



Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

NIKO MAST

TYYPPISET ONGELMATAPAUKSET; KAUPPAMATKUSTAJAN KIERROS

TIETOJENKÄSITTELY

2021

Tekijä(t) Mast, Niko	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Toukokuu 2021
	Sivumäärä 26	Julkaisun kieli Suomi
Julkaisun nimi		
Tyypilliset ongelmatapaukset; Kauppatatkustajan kierros		
Tutkinto-ohjelma Tietojenkäsittelyn koulutusohjelma		
<p>Työn tarkoituksena oli ohjelmointi taitojeni kehittäminen niin, että pyrin ymmärtämään minkä takia tietokoneilla ei voi ratkaista kaikkia ongelmia tehokkaasti, tarkoituksena oli myös oppia intuitiivisesti tunnistamaan sellaiset ongelmat, joita ei ole helppo ratkaista ja se, miten tällaisten ongelmien ratkaisua kannattaa lähestyä. Tutustuin työssä laskennalliseen kompleksisuuteen sekä kombinatoriseen optimointiin. Tutustuminen tapahtui esimerkkitapauksen tutkimisen kautta. Ongelma jota tutkin tunnetaan kauppatatkustajan ongelmana.</p> <p>Työssä käsittelen ongelman historiaa, sekä siihen liittyvää termistöä ja ratkaisutapoja joiden avulla ongelmaa voidaan lähteä ratkaisemaan. Työ sisältää myös koodiesimerkkejä tavoista joilla mm. kauppatatkustajan ongelmaa voidaan lähteä ratkaisemaan.</p> <p>Viimeinen kappale sisältää mielteitä joita minulle jäi tästä työstä.</p>		
Asiasanat haetaan asiasanaluettelosta, mutta siihen ei tehdä linkitystä Kauppatatkustajan ongelma, c++, heuristiikat		

Author(s) Mast, Niko	Type of Publication Bachelor's thesis	Date May, 2021
	Number of pages 26	Language of publication: Finnish
Title of publication Typical problem cases; Travelling salesman problem		
Degree programme Business information systems		
<p>The purpose of this thesis was to increase my programming skills, by understanding why some problems cannot be solved efficiently. I also wanted to learn how to recognize a problem that can't be solved efficiently and to know how solving these kinds of problems should be approached. In this thesis I explored computational complexity and combinatorial optimization. The exploration was done through a problem called the travelling salesman problem which is a good example of combinatorial complexity and solving the problem requires combinatorial optimization. I tried to understand the problem by dividing the problem into its parts and analyzing the parts.</p> <p>This thesis includes short explanation of the problem and its history. I also covered some more technical aspects related to these kinds of problems which include example codes and explanations of algorithms.</p> <p>In the last chapter I go through some ideas about the subject matter of this thesis.</p>		
<p><u>Key words</u> search from key word list but not link Travelling salesman problem, c++, heuristics</p>		

SISÄLLYS

1 JOHDANTO.....	5
2 KAUPPAMATKUSTAJAN ONGELMA.....	6
2.1 Historia.....	6
2.2 Ongelman abstraktion historia.....	7
2.2.1 Hamilton.....	8
2.2.2 Eulerin ja Hamiltonin jälkeen.....	8
2.2.3 Nykyaika.....	9
3 ONGELMAN VAIKEUS JA KOVUUS.....	9
4 TUOTANTOTALOUDESSA.....	11
4.1 NVIDIA.....	11
4.2 BÖWE CARDTEC.....	12
5 KIERROKSEN ETSINTÄ.....	12
5.1 Approksimointi/Heuristiset algoritmit.....	13
5.1.1 Nearest Neighbor.....	13
5.1.2 Cristofidesin algoritmi.....	15
5.1.3 K-opt optimointi.....	19
5.1.4 Lin-Kernighan.....	22
5.1.5 Lin-Kernighan-Helsgaun.....	23
5.2 Lyhyimmän kierroksen etsintä.....	24
5.2.1 Lineaarinen ohjelmointi.....	24
6 LOPUKSI.....	25

LÄHTEET

LIITTEET

1 JOHDANTO

Tässä työssä tutustuin mielenkiintoiseen ongelmaan, jonka melkein pä kuka tahansa ymmärtää mutta sen ratkaiseminen lähestyy mahdotonta eksponentiaalisesti suhteessa ongelmassa olevien muuttujien lukumäärään. Valitsin työn aiheen kuultuani siitä ohjelmointikurssilla, koska se vaikutti mielenkiintoiselta. Kiinnostuin erityisesti ongelman universaalisuudesta ja siihen liittyvästä termistöstä. Tutustuttuani ongelmaan huomasin, että sen vaikeus johtuu luonnonlakien alaisuudessa toimivasta laskettavuudesta, joka antaa huvittavan kontekstin sille kuinka yksinkertaisilta vaikuttavat asiat saattavat olla ovat mahdottoman vaikeita, vaikka niiden ratkaisemiseen käytettäisiin ihmiskunnan tehokkaimpia laskentakoneita.

Työtä kirjoittaessa kohtasin tieteellisiä julkaisuja, jotka olivat minulle ennennäkemättömän monimutkaisia yksityiskohtaisuutensa ja matemaattisten notaatioidensa vuoksi. Luulen, että onnistuin ymmärtämään ongelmaa ja siihen liittyviä tieteellisiä julkaisuja sen verran, että tämän työn avulla voin antaa lukijalle jonkinlaisen kuvan ongelmasta, jota siinä käsitellen. Tähän kuvaan, jonka lukijalle toivon tästä työstä jäävän sisältyy ainakin ohjeet, minne kannattaa katsoa, jos ongelma kiinnostaa.

Työn ensimmäisessä kappaleessa määritellään ongelma, jonka jälkeen käyn läpi ongelman historiaa. Historiaa käsittelevä kappale esittelee käytännön tilanteita, joissa ongelma on esiintynyt, sekä sitä miten ja milloin ongelman abstraktion tutkimus on aloitettu. Esittelen työssä myös laskettavuuteen liittyvää termistöä, jonka tarkoituksena on selventää, sitä minkä vuoksi kyseinen ongelma on niin vaikea ja tärkeä. Käyn läpi käytännön esimerkkejä tuotantotaloudesta, jossa ongelma aiheuttaa vaikeuksia. Edellisten asiakokonaisuuksien jälkeen esittelen lähestymistapoja, joita tämän kyseisen ongelman ratkaisussa käytetään, ja tämä kappale sisältää myös C++ esimerkkejä. Viimeisessä kappaleessa käyn läpi joitakin ajatuksia mitä työssä kohtaamani asiat herättivät.

2 KAUPPAMATKUSTAJAN ONGELMA

Ongelman kuvaus sellaisena kuin minä olen sen ymmärtänyt: Kauppatkustajan ongelmassa valitaan jokin määrä kaupunkia kartalta. Jokaisesta kaupungista pääsee jokaiseen muuhun kaupunkiin. Jokaisesta kaupungista jokaiseen kaupunkiin on tietysti myös tietty matka. Valitut kaupungit on tarkoitus kiertää niin, että kierros päätetään siihen kaupunkiin, josta se aloitettiin. Kierroksen pituuden tulee olla mahdollisimman pieni ja jokaisessa kaupungissa saa käydä vain kerran.

2.1 Historia

Kauppamatkustajan ongelmaa vastaavasta käytännön ongelmasta löytyy merkintöjä 1500-luvulta ja matemaattisempi lähestyminen ongelmaan aloitettiin 1700-luvulla. (Cook, 2012, 45.)

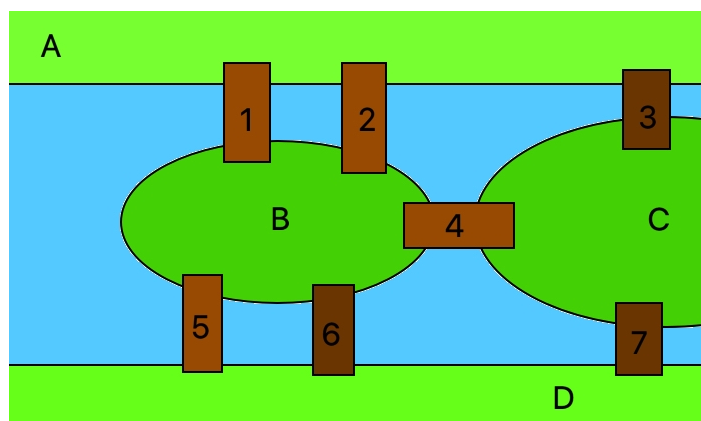
1500-luvulla lakimiehet ja tuomarit kiersivät yhdistyneissä kansakunnissa piirikuntia, joihin sisältyi kaupunkeja ja kyliä. Tämä piirikuntien kiertäminen siirtyi myöhemmin myös Yhdysvaltoihin, jossa niihin viitattiin nimenomaan kierroksina. Sanalla piirikunta ja kierros on tärkeä merkitys, koska se viittaa siihen, että matka aloitetaan tietystä kaupungista, matkalla käydään tietyissä kaupungeissa ja kylissä, jonka jälkeen palataan sinne mistä matka aloitettiin, joka vastaa kauppamatkustajan ongelman määrittelyä. (Cook, 2012, 53–54.)

