Algoritam inteligencije rojeva za problem usmjeravanje vozila

Kvesić, Božo

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:742945

Rights / Prava: Attribution-NonCommercial-NoDerivs 3.0 Unported/Imenovanje-Nekomercijalno-Bez prerada 3.0

Download date / Datum preuzimanja: **2024-07-27**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Božo Kvesić

Algoritam inteligencije rojeva za problem usmjeravanje vozila

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Božo Kvesić

Matični broj: 45866/17-R

Studij: ***Informacijsko i programsko inženjerstvo***

JMBAG: 0016129614

Algoritam inteligencije rojeva za problem usmjeravanje vozila

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Nikola Ivković

Varaždin, prosinac 2023.

Božo Kvesić

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U ovom radu je implementiran algoritam računalne inteligencije koji rješava problem usmjeravanja vozila (eng. *Vehicle routing problem*, VRP). Algoritam je implementiran u jeziku C++, u okruženju Microsoft Visual Studio 2019. Korišten je mravlji algoritam, preciznije MMAS algoritam na kojemu su kasnije provođeni eksperimenti na primjercima problema A-n32-k5, A-n55-k9, A-n63-k9, A-n80-k10, B-n35-k5, B-n45-k5, B-n50-k7, B-n78-k10, E-n101-k14, E-n22-k4, E-n30-k3, E-n33-k4, E-n51-k5, F-n45-k4, F-n72-k4, M-n101-k10, P-n16-k8 i P-n40-k5. Cilj istraživanja je bio postići napredak algoritma kroz vrijeme, odnosno njegovo učenje i konstrukciju optimalnog rješenja. Problem usmjeravanja vozila je kombinatorni problem koji odgovara na pitanje: „Koji je optimalni skup ruta za flotu vozila kako bi se nešto prikupilo ili isporučilo na određene lokacije?“. Radi se o generalizaciji dobro poznatog problema trgovačkog putnika (TSP). VRP se često izravno primjenjuje u industriji, a sektor prijevoza čini osjetan udio BDP-a primjerice u EU. Cilj rada je rješavanje NP-teškog problema koji standardnim matematičkim metodama nije rješiv u razumnom vremenu, a korištenjem mravljeg algoritma može se dobiti približno optimalno rješenje koje će biti prihvatljivo. Prvi put se VRP pojavio u radu Georgea Dantziga i Johna Ramsera 1959. godine u kojem je opisan prvi algoritamski pristup koji je primijenjen na problem isporuke benzina. Godine 1964.,Clarke i Wright poboljšali su Dantzigov i Ramserov pristup koristeći pohlepnog algoritma nazvanog algoritam štednje. U problemu usmjeravanja vozila (VRP), dan nam je skup kupaca s poznatim zahtjevima, skup vozila i početna točka kretanja. Problem je osmisliti najjeftinije rute za vozila koje prolaze i završavaju u početnoj točki kretanja kako bi uslužile korisnike ovisno o ograničenjima kapaciteta vozila. Cilj je zadovoljiti svakog kupca, odnosno posjetiti ga jedanput i ispuniti njegove zahtjeve, a pri tome koristiti više vozila kako bi se troškovi puta smanjili. Nakon provedenih eksperimenata došao sam do zaključka kako MMAS algoritam pronalazi dobra rješenja za veliki broj instanci problema. Proveli smo 23 različitih eksperimenata čiji su rezultati unutar 10% razlike od optimalno prikazanih, dok smo za 1 primjer dobili bolje rješenje od znanstveno predstavljenog te smo za 9 eksperimenta dobili optimalno rješenje. Lokalna optimizacija često uvelike doprinosi poboljšavanju rješenja, ali algoritam bez lokalne optimizacije ne daje nužno uvijek najgore rezultate. Lokalna optimizacija je veće instance problema uvelike usporila, ali je bila nužna strategija za korištenje uz listu favorita.

Ključne riječi: programiranje, algoritam, inteligencija rojeva, problem usmjeravanja vozila

Sadržaj

Zahvale

Želim zahvaliti Elvisu Popoviću za pomoć oko pronalaska optimalnih ulaznih parametara algoritma u alatu iRace.

1. Uvod

Algoritam računalne inteligencije predstavlja skup pravila i postupaka koji se koriste za rješavanje problema koji su nemogući za rješavanje konvencionalnim metodama. U ovom radu objasnit ćemo detaljnije karakteristike algoritama računalne inteligencije te ćemo implementirati jedan takav za skup problema usmjeravanja vozila (eng. *Vehicle routing problem*, VRP). S obzirom na to da postoje različiti tipovi algoritama računalne inteligencije, radit ćemo na implementaciji mravljeg algoritma (eng. *ant colony algorithm*) na kojemu ćemo kasnije provoditi eksperimente na različitim primjercima problema usmjeravanja vozila te prikazati i analizirati rezultate. Glavna karakteristika toga algoritma je ta da u početku postoji jednaka vjerojatnost za odabir zadovoljavanja zahtjeva klijenata, a onda kasnije, nakon određenog broja iteracija vjerojatnost se povećava za one čvorove koji su korišteni pri konstrukciji najboljeg rješenja, odnosno smanjuje za ostale ako zahtjevi na toj relaciji nisu isplativi. S obzirom na to da imamo veći broj iteracija, a nakon svake će se nagrađivati određeni putevi povećavanjem koeficijenata dok će se svi ostali koeficijenti smanjivati, mravi koji konstruiraju rješenje će početi odabirati put koji je optimalan. Kasnije u radu ćemo pokazati razne grafove koji će opisivati krivulju učenja za različite instance problema te ćemo analizirati dobivene rezultate. Mravlji algoritam ćemo implementirati kao aplikaciju za konzolu s raznim parametrima koje ćemo mijenjati ovisno o instanci problema. Također, u svrhu poboljšanja ulaznih parametara, koristit će se program iRace kojemu je glavna upotreba automatska konfiguracija algoritma za optimizaciju i odlučivanje, odnosno pronalaženje najprikladnijih postavki za algoritam danih na skup instanci problema.

Primarni cilj ovog rada je razviti mravlji algoritam unutar konzolne aplikacije koja je napisana u C++ jeziku, te provesti analizu rezultata koje dobijemo. Cilj je također analizirati dobivena rješenja za postavljene početne parametre poput utjecaja ograničavanja jačine feromona, jačine isparavanja feromona, broja mravi, broja iteracija, broja vozila, lokalnih i globalnih parametara za optimizaciju nad kreiranim rješenjima te još mnogih drugih parametara.

Aplikacija je sposobna pokrenuti mravlji algoritam s određenim brojem iteracija i određenim brojem mrava u svakoj iteraciji, uz različite druge parametre koje ćemo detaljno objasniti. Svaki pojedini mrav konstruira vlastito rješenje koje se potom

evaluira te se vrše provjere u odnosu na ostala rješenja u iteraciji, odnosno na najbolje globalno rješenje. S obzirom na to da se koristi MMAS algoritam, gornja i donja granica feromonskih tragova se određuju na osnovu veličine učitanog problema te su granice dinamički promjenjive ovisno o unesenim parametrima i problemu kojeg se rješava. Aplikacija će biti implementirana korištenjem objektno-orijentiranog načina uz korištenje headera te je zahtjevna u slučaju većeg broja klijenata (čvorova). Uzorci dizajna će također biti implementirani kako bi nastavak rada na aplikaciji bio znatno olakšan. Algoritam od uzoraka dizajna implementira *strategy* i *observer* koji služe za bolji objektno orijentirani pristup samoj implementaciji.

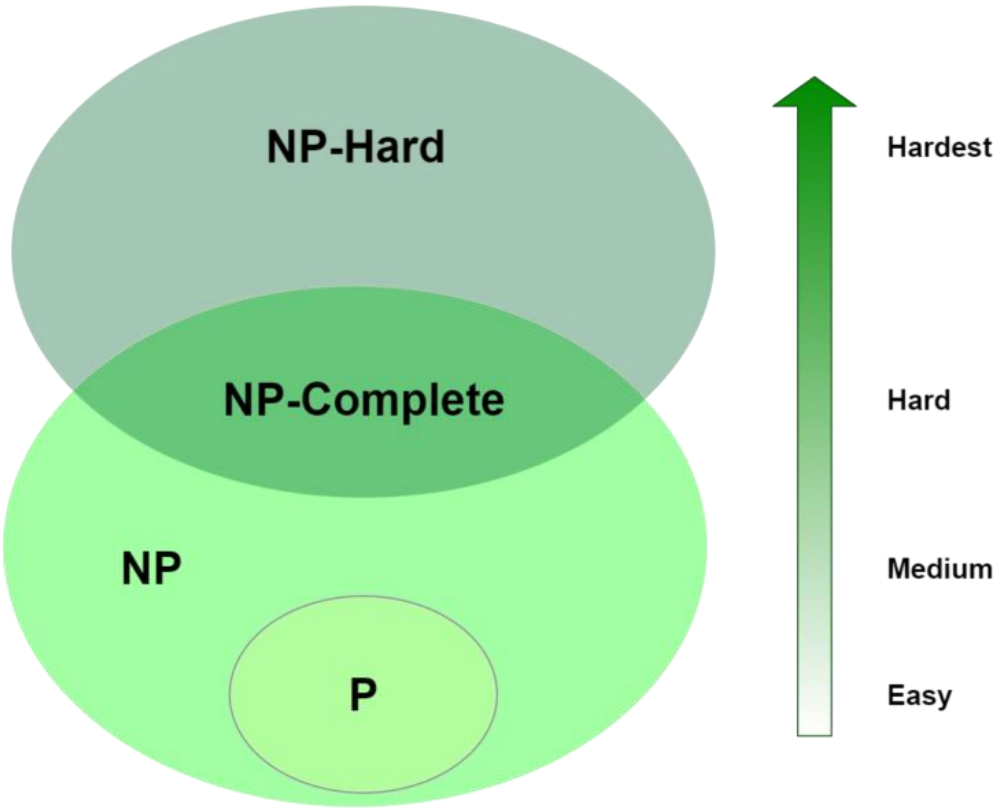
Eksperimenti koji se provode istražuju različite verzije istog problema koje imaju različite početne uvjete. Glavna funkcija cilja je minimizacija troška udaljenosti pa će tako učinkovitost pojedinog mrava biti mjerena po ukupnoj sumi svih udaljenosti koje je pojedino vozilo prešlo računanoj po matrici udaljenosti definiranoj implicitno ili eksplicitno u samoj datoteci problema. Uz to, koristit ćemo različite metode lokalne optimizacije koje su posebno dizajnirane za problem usmjeravanja vozila. Također ćemo provesti dodatnu analizu različitih postavki za lokalne optimizacije koje su predviđene za svako pojedino rješenje, najbolje rješenje u svakoj iteraciji i najbolje globalno rješenje.

Ovaj rad se sastoji od šest poglavlja. Prvo poglavlje donosi opis klase NP-problema, dok drugo poglavlje pruža detalje o problemu usmjeravanja vozila. Treće poglavlje se bavi opisom algoritama za rješavanje NP-problema. Četvrto poglavlje se fokusira na mravlji algoritam i pruža detaljne informacije o njemu. Peto poglavlje uključuje analizu koda i ističe ključne aspekte koji ovaj algoritam čine pravim mravljim algoritmom. Kao završno poglavlje, rad donosi analizu provedenih eksperimenata i zaključak.

Na kraju ovog rada pokušat ćemo odgovoriti na sljedeća pitanja: dobivamo li poboljšanje korištenjem lokalne optimizacije za dobivanje optimalnog rješenja? Koliko je programsko rješenje dobro u usporedbi s razvijenim znanstvenim algoritmima za ovaj problem korištenjem manjeg i većeg broja klijenata? Mijenjanjem parametara dobivamo li na kvaliteti programskog rješenja?

2. NP problemi

Algoritmi koriste računalne resurse za svoje izvršavanje pa se tako algoritamska složenost karakterizira kao vremenska ili prostorna složenost u ovisnosti o veličini ulaznog problema. Vremenska složenost algoritma daje red veličine broja elementarnih instrukcija koje se izvode u vremenu izvođenja samog programa. Koristi se za usporedbu različitih algoritama neovisno o karakteristikama samog računala ili programskog jezika. Također, vremenska složenost obično ovisi o veličini ulaznih podataka. S obzirom na to da je teško izračunati točan broj instrukcija, vremenska složenost nam daje red veličine. Cilj vremenske složenosti je dati informaciju o skalabilnosti samog algoritma. U tu svrhu koristimo O notaciju te je vremenska složenost algoritma $O(f(n))$. Složenost određenog problema ocjenjuje se s obzirom na složenost najboljeg algoritma za rješavanje tog problema. Shodno tome, najbolji algoritam je onaj čija je složenost najmanja. Za probleme odlučivanja uvedene su klase složenosti. Tako postoje klase P i NP. Klasa P sadrži sve probleme koji se mogu riješiti u polinomijalnom vremenu s pomoću Turingovog stroja, koji se može promatrati kao teorijski računalni model. Pojednostavljeno to bi značilo da se svaki problem ove klase može riješiti s pomoću algoritma čija je vremenska složenost manja ili jednaka $O(n^k)$ gdje je n veličina ulaznih podataka, a k konstanta neovisna o ulaznim podacima. Tako u toj klasi problema imamo primjere poput traženja vrijednosti u nizu te odgovor na pitanja je li zadani cijeli broj prost. Klasa NP sadrži sve probleme koji se mogu riješiti u polinomijalnom vremenu na nedeterminističkom Turingovom stroju i on ima mogućnost prijelaza u više stanja. [1] Osim toga, radi se o skupu problema za koje možemo provjeriti rješenje u polinomijalnom vremenu. Algoritmi klase NP su algortimi koji se koriste za rješavanje problema koji bi inače bili teški ili nemogući za rješavanje pomoću konvencionalnih metoda. Primjeri algoritama klase NP uključuju *brute force* algoritme, podijeli i vladaj algoritme, dinamičke algoritme, pohlepne (*greedy*) algoritme, te algoritme za sortiranje i pobrojavanje. Neki su znanstvenici smatrali da će za većina algoritama klase NP s vremenom i napretkom znanosti biti svedene na klasu P [2], ali nakon pola stiljeća istraživanja to se nije dogodilo.



Slika 1: Eulerov dijagram za klasu algoritama P, NP, NP-potpune i NP-teške skupove problema (Izvor: https://www.baeldung.com/cs/p-np-np-complete-np-hard)

Na slici 1 vidimo Eulerov dijagram koji prikazuje odnos između različitih klasa problema u teoriji računalne složenosti koji uključuje klasu P, klasu NP, klasu NP-potpunu i klasu NP-tešku. Iz grafa možemo vidjeti kako klasa P, odnosno klasa problema koja se može riješiti u polinomijalnom vremenu, nalazi unutar kruga za NP, odnosno problema za koje se rješenje može provjeriti u polinomijalnom vremenu. NP-teški skupovi problema su izvan kruga NP jer osim što je teško pronaći njihova rješenja, također ih je teško provjeriti. NP-teški problemi često se javljaju u stvarnim aplikacijama. Na primjer, u približnom računanju, cilj je zamijeniti točnost za računsku učinkovitost. Mnogi optimizacijski problemi u ovoj domeni, poput optimiziranja aproksimacijskih algoritama, mogu biti NP-teški. [3] To implicira da pronalaženje optimalnih rješenja može biti nepraktično i istraživači moraju osmisliti heurističke ili aproksimacijske metode kako bi dobili rezultate blizu optimalnih. U rudarenju podataka, NP-teški problemi često se javljaju u zadacima kao što su klasteriranje, odabir značajki i rudarenje pravila pridruživanja. Budući da rudarenje podataka uključuje pronalaženje obrazaca i odnosa u velikim skupovima podataka, uobičajeno je naići na probleme optimizacije koje je teško točno riješiti. Problemi s rutama, poput

pronalaženja najkraćeg puta ili optimizacije ruta za vozila, često su NP-teški. Ovi problemi nastaju u logistici, transportu i umrežavanju. Praktična rješenja uključuju aproksimacijske algoritme i heuristiku zbog složenosti scenarija iz stvarnog svijeta.

Zaključno, NP-teški problemi predstavljaju značajan izazov u teoriji složenosti računanja. Iako ih je teško točno riješiti, ključni su za mnoga područja računalne znanosti i matematike. Tekuća istraživanja ovih problema nastavljaju donositi fascinantne uvide u prirodu složenosti i računanja. Unatoč inherentnim poteškoćama, NP-teški problemi nisu nepremostivi. Upotrebom pametnih algoritama i heuristike često je moguće pronaći rješenja koja su „dovoljno dobra“ za praktične svrhe. Kako se razumijevanje ovih problema povećava, očekujemo kontinuirani napredak za rješavanje ovih problema ili dobivanje „dovoljno dobrih“ rješenja za praktične svrhe.

Povijest NP-problema

Dana 4. svibnja 1971. godine, znanstvenik Steve Cook predstavio je svijetu problem NP u svom radu „The Complexity of Theorem proving procedures“. Trenutno je taj problem i dalje aktualan te svijet još uvijek traga za rješenjem. Prije 13 godina je objavljen članak pod nazivom „The Status of P versus NP Problems“. U tom članku je predstavljena teorija klasa problema tako da je postavljeno pitanje kako upariti veliku grupu učenika u timove od dva kompatibilna člana. Pretraživanje svih mogućih parova za manji broj učenika predstavlja veliki broj mogućih uparivanja koje bi trebalo provjeriti. Godine 1965.. Jack Edmonsds dao je učinkovit algoritam za rješavanje ovog problema te je predložio definiciju „učinkovitog računanja“. Klasa problema s učinkovitim računanjem je kasnije poznata kao klasa P. Kasnije se uspostavilo kako povezani problemi nemaju tako učinkovit algoritam pa je tako problem ako želimo napraviti grupe od po tri učenika tako da svaki par u toj trojci bude kompatibilan? Što ako želimo posjesti učenike velikog okruglog stola pri čemu ne sjede nekompatibilni učenici jedni do drugih? Svim ovim problemima možemo učinkovito potvrditi rješenja te su takvi problemi kojima možemo učinkovito potvrditi rješenja, a teže doći do rješenja poznati kao klasa NP. Problem P vs. NP je ubrzo postao važno računalno pitanje u gotovo svakoj znanstvenoj disciplini. Stoga učinkovito rješavanje bilo kojeg NP problema bi značilo da je P=NP što nije dokazano, ali također nije dokazano ni P≠NP. [4]

3. Problem usmjeravanja vozila

Problem usmjeravanja vozila (eng. *vehicle routing problem*, VRP) je generički naziv za čitavu klasu problema koji se tiču optimalnog dizajna ruta koje će koristiti skup vozila ograničenog kapaciteta smještenih u središnjem depou za ispunjavanje zahtjeva skupa kupaca. Cilj problema je minimizirati ukupne troškove puta. Problem usmjeravanja započeo je 26. kolovoza 1735. godine kada je Leonhard Euler predstavio svoje rješenje problema Königsergberskog mosta gdje je zadan povezani graf $G = (N, E)$ te je potrebno pronaći zatvorenu turu koja posjećuje svaki rub u E točno jednom. Euler je dokazao da postoji ako i samo ako svaki čvor u G ima parni stupanj. Sljedeći problem usmjeravanja koji je trebalo proučavati je bio problem kineskog poštara koji ima zadan povezani graf $G = (N, E, C)$, gdje je C matrica udaljenosti te je potrebno pronaći obilazak koji prolazi svakim rubom barem jednom i to na najkraći mogući način. Problem je moguće riješiti u polinomnom vremenu kada je G potpuno usmjeren ili potpuno neusmjeren, ali kada je G mješoviti graf onda taj problem postaje NP-težak. [5] Problem putujućeg lopova je kombinacija trgovačkog putnika (TSP) i problema ruksaka (KP). Ovaj problem se koristi za modeliranje i rješavanje stvarnih problema koji uključuju više komponenti koje su međusobno ovisne. Unatoč njegovoj složenosti, postoji rasprava među znanstvenicima o tome koliko je ovaj problem realističan, s obzirom na to da model omoguućuje jednom „lopovu“ da putuje kroz veliki broj gradova kako bi prikupio predmete. [6] Upravo problem usmjeravanja vozila se bavi tim pitanjem gdje imamo skup vozila koji će kreirati vlastiti graf, a kada uzmemo svako to vozilo i njegovu rutu stavimo na graf, on će kreirati skup svih kupaca. Problem usmjeravanja vozila je problem kojim se ponajviše bavi distribucija tisuća i tisuća tvrtki. Prvi objavljeni članak na temu problema usmjeravanja vozila je objavljen 1959. godine od strane Dantziga i Ramsera s naslovom „The truck dispatching problem“ gdje je predstavljeno nekoliko varijanti osnovnog problema. Tradicionalni problem usmjeravanja vozila smatra se jednim od najčešće proučavanih problema u području optimizacije. Istraživanjem ovog problema rezultiralo je razvojem brojnih preciznih i heurističkih metoda rješavanja koje se mogu primijeniti na širok spektar problema. VRP se često definira prema ograničenjima kapaciteta i duljine rute. Kada su prisutna samo ograničenja kapaciteta onda se problem označava kao CVRP. [7]

Karakteristike problema CVRP

Skup vozila treba posjetiti sve klijente tako da kapacitet vozila prilikom odabira klijenata ne bude prekoračen. Svaki klijent nosi svoju vrijednost te vozilo prilikom posjeta klijentu tu vrijednost stavlja kod sebe u vozilo te si tako smanjuje kapacitet. Cilj je da više vozila pažljivo odabere klijente kako bi se trošak puta minimizirao. Trošak puta je suma svih udaljenosti svakog vozila koje je prešao prilikom kreiranja svoje rute. Rješenje je bolje čim je trošak puta manji te nam je cilj minimizirati trošak puta. Cilj je prikazan formulom koja ima sljedeći oblik:

$$Trošak\_puta = Min(\sum_{k \in Vozila} \sum_{i \in Lista\_klijenata} Udaljenost(Matica[k,i], Matrica[k,i+1]))$$

Skup vozila kreće s iste lokacije na kojoj se nalazi skladište ili distribucijski centar. S obzirom na to da svako vozilo ima ograničen kapacitet tada ruta vozila može posjetiti samo kupce ili lokacije čiji zahtjevi ne premašuju njegove kapacitete. Optimizacija je usmjerena na određivanje potrebnih ruta i redoslijed posjeta kako bi se smanjila udaljenost putovanja prema formuli koja je prethodno napisana za trošak puta. [8]

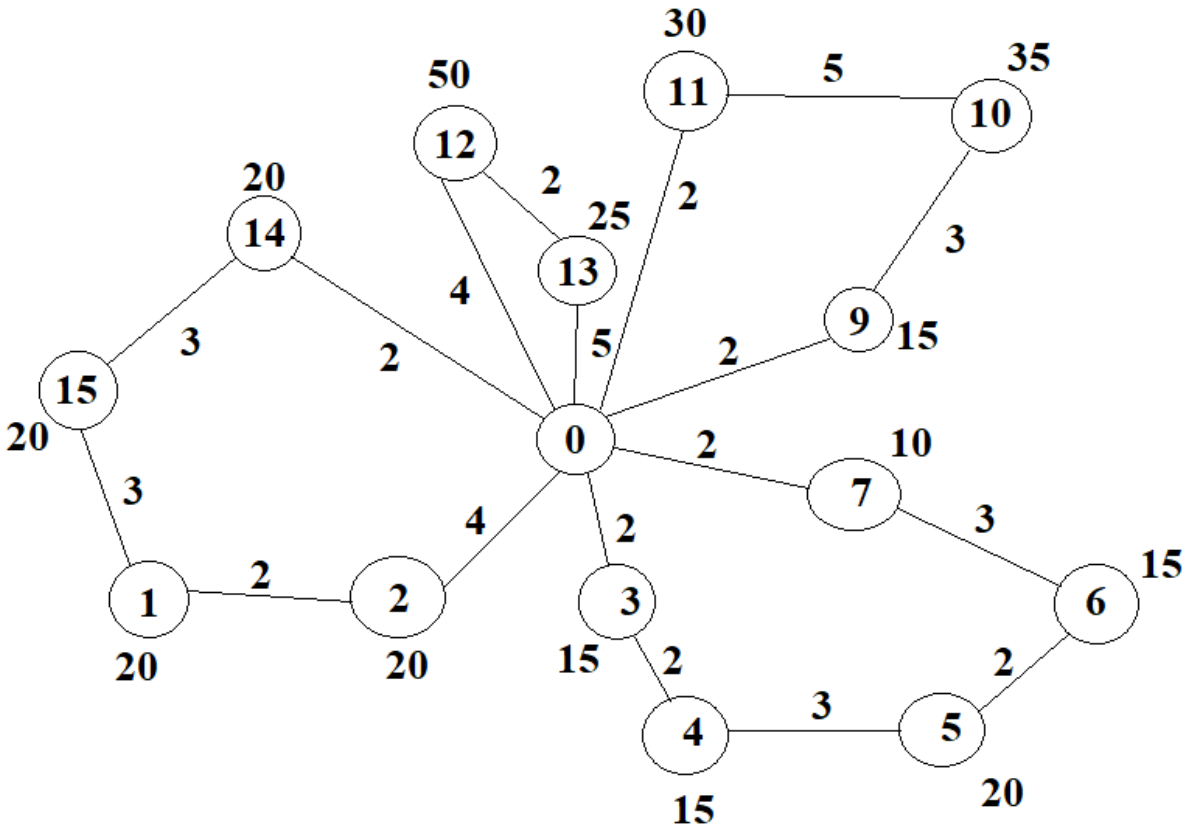
Svrha problema

Problem usmjeravanje vozila je jedan od najizazovnijih problema unutar logističkog polja zbog svoje složenosti. Tako naprimjer usluge dostave imaju za cilj optimizirati rute za skup vozila koja koriste za dostavu kako bi im trošak bio što manji, a zarada od prodaje što veća. CVRP se može koristit u planiranju prijevoza za optimizaciju autobusnih ruta, rasporeda školskih autobusa i drugih usluga javnog prijevoza. Svi oni se bave tim problemom nesvjesno. U suvremenom poslovanju, čak i manje tvrtke često se susreću s izazovima izuzetne kompleksnosti. Michalewicz je identificirao nekoliko razloga za nesklad između akademske i stvarne stvarnosti. Jedan od tih razloga je fokus akademskih eksperimenata na jednostavne referentne probleme, dok su stvarni poslovni problemi često složeni i višekomponentni. Kako bi se naglasila važnost optimizacije u stvarnom svijetu, uvedeni su višekomponentni problemi, ilustrirajući kompleksnosti koje proizlaze iz višestrukih interaktivnih komponenti. Rješavanjem problema usmjeravanja vozila temeljenog na realnom primjeru iz svijeta poslovanja

može smanjiti troškove logistike, odnosno dovesti do učinkovitog rješenja za planiranje rute i može dovesti do učinkovitije i isplativije logistike i operacija isporuke, što u konačnici može dovesti do povećanog zadovoljstava klijenata te održivog rasta poslovanja. [9]

Matematički zapis CVRP

Kod problema kapacitetnog usmjeravanja vozila zadan je usmjereni graf $G = (N, A)$ gdje je $N = \{0, \dots, n+1\}$ skup svih lokacija takvih da 0 predstavlja početnu lokaciju svih vozila, a $n+1$ posljednju lokaciju klijenta. $N' = \{1, \dots, n\}$ je skup svih klijenata te je $A = \{(i,j) : i,j \in N, i \neq j, i \neq n+1, j \neq 0\}$. Svaki klijent $i \in N'$ ima potražnju $q_i > 0$. Ovdje smo matematički postavili da imamo skup kupaca gdje svaki osim početnog ima određenu potražnju, a početna lokacija s koje kreću i vraćaju se vozila ima potražnju 0. Postavimo radi jednostavnosti da je $q_0 = q_{n+1} = 0$ što bi značilo da početno i završno odredište svakog vozila ima potražnju 0 kako smo ranije naveli. Na raspolaganju imamo neograničenu količinu vozila p kapaciteta Q koja je dostupna za zadovoljavanje potrebe klijenata. Vozila prelaze elementarnu statu od 0 do n+1 koja se naziva ruta kako bi zadovoljili potražnju svih kupaca na putu. Ukupna potražnja kupaca na jednoj ruti ne može premašiti kapacitet vozila, a potražnja se ne može podijelit što bi značilo da se svaki kupac posjećuje točno jednom. Pretpostavimo dalje da je $q_i \leq Q$ za sve $i \in N$. Trošak puta između dva klijenta $(i,j) \in A$ je $c_{ij} \geq 0$. Problem usmjeravanja vozila s kapacitetom je problem pronalaženja ruta koje zadovoljavaju sve zahtjeve klijenata dok su ukupni troškovi minimizirani. Ovo ovdje je osnovi problem usmjeravanja vozila koji ima ograničenje kapaciteta nad vozilima, ali također postoje razne varijante koje imaju dodatna ograničenja koja su matematički postavljena. Minimalan broj vozila potrebnih za ispunjenje potražnje svih klijenata iznosi $\left\lceil \frac{\sum_{i=1}^{n} q_i}{Q} \right\rceil$. [10] Na slici 2 možemo vidjeti rješenje CVRP problema gdje je kapacitet vozila 80. Prema formuli za ovaj problem imamo $\left\lceil \frac{310}{80} \right\rceil = \lceil 3{,}875 \rceil = 4$. Dobili smo rezultat 4 prema kojemu minimalan broj vozila za rješavanje ovog problema iznosi 4.



Slika 2: Primjer CVRP rješenja za odabir najmanjeg broja vozila (Izvor: vlastita izrada prema Borčinova, Z. 2017.)

U ovom problemu postoje matematički zapisi ograničenja koja se trebaju poštivati.

$$Min. \sum_{r=1}^{p} \sum_{i=0}^{n} \sum_{j=0, i \neq j}^{n} C_{ij}\, x_{rij} \qquad (1)$$

Ograničenje 1 pokazuje funkciju cilja za ukupni trošak svih vozila koji treba minimizirati te je binarna varijabla odluke x_{rij} definirana da pokaže da li vozilo r gdje je $r \in \{1,2, \dots, p\}$ prelazi luk (i, j) u optimalnom rješenju. [10]

$$\sum_{r=1}^{p} \sum_{i=0, i \neq j}^{n} x_{rij} = 1\,, \forall \mathrm{j} \in \{1, \dots, \mathrm{n}\} \qquad (2)$$

Ograničenje 2 osigurava da je stupanj svakog klijentovog čvora 2, odnosno da svakog klijenta posjećuje točno jedno vozilo. [10]

$$\sum_{j=1}^{n} x_{r0j} = 1\,, \forall \mathrm{r} \in \{1, \dots, \mathrm{p}\} \qquad (3)$$

$$\sum_{i=0,i\neq j}^{n} x_{rij} = \sum_{i=0}^{n} x_{rji}\,, \forall \mathrm{j} \in \{0, \dots, \mathrm{n}\}, \mathrm{r} \in \{1, \dots, \mathrm{p}\} \tag{4}$$

Ograničenja 3 i 4 osiguravaju da svako vozilo može samo jednom napustiti skladište, odnosno čvor 0, a broj vozila koja dolaze do svakog kupca i ulaze u skladište jednak je broju vozila koja izlaze. [10]

$$\sum_{i=0}^{n} \sum_{j=1,i\neq j}^{n} q_i\, x_{rij} \leq Q\,, \forall \mathrm{r} \in \{1, \dots, \mathrm{p}\} \tag{5}$$

Ograničenje 5 osigurava da zbor potražnje posjećenih klijenata na ruti manji ili jednak kapacitetu vozila koje pruža usluge, odnosno osigurava da vozilo ne pređe svoj kapacitet. [10]

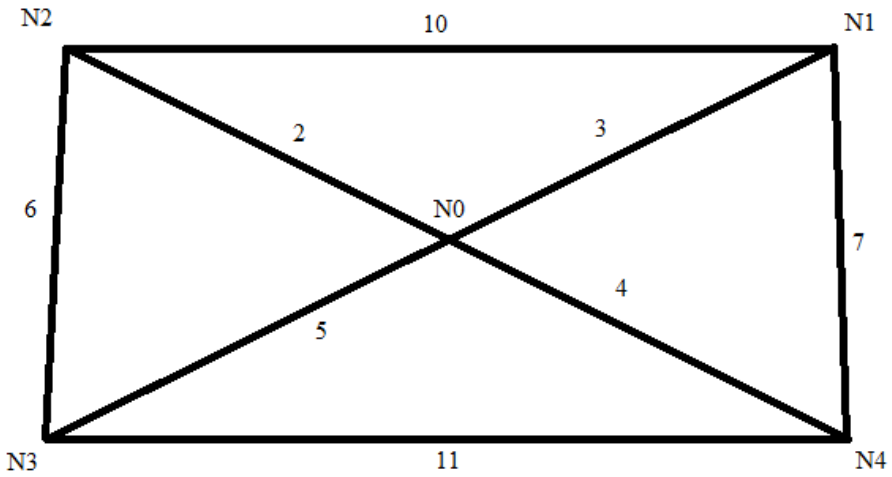
$$\sum_{r=1}^{p} \sum_{i\in S} \sum_{j\in S,i\neq j} x_{rij} \leq |S| - 1\,, \forall \mathrm{S} \in \{1, \dots, \mathrm{n}\} \tag{6}$$

Ograničenje 6 osigurava da rješenje S ne sadrži cikluse koji nisu povezani sa skladištem, odnosno vozilo mora krenuti iz skladišta i u njega se vratiti. [10]

$$x_{rij} \in \{0,1\}, \forall \mathrm{r} \in \{1, \dots, \mathrm{p}\}, \mathrm{i}, \mathrm{j} \in \{0, \dots, \mathrm{n}\}, \mathrm{i} \neq \mathrm{j} \tag{7}$$

Ograničenje 7 određuje definiciju domene varijabli, odnosno pokazuje preko varijable x_{rij} da li vozilo r prelazi luk (i,j) u optimalnom rješenju pa je vrijednost 1 u slučaju da prelazi, odnosno 0 ako ne prelazi. [10]

Primjer CVRP problema i rješenja



Slika 3: primjer CVRP-a (Izvor: vlastita izrada)

Na slici 2 prikazan je jednostavni primjer gdje N0 predstavlja početnu lokaciju svih vozila, a N1,N2,N3 i N4 su klijenti koje vozila trebaju posjetiti. Svakom klijentu, osim prvom (N0) dodijeljena je potražnja veća od nula što možemo vidjeti u tablici 1. Na raspolaganju imamo neograničeni broj vozila, a svakom vozilu je postavljeno ograničenje kapaciteta na 15. Kako bi ovaj primjer odgovarao primjerima problema usmjeravanja vozila, možemo vidjeti kako suma potražnje svih klijenata iznosi 23 što je veće od 15 te možemo zaključiti kako jedno vozilo neće biti dovoljno da obiđe sve klijente već je potrebno korištenje više od jednog vozila.

Tablica 1: Prikaz potražnje klijenata

Oznake klijenata	Potražnja klijenata
N0	0
N1	5
N2	3
N3	9
N4	6

(Izvor: vlastita izrada)

U tablici 2 prikazana je matrica udaljenosti između klijenata koja se temelji na slici 2 gdje je grafički prikazan odnos udaljenosti između pojedinih klijenata. Ovdje također možemo primijetiti kako se radi o simetričnoj matrici udaljenosti jer svi problemi koje ćemo rješavati razvijenim algoritmom će također imati simetričnu matricu udaljenosti.

Tablica 2: Matrica udaljenosti

	N0	N1	N2	N3	N4
N0	0	3	2	5	4
N1	3	0	10	8	7
N2	2	10	0	6	6
N3	5	8	6	0	11
N4	4	7	6	11	0

(Izvor: vlastita izrada)

Zamislimo da imamo poduzeće s jednim skladištem koje našem slučaju predstavlja vrh N0 i četiri kupca koja predstavljaju ostale vrhove. Cilj je pronaći najučinkovitiju rutu koja zadovoljava sve zahtjeve kupaca bez prekoračenja kapaciteta vozila. Vozilo kreće iz skladišta, isporučuje robu kupcima i mora se vratiti u skladište na kraju svoje rute.

