Pienimmän virittävän puun käyttö kauppamatkustajan ongelman ratkaisussa

Teemu Nenonen

Pro gradu –tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Joulukuu 2019

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

Nenonen, Teemu Kalevi: Pienimmän virittävän puun käyttö kauppamatkustajan ongelman ratkaisussa
Pro gradu –tutkielma, 47 s.
Pro gradu –tutkielman ohjaaja: FT Pasi Fränti
Joulukuu 2019

Tässä tutkielmassa perehdytään kauppamatkustajan ongelmaan ja kuinka pienintä virittävää puuta voidaan käyttää sen ratkaisun apuna. Aluksi selvitetään, mikä on pienin virittävä puu ja miten se luodaan algoritmien avulla. Sen jälkeen tutkielmassa käsitellään mikä kauppamatkustajan ongelma ylipäätänsä on. Pääkohteena on tutkia, kuinka pienimmästä virittävästä puusta voidaan muokata ratkaisu avoimen kierroksen kauppamatkustajan ongelmaan. Algoritmin toiminta selitetään yksityiskohtaisesti. Myös algoritmin sisältäviä heikkouksia käsitellään yksityiskohtaisesti ja algoritmin aikavaativuutta tutkitaan. Tämän jälkeen esitellään vielä parannusehdotus algoritmiin. Satunnaistettu puunluonti esitellään yksityiskohtaisesti. Seuraavaksi analysoidaan, kuinka hyviä tuloksia perusalgoritmilla ja satunnaistetulla algoritmilla saadaan. Lopuksi vielä vertaillaan algoritmin tuloksia muihin kauppamatkustajan ongelman ratkaiseviin algoritmeihin.

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

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, Teemu Kalevi: Using minimum spanning tree to solve travelling salesman problem
Master's Thesis, 47 p.
Supervisor of the Master's Thesis: PhD Pasi Fränti
December 2019

In this thesis travelling salesman problem is explored and how minimum spanning tree can be used in resolving it. First is explained what minimum spanning tree is and how it can be created with algorithms. After that is discussed what travelling salesman problem is. Main point of the thesis is to examine how minimum spanning tree can be modified to open loop travelling salesman route. Algorithm's working logic is explained in detail. Also, algorithm's weaknesses are discussed thoroughly, and its time complexity is examined. After that, improvement is presented to algorithm. Randomized tree creation is explained thoroughly. Next, basic and randomized algorithms' results are analyzed. Lastly algorithm's results are compared to some other travelling salesman problem solving algorithms.

Keywords: Minimum spanning tree, Travelling salesman problem, Shortest route, Prim's algorithm.

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 ohjaajaani FT Pasi Fräntiä hänen ohjauksestaan tämän tutkielman aikana, sekä TkT Saku Kukkosta tutkielman tarkistamisesta ja avusta tutkielman hiomisessa.

Lyhenneluettelo

ACM	Association for Computing Machinery; maailmanlaajuinen tietotekniikka-alan tieteellinen yhdistys
MST	Minimum Spanning Tree, pienin virittävä puu
TSP	Travelling Salesman Problem, Kauppamatkustajan ongelma
ACO	Ant Colony Optimization
TS	Tabu Search

Sisällysluettelo

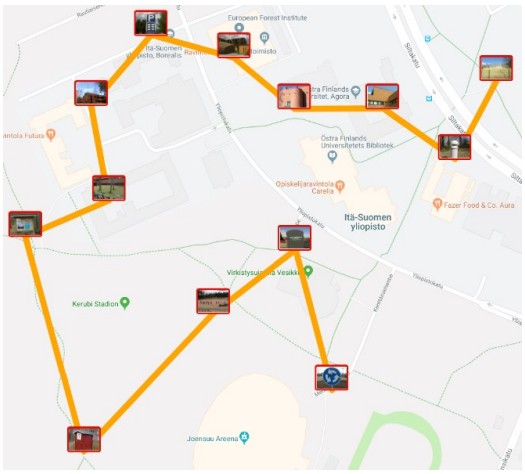
1 Johdanto

Kauppamatkustajan ongelmassa etsitään lyhin reitti vierailtavien kaupunkien läpi siten, että alku- ja loppupiste ovat samat ja jokaisessa kaupungissa käydään yhden kerran [12].

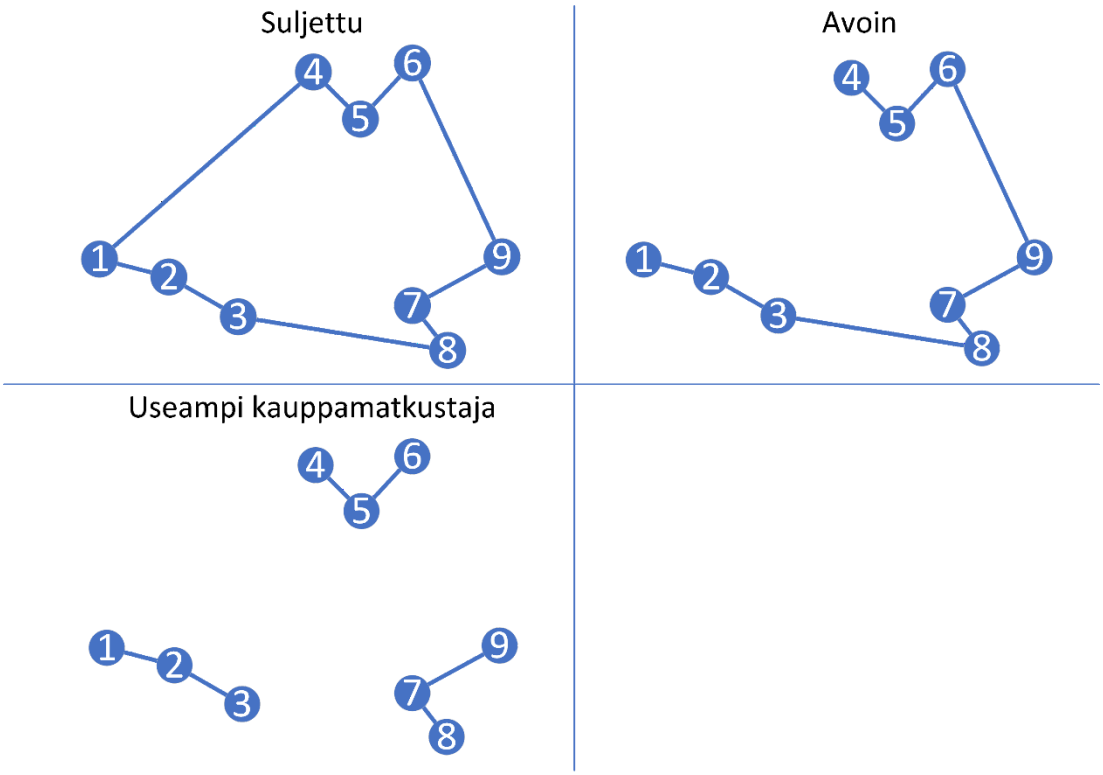
Kauppamatkustajan ongelmalla on monia käyttökohteita, kuten kulkuvälineiden reititys, tietokoneiden johdotus, työvaiheiden sekventointi ja klusterointi sekä monia muita [6]. Sovelluksia käsitellään vielä tarkemmin omassa kappaleessaan myöhemmin.

Kauppamatkustajan ongelmalla on myös useita sitä lähellä olevia ongelmia. Usean kauppamatkustajan ongelma on yksinkertaistetumpi versio kulkuneuvojen reititysongelmasta [6]. Kauppamatkustajia on useampi kuin yksi ja jokaisessa kaupungissa täytyy vierailla yksi kauppamatkustaja. Jokaisella kauppamatkustajalla on määrätty alku- ja loppupiste. Jokaisen uuden kauppamatkustajan palkkaaminen maksaa vakion verran. Tavoitteena on vierailla jokaisessa kaupungissa siten, että reittien kulkemisesta aiheutuva yhteenlaskettu summa ja palkkaukseen menevä summa on mahdollisimman pieni.

Avoimen kierroksen kauppamatkustajan ongelmassa vieraillaan jokaisessa verkon solmussa kerran ja kaarien eli solmujen välisten etäisyyksien summa on pienin mahdollinen siten, ettei palata takaisin lähtöpisteeseen [10]. Tutkielmassa keskitytään avoimen kierroksen versioon yhden kauppamatkustajan tapauksessa.



Kuva 1: Esimerkki avoimen kierroksen kauppamatkustajan reitistä



Kuva 2: Esimerkki erilaisista kauppamatkustajan ongelmista

Myös virittävät puut liittyvät käsiteltävään aiheeseen. Virittävä puu yhdistää kaikki verkon solmut [6]. Pienin virittävä puu on helpotettu versio kauppamatkustajan ongelmasta ja siten kauppamatkustajan ongelma on rajoitettu versio pienimmän virittävän puun ongelmasta. *Hamiltonialainen* reitti on sellainen, jossa jokaisessa solmussa vieraillaan yhden kerran [4]. Jokainen Hamiltonialainen reitti on virittävä puu. Virittävä puu on myös Hamiltonialainen reitti, jos se täyttää ehdon, että jokaisesta puun solmusta lähtee korkeintaan kaksi kaarta. Siten nämä ongelmat liittyvät hyvinkin läheisesti toisiinsa.

Tutkielmassa esitellään algoritmi, jonka toimintaperiaatteena on luoda ensin pienin virittävä puu ja sitä muokkaamalla saada mahdollisimman hyvä ratkaisu kauppamatkustajan ongelmaan. Eli ensin luodaan pienin virittävä puu esimerkiksi Primin algoritmin avulla, ja sitten puusta aletaan poistamaan kaaria ja lisäämään kaaria poistettujen tilalle siten, että saadaan valmis kauppamatkustajan reitti.

Luvussa 2 käsitellään pienintä virittävää puuta sekä esitellään algoritmeja, joiden avulla verkosta voidaan luoda pienin virittävä puu. Kolmas luku käsittelee kauppamatkustajan ongelmaa, sen vaikeutta ja käyttökohteita. Neljännessä luvussa

esitellään, miten pienimmästä virittävästä puusta saadaan kauppamatkustajan ongelmaan ratkaisu. Algoritmin toiminta ja heikkoudet käydään läpi. Viidennessä luvussa esitellään parannusehdotuksia algoritmiin. Kuudennessa luvussa käydään läpi algoritmin avulla saatuja tuloksia eri datajoukoille ja käydään läpi myös satunnaistetun version tulokset. Lisäksi algoritmin antamia tuloksia verrataan myös muihin saman ongelman ratkaiseviin algoritmeihin. Lopuksi tehdään yhteenveto käsitellystä aiheesta.

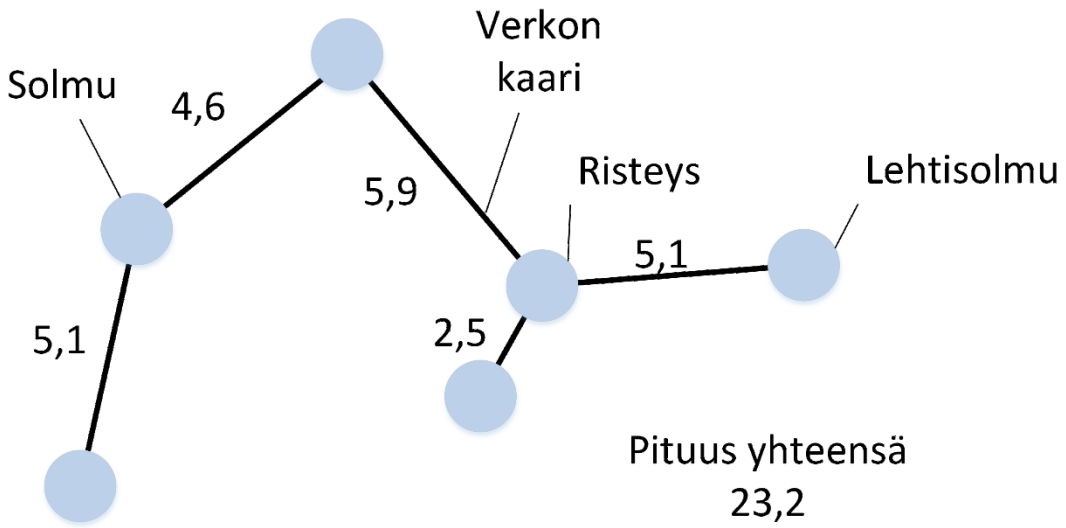
2 Pienin virittävä puu

2.1 Määritelmä

Verkko koostuu n määrästä solmuja. Solmut ovat yhteydessä toisiinsa *kaarilla*. Verkkoon ja siten myös puuhun voi sisältyä useampia eri tyyppisiä solmuja. *Risteyssolmu* eli risteys on verkon solmu, josta lähtee useampi kuin kaksi kaarta. *Lehtisolmu* on myös yksi solmutyyppi. Lehtisolmusta ei lähde kuin korkeintaan yksi kaari.

Virittävä puu määritellään siten, että se on verkon aliverkko, joka sisältää kaikki verkon solmut kaarilla yhdistettyinä ja verkko muodostaa puurakenteen [1]. Puuhun ei sisälly *syklejä* eli kierroksia. Virittävällä puulla on useita käyttötarkoituksia. Virittävää puuta voidaan käyttää, jos halutaan löytää tehokas ratkaisu eri päätepisteiden yhdistämiseen. Esimerkkinä voi olla kaupungit, joiden välille halutaan rakentaa tieverkosto. Välttämättä jokaisesta kaupungista ei haluta rakentaa tietä jokaiseen toiseen kaupunkiin, mutta jokaisesta kaupungista pitää kuitenkin jotain kautta päästä myös muihin kaupunkeihin. Virittävä puu sisältää kaaria yhden vähemmän kuin on solmujen määrä virittävässä puussa.

Pienin virittävä puu (Minimum spanning tree, MST) eroaa tavallisesta virittävästä puusta siten, että puussa olevien kaarien pituuksien summa on pienin, mikä on kyseisessä verkossa mahdollista [1]. Pienimpään virittävään puuhun pitää sisältyä jokainen verkon solmu ja kaarien summa täytyy olla pienin mahdollinen. Pienin virittävä puu on käytännöllinen, kun halutaan yhdistää verkon solmuja toisiinsa mahdollisimman kustannustehokkaasti. Pienimmän virittävä puun löytämiseksi verkosta on kehitetty erilaisia algoritmeja, joista tunnetuimpia ovat *Primin algoritmi* [2] ja *Kruskalin algoritmi* [7]. Kuvassa 3 on esimerkki pienimmästä virittävästä puusta.



Kuva 3: Pienin virittävä puu

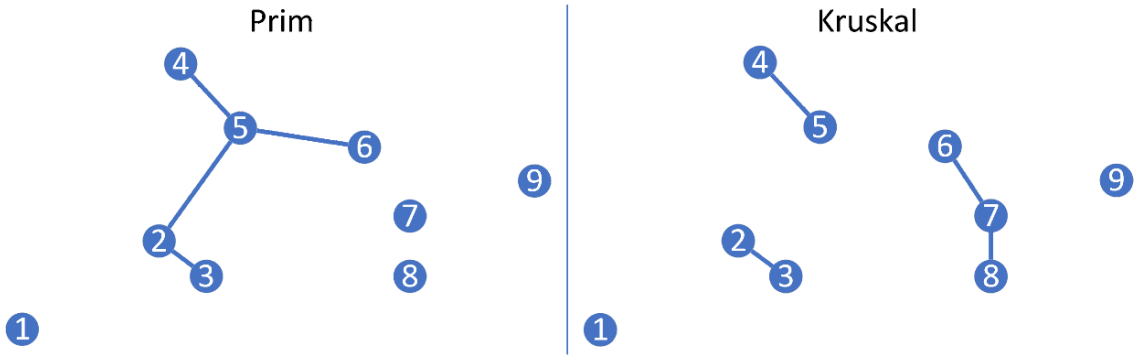
2.2 Kruskalin algoritmi

Kruskalin algoritmin esitteli vuonna 1956 Joseph Kruskal [1]. Kruskalin algoritmin toimintaperiaatteena on luoda metsä, jossa jokainen verkon solmu on aluksi erillinen puu. Tämän jälkeen kaikki verkon kaaret järjestetään kaarien painon mukaiseen järjestykseen. Seuraavassa vaiheessa kaaret käydään järjestyksessä läpi ja jos kaaren alkusolmu ja loppusolmu kuuluvat eri puihin lisätään molemmat rakennettavaan metsään, jolloin kaksi puuta yhdistetään yhdeksi puuksi. Tätä jatketaan, kunnes kaikki verkkoon kuuluvat kaaret on käyty läpi.

2.3 Primin algoritmi

Primin algoritmin esitteli vuonna 1957 Robert Prim [1]. Primin algoritmin toiminta perustuu kahdelle säännölle, joiden avulla pienin virittävä puu rakennetaan [2]. Ensimmäinen sääntö on, että yksittäinen verkon solmu voidaan yhdistää lähimpään naapuriin. Toinen sääntö on, että jo yhdistetyt verkon solmut muodostavat oman alipuun, joka voidaan yhdistää lähimpään toiseen alipuuhun lyhimmällä olemassa olevalla kaarella. Aluksi valitaan lähtösolmu satunnaisesti, jonka jälkeen siihen yhdistetään lähin naapuri [1]. Seuraavassa vaiheessa rakennettavaan puuhun lisätään lähin solmu, joka ei jo sisälly luotuun alipuuhun. Tätä jatketaan, kunnes kaikki solmut sisältyvät puuhun. Solmun lisäyssääntö estää silmukoiden syntymisen. Lopulta kaikki

verkon solmut sisältyvät rakentuneeseen puuhun ja lopputuloksena saadaan pienin virittävä puu.



