Tampereen yliopisto

John Mäkelä

SUUNNISTUSONGELMA JA HEURISTISTEN RATKAISUJEN EMPIIRINEN ARVIOINTI

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Maaliskuu 2024

Tiivistelmä

John Mäkelä: Suunnistusongelma ja heuristisiten ratkaisujen empiirinen arviointi
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelyopin maisteriohjelma
Maaliskuu 2024

Kombinatorisessa optimoinnissa tavoitteena on löytää paras ratkaisu äärellisestä vaihtoehtojen joukosta: monet tunnetut laskennan rajoja koettelevat ongelmat kuuluvat tähän luokkaan. Useille näistä ongelmista ei ole esitetty tehokasta algoritmia, jonka laskenta-aika skaalautuisi kohtuullisesti (polynomisesti) syötteen koon kasvaessa. Tässä tutkimuksessa käsitellään kombinatoristen ongelmien heuristisia ratkaisumenetelmiä, jotka eivät takaa parhaan mahdollisen ratkaisun löytymistä. Teoreettisten takuiden puuttuessa empiiristen mittausten rooli korostuu, mutta samaan aikaan mittausten tekeminen vertailukelpoisella tavalla ja oikean kokoluokan ongelmissa on haastavaa. Syntyy tarve kehittää erilaisia tapoja mitata ratkaisimen suorituskykyä ja lisätä varmuutta heuristisiten ratkaisujen laadusta. Tässä työssä pyritään vastaamaan kyseiseen tarpeeseen soveltamalla heuristisia ratkaisumenetelmiä ja ratkaisimen suorituskyvyn mittaustapoja erityisesti suunnistusongelmaan, jota käsitellään ensin teorian tasolla ja myöhemmin vertaillaan toteutetun ratkaisimen tuloksia kirjallisuudessa esitettyihin ratkaisimiin. Lopputuloksena julkaistaan myös lähdekoodi suunnistusongelmageneraattoriin, joka tuottaa ennalta määritellyn optimin sisältäviä suunnistusongelmia mittaustarkoituksiin.

Avainsanat: suunnistusongelma, kombinatorinen optimointi, lineaariohjelmointi, heuristiset menetelmät, ahne heuristiikka, painotettu satunnaistaminen, ongelmageneraattori, kauppamatkustajan ongelma, reppuongelma, tilastollinen optimin estimointi, linkkuveitsimenetelmä, bootstrap-menetelmä, 3-opt, paikallinen etsintä, MAXIMP.

Abstract

John Mäkelä: The orienteering problem and empirical evaluation of heuristic solutions
Master's thesis
Tampere University
Master's Programme in Computer Science
March 2024

Combinatorial optimization concerns finding the best solution from a finite set of alternatives; several well-known problems that test the limits of computation fall into this category. For many of these problems, efficient (polynomial time) algorithms have not been presented. This thesis explores heuristic methods for solving combinatorial problems that do not guarantee finding the best solution. In the absence of theoretical guarantees, empirical measurements are of major importance, but at the same time acquiring comparable measurements for problems of the right size is challenging. To address this research gap, different ways to measure the performance of a solver and to increase confidence in the quality of heuristic solutions are explored by applying heuristic solution methods and measurement techniques to the orienteering problem, which is first discussed at a theoretical level and then the results from the implemented heuristic are compared with existing solvers from the literature. Finally, a problem generator for the orienteering problem, which generates problems with known optima suitable for measurement purposes, is published with source code.

Keywords: orienteering problem, combinatorial optimization, linear programming, heuristic methods, greedy heuristic, biased randomization, problem generator, traveling salesman problem, knapsack problem, statistical optimum estimation, jackknife method, bootstrap method, 3-opt, local search, MAXIMP.

Sisällys

1 Johdanto

> Kultakutri näki pöydällä höyryävät puurolautaset. Puuro tuoksui niin hyvältä, että hän päätti maistaa sitä. Ensin hän maistoi suuren suuresta lautasesta. ”Tämä on aivan liian kuumaa”, hän sanoi. Sitten hän maistoi aika suuresta lautasesta. ”Tämä on ihan liian kylmää”, hän tuumi. Ja viimein hän maistoi pikkuruiselta lautaselta. Se puuro oli juuri sopivaa, ja niin Kultakutri söi hyvällä ruokahalulla koko lautasellisen.
>
> *Kultakutri ja kolme karhua*
> *Robert Southey*
> *suom. Kati Weiss*

Monessa reaalimaailman ongelmassa on kyse parhaan vaihtoehdon löytämisestä, toisin sanoen *optimoinnista*. Jos vaihtoehtoja on vähän, on mahdollista kokeilla jokaista vaihtoehtoa ja valita paras vaihtoehto eli *optimi*. Niin Kultakutri teki vanhassa englantilaisessa sadussa maistettuaan ensin liian kuumaa puuroa, sitten liian kylmää puuroa, kunnes löysi juuri sopivan lämpöisen lautasellisen. Kultakutrin menetelmä, jota kutsutaan myös raa’an voiman hauksi, ei kuitenkaan sovellu ongelmiin, joissa vaihtoehtojen määrä on hyvin suuri. Tällöin kaikki aika menee maisteluun, kunnes jäljellä on vain kylmää puuroa.

Asiaan perehtymätön voi ajatella, että Kultakutrin kohtaamat aikarajoitteet eivät päde tietokoneella tehtävään laskentaan. Moderni tietokone pystyy kokeilemaan häkellyttävän määrän vaihtoehtoja pienessä ajassa ja esimerkiksi koneoppimisen viimeaikainen kehityskulku nojaakin alun alkaen peli- ja simulaattorimarkkinoille kehitettyjen grafiikkasuorittimien laskentatehon voimakkaaseen kasvuun. Samaan aikaan on kuitenkin olemassa ongelmia, joiden ratkaiseminen on edelleen vaikeaa mille tahansa tietokoneelle. Tässä tutkielmassa tarkasteltu *suunnistusongelma* (*orienteering problem*) kuuluu tähän joukkoon (epämuodollisesti) vaikeita ongelmia. Yksinkertaisimmillaan suunnistusongelmassa on kyse rogaining-suunnistuksen reitin suunnittelusta, jossa tavoitteena on kerätä mahdollisimman paljon pisteitä rajatussa ajassa käymällä rasteilla, joiden pistemäärä vaihtelee. Suunnistusongelmalla on kuitenkin sovelluksia urheilun ulkopuolella, esimerkiksi turistireittien suunnittelussa tai kun maksimoidaan mitä tahansa mitattavaa asiaa, joka on jakautunut kohteisiin, joiden välillä liikkumisesta seuraa kustannus, kun kokonaiskustannuksille täytyy asettaa raja.

Palaten yleiselle tasolle: mikä neuvoksi jos Kultakutri haluaisi kehittää nopeamman menetelmän, joka toimii suurella määrällä puurolautasia? Jos lautaset (ongelman ratkaisut) olisi järjestetty vasemmalta oikealle kylmimmästä kuumimpaan, hän

voisi kenties suorittaa *puolitushaun*, jossa lautassarjaa maistetaan keskeltä ja jatketaan etsintää vasemmalta puolelta, jos puuro on liian kuumaa, tai oikealta puolelta, jos puuro on liian kylmää, kunnes löydetään juuri sopivan lämpöinen lautanen. Tämä olisi *eksakti algoritmi*, joka päätyy todistettavasti parhaaseen mahdolliseen vaihtoehtoon lyhyemmässä ajassa kuin lautasellisten maistaminen järjestyksessä. Puolitushaku on nopeampi menetelmä, koska se sulkee kerralla pois laajoja alueita, joilla sijaitsee pelkästään epätyydyttäviä vaihtoehtoja.

Entä jos lautasia ei ole edes mahdollista laittaa nopeasti tarkkaan järjestykseen? Tällöin näyttää siltä, ettei pahimmassa tapauksessa ole muuta tapaa kuin maistaa kaikilta lautasilta, jos tavoitellaan parasta mahdollista puuroa. Toisaalta, jos ei vaadita parasta mahdollista, tyydyttävä vaihtoehto saattaisi löytyä nopeastikin esimerkiksi tunnustelemalla kämmenellä höyryäviä lautasia. Tämä olisi *heuristinen algoritmi* tai *heuristiikka* eli peukalosääntö, joka ei takaa parhaan mahdollisen vaihtoehdon löytymistä, mutta löytää riittävän usein riittävän hyvän vaihtoehdon. Mutta miten edes määritellä riittävän hyvä, jos emme ensin tiedä mikä on paras mahdollinen ratkaisu? Kultakutri saattaisi olla melko tyytyväinen, jos riittävän hyvä on lähes yhtä tyydyttävää kuin täydellinen puuro, mutta jos riittävän hyvä on vain puoliksi yhtä hyvä kuin täydellinen lautasellinen, hän saattaisi pettyä, jos tietäisi saatavilla olleen paljon parempaakin puuroa. Tämä on perustavanlaatuinen ongelma heuristiikkojen suorituskyvyn arvioinnissa.

Glover [19] maalaa kuvan siitä, miten eksaktien algoritmien ajatellaan kehittyvän akatemian korkeiden tornien analyyttisessä puhtaudessa, kun taas heuristiikat syntyvät työelämän pimeissä tyrmissä konkreettiseen tarpeeseen. Tämän tutkielman tavoitteena on kaventaa kuilua heuristiikkojen ja eksaktien algoritmien välillä soveltamalla ja laajentamalla kirjallisuudessa esitettyjä robusteja heuristiikkojen arviointimenetelmiä uusiin ongelmatyyppeihin. Toissijaisena tavoitteena on myös toteuttaa ratkaisinohjelma suunnistusongelmalle ja mitata sen suorituskykyä esitetyillä arviointimenetelmillä.

Työ etenee seuraavasti: luvussa 2 tarkastellaan kirjallisuudessa esitettyjä heuristiikkojen arviointimenetelmiä, esitellään suunnistusongelman matemaattinen määritelmä ja johdetaan ongelmageneraattori, joka tuottaa etukäteen määritellyn optimin sisältäviä satunnaisia suunnistusongelmia. Arviointimenetelmien hyviä ja huonoja puolia tarkastellaan teoreettisella tasolla ja valitaan myöhempään empiiriseen tarkasteluun sekä enemmän että vähemmän teoreettisia takuita antavia menetelmiä. Luvussa 3 käsitellään ongelmageneraattorin toteutusyksityiskohtia ja esitellään heuristinen ratkaisin suunnistusongelmalle. Luvussa 4 mitataan heuristisen ratkaisimen suorityskykyä testiajoilla ja tarkastellaan tulosten valossa arviointimenetelmien pätevyyttä. Viimeiseksi luvussa 5 pohditaan tulosten vaikutusta heuristiikkojen tutkimukseen ja menetelmien laajentamismahdollisuuksia jatkotutkimuksen kannalta.

2 Teoria

Rardin ja Uszoy [40] esittävät, että optimointiongelmien tutkimus käy läpi tiettyjä tunnistettavissa olevia yleisiä vaiheita. Jos tietyn ongelman ratkaisemiseen ei ole aiemmin esitetty ainuttakaan algoritmia, ensimmäisen toimivan heuristiikan löytäminen voi olla merkittävä saavutus. Tälle alkuvaiheelle tyypillistä on, että algoritmeja testataan verrattain pienikokoisilla ja usein satunnaisesti generoiduilla testiongelmilla. Mittaamisen ja vertailun tarve lisääntyy vasta myöhemmin, kun useita vaihtoehtoisia menetelmiä on jo esitetty, eikä yhden uuden algoritmin esittäminen ole enää riittävä tieteellinen kontribuutio alalle. Tällöin testiongelmista kehitetään standardoituja testipankkeja, jotta algoritmeja voitaisiin paremmin verrata keskenään ja osoittaa uuden menetelmän paremmuus vanhoihin nähden.

Testipankkien käyttöön liittyy useita ongelmia. Valitut ongelmat saattavat olla liian helppoja joillekin algoritmeille, esimerkiksi jos niissä on matemaattista rakennetta, joka helpottaa ratkaisua. Toisaalta pankeissa voi olla liiankin vaikeita patologisia tapauksia, joita esiintyy vain harvoin oikeassa maailmassa [40]. Pankkien käyttöön liittyvät seikat, kuten algoritmien alkuarvovalinnat, voivat myös haitata kokeellisten vertailujen tulkintaa [5].

Pankkien koostumukseen ja käyttöön liittyvien ongelmien ohella on edelleen johdantoluvussa mainittu perustavanlaatuisempi ongelma: jos testiongelmien todellista optimia ei tiedetä, ei ole mitään vertailukohtaa heuristiikan tuottamille ratkaisuille. Testiongelma voidaan toki yrittää ratkaista eksaktilla algoritmilla, mutta se ei usein ole mahdollista täsmälleen niille suurikokoisille ongelmille, joihin heuristiikkoja tarvitaan eniten. Vastaavasti pienikokoisille ongelmille tarkoitettujen heuristiikkojen testaamisessa eksakteista algoritmeista voi olla hyötyä [40].

Eksaktin ratkaisun sijaan toiseksi paras vaihtoehto voisi olla verrata heuristista ratkaisua teoreettiseen ala- tai ylärajaan [40], jos sellainen on mahdollista määrittää nopeammin kuin ongelman todellinen optimi. Joillekin paljon tutkituille ongelmille, esimerkiksi *kauppamatkustajan ongelmalle* (*traveling salesman problem*), onkin saatavilla riittävän tiukkoja alarajoja, jotka voivat päästä keskimäärin 0,1% päähän ja pahimmassakin testitapauksessa n. 3% päähän todellisesta optimista tai parhaasta mahdollisesta ratkaisusta TSPLIB-testipankissa, jonka testiongelmissa on kymmeniä tuhansia pisteitä [22]. Teoreettisten rajojen määrittäminen voi kuitenkin olla yhtä vaikea tai vaikeampi tehtävä kuin itse heuristiikan toteutus: suunnistusongelmalle esitetyt ylärajat ovat 3,8%-6,3% päässä parhaasta mahdollisesta ratkaisusta testipankissa, jonka sisältämissä ongelmissa on vain 21-32 pistettä [29]. Teoreettisissa ala- ja ylärajoissa on sama ongelma kuin heuristisissa ratkaisuissakin: ne vaativat empiirisen tarkastelun, jotta voidaan osoittaa teoreettisen rajan sijaitsevan lähellä

todellista optimia; ongelma ei ratkea, vaan ainoastaan siirtyy heuristiikan tasolta teoreettisten rajojen tasolle.

Kolmas tutkimussuunta on tilastollinen optimin estimointi, jolla on pitkä historia 1970-luvulta alkaen [18], vaikkakin se on jäänyt 1990-luvun jälkeen vähemmälle huomiolle [9]. Tilastollinen lähestymistapa perustuu oletukseen, että heuristiikan yksittäisen ajon tuottamaa ratkaisua voidaan ajatella ikään kuin otosminiminä- tai maksimina heuristiikan läpi käymien ratkaisujen joukosta (populaatiosta). Jos heuristiikka ylläpitää enemmän kuin yhtä keskeneräistä ratkaisua kerrallaan, kuten on tyypillistä esimerkiksi evoluutioalgoritmeille, tämä joukko voidaan samaistaa vastaavasti joukoksi otosminimejä- tai maksimeja [18]. Tilastotieteessä otosminimien ja maksimien analyysi kuuluu ääriarvoteorian alle, jossa ääriarvojen rajajakaumaa tarkastelivat ensimmäisinä 1920-luvulla R. A. Fisher ja L. H. C. Tippett [16]. Optimointikirjallisuudessa Fisher-Tippett-tulosta hyödyntänyt tutkimussuunta kulminoitui 1970-luvun lopulla Goldenin ja Altin julkaisuun, jossa sitä sovelletaan kauppamatkustajan ongelmaan [20].

Tilastollisessa optimin estimoinnissa on käytetty myös *katkaisupistemenetelmää*, joka vaatii vähemmän oletuksia kuin ääriarvoteoreettinen lähestymistapa. Katkaisupiste tarkoittaa pistettä satunnaismuuttujan jakaumalla, jonka jälkeen ei voi enää seurata suurempia tai pienempiä arvoja. Robson ja Whitlock [43] julkaisivat vuonna 1964 *linkkuveitsimenetelmän* (*jackknife, lyh. JK*) jatkuvan jakauman katkaisupisteen määrittämiseksi. Dannenbring sovelsi tulosta myöhemmin esittäen joukon optimiestimaattoreita [11]. Tässäkin lähestymistavassa heuristiset ratkaisut ovat otosminimejä-/maksimeja mahdollisten ratkaisujen joukosta, jonka ajatellaan samaistuvan likimääräisesti jatkuvaan jakaumaan; katkaisupiste vastaa optimia. Carling ja Xiangli [9] ovat myöhemmin jatkaneet katkaisupistemenetelmän empiiristä tutkimusta.

Dannenbring [11] suosittelee käytettäväksi pääasiallisesti satunnaista kokeilua ja 1. asteen linkkuveitsiestimaattoria $\hat{\theta}_{JK}^{(1)} = 2\tilde{x}_{(1)} - \tilde{x}_{(2)}$, jossa $\tilde{x}_{(i)}$:t ovat suuruusjärjestykseeen asetetut satunnaiskokeilun tuottamat ratkaisut eli otosminimit- tai maksimit. Järjestys on nouseva minimointiongelman tapauksessa ja laskeva maksimointiongelman tapauksessa. Carling ja Xiangli [9] esittävät, että voidaan käyttää myös voimakkaan heuristiikan tuottamia ratkaisuja satunnaisen kokeilun sijaan, kunhan heuristiset ratkaisut ovat jakauman hännässä riittävän lähellä optimia: kriteeriksi tähän he esittävät skaalattua keskihajontaa $SR = 1000\sigma\left(\tilde{x}_i\right)/\hat{\theta}_{JK}^{(1)}$. Carlingin ja Xianglin [9] mukaan voidaan myös määrittää luottamusväli arvioidulle optimille *bootstrap-menetelmällä*. Bootstrap-menetelmässä simuloidaan otantaa otosminimien-/maksimien populaatiosta tekemällä alkuperäisestä otoksesta uusi satunnaisotanta takaisinpanolla (sama alkuperäisen otoksen alkio voidaan valita useita kertoja) [12].

Sekä katkaisupistemenetelmä että ääriarvoteoriaan nojaava lähestymistapa olettavat, että heuristiikan läpikäymät pseudo-otokset ovat toisistaan riippumattomia. Parhaiten tämä toteutuu kokeilemalla täysin satunnaisesti ratkaisuja. On osoitettu [35], että satunnaista kokeilua vahvempi heuristiikka aiheuttaa harhaa tuloksiin. Tästä huolimatta tilastollisessa lähestymistavassa lupaavaa on sen yleisyys, sillä samaa menetelmää voisi soveltaa useaan erilaiseen heuristiikkaan ja ongelmaan.