1700-luvulla Kristinuskon saarnaajat levittivät sanomaansa piirikunnissa ja voidaan olettaa, että matkojen suunnittelu kustannusten minimoimiseksi oli vähintään ohi menevä ajatus, joten käytännöllisessä muodossa ongelmasta löytyy merkintöjä jo yli kahdelta vuosisadalta. (Cook, 2012, 55.)

Kauppamatkustajat olivat 1800 ja 1900-luvulla erityisesti Yhdysvalloissa yleinen näky. Joidenkin arvioiden mukaan vuonna 1883 Yhdysvalloissa toimi noin 200,000 kauppamatkustajaa. Kauppamatkustajien määrä nousi 1900-luvulle tultaessa. Kauppamatkustajien avuksi kirjoitettiin kirjoja, jotka sisälsivät valmiita reittejä. Kirjoista he pystyivät valitsemaan reittinsä tehokkaasti. Iso osa kauppamatkustajien matkoista suunniteltiin keskitetysti, joten kauppamatkustaja ei välttämättä saanut henkilökohtaisesti, valita reittiään. Reittejä sisältäviä kirjoja löytyi myös eurooppalaisille kauppamatkustajille. Saksan ja Sveitsin alueille on olemassa Kauppamatkustajan ongelman määrittelyn täyttävä kaupunkien kierros, joka on vuodelta 1832. (Cook, 2012, 46–53.)

2.2 Ongelman abstraktion historia

1700-luvulla, matemaatikko nimeltä Leonhard Euler kirjoitti matemaattisia artikkeleita liittyen kiertomatkoihin. Euler asui kaupungissa nimeltä Königsberg. Samassa kaupungissa oli mielenkiintoinen siltoja ja maamassoja sisältävä verkosto. Vanhan Königsbergiläisen legendan mukaan kaupungin asukkailla oli tapana miettiä vastausta kysymykseen, onko olemassa reittiä, joka johtaa kaupungin läpi niin, että kyseistä reittiä kävelemällä ylittää kaupungin jokaisen sillan tasan kerran? Kaupunkilaisten sanotaan yrittäneen vastata tähän kysymykseen piirtelemällä mahdollisia reittejä paperille, kunnes vuonna 1735 Euler esitti tähän kysymykseen pelkistetyn matematiikkaan nojaavan vastauksen. (Carlson, 2020.)



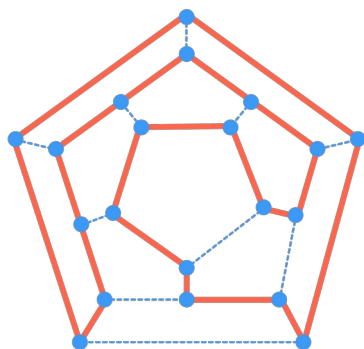
Kuva 1. Königsbergin sillat.

Vaatimaton versio Eulerin vastauksesta voidaan esittää seuraavalla tavalla: Kuvassa 1 maa massalle B johtaa viisi siltaa ja muihin maamassoihin johtaa kolme siltaa. Maamassalle siirtymiseen tarvitaan yksi silta ja sieltä poistumiseen tarvitaan yksi silta. Niiltä maamassoilta, josta reitti aloitetaan tai johon se lopetetaan ei tarvitse johtaa kuin yksi silta, mutta maamassoille, joissa käydään pitää johtaa kaksinkertainen määrä siltoja suhteessa käyntien määrään, jotta siellä käyminen on mahdollista. Königsbergin tilanteessa niin ei ole, joten sellainen reitti, jossa jokaista siltaa käytetään mutta vain kerran ei ole mahdollinen. (Carlson, 2020.)

Tämän vastauksen virallinen todistus toimii pohjana verkkoteorialle ja topologialle, jotka ovat matematiikan haaroja. Verkkoteoriassa verkko on sellainen olio, jossa on pisteitä ja pisteet ovat yhdistettyinä toisiinsa kaarilla. Königsbergin siltojen tapauksessa maa massat ovat pisteitä ja sillat ovat kaaria. Eulerin mukaan on nimetty verkkoteoriassa sellainen polku, jossa käytetään verkon jokaista kaarta tasan kerran, Eulerin poluksi. Sellainen kierros, jossa käytetään jokaista kaarta vain kerran ja se päätetään samaan pisteeseen, josta se aloitettiin on nimetty Eulerin kierrokseksi. (Carlson, 2020.)

2.2.1 Hamilton

Satavuotta Eulerin jälkeen Irlantilainen Sir William Rowan Hamilton tutkiskeli mahdollisia tapoja kiertää Dodekaedrin jokainen kulma. Dodekaedrissa on 20 kulmaa.



Kuva 2. Hamiltonin kierros (Brilliant.org, 2020, Hamiltonian circuit).

Kierros pitää lopettaa samaan kulmaan, josta se aloitettiin. Kuvio on rakenteeltaan verkko, kuvion kulmat ovat verkon pisteitä ja kuvion kulmia yhdistävät suorat ovat verkon kaaria. (Brilliant.org, 2020, Hamiltonian circuit; Cook, 2012, 62–63.)

Hamilton kehitti tätä ongelmaa varten algebrallisen menetelmän. Menetelmänsä avulla hän todisti, että aloitettiinpa kierros mistä tahansa kuvion uloimmasta kulmasta on mahdollista kiertää kuvion kulmat käyttämättä yhdessäkään niistä yli yhtä kertaa, kierroksen voi myös lopettaa samaan kulmaan, josta se aloitettiin. Hamiltonin tulosten ansiosta verkkoteoriassa sellainen polku, joka kiertää verkon jokaisen pisteen tasan kerran on nimetty Hamiltonin poluksi. Sellainen kierros, joka kiertää jokaisen pisteen tasan kerran ja päättyy siihen pisteeseen mistä se aloitettiin, on nimetty Hamiltonin kierrokseksi. Kauppatarkkailijan ongelmassa etsitään Hamiltonin kierrosta kaupunkien verkostosta. (Brilliant.org, 2020. Hamiltonian Path; William J. Cook, 2012, 64.)

2.2.2 Eulerin ja Hamiltonin jälkeen

1920 matemaatikko ja ekonomisti Karl Menger julkaisi kollegoidensa kanssa ”Postimiehen ongelmaa” käsittelevän artikkelin. Postimiehen ongelman voidaan sanoa olevan samanlainen kuin kauppatarkkailijan ongelman sillä eroavaisuudella, että siinä etsitään tehokkainta tapaa kiertää postilaatikoita, kaupunkien kiertämisen sijaan. (Cook, 2012, 70–71.)

1940 luvulla intialaiset tilastotieteilijät miettivät miten he voisivat kerätä satoihin liittyvää dataa maatiloilta. Aluksi ajatuksena oli, että valittaisiin satunnaisesti maatiloja, joilta data sitten haettaisiin. Budjetin kannalta paremmaksi suunnitelmaksi osoittautui maan jakaminen geografisesti toisiaan muistuttaviin osiin. Näistä maan osista valittiin satunnainen määrä maatiloja. Maan osiin jakamisen ja satunnaisten maatilojen valitsemisen jälkeen piti suunnitella kustannustehokkaat reitit. Siitä miten reitit suunniteltiin ei ole tietoa vaan tutkimukset keskittyivät lyhyimpien kierrosten keskiarvoihin tietyllä määrällä kaupunkeja. Keskimääräisen kierroksen selvittäminen antoi viitteitä kaavasta. Kaavan tutkimuksista on saatu selville, että se vahvistuu sitä mukaan, kun kaupunkien määrä kasvaa mutta tarkkaa arvoa kaavan tuotavalle vakiolle ei osata määrittää. (Cook, 2012, 76–82.)

2.2.3 Nykyaika

Informaatio aikakaudella ongelmasta on muodostunut suosittu ja tunnettu, siitä on tehty elokuvia ja sen ratkaisemisesta on luvattu ”kuuluksia” palkkio, jonka suuruus on miljoona dollaria ja omasta mielestäni ongelman ratkaiseminen kyseisen palkkion vuoksi olisi maailman vaikein tapa hankkia miljoona dollaria olettaen, että ongelman ratkaisija vaihtaisi ratkaisunsa vain miljoonaan dollariin. Ongelmaan ei siis ole täydellistä ratkaisua mutta siihen on joitakin ratkaisukeinoja.