Kuva 4: Esimerkki samasta välivaiheesta pienimmän virittävän puun luonnissa Primin ja Kruskalin algoritmilla

2.4 Fast-MST-algoritmi

Perinteisten pienimmän virittävän puun luovien algoritmien käyttö suurelle datajoukolle on haastavaa, koska niiden aikavaativuus on neliöllinen solmujen määrän suhteen [8]. Fast-MST-algoritmi käyttää hajota ja hallitse -rakennetta luodakseen likimääräisen pienimmän virittävän puun [8]. Tämän etuna on, että teoreettinen aikavaativuus on O($n^{1.5}$) ja siten pienempi kuin perinteisillä algoritmeilla.

Algoritmi voidaan jakaa kahteen päävaiheeseen [8]. Ensimmäinen päävaihe koostuu kolmesta osasta. Ensimmäiseksi K-means-klusterointimenetelmän avulla datajoukko jaetaan $\sqrt{N}$ määrään klustereita. *Klusteroinnilla* tarkoitetaan datan järjestelemistä ryhmiin samanlaisten ominaisuuksien perusteella [3].

Tämän jälkeen käytetään jotain perinteistä pienimmän virittävän puun luovaa algoritmia, kuten Primiä, rakentamaan pienin virittävä puu jokaisesta alijoukosta [8]. Seuraavaksi luodut alipuut yhdistetään toisiinsa, jotta saadaan luotua likimääräinen pienin virittävä puu.

Toinen päävaihe koostuu myös kolmesta osasta. Ensiksi luodaan vaihtoehtoiset alijoukot, jotka keskittyvät klustereiden laidoille. Seuraavaksi luodaan toinen likimääräinen pienin virittävä puu kuten ensimmäisessä päävaiheessa. Lopuksi saadut likimääräiset pienimmät virittävät puut yhdistetään, ja uusi tarkempi tulos saadaan

käyttämällä perinteistä pienimmän viritettävän puun luovaa algoritmia tähän tulokseen. [8]

2.5 Pienimmän virittävän puun sovelluksia

Yksi pienimmän virittävän puun käyttökohteista on kaapelin vetäminen asuntoalueelle [3]. Kaapelin kustannus on yleensä suoraan riippuvainen sen pituudesta, joten virittävän puun avulla saadaan kaikki asunnot yhdistettyä kaapeliverkkoon mahdollisimman pienin kustannuksin.

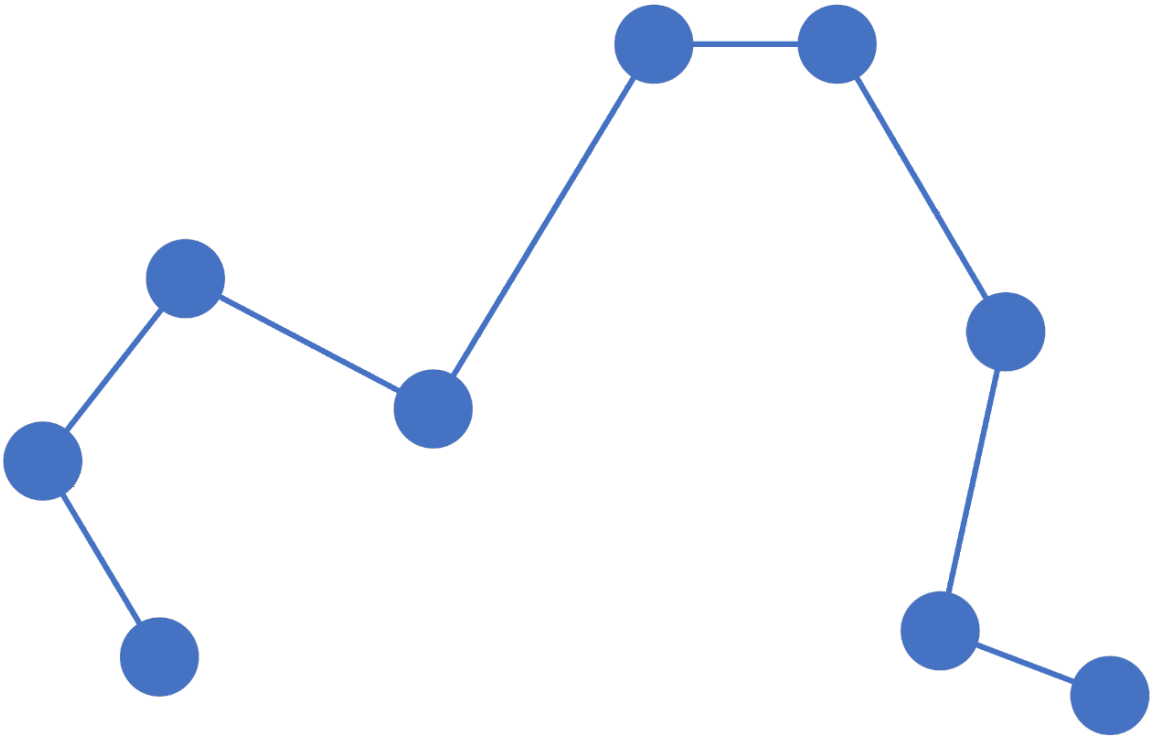
Toinen sovelluskohde on piirien suunnittelu [3]. Sähköpiirissä tarvitaan usein yhdistää pinnejä toisiinsa. Tällöin pienimmän virittävän puun avulla tarvitaan vähiten johtoa luomaan yhteydet.

Saariston yhdistämisessä toisiinsa voidaan myös käyttää pienintä virittävää puuta apuna [3]. Tarkoituksena on, että jokaisesta saaresta pääsee jotain kautta toisiin saariin. Jos jokaisen sillan rakentamiskustannukset voidaan arvioida etukäteen, voi siltojen rakennuspaikat valita siten, että kaikki saaret saadaan yhdistettyä, mutta rakentamiseen menee mahdollisimman vähän rahaa. Virittävällä puulla on myös muita sovelluskohteita, mutta edellä mainittiin esimerkin vuoksi muutama sellainen.

3 Kauppamatkustajan ongelma

Kauppamatkustajan ongelma (TSP) määritellään siten, että myyntimiehellä on joukko kaupunkeja, joissa hänen täytyy vierailla jokaisessa ja palata lopulta lähtöpisteeseen. Tehtävänä on löytää lyhin reitti, jonka avulla hän pääsee käymään jokaisessa kaupungissa [3]. Kauppamatkustajan ongelma on tunnetuimpia ongelmia kombinatorisessa optimoinnissa [5]. Symmetrisessä kauppamatkustajan ongelmassa matka solmusta A solmuun B on sama kuin myös toisinpäin B:stä A:han [4]. Kauppamatkustajan ongelmasta on olemassa myös epäsymmetrinen versio, jossa matka/kustannus solmusta A solmuun B ei ole sama kuin B:stä mennessä A:han [4]. Verkossa saattaa olla myös yksisuuntaisia kaaria, jolloin samaa kaarta pitkin ei pääse molempiin suuntiin [4].

Kuvassa 5 on esimerkki reitistä, joka käy jokaisessa solmussa. Kyseessä on *avoimen kierroksen (open loop) TSP* -variantti, jolloin ei palata lähtöpisteeseen. Verkossa ei ole yksisuuntaisia kaaria, ja verkko on muutenkin symmetrinen. Tutkielmassa keskitytään symmetriseen avoimen kierroksen kauppamatkustajan ongelmaan.



Kuva 5: TSP-ratkaisu

3.1 Ongelman historia ja vaikeus

Laporte [5] kertoo omassa artikkelissaan, että osasyy TSP:n suosiolle on sen määritelmän yksinkertaisuus ja samalla ongelman ratkaisun laskennallinen monimutkaisuus. Toisaalta Applegate [3] omassa kirjassaan esittää kysymyksen, onko TSP oikeasti hankala ratkaista. Usein TSP määritellään hankalaksi ratkaista, mutta Applegaten mukaan totuus on, että sitä ei oikeasti tiedetä, kuinka hankala ongelma TSP oikeasti on [3].

Löytyykö TSP:hen sitten hyvää ratkaisualgoritmia? Applegaten [3] mukaan tuohonkaan kysymykseen ei ole vielä löytynyt vastausta. Todisteena hän käyttää myös sitä, että Clay Mathematics Institute tarjoaisi miljoonan dollarin palkinnon taholle, joka kehittäisi polynomiajassa toimivan algoritmin kyseiseen ongelmaan tai todistaisi sellaisen algoritmin olemassa olon tai olemassa olemattomuuden.

Kauppamatkustajan ongelman nimen alkuperä on hämärän peitossa. Missään dokumentissa ei ole mainittu termin alkuperäistä kehittäjää, eikä tietoa ole milloin termiä alettiin alun perin käyttää. Joka tapauksessa 1950-luvun puolivälissä termi oli jo laajassa käytössä. On spekuloitu, että TSP:n nimi syntyi 1930–1940-luvuilla, jolloin ongelmaa alettiin tutkia tosissaan [3].

3.2 Esimerkkejä käyttökohteista

Logistiikka on yksi TSP:n käyttökohteista. Ihmisten, tavaroiden ja kulkuneuvojen liikkeiden suunnittelu erilaisia matkoja varten on toinen sen yleisistä käyttökohteista. [3]

Loma- tai työmatkaa suunnitellessa harva kuitenkaan etsii ratkaisua reititysongelmaan TSP:tä käsittelevistä kirjoista. Usein kuitenkin matkaa suunnitellessa saatetaan käyttää apuna ohjelmaa, joka käyttää TSP:n ratkaisevaa algoritmia. Karttaohjelmat voivat ratkaista TSP-ongelman pienelle määrälle kaupunkeja, jolloin se riittää usealle turistille tai kauppamatkustajalle. [3]

Myös koulubussin ajoittaminen ja reitin valitseminen siten, että kaikki oppilaat saadaan kyytiin ja lopulta päästään kouluun, on yksi TSP:n käytännön sovellus.

Useampi yritys on tänä päivänäkin erikoistunut koulubussien reititys ohjelmistoihin, joiden avulla oppilaiden kyytien reittejä ja aikatauluja voidaan optimoida. [3]

TSP:tä voidaan käyttää apuna myös genomien sekvensoinnissa. TSP tarjoaa työkalun sekvenssien rakentamiseksi kokeellisesta datasta. [3]

Tulostettuja piirilevyjä käytetään usein elektroniikassa liittämään piirejä, jolloin piirisiruja voidaan liittää muihin laitteistoihin jonkin tietyn toiminnallisuuden saavuttamiseksi. Piirilevyt sisältävät suuren määrän reikiä mikrosirujen liittämistä varten sekä eri piirilevyjen kerrosten yhdistämisen takia. Usein reiät porataan käyttämällä automatisoitua porauskonetta, joka liikkuu määriteltyjen porauskohtien välillä. TSP:n avulla minimoidaan poranterän paikasta toiseen kulkuun menevä aika. [3]

Valmistuksen jälkeinen testaus on kriittinen askel tietokoneen piirisirujen tuottamisessa. Sirujen ja piirilevyjen monimutkaisuus tekee niiden luotettavasta valmistamisesta vaikeaa, mikä aiheuttaa turhaa hävikkiä. Skannausketju yhdistää komponentit/skannauspisteet toisiinsa siten, että niistä syntyy polku syöteyhteyden ja ulostulon välille. TSP:n avulla skannauspisteiden järjestys voidaan määritellä siten, että ketju on mahdollisimman lyhyt. [3]

Planeettoja, tähtiä ja galakseja voidaan tutkia teleskooppien avulla [3]. TSP:tä voidaan käyttää tarkkailussa käytetyn laitteiston kääntämiseen oikeaan asentoon [3]. Suurikokoisen teleskoopin kääntäminen oikeaan kohtaan on aikaa vievää. Kääntyminen toimii tietokoneohjattujen moottoreiden avulla. TSP:n avulla voidaan minimoida kääntymiseen käytettävää aikaa joukolle avaruudessa olevia tutkimuskohteita. Hyvin kalliiden laitteiden tehokas käyttö vaatii hyvän ratkaisun reititysongelmaan.

Yksi data-analyysin ja koneoppimisen perusongelmista on järjestellä informaatiota ryhmiin samanlaisten ominaisuuksien perusteella eli klusterointi. Kauppamatkustajan ongelman käyttö klusterointiongelman ratkaisun apuna on ollut usean tutkimuksen aihe. [3]

3.3 Ratkaisualgoritmeja

3.3.1 Cristofidesin algoritmi

Tarkkojen algoritmien haittapuolena on, että niiden suoritusaika voi olla hyvinkin pitkä. Heuristisen metodien etuna on taas, että ne tuottavat hyviä ratkaisuja kohtalaisen isoihinkin ongelmiin kohtuu ajassa, mutta eivät takaa optimaalisuutta [13].

Cristofidesin algoritmi on O(n^3) aikavaativuuden heuristinen algoritmi kauppamatkustajan ongelman ratkaisuun [9]. Cristofidesin algoritmin tekee mielenkiintoiseksi se, että myös siinä käytetään pienintä virittävää puuta apuna kauppamatkustajan ongelman ratkaisuun. Ratkaistavan kauppamatkustajan ongelman etäisyysmatriisi täytyy täyttää kolmioepäyhtälön vaatimuksen, jotta algoritmia voidaan käyttää [9].

Algoritmi voidaan jakaa useampaan eri alivaiheeseen. Ensimmäisessä vaiheessa lasketaan pienin virittävä puu graafille, josta halutaan ratkaista kauppamatkustajan ongelma [9]. Tämä vaihe on sama kuin pienimmän virittävän puun muutto kauppamatkustajan ongelmaan algoritmissa, joka esitellään myöhemmässä luvussa.

Tämän jälkeen luodaan joukko, joka sisältää ne solmut, joiden kaarien määrä on pariton. Luodun joukon koko on aina parillinen. Luodusta joukosta saadaan aikaan aliverkko, jossa täydellinen yhteensovittaminen on aina mahdollista, parillisesta kardinaliteetista johtuen. Täydellisellä yhteensovittamisella tarkoitetaan sitä, että käytettyjen kaarien määrä on tasan puolet joukossa olevien solmujen määrästä. Tämä yhteensovitus yhdistetään aiemmin luotuun pienimpään virittävän puuhun. Tuloksena on verkko, jolle suoritetaan Eulerin kierros. Lopulta verkosta voidaan laskea Hamiltonin kierros poistamalla reitistä useammat vierailut samoihin solmuihin. Oikoreittien seurauksena on valmis kauppamatkustajan reitti. [9]

Algoritmin antama tuloksen suhdeluku on vähemmän kuin 3/2 optimaalisesta kauppamatkustajan ongelman ratkaisusta [9].

3.3.2 Lin-Kernighan-algoritmi

Lin-Kernighan-algoritmi on myös heuristinen algoritmi kauppamatkustajan ongelman ratkaisuun [13]. Sen avulla voidaan generoida optimaalisia ja lähes optimaalisia ratkaisuja symmetriseen kauppamatkustajan ongelmaan.

Lin-Kernighan-algoritmi kuuluu paikallishakualgoritmien luokkaan [14]. Paikallishakualgoritmit aloittavat jostain kohtaa hakualuetta (search space) ja liikkuvat sitten nykyisestä ratkaisusta johonkin sen naapuriratkaisuun [14]. Paikallishakualgoritmeissa tehdään erilaisia vaihtoja, joiden avulla voidaan muuttaa yksi mahdollinen ratkaisuvaihtoehto toiseen. Algoritmi tekee toistuvasti kaarien vaihtoa sen hetkiselle kauppamatkustajan reitille sen pituuden lyhentämiseksi, kunnes se päätyy pisteeseen, jolloin mikään vaihto ei enää paranna tulosta.

Lin-Kernighan-algoritmi tekee *k-opt* nimellä kutsuttuja muutoksia reiteille. Yksittäinen k-opt-muutos muuttaa reittiä korvaamalla k määrän kaaria olemassa olevasta reitistä k määrällä uusia kaaria siten, että lopputuloksena on lyhyempi reitti. [14]

3.3.3 Paranneltu Christofidesin algoritmi

Hyong-Chan, Kleinberg ja Shmoys [16] esittelevät parannellun version Cristofidesin algoritmista reittiversiolle kauppamatkustajan ongelmasta. Cristofidesin 3/2 likimääräinen algoritmi antaa silti parhaan tuloksen normaalille kauppamatkustajan ongelmalle, mutta reittivariantille voidaan saada parempia tuloksia [16]. Tutkimuksessa esitellään deterministinen $(1+\sqrt{5})/2$ [1] -approksimaatioalgoritmi metriselle ST-polku kauppamatkustajan ongelmalle. Tällä saavutetaan parannus verrattuna ST-polulle aiemmin tunnetulle 5/3 rajalle [16].

