Algoritam računalne inteligencije za rješavanje problema putujućeg lopova

Kvesić, Božo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Božo Kvesić

Algoritam računalne inteligencije za rješavanje problema putujućeg lopova

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Božo Kvesić

Matični broj: 45866/17–R

Studij: Informacijski sustavi

JMBAG: 0016129614

Algoritam računalne inteligencije za rješavanje problema putujućeg lopova

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Nikola Ivković

Varaždin, rujan 2020.

Božo Kvesić

Izjava o izvornosti

Izjavljujem da je moj završni/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 putujućeg lopova. 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 s 51, 76, 100 i 280 gradova te od 1 do 5 predmeta u svakom pojedinom gradu osim početnom. Cilj istraživanja bio je postići napredak algoritma kroz vrijeme, odnosno njegovo učenje i konstrukciju optimalnog rješenja. Problem putujućeg lopova je noviji višekomponentni problem koji je približan stvarnim problemima s kojima se tvrtke danas susreću. Problem je kombinacija problema trgovačkog putnika (TSP) i problema naprtnjače (KP) koji su veoma poznati u svijetu optimizacije. Cilj rada je rješavanje NP-teškog problema koji standardnim matematičkim metodama nije rješiv u realnom vremenu, a korištenjem mravljeg algoritma ćemo dobiti približno optimalno rješenje koje će biti prihvatljivo. Sergey Polyakovskiy predstavlja prve primjerke problema putujućeg lopova. Algoritam na kojemu su se vršili eksperimenti je testiran s 20 mrava i 1000 iteracija. Nakon provedenih eksperimenata došao sam do zaključaka kako MMAS algoritam relativno dobro napreduje kroz iteracije te konstruira bolja rješenja kako vrijeme prolazi. Lokalna optimizacija uvelike doprinosi napretku algoritma na način da u početku pomaže algoritmu konstruirati bolja rješenja kako bi u kasnijim iteracijama algoritam bolje učio. Loša karakteristika lokalne optimizacije je usporavanje izvođenja samog algoritma, ali s obzirom na dobivena kvalitetnija rješenja onda možemo zanemariti tu činjenicu.

Ključne riječi: optimizacija, računalna inteligencija, NP-težak problem, programiranje, algoritam

Sadržaj

1. Uvod

U ovom radu ćemo implementirati algoritam računalne inteligencije koji rješava problem putujućeg lopova. Koristiti ćemo mravlji algoritam (eng. ant colony algorithm) na kojemu ćemo kasnije provoditi eksperimente na različitim primjercima problema putujućeg lopova te prikazati i analizirati rezultate. Algoritam funkcionira na temelju računalne inteligencije na način da u početku postoji jednaka vjerojatnost za uzimanje svakog predmeta u svakom gradu, a onda kasnije, nakon određenog broja puštenih mravi se ta vjerojatnost povećava za pojedine predmete, odnosno smanjuje za ostale ako predmeti nisu isplativi. Provođenjem većeg broja povećavanja i smanjivanja bi trebali kasnije dobiti mrava koji će odabrati optimizirani put uz minimiziranje troška puta, a maksimiziranje profita. Razni grafovi će opisivati računalnu inteligenciju, odnosno algoritamsko učenje kroz vrijeme i broj puštenih mrava. Ovisno o toj krivulji ćemo podešavati početne parametre kako bi ta krivulja izgleda što je bliže moguće krivulji učenja.

Cilj ovog rada je načiniti algoritam optimiranja kolonijom mrava u aplikaciji za konzolu napisanoj u jeziku C++ te komentirati rješenja dobivena ovim algoritmom za problem putujućeg lopova za određene parametre, poput, broja mrava, jačine isparavanja feromona, utjecaj ograničavanja jačine feromona te još mnogi drugi parametri koji bi mogli utjecati na konačno optimalno rješenje.

Aplikacija koja je izrađena u svrhu ovog rada može izvoditi mravlji algoritam na zadanom broju kolonija mrava te zadanom broju mrava u pojedinoj iteraciji uz predefinirane postavke koeficijenta smanjenja, odnosno isparavanja feromonskih tragova te koeficijenta nagrađivanja i definiranje maksimalne te minimalne vrijednosti feromonskog traga za MMAS problem. Koristi funkcijski pristup uz korištenje *headera* te je zahtjevna za izvođenje u slučaju većeg broja gradova i predmeta po gradovima. Radi se o aplikaciji za konzolu, a napisana je u jeziku C++, u okruženju Microsoft Visual Studio 2019.

Eksperimenti koji se vrše, gledaju različite instance istog problema koji zadaju različite početne konfiguracije. Učinkovitost pojedinog mrava u koloniji se mjeri po ukupnoj sumi svih profita „ukradenih“ predmeta umanjenoj za troškove puta i troškove najma ruksaka. Također, uspoređivati ćemo rješenja problema koristeći lokalnu optimizaciju s rješenjima problema gdje ju nećemo koristiti pa ćemo ovisno o rezultatima donijeti i analizirati zaključke.

Rad čini pet poglavlja. Od toga prvi će sadržavati opis klase NP-problema, drugi će sadržavati pojedinosti vezano za mravlji algoritam, potom slijedi opis problema, nakon toga slijedi analiza koda te njegovih najznačajnijih dijelova zbog kojih se taj algoritam zaista može

nazvati mravljim algoritmom i kao posljednje poglavlje će biti analiza rješenja dobivenih tim algoritmom uz zaključak na kraju.

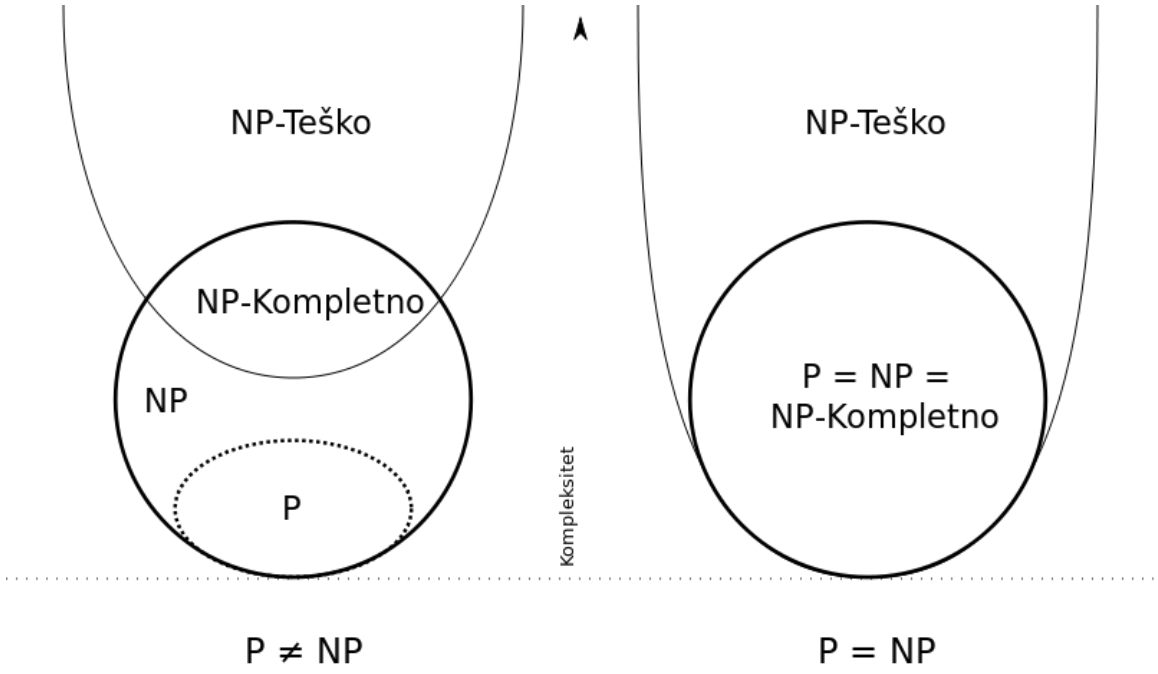
Dobivamo li slična ili različita rješenja puštanjem istog algoritma s istim predefiniranim postavkama? Koliko je predstavljeno programsko rješenje dobro u usporedbi sa razvijenim znanstvenim algoritmima za ovaj problem? Zbog čega koristimo ovakve algoritme kod ovakvih problema? Ovo su samo neka od pitanja na koje će ovaj rad pokušati dati odgovor.

2. NP problemi

Da bismo se osvrnuli na navedeni pojam, potrebno je prvo navesti pojam klasa složenosti unutar kojih smještamo razne algoritme ovisno određenoj mjeri njihove složenosti. Složenost algoritma možemo promatrati prostorno (količina memorijskog prostora koja je potrebna da se izvrši operacija nad operandima na zadanom izvršitelju), te vremenski (broj vremenskih jedinica ili koraka/taktova potrebnih da se algoritam izvrši na zadanom izvršitelju). Posebice se proučavaju vremenske mjere složenosti jer one u današnje vrijeme predstavljaju velik problem te onemogućuju da algoritam iznese rezultat u zadovoljavajućem prihvatljivom vremenu. Unatoč ogromnim brzinama izvođenja operacija koje nude današnji procesori (čak do 10^7 operacija po sekundi), neki algoritmi i problemi ne mogu biti riješeni u kratkom vremenu, već postaju neodgonetnuti u praksi na suvremenim računalima u realnom vremenu. Kad nailazimo na mjeru složenosti određenog algoritma, najčešće se misli na njegovu kompleksnost u najgorem mogućem slučaju.