U prvom primjeru koji dobivamo imamo sljedeće:

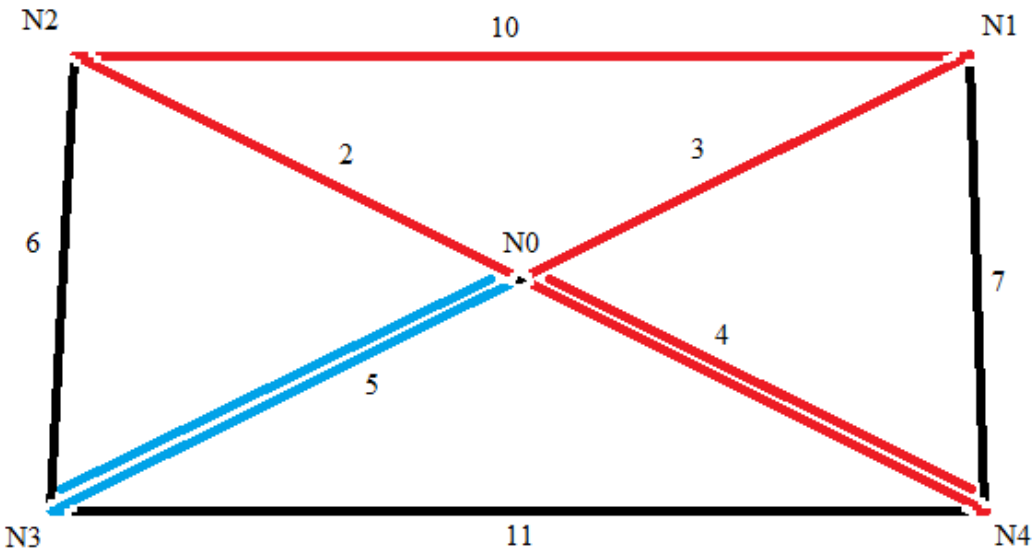
$$vozilo_1 = \{N0, N1, N2, N4, N0\}$$

$$vozilo_2 = \{N0, N3, N0\}$$

Ovdje vidimo dva vozila koja su kreirala svoju rutu i ako uzmemo prvo vozilo i izračunamo ukupnu potražnju vidimo da $vozilo_1 = 5 + 3 + 6 = 14$ što je manje od zadanog kapaciteta te je ruta prvog vozila u skladu sa zadanim ograničenjima. Ako uzmemo drugo vozilo i izračunamo ukupnu potražnju vidimo da $vozilo_2 = 9$ što je također manje od zadanog kapaciteta i također možemo zaključiti kako je u skladu sa zadanim ograničenjima. Na slici 3 vidimo kretanje tih vozila prikazano grafički. Crvenom bojom je prikazano kretanje vozila 1, a plavom bojom je prikazano kretanje vozila 2. Sada trebamo evaluirati rješenje, odnosno izračunati trošak puta.

$$Ukupni\_trošak\_puta = Trošak\_puta\_vozila_1 + Trošak\_puta\_vozila_2$$
$$Trošak\_puta\_vozila_1 = Udaljenost(N0, N1) + Udaljenost(N1, N2) + Udaljenost(N2, N4) + Udaljenost(N4, N0)$$
$$Trošak\_puta\_vozila_1 = 3 + 10 + 6 + 4 = 23$$
$$Trošak\_puta\_vozila_2 = Udaljenost(N0, N3) + Udaljenost(N3, N0)$$
$$Trošak\_puta\_vozila_1 = 5 + 5 = 10$$
$$Ukupni\_trošak\_puta = 23 + 10 = 33$$

Ukupni trošak ovog rješenja iznosi 33.



Slika 4: Kretanje vozila u prvom primjeru rješenja (Izvor: vlastita izrada)

U drugom primjeru koji dobivamo imamo sljedeće:

$$vozilo_1 = \{N0, N2, N3, N0\}$$

$$vozilo_2 = \{N0, N1, N4, N0\}$$

Ovdje vidimo dva vozila koja su kreirala svoju rutu i ako uzmemo prvo vozilo i izračunamo ukupnu potražnju vidimo da $vozilo_1 = 3 + 9 = 12$ što je manje od zadanog kapaciteta te je ruta prvog vozila u skladu sa zadanim ograničenjima. Ako uzmemo drugo vozilo i izračunamo ukupnu potražnju vidimo da $vozilo_2 = 5 + 6 = 11$ što je također manje od zadanog kapaciteta i također možemo zaključiti kako je u skladu sa zadanim ograničenjima. Na slici 4 vidimo kretanje tih vozila prikazano grafički. Crvenom bojom je prikazano kretanje vozila 1, a plavom bojom je prikazano kretanje vozila 2. Sada trebamo evaluirati rješenje, odnosno izračunati trošak puta.

$$Ukupni\_trošak\_puta = Trošak\_puta\_vozila_1 + Trošak\_puta\_vozila_2$$

$$Trošak\_puta\_vozila_1 = Udaljenost(N0, N2) + Udaljenost(N2, N3) + Udaljenost(N3, N0)$$

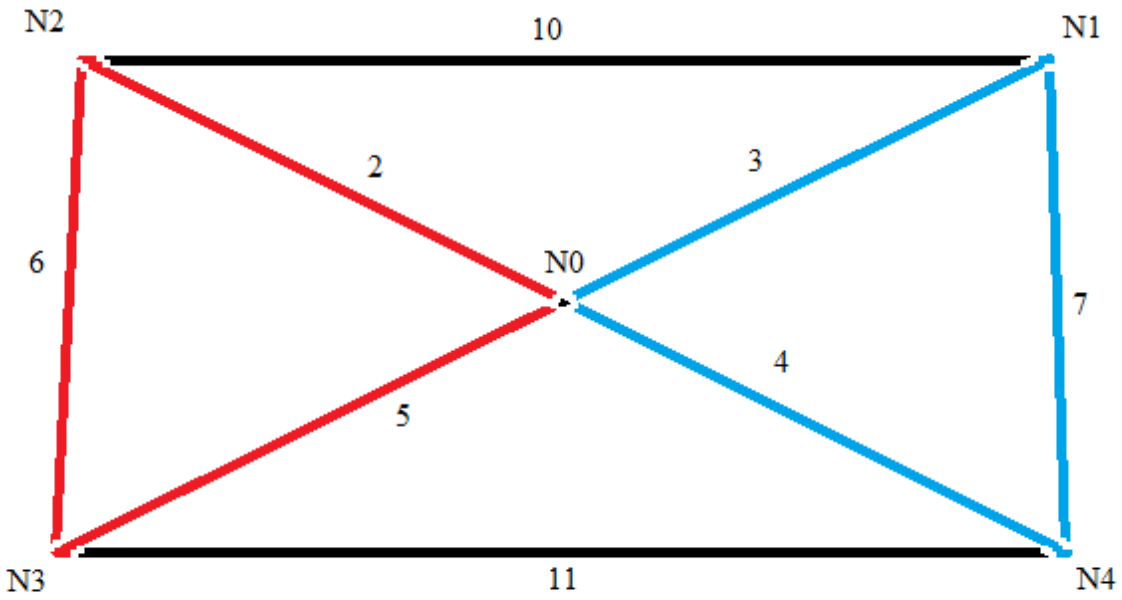
$$Trošak\_puta\_vozila_1 = 2 + 6 + 5 = 13$$

$$Trošak\_puta\_vozila_2 = Udaljenost(N0, N1) + Udaljenost(N1, N4) + Udaljenost(N4, N0)$$

$$Trošak\_puta\_vozila_1 = 3 + 7 + 4 = 14$$

$$Ukupni\_trošak\_puta = 13 + 14 = 27$$

Ukupni trošak ovog rješenja iznosi 27.



Slika 5: Kretanje vozila u drugom primjeru rješenja (Izvor: vlastita izrada)

U ovom primjeru smo vidjeli dva validna rješenja koja odgovaraju postavljenim ograničenjima, ali vidimo kako je trošak puta u drugom primjeru manji nego u prvom te u tom slučaju uzimamo rješenje primjera 2 kao optimalno.

Definiranje problem CVRP za rješavanje

Problem usmjeravanja vozila s ograničenim kapacitetima zahtjeva dobro definiranje koje je uskladu sa svim ograničenjima koje smo postavili kod rješavanja problema. Tako udaljenosti gradova ne smiju previše odskakati kako ne bi program ušao u stagnaciju te bi određene rute bile konstantno kreirane kod svakog rješenja.

```
TYPE : CVRP
DIMENSION : 22
EDGE_WEIGHT_TYPE : EUC_2D
CAPACITY : 6000
NODE_COORD_SECTION
1 145 215
2 151 264
3 159 261
4 130 254
5 128 252
6 163 247
7 146 246
8 161 242
9 142 239
10 163 236
11 148 232
12 128 231
13 156 217
14 129 214
15 146 208
16 164 208
17 141 206
18 147 193
19 164 193
20 129 189
21 155 185
22 139 182
DEMAND_SECTION
1 0
2 1100
3 700
4 800
5 1400
6 2100
7 400
8 800
9 100
10 500
11 600
12 1200
13 1300
14 1300
15 300
16 900
17 2100
18 1000
19 900
20 2500
21 1800
22 700
DEPOT_SECTION
1
```

Slika 6: Opis problema za 21 klijenta (Izvor: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/)

Na slici 6 vidimo primjer jednog definiranja problema kojemu su zadane koordinate čvorova odnosno klijenata, potražnja po čvorovima, kapacitet pojedinog vozila te koordinacije skladišta. Matricu udaljenosti je potrebno izračunati koristeći Euklidovu formulu za udaljenost.

```
NAME : cvrp-S-G-150-14
COMMENT : described in Salazar & Letchford (EJOR, 2015)
TYPE : ACVRP
DIMENSION : 16
EDGE_WEIGHT_TYPE : EXPLICIT
EDGE_WEIGHT_FORMAT : FULL_MATRIX
DISPLAY_DATA_TYPE : NO_DISPLAY
CAPACITY : 150
VEHICLES : 16
EDGE_WEIGHT_SECTION
0 229 175 174 135 162 181 179 104 289 195 168 156 213 201 193
229 0 144 88 268 198 409 50 329 468 354 143 137 296 429 78
175 144 0 59 281 58 336 114 249 461 224 8 201 153 352 179
174 88 59 0 255 111 349 57 264 447 269 56 154 209 367 123
135 268 281 255 0 286 223 229 191 202 322 274 137 348 244 199
162 198 58 111 286 0 300 162 214 449 168 57 232 99 314 222
181 409 336 349 223 300 0 360 88 221 208 330 320 294 22 369
179 50 114 57 229 162 360 0 279 428 309 111 109 260 379 67
104 329 249 264 191 214 88 279 0 266 151 243 256 219 104 296
289 468 461 447 202 449 221 428 266 0 415 454 337 480 230 400
195 354 224 269 322 168 208 309 151 415 0 221 338 105 211 354
168 143 8 56 274 57 330 111 243 454 221 0 196 154 346 174
156 137 201 154 137 232 320 109 256 337 338 196 0 321 341 63
213 296 153 209 348 99 294 260 219 480 105 154 321 0 302 318
201 429 352 367 244 314 22 379 104 230 211 346 341 302 0 389
193 78 179 123 199 222 369 67 296 400 354 174 63 318 389 0
DEMAND_SECTION
1 0
2 31
3 25
4 32
5 28
6 30
7 30
8 25
9 31
10 29
11 26
12 30
13 26
14 27
15 31
16 32
DEPOT_SECTION
1
EOF
```

Slika 7: Opis problema za 15 klijenata (Izvor: http://www.vrp-rep.org/variants.html)

Na slici 7 vidimo drugi tip zadavanja problema. Jedina razlika je što su ovdje udaljenosti zadane eksplicitno te nije potrebno računati udaljenosti jer su one unaprijed zadane. Osim toga napisane su potražnje pojedinih klijenta, kapacitet vozila kao što su zadane na slici 6. Također ovdje je potrebno staviti dobar omjer potražnje i postavljanja kapaciteta na pojedino vozilo kako ne bi došlo do toga da jedno vozilo može samostalno kreirati rješenje jer mu je kapacitet veći od sume svih potražnji klijenata te za takav primjer možemo reći kako nije dobro definiran problem. Osim toga matrica udaljenosti bi trebala biti simetrična što je na slici 7 vidljivo i udaljenosti također

moramo biti dobro definirane kako se ne bi dogodilo da određena udaljenost između klijenata previše odskače od preostalih te će tako luk s prevelikim troškom biti zanemaren za svako kreirano rješenje. Za rješavanjem ovog problema koristit ćemo unaprijed definirane problema i analizirati dobivene rezultate.

4. Algoritmi za rješavanje NP problema

Povijest razvoja algoritama koji samostalno uče seže daleko u povijest pa tako imamo Aristotela u staroj Grčkoj koji je vjerojatno prva osoba koja je krenula prema konceptu umjetne inteligencije. Njegov je cilj bio objasniti stilove deduktivnog zaključivanja koje je kasnije nazvao silogizam. U 1854. godini George Boole razvija temelj propozicijske logike. Tek je kasnije u 1950-im godinama Alan Turing dao prvu definiciju umjetne inteligencije. Turing je proučavao kako se strojevi mogu koristiti za oponašanje procesa ljudskog mozga. Sam naziv umjetna inteligencija nastao je na konferenciji u Dartmouthu koju je organizirao John MacCarthy 1956. godine. Danas se John Maccarthy smatra ocem umjetne inteligencije. Razvoj umjetne inteligencije započeo je s genetskim algoritmima 1950-ih s radom Frasera, Bremermanna i Reeda [11]. Praksa nas podučava da kod rješavanja NP problema često nije nužno riješiti probleme s potpunom preciznošću već je dovoljno pronaći dovoljno dobro rješenje, odnosno približno rješenje. U tu svrhu razvijeni su heuristički algoritmi koji su nastali eksperimentiranjem kako bi se dobila zadovoljavajuća rješenja. Heuristički algoritmi su vrsta algoritama koji se primjenjuju na probleme koji su izazovni ili čak nemogući za rješavanje standardnim metodama. Iako ne nude uvijek najbolje rješenje, oni su sposobni pružiti zadovoljavajuće rješenje u određenom vremenskom okviru. Ovi algoritmi su korisni za generiranje aproksimativnih rješenja kada klasične metode ne daju rezultate ili su prespore. Karakterizira ih relativno niska računalna složenost, obično polinomna, ali ne pružaju garanciju za pronalaženje optimalnog rješenja. Kvaliteta, preciznost i potpunost rješenja često su žrtvovane zbog brzine. Heuristički algoritmi su posebno efikasni za rješavanje problema s eksponencijalnom i faktorijelnom složenošću [12]. Ovisno o problemu, heuristički algoritmi pokazuju različite rezultate pa tako za pojedine probleme pokazuju relativno loše rezultate. Heuristički algoritmi se razliku po svojim funkcijama cilja koji se razliku ovisno o situacijama u kojima se koriste. Heurističke metode moraju biti zasnovane na

jednostavnim i lako razumljivim pravilima. Svi koraci metode moraju biti u skladu s pravilima kojima je metaheuristika definirana. Uz sve to potrebno je precizno definirati metaheurističke metode matematičkim terminima. Za svaki problem potrebno je osigurati optimalno rješenje ili dovoljno dobro u razumnom vremenu. Osim toga heuristika mora davati kvalitetne rezultate za širok raspon problema iste klase i metode moraju biti jasno definirane kako bi se lako implementirale. Heuristika je pravilo koje se temelji na prethodnom iskustvu i koristi se za olakšavanje pronalaženja rješenja za određeni problem. Heuristički algoritmi, koji se zasnivaju na ovim pravilima, često se koriste za rješavanje optimizacijskih problema za koje ne postoje poznati algoritmi s polinomskom složenošću. Ključna prednost heurističkih algoritama leži u njihovoj sposobnosti da smanje prostor pretrage kroz primjenu iskstvenih uvida, što značajno ubrzava proces pronalaženja rješenja. Ovi algoritmi su posebno korisni za rješavanje problema s eksponencijalnom i faktorijelnom složenošću. Algoritmi se mogu podijeliti na egzaktne i heurističke algoritme. Egzaktni uvijek pružaju optimalno rješenje, dok heuristički ne moraju uvijek pruužiti optimalno rješenje. Postoje različite klasifikacije heurističkih algoritama, a jedna od podjela je sljedeća:

- Stohastički algoritmi
- Evolucijski algoritmi
- Fizički algoritmi
- Probabilistički algoritmi
- Algoritmi roja
- Imunološki algoritmi
- Neuronski algoritmi

Stohastički algoritmi su algoritmi koju pružaju razne različite strategije s pomoću kojih se mogu generirati „bolje“ i raznolike početne točke i dati tehniku pretraživanja susjedstva za usavršavanjem, proces koji se ponavlja s potencijalnom poboljšanim ili neistraženim područjima za pretraživanje [13].

Evolucijski algoritmi su algoritmi koju su inspirirani procesom i mehanizmima biološke evolucije. Darwin je predložio proces evolucije s pomoću prirodne selekcije (podrijetlo s modifikacijom) kako bi objasnio raznolikost života i njegovu prikladnost (adaptivno prilagođavanje) okolišu. Mehanizmi evolucije opisuju kako se evolucija

zapravo odvija kroz modifikaciju i razmnožavanje genetskog materijala (proteina). Evolucijski algoritmi bave se istraživanjem računalnih sustava koji nalikuju pojednostavljenim verzijama procesa i mehanizama evolucije prema postizanju učinaka tih procesa i mehanizama [13].

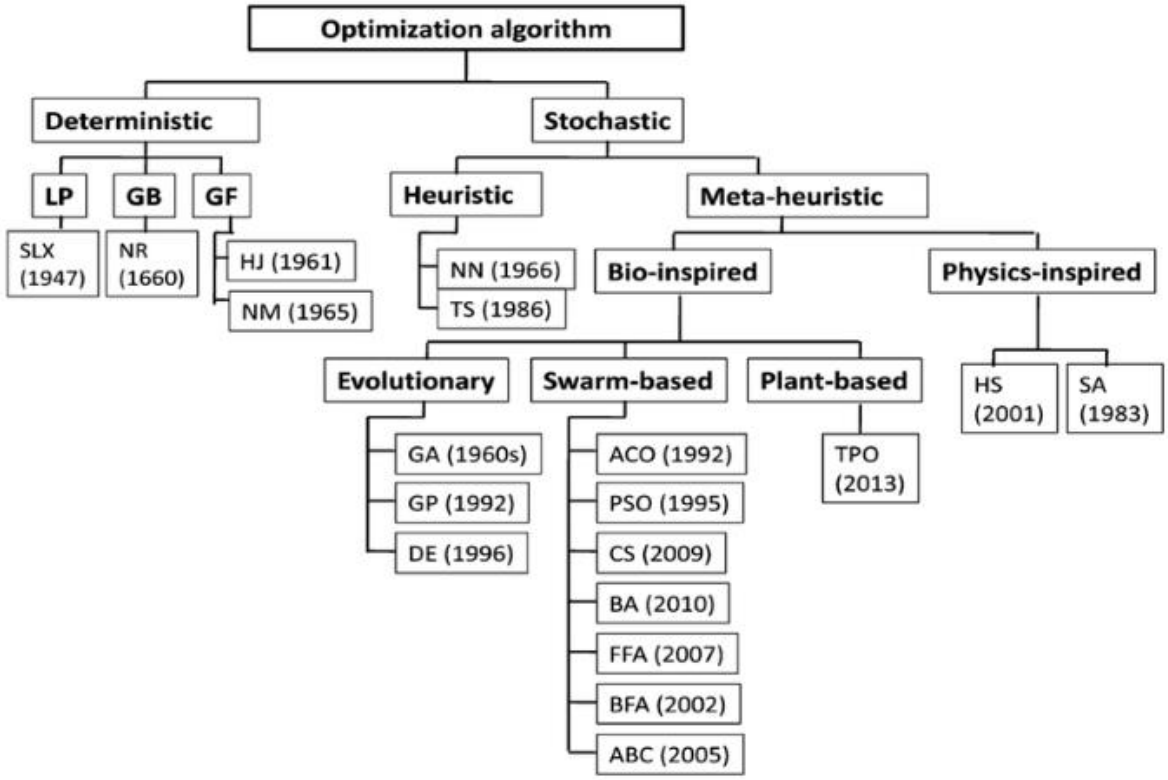
Fizički algoritmi su algoritmi inspirirani fizičkim procesom te općenito pripadaju poljima metaheuristike i računalne inteligencije. To su algoritmi inspirirani prirodom koji se svode na algoritme stohastičke optimizacije s mješavinom lokalnih i globalnih tehnika optimizacije [13].

Probabilistički algoritmi su algoritmi koji modeliraju problem ili pretražuju problemski prostor koristeći probabilistički model mogućih rješenja. Većina algoritama ovog skupa se još naziva i algoritmima procjene distribucije [13].

Algoritmi roja su skupina algoritama koja je inspirirana „kolektivnom inteligencijom“. Kolektivna inteligencija nastaje kroz suradnju velikog broja homogenih agenata u okruženju. Primjeri uključuju jata riba, jata ptica i kolonije mrava. Takva inteligencija je decentralizirana, samoorganizirajuća i distribuirana kroz okoliš. U prirodi se takvi sustavi obično koriste za rješavanje problema kao što je učinkovito traženje hrane, izbjegavanje plijena ili premještanje kolonije. Informacije se tipično pohranjuju među homogenim agensima koji sudjeluju ili se pohranjuju ili prenose u samom okolišu, poput upotrebe feromona kod mrava, plesa kod pčela i blizine kod riba i ptica. Primjer takvog algoritam je mravlji algoritam [13].

Imunološki algoritmi su algoritmi koji pripadaju području proučavanja umjetnog imunološkog sustava koje se bavi računalnim metodama inspiriranim procesima i mehanizmima biološkog imunološkog sustava [13].

Neuronski algoritmi su algoritmi koji se temelje na elementima obrade informacija inspirirane živčanim sustavom gdje postoji skup neurona koji su međusobno povezani u mreže i međusobno djeluju s pomoću elektrokemijskih signala [13].



LP	Linear programming	NM	Nelder–Mead	NN	Nearest neighbor	ACO	Ant colony optimization
GB	Gradient-based	HS	Harmony search	HJ	Hooke–Jeeves	ABC	Artificial bee colony
GF	Gradient free	CS	Cuckoo search	TS	Tabu search	TPO	Tree physiology optimization
SLX	Simplex algorithm	BA	Bat algorithm	GA	Genetic algorithm	PSO	Particle swarm optimization
NR	Newton–Raphson	FFA	Firefly algorithm	GP	Genetic programming	SA	Simulated annealing
				DE	Differential evolution	BFA	Bacterial foraging algorithm

Slika 8: Podjela heurističkih algoritama (Izvor: Dodig, M. and Smith, M. (2020))

Na slici 8 možemo vidjeti grafički prikaz podjele optimizacijskih algoritama s godinom nastanka pojedinog algoritma. S obzirom na to da ćemo se u radu baviti mravljim algoritmom, možemo vidjeti kako on pripada algoritmima roja te je predstavljen 1992. godine. Detaljnije ćemo obraditi mravlji algoritam u sljedećem poglavlju gdje ćemo predstaviti najvažnije karakteristike takvog algoritma.

5. Mravlji algoritam (ACO)

Mravlji algoritam (eng. *ant colony optimization*) je heuristički algoritam inspiriran ponašanjem kolonija mrava u traženju hrane. Prvi ga je predstavio Marco Dorigo 1992. godine u svojoj doktorskoj disertaciji. Algoritam se temelji na principu da mravi ostavljaju tragove feromona dok se kreću od svog gnijezda do izvora hrane. Mravi su životinje koje se kreću poluslijepo, bez korištenja osjetila vida. Kretanje ovih organizama je rezultat socijalne interakcije između pojedinaca. U prirodi mravi prilikom pronalaska izvora hrane ostavljaju za sobom feromonski trag. Taj feromonski trag omogućava drugim mravima da pronađu izvor hrane. Kada veći broj mrava koristi istu stazu, trag feromona na toj stazi postaje jači za ostale mrave. S druge strane, trag feromona može ispariti ako mravi ne koriste tu stazu često. Staze koje su najkraće i otkrivene u početku ovično imaju najviše feromona što privlači veći broj mrava [15]. U mravljem algoritmu, umjetni mravi se koriste za konstrukciju pojedinih rješenja koja se kasnije evaluiraju te se najboljem rješenju povećavaju feromoni kako bi mravi pri idućim konstrukcijama rješenja uzimali već provjerene bolje lukove. Algoritam nagrađuje najbolje povećavajući koeficijente odabira puta kojim je prošao za određeni delta_tau[1], dok istovremeno smanjuje vrijednosst feromona za određeni rho na svim ostalim putovima, simulirajući na taj način isparavanje feromona. U svakoj sljedećoj iteraciji, mravi odabiru svoj put uzimajući u obzir koeficijente feromona. Nakon određenog broja iteracija, koeficijenti feromona su također s pomoću delta_tau i rho izvršili određeni broj pojačavanja i isparavanja feromona te je veća vjerojatnost da će sljedeći mrav odabrati sličan put koji je u prethodnim iteracija pokazan kao najbolji. Nakon velikog broja iteracija kolonije mrava, algoritam dolazi do optimalnog rješenja. Budući da se radi o heurističkom algoritmu, ne možemo sa sigurnošću reći je li dobiveno rješenje optimalno, približno optimalno ili loše. Svaki problem zahtijeva jedinstven pristup mravljeg algoritma, što znači da nema univerzalnog koda za ovaj algoritam. Programer ima veliku slobodu u odabiru strategije kojom će mravlji algoritam rješavati njegov specifični problem. Mravi se kreću nasumično, ali s većom vjerojatnošću prema smjeru s jačim feromonskim tragom. Nakon nekoliko pokretanja programa, može se uočiti da li algoritam ima velike oscilacije u kvaliteti odabranih rješenja. Ako algoritam pokazuje velike oscilacije, potrebno je prilagoditi neke postavke

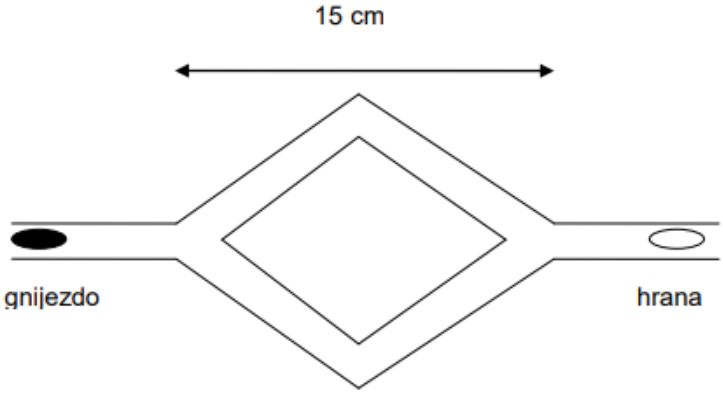
[1] Delta_tau je vrijednost kojom se nagrađuje, odnosno uvećava koeficijent puta za najboljeg mrava.

kako bi algoritam bolje funkcionirao. Važno je pravilno postaviti delta_tau kako se rješenja generirana u početnim iteracijama ne bi prebrzo nagrađivala, što bi moglo dovesti do konstrukcije suboptimalnog rješenja. Također, bitno je pravilno postaviti rho kako feromoni dobrih rješenja ne bi prebrzo isparili. U suprotnom, mravlji algoritam bi se nakon nekoliko iteracija sveo na slučajan odabir, što bi rezultiralo velikim varijacijama u kvaliteti rješenja koje generiraju pojedini mravi, što bi na kraju dovelo do loših ukupnih rezultata.

Eksperiment s dvokrakim mostom

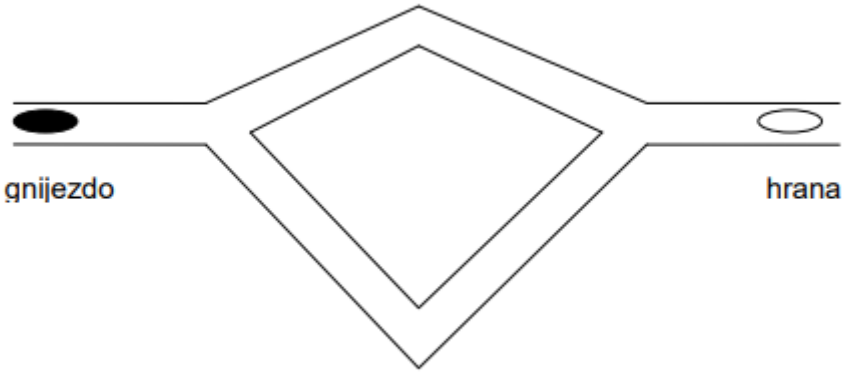
Mravlji algoritam je inspiriran istraživanjem koje su proveli Denebourg i njegovi suradnici, u kojem su koristili most s dva grananja koji povezuje mravinjak s izvorom hrane. Na temelju ovog eksperimenta, mravlji algoritam koristi feromonski trag koji mravi ostavljaju za sobom kako bi pronašao najbolje rješenje za problem. U eksperimentu su korišteni argentinski mravi čija je specifičnost da ostavljaju trag u oba smjera, odnosno na putu prema hrani te u povratku. Eksperiment su podešavali tako da su varirali omjer $r = \frac{l_l}{l_s}$ gdje je bila l_l dužina dulje grana, a l_s dužina kraće grane. U prvom eksperimentu postavili su jednaku duljinu l_l i l_s te je omjer r iznosio 1. Rezultat koji su dobili je bio da su u početnoj fazi mravi slučajno odabirali obje grane, ali na kraju su svi mravi koristili istu granu. Objašnjenje za ovakav ishod je sljedeće, mravi u početku nemaju preferenciju te s istom vjerojatnošću odabiru jedan ili drugi krak mosta, no ipak zbog nasumičnih odabira veći broj mrava će odabrati jedan krak mosta te će to rezultirati tome da se na jednom kraku mosta stvara veća količina feromona što rezultira tome da sljedeći mravi odabiru krak mosta koji ima jači feromonski trag. U završnoj fazi se može vidjeti kako svi mravi koriste isti krak mosta zbog količine feromonskih tragova koji su veći u odnosu na drugi krak mosta. To je primjer stigmergične[2] komunikacije gdje mravi iskorištavaju neizravnu komunikaciju posredovanu promjenom okoline u kojoj se kreću [15]. U idealnim uvjetima očekivali bi da će mravi podjednako izabrati oba puta te će feromonski trag na oba kraka mosta biti jednak.

[2] Stigmergija – oblik neizravne komunikacije posredovane modifikacijama okoliša



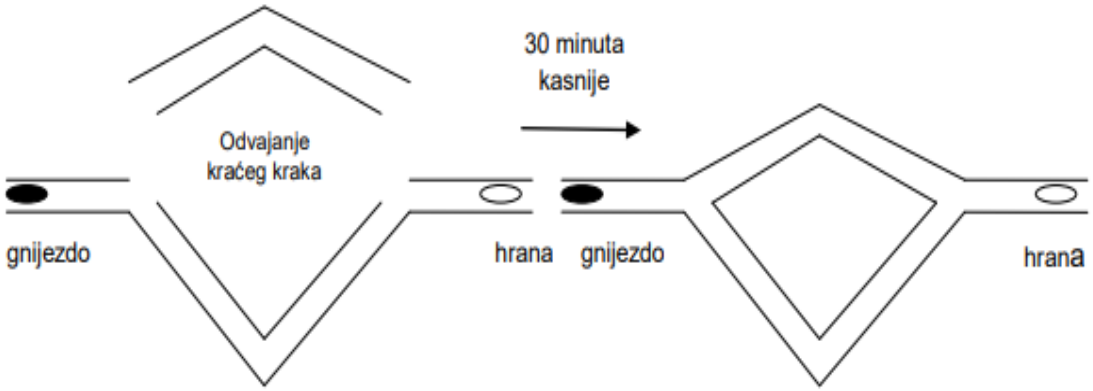
Slika 9: Prvi eksperiment (Izvor: M..Dorigo i T. Stutzle, 2004.)

Drugi scenarij eksperiment je imao jedan dvostruko dulji krak mosta od drugog. a mravi su opet imali slobodnu volju odabira puta do hrane. Omjer r u ovom scenariju je bio 2. Na slici 10 vidimo grafičku skicu drugog scenarija. Dobiveni rezultat bio je očekivan jer su mravi nakon nekog vremena odabrali kraći put. U početku su, kao i u prvom scenariju, mravi nasumično birali put. Međutim, za razliku od prvog pokusa gdje su oba puta bila jednake duljine, ovdje su mravi koji su odabrali kraći put brže došli do hrane i brže se vratili, što je rezultiralo većom koncentracijom feromona na kraćem putu.
U konačnici mravi odabiru kraći krak mosta. Povećana koncentracija feromona na kraćem putu potaknula je mrave da se kreću tim smjerom. Rezultati su pokazali da se feromon brže nakuplja na kraćem putu, što potiče većinu mrava da odabere taj put. Jedan od zanimljivih zaključaka do kojeg su znanstvenici došli je da neće svi mravi odabrati najkraći put do hrane, čak i ako je jedan put dvostruko duži od drugog. Mali broj mrava će se kretati dužim putem i ostavljati tragove feromona na tom putu, ali to neće utjecati na izbor većine jer su tragovi feromona na kraćem putu jači. Deneborgh i suradnici su ponašanje tih mrava tumačili kao „istraživanje puta“. Znanstvenici su otkrili da mravi koriste feromone kako bi pronašli hranu. Kada je koncentracija feromona veća na kraćem putu, to privlači većinu mrava da se kreću tim smjerom. Feromoni se brže gomilaju na kraćem putu, što potiče većinu mrava da odabere taj put. Unatoč tome što je jedan put dvostruko duži od drugog, neće svi mravi odabrati najkraći put do hrane. Mali broj mrava će se kretati duljim krakom te ostavljati feromonske tragove na tome putu, ali oni neće utjecati na odabir većine jer su jači feromonski tragovi na kraćem kraku. Znanstvenici su ovo ponašanje protumačili kao “istraživanje puta” [15].



Slika 10: Drugi eksperiment (Izvor: M.Dorigo i T.Stutzle, 2004.)

U trećem eksperimentu, znanstvenici su povezali mravinjak i izvor hrane s jednim krakom mosta. Nakon pola sata, dodan je drugi krak mosta koji je bio dvostruko kraći od prvog. Mravi su na početku imali pristup samo jednom kraku mosta, pa su na tom kraku ostavljali feromonski trag. Nakon dodavanja kraćeg kraka mosta, mravi su nastavili odabirati dulji krak mosta jer je na njemu bila velika koncentracija feromona. Zaključak znanstvenika je bio da je hlapljenje bilo presporo što je rezultiralo nemogućnost mravi da istraže nove puteve i otkriju kraći krak mosta. Istraživanje pravih kolonija mrava potvrdilo je da mravi koji ostavljaju trag feromona samo kada se vraćaju s hrane u gnijezdo ne mogu pronaći najkraći put. Slika 11 prikazuje skicu trećeg eksperimenta [15].



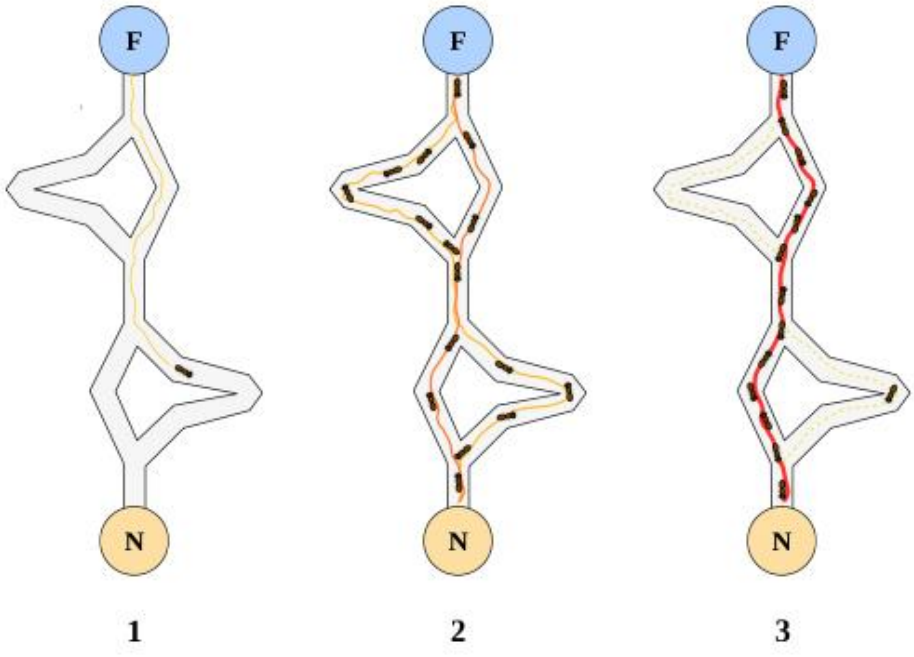
Slika 11: Treći eksperiment (Izvor: M.Dorigo i T.Stutzle, 2004.)

Goss je razvio statistički model ponašanja mrava koji je prikazan sljedećom formulom.

$$p_1 = \frac{(m_1 + k)^h}{(m_1 + k)^h + (m_2 + k)^h}$$

Ako bi određen broj mrava (m_1) odabralo prvi most, a određen broj mrava (m_2) odabralo drugi most. Vjerojatnost da mrav odabere prvi most iznosi p_1. U tom slučaju vrijedi $p_2 = 1 - p_1$. Monte Carlo simulacija pokazala je da je dobar odabir $k \approx 20$ i $h \approx 2$. Razvoj ovog modela ponašanja dovodi do inspiracije za prvi mravlji algoritam [8].

Eksperiment s dvostrukim mostom



Slika 12: Skica dvostrukog mosta (Izvor: A. Saxena i C. Mueller, 2018.)