Lineaarinen ohjelmointi tuottaa erittäin hyviä tuloksia Kauppamatkustaja ongelmassa. Concorde algoritmi, jossa käytetään lineaarista ohjelmointia, on saatavissa internetistä ja sen avulla on ratkaistu Kauppamatkustaja ongelma, jossa $n = 85,900$. Toinen tärkeä lineaarisen ohjelmoinnin työkalu on Simplex algoritmi. (Cook, 2012, 31, 134, 138.)

Heuristiset algoritmit, eli valmista kierrosta parantelevat algoritmit tai mahdollisimman hyvän kierroksen nopeasti tuottavat algoritmit ovat myös hyviä tuloksiensa puolesta. Heuristisista algoritmeista kerron työssä lisää.

3 ONGELMAN VAIKEUS JA KOVUUS

Tietokoneilla ongelmia ratkaistaan kehittämällä ongelmaa ratkaiseva algoritmi. Intuitiivisesti algoritmi tarkoittaa ohjeita, joita seuraamalla ongelma ratkaistaan. Algoritmien luokitteluun käytetään iso O notaatiota. Iso O notaatio kertoo, kuinka paljon aikaa algoritmin suorittamiseen kuluu suhteessa algoritmin syötteen suuruuteen. Algoritmin kuluttaman ajan suhde algoritmin syötteeseen mittaa algoritmin aikakompleksisuutta. $O(1)$, tarkoittaa, että syötteen pituus ei vaikuta suoritusaikaan. $O(n^2)$ tarkoittaa, että algoritmin suoritusaika on syötteen neliö. $O(a^n)$ tarkoittaa, että algoritmin suoritusaika muuttuu eksponentiaalisesti ja eksponenttina on syötteen koko. $O(n!)$ tarkoittaa, että algoritmin suoritusaika on syötteen kertoma. (Brilliant.org, 2020. Big O Notation; Cook, 2012, 220.)

Kauppamatkustajan ongelmassa etsitään kaupunkien joukosta lyhintä mahdollista kierrosta, johon kuuluu jokainen kaupunkien joukon kaupunki. Mahdollisimman lyhyen kierroksen löytämiseksi pitäisi tarkistaa jokainen mahdollinen kierros. Näin on ainakin vielä. Oletetaan etäisyysmatriisi, joka koostuu kaupungeista ja niiden välisistä etäisyyksistä.

Kaupunki	Turku	Helsinki	Tampere	Vaasa	Kuopio	Joensuu	Oulu
Turku	0	166	163	330	454	573	673
Helsinki	166	0	178	422	391	437	606
Tampere	163	178	0	242	294	394	487
Vaasa	330	422	242	0	377	493	318
Kuopio	454	391	294	377	0	137	287
Joensuu	573	437	394	493	137	0	393
Oulu	673	606	487	318	287	393	0

Tarkoituksena olisi etsiä etäisyysmatriisista lyhin mahdollinen kierros. Ensin pitää etsiä kaikki mahdolliset matkat ensimmäisestä kaupungista.

Edellisen jälkeen pitäisi etsiä toisesta kaupungista kaikki mahdolliset matkat, sellaisiin kaupunkeihin, joissa ei olla vielä käyty. Tätä jatkettaisiin toiseksi viimeiseen kaupunkiin asti. (William J. Cook, 2012, 18.)

Kaikkien mahdollisten reittien etsiminen voidaan ilmaista kaavalla $(n-1)!$, jossa n on kaupunkien lukumäärä.

Usein ongelman määrittelyssä käytetään matkojen symmetrisyyttä ja kolmikulmaisuuuden sääntöä. Matkojen symmetrisyys tarkoittaa, että matka $A \rightarrow B$ on yhtä suuri kuin matka $B \rightarrow A$. Kolmikulmaisuuuden sääntö tarkoittaa, että matka $A \rightarrow B \rightarrow C$ on suurempi tai yhtä suuri kuin matka $A \rightarrow C$.

Matkojen symmetrisyys muuttaa esimerkin kaavan $(n-1)!$ muotoon $(n-1)!/2$. Esimerkiksi kierros $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ on yhtä suuri kuin kierros $A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow A$. Vaikka matkat ovat symmetrisiä tilanteessa, jossa n on 33 tarkistettavaa kierroksia on 131,565,418,466,846,765,083,609,006,080,000 kappaletta. Jos oletetaan, että tietokoneen pitäisi tehdä reitin tarkistamista varten yksi aritmeettinen operaatio. Samainen tietokone pystyy myös tekemään 1,470 triljoonaa operaatiota sekunnissa. Menisi tietokoneella silti 28 triljoonaa vuotta lyhyimmän reitin löytämiseen. Auringon arvioitu jäljellä oleva elinikä on noin viisi miljardia vuotta. Triljoona on miljardi miljardia. (ursa.fi, 2020; Cook, 2012, 21.)

Tietokoneilla ratkaistavia ongelmia luokitellaan. Luokittelu tapahtuu sen perusteella, kuinka nopeita algoritmeja kyseisen ongelman ratkaisemista varten on kehitetty. Mielestäni kaksi tärkeintä luokkaa ovat P ja NP. Luokka P tarkoittaa, että ongelmaan on olemassa algoritmi, joka ratkaisee sen polynomisessa ajassa. Luokassa P olevan ongelman ratkaisevan algoritmin aikakompleksisuus on siis $O(n^p)$, jossa p on vakio. NP luokka tarkoittaa, että ongelma voidaan ratkaista polynomisessa ajassa tietokoneilla, jotka toimivat rinnakkain mutta eivät kommunikoi keskenään. NP luokassa olevan ongelman vastaus voidaan myös tarkistaa polynomisessa ajassa. NP-kova ongelma on sellainen, että se voidaan muuttaa miksi tahansa muuksi NP luokan ongelmaksi. NP-kovan ongelman ratkaisevaa algoritmia voidaan käyttää kaikkiin NP luokan ongelmiin. Kauppatmatkustajan ongelma on luokassa NP ja se on myös NP-kova ongelma. (Brilliant.org, 2020, Complexity classes).

4 TUOTANTOTALOUDESSA

Kauppatmatkustajan ongelman elinvoimaisuus perustuu siihen, että sen ydin esiintyy useassa paikassa ja sen vuoksi sitä ei voida sivuttaa pelkkänä abstraktina palapelinä. (Cook, 2012, 83.)

Paikkoja, joissa kauppatmatkustajan ongelma esiintyy ovat mm, koulukuljetusten järjestäminen, navigointi ohjelmistoissa tapahtuva reittien suunnittelu, genomien kartoitus, planeettojen etsintä, tuotantotalouden optimointi ja piirilevyjen suunnittelu. (Cook, 2012, 70–92.)

4.1 NVIDIA

NVIDIA on laskentateknologiaa valmistava yritys. NVIDIA käyttää Concorde algoritmia näytönohjaimien testauksessa käyttävien piirisirujen optimoinnissa. Concorde algoritmi on lineaarista ohjelmointia hyödyntävä ohjelma, jota käytetään optimointi ongelmiin, joihin sisältyy symmetrinen kauppamatkustajan ongelma. Laitteiston valmistuksen jälkeinen testaus on kriittinen vaihe laitteiston tuotanto prosessissa ja siksi tarkka lähestymistapa on yleinen modernien laitteistojen suunnittelussa. Piirisirut on asennettu NVIDIAN tapauksessa näytönohjaimiin. Piirisuruissa on syöttö ja syöte portit. Syöte porttiin syötetään dataa ja se, miten laitteisto käsittelee dataa, saadaan selville piirisirun syöteportista tulevasta syötteestä. Concorde avulla määritetään mihin syöte portteja tulee asentaa, jotta testaus vaihe olisi kattavaa ja samaan aikaan mahdollisimman lyhyt. (Cook, 2012, 88–89.)

4.2 BÖWE CARDTEC

Saksalainen yritys BÖWE CARDTEC toimittaa laitteistoa ja ohjelmistoja mm. Luottokorttien ja tunnistuskorttien valmistusta varten. Yhdestä laitteistosta ja ohjelmistosta voidaan valmistaa monia erilaisia kortteja, joten laitteiston tai ohjelmiston konfigurointiin kuluva aika on suuri. BÖWE CARDTEC käsittelee konfigurointi prosessia kauppamatkustajan ongelmana. Konfigurointi työt ovat tässä tapauksessa kaupunkia, ja työhön kuluva aika vastaa kahden kaupungin välistä matkaa. Concordea käyttäen BÖWE CARDTEC on raportoinut vähentäneensä tyypillisissä konfigurointi tehtävissä kuluva aika 65 %. (Cook, 2012, 90.)

