Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jakob Gaberc Artenjak

Pregled hevristik za problem trgovskega potnika

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Borut Robič

Ljubljana, 2020

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite in analizirajte hevristične metode in algoritme za reševanje problema trgovskega potnika. Poiščite njihove prednosti in slabosti ter razmislite o njihovih morebitnih izboljšavah. Delovanje algoritmov eksperimentalno ovrednotite.

Zahvaljujem se vsem, ki so mi omogočili študij in čas za izdelavo diplomske naloge.

Kazalo

Povzetek

Naslov: Pregled hevristik za problem trgovskega potnika

Avtor: Jakob Gaberc Artenjak

Problem trgovskega potnika je iskanje najkrajše poti med vsemi mesti, kjer obiščemo vsako mesto natanko enkrat in se vrnemo nazaj na začetno mesto. Na problem lahko gledamo kot na iskanje najcenejšega cikla v grafu, ki obišče vse točke natanko enkrat.

Pridobivanje optimalne rešitve problema trgovskega potnika je praktično neuporabno zaradi časovne zahtevnosti problema. Hevristični algoritmi so dobra alternativa optimalnemu reševanju problema, saj pridobijo rešitev v praktično izvedljivem času, a izgubijo jamstvo optimalne rešitve.

Uvod diplomske naloge vsebuje osnovni opis problema trgovskega potnika. Temu sledijo primeri praktične uporabe problema, natančnejši opis problema, opredelitev hevristik in opis testnega okolja.

Glavni del naloge vsebuje šest hevurističnih algoritmov, ki smo jih implementirali in jih testirali. Izbrali smo algoritem Lokalnega iskanja z operacijo k-opt, Lin-Kernighanov algoritem, algoritem Simuliranega ohlajanja, algoritem Kolonije mravelj, algoritem Optimizacije z roji delcev in algoritem Oponašanja volkov.

Končni del naloge vsebuje primerjavo eksperimentalnih rezultatov in komentar nad uporabljeno metodologijo za primerjavo algoritmov.

Ključne besede: hevristika, Problem trgovskega potnika, simetrični Problem trgovskega potnika, Lokalno iskanje, k-opt, Lin-Kerninghan, Simulirano

ohlajanje, Kolonija mravelj, Optimizacija z roji delcev, Oponašanje volkov.

Abstract

Title: Overview of heuristics for The Traveling Salesman Problem

Author: Jakob Gaberc Artenjak

The Traveling Salesman Problem is finding the shortest path through all the cities, where each city is visited precisely once, while the first and the last city are the same. This can be formulated as searching for the shortest cycle in a graph which visits each vertex exactly once.

Finding the optimal solution is practically fruitless due to the time complexity of the problem. Heuristic algorithms are good alternatives to algorithms, which search for the optimal solution, because a solution can be found in a practically achievable time frame; however, the guarantee of the solution being optimal is lost.

The introduction of this work includes a basic description of The Traveling Salesman Problem, which is followed by a list of practical applications, a detailed description of the problem, the classification of heuristic algorithms and the details of the experimental environment.

The main part of this work includes six algorithms, which were implemented and tested. The selected algorithms are Local Search with the k-opt operation; Lin-Kernighan algorithm; Simulated Annealing; Ant Colony Algorithm; Particle Swarm Optimization and Wolfpack algorithm.

The final part of this work is a comparison of the results from each algorithm and a commentary on the methodology that was used for the comparison of the algorithms.

Keywords: heuristic, Traveling Salesman Problem, symetric Traveling Salesman Problem, Local search, k-opt, Lin-Kerninghan, Simulated annealing, Ant colony, Particle swarm optimization, Wolfpack.

Poglavje 1

Uvod

Problem trgovskega potnika je enostavno razumeti. Imamo N mest in določene razdalje med njimi. Naša naloga je najti najkrajšo pot, ki obišče vsako mesto natanko enkrat in se ob koncu poti vrne nazaj v začetno mesto.

Problem trgovskega potnika pogosto srečamo v vsakdanjem življenju. Se odpravimo po nakupih? Želimo obiskati vse turistične točke v počitniškem mestu? Kako naj obiščemo vse tobogane v kopališču, da bomo pri tem prehodili najkrajšo pot? Pri navedenih problemih poskušamo najti najkrajši vrstni red obiska določenih prostorov, pri čemer se želimo ob koncu poti vrniti na izhodiščno točko. Po obisku vseh trgovin se želimo vrniti nazaj domov, po obisku vseh turističnih točk se želimo vrniti nazaj v hotel, po obisku vseh toboganov se želimo vrniti nazaj na svojo brisačo pod senco.

Če imamo na voljo manjše število mesto, lahko problem rešimo intuitivno. Pri treh mestih je optimalna rešitev trivialna, saj imamo le eno možnost. Pri štirih mestih imamo tri možne rešitve. Kaj pa za $N > 20$ mest? Število različnih rešitev za problem trgovskega potnika z N mesti je $\frac{(N-1)!}{2}$. Kadar ima problem veliko mest, intuitivno reševanje odpade in moramo najti bolj rigorozen način reševanja.

Na tem mestu nastopijo algoritmi. Naiven algoritem za reševanje problema trgovskega potnika je ustvariti vseh $\frac{(N-1)!}{2}$ možnih rešitev in ugotoviti, katera pot je najkrajša. Tak algoritem je, kakor rečeno, naiven. Pri problemu

z $N = 50$ je število možnih rešitev dolgo 63 števk. Ustvariti, oceniti in primerjati vse možne rešitve postane praktično neizvedljivo.

Izkaže se, da problem trgovskega potnika pripada skupini NP-težkih[1] problemov, kar pomeni, da zanj ne poznamo algoritma, ki bi v polinomskem času pridobil optimalne rešitve, in predpostavimo, da takšen algoritem sploh ne obstaja. Zaradi tega se za reševanje NP-težkih problemov (vključno s problemom trgovskega potnika) uporabljajo hevristični algoritmi. Hevristični algoritmi so algoritmi, ki vrnejo čim boljšo rešitev, ki pa ni nujno optimalna. Ta olajšava jim omogoča delovanje v praktično sprejemljivem času.

Za pregled zgodovine problema trgovskega potnika in njegovega zgodnjega reševanja priporočamo delo [7].

Diplomska naloga je razdeljena na enajst poglavij. Prvo poglavje je uvod. Drugo poglavje vsebuje primere praktične uporabe problema trgovskega potnika. Tretje poglavje opiše različne zvrsti problema, vsebuje definicijo za izbrano zvrst problema, opiše lastnosti povezav problema, prikaže možne načine predstavitve rešitev problema, definira različne zvrsti hevristik problema in opredeli pojem hevristike ter opiše testno okolje. Naslednjih šest poglavij vsebuje algoritme. Predstavljeni so algoritem Lokalnega iskanja, Lin-Kernighanov algoritem, algoritem Simuliranega ohlajanja, algoritem Kolonije mravelj, algoritem Optimizacije z roji delcev in algoritem Oponašanja volkov. Posamezna poglavja algoritmov sprva predstavijo idejo algoritma in postopoma prikazujejo aplikacijo ideje v algoritem za reševanje problema trgovskega potnika. Na koncu posameznih poglavij algoritmov so predstavljeni rezultati algoritma s komentarjem le-tega, možne spremembe ali izboljšave algoritma in navedena uporabljena in dodatna literatura algoritma. Deseto poglavje vsebuje primerjavo rezultatov in komentar o uporabljeni metodologiji primerjave algoritmov. Enajsto poglavje je zaključek.

[1] angl. NP-hard

Poglavje 2

Uporaba v praksi

Zgodnja praktična uporaba problema trgovskega potnika je bila, kakor pove že ime, namenjena trgovskim potnikom. Trgovski potniki so želeli obiskati določena mesta, kjer so trgovali z izdelki, in se nato v najkrajšem možnem času vrniti domov.

Ta panoga v sedanjosti ni več tako aktualna, a obstajajo druge veščine, ki jim optimalnejše reševanje problema izboljša produktivnost:

Vozni red avtobusov – Imamo avtobus in želimo najti najkrajšo pot, po kateri bomo obiskali vse postaje in se vrnili nazaj na izhodiščno avtobusno postajo.

Oblikovanje kristalov s pomočjo laserjev – Imamo kristal, ki ga želimo preoblikovati. Podane imamo točke, po katerih moramo z laserskim žarkom zvrtati kristal. Naloga je najti najkrajšo pot glave laserja, da obišče vse vrtalne točke in se vrne nazaj v neaktivno stanje.

Luknjanje vezij – Imamo načrt vezja. Želimo najti najkrajšo pot glave vrtalnika tako, da bo obiskal vse točke, ki jih želimo prevrtati, in se vrnil nazaj v neaktivno stanje.

Pošta – Najti želimo najkrajšo pot, po kateri poštar obišče vse hiše, za katere ima pošto, in se vrne nazaj v poštni center, v katerem je začel

svojo pot. V določenih primerih, kadar so hiše skupaj v gručah (npr. bloki), lahko več točk obravnavamo kot eno.

Pregled – Imamo določene točke, ki jih želimo pregledovati zaradi npr. zbiranja informacij ali preverjanja pravilnega delovanja. Najti želimo najkrajšo pot, ki obišče vse točke in se vrne nazaj v začetno bazo, iz katere smo začeli.

Ciljanje teleskopov – Imamo astronomske observatorije, ki redno preverjajo stanje določenih zvezd. Najti želimo vrstni red preverjanja izbranih zvezd, pri katerem minimalno spreminjamo pozicijo teleskopa, ob koncu pregleda pa se teleskop vrne v neaktivno stanje.

Odvoz predmetov iz skladišča na drugo lokacijo – Imamo predmete, ki jih želimo vzeti iz skladišča in jih dostaviti na drugo lokacijo. Najti želimo najkrajši vrstni red odvzema izbranih predmetov iz skladišča s pomočjo vozila, tako da bo pot vozila najkrajša, ob odvzemu vseh izbranih predmetov iz skladišča pa se vrnemo na lokacijo prevzema predmetov za nadaljnjo dostavo.

Vsak problem, ki ga lahko formuliramo kakor iskanje najcenejše poti skozi več stanj, kjer stanja obiščemo natanko enkrat, začnemo in končamo pa v izhodiščnem stanju, lahko rešujemo s pomočjo algoritmov za problem trgovskega potnika.

Za širši in podrobnejši pregled navedenih in dodatnih primerov uporabe problema trgovskega potnika v praksi priporočamo dela [6, 27, 20].

Poglavje 3

Definicije, pojmi in testno okolje

3.1 Zvrsti problema trgovskega potnika

Problem trgovskega potnika (v nadaljevanju PTP[1]) ima več različnih zvrsti. V tej nalogi bomo pregledovali algoritme za standarden PTP. Standarden PTP išče najkrajši cikel v polnem grafu, ki obišče vsako mesto natanko enkrat. Imamo tudi druge zvrsti PTP-ja, na primer:

- splošni PTP[2], ki mora obiskati natanko eno mesto v vseh podskupinah mest;
- nepopoln PTP, kjer podani graf ni poln;
- PTP z ozkim grlom, kjer želimo pridobiti cikel, pri katerem je najdražja povezava minimalna.

Daljši seznam vrst PTP-jev se najde v delu [27].

Problem usmerjanja vozil[3] je generaliziran PTP, pri katerem so dodane dodatne omejitve, kot sta recimo časovna omejitev doseganja določenih mest

[1] angl. TSP, kratica za Traveling Salesman Problem
[2] angl. General Traveling Salesman Problem
[3] angl. Vehicle Routing Problem

ali več možnih vozil.

Vse navedene zvrsti PTP-ja se lahko pretvorijo v standarden PTP, zato so v enakem razredu časovne zahtevnosti, a takšne pretvorbe niso priporočljive, saj po navadi dodajo dodatna mesta. Bolje je uporabiti algoritem, ki zna izkoristiti specifične lastnosti PTP-ja določene zvrsti.

3.2 Definicija problema trgovskega potnika

Standarden PTP z N mesti definiramo s polnim grafom $G = (V, E)$, pri katerem vozlišča $V = \{1, 2, ..., N\}$ predstavljajo mesta, povezave v $E = \{(i, j) \in V^2\}$ predstavljajo povezave med mesti in dodatno matriko cen $C_{N \times N}$, kjer vsak element $c_{i,j} \in \mathbb{R}$ predstavlja dolžino povezave od mesta i do mesta j. Naša naloga je najti Hamiltonov cikel, kjer je vsota vseh cen povezav najnižja.

3.3 Lastnost povezav problema trgovskega potnika

PTP katere koli zvrsti je lahko simetričen ali asimetričen. Za simetričen PTP velja, da je cena povezave od mesta i do mesta j enaka ceni povezave od mesta j do mesta i, $c_{i,j} = c_{j,i}$. Za asimetrične PTP-je pa to ne drži. Asimetrične PTP-je lahko transformiramo v simetrične, a moramo dodati dodatna mesta. Postopek transformacije je opisan v delu [17].

Cene povezav so lahko podane vnaprej za vsako povezavo ali pa izračunane s pomočjo enačbe in dodanih informacij mesta, kot so npr. koordinate mest.

Dve pogosto uporabljeni funkciji za izračun cen povezav s pomočjo podanih koordinat mest sta:

evklidska razdalja – Pri podanih koordinatah mest je cena povezave od mesta i do mesta j pridobljena z enačbo
$c_{j,i} = c_{i,j} = \sqrt{(i_1 - j_1)^2 + (i_2 - j_2)^2 + ... + (i_N - j_N)^2}$ pri čemer je i_t

t-ta koordinata mesta i.

manhattan razdalja – Pri podanih koordinatah mest je cena povezave od mesta i do mesta j pridobljena z enačbo $c_{j,i} = c_{i,j} = \sum_{t=1}^{N} |i_t - j_t|$ pri čemer je i_t t-ta koordinata mesta i.

Kakor pri različnih zvrsteh PTP-ja so najboljši algoritmi tisti, ki znajo izkoristijo lastnosti povezav v izbranem PTP-ju.

Naša naloga bo pregledovala algoritme za standarden PTP s simetričnimi cenami povezav.

3.4 Predstavitev rešitev problema trgovskega potnika

Pogledali bomo dva pogosto uporabljena načina za predstavitev rešitev PTP-ja.

Predpostavimo, da imamo standarden in simetričen PTP z N mesti. Prva predstavitev rešitev je z uporabo množice vseh povezav R v PTP-ju:

$$R = \{r_{1,1}, r_{1,2}, ..., r_{1,N}, r_{2,2}, r_{2,3}, ..., r_{N,N}\} \tag{3.1}$$
$$r_{i,j} \in [0,1] \mid \forall i,j$$

kjer $r_{i,j}$ predstavlja, do kolikšne mere je povezava od mesta i do mesta j vključena v rešitev.

Za izračun cene rešitev $C(R)$ uporabimo enačbo

$$C(R) = \sum_{j=1}^{N} \sum_{i=j}^{N} c_{i,j} r_{i,j}$$

Nekatera stanja te predstavitve rešitev niso Hamiltonov cikel in zato niso veljavne rešitve PTP-ja. Z njimi lahko pridobimo kakršen koli podgraf začetnega grafa $G = (V, E)$.

Kljub temu se ta predstavitev uporablja pri reševanju PTP-ja s pomočjo Linearnega programiranja[4]. Pri tem načinu reševanja PTP-ja postopoma

[4]angl. Linear Programming

dodajamo omejitve, dokler ne dobimo veljavne rešitve PTP-ja, ki je tudi optimalna.

Primer Linearnega Programiranja na PTP-ju:

$$\begin{aligned} &\text{minimiziraj} && \sum_{j=1}^{N}\sum_{i=j}^{N} c_{i,j}r_{i,j} \\ &\text{z omejitvami} && \sum_{j}^{N} r_{i,j} = 2, \quad i = 1, ..., N \\ & && r_{i,j} \in \{0, 1\}, \quad i, j = 1, ..., N \end{aligned}$$

Prva omejitev Linearnega programiranja zahteva, da je vsota vseh povezav v in izven katerega koli mesta enaka dva. Druga omejitev zahteva, da so vse povezave 0 ali 1 (kar pomeni, da problem rešujemo s Celoštevilskim Programiranjem[5]).

Naša naloga vsebuje izključno hevristične algoritme, pri katerih se ta predstavitev rešitev po navadi ne uporablja. Za pregled uporabe Linearnega Programiranja za reševanje PTP-ja priporočamo deli [7, 22].

Kakor pri prvem načinu predstavitve rešitev predpostavimo, da imamo standarden PTP z N mesti. Pri drugem načinu predstavitve rešitev uporabimo permutacijo R množice $P_N = \{1, 2, ..., N\}$:

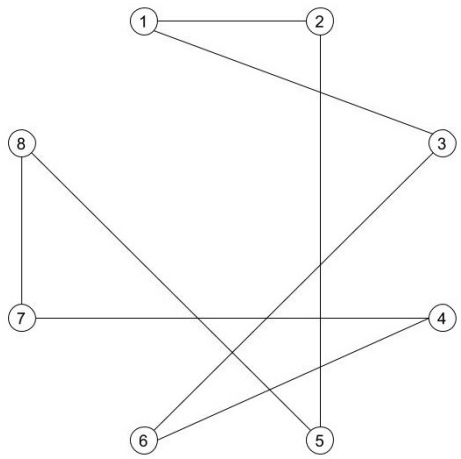
$$\begin{gathered} R = (r(1), r(2), ..., r(N)) \\ r(i) \in P_N \mid \forall i \end{gathered} \tag{3.2}$$

Za izračun cene rešitev $C(R)$ uporabimo enačbo

$$C(R) = \sum_{i=1}^{N-1} c_{r(i),r(i+1)} + c_{r(N),r(1)} \tag{3.3}$$

Ta predstavitev rešitev prikazuje vrstni red obiskanih mest, kjer sta prvo in zadnje mestu v R-ju povezani. Začetno ali prvo mesto je lahko katero koli mesto v R (po navadi je začetno ali prvo mesto $r(1)$). Primer $R = (6, 3, 1, 2, 5, 8, 7, 4)$ na namišljenem grafu je prikazan na sliki 3.1.