Algoritmille annetaan syötteeksi täydellinen verkko kustannusfunktiolla sekä alku- ja loppupiste, joiden avulla lasketaan optimitulos polkuvariantille *Held-Karp-väljennyksestä* [16]. Tämän jälkeen edellinen tulos hajotetaan yhdistelmäksi useampia virittäviä puita eri kertoimilla. Varsinainen virittävä puu tehdään näiden erillisten virittävien puiden kerrointen todennäköisyysjakauman perusteella. T on joukko

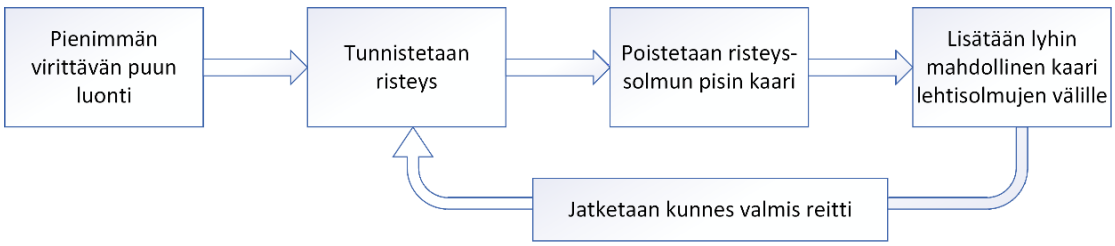
[1] (noin 1,62)

solmuja väärillä pariteeteilla, eli se sisältää puuhun sisältyvät tavalliset solmut, joista lähtee pariton määrä kaaria ja päätepisteet, joista lähtee parillinen määrä kaaria. *J* on kaarien joukko. Algoritmi etsii pienimmän *T* liittymän *J*:hin ja ST-Eulerin polun multigraafille. Eulerin polun avulla saadaan laskettua varsinainen Hamiltonilainen reitti. Lopputuloksena on Hamiltonilainen reitti alku- ja loppupisteen välillä. [16]

Algoritmi on siis muokattu Cristofidesin algoritmista siten, että algoritmi valitsee alustavan virittävän puun *Held-Karp-väljennyksen* optimituloksen perusteella eikä käytä normaalisti käytettyä pienintä virittävää puuta [16]. Tämän parannuksen ansiosta algoritmilla päästään parempaan approksimaatiosuhteeseen kuin perinteisellä Cristofideksen algoritmilla [16].

4 Pienimmän virittävän puun käyttö kauppamatkustajan ongelmaan

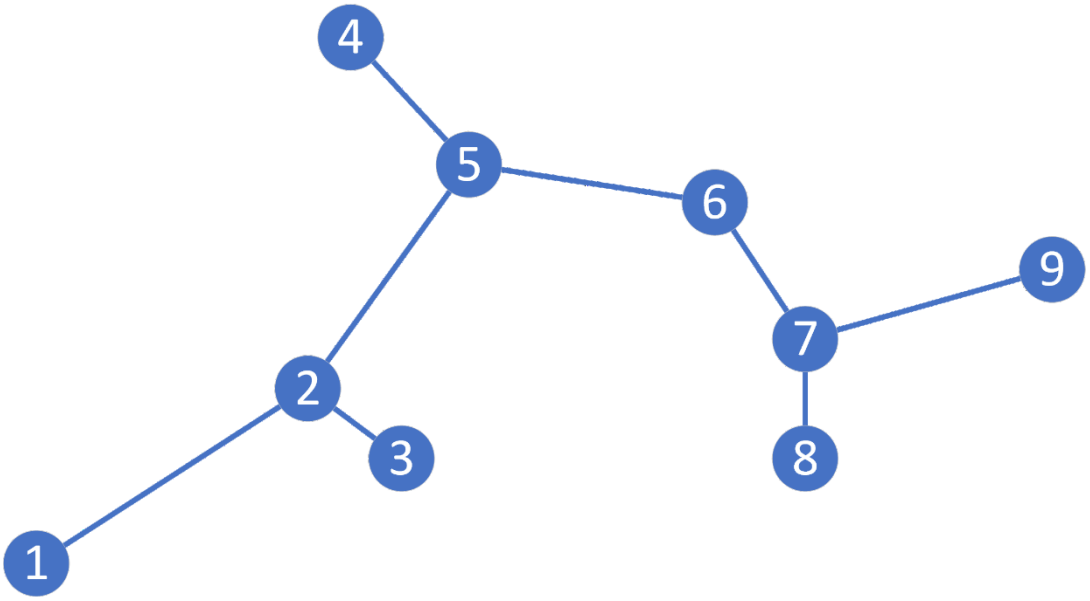
Algoritmin toimintaperiaatteena on ottaa valmis pienin virittävä puu ja muuttaa se ratkaisuksi avoimen kierroksen kauppamatkustajan ongelmaan. Esimerkiksi aiemmin mainitulla Primin algoritmilla luodaan syötteenä olleesta verkosta pienin virittävä puu. Sen jälkeen puusta tunnistetaan risteys, jossa puu haarautuu. Risteyksen pisin kaari poistetaan. Tämän jälkeen lisätään kaari sellaisten lehtisolmujen välille, jotka kuuluvat eri aligraafiin ja kaari ei ole sellainen, joka poistettiin jossain aiemmassa vaiheessa. Verkon solmujen läpikäyntiä sekä kaarien poistoa ja lisäystä jatketaan, kunnes verkossa ei ole enää risteyksiä. Poistojen määrä riippuu virittävässä puussa olevien risteysten määrästä sekä risteyksistä lähtevien kaarten määrästä. Poistoja tarvitaan ylimääräisten kaarien määrä. Tässä vaiheessa tuloksena on avoimen kierroksen kauppamatkustajan reitti, joka voi sisältää risteäviä kaaria. Reittiin sisältyy tasan kaksi lehtisolmua. Lopuksi tulosta voidaan optimoida poistamalla ristikkäiset kaaret, jos sellaisia sattuu kyseiseen ratkaisuun sisältymään.



Kuva 6: Kaavio algoritmin toiminnasta

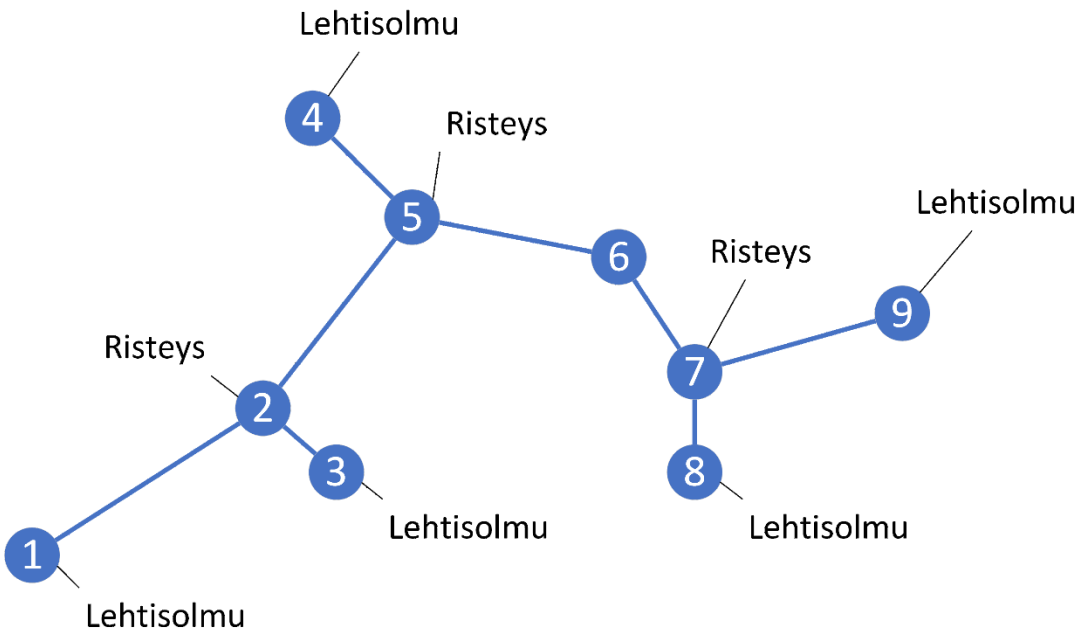
4.1 Algoritmin yksityiskohtainen toiminta

Aluksi syötteenä olevasta verkosta luodaan pienin virittävä puu (MST). Tämä voidaan tehdä esimerkiksi jo aiemmin mainitulla Primin algoritmilla. MST:n luonnissa käytetyllä algoritmilla ei ole vaikutusta varsinaisen algoritmin toimintaan, vaan tärkeää on vain, että lopputulokseksi saadaan virittävä puu. Kuvassa 7 esimerkki pienimmästä virittävästä puusta esimerkkinä käytetylle verkolle.



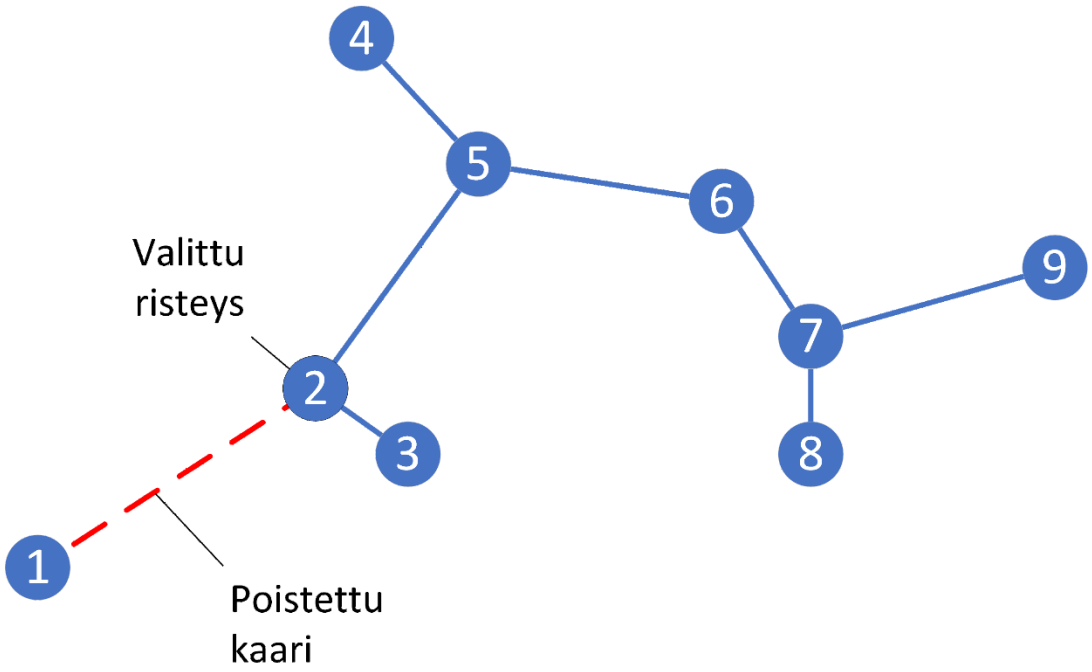
Kuva 7: Pienin virittävä puu

Seuraavaksi pienimmästä virittävästä puusta identifioidaan risteykset (knots). Tämä sen takia, että kaikista risteyksistä halutaan päästä eroon, koska lopullinen ratkaisu ei voi sisältää yhtään risteystä. Verkon solmut käydään yksi kerrallaan läpi ja jos huomataan, että solmusta lähtee useampi kuin kaksi kaarta, otetaan kyseinen solmu käsittelyyn. Verkon solmut on numeroitu, koska algoritmi ottaa syötteenä listan, jossa solmut ovat, ja solmujen läpikäynti tapahtuu numerojärjestyksestä. Tämän takia ensimmäiseksi valittu risteys on solmu numero 2. Algoritmi antaa eri tuloksen, jos solmut ovat listassa eri järjestyksessä. Silloin puun läpikäyntijärjestys on erilainen. Kuva 8 havainnollistaa tätä algoritmin vaihetta.



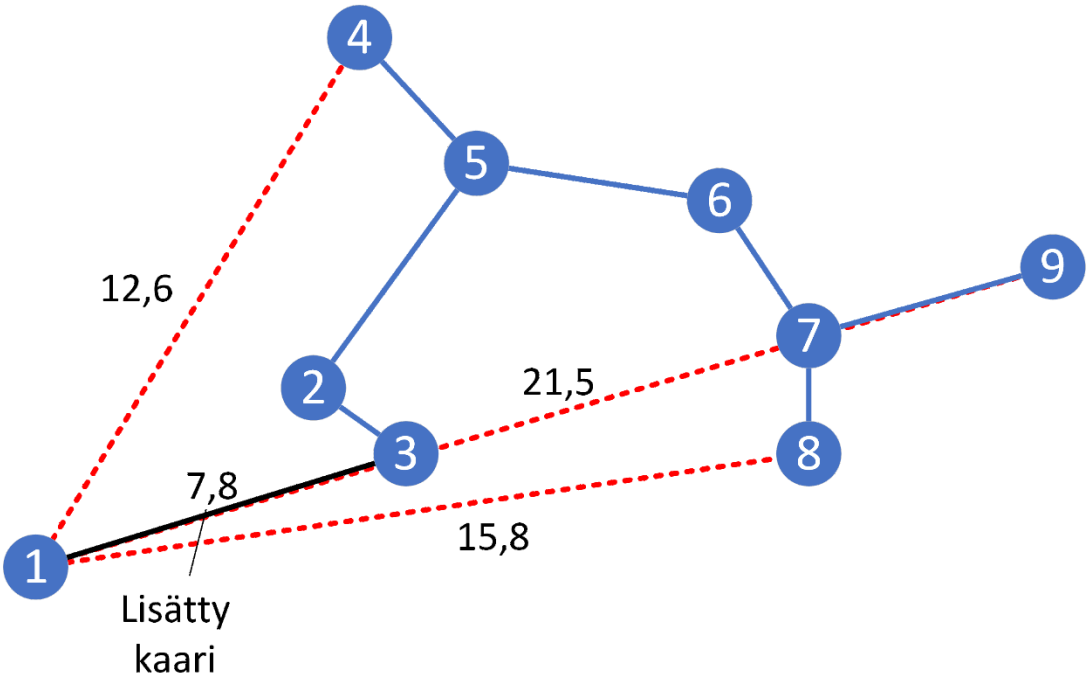
Kuva 8: Risteyksen valitseminen

Tämän jälkeen algoritmi poistaa valitusta risteyksestä lähtevän pisimmän kaaren. Tässä vaiheessa toimitaan ahneen algoritmin tavoin eli poistetaan aina solmun pisin kaari. Myös jokaisesta solmusta lähtevien kaarien määrä päivitetään. Kuvassa 9 on merkitty punaisella katkoviivalla poiston kohteena oleva kaari.



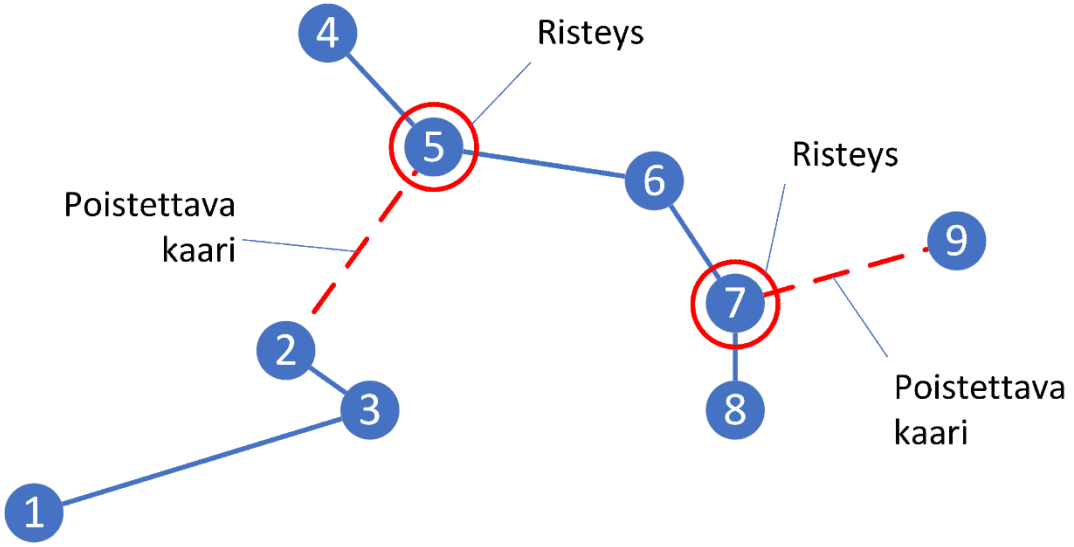
Kuva 9: Pisimmän kaaren poisto valitusta risteyksestä

Kaaren poiston seurauksena syntyy kaksi aligraafia, jotka eivät ole kytkeytyneinä toisiinsa. Niinpä syntyneet graafit täytyy yhdistää toisiinsa uudella kaarella. Uusi kaari lisätään kahden toisiaan lähimpänä sijaitsevan lehtisolmun välille, jolloin päästään eroon yhdestä risteyksestä. Myös kaaren lisäys suoritetaan ahneesti eli lyhin kaari valitaan lisättäväksi. Kaaren lisäyksen jälkeen päivitetään solmuista lähtevien kaarien määrä jokaiselle solmulle. Tämä sen takia, jotta risteysten tarkistaminen olisi mahdollista. Kuva 10 havainnollistaa kaaren lisäystä lehtisolmujen välille. Kaari lisätään solmujen 1 ja 3 välille, koska se on lyhin mahdollinen lisäys. Katkoviivalla on esitetty muut lisäysvaihtoehdot.