5 KIERROKSEN ETSINTÄ

Kauppamatkustajan ongelmaa voidaan lähteä ratkaisemaan ainakin kahdella tavalla, joista ensimmäinen on sellainen, että yritetään saada mahdollisimman hyvä ratkaisu järkevässä ajassa. Toinen tapa on lähteä suoraa etsimään parasta mahdollista ratkaisua. Kummatkin näistä ratkaisutavoista ovat vaativia ja molemmilla on omat hyvät sekä huonot puolensa. Esittelen tässä kappaleessa ratkaisu metodeja kauppamatkustajan ongelmaan kummastakin edellä mainitusta kategoriasta. Kappale sisältää myös koodiesimerkkejä, jotka ovat kirjoitettu C++ kielellä.

Esimerkkiohjelma sisältää luokan Kartta, johon kuuluu työssä jo aikaisemminkin esitelty etäisyysmatriisi. Luokassa on myös merkkijonoista koostuva vektori, jossa on kaupunkien nimet etäisyysmatriisissa olevien kaupunkien nimet.

```

class kartta{
public:
    vector<int>
        Turku{0,166,163,330,454,573,673},
        Helsinki{166,0,178,422,391,437,606},
        Tampere{163,178,0,242,294,394,487},
        Vaasa{330,422,242,0,377,493,318},
        Kuopio{454,391,294,377,0,137,287},
        Joensuu{573,437,394,493,137,0,393},
        Oulu{673,606,487,318,287,393,0};

    vector<vector<int>>n {Turku, Helsinki, Tampere, Vaasa, Kuopio, Joensuu, Oulu};

    vector<string> kaupungit{"Turku","Helsinki","Tampere","Vaasa","Kuopio","Joensuu","Oulu"};

```

Kuva 3. Kartta luokka.

Kartta luokkaan sisältyvät metodit ovat, PoistaKaupunki, joka poistaa tietyn kaupungin vektorista, metodiin pitää syöttää poistettavan kaupungin järjestysnumero.

```

void PoistaKaupunki(int i){
    kaupungit.erase(kaupungit.begin()+i);
    for(int j = 0; j < n.size(); j++){
        n[j].erase(n[j].begin()+i);
    }
    n.erase(n.begin()+i);
}

```

Kuva 4. PoistaKaupunki metodi.

PoistaKohde metodilla voidaan poistaa tietty matkustuskohde kaikista muista kaupungeista. Tämän metodin avulla voidaan pienentää etsintäavaruutta sitä mukaan, kun tiettyyn kaupunkiin ei enää tarvitse matkustaa. Tämäkin metodi ottaa syötteen kaupunkin järjestysnumeron.

Luokassa on myös kaksi tulostus metodia, joiden avulla voidaan tulostaa kaikki kaupungit tai tietystä kaupungista jäljellä olevat matkustus kohteet. Tämän metodin tarkoitus on selvittää ohjelman toimintaa.

```

void TulostaKaupungit(){
    if(kaupungit.size() == n.size()){
        cout << kaupungit.size() << "\n";
    }
    for(string s : kaupungit){
        cout << s << "\n";
    }
    cout << "\n";
}

```

Kuva 5. TulostaKaupungit metodi.

```

void TulostaKaikkiKohteet(){
    for(int i = 0; i < n.size(); i++){
        cout << "Kaupungista " << kaupungit[i] << " voidaan matkustaa kaupunkeihin: " << "\n";
        for(int j = 0; j < n[i].size(); j++){
            if(n[i][j] != 0){
                cout << kaupungit[j] << "\n";
            }
        }
        cout << "\n";
    }
}

```

Kuva 6. TulostaKaikkiKohteet metodi.

Ohjelman algoritmeissa muokataan Kartta luokan ominaisuuksia niin, että niitä ei voi suoraan käyttää luokan muissa algoritmeissa, joten luokassa Kartta on myös Paivita metodi, jonka avulla päivitetään Kartta luokan ominaisuudet.

5.1 Approksimointi/Heuristiset algoritmit

Approksimointi/heuristiset algoritmit eivät tuota parasta mahdollista tulosta mutta ne tuottavat hyvän tuloksen käytännöllisessä ajassa. (Brilliant.org 2020, Algorithm.)

5.1.1 Nearest Neighbor

Tässä kappaleessa esittelen intuitiivisesti hyvältä vaikuttavan algoritmin, jonka avulla voidaan muodostaa kauppatkustajan kierros.

Algoritmissa mennään aina siihen kaupunkiin, joka on lähimpänä sitä kaupunkia, jossa ollaan, ja kaupungin pitää olla myös sellainen, jossa ei olla vielä käyty. Tämä lähestymistapa ei kuitenkaan tuota optimaalista ratkaisua kovinkaan usein. Alussa algoritmi löytää hyviä ratkaisuja mutta se saattaa johtaa harhaan niin, että lopussa jäljellä olevat vaihtoehdot ovat järkyttävän huonoja. Tulokset ovat huonoja varsinkin tilanteissa, joissa kolmikulmaisuuksien sääntöä ei käytetä. Kun kolmikulmaisuuksien sääntö on käytössä, algoritmin huonoin mahdollinen tulos on $1 + \log_2(n)/2$, kertaa pidempi kuin optimaalisimman kierroksen pituus. (Cook, 2012, 96–97.)

Esimerkkejä sisältävässä ohjelmassa on luokka Algoritmit, jossa on ominaisuutena, luokan kartta instanssi, summa vektori, johon kerätään polun matkojen pituudet ja merkkijonoja sisältävä vektori, johon kerätään polun kaupungit samassa järjestyksessä kuin niissä on käyty.

Toimintana on mm. NearestNeighbor algoritmi. Tämän algoritmin toteutus tapahtuu niin, että ensin valitaan aloituskaupunki, lisätään se vektoriin, johon kerätään kaupunkia, joissa on jo käyty ja poistetaan se vektorista, josta etsitään kaupunkia, joihin pitäisi vielä matkustaa. Seuraavaksi etsitään viimeksi käytyjen kaupunkien vektoriin lisäystä kaupungista lyhintä reittiä sellaiseen kaupunkiin, johon pitää vielä matkustaa ja tätä toistetaan, kunnes matkustuskohdetta sisältävä vektori on tyhjä. Matkan pituuteen lisätään uuden matkan pituus, joka kerta kun käytyjen kaupunkien listaan lisätään uusi kaupunki.

```

void NearesNeighbor(int sijainti){
    summa.clear();
    do{
        matka.push_back(rakenne.kaupungit[sijainti]);
        int pituus = 9999;
        int temp = 0;
        for(int i = 0; i < rakenne.n[sijainti].size(); i++){

            if((rakenne.n[sijainti][i] != 0)&&(rakenne.n[sijainti][i] < pituus)){
                pituus = rakenne.n[sijainti][i];
                temp = i;
            }

        }
        summa.push_back(rakenne.n[sijainti][temp]);
        rakenne.PoistaKaupunki(sijainti);

        if(temp == 0){
            sijainti = temp;
        }
        else{
            sijainti = temp-1;
        }
    }while(rakenne.n.size() != 0);
}

```

Kuva 7. NearestNeighbor metodi.

Lopuksi aikaisemmin luodusta polusta muodostetaan kierros lisäämällä viimeisestä kaupungista matka polun ensimmäiseen kaupunkiin, joka tulostetaan.

```

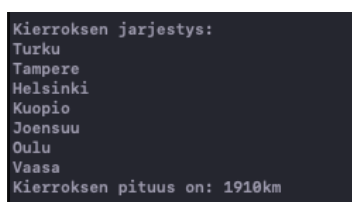
rakenne.Paivita();

int mista = 999, mihin = 999;
for(int i = 0; i < rakenne.n.size(); i++){
    if(matka[0] == rakenne.kaupungit[i]){
        mista = i;
    }
    if(matka[matka.size()-1] == rakenne.kaupungit[i]){
        mihin = i;
    }
}

summa.push_back(rakenne.n[mista][mihin]);
TulostaKierros();
rakenne.Paivita();

```

Kuva 8. Polun viimeistely.



```

Kierroksen järjestys:
Turku
Tampere
Helsinki
Kuopio
Joensuu
Oulu
Vaasa
Kierroksen pituus on: 1910km

```

Kuva 9. Muodostettu kierros.