[5]angl. Integer Programming



Slika 3.1: Rešitev $(6, 3, 1, 2, 5, 8, 7, 4)$ prikazana grafično

Pri tem načinu predstavitve so vse rešitve veljavne, a niso enolične. Če celotno rešitev premaknemo na levo ali desno, dobimo enako rešitev, prav tako dobimo enako rešitev, če celotno rešitev zrcalimo.

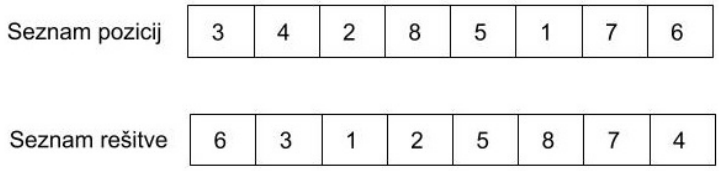
Ta predstavitev se najpogosteje uporablja v hevristikah, pri njihovi uporabi moramo paziti le na to, da R ostane permutacija množice P_N. Pri vseh hevristikah v tej diplomski nalogi se uporablja druga predstavitev rešitev.

3.4.1 Podatkovna struktura

Za izbrano predstavitev rešitev in podani hevristični algoritem je pomembna izbira primerne podatkovne strukture. Hevristični algoritmi s pomočjo operacij spreminjajo rešitve. Izbrati želimo podatkovno strukturo, ki v najmanj korakih izvede operacije, uporabljene v hevurističnem algoritmu.

V diplomski nalogi bomo za shranjevanje rešitev uporabili podatkovno strukturo dveh seznamov. Obstajajo tudi druge podatkovne strukture, ki jih ne bomo opisali, a se najdejo v delu [13].

Struktura dveh seznamov vsebuje dva seznama. Prvi seznam, seznam pozicij, nam pove, v kateri poziciji se nahaja določeno mesto. Prvi element



Slika 3.2: Primer podatkovne strukture dveh seznamov za rešitev $(6, 3, 1, 2, 5, 8, 7, 4)$

v seznamu pozicij vsebuje pozicijo prvega mesta v rešitvi, drugi element v seznamu pozicij vsebuje pozicijo drugega mesta v rešitvi itd. Drugi seznam, seznam rešitve, vsebuje samo rešitev. Primer je prikazan na sliki 3.2.

Nekaj primerov pogosto uporabljenih operacij nad rešitvami:

naslednji/prihodnji(i) – Pridobimo mesto, ki je naslednik/predhodnik mestu i. Zaradi seznama pozicij lahko v času reda $O(1)$ pridobimo pozicijo mesta i in nato s pomočjo seznama rešitve pridobimo iskano mesto v času reda $O(1)$.

vstavi na levo/desno(i, j) – Mesto i izvzamemo iz rešitve in ga vstavimo levo/desno od mesta j. Premakniti moramo mesta od i do j (po možnosti premik vključuje mesto j v primeru, ko mesto i vstavljamo levo od mesta j, a je mesto i desno od mesta j) in nato vstaviti mesto i. Največja časovna zahtevnost operacije je $O(N)$. Primer uporabe operacije je prikazan na sliki 3.3.

zrcali(i, j) – Zrcalimo vrstni red, vključno z mestom i in mestom j. Posodobiti moramo vsa mesta med podanima mestoma, vključno z njima. Največja časovna zahtevnost operacije je $O(N)$. Primer uporabe operacije je prikazan na sliki 3.3.

zrcali povezave med($(i, j), (k, l)$) – Enakovredno zrcali(j, k) ali zrcali(i, l), le da podamo povezave namesto mesta.

Slika 3.3: Prikaz operacij vstavi na desno$(3, 5)$ in zrcali$(1, 8)$ nad rešitvijo $(6, 3, 1, 2, 5, 8, 7, 4)$

Razlog za izbiro podatkovne strukture dveh seznamov je enostavnost implementacije in podpora potrebnih operacij v smiselnem času.

3.5 Splošno o hevrističnih metodah za PTP

Hevristike za PTP lahko razdelimo v tri skupine:

Konstrukcijska hevristika [6] – Sestavi rešitev.

Izboljševalna hevristika [7] – Izboljša že obstoječo rešitev.

Kombinirana hevristika [8] – Kombinacija zgoraj navedenih hevristik.

[6] angl. Construction heuristics
[7] angl. Improvement heuristics
[8] angl. Composite heuristics

Kombinirane hevristike so boljše kakor le konstrukcijske ali izboljševalne, saj omogočajo najboljšo možnost za pridobivanje dobrih rešitev, a po navadi trajajo dlje časa in so kompleksnejše. Primer konstrukcijske hevristike je požrešna metoda[9], kjer se dodaja najbližje, še ne obiskano mesto nazadnje dodanega mesta, vse dokler ne dodamo vseh mest. Primer izboljševalne hevristike je algoritem Lokalnega iskanja. Primer kombinirane hevristike pa je algoritem Lokalnega iskanja z začetno rešitvijo, pridobljeno s pomočjo požrešne metode.

Za lažje razumevanje hevrističnih algoritmov uporabljamo ideji izkoriščanja[10] ali intenzificiranja[11] in raziskovanja[12] ali diverzificiranja[13]. Namen izkoriščanja je približevanje lokalnemu optimumu rešitev. Namen raziskovanja je preiskovanje čim širšega območja v P_N. Pri vsakem algoritmu se lahko vprašamo, kateri del algoritma pripomore k izkoriščanju in kateri k raziskovanju.

V literaturi hevristik se najde pojem meta-hevristike[14], ki pa nima jasne definicije. Nekateri viri ga povezujejo z idejo, da se algoritem lahko izogne lokalnim optimumom, drugi pa kot meta-hevristični algoritem označijo vsak hevristični algoritem, ki je opisan na dovolj abstraktnem nivoju. Zaradi neenotne definicije tega pojma v diplomski nalogi ne bomo uporabljali.

3.6 Testi in testno okolje

Za testiranje naših algoritmov smo iz knjižice TSPLIB [24] izbrali naslednje primerke PTP-ja:

bayg29 – Primerek PTP-ja z 29 mesti. Cena povezav je podana eksplicitno za vsako povezavo. Najboljša rešitev je 1610.

[9] angl. Greedy method
[10] angl. Exploitation
[11] angl. Intensification
[12] angl. Exploration
[13] angl. Diversification
[14] qngl. Meta-Heuristics

brg180 – Primerek PTP-ja s 180 mesti. Cena povezav je podana eksplicitno za vsako povezavo. Najboljša rešitev je 1950.

fl417 – Primerek PTP-ja s 417 mesti. Cena povezav se izračuna z evklidsko razdaljo med podanimi dvodimenzionalnimi koordinatami mest. Najboljša rešitev je 11861.

si1032 – Primerek PTP-ja s 1032 mesti. Cena povezav je podana eksplicitno za vsako povezavo. Najboljša rešitev je 92650.

u1432 – Primerek PTP-ja s 1432 mesti. Cena povezav se izračuna z evklidsko razdaljo med podanimi dvodimenzionalnimi koordinatami mest. Najboljša rešitev je 152970.

rl1889 – Primerek PTP-ja s 1889 mesti. Cena povezav se izračuna z evklidsko razdaljo med podanimi dvodimenzionalnimi koordinatami mest. Najboljša rešitev je 316536.

Naši kriteriji pri izbiri primerov so bili sledeči:

- primeri morajo biti primerni za algoritme, ki rešujejo standarden PTP s simetričnimi povezavami;
- opazovati želimo delovanje algoritmov pri primerkih s postopoma več mesti;
- testiranje algoritmov na izbranih primerkih naj bo možno v praktično sprejemljivem času.

Testiranje je potekalo na računalniku s procesorjem Intel i7 s 3,7 GHz, 32 GB RAM-a in Java verzijo 12.0.2. Vsa koda, uporabljena v tej nalogi, se najde na githubu [25], rezultati algoritmov pa v Google preglednici [26].

Sami rezultati algoritmov bodo prikazani in opisani v poglavju posameznega algoritma in v desetem poglavju, kjer bomo rezultate algoritmov primerjali.

Poglavje 4

Lokalno iskanje

Ideja Lokalnega iskanja[1] je postopno približevanje k lokalnemu optimumu. Eden izmed načinov, kako to dosežemo, je ustvarjanje novih rešitev, podobnih trenutni rešitvi, in sprejemanje boljših rešitev, dokler ne dosežemo končnega pogoja.

Za implementacijo algoritma Lokalnega iskanja na PTP-ju potrebuje začetno rešitev R_z in operacijo, ki spremeni eno rešitev v drugo. Primera operacije sta zamenjava dveh mest v rešitvi ali premik izbranega mesta na konec rešitve.

Definirajmo funkcijo $SOSED(R) = \{R_1, R_2, R_3, ..., R_s\}$, kjer je $\{R_1, R_2, R_3, ...R_s\}$ množica vseh možnih rešitev, če uporabimo določeno operacijo nad rešitvijo R. Rešitve v množici $\{R_1, R_2, R_3, ..., R_s\}$ se imenujejo sosedi rešitve R.

Na začetku algoritem sprejeme začetno rešitev $R = R_z$. V naslednjem koraku pridobi soseda trenutne rešitve $R' = SOSED(R)$. Novo pridobljen sosed zamenja trenutno rešitev v primeru, ko je njegova cena nižja od cene trenutne rešitve $C(R') < C(R)$. Algoritem ponavlja korak, dokler ne doseže končnega pogoja, ki vrne njegovo trenutno rešitev.

Pseudo koda algoritma Lokalnega iskanja je prikazana v 1.

Algoritem Lokalnega iskanja ima dve pogosto uporabljeni strategiji za

[1] angl. Local Search

pregledovanje množice sosedov pri kakršni koli operaciji.

Prva strategija je naključno ustvarjanje sosedov trenutne rešitve, pri čemer sprejeme prvega soseda, ki ima nižjo ceno od cene trenutne rešitve. Imenujemo jo strategija prvega boljšega.

Druga strategija pregleda vse sosede trenutne rešitve, pri čemer si zapomni soseda z najnižjo ceno. Soseda z najnižjo ceno sprejeme v primeru, ko je njegova cena nižja od cene trenutne rešitve. Imenujemo jo strategija najboljšega. Ko s strategijo najboljšega soseda ne sprejmemo, se nahajamo v lokalnem optimumu.

Izbira operacije je najpomembnejši del algoritma. Predstavljamo dva indikatorja dobre operacije:

Prvi indikator je velikost množice možnih sosedov $|\{R_1, R_2, R_3, ..., R_s\}| = s$. Večja množica omogoča pregledovanje več sosedov, a za celoten pregled zahteva več časa.

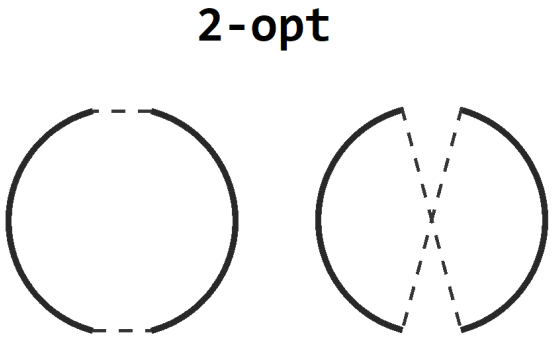
Drugi indikator je podobnost med sosedi in rešitvijo. Ne želimo, da so sosedi določene rešitve sestavljeni iz povezav, ki niso prisotne v podani rešitvi. Želimo, da so sosedi sestavljeni v večini iz povezav, prisotnih v podani rešitvi, saj želimo z manjšimi spremembami prikorakati do lokalnega optimuma. Ta indikator lahko ocenimo s pomočjo povprečnega števila povezav, ki so prisotne v podani rešitvi, a niso prisotne v sosedih. Ocena indikatorja nima optimalne vrednosti in je lahko zavajajoča. Lahko bi sklepali, da želimo imeti nižjo oceno indikatorja, saj želimo raziskati prostor blizu podane rešitve. A obstajajo operacije z višjo oceno indikatorja, ki vključujejo vse možne sosede iz operacije z nižjo oceno indikatorja.

4.1 Operacija k-opt

Operacija k-opt izbere k povezav v rešitvi, ki jih nato odstrani. Rešitev je tako razdeljena na k poti. Nato jih ponovno poveže tako, da je pridobljen sosed veljavna rešitev. Možni sosedi operacije 2-opt so prikazani na sliki 4.1, možni sosedi operacije 3-opt pa na sliki 4.2.

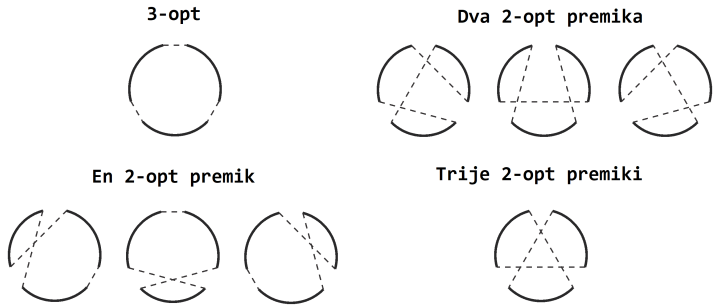
Algorithm 1 Lokalno iskanje

1: $R = R_z$
2: $R' = SOSED(R)$
3: **while** Končni pogoj **do**
4: **if** $C(R') < C(R)$ **then**
5: $R = R'$
6: $R' = SOSED(R)$
7: **return** R



Slika 4.1: Grafični prikaz vseh možnih sosedov operacije 2-opt

Operacijo k-opt implementiramo s pomočjo zrcaljenja določenega dela rešitve, saj če zamenjamo le pozicijo mest, ne pridobimo želenega soseda. Zakaj moramo zrcaliti določen del rešitve, bomo prikazali s pomočjo naslednjega primera. Vzemimo rešitev $(1, 2, 3, 4, 5, 6, 7, 8, 9)$. Z izbranima povezavama $(2, 3)$ in $(7, 8)$ želimo narediti operacijo 2-opt, s katero želimo pridobiti soseda, ki se razlikuje od podane rešitve. Če le zamenjamo mesti 3 in 7, pridobimo soseda $(1, 2, 7, 4, 5, 6, 3, 8, 9)$. Pridobljen sosed poleg želenih izgubljenih povezav $(2, 3)$ in $(7, 8)$ izgubi še povezavi $(3, 4)$ in $(6, 7)$, poleg želenih pridobljenih povezav $(2, 7)$ in $(3, 8)$ pridobi še povezavi $(7, 4)$ in $(6, 3)$. Pravilna operacija je zrcaljenje vseh mest med mestoma 3 in 7 ali zrcaljenje vseh mest



Slika 4.2: Grafični prikaz vseh možnih sosedov operacije 3-opt, razdeljenih po tem, koliko operacij 2-opt potrebujemo, da jih dosežemo

med mestoma 8 in 2, pri čemer pridobimo soseda $(1, 2, 7, 6, 5, 4, 3, 8, 9)$ ali $(9, 8, 3, 4, 5, 6, 7, 2, 1)$. Pridobljena soseda prikazujeta enako rešitev.

Če je podana rešitev k-opt optimalna, pomeni, da ne obstaja sosed z nižjo ceno v sosedju operacije k-opt podane rešitve. Za operacijo k-opt je značilno, da so vsi sosedi nižje stopnje operacije k-opt vključeni v sosedje višje stopnje operacije k-opt, $SOSED_{k_1\text{-opt}}(R) \subset SOSED_{k_2\text{-opt}}(R) \ : \ k_2 > k_1$. Če je rešitev k_2-opt optimalna, je tudi k_1-opt optimalna. $(N-1)$-opt optimalna rešitev je optimalna rešitev. S kombinacijo več operacij 2-opt lahko pridobimo priljubljenega soseda operacije k-opt. Če je število mest v primerku problema PTP N in uporabljamo operacijo k-opt, izbiramo med $\binom{N}{k}$ možnimi povezavami, ki jih odstranimo, kasneje pa jih lahko ponovno povežemo na $(k-1)!2^{k-1}$ načinov. Zaradi hitre rasti števila sosedov se večje stopnje operacije k-opt v praksi ne uporabljajo.

Algoritem Lokalnega iskanja smo implementirali z operacijo 2-opt in operacijo 3-opt z uporabo strategije prvega boljšega in strategije najboljšega. Na vseh primerkih smo ju testirali 30-krat. Algoritem Lokalnega iskanja z operacijo 3-opt in strategijo najboljšega smo 30-krat testirali le na primerkih bayg29, brg180, fl417 in si1032, saj bi za testiranje primerkov u1432 in rl1889 potrebovali preveč časa.

