoritmin aikavaativuudeksi pseudokoodia tutkimalla $O(n^3)$. Dots-datajoukon testauksella aika näyttää kasvavan enemmän kuin lineaarisesti, vaikkakin tuloksissa näkyy hyvin paljon vaihtelua.



Kuva 26: Solmujen määrä ja reitinmuodostamiseen käytetty aika Dots-datajoukolla

Kuvassa 27 näkyy pienemmällä datajoukolla eli O-Mopsi-datajoukolla ajettujen reittien muodostamiseen kulunut aika. Tässä kuvassa on myöskin havaittavissa samaa muotoa kuin Dots-datajoukolla saaduissa tuloksissa.



Kuva 27: Solmujen määrä ja reitin muodostukseen käytetty aika O-Mopsi-datajoukolla

6.4 Kruskal-TSP verrattuna muihin algoritmeihin

Kruskal-TSP:tä ja satunnaistettua Kruskal-TSP:tä vertailtiin pariin muuhun kauppamatkustajan ongelman ratkaisevaan algoritmiin. Vertailtavina algoritmeina oli Ant Colony Optimization (ACO) [17] ja Tabu Search (TS) [18] -algoritmit. ACO:n ja TS -algoritmien vertailu tehtiin jo olemassa olevilla toteutuksilla [19][20]. Alla olevissa taulukoissa 3 ja 4 näkyvät olennaisimmat tulokset algoritmeista O-Mopsi ja Dots -datajoukkoa käytettäessä.

Taulukosta 3 nähdään, että perus-Kruskal-TSP ei pärjää vertailussa muille algoritmeille O-Mopsi-datajoukolla, vaan jää aina huonoimmaksi selvällä erolla. Satunnaistettu Kruskal-TSP pärjää huomattavasti paremmin. Se voittaa vertailussa TS-algoritmin, mutta jää kuitenkin selvästi ACO:sta.

Taulukko 3: Algoritmien vertailua O-Mopsi-datajoukolla

O-Mopsi tulokset	Keskimääräinen ero optimaaliseen	Optimaalisten reittien määrä prosentteina
Kruskal-TSP	4,63 %	36,73 %
Satunnaistettu Kruskal-TSP, 100 toistoa	0,65 %	74,15 %
ACO	0,14 %	89,12 %
TS	2,14 %	55,78 %

Taulukosta 4 nähdään Dots-datajoukolla hyvin samankaltaiset tulokset kuin O-Mopsi-datajoukolla. Perus-Kruskal-TSP on selvästi huonoin. Paremmuusjärjestys on sama kuin aiemmin. Kuitenkin Dots-datajoukolla satunnaistettu Kruskal-TSP pääsee lähemmäksi ACO:n tuloksia.

Taulukko 4: Algoritmien vertailua Dots-datajoukolla

Dots tulokset	Keskimääräinen ero optimaaliseen	Optimaalisten reittien määrä prosentteina
Kruskal-TSP	2,66 %	56,93 %
Satunnaistettu Kruskal-TSP, 100 toistoa	0,26 %	87,82 %
ACO	0,03 %	96,22 %
TS	0,96 %	72,78 %

7 Yhteenveto

Työssä esiteltiin ja toteutettiin avoimen kierroksen kauppamatkustajan ongelman ratkaiseva algoritmi Kruskal-TSP käyttäen pohjana Kruskalin algoritmia. Lisäksi algoritmista tehtiin satunnaistettu versio, jonka tarkoituksena oli parantaa perus-Kruskal-TSP:n tuloksia.

Tuloksista sai selville sen, että Kruskalin algoritmia voidaan käyttää avoimen kierroksen kauppamatkustajan ongelman ratkaisemiseksi. Perus-Kruskal-TSP:n antamat tulokset jättävät kuitenkin toivomisen varaa sekä O-Mopsi että Dots -datajoukoilla. O-Mopsi-datajoukolla algoritmin antama keskimääräinen ero optimaaliseen tulokseen on 4,63 %. Huonoin tulos on 23,64 % pitempi kuin optimaalinen tulos. Lisäksi vain 36,73 % tuloksissa algoritmi antaa optimaalisen reitin. Algoritmin ongelma on ahneus, mikä johtaa siihen, että usein ei päästä lähelle optimipituutta.