5.1.2 Cristofidesin algoritmi

Algoritmin aikakompleksisuus on $O(n^3)$, kun kolmikulmaisuuuden sääntö on käytössä. Algoritmi koostuu vaiheista, ensimmäisessä vaiheessa kaupunkien välille luodaan minimum spanning tree -puurakenne. Ensimmäisen vaiheen tuottamalle rakenteen osalle suoritetaan minimum perfect matching, josta löytyy Eulerin polku, joka sitten muutetaan kierrokseksi. Huonoimmassa tapauksessa algoritmin tuottaman kierroksen pituus on $2/3$ lyhyintä mahdollista kierrosta pidempi, joka oli algoritmin

keksimisen aikaan 50 % parempi kuin paras mahdollinen polynomisessa ajassa toimiva algoritmi. (Nicos Christofides, 1976.)

Ensimmäiseksi pitää luoda kaupunkien välille minimum spanning tree -puurakenne. Puurakenne sisältää verkoston kaikki pisteet, jotka ovat tässä tapauksessa kaupunkia. Kaupunkien väliset tiet ovat verkoston kaaria. Puurakenteessa ei saa esiintyä kiertokulkua. Puurakenne luodaan valitsemalla siihen ensimmäinen kaari. Rakenteeseen lisätään kaaria niin, että lisättävässä kaareissa pitää olla yksi sellainen piste, joka kuuluu rakenteeseen ja kaaren toisen pisteen tulee olla sellainen, että se ei kuulu rakenteeseen. Minimum spanning tree on puurakenne, jossa rakenteen kaarien kustannus/pituus on mahdollisimman pieni. (tutorialspoint.com, 2020, Data structures & Algorithms- Spanning trees.)

Toteutin esimerkki ohjelmaani minimum spanning tree rakenteiden luonnin seuraavalla tavalla. Luokassa Algoritmit on pareja sisältävä vektori. Vektorissa oleva pari ilmaisee rakenteeseen jo kuuluvaa kaarta. MinimumSpanningTree metodissa etsitään lyhyintä reittiä sellaiseen kaupunkiin, jossa ei olla käyty sellaisesta kaupungista, jossa on käyty, ja etsittävien kaupunkien avaruus pienenee sitä mukaan, kun kaupungeissa käydään. Joka kierroksella muodostetaan kaupunkien välille pari ja se lisätään parien vektoriin. Ohjelmaa toistetaan, kunnes jokaisessa kaupungissa on käyty.

```
void MinimumSpanningTree(int mista){
    summa.clear();
    vector<int> kaydyt;
    int mihin = 0;
    do{
        rakenne.PoistaKohde(mista);
        kaydyt.push_back(mista);
        int pituus = 9999;
        for(int i = 0; i < kaydyt.size(); i++){
            for(int j = 0; j < rakenne.n[kaydyt[i]].size(); j++){
                if((rakenne.n[kaydyt[i]][j] != 0)&&(rakenne.n[kaydyt[i]][j] < pituus)){
                    pituus = rakenne.n[kaydyt[i]][j];
                    mista = kaydyt[i];
                    mihin = j;
                }
            }
        }
        summa.push_back(rakenne.n[mista][mihin]);
        puu.push_back(make_pair(mista, mihin));
        mista = mihin;
    }while(kaydyt.size() < rakenne.n.size()-1);
    sort(puu.begin(), puu.end());
    rakenne.Paivita();
    TulostaPuu();
}
```

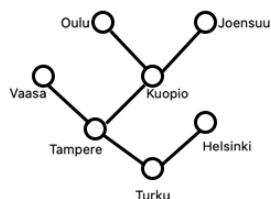
Kuva 10. MinumumSpanningTree metodi.

Kun parien etsintä on suoritettu, parit järjestetään ja tulostetaan.

```
Turku -> Helsinki
Turku -> Tampere
Tampere -> Vaasa
Tampere -> Kuopio
Kuopio -> Joensuu
Kuopio -> Oulu
Kaarien summa on: 1289km
```

Kuva 11. Parit.

Puu visualisoituna:



Kuva 12. MinimumSpanning tree.

Cristofidesin algoritmin seuraavassa vaiheessa pitää minimum spanning tree -puurakenteen sellaisille pisteille, joihin johtaa pariton määrä kaaria suorittaa minimum perfect matching. Matching tarkoittaa, että verkostosta valitaan sellaiset kaaret, joilla ei ole yhteisiä pisteitä. Jokaiseen pisteeseen johtaa nolla tai yksi kaarta. Minimum matching suosii sellaisia verkoston kaaria joiden kustannus/pituus on mahdollisimman pieni. Perfect matching tarkoittaa, että verkoston jokainen piste on yhdistetty tasan yhteen kaareen. Minimum perfect matching tuottaa sellaisen verkoston, jossa jokainen piste on yhdistetty tasan yhteen kaareen ja sen kokonais kustannus/pituus on mahdollisimman pieni. (Brilliant.org, 2020, Matching Graph Theory.)

Tein esimerkin siitä, miten minimum spanning rakenteelle tehdään minimum perfect matching.

```

void EulerinPolku(){
    vector<int> kaaret;
    for(int i = 0; i < puu.rakenne.n.size(); i++){
        kaaret.push_back(0);
    }

    for(int i = 0; i < puu.puu.size(); i++){
        kaaret[puu.puu[i].first]++;
        kaaret[puu.puu[i].second]++;
    }

    int j = 0;
    for(int i : kaaret){
        if(i % 2 != 0){
            lisaa.push_back(j);
        }
        j++;
    }
}

```

Kuva 13. EulerinPolku metodi.

```

int i = 0, temp = 0;

do{
    for(int j = 0; j < lisaa.size(); j++){
        if((puu.rakenne.n[lisaa[i]][lisaa[j]] != 0) && (puu.rakenne.n[lisaa[i]][lisaa[j]] < pituus)){
            pituus = puu.rakenne.n[lisaa[i]][lisaa[j]];
            mista = lisaa[i];
            mihin = lisaa[j];
            temp = j;
        }
    }

    puu.puu.push_back(make_pair(mista, mihin));
    puu.summa.push_back(puu.rakenne.n[mista][mihin]);
    pituus = 999;
    lisaa.erase(lisaa.begin()+0);
    lisaa.erase(lisaa.begin()+temp-1);
}while(lisaa.size() != 0);

puu.TulostaPuu();

```

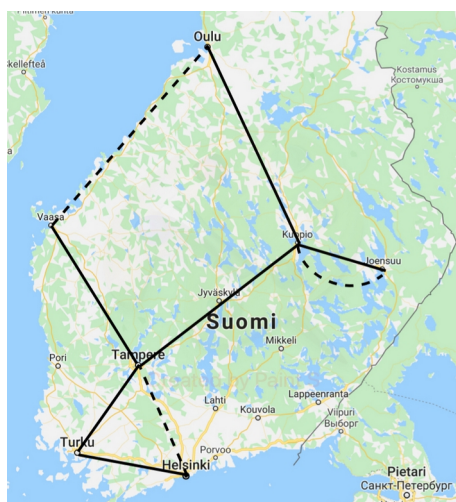
Kuva 14. Metodin toiminta.

Lopuksi tulostetaan lopullinen rakenne.

```
Turku -> Helsinki
Turku -> Tampere
Tampere -> Vaasa
Tampere -> Kuopio
Kuopio -> Joensuu
Kuopio -> Oulu
Helsinki -> Tampere
Vaasa -> Oulu
Kuopio -> Joensuu
Kaarien summa on: 2543km
```

Kuva 15. Eulerin polku.

Visualisointi, jossa mustat viivat ovat minimum spanning tree rakennetta kuvaavia ja katkoviivat ovat edellisessä metodissa lisätyt kaaret.



Kuva 16. Polun visualisointi.

Cristofidesin algoritmin seuraavassa vaiheessa muodostetaan kierros aikaisemmin luodusta rakenteesta.

Tein myös tästä vaiheesta esimerkin. Aloitetaan lisäämällä ensimmäinen kaupunkien pari vektoriin, jonka jälkeen tämä kaupunkien pari poistetaan etsittävien parien joukosta. Käynnistetään toistorakenne, jota toistetaan, kunnes jokainen kaupunki kuuluu lyhennettyyn kierrokseen. Kierrokseen lisätään pareja niin, että etsitään Eulerin polkuun kuuluvien parien joukosta sellaista paria, joka alkaa siitä kaupungista, johon lyhennettyyn kierrokseen viimeksi lisätty pari loppui. Etsittävä pari ei myöskään saa päättyä siihen kaupunkiin, josta viimeksi lisätty pari alkaa. Heti, kun ehdot täyttävä kaupunki löytyy, se lisätään lyhennettyyn kierrokseen, lisätyn parin järjestysnumero tallennetaan, jonka avulla se poistetaan etsittävien joukosta.