Kot začetno rešitev smo izbrali naključno ustvarjeno rešitev. Če bi upora-

bili rešitev, pridobljeno s pomočjo katerega koli konstrukcijskega algoritma, bi naleteli na problem pri ocenjevanju algoritma. Testirati želimo sposobnosti algoritma, pri uporabi konstrukcijskega algoritma pa bi morali ugotoviti, koliko je doprinesel do končnih rezultatov. Podanem problemu se lahko izognemo s pomočjo naključno ustvarjene začetne rešitve. Če bi uporabljali vnaprej ustvarjene naključne rešitve, ki bi jih ponovno uporabljali skozi vse algoritme v nalogi, bi v primerih, kjer določeni algoritmi kot vhod sprejmejo nobene ali več rešitev nastal problem.

Algoritem Lokalnega iskanja z operacijo 2-opt in 3-opt ter strategijo najboljšega se konča, kadar ne izboljša rešitve.

Algoritem Lokalnega iskanja z operacijo 2-opt in 3-opt ter strategijo prvega boljšega pregleda kN^2 možnih izbir povezav, kjer je N število mest primerka, k pa stopnja operacije k-opt, pri čemer preveri vse možne načine ponovnega povezovanja povezav, in se konča v primeru, ko ne izboljša rešitve.

Izbrali smo kN^2, saj smo želeli imeti enako enačbo za število možnih izbir povezav pri algoritem Lokalnega iskanja z operacijo 2-opt in operacijo 3-opt. Poskusili smo $\frac{\binom{N}{k}}{x}$, kjer je x nekakšno pozitivno število, a nismo našli x, ki bi omogočil dober pregled množice sosedov operacije 2-opt in omogočil smiselno časovno delovanje algoritma z operacijo 3-opt. Za algoritem Lokalnega iskanja z operacijo 2-opt velja, da bomo najverjetneje pregledali vse možne izbire povezav, saj $\lim_{N\to\infty} \frac{2N^2}{\binom{N}{2}} = 4$, za algoritem Lokalnega iskanja z operacijo 3-opt pa nimamo takšne garancije.

4.1.1 Rezultati

Rezultati so predstavljeni v tabelah 4.1 za Lokalno iskanje z operacijo 2-opt in strategijo prvega boljšega, 4.2 za operacijo 2-opt s strategijo najboljšega, 4.3 za operacijo 3-opt s strategijo prvega boljšega in 4.4 za operacijo 3-opt s strategijo najboljšega. V prvem stolpcu vseh tabel se nahajajo poimenovanja primerkov, v drugem stolpcu povprečna cena rešitev, v tretjem stolpcu oddaljenost povprečne cene rešitve od optimalne, izračunana z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu so zapisani

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1683	4,53	0>	0>
brg180	2403	23,23	10	8
fl417	12861	8,43	43	30
si1032	94126	1,59	449	345
u1432	176950	15,68	979	674
rl1889	370155	16,94	2694	2021

Tabela 4.1: Rezultati algoritma Lokalnega iskanja z operacijo 2-opt in strategijo prvega boljšega

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1654	2,73	1	1
brg180	2415	23,85	36	35
fl417	12614	6,35	538	537
si1032	94004	1,46	8747	8741
u1432	171562	12,15	26293	26279
rl1889	354047	11,85	86708	86676

Tabela 4.2: Rezultati algoritma Lokalnega iskanja z operacijo 2-opt in strategijo najboljšega

povprečni časi algoritma v milisekundah, v petem stolpcu pa povprečni časi v milisekundah, ki so bili potrebni, da je algoritem našel najboljšo rešitev od začetka algoritma.

Strategija najboljšega pridobi boljše rešitve kot strategija prvega boljšega, a za to porabi veliko več časa. To ni presenetljivo, saj strategija najboljšega pregleda vse sosede in se zmeraj ustavi v lokalnem optimumu, kar ji omogoči pridobivanje boljših rešitev. Strategija prvega boljšega pa ustavi pregled sosedov, ko najde boljšega soseda, kar ji omogoči hitrejše delovanje.

Opazna razlika med operacijo 2-opt in 3-opt ni presenetljiva. Rešitve

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1654	2,73	2	1
brg180	2019	3,54	32	23
fl417	12627	6,46	129	76
si1032	94041	1,5	1087	690
u1432	175113	14,48	6072	4717
rl1889	363757	14,92	16612	13502

Tabela 4.3: Rezultati algoritma Lokalnega iskanja z operacijo 3-opt in strategijo prvega boljšega

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1641	1,93	5	4
brg180	1964	0,72	2454	2432
fl417	12208	2,93	97687	97170
si1032	93080	0,46	3794128	3786070

Tabela 4.4: Rezultati algoritma Lokalnega iskanja z operacijo 3-opt in strategijo najboljšega

pri operaciji 3-opt so boljše od rešitev pri operaciji 2-opt, a je operacija 2-opt hitrejša od operacije 3-opt. Na primerku brg180 bomo izpostavili in komentirali razlike v rezultatih. Ta primerek dobro prikazuje idejo, da se lahko boljša rešitev skriva za kompleksnejšo operacijo, ki je enostavnejša operacija ne mora izvesti, saj bi morala sprejeti spremembo, ki za en ali več korakov poslabša trenutno rešitev. Za primerek brg180 lahko rečemo, da dodatna širina operacije 3-opt občutno pripomore k iskanju optimalne rešitve.

Zgoraj omenjeno idejo bomo imenovali skrite lastnosti primerka. Skrite lastnosti primerka so lastnosti primerka, ki jih je težko razbrati, a imajo vpliv na to, kako dobro določen algoritem ali operacija deluje nad primerkom. Skrite lastnosti lahko opazimo v rezultatih, kjer določen algoritem ali operacija najde boljšo rešitev, ki je drugi algoritem ali operacija ne more, ali pa določeno rešitev pridobi v krajšem času.

V tej nalogi bomo gledali tudi na kontrolni čas[2]. Njegov namen je predstaviti, kako dolgo je algoritem deloval ob koncu njegovega poteka, ne da bi izboljšal rešitve. Kontrolni čas pridobimo kot razliko med celotnim časom in časom, potrebnim za pridobitev najboljše rešitve, kakor je prikazano v enačbi: Celotni čas−Čas za najboljšo rešitev. Za naše algoritme Lokalnega iskanja kontrolni čas ni relevanten, saj se algoritmi končajo malo po najdbi najboljše rešitve.

4.1.2 Spremembe in izboljšave

Za algoritem Lokalnega iskanja bi lahko uporabili kakšno drugo operacijo, ki po možnosti najde boljšo rešitev v hitrejšem času. Lahko bi tudi povečali stopnjo operacije k-opt.

Rešitev, pridobljeno iz algoritma Lokalnega iskanja z določeno operacijo, lahko podamo kot začetno rešitev v algoritem Lokalnega iskanja z drugo operacijo.

[2]angl. Check-out time

Algoritem lahko izboljšamo z dobro začetno rešitvijo, pridobljeno iz konstrukcijskih algoritmov, kot so požrešna metoda, Christofidov algoritem ali kakšen drug konstrukcijski algoritem.

Operacijo k-opt lahko izboljšamo s prioritetno listo[3], ki namesto naključnega izbiranja povezav uporabi določeno logiko. Če je prioritetna lista dobro sestavljena, zmanjšamo čas delovanja algoritma, a ne izgubimo kakovosti rešitev. Pogosto uporabljena prioritetna lista je najbližji sosed[4], kjer se za vsako mesto sestavi seznam najbližjih mest. Prioritetno listo je mogoče sestaviti tudi s pomočjo drugih pripomočkov, npr. z minimalnim vpetim drevesom[5] PTP-ja.

Algoritem lahko izboljšamo z idejo ne poglej bit[6]. Izboljšava vsakemu mestu doda dodaten bit, ki nam pove, ali se je povezava blizu določenega mesta v preteklih korakih algoritma spremenila ali ne. Ob naslednjem koraku pregledamo le povezave, povezane z mesti, ki imajo dodaten bit nastavljen. S tem se poskusimo izogniti ponovnemu pregledu slabših izbir povezav.

4.1.3 Uporabljena literatura in dodatna literatura

Delo [28] vsebuje osnovni pregled različnih operacij in prioritetnih list. Natančnejši pregled in dodatno analizo prioritetnih list najdemo v delu [1], konstrukcijskih algoritmov v delu [2] in operacij v delu [3].

[3] angl. Candidate list
[4] angl. Nearest neighbors list
[5] angl. Minimal spanning tree
[6] angl. Don't look bit

Poglavje 5

Lin-Kernighanov algoritem

Slabosti algoritma Lokalnega iskanja z operacijo k-opt je število sosedov pri višjih stopnjah operacije k-opt. Lin-Kernighanov algoritem je algoritem, ki pridobi sosede višjih stopenj operacije k-opt s preiskovanjem le določenega dela vseh sosedov operacije. To doseže z večkratno uporabo operacije, podobne operaciji 2-opt. Algoritem je fleksibilen, kar mu omogoča končati njegov potek v katerem koli koraku, če oceni, da dodatne spremembe ne bodo izboljšale rešitve.

Definirajmo množico povezav $A = (A_1, A_2, ..., A_v)$. Vsak element v množici A_i je povezava med mestom a_{i_1} in mestom a_{i_2}, torej $A_i = (a_{i_1}, a_{i_2})$. Množica ob določeni točki vsebuje v elementov.

Lin-Kernighanov algoritem sestavlja dve množici povezav X in Y. Množica X vsebuje povezave, ki bodo odstranjene iz rešitve, medtem ko množica Y vsebuje povezave, ki bodo v rešitev dodane. V vsakem koraku algoritem odstrani povezavo tako, da jo doda v množico X, in doda povezavo tako, da jo doda v množico Y. Korak ponavlja, dokler ne doseže končnega pogoja.

Obstajajo štirje kriteriji, ki nam povedo, kako sestavljati množici X in Y in kaj je končni pogoj algoritma:

Kriterij disjunktivnosti množice X in Y [1] – Če smo iz rešitve odstranili povezavo, je ne smemo ponovno dodati. Če smo v rešitev dodali

[1] angl. The disjunctivity criterion

povezavo, je ne smemo ponovno odstraniti. To pomeni, da mora biti presek množice X in Y prazen $X \cap Y = \emptyset$.

Kriterij izvedljivosti [2] – Algoritem se lahko konča v katerem koli koraku. To doseže tako, da v množico X doda povezavo, ki bo spremenila rešitev v pot. Nato v množico Y doda povezavo, ki bo povezala konce poti.

Ta kriterij zahteva tudi, da algoritem za vsak X_i, kjer je $i > 1$, doda enako povezavo X_i, kakor povezavo X_i, ki bi jo dodal v primeru, če bi se zaključil. Če tega ne upoštevamo, lahko pride do situacije, kjer ta kriterij v naslednjem koraku algoritma ne bo držal.

Kriterij zaporedne izmenjave povezav v množicah X in Y [3] –

Ta kriterij zahteva, da za vsak par povezav X_i in Y_i velja, da je prvo mesto v Y_i povezavi enako drugemu mestu v X_i, torej $y_{i_1} = x_{i_2}$. To pomeni, da se vse dodane povezave začnejo z mestom, v katerem se je odstranjena povezava končala. Kriterij tudi zahteva, da je prvo mesto v vsakem X_i (razen za prvi element X_1) enako drugemu mestu v Y_{i-1}, torej $x_{i_1} = y_{(i-1)_2}$. To pomeni, da se povezava, ki bo v določenem koraku odstranjena, začne v mestu, v katerem se je končala dodana povezava v preteklem koraku algoritma.

Kriterij pridobička [4] – Poznamo dva pridobička:

Prvi pridobiček, pridobiček za i-ti korak algoritma, se izračuna po enačbi $G_i = \sum_{j=1}^{i} c_{x_{j_1},x_{j_2}} - \sum_{j=1}^{i} c_{y_{j_1},y_{j_2}}$. Predstavlja, koliko nižja je cena od začetne rešitve v koraku i. Ta pridobiček je le ocena cene rešitve ob trenutnem koraku. Če bi iz rešitve odstranili vse povezave v X množici in dodali vse povezave v Y množici, ne bi pridobili veljavne rešitve.

Drugi pridobiček, končni pridobiček za i-ti korak algoritma, se izračuna

[2] angl. The feasiblity criterion
[3] angl. The sequential exchange criterion
[4] angl. Gain criterion

po enačbi $G_i^* = \sum_{j=1}^{i} c_{x_{j_1},x_{j_2}} - \sum_{j=1}^{i} c_{y_{j_1},y_{j_2}} + c_{y_{i_2},x_{(i+1)_2}} - c_{x_{(i+1)_2},x_{1_1}}$. Odstranjeno povezavo $(y_{i_2}, x_{(i+1)_2})$ in dodano povezavo $(x_{(i+1)_2}, x_{1_1})$ pridobimo s pomočjo kriterija izvedljivosti. Če zaključimo algoritem v trenutnem koraku, nam ta pridobiček pove, koliko nižja bo cena pridobljene rešitve od začetne rešitve. Obstaja tudi najboljši najden končni pridobiček G^*.

Kriterij pridobička vsebuje tudi končni pogoj. Ta zahteva, da se naj algoritem zaključi v primeru, kadar so spremembe v trenutnem koraku slabše od začetne rešitve, torej $G_i < 0$, ali kadar so trenutne spremembe v trenutnem koraku slabše od najboljšega končnega pridobička, torej $G_i < G^*$.

Ob končnih pogojih, navedenih v kriteriju pridobička, se algoritem konča, kadar ne more dodati nove povezave v množico X ali Y.

Pseudo koda Lin-Kernighan algoritma je prikazana v 2.

Za lažje razumevanje Lin-Kernighanovega algoritma bomo njegov potek prikazali s pomočjo primera na sliki 5.1.

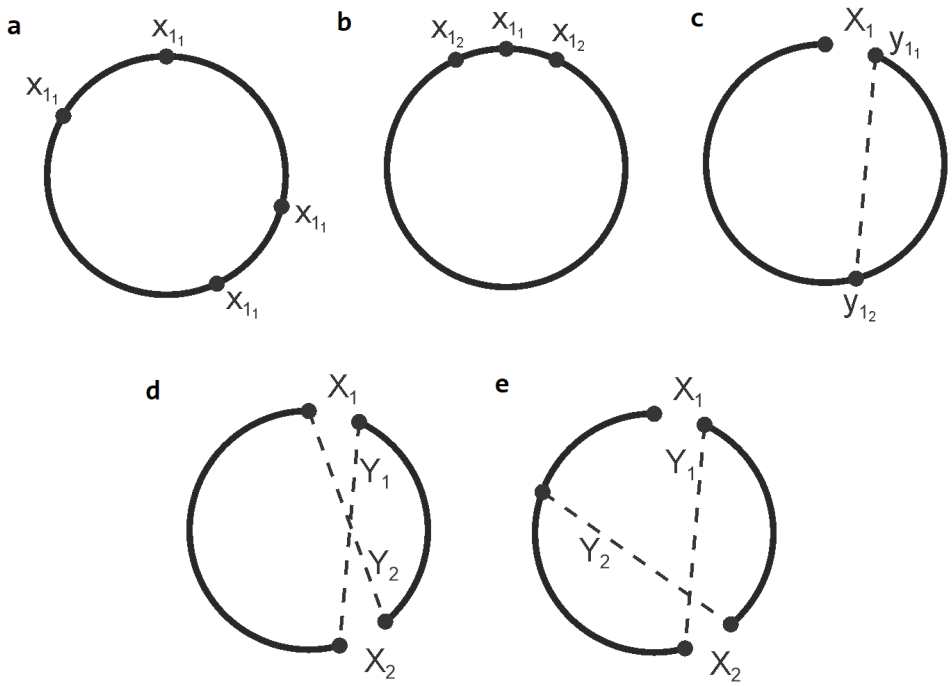
Začnemo v delu a), kjer iščemo prvo mesto v povezavi X_1, x_{1_1}. Nato končamo sestavljanje povezave X_1 tako, da izberemo enega izmed sosednjih mest mesta x_{1_1}, označenega z x_{1_2}, kakor je prikazano v delu b). Mesto x_{1_2} mora biti sosedno mestu x_{1_1}, sicer povezava (x_{1_1}, x_{1_2}) ni obstoječa povezava v rešitvi.

Po izbiri X_1 vemo, da bo prvo mesto v dodani povezavi Y_1 enako drugemu mestu v odstranjeni povezavi X_1 $y_{1_1} = x_{1_2}$ zaradi kriterija zaporedne izmenjave povezav. Izbira drugega mesta v dodani povezavi y_{1_2} je odvisna od implementacije algoritma, a mora ohranjati kriterij disjunktivnosti množic in omogočati odstranjevanje povezave X_2 v naslednjem koraku algoritma. Povezava, ki jo bomo odstranili v naslednjem koraku, je po izbiri y_{1_2} enolično določena zaradi kriterija izvedljivosti. V delu c) prikazujemo, kako je videti rešitev po izbiri y_{1_2}.