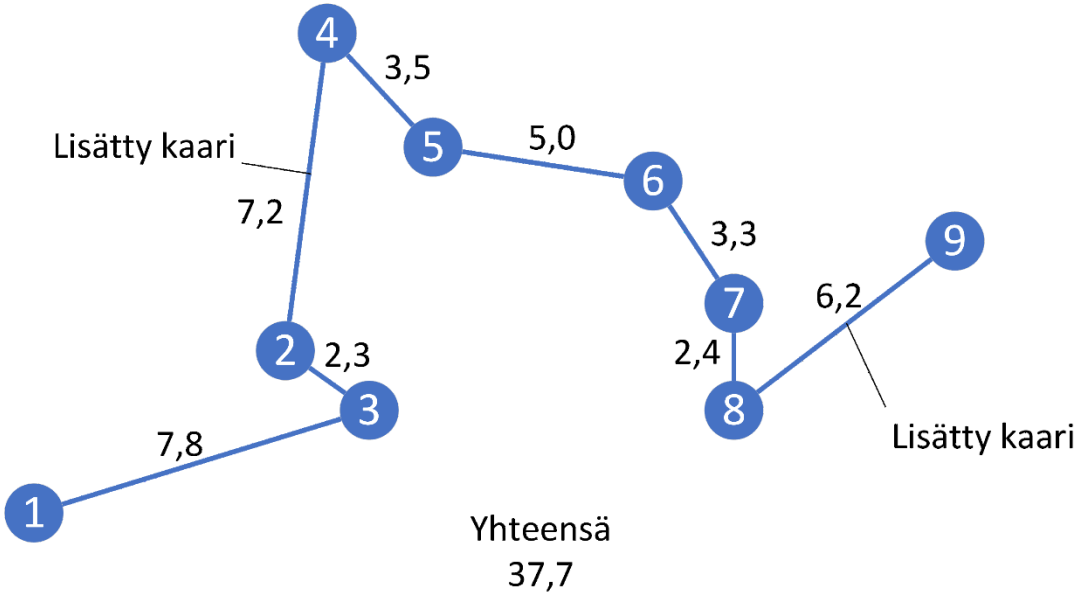
Kuva 10: Kaaren lisäys lehtisolmujen välille

Kaaren lisäyksen jälkeen tarkistetaan, sisältyykö graafiin risteyksiä ja onko lehtisolmuja jäljellä tasan kaksi kappaletta. Jos edellä mainitut ehdot eivät täyty, valitaan seuraava risteys, jonka pisin kaari poistetaan ja tilalle lisätään kaari eri aligraafiin kuuluvien lehtisolmujen välille. Algoritmin edellisiä vaiheita toistetaan, kunnes kaikki risteykset on saatu poistettua.



Kuva 11: Risteyksistä halutaan päästä eroon

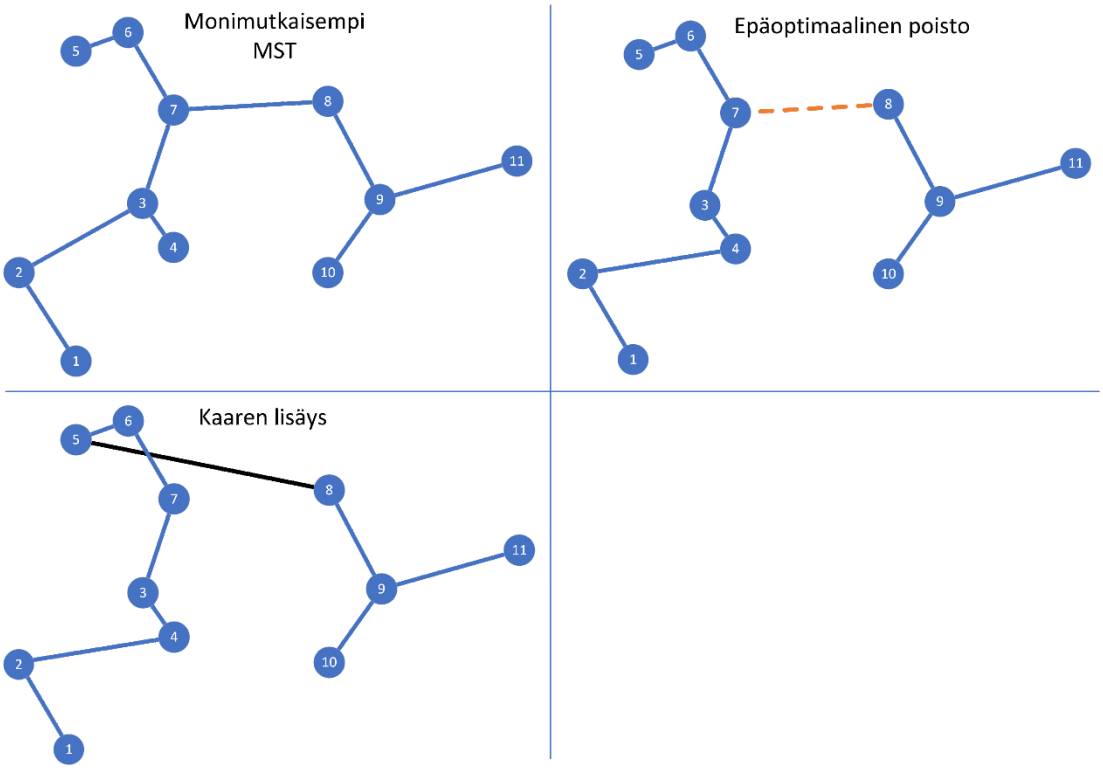
Lopputuloksena on valmis kauppamatkustajan reitti. Valmiissa reitissä ei ole jäljellä yhtään risteystä ja lehtisolmuja on jäljellä tasan kaksi kappaletta. Kuvassa 12 on valmis reitti. Kuvaan on lisätty kaarten pituudet ja reitin kokonaispituus



Kuva 12: Valmis kauppamatkustajan reitti

4.2 Algoritmin heikkouksia

Joissakin tilanteissa algoritmi ei toimi optimaalisesti. Ahneen toimintaperiaatteen vuoksi esimerkiksi kaaren poistovaiheessa valitaan aina solmusta lähtevä pisin mahdollinen kaari, vaikka reitin muodostuksen kannalta se ei aina olisi paras mahdollinen vaihtoehto. Joissain tapauksissa olisi parempi poistaa jokin muu kaari. Joskus jopa lyhin voi tuottaa parhaimman tuloksen. Lisäksi ahne suoritusperiaate voi aiheuttaa sen, että valmiiseen TSP-reittiin sisältyy leikkauskaaria eli kaaria, jotka risteävät jossakin kohdassa. Optimaalisessa ratkaisussa ei voi olla leikkauksia, koska näitä sisältävä ratkaisu on aina huonompi kuin sellainen, joka ei sisällä niitä. Alla esitellään tilanne, jossa nämä algoritmin heikkoukset tulevat esiin. Kuvassa 13 on esimerkki monimutkaisemmasta pienimmästä virittävästä puusta, jolle algoritmi ei löydä optimaalista reittiä.

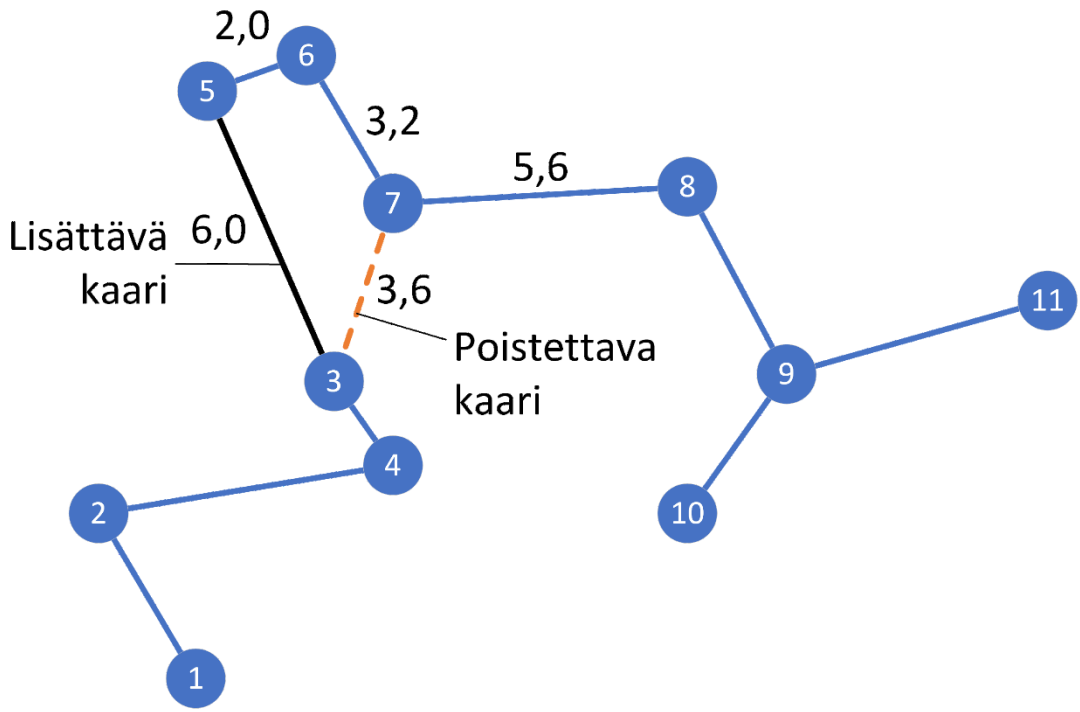


Kuva 13: Esimerkki, jossa epäoptimaalinen poisto tuottaa leikkauksen

Suoritettaessa algoritmia päädymme tilanteeseen, jossa poistettavana kohteena katkoviivalla merkitty kaari. Tässä kohdassa ahne algoritmi ei toimi optimaalisesti, koska se poistaa kyseisen kaaren. Tämän jälkeen algoritmi ei voi enää antaa optimitulosta kyseiselle puulle.

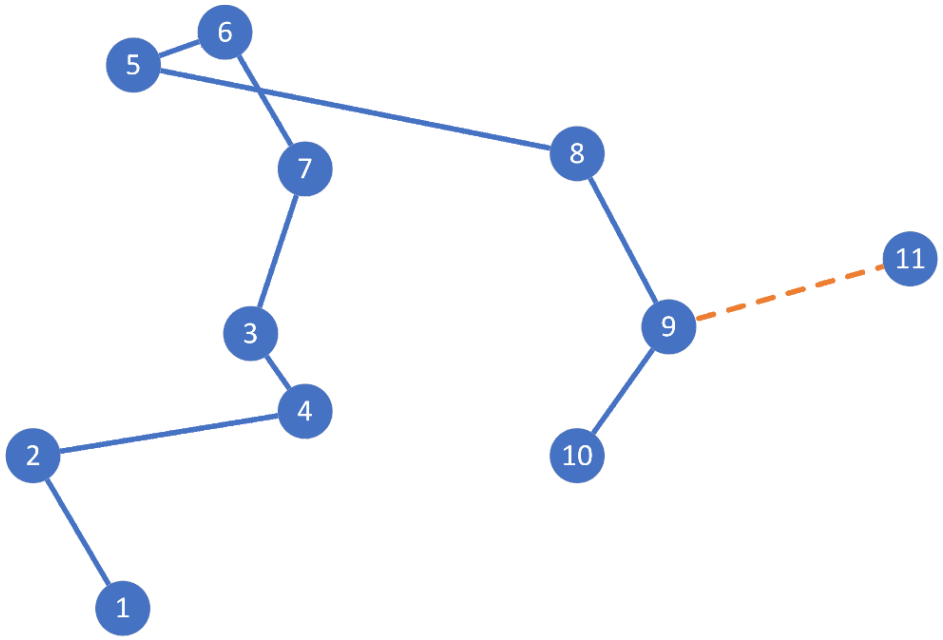
Lisättäessä mahdollisimman lyhyen kaaren kahden lehtisolmun välille päädymme kuvan esittämään viimeiseen vaiheeseen. Vaikka lisäämme lyhyimmän mahdollisen kaaren, reitti sisältää kaarien leikkauksen, koska edellinen kaaren poisto oli huono.

Parempi vaihtoehto olisi olla poistamatta pisintä kaarta, ja valita tässä tapauksessa toiseksi pisin kaari poistettavaksi. Tällöin vältyttäisiin turhilta kaarien leikkauksilta ja reitti olisi siten parempi. Tällöin solmujen 3–8 välillä olevan reitin summaksi tulee 16,8. Jos taas reittiin sisältyy leikkaus, kuten edellisen kuvan 14 esimerkissä, osareitin summaksi tulee 18,1 jolloin kokonaisreitin pituus on suurempi.



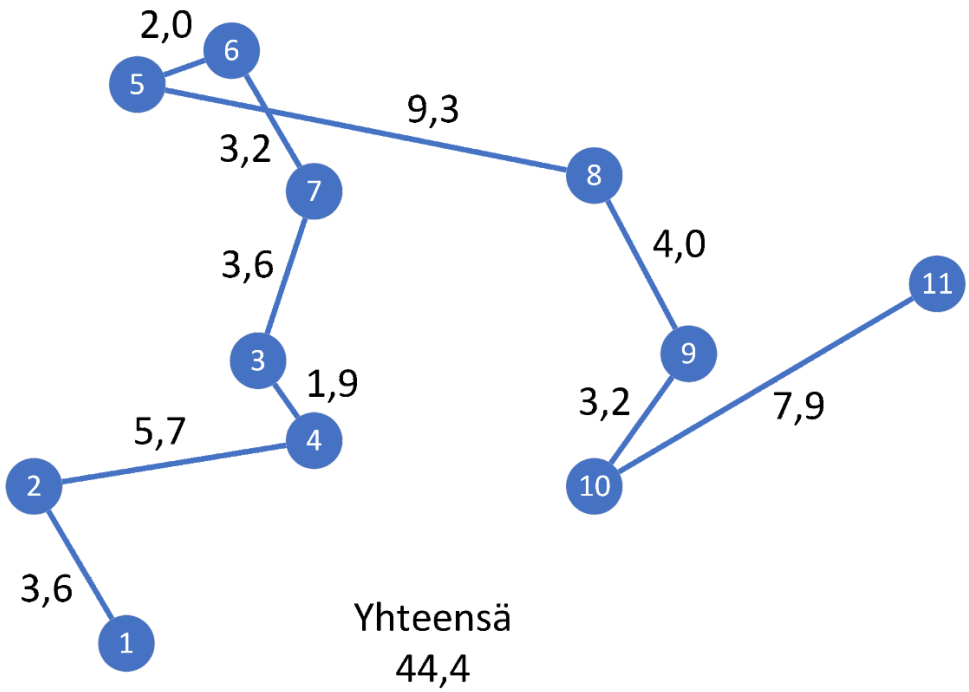
Kuva 14: Paremman tuloksen antava kaaren poisto ja lisäys

Myös kuvan 15 esittämässä vaiheessa algoritmi tekee epäoptimaalisen poiston. Vaikka katkoviivalla merkityn kaaren poisto ja sen jälkeinen kaaren lisäys ei aiheuta tässä tapauksessa risteävien kaarien muodostumista, tuloksena saatu reitti ei ole optimaalinen. Tässä nähdään toistamiseen algoritmin ongelma. Ahne valintaperiaate kaaren poistolle aiheuttaa sen, että saatu reitti ei ole välttämättä optimaalinen.



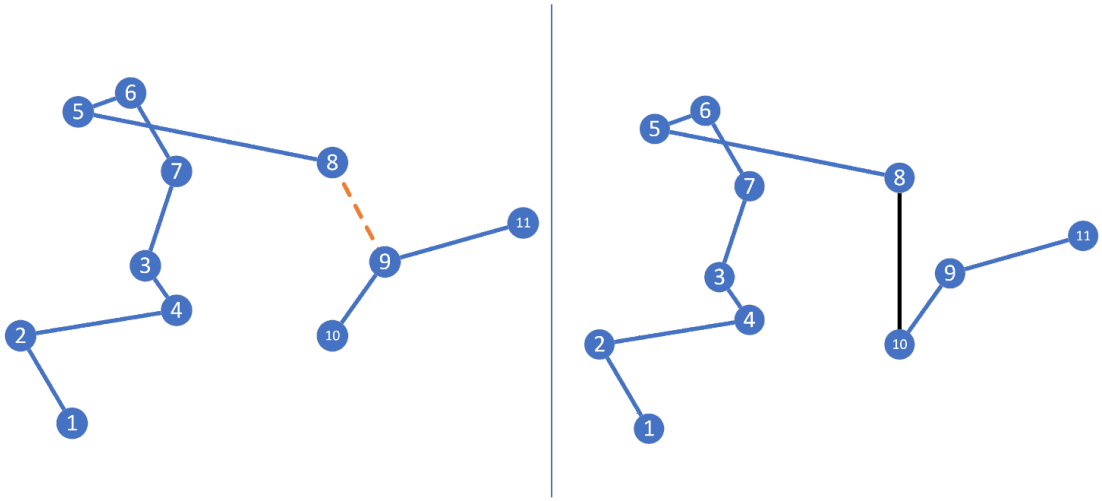
Kuva 15: Epäoptimaalinen poisto

Algoritmin antamana lopputuloksena on kuvan 16 mukainen reitti. Ratkaisuun sisältyy leikkauskohtia, ja lyhyempi reitti olisi mahdollinen. Algoritmi antaa optimituloksen vain silloin kun reitin muodostuksen kannalta on optimaalisinta poistaa solmukohdasta lähtevä pisin kaari. Jos taas olisi optimaalisempaa poistaa lyhin tai jokin pisimmän ja lyhimmän väliltä, ei lopputulos ole paras mahdollinen.



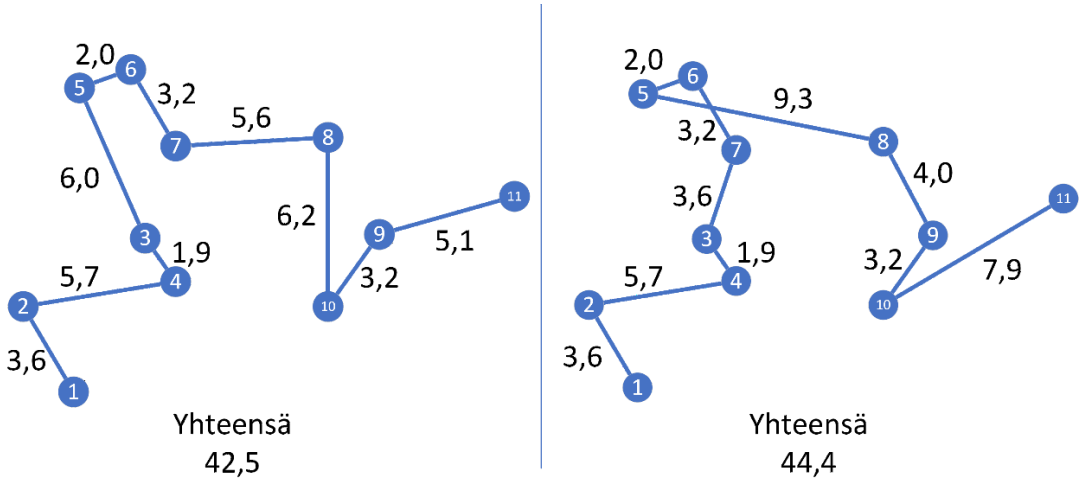
Kuva 16: TSP-reitti monimutkaisemmalle MST:lle

Parempi vaihtoehto tässäkin tapauksessa olisi ollut poistaa toiseksi pisin kaari, kuten kuvasta 17 näkyy. Tällöin päädytään tulokseen, jolloin reitti on lyhempi ja siten parempi.