Viimeinen tutkimussuunta on ollut rakentaa ongelmageneraattoreita, jotka tuottavat ennalta tiedetyn optimin sisältäviä testiongelmia. Naiivi satunnaisgenerointi on melko helppoa, mutta vaikeampaa on tuottaa ongelmainstansseja siten, että generoinnin lopussa tunnetaan kunkin instanssin optimi [40]. Onnistuessaan satunnaisgenerointi antaisi kuitenkin vahvimmat teoreettiset takuut suurikokoisille ongelmille, joille ei ole olemassa tiukkoja teoreettisia rajoja eikä eksakti ratkaisualgoritmi anna realistisessa ajassa ratkaisua. Kirjallisuudesta ei löytynyt ennestään suunnistusongelmalle sopivaa generaattoria; relevanteimmat esitetyt generaattorit ovat Pilcherin ja Rardinin [38] sekä Arthurin ja Frendeweyn [2] generaattorit kauppamatkustajan ongelmalle. Arthur-Frendewey-generaattori on näistä huomattavasti yksinkertaisempi, joten myöhemmin käsitellään vastaavan kaltaista generaattoria suunnistusongelmalle.

2.1 Suunnistusongelma (OP)

Muodollisesti suunnistusongelma määritellään graafilla $G = (V, A)$, jossa $V = \{0, 1, 2, \ldots, n\}$ on joukko solmuja, joista $\{0\}$ on alkuterminaali ja $\{n\}$ on lopputerminaali. Useimmiten alkuterminaali ja lopputerminaali vastaavat samaa kohdetta, eli palataan lähtöpisteeseen. Jokaisella solmulla i on pistemäärä p_i, esimerkiksi turistikohteen viihdyttävyys tai kohteesta kerättävän rahan odotusarvo. A on joukko kaaria (i, j), joista jokaisella on kustannus c_{ij}, esimerkiksi matka-aika tai polttoaineen määrä. Tavoitteena on löytää pistemäärän p_i maksimoiva polku, jonka kustannukset eivät ylitä budjettia $C_{\max}$. Päätösmuuttuja x_{ij} saa arvon 1, jos ratkaisussa kuljetaan solmusta i solmuun j; muuten se saa arvon 0. Tavoitefunktio (2.1) ja ehdot (2.2)–(2.9) riittävät määrittelemään suunnistusongelman lineaarisena optimointitehtävänä:

$$\max \sum_{i=0}^{n-1} \sum_{j=1}^{n} p_i x_{ij} \tag{2.1}$$

siten, että

$$\sum_{i=1}^{n} x_{0i} = 1 \tag{2.2}$$

$$\sum_{i=0}^{n-1} x_{in} = 1 \tag{2.3}$$

$$\sum_{i=0}^{n-1} x_{ij} \leq 1, \qquad j = 1, 2, \ldots, n-1 \tag{2.4}$$

$$\sum_{j=0}^{n} x_{ij} \leq 1, \qquad i = 1, 2, \ldots, n-1 \tag{2.5}$$

$$\sum_{i=0}^{n-1} x_{ij} = \sum_{i=1}^{n-1} x_{ji}, \qquad j = 1, 2, \ldots, n-1 \tag{2.6}$$

$$\sum_{i=0}^{n-1} \sum_{j=1}^{n} c_{ij} x_{ij} \leq C_{\max} \tag{2.7}$$

$$+ \text{ Osapolkujen poistamisrajoitukset} \tag{2.8}$$

$$x_{ij} \in \{0, 1\} \tag{2.9}$$

Ehdot (2.2) ja (2.3) takaavat, että polku alkaa ja päättyy terminaaliin. Ehdot (2.4) ja (2.5) takaavat, että jokaiseen solmuun tulee ja jokaisesta solmusta lähtee korkeintaan yksi kaari. Ehto (2.6) takaa, että jokaiseen solmuun tulee yhtä monta kaarta kuin siitä lähtee, 0 tai 1 ehtojen (2.4) ja (2.5) nojalla. Ehto (2.7) takaa, että kustannusten summa ei ylitä maksimibudjettia. Osapolkujen poistamisrajoitukset (2.8) jätetään implisiittisiksi, koska rajoitteiden määrä on suuri ja ne voi muodostaa usealla eri tavalla; kaikkien tapojen tarkoituksena on varmistaa, että ratkaisu muodostaa yhtenäisen polun. Myöhemmin esiteltävä ongelmageneraattori ei myöskään vaadi osapolkujen poistamisrajoitusten käsittelyä. Viimeinen ehto (2.9) rajaa päätösmuuttujan vain kokonaislukuarvoihin 0 tai 1.

2.2 Arthur-Frendewey-generaattori

Arthur ja Frendewey [2] esittivät kauppamatkustajan ongelmalle generaattorin, joka tuottaa ennalta määrätyn optimireitin sisältäviä ongelmia. Alkuperäisessä generaattorissa ainoastaan optimireitin solmujen (kaupunkien) järjestys on ennalta määrätty: reitin pituus saa vaihdella satunnaisesti. Pienillä muutoksilla Arthur-Frendewey-generaattorin saa kuitenkin tuottamaan ongelmia, joissa myös optimireitin solmujen väliset etäisyydet saavat halutut ennalta määrätyt arvot. Tästä on myöhemmin hyötyä, koska muunneltua generaattoria on tarkoitus käyttää alirutiinina vastaavassa generaattorissa suunnistusongelmalle, jossa on rajoite reitin maksimipituudelle.

Aloitetaan Arthur-Frendewey-generaattorin käsittely kauppamatkustajan ongelman määritelmästä.

2.2.1 Kauppamatkustajan ongelma (TSP)

Kauppamatkustajan ongelma määritellään graafilla $G = (V, A)$, jossa $V = \{1, 2, \ldots, n\}$ on joukko solmuja ja A on joukko suunnattuja kaaria (i, j), joilla on kustannus (etäisyys) c_{ij}. Päätösmuuttuja x_{ij} saa jälleen arvon 1, jos ratkaisussa kuljetaan solmusta i solmuun j; muuten se saa arvon 0. Tavoitteena on löytää lyhin mahdollinen reitti, joka käy läpi kaikki n solmua täsmälleen kerran ja palaa lähtöpisteeseen. Tavoitefunktio (2.10) ja ehdot (2.11)–(2.14) määrittelevät kauppamatkustajan ongelman lineaarisena optimointitehtävänä:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.10}$$

siten, että

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, n \tag{2.11}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, \ldots, n \tag{2.12}$$

$$+ \text{ Osapolkujen poistamisrajoitukset} \tag{2.13}$$

$$x_{ij} \in \{0, 1\} \tag{2.14}$$

Ehdot (2.11) ja (2.12) varmistavat, että jokaisesta solmusta lähtee ja jokaiseen solmuun tulee vain yksi kaari. Osapolkujen poistamisrajoitukset (ehto (2.13)) varmistavat, että ratkaisu on *Hamiltonin polku*, eli että jokaisessa solmussa käydään täsmälleen kerran ja palataan lähtöpisteeseen. Ehto (2.14) rajaa jälleen päätösmuuttujan arvoihin 0 tai 1.

Tarkastellaan seuraavaksi erästä kauppamatkustajan ongelman relaksaatiota. Relaksaatio on alkuperäisen ongelman muunnelma, josta on poistettu ehtoja. Tavoitteena on keksiä ongelmageneraattori ensin relaksaatiolle, sitten varmistaa jollain tapaa myös poistettujen ehtojen täyttyminen. Jos $\{x_{ij}\}$ on relaksaation optimaalinen ratkaisu ja täyttää poistetut ehdot, se on myös alkuperäisen ongelman optimaalinen ratkaisu ja olemme luoneet generaattorin alkuperäiselle ongelmalle.

2.2.2 Työnjako-ongelma (AP)

Poistetaan ehto (2.13) ja löysätään ehtoa (2.14) muotoon (2.18). Tavoitefunktio (2.15) ja muut ehdot (2.16)–(2.17) säilyvät. Syntyy niin kutsuttu *työnjako-ongelma* (*assignment problem*):

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.15}$$

siten, että

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, n \tag{2.16}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, \ldots, n \tag{2.17}$$

$$x_{ij} \geq 0, \qquad i, j = 1, \ldots, n \tag{2.18}$$

Lineaariselle optimointitehtävälle voidaan muodostaa *duaalitehtävä* noudattamalla yksinkertaista menetelmää [34]; tällöin alkuperäistä tehtävää sanotaan *primaalitehtäväksi*. Primaalitehtävä ja duaalitehtävä ovat ikään kuin toistensa peilikuvia: minimointiongelma ja maksimointiongelma, joiden optimiratkaisut kohtaavat toisensa samassa tavoitefunktion arvossa. Duaalitehtävässä primaalitehtävän muuttujista tulee ehtoja, ehdoista tulee muuttujia ja tavoitefunktion suunta muuttuu (tässä tapauksessa minimoinnista maksimoinniksi). Työnjako-ongelman duaalitehtävä on yleisesti tunnettu [2, 3, 10], mutta yritetään kuitenkin seuraavaksi havainnollistaa, miten sen voisi johtaa työnjako-ongelman määritelmästä.

2.2.3 Työnjako-ongelman duaali (DAP)

Menetelmä [34] on määritelty maksimointitehtäville, joten muutetaan tavoitefunktio (2.15) minimoinnista maksimointiin seuraavasti (seuraavan maksimointitehtävän duaalitehtävä on tällöin minimointimuodossa):

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} -c_{ij} x_{ij} \tag{2.19}$$

Vaihtoehtoisesti voitaisiin käyttää alkuperäistä tavoitefunktiota (2.15) ja noudattaa menetelmää päinvastaiseen suuntaan, jolloin lopputuloksena olisi suoraan maksimointitehtävä. Kummassakin tapauksessa huomataan, että ehdot (2.16)–(2.18) muodostavat itse asiassa joukon ehtoja, jokaiselle j:n arvolle ja jokaiselle i:n arvolle. Määritellään ensin primaalitehtävän ehtojoukkoja (2.16)–(2.17) vastaavat du-

aalimuuttujat α_i ja β_j. Menetelmän mukaan duaalitehtävää muodostettaessa ei-negatiivisuusrajoitteille ((2.18)) ei määritellä duaalimuuttujia. Siirretään menetelmän mukaisesti tavoitefunktiosta (2.19) kerroin $-c_{ij}$ primaalitehtävän ehtoa (2.18) vastaavan duaalitehtävän ehdon (2.20) oikealle puolelle ja vasemmalle puolelle duaalimuuttujat kertoimineen (kertoimet saavat arvon 1):

$$\alpha_i + \beta_j \geq -c_{ij}, \qquad i, j = 1, \ldots, n \tag{2.20}$$

Ehdon (2.20) merkit vaihtuvat myöhemmin, kun tehtävä muunnetaan minimointitehtävästä maksimointitehtäväksi. Koska primaalitehtävässä kaavan mukaan huomioitavat ehdot (2.16)–(2.17) ovat yhtäsuuruusehtoja, niitä vastaavat duaalitehtävän ei-negatiivisuus-, ei-positiivisuus- tai reaalilukurajoitteet typistyvät duaalitehtävän ehdossa (2.23) muotoon $\alpha_i \in \mathbb{R}$ ja $\beta_j \in \mathbb{R}$. Primaalitehtävän ehtojoukkojen (2.17) ja (2.16) oikealta puolelta siirretään menetelmän mukaisesti kertoimet ja niitä vastaavat duaalimuuttujat duaalitehtävän tavoitefunktioon (2.21); jokaisen duaalimuuttujan kertoimen arvo on jälleen 1, joten seuraa summa $\sum_{i=1}^{n} \alpha_i + \sum_{j=1}^{n} \beta_j$. Johdettu duaalitehtävä on minimointimuodossa, mutta sen voi muuntaa maksimointimuotoon, jolloin seuraa halutun muotoinen duaalitehtävä:

$$\max(\sum_{i=1}^{n} \alpha_i + \sum_{j=1}^{n} \beta_j) \tag{2.21}$$

siten, että

$$\alpha_i + \beta_j \leq c_{ij}, \qquad i, j = 1, \ldots, n \tag{2.22}$$

$$\alpha_i, \beta_j \in \mathbb{R} \tag{2.23}$$

Lineaariohjelmoinnin duaalilauseen nojalla [2, 34] $\{x_{ij}\}$ ja $\{\alpha_i, \beta_j\}$ ovat optimaalisia ratkaisuja vastaaviin tehtäviinsä, jos $\{x_{ij}\}$ on käypä ratkaisu luvun 2.2.2 primaalitehtävään AP ja $\{\alpha_i, \beta_j\}$ on käypä ratkaisu luvun 2.2.3 duaalitehtävään DAP siten, että niin kutsutut *complementary slackness*-ehdot toteutuvat kaikille i ja j. Complementary slackness-ehtojen tulee varmistaa joko, että $x_{ij} = 0$, tai muussa tapauksessa varmistaa, että kaikki primaali- tai duaalitehtävän ei-negatiivisuusehdot toteutuvat täsmälleen niiden sallimissa yhtäsuuruustapauksissa ($= 0$) [34]. Koska ehto (2.22) ei ole ei-negatiivisuusmuodossa, muokataan sen epäyhtälöä siirtämällä c_{ij} vasemmalle puolelle ja kertomalla puolittain luvulla -1, jolloin seuraa ei-negatiivisuusehto $c_{ij} - \alpha_i - \beta_j \geq 0$, josta muokataan edelleen complementary slackness-ehto (2.24) merkitsemällä puolet yhtä suuriksi ja huomioimalla samassa

ehdossa molemmat päätösmuuttujan x_{ij} tapaukset:

$$(c_{ij} - \alpha_i - \beta_j)\, x_{ij} = 0 \tag{2.24}$$

Jos päätösmuuttuja x_{ij} saa arvon 0, ehto (2.24) toteutuu triviaalisti; muuten ehto (2.24) varmistaa ehdon (2.22) toteutumisen yhtäsuuruudessa.

Tarkastellaan seuraavaksi Arthurin ja Frendeweyn kauppamatkustajan ongelmalle kehittämää generointialgoritmia [2]. Algoritmi hyödyntää luvun 2.2.1 kauppamatkustajan ongelman tehtävää TSP ja sen relaksaatiota, luvun 2.2.3 duaalitehtävää DAP.

2.2.4 Arthurin ja Frendeweyn generointialgoritmi

Olkoon n generoitavan ongelman solmujen määrä ja R suurin sallittu etäisyys solmujen välillä. Generoidaan ensin satunnainen järjestys, jossa optimiratkaisun kaikki solmut käydään läpi, toisin sanoen kokonaislukujen $\{1, \ldots, n\}$ satunnainen permutaatio $\{i_1, \ldots, i_n\}$. Generoitua permutaatiota vastaava optimiratkaisu $\{x_{ij}\}$ täyttää osapolkujen poistamisrajoitukset (2.13), kunhan määrittelemme lisäksi $i_{n+1} = i_1$, jolloin haluttu optimiratkaisu käy kaikki solmut läpi ja palaa lähtöpisteeseen. Ainoastaan permutaation peräkkäisiä solmuja vastaavat kaaret kuuluvat optimireittiin, eli $x_{i_k i_{k+1}} = 1, \;\; k = 1, \ldots, n$ ja muut kaaret saavat arvon 0. Tapauksissa $x_{ij} = 0$ complementary slackness-ehto (2.24) toteutuu triviaalisti, joten riittää käsitellä tapaukset $x_{ij} = 1$, joilta vaaditaan:

$$\alpha_{i_k} + \beta_{i_{k+1}} = c_{i_k i_{k+1}}\,, \qquad k = 1, \ldots, n \tag{2.25}$$

Alkuperäisessä Arthurin ja Frendeweyn [2] menetelmässä generoidaan satunnaisesti duaalimuuttujien arvot α_i ja β_j. Seuraavaksi täytetään yhtälön (2.25) mukaan kustannus-/etäisyysmatriisin arvot c_{ij}, jotka vastaavat generoidun permutaation peräkkäisiä solmuja. Jotta arvot c_{ij} päätyvät sallitulle välille $[1, R]$, täytyy rajata α_i ja β_j välille $[\rho R, \sigma R]$, jossa $0 \leq \rho \leq \sigma \leq 0.5$. Lopuille tapauksille $x_{ij} = 0$ ainoa tarkistettava ehto on (2.22), jonka täyttymiseksi riittää, että loput kustannusmatriisin arvot c_{ij} generoidaan satunnaisesti välille $[\alpha_i + \beta_j, R]$. Edellä mainitulla menetelmällä kustannusmatriisista tulee epäsymmetrinen, mutta symmetrisen matriisin luomiseksi riittää asettaa arvot α_i ja β_i yhtä suuriksi [2]; käytännössä symmetrisessä tapauksessa voimme siis ylläpitää vain α-arvoja, koska $\alpha_i = \beta_i$.