Izračunamo pridobiček $G_1 = c(x_{1_1}, x_{1_2}) - c(y_{1_1}, y_{1_2})$ in končni pridobiček $G_1^* = c(X_1) - c(Y_1) + c(X_2) - c(Y_2)$, kjer po kriteriju zaporedne izmenjave

Algorithm 2 Lin-Kernighan

1: $G^* = 0$
2: Inicializacija množice X in Y
3: Izberi povezavi X_1 ter Y_1 tako, da zadržujejo kriterije algoritma in omogočajo odstranjevanje X_2 (ki je enolično določen po izbiri Y_1 zaradi kriterija izvedljivosti in kriterija zaporedne izmenjave povezav)
4: Dodaj X_1 v množico X ter Y_1 v množico Y
5: Izračunaj G_1 ter G_1^*
6: **if** $G^* < G_1^*$ **then**
7: $G^* = G_1^*$ in shrani rešitev
8: $i = 1$
9: **while** $0 < G_i$ in $G_i^* < G_i$ **do**
10: $i = i + 1$
11: **if** Ali obstajata povezavi X_i ter Y_i, ki ohranjata kriterije in omogočata odstranjevanje povezave X_{i+1} **then**
12: Dodaj povezavo X_i v množico X in povezavo Y_i v množico Y
13: **else**
14: **return** Najboljšo rešitev
15: Izračunaj G_i ter G_i^*
16: **if** $G^* < G_i^*$ **then**
17: $G^* = G_i^*$ in shrani rešitev
18: **return** Najboljšo rešitev



Slika 5.1: Primer Lin-Kernighan algoritma

in kriteriju izvedljivosti pridobimo povezavi X_2 in Y_2. Kakšna je rešitev v primeru, če se algoritem zaključi v tem koraku, je prikazano v delu d).

Če se v tem koraku algoritem ne zaključi, povečamo trenutni korak i in iščemo nov Y_2. Povezava X_2 ostane enaka kot v primeru, če se bi algoritem zaključil zaradi kriterija izvedljivosti. Povezava Y_2 se začne v drugem mestu v povezavi X_2 zaradi kriterija zaporedne izmenjave povezav $y_{2_1} = x_{2_2}$. Izbira mesta y_{2_2} je podobna izbiri mesta y_{1_2} v smislu, da je odvisna od implementacije algoritma. Paziti moramo, da omogočimo odstranjevanje naslednje povezave (v tem primeru X_3) in ohranjamo kriterije. Videz trenutne rešitve je prikazan v delu e).

Ponovno izračunamo pridobička, preverimo, ali smo našli boljšo najboljšo rešitev in preverimo končne pogoje, povezane s pridobičkom.

Algoritem nadaljuje njegovo delovanje, dokler ne doseže enega izmed

končnih pogojev in nam nato vrne najboljšo rešitev.

Avtorja algoritma predlagata naslednje izboljšave:

Vračanje [5] – Štirje nivoji vračanja, na katere se algoritem vrne v primeru končnega pogoja.

Prvi nivo je ponovna izbira x_{1_1}. Drugi nivo je izbira drugega x_{1_2}. Tretji nivo je ponovna izbira y_{2_2}. Četrti nivo je izbira drugega x_{2_2}, pri čemer zanemarimo kriterij izvedljivosti za trenuten korak in po možnosti še za naslednji korak. Ta izboljšava nam omogoča pregled več sprememb rešitve, preden se algoritem konča.

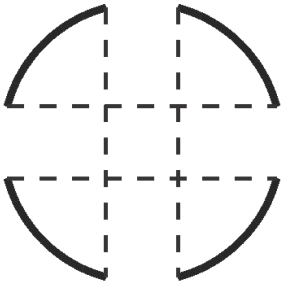
Prioritetna lista [6] – Uporaba prioritetne liste za izbiro mest y_{i_2}. Skrajša čas delovanja algoritma in nam poda logiko za izbiro y_{i_2}.

Naprej pogled [7] – Zaradi kriterija izvedljivosti in kriterija zaporedne izmenjave poznamo X_{i+1} ob izbiri y_{i_2}. To nam omogoča izboljšati izbiro mesta y_{i_2} z vključitvijo X_{i+1}. Namesto enačbe $min(c_{y_{i_1},y_{i_2}})$ bi za izbiro najboljšega mesta y_{i_2} uporabili enačbo $min(c_{y_{i_1},y_{i_2}} - c_{y_{i_2},x_{(i+1)_2}})$.

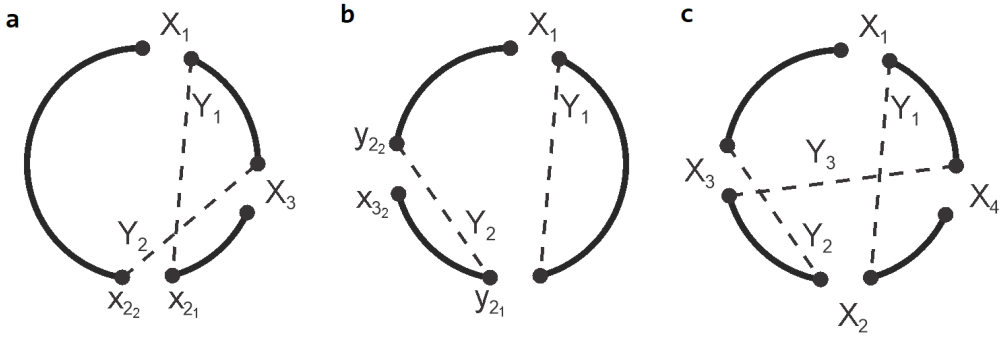
Redukcija [8] – Prepoznamo skupne povezave, vključene v dobre rešitve. Ob naslednjih pogonih algoritma izbrane povezave ne odstranjujemo. Ta izboljšava nam skrajša čas preiskovanja, a omeji pregled več rešitev.

Nezaporedne spremembe [9] – Določene spremembe rešitve ne moremo doseči zaradi kriterija izvedljivosti in kriterija zaporedne izmenjave, npr. premika dvojnega mostu[10], prikazanega na sliki 5.2. Zaradi tega lahko ob koncu algoritma poskusimo nekaj takšnih sprememb, saj nam lahko prinesejo optimalno rešitev.

[5] angl. Backtracking
[6] angl. Priority list
[7] angl. Lookahead
[8] angl. Reduction
[9] angl. Nonsequential exchanges
[10] angl. Double bridge



Slika 5.2: Premik dvojnega mostu



Slika 5.3: Primer četrtega nivoja vračanja

Četrti nivo vračanja, alternativno izbiro x_{2_2}, pri katerem zanemarimo kriterij izvedljivosti za trenuten in po možnosti za naslednji korak, bomo prikazali s pomočjo primera na sliki 5.3, saj je to edini nivo vračanja z netrivialnim potekom.

Četrti nivo začnemo tako, da izberemo x_{2_2}, ki razdeli rešitev na en cikel in eno pot. Če bi v tem koraku zaključili algoritem s trenutno izbranim X_2, bi pridobili rešitev z dvema cikloma, kar pa ni veljavna rešitev. Tako izbira alternativnega x_{2_2} zanemari kriterij izvedljivosti za en korak. Dobimo dva različna izida, ki sta odvisna od tega, kje se nahaja izbrani y_{2_2}.

Kadar se y_{2_2} nahaja v ciklu, kot je prikazano v delu a), lahko izberemo kateri koli x_{3_2} in vzpostavimo kriterij izvedljivosti. V naši implementaciji bomo zmeraj izbrali x_{3_2}, ki se nahaja na bližnji poti med mestoma x_{3_1} in x_{2_1}. Po izbiri x_{3_2} algoritem normalno nadaljuje.

Kadar se y_{2_2} nahaja na poti, kot je prikazano v delu b), izberemo x_{3_2}, ki se nahaja na bližnji poti med mestoma x_{3_1} in y_{2_1}. Na izbiro x_{3_2} lahko gledamo tudi kot na izbiranje povezave, ki ohrani sestavo rešitve iz enega cikla in ene poti. Če izberemo drug x_{3_2}, pridobimo dva cikla in eno pot. Ta izid zanemari kriterij izvedljivosti za dodaten korak.

Sedaj moramo ponovno vzpostaviti kriterij izvedljivosti. To naredimo z izbiro povezave Y_3, ki poveže pot in cikel rešitve. Rezultat je prikazan v delu c). Za X_4 lahko izberemo katerega koli soseda, a v naši implementaciji zmeraj izberemo x_{4_2}, ki se nahaja na bližnji poti med mestoma x_{4_1} in x_{2_1}. Po izbiri x_{4_2} algoritem normalno nadaljuje.

V opisu algoritma nismo navedli, kako v našo podatkovno strukturo vpeljemo odstranjevanje povezav v množici X in dodajanje povezav v množici Y. To smo implementirali tako, da smo po izbiri X_1 rešitev spremenili tako, da je x_{1_1} prvo mesto in x_{1_2} zadnje mesto v rešitvi. Nato si predstavljamo, da povezava med prvim in zadnjim mestom v rešitvi ne obstaja. Za vsak korak po prvem zrcalimo vsa mesta od $x_{(i-1)_2}$ do x_{i_2}. Ko dosežemo konec algoritma, že imamo želeno rešitev.

Opis sprememb nad podatkovno strukturo v primeru vračanja četrtega nivoja bomo zaradi kompleksnosti izpustili, a je podoben zgoraj omenjenemu postopku.

Implementirali smo Lin-Kernighanov algoritem in ga na vsakem primerku testirali 30-krat. Algoritem vsebuje vse nivoje vračanja in uporablja prioritetno listo 10 najbližjih sosedov. Izboljšave najprej pogleda, redukcije in nezaporednih sprememb nismo implementirali.

Na prvi nivo vračanja se vrnemo 10-krat, pri čemer zmeraj izberemo naključno mesto. Drugi nivo vračanja preveri alternativno izbiro x_{1_2}. Na tretji nivo vračanja se vrnemo, kadar še nismo pregledali vseh mest v prioritetni

Ime primerka	Rešitev	Oddaljenost (%)	Čas(ms)	Najboljša(ms)
bayg29	1620	0,62	2	1
brg180	2002	2,67	4	3
fl417	16487	39	33	27
si1032	94559	2,06	74	61
u1432	159661	4,37	217	193
rl1889	398767	25,98	485	438

Tabela 5.1: Rezultati Lin-Kernighan algoritma

listi. Za četrti nivo preverimo alternativno izbiro x_{2_2} in le enkrat izberemo najboljšo povezavo Y_2 in po možnosti povezavo Y_3 po prioritetni listi. Za preostali del algoritma izberemo najboljši Y_i po prioritetni listi, a se ne vrnemo, tudi če nismo pregledali celotne prioritetne liste.

Algoritem se konča v primeru, ko $\frac{N}{10} + 15$-krat zaporedoma ne izboljša rešitve.

Algoritem deluje deterministično po izbiri mesta x_{1_1}. Izbrali smo $\frac{N}{10} + 15$, saj vemo, da smo v primeru, ko bi se algoritem končal, pregledali $10 * (\frac{N}{10} + 15)$ izbir x_{1_1} zaradi vračanja na prvi nivo. Algoritem lahko ponovno izbere enako mesto x_{1_1}, a kljub temu bo pregledal vsaj večino, če že ne vseh možnih izbir mesta x_{1_1}, ki ga bo ponovno izbral skupno $10 * \frac{N}{10} + 15 = N + 150$-krat.

5.1 Rezultati

Rezultati so predstavljeni v tabeli 5.1. V prvem stolpcu tabele se nahajajo poimenovanja primerkov, v drugem stolpcu povprečna cena rešitev, v tretjem stolpcu oddaljenosti povprečne cene rešitev od optimalne, izračunane z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu povprečne čase algoritma v milisekundah, v petem stolpcu pa imamo povprečne čase, potrebne, da je algoritem od začetka našel najboljšo rešitev (v milisekundah).

Rešitve so dobre, razen za primerka brg180 in fl417. Algoritem je tudi

zelo hiter, saj potrebuje manj kot sekundo za največji primerek - rl1889. Sklepamo, da algoritem zaradi skritih lastnosti primerkov, ki otežijo pridobivanje boljših rešitev, z operacijami, uporabljenimi v algoritmu, slabše deluje na primerkih brg180 in fl417.

Kontrolni čas nam ne pove veliko, saj se, kakor algoritem Lokalnega iskanja, Lin-Kernighanov algoritem konča malenkost pred najdbo njegove najboljše rešitve.

5.2 Spremembe in izboljšave

Lin-Kernighanov algoritem lahko izboljšamo z boljšo začetno rešitvijo, boljšo prioritetno listo in ne poglej bitom.

Dobra sprememba Lin-Kernighanovega algoritma je sprememba kriterija izvedljivosti. Trenutno kriterij zahteva, da moramo algoritmu po vsakem dodanem paru X in Y (razen v primeru vračanja) omogočiti pridobivanje rešitev z odstranitvijo primerne povezave in povezavo koncev poti. Pravilo spremenimo tako, da namesto po vsakem paru to zahtevamo le po vsakih j parih. Posledica tega je, da ne moremo izračunati končnega pridobička G^* za vse pare X in Y. Z vidika raziskovanja je to dobro, saj končni pridobiček predstavlja omejitev pregleda slabših sprememb na krajši rok. Te spremembe nam lahko omogočijo pridobitev boljših rešitev na daljši rok.

5.3 Uporabljena literatura in dodatna literatura

Izvorno delo Lin-Kernighanovega algoritma je [19].

Dobra izboljšava Lin-Kernighanovega algoritma je predstavljena v delu [16]. Delo predlaga izboljšano prioritetno listo na podlagi α-bližine[11] mest, pridobljene iz L-dreves, spremembo določenih kriterijev algoritma, začetek z boljšo

[11] angl. α-nearness

začetno rešitvijo in odstranitev vračanja, ki postane z vsemi izboljšavami nepotrebno.

Poglavje 6

Simulirano ohlajanje

Simulirano ohlajanje[1] izhaja iz metalurgije[2]. Žarjenje[3] je proces segrevanja materiala do določene temperature, pri kateri se atomi prosto premikajo, in počasnega ohlajanja, da pridobimo stanje s čim manj energije. Algoritem Simuliranega ohlajanja simulira proces žarjenja.

Za implementacijo ideje Simuliranega ohlajanja v algoritem za reševanje PTP-ja potrebujemo začetno rešitev R_z, operacijo, ki spremeni eno rešitev v drugo, verjetnostno funkcijo za sprejem pridobljene rešitve, začetno temperaturo, način nižanja temperature in končni pogoj.

Ko izberemo vse potrebne dele algoritma, jih sestavimo po sledečem postopku: Začnemo z začetno temperaturo in začetno rešitvijo. V vsaki iteraciji algoritma dobimo novo rešitev s pomočjo operacije nad trenutno rešitvijo. Novo rešitev sprejmemo odvisno od verjetnostne funkcije. Ob koncu iteracije znižamo temperaturo po načinu nižanja temperature. Iteracije ponavljamo, dokler ne dosežemo končnega pogoja.

Algoritem Simuliranega ohlajanja je podoben algoritmu Lokalnega iskanja s strategijo prvi boljši, a ima možnost, da sprejme slabšo rešitev - odvisno od verjetnostne funkcije. Ta možnost omogoči algoritmu izhod iz lokalnih optimumov.

[1] angl. Simulated Annealing

[2] znanstvena veda o pridobivanju kovin iz rude

[3] angl. Annealing

Za izbiro operacije je podobnost še zmeraj pomembna kot pri algoritmu Lokalnega iskanja, a njena velikost ne prinaša dodatnega časa, kajti v vsaki iteraciji algoritma pridobimo le enega naključnega soseda trenutne rešitve. Operacija 2-opt ali po možnosti tudi operacija 3-opt je dobra izbira za operacijo, pri čemer nobena nima očitne prednosti pred drugo.

Verjetnostna funkcija je podobna Metropolisovemu kriteriju[4]. Izračunamo jo po enačbi:

$$P(\text{Sprejemi } R') = \begin{cases} exp(\frac{C(R)-C(R')}{T}) & C(R) < C(R') \\ 1 & C(R) \geq C(R') \end{cases} \tag{6.1}$$

Pri tem je R trenutna rešitev, R' nova rešitev, T pa trenutna temperatura. Verjetnostna funkcija deluje po naslednjem principu: če je nova rešitev boljša od trenutne, jo sprejmemo. Če je slabša, jo hitreje sprejmemo ob visoki temperaturi in ob manjši razliki med cenama nove in trenutne rešitve.

Začetna temperatura in način nižanja temperature sta pomembna dela algoritma. Potrebujemo dovolj visoko začetno temperaturo, saj je v nasprotnem primeru delovanje algoritma podobno delovanju algoritma Lokalnega iskanja s strategijo prvega boljšega. Pazljivi moramo biti pri načinu nižanja temperature. Algoritem lahko razdelimo v dve fazi - odvisno od trenutne temperature. Prva faza, je faza visoke temperature. Takrat algoritem pogosto sprejme slabšo rešitev. Druga faza, je faza nizke temperature. V tej fazi algoritem deluje podobno kot algoritem Lokalnega iskanja s strategijo prvega boljšega. Naš cilj sta takšna začetna temperatura in takšen način nižanja temperature, da bosta obe fazi algoritma primerno obdelani. Začetno temperaturo in način nižanja temperature imenujemo temperaturni urnik[5]. Dobro izbiro temperaturnega urnika predstavlja kakršna koli dovolj visoka začetna temperatura T_0 z geometrijskim nižanjem. To je predstavljeno v enačbi

$$T_{i+1} = T_i * f \tag{6.2}$$

[4]angl. Metropolis criterion
[5]angl. Temperature schedule

T_i predstavlja temperaturo ob i-ti iteraciji algoritma, f pa faktor nižanja. Geometrijsko nižanje temperature izniči previsoke začetne temperature T_0 v smiselnem številu iteracij algoritma. Tako algoritmu omogočimo primerno obdelavo obeh faz temperature. Faktor nižanja je po navadi $f \in [0,8, 0,999]$.