Klasa *NP* (*nondeterministic polynomial time*) zapravo predstavlja skup problema odlučivanja rješivih u polinomskom vremenu na nedeterminističkom Turingovom stroju (Turingovom stroju koji ima mogućnost prijelaza u više stanja, ovisno o isplativosti i potencijalnom ishodu ako se krene određenim putem, a to se pak određuje pomoću stabala i grafova). Ekvivalentno, to je skup problema čija se rješenja mogu provjeriti na determinističkom Turingovom stroju u polinomskom vremenu. Većinu algoritama iz klase *NP* je za sada moguće optimizirati, te im povećati performanse. Tako s obzirom na metodologiju dizajna, algoritme možemo podijeliti na *brute force* algoritme, *podijeli i vladaj* algoritme, dinamičke algoritme, pohlepne (*greedy*) algoritme, te algoritme za sortiranje i pobrojavanje. Od navedenih algoritama prva 3 daju točan rezultat nakon određenog vremena, četvrti ne vraća precizan rezultat, već aproksimativan rezultat koji je u određenim slučajevima neprihvatljiv, te peti koji većinom daju zadovoljavajuće rezultate, a po potrebi se mogu dodatno optimizirati primjenom teorije grafova. Kao što je spomenuto, većina algoritama iz klase *NP* je moguće poboljšati, te preostaje mogućnost da će s vremenom i napretkom znanosti biti

svedene na *P* razinu (*deterministic polynomial time*). Međutim, preostale algoritme ubrajamo u klasu algoritama za *NP*-teške probleme koji će, i u slučaju da se dokaže *P=NP*, dalje vrlo vjerojatno ostati izvan nje. U te probleme spadaju SAT problem, N-slagalica, problem ruksaka, problem Hamiltonovog ciklusa, problem trgovačkog putnika, problem izomorfizma podgrafa, problem sume podskupa i još brojni drugi. [1]



Slika 1: Eulerov dijagram za P, NP, NP-kompletne i NP-teške skupove problema. Postojanje problema u klasi NP ali van P i NP-kompletne klase je pod ovom pretpostavkom osnovao Ladner.

(Izvor: https://bs.math.org)

Algoritamski nerješivi problemi

Prethodna dva stoljeća donose veliki zamah znanosti, pa tako i matematike - otkrivaju se sve jači algoritmi koji rješavaju sve općenitije probleme. Veliki san i nadu matematike izrazio je njemački matematičar i filozof Gottfried Wilhelm Leibniz na prelasku u 18. stoljeće: cilj je pronaći algoritam nad algoritmima, koji rješava sve matematičke i filozofske probleme. Otkriće takvog algoritma značilo bi vrhunac, pa možda i sam kraj matematike. No, 1936. američki matematičar Alonzo Church dokazao je da takvog algoritma nema. Nakon toga događaja mnogi su problemi identificirani kao algoritamski nerješivi. Algoritamski nerješivi problemi (zovu se još i neodlučivi problemi (*undecidable problems*)) se ne mogu

riješiti računalom jer je problem i samo algoritamsko rješenje nemoguće formalizirati standardnim matematičkim jezikom. S druge strane, algoritme algoritamsko rješivih problema je moguće opisati i to na razne načine: pomoću govornog jezika, algoritamskom shemom (blok-dijagramom), pseudokodom, pomoću programskog jezika, itd.. Najpoznatiji algoritamski nerješivi problemi su problem zaustavljanja Turingovog stroja, deseti Hilbertov problem, problem ekvivalentnosti riječi u asocijativnom računu, te problem izomorfizma za grupe.[2]

3. Heuristički i metaheuristički algoritmi

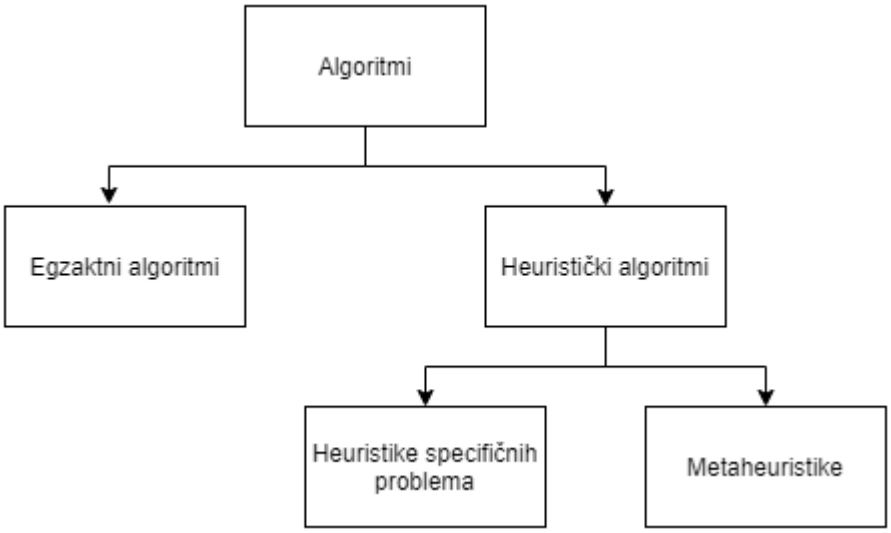
Jedan od način pristupa problemima koji spadaju u NP-teške i NP-potpune probleme je poboljšanje sklopovskih performansi računala. Dobar primjer za to je kvantno računalo koje bi zbog svojih sklopovskih performansi i mogućnosti, moglo riješiti probleme eksponencijalne složenosti u razumnom vremenu. Kvantno računalo kao takvo još uvijek ne postoji, ali se nadamo da će u bliskoj budućnosti biti razvijeno. Kako trenutno ne postoji računalo koje bi takav tip problema riješio u razumnom vremenu onda nam ne preostaje ništa drugo nego pristupiti problemu na drukčiji način. Inženjerska praksa nas uči da često nije potrebno riješiti probleme egzaktno, tj. dovoljno ih je riješiti približno. U tu svrhu koristimo se nekakvim iskustvenim metodama čija je učinkovitost eksperimentalno potvrđena. Neke od tih metoda su i heuristički algoritmi. Riječ heuristika potječe od grčke riječi heurisko što znači pronašao sam. Odavde se da naslutiti da su heuristički algoritmi zapravo algoritmi nastali eksperimentiranjem u svrhu dobivanja zadovoljavajućeg rješenja. Bitno svojstvo heurističkih algoritama je da mogu približno (dovoljno dobro) riješiti probleme eksponencijalne i faktorijelne složenosti. Heuristički algoritmi su algoritmi koji ne daju najbolje rješenje, ali mogu dati dobro rješenje u nekom vremenskom razdoblju. Primjenjuju se za pronalaženje okvirnih rješenja kada konvencionalne metode zakažu ili su prespore. Takvi algoritmi imaju relativno nisku računsku složenost, radi se o polinomskoj složenosti i ne jamče da će uspjeti pronaći optimalno rješenje. Često se zbog brzine gubi na kvaliteti, preciznosti i potpunosti rješenja. [4] Ipak, valja napomenuti da rješavanje problema heurističkim algoritmima ne mora voditi k zadovoljavajućem rješenju, a za neke probleme, heuristički algoritmi pokazuju relativno loše rezultate. Isto tako, heuristički algoritmi nisu jednoznačno određeni. Pojedini dijelovi heurističkih algoritama se razlikuju ovisno o situaciji u kojoj se koriste. Ti su dijelovi uglavnom funkcije cilja (transformacije) i njihovo definiranje znatno utječe na efikasnost algoritma. [3] Heuristički algoritmi mogu se podijeliti na dvije skupine:

Konstrukcijski algoritmi: rješenja se grade korak po korak te na kraju konstruiraju cjelokupno rješenje. Brzi su, jednostavni i usko vezani uz problem koji rješavaju.[3]

Algoritmi lokalne pretrage: najčešće se javljaju uz neki konstrukcijski algoritam i iterativnim postupkom pokušavaju poboljšati dobivena rješenja. Ovi su algoritmi za razliku od konstrukcijskih mnogo sporiji, ali daju bolje rezultate. [3]

Heuristika je pravilo temeljeno na našem iskustvu pomoću kojeg mi tražimo rješenje nekog problema. Pomoću heuristike se mogu pronalaziti optimalna rješenja, ali i procjenjivati odluke. Heuristički algoritmi se temelje na heuristici i uglavnom se koriste u rješavanju optimizacijskih problema za čije rješavanje nisu poznati algoritmi polinomske složenosti. Glavna snaga heurističkih algoritama je da smanjuju prostor pretrage koristeći neke iskustvene spoznaje, te time znatno ubrzavaju proces pronalaženja rješenja. No, to znači i da heuristički algoritmi ne moraju uvijek dati optimalno rješenje, a ponekad njihovo izvođenje može trajati duže od algoritama koji koriste egzaktne metode rješavanja. [5]

.