Eksperiment s dvostrukim mostom bio je sličan prethodnom scenariju eksperimenta, s različitim duljinama l_l i l_s. U početku, mravi su imali slobodu odabira puta, što je vidljivo na slici 12, primjeru 2. Međutim, kasnije se dogodila situacija slična onoj u eksperimentu dva - mravi koji su odabrali kraći put vratili su se prije od onih koji su odabrali duži put, ostavljajući jači feromonski trag na kraćem putu. Kroz određeno vrijeme, vidljivo je na primjeru 3 da mravi odabiru najkraći put jer sadrži najveću koncentraciju feromona. Znanstvenici navode da mravi istražuju nove rute, ali zbog

sporog isparavanja tragova feromona i intenzivnog feromonskog traga na najkraćem putu, taj put se rijetko bira jer nije najbolji izbor [16]. Ovaj eksperiment s eksperimentima prikazanim u prethodnom poglavlju su potaknuli znanstvenike na razvoja mravljeg algoritma koji će funkcionirati na temelju ostavljanja feromonskih tragova koji će kroz određeno vrijeme pronaći optimalno rješenje. S obzirom na prikazano u eksperimentima, potrebno je dobro postaviti parametre pojačavanja i isparavanja feromonskih tragova te omogućiti mravima da kroz određeno vrijeme mogu konstruirati rješenje koje je bolje od trenutnog najbolje, a nema visoku razinu feromona. Prema Saxenu i Muelleru, mravlji algoritam bi se trebao sastojati od tri glavne procedure. Prva je konstrukcija rješenja koja upravlja kolonijom mrava koji istovremeno i asinkrono posjećuju susjedne čvorove grafa problema. Oni konstruiraju rješenje primjerom stohastičke politike lokalnog odlučivanja koja koristi tragove feromona i heurističke informacije. Druga procedura je ažuriranje feromonskih tragova kojom se mijenjaju feromonski tragovi, odnosno koeficijenti. Veće feromonske vrijednosti povećavaju vjerojatnost da će lukovi koji su odabrani od strane velikog broja mravi ili se koriste u trenutno najboljem rješenju biti ponovno korišteni od mrava u idućim iteracijama. Treća procedura je uključivanje lokalnih optimizacija gdje se u početnim iteracijama pokušava navesti mrave da kreiraju bolja rješenja kako bi što prije kreirali optimalno rješenje [17].

Karakteristike mrava

Ključne značajke priodnih mrava koje su pomogle razvoju mravljeg algoritma su:

- Kolonije mrava
- Feromonski trag
- Pronalaženje najkraćeg puta
- Stohastičke odluke

U prethodno prikazanim eksperimentima je prikazano kako mravi imaju ugrađenu sposobno optimizacije koja se temelji na ostavljanu feromona s pomoću kojih je kolonija mrava sposobna pronaći najkraći put između dvije točke u svom okruženju.

Kolonije mrava kao značajka za razvoj mravljeg algoritma znači da virtualni mrav u početku ne treba konstruirati optimalno rješenje, nego ga gradi u suradnji s ostatkom koloniji. Svaki biološki mrav kreira svoj put na kojemu ostavlja feromonski trag, što bi značilo da će svaki virtualni mrav kreirati rješenje koje će kolonija uzimati u obzir ako je najbolje te će ga nagraditi povećanjem feromonskih tragova, dok će svi ostali feromonski tragovi kreiranih rješenja hlapiti [15].

Feromonski trag kao značajka znači da virtualni mrav kreira svoje rješenje na temelju koeficijenata lukova pojedinih čvorova. Kod mrava u prirodi, feromoni su kemijske tvari koje ostali mravi iz iste kolonije mogu detektirati. U slučaju virtualnih mrava, numerički koeficijenti se povećavaju prema formuli $tau\_ij = tau\_ij + delta\_tau$ ako se tau ako se otkrije optimalan put. Ovdje tau_ij predstavlja koeficijent ili trag tau za određenog klijenta (točku u grafu), a $delata\_tau$ je konstantna vrijednost kojom se nagrađuju koeficijenti. Isparavanje feromona izračunava se formulom $tau\_ij = (1 - rho) * tau\_ij$ gdje tau_ij predstavlja trenutni koeficijent trag za pojedinog klijenta, a rho postotnu vrijednost smanjenja. Važno je dobro postaviti $delta\_tau$ i rho kako feromonski tragovi ne bi prebrzo rasli za početne iteracije ili kako ne bi došlo do brzog hlapljenja feromonskih tragova [15].

Pronalaženje najkraćeg puta je glavni cilj virtualnih mravi s obzirom na to da se radi o optimizacijskom algoritmu. Biološki mravi kroz vrijeme uz pomoć feromonskih tragova pronalaze najkraći put, no vidjeli smo u eksperimentima da nije uvijek tako [15]. Naime, ako određena skupina mrava prođe nekim pute koji ne mora nužno biti optimalna, zbog visoke razine feromonskih tragova, ostatak kolonije će pratiti taj put umjesto nekog drugog puta koji je optimalan. Važno je postaviti dobre parametre kako virtualni mravi ne bi kreirali rješenja koja su daleko od optimalnih, ali su početne iteracije ostavile dobar feromonski trag koji će kolonija pratiti u daljnjim iteracijama.

Stohastičke odluke su one koje se donose u nepoznatim i nejasnim okolnostima. Kada se kreira rješenje, analiziraju se samo ograničenja koja su postavljena u samom problemu. Feromoni su odgovor na donošenje takvih odluka. Mravi prilikom kreiranja puta koriste vjerojatnosno pravilo, odnosno postoji jednaka šansa da će mrav krenuti na jedan od N mogućih puteva. U kasnijim iteracijama stohastičke odluke se smanjuju zbog utjecaja feromonskih tragova. U eksperimentima je uočeno da mravi odabiru duži

put čak i nakon pojačanog feromonskog traga na optimalnom putu. Ova pojava naziva se “istraživanje puta” [15].

Vrste ACO algoritama

Postoje razne vrste mravljeg algoritma koje koriste osnove karakteristike mravljeg algoritma te imaju određene posebnosti specifične za pojedinu vrstu ACO algoritma. Postoji nekoliko popularnih varijacija mravljeg algoritma koje ćemo kratko predstaviti u ovom poglavlju, a to su:

- Mravlji sustav (eng. *Ant system* – AS)
- Sustav kolonija mrava (eng. *Ant colony system* – ACS)
- Elitistički mravlju sustav (eng. *Elitist ant system* – EAS)
- Max-min mravlji sustav (eng. *Max-min ant system* – MMAS)
- Sustav mrava s trima granica (eng. *Three bound ant system* – TBAS)
- Mravlji sustav temeljen na rangu (eng. *Rank-based ant system* – ASrank)
- Paralelna optimizacija kolonije mrava (eng. *Parallel ant colony optimization* - PACO)
- Ortogonalni mravlji sustav (eng. *Continuous orthogonal ant colony* – COAC)
- Rekurzivni mravlji sustav (eng. *Recursive ant colony optimization* - RACO)

Mravlji sustav

Prvi mravlji algoritam bio je mravlji sustav (eng. *Ant system – AS),* a predstavio ga je 1992. godine Marco Dorigo 1992. Mrav pri konstrukciji rješenja primjenjuje slučajno pravilo odlučivanja kako bi donio odluku koji će vrh (klijenta) posjetiti sljedeće Vjerojatnost s kojom će mrav k, koji se trenutno nalazi u vrhu i, posjetiti vrh j, dana je sljedećom formulom:

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}$$

gdje je $\eta_{ij=1/d_{ij}}$ heuristička vrijednost, a α i β određuju relativan utjecaj traga feromona i heurističke vrijednosti. N_i^k su neposjećeni gradovi mrava k kada se nalazi u gradu i, a vjerojatnost odabira nekog grada izvan N_i^k je 0. Korištenjem pravila slučajnog

odlučivanja mrava, vjerojatnost izbora određenog grada se povećava s povećanjem vrijednosti traga feromona τ_{ij} i heuristike η_{ij}. Uloga parametara α i β je sljedeća. Ako je $\alpha = 0$, najvjerojatnije će biti odabrani gradovi koji su najbliži. Ako je $\beta = 0$, na izbore sljedećeg čvora utječe isključivo koeficijent feromona, bez heuristike. Za vrijednosti $\alpha > 1$, konstrukcija rješenja mrava brzo dolazi do zastoja, odnosno do stanja u kojem svi mravi prate isti put i stvaraju identično rješenje, koje je često zna biti loše. Karakteristično za mravlji sustav je to da su se feromonski tragovi ažurirali od strane svih mrava koje konstruiraju rješenje. Isparavanje feromonskog traga implementirano je s pomoću sljedeće formule:

$$\tau_{ij} = (1 - \rho) * \tau_{ij}, \forall (i, j) \in L$$

gdje je $0 < \rho \leq 1$ stopa isparavanja feromona. Parametar ρ koristi se kako bi se spriječilo beskonačno povećanje feromonskog traga te omogućuje algoritmu da zaboravi na loše odluke koje je donio, omogućavajući mu da istraži druga potencijalno bolja rješenja. Vrijednost feromona eksponencijalno se smanjuje s brojem iteracija ako određeni vrh nije odabran u nekoliko iteracija. Povećavanje tragova feromona izračunava se prema formuli:

$$\tau_{ij} = \tau_{ij} + \sum_{k-1}^{m} \Delta\tau_{ij}^{k}, \forall (i, j) \in L$$

gdje je $\Delta\tau_{ij}^{k}$ količina feromona koju virtualni mrav k odloži u luk koji je posjetio iz prethodnog vrha, odnosno poveća im koeficijent. Što više mrava u konstruiranom rješenju sadrži određeni vrh to će vrijednost feromona luka prema tom vrhu sadržavati veću vrijednost feromonskog traga [15].

Sustav kolonije mrava

Sustav kolonije mrava, poznat kao *Ant Colony System* (ACS), predstavlja varijantu mravljeg algoritma. Njegova se jedinstvenost ogleda u tome što proces isparavanja i taloženja feromona provodi samo na najučinkovitijem putu. Osim toga, druga ključna značajka ovog sustava je lokalno ažuriranje feromona. Svaki put kada mrav tijekom konstrukcije rješenja prelazi na sljedeći nasumično odabrani vrh, uklanja dio feromonskih tragova u trenutnom gradu, čime potiče istraživanje alternativnih puteva. Ovaj algoritam se temelji na biološkim mravima, ali za razliku od njih, njega

karakterizira brisanje prethodno ostavljenih feromonskih tragova. Ažuriranje feromonskih tragova se događa isključivo za globalno najbolje rješenje. Osim globalnog ažuriranja feromonskih tragova, koje se provodi samo na najboljem globalnom rješenju, koristi se i lokalno ažuriranje tragova feromona. To znači da mrav, nakon što prijeđe u sljedeći grad tijekom izgradnje rješenja, uklanja dio tragova feromona prema formuli:

$$\tau_{ij} = (1 - \xi) * \tau_{ij} + \xi\tau_0$$

gdje su ξ, $0 \leq \xi \leq 1$, i τ_0 dva parametra. Vrijednost τ_0 postavljena je da odgovara početnoj vrijednosti tragova feromona. Jedan od učinaka lokalnog ažuriranja tragova feromona je da svaki put kada mrav posjeti određeni grad, taj grad postaje manje privlačan za ostale mrave tijekom njihove izgradnje ture, a trag feromona tog grada τ_{ij} se smanjuje. Ova metoda lokalnog ažuriranja feromonskih tragova omogućuje istraživanje novih puteva i konstrukciju rješenja koja do sada nisu bila generirana. Ovaj pristup problemu ima značajnu prednost jer sprječava algoritam da upadne u fazu stagnacije.

Elitistički mravlji sustav

Elitistički mravlji sustav (eng. *Elitist ant system* – EAS) je mravlji algoritam čija je specifičnost što globalno najbolje rješenje poveća feromone na svom putu nakon svake iteracija. To se događa i kada najbolje globalno rješenje nije ni konstruirano u toj iteraciji. Cilj ovakve strategije je usmjeriti potragu svih mrava da pri konstrukciji rješenja koriste poveznice trenutno najbolje rute. Ovaj pristup inspiriran je ponašanjem pravih mrava. Kada mravi pronađu hranu, vraćaju se u svoju koloniju dok ostavljaju tragove feromona. Ako drugi mravi pronađu takav put, vjerojatno neće nastaviti putovati nasumično, već će slijediti trag, vraćati se i pojačavati ga svojim feromonima [18].

Max-min mravlji sustav

Max-min mravlji sustav (eng. *Max-min ant system* – MMAS) donosi neke izmjene u algoritam Mravljeg sustava. Ovaj algoritam se usredotočuje na najbolje rješenje u svakoj iteraciji ili na najbolje globalno rješenje te povećava feromonske tragove na najbolje lokalnom rješenju ili na najboljem globalnom rješenju [19]. Kao što smo naveli u primjeru sustav kolonije mrava, ovakav pristup ažuriranja feromonskih

tragova može dovesti algoritam do stagnacije gdje će svaki mrav konstruirati isto rješenje. Ova situacija može nastati kada se u početnim iteracijama feromoni na putu koji ne predstavlja optimalno rješenje pojačavaju, što dovodi do toga da ostali mravi slijede taj put umjesto da istražuju alternativne puteve. Kako bi se ovakva situacija izbjegla, koristi se metoda koja ograničava moguće vrijednosti feromonskih tragova unutar određenog intervala:

$$[\tau_{min}, \tau_{max}].$$

Ograničavanje vrijednosti feromonskih tragova je specifična značajka MMAS algoritma, sve vrijednosti feromonskih tragova postavljaju se na gornju granicu, odnosno τ_{max}. Ova karakteristika omogućuje mravima da na početku pretraživanja intenzivnije istražuju različite puteve, čime se smanjuje vjerojatnost stvaranja suboptimalnih rješenja u početnim iteracijama algoritma, što bi kasnije moglo dovesti do stagnacije. Uz početno postavljanje na τ_{max}, MMAS algoritmi se odlikuju malim postotkom isparavanja svih feromonskih tragova, što zajedno omogućuje bolju konstrukciju rješenja od samog početka. Ako dođe do stagnacije ili ako nema poboljšanja kroz određeni broj uzastopnih iteracija, trag feromona se ponovno inicira. Isparavanje feromonskih tragova provodi se prema formuli navedenoj u mravljem sustavu, a dodavanje novih feromona implementira se na sljedeći način:

$$\tau_{ij} = \tau_{ij} + \Delta\tau$$

U primjenama MMAS algoritma, koriste se oba pravila ažuriranja (najbolji do sada ili najbolji u trenutnoj iteraciji), a koji će se od njih koristiti ovisi o vrsti problema. Koliko često se koristi jedno pravilo u odnosu na drugo određuje koliko je algoritam pohlepan [16].

Sustav mrava s trima granicama

Da bi se poboljšala određena svojstva MMAS algoritma, razvijen je “Sustav mrava s trima granicama” (eng. *Three Bound Ant System* - TBAS) [20, 21]. Ključne prednosti ovog algoritma uključuju manju vremensku složenost i veću prilagodljivost u održavanju granica tragova feromona. U usporedbi s Max-min mravljim sustavom, TBAS donosi nekoliko ključnih razlika:

- postoje tri granice feromona (donja granica τ_{LB}, gornja granica τ_{UB} i kontrakcijska granica $\tau_{CB} = \omega \cdot \tau_{UB}$)
- ispravanje feromona je periodično

• postoji jedinstveni postupak za pojačanje koeficijenta feromona
• vrijednost donje granice τ_{LB} jednaka je početnoj vrijednosti tragova feromona

Tijekom konstrukcije rješenja, TBAS koristi pravilo odlučivanja koje je slučajno-proporcionalno, baš kao i MMAS. Također koristi sve parametre kao i MMAS, ali s dodatkom parametra ω, gdje je $\frac{\tau_{LB}}{\tau_{UB}} \leq \omega \leq 1$. Proces ažuriranja feromona u TBAS-u započinje povećavanjem koeficijenta feromona, a zatim slijedi isparavanje feromona, ali samo ako feromonski trag premašuje gornju granicu feromona τ_{UB}. U tom slučaju, feromonski trag se prilikom isparavanja postavlja na vrijednost τ_{UB}. Kod nagrađivanje je potrebno prvotno odabrati strategiju nagrađivanja. Najčešće korištene strategije u literaturi su nagrađivanje najboljeg rješenja u trenutnoj iteraciji i nagrađivanje najboljeg rješenja do sada. Ako su s^{bs} komponente rješenja koje trebaju biti nagrađene, tada se nagrada dodjeljuje prema formuli:

$$\tau_c = \tau_c + \frac{Q_i}{f(s^{bs})}, \forall \mathrm{c} \in s^{bs}$$

U prvoj iteraciji, vrijednost Q_0 je postavljena na 1, dok se u svakoj sljedećoj iteraciji vrijednost Q_{i+1} izračunava kao $Q_i/(1-p)$. Istraživanja su pokazala da je TBAS algoritam brži za 1% do 20% u usporedbi s MMAS algoritmom. Osim toga, provedeni su i eksperimenti koji su se fokusirali na kvalitetu rješenja, gdje se TBAS algoritam također pokazao kao superiorna opcija u odnosu na ostale mravlje algoritme [20, 21].

Mravlji sustav temeljen na rangu

Mravlji sustav temeljen na rangu (eng. *Rank-based ant system* – ASrank) je mravlji algoritam gdje su sva rješenja rangirana prema duljini. Fiksno broj najbolji rješenja u iteraciji smije ažurirati svoje feromone. Ključna značajka mravljeg sustava temeljenog na rangu je uključivanje ideje rangiranja u proces ažuriranja feromona. Najbolji mravi u iteraciji ažuriraju feromonske tragove proporcionalne njihovom rangu [22].

Paralelna optimizacija kolonije mrava

Paralelna optimizacija kolonije mrava (eng. *Parallel ant colony optimization* - PACO) je sustav kolonije mrava s komunikacijskim strategijama. Specifično je da su kod ovog algoritma mravi podijeljeni u nekoliko skupina. Za rješavanje problema trgovačkog putnika predlaže se sedam komunikacijskih metoda za ažuriranje razine feromona između grupa. Karakteristično je također da se nekoliko sekvencijalnih ACO koji koriste identične ili različite parametre istovremeno izvršava na skupu procesora [23].

Ortogonalni mravlji sustav

Ortogonalni mravlji sustav (eng. *Continuous orthogonal ant colony* – COAC) koristi ortogonalni dizajn te mravi u izvedivoj domeni mogu istraživati odabrana područja brzo i učinkovito uz poboljšanu mogućnost globalnog pretraživanja i točnosti. Implementacijom metode "adaptivnog regionalnog radijusa", predloženi algoritam može smanjiti vjerojatnost zarobljavanja u lokalnim optimumima i stoga poboljšati sposobnost i točnost globalnog pretraživanja. Također se koristi elitistička strategija za zadržavanje najvrjednijih rješenja.

Rekurzivni mravlji sustav

Rekurzivni mravlji sustav je (eng. *Recursive ant colony optimization* - RACO) je rekurzivni oblik mravljeg sustava koji cijelu domenu pretraživanja dijeli na nekoliko poddomena i rješava cilj na tim poddomenama. Potom se rezultati iz svih poddomena uspoređuju i nekoliko najboljih od njih se promiče za sljedeću razinu. Poddomene koje odgovaraju odabranim rezultatima dalje se dijele i proces se ponavlja dok se ne dobije rezultat željene preciznosti. RACO primjenjuje ACO rekurzivno uvodeći dodatni termin 'dubina' koji odlučuje o opsegu rekurzije. Svaka dubina je uobičajeni ACO s tri koraka po iteraciji, 'praćenje feromona', 'ažuriranje feromona' i 'odabir grada'. Rezultati svake dubine doprinose konstruiranju modela za sljedeću dubinu, a raspon vrijednosti za svaki parametar smanjen je oko stvarnog rješenja [25].

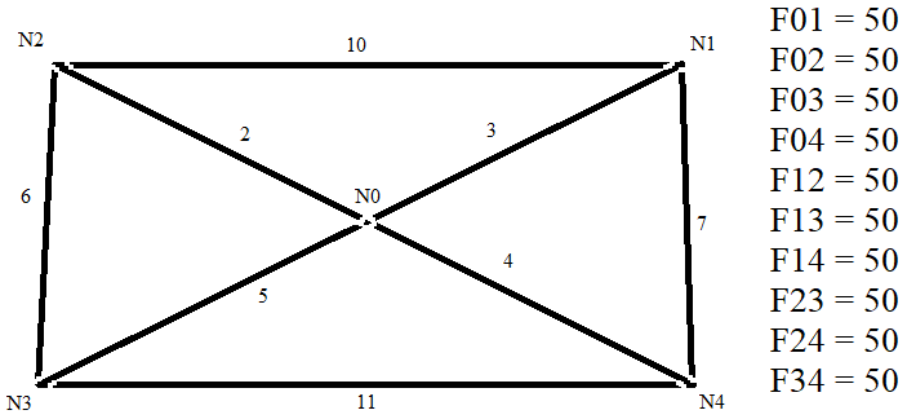
Implementacija mravljeg algoritma

```
1. Mravlji algoritam
2. Početak
3.      Postavi početne parametre, inicijaliziraj feromonske tragove
4.      Dok sve iteracije kolonija mravi ne završe Radi
5.              Konstruiraj rješenje
6.              Ažuriraj feromonske tragove
7.      Kraj
8. Kraj
```

Slika 13: Pseudokod jednostavnog mravljeg algoritma (Izvor: vlastita izrada)

Na slici 13 možemo vidjeti pseudokod jednostavnog mravljeg algoritma koji u početku postavlja parametre, a to uključuje delta_tau, rho, računa matricu udaljenosti, učitava datoteku instance problema, inicijalizira vektor vozila s kapacitetom te inicijalizira feromonske tragove. Nakon toga nam slijedi while petlja koja ima ograničeni broj iteracija, a svaka iteracija sadrži unaprijed zadani N broj mrava. Unutar while petlje svaki pojedini mrav radi na konstrukciji svog rješenja uzimajući u obzir koeficijente feromonskih tragova i nakon što iteracija završi, slijedi provjeravanje rješenja te ovisno o vrsti mravljeg algoritma se radi ažuriranje feromonskih tragova. Najbolje globalno rješenje se pamti te se nakon završetka while petlje ispisuje.

Ideja mravljeg algoritma na primjeru manjeg problema

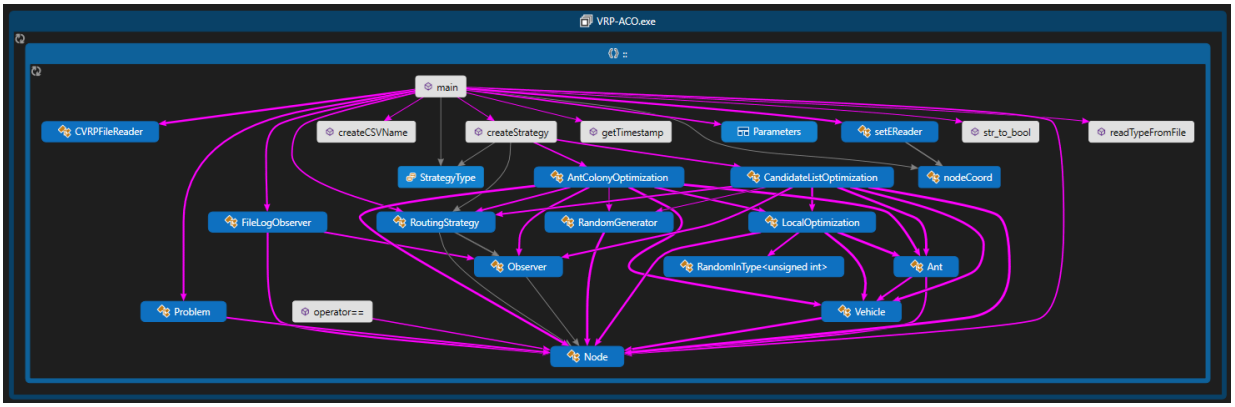


Slika 14: primjer mravljeg algoritma za CVRP (Izvor: vlastita izrada)

Na slici 14 ću objasniti ideju mravljeg algoritma za problem usmjeravanja vozila s ograničenim kapacitetima. Ova problem smo rješavali u prethodnim poglavljima te smo sada samo dodali feromonske tragove koji se nalaze s desne strane slike, a dodani su na lukove između pojedinih klijenata, odnosno čvorova. Početna vrijednost feromonskih tragova iznosi 50 što je karakteristika MMAS algoritma te ćemo se fokusirati na implementaciju jednog MMAS algoritma za rješavanje ovog problema. U prvoj iteraciji kolonije mrava, svaki mrav konstruira rješenje. Svaki mrav sadrži listu vozila, a svako vozilo sadrži listu klijenta koje je obišao. Moramo paziti na ograničenja postavljena vezano za pojedino vozilo kod odabira klijenta. Nakon što mrav konstruira rješenje onda slijedi evaluacija puta koju smo pokazali u prethodnim poglavljima, a radi se o računanju sume udaljenosti svakog vozila koje je prešao na svojoj ruti. Najbolje rješenje se pamti, a nakon što završi iteracija kolonije mrava slijedi ažuriranje feromonskih tragova koje funkcionira na principu da se prvo svima smanji koeficijent feromonskih tragova za rho. Recimo da je u našem primjeru rho postavljen na 0.02, onda bi to značilo da će u prvom dijelu ažuriranja svi feromonski tragovi imati 49, a ruta koja će biti najbolja globalna će imati također povećavanje koeficijenata feromonskih tragova za $delta\_tau$. U našem primjeru neka je $delta\_tau$ postavljen na 1 te bi to značilo da će već nakon prve iteracije određen broj feromonskih tragova imati 49, a određen broj 50. Ovdje smo vidjeli kako funkcionira ažuriranje feromona te u sljedećoj iteraciji mravi imaju mogućnost odabrati bolja rješenja. Naravno radi se o maloj razlici koja u početnim iteracijama može biti zanemarena kako bi mravi mogli samostalno istraživati puteve, ali prolaskom svake iteracija feromonski tragovi se počinju razlikovati te u posljednjoj iteraciji bi trebali dobiti mrava koji će konstruirati optimalno rješenje korištenjem feromonskih tragova jer će veliki broj koeficijenata feromona biti približno jednak tau_min, dok će koeficijenti feromona na optimalnom putu biti jednaki tau_max. Ovdje je prikazan jednostavan primjer izvođenja MMAS algoritma s 4 klijenta. U nastavku ćemo objasniti implementaciju koda koji će biti predviđen za rješavanje većih instanci problema od ovih koje smo ovdje objašnjavali.

6. Programski kod

U uvodu smo rekli kako je programski kod napisan u C++ programskog jeziku u okruženju Microsoft Visual Studio 2019. Programski kod je napisan objektno-orijentiranim pristupom za razliku od koda kojega smo koristili u radu Algoritam računalne inteligencije za rješavanje problema putujućeg lopova gdje smo koristili funkcijski pristup [6]. Programski kod sadrži 13 headera: Ant.h, AntColonyOptimization.h, RoutingStategy.h, CVRPFileReader.h, FileLogObserver.h, LocalOptimization.h, Node.h, Observer.h, Problem.h, RandomGenerator.h, RandomInType.h, setEReader.h i Vehicle.h. Također, projekt se sastoji od 11 klasa koje su: Ant, AntColonyOptimization, CandidateListOptimization, CVRPFileReader, FileLogObserver, LocalOptimization, Node, RandomGenerator, setEReader, Vehicle i Main upravlja svim ostalim klasama. Na ovaj način je implementiran objektno-orijentirani pristup.



Slika 15: Prikaz stukture implemenitiranih klasa (Izvor: code map u Visual Studio Enerprise 2022)

Struktura podataka

Na slici 15 vidimo strukturu prikaza korištenja pojedinih klasa i odnosa među njima te ćemo sada proći pojedinu klasu i objasniti njezinu strukturu podataka. Strukture podataka igraju ključnu ulogu u strukturiranju i organiziranju podataka u objektno-orijentiranom pristupu olakšavajući dizajn, razumijevanje te nadogradnju softverskih sustava. Oni pružaju način modeliranja entiteta i odnosa iz stvarnog svijeta što dovodi do intuitivnijeg i učinkovitijeg dizajna softvera. Za razvoj algoritma koji će rješavati

problem usmjeravanja vozila s ograničenima važna je struktura podataka i dobro postavljeni odnosi koji će u kasnijem razvoju bitno olakšati samu implementaciju MMAS algoritma.

Klasa Node

```
class Node {
public:
  Node(int id, int demand);
  int getId() const;
  int getDemand() const;
  bool isRouted() const;
  void setIsRouted(bool routed);
  std::vector<int> candidateList;

private:
  int NodeId;
  int demand;
  bool IsRouted;
};
```

Slika 16: Prikaz strukture podataka klase $Node$ (Izvor: vlastita izrada)

Na slici 16 možemo vidjeti prikaz klase $Node$. Klasa $Node$ sadrži karakteristike pojedinog klijenta, odnosno možemo vidjeti kako sadrži $NodeId$ odnosno id klijenta, $demand$ koji predstavlja potražnju te bool tip podataka $IsRouted$ koji predstavlja je li pojedini klijent već korišten pri konstrukciji rješenja. Od metoda možemo vidjeti konstruktor koji kreira objekt klijenta tako da mu se pošalje id i potražnja te vidimo tri geter metode za dohvaćanje podataka koje smo upravo predstavili te jednu seter metodu za postavljanje varijable $IsRouted$ kada se pojedini klijent iskoristi za konstrukciju rješenje. $NodeId$ i $demand$ su integer tip podataka što znači da te varijable mogu biti samo cjelobrojne, dok je $IsRouted$ bool tip podataka što znači da može biti true ili false. Uz to možemo vidjeti kako svaki klijent sadrži vektor cjelobrojnih

vrijednosti pod nazivom $candidateList$ koja se koristi prilikom strategije s listom favorita o kojoj ćemo kasnije govoriti.

Klasa Vehicle

```
class Vehicle
{
public:
  double totalCost;


  Vehicle(int capacity);

  void AddNode(Node& Customer);
  bool CheckIfFits(int dem);


  std::vector<Node> getRoute() const;

  int getLoad();

  void setLoad(int load);

  int getCurrentLocation();

  void setCurrentLocation(int currentLocation);

  int getCapacity();

  std::vector<Node>& getNonConstRoute();

  int getLeftOverCapacity() const;
  void reverse(size_t i1, size_t j);

  void removeNode(Node& node);
  void addNodeAtIndex(Node &node, int index);

  void removeNodeAtIndex(int index);

private:
  std::vector<Node> route;
  int capacity;
  int load;
  int currentLocation;
  int leftOverCapacity;

};
```

Slika 17: Prikaz strukture podataka klase $Vehicle$ (Izvor: vlastita izrada)

Na slici 17 možemo vidjeti prikaz klase $Vehicle$ koja sadrži karakteristike pojedinog vozila s ograničenim kapacitetom. U klasi $Vehicle$ možemo vidjeti varijablu $route$ koja se sastoji do vektora klase $Node$ te ona predstavlja rutu koju će pojedino vozilo kreirati prilikom konstrukcije rješenja, također možemo vidjeti varijablu $capacity$ koja predstavlja ograničeni kapacitet pojedinog vozila, varijablu $load$ koji predstavlja trenutne ispunjene zahtjeve klijenata na svojoj ruti, varijablu $currentLocation$ koji

predstavlja id lokacije klijenta kod kojeg se trenutno nalazi te varijablu $leftOverCapacity$ koja predstavlja razliku varijable $capacity$ i $load$. Od metoda možemo vidjeti konstruktor koji kreira objekt vozila s parametrom kapaciteta vozila koje mu pri kreiranju postavlja kao ograničenje. Vidimo pet getera za dohvaćanje varijabli rute, kapaciteta, opterećenja, trenutne lokacije i preostalog slobodnog kapaciteta vozila. Od metoda postavljanja vrijednosti imamo seter za opterećenje i seter za trenutnu lokaciju. Uz to imamo metodu dodavanja $Noda$ u rutu koja imam parametar $Node$ te ga dodaje u vektor rute. Možemo vidjeti kako klasa $Vehicle$ implementira metodu CheckIfFits kojoj je zadaća provjeravanje potražnje novog klijenta te vraćanje istinite tvrdnje da li ona stane pojedino vozilo nakon što je to vozilo smanjilo svoj početni kapacitet potražnjom ostalih klijenata na svojoj ruti. Kasnije ćemo objasniti implementaciju lokalne optimizacije, a za sada ovdje vidimo četiri metode koje se koriste kod lokalne optimizacije, a to su metoda reverse koja zamjenjuje redoslijed dva klijenta, metoda removeNode koja briše jednog klijenta iz rute, metoda addNodeAtIndex koja dodaje klijenta na točno određeno mjesto u ruti te metoda removeNodeAtIndex koja briše klijenta na specifičnom indexu u ruti.

Klasa Ant

```
class Ant {
public:
  Ant(int numberOfVehicles, int capacity);
  std::vector<Vehicle>& getVehicles();
  double evaluate(std::vector<std::vector<double>> distances);
  double evaluateVehicle(
      std::vector<Node> route,std::vector<std::vector<double>> distances);
  Ant();

private:
  std::vector<Vehicle> vehicles;
};
```

Slika 18: Prikaz strukture podataka klase Ant (Izvor: vlastita izrada)

Na slici 18 možemo vidjeti prikaz klase Ant koja sadrži karakteristike pojedinog mrava u koloniji. U klasi Ant možemo vidjeti varijablu $vehicles$ koja se sastoji do vektora klase $Vehicle$ te ona predstavlja skupinu vozila koja će jednom mravu treba za konstrukciju rješenja. Možemo vidjeti prazni konstruktor i konstruktor koji kreira objekt mrava s parametrom broja vozila i kapaciteta pojedinog vozila. Vidimo geter za

dohvaćanje varijable vozila. Od metoda imamo evaluate i evaluateVehicle gdje prva metoda radi izračun troškova cjelokupnog rješenja dok druga radi na izračunu troškova pojedinog vozila.

Klasa CVRPFileReader

```
#include <string>
#include <vector>
class CVRPFileReader {
public:
  CVRPFileReader(const std::string &filename);

  int dimension() const { return dimension_; }
  int capacity() const { return capacity_; }
  int vehicles() const { return vehicles_; }
  std::string type() const { return type_; }
  int cities() const { return demand().size()-1; }
  const std::vector<std::vector<int>> &edge_weight() const {
    return edge_weight_;
  }
  const std::vector<int> &demand() const { return demand_; }

private:
  int dimension_;
  int capacity_;
  int vehicles_;
  std::string type_;
  std::vector<std::vector<int>> edge_weight_;
  std::vector<int> demand_;
};
```

Slika 19: Prikaz strukture podataka klase $CVRPFileReader$ (Izvor: vlastita izrada)

Na slici 19 možemo vidjeti prikaz klase $CVRPFileReader$ koja predstavlja čitanje datoteke problema koji ima matricu udaljenosti postavljenu eksplicitno. U klasi $CVRPFileReader$ možemo vidjeti integer varijable $dimension\_$, $capcaity\_$ i $vehicles\_$ te string varijablu $type\_$ koja predstavlja tip datoteke. Tip datoteke se određuje na temelju matrice udaljenosti. Eksplicitno zadane su one koje nemaju koordinate nego je u samoj

datoteci zadana matrica udaljenosti dok implicitno zadana matrica udaljenosti je ona gdje je zapisana x i y os svakog vrha te ju je potrebno izračunati korištenjem Euklidove formule za udaljenost. Varijabla $edge\_weight\_$ predstavlja matricu udaljenosti i varijabla $demand\_$ koja je vektor integer vrijednosti predstavlja potražnju pojedinog klijenta za cijeli problem. Ova klasa služi za učitavanje problema i ona se kreira na početnu u slučaju da se radi o eksplicitno zadanoj matrici udaljenosti za problem usmjeravanja vozila. Možemo vidjeti konstruktor koji kao parametar ima naziv datoteke u kojoj se nalazi instanca problema. Vidimo šest getera za dohvaćanje varijabli $dimension\_$, $capcaity\_$, $vehicles$, $type\_$, $edge\_weight\_$ i $demand\_$ te metodu cities() koja vraća broj klijenata na temelju pročitanje potražnje za pojedinog klijenta.

Klasa setEReader i nodeCoord

```
class nodeCoord {
public:
  int id;
  int x;
  int y;
  int demand;
};

class setEReader {
public:
  setEReader(const std::string &filename);
  void print_distance_matrix() const;
  std::vector<nodeCoord> get_nodes() const;
  int get_capacity() const;
  std::vector<std::vector<double>> get_distance_matrix() const;
  std::string getType() const;

private:
  std::vector<nodeCoord> nodes_;
  std::vector<std::vector<double>> distance_matrix_;
  int capacity_;
  std::string type;

  double euclidean_distance(const nodeCoord &a, const nodeCoord &b) const;
};
```

Slika 20: Prikaz strukture podataka klasa $setEReader\ i\ nodeCoord$ (Izvor: vlastita izrada)

Na slici 20 možemo vidjeti prikaz klase $setEReader$ koja predstavlja čitanje datoteke problema koji ima matricu udaljenosti postavljenu implicitno. U klasi $setEReader$ možemo vidjeti integer varijablu $capacity\_$ te string varijablu $type\_$ koja predstavlja tip datoteke. Ovaj reader je namijenjen čitanju instance problema koja ima implicitno zadanu matricu udaljenosti. Varijabla $distance\_matrix\_$ predstavlja matricu

udaljenosti i varijabla $nodes\_$ koja je vektor klase $nodeCoord$ za zapis implicitno zadanih vrhova. Unutar klase $nodeCoord$ vidimo da je sadržan $demand$ varijabla za potražnju pojedinog klijenta koja je u prethodnom poglavlju bila unutar reader klase. Ova klasa služi za učitavanje problema i ona se kreira na početnu u slučaju da se radi o implicitno zadanoj matrici udaljenosti za problem usmjeravanja vozila te također možemo primijetiti privatnu metodu $euclidean\_distance$ koja računa udaljenost između dva vrha korištenjem euklidske formule za udaljenost te ona kao parametre prima dva objekta tipa $nodeCoord$ i na temelju x i y osi koje su sastavi dio te klase računa udaljenost između dva klijenta. Možemo vidjeti konstruktor koji kao parametar ima naziv datoteke u kojoj se nalazi instanca problema. Vidimo četiri getera za dohvaćanje varijabli $nodes\_$, $distance\_matrix\_$, $capacity$ i $type\_$ te metodu print_distance_matrix() koja ispisuje matricu udaljenosti.