Algoritmissa on myös toinen ehto, joka on täysin instanssi kohtainen. Eli se toimii tässä esimerkissä käytettävällä etäisyysmatriisilla mutta yleisellä tasolla se ei toimi. Jos ensimmäisiä ehtoja täyttävää paria ei löydy, etsitään sellaista paria, jossa aloitus kaupunki ei ole sama kuin viimeksi lyhennettyyn kierrokseen lisätyn mutta toinen kaupunki on sama. Jos tällainen pari löytyy, muutetaan sen järjestys päinvastaiseksi, jonka jälkeen se lisätään lyhennettyyn kierrokseen. Tämän ehdon ansiosta algoritmi

lisää kierrokseen parin, joka sisältää matkan Kuopiosta Ouluun mutta tallentaa sen matkana Oulusta Kuopioon.

Etsinnän päätyttyä, jos ensimmäinen ehto oli tosi, poistetaan se pari, joka viimeksi lisättiin lyhennettyyn kierrokseen etsittävien joukosta ja toistetaan etsimisprosessi mutta tällä kertaa se tehdään viimeksi lisätystä parista.

```
void oikaisu(){
    vector<pair<int, int>> oikaisu;
    int temp = 0, kayty = 0;
    oikaisu.push_back(make_pair(puu.puu[0].first, puu.puu[0].second));
    puu.PoistaKaari(0);
    do{
        for(int i = 0; i < puu.puu.size(); i++){
            if((puu.puu[i].first == oikaisu[temp].second)&&(puu.puu[i].second != oikaisu[temp].first)){
                oikaisu.push_back(make_pair(puu.puu[i].first, puu.puu[i].second));
                temp++;
                kayty = temp;
                puu.PoistaKaari(kayty);
                i = (int)puu.puu.size();
            }
            else if((puu.puu[i].first != oikaisu[temp].first)&&(puu.puu[i].second == oikaisu[temp].second)){
                oikaisu.push_back(make_pair(puu.puu[i].second, puu.puu[i].first));
                temp++;
                i = (int)puu.puu.size();
            }
        }
    }while(oikaisu.size() < puu.rakenne.kaupungit.size()-1);
}
```

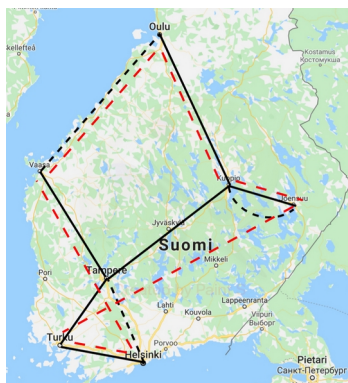
Kuva 17. Kierroksen oikaisu.

Sitten kun jokainen kaupunki kuuluu oikaistuun kierrokseen ja toistorakenteen suorittaminen on lopetettu, lisätään kierrokseen vielä paluu siihen kaupunkiin, josta se aloitettiin. Se tehdään lisäämällä kierrokseen sellainen pari, jonka ensimmäinen osa on se kaupunki, johon kierros loppuu ja toinen osa on se kaupunki, josta kierros alkaa.

```
temp = (int)oikaisu.size();
oikaisu.push_back(make_pair(oikaisu[temp-1].second, oikaisu[0].first));
for(pair<int, int> p : oikaisu){
    cout << puu.rakenne.kaupungit[p.first] << " -> " << puu.rakenne.kaupungit[p.second] << "\n";
}
cout << "\n";
```

Kuva 18. Polun muodostaminen.

Algoritmin luoma rakenne tulostettuna ja sen vieressä viimeksi luotu kierros merkittynä karttaan punaisella katkoviivalla, minimum spanning tree mustalla ja Eulerin polun lisäykset edelliseen merkittynä mustalla katkoviivalla. Kuten kuvasta näkee, kierros menee ns. itsensä yli, joten sitä voi vielä parannella.



Kuva 19. Polut kartalla.

5.1.3 K-opt optimointi

Kierrosta voidaan vielä parantaa ja se tehdään k-opt siirtojen avulla, joka tapahtuu niin, että etsitään virheitä ja sen jälkeen virheitä yritetään korjata. Virheiden korjaamisessa etsitään jokin määrä kaaria, jotka voidaan korvata lyhyemmällä kaarilla. Korvattavien kaarien määrä ilmaistaan termillä k-opt, jossa k on korvattavien kaarien lukumäärä. Esimerkiksi, jos kierroksesta korvataan kaksi kaarta lyhyemmällä kaarilla se tarkoittaisi, että kierrokselle tehtiin yksi 2-opt siirto. Yksi vanhimmista teoreemoista koskien kauppamatkustajan instansseja on, että Euklidisessa avaruudessa lyhin mahdollinen kierros ei koskaan ylitä itseään tai toisin sanoen mene ristiin itsensä kanssa. Edellinen teoreema voidaan todistaa tekemällä ristin sisältävälle kierrokselle 2-opt siirto, jonka avulla poistetaan risti ja se johtaa aina siihen, että kierros lyhenee. (Cook, 2012, 130–132.)

K-opt siirtoja tehdessä k:n arvo voitaisiin kasvattaa mutta se johtaa epäkäytännöllisyyteen, jos k on paljon suurempi kuin kaksi tai kolme. (Cook, 2012, 134.)

Tämän algoritmin tarkoituksena on tehdä aikaisemmin luodusta kierroksesta sellainen, että se ei mene itsensä kanssa ristiin. Siinä on viisi integer tyyppistä muuttujaa, joihin tallennetaan tietoa toistorakenteesta. Toistorakenteesta etsitään aluksi ensimmäisestä kaupungista lyhintä mahdollista reittiä toiseen kaupunkiin. Lyhimmän kaupungin löydyttyä sen järjestysnumero tallennetaan muuttujaan, jonka avulla lyhennetystä kierroksesta poistetaan kaikki muuttujan kaupunkiin menevät parit, samalla otetaan talteen se kaupunki, josta poistettava pari alkavat. Samassa rakenteesta poistetaan jokainen pari, joka päättyy siihen kaupunkiin, josta etsittiin lyhintä reittiä ja näistä poistettavista pareista otetaan talteen lähtökaupungit. Viimeiseksi etsitään sellaisia pareja, jotka alkavat siitä kaupungista, josta aluksi etsittiin lyhintä reittiä ja jokaisen löytyneen parin viimeinen kaupunki vaihdetaan siksi kaupungiksi, johon oli lyhin matka ensimmäisessä etsinnässä.

```
void tarkistus(vector<pair<int, int>> kierros){
    int pituus = 999;
    int mihin = 0, mista = 0, i = 0, temp = 0, tempp = 0;
    mista = kierros[i].first;
    do{
        for(int j = 0; j < puu.rakenne.n[kierros[i].first].size(); j++){
            if((puu.rakenne.n[kierros[i].first][j] < pituus)&&(puu.rakenne.n[kierros[i].first][j] != 0)){
                pituus = puu.rakenne.n[kierros[i].first][j];
                mihin = j;
            }
        }

        for(int k = 0; k < kierros.size(); k++){
            if(kierros[k].second == mihin){
                temp = kierros[k].first;
                kierros.erase(kierros.begin()+k);
                k--;
            }
            if(kierros[k].second == mista){
                tempp = kierros[k].first;
                kierros.erase(kierros.begin()+k);
                k--;
            }
        }

        for(int k = 0; k < kierros.size(); k++){
            if(kierros[k].first == mista){
                kierros[k].second = mihin;
            }
        }

        i++;
    }while(i == 0);
}
```

Kuva 20. Tarkistus metodin toiminta.

Nyt lisätään kierrokseen kaksi uutta paria. Sen parin kaupunki, joka poistettiin koska se sisälsi sen kaupungin, johon oli lyhin matka algoritmin ensimmäisessä vaiheessa, asetetaan ensimmäisen uuden parin ensimmäiseksi kaupungiksi ja toisen uuden parin toiseksi kaupungiksi.

Sen parin kaupunki, joka poistettiin, koska se päättyi siihen kaupunkiin, josta etsittiin lyhintä reittiä algoritmin ensimmäisessä vaiheessa, asetetaan ensimmäisen uuden parin toiseksi kaupungiksi.