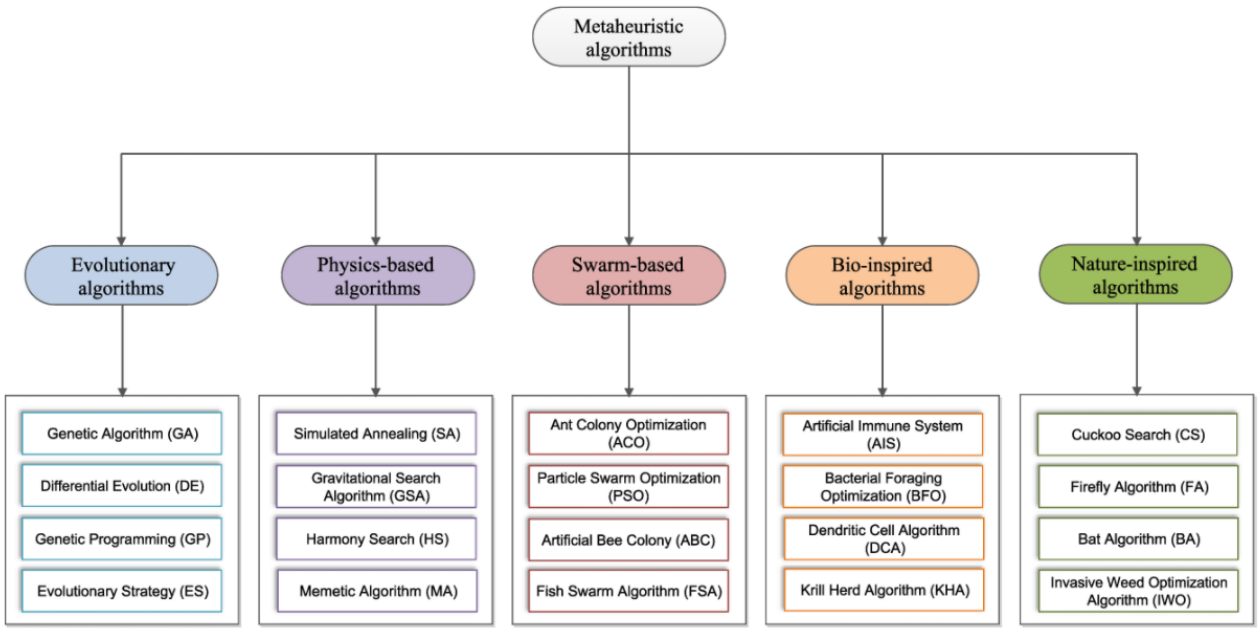


Slika 2: Podjela algoritama (Izvor: Radanović G., 2007.)

Na slici 2. možemo vidjeti podjelu algoritama na egzaktne i heurističke algoritme. Egzaktni algoritmi uvijek daju optimalno rješenje, ali za NP-teške probleme su primjenjivi na jako malom broju primjera zbog složenosti tih problema. Ovakvi algoritmi nam uvijek daju isto rješenje prilikom njihovog pokretanja. [6] Heuristički algoritmi ne moraju uvijek davati optimalno rješenje, nego daju približno optimalno rješenje. Glavna snaga heurističkih algoritama je da smanjuju prostor pretrage koristeći neke iskustvene spoznaje, te time znatno ubrzavaju proces pronalaženja rješenja. [4]

Metaheuristički algoritmi obuhvaćaju širok skup algoritama namijenjenih optimizacijskim problemima. Sadržavaju skup algoritamskih koncepata koji se primjenjuju za definiranje heurističkih metoda na način da iterativnim postupkom pokušavaju poboljšati postojeće rješenje s obzirom na neku evaluacijsku funkciju. Oni su se pokazali izuzetno dobri u rješavanju NP-teških problema: problema nerješivih egzaktnim algoritmima u stvarnom vremenu. Metaheurističke algoritme možemo podijeliti po različitim osnovama, a najvažnije podjele su:

- Prirodom-inspirirani algoritmi i algoritmi koji nisu prirodom inspirirani.
- Algoritmi koji imaju dinamičku funkciju objekta i algoritmi koji imaju statičku funkciju objekta.
- Algoritmi koji koriste jednu strukturu okoline i algoritmi koji koriste skup struktura okoline
- Algoritmi koji imaju mogućnost pamćenja prethodnih rješenja i oni koji to nemaju
- Konstruktivni, poboljšavajući i hibridni algoritmi
- Algoritmi bazirani na populaciji rješenja i algoritmi putanje [4]



Slika 3: Podjela metaheurističkih algoritama (Izvor: G. Dhiman i V. Kumar, 2007. , https://bit.ly/2EZgvuy)

Na slici 3 možemo vidjeti još jednu podjelu metaheurističkih algoritama koja ih dijeli u pet kategorija te pojedine primjere algoritama razvrstanih u tih pet kategorija. Ovo ovdje nisu svi metaheruistički algoritmi jer ih postoji puno više od ovdje 20 nabrojanih, ali ovo ovdje su primjeri za pojedine kategorije tih algoritama koji koriste slične karakteristike izvršavanja po kojima se i svrstavaju u njih.

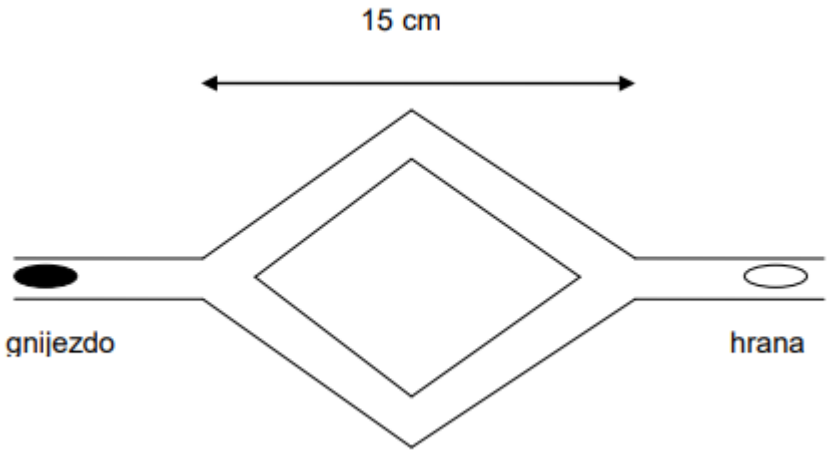
4. Mravlji algoritam

Mravi su poluslijepe životinje te prilikom kretanja ne koriste osjetilo vida, već je njihovo kretanje uvjetovano socijalnom interakcijom među jedinkama. U prirodi, mravi slobodno lutaju, a nakon što pronađu izvor hrane te se krenu vraćati u gnijezdo, za sobom ostavljaju hormonski trag, feromone.[7] Mravlji algoritam je algoritam nastao proučavanjem stvarnog života mrava te je po tome dobio naziv. Mravi u svojoj prirodnom okruženju žive u kolonijama koje su veoma organizirane pa tako prilikom odlaska po hranu ostavljaju feromonske tragove koje ostali mravi mogu osjetiti. Prolaskom većeg broja mrava istim putem pojačava se feromonski trag za ostale mrave, a isto tako feromonski trag isparava u slučaju da mravi ne prolaze često tim putem. Upravo taj način se primjenjuje u mravljem algoritmu koji u početku pušta određen broj „mrava“ (u algoritmu se mravi smatraju izvršenje jednog izračuna puta te usporedba njegovog rješenja s ostalim izračunima puta) te svaki taj mrav odabire svoj put, a potom algoritam nagrađuje određenog mrava ovisno o vrsti mravljeg algoritma na način da tragove odabira puta na putu kojim je on prošao poveća za određeni delta_tau[1]. Iduća iteracija mrava također dopušta da svaki mrav bira svoj put, ali uzimajući u obzir tragove koje ako povučemo crtu sa stvarnim životom predstavljaju feromonske tragove. Što više mrava odabere isti ili sličan put, veća je vjerojatnost idućeg mrava da će odabrati taj put. Nakon određenog broja puštenih mrava, algoritam ispisuje najbolje pronađeno rješenje koje može biti i optimalno. Postavlja se pitanje je li pronađeno rješenje zaista optimalno? Općenito ne možemo znati je li rješenje optimalno, to se može znati ako je problem umjetno konstruiran na način da je poznato optimalno rješenje ili ako je moguće nekom metodom pokazati da bolje rješenje nije moguće. U slučaju da se radi o algoritmu koji ima velike skokove i padove onda je potrebna modifikacija određenih postavki kako bi algoritam bolje radio. Mravlji algoritam nema svoj jedinstven kod nego je mravlji algoritam jedinstven za svaki problem, odnosno mogli bi reći kako je mravlji algoritam koncept koji zahtjeva određene elemente kako bi se takav mogao nazvati, ali je na programeru velika sloboda i zadatak da odabere recept po kojemu će mravlji algoritam funkcionirati za njegov problem. Mrav se kreće slučajno, ali s većom vjerojatnošću u smjeru u kojem osjeti jači feromonski trag. Spomenuli smo to kod metaheurističkih algoritama koji definiraju funkciju cilja ovisno o problemu te je funkcija cilja jedinstvena za svaki problem. Na svim putovima gdje je barem jedan mrav prošao postojati će feromonski trag od gnijezda do hrane i s vremenom će se istaknuti najjači trag koji će svi mravi slijediti. Pokus koji najbolje predstavlja proučavano ponašanje mrava su opisali pomoću pokusa s dvostrukim mostom. [7]