Klasa RandomGenerator

```
#include <vector>
#include "Node.h"

class RandomGenerator {
public:
  static Node
  selectNextNode(const std::vector<Node> &nodes,
                 const std::vector<std::vector<double>> &pheromoneTrails,
                 const Node &currentNode);

  static Node selectNextNodeWithHeuristic(
      const std::vector<Node> &nodes,
      const std::vector<std::vector<double>> &pheromoneTrails,
      const std::vector<std::vector<int>> &distances,
      const Node &currentNode, double alpha, double beta);

  static Node* selectNextNodeWithHeuristicCompleted(
      std::vector<Node *> &nodes,
      const std::vector<std::vector<double>> &pheromoneTrails,
      const std::vector<std::vector<double>> &distances, Node *currentNode,
      double alpha, double beta, double u);
};
```

Slika 21: Prikaz strukture podataka klase $RandomGenerator$ (Izvor: vlastita izrada)

Na slici 21 možemo vidjeti prikaz klase $RandomGenerator$ koja sadrži metode korištene za odabir sljedećeg klijenta s pomoću feromonskih tragova i heuristike. Ovdje možemo vidjeti tri statičke metode koje na temelju feromonskih tragova i heuristike vračaju klijenta koji još nije posjećen. Trenutno se u algoritmu koristi samo metoda $selectNextNodeWithHeuristicCompleted$.

Klasa FileLogObserver

```
class FileLogObserver : public Observer {
private:
  std::ofstream logFile;
  std::string filename;

public:
  FileLogObserver(const std::string &filename);
  ~FileLogObserver();
  void update(int vehicleId, int location) override;
  void addLocalBest(int localBest, bool lastRow) override;
  void writeRoute(std::vector<Node> route) override;
  void writeDistance(std::vector<std::vector<double>> distance) override;
  void addGlobalBest(double globalBest) override;
};
```

Slika 22: Prikaz strukture podataka klase $FileLogObserver$ (Izvor: vlastita izrada)

Na slici 22 možemo vidjeti prikaz klase $FileLogObserver$ koja implementira klasu Observer te je ovdje implementiran uzorak dizajna observer za kreiranje pojedinih datoteka o kojemu ćemo kasnije više pisati. Klasa $FileLogObserver$ sadrži dvije privatne varijable. To su $logFile$ koja je tipa ofstream za upravljanje kreiranjem i zapisom kreirane datoteke te varijabla $filename$ tipa string koja označava sam naziv datoteke s pripadnom ekstenzijom. Od metoda možemo vidjeti konstruktor s parametrom naziva datoteke i destruktor koji će tu datoteku zatvoriti kako ne bi došlo do gubljenja podataka ili kako ne bi došlo do nemogućnosti otvaranja datoteke zbog korištenja datoteke od strane aplikacije. Ostale metode su kreirane kako bi pomogle kod unaprijed određenih formata datoteke. Metoda $update$ se koristi kako bi se svaki korak vozila mogao zapisati u datoteku te bi se mogla vidjeti sama konstrukcija rješenja pojedinog mrava. Metoda $addLocalBest$ služi za dodavanje najboljeg rezultata u iteraciji. Metoda $writeRoute$ zapisuje rutu pojedinog vozila u datoteku. Metoda $writeDistance$ zapisuje matricu udaljenosti u datoteku u eksplicitnom obliku. Metoda $addGlobalBest$ zapisuje najbolji globalni rezultat u datoteku. Ova klasa samo implementira nabrojane metode te je na nama da odlučimo o njihovom korištenju.

Klasa RandomInType

```cpp
#include <ctime>
#include <random>

template <typename T> class RandomInType {
private:
  std::mt19937 generator;

public:
  RandomInType(T seed = (T)std::time(0)) : generator(seed) {}
  void SetSeed(T seed) { generator.seed(seed); }
  T GetRandomInt(T min, T max) {
    std::uniform_int_distribution<T> distribution(min, max);
    return distribution(generator);
  }
  T next(T max) { return GetRandomInt(0, max); }
};
```

Slika 23: Prikaz strukture podataka klase $RandomInType$ (Izvor: vlastita izrada)

Na slici 23 možemo vidjeti prikaz klase $RandomInType$ koja sadrži metode dobivanja uniformne cjelobrojne distribucije, odnosno dobivanje slučajnog integer broja u rasponu. Možemo vidjeti kako konstruktor određuje seed za random s pomoću trenutnog vremena koji se kasnije koristi prilikom dobivanja random integer broja u rasponu. Metoda $GetRandomInt$ prima dva parametra koji predstavljaju donju i gornju granicu raspona iz kojega se dobiva slučajni broj. Metoda next s parametrom max je pojednostavljena $metodaGetRandomInt$ koja prima gornju granicu te donju postavlja na nula što znači da se odnosi na pozitivni raspon cjelobrojnog broja. Klasa sadrži privatni član pod nazivom generator, koji je instanca $std::mt19937$. To je Mersenne Twister pseudoslučajni generator koji proizvodi 32-bitne brojeve i ima veličinu stanja od 19937 bita. Ovaj generator se koristi za proizvodnju nasumičnih brojeva. Početna vrijednost generatora ključna je u generiranju nasumičnog broja jer određuje početno stanje generatora. Ako tijekom instanciranja klase nije navedena početna vrijednost, konstruktor koristi trenutno sistemsko vrijeme kao zadano početno mjesto.

Klasa LocalOptimization

```
class LocalOptimization {
private:
  RandomInType<unsigned int> randomsource;
  std::vector<char> dntLookBit;
  std::vector<Node> segment;

public:
  LocalOptimization(unsigned int seed = (unsigned int)std::time(0))
      : randomsource(seed) {}

  double Opt2_FIR(Ant &ant,
                  const std::vector<std::vector<double>> & distances, unsigned attempts);
  double Opt2_FIC(Ant &ant, std::vector<std::vector<double>> & distances);

  double reverseNodesBetweenVehicles(Ant &ant, unsigned maxAttempts,
      const std::vector<std::vector<double>> &distances);

  double addRemoveOneNode(Ant &ant, unsigned maxAttempts,
                          const std::vector<std::vector<double>> &distances);

  bool checkIfSolutionValid(std::vector<Node>& route, int capacity);
};
```

Slika 24: Prikaz strukture podataka klase $LocalOptimization$ (Izvor: vlastita izrada)

Na slici 24 možemo vidjeti prikaz klase $LocalOptimization$ koja se koristi za implementaciju lokalne optimizacije. U kasnijim poglavljima ćemo detaljnije objasniti lokalnu optimizaciju korištenu za problem usmjeravanja vozila. Klasa sadrži privatni član $randomsource$ koji je instanca klase $RandomInType < unsigned\ int >$. Ovaj se objekt koristi za generiranje nasumičnih brojeva tipa unsigned int. Klasa također sadrži dva privatna vektora, $dntLookBit$ i $segment$, koji se koriste u optimizacijskim algoritmima implementiranim u klasi. Konstruktor inicijalizira slučajni izvor početnom vrijednošću. Ako nije navedena početna vrijednost onda se koristi trenutno vrijeme sustava kao zadana početna vrijednost seed-a. Klasa još sadrži metode $Opt2\_FIR$ koja radi optimizaciju na pojedinom vozilo, $reverseNodesBetweenVehicles$ koja radi zamjenu klijenata među vozili, $addRemoveOneNode$ koja jednog vozilu briše klijenta kojega potom dodaje drugom vozilu te $checkIfSolutionValid$ koja nakon zamjere provjerava da li konstruirana rješenja poštuju ograničenja vozila zadana na početku programa. U ovoj klasi implementirano je nekoliko strategija za lokalnu optimizaciju primjenjivih za problem usmjeravanja vozila s ograničenim kapacitetom, a detaljnije na koji način je pojedina metoda implementirana će biti objašnjeno u sljedeći poglavljima ovog rada.

Klasa AntColonyOptimization

```
public:

  AntColonyOptimization(int numberOfAnts, int numberOfVehicles, int capacity,
                        int maxIteration, double deltaTau, double tauMax,
                        double tauMin, double rho, int S_FIR, int S_ADDREMOVE,
                        int S_REVERSE, int LO_FIR, int LO_ADDREMOVE,
                        int LO_REVERSE, int GO_FIR, int GO_ADDREMOVE,
                        int GO_REVERSE);
  void solve(std::vector<Node *> &route,
             std::vector<std::vector<double>> distances, double alpha, double beta) override;
  void attach(Observer* observer);
  void
  initializePheromoneTrails(std::vector<std::vector<double>> &pheromone_trails,
                            int num_cities);
  void initializeAnts(std::vector<Ant> &ants);

  void DisplayFinalPrice(double solutionCost, int iteration);

private:
  Node*
  choose_next_city(Node* currentNode, std::vector<Node *> &route,
                        const std::vector<std::vector<double>> &pheromoneTrails,
                   std::vector<std::vector<double>> distances, double alpha,
                   double beta, double u);
  void update_pheromones(double bestColonySolution, double bestEverSolution,
                    Ant bestLocalSolution, Ant bestGlobalSolution,
                    std::vector<std::vector<double>> &pheromoneTrails);
  std::vector<Observer*> observers; // List of observers
};
```

Slika 25: Prikaz strukture podataka klase $AntColonyOptimization$ (Izvor: vlastita izrada)

Na slici 25 možemo vidjeti prikaz klase $AntColonyOptimization$ koja se koristi za implementaciju mravljeg algoritma. Kako vidimo konstruktor ove klase prima puno parametara koji se unose na početku bitnih za pokretanje mravljeg algoritma uključujući broj mrava, broj vozila, kapacitet vozila, maksimalni broj ponavljanja i nekoliko parametara povezanih s ažuriranjem feromona kao što su $deltaTau$, $tauMax$, $tauMin$ i rho. Također vidimo metodu $initializePheromoneTrails$ koja služi za inicijalizaciju feromonskih tragova. Uz nju imamo također metodu $initalizeAnts$ za inicijalizaciju mravlje kolonije te imamo metode $choose\_next\_city$ za odabir idućeg klijenta pri konstrukciji rješenja i metodu $update\_pheromones$ koja služi za ažuriranje feromonskih tragova. U konstruktroru imamo parametre S_FIR, $S\_ADDREMOVE$, $S\_REVERSE$, LO_FIR, $LO\_ADDREMOVE$, $LO\_REVERSE, GO\_FIR$, $GO\_ADDREMOVE$, $GO\_REVERSE$ koji su parametri lokalne optimizacije koja će se koristiti. U ovoj klasi možemo metodom attach dodati observere odnosno datoteke u koje ćemo zapisivati određene statistike.

Klasa CandidateListOptimization

```
public:
  CandidateListOptimization(int numberOfAnts, int numberOfVehicles,
                            int capacity,
                        int maxIteration, double deltaTau, double tauMax,
                        double tauMin, double rho, int S_FIR, int S_ADDREMOVE,
                        int S_REVERSE, int LO_FIR, int LO_ADDREMOVE,
                        int LO_REVERSE, int GO_FIR, int GO_ADDREMOVE, int GO_REVERSE,
                         int candidateFavorites);
  void solve(std::vector<Node *> &route,
             std::vector<std::vector<double>> distances, double alpha,
             double beta) override;
  void attach(Observer *observer);
  void
  initializePheromoneTrails(std::vector<std::vector<double>> &pheromone_trails,
                            int num_cities);
  void initializeAnts(std::vector<Ant> &ants);

  void DisplayFinalPrice(double solutionCost, int iteration);

  void CalculateCandidateList(std::vector<std::vector<double>> &distances,
                              std::vector<Node *> &nodes);

  void removeVisitedFromFavoriteLists(int visitedNodeId,
                                      std::vector<Node*> &nodes);

  Node *findNodeById(std::vector<Node*> &nodes, int &id);

private:
  Node *choose_next_city(Node *currentNode, std::vector<Node *> &route,
                  const std::vector<std::vector<double>> &pheromoneTrails,
                  std::vector<std::vector<double>> distances, double alpha,
                  double beta, double u);
  void update_pheromones(double bestColonySolution, double bestEverSolution,
                         Ant bestLocalSolution, Ant bestGlobalSolution,
                         std::vector<std::vector<double>> &pheromoneTrails);
  std::vector<Observer *> observers; // List of observers
};
```

Slika 26: Prikaz strukture podataka klase $CandidateListOptimization$ (Izvor: vlastita izrada)

Na slici 26 možemo vidjeti prikaz klase $CandidateListOptimization$ koja se koristi za implementaciju mravljeg algoritma, odnosno druge strategije koja se primjenjuje za velike instance problema. Kako vidimo metode su skoro iste kao kod klase $AntColonyOptimization$. U ovoj klasi se nazale još tri dodatne metode: $CalculateCandidateList$, $removeVisitedFromFavoriteLists$ i $findNodeById$ koje služe za rad s listom favorita koja je karakteristična za ovu strategiju. Lista kandidata se koristi za velike instance problema kako bi se pomoglo mravima pri konstrukciji rješenja tako da svaki klijent ima listu svojih favorita, odnosno određen broj klijenta koji su mu najbliže.

Uzorci dizajna

Uzorci dizajna (eng. *design patterns*) predstavljaju neke od najboljih praksi koje su prilagodili iskusni programeri objektno-orijentiranog pristupa. Oni su kao unaprijed izrađeni nacrti koje možemo prilagoditi za rješavanje problema koji se ponavljaju u kodu. Uzorci dizajna predstavljaju opći koncept za rješavanje određenog problema te je potrebno slijediti detalje obrasca i implementirati rješenje koje odgovara programu [26]. U razvoju mravljeg algoritma koristili smo strategy i observer. Strategy smo koristili za odabir ponašanja objekta tijekom izvođenja dok smo observer koristili za komunikaciju s klasom koja implementira mravlji algoritam te joj preko observera pošaljemo datoteke koje će ona promatrati i ovisno o situaciji zapisivati određene stvari u pojedine datoteke.

Strategy

Uzorak dizajna strategy predlaže da klasu koja radi specifično na mnogo različitih načina izdvojimo u zasebne klase koje se nazivaju strategije. Razlog zbog kojeg smo koristili ovaj uzorak dizajna je mogućnost korištenja različitih mravljih algoritama u budućnosti te mijenjana pojedinosti trenutno implementiranog algoritma bez modificiranja postojećeg.

```
#include <vector>
#include "Node.h"
#include "Observer.h"

class RoutingStrategy {
public:
  virtual ~RoutingStrategy() {}
  virtual void solve(std::vector<Node *> &route,
                     std::vector<std::vector<double>> distances, double alpha, double beta) = 0;
  virtual void attach(Observer* observer) = 0;
};
```

Slika 27: Prikaz strukture podataka klase $RoutingStrategy$ (Izvor: vlastita izrada)

Na slici 27 možemo vidjeti prikaz strukture klase $RoutingStrategy$ koja ima dvije metode koje će biti univerzalne za svaku kreiranu strategiju. Prva metoda je solve koja kao parametar ima vektor tipa $Node$, matricu udaljenosti, alphu i betu, dok druga metoda ima mogućnost dodavanje datotečnih observera. Ovisno o vrsti strategije, možemo kreirati i različite statistike koje ćemo koristeći observer vrlo lako moći implementirati bez modifikacije postojećeg koda. Do sada smo definirali dvije

strategije. Prva je korištenjem običnog mravljeg algoritma s lokalnom optimizacijom dok je druga mravlji algoritam koji koristi listu favorita te se koristi za instance velikih problema.

```
enum StrategyType {
  ANT_COLONY_OPTIMIZATION,
  ACO_CANDIDATE_LIST
};

RoutingStrategy *createStrategy(StrategyType type, int numberOfAnts,
                                int numberOfVehicles, int capacity,
                                int maxIteration, double deltaTau,
                                double tauMax, double tauMin, double rho,
                                int S_FIR, int S_ADDREMOVE, int S_REVERSE,
                                int LO_FIR, int LO_ADDREMOVE, int LO_REVERSE,
                                int GO_FIR, int GO_ADDREMOVE, int GO_REVERSE,
                                int numberOfFavorits) {
  switch (type) {
  case ANT_COLONY_OPTIMIZATION:
    return new AntColonyOptimization(
        numberOfAnts, numberOfVehicles, capacity, maxIteration, deltaTau,
        tauMax, tauMin, rho, S_FIR, S_ADDREMOVE, S_REVERSE, LO_FIR,
        LO_ADDREMOVE, LO_REVERSE, GO_FIR, GO_ADDREMOVE, GO_REVERSE);
  case ACO_CANDIDATE_LIST:
    return new CandidateListOptimization(
        numberOfAnts, numberOfVehicles, capacity, maxIteration, deltaTau,
        tauMax, tauMin, rho, S_FIR, S_ADDREMOVE, S_REVERSE, LO_FIR,
        LO_ADDREMOVE, LO_REVERSE, GO_FIR, GO_ADDREMOVE, GO_REVERSE,
        numberOfFavorits);
  default:
    return nullptr;
  }
}
```

Slika 28: prikaz korištenja uzorka dizajna strategy (Izvor: vlastita izrada)

Na slici 28 možemo vidjeti prikaz kreiranja strategije ovisno o tipu. Imam enum vrijednost $StrategyType$ te metodu $createStrategy$ koja na osnovi tip strategije kreira objekt strategije koju želimo. Trenutno vidimo kako imamo dvije strategije koja kreira objekt $antColonyOptimization$ ili objekt $candidateListOptimization$ klase. Ovaj pristup je korišten kako bi se olakšala nadogradnja koda u budućnosti te omogućio dodatni razvoj mravljeg algoritma na postojećem kodu. Odabir strategije se događa pri pokretanju programa preko parametara.

Observer

Uzorak dizajna observer predlaže dodavanje mehanizma pretplate klasi izdavača kako bi se pojedinačni objekti mogli pretplatiti ili otkazati pretplatu na tok događanja koji dolaze do tog izvođača. Ovaj uzorak dizajna je korišten kako bi olakšali implementaciju pojedinačnih statistika za određene statistike. Na ovaj trenutno implementiran način možemo jednostavno dodati još neodređeno datoteka od kojih će svaka imati svoju jedinstvenu statistiku koju unaprijed definiramo. Kao što smo pokazali kod klase $observer$ i klase $fileLogObserver$ gdje imamo implementirane osnovne metode za upravljanje datotekom. Mi trenutno možemo jednostavno kreirati csv datoteku koju ćemo ju pretplatiti na određenu strategiju i unutar nje će ona bilježiti statistiku. Kada nam statistika ne bude trebala jednostavno samo maknemo pretplatu datoteke sa strategije i riješili smo problem.

```
FileLogObserver observer(createCSVName(
    params.instance.substr(0, params.instance.find_last_of(".")),
    params.alpha, params.beta, params.rho, params.delta_tau, params.tau_max,
    params.tau_min, params.iterations, params.maxVehicle, params.S_FIR,
    params.S_ADDREMOVE, params.S_REVERSE, params.LO_FIR,
    params.LO_ADDREMOVE, params.LO_REVERSE, params.GO_FIR,
    params.GO_ADDREMOVE, params.GO_REVERSE, params.numberOfAnts,
    params.candidateList, params.numberOfFavorits));

strategy->attach(&observer);

FileLogObserver solution(
    "SOLUTION_" +
    params.instance.substr(0, params.instance.find_last_of(".")) + "_" +
    getTimestamp() + ".txt");
strategy->attach(&solution);
```

Slika 29: prikaz kreiranja observera s datotekom i pretplata na strategiju (Izvor: vlastiti izvor)

Na slici 29 možemo vidjeti primjer kreiranja datoteke observera. Vidimo da se prvo kreira objekt observera s csv ekstenzijom datoteke, a potom objekt observera s txt ekstenzijom datoteke. Na primjeru također možemo vidjeti kako jednostavno možemo dodati observer u strategiju korištenjem metode $attach$. Ovaj pristup možemo objasniti da se svaka datoteka koja želi određene podatke mora prvo pretplatiti na strategiju te tek onda će biti obaviještena o rezultatima koje će moći zapisati. Ovakav pristup se činio dosta dobar zbog toga što su trebale različite statistike zapisane u različitim datotekama, a ni tada nismo bili sigurni jesu li to sve ili će prilikom eksperimentiranja biti potrebne još neke dodatne.

Parametri

S obzirom na to da se radi o konzolnoj aplikaciji, parametri su važan dio same implementacije mravljeg algoritma. Kako smo ranije spomenuli važnost određenih parametra za mravlji algoritam kao što su delta_tau, rho, tau_min, tau_max i drugih tako smo implementira da ti parametri imaju mogućnost unosa pri samom početku programa.

Slika 30: Prikaz svih parametara (Izvor: vlastiti izvor)

Na slici 29 vidimo prikaz svih parametara koje možemo uključiti i mijenjati prilikom pokretanja programa. Relase mode kreira ekstenziju .exe koju onda možemo pokretati u command promptu s prikazanim parametrima.
Primjer pokretanja jedne instance problema:

VRP-ACO.exe $-instance\ vrp-set-E-D22.txt\ -alpha\ 1.0\ -beta\ 2.0\ -rho\ 0.02\ -deltaTau\ 1\ -Ants\ 10\ -iterations\ 1000\ -maxVehicle\ 10\ -sFirAttempts\ 5\ -sAddRemoveAttempts\ 5\ -sReverseAttempts\ 5\ -loFirAttempts\ 10\ -loAddRemoveAttempts\ 10\ -loReverseAttempts\ 10\ -goFirAttempts\ 30\ -goAddRemoveAttempts\ 30\ -goReverseAttempts\ 30-candidateList\ 1-numberOfCandidate\ 7$

Možemo vidjeti kako se pokreće instanca problema $vrp\_set\_E\_D22.txt$ koja će parametar $alphe$ imati 1, parametar bete 2, parametar rho 0.02, parametar deltaTau 1. U slučaju da se tau_min i tau_max ne unesu ručno u program oni se automatski računaju prema formulama tako da će tau_min i tau_max ovisi o instanci problema te broju klijenta. Nadalje, možemo vidjeti kako se kolonije sastoji od 10 mrava te da će program u 1000 iteracija pokušati pronaći optimalno rješenje. Broj vozila koje će program imati na raspolaganju je 10. Svi ostali parametri vezani su za lokalnu optimizaciju svakog konstruiranog rješenja, najboljeg u iteraciji te globalno najbolje konstruiranog rješenja. Parametar $candidateList$ označava da ćemo koristiti strategiju s listom favorita te da će svaki klijent imati 7 favorita koje će prve gledati, a tek onda ostale vrhove. U slučaju da ne želimo koristiti listu favorita samo zanemarimo parametar $candidateList$ ili ga postavimo na 0.

Candidate List

Lista favorita je koncept koji igra ključnu ulogu u poboljšavanju izvođenja mravljeg algoritma. Popis kandidata je strateška komponenta koja značajno smanjuje vrijeme izvršenja algoritma. Ograničavanjem broja opcija koje mrav može izabrati, popis kandidata učinkovito sužava prostor pretraživanja, čime se ubrzava proces izračunavanja. Lista favorita služi kao vodič za mrave u algoritmu te u slučaju velikih instanci problema ograničava prostor pretraživanja što usmjerava mrave prema konstrukciji optimalnog rješenja [27]. Korištenjem liste favorita, mravi se sužava prostor pretraživanja te ga se navodi na korištenje optimalne rute te se također smanjuje vrijeme izvršavanja algoritma. U poglavlju eksperimenata ćemo detaljnije pojasniti prednosti korištenja ove strategije.

Mravlji algoritam za CVRP

```
volatile int numberOfNodes = 0;
volatile double numberOfNodesAVG = 0;
volatile double theta = 0;

std::string typeOfFile = readTypeFromFile(params.instance);
StrategyType type = ANT_COLONY_OPTIMIZATION;
if (params.candidateList)
  type = ACO_CANDIDATE_LIST;
if (typeOfFile == "EXPLICIT") {
  CVRPFileReader cvrp(params.instance);
  Problem problem;
  problem.capacity = (cvrp.*(&CVRPFileReader::capacity))();
  problem.vehicles = (cvrp.*(&CVRPFileReader::vehicles))();
  problem.createNodes((cvrp.*(&CVRPFileReader::demand))());
  problem.distances = (cvrp.*(&CVRPFileReader::edge_weight))();
  problem.type = (cvrp.*(&CVRPFileReader::type))();
  problem.convertDistances();
  if (!tauMinPresent) {
    double p = 0.05;
    numberOfNodes = problem.nodes.size() - 1;
    numberOfNodesAVG = numberOfNodes / 2;
    theta = pow((1.0 - pow(p, 1.0 / numberOfNodes)) /
                  ((numberOfNodesAVG - 1.0) * pow(p, 1.0 / numberOfNodes)),
              (1.0 / params.alpha));
    params.tau_min = params.tau_max * theta;
  }
  RoutingStrategy *strategy = createStrategy(
      type, params.numberOfAnts, problem.vehicles, problem.capacity, 1000,
      params.delta_tau, params.tau_max, params.tau_min, params.rho,
      params.S_FIR, params.S_ADDREMOVE, params.S_REVERSE, params.LO_FIR,
      params.LO_ADDREMOVE, params.LO_REVERSE, params.GO_FIR,
      params.GO_ADDREMOVE, params.GO_REVERSE, params.numberOfFavorits);

  if (strategy == nullptr) {
    cout << "STRATEGY DOES NOT EXIST!" << endl;
    return 1;
  }

  FileLogObserver observer(createCSVName(
      params.instance.substr(0, params.instance.find_last_of(".")),
      params.alpha, params.beta, params.rho, params.delta_tau, params.tau_max,
      params.tau_min, params.iterations, params.maxVehicle, params.S_FIR,
      params.S_ADDREMOVE, params.S_REVERSE, params.LO_FIR,
      params.LO_ADDREMOVE, params.LO_REVERSE, params.GO_FIR,
      params.GO_ADDREMOVE, params.GO_REVERSE, params.numberOfAnts,
      params.candidateList, params.numberOfFavorits));

  strategy->attach(&observer);

  FileLogObserver solution(
      "SOLUTION_" +
      params.instance.substr(0, params.instance.find_last_of(".")) + "_" +
      getTimestamp() + ".txt");
  strategy->attach(&solution);

  strategy->solve(problem.nodes, problem.doubleDistances, params.alpha,
                  params.beta);

  std::string fileNameDistances =
      "Distances_" +
      params.instance.substr(0, params.instance.find_last_of(".")) + ".csv";
  FileLogObserver distancesCSV(fileNameDistances);
  distancesCSV.writeDistance(problem.doubleDistances);

} else if (typeOfFile == "EUC_2D") {
```

Slika 31: Prikaz eksplicitno zadane matrice udaljenosti te pokretanja mravljeg algoritma (Izvor: vlastita izrada)

Na slici 31 možemo vidjeti početne postavke te odabir strategije za početak mravljeg algoritma. Svi uneseni parametri se učitavaju, računaju se vrijednosti tau_min

i tau_max te se kreira strategija ovisno o korištenju liste favorita. Također možemo vidjeti kako kreiramo tri datoteke za kasniju analizu, a to su: datoteka za finalno rješenje, datoteka za prikaz svih najboljih rješenja u koloniji u svakoj iteraciji te datoteka koja predstavlja matricu udaljenosti.

```
} else if (typeOfFile == "EUC_2D") {
  setEReader setE(params.instance);
  if (!tauMinPresent) {
    double p = 0.05;
    numberOfNodes = setE.get_nodes().size() - 1;
    numberOfNodesAVG = numberOfNodes / 2;
    theta = pow((1.0 - pow(p, 1.0 / numberOfNodes)) /
                ((numberOfNodesAVG - 1.0) * pow(p, 1.0 / numberOfNodes)),
              (1.0 / params.alpha));
    params.tau_min = params.tau_max * theta;
  }
  RoutingStrategy *strategy = createStrategy(
      type, params.numberOfAnts, params.maxVehicle, setE.get_capacity(),
      params.iterations, params.delta_tau, params.tau_max, params.tau_min,
      params.rho, params.S_FIR, params.S_ADDREMOVE, params.S_REVERSE,
      params.LO_FIR, params.LO_ADDREMOVE, params.LO_REVERSE, params.GO_FIR,
      params.GO_ADDREMOVE, params.GO_REVERSE, params.numberOfFavorits);

  if (strategy == nullptr) {
    cout << "STRATEGY DOES NOT EXIST!" << endl;
    return 1;
  }
  std::vector<Node *> route;
  for (nodeCoord var : setE.get_nodes()) {
    route.push_back(new Node(var.id - 1, var.demand));
  }

  FileLogObserver observer(createCSVName(
      params.instance.substr(0, params.instance.find_last_of(".")),
      params.alpha, params.beta, params.rho, params.delta_tau, params.tau_max,
      params.tau_min, params.iterations, params.maxVehicle, params.S_FIR,
      params.S_ADDREMOVE, params.S_REVERSE, params.LO_FIR,
      params.LO_ADDREMOVE, params.LO_REVERSE, params.GO_FIR,
      params.GO_ADDREMOVE, params.GO_REVERSE, params.numberOfAnts,
      params.candidateList, params.numberOfFavorits));

  strategy->attach(&observer);

  FileLogObserver solution(
      "SOLUTION_" +
      params.instance.substr(0, params.instance.find_last_of(".")) + "_" +
      getTimestamp() + ".txt");
  strategy->attach(&solution);

  strategy->solve(route, setE.get_distance_matrix(), params.alpha,
                  params.beta);

  std::string fileNameDistances =
      "Distances_" +
      params.instance.substr(0, params.instance.find_last_of(".")) + ".csv";
  FileLogObserver distancesCSV(fileNameDistances);
  distancesCSV.writeDistance(setE.get_distance_matrix());

} else {
  cout << "Unknown reader type of matrix" << endl;
}
```

Slika 32: Prikaz implicitno zadane matrice udaljenosti te pokretanja mravljeg algoritma (Izvor: vlastita izrada)

Na slici 32 možemo vidjeti istu stvar s pripremom algoritma samo u slučaju korištenja implicitno zadanih koordinata matrice udaljenosti. Obe slike prikazuju učitavanje parametara, odabir strategije te početak kreiranja mravljeg algoritma.

```
void AntColonyOptimization::update_pheromones(
    double bestColonySolution, double bestEverSolution, Ant bestLocalSolution,
    Ant bestGlobalSolution, std::vector<std::vector<double>> &pheromoneTrails) {

  for (auto &row : pheromoneTrails) {
    for (auto &tau : row) {
      tau = (1 - rho) * tau;
      if (tau < tauMin) {
        tau = tauMin;
      }
    }
  }

  for (Vehicle vehicle : bestGlobalSolution.getVehicles()) {
    int previousNode = 0;
    for (Node node : vehicle.getRoute()) {
      pheromoneTrails[previousNode][node.getId()] =
          pheromoneTrails[previousNode][node.getId()] + deltaTau;
      if (pheromoneTrails[previousNode][node.getId()] >= tauMax)
        pheromoneTrails[previousNode][node.getId()] = tauMax;
      previousNode = node.getId();
    }
  }
}
```

Slika 33: prikaz metode za ažuriranje feromonskih tragova (Izvor: vlastita izrada)

Na slici 33 možemo vidjeti implementaciju metode za ažuriranje feromonskih tragova gdje vidimo kako prvi dio služi za „isparavanje feromonskih tragova“ tako da se svaki koeficijent feromonskih tragova množi s $1 - rho$, odnosno kada bismo rho pretvorili u postotke onda bi se nakon svake iteracije feromonski koeficijent smanjio za taj postotak. U drugom dijelu možemo vidjeti pojačavanje feromonskih tragova za najbolje trenutno globalno rješenje tako da mu se dodaje $deltaTau$. U ovoj metodi također možemo primijetiti kako se gornja i donja granica feromonskih tragova uzimaju u obzir te tako nije moguće smanjiti feromonski koeficijent ako je on jednak tau_min i nije moguće povećati koeficijent ako je on jednak tau_max. U literaturu možemo pronaći da se najčešće koriste dvije vrste nagrađivanja, a to su: nagrađivanje najboljeg trenutnog rješenja i nagrađivanje najboljeg rješenja u iteraciji [28,29].

```
Node *AntColonyOptimization::choose_next_city(
    Node *currentNode, std::vector<Node *> &route,
    const std::vector<std::vector<double>> &pheromoneTrails,
    std::vector<std::vector<double>> distances, double alpha, double beta,
    double load) {

  double u = load / capacity;
  return RandomGenerator::selectNextNodeWithHeuristicCompleted(
      route, pheromoneTrails, distances, currentNode, alpha, beta, u);
}
```

Slika 34: prikaz metode za odabir idućeg klijenta bez lista favorita (Izvor: vlastita izrada)

Na slici 34 možemo vidjeti prikaz implementacije metode za odabir idućeg klijenta u slučaju korištenja strategije ant colony optimization, odnosno strategije koja ne koristi listu favorita. Možemo vidjeti na slici 35 da se ova metoda razlikuje kod korištenja strategije liste favorita gdje se prvo gleda jesu li svi klijenti iz liste favorita posjećeni te ako nisu onda se idući klijent bira među preostalim ne posjećenim klijentima, a ako jesu onda se odabire idući klijent iz cijelog skupa neposjećenih klijenta.

```
Node *CandidateListOptimization::choose_next_city(
    Node *currentNode, std::vector<Node *> &route,
    const std::vector<std::vector<double>> &pheromoneTrails,
    std::vector<std::vector<double>> distances, double alpha, double beta,
    double load) {

  double u = load / capacity;
  std::vector<Node *> candidateList;
  for (int candidateNode : currentNode->candidateList) {
    Node* node = findNodeById(route,candidateNode);
    if (!node->isRouted())
      candidateList.push_back(node);
  }
  if (candidateList.size() > 0) {
    return RandomGenerator::selectNextNodeWithHeuristicCompleted(
        candidateList, pheromoneTrails, distances, currentNode, alpha, beta, u);
  } else {
    return RandomGenerator::selectNextNodeWithHeuristicCompleted(
        route, pheromoneTrails, distances, currentNode, alpha, beta, u);
  }

}
```

Slika 35: prikaz metode za odabir idućeg klijenta sa listom favorita (Izvor: vlastita izrada)

Na slici 36 možemo vidjeti implementaciju metode $selectNextNodeWithHeuristicCompleted$ gdje vidimo kako se uz jačinu feromonskih tragova koristi heuristika za odabir bližih klijenta. U komentarima nam se nalazi pokušaj korištenja dvije ete gdje bi vozila pokušala birati klijente koji su bliži odredištu ovisno o popunjenosti kapacitete pojedinog vozila, no testiranjem se pokazalo kako nam druga eta ne daje željene rezultate te smo odustali od toga i primijenili smo druge metode za poboljšavanje mravljeg algoritma. Obe strategije koriste ovu metodu samo što su slučaju korištenja liste favorita šaljemo isključivo klijente favorite te se radi računanje ete i uključivanje feromonskih tragova samo na favoritima dok u drugom slučaju radimo računanje na skupu svih neposjećenih klijenata neovisno jesu li favoriti ili ne. Prvo stvara vektor klijenata koji još nisu posjećeni. To se radi iteracijom preko svih klijenata i provjerom je li klijent posjećen ili ne. Za svaki neposjećeni vrh ili neposjećenog klijenta se izračunava umnožak feromonskog traga i heurističkih informacija. Trag feromona (tau) dohvaća se iz matrice pheromoneTrails, a heurističke informacije (ETAij)

izračunavaju se kao inverzna vrijednost udaljenosti između trenutnog čvora i neposjećenog čvora, odnosno trenutnog čvora i početnog čvora. Potencije alfa i beta su parametri funkcije. Nakon toga se vektor izračunatih feromonski i heurističkih vrijednosti koristi za stvaranje diskretne distribucije koja generira slučajni indeks te on odabire klijenta na tom indeksu iz neposjećenih klijenata. Ova metoda implementira probabilističku selekciju sljedećeg klijenta favorizirajući čvorove s većim feromonskih koeficijentima i manjim udaljenostima.

```
Node* RandomGenerator::selectNextNodeWithHeuristicCompleted(
    std::vector<Node *> &nodes,
    const std::vector<std::vector<double>> &pheromoneTrails,
    const std::vector<std::vector<double>> &distances, Node* currentNode,
    double alpha, double beta, double u) {

  std::vector<Node *> unvisitedNodes;
  int betaMAX = 2;
  int betamin = 1;

  for (auto &node : nodes) {
    if (node->getId() != currentNode->getId() && !node->isRouted()) {
      unvisitedNodes.push_back(node);
    }
  }

  if (unvisitedNodes.size() == 0)
    return nodes[0];

  std::vector<double> products;
  for (const auto &node : unvisitedNodes) {
    double tau = pheromoneTrails[currentNode->getId()][node->getId()];
    double ETAij = 1.0 /
                distances[currentNode->getId()]
                         [node->getId()];
    double ETAjs = 1.0 / distances[currentNode->getId()][0];
    double beta2 = 4 * (betaMAX - betamin) * pow(u, 2) -
                   4 * (betaMAX - betamin) * u + betaMAX;
    //products.push_back(pow(tau, alpha) * pow(ETAij, beta) * pow(ETAjs, beta2));
    products.push_back(pow(tau, alpha) * pow(ETAij, beta));
  }


  std::random_device rd;
  std::mt19937 gen(rd());


  std::discrete_distribution<> d(products.begin(), products.end());

  int randomIndex = d(gen);

  return unvisitedNodes[randomIndex];
}
```