Kuva 17: Parempi kaaren poisto ja lisäys

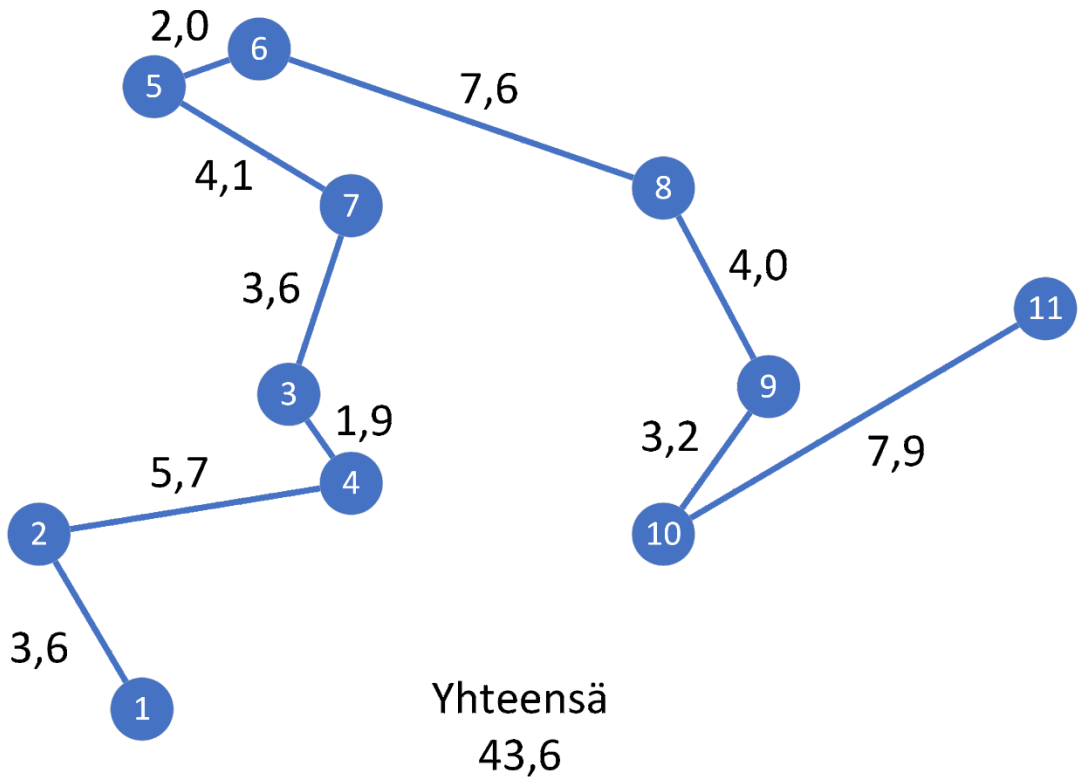
Poistamalla aina parhaan vaihtoehdon eli ensimmäisellä kerralla pisimmän ja toisella ja kolmannella kerralla toiseksi pisimmän kaaren, lopputuloksesta olisi tullut kuvan 18 vasemmanpuolisen reitin mukainen. Sen antama tulos on parempi, kuin varsinaisen algoritmin antama tulos oikealla.



Kuva 18: Reitti, jos ei poisteta aina pisintä kaarta, sekä algoritmin antama tulos

4.3 Saadun reitin optimointi

Reitin epäoptimaalisen luonnin jälkeen vaihtoehtona on myös muokata saatua reittiä. Kuvassa 19 kaarien leikkaus on poistettu. Tällöin reitin pituus lyhenee. Toisaalta sekään, että reittiin ei sisälly leikkaavia kaaria, ei vielä takaa reitin optimaalisuutta. Tässäkään tapauksessa reitin loppupää (oikealla) ei ole optimaalinen.



Kuva 19: Leikkauksien poisto

4.4 Algoritmin aikavaativuus sekä pseudokoodi

Alla on lueteltu algoritmin toiminnan kannalta tärkeimmät metodit sekä esitetty ne pseudokoodina. Ensimmäisenä esittelyssä on Primin algoritmi. Se luo pienimmän virittävän puun, jota tarvitaan varsinaisen algoritmin toimintaa varten. Kyseisen toteutuksen aikavaativuus O(n^3). n on solmujen määrä syötteessä

prim() O(n^3)

```
Prim(edges,numberofpoints)
    visited <- array size of numberofpoints
    FOR each visited <- false;
    mst <- empty
    ADD first edge TO mst
    status of added edge <- false
    visited[edges.from(0)-1] <- true
    visited[edges.to-1(0)] <- true

    WHILE size of mst < numberofpoints-1 DO
        IF (visited[edges.from(i)-1] AND !visited[edges.to(i)-1]) THEN
            visited[edges.to(i)-1] <- true
            status of current edge <- false
            ADD current edge TO mst
           i<-0

        ELSE IF (!visited[edges.from(i)-1] AND visited[edges.to(i)-1] THEN
            visited[edges.from(i)-1] <- true
            status of current edge <- false
            ADD current edge TO mst
           i<-0
       i++

    RETURN mst
```

replaceEdge: Alla varsinainen päämetodi *replaceEdge()*, joka muuttaa virittävän puun ratkaisuksi kauppamatkustajan ongelmaan. Käyttää apuna metodeja *isFinalForm, getFirstKnot, getHigherFromKnot, getPointsDegree* ja *findEdgeToAdd.*

replaceEdge() O(n^4)

```
  replaceEdge(edgelist,currenttsp, numberofpoints)
    veriticesdegree <- getPointsDegree(currenttsp,numberofpoints)
    WHILE !isFinalForm() DO
        idfirstknot <- getFirstKnot()
        removededge <- getHigherFromKnot(currenttsp, idfirstknot)

        edgelist(removededge).status <-false
        REMOVE edge FROM currenttsp

        veriticesdegree <- getPointsDegree(currenttsp,numberofpoints)
```

```
        edge <- findEdgeToAdd(edgelist,currenttsp)
        ADD edge TO currenttsp

        veriticesdegree <- getPointsDegree(currenttsp, numberofpoints)

    RETURN currenttsp
```

getHigherFromKnot: Apumetodi, joka palauttaa pisimmän kaaren valitusta risteyksestä.

getHigherFromKnot() O(n)

```
GetHigherFromKnot(tsp, knotid)
    FOR i<-0 TO size of tsp DO
        IF currentstartnode = knotid OR currentendnode = knotid THEN
            IF edgeweight > maxweight THEN
                maxlink <- currentlink
                maxweight <- edgeweight
    maximumweightlink.status <- false
    RETURN maximumweightlink
```

getPointsDegree: Apumetodi, joka tarkistaa kuinka monta kaarta jokaisesta solmusta lähtee.

getPointsDegree() O(n)

```
getPointsDegree(tsp,numberofpoints)
    array degrees <- numberofpoints
    FOR each edge in tsp DO
        degrees[edge.to-1]++
        degrees[edge.from-1]++
    RETURN degrees
```

getFirstKnot: Apumetodi, joka etsii ensimmäisen risteyksen virittävästä puusta, eli solmun, josta lähtee enemmän kuin kaksi kaarta.

getFirstKnot() O(n)

```
getFirstKnot()
    firstknot <- 0
    WHILE firstknot.degree<=2 DO
        firstknot <- firstknot+1

    RETURN ++firstknot
```

findEdgeToAdd: Apumetodi kaaren lisäykseen kauppamatkustajan reittiin. Kutsuu metodia *compatibleLink.*

findEdgeToAdd() O(n^2)

```
findEdgeToAdd(edgelist,tsp)
    edge <- new edge
    WHILE edge.from = 0 AND edge.to = 0 DO
        IF edgelist currentedge.status = true AND
         compatibleLink(currentedge,tsp) = true THEN
            edge <- edgelist(currentedge)
    RETURN edge
```

compatibleLink: Apumetodi, joka tarkistaa onko kaari mahdollista lisätä vaiko ei. Käyttää apuna kahta metodia, jotka ovat *onlyLeafs* ja *differentGraph.*

compatibleLink() O(1)

```
compatibleLink(edge, tsp)
    RETURN (result of onlyLeafs(edge) and differentGraph(edge,tsp))
```

onlyLeafs: Tarkistaa, että kaaren molemmat päät ovat lehtisolmuja. Eli niistä ei lähde kuin korkeintaan yksi kaari kummastakin.

onlyLeafs() O(1)

```
onlyLeafs(edge)
    RETURN result of degree edge.from-1 < 2 and degree edge.to-1 < 2
```

differentGraph: Apumetodi, joka tarkistaa, että kaaren molemmissa päissä sijaitsevat solmut eivät kuulu samaan aligraafiin. Tällä estetään silmukoiden syntyminen valmiiseen reittiin.

differentGraph() O(n)

```
differentGraph(edge,tsp)
    set <- empty
    ADD edge.to TO set

    FOR i<-0 TO size of tsp DO
        IF set contains(tsp.to(i)) and does not contain (tsp.from(i)) THEN
            ADD tsp.get(i).from TO set
                i=-1;
        ELSE IF (set does not contain(tsp.to(i)) and contains(tsp.from(i))
         THEN
            ADD tsp.get(i).to TO set
                i=-1;
    RETURN !contain(edge.from) FROM set
```

isFinalForm: Metodi, joka tarkistaa, onko reitti valmis. Jos jostain solmusta lähtee enemmän kuin kaksi kaarta tai ei yhtään kaarta, niin palauttaa epätoden. Palauttaa totta, jos löytyy tasan kaksi lehtisolmua.

isFinalForm() O(n)

```
isFinalForm()
    countleafs <- 0
    FOR EACH verticesdegree
        IF veriticesdegree > 2 or veriticesdegree = 0 THEN
            RETURN false
        ELSE IF verticesdegree = 1 THEN
            countleafs++

    RETURN countleafs=2
```

Algoritmin rakenne on esitetty alla.

```
ConvertMSTtoTSP()
       Prim() O(n^3)
       replaceEdge()
              getPointsDegree() O(n)
              while !isFinalForm() O(n)
                     getFirstKnot() O(n)
                     getHigherFromKnot() O(n)
                     remove edge() O(n)
                     getPointsDegree() O(n)
                     findEdgeToAdd() O(n^2)
                            compatibleLink() O(1)
                                   onlyLeafs() O(1)
                                   differentGraph() O(n)
                     add edge() O(1)
                     getPointsDegree() O(n)
              Return Tsp
```

Solmuja on *n* määrä ja kaikkia kaaria yhteensä $(n * (n-1))/2$ kappaletta verrattuna solmuihin. Kauppamatkustajan reitissä kaaria on *n*-1 kappaletta. Pienimmän virittävän puun luonti vie käytetyllä Primin algoritmin toteutuksella O(n^3).

Tarkistus joka kierroksen alussa, onko reitti valmis, vie ajan O(n). Risteyksen löytäminen, pisimmän kaaren valitseminen ja poisto siitä vie yhteensä O(n) ja solmujen kaarimäärien tarkistus O(n).

Kaaren lisäys vie yhteensä O(n^3), koska läpikäytävänä on kaikki verkon kaaret eli $(n * (n-1))/2$ kappaletta, josta tulisi O(n^2). Lisäksi metodin sisällä on kutsu *DifferentGraph* metodiin, joka vie itsessään O(n) aikaa.

Yhteensä tässä toteutuksessa on kolme sisäistä silmukkaa, joista ulommainen vie O(n), keskimmäinen vie O(n^2) ja sisimmäinen O(n). Siksi aikavaativuus on O(n^4).

Algoritmia voitaisiin optimoida laskemalla kaarien määrän etukäteen esimerkiksi puunluonnin aikana. Sitten kaarien määrää voitaisiin päivittää aina poiston yhteydessä ja risteysten määrän voisi lakea yksinkertaisella for-silmukalla. Tällöin *isFinalForm* muuttuisi yksinkertaisemmaksi.

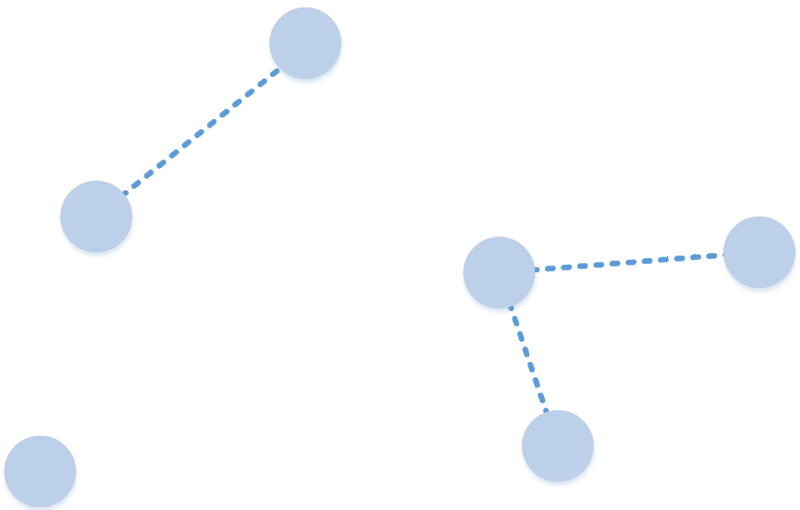
5 Parannusehdotukset algoritmiin

Koska algoritmi ei anna aina optimaalista tulosta, pyrittiin tuloksia parantaa lisäämällä satunnaisuutta. Ideana oli, että algoritmille ei annettaisikaan pienintä virittävää puuta, vaan satunnainen virittävä puu. Tällöin kauppamatkustajan reitistä muodostuisi erilainen ja potentiaalisesti myös lyhyempi.

5.1 Satunnaistettu Prim

Muodostetaan virittävä puu, mutta ei pienintä virittävää puuta. Satunnaisuuden takia tosin myös pienin virittävä puu on mahdollinen tulos, mutta kuitenkin harvinainen. Toimintaperiaate on sama kuin tavallisella Primin algoritmilla ratkaistuna, mutta valitaan lisättävä kaari satunnaisesti aina kolmesta lyhimmästä mahdollisesta vaihtoehdosta.

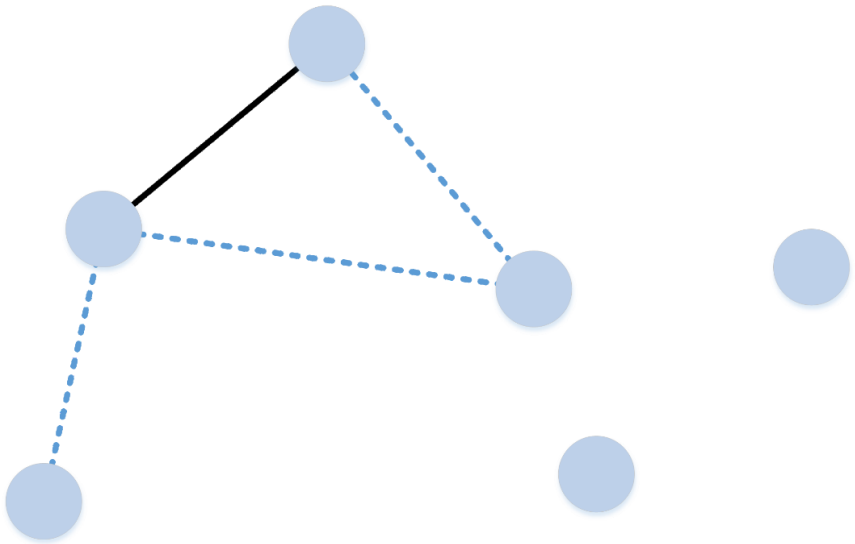
Kuvassa 20 on yksi esimerkki satunnaistetun Primin toiminnasta. Aluksi valitaan kolmesta lyhimmästä kaaresta satunnaisesti yksi, jonka perusteella aletaan rakentamaan puuta. Lyhimmät mahdolliset kaarivaihtoehdot on merkitty sinisellä katkoviivalla.



Kuva 20: Satunnaistettu Prim ensimmäisen kaaren valinta

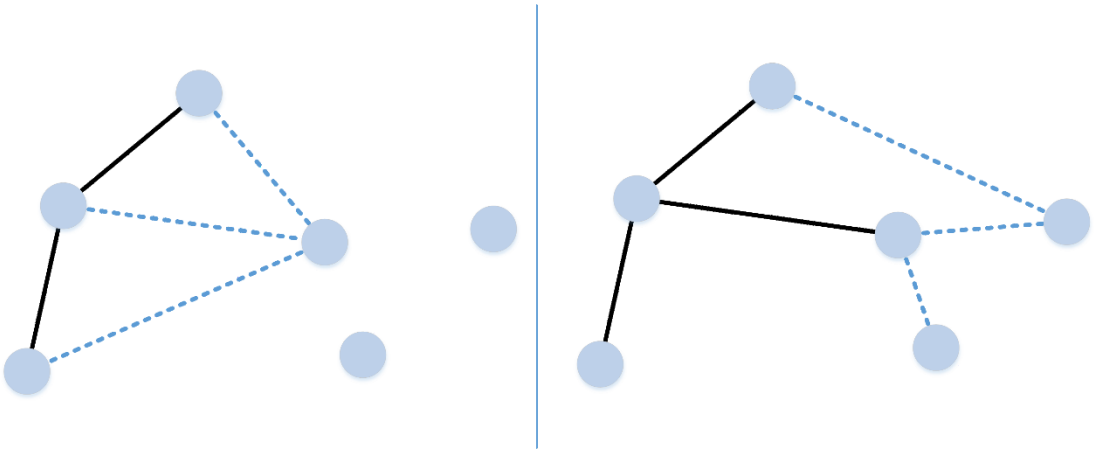
Kuvassa 21 on valittu näistä kaarista satunnaisesti yksi. Valittu kaari on merkitty mustalla kokoviivalla. Lisäksi kuvassa on esitetty seuraavat kolme lyhintä mahdollista kaarta. Kaarten lisäyksessä noudatetaan Primin algoritmin toimintalogiikkaa eli

lisättävän kaaren toisen pään pitää kuulua jo luotuun puuhun ja toisen pään ei tule kuulua puuhun.