Toisen uuden parin toinen kaupunki on se kaupunki, josta aluksi etsittiin lyhintä reittiä toiseen kaupunkiin.

```

mihin = (int)kierros.size();
kierros.push_back(make_pair(temp, temp));
kierros.push_back(make_pair(temp, mista));
for(pair<int, int> p : kierros){
    cout << puu.rakenne.kaupungit[p.first] << " -> " << puu.rakenne.kaupungit[p.second] << "\n";
}

```

Kuva 21. Ohjelman toimintaa.

Algoritmin tuotos tulostettuna:

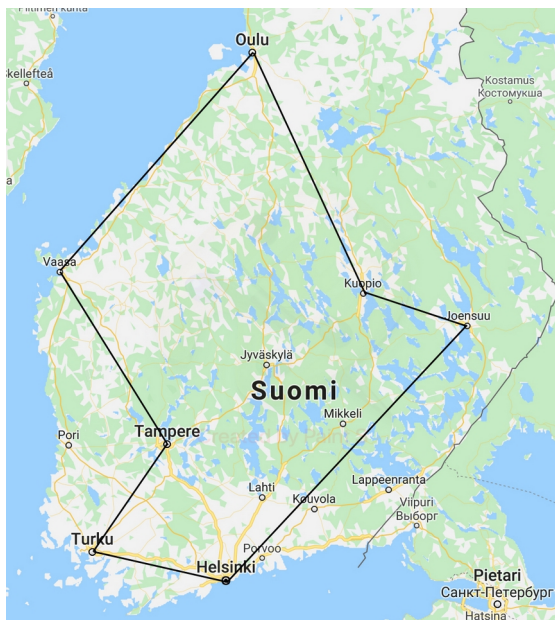
```

Turku -> Tampere
Tampere -> Vaasa
Vaasa -> Oulu
Oulu -> Kuopio
Kuopio -> Joensuu
Joensuu -> Helsinki
Helsinki -> Turku

```

Kuva 22. Polku tulostettuna.

Edellinen kierros kartalla:



Kuva 23. Polun visualisointi.

5.1.4 Lin-Kernighan

Erittäin tehokas menetelmä, jossa käytetään k-opt optimointia ja jonka avulla saadaan aikaan suhteellisen lyhyt kierros nopeasti, vaikka kaupunkien määrä olisi suuri. Lyhyimmän mahdollisen kierroksen löytymisestä nopeasti, ei kuitenkaan ole takuita.

Menetelmässä etsitään jo muodostetusta kierroksesta lyhyempää versiota muokkaamalla jo muodostettua kierrosta k-opt siirtojen avulla. Jos lyhyempi versio löytyy, korvataan muodostettu kierros tällä lyhyemmällä kierroksella, jonka jälkeen etsitään lyhyempää versiota aikaisemmin luodusta lyhyemmästä kierroksesta. Jos lyhyempää kierrosta ei löydy niin kierros on paikallisesti optimaalinen, tämän jälkeen voidaan muodostaa kokonaan uusi kierros ja etsiä siitä lyhyempiä versioita, kunnes käytettävissä oleva aika loppuu tai tulos on hyväksyttävä. (Lin & Kernighan, 1973.)

Kierroksesta lyhyempien versioiden etsintä tapahtuu niin, että

- 1 Valitaan kierroksesta K, jokin kaupunki x_1 ja siitä kaupungista lähtevä kaari k_1 . Määritetään myös muuttuja i ja sen arvo 1.
- 2 Kaupungista x_1 lähtevän kaaren k_1 toisessa päässä olevasta kaupungista x_2 valitaan kaari y_1 , jonka lisääminen kierrokseen K, kaaren k_1 tilalle lyhentäisi kierrosta K. Jos ehtoja täyttävää kaarta y_1 ei löydy siirrytään vaiheeseen kuusi.
- 3 Muutetaan muuttujaa $i = i + 1$. Valitaan kaaret k_i ja y_i niin, että
 - a Jos kaupunki x_{2i} , joka on tässä tapauksessa kaaren k_i päässä, liitetään kaupunkiin x_1 muodostuu kierros.
 - b Kaari y_i :n toinen pää sijaitsee kaupungissa x_{2i} , josta lähtee kaari k_i . Jos tällaista kaarta y_i ei ole siirrytään vaiheeseen neljä.
 - c Sen varmistamiseksi, että k-kaari ei jatka y-kaarta, kaari x_i ei saa olla sellainen, joka on juuri lisätty kierrokseen K ja kaari y_i ei saa olla sellainen kaari, joka aikaisemmin poistettu kierroksesta K.
 - d Lisättyjen kaarien pitää lyhentää kierrosta K.
 - e Lisättävän kaaren y_i pitää sallia kaaren k_{i+1} poistaminen.
 - f Ennen kaaren y_i rakentamista tarkistetaan, että lyhennetty kierros on toimiva. Se tehdään niin, että liitetään kaupunki x_{2i} , kaupunkiin x_1 ja jos tämä liitos lyhentää kierrosta on kierros toimiva.
- 4 Perutaan kaarien x_i ja y_i muodostus vaiheissa yksi-kolme, jos sellaisia linkkejä x_i ja y_i , jotka toteuttavat vaiheen neljä ehdot c-e ei löydy tai jos

kierrosta K lyhentävää kaarta ei löydy. Jos tähän mennessä ollaan, löydetty alkuperäistä kierrosta K lyhyempi kierros aloitetaan koko prosessi alusta käyttäen sitä kierrosta, joka löydettiin ennen tätä vaihetta.

- 5 Jos kierrokseen K ei löytynyt parannuksia mennään algoritmista taaksepäin.
 - a Toistetaan askeleet neljä ja viisi. Etsitään vaihtoehtoisia kaaria y_2 , joiden lisääminen kierrokseen lyhentää sitä.
 - b Jos kaikki vaihtoehtoiset kaaret y_2 ovat sellaisia, että ne eivät paranna kierrosta, suoritetaan vaihe 3 (a), jossa etsitään vaihtoehtoisia kaaria kaaren k_2 tilalle.
 - c Jos edellisenkään vaihe ei tuota parannusta kierrokseen palataan vaiheeseen kaksi, jossa etsitään vaihtoehtoisia kaaria kaaren y_1 tilalle.
 - d Jos edellinen vaihe ei tuota parannusta kierrokseen, etsitään vaihtoehtoisia kaaria kaaren k_1 tilalle.
 - e Jos mikään edellisistä vaiheista ei tuota parannusta kierrokseen palataan vaiheeseen yksi mutta tällä kertaa etsintä suoritetaan toisesta kaupungista x_1 kuin viimeksi.
- 6 Prosessi lopetetaan, kun jokainen kaupunki on toiminut aloitus kaupunkina x_1 ja parannuksia ei ole löytynyt. Algoritmin voi aloittaa alusta uudella satunnaisesti luodulla kierroksella. (Lin & Kernighan, 1973).

5.1.5 Lin-Kernighan-Helsgaun

Lin-Kernighan metodin tehokkuutta on ylläpitänyt tutkija yhteisöltä tasaiseen tahtiin tulleet parannukset. Suurin osa tästä tuesta oli siis tullut hienosäädön muodossa, kunnes tietokonetieteilijä Keld Helsgaun muokkasi etsintään liittyvän algoritmin ytimen uuteen muotoon vuonna 1998. (Cook, 2012, 114.)

Lin-Kernighan-Helsgaun menetelmän muutokset muuttivat alkuperäisen algoritmin yksityiskohtia. Isoimmat muutokset alkuperäiseen algoritmiin liittyivät siinä tapahtuvaan lyhyempien kierroksien etsintään niin, että Keld Helsgaunin versiossa etsintä avaruus on isompi ja kompleksisempi kuin alkuperäisessä algoritmista. Algoritmin uudistettu versio oli kasvaneesta kompleksisuudesta huolimatta tehokkaampi kuin alkuperäinen algoritmi ja tämä tehokkuus mahdollisti lyhyimmän mahdollisen kierroksen löytymisen, vaikka ongelmassa olisi ollut suuri määrä kaupunkeja. Tyypillisestä 100 kaupungin ongelmasta algoritmi löytää lyhimmän mahdollisen reitin alle sekunnissa ja 1000 kaupungin ongelmasta se selvittää lyhimmän mahdollisen reitin alle minuutissa. (Helsgaun, 1976.)

Algoritmi on todella kompleksinen enkä aio selventää sitä enempää tässä työssä mutta jos se herättää kiinnostusta lähdeluettelosta löytyy menetelmää käsittelevän alkuperäisen julkaisun nimi, jossa algoritmin kehittäjä selventää sen toimintaa.

5.2 Lyhyimmän kierroksen etsintä

Kerron tässä kappaleessa Lineaarisesta ohjelmoinnista. Lineaarinen ohjelmointi oli minulle ennen tätä työtä melkein täysin tuntematon aihe. Työssä pääsin tutustumaan siihen hieman ja voin sanoa, että se on mielenkiintoinen ja erittäin käytännöllinen työkalu tietyissä tilanteissa.