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

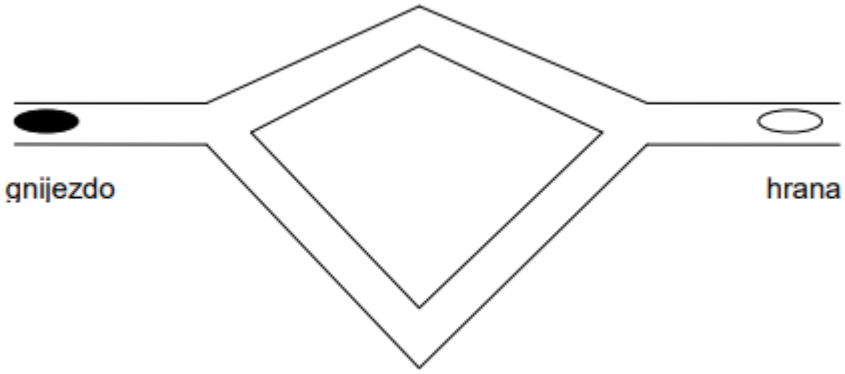
Eksperiment s dvokrakim mostom

Mravlji algoritam (eng. *ant colony optimization*, ACO) je inspiriran pokusom Deneborgha i njegovih suradnika u kojemu su koristili most sa dva grananja koji spaja mravinjak s hranom. Na taj su način pratili trag feromona koji su mravi ostavljali za sobom. U eksperimentu dvokrakog mosta, mravinjak argentinskih mrava povezan je s hranom s dva mosta. Mravi mogu doći do hrane i vratiti se koristeći bilo koji od ta dva mosta. Argentinski mravi specifični su po tome što ostavljaju trag feromona u oba smjera, dakle u smjeru kretanja prema hrani i vraćanja prema gnijezdu. Cilj eksperimenta je bio proučiti ponašanje kolonije. Provodili su eksperiment na način da su varirali omjer $r = \frac{l_l}{l_s}$ kao udaljenost između hrane i gnijezda, gdje je bila l_l duljina dulje grane, a l_s kraće grane. [7]



Slika 4: Prvi eksperiment (Izvor: M.Dorigo i T.Stutzle, 2004.)

Prvi eksperiment je proveden na način da su l_l i l_s bili jednake duljine, odnosno r=1 te su kao rezultat dobili da su se mravi u podjednakom broju kretali na oba kraka mosta. Na određenom kraku mosta je u jednom trenutku prošla veća skupina mrava te je ostavila jače feromonske tragove nakon čega je taj krak postao dominantniji. Kako je udaljenost kraćeg i duljeg kraka bila jednaka kao što možemo vidjeti na slici 4, onda su znanstvenici odlučili isti eksperiment ponoviti veći broj puta te su došli do zaključka kako je u prosjeku mravi u 50% slučajeva biraju jedan krak, a u 50% slučajeva biraju drugi krak.

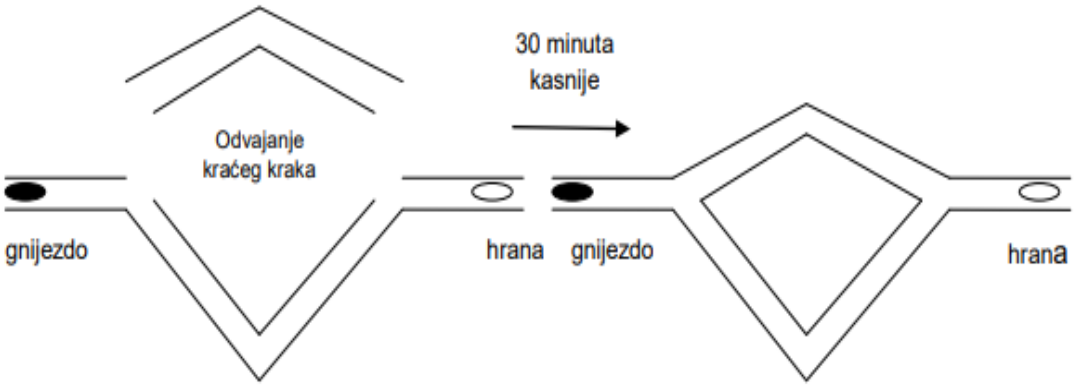


Slika 5: Drugi eksperiment (Izvor: M.Dorigo i T.Stutzle, 2004.)

Drugi eksperiment je proveden na način da je jedan krak mosta bio dvostruko dulji od drugog, a mravi su opet imali slobodnu volju odabira puta do hrane, odnosno nije bilo dodatnih prepreka koje bi utjecale na odabir puta mrava osim duljine krakova mosta. Na slici 5 je prikazana skica drugog eksperimenta gdje je krak jednog mosta dvostruko dulji od kraka drugog mosta. U početku su kao i kod prvog eksperimenta mravi birali put nasumično, ali se nakon većeg broja puštenih mrava pokazalo da je većina mrava odabrala najkraći put, u našem slučaju je to put kraćim krakom i povratak istim.

Ako u obzir uzmemo duljinu krakova, onda možemo zaključiti kako su mravi koji su odabrali kraći put prvi stizali do hrane i prvi se vraćali u gnijezdo zbog čega su ostavljali feramonski trag prije ostalih mrava. Veća količina feromona na kraćoj relaciji natjerala je mrave da se kreću tim putem. Rezultat je pokazao da feromon brže akumulira na kraćoj relaciji što navodi većinu mrava da se kreću upravo u tom smjeru.

Jedna od interesantnih zaključaka do koji su znanstvenici došli je ta da neće svi mravi birati najkraći put do hrane iako je jedna grana dvostruko dulja od druge. 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. Deneborgh i suradnici su ovakvo ponašanje interpretirali kao „istraživanje puta“. [7]



Slika 6: Treći eksperiment (Izvor: M.Dorigo i T.Stutzle, 2004.)

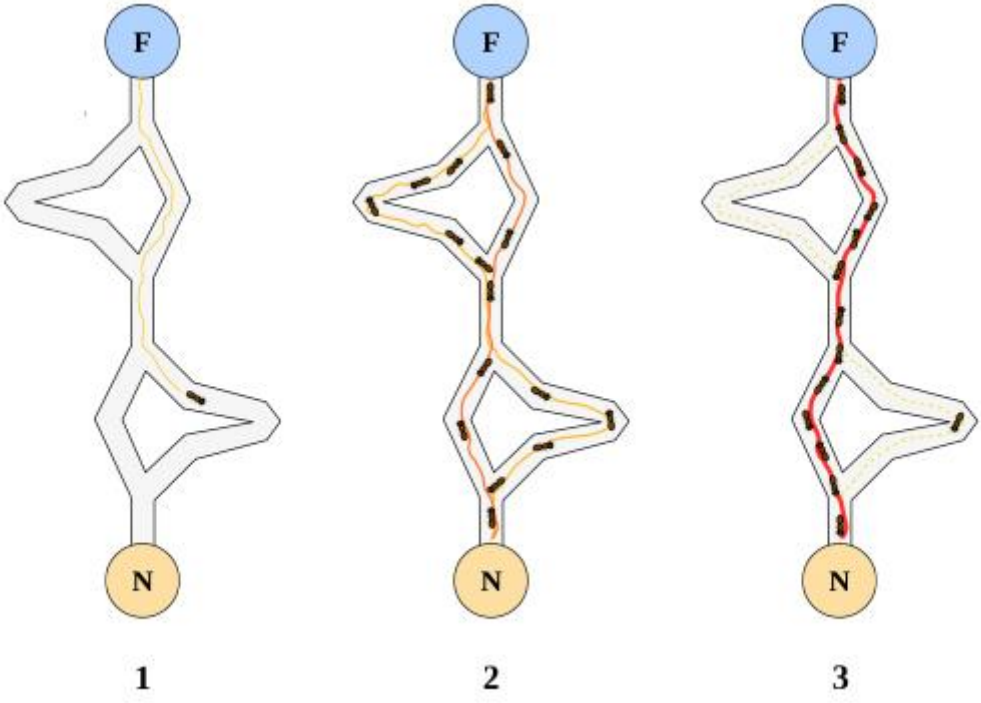
Treći eksperiment je proveden na način da su mravinjak i izvor hrane prvotno bili povezani samo jednim krakom mosta, a potom je kasnije nadodan drugi krak mosta. Početni krak mosta je bio dulji od onoga koji je kasnije nadodan kao što je prikazano na slici 6. U početku su mravi imali slobodnu volju kretanja mostom, ali kako je tada postojao samo jedan krak mosta, mravi su se kretali samo njime. Nakon 30 minuta, mostu je dodan drugi kraći krak, ali, mravi su se nastavili kretati duljim putem jer je na tom putu bila veća koncentracija feromona. Znanstvenici su došli do zaključka kako je u ovom slučaju hlapljenje feromona bilo presporo što je uskratilo mravima mogućnost odabira kraćeg kraka mosta te istraživanja novih puteva. Ovdje se dokazalo da feromonski trag sporo hladi te na taj način dozvoljava mravima da s vremenom zaborave suboptimalne puteve kojima teže na način da se novi putevi mogu „naučiti“ i istražiti. [7]

Istraživanje stvarnih mravljih kolonija potvrdila su da mravi koji ostavljaju feromonski trag samo kada se vraćaju od hrane u gnijezdo nisu u mogućnosti pronaći najkraći put između njihovog gnijezda i izvora hrane. [7] Goss je razvio model opisanog ponašanja: uz pretpostavku da je, u danom trenutku, m_1 mrava odabralo prvi most i m_2 mrava odabralo drugi most, vjerojatnost p_1 da mrav odabere prvi most je

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

gdje su parametri k i h dobiveni eksperimentalno (očito je $p_2 = 1 - p_1$). Monte Carlo simulacija je pokazala da je dobar odabir $k \approx 20$ i $h \approx 2$. Ova jednadžba je poslužila kao inspiracija za Mravlji sistem, prvi ACO algoritam [8].