Slika 36: prikaz metode odabira idućeg klijenta korištenjem feromona i heuristike (Izvor: vlastita izrada)

Na slici 37 vidimo prikaz cijelog algoritma bez korištenja liste favorita gdje vidimo da se u početku inicijaliziraju feromonski tragovi tako da su jednaki gornjoj granici, odnosno tau_max te dolazi do for petlje koja koristi parametarski unos za iteracije i nakon toga kolonija kreće s kreiranjem rješenja. Svaki mrav u početku postavlja svoju poziciju u početnu točku te dohvaća vozila na raspolaganju. Nakon toga slijedi odabir neposjećenih klijenata te se dodaju vozilu dok god ima kapaciteta za ispunjenje. U slučaju kada nema mjesta, onda se odabire iduće vozilo te on posjećuje odabranog klijenta. Nakon što mrav konstruira rješenje, radi se evaluacija kako bi se rješenje moglo dalje uspoređivati.

```
void AntColonyOptimization::solve(std::vector<Node *> &route,
                                  std::vector<std::vector<double>> distances,
                                  double alpha, double beta) {

  Ant bestLocalSolution;
  Ant bestGlobalSolution;
  LocalOptimization LO;
  std::vector<std::vector<double>> pheromone_trails;
  initializePheromoneTrails(pheromone_trails, route.size());
  int bestSolutionIteration = 0;
  for (int iteration = 0; iteration < maxIterations; ++iteration) {
    int mrav = 1;
    initializeAnts(ants);
    bestColonySolution = DBL_MAX;
    for (Ant &ant : ants) {
      for (Node *node : route) {
        node->setIsRouted(false);
      }

      Node *current_city = route[0];
      current_city->setIsRouted(true);
      std::vector<Vehicle> &vehicles = ant.getVehicles();
      int vehicleNumber = 0;
      Vehicle *currentVehicle = &vehicles[vehicleNumber];
      bool allRouted = false;

      Node *depot = route[0];
      while (!allRouted) {
        allRouted = true;
        for (Node *node : route) {
          if (!node->isRouted()) {
            allRouted = false;
            break;
          }
        }
        depot->setIsRouted(true);
        Node *next_city =
            choose_next_city(current_city, route, pheromone_trails, distances,
                             alpha, beta, currentVehicle->getLoad());
        if (currentVehicle->CheckIfFits(next_city->getDemand())) {
          if (next_city->getId() != 0)
            currentVehicle->AddNode(*next_city);
        } else {
          vehicleNumber++;
          currentVehicle = &vehicles[vehicleNumber];
          currentVehicle->AddNode(*next_city);
        }
        current_city = next_city;
      }

      double solutionValue = ant.evaluate(distances);
```

Slika 37: prikaz mravljeg algoritma bez liste favorita (Izvor: vlastita izrada)

Na slici 38 vidi prikaz implementacije mravljeg algoritma uz korištenje liste favorita, odnosno prikaz druge strategije. Možemo primijetiti kako je većina ista osim početnog dijela gdje se poziva metoda za izračun liste favorita. Lista favorita se koristi kod odabira neposjećenih klijenata.

```
void CandidateListOptimization::solve(
    std::vector<Node *> &route, std::vector<std::vector<double>> distances,
    double alpha, double beta) {

  Ant bestLocalSolution;
  Ant bestGlobalSolution;
  LocalOptimization LO;
  std::vector<std::vector<double>> pheromone_trails;
  CalculateCandidateList(distances, route);
  initializePheromoneTrails(pheromone_trails, route.size());

  int bestSolutionIteration = 0;
  for (int iteration = 0; iteration < maxIterations; ++iteration) {
    int mrav = 1;

    initializeAnts(ants);
    bestColonySolution = DBL_MAX;
    for (Ant &ant : ants) {
      for (Node *node : route) {
        node->setIsRouted(false);
      }

      Node *current_city = route[0];
      current_city->setIsRouted(true);
      std::vector<Vehicle> &vehicles = ant.getVehicles();
      int vehicleNumber = 0;
      Vehicle *currentVehicle = &vehicles[vehicleNumber];
      bool allRouted = false;

      Node *depot = route[0];
      while (!allRouted) {
        allRouted = true;
        for (Node *node : route) {
          if (!node->isRouted()) {
            allRouted = false;
            break;
          }
        }
        depot->setIsRouted(true);
        Node *next_city =
            choose_next_city(current_city, route, pheromone_trails, distances,
                             alpha, beta, currentVehicle->getLoad());

        if (currentVehicle->CheckIfFits(next_city->getDemand())) {
          if (next_city->getId() != 0)
            currentVehicle->AddNode(*next_city);
        } else {

          vehicleNumber++;
          currentVehicle = &vehicles[vehicleNumber];
          currentVehicle->AddNode(*next_city);
        }
        current_city = next_city;
        //removeVisitedFromFavoriteLists(next_city->getId(), route);
      }

      double solutionValue = ant.evaluate(distances);
```

Slika 38: prikaz mravljeg algoritma s listom favorita (Izvor: vlastita izrada)

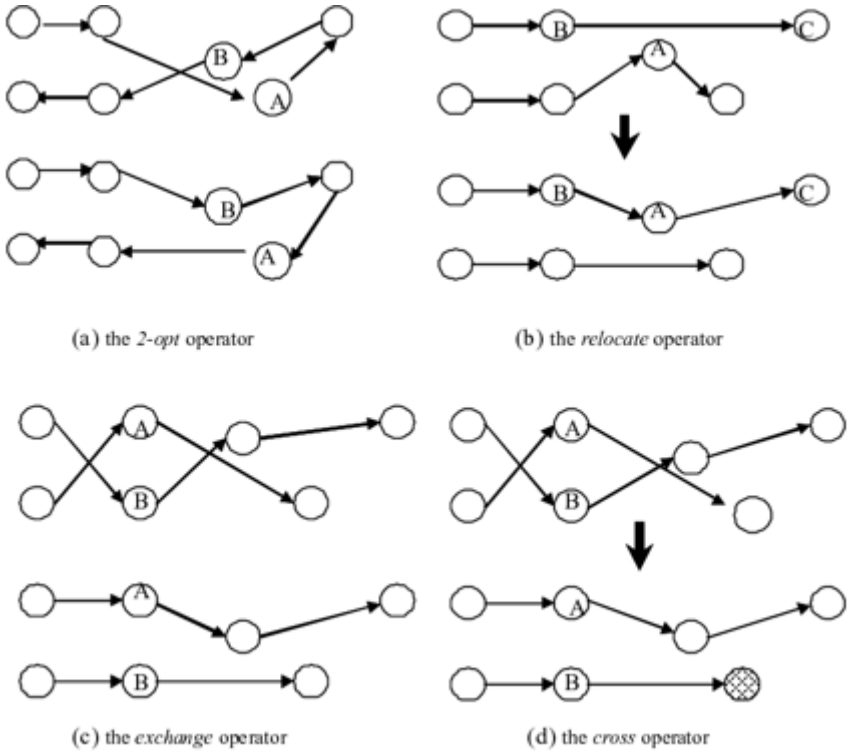
```
void CandidateListOptimization::CalculateCandidateList(
    std::vector<std::vector<double>> &distances, std::vector<Node *> &nodes) {
  int numNodes = nodes.size();

  for (int i = 0; i < numNodes; ++i) {
    std::vector<std::pair<double, int>> distIdPairs;
    for (int j = 0; j < numNodes; ++j) {
      if (i != j) {
        distIdPairs.push_back(std::make_pair(distances[i][j], nodes[j]->getId()));
      }
    }

    sort(distIdPairs.begin(), distIdPairs.end());

    for (int k = 0; k < std::min(candidateFavorites, (int)distIdPairs.size()); ++k) {
      nodes[i]->candidateList.push_back(distIdPairs[k].second);
    }
  }
}
```

Slika 39: prikaz metode za izračun liste favorita (Izvor: vlastita izrada)

Na slici 39 možemo vidjeti implementaciju za izračun lista favorita gdje se za svaki čvor stvara vektor parova gdje se svaki par sastoji od udaljenosti od trenutnog čvora i identifikacijskog broja čvora klijenta kojega gledamo. Nakon toga se radi sortiranje prema udaljenosti te se korištenjem parametra za broj favorita dodaje svakog klijentu id klijenta i tako se stvara njegov popis favorita.

Lokalna optimizacija

Lokalna optimizacija je tehnika koja je koristi za poboljšavanje performansi algoritma te njegovo usmjeravanje prema boljim rješenjima. Često se mravlji algoritam zna zarobiti u lokalnim optimumima što znači da je rješenje optimalno unutar susjednog skupa mogućih rješenja, ali nije optimalno općenito. Upravo zbog toga koristimo lokalnu optimizaciju koja usavršava rješenja koja su pronašli mravi pomažući im da izbjegnu lokalne optimume te se usmjere na potragu prema globalnom optimumu. Jedan pristup je korištenje lokalnog algoritma pretraživanja kao što je 2-opt [30]. Lokalna optimizacija funkcionira tako da iterativno unose male promjene u rješenje kao što je zamjena dva grada na ruti te prihvaćaju promjene ako je ukupni trošak rješenja manji od onog prije promjene.



Slika 40: Prikaz mogućnosti strategija lokalne optimizacije (Izvor: Vincent W.L. Tam, link)

Na slici 40 možemo vidjeti određene strategije pristupa lokalnoj optimizaciji. 2-opt je operator unutar rute koji mijenja dio rute brisanje lukova i zamjenom s bilo koja 2 luka kako bi se reformirala ruta. Operator pomicanja premješta posjet s pozicije na jednoj ruti na drugu poziciju na istoj ili drugoj ruti. Razmjena mijenja 2 posjeta istim ili različitim rutama, dok operator prijelaza mijenja krajnje točke 2 rute [31]. U slučaju našeg problema usmjeravanja vozila s ograničenim kapacitetom implementirali smo tri vrste strategija lokalne optimizacije. Prva je 2-opt na svakoj ruti pojedinog vozila gdje se zamjenom dva klijenta istog vozila pokušava dobiti ukupno bolje rješenje. Druga je zamjenom dva klijenta između dva vozila s određenim brojem pokušaja gdje se također pokušava dobiti bolje rješenje. Treće je brisanje jednog klijenta na jednom vozilu te dodavanje u rutu drugog vozila uz provjeru kapaciteta. U nastavku ćemo objasniti detaljnije njihovu implementaciju, ali kombiniranjem svih triju strategija algoritam prije pronalazi optimalno rješenje.

```
double LocalOptimization::addRemoveOneNode(
    Ant &ant, unsigned maxAttempts,
    const std::vector<std::vector<double>> &distances) {
  if (maxAttempts == 0) {
    return -1;
  }
  std::vector<Vehicle> &solutions = ant.getVehicles();
  if (ant.getVehicles().size() < 2) {
    return NULL;
  }
  double totalOriginalCost = ant.evaluate(distances);
  size_t vehicleIndex1, vehicleIndex2, nodeIndex1, nodeIndex2, randomIndex;
  double totalNewCost;
  bool improved = false;

  while (maxAttempts) {
    maxAttempts--;

    do { ... } while (solutions[vehicleIndex1].getRoute().empty());

    do { ... } while (vehicleIndex2 == vehicleIndex1 ||
            solutions[vehicleIndex2].getRoute().empty());

    nodeIndex1 =
        randomsource.next(solutions[vehicleIndex1].getRoute().size() - 1);

    nodeIndex2 =
        randomsource.next(solutions[vehicleIndex2].getRoute().size() - 1);

    bool selectNodeIndex1 = randomsource.next(2);
    int randomNodeIndex = selectNodeIndex1 ? nodeIndex1 : nodeIndex2;

    if (selectNodeIndex1) {
      randomIndex =
          randomsource.next(solutions[vehicleIndex2].getRoute().size() - 1);
      solutions[vehicleIndex2].addNodeAtIndex(
          solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1], randomIndex);
      solutions[vehicleIndex1].removeNode(
          solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1]);
    } else { ... }

    totalNewCost = 0;
    for (auto &solution : ant.getVehicles()) {
      solution.totalCost = ant.evaluateVehicle(solution.getRoute(), distances);
    }
    for (auto &solution : solutions) {
      totalNewCost += ant.evaluateVehicle(solution.getRoute(), distances);
    }

    if (totalNewCost >= totalOriginalCost ||
        !checkIfSolutionValid(solutions[vehicleIndex1].getNonConstRoute(),
                              solutions[vehicleIndex1].getCapacity()) ||
        !checkIfSolutionValid(solutions[vehicleIndex2].getNonConstRoute(),
                              solutions[vehicleIndex2].getCapacity())) {
      if (selectNodeIndex1) {
        solutions[vehicleIndex1].addNodeAtIndex(
            solutions[vehicleIndex2].getNonConstRoute()[randomIndex],
            nodeIndex1);
        solutions[vehicleIndex2].removeNodeAtIndex(randomIndex);
      } else {
        solutions[vehicleIndex2].addNodeAtIndex(
            solutions[vehicleIndex1].getNonConstRoute()[randomIndex],
            nodeIndex2);
        solutions[vehicleIndex1].removeNodeAtIndex(randomIndex);
      }
    } else {
      improved = true;
      totalOriginalCost = totalNewCost;
    }
  }
  if (!improved) { ... }

  return totalOriginalCost;
```

Slika 41: prikaz implementacije lokalne optimizacije dodavanja-oduzimanja čvora među vozilima (Izvor: vlastita izrada)

Na slici 41 možemo vidjeti prikaz implementacije strategije lokalne optimizacije gdje se jednom vozilu oduzima klijent kojeg posjećuje na svojoj ruti te se dodjeljuje drugom klijentu. Prvo se odabiru dva različita vozila koja su korištenja za konstrukciju

rješenja te se potom slučajnim odabirom bira jedan klijent iz njihove dvije rute koji se potom briše vozilu koji ga trenutno posjećuje i dodjeljuje se drugom slučajno odabranom vozilu. Nakon toga slijedi promjera jesu li obje rute zadovoljile kapacitet svojih vozila, odnosno je li kapacitet svakog klijenta manji od kapaciteta pojedinog vozila te se također provjerava je li novokreirano rješenje bolje od prethodnog. U slučaju da je bolje rješenje se zapisuje, a ako nije onda se rute vraćaju na početni raspored.

```
double LocalOptimization::reverseNodesBetweenVehicles(
    Ant &ant, unsigned maxAttempts,
    const std::vector<std::vector<double>> &distances) {
  if (maxAttempts == 0) {
    return -1;
  }

  std::vector<Vehicle> &solutions = ant.getVehicles();
  if (ant.getVehicles().size() < 2) {
    return NULL;
  }

  double totalOriginalCost = ant.evaluate(distances);
  size_t vehicleIndex1, vehicleIndex2, nodeIndex1, nodeIndex2;
  double totalNewCost;
  bool improved = false;

  while (maxAttempts) {
    maxAttempts--;

    do {
      vehicleIndex1 = randomsource.next(solutions.size() - 1);
    } while (solutions[vehicleIndex1].getRoute().empty());

    do {
      vehicleIndex2 = randomsource.next(solutions.size() - 1);
    } while (vehicleIndex2 == vehicleIndex1 ||
             solutions[vehicleIndex2].getRoute().empty());

    nodeIndex1 =
        randomsource.next(solutions[vehicleIndex1].getRoute().size() - 1);
    nodeIndex2 =
        randomsource.next(solutions[vehicleIndex2].getRoute().size() - 1);

    Node temp = solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1];
    solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1] =
        solutions[vehicleIndex2].getNonConstRoute()[nodeIndex2];
    solutions[vehicleIndex2].getNonConstRoute()[nodeIndex2] = temp;

    totalNewCost = 0;
    for (auto &solution : ant.getVehicles()) {
      solution.totalCost = ant.evaluateVehicle(solution.getRoute(), distances);
    }
    for (auto &solution : solutions) {
      totalNewCost += ant.evaluateVehicle(solution.getRoute(), distances);
    }

    if (totalNewCost >= totalOriginalCost ||
        !checkIfSolutionValid(solutions[vehicleIndex1].getNonConstRoute(),
                              solutions[vehicleIndex1].getCapacity()) ||
        !checkIfSolutionValid(solutions[vehicleIndex2].getNonConstRoute(),
                              solutions[vehicleIndex2].getCapacity())) {
      Node temp2 = solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1];
      solutions[vehicleIndex1].getNonConstRoute()[nodeIndex1] = temp;

      solutions[vehicleIndex2].getNonConstRoute()[nodeIndex2] = temp2;
    } else {
      improved = true;
      totalOriginalCost = totalNewCost;
    }
  }

  if (!improved) {
    return -1.0;
  }
  return totalOriginalCost;
}
```

Slika 42: prikaz implementacije lokalne optimizacije zamjene dva čvora među vozilima (Izvor: vlastita izrada)

Na slici 42 možemo vidjeti prikaz implementacije lokalne optimizacije gdje se koristi strategija zamjene dva klijenta među dva vozila. Kao i u prethodnoj implementaciji ovdje se također odabiru slučajno dva vozila koja su korištena pri konstrukciji rješenja te se nakon toga iz svake rute odabire slučajni klijent. Nakon toga se mijenjaju ti klijenti i vrši se provjera je li rješenje u skladu s ograničenjem kapaciteta te ako je, je li rješenje bolje od onog prije zamjene. Ako je rješenje bolje onda se pamti i vraća kao uspješno odrađen korak lokalne optimizacije, a ako rješenje nije bolje samo se rute vraćaju na početni raspored.

```
double
LocalOptimization::Opt2_FIR(Ant &ant,
                            const std::vector<std::vector<double>> &distances,
                            unsigned maxAttempts) {
  if (maxAttempts == 0) {
    return -1;
  }
  double originalCost = ant.evaluate(distances);
  unsigned int maxindex;
  double dii1_jj1, dij_i1j1;
  size_t i, j, i1, j1;
  bool improved = false;
  while (maxAttempts) {
    maxAttempts--;
    for (Vehicle &solution : ant.getVehicles()) {
      if (solution.getRoute().empty())
        continue;

      solution.totalCost = ant.evaluateVehicle(solution.getRoute(), distances);
      auto route = solution.getRoute();
      maxindex = solution.getRoute().size() - 1;
      i = randomsource.next(maxindex);
      i1 = (i + 1) % solution.getRoute().size();
      j = (i + 2 + randomsource.next(solution.getRoute().size() - 4)) %
          solution.getRoute().size();
      j1 = (j + 1) % solution.getRoute().size();

      dii1_jj1 = distances[route[i].getId()][route[i1].getId()] +
                 distances[route[j].getId()][route[j1].getId()];
      dij_i1j1 = distances[route[i].getId()][route[j].getId()] +
                 distances[route[i1].getId()][route[j1].getId()];
      if (dii1_jj1 > dij_i1j1) {
        solution.reverse(i1, j);

        if (ant.evaluate(distances) >= originalCost) {
          solution.reverse(i1, j);
        } else {
          improved = true;
          solution.totalCost -= (dii1_jj1 - dij_i1j1);
          originalCost = ant.evaluate(distances);
          for (auto &solution : ant.getVehicles()) {
            solution.totalCost =
                ant.evaluateVehicle(solution.getRoute(), distances);
          }
          solution.totalCost =
              ant.evaluateVehicle(solution.getRoute(), distances);
        }
      }
    }
  }

  double totalNewCost = 0;
  for (auto &solution : ant.getVehicles()) {
    totalNewCost += ant.evaluateVehicle(solution.getRoute(), distances);
  }

  return improved ? totalNewCost : -1;
}
```

Slika 43: prikaz implementacije 2-opt lokalne optimizacije (Izvor: vlastita izrada)

Na slici 43 možemo vidjeti prikaz 2-opt lokalne optimizacije gdje se koristi strategija zamjene dva klijenta na istoj ruti pojedinog vozila. Za razliku od prethodnih implementacija ovdje se prolazi kroz cijelo rješenje, odnosno pokušava se optimizirati svaka ruta vozila. U slučaju da jedna zamjena smanji trošak, rješenje se odmah pamti. Kao i kod prethodnih strategija koristimo broj pokušaja kako lokalna optimizacija ne bi ušla u beskonačnu petlju koji definiram s pomoću parametara na početku programa. Sada kad smo vidjeli sve tri strategije lokalne optimizacije u nastavku rada ćemo pokazati njihovo korištenje nad različitim rješenjima.

```
Ant helperAntS = ant;
if (LO.reverseNodesBetweenVehicles(helperAntS, S_REVERSE, distances) !=
    -1.0) {
  ant = helperAntS;
}

Ant helperAnt2S = ant;
if (LO.addRemoveOneNode(helperAnt2S, S_ADDREMOVE, distances) != -1.0) {
  ant = helperAnt2S;
}

Ant helperAnt3S = ant;
if (LO.Opt2_FIR(helperAnt3S, distances, S_FIR) != -1.0) {
  ant = helperAnt3S;
}

solutionValue = ant.evaluate(distances);

if (solutionValue < bestColonySolution) {
  bestLocalSolution = ant;
  bestColonySolution = solutionValue;
}
if (solutionValue < bestEverSolution) {
  bestSolutionIteration = iteration+1;
  bestGlobalSolution = ant;
  bestEverSolution = solutionValue;
}
```

Slika 44: prikaz korištenja lokalne optimizacije za svako konstruirano rješenje (Izvor: vlastita izrada)

Na slici 44 možemo vidjeti kako koristimo sve tri strategije te se one primjenjuju za svako konstruirano rješenje kako bi se sama kolonija pokušala optimizirati. Parametri $S\_REVERSE$, $S\_ADDREMOVE$ i S_FIR predstavlja broj pokušaja za svaku od tih strategija. U poglavlju s eksperimentima ćemo raditi na podešavanju parametara kako bi dobili optimalno rješenje u kraćem vremenskom roku.

```
Ant helperLocalAnt = bestLocalSolution;
if (LO.reverseNodesBetweenVehicles(helperLocalAnt, LO_REVERSE, distances) !=
    -1.0) {
 bestLocalSolution = helperLocalAnt;
 bestColonySolution = bestLocalSolution.evaluate(distances);
}


Ant helperLocalAnt2 = bestLocalSolution;
if (LO.addRemoveOneNode(helperLocalAnt2, LO_ADDREMOVE, distances) != -1.0) {
 bestLocalSolution = helperLocalAnt2;
 bestColonySolution = bestLocalSolution.evaluate(distances);
}

Ant helperLocalAnt3 = bestLocalSolution;
if (LO.Opt2_FIR(helperLocalAnt3, distances, LO_FIR) != -1.0) {
 bestLocalSolution = helperLocalAnt3;
 bestColonySolution = bestLocalSolution.evaluate(distances);
}

Observer *observer = observers[0];
if (iteration != maxIterations - 1) {
 observer->addLocalBest(bestLocalSolution.evaluate(distances), false);
} else {
 observer->addLocalBest(bestLocalSolution.evaluate(distances), true);
}

if (bestColonySolution < bestEverSolution) {
 bestSolutionIteration = iteration;
 bestEverSolution = bestColonySolution;
 bestGlobalSolution = bestLocalSolution;
}
```

Slika 45: prikaz korištenja lokalne optimizacije za najbolje rješenje iteracije (Izvor: vlastita izrada)

Na slici 45 možemo vidjeti kako koristimo sve tri strategije te se one primjenjuju za najbolje rješenje u iteraciji. Na ovaj način najbolje rješenje iteracije koje ne mora biti najbolje globalno rješenje se može poboljšati te tako dobiti optimalno rješenje u ranijim iteracijama. Ovdje također možemo vidjeti kako se nakon izvršenja lokalne optimizacije za najbolje rješenje iteracije vrijednost zapisuje u observer, odnosno observer dobiva obavijest o promjeni te ju ažurira u datoteku. Na ovaj način kreiramo statistiku rješenja kroz iteracije.

```
Ant helperAnt = bestGlobalSolution;
if (LO.reverseNodesBetweenVehicles(helperAnt, GO_REVERSE, distances) !=
    -1.0) {
  bestSolutionIteration = iteration;
  bestGlobalSolution = helperAnt;
  bestEverSolution = bestGlobalSolution.evaluate(distances);
}


Ant helperAnt2 = bestGlobalSolution;
if (LO.addRemoveOneNode(helperAnt2, GO_ADDREMOVE, distances) != -1.0) {
  bestSolutionIteration = iteration;
  bestGlobalSolution = helperAnt2;
  bestEverSolution = bestGlobalSolution.evaluate(distances);
}

Ant helperAnt3 = bestGlobalSolution;
if (LO.Opt2_FIR(helperAnt3, distances, GO_FIR) != -1.0) {
  bestSolutionIteration = iteration;
  bestGlobalSolution = helperAnt3;
  bestEverSolution = bestGlobalSolution.evaluate(distances);
}
```

Slika 46: prikaz korištenja lokalne optimizacije za najbolje globalno rješenje (Izvor: vlastita izrada)

Na slici 46 možemo vidjeti kako koristimo sve tri strategije te se one primjenjuju za najbolje globalno rješenje. Kao i kod prethodnih rješenja, koristimo sve tri strategije s brojem pokušaja za globalne strategije lokalne optimizacije koje nakon svake iteracije pokušavaju poboljšati najbolje globalno rješenje.

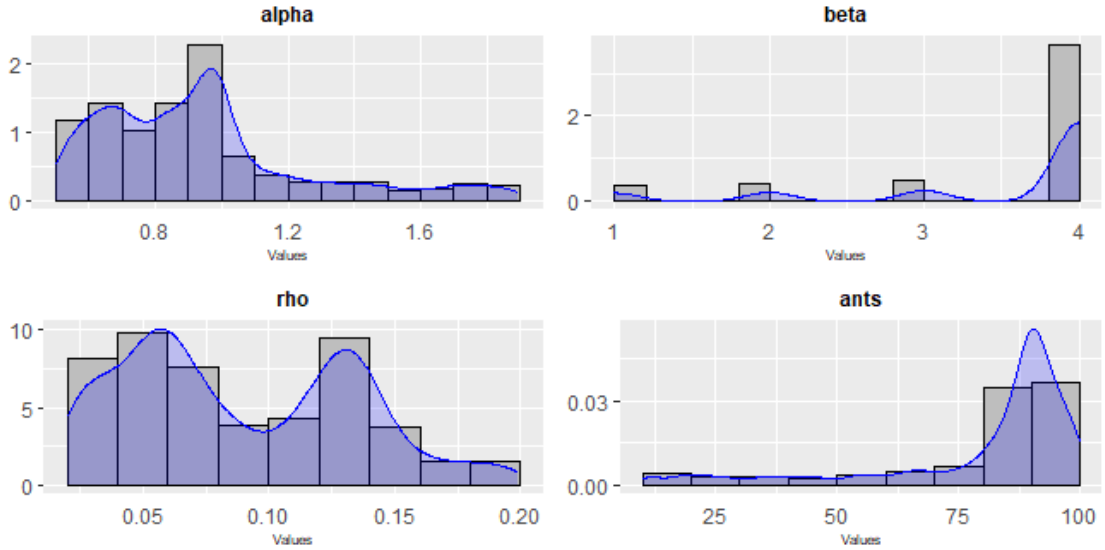
7. Eksperimenti

Različiti eksperimenti su provedeni na problemu usmjeravanja vozila s ograničenim kapacitetom koristeći prethodno opisani algoritam. U ovom poglavlju, rezultati tih eksperimenata su prikazani i analizirani. Problem usmjeravanja vozila predstavlja jedan od najvećih izazova u logističkom sektoru jer je to složeni problem koji odražava stvarne probleme s kojima se suočavaju mnogi logistički odjeli u poduzećima. Postoji mnogo varijacija ovog problema koje se razlikuju po broju klijenata, kapacitetu svakog vozila, zahtjevima klijenata i broju dostupnih vozila. Poznato je da performanse algoritama računalne inteligencije, pa tako i algoritama ACO, ovise o dobrim postavkama algoritamskih parametara. Zbog toga je prije ostalih eksperimenata prvo proveden postupak pronalaska dobrih parametara s pomoću alata iRace,

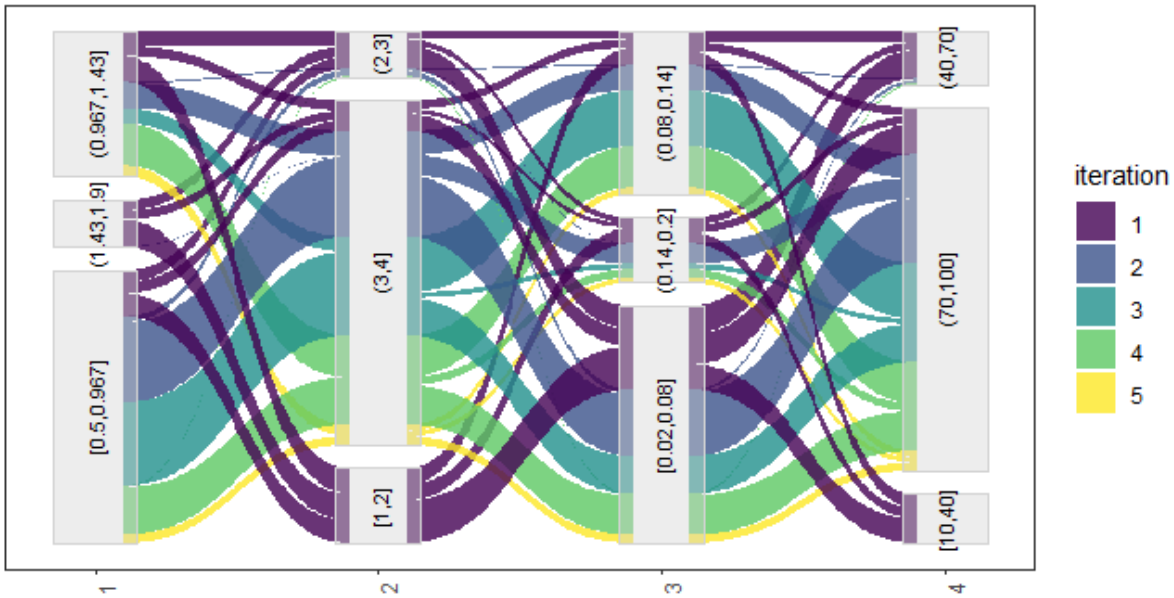
Rezultati iRace

iRace je aplikacija koja služi za automatsku konfiguraciju optimizacijskih algoritama. U prethodnim poglavljima spominjali smo razne ulazne parametre koje ovaj program uzima u obzir te pronalazi najprikladnije postavke s obzirom na skup instanci problema. Implementiran je u jezik R te se jednostavno može pokretati na windows operacijskim sustavima te linux i MacOs.

Na slici 47 možemo vidjeti prikaz kretanja četiri parametra za instancu problema A-n63-k9. Ovdje možemo vidjeti kako je za veći broj čvorova najbolje imati oko 80 mrava u koloniji, betu postaviti na 4, alfu na 1, a rho otprilike na 0.07. Važno je napomenuti kako smo prilikom izračunavanja ovih parametara uzeli količinu resursa u obzir pa smo tako imali izvedenu varijablu koja se zvala iterationsxAnts koja je predstavljala broj iteracija pomnožen s brojem mrava u koloniji te smo tako dobili da iRace program ima ograničen broj resursa te da izračuna koliko nam je zaista najbolje uzeti mrava u koloniji.

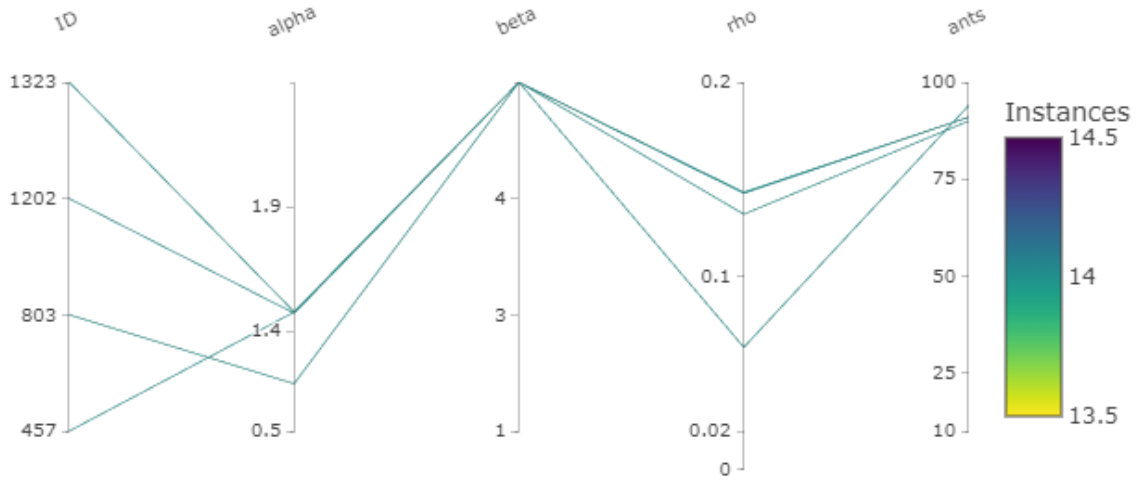


Slika 47: rezultati iRace za instancu problema A-n63-k9 (Izvor: slika izrađena pomoću iRace alata)

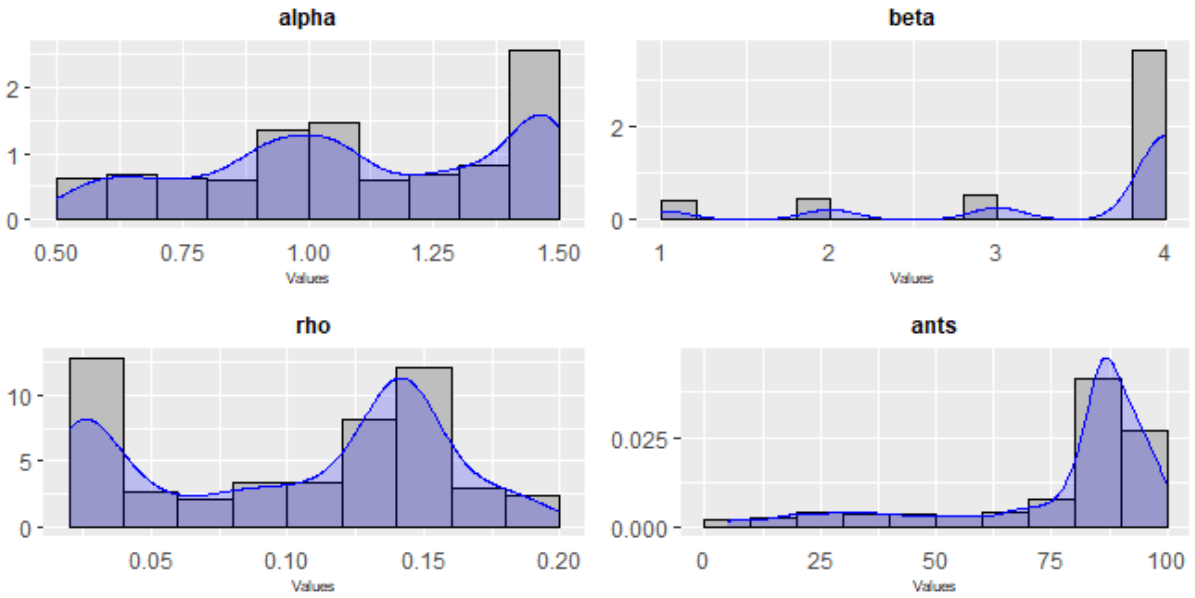


Slika 48: Rplot za instancu problema A-n63-k9 (Izvor: slika izrađena pomoću iRace alata)

Na slici 48 možemo vidjeti rplot za instancu problema A-63-k9 na kojoj možemo vidjeti intervala pojedinih parametara. Tako možemo vidjeti da interval parametra alphe [0.5,0.967] daje najbolje rezultate kada je beta iz intervala [3,4], rho gornji dio intervala [0.02,0.08] te broj mrava u koloniji u intervalu [70,100]. Na slici 49 možemo vidjeti isto to samo prikazano na drukčiji način.