Lopputuloksena on kustannusmatriisi ja sen optimaalisen ratkaisun polku $\{i_1, \ldots, i_n\}$. Jos poiketen alkuperäisestä Arthur-Frendewey-generaattorista tarvitaan lisäksi symmetrinen kustannusmatriisi ($\alpha_i = \beta_i$), jossa optimaalisen ratkaisun kaaret saavat ennalta määritellyt kustannukset, yhtälöstä (2.25) β:n ja $c_{i_k i_{k+1}}$:n

korvaamalla sekä yksinkertaistamalla seuraa:

$$\alpha_{k+1} = c_k - \alpha_k\,, \qquad k = 1, \ldots, n \tag{2.26}$$

jossa α_k on optimiratkaisun k:tta kaarta vastaava duaalimuuttuja ja c_k kyseisen kaaren haluttu kustannus (indeksi $k = n+1$ saa jälleen määritelmällisesti arvon 1). Löytämällä oikea alkuarvo muuttujalle α_1 saadaan yksi kerrallaan generoitua sarja duaalimuuttujia, jotka vastaavat haluttuja kustannuksia. Alkuarvon määrittämiseksi voidaan asettaa $\alpha_1 = 0$ ja käydä duaalimuuttujien laskentaprosessi kertaalleen läpi, jolloin saadaan α_1:n ja α_2:n implikoima kustannus $\alpha_1+\alpha_2$, joka eroaa halutusta arvosta c_1. Laskemalla halutun arvon ja implikoidun arvon välinen erotus eli virhekertymä $c_{err} = c_1 - (\alpha_1 + \alpha_2)$ saadaan generoinnin aikana kerääntynyt poikkeama halutusta arvosta c_1. Koska arvo α_1 asetetaan kumuloituvassa generointiprosessissa kaksi kertaa, alkuarvon asettamisen aikana ja kun $k = n$, saadaan toivottu kustannus c_1 aikaan asettamalla uudeksi alkuarvoksi $\alpha_{1_{uusi}} = \alpha_{1_{vanha}} + c_{err}/2$, jolloin puolet virhekertymästä korjaantuu α_0:n alkuarvoa asetettaessa ja puolet korjaantuu viimeisellä iteraatiolla $k = n$. Virhekertymän korjaus voidaan toteuttaa esimerkiksi seuraavalla Python-ohjelmalla:

```
1  alfa[0] = 0.0

3  for k in range(0, n):
       alfa[(k+1)%n] = halutut_kustannukset[k] - alfa[k]
5
   alfa0_implikoitu_kustannus = alfa[0] + alfa[1]
7  kertynyt_poikkeama = halutut_kustannukset[0] - alfa0_implikoitu_kustannus

9  alfa[0] = alfa[0] + kertynyt_poikkeama / 2.0

11 for k in range(0, n):
       alfa[(k+1)%n] = halutut_kustannukset[k] - alfa[k]
```

Ohjelma 2.1 *Koodiesimerkki duaalimuuttujan α arvojen määrityksestä.*

Edelleen on kuitenkin vaarana, että yhtälön (2.26) erotus $c_k - \alpha_k$ saa pienemmän arvon kuin ρR, jolloin duaalimuuttuja α_{k+1} ei enää sijoitu vaaditulle välille $[\rho R, \sigma R]$. Eräs keino tämän välttämiseksi on järjestää arvot c_k nousevaan suuruusjärjestykseen, jolloin peräkkäisten arvojen erotus on aina positiivinen ja kertymä seuraavalle iteraatiolle kasvaa, mutta tällöin optimiratkaisuun tulee epätoivottua säännöllisyyttä. Nousevan järjestyksen rikkomiseksi voidaan käyttää sekoitusmenetelmää, joka vaihtaa kahden satunnaisen kustannusarvon c_{rnd1} ja c_{rnd2} paikkaa optimiratkaisun polulla, laskee α-arvot ja tarkistaa alarajan ρR täyttymisen. Jos vaihdoksen takia yksikään α-arvo rikkoo alarajaa ρR, vaihdos perutaan ja yritetään uudelleen, kunnes riittäväksi katsottu määrä vaihdoksia on saavutettu tai vaihdoksessa on epä-

onnistuttu liian monta kertaa. Edellä mainitut menetelmät soveltuvat tapauksiin, joissa edellytetään, että optimiratkaisu sisältää jokaisen toivotun kustannusarvon c_k kaaren, mutta niiden järjestyksellä ei ole väliä. Juuri tällaista generaattoria kauppamatkustajan ongelmalle voidaan hyödyntää osana laajempaa generaattoria suunnistusongelmalle.

2.3 Ongelmageneraattori suunnistusongelmalle

Olennainen ero luvun 2.1 suunnistusongelman (OP) ja luvun 2.2.1 kauppamatkustajan ongelman (TSP) välillä on, että suunnistusongelmassa ratkaisuun valitaan osajoukko kaikista solmuista, kun taas kauppamatkustajan ongelmassa ratkaisun tulee käydä jokaisessa solmussa. Tämä viitaa mahdolliseen generointitapaan: generoidaan ensin luvun 2.2.4 mukaisesti kauppamatkustajan ongelman kustannusmatriisi, joka sisältää kaikki suunnistusongelman optimiratkaisuun kuuluvat *valitut solmut*, sitten lisätään loput optimiratkaisuun kuulumattomat *ei-valitut solmut* kustannusmatriisiin siten, että niiden lisääminen ei muuta suunnistusongelman optimia. Kysymykseksi nousee, miten taata, ettei lisääminen vaikuta optimiin; seuraava *0/1-reppuongelman* (*0/1 knapsack problem*) käsittely tarjoaa erään vastauksen.

2.3.1 0/1-reppuongelma (KP)

Kuvitellaan reppureissaaja, jonka täytyy päättää mitkä tavarat pakata mukaan matkalle n:n tavaran joukosta. Jokaisella tavaralla on paino w_i ja arvo v_i; reppuun valittujen tavaroiden paino ei saa ylittää toivottua painorajaa W. Jos tavoitteena on maksimoida mukaan otettavien tavaroiden arvo, kyseessä on 0/1-reppuongelma, jonka voi muotoilla lineaarisena optimointitehtävänä:

$$\max \sum_{i=1}^{n} v_i x_i \tag{2.27}$$

siten, että

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2.28}$$

$$x_i \in \{0, 1\} \tag{2.29}$$

Reppuongelman ja suunnistusongelman välillä on tiettyjä samankaltaisuuksia, jotka sallivat rinnastusten tekemisen ongelmien välillä, vaikkakin reppuongelma ei ole suoraviivaisesti suunnistusongelman relaksaatio. Erityisesti voimme rinnastaa reppuongelman jokaisen tavaran i suunnistusongelman jokaiseen solmuun i, ja määritellä tavaran arvon v_i ja solmun pistemäärän p_i yhtä suuriksi. Määritellään lisäksi

repun painoraja W yhtä suureksi reitin maksimipituuden C_{max} kanssa. Eroavaisuudeksi jää, että reppuun valikoidaan tavaroita/solmuja, kun taas reitille valikoidaan kaaria solmujen/tavaroiden välillä; lisäksi suunnistusongelman ratkaisun täytyy olla polku, joka noudattaa osapolkujen poistamisrajoituksia.

Nyt jokaista reppuongelman tavaraa i vastaa suunnistusongelman solmu i, jolla on vastaavassa suunnistusongelmassa useita mahdollisia kustannuksia c_{ij}, kun taas reppuongelmassa tavaralla on vain yksi paino/kustannus w_i. Voimme kytkeä tavaran painon w_i ja kaaren kustannuksen c_{ij} määrittelemällä, että jokaisen kaaren, joka lähtee solmusta i (ja päättyy mihin tahansa solmuun j), kustannus on vähintään w_i. Tällöin reppuongelman painot asettavat alarajan suunnistusongelman kustannuksille.

Huomioidaan lisäksi, että reppuongelman optimi $\{x_i\}$ ei muutu, vaikka tehtävään lisättäisiin tavaroille vaihtoehtoja suuremmilla painoilla (määritellen, että vain yksi vaihtoehto tavarasta voi tulla valituksi). Ei-valittujen tavaroiden ($x_i = 0$) osalta uusi suuremman painon vaihtoehto ei voi kuulua optimireppuun, jos saman arvoinen tavara ei olisi kuulunut sinne pienemmälläkään painolla. Optimi $\{x_i\}$ ei myöskään muutu, jos valituista tavaroista ($x_i = 1$) lisättäisiin vaihtoehtoja suuremmilla painoilla, jos vain yksi vaihtoehto voi päätyä ratkaisuun. Kaikissa tapauksissa, joissa on useita painovaihtoehtoja samalle tavaralle ja vaihtoehdoista voidaan valita vain yksi, valinnalla ei joko ole väliä tai kannattaa valita pienimmän painoinen vaihtoehto, jotta reppuun jää enemmän tilaa muille tavaroille. Toisin sanoen kaikki reppuongelman tavaran i painon w_i asettamaa alarajaa kalliimmat suunnistusongelman kaarivaihtoehdot c_{ij} voivat ainoastaan huonontaa ratkaisua verrattuna tapaukseen $w_i = c_{ij}$, koska solmu i ei voi kuulua suunnistusongelman ratkaisuun, jos ratkaisuun ei ole valittu yhtään siitä lähtevää kaarta (josta seuraa kustannus c_{ij}).

Seuraa, että reppuongelman ja suunnistusongelman optimit vastaavat toisiaan, jos optimin $\{x_i\}$ valittujen tavaroiden ja niitä vastaavien solmujen arvot/pistemäärät ja painot/kustannukset ovat yhtä suuret, ja suunnistusongelman optimiratkaisun valittujen solmujen läpi on olemassa osapolkujen poistamisrajoituksia noudattava polku, jonka kaarien kustannukset vastaavat reppuongelman tavaroiden painoja. Tällöin suunnistusongelman optimiratkaisun $\{x_{ij}\}$ yhteenlaskettu pistemäärä on yhtä suuri kuin reppuongelman optimiratkaisun $\{x_i\}$ tavaroiden yhteenlaskettu arvo.

2.3.2 Generointialgoritmi

Olkoon jälleen n generoitavien solmujen määrä ja R suurin sallittu etäisyys solmujen välillä. Olettaen, että käytössä on eksakti ratkaisualgoritmi reppuongelmalle, joka pystyy ratkaisemaan n:n tavaran ongelman, on mahdollista luoda määrätyn optimin sisältävä suunnistusongelma yhdistämällä Arthur-Frendewey-generaattori

2.2.4 ja reppuongelman 2.3.1 optimiratkaisu. Tällöin muunneltu Arthur-Frendeweygeneraattori takaa halutun ratkaisun olemassaolon ja reppuongelman optimiratkaisun antamat kaarikustannusten alarajat takaavat, että haluttuun ratkaisuun valittu solmujoukko on optimaalinen.

0/1-reppuongelman ratkaisemiseksi on olemassa yksinkertainen taulukointialgoritmi, jossa ongelma ratkaistaan rekursiivisesti hyödyntäen aiempia välituloksia. Menetelmää kutsutaan myös *dynaamiseksi ohjelmoinniksi* ja sen suoritusaika on luokkaa $O(nc)$, jossa n on tavaroiden määrä ja c on repun kapasiteetti [24]. Taulukointi-algoritmi on riittävän nopea tähän tutkielmaan; suurimmat generoitavat ongelmat tulevat olemaan kokoluokkaa $n = 50000$.

Ennen generointia määritellään Arthurin ja Frendeweyn [2] suositusten mukaisesti alkuarvot $\rho = 0.1$ ja $\sigma = 0.25$. Lisäksi määritellään maksimikustannus $w_{max} = c_{max} = R = 100$, maksimiarvo $v_{max} = 1000$ ja pienin mahdollinen α-arvo $\alpha_{min} = 0.05 * c_{max}$. Valitaan myös halutun ongelman koko n ja repun kapasiteetti/reitin maksimikustannus $W = C_{max} = pnc_{max}$, jossa p kertoo kuinka suuri osuus kaikista solmuista n mahtuu optimireitille. Generointialgoritmi etenee seuraavasti:

1. Satunnaisgeneroidaan reppuongelman tavaroiden arvot v tasaisesti väliltä $[1, v_{max}]$ ja tavaroiden painot w tasaisesti väliltä $[2\rho R, 2\sigma R]$. Tavaroiden painoille käytetään kerrointa 2, koska paino/kaarikustannus tulee olemaan kahden α-arvon summa ja yksittäinen α-arvo kuului rajata välille $[\rho R, \sigma R]$.

2. Ratkaistaan reppuongelma taulukointi-algoritmilla. Lopputuloksena on valittujen tavaroiden järjestetty joukko $\{x_s\}$ ja sitä vastaava jäljelle jäävien eivalittujen tavaroiden järjestetty joukko $\{x_u\}$.

3. Järjestetään tavarat nousevaan kustannusjärjestykseen joukossa $\{x_s\}$ ja satunnaiseen järjestykseen joukossa $\{x_u\}$.

4. Generoidaan muunnellulla Arthur-Frendewey-generaattorilla (luku 2.2.4) valittuja tavaroita $\{x_s\}$ vastaava kauppamatkustajan ongelma. Kustannusten nousevan järjestyksen rikkomiseksi sovelletaan sekoitusmenetelmää, jossa vaihdos perutaan, jos se aiheuttaa pienemmän α-arvon kuin α_{min}.

5. Lisätään generoidun kauppamatkustajan ongelman kustannusmatriisiin eivalitut tavarat $\{x_u\}$ siten, että valitun tai ei-valitun tavaran/solmun i ja eivalitun tavaran/solmun j välinen kustannus c_{ij} on vähintään $max(w_i, w_j)$. Kahden valitun tavaran/solmun välinen kustannus pysyy samana. Lopputuloksena on suunnistusongelman kustannusmatriisi. Tavaroiden arvot v vastaavat suunnistusongelman pistemääriä p.

Jotta ei-valituista solmuista lähtevien kaarien kustannukset eivät kasvaisi keinotekoisen suuriksi verrrattuna valittujen solmujen välisiin kaariin, voidaan vaiheessa 5 generoida kustannukset *kolmiojakaumalta*, jossa todennäköisyys on suurin kustannusarvolle $max(w_i, w_j)$ laskien kustannusarvojen maksimia R kohti.

2.4 Heuristiikka suunnistusongelmalle

Suunnistusongelman ratkaisemiseen on esitetty monia heuristiikkoja sekä jonkin verran eksakteja algoritmeja. Tsiligirides [46] esitti kenties varhaisimman heuristiikan; myöhemmin suunnistusongelmaan on sovellettu esimerkiksi tabuhakua [17, 44], evoluutioalgoritmeja [45, 14, 33, 36], GRASP-menetelmää [8, 25] ja simuloitua jäähdytystä [31]. Suunnistusongelmalle esitetyt eksaktit algoritmit ovat yleisesti *branch-and-cut*-tyyppiä ja niillä on mahdollista löytää todellinen optimi, kun ongelmakoko on maksimissaan tuhansien, ei kymmenien tai satojen tuhansien solmujen luokkaa [15, 27]. Yli 400 solmun ongelmissa paras tunnettu eksakti algoritmi ([27]) vaatii yleensä useita tunteja laskenta-aikaa löytääkseen ratkaisulle tiukat ala- ja ylärajat, eikä optimin saavuttaminen ole varmaa; heuristinen ratkaisin voi löytää lähes yhtä hyvän ratkaisun alle minuutissa [27].

Tarkastellaan yleisten *metaheurististen* lähestymistapojen kuten tabuhaun tai evoluutioalgoritmien sijaan varta vasten suunnistusongelmalle kehitettyjä heuristiikkoja. Toivona on, että ongelmaspesifi heuristiikka skaalautuu paremmin suurikokoisiin ongelmiin, koska se rajaa hakuavaruutta voimakkaammin kuin tyypillinen metaheuristiikka. Gendreau ja kumpp. kommentoivat [17], että suunnistusongelmalle sopivan heuristiikan kehittäminen on vaikeaa, koska solmujen pistemäärät ja kustannukset ovat toisistaan riippumattomia, jolloin pistemäärää painottava heuristiikka ei huomioi riittävästi kustannuksia ja kustannuksia painottava heuristiikka ei huomioi riittävästi pistemääriä. Toisin sanoen on hankala löytää yleistä kaavaa, joka valitsisi kulloinkin parhaan kompromissin pistemäärien ja kustannusten välillä. Tarkastellaan kuitenkin seuraavaksi erästä kirjallisuudessa esitettyä vaihtoehtoa ja yritetään laajentaa sitä.

Butt ja Cavalier [7] esittivät suunnistusongelmalle konstruktiivisen *MAXIMP*-heuristiikan, joka rakentaa reitin solmu kerrallaan. Menetelmän ytimessä on houkuttavuusarvo W_{ij}, joka lasketaan jokaiselle suunnistusongelman solmujen parille (i, j) seuraavasti:

$$W_{ij} = \alpha \left[(p_i + p_j) / C_{max} \right] t_{ij} + (1 - \alpha) \left[(p_i + p_j) / t_{ij} \right] C_{max} \tag{2.30}$$

Yhtälössä (2.30) t_{ij} on pelkästään solmut i ja j keräävän reitin kustannus $(c_{0i} + c_{ij} + c_{jn})$, C_{max} on kustannusten maksimibudjetti ja p on solmusta kerättävä pistemäärä suunnistusongelman 2.1 määritelmän mukaisesti. Houkuttavuusarvo

W_{ij} on sitä suurempi, mitä suurempi kahden solmun pistemäärä on suhteessa niistä koituviin kustannuksiin. Houkuttavuusarvo huomioi kaksi näkökulmaa: pistemäärän suhteen kustannuksiin osuutena maksimibudjetista (t_{ij}/C_{max}) ja pistemäärän suhteen kustannuksiin (t_{ij}); α-kerroin tasapainottelee näiden kahden näkökulman välillä. Butt ja Cavalier suosittelevat arvoa $\alpha = 0.5$ [7].

Alkuperäinen MAXIMP-heuristiikka on ahne, eli reitinmuodostuksessa valitaan jokaisella askeleella paikallisesti paras solmuvaihtoehto etsimällä kaikki solmuparit (i, j), joista vain toinen sisältyy reittiin, järjestämällä ne houkuttavuusarvon W_{ij} mukaan ja lisäämällä reittiin suurimman houkuttavuusarvon parin ehdottama solmu. Heuristiikan voi muokata vähemmän ahneeksi valitsemalla listan ensimmäisen alkion sijaan satunnaisen alkion, kuitenkin painotetusti järjestetyn listan alkupäästä. Tätä *painotettua satunnaistamista* (*biased randomization*) on sovellettu moniin heuristiikkoihin mukaan lukien suunnistusongelmalle [21, 23, 37, 42]. Yleisenä tapana on, että painotukseen käytetyt indeksit generoidaan geometriselta jakaumalta [21, 37], jolloin indeksi 0 on todennäköisin vaihtoehto ja indeksin kasvaessa sen todennäköisyys laskee.

Painotettu satunnaistaminen muuttaa ahneen heuristiikan *etsintä-hyödynnyskompromissia* (*exploration vs. exploitation tradeoff*). Ahne heuristiikka keskittyy pelkästään hyödyntämään paikallisesti parasta vaihtoehtoa; painotettu satunnaistaminen sallii ajoittain myös huonomman kuin paikallisesti parhaan vaihtoehdon valitsemisen siinä toivossa, että paikallisesti huonompi valinta tuottaa pitkällä tähtäimellä globaalisti paremman lopputuloksen. Eräs painotetun satunnaistamisen etu on, että menetelmä rinnakkaistuu usealle prosessorille yksinkertaisesti vaihtamalla käytetyn satunnaislukugeneraattorin siemenlukua [21].

Siemenluvun vaihtaminen tuottaa joka kerta hieman erilaisen ratkaisun täysin riippumattomasti muista ajokerroista. Tämä ei kuitenkaan välttämättä ole tavoiteltavaa, sillä parempi lähestymistapa saattaisi olla parantaa samaa ratkaisua uudelleen ja uudelleen, tehden alussa suuria ja lopussa pieniä muutoksia. Tämä muistuttaa *simuloitua jäähdytystä* (*simulated annealing*), jossa laskeva *lämpötilaparametri* ohjaa muutosten suuruutta [39]. On mahdollista toteuttaa konstruktiiviselle heuristiikalle vastaava mekanismi, joka pitää kirjaa parhaasta löydetystä ratkaisusta ja rajoittaa lisättävät solmut ainoastaan niihin solmuihin, jotka sisältyvät parhaaseen löydettyyn ratkaisuun. Kutsutaan tätä *estoksi* ja määritellään eston todennäköisyys p_{esto}, joka kasvaa ratkaisuprosessin aikana. Esto ohjaa työn alla olevaa ratkaisua parhaan löydetyn ratkaisun suuntaan. Painotetun satunnaistamisen ja estot toteuttava muunneltu MAXIMP-heuristiikka etenee seuraavasti:

> *Esikäsittely.* Muodostetaan lista RWD, joka sisältää jokaisen solmun paitsi lähto- ja päätössolmun (0 ja n) järjestettynä laskevasti pistemäärän mukaan.