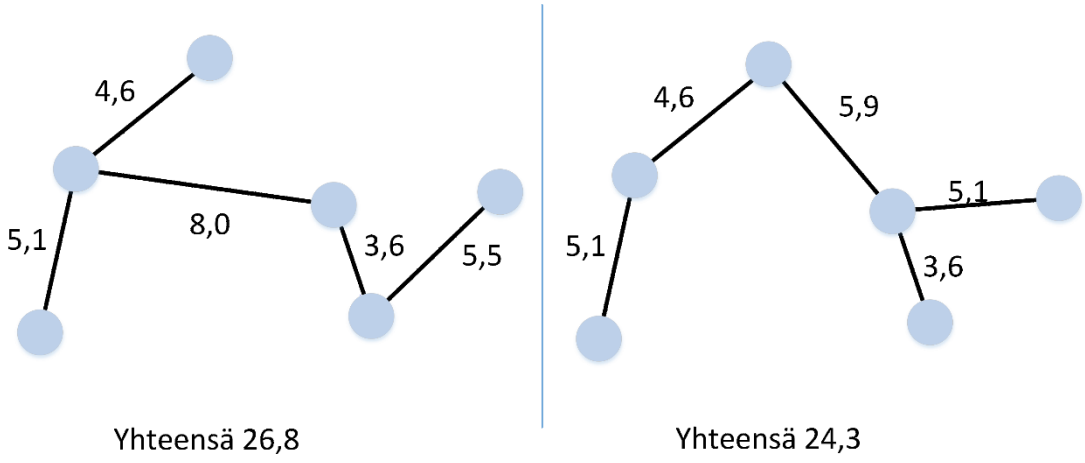
Kuva 21: Ensimmäinen kaari lisätty

Kaarten lisäystä jatketaan samalla tavalla, kunnes tuloksena on virittävä puu. Kuvassa 22 vasemmalla puolella on toisena lisättävän kaaren lisäys ja seuraavat kolme lyhintä vaihtoehtoa. Kuvassa 22 oikealla puolella on lisätty kolmas kaari, sekä seuraavat potentiaaliset vaihtoehdot.



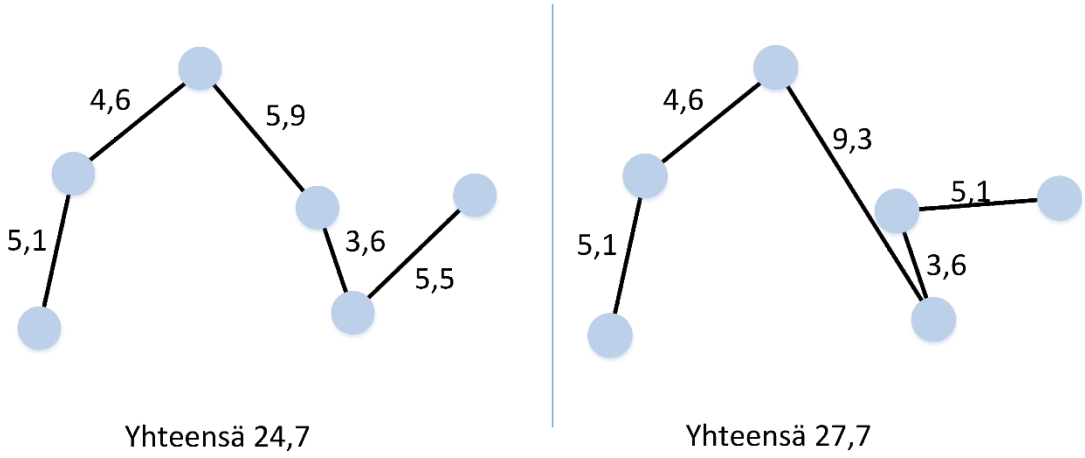
Kuva 22: Toisen kaaren sekä kolmannen kaaren lisäys

Kuvan 23 vasen puoli esittää yhtä mahdollista algoritmin antamaa lopputulosta, jossa kaarten pituudet on merkitty kuvaan ja yhteispituus laskettu. Luotu virittävä puu eroaa huomattavasti verrokkina kuvan oikealla puolella olevaan pienimpään virittävään puuhun. Tässä esimerkkitapauksessa satunnaisesti luotu virittävä puu on 10,3 % pidempi kuin pienin virittävä puu.



Kuva 23: Satunnaistettu Prim -tulos ja pienin virittävä puu samasta verkosta

Algoritmin luomista kauppamatkustajan reiteistä kuvassa 24 voidaan huomata, että satunnaistetulla puunluonnilla saadaan tässä tapauksessa parempi lopputulos kuin käyttämällä pienintä virittävää puuta. Satunnaistuksella saatu reitti on 10,8 % lyhyempi kuin pienimmällä virittävällä puulla saatu lopputulos.



Kuva 24: Satunnaistettu Prim -lopputulos sekä pienin virittävä puu -lopputulos

5.2 Pseudokoodi

Seuraavaksi esitetään satunnaistettu Primin algoritmi pseudokoodina. Se toimii Primin algoritmin tavoin, mutta valitsee aina lisättävän kaaren satunnaisesti kolmesta lyhimmästä mahdollisesta kaaresta, jotka täyttävät lisäysehdon.

random prim() O(n^3)

```
Spanningtree(edges, numberofpoints)

    FOR i <- TO size of edges DO
        current edges.status <- true

    visited2 <- array size of numberofpoints;
    FOR each visited2 <- false
    spannigtreelist <- empty
    indiceslist <- empty
    numberofedges <- size of edges;

    edges_best <- 3;
    IF numberofedges < edges_best THEN
        edges_best <- numnerofedges;

    index <- random integer between 0 and edges_best
    ADD edge TO spanning tree FROM  list with chosen index
    chosen edges.status <- false
    edges.from-1 node in visited2 <- true
    edges.to-1 node in visited2 <- true

    WHILE size of spannigtreelist < numberofpointspoints-1 DO

        FOR i <- 0 TO size of edges DO
            IF (visited2[current edges.from-1] AND !visited2[current
             edges.to-1]) THEN
                IF current edge.status = true THEN
                    ADD i TO indiceslist

            ELSE IF (!visited2[current edges.from-1] AND visited2[current
             edges.to-1]) THEN
                IF current edge.status = true THEN
                    ADD i TO indiceslist

        index2 <- 0;
        best3 <- 3;

        IF size of indiceslist > best3 THEN
            index2 <- Random integer between 0 and best3

        ELSE
            index2 <- Random integer between 0 and size of indiceslist

        ind3 <- 0;
        ind3 <- value of indiceslist FROM index2

        ADD edge TO spanning tree from edges list with chosen index ind3

        chosen edges.status <- false
        edges.from-1 node in visited2 <- true
        edges.to-1 node in visited2 <- true
        CLEAR spannigtreelist

    RETURN st
```

6 Tulokset

Algoritmin toimintaa testattiin myös käytännössä. Algoritmi on toteutettu Java-ohjelmointikielellä ja syötteenä käytettiin O-Mopsi [11] ja Dots [15] -datajoukkoja. O-Mopsi-datajoukko koostuu leveys- ja pituusasteen sisältävistä pisteistä, joiden avulla kauppamatkustajanreittejä laskettiin käyttäen Haversine-funktiota.

$$d = 2r\ arcsin\left(\sqrt{\sin^2(\frac{\varphi_2 - \varphi_1}{2}) + \cos(\varphi_1)\cos(\varphi_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2})}\right)$$

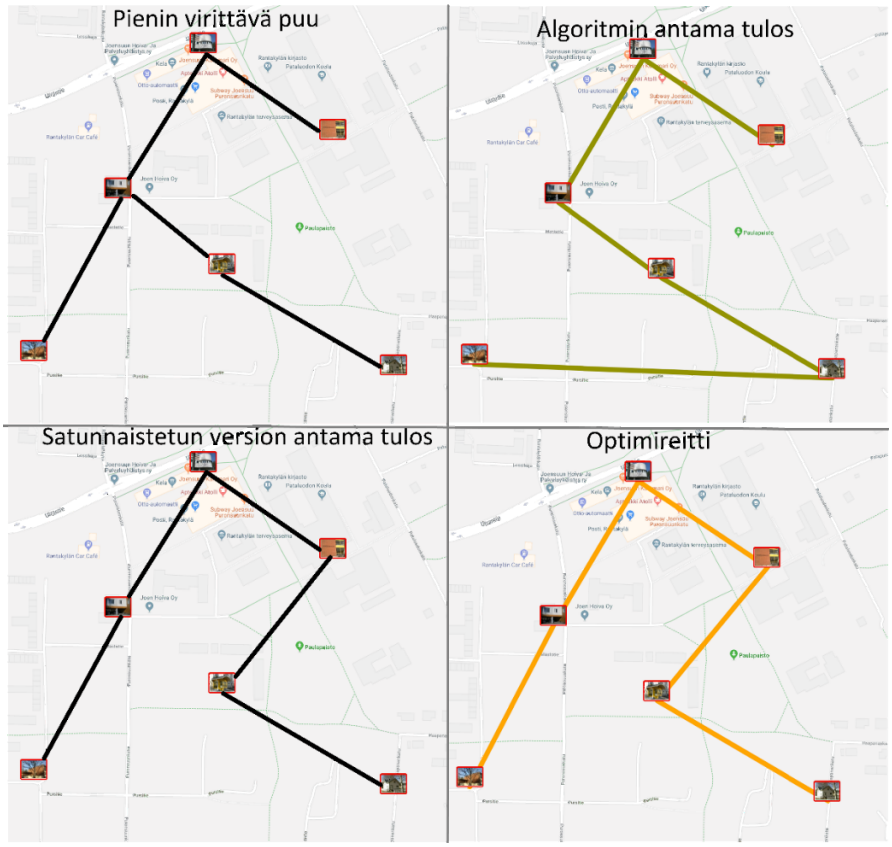
Kaavassa kahden pisteen välistä laskettua pituutta merkitään d:llä. Maapallon säteen r arvona on käytetty lukua 6356.752. Pisteen yksi leveysastetta radiaaneina on merkitty φ-merkillä ja pisteen kaksi leveysastetta radiaaneina φ_2-merkillä. Pisteen yksi ja kaksi pituusasteita radiaaneina on merkitty merkeillä λ_1 ja λ_2.

Dots-datajoukko koostuu myös koordinaateista, mutta koordinaatistona toimi kaksiulotteinen karteesinen koordinaatisto. Satunnaistettua versiota algoritmista ajettiin aina 100 kertaa jokaiselle syötteelle ja reiteistä valittiin lyhin. Näiden avulla algoritmin toimivuutta arvioitiin. Tutkimuksen kohteena oli, kuinka lähelle optimaalista tulosta algoritmi pääsee ja kuinka hyvin se toimii verrattuna muihin saman ongelman ratkaiseviin algoritmeihin. Ensiksi esitellään yksittäisiä tuloksia tarkemmin, jonka jälkeen siirrytään algoritmin avulla saatuihin kokonaistuloksiin O-Mopsi ja Dots -datajoukoille.

6.1 Esimerkki O-Mopsi-datasta

Kuvassa 25 on esitetty pienin virittävä puu yhdelle O-Mopsi-datajoukon instanssille, algoritmin antama lopputulos, satunnaistetun version tulos sekä viimeisenä optimireitti. Kuvasta nähdään, että normaalilla versiolla ei saada optimaalista tulosta kyseiselle syötteelle. Toisaalta nähdään, että satunnaistetun version antama reitti on sama kuin optimireitti. Perusversio antoi reitin pituudeksi 1,345 kilometriä, kun taas optimireitin pituus oli 1,170 km. Joten tässä yksittäistapauksessa perusversion antama

tulos oli n. 15 % pidempi, kun taas satunnaistetulla algoritmilla päästiin optimitulokseen.

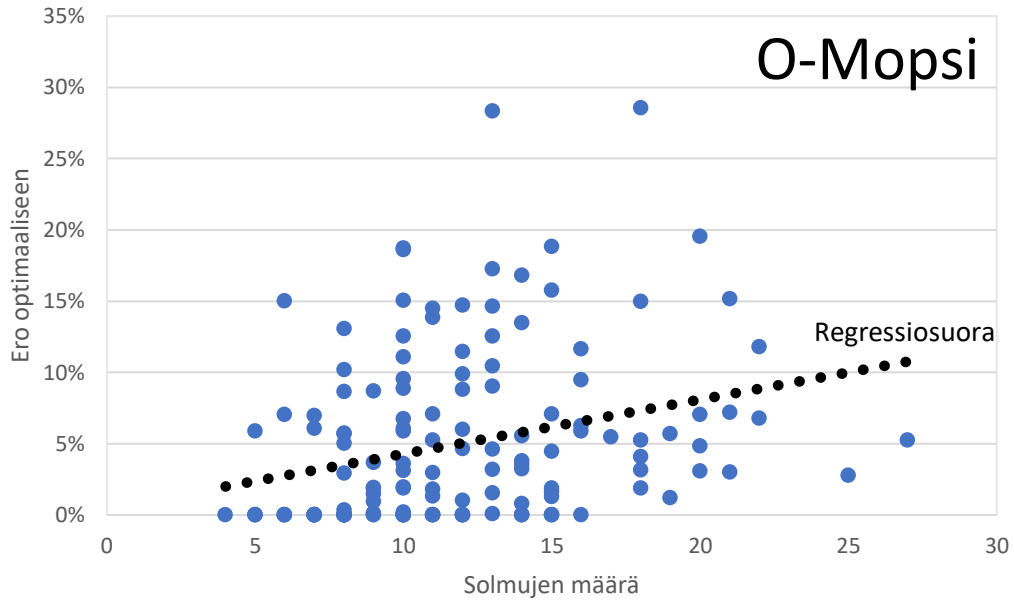


Kuva 25: Esimerkki O-Mopsi-datasta

6.2 Kokonaistulokset

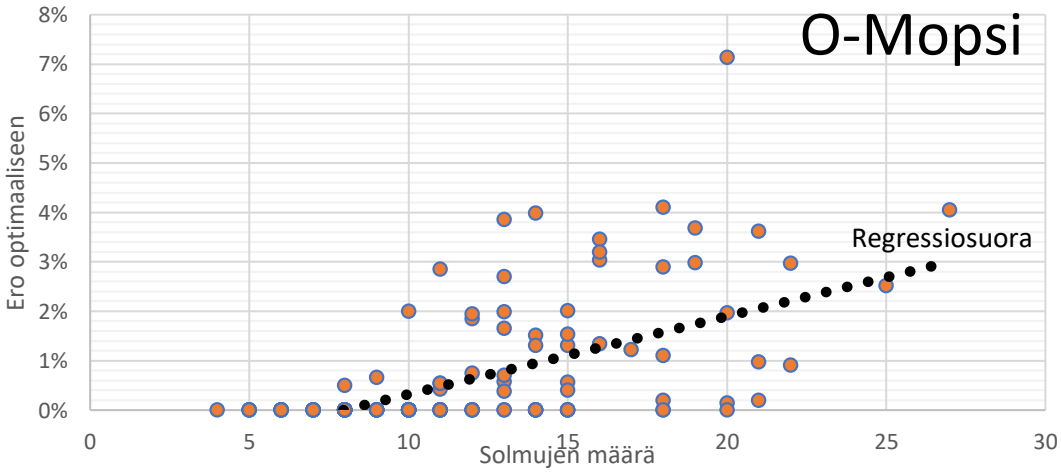
Kuvassa 26 nähdään algoritmin antamat tulokset O-Mopsi-datajoukolle tiivistettynä yhteen kuvaan. Jokainen piste vastaa yhtä datajoukon tulosta, ja vaaka-akselilla nähdään, kuinka monta solmua kyseisessä syötteessä on sekä pystyakselilla, kuinka paljon saatu tulos eroaa optimaalisesta ratkaisusta. Tummemmat pisteet tarkoittavat useampaa tulosta päällekkäin.

Kaikkien tulosten keskiarvoksi saatiin 4,96 %, eli perusalgoritmin antama tulos on keskimäärin tämän verran heikompi kuin optimaalinen. O-Mopsi-datajoukkoon kuuluu 147 eri koordinaattipisteistä koostuvaa syötettä, joista algoritmi antoi optimaalisen tuloksen 50 syötteelle ja huonomman tuloksen 97 syötteelle. Eli noin kolmannekselle syötteistä saatiin optimitulos algoritmin avulla. Huonoimmissa tapauksissa ero optimaaliseen on n. 30 %.