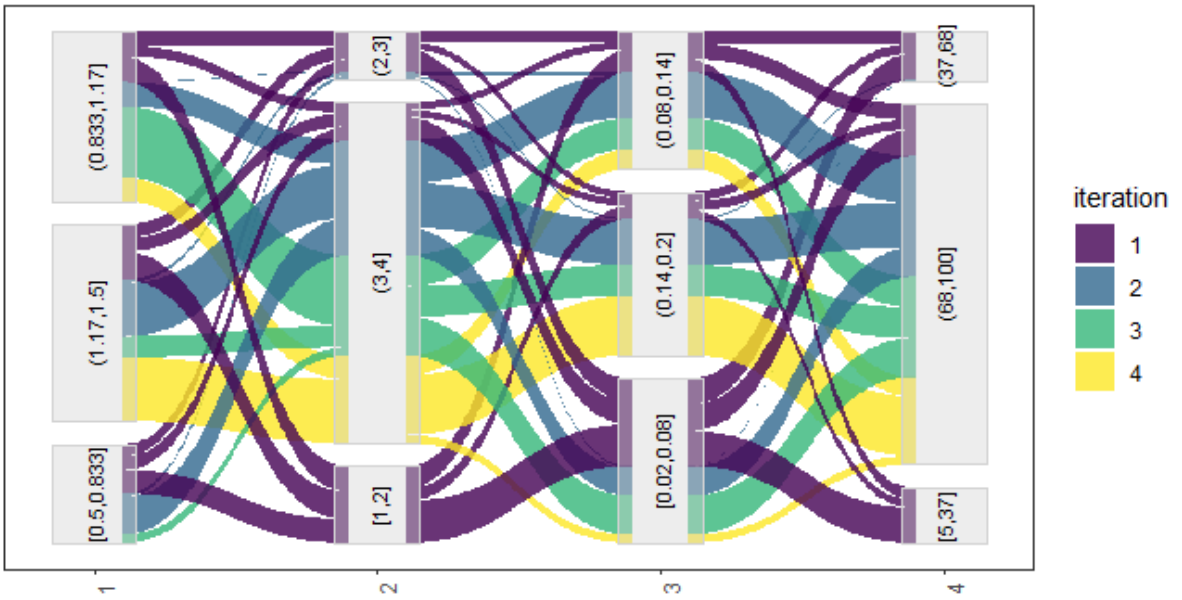


Slika 49: Rplot za instancu problema A-n63-k9 drugi primjer (Izvor: slika izrađena pomoću iRace alata)



Slika 50: rezultati iRace za instancu problema P-n40-k5 (Izvor: slika izrađena pomoću iRace alata)

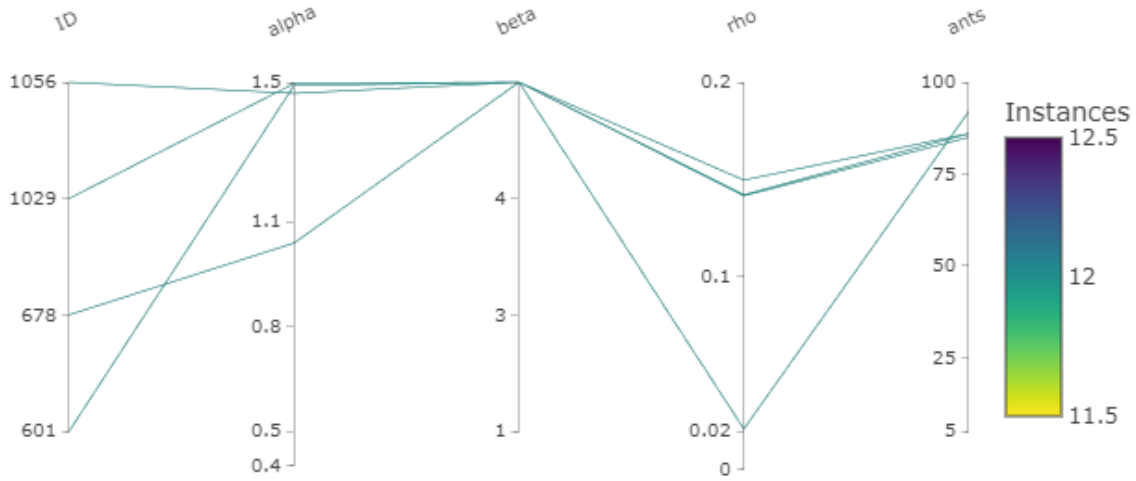
Na slici 50 možemo vidjeti prikaz kretanja četiri parametra za instancu problema P-n40-k5. Ovdje možemo vidjeti kako je za manji broj čvorova najbolje imati oko 90 do 100 mrava u koloniji, betu postaviti na 4, alfu na 1.5, a rho otprilike na 0.15.



Slika 51: Rplot za instancu problema P-n40-k5 (Izvor: slika izrađena pomoću iRace alata)

Kao i kod prethodne instance, na slici 51 možemo vidjeti rplot za instancu problema P-n40-k5. Tako možemo vidjeti da interval parametra alphe [1.17,1.5] daje najbolje rezultate kada je beta iz intervala [3,4], rho gornji dio intervala [0.14,0.2] te broj mrava

u koloniji u intervalu [68,100]. Na slici 52 možemo vidjeti najbolje odabrane vrijednosti parametara za instancu problema P-n40-k5. S obzirom na to da smo u iRace alatu analizirali samo četiri parametra, ostale smo jasno definirali kao konstantne parametre. Primjer iterationsxAnts je uvijek definiran kao 50000 što bi značilo da postoji 50000 resursa, a onda nam je iRace pokazao za manje problema kako je u redu da uzmemo oko 80 mrava što dovodi do 625 iteracija, a za veće da uzmemo oko 100 mrava što dovodi do 500 iteracija. Eksperimenti s lokalnim optimizacijama su rađeni na različite načine. Listu favorita ili kandidacijsku listu smo odlučili koristiti za problema veće od 100 čvorova, dok za manje instance problema ona neće biti korištena.



Slika 52: Rplot za instancu problema P-n40-k5 drugi primjer (Izvor: slika izrađena pomoću iRace alata)

Popis scenarija

U tablici 3 možemo vidjeti prikaz svih instanci problema koji su se pokretali na implementiranom mravljem algoritmu. U nastavku je svaka instanca detaljnije proučena. Za 4 instance implementirani algoritam je pronašao optimalno rješenje, za instancu problema E-n30-k3 je pronašao rješenje koje je bolje od znanstveno prikazanog rješenja, ostale instance su unutar 10 posto od optimalnog rješenja te imamo instancu M-n101-k10 koja je izvan 10 posto od optimalnog rješenja. Svaku ovu situaciju ćemo u nastavku detaljnije priučiti. Svi scenariji su pokretani na prijenosnom računalu Asus Strix koji ima procesor Intel core i7-7700HQ, 2.80GHz. Računalo ima 16 GB RAM-a te 64-bitni operacijski sustav.

Tablica 3: prikaz svih scenarija eksperimenata

Instanca problema	LO_FIR	LO_ADDREMOVE	LO_REVERSE	Ants	Najbolje dobiveno rješenje	Najbolje optimalno znanstveno rješenje	Iteracije	Alpha	Rho	Beta	Candidate list	List of favorits	Vrijeme izvršavanja eksperimenta
A-n32-k5	100	100	100	80	786	784	625	1.5	0.15	4	false	-	25h 10min 16sec
A-n55-k9	100	100	100	80	1074	1073	625	1.5	0.15	4	false	-	39h 33min 26sec
A-n63-k9	0	0	0	80	1672	1616	625	1	0.07	4	false	-	41min 44sec
A-n63-k9	10	10	10	80	1635	1616	625	1	0.07	4	false	-	7h 5min 58sec
A-n63-k9	0	10	10	80	1686	1616	625	1	0.07	4	false	-	6h 10min 30sec
A-n63-k9	10	0	10	80	1678	1616	625	1	0.07	4	false	-	4h 48min 27sec
A-n63-k9	10	10	0	80	1643	1616	625	1	0.07	4	false	-	4h 47min 34sec
A-n80-k10	100	100	100	100	1816	1763	500	1	0.1	4	true	20	68h 59min 5sec
B-n35-k5	100	100	100	80	956	955	625	1.5	0.15	4	false	-	34h 5min 13sec

Instanca problema	LO_FIR	LO_ADDREMOVE	LO_REVERSE	Ants	Najbolje dobiveno rješenje	Najbolje optimalno znanstveno rješenje	Iteracije	Alpha	Rho	Beta	Candidate list	List of favorits	Vrijeme izvršavanja eksperimenta
B-n45-k5	20	20	20	80	766	751	625	1.5	0.15	4	false	-	3h 2min
B-n50-k7	100	100	100	80	755	741	625	1.5	0.15	4	false	-	43h 52min 52sec
B-n78-k10	100	100	100	80	1240	1221	625	1.5	0.15	4	false	-	30h 46min 28sec
E-n22-k4	30	30	30	80	375	375	625	1.5	0.15	4	false	-	2h 13min 28sec
E-n30-k3	10	10	10	80	505	534	625	1.5	0.15	4	false	-	1h 15min 31sec
E-n33-k4	20	20	20	80	839	835	625	1.5	0.15	4	false	-	2h 11min 9sec
E-n51-k5	20	20	20	80	521	521	625	1.5	0.15	4	false	-	3h 16min 20sec
E-n101-k14	10	10	10	100	1168	1067	500	1	0.1	4	true	20	9h 54min 50sec
F-n45-k4	10	10	10	80	744	724	625	1.5	0.15	4	false	-	1h 59min 1sec
F-n72-k4	20	20	20	100	243	237	500	1	0.1	4	true	20	8h 31min 47sec
M-n101-k10	20	20	20	100	900	820	500	1	0.1	4	true	20	18h 03min 4sec
M-n101-k10	0	20	20	100	922	820	500	1	0.1	4	true	20	15h 14min 18sec
M-n101-k10	20	0	20	100	903	820	500	1	0.1	4	true	20	11h 51min 1sec
M-n101-k10	20	20	0	100	897	820	500	1	0.1	4	true	20	11h 54min 9sec
P-n16-k8	100	100	0	80	450	450	625	1.5	0.15	4	false	-	22h 25min 47sec
P-n16-k8	10	10	10	80	450	450	625	1.5	0.15	4	false	-	1h 17min 24sec

Instanca problema	LO_FIR	LO_ADDREMOVE	LO_REVERSE	Ants	Najbolje dobiveno rješenje	Najbolje optimalno znanstveno rješenje	Iteracije	Alpha	Rho	Beta	Candidate list	List of favorits	Vrijeme izvršavanja eksperimenta
P-n16-k8	0	0	0	80	450	450	625	1.5	0.15	4	false	-	3min 24sec
P-n16-k8	100	100	100	80	450	450	625	1.5	0.15	4	false	-	23h 22min 20sec
P-n16-k8	0	100	100	80	450	450	625	1.5	0.15	4	false	-	20h 55min 28sec
P-n16-k8	100	0	100	80	450	450	625	1.5	0.15	4	false	-	22h 28min
P-n40-k5	100	100	100	80	458	458	625	1.5	0.15	4	false	-	37h 30min 30sec
P-n40-k5	0	100	100	80	459	458	625	1.5	0.15	4	false	-	36h 29min 43sec
P-n40-k5	100	0	100	80	461	458	625	1.5	0.15	4	false	-	20h 40min 36sec
P-n40-k5	100	100	0	80	459	458	625	1.5	0.15	4	false	-	20h 36min 41sec

(Izvor: vlastita izrada)

Rezultati analize

U tablici 6 se nalazi prikaz rezultata koje ćemo detaljno obraditi u sljedećim poglavljima. Možemo vidjeti prikaz svih scenarija te rezultati percentila 10, medijana te percentila 90.

Tablica 4: prikaz P10, mediana i P90 za svaki provedeni scenarij

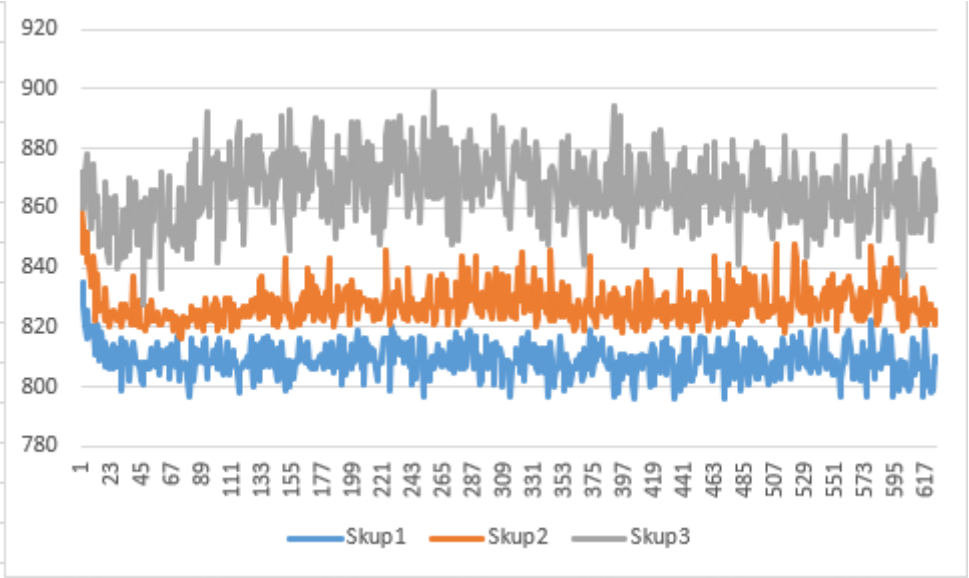
Instanca problema	LO_FIR	LO_ADDREMOVE	LO_REVERSE	P10	Medijan	P90
A-n32-k5	100	100	100	796	816	828
A-n55-k9	100	100	100	1086	1129	1165
A-n63-k9	0	0	0	1695	1775	1851
A-n63-k9	10	10	10	1683	1776	1850
A-n63-k9	0	10	10	1714	1787	1868
A-n63-k9	10	0	10	1697	1766	1841
A-n63-k9	10	10	0	1694	1763	1845
A-n80-k10	100	100	100	1834	1888	1954
B-n35-k5	100	100	100	962	972	985
B-n45-k5	20	20	20	786	808	832
B-n50-k7	100	100	100	774	792	809
B-n78-k10	100	100	100	1262	1295	1328
E-n22-k4	30	30	30	375	376	377
E-n30-k3	10	10	10	507	526	553
E-n33-k4	20	20	20	841	850	865
E-n51-k5	20	20	20	535	560	581
E-n101-k14	10	10	10	1188	1241	1279
F-n45-k4	10	10	10	755	791	830
F-n72-k4	20	20	20	244	252	266
M-n101-k10	20	20	20	927	987	1031

M-n101-k10	0	20	20	951	1006	1052
Instanca problema	LO_FIR	LO_ADDREMOVE	LO_REVERSE	P10	Medijan	P90
M-n101-k10	20	0	20	929	974	1044
M-n101-k10	20	20	0	920	979	1040
P-n16-k8	100	100	0	450	450	450
P-n16-k8	10	10	10	450	450	450
P-n16-k8	0	0	0	450	450	450
P-n16-k8	100	100	100	450	450	450
P-n16-k8	0	100	100	450	450	450
P-n16-k8	100	0	100	450	450	450
P-n40-k5	100	100	100	461	471	480
P-n40-k5	0	100	100	472	484	498
P-n40-k5	100	0	100	466	474	484
P-n40-k5	100	100	0	462	472	485

(Izvor: vlastita izrada)

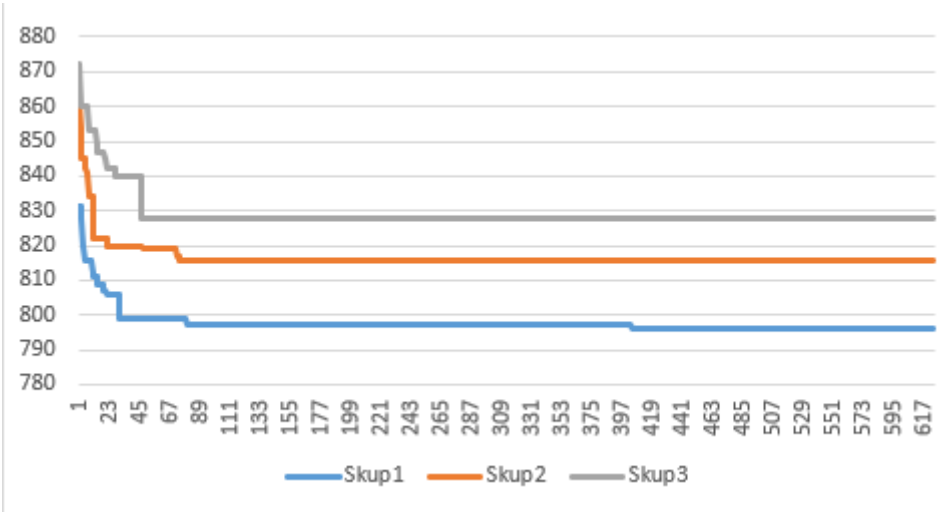
A-n32-k5

Instanca problema A-n32-k5 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 25 sati, 10 minuta i 16 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 786 što je za samo dvije jedinice lošije od optimalnog. Možemo reći kako je implementirani algoritam s postavkama odradio dobar posao. Vjerojatno bi kroz malo veći broj pokušaja lokalne optimizacije došli do optimalnog rješenja, ali zbog vremenski zahtjevnog eksperimenta smo zadovoljni dobivenim rezultatima.



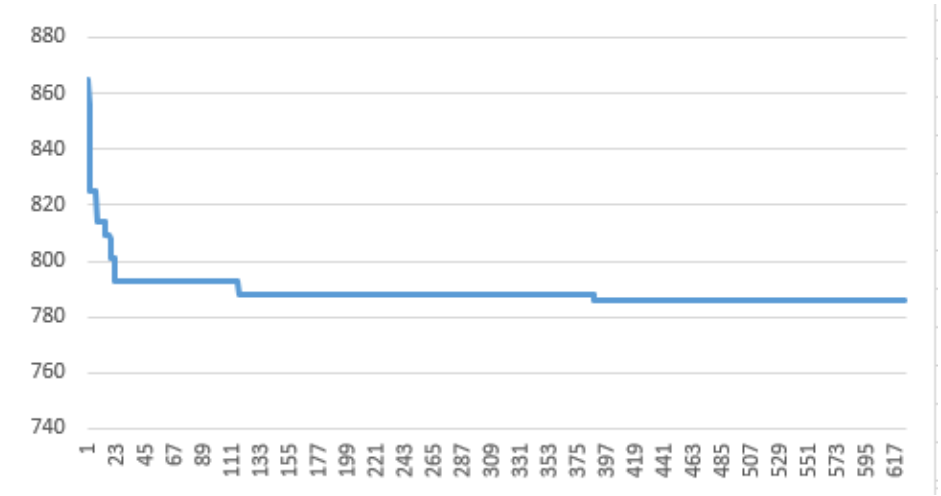
Slika 53: prikaz vršne, prosječne i donje performanse za instancu A-n32-k5 (Izvor: izrada u Excelu)

Na slici 53 možemo vidjeti grafički prikaz gdje skup 1 predstavlja percentile kroz iteracije za one koji postignu 10% najboljih rezultata, skup 2 predstavlja prosječnu vrijednost po iteraciji dok skup 3 predstavlja percentile funkciju kroz iteracije za one koji postignu 90% najboljih rezultata. Možemo vidjeti kako na početku su rezultati lošiji u odnosu na one u kasnijim iteracijama.



Slika 54: prikaz vršne, prosječne i donje performanse za instancu A-n32-k5 (Izvor: izrada u Excelu)

Na slici 54 možemo vidjeti konstrukciju rješenja kroz iteracije te prikaz vršne, prosječne i donje performanse za instancu te možemo vidjeti kako se kroz iteracije krivulja smanjuje što dovodi do zaključka kako mravi kroz iteracije konstruiraju bolja rješenja. S obzirom na to da je ovo manja instanca problema, možemo vidjeti kako u kasnijim iteracijama mravi sporije konstruiraju bolje rješenje što je logično zbog toga što u tim iteracijama feromonski tragovi imaju veliki utjecaj te mravi nemaju veliku mogućnost istraživanja suboptimalnih puteva.

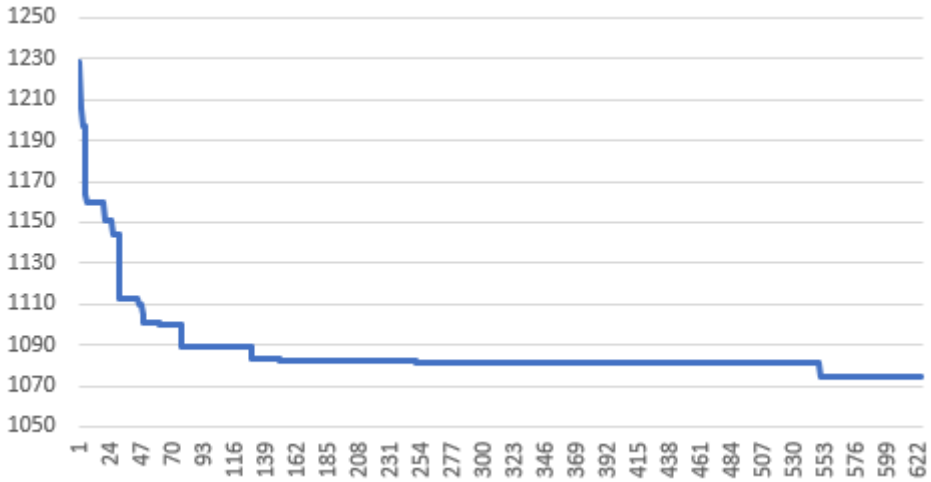


Slika 55: prikaz instance algoritma koji je dao najbolje rješenje za instancu A-n32-k5 (Izvor: izrada u Excelu)

Na slici 55 možemo vidjeti prikaz algoritma za instancu ovog problema koje je dao najbolje rješenje te također možemo vidjeti kako se u početku vidi veliki napredak dok u kasnijim fazama zbog utjecaja feromonskih tragova dolazi do sporijeg pronalaska boljih rješenja kroz iteracije.

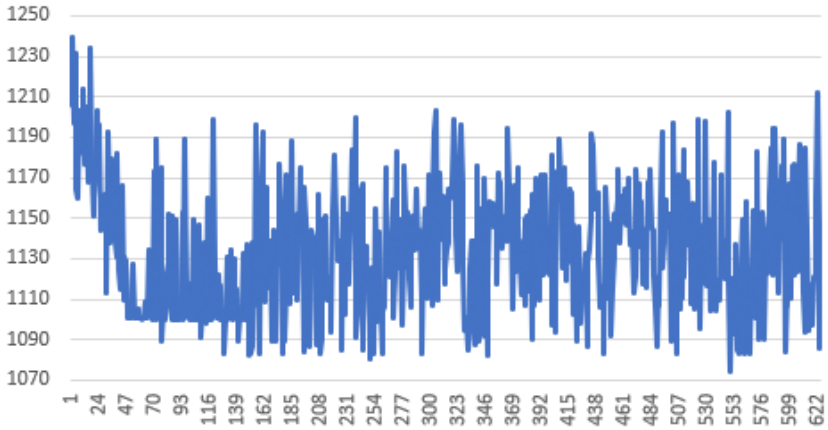
A-n55-k9

Instanca problema A-n55-k9 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 39 sati, 33 minuta i 26 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 1074 što je za samo jednu jedinice lošije od optimalnog.



Slika 56: prikaz instance algoritma koji je dao najbolje rješenje za instancu A-n55-k9 (Izvor: izrada u Excelu)

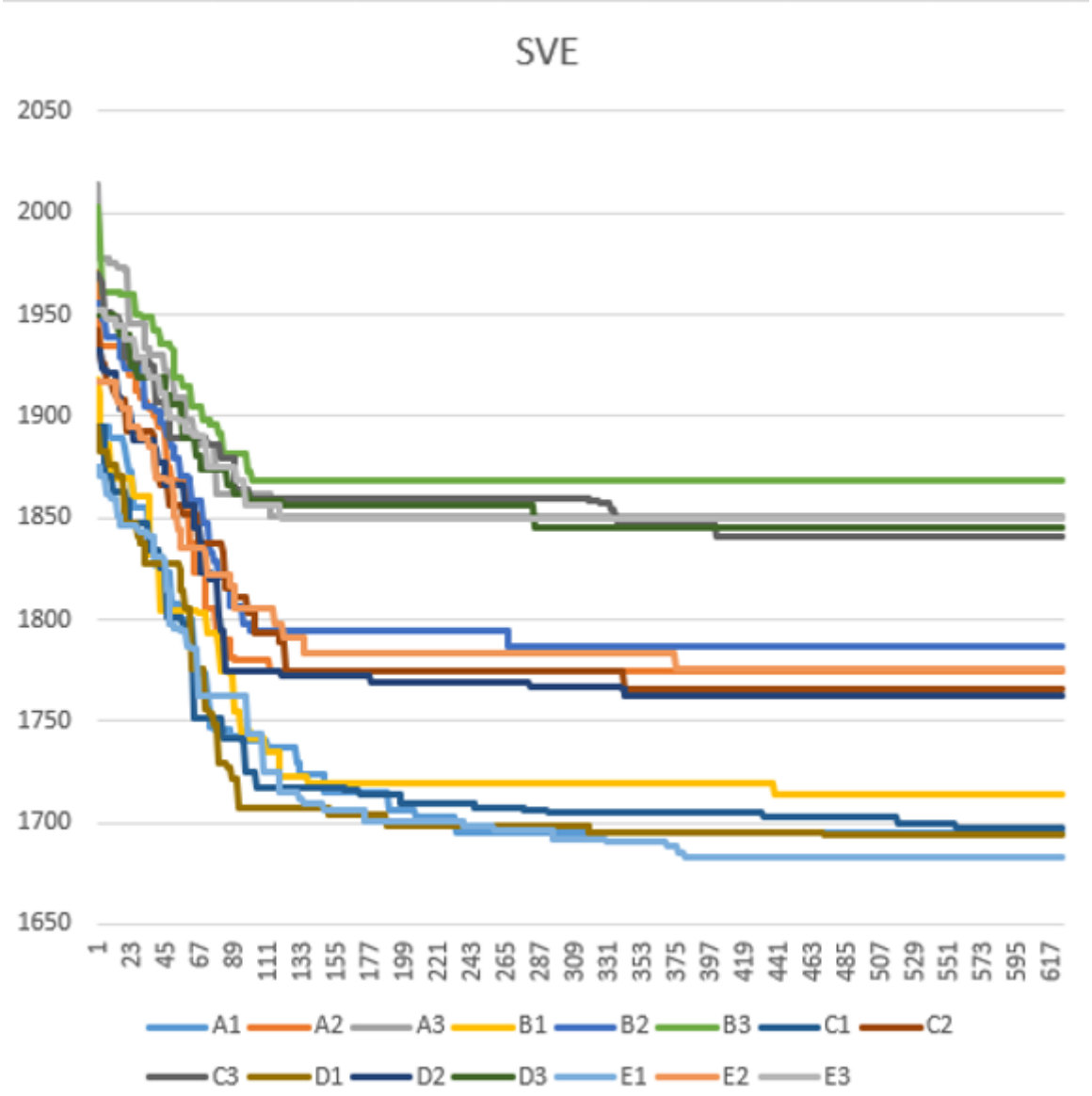
Na slici 55 i slici 56 možemo vidjeti sličnu situaciju kao i kod prethodne instance problema. U početnim iteracijama vidimo veliki napredak u konstrukciji rješenja dok u kasnijim fazama zbog utjecaja feromonskih tragova dolazi do sporijeg pronalaska boljih rješenja kroz iteracije. Na slici 56 još možemo vidjeti kako određene iteracije ne konstruiraju uvijek dobro rješenja, ali s vremenom napreduju u konstrukciji boljih rješenja.



Slika 57: prikaz najbolje rješenja svake iteracije za instancu problema A-n55-k9 (Izvor: izrada u Excelu)

A-n63-k9

Instanca problema A-n63-k9 je pokrenuta s pet različitih scenarija gdje su se različito podešavale lokalne optimizacije. Najbolje rješenje smo dobili u scenariju gdje smo koristili sve metode lokalne optimizacije i ono je iznosilo 1635 te se taj scenarij izvršavao najduže dok najlošije rješenje smo dobili u scenariju u kojemu nismo koristili prvu metodu lokalne optimizacije koja optimizira rutu svakog pojedinog auta zamjenom čvorova na njegovom putu. U svakom scenariju smo koristili 625 iteraciji s 80 mravi u svakoj. Optimalno znanstveno rješenje iznosi 1616. Možemo također zaključiti kako bez korištenja lokalne optimizacije ne dobivamo najgore rješenje.



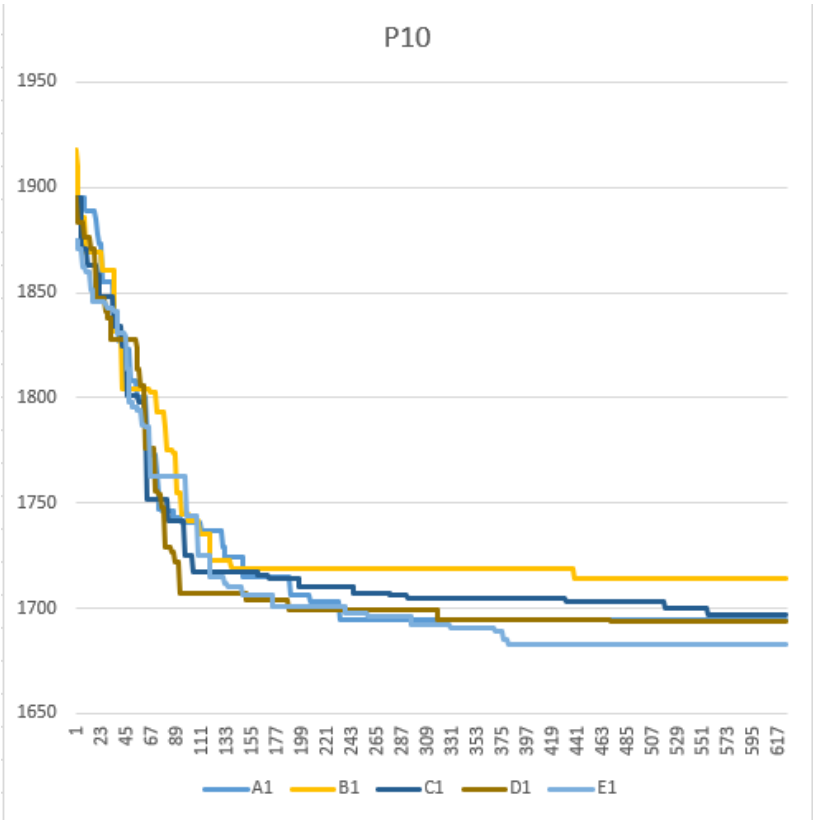
Slika 58: prikaz percentila vrijednosti u rasponu i medijana za instancu A-n63-k9 za svih 5 scenarija (Izvor: izrada u Excelu)

Scenarije smo označili slovima gdje A predstavlja scenarij bez korištenja lokalne optimizacije, slovo B scenarij bez korištenja prve metode lokalne optimizacije, slovo C scenarij bez korištenja druge metode lokalne optimizacije, slovo D scenarij bez korištenja treće metode lokalne optimizacije te slovo E koje predstavlja korištenje svih metoda lokalne optimizacije. Indeks 1 uz slova označava 10-percentila, indeks 2 označava 50-percentila, a indeks 3 označava 90-percentila. Sa slike možemo vidjeti graf koji prikazu performanse svakog scenarija, dok u tablici 4 možemo vidjeti vršne, prosječne i donje performanse. Najbolji rezultati za vršne performanse su za scenarij gdje koristimo sve metode lokalne optimizacije. U prosječnim performansama je najbolja za scenarij D gdje ne koristimo treću metodu lokalne optimizacije dok su donje performanse za scenarij C gdje ne koristimo drugu metodu lokalne optimizacije. Iz tablice možemo također zaključiti kako scenarij u kojemu ne koristimo lokalnu optimizaciju nije najbolji, ali nikad ni najgori.

Tablica 5: prikaz performansi za instancu A-63-k9

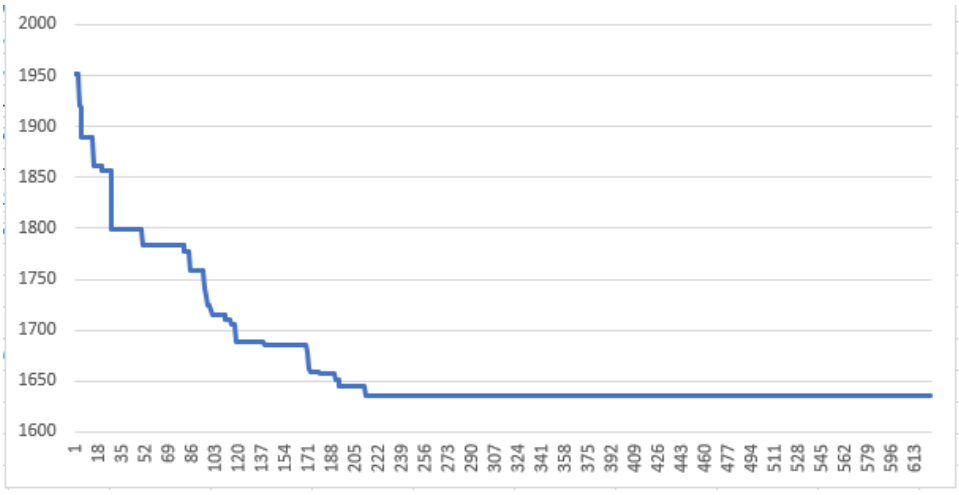
Vršne performanse	1	1683	E1
	2	1694	D1
	3	1695	A1
	4	1697	C1
	5	1714	B1
Prosječne performanse	1	1763	D2
	2	1766	C2
	3	1775	A2
	4	1776	E2
	5	1787	B2
Donje performanse	1	1841	C3
	2	1845	D3
	3	1850	E3
	4	1851	A3
	5	1868	B3

(Izvor: vlastita izrada)



Slika 59: prikaz vršnih performansi za svaki scenarij instance problema A-n63-k9 (Izvor: izrada u excelu)

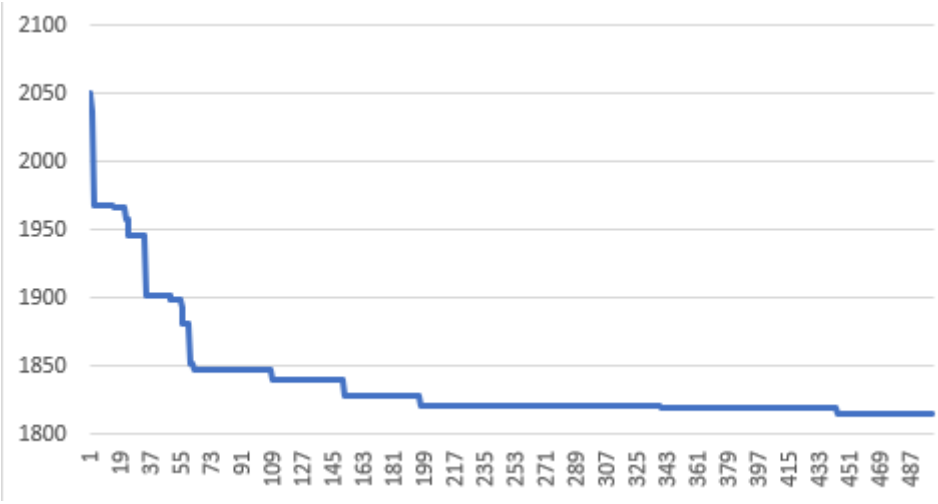
Na slici 59 možemo vidjeti kako je najbolji scenarij E u kojemu koristimo sve lokalne optimizacije, dok scenarij A gdje ne koristimo lokalne optimizacije nije najgori. Na slici 60 možemo vidjeti prikaz algoritma za instancu ovog problema koje je dao najbolje rješenje te također možemo vidjeti kako se napredak vidi kroz puno veći broj iteracija u odnosu na prošle instance problema.