Lineaarinen ohjelmointi on metodi, jota käytetään tilanteissa, joissa yritetään muodostaa lyhin mahdollinen kierros. Lineaarisen ohjelmoinnin tarina alkaa 1939 luvulla, kun George Dantzig saapui myöhässä oppitunnille, jossa hän näki taululla kaksi ongelmaa ja laitto ne itselleen muistiin, koska hän luuli ongelmien olevan kotiläksyjä. George palautti tehtävät muutaman päivän päästä opettajalleen samalla pahoitellen sitä, että niiden ratkaisussa oli mennyt normaalia pidempi aika. Kuuden viikon kuluttua Georgelle selvisi hänen opettajansa toimesta, että ongelmat, jotka hän ratkaisi, olivat kuuluisia ratkaisemattomia tilastotieteen ongelmia. Nämä kotiläksyjen vastaukset toimivat lineaarisen ohjelmoinnin teorian pohjana. Myöhemmin tämä tarina on esiintynyt elokuvissa ja uskonnollisten saarnaajien todisteena siitä miten positiivinen ajattelu auttaa ratkaisemaan ongelmia. (Davis, 2019; Cook, 2012, 154.)

5.2.1 Lineaarinen ohjelmointi

Lineaarinen ohjelmointi on optimointi tekniikka, jota voidaan käyttää systeemeissä, joissa on lineaarisia muuttujia ja lineaarinen objektiivinen funktio. Objektiivinen funktio määrittelee sen, minkälaiseen muuttujan tai muuttujien arvoon optimoinnilla pyritään. Lineaarisen ohjelmoinnin tarkoituksena on siis löytää sellaiset muuttujien arvot, jotka maksimoivat tai minimoivat objektiivisen funktion. Lineaarisesta ohjelmoinnista on hyötyä tilanteissa, joissa resurssien jakoa pitää optimoida, kuten tuotantotaloudessa tai logistiikassa. (Brilliant.org, 2020, Linear Programming).

Tilanteissa, joissa muuttujia ja rajoitteita on muutamia, voidaan soveltaa lineaarista ohjelmointia. Tilanteessa, jossa muuttujia ja rajoitteita on satoja tai tuhansia on kannattavampaa käyttää lineaarisen ohjelmoinnin menetelmää, jonka nimi on Simplex algoritmi. Simplex algoritmin toiminta perustuu iterointiin. Jokaisella iterointi kerralla parannellaan yhtälöiden arvoja, kunnes optimaalinen ratkaisu löytyy. (Brilliant.org, 2020, Linear Programming; Cook, 2012, 1.)

6 LOPUKSI

Kokonaisuutena työtä oli mukava kirjoittaa, koska käsittelen siinä itselleni mielenkiintoista aihetta, mutta osaamiseni ei kuitenkaan ollut, eikä ole riittävällä tasolla, eikä myöskään riittävän spesifioitua, että voisin kirjoittaa tästä aiheesta teknisesti todella hyvin. En kuitenkaan tiennyt sitä ennen kuin aloin kirjoittamaan tätä työtä, nyt kuitenkin ymmärrän, että esimerkiksi jokaisesta ongelman ratkaisutavasta, joita työssä esittelin voisi luultavasti melko helposti kirjoittaa erillisen työn.

Ongelman historian selvittäminen oli mielenkiintoista, kuin myös termistöihin tutustuminen. Historian avulla oli helpompi ymmärtää, kuinka paljon työtä tämän tyyppisten ongelmien ratkaisemisen eteen tehty ja miksi. Termistöjen osalta työhön päätyi vain pieni osa siitä mitä aluksi olin ajatellut mutta mitään välttämätöntä ei mielestäni jäänyt puuttumaan.

Heuristiset menetelmät olivat sellaisia, joista olin kuullut aiemmin mutta niihin liittyvät tieteelliset julkaisut olivat todella yksityiskohtaisia ja komplekseja. Tämä kompleksisuus tuli esiin myös tilanteissa, joissa löysin tätä minun työtäni muistuttavia töitä samalla, kun etsin lähdemateriaaleja ja näistä töistä kaikki olivat, joko matematiikan opiskelijoiden tai ammattilaisten kirjoittamia ja niissäkin oli tiivistetty yksityiskohtia sen vuoksi, että algoritmit olivat kompleksisia. Itse yritin selittää Lin-Kernighan algoritmin toiminnan vaihevaiheelta mutta Lin-Kernighan-Helsgaun algoritmin toiminnan jouduin tiivistämään. Toiminnoiltaan molemmat ovat minun mielestäni erittäin hienosti rakennettuja. Kompleksisuudestaan huolimatta ne olivat erittäin yksinkertaisia ja jollakin tapaa jopa taiteellisia. Christofidesin algoritmi oli mielestäni kaikessa yksinkertaisuudessaan mielenkiintoisin ja se oli myös suhteellisen helposti löydettävissä.

Concorde algoritmissa, joka nousi esiin tuotantotalouden esimerkeissä, käytettävä lineaarinen ohjelmointi oli minulle täysin uusi asia, joten tässä työssä en päässyt sitä avaamaan melkein yhtään mutta ilman tätä työtä en luultavasti olisi tutustunut siihen ollenkaan ja aion tutustua siihen lisää, koska sen hyödyllisyys tilanteissa, joissa muuttujien suhteet ovat lineaarisia on ilmiselvää.

Koodien kirjoittaminen oli myös mielenkiintoista. Yritin tehdä ja teinkin sen niin, että kirjoitin koodit sen perusteella mitä olin lukenut, en siis etsinyt mistään koodeja ja kopioinut niitä tähän työhön.

Kokonaisuutena aihe on sellainen, että mitä enemmän siitä lukee sitä enemmän ymmärtää, kuinka vähän koko aiheesta tietää. Maailmassa on erittäin päteviä ihmisiä, jotka käyttävät melkeinpä koko työuransa tällaisten ongelmien ymmärtämiseen ja ratkaisu metodien kehittämiseen. Luulen, että isoin asia, jonka tässä työssä opin liittyy tieteellisten lähteiden käyttöön sekä ymmärtämiseen ja sitä kautta siihen, että osaan etsiä vastausta oikeasta paikasta tilanteissa, joissa esiintyy kombinatorista optimointia.

LÄHTEET

William J. Cook, 2012. In pursuit of the traveling salesman. Princeton University Press.

Stephan C. Carlson, 2020. Königsberg bridge problem. Viitattu 20.05.2020. <https://www.britannica.com/science/Konigsberg-bridge-problem>

Brilliant.org, 2020. Hamiltonian Path. Viitattu 21.05.2020. <https://brilliant.org/wiki/hamiltonian-path/>

Brilliant.org, 2020. Big O Notation. Viitattu 21.05.2020. <https://brilliant.org/wiki/big-o-notation/>

ursa.fi, 2020. Auringon ja tähtien elinkaari. Viitattu 07.05.2020. <https://www.ursa.fi/yhd/komeetta/esitelma/aurinko.html>

Brilliant.org, 2020. Complexity classes. Viitattu 08.05.2020. <https://brilliant.org/wiki/complexity-classes/>

Brilliant.org. 2021. Algorithm. Viitattu. 08.05.2020. <https://brilliant.org/wiki/algorithm/>

tutorialspoint.com, 2020. Data structures & Algorithms- Spanning trees. Viitattu 27.05.2020. https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm

developers.google.com, 2020. C++ References: Cristofides. Viitattu 02.06.2020. <https://developers.google.com/optimization/reference/graph/christofides>

Brilliant.org, 2020. Linear Programming. Viitattu 09.06.2020. <https://brilliant.org/wiki/linear-programming/>

britannica.com, 2020. Simplex method. Viitattu 09.06.2020. <https://www.britannica.com/topic/simplex-method>

Nicos Christofides, 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Carnegie-Mellon university.

Brilliant.org, 2020. Matching (Graph Theory). Viitattu 10.06.2020. <https://brilliant.org/wiki/matching/>

S. Lin ja B.W Kernighan, 1973. An Effective Heuristic Algorithm for the Travelling-Salesman Problem. INFORMS.

Keld Helsgaun. An Effective Implementation ff the Lin-Kernighan Traveling Salesman Heuristic. Department of Computer Science Roskilde University.

Matt Davis, 29.03.2019. Remembering George Dantzig: The real Will Hunting.
Viitattu 12.06.2020. <https://bigthink.com/culture-religion/george-dantzig-real-will-hunting>