Ahneen algoritmin heikkouksia voidaan kuitenkin korjata satunnaisuuden avulla. Satunnaistettu versio Kruskal-TSP-algoritmista toimii hyvin sekä O-Mopsi että Dots -datajoukoilla. Kyseisellä versiolla päästään lähelle optimaalisia tuloksia. O-Mopsi-datajoukolla keskimääräinen ero optimaaliseen tulokseen on 0,65 %. Huonoin tulos on 9,70 % pitempi kuin optimaalinen tulos, joka sekin on huomattava parannus verrattuna perus-Kruskal-TSP:n antamaan tulokseen. Satunnaistettu Kruskal-TSP antaa optimaalisia tuloksia 74,15 % reiteistä, joka on jo hyvä tulos. Aikavaativuus NP-vaikeaan ongelmaan verrattuna on ihan siedettävä $O(n^3)$. Hyvä puoli on myös se, että algoritmi on kohtalaisen helppo toteuttaa.

Molempien algoritmien perusteella voidaan sanoa, että mitä useampi solmu reitissä on, sitä hankalampi optimaalisen reitin laskeminen on. Satunnaisuus auttaa kuitenkin tässä. Toistoja kuitenkin tarvitaan tällöin useita. Satunnaistetulla Kruskal-TSP-algoritmilla voi olla hyötyä käytännössä, mikäli halutaan löytää lähellä optimia oleva reitti suhteellisen nopeasti.

8 Viitteet

[1] Laporte, Gilbert. "A concise guide to the traveling salesman problem." *Journal of the Operational Research Society* 61.1 (2010): 35-40.

[2] Applegate, David., Bixby, R., Cook, W., Chvátal, V. "On the solution of traveling salesman problems." (1998).

[3] Graham, Ronald L., Hell, Pavol. "On the history of the minimum spanning tree problem." *Annals of the History of Computing* 7.1 (1985): 43-57.

[4] Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." *Proceedings of the American Mathematical society* 7.1 (1956): 48-50.

[5] Wu, Bang Ye., Chao, Kun-Mao. "Spanning trees and optimization problems." *CRC Press*, (2004).

[6] Lawler, Eugene L. "The traveling salesman problem: a guided tour of combinatorial optimization." *Wiley-Interscience Series in Discrete Mathematics* (1985).

[7] Applegate, David L., Bixby, R., Cook, W., Chvátal, V. "The traveling salesman problem: a computational study." *Princeton university press*, (2006).

[8] Jünger, Michael., Reinelt, Gerhard., Rinaldi, Giovanni. "The traveling salesman problem." *Handbooks in operations research and management science 7* (1995): 225-330.

[9] Borůvka, Otakar. "O jistém problému minimálním.", *Prace Moravske Pridovedecke Spolecnosti, vol. 3*, (1926).

[10] Prim, Robert Clay. "Shortest connection networks and some generalizations." *Bell system technical journal 36.6* (1957): 1389-1401.

[11] Dantzig, George., Fulkerson, Ray., Johnson, Selmer. "Solution of a large-scale traveling-salesman problem." *Journal of the operations research society of America* 2.4 (1954): 393-410.

[12] Adleman, L. M. “Molecular computation of solutions to combinatorial problems.” *Science 266* (1994): 1021–1024.

[13] Fränti, Pasi., Mariescu-Istodor, Radu., Sengupta, Lahari. "O-Mopsi: Mobile orienteering game for sightseeing, exercising, and education." *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 13.4 (2017): 56.

[14] Laporte, Gilbert. "The traveling salesman problem: An overview of exact and approximate algorithms." *European Journal of Operational Research* 59.2 (1992): 231-247.

[15] Sengupta, Lahari., Mariescu-Istodor, Radu., Fränti, Pasi. "Which local search operator works best for open loop Euclidean TSP", *Applied Intelligence*, 9 (19), 3985, 2019.

[16] Sengupta, Lahari., Fränti, Pasi. "Predicting difficulty of TSP instances using MST", *IEEE Int. Conf. on Industrial Informatics (INDIN)*, 847-852, Helsinki, 2019.

[17] Dorigo, Marco., Di Caro, Gianni. "Ant colony optimization: a new meta-heuristic." *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. Vol. 2. IEEE, 1999.

[18] Glover, Fred., Laguna, Manuel. "Tabu search." *Handbook of combinatorial optimization*. Springer, Boston, MA, 1998. 2093-2229.

[19] Behravan, Hamid, ACO implementation, [online] https://www.researchgate.net/profile/Hamid_Behravan, (viitattu 1.12.2019)

[20] Sanio, Toni, Tabu Search implementation, [online] http://tonisanio.fi/portfolio/index.html, (viitattu 1.12.2019)

[21] Christofides, Nicos. “Worst-case analysis of a new heuristic for the travelling salesman problem.” *No. RR-388*. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.