Slika 60: prikaz instance algoritma koji je dao najbolje rješenje za instancu A-n63-k9 (Izvor: izrada u Excelu)

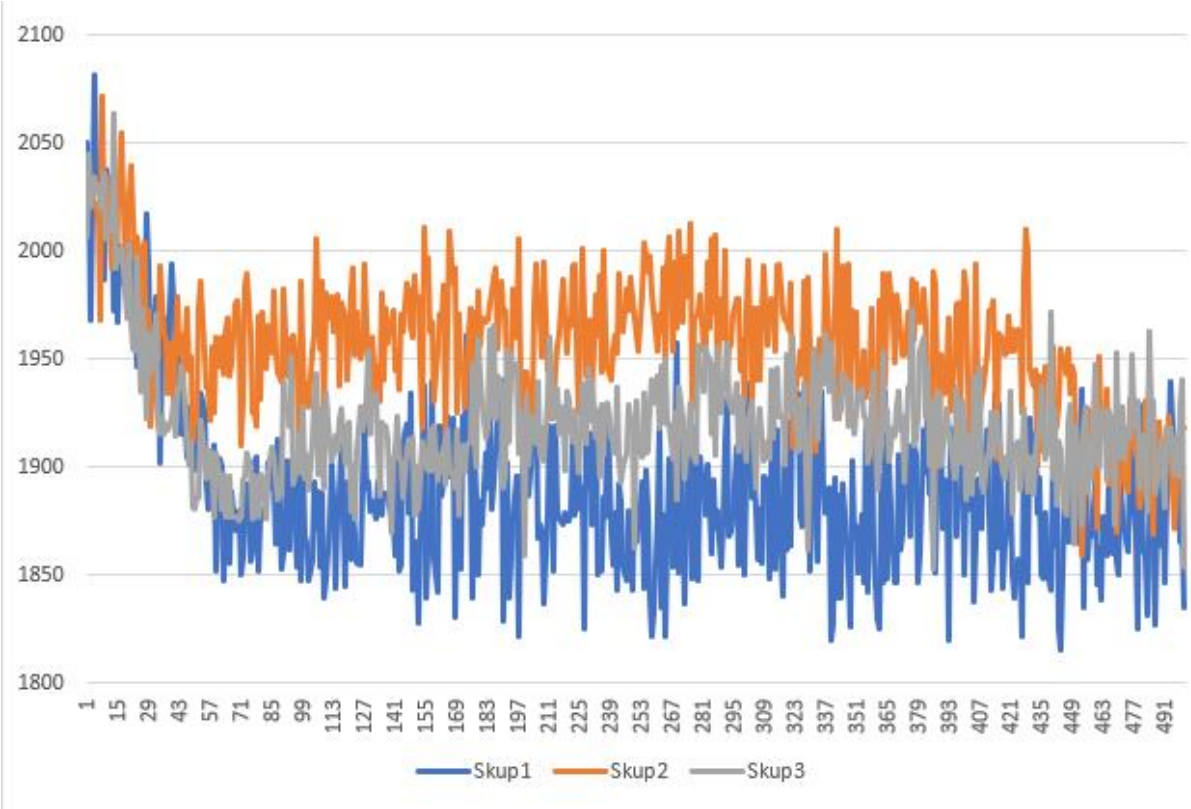
A-n80-k10

Instanca problema A-n80-k10 je pokrenuta s korištenjem sve tri metode lokalne optimizacije i korištenjem liste favorita te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode lokalne optimizacije. Lista favorita je postavljena na 20 što bi značilo da je svaki čvor imao 20 najbližih vrhova koje je prvo gledao, a u slučaju da su svi posjećeni onda bi gledao ostale vrhove. Bilo je 500 iteracija po 100 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 68 sati, 59 minuta i 5 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 1816. Optimalno rješenje iznosi 1763 što znači da je dobiveno rješenje 3.01% lošije od optimalnog.



Slika 61: prikaz instance algoritma koji je dao najbolje rješenje za instancu A-n80-k10 (Izvor: izrada u Excelu)

Na slici 61 možemo vidjeti prikaz algoritma za instancu ovog problema koje je dao najbolje rješenje. Kao i kod prethodnih instanci možemo vidjeti učenje kroz iteracije te konstrukciju boljih rješenja. Lista favorita je znatno pomogla da graf ima sličnu krivulju kao i prethodne instance jer instance problema s više od 100 čvorova ne daju dobra rješenja ako se samo upotrebljava lokalna optimizacije upravo zato što je to velik broj čvorova. Povećavanjem broja pokušaja lokalne optimizacije nije bilo dobro jer se znatno produžava vrijeme izvođenja programa, a rješenje ne bi nužno moralo biti bolje.

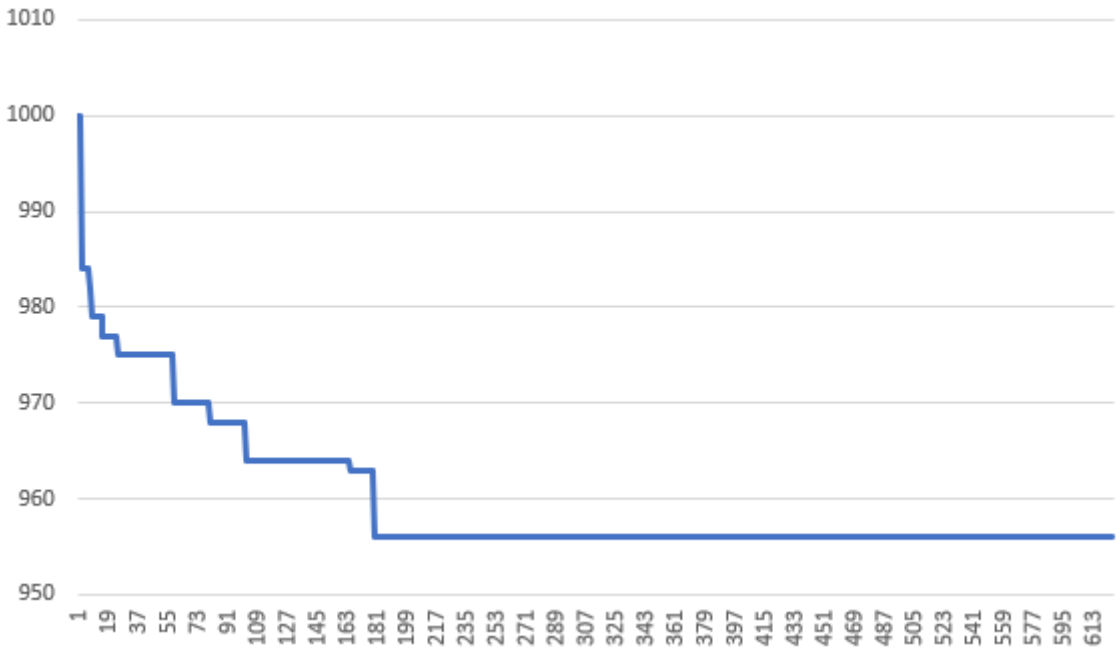


Slika 62: prikaz 3 instance algoritma rješenje za instancu A-n80-k10 (Izvor: izrada u Excelu)

Na slici 62 možemo vidjeti prikaz 3 instance algoritma za ovu instancu problema te možemo vidjeti kako određene instance konstruiraju bolja rješenja od ostalih. Kod sve tri instance vidimo kako se konstruiraju bolja rješenja sa svakom iteracijom samo kod skupa jedan vidimo da je graf puno niži u odnosu na ostala dva zbog bolje konstruiranih rješenja koje imaju manji trošak. Prilikom konstrukcije takvih rješenja u početku, mravi pronalaze taj put te se feromonski tragovi povećavaju i zbog toga mravi u kasnijim iteracijama pronalaze bolja rješenja.

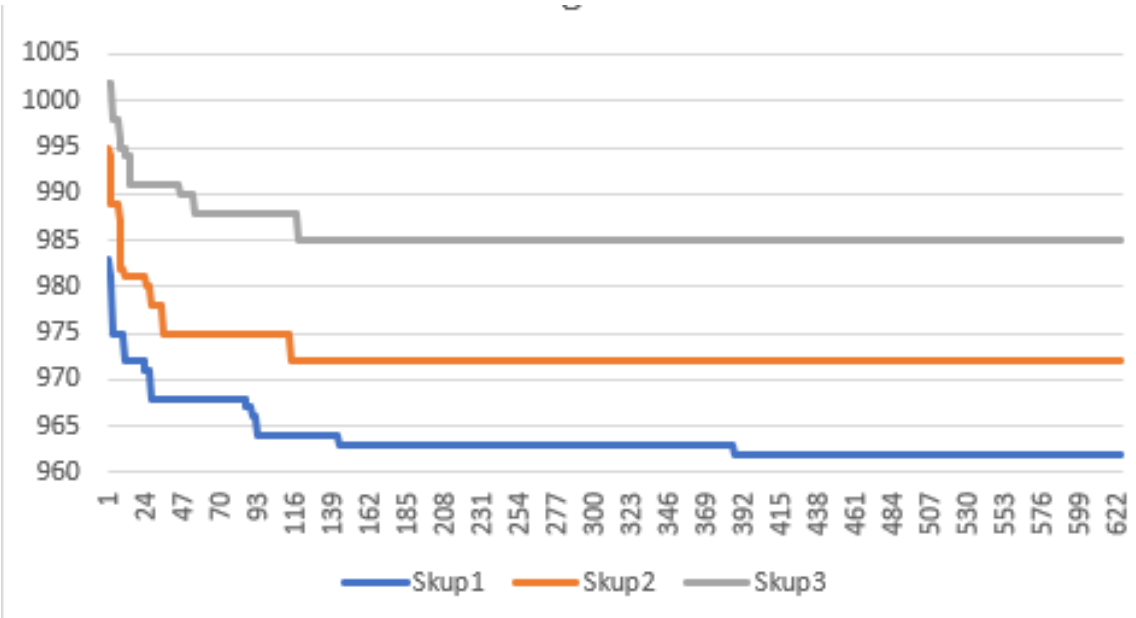
B-n35-k5

Instanca problema B-n35-k5 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 34 sata, 5 minuta i 13 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 956 što je za samo jednu jedinicu lošije od optimalnog.



Slika 63: prikaz instance algoritma koji je dao najbolje rješenje za instancu B-n35-k5 (Izvor: izrada u Excelu)

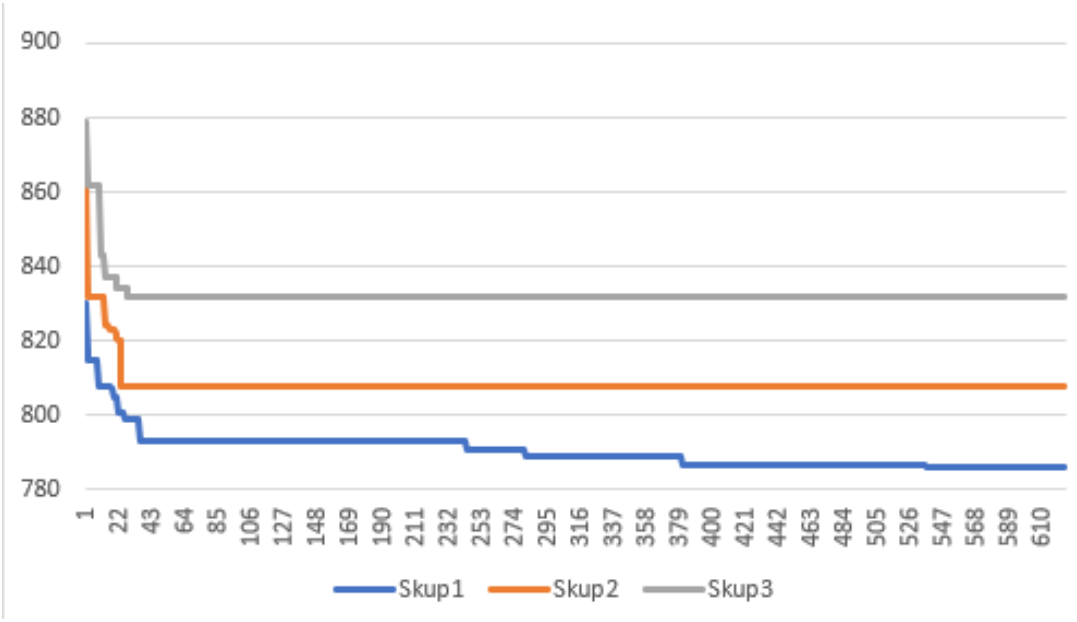
Na slici 63 možemo vidjeti prikaz algoritma za instancu ovog problema koje je dao najbolje rješenje. Kao i kod prethodnih instanci možemo vidjeti učenje kroz iteracije. Na slici 64 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma. Možemo primijetiti kako vršne performanse napreduju kroz iteracije dok donje i prosječne nakon određenog vremena stagniraju zbog malog poboljšanja rješenja.



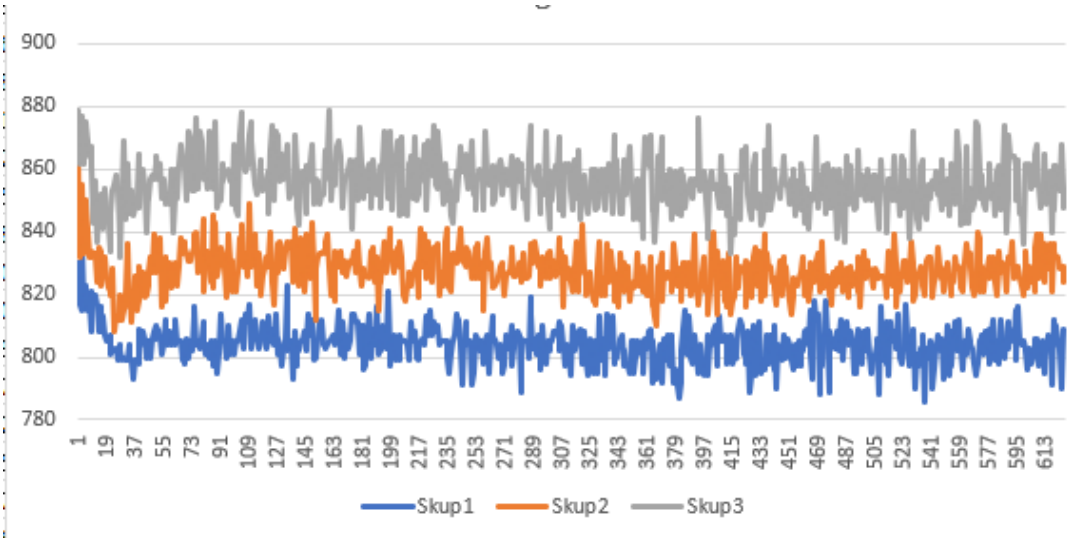
Slika 64: prikaz vršne, prosječne i donje performanse za instancu B-n35-k5 (Izvor: izrada u Excelu)

B-n45-k5

Instanca problema B-n45-k5 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 20 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 3 sata i 2 minute. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 766, dok je optimalno rješenje 751. Dobiveno rješenje je lošije za 2%. Na slici 65 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma.



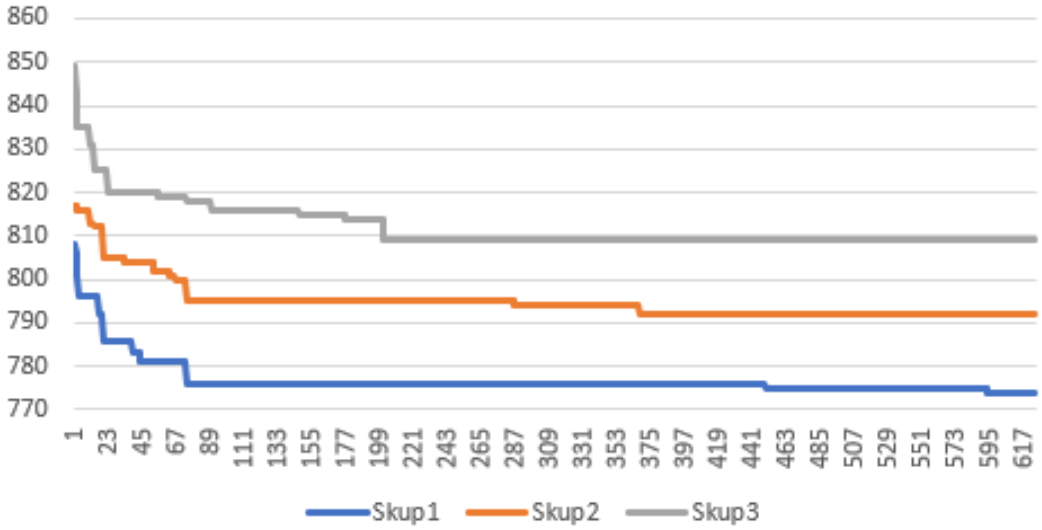
Slika 65: prikaz vršne, prosječne i donje performanse za instancu B-n45-k5 (Izvor: izrada u Excelu)



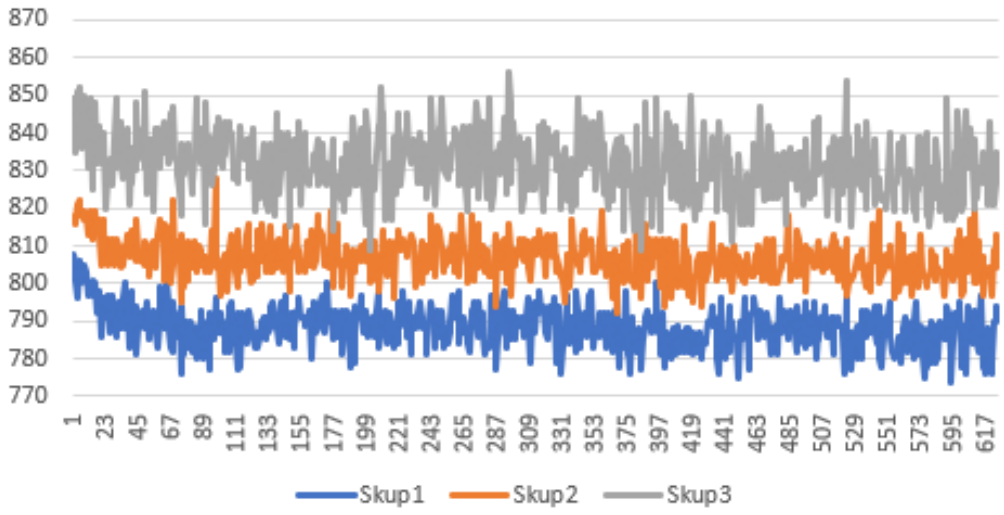
Slika 66: prikaz vršnih percentila, medijana i donjih percentzila za instancu B-n45-k5 (Izvor: izrada u Excelu)

B-n50-k7

Instanca problema B-n50-k7 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 43 sata, 52 minute i 52 sekunde. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 755, dok je optimalno rješenje 741. Dobiveno rješenje je lošije za 1.89%. Na slici 67 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje, dok na slici 68 možemo vidjeti prikaz donjih, gornjih i prosječnih performansi za svaku iteraciju.



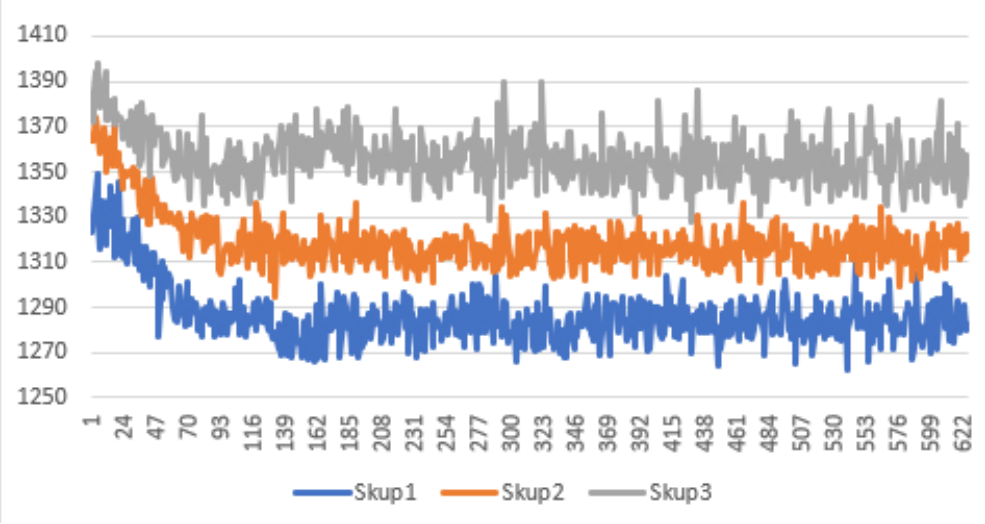
Slika 67: prikaz vršne, prosječne i donje performanse za instancu B-n50-k7 (Izvor: izrada u Excelu)



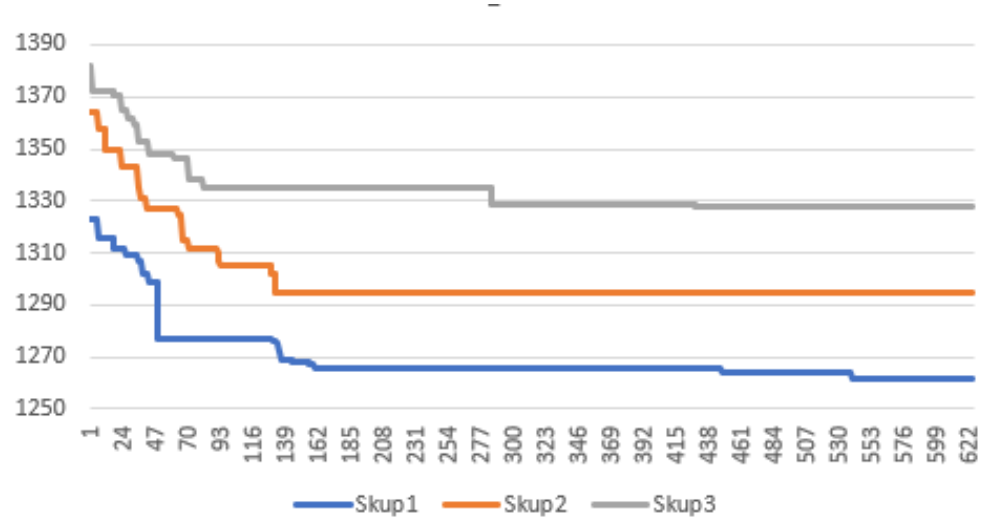
Slika 68: prikaz vršnih percentila, medijana i donjih percentzila za instancu B-n50-k7 (Izvor: izrada u Excelu)

B-n78-k10

Instanca problema B-n78-k10 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 100 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 30 sati, 46 minuta i 28 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 1240, dok je optimalno rješenje 1221. Dobiveno rješenje je lošije za 1.56%. Na slici 69 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje, dok na slici 70 možemo vidjeti prikaz donjih, gornjih i prosječnih performansi za svaku iteraciju.



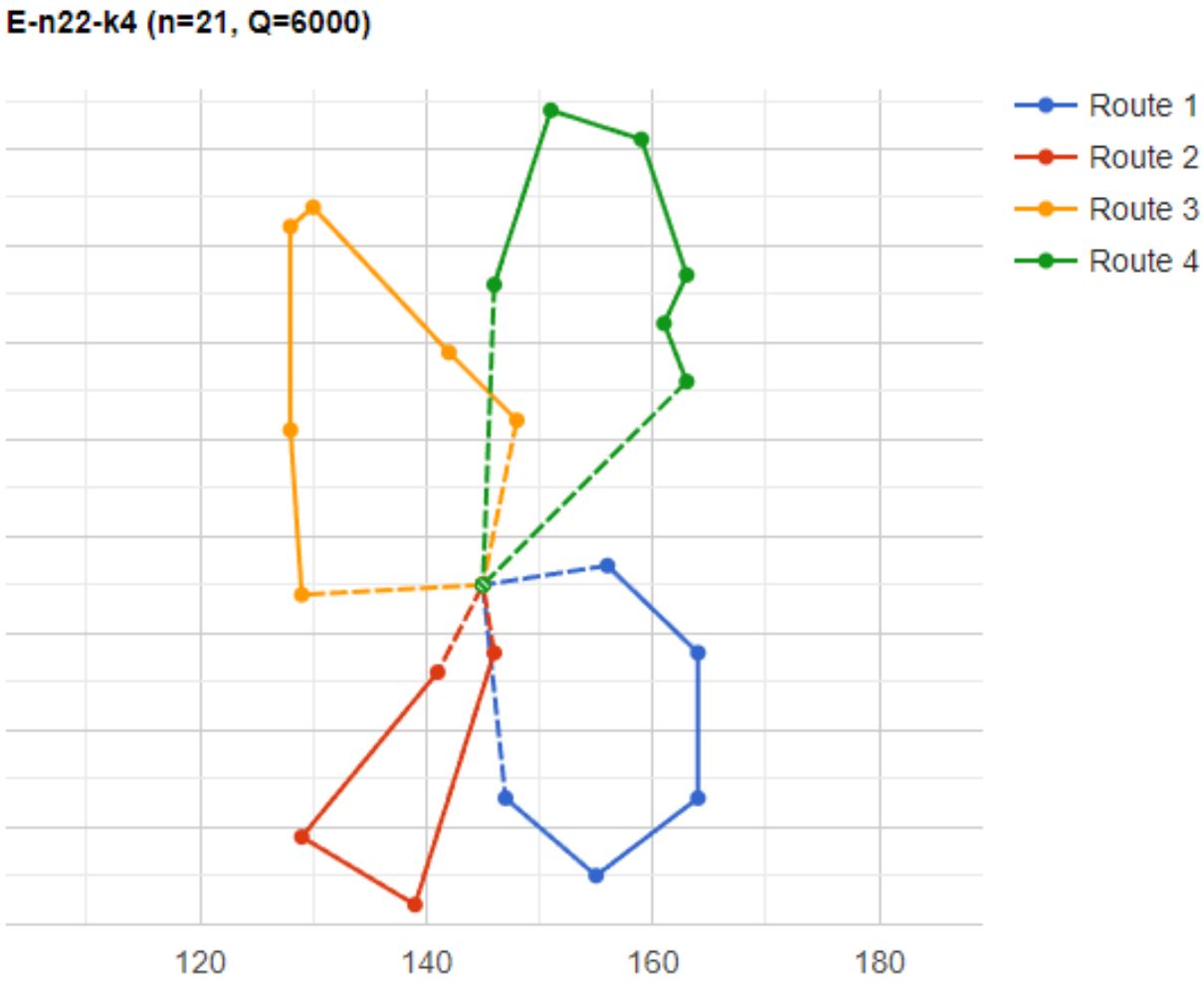
Slika 69: prikaz vršnih percentila, medijana i donjih percentzila za instancu B-n78-k10 (Izvor: izrada u Excelu)



Slika 70: prikaz vršne, prosječne i donje performanse za instancu B-n78-k10 (Izvor: izrada u Excelu)

E-n22-k4

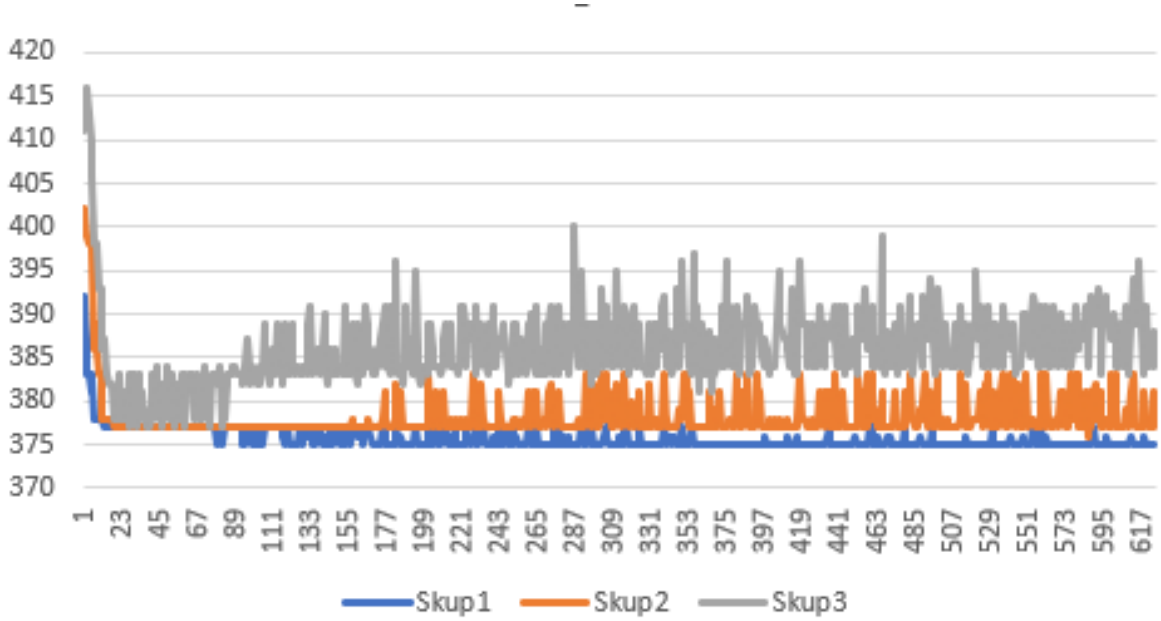
Instanca problema E-n22-k4 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 30 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 2 sata, 13 minuta i 28 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 375 koje je ujedno i optimalno rješenje. Na slici 71 možemo vidjeti grafički prikaz optimalnog rješenja gdje je korišteno 4 vozila. Ruta svakog vozila je prikazana drugom bojom.



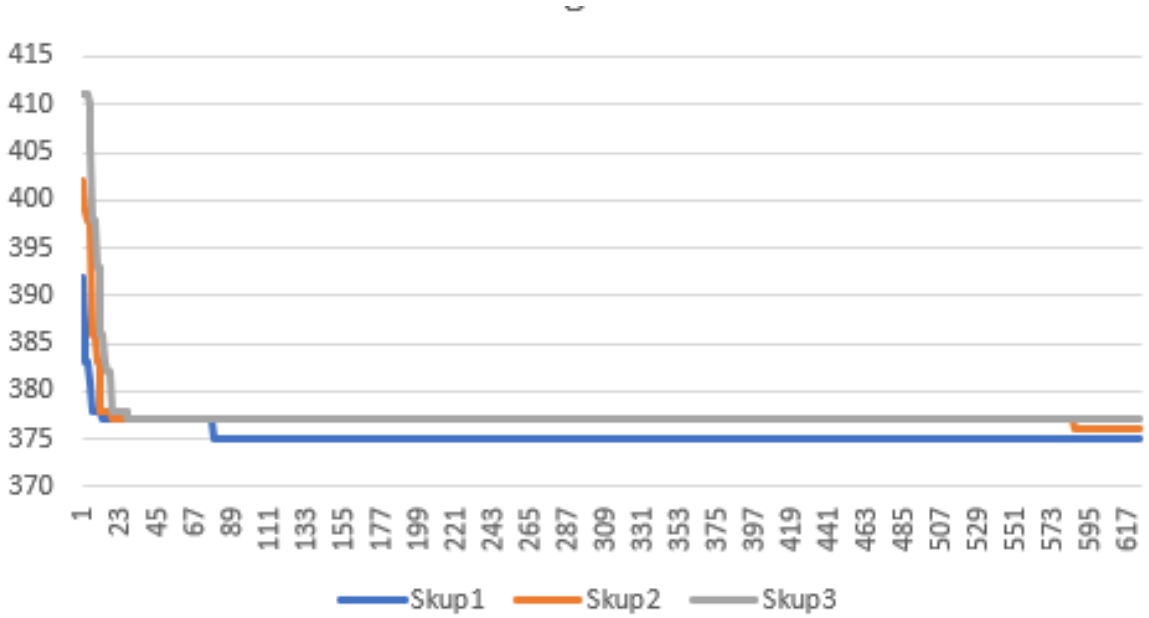
Slika 71: grafički prikaz optimalnog rješenja za instancu E-n22-k4 (Izvor: CVRPLIB, link)

Na slici 72 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje, dok na slici 73 možemo vidjeti prikaz donjih, gornjih i prosječnih performansi za svaku iteraciju. S obzirom na

to da se radi o manjoj instanci problema možemo vidjeti kako se optimalno rješenje pronalazi već u početnim iteracijama.



Slika 72: prikaz vršnih percentila, medijana i donjih percentzila za instancu E-n22-k4 (Izvor: izrada u Excelu)



Slika 73: prikaz vršne, prosječne i donje performanse za instancu E-n22-k4 (Izvor: izrada u Excelu)

E-n30-k3

Instanca problema E-n30-k3 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 10 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 1 sat, 15 minuta i 31 sekunda. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje

smo dobili je bilo 505, dok je optimalno znanstveno rješenje 534. Ovdje dolazimo do jedne zanimljive situacije gdje smo dobili bolje rješenje od onog predstavljenog, a razlog je korištenje 4 vozila umjesto 3. Oni su u postavkama programa naveli kako je minimalan broj vozila 3 što je točno, no optimalno rješenje kojeg smo mi dobili koristi 4 vozila.

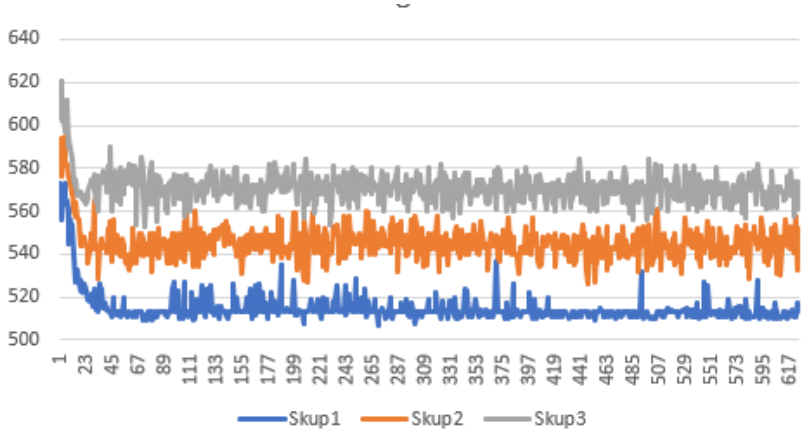
```
Route #1: 26 28 27 29 25 24 6 21
Route #2: 20 3 4 1 5 2 22
Route #3: 19 15 16 13 7 17 9 14 8 12 11 10 23 18
Cost 534
```

Slika 74: prikaz optimalnog rješenja korištenjem 3 vozila (Izvor: CVRPLIB, izvor)

```
18 23 10 11 12 8 14 9 17 7 13 16 15
19 6 1 24 25 29 27 28 26
3 4 5 2 22 20
21
SOLUTION VALUE: 505
```

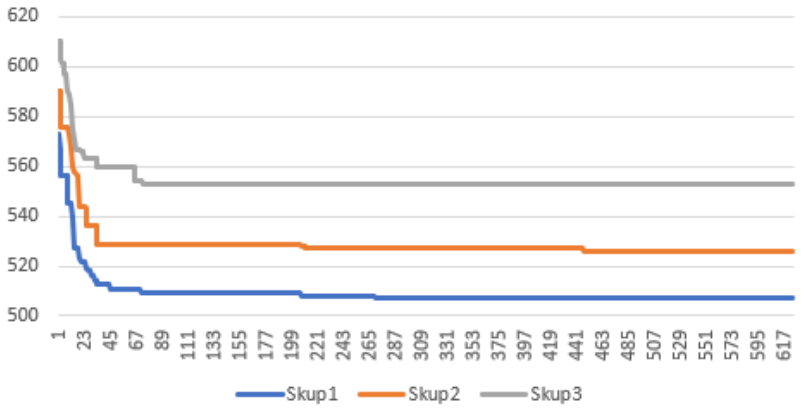
Slika 75: prikaz optimalnog rješenja kojega smo mi dobili korištenjem 4 vozila (Izvor: vlastita izrada)

Na slici 74 je prikazano optimalno rješenje korištenjem 3 vozila, dok je na slici 75 prikazano optimalno rješenje korištenjem 4 vozila. Ovdje također možemo primijetiti kako je manja cijena korištenjem 4 vozila.



Slika 76: prikaz vršnih percentila, medijana i donjih percentzila za instancu E-n30-k3 (Izvor: izrada u Excelu)

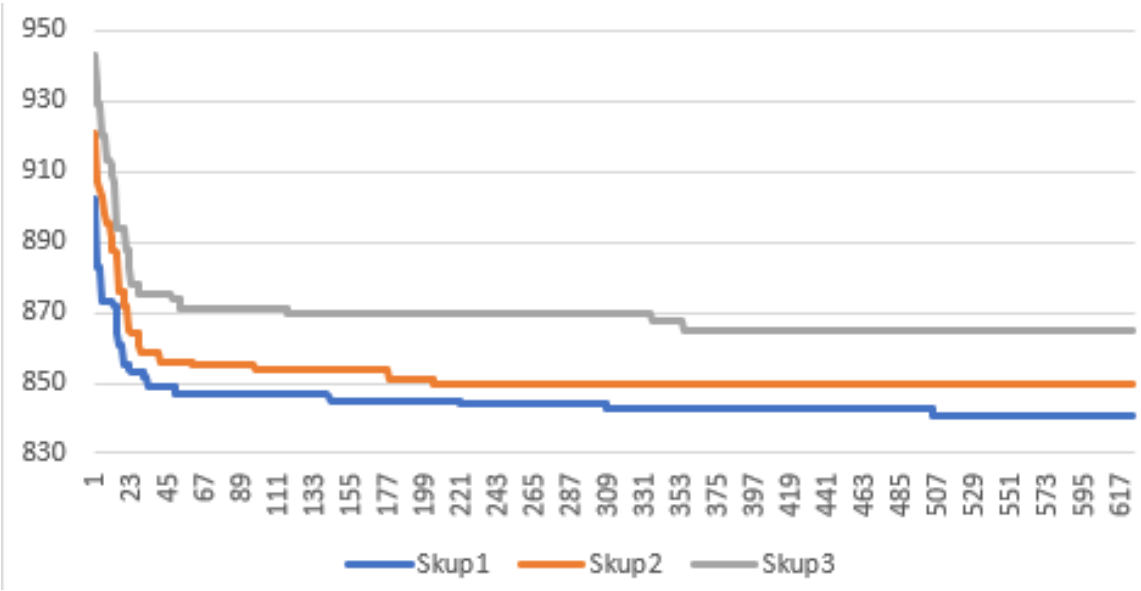
Na slici 76 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje, dok na slici 77 možemo vidjeti prikaz donjih, gornjih i prosječnih performansi za svaku iteraciju.