Najpogostejši končni pogoj algoritma je spodnja meja temperature ali vnaprej določeno število iteracij.

Namesto nižanja temperature ob vsaki iteraciji algoritma lahko temperaturo, preden jo znižamo, zadržimo za nekaj iteracij. S tem algoritmu dovolimo stabiliziranje rešitve ob določeni temperaturi. Alternativno, če imamo geometrijsko nižanje temperature in končno temperaturo za končni pogoj, lahko le primerno povečamo faktor nižanja. Tako ohranimo enako število iteracij do zaključka algoritma, kakor bi jih imeli z dodatnimi iteracijami ob zadržani temperaturi. Oba načina imata podoben, a ne isti, potek temperature skozi potek algoritma.

Pseudo koda algoritma Simuliranega ohlajanja je prikazana v 3.

Algorithm 3 Simulirano ohlajanje

```
1: Pridobi začetno temperaturo T_0 in začetno rešitev R = R_z
2: while Končni pogoj do
3:     for Št. iteracij ob temperaturi do
4:         Dobi novo rešitev R' = SOSED(R)
5:         if C(R) > C(R') then
6:             R = R'
7:         else
8:             Sprejmi R' po enačbi (6.1)
9:     Zmanjšaj temperaturo po enačbi (6.2)
10: return Najboljšo rešitev
```

V našem algoritmu Simuliranega ohlajanja bomo začetno temperaturo izračunali s pomočjo algoritma, predstavljenega v delu [8]. Ta algoritem poskuša ugotoviti, kakšna mora biti temperatura, da dosežemo naprej podano verjetnost sprejema nove rešitve. To doseže s pomočjo novih rešitev, pri-

dobljenih iz uporabe operacije nad začetnimi rešitvami in samimi začetnimi rešitvami. Kot smo omenili v poglavju Lokalnega iskanja, so začetne rešitve naključno ustvarjene rešitve.

Algoritem Simuliranega ohlajanja smo implementirali z operacijama 2-opt in 3-opt ter ga na vsakem primerku 30-krat testirali. Pri operaciji 2-opt izberemo dve naključni povezavi in zmeraj vrnemo rešitev, ki je drugačna od podane rešitve. Pri operaciji 3-opt izberemo tri naključne povezave in vrnemo najboljšo zamenjavo, ki ni enaka podani rešitvi. Začetna temperatura je izračunana po omenjenem algoritmu, ki bo poskusil najti temperaturo z verjetnostjo $0,95$ s pomočjo 2000 začetnih rešitev. Nižanje temperature je geometrijsko, pri čemer je faktor nižanja $0,99$. Končni pogoj algoritma je število iteracij. Temperaturo znižamo 1400-krat, pri čemer vsako temperaturo zadržimo za 10 000 iteracij.

Parametri so bili pridobljeni na osnovi predlogov iz literature in testiranja algoritma na primerkih z manj mesti.

6.1 Rezultati

Rezultati za Simulirano ohlajanje z operacijo 2-opt so predstavljeni v tabeli 6.1, za Simulirano ohlajanje z operacijo 3-opt pa v tabeli 6.2. V prvem stolpcu tabel so zapisana imena primerkov, v drugem stolpcu povprečna cena rešitev, v tretjem oddaljenosti povprečne cene rešitev od optimalne, izračunane z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu imamo povprečne čase algoritma v milisekundah, v petem pa povprečne čase, potrebne za to, da je algoritem našel najboljšo rešitev od začetka algoritma (v milisekundah).

Simulirano ohlajanje z operacijo 2-opt ima boljše ali enake rešitve kakor Simulirano ohlajanje z operacijo 3-opt za vse primerke, razen za brg180. Sklepamo, da ima operacija 3-opt slabše rešitve zato, ker zmeraj vrne najboljšo izmed možnih izmenjav in zaradi tega težje skoči izven lokalnih optimumov. Časovno sta si algoritma podobna, a je operacija 3-opt malenkost daljša, saj

Ime primerka	Rešitev	Oddaljenost (%)	Čas(ms)	Najboljša(ms)
bayg29	1610	0	7803	2937
brg180	1957	0,36	38451	31510
fl417	12085	1,89	87571	66263
si1032	93352	0,76	293909	284142
u1432	168561	10,19	658646	653550
rl1889	373362	17,95	865289	862286

Tabela 6.1: Rezultati algoritma Simuliranega ohlajanja z operacijo 2-opt

Ime primerka	Rešitev	Oddaljenost (%)	Čas(ms)	Najboljša(ms)
bayg29	1610	0	9309	3812
brg180	1952	0,1	41039	36342
fl417	12114	2,13	91029	81394
si1032	94143	1,61	296211	294823
u1432	177440	16	641647	640371
rl1889	419747	32,61	897327	896609

Tabela 6.2: Rezultati algoritma Simuliranega ohlajanja z operacijo 3-opt

mora ob vsaki iteraciji izbrati tri povezave in pregledati 7 sosedov operacije 3-opt in nato vrniti najboljšega. Operacija 2-opt pa le naključno izbere dve povezavi in vrne novo rešitev.

Kontrolna časa za operaciji 2-opt in 3-opt sta podobna. V primerkih z manj mesti se najboljša rešitev najde, še preden se algoritem konča. Pri bayg29 se najde najboljša rešitev približno v $\frac{1}{3}$ časa delovanja algoritma. V primerkih z več mesti pa se pozno v poteku algoritma najdejo boljše rešitve. To bi lahko pomenilo, da bi v primeru, če bi algoritmu dali več časa (v našem primeru več iteracij), ta lahko našel boljšo rešitev.

6.2 Spremembe in izboljšave

Algoritem Simuliranega ohlajanja lahko izboljšamo z boljšo začetno rešitvijo ali kakšno drugo operacijo.

Temperaturne urnike lahko razdelimo v dve skupini. Prva skupina so statični urniki, kjer naprej podamo temperaturni urnik, ki se skozi potek algoritma ne spremeni. Druga skupina so dinamični urniki, kjer se lahko način nižanja temperature ob poteku algoritma spremeni. Obstaja tudi več načinov nižanja temperature, kot je npr. linearno nižanje temperature, kjer pri vsakem koraku od trenutne temperature odštejemo določeno število, a se izkaže, da je lažje uporabljati geometrijsko nižanje temperature.

6.3 Uporabljena literature in dodatna literatura

Dober pregled algoritma Simuliranega ohlajanja se najde v delih [23, 5].

V delu [14] se predlaga pridobivanje več novih rešitev in izbira najboljše namesto le ene.

Poglavje 7

Kolonija mravelj

Algoritem Kolonije mravelj izhaja iz obnašanja mravelj pri iskanju najkrajše poti do hrane.

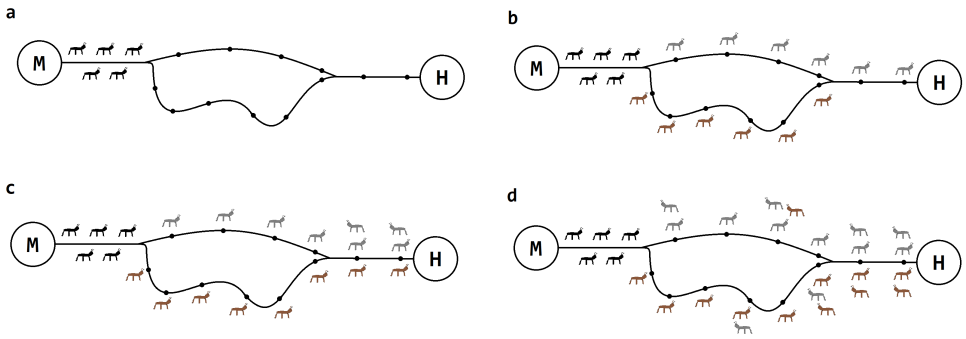
Kadar koli kolonija mravelj najde hrano, je naloga mravelj, da jo čim prej prinesejo nazaj v mravljišče. Iskanje najkrajše poti od mravljišča do hrane je pomemben del pospešitve tega procesa. Zato imajo mravlje določen sistem, ki jim omogoči najti najkrajšo pot.

Mravlje na svoji poti puščajo feromon, ki čez čas izhlapi. Feromon pomaga drugim mravljam pri določanju njihovih poti – več feromona kakor ima pot, večja je verjetnost, da bo jo mravlja izbrala. Vendar lahko mravlja izbere pot z manj ali čisto brez feromona.

Zakaj in kako ta sistem omogoči mravljam najti najkrajšo pot od mravljišča do hrane, bomo prikazali s pomočjo primera na sliki 7.1.

Slika je sestavljena iz štirih delov. V vsakem delu M predstavlja mravljišče, H pa hrano. Vsak del sestavljata enaki zgornja in spodnja pot. Na določenih delih poti se nahajajo točke, ki bodo delovale kot postajališče za mravlje. Razdalja med vsemi sosednimi točkami je enaka. Zgornja pot ima štiri točke, spodnja pa šest, torej je zgornja pot krajša kot spodnja pot. V našem primeru bodo iz mravljišča nenehno prihajale nove mravlje.

V delu a) mravlje pridejo iz mravljišča in se odločajo, ali naj izberejo zgornjo ali spodnjo pot. Trenutno sta obe poti brez feromona, saj še nista



Slika 7.1: Grafično prikazan potek sistema mravelj pri iskanju najkrajše poti od mravljišča do hrane

bili prehojeni, zato se bo polovica mravelj odločila za zgornjo pot, druga polovica pa za spodnjo.

Del b) prikazuje stanje, kjer je 12 mravelj že izbralo svojo pot do hrane. Šest mravelj je na zgornji poti, šest na spodnji. Mravlje, ki so izbrale zgornjo pot, smo označili s sivo barvo, mravlje, ki so izbrale spodnjo pot, smo označili z rjavo barvo. Sive mravlje so že pri hrani, medtem ko so rjave mravlje šele na koncu spodnje poti. Feromon, ki ga je na prvi točki zgornje poti pustila prva siva mravlja, je po večini izhlapel, feromon, ki ga je na prvi točki pustila druga siva mravlja, je malo manj izhlapel... Podobno velja za preostale štiri sive mravlje in za rjave mravlje pri spodnji poti. Zaradi tega se za zgornjo in spodnjo pot še zmeraj odloča enako število mravelj.

Del c) prikazuje stanje po tem, ko so štiri dodatne mravlje izbrale svojo pot do hrane. Sive mravlje se morajo ponovno vprašati, za katero pot naj se odločijo na poti nazaj do mravljišča. Zadnjo točko zgornje poti je prekorakalo več mravelj kot zadnjo točko spodnje poti, zato je na njej več feromona, tudi če skozi čas izhlapeva. Ta razlika je pomembna, saj bi od tega trenutka več mravelj začelo izbirati zgornjo pot. V našem primeru bomo to prikazali tako, da bo prva mravlja na poti nazaj do mravljišča izbrala zgornjo pot, druga

spodnjo, tretja zgornjo, četrta spodnjo itd.

V delu d) je prva mravlja, ki je izbrala zgornjo pot, na poti nazaj do mravljišča končala zgornjo pot in se nahaja na prvi točki zgornje poti. V tem trenutku sta na prvi točki zgornje poti feromon pustili dve mravlji, obenem pa je na prvi točki spodnje poti svoj feromon pustila le ena mravlja. To pomeni, da bo večina mravelj, ki si šele izbirajo svojo pot do hrane, izbrala zgornjo pot. Takšno stanje bo ohranjeno, dokler se prva mravlja, ki je izbrala spodnjo pot, do mravljišča ne vrne po spodnji poti. To bo trajalo dovolj dolgo, da se bo zgornja pot uveljavila kot boljša pot z več feromona.

Nadaljevanje sistema je trivialno. Zgornja pot bo intenzivneje potrjena kot boljša, saj je krajša. Na njej se bo dogajalo isto kot v prikazanem primeru, ko se je zgornja pot uveljavila kot boljša. Spodnjo pot bo še zmeraj izbralo nekaj mravelj, ki želijo najti novo boljšo pot, a je ta daljša, zato se to ne bo zgodilo.

Prikazali smo, kako in zakaj sistem deluje v primeru, ko mravlje iščejo najkrajšo pot do hrane. Ta sistem deluje tudi v primeru, ko določena pot izgine ali pa nastane nova, krajša pot.

Sprememba sistema mravelj v algoritem za reševanje PTP-ja je opisana v nadaljevanju.

Definirati moramo, kaj je mravlja in kaj je njena pot, kako deluje puščanje in izhlapevanje feromona ter kako se mravlja odloči, kam naj gre.

Algoritem je sestavljen iz m mravelj. Posamezna mravlja je agent, ki koraka od mesta do mesta in sestavlja pot. Vsako mesto lahko obišče le enkrat. Ko ne more obiskati nobenega mesta več, konča s korakanjem. Končna pot mravlje prikazuje rešitev za PTP, pri čemer povežemo začetno in zadnje obiskano mesto mravlje.

Feromon je dodatna vrednost vsake povezave v PTP-ju, označena s τ_{ij}. Velja $\tau_{ij} = \tau_{ji}$, saj imamo simetrični PTP. Feromon se posodobi, ko vse mravlje zaključijo svoje poti. Posodabljanje feromona vsebuje izhlapevanje, s katerim pomanjšamo trenutne vrednosti feromona vseh povezav, in dodajanje feromona, ko prehojenim povezavam povečamo feromon.

Za izračun, koliko feromona bo pustila mravlja k na povezavi med mestom i in mestom j, uporabimo enačbo

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{C(R_k)} & \text{če rešitev } k\text{-te mravlje vsebuje povezavo } (i,j) \\ 0 & \text{drugače} \end{cases} \tag{7.1}$$

k predstavlja indeks mravlje, R_k rešitev mravlje k in Q naprej podan parameter.

Končno število dodanega feromona na povezavi med mestom i in mestom j izračunamo z enačbo

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{7.2}$$

m v enačbi predstavlja število mravelj.

Končno posodobitev feromona za povezavo med mestoma i in j, ki vključuje tudi izhlapevanje feromona, izračunamo z enačbo

$$\tau_{ij}(t+1) = \rho * \tau_{ij}(t) + \Delta\tau_{ij} \tag{7.3}$$

$\tau_{ij}(t)$ predstavlja feromon na povezavi med mestoma i in j ob iteraciji t, ρ pa koeficient izhlapevanja, podan kot parameter.

Pri izbiri poti mora imeti mravlja na razpolago pot z manj ali pa brez feromona. V našem algoritmu to pomeni, da mora imeti mravlja v določenem mestu možnost izbrati katero koli povezavo do še ne obiskanega mesta.

Vpeljemo hevristično oceno za povezavo med mestoma i in j η_{ij}, ki jo izračunamo kot inverz cene povezave med mestoma i in j, $\eta_{ij} = \frac{1}{c_{i,j}}$.

Verjetnost, da mravlja k, ki se trenutno nahaja v mestu i, izbere povezavo, ki jo pripelje do mesta j, izračunamo z enačbo

$$P_{ij}^{k} = \begin{cases} \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{t \in J_k}[\tau_{it}]^{\alpha}[\eta_{it}]^{\beta}} & \text{če } j \in J_k \\ 0 & \text{drugače} \end{cases} \tag{7.4}$$

α predstavlja koeficient za kontrolo pomembnosti hevristične ocene, β koeficient za kontrolo pomembnosti feromona in J_k množico vseh neobiskanih mest za mravljo k.

Algoritem Kolonije mravelj deluje v več iteracijah. V začetnem delu iteracije sestavljamo poti mravelj, pri čemer je začetno mesto mravelj naključno. Mravlje sestavljajo svoje poti s pomočjo enačbe (7.4), dokler ne obiščejo vseh mest. Rešitve vseh mravelj ocenimo in preverimo, ali je najboljša rešitev boljša od do sedaj najboljše rešitve. Če je, posodobimo do sedaj najboljšo rešitev. Iteracijo zaključimo s posodabljanjem feromona. Koliko feromona moramo dodati na posamezni povezavi, izračunamo z enačbama (7.1) in (7.2). Feromon na vseh povezavah posodobimo po enačbi (7.3). Algoritem ponavlja iteracijo, dokler ne doseže končnega števila iteracij.

Pseudo koda algoritma Kolonije mravelj je prikazana v 4.