Eksperiment s dvostrukim mostom



Slika 7: Skica dvostrukog mosta (Izvor: A. Saxena i C. Mueller, 2018.)

Proveden je još jedan eksperiment u kojemu je napravljen jedan put koji je sadržavao dva mosta od kojih se svaki sastojao od duljeg i kraćeg kraka. U početku su mravi opet imali slobodnu volju odabira puta te kao što možemo vidjeti na primjeru 1 sa slike 7, mrav odabire bilo koji put jer u početku ne postoje feromonski tragovi između točke N i točke F. Nakon većeg broja puštenih mravi vidljivo je da mravi koji odabiru kraći put se brže vraćaju od točke F do točke N pa zbog toga brže ostavljaju i feromonski trag koji ostali mravi prate. Sve veći broj mrava osjeti jači feramonski trag na kraćim krakovima mosta te se krenu kretati time i na taj način pronalaze najkraći put. Na primjeru 3 možemo vidjeti da se većina mrava kreće najkraćim mogućim putem prilikom odlaska po hranu, ali je također i vidljiv jedan mrav koji i dalje odabire duži put. Upravo taj mrav je mrav koji po znanstvenicima istražuje nove puteve, ali zbog sporog ishlapljivanja feromonskih tragova i jakog feramonskog traga na najkraćem putu, taj put neće biti često izabran jer nije optimalan. Razvoj ACO algoritma je potaknut upravo ovim eksperimentima zbog kojih je vidljivo da cijela kolonija uspijeva pronaći optimalni put do hrane. Upravo sve ove značajke feromonskih tragova, odabira puta po jačini feromona te ishlapljivanje feromona su uzeti i primjenjeni na mravlji algoritam. [7]

Karakteristike mrava

Kao što je ranije navedeno, eksperimenti s dvostrukim mostom i eksperimenti s dvokrakim mostom u kojima su korišteni biološki mravi, odnosno vrsta argentinski mravi su doveli do razvoja ideje za mravlji algoritam ili ACO. Ključne značajke bioloških mrava koje su pomogle razvoju mravljeg algoritma su:

1) kolonije mrava

2) trag feromona koji služi kao putokaz ostatku kolonije

3) pronalaženje najkraćeg puta

4) stohastičke odluke [7]

1) Biološki mravi žive u kolonija koje skupa surađuju te skupljaju hranu, a upravo je to značajna karakteristika mravljeg algoritma. Virtualni mrav ne mora odmah pronaći optimalno rješenje, nego se optimalno rješenje gradi suradnjom sa ostatkom kolonije koje na kraju daje rješenje visoke kvalitete. Svaki mrav može stvoriti moguće rješenje, odnosno kod bioloških mravi bi to značilo da svaki mrav može pronaći put od mravinjaka do gnijezda te se svako rješenje mrava uspoređuje i uzima u obzir, a samo ono najbolje nagrađuje. [7]

2) Trag feromona kod bioloških mrava je kemijska supstanca koju ostali mravi prepoznaju i kreću se, dok kod virtualnih mravi feromonski trag bi označavao tragove odabira posjećivanja pojedinih gradova i kupljenja pojedinih predmeta u gradovima. Virtualni mravi povećavaju te numeričke tragove po formuli $tau\_ij = tau\_ij + delta\_tau$ ako se radi o pronalasku puta koji je optimalan gdje tau_ij predstavlja feromonski trag ili tau trag za taj grad, a $delata\_tau$ konstantnu vrijednost kojom se nagrađuju kofecijenti, dok se funkcija isparavanja feromonski tragova računa po sljedećoj formuli $tau\_ij = (1 - rho) * tau\_ij$ gdje tau_ij predstavlja trenutni feromonski trag u tome gradu, a rho predstavlja vrijednost postotka u decimalnom zapisu za koju će se vrijednost traga smanjiti. Najčešće se u ACO algoritmima dodaje mehanizam isparavanja feromona, koji simulira pravo isparavanja feromonski tragova, a samo isparavanje omogućuje mravljoj koloniji zaboravljanje prošlih rješenja te istraživanja novih puteva bez utjecaja na prošle odluke. Ovaj mehanizam isparavanja feromonski tragova je koristan kod ispravljanja rješenja ako se veći broj mrava u početku počeo kretati putem koji nije optimalan te onda postoji mogućnost da se kroz određen broj iteracija rješenje ispravi, a feromonski tragovi pojačaju na tome novom putu koji bi mogao biti optimalan, dok kod virtualnih mravi bi to značilo povećavanje feromonskih tragova na tome novome putu.[7]

3) Glavni cilj mravljeg algoritma je pronaći optimalno rješenje, odnosno kod bioloških mravi to bi značilo pronalazak najkraćeg puta do hrane te povratka u gnijezdo. Biološki mravi istražuju korak po korak te ostavljaju feromonske tragove u slučaju pronalaska puta. Nakon većeg broja puštenih mravi u istraživanju puta do hrane, ostatak mravi prepoznaje optimalni put kojim se kasnije kreću kako bi što prije obavili odlazak po hranu te njezino kupljenje i povratak u mravinjak. [7]

4) Stohastičko odlučivanje podrazumijeva nepoznate i nejasne okolnost u kojima je donesena određena odluka te ju možemo usporediti s pojedinim mravom u potrazi za hranom koji donosi upravo takve odluke koje ga u konačnici dovode do hrane, a mi te odluke ne analizirano jer su one donesene u nepoznatim i nejasnim okolnostima, odnosno nije u potpunosti jasno zbog čega se određeni mrav kreće baš tim putem, osim u slučaju pojačanih feromonskih tragova kada su oni odgovor donošenja takvih odluka, ali smo i u eksperimentima vidjeli da i nakon pojačanog feramonskog traga na optimalnom putu, mravi odabiru duži put te smo tu pojavi nazvali „istraživanjem puteva“. Ovakve odluke podrazumijevaju korištenje informacija drugih mrava, odnosno njihovih feromonskih tragova, potom korištenje informacija o određenim preprekama te ne rabe mogućnost planiranja postupaka unaprijed za predviđanje budućih položaja. [7]

Vrste ACO algoritama

U ovom poglavlju ćemo opisati najuspješnije mravlje algoritme. To su mravlji sustav kojega je predstavio Dorigo 1992. godine, potom sustav kolonija mrava (ACS) kojega je također predstavio Dorigo , ali u suradnji s Gambardella-om 1997. godine i MAX-MIN mravlji sustav (MMAS) koji je predstavljen od Stützle-a i Hoos-a 2000. godine. Opisati ćemo sve potrebne korake za implementaciju takvog algoritma. Potom ćemo pokazati implementaciju jednog takvog algoritma.

4.1.1. Mravlji sustav

Mravlji sustav (eng. *ant system*, AS) bio je prvi mravlji algoritam (ACO algoritam) predložen u literaturi, a predstavio ga je Marco Dorigo 1992. godine. Prilikom kreiranja rješenja mrav primjenjuje slučajno-proporcijonalno pravilo odlučivanja da odluči koji će grad posjetiti sljedeći.

Detaljnije, vjerojatnost s kojom će mrav k, koji je trenutno u gradu i, posjetiti grad j, dana je s

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

gdje je $\eta_{ij=1/d_{ij}}$ heuristička vrijednost, a α i β su dva parametra koji odreduju relativan utjecaj traga feromona i heurističke vrijednosti. N_i^k su neposjećeni gradovi mrava k kada se nalazi u gradu i. Vjerojatnost odabira nekog grada izvan N_i^k je 0. Prema pravilu 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 najbliži gradovi, što odgovara klasičnom stohastičkom pohlepnom algoritmu (s višestrukim počecima, jer su mravi, na početku, slučajno distribuirani po gradovima). Ako je $\beta = 0$, na izbor utječe jedino vrijednost feromona, bez heuristike. Za vrijednosti $\alpha > 1$, konstrukcija rješenja mrava ubrzo dolazi do stagnacije, to jest, do situacije u kojoj svi mravi slijede isti put i konstruiraju isto rješenje, koje je (vrlo često) jako loše.

Glavna karakteristika mu je u tome što su vrijednosti feromona ažurirane od svih mrava koje su završile obilazak. Isparavanje feromonskog traga implementira se po sljedećoj formuli

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

gdje je $0 < \rho \leq 1$ stopa isparavanja feromona. Parametar ρ se koristi da se izbjegne neograničeno povećanje traga feromona i omogućuje algoritmu da zaboravi na loše odluke koje je donio te kako bi mogao istraživati druga rješenja koja su možda bolja od trenutnih.

Ukoliko mravi ne odabiru određeni grad u nekoliko iteracija, vrijednost feromona se smanjuje eksponencijalno u broju iteracija. Poslije isparavanja, svi mravi ispuštaju feromone na bridove koje su posjetili u svojoj turi prema sljedećoj 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 mrav k odloži u gradove koje je posjetio, odnosno poveća im vrijednost feromonskog traga.