Houkuttavuuden laskenta. Lasketaan houkuttavuusarvot W_{ij} jokaiselle parille (i, j), jossa $i \neq j, i \neq 0, j \neq 0, i \neq n, j \neq n$. Symmetrisessä ongelmassa parin indeksien järjestyksellä ei ole väliä eli tarvitsee laskea vain parit, joissa $i < j$. Järjestetään parit houkuttavuuden mukaan listaksi WGT. Poistetaan WGT-listalta kaikki parit, joiden $t_{ij} > C_{max}$, koska tällainen pari ylittäisi kustannusten maksimibudjetin. Muodostetaan $(i \times j)$-kokoinen matriisi P pareista (j, r_{WGT}), missä r_{WGT} on parin (i, j) järjestysluku WGT-listalla, siten, että parit on järjestetty jokaisella rivillä i nousevasti järjestysluvun mukaan (eli laskevasti houkuttavuuden mukaan). Tällöin P_{i0} on kaikista houkuttavin solmun i sisältävä pari (j, r_{WGT}), P_{i1} on toiseksi houkuttavin pari ja niin edelleen. Matriisin P kokoa voi valinnanvaraisesti rajoittaa jättämällä osan (vähemmän suosituista) pareista pois, mutta tässä tutkielmassa lasketaan varalta koko matriisi (tähän kuluva aika ei ole merkittävä osuus). Matriisi voi sisältää myös tyhjiä alkioita, jos WGT-listalta on poistettu pareja, joiden $t_{ij} > C_{max}$. Symmetriselle ongelmalle tarvitsee laskea vain matriisin P diagonaalin yläpuolinen osuus (yläkolmio).

Reitin muodostaminen. Muodostetaan reitti seuraavasti:

1. Generoidaan satunnainen indeksi i geometriselta jakaumalta $Geom(p)$, jossa $p = 0.01$ ja alustetaan reitti parin WGT[i] (modulo listan pituus) kahdella solmulla. Tarkistetaan, toteutuuko esto (todennäköisyys p_{esto}). Jos esto toteutui, valitaan sen parin WGT[i] sijaan parhaassa löydetyssä ratkaisussa käytetty alustussolmu.

2. Tarkistetaan, toteutuuko esto (todennäköisyys p_{esto}). Muodostetaan tilapäinen lista TMP etsimällä m (vähintään yksi per solmu) houkuttavinta paria matriisista P iteroimalla ensin rivien (nykyisen reitin solmujen), sitten sarakkeiden (houkuttavuusjärjestyksen) yli; etsitään vain ne parit, joiden toinen solmu ei kuulu reittiin. Jos esto toteutui, rajoitetaan lisäksi haku ainoastaan niihin solmuihin, jotka ovat parhaassa löydetyssä ratkaisussa. Lukumäärä m asetetaan siten, että todennäköisyys generoida i, joka menee listan yli, on pieni (noin 0.01). Järjestetään TMP-lista laskevasti houkuttavuuden (nousevasti järjestysluvun) mukaan. Generoidaan satunnainen indeksi i geometriselta jakaumalta $Geom(p)$, jossa $p = min[max(-0.000045n + 0.01225, 0.001), 0.01]$ ja lisätään parin TMP[i] (modulo listan pituus) puuttuva solmu reittiin. Eston toteutuessa lisätään solmu ensisijaisesti sen solmun viereen, jonka vasemmalla tai oikealla puolella se on parhaassa löydetyssä ratkaisussa; muussa tapauksessa lisätään solmu satunnaiseen paikkaan reitillä.

3. Ratkaistaan reittiä vastaava kauppamatkustajan ongelma (matka alkuterminaalista 0 reittiin valittujen solmujen kautta lopputerminaaliin k). Jos lopputuloksen pituus on pienempi tai yhtä suuri kuin kustannusten maksimibudjetti C_{max}, säilytetään edellisessä vaiheessa lisätty solmu ja solmujen uusi järjestys reitillä seuraavaa iteraatiota varten, muussa tapauksessa poistetaan lisätty solmu ja säilytetään aiempi reitti. Epäonnistunutta solmua ei yritetä lisätä uudelleen. Edellä mainittua tarkistusta ei tarvitse tehdä joka kerta, vaan aluksi voidaan lisätä solmuja tarkistamatta uuden reitin pituutta. Suoritetaan tarkistus vain joka $max(floor(n/100), 1)$ lisäyksellä. Kun pituuden tarkistus epäonnistuu ensimmäisen kerran, tarkistetaan reitin pituus siitä eteenpäin jokaisella iteraatiolla. Jos uutta solmua on yritetty lisätä enemmän kuin $i_{max} = 50$ kertaa, lopetetaan reitin muodostaminen; muussa tapauksessa palataan vaiheeseen 2.

Lopputarkistus. Palautetaan edellisessä vaiheessa muodostettu reitti, jos se on pistemäärältään parempi kuin ainoastaan solmun RWD[0] sisältävä reitti. Muussa tapauksessa palautetaan ainoastaan solmun RWD[0] sisältävä reitti, jos sen kustannus on pienempi tai yhtä suuri kuin kustannusten maksimibudjetti C_{max}; viimeisessä tapauksessa palautetaan tyhjä reitti.

Edellä kuvailtu heuristiikka rinnakkaistuu n:lle prosessorille vaihtamalla satunnaislukugeneraattorin siemenlukua ja pitämällä kirjaa n:stä parhaasta ratkaisusta prosessorikohtaisesti. Ajamalla heuristiikkaa toistuvasti ja kasvattamalla eston todennäköisyyttä p_{esto} jokainen n:stä ratkaisuista tarkentuu hieman eri suuntaan. On myös mahdollista jakaa informaatiota n:n rinnakkaisen ratkaisuprosessin välillä, esimerkiksi korvaamalla prosessorikohtaisen parhaan tunnetun ratkaisun sijaan kaikista säilytetyistä ratkaisuista huonoin, jolloin säilytettyjen ratkaisujen diversiteetti laskee, mutta menetelmä saattaa löytää optimin nopeammin. Luvun 4 tuloksissa on korvattu aina huonoin siihen mennessä löydetty ratkaisu. Reitinmuodostuksessa käytetty parametri $p = 0.01$ on kokeilemalla etsitty alustava arvaus; jatkotutkimus olisi tarpeen p:n, ongelmakoon ja iteraatioiden määrän riippuvuuksien tarkastelemiseksi; tämä vaatisi pitkiä ajoja.

3 Toteutus

Luvussa 2 käsiteltiin ongelmageneraattoria (aliluku 2.3) ja MAXIMP-heuristiikkaa (aliluku 2.4) teorian tasolla, mutta niiden toteuttaminen tehokkaasti ja oikein on lähes yhtä merkittävä haaste. Erityisesti MAXIMP-heuristiikan sisimmän silmukan vaihe 3, jossa ratkaistaan kauppamatkustajan ongelma *jokaista solmun lisäystä kohden*, muodostaa pullonkaulan heuristiikan suorituskyvylle. Jos vaihe 3 on hyvin optimoitu, myös vaiheen 2 haku saattaa olla merkittävä suoritusajan kannalta. Muistinkäyttö voi kasvaa liian suureksi sekä MAXIMP-heuristiikan tarvitsemien matriisien että ongelmageneraattorin taulukointimenetelmän kohdalla, varsinkin jos suurimmat käsiteltävät ongelmat ovat sadan tuhannen solmun kokoluokkaa. Tarkastellaan aluksi toteutuksessa käytettyjä tekniikoita yleisellä tasolla, sitten MAXIMP-heuristiikan toteutuksessa tehtyjä valintoja ja viimeiseksi luvun 2 tilastollisten menetelmien soveltamista.

3.1 Yleiset tekniikat

Yleisesti optimointitutkimuksessa on ollut tapana toteuttaa ratkaisinohjelmat C- tai C++-ohjelmointikielillä [32]. Tulkattu ohjelmointikieli kuten Python on normaalisti liian hidas ratkaisinohjelman ydinosien toteuttamiseen, joten sen rooliksi on jäänyt toimia ”liimakoodina” erilaisten C- ja C++-kirjastojen välillä. Viime vuosina on kuitenkin kehitetty kilpailukykyisiä ajonaikaiseen kääntämiseen (JIT) perustuvia vaihtoehtoja kuten Julia-ohjelmointikieli [32] ja Numba-kääntäjä [28] Pythonille. Numban lisäksi Cython on ollut toinen perinteisempi nopeutusmenetelmä, joka perustuu tavalliseen kääntäjään ja Cython-kieleen, joka sallii Python-tyylisen koodin kirjoittamisen (pienillä muutoksilla) ja kääntää sen C-koodiksi [4]. Edellä mainittujen kehityskulkujen innoittamana toteutuskieleksi valikoitui Python, jota on nopeutettu Numban ja Cythonin avulla.

MAXIMP-heuristiikan toteutus on rinnakkaistettu käyttämällä Numbaa ja Cythonia. Numba mahdollistaa usean säikeen ajamisen samanaikaisesti, koska se pystyy vapauttamaan Pythonin *global interpreter lock*-lukon [28]. Ongelmageneraattorin toteutuksessa on myös käytetty Numbaa, mutta rinnakkaistaminen ei ollut tarpeen. Cythonia on käytetty houkuttavuusmatriisi P:n laskevassa osuudessa ja hyödynnetty myös Cythonin OpenMP-tukea silmukoiden rinnakkaistamiseen (*parallel for*).

3.2 Ongelmageneraattorin toteutus

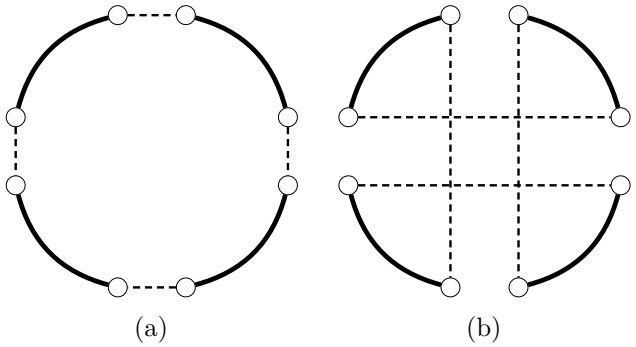
Ongelmageneraattori on pyritty toteuttamaan mahdollisimman tarkasti luvussa 2 kuvaillulla tavalla. Laskutoimituksissa on käytetty pääasiassa kokonaislukuja, jotta

mahdolliset liukulukulaskutoimitusten epätarkkuudet eivät aiheuttaisi virheellisiä tuloksia. Generaattori varmistaa jälkikäteen, että toivottu optimi on myös olemassa, mutta yksinkertaiset jälkitarkistukset eivät voi täysin sulkea pois sitä mahdollisuutta, että generoidussa ongelmassa on myös toivottua suuremman pistemäärän optimi: niin kauan kuin generaattori noudattaa täysin luvun 2 matemaattista käsittelyä, kyseisen tilanteen pitäisi kuitenkin olla mahdoton. Toteutetun generaattorin lähdekoodi on liitteessä A.

3.3 MAXIMP-heuristiikan toteutus

MAXIMP-heuristiikan suoritusajan kannalta merkittävin vaihe 3 saattaa aluksi vaikuttaa jopa toteutuskelvottomalta, koska jokaista solmua lisättäessä ratkaistaan kauppamatkustajan ongelma. Keskeinen oivallus on siinä, että reitin pituuden tarkistamiseksi kauppamatkustajan ongelmaa ei tarvitse ratkaista kokonaan uudelleen jokaisella iteraatiolla, koska yksittäisen solmun lisäämisen jälkeen uusi reitti poikkeaa vain hieman edellisestä. Luvun 3.3 mukaisesti lisäyksiä voidaan tehdä myös useampi kuin yksi jokaisen pituustarkistuksen välissä, kunhan yli menneet lisäykset perutaan ja aloitetaan uudestaan edellisestä tarkistetusta reitistä. Lisäksi suoritus voidaan keskeyttää heti kun kustannusten maksimibudjetti on saavutettu. Riittää siis, että vaiheessa 3 käytettävä menetelmä ”pysyy lisäysten kintereillä”. Eksakti algoritmi ei välttämättä taivu kyseiseen tarkoitukseen, mutta kauppamatkustajan ongelmaan on kehitetty monia *lokaalin haun* (*local search*) heuristiikkoja, jotka yrittävät parantaa olemassaolevaa ratkaisua [1]. Näistä mahdollisesti tunnetuin on Lin-Kernighan-heuristiikka [30].

Lin-Kernighan-heuristiikka on yleistys 2-opt- ja 3-opt-heuristiikoista [6], joissa vaihdetaan kahden tai kolmen solmun paikat reitillä (väliin jäävien solmujen järjestys käännetään eri tavoin) ja lasketaan lyhentäisikö vaihdos/käännös reittiä. Käytännössä nämä k-opt-menetelmät ”suoristavat” kohtia, joissa (tasolle piirretty) reitti leikkaa itsensä. Lin-Kernighan-heuristiikka on huomattavasti hankalampi toteuttaa kuin 2-opt ja 3-opt, joten tässä tutkielmassa on valittu Johnsonin ja McGeochin [1] suositusten mukaisesti 3-opt-menetelmä ja erityisesti sen iteroitu variantti (*iterated 3-opt*), jossa menetelmä ajetaan useita kertoja peräkkäin tekemällä jokaisen iteraation jälkeen *double-bridge-liike* (*double-bridge move*), jossa reitti katkaistaan neljästä satunnaisesta kohdasta ja yhdistetään uudelleen kuvan 3.1 tapaan. Iteraatioiden määrää voi muuttaa suoritusaikavaatimusten mukaan, mutta tässä tutkielmassa käytetään kymmentä iteraatiota. Lisäksi käytetään *naapurilistoja* (*neighbor lists*) ja *sulkubittejä* (*don't look bits*) [1]. Naapurilistoissa säilytetään jokaiselle solmulle vain n lähintä solmua (läheisyysjärjestyksessä) ja jätetään loput tarkastelun ulkopuolelle. Myös sulkubitti jättää solmun tarkastelun ulkopuolelle ja se asetetaan,

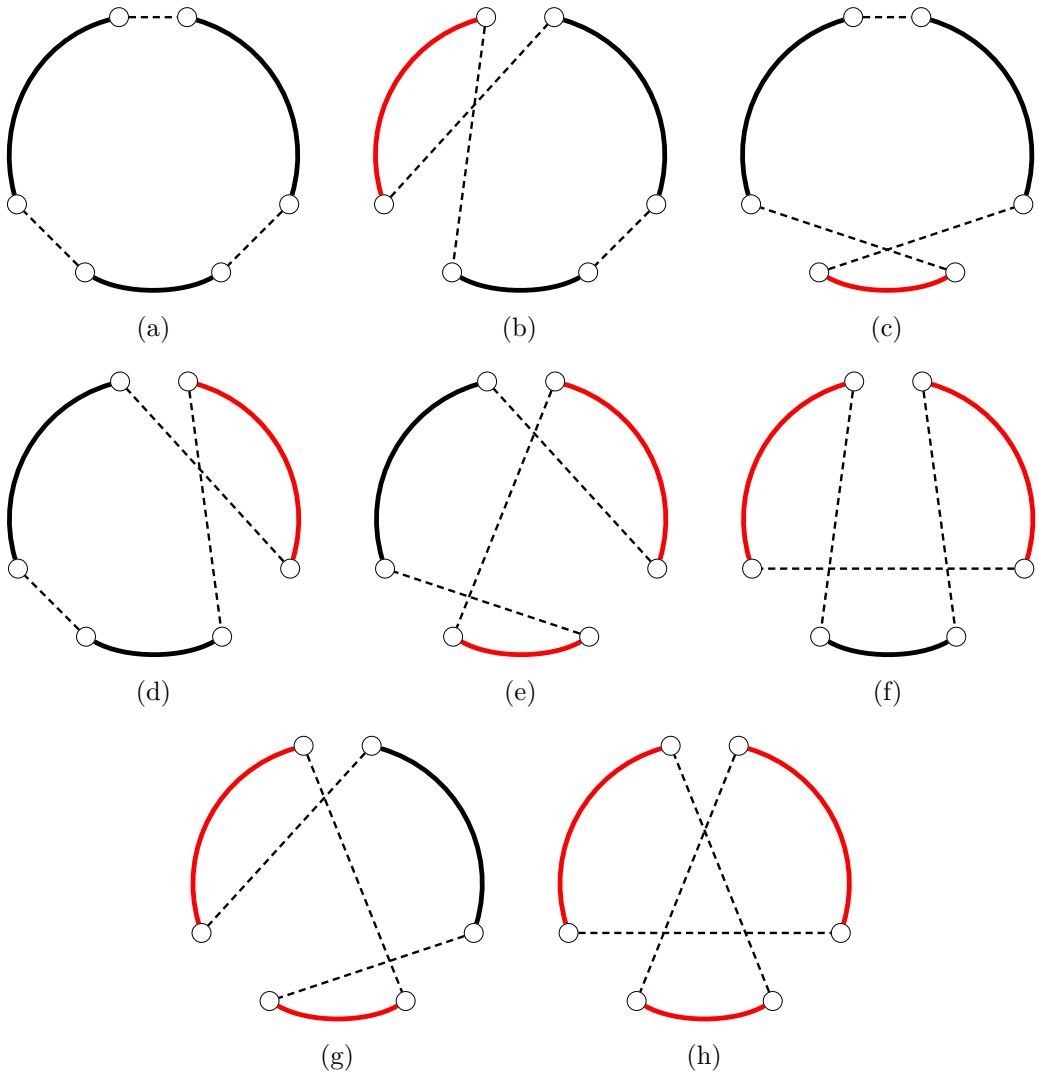


Kuva 3.1 *Double-bridge-liike ennen (a) ja jälkeen (b).*

kun solmu on käyty läpi mutta ei ole löydetty yhtään siitä lähtevää reittiä lyhentävää vaihdosta. Jos solmu osallistuu myöhemmin reittiä lyhentävään muutokseen, sulkubitti nollataan.