Algorithm 4 Kolonija mravelj

```
1: Inicializacija mravelj
2: for all Iteracije do
3:     for all Mravlje do
4:         Ponastavi mravljo
5:         Pridobi začetno mesto mravlje
6:         while Dokler trenutna mravlja ni dokončala svoje poti do
7:             Izberi naslednjo mesto mravlje po (7.4)
8:     Posodobi najboljšo rešitev
9:     for all Povezave do
10:        for all Mravlje do
11:            Izračunaj doprinos feromona mravlje za povezavo po (7.1)
12:        Izračunaj skupen doprinos feromona vseh mravelj za povezavo
   po (7.2)
13:        Posodobi feromon na povezavi po (7.3)
14: return Najboljšo rešitev
```

Algoritem Kolonije mravelj smo implementirali in ga 30-krat testirali na primerkih bayg29, brg180, fl417 in si1032. Zaradi trajanja izvedbe algoritma smo izpustili primerka u1432 in rl1889.

Uporabili smo parametre 20 mravelj, 1500 iteracij, $0,95$ koeficienta izhla-

Ime primerka	Rešitev	Oddaljenost (%)	Čas(ms)	Najboljša(ms)
bayg29	1690	4,97	603	93
brg180	2565	31,54	50009	25443
fl417	27639	133,02	516323	263738
si1032	230227	148,49	7161785	3669255

Tabela 7.1: Rezultati algoritma Kolonije mravelj

pevanja, $\alpha = 2$, $\beta = 2$, $Q = 1$ in 0 feromona na povezavah ob začetku algoritma.

Parametri so bili pridobljeni na osnovi predlogov iz literature in testiranja algoritma na primerkih z manj mesti.

7.1 Rezultati

Rezultati so predstavljeni v tabeli 7.1. V prvem stolpcu tabele se nahajajo poimenovanja primerkov, v drugem stolpcu so zapisane povprečna cena rešitev, v tretjem oddaljenosti povprečne cene rešitev od optimalne, izračunane z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu so zapisani povprečni časi algoritma v milisekundah, v petem pa povprečni časi, potrebni za to, da je algoritem našel najboljšo rešitev od začetka (v milisekundah).

Rezultati algoritma Kolonije mravelj so slabi. Rešitve so slabe, saj so zelo oddaljene od optimalne rešitve, čas izvajanja algoritma pa je zelo dolg. Algoritem je za primerek si1032 potreboval malo manj kot dve uri.

Razlog za slab čas algoritma je posodabljanje matrike feromona velikosti $N \times N$ ob koncu vsake iteracije in ponovno računanje verjetnosti izbire naslednjega mesta za vsako izbiro naslednjega mesta vseh mravelj.

Kontrolni čas nakazuje, da bi algoritem v dodatnem času težko našel boljšo rešitev, saj najde najboljšo rešitev za primerek si1032 v približno polovici celotnega časa, kar pomeni, da v drugi polovici poteka ni mogel izboljšati

rešitve.

7.2 Spremembe in izboljšave

Algoritmu Kolonije mravelj lahko dodamo elitne mravlje[1]. Ta sprememba omogoči puščanje feromona le najboljšim mravljam in ne vsem.

Možno je dodati lokalno posodabljanje feromona. Pri tej spremembi povezavam posodobimo feromon v trenutku, ko so prehojene, ne le ob koncu iteracije.

Spremenimo lahko tudi izbiro naslednjega mesta mravlje. Pri tem potrebujemo dodaten parameter $q_0 \in [0, 1]$, ki predstavlja verjetnost, ali naj gre mravlja k mestu z največjo verjetnostjo ali pa naj za izbiro naslednjega mesta uporabi enačbo (7.4).

7.3 Uporabljena literatura in dodatna literatura

Algoritem je bil predlagan v delu [11], ki poleg osnovnega opisa algoritma vsebuje tudi analizo parametrov in določenih izboljšav.

Delo [4] vsebuje dober pregled algoritma in nekaj njegovih različic.

Dobro delo za lokalno posodabljanje, elitizem, alternativni način, izbiro naslednjega mesta in analizo algoritma se najde v [10], kjer je tudi predstavljen algoritem Kolonija mravelj 3-opt[2].

Zanimivo je delo [12], v katerem so mravlje razdeljene na dve skupini: prva skupina se osredotoči na raziskovanje, druga pa na izkoriščanje.

[1] angl. Elite ants

[2] angl. Ant Colony System-3-opt

Poglavje 8

Optimizacija z roji delcev

Algoritem Optimizacije z roji delcev[1] izhaja iz obnašanja živali, primarno ptic, ki živijo v skupinah. Življenje v skupini je lahko slabost, saj mora posameznik deliti svoje vire s celotno skupino, a se izkaže, da v primeru ptic moč skupine in informacije, pridobljene od skupine, kompenzirajo s slabostmi.

Algoritem Optimizacije z roji delcev simulira obnašanje živali, specifično ptic, v skupinah. Simulira letenje ptic, ki si med seboj izmenjujejo informacije o obiskanih lokacijah.

Algoritem je sestavljen iz več agentov, imenovanih delci[2]. Vsak delec ima svojo trenutno lokacijo p_i, smer potovanja v_i in najboljšo do sedaj najdeno lokacijo $pbest_i$. V spremenljivki *best* se hrani tudi indeks delca, ki je našel najboljšo lokacijo v celotni skupini.

Vsak delec naredi korak po enačbi

$$p_i(t+1) = p_i(t) + v_i(t+1) \tag{8.1}$$

$$v_i(t+1) = w * v_i(t) + c_1 r_1(pbest_i - p_i(t)) + c_2 r_2(pbest_{best} - p_i(t)) \tag{8.2}$$

t predstavlja trenutno iteracijo algoritma, i indeks delca, w koeficient, koliko pretekle smeri delca želimo pozabiti, c_1 in c_2 sta koeficienta za konvergenco

[1] angl. Particle Swarm Optimization
[2] angl. Particle

delca proti svoji najboljši lokaciji in konvergenco delca do najboljše najdene lokaciji celotne skupine, r_1 in r_2 pa sta naključno ustvarjena vektorja z vrednostmi v intervalu $[0, 1]$.

Ideja za korakom je, da vsak delec naredi korak v svojo smer, se približa svoji do sedaj najboljši lokaciji in najboljši lokacije celotne skupine.

Ta algoritem ne deluje na PTP-ju, saj so lokacije, ki predstavljajo rešitve problema, v zveznem prostoru $\mathbb{R}^N$ in PTP ne sprejema takšnih rešitev.

Kljub temu lahko poskusimo idejo Optimizacije z roji delcev pretvoriti v algoritem za reševanje PTP-ja. Delo [15] prikazuje en način uporabe ideje Optimizacije z roji delcev za reševanje PTP-ja.

V omenjenem delu se lokacija posameznega delca p_i in do sedaj najboljša lokacija posameznega delca $pbest_i$ ohranita, a sedaj vsebujeta rešitev za PTP. Ohrani se tudi shranjevanje indeksa $best$, zavrže pa se smer potovanja posameznega delca v_i.

Korak delca se razdeli na tri korake, pri čemer posamezen korak predstavlja en del vsote enačbe (8.2). V vsaki iteraciji vsak delec izbere le enega izmed treh korakov. Verjetnost izbire določenega koraka je podana kot parameter, pri čemer P_1 predstavlja verjetnost za prvi korak, P_2 verjetnost za drugi korak, P_3 pa verjetnost za tretji korak.

Verjetnost izbire določenega koraka se ob koncu iteracije spremeni po enačbi

$$P_1 = P_1 * f_1;\ P_2 = P_2 * f_2;\ P_3 = 1 - (P_1 + P_2) \tag{8.3}$$

Faktorja f_1 in f_2 sta podana kot parametra.

V naši implementaciji bomo verjetnosti izbire določenega koraka ob koncu iteracije spreminjali na drugačen način. Verjetnosti se bodo spreminjale po naslednji enačbi

$$P_1 = P_1 * f_1;\ P_2 = P_2 * f_2;\ P_3 = P_3 * f_3$$

$$P_1 = \frac{P_1}{P_1 + P_2 + P_3};\ P_2 = \frac{P_2}{P_1 + P_2 + P_3};\ P_3 = \frac{P_3}{P_1 + P_2 + P_3} \tag{8.4}$$

Faktorji f_1, f_2 in f_3 so podani kot parametri. Tega načina se poslužujemo, ker nam omogoča lažje spreminjanje in boljši pregled nad verjetnostmi izbire določenega koraka skozi potek algoritma.

Prvi korak, ki predstavlja prvi del vsote, je korak delca v svojo smer. Implementiran je s pomočjo dodatnega algoritma, ki se približa lokalnemu optimumu rešitve p_i delca. Avtorica algoritma za ta korak predlaga uporabo algoritma Lokalnega iskanja z operacijo, ki je podobna operaciji 2-opt, in Lin-Kernighanov algoritem.

Naša implementacija bo uporabljala algoritem Lokalnega iskanja z operacijo 2-opt in strategijo prvega boljšega ter Lin-Kernighanov algoritem. Prvemu koraku smo dodali možnost, da se lahko izbran algoritem izvede večkrat ob izbiri prvega koraka, pri čemer se kot vhod za naslednjo izvedbo izbranega algoritma uporabi rešitev, pridobljena od pretekle uporabe izbranega algoritma.

Drugi in tretji korak predstavljata približevanje delca do svoje najboljše lokacije in do najboljše lokacije celotne skupine. Implementirana sta s pomočjo algoritma Vzpostavljanja poti[3].

Algoritem Vzpostavljanja poti potrebuje začetno rešitev R_z in ciljno rešitev R_c. Algoritem postopoma premika mesta v začetni rešitvi na levo ali desno za eno pozicijo, dokler začetna rešitev ni enaka končni rešitvi.

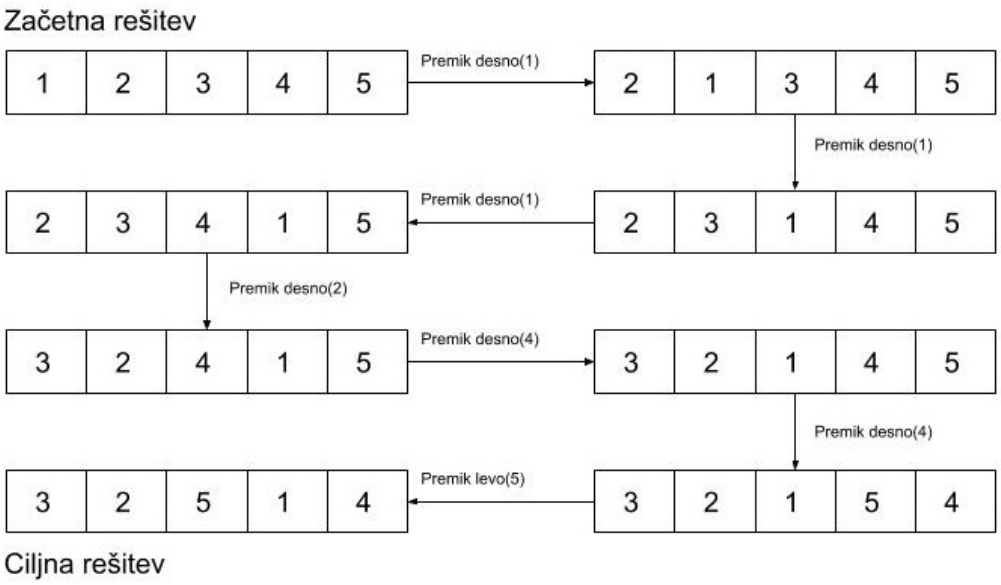
Primer poteka algoritma Vzpostavljanja poti se najde na sliki 8.1.

Pri drugem koraku se uporabi algoritem Vzpostavljanja poti, pri katerem je rešitev delca p_i začetna rešitev, najboljša rešitev delca $pbest_i$ pa ciljna rešitev. Pri tretjem koraku je rešitev delca p_i začetna rešitev, najboljša rešitev v celotni skupini $pbest_{best}$ pa ciljna rešitev.

Ne želimo si, da bi v drugem in tretjem koraku trenutna rešitev delca p_i postala najboljša rešitev delca $pbest_i$ ali najboljša rešitev celotne skupine $pbest_{best}$. Želimo, da se ji le približa. Zato moramo algoritem Vzpostavljanja poti v določenem koraku ustaviti.

To dosežemo tako, da pridobimo oceno, koliko korakov je potrebnih,

[3]angl. Path relinking



Slika 8.1: Primer algoritma vzpostavljanja poti z začetno rešitvijo $(1, 2, 3, 4, 5)$ in ciljno rešitvijo $(3, 2, 5, 1, 4)$

da algoritem Vzpostavljanja poti spremeni začetno rešitev v ciljno. Oceno izračunamo kot vsoto vseh razlik pozicij med mesti v začetni in ciljni rešitvi. Če bi imeli začetno rešitev $(1, 2, 3, 4, 5)$ in ciljno rešitev $(3, 2, 5, 1, 4)$, bi pridobili oceno $3 + 0 + 2 + 1 + 2 = 7$, kjer prvi del vsote 3 prikazuje, koliko pozicij moramo prestaviti mestu 1 v začetni rešitvi, da pridobi pozicijo mesta 1 v ciljni rešitvi, drugi del vsote pa predstavlja enako kot prvi del, vendar za mesto 2 itd. Pridobljeno oceno pomnožimo s faktorjem, ki ga pridobimo kot parameter. Rezultat uporabimo kot število korakov, ki naj jih algoritem Vzpostavljanja poti naredi.

Pseudo koda algoritma Optimizacije z roji delcev je prikazana v 5.

Avtorica predlaga, da naj bo v začetku algoritma verjetnost za izbiro prvega koraka visoka, verjetnost za drugi korak naj bo možna, a ne prepogosta, verjetnost za tretji korak pa naj bo zelo nizka. Skozi potek algoritma naj se verjetnost prvega koraka niža, verjetnost drugega koraka malenkost viša,

Algorithm 5 Optimizacija z roji delcev

```
Pridobi verjetnosti korakov in inicializacija delcev
for all Iteracije do
    for all Delci i do
        if Verjetnost za prvi korak then
            for Število izvedb algoritma do
                p_i = Izbran_Algoritem(p_i)
        else if Verjetnost za drugi korak then
            p_i = Vzpostavi_pot(p_i, pbest_i)
        else
            p_i = Vzpostavi_pot(p_i, pbest_best)
    Posodobi best, če obstaja delec, ki ima boljšo rešitev kot pbest_best
    Posodobi verjetnosti po (8.4)
return Najboljšo rešitev
```

verjetnost tretjega pa precej viša. Ob koncu algoritma naj bo verjetnost prvega koraka redka in najnižja, verjetnost drugega koraka smiselno možna, verjetnost tretjega koraka pa najpogostejša.

Implementirali smo algoritem Optimizacije z roji delcev, ki ima kot prvi korak algoritem Lokalnega iskanja z operacijo 2-opt in strategijo prvega boljšega ter Lin-Kernighanov algoritem. Na vsakem primerku smo ga testirali 30-krat.

Algoritem Optimizacije z roji delcev, ki ima za prvi korak algoritem Lokalnega iskanja z operacijo 2-opt in strategijo prvega boljšega, bo ob vsaki izbiri prvega koraka izvedel algoritem Lokalnega iskanja $\frac{N}{10} + 5$-krat. Ob posamezni izvedbi bo pregledal N možnih izbir povezav.

Če rešitev ob izbiri prvega koraka z izbranima parametroma ne bo izboljšana, vemo, da smo pregledali $N(\frac{N}{10} + 5)$ možnih izbir povezav. Parametra smo izbrali na podlagi razširljivosti na primerke z večjimi mesti. Pri več ponovnih izbirah prvega koraka, kjer se rešitev ne izboljša, bomo pregledali vse možne izbire povezav, saj $\lim_{N\to\infty} \frac{N(\frac{N}{10}+5)}{\binom{N}{2}} = \frac{1}{5}$.

Algoritem Optimizacije z roji delcev, ki ima za prvi korak Lin-Kernighanov algoritem, bo ob izbiri prvega koraka 10-krat izvedel Lin-Kernighanov algoritem, pri čemer bo uporabljal vse nivoje vračanja, seznam bližnjih sosedov dolžine 10 in vračanje na prvi nivo $\frac{N}{20} + 10$-krat.

Lin-Kernighanov algoritem lahko za prvo mesto izbere N mest. Po izbiri prvega mesta je njegovo delovanje deterministično. Ob vsaki izvedbi algoritma preverimo $\frac{N}{20} + 10$ možnih izbir prvega mesta. Če ne izboljšamo rešitve ob izbiri prvega koraka, vemo, da smo pregledali $(\frac{N}{20} + 10) * 10 = \frac{N}{2} + 100$ prvih mest za podano rešitev, kar pokriva vsa mesta v primerkih z manj mesti in dobro polovico mest v primerkih z več mesti.

Drugi parametri, ki so bili pridobljeni na osnovi predlogov iz literature in testiranja algoritma na primerkih z manj mesti, so enaki za oba algoritma. Preostali parametri so 25 delcev, 2000 iteracij, $0,6$ faktor koraka za algoritem Vzpostavljanja poti, $P_1 = 0,68$, $f_1 = 0,999$, $P_2 = 0,3$, $f_2 = 1,0006$, $P_3 = 0.02$ ter $f_3 = 1,002$. Verjetnosti ob koncu algoritma so $P_1 = 0,042265$, $P_2 = 0,457736$, $P_3 = 0,499999$.