Slika 77: prikaz vršne, prosječne i donje performanse za instancu E-n30-k3 (Izvor: izrada u Excelu)

E-n33-k4

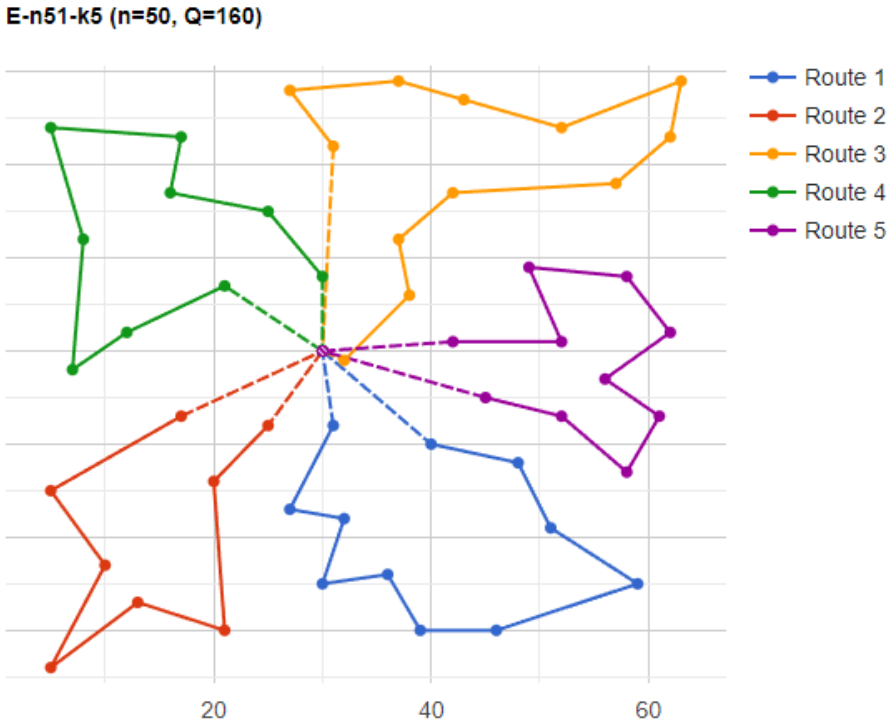
Instanca problema E-n33-k4 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 20 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 2 sata, 11 minuta i 9 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 839, dok je optimalno rješenje 835. Dobiveno rješenje je lošije za 0.48%. Na slici 78 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje u svakoj iteraciji.



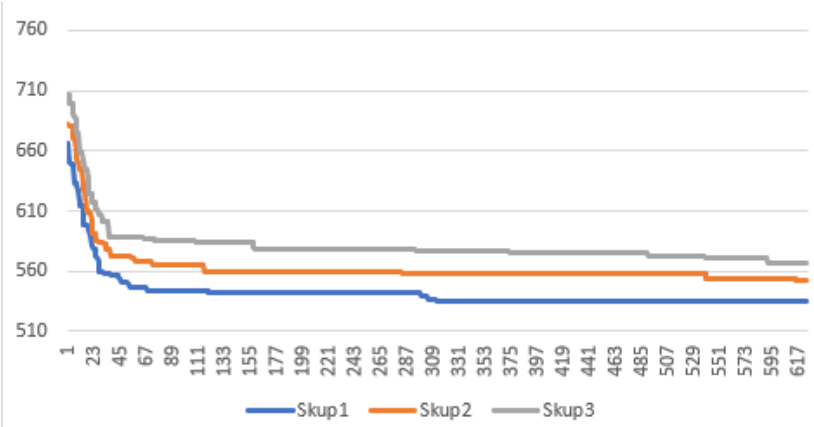
Slika 78: prikaz vršne, prosječne i donje performanse za instancu E-n33-k4 (Izvor: izrada u Excelu)

E-n51-k5

Instanca problema E-n51-k5 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 20 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 3 sata, 16 minuta i 20 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 521 koje je ujedno i optimalno rješenje. Na slici 79 možemo vidjeti grafički prikaz optimalnog rješenja koje koristi pet vozila.



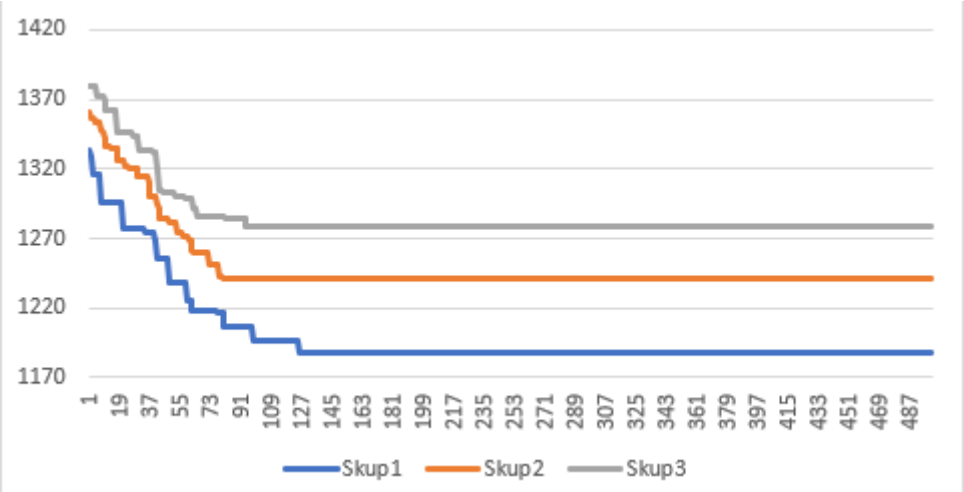
Slika 79: grafički prikaz optimalnog rješenja za instancu E-n51-k5 (Izvor: CVRPLIB, link)



Slika 80: prikaz vršne, prosječne i donje performanse za instancu E-n51-k5 (Izvor: izrada u Excelu)

E-n101-k14

Instanca problema E-n101-k14 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 10 pokušaja svake metode. Bilo je 500 iteracija po 100 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 9 sati, 54 minute i 50 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 1168, dok je optimalno rješenje 1067. Dobiveno rješenje je lošije za 9.47%. Na slici 81 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma tako da pratimo samo najbolje globalno rješenje u svakoj iteraciji.



Slika 81: prikaz vršne, prosječne i donje performanse za instancu E-n101-k14 (Izvor: izrada u Excelu)

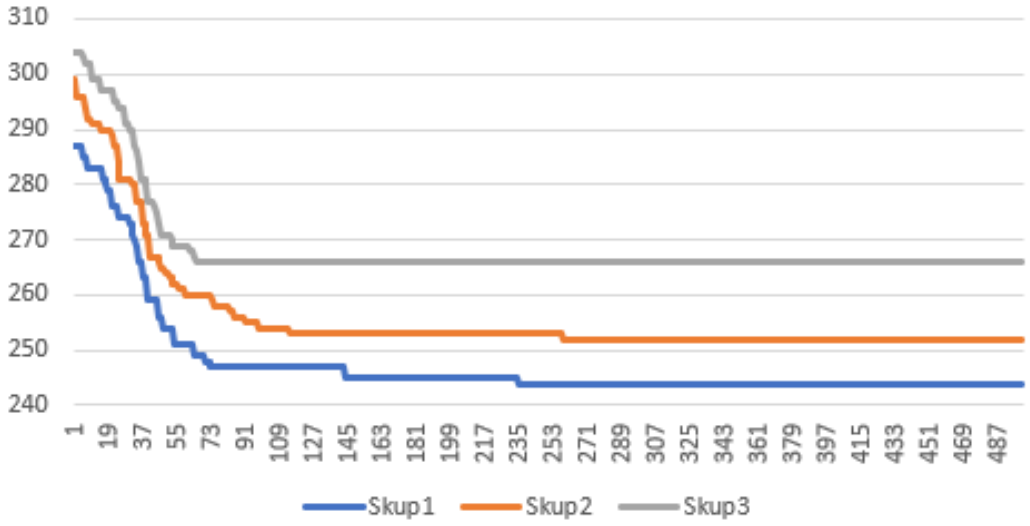
F-n45-k4

Instanca problema F-n45-k4 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 10 pokušaja svake metode. Bilo je 625 iteracija po 80 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 1 sat, 59 minuta i 1 sekunda. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 744, dok je optimalno rješenje 724. Dobiveno rješenje je lošije za 2.76%. Na slici 82 možemo vidjeti vršne, prosječne i donje performanse za sva pokretanja algoritma.

Slika 82: prikaz vršne, prosječne i donje performanse za instancu F-n45-k4 (Izvor: izrada u Excelu)

F-n72-k4

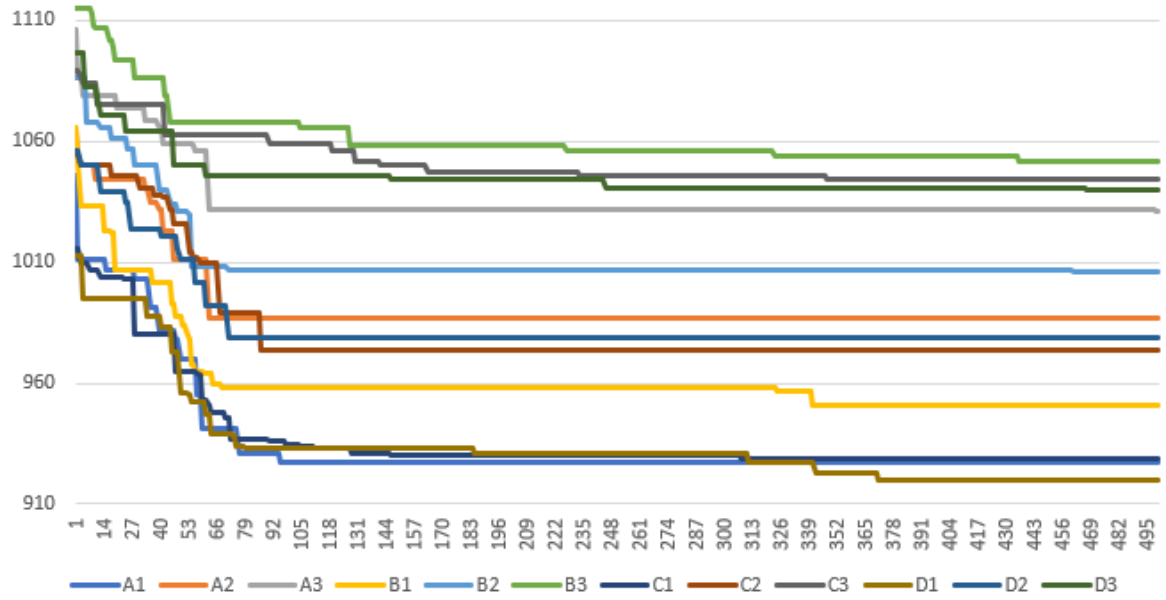
Instanca problema F-n72-k4 je pokrenuta s korištenjem sve tri metode lokalne optimizacije te je za svako konstruirano rješenje bilo po 20 pokušaja svake metode. Bilo je 500 iteracija po 100 mravi u svakoj. Ukupno vrijeme izvođenja je bilo 8 sati, 31 minuta i 47 sekundi. Problem smo pokrenuli ukupno 21 put, a najbolje rješenje koje smo dobili je bilo 243, dok je optimalno rješenje 237. Dobiveno rješenje je lošije za 2.53%.



Slika 83: prikaz vršne, prosječne i donje performanse za instancu F-n72-k4 (Izvor: izrada u Excelu)

M-n101-k10

Instanca problema M-n101-k10 je pokrenuta s četiri različita scenarija gdje su se različito podešavale lokalne optimizacije. Najbolje rješenje smo dobili u scenariju gdje nismo koristili sve metode lokalne optimizacije i ono je iznosilo 897 te se taj scenarij izvršavao puno kraće od ostalih dok najlošije rješenje smo dobili u scenariju u kojemu nismo koristili treću metodu lokalne optimizacije koja optimizira rutu svakog pojedinog auta zamjenom čvorova na njegovom putu. U svakom scenariju smo koristili 500 iteraciji sa 100 mravi u svakoj. Optimalno znanstveno rješenje iznosi 820.



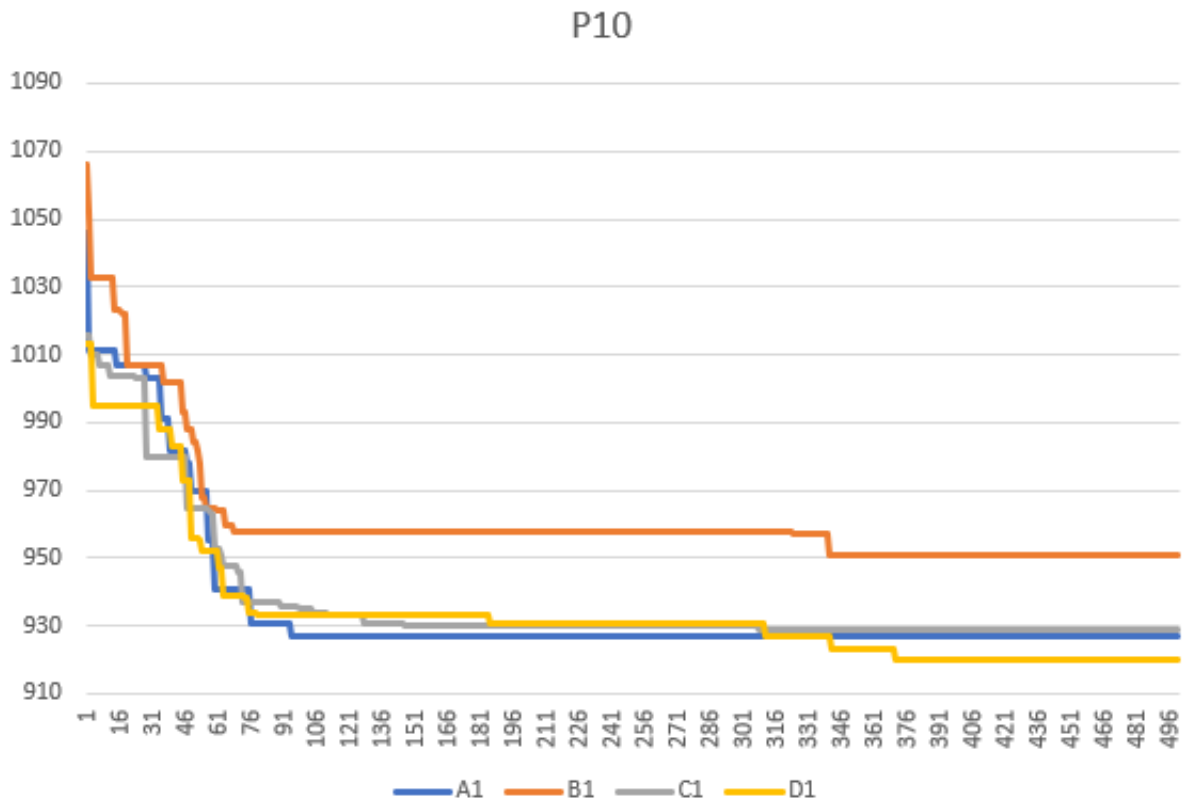
Slika 84: prikaz percentila vrijednosti u rasponu i medijana za instancu M-n101-k10 za sva 4 scenarija (Izvor: izrada u Excelu)

Scenarije smo označili slovima gdje A predstavlja scenarij sa korištenjem svih lokalnih optimizacija, slovo B scenarij bez korištenja prve metode lokalne optimizacije, slovo C scenarij bez korištenja druge metode lokalne optimizacije i slovo D scenarij bez korištenja treće metode lokalne. Sa slike možemo vidjeti graf koji prikazu performanse svakog scenarija, dok u tablici 4 možemo vidjeti vršne, prosječne i donje performanse. Najbolji rezultati za vršne performanse su za scenarij gdje koristimo ne koristimo treću metodu lokalne optimizacije. U prosječnim performansama je najbolja za scenarij C gdje ne koristimo drugu metodu lokalne optimizacije dok su donje performanse za scenarij A gdje koristimo sve metode lokalne optimizacije.

Tablica 6: prikaz performansi za instancu M-n101-k10

Vršne performanse	1	920	D1
	2	927	A1
	3	929	C1
	4	951	B1
Prosječne performanse	1	974	C2
	2	979	D2
	3	987	A2
	4	1006	B2
Donje performanse	1	1031	A3
	2	1040	D3
	3	1044	C3
	4	1052	B3

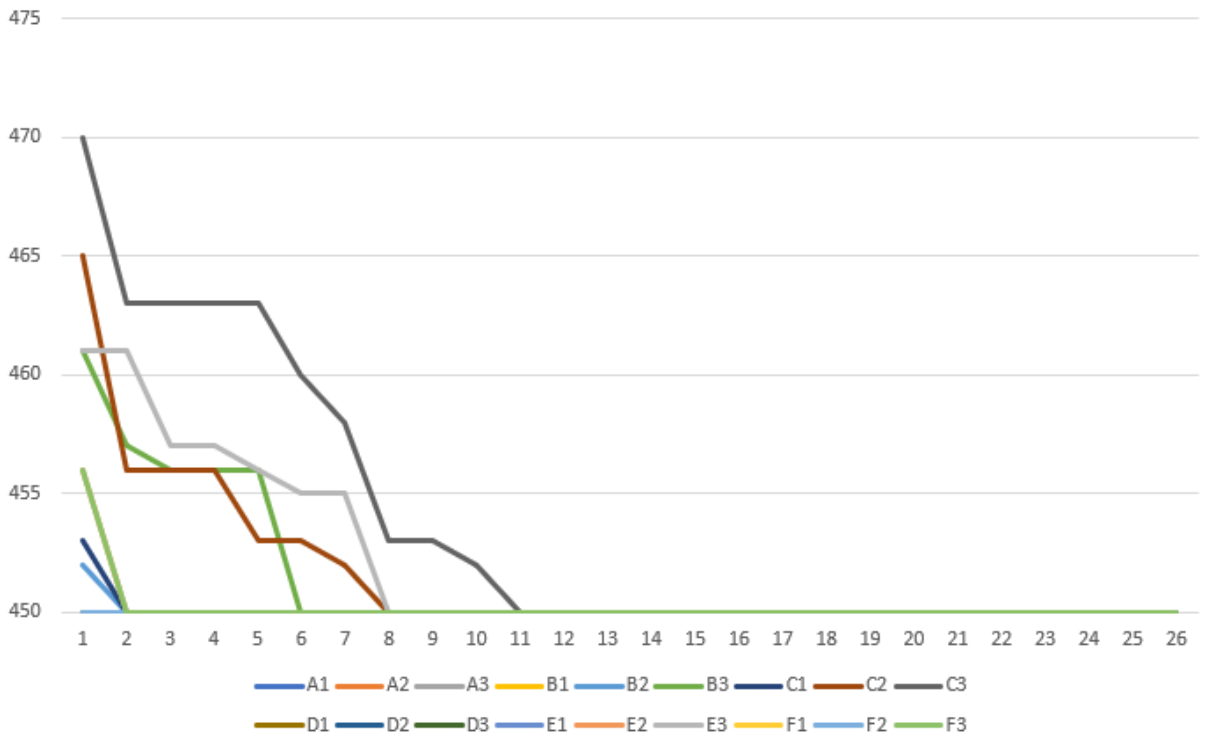
(Izvor: vlastita izrada)



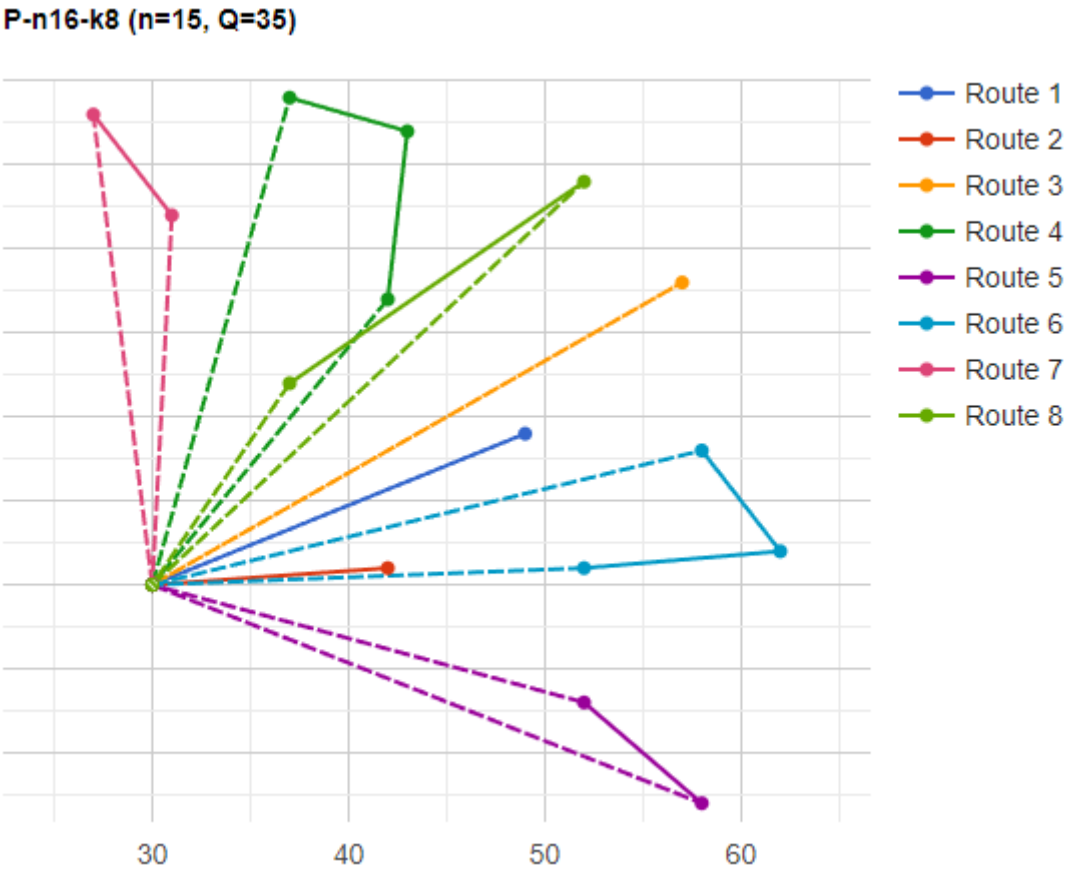
Slika 85: prikaz vršnih performansi za svaki scenarij instance problema M-n101-k10 (Izvor: izrada u excelu)

P-n16-k8

Instanca problema P-n16-k8 je pokrenuta sa šest različita scenarija gdje su se različito podešavale lokalne optimizacije. Najbolje rješenje smo dobili u svakom scenariju koje je ujedno i optimalno rješenje. Na slici 86 je prikazan prikaz prvih 26 iteracija te možemo zaključiti da se jako brzo pronađe optimalno rješenje. Ova instanca problema je bila premala za korištenje strategije lokalne optimizacije za sve tri metode po 100. U jednom scenariju nismo imali lokalnu optimizaciju te smo svejedno pronašli rješenje. Ova linija koja zadnja pronašla optimalno rješenje je upravo strategija koja nije koristila lokalnu optimizaciju. Na slici 87 možemo vidjeti grafički prikaz optimalnog rješenja koje koristi osam vozila.



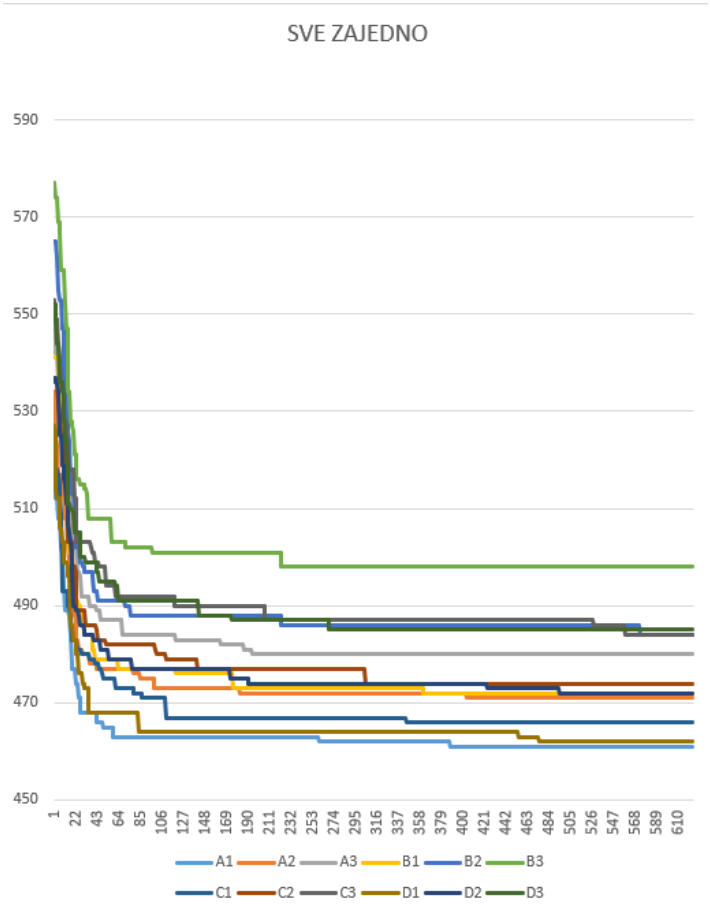
Slika 86: prikaz percentila vrijednosti u rasponu i medijana za instancu P-n16-k8 za svih 6 scenarija (Izvor: izrada u Excelu)



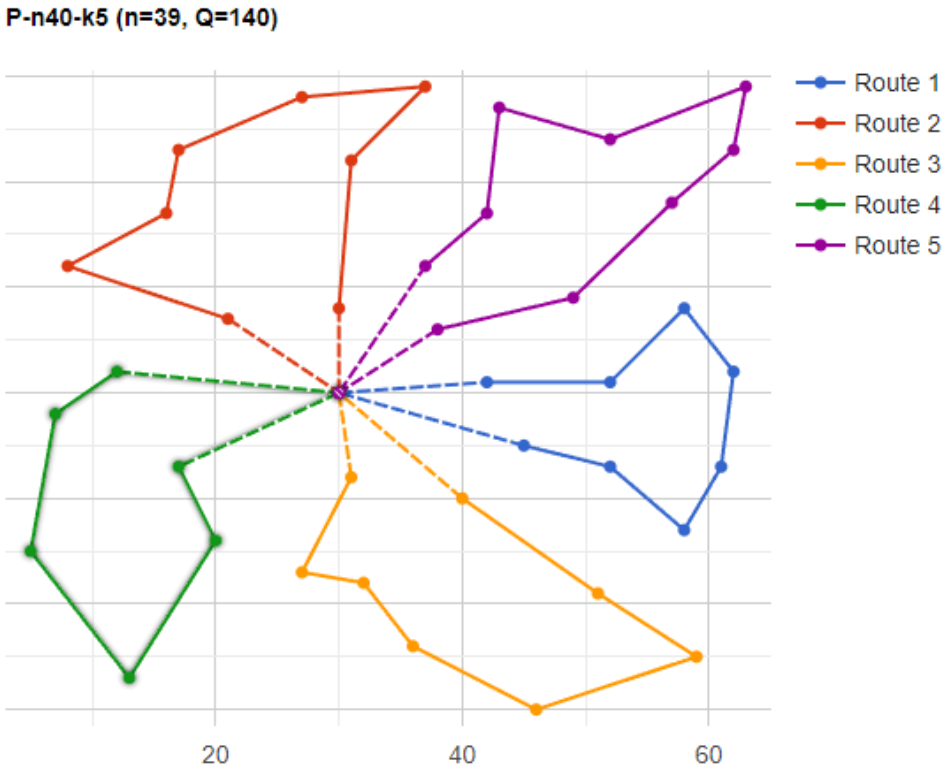
Slika 87: grafički prikaz optimalnog rješenja za instancu P-n16-k8 (Izvor: CVRPLIB, link)

P-n40-k5

Instanca problema P-n40-k5 je pokrenuta s četiri različita scenarija gdje su se različito podešavale lokalne optimizacije. Najbolje rješenje smo dobili u scenariju gdje smo koristili sve metode lokalne optimizacije i ono je iznosilo 458 te je to ujedno bilo i optimalno rješenje. Scenarije smo označili slovima gdje A predstavlja scenarij s korištenjem svih lokalnih optimizacija, slovo B scenarij bez korištenja prve metode lokalne optimizacije, slovo C scenarij bez korištenja druge metode lokalne optimizacije i slovo D scenarij bez korištenja treće metode lokalne. Na slici 89 možemo vidjeti grafički prikaz optimalnog rješenja koje koristi pet vozila.



Slika 88: prikaz percentila vrijednosti u rasponu i medijana za instancu P-n40-k5 za sva 4 scenarija (Izvor: izrada u Excelu)



Slika 89: grafički prikaz optimalnog rješenja za instancu P-n40-k5 (Izvor: CVRPLIB, link)

8. Zaključak

Problem usmjeravanja vozila predstavlja jedan od najvećih izazova u logističkom sektoru jer je to složeni problem koji odražava stvarne probleme s kojima se suočavaju mnogi logistički odjeli u poduzećima. Prvi put se VRP pojavio u radu Georgea Dantziga i Johna Ramsera 1959. godine u kojem je opisan prvi algoritamski pristup koji je primijenjen na problem isporuke benzina. Postoji mnogo varijacija ovog problema koje se razlikuju po broju klijenata, kapacitetu svakog vozila, zahtjevima klijenata i broju dostupnih vozila.

Diplomski rad „Algoritam inteligencije rojeva za problem usmjeravanja vozila“ je kroz uvodno poglavlja objasnio postojanje NP-teških problema i objašnjeno je kako su to problemi koji nisu rješivi u razumnom vremenu te da postoje heuristički algoritmi koji se temelje na iskustvu koji u razumnom vremenu mogu pronaći dovoljno dobro rješenje. Postom smo objasnili postojanje takvih algoritma i odlučili se za implementaciju mravljeg algoritma kojega smo potom detaljno objasnili. U jeziku C++ smo implementirali MMAS algoritam na kojemu su kasnije provođeni eksperimenti na instancama problema A-n32-k5, A-n55-k9, A-n63-k9, A-n80-k10, B-n35-k5, B-n45-k5, B-n50-k7, B-n78-k10, E-n101-k14, E-n22-k4, E-n30-k3, E-n33-k4, E-n51-k5, F-n45-k4, F-n72-k4, M-n101-k10, P-n16-k8 i P-n40-k5.

Kroz različite scenarije eksperimenata došli smo do zaključka kako lokalna optimizacija uvelike doprinosi poboljšanju rješenja te smo također pokazali kako metoda lokalne optimizacije oduzimanja i dodavanja određenog čvora u rutama dva vozila u velikoj mjeri utječe na kvalitetu rješenja. Određeni scenariji eksperimenata su nam pokazali kako bez lokalne optimizacije ne dobivamo najgore rješenje što je bilo određeno iznenađenje. U jednom scenariju s instancom E-30-k3 smo došli do rješenja koje je bolje od znanstveno prikazanog zbog korištenja dodatnog vozila te smo također zaključili kako lokalna optimizacija uvelike produljiva vrijeme izvršavanja programa. Za instance problema s više od sto čvorova smo koristili listu favorita koja je pokazala svoju učinkovitost. U 23 scenarija koja smo proveli smo dobili rješenja koja odstupaju od optimalno prikazan do 10%, u 9 scenarija smo dobili optimalno rješenje, a u jednom smo dobili bolje rješenje od najboljeg objavljenog rješenje zbog korištenja dodatnog vozila koje je uvelike poboljšalo kvalitetu konačnog rješenja.

9. Popis literature

[1] Solnon, C. (2013). *Ant Colony Optimization and Constraint Programming*. John Wiley & Sons.

[2] Ladner, R. E. (1975). "On the Structure of Polynomial Time Reducibility". *Journal of the ACM*., New York, NY, United States

[3] Daniel Pierre Bovet and Crescenzi, P. (1994). *Introduction to the Theory of Complexity*. Prentice Hall PTR.

[4] Fortnow, L. (2009). *„The status of the P versus NP problem“.* Communications of the ACM, 52(9), pp.78–86. doi:https://doi.org/10.1145/1562164.1562186.

[5] Golden, B.L., S Raghavan, Wasil, E.A. and Springerlink (2008). *The Vehicle Routing Problem: Latest Advances and New Challenges*. New York, Ny: Springer Us.

[6] Kvesić, B. (2020.) *Algoritam računalne inteligencije za rješaanje problema putujućeg lopova* (Završni rad), Fakultet organizacije i informatike, Zagreb, Sveučilište u Zagrebu

[7] Cordeau, Jean-François (2007). *[Handbooks in Operations Research and Management Science] Transportation Volume 14 || Chapter 6 Vehicle Routing.* , 367–428. doi:10.1016/s0927-0507(06)14006-2

[8] Mazidi, A., Fakhrahmad, M., & Sadreddini, M. H. (2016). *A meta-heuristic approach to CVRP problem: local search optimization based on GA and ant colony*.

[9] Anon, (2022). *Vehicle Routing Problem - Upper Route Planner*. [online] Preuzeto s: https://www.upperinc.com/blog/vehicle-routing-problem/

[10] Borčinova, Z. (2017). Two models of the capacitated vehicle routing problem. *Croatian Operational Research Review*, pp.463–469. doi:https://doi.org/10.17535/crorr.2017.0029.

[11] Engelbrecht, A.P. (2007). Computational Intelligence. John Wiley & Sons.

[12] Radanović, G. (2007). Pregled heurističkih algoritama (Seminarski rad), Fakultet elektrotehnike i računarstva, Zagreb, Sveučilište u Zagrebu

[13] Brownlee, J. (2012). *Clever algorithms : nature-inspired programming recipes*. United Kingdom

[14] Jungić, B. (2018). Meta-heuristički algoritmi za problem odabira tima (Diplomski rad), Prirodoslovno–matematički fakultet, matematički odsjek, Zagreb, Sveučilište u Zagrebu.

[15] Dorigo, M., Stützle, T., (2004). Ant Colony Optimization, MIT Press, Cambridge, MA

[16] Ant colony optimization, (2007). U Scholarpedia online. Preuzeto s http://www.scholarpedia.org/article/Ant_colony_optimization

[17] Saxena, A., & Mueller, C. (2018). Intelligent intrusion detection in computer networks using swarm intelligence. *International Journal of Computer Applications*, *179*, 1-9.

[18] Negulescu, Sorin & Oprean, Constantin & Kifor, Claudiu & Carabulea, Ilie. (2008). *Elitist ant system for route allocation problem*.

[19] Stutzle, T., i Hoos, H. H. (2000). *MAX–MIN Ant System, Future Generation Computer Systems 16* , Vancouver, BC, Canada

[20] Ivković, Nikola ; Golub, Marin. A New Ant Colony Optimization Algorithm : Three Bound Ant System // Lecture notes in computer science, 8667 (2014.), 280-281

[21] Ivković, Nikola; Modeliranje, analiza i poboljšanje algoritama optimizacije kolonijom mrava, 2014., doktorska disertacija, Fakultet elektrotehnike i računarstva

[22] Pérez-Carabaza, S., Gálvez, A. and Iglesias, A. (2022). *Rank-Based Ant System with Originality Reinforcement and Pheromone Smoothing*. Applied Sciences, 12(21), p.11219. doi:https://doi.org/10.3390/app122111219

[23] Pedemonte, M., Nesmachnow, S. and Cancela, H. (2011). *A survey on parallel ant colony optimization*. Applied Soft Computing, 11(8), pp.5181–5197. doi:https://doi.org/10.1016/j.asoc.2011.05.042.

[24] Hu, XM., Zhang, J. & Li, Y. *Orthogonal Methods Based Ant Colony Search for Solving Continuous Optimization Problems.* J. Comput. Sci. Technol. 23, 2–18 (2008). https://doi.org/10.1007/s11390-008-9111-5

[25] Gupta, D.K., Arora, Y., Singh, U.K. and Gupta, J.P. (2012). *Recursive Ant Colony Optimization for estimation of parameters of a function*. doi:https://doi.org/10.1109/RAIT.2012.6194620.

[26] Pikus, F.G. (2019). Hands-On Design Patterns with C++. Packt Publishing Ltd.

[27] Mariusz Boryczka (2008). *Ant Colony Programming with the Candidate List. Springer eBooks*, pp.302–311. doi:https://doi.org/10.1007/978-3-540-78582-8_31.

[28] Ivković, Nikola; Maleković, Mirko; Golub, Marin *Extended Trail Reinforcement Strategies for Ant Colony Optimization // Swarm, Evolutionary, and Memetic Computing, Lecture Notes in Computer Science*, 7076 (2011), 1; 662-669 doi:10.1007/978-3-642-27172-4_78

[29] Ivković, Nikola; Golub, Marin; Jakobović, Domagoj, Designing DNA Microarrays with Ant Colony Optimization // Journal of computers, 11 (2016), 6; 528-536 doi:10.17706/jcp.11.6.528-536 (međunarodna recenzija, članak, znanstveni)

[30] Zhu, J. (2022). *Solving Capacitated Vehicle Routing Problem by an Improved Genetic Algorithm with Fuzzy C-Means Clustering*. Scientific Programming, 2022, pp.1–8. doi:https://doi.org/10.1155/2022/8514660.

[31] Tam, V. and K.T. M (2008). An Effective Search Framework Combining Meta-Heuristics to Solve the Vehicle Routing Problems with Time Windows. doi:https://doi.org/10.5772/5638.

10. Popis slika

11. Popis tablica