3-opt-heuristiikka katkaisee reitin kolmesta kohtaa (i, j, k) ja yrittää yhdistää reitin uudelleen kokeillen jokaista mahdollista tapaa. Kuvassa 3.2 esitellään 3-opt-heuristiikan kahdeksan tapausta, joista ensimmäinen tapaus 3.2(a) kuvaa lähtötilannetta, jossa yhtään reittiosuutta ei ole käännetty (merkitty punaisella). Jäljelle jäävistä seitsemästä tapauksesta ensimmäiset kolme tapausta 3.2(b)–3.2(d) ovat 2-opt-tapauksia, joissa käännetään vain yksi reittiosuus. Viimeiset neljä tapausta 3.2(e)–3.2(h) ovat puhtaita 3-opt-tapauksia, joissa käännetään kaksi tai kolme reittiosuutta. Toteutettu 3-opt-heuristiikka laskee jokaisesta tapauksesta seuraavat kustannussäästöt ja suorittaa käännökset tapauksesta, joka tuottaa eniten säästöjä. Jos säästöjä ei seuraa yhdestäkään tapauksesta, suoritus jatkuu, mutta solmun i sulkubitti asetetaan. Reittiä iteroidaan käyden läpi solmun i ja j naapurilistoja uudelleen ja uudelleen, kunnes ei löydy yhtään mahdollista säästöä. Käytännössä 3-opt:in voi siis toteuttaa while-silmukkana, joka tarkistaa löytyikö yhtään säästöä: while-silmukan sisällä naapurilistojen iterointi toteutetaan kolmena sisäkkäisenä for-silmukkana, missä sulkubitit nopeuttavat uloimman i:n yli iteroivan silmukan suoritusta. Naapurilistojen koko vaikuttaa myös suoritusaikaan; tässä tutkielmassa niissä säilytetään jokaiselle solmulle 20 lähintä solmua Johnsonin ja McGeochin suositusten mukaisesti [1].

3-opt-menetelmän suorituskykyyn vaikuttaa huomattavasti valittu alkureitti, jota lähdetään parantamaan. Lyhyiden kokeilujen perusteella näyttää siltä, että joskus paras taktiikka on käyttää reitin olemassaolevaa järjestystä lisäyksen jälkeen, joskus taas rakentaa kokonaan uusi alkureitti, mihin tarvitaan mielellään vielä nopeampi heuristiikka kuin 3-opt. Tämä on melko ymmärrettävää, sillä joskus lisäys voi olla hyvä, mutta osua huonoon paikkaan reitillä, jolloin alkureitti pitenee huomattavasti. Tässä tutkielmassa lasketaan yksinkertaisella lähimmän naapurin menetelmällä



Kuva 3.2 *3-opt:in tapaukset.*

vaihtoehtoinen alkureitti ja käytetään sitä, jos se on lyhyempi kuin olemassaoleva järjestys.

Normaalitilanteessa yhden solmun lisäämisestä seuraavien 3-opt-tarkistusten määrä pysyy pienenä ja näin ollen vaiheen 3 suoritus nopeutuu huomattavasti verrattuna naiiviin toteutukseen. MAXIMP-iteraation lopussa 3-opt-ratkaisu kannattaa hioa loppuun asti ajamalla enemmän kuin kymmenen 3-opt-iteraatiota (tässä tutkielmassa käytetään 1000 iteraatiota). Joissakin tapauksissa vaiheen 2 haku kestää jopa kauemmin kuin kauppamatkustajan ongelman ratkaiseminen vaiheessa 3. Vaiheen 2 haun nopeuttamiseksi matriisia P kannattaa iteroida siten, että peräkkäin haettavat alkiot ovat myös muistissa vierekkäin (rivi ensin tai sarake ensin riippuen ohjelmointiympäristöstä), jolloin prosessori tekee vähemmän välimuistihuteja (välimuisti varastoi tyypillisesti useita muistissa vierekkäisiä elementtejä kerrallaan).

Suoritusajan lisäksi myös muistinkäyttö voi estää suurikokoisten ongelmien ratkaisemista: matriisilla P tehdään klassinen muistinkäytön ja suoritusajan välinen kompromissi (*space-time tradeoff*) tallentamalla esilaskettuja houkuttavuusjärjestyksiä taulukkoon sen sijaan, että houkuttavuus laskettaisiin ajon aikana monta kertaa uudelleen. Symmetrisen ongelman tapauksessa kustannus- ja houkuttavuusmatriiseista tarvitsee säilyttää vain diagonaalin yläpuolella olevat elementit, jolloin muistinkäyttö puolittuu. Muistinkäytön vähentämiseksi edelleen on mahdollista rajoittaa matriisien arvot esimerkiksi 16-bittisiksi positiivisiksi kokonaisluvuiksi, mutta tällöinkin $n = 100000$ kokoluokan ongelman symmetrinen kustannusmatriisi varaa n. 10 gigatavua keskusmuistia. Suorittimen ytimet voivat jakaa kustannusmatriisin ja houkuttavuusmatriisin P, mutta eivät omia tilapäisiä listojaan eivätkä 3-opt:in käyttämää pienempää kustannusmatriisia, joka sisältää vain reittiin sillä hetkellä kuuluvat solmut. Perinteisen suorittimen tapauksessa on helppo lisätä erillistä keskusmuistia, mutta jos haluttaisiin käyttää GPU-laskentaa, tarvittavat tietorakenteet eivät välttämättä mahtuisi yksittäisen grafiikkasuorittimen muistiin.

3.4 Tilastolliset menetelmät

Luvussa 2 käsiteltiin kahta tilastollista lähestymistapaa optimin estimointiin: ääriarvoteoriaa ja linkkuveitsimenetelmää. Yleinen M:n asteen linkkuveitsiestimaattori on [9]:

$$\hat{\theta}_{JK}^{(M)} = \sum_{i=1}^{M+1} (-1)^{(i-1)} \left(\begin{array}{c} M+1 \\ i \end{array} \right) \tilde{x}_{(i)}, \tag{3.1}$$

jossa $\tilde{x}_{(i)}$ on i:ksi pienin heuristinen ratkaisu minimointiongelman tapauksessa (maksimointiongelmalle arvot voi kertoa luvulla -1 ja käsitellä kuten minimointiongelmaa). Linkkuveitsimenetelmä on joustavampi kuin ääriarvoteoreettinen menetelmä, koska M:n arvoa kasvattamalla voidaan vähentää linkkuveitsimenetelmän tuottamaa harhaa keskineliövirheen kasvattamisen kustannuksella [9], jolloin käyttäjä voi valita haluamansa kompromissin. Käytännössä suuremmat M:n arvot kasvattavat estimaatin vaihteluväliä ja antavat samalla varovaisemman estimaatin: minimointiongelman optimiarvio pienenee M:n kasvaessa ja maksimointiongelman arvio suurenee. Yleensä suositellaan käytettäväksi 1. tai 2. asteen linkkuveitsiestimaattoria [9], mutta Zhao ja kumpp. mukaan jopa 4. asteen estimaattori voi olla hyödyllinen [47]. 2. ja 3. asteen estimaattorin tulokset ovat aiemmassa tutkimuksessa sijoittuneet 1. ja 4. asteen estimaattorin väliin [47], joten riittää testata ainoastaan 1. ja 4. asteen ääritapaukset. Piste-estimaatti ei yksinään kerro kovin paljoa, joten linkkuveitsiestimaatille lasketaan lisäksi *persentiilimenetelmällä* (*percentile method*) 95 %:n bootstrap-luottamusvälit ($n = 10000$) [13].

4 Tulokset

Tässä luvussa ratkaisintoteutusta testataan OPLib-testipankilla [26] ja itse generoiduilla testiongelmilla, joista jälkimmäisille on tiedossa myös generaattorin takaama todellinen optimi. OPLib-testipankista on valittu joukko `gen2`, jossa pistemäärät ovat satunnaisgeneroituja, koska joukossa `gen1` kaikki pistemäärät saavat arvon 1 ja joukossa `gen3` pistemäärät riippuvat järjestelmällisesti kustannuksista, joka saataisi suosia tai haitata MAXIMP-heuristiikkaa. Itse generoidut testiongelmat mahdollistavat ratkaisimen skaalautumisen tarkastelun suurikokoisilla ongelmilla ja todellisen optimin vertailun bootstrap-luottamusväliin. Erityisesti kiinnostavaa on, sijoittuuko todellinen optimi bootstrap-luottamusvälille suuren n:n ongelmassa, jossa ratkaisin ei välttämättä pääse yhtä lähelle todellista optimia kuin pienikokoisissa ongelmissa. Tarkka optimiestimaatti antaisi heuristiikalle selkeän lopetuskriteerin, mutta arvioon ei voi luottaa, jos todellinen optimi sijaitsee usein tai systemaattisesti 95 %:n luottamusvälin ulkopuolella.

Itse generoiduissa ongelmissa tarkastellaan kahta joukkoa, joissa optimireitille valittujen solmujen osuus vaihtelee. Parametri c asettaa generointivaiheessa ratkaistavan reppuongelman kapasiteetin per tavara: arvolla $c = 8$ noin 25% tavaroista/-solmuista mahtuu optimireitille ja arvolla $c = 16$ noin 50% tavaroista/solmuista mahtuu optimireitille. OPLib-testipankissa vastaava kapasiteetti/kustannusraja, eli suunnistusongelman kustannusrajan prosentuaalinen osuus saman instanssin kaikki solmut sisältävän TSP-ratkaisun kustannuksesta, on noin 50%. Lisäksi kuvaajissa 4.1 ja 4.2 tarkastellaan MAXIMP-toteutuksen suoritusajan ja sen antamien ratkaisujen laadun skaalautumista ongelmakoon mukaan. Kuvaajissa käytetyt testiongelmat ovat muuten samoja kuin taulukossa 4.3, mutta koska kokojen $n = 100$ ja $n = 200$ ongelmissa satunnaisvaihtelu oli suurempaa, kuvaajissa on keskiarvotettu suurempi määrä ongelmia (sata) kuin muille ongelmako'oille (kolme).

OPLib-testipankin ongelmien kohdalla tuloksia vertaillaan kirjallisuudesta löytyviin EA4OP-ratkaisimen [26] ja eksaktin branch-and-cut-ratkaisimen (RBC) [27] tuloksiin samoille ongelmille. EA4OP on suunnistusongelmalle räätälöity evoluutioalgoritmi, jonka suorituskyky on heuristisen ratkaisimien kärkeä, sillä se löysi uusia julkaisuhetkellä parhaaksi tiedettyjä ratkaisuja (*best known solutions*) [26]. Itse generoiduilla testiongelmilla saavutettuja tuloksia verrataan generaattorin takaamaan optimiin. Jokaiselle ajolle raportoidaan suoritusaika sisältäen esilaskennan, saavutettu pistemäärä, *goodness gap* eli prosentuaalinen ero todellisesta optimista ("RBC", kun $n \leq 400$ ja generaattorin takaama optimi rnd-luokassa) tai parhaasta tunnetusta ratkaisusta ("Paras", kun $n > 400$) [27], skaalattu keskihajonta SR, linkkuveitsiestimaatit $\hat{\theta}_{JK}^{(1)}$ ja $\hat{\theta}_{JK}^{(4)}$ ja linkkuveitsiestimaattien 95 %:n

bootstrap-luottamusvälin antamat ala- ja ylärajat. Skaalattu keskihajonta $SR = 1000\sigma\left(\tilde{x}_i\right)/\hat{\theta}_{JK}^{(1)}$ kertoo miten lähellä vaihtoehtoisten ratkaisujen pistemäärät $\tilde{x}_i$ ovat toisiaan; skaalaus tehdään jakamalla 1. asteen linkkuveitsiestimaatilla $\tilde{x}_i$, jotta luku olisi vertailukelpoinen ongelmien välillä [9].

MAXIMP-ajot suoritettiin 16-ytimisellä AMD Ryzen 9 5950X-prosessorilla. Suoritusajan rajaamiseksi ajoissa ylläpidetään 32 ratkaisua (yksi per säie) ja jokaista ratkaisua yritetään parantaa 50 kertaa luokan $n \leq 400$ OPLib-ongelmille, jolloin MAXIMP-heuristiikan iteraatioiden määrä on $32 * 50 = 1600$. Luokan $n > 400$ ja rnd-luokan satunnaisgeneroiduissa ongelmissa jokaista ratkaisua yritetään parantaa 6 kertaa, jolloin iteraatioiden määrä on $32 * 6 = 192$. Eston todennäköisyys on alussa 0.1, kasvaa lineaarisesti arvosta 0.1 arvoon 0.9 kun 5-50% iteraatioista on kulunut ja pysyy arvossa 0.9 kun 50% iteraatioista on kulunut.

Taulukoiden 4.1 ja 4.2 tulokset osoittavat, että esitetty MAXIMP-heuristiikan muunnelma ei ole useimpien OPLib-ongelmien kohdalla kilpailukykyinen ratkaisujen laadussa (GGap) verrattuna ratkaisimien parhaimmistoon. Aivan pienikokoisimmissa ongelmissa MAXIMP-ratkaisin saavuttaa toisinaan todellisen optimin, mutta pidemmässä ajassa kuin eksakti RBC-ratkaisualgoritmi tai EA4OP. Muutamassa ongelmassa kuten `brazil58`, `gr137` ja `pr264` MAXIMP saavuttaa kuitenkin paremman ratkaisun kuin EA4OP. Suoritusaikojen kohdalla vertailu on hankalaa, koska ratkaisujen laadun ja suoritusajan vuorovaikutusta pitäisi tarkastella useilla eri iteraatiomäärillä. Suurempien $n > 400$ ongelmien kohdalla taulukon 4.2 tulosten vertailu on hankalaa, koska MAXIMP vie 192 iteraatiolla tyypillisesti vähemmän aikaa. Vaikuttaa siltä, että MAXIMP vaatii enemmän suoritusaikaa kuin EA4OP saman pistemäärän saavuttamiseksi OPLib-ongelmissa, mutta tämän selvittäminen perusteellisesti ei ollut mahdollista suoritusaikarajoitteiden takia. Iteraatioiden määrää kasvattamalla olisi ehkä päästy lähemmäksi muita ratkaisimia pidemmän suoritusajan kustannuksella. Eksaktia ratkaisinta tai EA4OP-heuristiikkaa ei testata itse generoiduilla ongelmilla, koska niiden suoritusajat ovat jo OPLib-testipankin luokan $n > 400$ suurimmissa ongelmissa tuntien luokkaa.

Taulukon 4.3 tulosten perusteella näyttää siltä, että generaattorin tuottamat suurikokoiset ongelmat ovat helpompia kuin OPLib-testipankin vaikeimmat ongelmat. Prosentuaalinen ero saavutetun pistemäärän ja optimin välillä (GGap) pysyy kohtalaisena ongelmien suuresta koosta huolimatta ja suoritusajat pysyvät pieninä, kun ratkaisua ei yritetä parantaa ylimääräisillä iteraatioilla. Samaan aikaan vaikuttaa kuitenkin myös siltä, että on mahdollista tuottaa ei-triviaaleja ongelmia, jotka aidosti haastavat heuristiikkaa: käytetyillä iteraatiomäärillä prosentuaalinen ero laadussa (GGap) on yli 20% `rnd250`-ongelmissa, joille $c = 8$. Vertaamalla tuloksia esimerkiksi OPLib-testipankin suurimpaan ongelmaan `pla7397` taulukossa 4.2 huomataan, että prosentuaalinen ero voi olla jopa yli 40% haastavimmissa OPLib-

	MAXIMP										EA4OP			RBC	
Nimi	Pisteet	Aika (s)	GGap (%)	SR	JK1	JK1 p2,5	JK1 p97,5	JK4	JK4 p2,5	JK4 p97,5	Pisteet	GGap (%)	Aika (s)	Pisteet	Aika (s)
a280	7345	64,71	12,85	8,47	7457	7224	7473	7780	6609	7885	8304	1,47	2,85	8428	519,95
att48	1717	10,03	*	0,00	1717	1717	1717	1717	1717	1717	1717	*	0,32	1717	0,04
berlin52	1897	11,68	*	0,00	1897	1897	1897	1897	1897	1897	1897	*	0,35	1897	3,23
bier127	5277	37,69	1,97	1,96	5277	5277	5283	5271	5228	5349	5381	0,04	1,71	5383	0,96
brazil58	2220	14,68	*	0,00	2220	2220	2220	2220	2220	2220	2218	0,09	1,52	2220	0,46
d198	6114	49,47	8,34	11,24	6124	6084	6157	6131	5794	6380	6660	0,15	7,33	6670	298,24
eil101	3592	21,88	1,72	2,83	3610	3574	3610	3655	3473	3708	3655	*	0,82	3655	4,15
eil51	1668	8,44	0,36	0,54	1668	1668	1668	1668	1668	1668	1668	0,36	0,18	1674	0,96
eil76	2515	14,87	1,37	5,08	2527	2492	2538	2577	2417	2607	2550	*	0,43	2550	0,62
gil262	7507	56,04	9,78	12,74	7606	7383	7643	7781	6751	8184	8175	1,75	3,47	8321	64,63
gr120	4232	29,12	3,18	1,97	4254	4201	4263	4346	4054	4359	4356	0,34	1,37	4371	6,57
gr137	4188	26,58	2,29	5,23	4247	4119	4258	4417	3798	4531	4099	4,36	3,09	4286	10,65
gr202	7294	51,28	6,36	7,22	7303	7237	7351	7305	6925	7590	7789	*	8,77	7789	139,90
gr229	8627	70,38	5,99	5,77	8686	8551	8703	8782	8186	8967	9174	0,03	13,19	9177	16,67
gr48	1749	8,98	0,68	0,00	1749	1749	1749	1749	1749	1749	1749	0,68	0,20	1761	1,32
gr96	3267	21,54	3,80	6,54	3349	3174	3364	3611	2726	3671	3394	0,06	1,44	3396	9,50
hk48	1614	7,23	*	8,80	1614	1614	1614	1614	1614	1614	1614	*	0,15	1614	0,10
kroA100	3092	16,93	3,74	5,13	3098	3086	3098	3116	3038	3153	3212	*	0,57	3212	0,70
kroA150	4375	26,33	11,04	4,23	4385	4354	4396	4456	4262	4471	4902	0,33	1,26	4918	60,43
kroA200	5912	35,96	9,70	16,03	5925	5777	6050	5271	5098	6480	6534	0,20	1,71	6547	16,18
kroB100	3109	19,45	4,07	1,07	3109	3109	3109	3109	3109	3109	3238	0,09	0,52	3241	13,28
kroB150	4281	25,66	12,08	11,63	4365	4139	4423	4603	3469	4869	4869	*	1,19	4869	16,94
kroB200	5845	36,35	8,94	10,71	5927	5735	5956	6228	5290	6293	6278	2,20	1,97	6419	20,62
kroC100	2818	15,03	4,38	2,61	2827	2809	2827	2854	2764	2854	2931	0,54	0,60	2947	2,22
kroD100	3208	18,52	2,99	5,83	3208	3203	3213	3178	3166	3268	3307	*	0,65	3307	3,62
kroE100	2940	20,49	4,85	1,68	2940	2933	2948	2897	2894	2984	3082	0,26	0,50	3090	11,31
lin105	3498	23,58	1,30	6,22	3508	3439	3557	3294	3147	3762	3530	0,40	1,10	3544	2,51
lin318	9335	81,67	14,54	7,76	9346	9298	9400	9363	8814	9749	10866	0,52	8,29	10923	367,53
pr107	2667	20,74	*	0,00	2667	2667	2667	2667	2667	2667	2667	*	1,05	2667	0,20
pr124	3893	22,71	0,61	0,00	3893	3893	3893	3893	3893	3893	3899	0,46	1,34	3917	1,07
pr136	3834	26,47	11,02	4,86	3835	3833	3840	3833	3743	3898	4309	*	1,15	4309	1,25
pr144	3807	25,64	4,90	5,00	3878	3736	3878	4091	3381	4091	3965	0,95	3,02	4003	32,23
pr152	4077	19,05	4,72	7,51	4150	3996	4159	4334	3594	4409	4245	0,79	3,47	4279	1,85
pr226	6068	40,32	8,92	10,96	6111	5987	6155	6080	5587	6440	6658	0,06	7,29	6662	2894,81
pr264	6654	45,03	*	0,00	6654	6654	6654	6654	6654	6654	6173	7,23	5,94	6654	13,33
pr299	7967	65,36	13,23	4,04	7978	7943	7992	8046	7821	8088	9112	0,76	3,23	9182	623,34
pr76	2657	15,52	1,88	4,22	2657	2657	2657	2657	2657	2657	2708	*	0,48	2708	1,46
rat195	5186	40,27	10,45	5,58	5211	5130	5245	5176	4876	5446	5703	1,52	1,55	5791	46,09
rat99	2829	17,03	3,91	4,01	2829	2829	2829	2829	2763	2873	2944	*	0,49	2944	3,25
rd100	3243	19,09	3,45	5,77	3267	3198	3288	3213	2973	3423	3359	*	0,50	3359	0,36
rd400	11401	91,44	16,49	7,01	11452	11319	11493	11519	10889	11853	13442	1,54	6,80	13652	769,66
st70	2255	13,30	1,36	0,78	2255	2249	2261	2279	2219	2279	2285	0,04	0,31	2286	1,77
ts225	6114	51,14	10,54	13,43	6156	6041	6189	6268	5699	6490	6819	0,22	1,47	6834	95,22
tsp225	6225	42,43	10,91	7,93	6314	6118	6335	6617	5609	6697	6936	0,73	1,87	6987	54,09
u159	4170	30,50	15,93	6,97	4171	4159	4181	4184	4104	4222	4941	0,38	1,44	4960	14,96