8.1 Rezultati

Rezultati za algoritem Optimizacije z roji delcev z operacijo 2-opt so predstavljeni v tabeli 8.1, za algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom pa v tabeli 8.2. V prvem stolpcu tabel se nahajajo pomenovanja za primerke, v drugem stolpcu imamo povprečni oceni rešitev, v tretjem so zapisane oddaljenosti povprečne cene rešitev od optimalne, izračunane z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu imamo povprečne čase algoritma v milisekundah, v petem pa povprečne čase (v milisekundah), potrebne za to, da je algoritem našel najboljšo rešitev od začetka.

Algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom ima pri primerkih brg180, u1432 in rl1889 boljše rešitve kot algoritem Optimizacije z roji delcev z operacijo 2-opt. Pri primerku bayg29 sta povprečno oba algoritma dosegla optimalno rešitev.

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1610	0	132	21
brg180	2094	7,38	3331	1024
fl417	12169	2,6	12459	6762
si1032	92975	0,35	94813	55652
u1432	172009	12,45	288220	127946
rl1889	355118	12,19	568200	276046

Tabela 8.1: Rezultati algoritma Optimizacije z roji delcev z operacijo 2-opt

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1610	0	4109	2
brg180	1950	0	4219	387
fl417	12199	2,85	11964	6990
si1032	93208	0,6	25957	11502
u1432	156017	1,99	52658	18848
rl1889	351378	11,01	83648	47957

Tabela 8.2: Rezultati algoritma Optimizacije z roji delcev z Lin-Kernighanovim algoritmom

Čas, potreben za algoritem, je pri obeh algoritmih praktično sprejemljiv. Za algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom potrebujemo na primerkih z manj mesti več časa, na primerkih z več mesti pa manj časa.

Zanimiv je tudi kontrolni čas. Pri tem algoritmu s kontrolnim časom ne moremo napovedati, ali bi dodaten čas algoritmu pomagal najti boljšo rešitev, saj algoritem deluje v več fazah, odvisno od verjetnosti korakov. Lahko bi sklepali, v kateri fazi je algoritem pridobil najboljšo rešitev, in na podlagi tega podaljšali faze pred in ob tem času.

8.2 Spremembe in izboljšave

Algoritem Optimizacije z roji delcev je narejen za zvezne probleme. Enoten algoritem Optimizacije z roji delcev za reševanje PTP-ja ne obstaja, zato je težko najti izboljšave. Posamezna dela obdelujejo svoj algoritem, ki kot podlago uporablja idejo Optimizacije z roji delcev.

Kljub temu se najde način, ki lahko algoritme za reševanje zveznih problemov pretvori v algoritme za reševanje PTP-ja, in sicer: Imamo zvezno rešitev $\mathbb{R}^N$, kjer je N število mest v primerku PTP-ja. Za vsako takšno zvezno rešitev lahko pridobimo ceno rešitve za PTP tako, da rečemo, da je na poziciji z najmanjšo vrednostjo mesto 1, na poziciji z drugo najmanjšo vrednostjo mesto 2 itd. Preostali deli algoritma ostanejo enaki. Način deluje, a ni priporočljiv, saj ne izkoristi lastnosti PTP-ja.

Izpostavili bomo algoritem Vzpostavljanja poti. Implementacija algoritma Vzpostavljanja poti gleda na pozicijo mest, ko spreminja začetno rešitev v ciljno. Bolje bi bilo, če bi algoritem gledal na to, katere povezave so v ciljni in ne v začetni rešitvi in nato postopoma s pomočjo operacij 2-opt vzpostavljal manjkajoče povezave. Za naš algoritem ta sprememba ni nujno izboljšava. Algoritem Vzpostavljanja poti zavzame v našem algoritmu Optimizacije z roji delcev vlogo raziskovanja. Trenutna implementacija to omogoča bolje kot algoritem Vzpostavljanja poti, ki gleda na povezave in

uporablja operacijo 2-opt.

V celoti lahko na ta algoritem gledamo kot na izbiro določenega algoritma, ki zavzame vlogo izkoriščanja rešitev in uporabe algoritma Vzpostavljanja poti, ki zavzame vlogo raziskovanja. Ta dva algoritma delujeta nad več rešitvami. Izbiramo lahko, ali bomo v trenutni iteraciji izkoriščali ali raziskovali rešitve, odvisno od vnaprej podanih verjetnosti. Če gledamo na celotni algoritem s tega vidika, bi lahko spremenili algoritem v vlogi izkoriščanja in algoritem v vlogi raziskovanja.

Za algoritem Optimizacije z roji delcev bi lahko dodali še dodatno število iteracij, ki jih mora algoritem narediti ob določenih verjetnostih. To bi omogočilo lažje dodajanje iteracij brez ponovnega ugotavljanja, kakšne bi morale biti začetne verjetnosti korakov in faktorji za spreminjanje verjetnosti korakov, da bi dosegli podoben potek verjetnosti korakov skozi delovanje algoritma.

8.3 Uporabljena literatura in dodatna literatura

Algoritem Optimizacije z roji delcev je bil predlagan v delu [18].

Poglavje 9

Oponašanje volkov

Algoritem Oponašanja volkov[1] izhaja iz obnašanja skupine volkov pri lovljenju plena. To obnašanje lahko razdelimo na tri korake. Prvi korak je skavtstvo za plen, drugi je obkroževanje plena, tretji pa napad plena.

Za zvezne probleme in za reševanje PTP-ja ne obstaja enoten algoritem Oponašanja volkov. Kljub temu imajo algoritmi, ki imajo kot podlago obnašanje volkov pri lovljenju, skupne zgoraj omenjene korake. Pri tem lahko posamezen algoritem določen korak razdeli na več korakov ali uporabi druge lastnosti skupine volkov, npr. hierarhijo skupine volkov. Delo [9] prikazuje način, kako lahko uporabimo idejo Oponašanja volkov za reševanje PTP-ja. Ta algoritem ohrani tri korake in doda dodaten korak za izogibanje lokalnih optimumov. Korak napad plena je implementiran s pomočjo algoritma Lokalnega iskanja.

Algoritem poteka v več iteracijah, z več agenti, imenovanimi volkovi. Vsak volk ima svojo rešitev v_i. Indeks volka z najboljšo rešitvijo v skupini je hranjen v spremenljivki *best*. Posamezno iteracijo lahko razdelimo na štiri korake.

Prvi korak predstavlja skavtstvo posameznega volka v_i. Imamo spremenljivko k, ki jo iteriramo skozi množico $(1, 2, ..., N)$. Za vsak k pridobimo k' po enačbi

[1] angl. Wolfpack

$$k' = \begin{cases} rand(N) & w \leq W \\ j & min(c_{k,j}),\ w > W \end{cases} \tag{9.1}$$

$$k \neq k'$$

kjer $rand(N)$ naključno ustvari število med 1 in N, $W \in [0, 1]$ je podan parameter, $w \in [0, 1]$ naključno pridobljeno število, $min(c_{k,j})$ pa je funkcija, ki nam pove, da je mesto j najbližje mestu k.

Zgornja enačba pridobi k', ki predstavlja naključno mesto ali pa mesto, ki je najbližje mestu k.

V primeru, ko sta mesti k in k' soseda v rešitvi trenutnega volka v_i, gremo na naslednji k in pridobimo nov k'. Če smo skozi celotno množico iterirali za k, gremo na naslednjega volka.

Po izbiri k' pridobimo novo rešitev v_i' z obratom vrstnega reda tako, da mesto k' postane sosed mesta k, ob čemer ne spreminjamo pozicije mesta k. Recimo, da imamo $v_i = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, $k = 3$ in $k' = 7$, pridobimo $v_i' = (1, 2, 3, 7, 6, 5, 4, 8, 9)$, kar predstavlja operacijo 2-opt med povezavama $(3, 4)$ in $(7, 8)$.

Če je $C(v_i') < C(v_i)$, posodobimo rešitev v_i. Če je $C(v_i) < C(v_{best})$, posodobimo *best*.

Drugi korak predstavlja zaokroževanje plena. Vsak volk v_i naredi korak proti volku z najboljšo rešitvijo v_{best}.

Pri volku v_{best} naključno izberemo mesto a, kjer je njegovi levi sosed a_L, desni sosed pa a_R. Na primer $v_{best} = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, kjer je $a = 9$, pridobimo $a_L = 8$ in $a_R = 1$.

V tem koraku ustvarimo tri nove rešitve v_{i_a}, v_{i_b} in v_{i_c}.

Rešitev v_{i_a} pridobimo tako, da damo mesto a_L v rešitvi v_i pred mesto a. Če je mesto a_L v rešitvi v_i levo od mesta a, se vrstni red mest obrne tako, da je mesto a_L pred mestom a, ob tem pa ne spreminjamo pozicije mesta a. Če je a_L desno od mesta a, vzamemo mesto a_L iz rešitve v_i in ga vstavimo na pozicijo levo od mesta a. Npr. pri $a = 5$, $a_L = 7$,

$a_R = 3$ in $v_i = (8, 7, 3, 4, 9, 6, 2, 1, 5)$ bi pridobili $v_{i_a} = (8, 1, 2, 6, 9, 4, 3, 7, 5)$, pri čemer, če bi imeli rešitev $v_i = (5, 1, 2, 6, 9, 4, 3, 7, 8)$, bi pridobili $v_{i_a} = (7, 5, 1, 2, 6, 9, 4, 3, 8)$.

Rešitev v_{i_b} pridobimo tako, da damo mesto a_R v rešitvi v_i po mestu a. Če je a_R desno od mesta a v rešitvi v_i, se vrstni red mest obrne tako, da mesto a_R sledi mestu a, pri tem ne spreminjamo pozicije mesta a. Če je a_R levo od mesta a, vzamemo mesto a_R iz rešitve v_i in ga vstavimo na pozicijo desno od mesta a.

Rešitev v_{i_c} pridobimo tako, da nad rešitvijo v_i naredimo operacijo, enako kot za v_{i_a}, nato pa še operacijo, enako kot za v_{i_b}.

Če je trenutna rešitev volka v_i slabša kot najboljša rešitev med v_{i_a}, v_{i_b} in v_{i_c}, jo posodobimo. Če je $C(v_i) < C(v_{best})$, posodobimo *best*.

Tretji korak se izogiba lokalnih optimumov. Kadar *best* ostane nespremenjen za določeno število iteracij, dodamo v skupino nove volkove. Kadar skupina doseže maksimalno število volkov, se namesto dodajanja novih volkov volka v_{best} prisili k skavtstvu. Ta korak smo spremenili. Algoritem vedno začne z maksimalnim številom volkov, kar onemogoči dodajanje volkov. Kadar koli pride do situacije, kjer je *best* nespremenjen za določeno število iteracij, se prisili volka v_{best} k določenemu številu korakov skavtstva, pri katerih mora sprejeti pridobljeno rešitev tudi v primeru, če je ta slabša.

Ta sprememba omogoči, da algoritem najde boljšo rešitev, kot je rešitev v volku v_{best} ob koncu algoritma. To je razlog, da dodatno hranimo informacijo o najboljši najdeni rešitvi.

Četrti korak predstavlja napad. Implementiran je s pomočjo algoritma Lokalnega iskanja.

Ta korak naredi vsak par volkov v_i in v_j. Naključno izberemo mesto a. Če je a zadnje mesto v prvem volku v_i ali drugem volku v_j, ponovno izberemo a. Mesto desno od mesta a v volku v_j je a_j. V volku v_i poiščemo mesti a in a_j. Pridobimo rešitev v_i' z obratom vrstnega reda mest v rešitvi v_i tako, da sta mesti a in a_j soseda, pri tem ne spreminjamo pozicije mesta a. Npr. $v_i = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, $v_j = (4, 6, 8, 7, 2, 1, 5, 3, 9)$, $a = 5$ in $a_j = 3$.

Pridobili bi rešitev $v_i' = (1, 2, 4, 3, 5, 6, 7, 8, 9)$, kar predstavlja operacijo 2-opt med povezavama $(2, 3)$ in $(4, 5)$. Če je pridobljena rešitev boljša kot prejšnja $C(v_i') < C(v_i)$, jo posodobimo in gremo na naslednji par volkov. Če ni, pridobimo rešitev v_j' tako, da naredimo operacijo 2-opt, ki poveže povezavo, odstranjeno v izdelavi v_i', ki vsebuje a_j, tako da ne spremenimo pozicije mesta a_j. Nadaljujmo na prejšnji primer in predpostavimo, da $C(v_i') \geq C(v_i)$. S pomočjo operacije 2-opt poskusimo vzpostaviti povezavi $(2, 3)$ v rešitvi v_j, saj je $a_j = 3$. To naredimo z 2-opt operacijo med povezavama $(7, 2)$ in $(5, 3)$ v rešitvi $v_j = (4, 6, 8, 7, 2, 1, 5, 3, 9)$. Rezultat je $v_j' = (4, 6, 8, 7, 5, 1, 2, 3, 9)$.

Ideja algoritma Lokalnega iskanja v četrtem koraku je izbrati naključno povezavo v drugi rešitvi in jo vzpostaviti v prvi rešitvi. Če je pridobljena rešitev slabša, predpostavimo, da je odstranjena povezava boljša od povezavea, ki smo jo vzpostavili. Zaradi tega poskusimo odstranjeno povezavo vzpostaviti v drugi rešitvi. Algoritem Lokalnega iskanja išče lokalni optimum s pomočjo medsebojnega učenja volkov.

Pseudo koda algoritma Oponašanja volkov je prikazana v 6.

Algoritem Oponašanja volkov smo implementirali in ga na vsakem primerku testirali 30-krat. Uporabili smo parametre 160 volkov, 17000 iteracij, $W = 0,75$ ter, če za 100 iteracij ostane *best* nespremenjen, prisilimo volka v_{best} k 300 korakom skavtstva.

Parametri so bili pridobljeni na osnovi predlogov iz literature in testiranja algoritma na primerkih z manj mesti.

9.1 Rezultati

Rezultati so predstavljeni v tabeli 9.1. V prvem stolpcu tabele so poimenovani primerki, v drugem stolpcu je zapisana povprečna cena rešitev, v tretjem stolpcu so zapisane oddaljenosti povprečne cene rešitev od optimalne, izračunane z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$, v četrtem stolpcu so navedeni povprečni časi algoritma v milisekundah, v petem pa povprečni časi, potrebni, da je algoritem našel najboljšo rešitev (v milisekundah).

Algorithm 6 Oponašanje volkov z Lokalnim iskanjem

1: Inicializacija volkov
2: **for all** Iteracije **do**
3: **for all** Volki i **do**
4: Skavtstvo volka i, kakor opisano (prvi korak)
5: Če je potrebno, posodobi *best*
6: **for all** Volki **do**
7: Zaokroževanje plena, kjer se vsak volk i približa volku *best*, kakor opisano (drugi korak)
8: Če je potrebno, posodobi *best*
9: **if** Ali je *best* nespremenjen za določeno število iteracij? **then**
10: volk *best* naredi nekaj korakov skavtstva, kakor opisano (tretji korak)
11: **for all** Volkovi i **do**
12: **for all** Volkovi $j = i + 1$ **do**
13: Lokalno iskanje med volkom i in volkom j kakor opisano (četrti korak)
14: **return** Najboljšo rešitev

Ime primerka	**Rešitev**	**Oddaljenost (%)**	**Čas(ms)**	**Najboljša(ms)**
bayg29	1610	0	12880	11
brg180	2026	3,9	19394	8709
fl417	12298	3,68	25038	5177
si1032	92797	0,16	39830	16578
u1432	165774	8,37	72755	60466
rl1889	347043	9,64	99636	72679

Tabela 9.1: Rezultati algoritma oponašanja volkov.

Algoritem Oponašanja volkov pridobi dobre rešitve v primernem času. Primerek rl1889 je zaključil v manj kot dveh minutah.

Sesatava algoritem je zelo zanimiva. Izven tretjega koraka, ki je strogo omejen, saj spremeni le enega volka in ima predpogoj, da mora biti najboljši za določeno število iteracij, ni skoraj nobenega raziskovanja.

Kontrolni čas nam pove, da bi algoritem lahko našel boljše rešitve v primerkih z več mesti, če bi mu dali več časa, saj je kontrolni čas zanj nizek. Pri primerkih z manj mesti pa je kontrolni čas visok. Iz tega lahko sklepamo, da je imel algoritem dovolj časa, da je pregledal večino rešitev, ki jih lahko pregleda s svojimi operacijami.

9.2 Spremembe in izboljšave

Kot pri algoritmu Optimizacije z roji delcev, za reševanje PTP-ja tudi ni enotnega algoritma Oponašanja volkov. Zaradi tega je v drugi literaturi težko najti izboljšave zanj.

Izboljšamo lahko raziskovanje algoritma. Tretji korak bi lahko nadomestili s tem, da bi iz trenutne skupine volkov izvzeli nekaj najboljših volkov in z njimi kreirali novo skupino.

9.3 Uporabljena literatura in dodatna literatura

Algoritem Oponašanja volkov, ki smo ga pregledali, izhaja iz algoritma, prestavljenega v delu [29]. Vir vsebuje algoritem Oponašanja volkov za zvezne probleme, vendar ni najbolje opisan. Ima tudi drugi vir, iz katerega črpa določene dele algoritma, a je napisan v kitajščini.

Najdemo lahko tudi druge algoritme, ki kot podlago za svoje delovanje uporabljajo obnašanje volkov pri lovu. V delu [21] najdemo algoritem, ki za reševanje zveznih problemov uporablja obnašanje volkov pri lovu in njihovo hierarhijo.