Što je tura bolja, to njezine komponente poprime više feromona. Općenito, lukovi između gradova kojima je prošlo puno mrava i koji su dio kratkih tura, poprime više feromona i time je vjerojatnije da će ih mravi u budućnosti odabrati. [8]

4.1.2. Max-min mravlji sustav (MMAS)

Max-Min mravlji sustav (eng. *max-min ant system*, MMAS) uvodi određene modifikacije u algoritam Mravljeg sustava. Algoritam se fokusira na najbolje rješenje u iteraciji ili najbolje rješenje do sada može povećati feromonske tragove, odnosno koeficijente koji predstavljaju feromonske tragove u algoritmu. [9]

Negativna strana ovakvog načina nagrađivanja rješenja je ta da može doći do stagnacije u kojoj će svi mravi konstruirati isto rješenje, odnosno to se dešava jer se u početnim iteracijama nagrađuje najbolje rješenje koje nije optimalno te ostali mravi krenu koristiti dijelove te konstrukcije misleći da su upravo ti dijelovi koji tvore optimalno rješenje. Upravo kako bi se to izbjeglo koristi se ograničavanje moguće vrijednosti feromonskih tragova na interval:

$$[\boldsymbol{\tau}_{min}, \boldsymbol{\tau}_{max}].$$

Ograničavanje vrijednosti feromonskih tragova je karakteristika MMAS algoritma. Na početku algoritma, svi feromonski tragovi su postavljeni na gornju granicu vrijednosti feromoskog traga, odnosno $\boldsymbol{\tau}_{max}$. Ova karakteristika omogućuje mravima povećano istraživanje puteva na početku pretraživanja te se time smanjuje šansa za kreiranjem loših rješenja u startu algoritma koje bi kasnije mogle dovesti do stagnacije. Uz početno postavljanje na $\boldsymbol{\tau}_{max}$, MMAS algoritme karakterizira mali postotak isparavanja svih feromonskih tragova te zajedno stvaraju mogućnost bolje konstrukcije rješenja u startu. Trag feromona se ponovo inicijalizira, ako se primijeti da je došlo do stagnacije ili da nema poboljšanja u određenom broju uzastopnih iteracija. Isparavanja feromonskih tragova se primjenjuje po formuli napisanoj kod Mravljeg sustava, a dodavanje novih feromona se implementira na sljedeći način:

$$\boldsymbol{\tau}_{ij} = \boldsymbol{\tau}_{ij} + \Delta\boldsymbol{\tau}_{ij}^{\boldsymbol{najbolje}}$$

gdje je $\Delta\boldsymbol{\tau}_{ij}^{\boldsymbol{najbolje}} = \mathbf{1}/\mathbf{C}^{\boldsymbol{najbolje}}$. Kao što smo ranije naveli, moguće je da najbolji mrav od početka poveća koeficijente feromonskih tragova na svome putu, i u tom slučaju je $\Delta\boldsymbol{\tau}_{ij}^{\boldsymbol{najbolje}} = \mathbf{1}/\mathbf{C}^{\boldsymbol{najbolje\ trenutno}}$, ili najbolji mrav u trenutnoj iteraciji, pa je $\Delta\boldsymbol{\tau}_{ij}^{\boldsymbol{najbolje}} = \mathbf{1}/\mathbf{C}^{\boldsymbol{najbolje\ u\ koloniji}}$. U implementacijama MMAS se koriste oba pravila ažuriranja (najbolji dosad ili najbolji u iteraciji) i to ovisno o vrsti problema. Korištenje jednog češće od drugog određuje koliko je algoritam pohlepan. Znanstvenici su dokazali da je za male TSP probleme najbolje koristiti samo ažuriranje feromona od mrava najboljeg u iteraciji, a za velike TSP probleme, sa stotinama gradova, najbolje performanse su dobivene tako da se povećava utjecaj najboljeg dosad. [8]

4.1.3. Sustav mravlje kolonije

Sustav mravlje kolonije (eng. *ant colony system*, ACS) razlikuje se od prva dva opisana algoritma u nekoliko karakteristika. Sustav mravlje vrši isparavanje i odlaganje feromona samo na najboljoj turi dosad. Druga bitna karakteristika je postojanje lokalnog ažuriranja feromona, odnosno svaki put kada mrav prilikom konstrukcije rješenja pređe u idući nasumično odabrani grad, on izbriše dio feromonskih tragova u tome grad kako bi povećao istraživanje drugih puteva. Ovaj algoritam se ponaša suprotno mravima u prirodi jer mravi u prirodi nisu u mogućnosti brisati prethodno ostavljene feromonske tragove.

Najvažnije je primijetiti da se ažuriranje traga feromona u ACS, pritom mislimo na isparavanje feromona i odlaganje novih feromona, odnosi samo na bridove iz $\mathbf{T}^{najbolje\ trenutno}$, a ne na sve bridove, kao u AS. Uz globalno ažuriranje traga feromona koje se događa samo na najboljem rješenju do sada, u ovom algoritmu mravi koriste i lokalno ažuriranje feromonskih tragova, i to odmah nakon što mrav prijeđe u idući grad tijekom konstrukcije ture:

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

gdje su $\boldsymbol{\xi}$, $\mathbf{0} \leq \boldsymbol{\xi} \leq \mathbf{1}$, i $\boldsymbol{\tau_0}$ dva parametra. Vrijednost $\boldsymbol{\tau_0}$ je postavljena da bude ista kao početna vrijednost traga feromona. Posljedica lokalnog ažuriranja feromona je da svaki put kad mrav koristi određeni grad, trag feromona tog grada $\boldsymbol{\tau_{ij}}$ je smanjen pa taj grad postaje manje poželjan da ga koriste budući mravi u svojoj konstrukciji. Ovim lokalnim ažuriranjem feromonskih tragova se omogućuje istraživanje puteva koji još nisu posjećeni te to ima efekt da algoritam ne dolazi u fazu stagnacije, odnosno mravi u kasnijoj fazi konstruiraju rješenja koja se bitno razliku od ostalih mravi te na taj način doprinose pronalasku optimalnog puta. Važno je primijetiti da za prijašnje varijante algoritma Mravljeg sustava nije bilo bitno ako mravi konstruiraju turu paralelno ili sekvencijalno. To se mijenja u algoritmu Sustava mravlje kolonije, zbog lokalnog ažuriranja feromona. Sekvencijalno konstruiranje rješenja znači da mrav prvo konstruira cijelo rješenje te nakon njega kreće idući mrav, dok paralelna konstrukcija označava da se mravi zajedno kreću u svaki idući grad te paralelno konstruiraju rješenja. [8]

4.1.4. Sustav mrava s trima granicama

Radi poboljšanja pojedinih svojstava MMAS algoritma osmišljen je Sustav mrava s trima granicama (eng. *three bound ant system*, TBAS) [12, 13]. Njegove glavne prednosti su nešto manja vremenska složenost i veća prilagodljivost u održavanju granica feromonskih tragova. U odnosu na MMAS, najvažnije su razlike:

- tri granice feromona (donja granica $\boldsymbol{\tau_{LB}}$, gornja granica $\boldsymbol{\tau_{UB}}$ i kontrakcijska granica $\boldsymbol{\tau_{CB} = \omega \cdot \tau_{UB}}$)
- povremeno isparavanje feromona umjesto redovitog isparavanja
- jedinstveni postupak pojačanja feromonskih tragova
- vrijednost donje granice $\boldsymbol{\tau_{LB}}$ jednaka je početnoj vrijednosti feromonskog traga

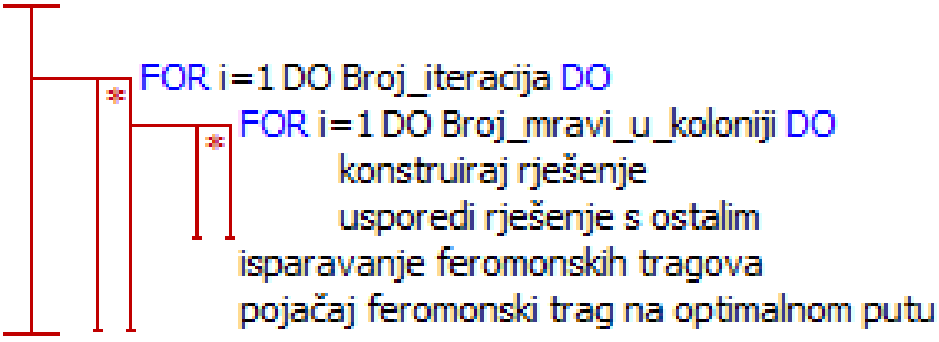
U konstrukciji rješenja, TBAS koristi primjenjuje slučajno-proporcijonalno pravilo odlučivanja kao i MMAS. TBAS koristi sve parametre kao i MMAS uz dodatni parametar $\boldsymbol{\omega}$, gdje je $\boldsymbol{\tau_{LB}/\tau_{UB} \leq \omega \leq 1}$. Postupak ažuriranja feromona za TBAS započinje pojačavanjem feromonskih tragova, nakon čega slijedi postupak isparavanja feromona, pod uvjetom da feromonski trag prelazi gornju granicu feromona $\boldsymbol{\tau_{UB}}$. Feromonski trag se kod isparavanja postavlja na vrijednost $\boldsymbol{\tau_{UB}}$. Za nagrađivanje određenog rješenja potrebno je odabrati strategiju nagrađivanja kao i kod ostalih ACO algoritama. Najčešće korištene strategije u literaturi su nagrađivanje najboljeg rješenja u iteraciji i nagrađivanje najboljeg trenutnog rješenja. Neka $\boldsymbol{s^{bs}}$ budu komponente rješenja koje trebaju biti nagrađene, onda se nagrađivanje implementira po sljedećoj formuli:

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

gdje je u prvoj iteraciji $\boldsymbol{Q_0 = 1}$, a u svakoj idućoj $\boldsymbol{Q_{i+1} = Q_i/(1-\rho)}$. Pokazano je da je TBAS brži za 1% do 20% od MMAS algoritma. Osim tih eksperimenata za brzinu izvođenja, provedeni su eksperimenti kvalitete rješenja gdje se TBAS također pokazao kao bolja opcija mravljeg algoritma. [12, 13].