Taulukko 4.1 *OPLib Gen2-tulokset, $n \leq 400$. Kolmen ratkaisimen keräämät pisteet, suoritusajat ja prosentuaalinen ero optimiin (RBC Pisteet). MAXIMP-ratkaisujen skaalattu keskihajonta SR, 1. ja 4. asteen linkkuveitsiestimaatit JK1 ja JK4 sekä niiden 95 %:n luottamusvälit.*

	MAXIMP										Paras	EA4OP			RBC	
Nimi	Pisteet	Aika (s)	GGap	SR	JK1	JK1 p2,5	JK1 p97,5	JK4	JK4 p2,5	JK4 p97,5	Pisteet	Pisteet	Aika (s)	GGap	Pisteet	Aika (s)
ali535	19095	36,42	13,02	6,22	19123	18913	19279	19375	17895	20055	21954	21910	95,05	0,20	21954	18000,00
att532	17993	27,81	8,36	4,63	18023	17960	18033	18115	17752	18243	19635	19265	23,43	1,88	19635	18000,00
d1291	29672	92,61	21,46	11,25	29689	29416	30104	28146	27046	31536	37778	35153	289,25	6,95	37778	18000,00
d1655	37969	121,42	23,01	8,91	38209	37613	38357	38218	35695	40098	49319	47211	683,17	4,27	46158	18000,00
d2103	42942	135,68	32,30	9,16	43318	42461	43423	44016	40152	44903	63426	57202	682,28	9,81	63426	16593,51
d493	11735	11,52	30,95	24,39	11826	11531	11961	11597	9810	13367	16995	16729	17,15	1,57	16995	18000,00
d657	17511	21,41	18,56	11,84	17598	17121	17930	16212	15136	19327	21503	21162	22,90	1,59	21503	554,67
dsj1000	29495	79,54	17,69	5,10	29665	29288	29703	29952	28247	30469	35835	34463	83,34	3,83	35835	18000,00
fl1400	35174	101,95	37,96	20,48	35922	33427	36921	33732	25492	42938	56692	56258	794,15	0,77	54124	18000,00
fl1577	38199	158,44	16,06	7,62	38199	37932	38498	38835	36400	39575	45505	45505	334,28	*	45326	18000,00
fl3795	94306	1670,25	8,79	7,19	94919	92997	95655	93652	86648	99862	103397	103397	4788,96	*	98998	18000,00
fl417	10608	8,30	11,10	20,22	10658	10480	10736	10321	9821	11196	11933	11787	16,73	1,22	11933	18000,00
fnl4461	116937	1472,12	20,51	2,77	117083	116687	117209	116932	115362	118275	147109	140424	2618,15	4,54	147109	18000,00
gr431	16467	30,46	10,10	9,22	16516	16413	16538	16670	16017	16892	18318	18287	51,38	0,17	18318	2809,41
gr666	23570	50,34	11,10	6,51	23692	23303	23837	24479	22176	24754	26514	26336	136,48	0,67	26514	18000,00
nrw1379	38300	119,73	17,94	3,69	38322	38157	38474	38577	37208	39170	46676	45602	117,51	2,30	46676	18000,00
p654	15367	17,25	14,15	24,55	15699	14992	15772	16420	12937	17935	17900	17821	42,82	0,44	17753	18000,00
pa561	17354	25,27	11,35	3,79	17414	17273	17437	17470	16866	17725	19576	18894	23,45	3,48	19576	1961,95
pcb1173	29340	75,24	20,78	5,49	29567	29033	29647	29855	27563	30740	37035	35826	69,94	3,26	37035	18000,00
pcb3038	73711	575,82	24,71	4,01	73731	73489	74013	72569	71859	75157	97902	91842	820,37	6,19	97902	18000,00
pcb442	11480	13,79	20,74	16,12	11614	11001	11964	12468	8810	13528	14484	14273	6,83	1,46	14484	13760,94
pla7397	166302	5563,02	41,02	1,73	166317	166169	166436	165923	165397	167066	281977	272452	18000,00	3,38	281977	18000,00
pr1002	24516	55,72	25,78	6,69	24548	24364	24670	24610	23648	25268	33030	31746	46,19	3,89	33030	18000,00
pr2392	55799	340,65	23,40	4,08	55800	55764	55941	55585	54936	56475	72843	71018	440,57	2,51	72843	18000,00
pr439	12598	13,92	22,10	23,29	12812	11867	13329	10420	8212	15850	16171	16085	11,77	0,53	16171	3765,86
rat575	14850	22,13	18,63	5,43	14909	14767	14939	15140	14394	15314	18251	17705	14,97	2,99	18251	9616,70
rat783	20354	31,17	20,10	5,40	20412	20256	20452	20412	19712	20962	25474	24861	32,36	2,41	25474	12246,90
rl1304	31851	100,35	24,66	6,82	31907	31522	32183	30970	29935	33255	42275	40561	97,68	4,05	42275	18000,00
rl1323	32674	108,52	24,67	6,47	32808	32382	32972	32840	31117	34053	43377	41459	89,78	4,42	43377	18000,00
rl1889	45794	264,27	27,66	4,70	45814	45536	46052	44884	44287	46914	63308	60084	286,07	5,09	63308	18000,00
rl5915	137208	4873,10	25,60	3,40	137678	136149	138319	137680	131700	141860	184424	176678	5512,40	4,20	184424	18000,00
rl5934	135464	4811,23	27,57	4,28	136045	134210	136722	134202	128064	140520	187034	171649	5757,80	8,23	187034	18000,00
u1060	26764	65,99	25,97	4,19	26769	26700	26842	26436	26286	27128	36151	35110	77,78	2,88	36151	18000,00
u1432	36477	99,48	22,30	3,55	36595	36308	36651	36718	35483	37318	46946	44810	100,91	4,55	46946	18000,00
u1817	42952	189,40	20,82	4,83	43012	42720	43217	42235	41398	44320	54245	50366	734,39	7,15	54245	18000,00
u2152	51984	303,29	19,59	4,45	52007	51703	52304	52613	50010	53472	64649	60211	1164,38	6,86	64649	18000,00
u2319	62721	265,71	22,48	3,84	62911	62443	62999	63605	61360	63985	80914	78102	447,06	3,48	80914	18000,00
u574	15021	23,48	22,38	8,11	15117	14745	15297	15669	13517	16189	19351	18966	16,33	1,99	19351	1026,82
u724	17987	28,02	25,74	7,62	18053	17654	18320	16879	16072	19399	24223	23793	28,71	1,78	24223	9829,42
vm1084	27491	78,73	32,58	6,21	27589	27230	27752	27061	25942	28951	40777	40308	55,67	1,15	40777	18000,00
vm1748	41670	203,30	38,76	5,48	41924	41195	42145	41458	38900	43818	68042	66685	195,85	1,99	68042	18000,00

Taulukko 4.2 *OPLib Gen2-tulokset, $n > 400$. Kolmen ratkaisimen keräämät pisteet, suoritusajat ja prosentuaalinen ero (GGap) optimiin tai parhaaseen tunnettuun ratkaisuun (Paras). MAXIMP-ratkaisujen skaalattu keskihajonta SR, 1. ja 4. asteen linkkuveitsiestimaatit JK1 ja JK4 sekä niiden 95 %:n luottamusvälit.*

MAXIMP											
Nimi	Optimi	Pisteet	GGap (%)	Aika (s)	SR	JK1	JK1 p2,5	JK1 p97,5	JK4	JK4 p2,5	JK4 p97,5
rnd100_2_8	19479	16260	16,53	0,54	21,91	16918	15513	17016	18885	11963	19406
rnd100_5_8	21975	19859	9,63	0,59	24,77	20831	18248	21470	22483	10741	26303
rnd100_7_8	22726	18265	19,63	0,55	27,31	18379	17880	18693	18184	15853	20149
rnd100_3500_16	35587	32839	7,72	1,20	9,01	33189	32403	33294	33850	30230	34797
rnd100_3501_16	32265	30166	6,51	1,20	11,27	31186	29139	31198	34218	23968	34766
rnd100_3502_16	32526	30279	6,91	1,20	9,09	30323	29871	30695	28259	27739	32259
rnd250_503_8	58135	43915	24,46	1,58	14,49	44013	42551	45285	38608	35762	49783
rnd250_504_8	53147	41117	22,64	1,55	18,33	41535	40615	41757	42697	37683	44237
rnd250_505_8	54862	41317	24,69	1,52	13,84	41405	41143	41496	41155	40167	42520
rnd250_4001_16	85763	77377	9,78	3,51	5,04	77486	77060	77778	76543	74691	79853
rnd250_4004_16	87285	79297	9,15	3,52	4,72	79452	78982	79612	78925	77169	81645
rnd250_4005_16	92929	84439	9,14	3,65	7,66	84498	83593	85403	80217	78537	89624
rnd500_1000_8	111960	94143	15,91	3,75	9,18	95331	92831	95496	98626	86440	99991
rnd500_1001_8	111473	93954	15,72	3,50	5,78	94127	93503	94409	93780	91558	95912
rnd500_1002_8	108935	91852	15,68	3,66	9,92	92097	91189	92605	91424	86399	96824
rnd500_4500_16	182902	170641	6,70	8,67	3,70	170769	169949	171732	167874	163722	176029
rnd500_4501_16	188503	178800	5,15	8,77	2,76	179162	178300	179300	179225	175605	181580
rnd500_4504_16	186805	175740	5,92	8,23	2,26	175876	174908	176628	173148	170625	179668
rnd1000_1501_8	217943	190885	12,42	9,00	2,46	190920	190386	191396	191839	187716	193341
rnd1000_1502_8	226891	196416	13,43	9,62	4,11	197249	194769	198176	194957	186055	205012
rnd1000_1503_8	219204	191245	12,75	9,07	2,82	191914	190520	191970	193429	186739	195771
rnd1000_5004_16	347524	332226	4,40	22,54	0,97	332264	331996	332485	332173	330677	333417
rnd1000_5008_16	360010	342640	4,82	23,89	1,56	342789	342421	342863	343288	341477	343751
rnd1000_5010_16	358686	342485	4,52	23,85	2,20	342827	341353	343740	341054	335587	348389
rnd2500_2000_8	551417	490094	11,12	36,34	1,56	491233	488519	491669	495662	482362	496922
rnd2500_2001_8	556315	488608	12,17	35,61	1,15	488974	487325	489902	488973	481676	494296
rnd2500_2003_8	565019	509443	9,84	39,91	1,21	510389	508169	510717	513581	502849	515051
rnd2500_5503_16	917489	863803	5,85	139,14	0,60	864170	863216	864395	866006	860429	866392
rnd2500_5504_16	911852	864217	5,22	126,01	0,80	864998	862537	865927	866567	856234	871245
rnd2500_5507_16	915317	867227	5,25	121,90	0,49	867649	866577	867909	869803	864038	870373
rnd5000_2500_8	1122574	1003446	10,61	141,95	0,84	1004473	1001833	1005059	1006572	995046	1011322
rnd5000_2501_8	1092986	980497	10,29	137,22	0,72	981443	978929	982105	980779	971064	987089
rnd5000_2502_8	1099726	980541	10,84	139,27	0,52	981682	979310	981774	984869	973251	985529
rnd5000_6000_16	1832672	1744239	4,83	612,26	0,40	1744660	1743419	1745169	1746082	1739035	1748547
rnd5000_6001_16	1834179	1727498	5,82	614,86	0,49	1729416	1725330	1729680	1733701	1714451	1736483
rnd5000_6011_16	1821803	1726861	5,21	579,49	0,29	1727092	1726118	1727604	1724636	1722312	1730621
rnd7500_7002_8	1658493	1472391	11,22	389,03	0,79	1475463	1468575	1476207	1481846	1450465	1488891
rnd7500_7003_8	1685986	1505849	10,68	395,77	0,71	1507923	1503054	1508661	1516783	1492337	1517513
rnd7500_7004_8	1663303	1478691	11,10	368,00	0,61	1479218	1477332	1480150	1480067	1470887	1484927
rnd7500_7500_16	2741036	2575210	6,05	1647,85	0,24	2575247	2574978	2575508	2575867	2572615	2578186
rnd7500_7504_16	2754834	2590367	5,97	1638,86	0,30	2590963	2589528	2591206	2592593	2582870	2597363
rnd7500_7505_16	2727945	2578490	5,48	1725,47	0,59	2579268	2577331	2579649	2580878	2572531	2584838
rnd10000_3004_8	2213672	1989710	10,12	661,64	0,45	1990253	1988327	1991172	1989839	1982227	1996494
rnd10000_3005_8	2223503	1990850	10,46	737,79	0,54	1991305	1989793	1991915	1989552	1984735	1995444
rnd10000_3006_8	2232905	2005344	10,19	787,29	0,55	2006994	2002776	2007912	2015193	1994196	2016456
rnd10000_6502_16	3652280	3443416	5,72	2869,76	0,20	3443637	3442692	3444164	3444204	3439628	3446504
rnd10000_6504_16	3637625	3457089	4,96	2743,36	0,28	3458947	3454563	3459615	3464729	3443879	3468474
rnd10000_6507_16	3633827	3436809	5,42	3085,86	0,53	3439846	3432827	3440926	3443647	3412782	3454586

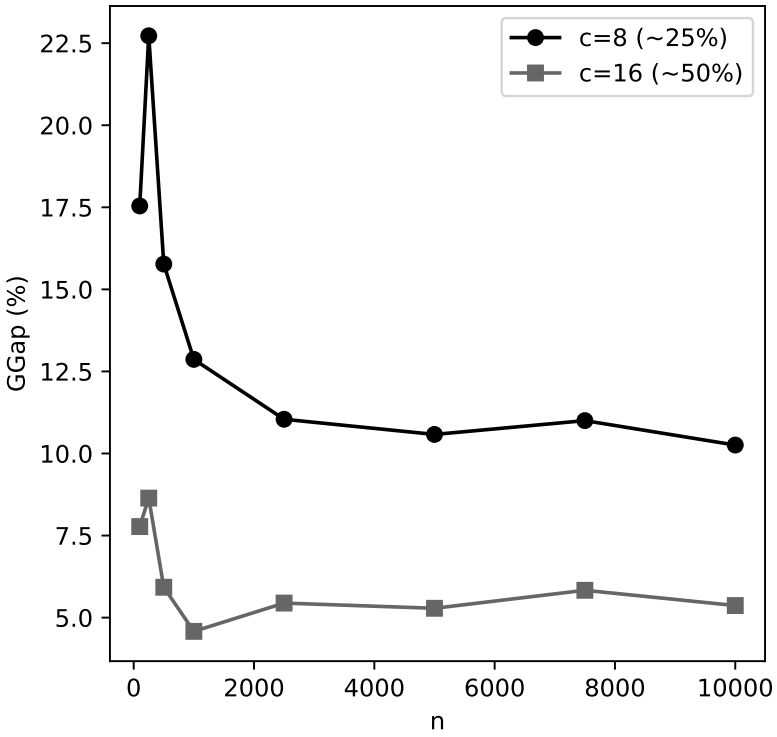
Taulukko 4.3 *Itse generoitujen ongelmien tulokset. Ongelman nimi on annettu muodossa "rnd<n>_<siemenluku>_<c>". MAXIMP-ratkaisimen keräämät pisteet, suoritusajat ja prosentuaalinen ero (GGap) generaattorin takaamaan optimiin. MAXIMP-ratkaisujen skaalattu keskihajonta SR, 1. ja 4. asteen linkkuveitsiestimaatit JK1 ja JK4 sekä niiden 95 %:n luottamusvälit.*

testiongelmissa, kun taas vastaavissa `rnd7500`-ongelmissa ($c = 16$) se on 5-6%. Generaattorin parametreja (erityisesti maksimikustannusta) tai kolmiojakauman tasajakaumaan vaihtamalla voisi todennäköisesti generoida vaikeampia ongelmia, mutta tässä käsitellyt ongelmat on generoitu vakioiduilla parametreilla, jotta jäljelle jäisi ainoastaan ongelmakoon vaikutus.

Tilastolliset optimiestimaatit ja niiden bootstrap-luottamusvälit eivät näytä olevan luotettavia ainakaan MAXIMP:n kaltaiselle melko ahneelle heuristiikalle, vaikka SR-statistiikkaa hyödyntäisikin esimerkiksi rajaamalla tapaukset $SR > 2$ tai $SR > 4$ tarkastelun ulkopuolelle Carlingin ja Xianglin mukaisesti [9]. Tämä saattaa johtua osittain valitusta huonoimman ratkaisun korvaavasta strategiasta, joka vähentää ratkaisujoukon diversiteettiä. Vielä suurempi ongelma saattaa olla houkuttavuusarvojen voimakas ratkaisua ohjaava vaikutus: MAXIMP-heuristiikka ei näytä vastaavan riittävästi tilastollisten menetelmien olettamaa satunnaista kokeilua. Taulukon 4.2 ongelmissa linkkuveitsiestimaattorit aliarvioivat optimin jokaisessa tapauksessa ja taulukon 4.3 tulokset ovat samankaltaisia. Ainoastaan tapaus $SR = 0$ (jolloin kaikki ylläpidetyt ratkaisut saavuttavat saman pistemäärän) näyttää olevan yhtä poikkeusta lukuun ottamatta taulukon 4.1 ongelmissa luotettava indikaatio todellisen optimin saavuttamisesta. Satunnaistamisellakin pehmennetty MAXIMP-heuristiikka aiheuttaa siis voimakkaan harhan tilastolliseen optimin estimointiin luvussa 2 ennakoidulla tavalla.