Poglavje 10

Primerjava rezultatov

V tem poglavju bomo komentirali rezultate algoritmov in komentirali metodologijo, uporabljeno za primerjavo algoritmov.

10.1 Oddaljenost algoritmov

V tabeli 10.1 so prikazane oddaljenosti vseh algoritmov. V prvem stolpcu so zapisana poimenovanja algoritmov. Naslednji stolpci ponazarjajo oddaljenost algoritmov pri določenem primerku, izračunano z enačbo $\frac{\text{Povprečna cena rešitev} - \text{Optimalna rešitev}}{\text{Optimalna rešitev}} * 100$. Zadnji stolpec ponazarja vsoto vseh oddaljenosti za algoritem v vrstici. V zadnji vrstici so zapisana povprečja oddaljenosti posameznih primerkov in povprečja skupnih vsot oddaljenosti, pri čemer pa zaradi pomanjkljivih podatkov ne vključujejo oddaljenosti algoritma Lokalnega iskanja z operacijo 3-opt in strategijo najboljšega in algoritma Kolonije mravelj.

Za poimenovanja algoritmov smo uporabili kratice, in sicer:

- PB2OPT za algoritem Lokalnega iskanja z operacijo 2-opt in strategijo prvega boljšega,
- N2OPT za algoritem Lokalnega iskanja z operacijo 2-opt in strategijo najboljšega,

Ime Algoritma	bayg29	brg180	fl417	si1032	u1432	rl1889	Vsota
PB2OPT	4,53	23,23	8,43	1,59	15,68	16,94	70,4
N2OPT	2,73	23,85	6,35	1,46	12,15	11,85	58,39
PB3OPT	2,73	3,54	6,46	1,5	14,48	14,92	43,63
N3OPT	1,93	0,72	2,93	0,46	N/A	N/A	6,04<
LK	0,62	2,67	39	2,06	4,37	25,98	74,7
SO2OPT	**0**	0,36	**1,89**	0,76	10,19	17,95	31,15
SO3OPT	**0**	0,1	2,13	1,61	16	32,61	52,45
KM	4,97	31,54	133,02	148,49	N/A	N/A	318,02<
ORD2OPT	**0**	7,38	2,6	0,35	12,45	12,19	34,97
ORDLK	**0**	**0**	2,85	0,6	**1,99**	11,01	**16,45**
OV	**0**	3,9	3,68	**0,16**	8,37	**9,64**	25,75
Povp.	1,18	7,23	8,15	1,12	10,63	17,01	45,32

Tabela 10.1: Oddaljenosti vseh algoritmov na vseh primerkih, vsote oddaljenosti posameznih algoritmov, povprečna oddaljenost za vsak primerek in povprečje vsote vseh oddaljenosti algoritmov, kjer povprečja ne vključujejo oddaljenosti od algoritma Kolonije mravelj in algoritma Lokalnega iskanja z operacijo 3-opt in strategijo najboljšega.

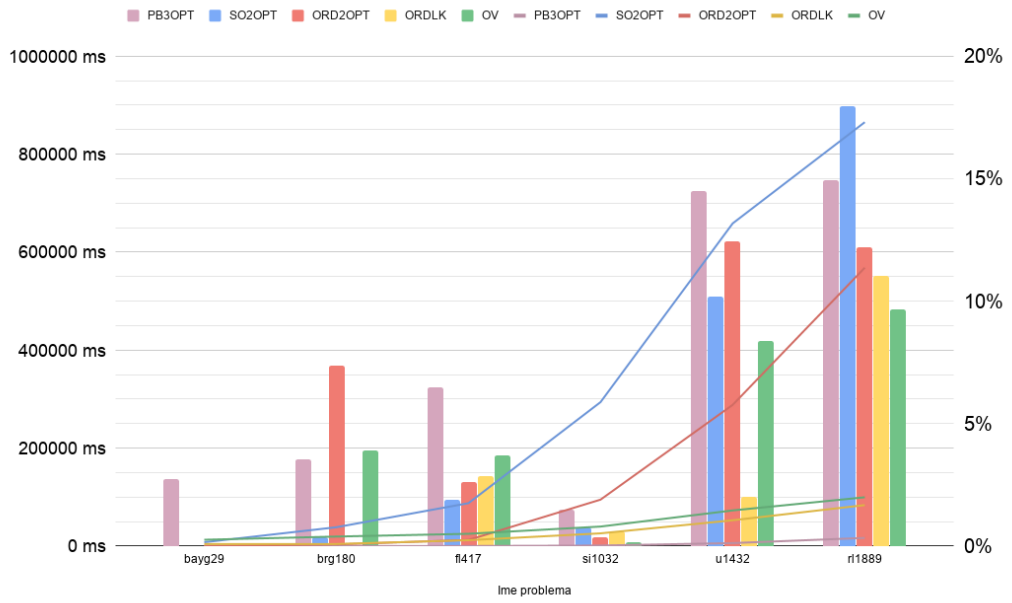
- PB3OPT za algoritem Lokalnega iskanja z operacijo 3-opt in strategijo prvega boljšega,
- N3OPT za algoritem Lokalnega iskanja z operacijo 3-opt in strategijo najboljšega,
- LK za Lin-Kernighanov algoritem,
- SO2OPT za algoritem Simuliranega ohlajanja z operacijo 2-opt,
- SO3OPT za algoritem Simuliranega ohlajanja z operacijo 3-opt,
- KM za algoritem Kolonije mravelj,
- ORD2OPT za algoritem Optimizacije z roji delcev z operacijo 2-opt,
- ORDLK za algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom,
- OV za algoritem Oponašanja volkov.

V tabeli so s krepko prikazane vrednosti algoritmov, ki imajo pri določenem primerku najnižjo oddaljenost. Označeno je, kateri algoritem ima najnižjo vsoto oddaljenosti.

Najnižjo vsoto oddaljenosti ima algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom, ki znaša $16,45$ in je $28,87$ nižja od povprečne vsote oddaljenosti algoritmov. To ne pomeni, da je algoritem najboljši, saj bi zanemarili čase izvajanja algoritmov, ki so pomemben del pri ocenjevanju algoritmov.

10.2 Oddaljenost s potrebnim časom algoritma

Predstavimo naslednji graf 10.1. Izbrali smo algoritme, ki imajo vsoto vseh oddaljenosti nižjo, kot je povprečna vsota vseh oddaljenosti. Razlog za izbiro teh algoritmov je preglednost grafa, saj bi ta postal nepregleden, če bi vanj vstavili vse algoritme. Algoritem Lokalnega iskanja z operacijo 3-opt



Slika 10.1: Graf oddaljenosti in časa izbranih algoritmov

in strategijo prvega boljšega je na grafu prikazan z roza barvo, algoritem Simuliranega ohlajanja z operacijo 2-opt z modro, algoritem Optimizacije z roji delcev z operacijo 2-opt z rdečo, algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom z rumeno in algoritem Oponašanja volkov z zeleno barvo.

Stolpci v grafu prikazujejo oddaljenost algoritmov pri določenem primerku. Same vrednosti stolpcev se navezujejo na desno navpično os grafa. Temnejše črte na garfu prikazujejo povprečen čas, potreben za to, da je algoritem zaključil svoje delovanje v določenem primerku. Vrednost črte se navezuje na levo navpično os grafa.

Po vrstnem redu si od najmanjše do največje vsote oddaljenosti izbranih algoritmov sledijo algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom z vsoto $16,45$, algoritem Oponašanja volkov z vsoto $27,75$, Simulirano ohlajanje z operacijo 2-opt z vsoto $31,15$, algoritem Optimizacije z roji delcev z operacijo 2-opt z vsoto $34,97$ in algoritem Lokalnega iskanja z

operacijo 3-opt in strategijo prvega boljšega z vsoto $43,63$.

Iz grafa je razvidno, da je algoritem z najmanj naraščajočim časom algoritem Lokalnega iskanja z operacijo 3-opt in strategijo prvega boljšega, a ima tudi najvišjo vsoto oddaljenosti. Temu sledi algoritem z najnižjo vsoto oddaljenosti, tj. algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom. Razlika med vsoto oddaljenosti teh dveh algoritmov je $27,18$, kar je dovolj, da lahko zaključimo, da razlika v vsotah oddaljenosti presega razlike v času. Glede na oddaljenost in čas, ki je bil potreben, da se je algoritem zaključil, lahko rečemo, da je algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom po naši metodologiji najboljši, saj ima najnižjo vsoto oddaljenosti in drugi najnižji čas med algoritmi, ki imajo vsoto oddaljenosti nižjo od povprečne vsote oddaljenosti vseh izbranih algoritmov.

10.3 Kritika metodologije primerjanja algoritmov

Izpostavili bomo pet razlogov, zakaj je uporabljena metodologija primerjave rezultatov pomanjkljiva.

Prvi razlog so skrite lastnosti primerkov. To smo izpostavili že pri algoritmu Lokalnega iskanja, kjer je operacija 3-opt pri primerku brg180 dosegla boljše rezultate od operacije 2-opt. Obstaja možnost, da smo izbrali primerke, ki so za ene algoritme primernejši kot za druge. Da bi lahko prišli do boljšega zaključka, bi morali izbrati več primerkov, tudi takšnih z enakim ali podobnim številom mest.

Drugi razlog je podatkovna struktura, kar smo omenili že v tretjem poglavju. Določen algoritem lahko pridobi časovno prednost s tem, da so njegove operacije v podatkovni strukturi bolje podprte kot operacije drugih algoritmov.

Tretji razlog je odvisen od same implementacije algoritma. Predstavljamo dva načina ocenjevanja in kako se bo cena rešitev z uporabo določene operacije spremenila. Prvi način je izvesti operacijo nad rešitvijo in preveriti ceno

nove pridobljene rešitve. Ta način je enostaven, saj potrebujemo le implementacijo operacije in standarden način ocenjevanja cen rešitev. Drugi način je teoretični izračun tega, za koliko se bo cena spremenila, če nad rešitvijo uporabimo določeno operacijo. Vsaka operacija iz rešitve odstrani in vanjo doda določene povezave. Namesto da izvedemo operacijo, lahko le primerno odštejemo cene odstranjenih povezav in k trenutni ceni rešitve prištejemo cene dodanih povezav ter tako pridobimo novo ceno rešitve. Ta način je težji, saj moramo predvideti, katere spremembe so bodo zgodile pri določeni operaciji, vendar pa je hitrejši. Algoritem Oponašanja volkov smo sprva implementirali s prvim načinom. Ob določenih parametrih je za primerek rl1889 potreboval 10 minut. Ko smo algoritem spisali z drugim načinom, je na primerku rl1889 ob enakih parametrih potreboval 50 sekund.

Četrti razlog so parametri. Možno je, da obstajajo parametri, ki skrajšajo čas algoritma, pri čemer ostanejo rešitve podobne, ali pa algoritem pridobi boljše rešitve ob podobnem času. Analiza parametrov je zahtevna, saj moramo imeti poleg znanja, kako določena sprememba parametrov vpliva na algoritem, tudi čas, da lahko hipoteze o parametrih potrdimo skozi testiranje algoritma na več različnih primerkih. V naši diplomski nalogi smo parametre poskusili optimizirati, vendar ne moramo potrditi, da so bili uporabljeni parametri najboljši. Poleg tega lahko izpostavimo tudi idejo, da bi lahko določeni parametri slabo delovali na enem primerku, na drugem pa zaradi skritih lastnosti primerkov bolje.

Peti razlog so mere, uporabljene za primerjavo algoritmov. Da smo zaključili, kateri algoritem je najboljši, smo uporabili smo le vsoto oddaljenosti in grafični prikaz časa. Naš zaključek iz izbranih mer ni nepravilen, je pa pomanjkljiv. Lahko bi rekli, da ima vsak algoritem končno oceno. Končna ocena algoritma je sestavljena iz manjših ocen, pridobljenih iz rezultatov algoritma na posameznih primerkih, ki jih tudi primerno otežimo. Ocena posameznega primerka bi lahko bila sestavljena iz več podatkov, ne le iz oddaljenosti in časa. Uporabili bi lahko najboljšo in najslabšo rešitev, standardni odklon cen rešitev, kontrolni čas, uporabo pomnilnika ipd.

Vsi navedeni razlogi privedejo do spoznanja, da je primerjava algoritmov zelo težka in kompleksna zadeva. Lahko opravimo osnovno primerjavo, kot smo jo v tej nalogi, a težko rečemo, da je določen algoritem zmeraj boljši od drugega.

Poglavje 11

Zaključek

Opisali smo problem trgovskega potnika in hevristične algoritme, ki smo jih uporabili za njegovo reševanje.

Posamezne algoritme smo testirali na vseh izbranih primerkih, razen nekaterih izjem, ki smo jih izpustili zaradi njihove časovne zahtevnosti.

Rezultate smo primerjali in prišli do zaključka, da je algoritem Optimizacije z roji delcev z Lin-Kernighanovim algoritmom najboljši med izbranimi algoritmi. Kljub temu je težko potrditi, da je izbran algoritem resnično najboljši, saj ima naša metodologija primerjanja algoritmov določene pomanjkljivosti, kot so skrite lastnosti primerkov, podatkovna struktura rešitev problema trgovskega potnika, implementacije posameznih algoritmov, optimizacija parametrov ter uporabljene mere za primerjanje rezultatov algoritmov.

Kljub negotovemu zaključku naloga še vedno predstavlja dober pregled nad problemom trgovskega potnika, izbranimi algoritmi in določenimi težavami, na katere naletimo, kadar poskusimo primerjati algoritme.

Literatura

[1] *Candidate Sets*, pages 64–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[2] *Construction Heuristics*, pages 73–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[3] *Improving Solutions*, pages 100–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[4] *Ant Colony Algorithms*, pages 123–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[5] Emile Aarts, Jan Korst, and Wil Michiels. *Simulated Annealing*, pages 187–210. Springer US, Boston, MA, 2005.

[6] David L. Applegate, Robert E. Bixby, Vašek Chvatál, and William J. Cook. *Applications*, pages 59–80. Princeton University Press, 2006.

[7] David L. Applegate, Robert E. Bixby, Vašek Chvatál, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[8] Walid Ben-Ameur. Computing the initial temperature of simulated annealing. *Computational Optimization and Applications*, 29(3):369–385, 2004.

[9] Ruyi Dong, Shengsheng Wang, Guangyao Wang, and Xinying Wang. Hybrid optimization algorithm based on wolf pack search and local search for solving traveling salesman problem. *Journal of Shanghai Jiaotong University (Science)*, 24(1):41–47, 2019.

[10] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *Trans. Sys. Man Cyber. Part B*, 26(1):29—41, February 1996.

[12] Jose B. Escario, Juan F. Jimenez, and Jose M. Giron-Sierra. Ant colony extended: Experiments on the travelling salesman problem. *Expert Systems with Applications*, 42(1):390–410, 2015.

[13] M.L. Fredman, D.S. Johnson, L.A. Mcgeoch, and G. Ostheimer. Data structures for traveling salesmen. *Journal of Algorithms*, 18(3):432–479, 1995.

[14] Xiutang Geng, Zhihua Chen, Wei Yang, Deqian Shi, and Kai Zhao. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. *Applied Soft Computing*, 11(4):3680–3689, 2011.

[15] Elizabeth F. Gouvêa Goldbarg, Givanaldo R. de Souza, and Marco César Goldbarg. Particle swarm for the traveling salesman problem. In Jens Gottlieb and Günther R. Raidl, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 99–110, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[17] Roy Jonker and Ton Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163, 1983.

[18] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[19] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21(2):498—516, April 1973.

[20] Rajesh Matai, Surya Singh, and Murari Lal Mittal. Traveling salesman problem: an overview of applications, formulations, and solution approaches. In Donald Davendra, editor, *Traveling Salesman Problem*, chapter 1. IntechOpen, Rijeka, 2010.

[21] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, 2014.

[22] Denis Naddef. *Polyhedral Theory and Branch-and-Cut Algorithms for the Symmetric TSP*, pages 29–116. Springer US, Boston, MA, 2007.

[23] Alexander G. Nikolaev and Sheldon H. Jacobson. *Simulated Annealing*, pages 1–39. Springer US, Boston, MA, 2010.

[24] TSPLIB. Dosegljivo: `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html`, 1997. [Dostopano: 2. 9. 2020].

[25] Programska koda uporabljena v tej nalogi. Dosegljivo: `https://github.com/TLOP/Heuristics-for-TSP`, 2020. [Dostopano: 2. 9. 2020].

[26] Rezultati algoritmov ter uporabljeni grafi. Dosegljivo: `https://docs.google.com/spreadsheets/d/1VE341EqUxXPwC71_nEKmYO4VaoZnHrwTHhg2bnGEP2A/edit?usp=sharing`, 2020. [Dostopano: 2. 9. 2020].

[27] Abraham P. Punnen. *The Traveling Salesman Problem: Applications, Formulations and Variations*, pages 1–28. Springer US, Boston, MA, 2007.

[28] César Rego and Fred Glover. *Local Search and Metaheuristics*, pages 309–368. Springer US, Boston, MA, 2007.

[29] C. Yang, X. Tu, and J. Chen. Algorithm of marriage in honey bees optimization based on the wolf pack search. In *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, pages 462–467, Oct 2007.