Kuva 26: Ero optimaaliseen tulokseen O-Mopsi-datajoukossa

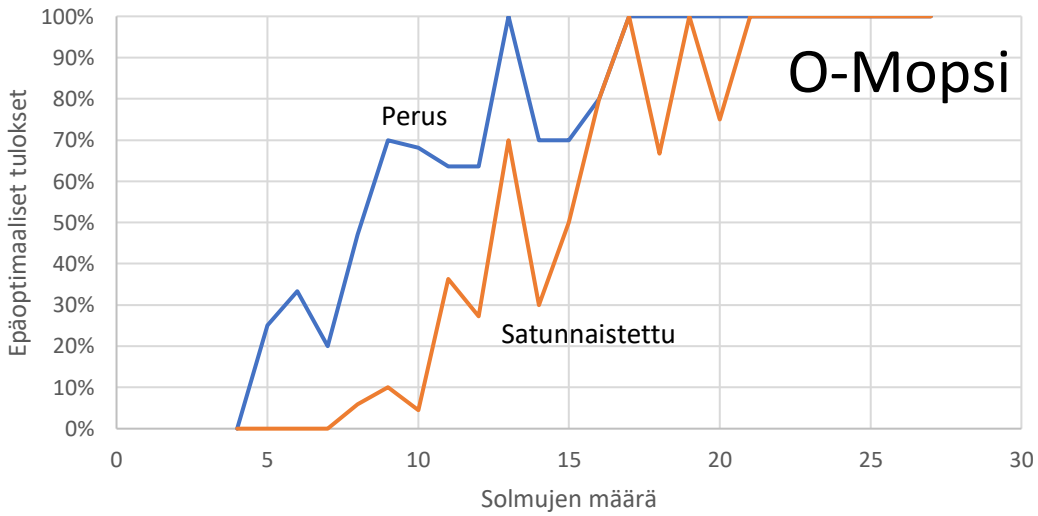
Satunnaistetussa versiossa satunnaistettu puunluonti toistettiin 100 kertaa jokaiselle O-Mopsi-datajoukon syötteelle ja näistä ratkaistiin kauppamatkustajan ongelma aiemmin esitellyn algoritmin avulla. Satunnaistetun version antamaksi tulokseksi saatiin 0,60 %. Toisin sanoen satunnaistuksella saadaan aikaan se, että keskimäärin tulos jää optimista alle yhden prosentin. Datajoukkoon kuuluneista 147 syötteestä optimitulos saatiin 101 syötteelle ja epäoptimi 46 syötteelle. Satunnaisuutta lisäämällä on saatu selkeä parannus verrattuna perusalgoritmin antamaan 50 optimitulokseen.



Kuva 27: Satunnaistetun algoritmin ero optimaaliseen tulokseen O-Mopsi-datajoukossa

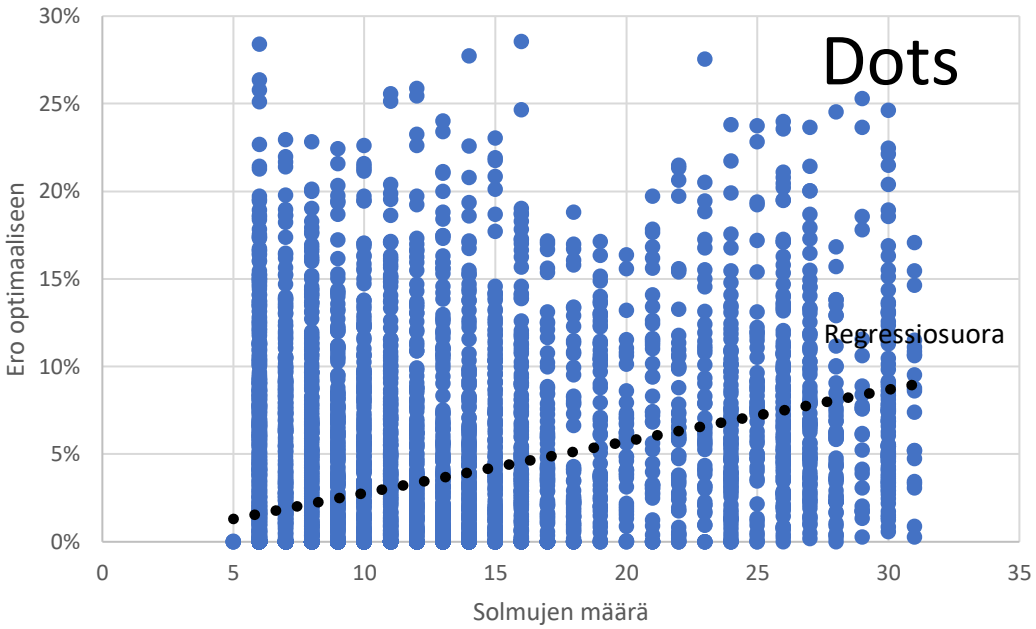
Lisäksi kuvasta 27 voidaan huomata, että satunnaistetun version huonoin tulos jää optimaalisesta alle 8 prosenttia. Tässäkin on siis tullut parannusta, koska perusversiossa ero optimaaliseen voi kasvaa pahimmassa tapauksessa n. 30 prosenttiin. Kuvista 26 ja 27 havaitaan myös, että ero optimiin on keskimäärin suurempi, kun solmujen määrä kasvaa.

Perusversion ja satunnaistetun version antamat epäoptimaaliset tuloskerrat esitetään seuraavassa kuvassa. Kuvasta 28 nähdään, kuinka monta prosenttia tuloksista on epäoptimaalisia kullakin solmumäärällä. O-Mopsi-datassa on vain vähän syötteitä useammissa solmumäärissä, joten yksittäisen tuloksen vaikutus on suurta. Esimerkiksi 17 solmun kohdalla on vain yksi syöte. Kuva kertoo vain epäoptimaalisten tulosten määrästä prosentteina, eikä siis kuinka paljon heikompia kyseiset tulokset ovat.



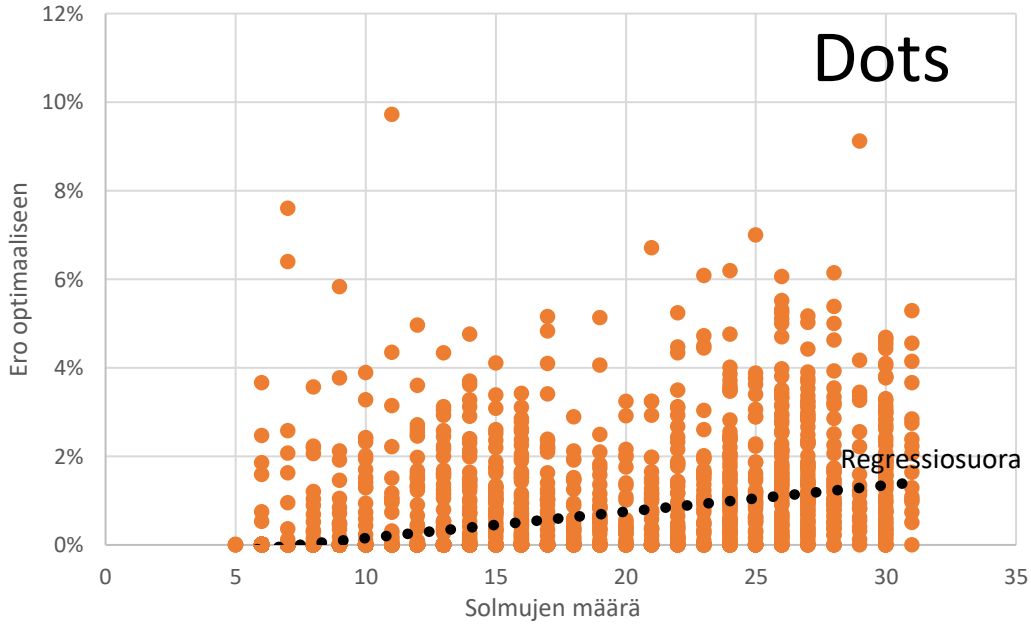
Kuva 28: Epäoptimaalisten tulosten jakautuminen O-Mopsi-datassa

Myös Dots-datajoukon tapauksessa perusversion heikoin tulos on melkein 30 % optimaalista huonompi. Dots-datajoukko sisältää huomattavasti enemmän eri syötteitä ja solmuja on maksimissaan 31 kappaletta yhdessä syötteessä. Kaikkien tulosten keskiarvoksi saatiin 3,05 %. Yhteensä 6447:stä pelistä optimitulos saatiin 3542 syötteelle. Ei-optimeita tuloksia oli 2905. Toisin sanoen optimitulos saatiin n. 55 % ajetuista testeistä.



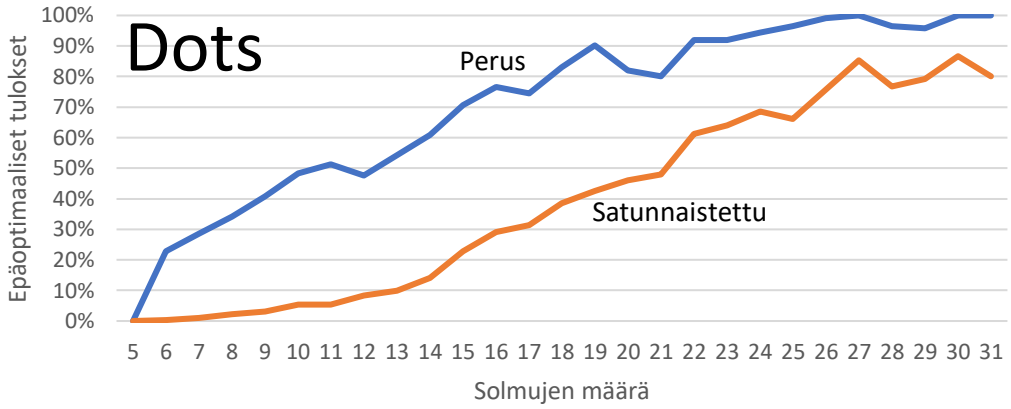
Kuva 29: Ero optimaaliseen Dots-datajoukossa

Dots-datajoukon tapauksessa satunnaistetulla algoritmilla heikoin tulos jää hieman alle 10 prosenttiin. Keskiarvotuloksen ero pienenee huomattavasti verrattuna perusversioon ollen 0,21 %. Lisäksi optimituloksia saatiin satunnaistuksella 5599 ja ei-optimeita 848. Siten optimitulos saatiin n. 87 % ajetuista Dots-joukon syötteistä.



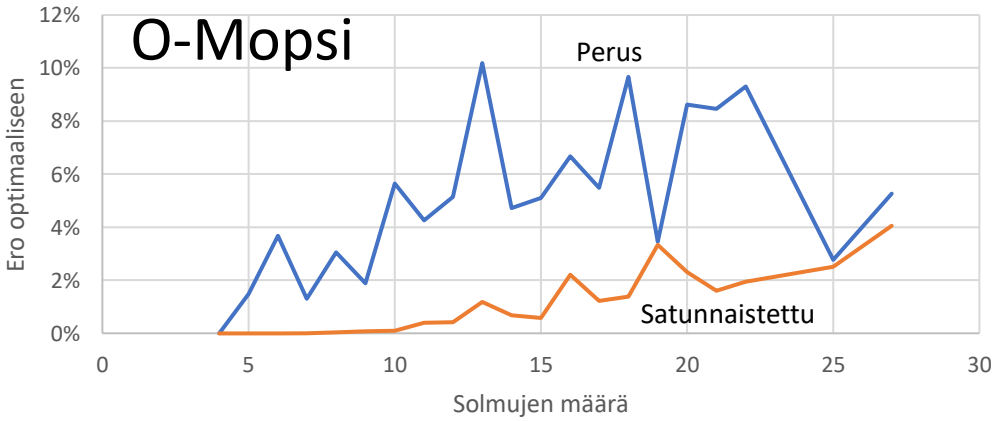
Kuva 30: Satunnaistettu ero optimaaliseen Dots-datajoukossa

Epäoptimaalisten tulosten määrä prosentteina kasvaa solmujen määrän kasvaessa Dots-datajoukossa. Satunnaistetulla algoritmilla on samanlaista kasvua solmumäärän mukaan, mutta epäoptimaalisten tulosten määrä on kuitenkin pienempi kuin perusalgoritmilla.



Kuva 31: Epäoptimaalisten tulosten jakautuminen Dots-datassa

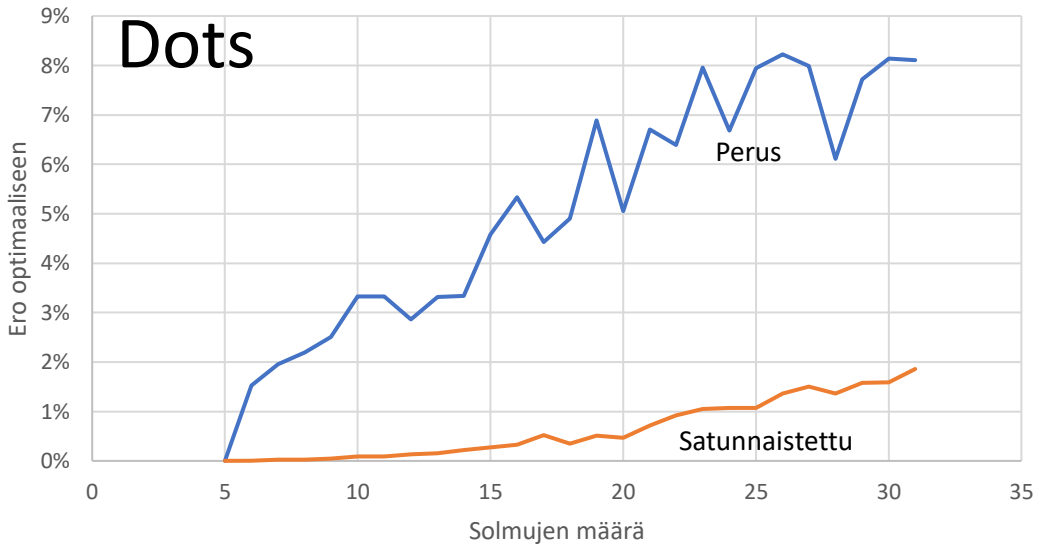
Kuva 32 esittää kuinka hyvän tuloksen perus- ja satunnainen algoritmi antavat keskimäärin jokaiselle O-Mopsi-datajoukon solmumäärälle. Oranssi viiva kuvaa satunnaisen algoritmin tuloksia ja sininen kuvaa perusalgoritmin tuloksia. Tämän kuvan perusteella voidaan päätellä, että solmumäärällä on vaikutusta tuloksiin. Esimerkiksi kun solmumäärä on 13, keskiarvo nousee perusversiolla yli kymmeneen prosenttiin. Vaihtelu on tosin melko suurta. Perusalgoritmin korrelaatiokertoimeksi solmujen määrän ja tuloksen hyvyyden välille saatiin n. 0,56, joten solmujen määrä korreloi saadun tuloksen kanssa. Satunnaistetulle algoritmille saatiin korrelaatiokertoimeksi n. 0,89 jolloin korrelaatio on jo vahva.



Kuva 32: Keskiarvoero O-Mopsi-datajoukolle

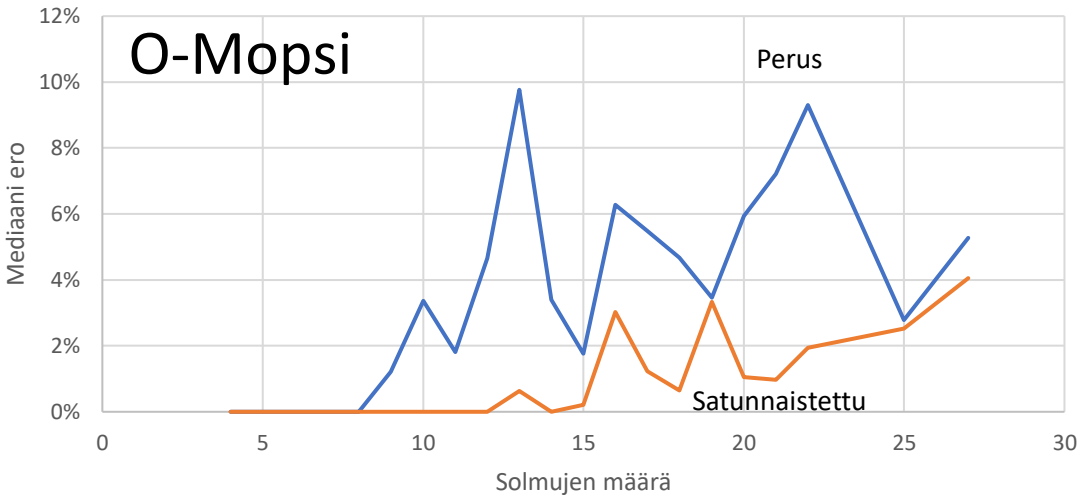
Kuvan 32 perusteella myöskin satunnaistetulla algoritmilla solmumäärällä näyttää olevan vaikutusta tuloksiin ja ero optimaaliseen vaikuttaa olevan kasvusuunnassa solmumäärän kasvaessa. Solmumäärän ollessa 27, keskiarvo on huipussaan n. 4 prosentissa. Keskiarvo jää kuitenkin jokaisessa muussa solmumäärässä alle neljään prosenttiin. Kuvasta voidaan huomata, että satunnaisuuden avulla saadaan kuitenkin saavutettua selvästi parempia tuloksia.

Dots-datajoukon tuloksista kuvassa 33 voidaan todeta, että satunnaisuudella saadaan parempia tuloksia. Keskiarvo jää alle kahteen prosenttiin kaikissa solmumäärissä, kun perusalgoritmilla huonoin keskiarvo nousee hieman yli kahdeksaan prosenttiin. Myös Dots-datajoukosta nähdään, että solmumäärän kasvaessa tulokset heikentyvät. Tosin satunnaisella algoritmilla päästään silti keskimäärin melko hyviin tuloksiin. Korrelaatiokertoimeksi perusalgoritmille saatiin n. 0,95 ja satunnaistetulle algoritmille n. 0,96. Molemmissa tapauksissa solmujen määrä korreloi hyvin vahvasti tuloksen hyvyyden kanssa.