Kuvissa 4.1 ja 4.2 tarkastellaan MAXIMP-toteutuksen skaalautumista ongelmakoon suhteen. Ratkaisujen laadun osalta (kuva 4.1) on huomattavaa, että hieman epäintuitiivisesti ratkaisujen laatu paranee, jos ongelmakokoa kasvatetaan ja muut generointi- ja ratkaisinparametrit pidetään samoina: satunnaisuudella on enemmän merkitystä pienissä ongelmakoissa (kaarivaihtoehtoja on pienempi määrä jolloin yksittäisen valinnan suhteellinen vaikutus on suurempi). Laatu näyttää lähenevän tiettyä raja-arvoa isommissa ongelmakoissa ja on huonompi tapauksissa $c = 8$ kuin tapauksissa $c = 16$, mahdollisesti koska ongelmakoon ollessa vakioitu pienen solmujoukon valitsemisessa on enemmän valinnanvaraa kuin suuren solmujoukon valitsemisessa: MAXIMP-houkuttavuuskaavan perusteella tehdyt valinnat ovat harvoin optimaalisia. Ratkaisujen laadun osalta tuloksiin vaikuttaa siis suuresti vakioitu MAXIMP-iteraatioiden määrä ja generaattorin parametrit. Näiden ollessa vakioituja kuvaajan perusteella näyttäisi löytyvän tietty ongelmakoko (piikki $n = 250$ kohdalla), jolle ongelmat ovat kaikkein vaikeimpia. Parametreja muuttamalla olisi kenties mahdollista siirtää piikin sijaintia ja generoida vaikeampia suurikokoisia ongelmia.

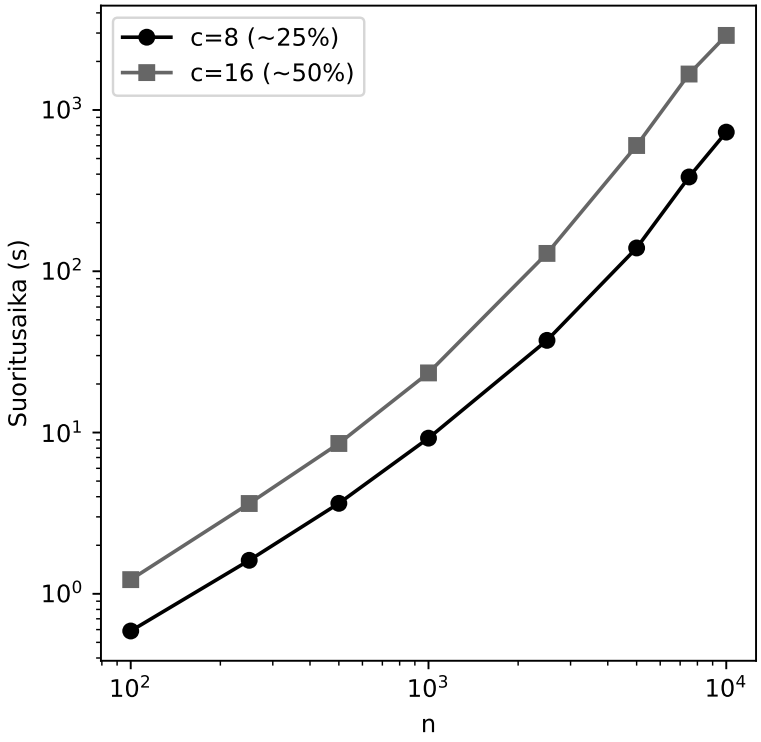
Suoritusajan osalta (kuva 4.2) voidaan todeta, että kuvaaja muistuttaa likimäärin suoraa, joka tarkoittaisi polynomista kasvua, koska kyseessä on log-log-kuvaaja. Suoritusaika kasvaa siis kohtuullisesti ongelmakoon kasvaessa ja MAXIMP-



Kuva 4.1 *Ratkaisujen keskimääräisen laadun (GGap) skaalautuminen ongelmakoon mukaan.*

toteutus on todennäköisesti käyttökelpoinen isommillekin ongelmako'oille. Jos sovelluskohteen ongelmat muistuttavat generaattorin tuottamia ongelmia, eli niissä ei ole esim. `pla7397`-ongelman kaltaista oletettavasti vaikeaa geometrista rakennetta, MAXIMP-heuristiikka skaalautuu hyvin.

Kaiken kaikkiaan toteutettu muunnelma MAXIMP-heuristiikasta näyttää olevan kohtalainen approksimaatio tilanteisiin, joissa ongelmakoon takia ei voi käyttää monimutkaisempaa heuristiikkaa tai eksaktia ratkaisinta ja ratkaisu tarvitaan lyhyessä ajassa. Mitä suurempi ongelma, sitä ahneemmaksi MAXIMP-heuristiikka muuttuu, koska vaihtoehtojen määrän kasvaessa ja p:n pysyessä vakiona suurin osa painotetun satunnaistamisen listalla olevista vaihtoehdoista on hyvin lähellä houkuttavuusjärjestyksen kärkeä. Tilastollisesta optimin estimoinnista ei näytä olevan merkittävää hyötyä MAXIMP-heuristiikan kohdalla, vaan iteraatioiden määrä täytyy asettaa jokseenkin mielivaltaisesti tavoitellun suoritusajan mukaan.



Kuva 4.2 *Keskimääräisen suoritusajan skaalautuminen ongelmakoon mukaan logaritmisella asteikolla.*

5 Pohdinta ja yhteenveto

Luvun 4 tulosten pohjalta näyttää siltä, että painotetulla satunnaistamisella ja estoilla saavutettu etu ei yksinään riitä viemään konstruktiivista heuristiikkaa ratkaisimien kärkeen. Tulokset ja teoria viittaavat useisiin mahdollisiin parannuksiin. Nykyisellään houkuttavuusarvojen laskenta on hyvin nopea operaatio, joten loogista olisi parantaa houkuttavuusarvoja esimerkiksi huomioimalla enemmän kuin kaksi solmua kerrallaan, vaikkakin muistinkäyttö kasvaa tällöin hyvin nopeasti. Useamman kuin kahden solmun huomioimiseksi voisi mahdollisesti ryhmitellä jollain tapaa lähekkäin olevia solmuja alueiksi ja laskea houkuttavuuden kolmen solmun sijaan solmuparin ja alueen yhdistelmälle, jolloin vaihtoehtojen määrä olisi $(n(n-1)/2)*a$, jossa a on alueiden määrä. Tällöin lisäykset tehtäisiin kaksi solmua kerrallaan yhden sijaan. Houkuttavuuskaavassa voisi käyttää myös useita alueita, jolloin houkuttavuus määräytyisi lähinnä tai pelkästään alueiden, ei yksittäisten solmujen houkuttavuuden mukaan. Alueellista resoluutiota vähentämällä olisi kenties mahdollista tallentaa esilaskettuun matriisiin useamman kuin kahden tai kolmen solmun yhdistelmien houkuttavuudet.

Toinen mahdollisuus olisi lisätä laskentaa ajon aikana tai soveltaa esilaskettuja houkuttavuusarvoja uudella tavalla, esimerkiksi tarkastelemalla useampaa kuin yhtä houkuttavuusarvoa per lisättävä solmu. Sopivan menetelmän löytäminen voi kuitenkin olla vaikeaa ja kirjallisuudesta löytyy ennestään melko hyviä vaihtoehtoja kuten EA4OP, jotka suorittavat enemmän laskentaa ajon aikana. Kolmas mahdollisuus olisi hyödyntää GPU-laskentaa heuristiikan kattaman hakuavaruuden laajentamiseksi pienentämällä geometrisen jakauman parametria p ja kasvattamalla ylläpidettyjen ratkaisujen määrää radikaalisti; optimaalinen parametrin p arvo ja sen yhteys iteraatioiden määrään olisi myös syytä määrittää tarkemmin. Jos rinnakkaisia ratkaisuja olisi tuhansia, iteraatioiden määrä voisi olla jopa 1. Esitetty ratkaisin rinnakkaistuu triviaalisti, mutta GPU-toteutus vaatisi käytettyjen tietorakenteiden suunnittelua uudelleen. On myös kyseenalaista, kuinka paljon raa'an voiman lisääminen voisi todella parantaa tuloksia.

Mielenkiintoinen täysin erilainen lähestymistapa olisi etsiä koneellisesti mahdollisten ratkaisinohjelmien avaruudesta parempia vaihtoehtoja, kuten on tehty esimerkiksi automatisoidun koneoppimisen kirjallisuudessa AutoML-Zero-menetelmässä [41]. Luvun 2 ongelmageneraattori muodostaisi selkeän tavoitefunktion prosessille, joka hakee parempia heuristiikkoja. Itse asiassa voidaan kuvitella jopa opetusmenetelmä, jossa heuristiikka ja generaattori kilpailevat nollasummapelissä, jossa generaattorin tavoite on tuottaa ongelmia (vaadittujen rajojen sisällä), joiden ratkaisemisessa heuristiikka suoriutuu huonosti ja heuristiikan tavoite on ratkaista ongelmat

mahdollisimman lähelle optimia. Hakuavaruus voisi olla joko suppea, jolloin tehtävä muistuttaisi esimerkiksi houkuttavuuskaavan symbolista regressiota, tai laaja mielivaltaisten ohjelmien haku, jolloin tehtävä vastaisi enemmän AutoML-Zeron lähestymistapaa [41]. Generaattorin suoritusaika on sitä luokkaa, että haku tai opetus olisi kenties mahdollista suorittaa järkevien laskentaresurssien puitteissa. Olettaen, että heuristiikka ei ylisovitu generaattoriin, tällainen heuristiikka olisi jossain mielessä paras mahdollinen. Generaattoria pitäisi kuitenkin ensin jatkokehittää teorian tasolla, jos sen nykyisellään antamat kustannusten alarajat tuottavat liian helppoja ongelmia voimakkaammalle heuristiikalle.

Tilastollisten menetelmien osalta todettiin, että ne eivät sovellu MAXIMP-heuristiikalle. Linkkuveitsimenetelmällä ei pysty arvioimaan ongelman todellista optimia, vaikka toteutettu heuristiikka hyödyntääkin painotettua satunnaistamista, jonka voisi ajatella vievän sen lähemmäksi satunnaista kokeilua. Tulos asettaa myös linkkuveitsimenetelmän soveltamisen muihin heuristiikkoihin hieman kyseenalaiseksi, koska menetelmä voi tuottaa selkeästi harhaisia tuloksia. Skaalatun keskihajonnan osalta tapaus $SR = 0$ oli välttävä indikaatio optimin saavuttamisesta, joten saattaisi kuitenkin olla mahdollista hyödyntää ratkaisujoukon antamaa informaatiota ainakin pysäytyskriteerin muodostamiseen. Tilastollisia menetelmiä pitäisi kehittää siten, että niiden taustaoletuksena olisi heuristiikan oikeasti toteuttama tavoitteellinen kokeilu, ei idealisoitu satunnaiskokeilumalli.

Kokonaisuudessaan tässä työssä on käsitelty suunnistusongelmaa yksityiskohtaisesti teorian tasolla ja toteutettu sille sekä generaattori että heuristinen ratkaisin, jota testattiin generaattorilla ja tunnetulla testipankilla. Ratkaisimeen yritettiin soveltaa tilastollisia optimiestimoinnin menetelmiä, mutta niistä ei saatu myönteisiä tuloksia. Ratkaisimen toteutus yhdistää erilaisia kirjallisuudessa esitettyjä ideoita uudella tavalla ja skaalautuu tehokkaasti suurikokoisille ongelmille, mutta sen suorituskyky jättää myös paljon parantamisen varaa. Generaattorin toteutus on mahdollisesti ensimmäinen laatuaan suunnistusongelmalle ja mahdollistaa heuristisen suunnistusongelmaratkaisimien tarkan vertailun suurikokoisilla ongelmilla, joiden eksakteja optimeja ei voida muilla tavoin määrittää kohtuullisessa ajassa. Kultakutri tietää siis näin ollen ainakin paremmin, kuinka lähellä parasta mahdollista puuroa ollaan.

Lähdeluettelo

[1] Emile Aarts ja Jan K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., 1997, s. 230–249, 298.

[2] Jeffrey L. Arthur ja James O. Frendewey. "Generating travelling-salesman problems with known optimal tours". *Journal of the Operational Research Society* 39.2 (1988), s. 153–159.

[3] Michel L. Balinski. "A competitive (dual) simplex method for the assignment problem". *Mathematical Programming* 34 (1986), s. 125–141.

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn ja Kurt Smith. "Cython: The best of both worlds". *Computing in Science & Engineering* 13.2 (2010), s. 31–39.

[5] Vahid Beiranvand, Warren Hare ja Yves Lucet. "Best practices for comparing optimization algorithms". *Optimization and Engineering* 18 (2017), s. 815–848.

[6] Frederick Bock. "An algorithm for solving travelling-salesman and related network optimization problems". Teoksessa: *Operations Research*. Vol. 6. 6. Informs, 1958, s. 897–897.

[7] Steven E. Butt ja Tom M. Cavalier. "A heuristic for the multiple tour maximum collection problem". *Computers & Operations Research* 21.1 (1994), s. 101–111.

[8] Vicente Campos, Rafael Martí, Jesús Sánchez-Oro ja Abraham Duarte. "GRASP with path relinking for the orienteering problem". *Journal of the Operational Research Society* 65 (2014), s. 1800–1813.

[9] Kenneth Carling ja Xiangli Meng. "On statistical bounds of heuristic solutions to location problems". *Journal of combinatorial optimization* 31 (2016), s. 1518–1549.

[10] Giorgio Carpaneto ja Paolo Toth. "Primal-dual algrorithms for the assignment problem". *Discrete Applied Mathematics* 18.2 (1987), s. 137–153.

[11] David G. Dannenbring. "Procedures for estimating optimal solution values for large combinatorial problems". *Management Science* 23.12 (1977), s. 1273–1283.

[12] Bradley Efron. "Bootstrap Methods: Another Look at the Jackknife". *The Annals of Statistics* 7.1 (1979), s. 1–26.

[13] Bradley Efron ja Trevor Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press, 2016, s. 185–190.

[14] João Ferreira, Artur Quintas, José A. Oliveira, Guilherme A. B. Pereira ja Luis Dias. "Solving the team orienteering problem: developing a solution tool using a genetic algorithm approach". Teoksessa: *Soft Computing in Industrial Applications: Proceedings of the 17th Online World Conference on Soft Computing in Industrial Applications*. Springer. 2014, s. 365–375.

[15] Matteo Fischetti, Juan Jose Salazar Gonzalez ja Paolo Toth. "Solving the orienteering problem through branch-and-cut". *INFORMS Journal on Computing* 10.2 (1998), s. 133–148.

[16] Ronald Aylmer Fisher ja Leonard Henry Caleb Tippett. "Limiting forms of the frequency distribution of the largest or smallest member of a sample". Teoksessa: *Mathematical proceedings of the Cambridge philosophical society*. Vol. 24. 2. Cambridge University Press. 1928, s. 180–190.

[17] Michel Gendreau, Gilbert Laporte ja Frédéric Semet. "A tabu search heuristic for the undirected selective travelling salesman problem". *European Journal of Operational Research* 106.2-3 (1998), s. 539–545.

[18] Angela P. Giddings, Ronald L. Rardin ja Reha Uzsoy. "Statistical optimum estimation techniques for combinatorial optimization problems: a review and critique". *Journal of Heuristics* 20 (2014), s. 329–358.

[19] Fred Glover. "Heuristics for integer programming using surrogate constraints". *Decision sciences* 8.1 (1977), s. 156–166.

[20] Bruce L. Golden ja Frank B. Alt. "Interval estimation of a global optimum for large combinatorial problems". *Naval Research Logistics Quarterly* 26.1 (1979), s. 69–77.

[21] Alex Grasas, Angel A. Juan, Javier Faulin, Jésica De Armas ja Helena Ramalhinho. "Biased randomization of heuristics using skewed probability distributions: A survey and some applications". *Computers & Industrial Engineering* 110 (2017), s. 216–228.

[22] David S. Johnson, Lyle A. McGeoch ja Edward E. Rothberg. "Asymptotic experimental analysis for the Held-Karp traveling salesman bound". Teoksessa: *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*. Vol. 341. ACM Press San Francisco. 1996, s. 350.

[23] A. A. Juan, A. Freixes, J. Panadero, C. Serrat ja A. Estrada-Moreno. "Routing drones in smart cities: A biased-randomized algorithm for solving the team orienteering problem in real time". *Transportation Research Procedia* 47 (2020), s. 243–250.

[24] Hans Kellerer, Ulrich Pferschy ja David Pisinger. ”Basic Algorithmic Concepts”. Teoksessa: *Knapsack Problems*. Springer Berlin Heidelberg, 2004, s. 15–42.

[25] Morteza Keshtkaran ja Koorush Ziarati. ”A novel GRASP solution approach for the Orienteering Problem”. *Journal of Heuristics* 22 (2016), s. 699–726.

[26] Gorka Kobeaga, María Merino ja Jose A. Lozano. ”An efficient evolutionary algorithm for the orienteering problem”. *Computers & Operations Research* 90 (2018), s. 42–59.

[27] Gorka Kobeaga, Jairo Rojas-Delgado, María Merino ja Jose A. Lozano. ”A revisited branch-and-cut algorithm for large-scale orienteering problems”. *European Journal of Operational Research* 313.1 (2024), s. 44–68.

[28] Siu Kwan Lam, Antoine Pitrou ja Stanley Seibert. ”Numba: A LLVM-based Python JIT compiler”. Teoksessa: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, s. 1–6.

[29] Adrienne C. Leifer ja Moshe B. Rosenwein. ”Strong linear programming relaxations for the orienteering problem”. *European Journal of Operational Research* 73.3 (1994), s. 517–523.

[30] Shen Lin ja Brian W. Kernighan. ”An effective heuristic algorithm for the traveling-salesman problem”. *Operations research* 21.2 (1973), s. 498–516.

[31] Shih-Wei Lin ja F. Yu Vincent. ”A simulated annealing heuristic for the team orienteering problem with time windows”. *European Journal of Operational Research* 217.1 (2012), s. 94–107.

[32] Miles Lubin ja Iain Dunning. ”Computing in operations research using Julia”. *INFORMS Journal on Computing* 27.2 (2015), s. 238–248.

[33] Yannis Marinakis, Michael Politis, Magdalene Marinaki ja Nikolaos Matsatsinis. ”A memetic-grasp algorithm for the solution of the orienteering problem”. Teoksessa: *Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences-MCO 2015-Part II*. Springer. 2015, s. 105–116.