Kuva 33: Keskiarvoero Dots-datajoukossa

O-Mopsi-datajoukon mediaanituloksista kuvassa 34 voidaan myös huomata, että perusalgoritmi ei anna aina optimaalisia tuloksia. Mediaanikuvaaja on samansuuntainen keskiarvokaavion kanssa. Myös mediaanikuvaajassa solmumäärän ollessa 13, ero nousee n. 10 prosenttiin. Kaikkien syötteiden mediaaniksi saatiin 3,02 %.



Kuva 34: Mediaaniero O-Mopsi-datajoukolle

Myös satunnaistetun version tapauksessa mediaanikuvaaja on samansuuntainen keskiarvokuvaajan kanssa. Myöskin mediaaniarvo jokaiselle solmumäärälle on korkeintaan neljä prosenttia. Lisäksi jos lasketaan kaikkien syötteiden mediaani, tuloksena on 0,00 % eli optimi tulos.

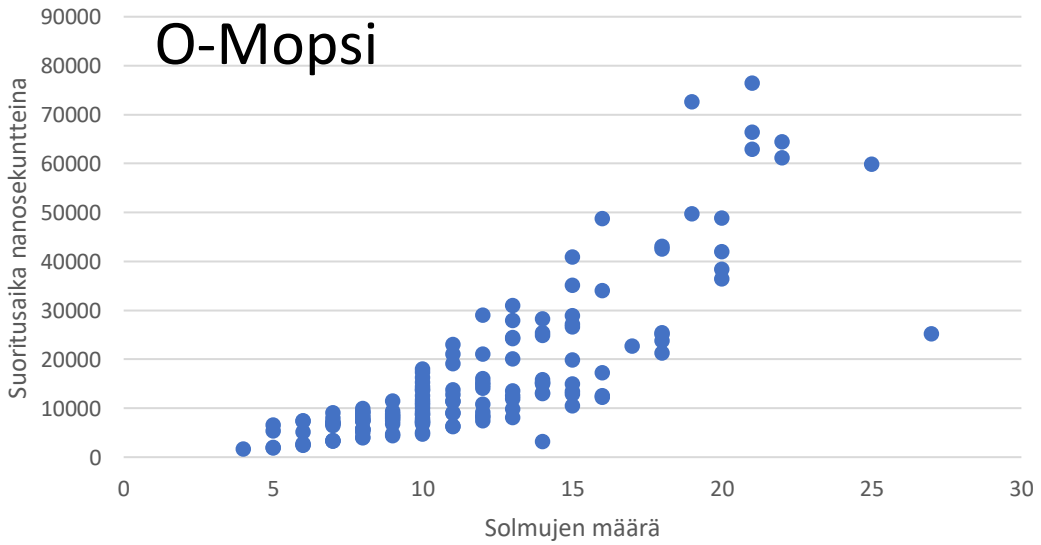
Taulukossa 1 on tulokset pelkästään O-Mopsi-datajoukon syötteille, joissa on 10 solmua kussakin. Tuloksista huomataan, että perusversion antamat tulokset vaihtelevat hyvin paljon myös samalla solmumäärällä. Välillä algoritmi antaa optimaalisia tuloksia ja välillä ero optimaaliseen voi olla jopa n. 19 prosenttia. Niinpä solmujen määrän lisäksi syötteen laatu vaikuttaa hyvin paljon tuloksiin. Satunnaistetulla algoritmilla vain Game_1624.txt antaa epäoptimaalisen tuloksen.

Taulukko 1: Kymmenen solmua sisältävät O-Mopsi-datajoukon syötteet

10 solmun syötteet	Perus	Satunnaistettu
Game_1624.txt	18,71 %	2,00 %
Game_1770.txt	18,63 %	0,00 %
Game_1382.txt	15,07 %	0,00 %
Game_1537.txt	12,57 %	0,00 %
Game_1818.txt	11,09 %	0,00 %
Game_1248.txt	9,58 %	0,00 %
Game_1810.txt	8,88 %	0,00 %
Game_1716.txt	6,76 %	0,00 %
Game_1808.txt	6,09 %	0,00 %
Game_1841.txt	5,88 %	0,00 %
Game_1219.txt	3,61 %	0,00 %
Game_1797.txt	3,14 %	0,00 %

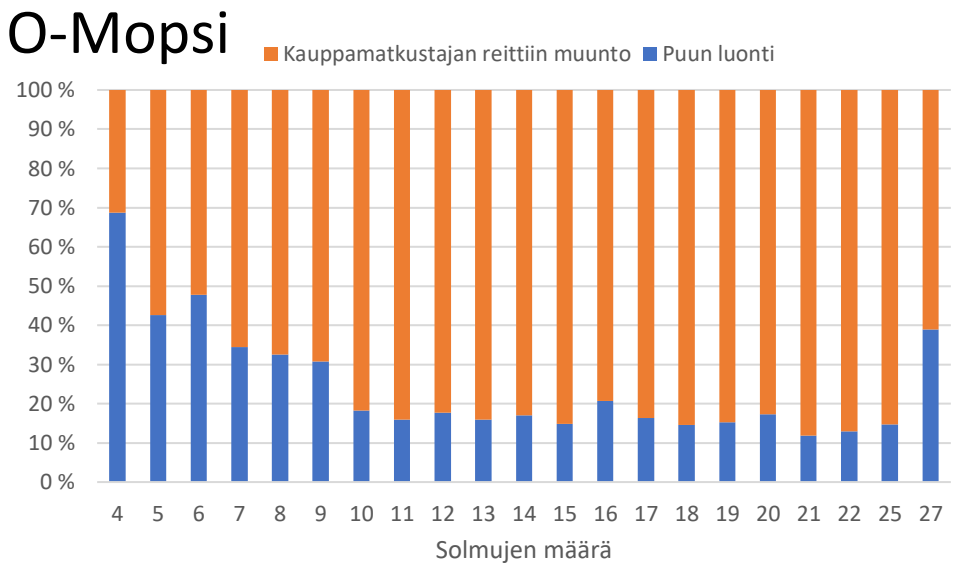
Game_1796.txt	1,95 %	0,00 %
Game_1126.txt	1,88 %	0,00 %
Game_1849.txt	0,20 %	0,00 %
Game_1431.txt	0,00 %	0,00 %
Game_1432.txt	0,00 %	0,00 %
Game_1758.txt	0,00 %	0,00 %
Game_1807.txt	0,00 %	0,00 %
Game_1385.txt	0,00 %	0,00 %
Game_1242.txt	0,00 %	0,00 %
Game_1850.txt	0,00 %	0,00 %

Algoritmin aikavaativuuden analysoinnin lisäksi suoritusaikaa testattiin käytännössä. Jokainen datajoukon syöte ajettiin 1000 kertaa ja mediaanitulos piirrettiin kuvaajaan. Testidataa ajettiin tietokoneella, jonka suorittimena toimi Intel i7 8700k ja käyttöjärjestelmänä Windows 10. Kuva 35 esittää kuinka suoritusaika kasvaa solmumäärän mukaan. Virittävän puun luontia ja puun muuttamista kauppamatkustajan reitiksi mitattiin nanosekunteina. Solmumäärän 25 ja 27 syötteitä oli testattavassa datassa vain yksi kappale kutakin, joten niiden antama tulos vaikuttaa poikkeavalta havainnolta. Muut pisteet näyttävät noudattavan samaa muotoa. Suoritusajan havaitaan nousevan nopeammin kuin lineaarisesti suhteessa solmujen määrään.



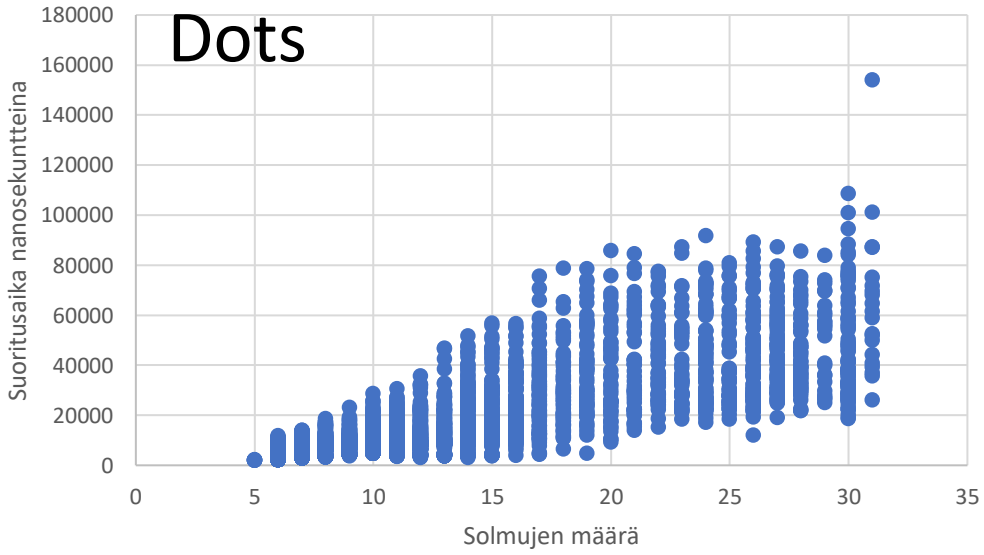
Kuva 35: Suoritusajan kasvu solmumäärän mukaan O-Mopsi-datalle

Suoritusajasta tutkittiin myös, kuinka paljon aikaa puun luonti vie suhteessa varsinaiseen kauppamatkustajan reittiin muokkaamiseen verrattuna. Kuten kuvasta 36 voi nähdä, puunluonti vie vähemmän aikaa verrattuna reitin luomiseen kaikissa muissa tapauksissa paitsi neljän solmun syötteessä. Useimmissa tapauksissa puunluontiin kuluu vain n. 20 % suoritusajasta. Lisäksi solmujen määrän kasvaessa puunluontiin menee keskimäärin vähemmän aikaa, paitsi 27 solmun tapauksessa.



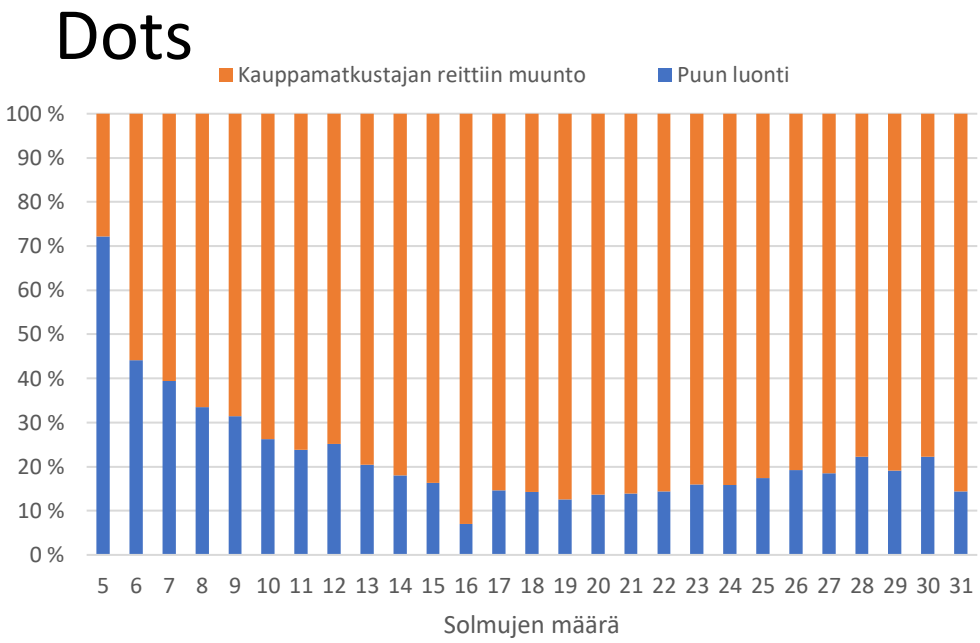
Kuva 36: Puun luonnin ja reitin luomisen suhde O-Mopsi-datalle

Suoritusaikaa testattiin myös Dots-datalle kuvassa 37. Tässäkin tapauksessa suoritusaika kasvaa solmumäärän mukaan.



Kuva 37: Suoritusajan kasvu solmumäärän mukaan Dots-datalle

Myös Dots-datasta nähdään kuvassa 38, että puunluonti vie vähemmän aikaa kuin varsinainen reitin luominen. Vain viiden solmun tapauksessa puun luontiin menee enemmän aikaa.



Kuva 38: Puun luonnin ja reitin luomisen suhde Dots-datalle

6.3 Vertailu muihin algoritmeihin

Taulukoissa 2 ja 3 on vertailtu algoritmin perusversion ja satunnaistetun version tuloksia pariin muuhun kauppamatkustajan ongelman ratkaisevaan algoritmiin. Vertailukohteena olivat Tabu Search (TS) [17] -algoritmi ja Ant Colonny Optimization (ACO) [18] -algoritmi. Vertailu tehtiin olemassa olevilla toteutuksilla molemmista algoritmeista [19][20].

Perusversio suoriutuu vertailluista algoritmeista heikoiten, mutta sekä O-Mopsi että Dots -datajoukkojen tapauksessa satunnaistetulla puunluonnilla varustettu versio toimii paremmin kuin TS-algoritmi. ACO-algoritmi suoriutuu silti parhaiten vertailluista algoritmeista. Sillä saavutetaan pienin keskimääräinen ero optimaaliseen sekä suurin määrä optimituloksia. Tästä huolimatta varsinkin satunnaistettu versio antaa hyviä tuloksia ja sitä voidaan käyttää kauppamatkustajan reitin laskemiseen.

Taulukko 2: Yhteenveto tuloksista O-Mopsi-datalle

O-Mopsi tulokset	Ero optimaaliseen keskimäärin	Optimitulosten määrä prosentteina
Perus	4,96 %	34,01 %
Satunnaistettu (100 toistoa)	0,60 %	68,71 %
ACO	0,14 %	89,12 %
TS	2,14 %	55,78 %

Taulukko 3: Yhteenveto tuloksista Dots-datalle

Dots tulokset	Ero optimaaliseen keskimäärin	Optimitulosten määrä prosentteina
Perus	3,05 %	54,94 %
Satunnaistettu (100 toistoa)	0,21 %	86,85 %
ACO	0,03 %	96,22 %
TS	0,96 %	72,78 %

7 Yhteenveto

Tutkielmassa esiteltiin tapa muuttaa pienin virittävä puu ratkaisuksi avoimen kierroksen kauppamatkustajan ongelmaan, sekä satunnaistettu versio samasta algoritmista, joka sisältää satunnaistetun puunluonnin

Tuloksista nähdään, että algoritmin avulla voidaan laskea kauppamatkustajan reittejä, mutta tulokset eivät ole aina optimaalisia. Perusversio ei anna aina lyhintä reittiä ja kuten O-Mopsi-datasta nähdään, tulos on keskimäärin hieman alle 5 % heikompi. Lisäksi optimaalisia tuloksia saatiin vain kolmannekselle annetuista syötteistä. Siten perusversio toimii heikonlaisesti, jos tavoitteena on löytää optimaalinen reitti. Dots-datalle perusversio antoi hieman parempia tuloksia reittien ollessa keskimäärin hieman yli 3 % pidempiä optimaalisesta ja optimaalisia tuloksia saatiin n. 55 % syötteistä.

Satunnaistetulla puunluonnilla parannettu versio taas toimii jo melko hyvin. O-Mopsi-datalle keskimääräinen tulos on 0,6 prosentin kasvu optimaaliseen reittiin verrattuna. Lisäksi optimaalisia tuloksia saatiin n. 69 % O-Mopsi-datan syötteistä. Dots-datalle saatiin vielä parempia tuloksia, keskimääräisen eron ollen vain 0,21 % ja optimaalisten tulosten määrän ollessa n. 87 %. Siten satunnaistetusta versiosta voidaan sanoa, että se toimii jo hyvin.

Tuloksista nähtiin myös, että solmujen määrä vaikuttaa tuloksiin. Keskimäärin mitä enemmän solmuja on syötteessä, sitä heikompi tulos. Lisäksi opittiin, että puunluonti vie yleisesti vähemmän aikaa, ja varsinainen reitiksi muunto vie suurimman osan suoritusajasta.

Algoritmia voidaan käyttää, jos halutaan laskea polynomiallisessa ajassa ratkaisua kauppamatkustajan ongelmaan.

8 Viitteet

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

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

[3] Applegate, David L., et al. "The traveling salesman problem: a computational study." *Princeton university press*, 2006.

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

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

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

[7] 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.

[8] Zhong, Caiming., Malinen, Mikko., Miao, Duoqian., Fränti, Pasi. "A fast minimum spanning tree algorithm based on K-means." *Information Sciences* 295 (2015): 1-17.

[9] 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.

[10] Chieng, Hock Hung., Wahid, Noorhaniza. "A performance comparison of genetic algorithm’s mutation operators in n-cities open loop travelling salesman problem." *Recent Advances on Soft Computing and Data Mining.* Springer, Cham, 2014. 89-97.

[11] 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.

[12] Rego, César., Gamboa, Dorabela., Glover, Fred., Osterman, Colin. "Traveling salesman problem heuristics: Leading methods, implementations and latest advances." *European Journal of Operational Research* 211.3 (2011): 427-441.

[13] Lin, Shen., Kernighan, Brian W. "An effective heuristic algorithm for the traveling-salesman problem." *Operations research* 21.2 (1973): 498-516.

[14] Helsgaun, Keld. "General k-opt submoves for the Lin–Kernighan TSP heuristic." *Mathematical Programming Computation* 1.2-3 (2009): 119-163.

[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] An, Hyung-Chan., Kleinberg, Robert., Shmoys, David B. "Improving christofides' algorithm for the st path TSP." *Journal of the ACM* (JACM) 62.5 (2015): 34.

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

[18] 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.

[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)