[34] Jiří Matoušek ja Bernd Gärtner. *Understanding and using linear programming*. Vol. 1. Springer, 2007, s. 84–86, 204.

[35] Robert L. Nydick Jr. ja Howard J. Weiss. ”An analytical evaluation of optimal solution value estimation procedures”. *Naval Research Logistics (NRL)* 41.2 (1994), s. 189–202.

[36] Krzysztof Ostrowski, Joanna Karbowska-Chilinska, Jolanta Koszelew ja Pawel Zabielski. "Evolution-inspired local improvement algorithm solving orienteering problem". *Annals of Operations Research* 253 (2017), s. 519–543.

[37] Javier Panadero, Majsa Ammouriova, Angel A. Juan, Alba Agustin, Maria Nogal ja Carles Serrat. "Combining parallel computing and biased randomization for solving the team orienteering problem in real-time". *Applied Sciences* 11.24 (2021), s. 12092.

[38] Martha G. Pilcher ja Ronald L. Rardin. "Partial polyhedral description and generation of discrete optimization problems with known optima". *Naval Research Logistics (NRL)* 39.6 (1992), s. 839–858.

[39] Ronald L. Rardin. *Optimization in Operations Research*. Pearson Education, 2016, s. 897–902.

[40] Ronald L. Rardin ja Reha Uzsoy. "Experimental evaluation of heuristic optimization algorithms: A tutorial". *Journal of Heuristics* 7 (2001), s. 261–304.

[41] Esteban Real, Chen Liang, David So ja Quoc Le. "AutoML-Zero: Evolving machine learning algorithms from scratch". Teoksessa: *International conference on machine learning*. PMLR. 2020, s. 8007–8019.

[42] L. Reyes-Rubiano, A. A. Juan, Christopher Bayliss, Javier Panadero, J. Faulin ja P. Copado. "A biased-randomized learnheuristic for solving the team orienteering problem with dynamic rewards". *Transportation Research Procedia* 47 (2020), s. 680–687.

[43] D. S. Robson ja J. H. Whitlock. "Estimation of a truncation point". *Biometrika* 51.1/2 (1964), s. 33–39.

[44] Hao Tang ja Elise Miller-Hooks. "A tabu search heuristic for the team orienteering problem". *Computers & Operations Research* 32.6 (2005), s. 1379–1407.

[45] M. Faith Tasgetiren. "A Genetic Algorithm with an Adaptive Penalty Function for the Orienteering Problem." *Journal of Economic & Social Research* 4.2 (2002).

[46] Theodore Tsiligirides. "Heuristic methods applied to orienteering". *Journal of the Operational Research Society* 35 (1984), s. 797–809.

[47] Fang Zhao, Zheqian Hu ja Xiangli Meng. "Statistical Bound of Genetic Solutions to Quadratic Assignment Problems". Teoksessa: *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. IEEE. 2022, s. 94–101.

Tekoälyn käytöstä

Tässä työssä on käytetty GitHub Copilot-työkalua ratkaisimen ja generaattorin kehityksen aikana. Työkalu perustuu kielimalliin, joka suosittelee koodiriville sopivia lopetuksia. Tutkielman kirjoittamiseen ei ole hyödynnetty tekoälyä.

LIITE A. Suunnistusongelmageneraattorin lähdekoodi

```
import math
import numpy as np
from numba import njit
import numba
import os

@njit(inline='always')
def calc_bitarray_size(n):
    return (n + 7) // 8

@njit(inline='always')
def set_bit(arr, i, j, value):
    byte_index = j // 8
    bit_index = j % 8
    if value == 1:
        arr[i][byte_index] |= 1 << bit_index
    else:
        arr[i][byte_index] &= ~(1 << bit_index)

@njit(inline='always')
def get_bit(arr, i, j):
    byte_index = j // 8
    bit_index = j % 8
    return (arr[i][byte_index] >> bit_index) & 1

@njit(parallel=True)
def knapsack_dynamic_programming(values, weights, capacity):
    n = len(values)

    # last two rows of the dynamic programming table
    # alternate between rows by using i modulo 2
    dp = np.zeros((2, capacity + 1), dtype=np.uint32)
    # save binary choices for backtracking
    # the idea for this space optimization
    # is from https://bayesianneuron.com/2019/02/0-1-knapsack-
        discrete-optimization-dynamic-programming
    choices = np.zeros((n + 1, calc_bitarray_size(capacity + 1)),
        dtype=np.uint8)

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
```

```
                newVal = max(dp[(i-1) % 2][w], values[i - 1] + dp[(i
                    -1) % 2][w - weights[i - 1]])
42              assert newVal <= 4294967295, "Overflow detected!"
                dp[i%2][w] = newVal
44              choice_result = int(dp[(i-1) % 2, w] != dp[i % 2, w
                    ])
                set_bit(choices, i, w, choice_result)
46          else:
                dp[i % 2][w] = dp[(i-1) % 2][w]
48              set_bit(choices, i, w, 0)

50      selected_items = []
        i, w = n, capacity
52      while i > 0 and w > 0:
            if get_bit(choices, i, w) == 1:
54              selected_items.append(i - 1)
                w -= weights[i - 1]
56          i -= 1

58      selected_items.reverse()
        return dp[n%2][-1], selected_items
60
    DEBUG = True
62  @njit()
    def debug_print(*msg):
64      if DEBUG:
            print(*msg)
66
    @njit(parallel=False)
68  def generate_tsp(selected_items, selected_weights, selected_values,
        d_max, d_min=10, swaps=500, tries=10000, rho=0.05):
        alpha = np.zeros(len(selected_weights), dtype=np.float64)
70      distance_matrix = np.full((len(selected_weights), len(
            selected_weights)), -1, dtype=np.int32)

72      selected_weights = np.roll(selected_weights, -len(
            selected_weights) // 2)  # shift left so the original sorted
                 weights don't produce negative values in the alpha
                calculation
74      selected_values = np.roll(selected_values, -len(selected_values)
            // 2)
        selected_items = np.roll(selected_items, -len(selected_items) //
            2)
76
        n = len(selected_weights)
78      assert len(selected_weights) == len(selected_values) == len(
            selected_items)
```

```
80      rho_alpha = rho * d_max

82      failed_swaps = 0

84      for swap_count in range(0, swaps):
            swap1 = np.random.randint(0, len(selected_weights))
86          swap2 = np.random.randint(0, len(selected_weights))
            while swap1 == swap2:
88              swap2 = np.random.randint(0, len(selected_weights))
            tmp1 = selected_weights[swap1]
90          tmp2 = selected_weights[swap2]
            selected_weights[swap1] = tmp2
92          selected_weights[swap2] = tmp1

94          tmp1_val = selected_values[swap1]
            tmp2_val = selected_values[swap2]
96          selected_values[swap1] = tmp2_val
            selected_values[swap2] = tmp1_val
98
            tmp1_item = selected_items[swap1]
100         tmp2_item = selected_items[swap2]
            selected_items[swap1] = tmp2_item
102         selected_items[swap2] = tmp1_item

104         alpha[0] = 0.0

106         # from 1 to n, wraparound because this is using previous
                values of alpha
            # (alpha=beta in the original paper) to calculate the next
                one instead of the other way around as in the original
                paper
108         for k in range(0, n):
                alpha[(k + 1) % n] = selected_weights[k] - alpha[k]
110
            alpha0_implied_distance = alpha[0] + alpha[1]
112         dist_offset = selected_weights[
                              0] - alpha0_implied_distance  # difference
                                   between wanted distance k0 and
                                  distance implied by (alpha0 and alpha1
                                  )
114
            alpha[0] = alpha[0] + dist_offset / 2.0  # offset here to
                fix first segment of tour
116
            # from 1 to n, wraparound because this is using previous
                values of alpha
```

```
118         # (alpha=beta in the original paper) to calculate the next
                one instead of the other way around as in the original
                paper
            for k in range(0, n):
120             alpha[(k + 1) % n] = selected_weights[k] - alpha[k]

122         success = True

124         # first check that the distance implied by alpha[0] and
                alpha[1] for the first segment of the tour is correct
            if math.fabs(alpha[0] + alpha[1] - selected_weights[0]) >
                0.01:
126             success = False

128         for i in range(0, len(alpha)):
                if alpha[i] <= rho_alpha:
130                 success = False
                    break
132
            if not success:
134             failed_swaps += 1
                selected_weights[swap1] = tmp1
136             selected_weights[swap2] = tmp2
                selected_values[swap1] = tmp1_val
138             selected_values[swap2] = tmp2_val
                selected_items[swap1] = tmp1_item
140             selected_items[swap2] = tmp2_item
            else:
142             failed_swaps = 0

144         if failed_swaps > tries:
                break
146
        alpha[0] = 0.0
148
        # from 1 to n, wraparound because this is using previous values
            of alpha
150     # (alpha=beta in the original paper) to calculate the next one
            instead of the other way around as in the original paper
        for k in range(0, n):
152         alpha[(k + 1) % n] = selected_weights[k] - alpha[k]

154     alpha0_implied_distance = alpha[0] + alpha[1]
        dist_offset = selected_weights[
156                       0] - alpha0_implied_distance  # difference
                              between wanted distance k0 and distance
                              implied by (alpha0 and alpha1)
```

```
158     alpha[0] = alpha[0] + dist_offset / 2.0  # offset here to fix
            first segment of tour

160     # from 1 to n, wraparound because this is using previous values
            of alpha
        # (alpha=beta in the original paper) to calculate the next one
            instead of the other way around as in the original paper
162     for k in range(0, n):
            alpha[(k + 1) % n] = selected_weights[k] - alpha[k]
164
        dists = []
166     for i in range(0, n):
            val = alpha[i] + alpha[(i + 1) % n]
168         distance_matrix[i, (i + 1) % n] = val
            distance_matrix[(i + 1) % n, i] = val
170         dists.append(val)
        debug_print("Dists", dists)
172
        debug_print("Selected swapped: ", selected_weights)
174     debug_print("Pre:", alpha)

176     # return None if any distances differ from selected weights
        # this shouldn't happen but it would mean we definitely need to
            retry generation
178     for i in range(0, len(selected_weights)):
            if math.fabs(selected_weights[i] - dists[i]) > 0.01:
180             debug_print("Error: calculated distance != selected
                    weight", i, selected_weights[i], dists[i])
                return None
182
        # return None for failed alpha calculations
184     for i in range(0, len(alpha)):
            if alpha[i] <= rho_alpha:
186             debug_print("Error: alpha < rho_alpha")
                return None
188
        min_alphabeta = np.min(alpha)  # alpha and beta are the same
190     max_alphabeta = np.max(alpha)
        debug_print("min alpha beta: ", min_alphabeta)
192     debug_print("max alpha beta: ", max_alphabeta)

194     for i in range(0, n):
            for j in range(i + 1, n):
196             if distance_matrix[i, j] != -1:
                    continue
```

```
198             val = math.ceil(np.random.uniform(alpha[i] + alpha[j],
                    d_max))  # ceil to avoid numerical errors
                distance_matrix[i, j] = val
200             distance_matrix[j, i] = val

202     for i in range(0, n):
            distance_matrix[i, i] = 0
204
        return distance_matrix, dists, selected_items, selected_weights,
            selected_values
206
    @njit()
208 def calc_tour_length(tour, distance_matrix):
        tour_length = 0
210     for i in range(0, len(tour)):
            tour_length += distance_matrix[tour[i], tour[(i + 1) % len(
                tour)]]
212     return tour_length

214 @njit
    def set_seed(value):
216     np.random.seed(value)

218 @njit(parallel=False)
    def construct_top(selected_items, selected_weights, selected_values,
        unselected_items, unselected_weights,
220                   unselected_values, d_max, d_min=10, swaps=500,
                          tries=10000, rho=0.05):
        result = generate_tsp(selected_items, selected_weights,
            selected_values, d_max, d_min, swaps, tries, rho)
222
        if result is None:
224         return None

226     selected_distance_matrix, selected_dists, selected_items,
            selected_weights, selected_values = result

228     n = len(selected_weights) + len(unselected_weights)

230     full_distance_matrix = np.full((n, n), 0, dtype=np.uint16)

232     for i in range(0, n):
            for j in range(i + 1, n):
234             if i < len(selected_weights):
                    if j < len(selected_weights):  # between selected
                        and selected
```

```
236                     full_distance_matrix[i, j] =
                            selected_distance_matrix[i, j]
                        full_distance_matrix[j, i] =
                            selected_distance_matrix[j, i]
238                 else:  # between selected and unselected;
                        selected_dists[i] should in any case equal
                        selected_weights[i]
                        # OPTIONAL uniform randomization
240                     # val = math.ceil(np.random.uniform(max(
                            selected_dists[i], unselected_weights[j-len(
                            selected_weights)]) + 1, d_max))
                        minval = max(selected_dists[i],
242                                  unselected_weights[j - len(
                                         selected_weights)]) + 0.01  #
                                         epsilon of 0.01
                        val = math.ceil(np.random.triangular(minval,
                            minval, d_max))
244                     full_distance_matrix[i, j] = val
                        full_distance_matrix[j, i] = val
246
            else:  # between unselected and unselected
248             # OPTIONAL uniform randomization
                # val = math.ceil(np.random.uniform(max(
                    unselected_weights[i-len(selected_weights)],
                    unselected_weights[j-len(selected_weights)]) +
                    1, d_max))
250             minval = max(unselected_weights[i - len(
                    selected_weights)],
                             unselected_weights[j - len(
                                 selected_weights)]) + 0.01  #
                                 epsilon of 0.01
252             val = math.ceil(np.random.triangular(minval, minval,
                     d_max))
                full_distance_matrix[i, j] = val
254             full_distance_matrix[j, i] = val

256 alt_sol = list(range(0, len(selected_weights)))
    alt_sol_dist = calc_tour_length(alt_sol, full_distance_matrix)
258 print("Alternative solution dist: ", alt_sol_dist)

260 combined_values = np.concatenate((selected_values,
        unselected_values))
    combined_indices = np.concatenate((selected_items,
        unselected_items))
262
    reindexed_distance_matrix = np.full((n, n), 0, dtype=np.uint16)
264 reindexed_values = np.full(n, 0, dtype=np.int32)
```

```
    for x in range(0, n):
266         for y in range(0, n):
                reindexed_distance_matrix[combined_indices[x],
                    combined_indices[y]] = full_distance_matrix[x, y]
268         reindexed_values[combined_indices[x]] = combined_values[x]

270     # distance matrix, point values, optimal tour
        return reindexed_distance_matrix, reindexed_values,
            selected_items
272
def generate_top(SEED, size=100, capacity_per_item=250, prefix="",
    value_max=1000, d_max=1000):
274     np.random.seed(SEED)
        set_seed(SEED)
276
        values = np.random.randint(1, value_max, size=size)
278
        sigma = 0.25
280     rho_minimum = 0.05  # 0.075 works
        rho_selected = 0.1  # 0.17 works
282
        weights = np.random.randint(2 * int(rho_selected * d_max), 2 *
            int(sigma * d_max), size=size)
284
        capacity = int(capacity_per_item * size)
286     print("Capacity: ", capacity)

288     max_value, selected_items = knapsack_dynamic_programming(values,
            weights, capacity)

290     if len(selected_items) % 2 == 0:
            print(
292             "Knapsack solution has even number of items which would
                    result in an imbalance during wraparound, retrying."
                    )
            return False
294
        unselected_items = np.random.permutation(np.asarray([i for i in
            range(0, size) if i not in selected_items]))
296     print(unselected_items)

298     unselected_weights = np.asarray(weights[unselected_items])
        unselected_values = np.asarray(values[unselected_items])
300
        # randomly shuffle selected items
302     selected_items = np.random.permutation(selected_items)
        selected_weights = np.asarray(weights[selected_items])
```

```
304
    # sort selected items by weight
306 selected_items = selected_items[selected_weights.argsort()]  #
        sort selected items by weight

308 # get the weights of the selected items
    selected_weights = np.asarray(weights[selected_items])
310
    # get the values of the selected items
312 selected_values = np.asarray(values[selected_items])

314 # generate the distance matrix
    result = construct_top(selected_items, selected_weights,
        selected_values, unselected_items, unselected_weights,
316                        unselected_values, d_max, rho=rho_minimum
                               , swaps=50000, tries=10000)

318 if result is None:  # skip invalid
        print("Failed")
320     return False

322 distance_matrix, values, optimal_tour = result

324 optimal_tour_distance = calc_tour_length(optimal_tour,
        distance_matrix)
    sum_of_weights = np.sum(selected_weights)
326 print("Sum of weights: ", sum_of_weights)
    print("n selected: ", len(selected_weights))
328 print("Optimal tour distance: ", optimal_tour_distance)
    print("Optimum:", sum(selected_values))
330 assert max_value == sum(selected_values), "Knapsack solution
        value does not match sum of selected item values"
    assert sum_of_weights == optimal_tour_distance, "Sum of weights
        does not match optimal tour distance"
332
    # ensure out dir exists
334 if not os.path.exists("/mnt/scratch2/out"):
        os.makedirs("/mnt/scratch2/out")
336
    with open("/mnt/scratch2/out/" + prefix + "_" + str(SEED) + "_"
        + str(capacity_per_item) + ".txt", 'w') as f:
338     f.write(f"n {size}\n")
        f.write("m 1\n")
340     f.write(f"tmax {capacity}\n")
        f.write("solution " + " ".join([str(i) for i in optimal_tour
            ]) + "\n")
342     f.write(f"optimum {sum(selected_values)}\n")
```

```
            f.write(f"optimumTime {optimal_tour_distance}\n")
344         f.write(f"scores " + " ".join([str(i) for i in values]) + "\
                n")
            #for i in range(0, len(distance_matrix)):
346         #    f.write(" ".join([str(j) for j in distance_matrix[i]])
                + "\n")
        # write distance matrix
348     np.save("/mnt/scratch2/out/" + prefix + "_" + str(SEED) + "_" +
            str(capacity_per_item) + ".npy", distance_matrix)

350     return True

352 def start_generation(SEED, size, problems_needed=5,
        capacity_per_item=150,

        value_max=1000, d_max=1000):
        success = False
354     success_count = 0
        while success_count < problems_needed:
356         success = generate_top(SEED, size=size, capacity_per_item=
                capacity_per_item, prefix="rnd" + str(size), value_max=
                value_max, d_max=d_max)
            if success:
358             success_count += 1
            SEED += 1
360         print("Success: ", success)

362     print("Seed: ", SEED - 1)

364 if __name__ == "__main__":
        seed = 0
366     steps = [100, 250, 500, 1000, 2500, 5000, 10000, 7500]
        for step in steps:
368         start_generation(seed, step, problems_needed=3,
                capacity_per_item=8, value_max=1000, d_max=100)
            seed += 500
370     for step in steps:
            start_generation(seed, step, problems_needed=3,
                capacity_per_item=16, value_max=1000, d_max=100)
372         seed += 500
```