Mravlji algoritmi su značajni pripadnici inteligentih sustava roja (eng. Swarm intelligent systems) i primjenjuju se za različite vrste problema. Kako bi se povećala efikasnost sustava, ACO algoritmi mogu se obogatiti sa dodatnim mogućnostima poput predviđanja koraka unaprijed, lokalne optimizacije te unatražnog pretraživanja koje se ne može vidjeti kod bioloških mrava. [8]

Implementacija mravljeg algoritma



Slika 8: Pseudokod jednostavnog mravljeg algoritma (Izvor: vlastita izrada prema Dorigo, M. i Stutzle, T., 2007.)

- Varijabla `Broj_iteracija` označava broj koliko će iteracija jedna kolonija pretraživati rješenje
- Varijabla `Broj_mravi_u_koloniji` označava broj mravi u svakoj iteraciji koji će konstruirati rješenje.
- Funkcija `konstruiraj rješenje` označava kreiranje krajnjeg rješenja temeljem feromonskih tragova, odnosno numeričkih koeficijenata sve dok ne prođe kroz sve čvorove zadanog problema. U TTP i TSP bi to značili prolazak kroz sve gradove i povratak u početni grad.
- Funkcija `usporedi rješenje s ostalim` označava vrednovanje svakog toga rješenja po određenim značajkama kao što su duljina puta, profit puta, troškovi puta te se krajnje rješenje svakog pojedinog virtualnog mrava uspoređuje s najboljim u koloniji ili najboljim općenito te u slučaju da se radi o boljem rješenju od onog s kojim uspoređujemo onda to rješenje postaje najbolje u koloniji ili najbolje općenito
- Funkcija `isparavanje feromonskih tragova` se implementira po sljedećoj formuli

$$\tau_{ij} = (1 - \rho) * \tau_{ij}$$

gdje je ρ, $0 < \rho < 1$, a označava postotak u decimalnom zapisu za koliko će se pojedini numerički koeficijenti koji predstavljaju virtualne feromonske tragove smanjivati ili ispravati.

- Funkcija `pojačaj feromonski trag na optimalnom putu` označava povećavanje numeričkih feromonskih tragova za najboljeg mrava u koloniji ili najboljeg mrava općenito ovisno o implementaciji kako bi se na tome putu pojačali feromonski tragovi, odnosno kako bi ti čvorovi imali veću vjerovatnost odabira prilikom puštanja iduće kolonije mravi za konstrukciju rješenja

5. Problem putujućeg lopova

Problem putujućeg lopova (eng. *traveling thief problem,* TTP) je noviji problem koji pokušava napraviti apstrakciju realnih višekomponentnih problema ovisnih o komponentama. Kombinira dva problema i stvara novi problem s dvije komponente. Konkretno, on kombinira problem trgovačkog putnika (TSP) i problem ruksaka (KP) jer su oba problema dobro poznata i proučavana su dugi niz godina na području optimizacije. Iako je to težak problem, mnogi znanstvenici se pitaju je li TTP dovoljno realan jer omogućuje samo jednom lopovu da putuje kroz stotine ili tisuće gradova kako bi prikupio (ukrao) predmete. Osim toga, lopov je dužan posjetiti sve gradove, bez obzira na to je li neki predmet ukraden ili ne. Znanstvenici Chand i Wagner su razgovarali o nedostacima trenutne formulacije i predstavili verziju problema koja omogućuje višestrukim lopovima putovanje kroz različite gradove s ciljem maksimiziranja kolektivnog profita grupe. Predložen je i niz brzih heuristika za rješavanje novopredloženog problema s više lopova koji putuju (MTTP). Uočeno je da bi mali broj dodatnih lopova mogao u mnogim slučajevima donijeti značajna poboljšanja objektivnih rezultata. [10]

Karakteristike problema

Lopov treba posjetiti skup gradova i odabrati neke predmete iz gradova te ih staviti u unajmljeni ruksak. Svaki predmet ima vrijednost i težinu. Naprtnjača ima ograničeni kapacitet koji ne može biti premašena ukupnom težinom odabranih predmeta. Na kraju, lopov mora platiti najamninu za ruksak, što ovisi o vremenu putovanja. TTP ima za cilj pronaći lopova za razgledavanje svih gradova točno jednom, pokupiti neke predmete na svome putu i konačno se vratiti u početni grad. Ukupni profit „krađe“ se računa kao ukupna vrijednost prikupljenih ili ukradenih predmeta umanjena za najam ruksaka. [10]

Svrha problema

U suvremenim poslovnim poduzećima složenost problema u stvarnom svijetu treba shvatiti kao jednu od najvećih prepreka u postizanju učinkovitosti. U suvremenom poslovanju dolazi do toga da se i relativno male tvrtke često suočavaju s problemima vrlo velike složenosti. Neki su znanstvenici istraživali značajke problema u stvarnom svijetu te su njihovi rezultati istraživanja poslužili kao objašnjenje za poteškoće koje Evolucijski algoritmi imaju za njihovo rješavanje. Brojni znanstvenici su se bavili ovim pitanjem, a napisani su i mnogi članci o Evolucijskom računanju (EC), no nije se tako lako mogla pronaći primjena u stvarnom svijetu. Michalewicz je utvrdio nekoliko razloga zbog kojih dolazi do neusklađenosti između

akademskog i stvarnog svijeta. Jedan od tih razloga je taj što su se akademski eksperimenti usredotočili na jednokomponentne referentne probleme, pri čemu su problemi u stvarnom svijetu često višekomponentni. Kako bi se zajednica usmjerila prema ovom sve važnijem aspektu optimizacije u stvarnom svijetu uveden je problem putujućeg lopova kako bi se ilustrirale složenosti koje nastaju zbog višestrukih interaktivnih komponenti. [10]

Vrednovanje rješenja

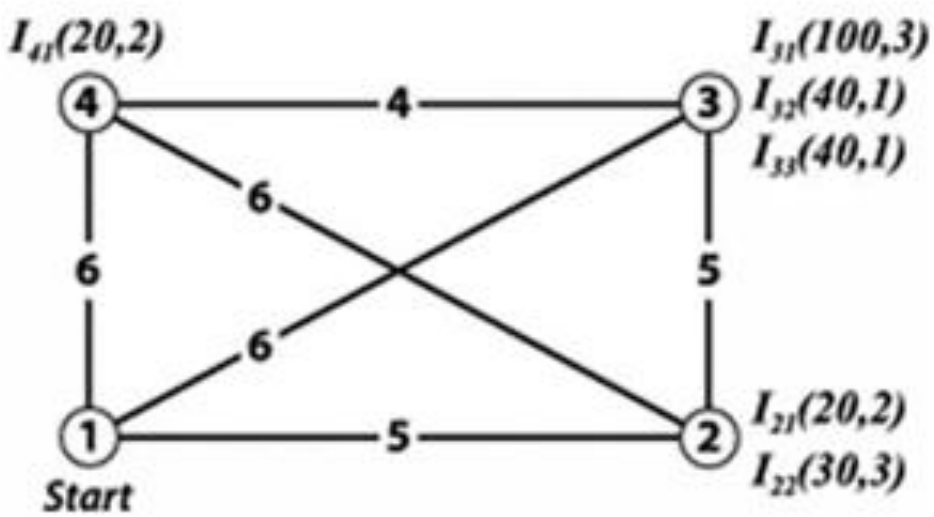
Dan je skup gradova $\boldsymbol{N} = \{\mathbf{1}, \dots, \boldsymbol{n}\}$ i skup predmeta $\boldsymbol{M} = \{\mathbf{1}, \dots, \boldsymbol{m}\}$ raspoređenih među gradovima. Za bilo koji par gradova $\boldsymbol{i}, \boldsymbol{j} \in \boldsymbol{N}$, znamo udaljenost $\boldsymbol{d_{ij}}$ između njih. Svaki grad $\boldsymbol{i}$, osim prvog, sadrži skup stavki $\boldsymbol{M_i} = \{\mathbf{1}, \dots, \boldsymbol{m_i}\}$, $\boldsymbol{M} = \cup_{i \in N} \boldsymbol{M_i}$. Svaki predmet $\boldsymbol{k}$ koji se nalazi u gradu i definiran je s vrijednošću $\boldsymbol{p_{ik}}$ i težinom $\boldsymbol{w_{ik}}$, dakle stavkom $\boldsymbol{I_{ik}} \sim (\boldsymbol{p_{ik}}, \boldsymbol{w_{ik}},)$. Lopov mora posjetiti sve gradove točno jednom, počevši od prvog grada, i na kraju se vratiti u njega. U bilo kojem gradu može se odabrati bilo koji predmet sve dok ukupna težina prikupljenih predmeta ne pređe navedeni kapacitet W. Za svaku vremensku jedinicu koja je uzeta za obilazak plaća se stopa najma ruksaka *R*. $\boldsymbol{v_{max}}$ i $\boldsymbol{v_{min}}$ označavaju maksimalne i minimalne brzine kojima se lopov može kretati. Cilj je pronaći put, zajedno s planom krađe predmeta, koji bi rezultirao maksimalnom dobiti. Ciljna funkcija koristi binarnu varijablu $\boldsymbol{y_{ik}} \in \{0,1\}$ koja je jednaka 1 kada je stavka *k* u gradu i odabrana, a u suprotnom nula. Također, neka $\boldsymbol{W_i}$ označi ukupnu težinu prikupljenih predmeta kada lopov napusti grad i. Zatim, ciljna funkcija za put $\Pi = (\boldsymbol{x_1}, \dots, \boldsymbol{x_n}), \boldsymbol{x_i} \in \boldsymbol{N}$ i plan krađe predmeta $\boldsymbol{P} = (\boldsymbol{y_{21}}, \dots, \boldsymbol{y_{nm_i}})$ ima sljedeći oblik:

$$\mathbf{Z}(\boldsymbol{\Pi}, \mathbf{P}) = \sum_{i=1}^{n} \sum_{k=1}^{m_i} \boldsymbol{p_{ik}} \boldsymbol{y_{ik}} - \boldsymbol{R} \left(\left(\sum_{i=1}^{n-1} \frac{\boldsymbol{d_{x_i x_i+1}}}{\boldsymbol{v_{max}} - \boldsymbol{\nu} \boldsymbol{w_{x_i}}} \right) + \frac{\boldsymbol{d_{x_n x_1}}}{\boldsymbol{v_{max}} - \boldsymbol{\nu} \boldsymbol{w_{x_n}}} \right)$$

gdje je $\boldsymbol{\nu} = \frac{\boldsymbol{v_{max}} - \boldsymbol{v_{min}}}{\boldsymbol{W}}$ konstantna vrijednost definirana ulaznim parametrima. Prvi korak je zbrojiti vrijednosti svih ukradenih predmeta, a drugi korak je oduzeti vrijednost koji lopov plaća za najam ruksaka (jednak ukupnom vremenu putovanja dužine Π pomnoženom s R). Prvi trošak je za putovanje od grada i do i+1, a drugi trošak je za putovanje iz posljednjeg grada u grad s ID-om 1. Imajte na umu da različite vrijednosti stope iznajmljivanja R rezultiraju u različitim TTP primjerima koje je moguće „teže“ ili „lakše“ riješiti. Na primjer, za male vrijednosti R (u odnosu na dobit), ukupna najamnina malo doprinosi konačnom objektivnom rezultatu. U ekstremnom slučaju R = 0, najbolje rješenje za datu instancu TTP-a je ekvivalentno najboljem rješenju KP komponente, što znači da uopće ne treba rješavati TSP komponentu. Slično tome,

visoke stope najma smanjuju učinak dobiti, a u krajnjem slučaju najbolje rješenje TTP-a je optimalno rješenje za datu komponentu TSP-a. [10]

Jednostavni primjer



Slika 9: primjer TTP-a (Izvor: Wagner, M., Lindauer, M., Mısır, M. et al., 2018.)

Na slici 9 pokazan je jednostavni primjer, a potpuni detalji dati su od Polyakovskiy-a. Svakom gradu, osim prvom je dodijeljen skup predmeta, npr. Grad 2 povezan je s predmetom $\boldsymbol{I_{21}}$ vrijednosti $\boldsymbol{p_{21} = 20}$ i težinom $\boldsymbol{w_{21} = 2}$ i s predmetom $\boldsymbol{I_{22}}$ vrijednosti $\boldsymbol{p_{22} = 30}$ i težinom $\boldsymbol{w_{22} = 3}$. Pretpostavimo da su maksimalna težina $\boldsymbol{W = 3}$, vrijednost iznajmljivanja $\boldsymbol{R = 1}$ i $\boldsymbol{v_{max}}$ i $\boldsymbol{v_{min}}$ postavljeni kao 1 i 0,1, respektivno. Tada je optimalna ciljna vrijednost $\boldsymbol{Z\,(\Pi, P) = 50}$ kada je obilazak $\boldsymbol{\Pi = (1, 2, 4, 3, 1)}$ i kad se uzmu stavke $\boldsymbol{I_{32}}$ i $\boldsymbol{I_{33}}$ (ukupna dobit od 80). Kako lopov u ruksaku ima težinu 2 na putu od grada 3 do grada 1 onda mu to smanjuje brzinu i rezultira povećanim troškovima iznosa 15. [10] Stoga je konačna ciljna vrijednost :

$$\boldsymbol{Z\,(\Pi, P) = 80 - 1 \cdot (5 + 6 + 4) - 1 \cdot \left(\frac{6}{1 - \frac{1.0 - 0.1}{3} * 2}\right) = 80 - 15 - 15 = 50}.$$ [10]

Ideja mravljeg algoritma na ovom problemu

$$I_{41} = 50$$

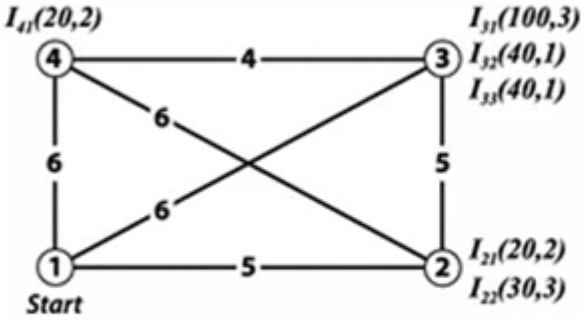
$$I_{31} = 50$$

$$I_{32} = 50$$

$$I_{33} = 50$$

$$I_{21} = 50$$

$$I_{22} = 50$$



Slika 10: primjer mravljeg algoritma za TTP (Izvor: vlastita izrada prema Wagner, M., Lindauer, M., Mısır, M. et al., 2018.)

Na slici 10 ću objasniti ideju mravljeg algoritma za problem putujućeg lopova. Feromonske tragove, odnosno numeričke koeficijente će sadržavati svaki pojedini predmet te će u početku njihova vrijednost biti maksimalna vrijednost feromonskog traga, a ona u našem primjeru iznosi 50. Ovo je karakteristika MMAS algoritma te ćemo se fokusirati na implementaciju jednog MMAS algoritma za rješavanje ovog problema. U prvoj iteraciji mrava, svaki mrav odabire slučajnim odabirom slijed predmeta koje bi trebao pokupiti, uključujući i predmete koji posjeduju nul-profit i nul-težinu, a postoje iz razloga kako bi lopov mogao „ne uzeti ništa“ iz nekoga grada pa je programski smišljen ovaj predmet kojega će lopov pokupiti ako mu ništa ne odgovara u tome gradu. Nakon što prvi mrav odabere slučajnim odabirom slijed predmeta, onda slijedi računanje stvarnog puta, odnosnog onog realnog. Realni put se računa na način da se ide kroz to polje slučajno izabranih slijeda kupljenja predmeta te se prvo gledaju karakteristike ruksaka, odnosno ako je predmet lakši (ima manju masu od trenutnog kapaciteta ruksaka) , onda se gleda karakteristika da li je lopov već bio u tome gradu kada je mogao pokupiti neki drugi predmet te ako nije onda stavlja predmet u ruksak. U slučaju da je bio već u tome gradu, onda

samo prelazi na idući predmet u tome nizu slučajno izabranih brojeva. Treba napomenuti kako taj slučajan izbor predmeta na početku pokretanja mrava nije posve slučajan jer se bira na temelju feromonskog traga i njihovih intervala pa će oni s većim koeficijentom imati veću šansu odabira od ostalih. Nakon svake iteracije, najbolje trenutno rješenje se nagrađuje i svi tragovi se smanjuju za određeni postotak, u našem slučaju smo dogovorili 2%.

Definiranje problema TTP za rješavanje

Problem putujućeg lopova je zahtjevan sam po sebi pa i samo definiranje problema zahtjeva određena istraživanja za njegovo dobro postavljanje. Počevši od broja gradova, nailazimo na prvi problem, a to su udaljenosti u gradovima koje ne smiju previše odskakati jedna od druge, odnosno mrav bi u svakom gradu trebao imati više bližih gradova koje bi mogao odabrati. Ako bi recimo određeni gradovi bili predaleko onda bi se njihovom posjetom uvelike naštetilo krajnjem optimalnom rješenju. Kako ne govorimo o problemu trgovačkog putnika nego o problemu putujućeg lopova onda dolazimo do drugog problema, a to je definiranje predmeta u svakome gradu osim početnoga. Omjer težine i profita treba postaviti tako da svi predmeti budu jednako ili približno jednako poželjni (ne bi imalo smisla da svi predmeti imaju istu težinu i profit). Problem koji će biti rješavan u ovome radu predstavljen je na natjecanju CEC-a na IEEE WCCI 2014: Optimizacija problema s više međusobno ovisnih komponenata u Beijing-u. Na natjecanju je bio zadan problem koji je obuhvaćao 280 gradova te 1 predmet u svakome gradu i problem koji je obuhvaćao 280 gradova te 5 predmeta u svakome gradu. Naravno bilo je zadano puno više instanci problema sa većim brojem gradova i predmeta, ali ćemo se fokusirati trenutno na ova dva problema. Iz ova dva problema ćemo kasnije kreirati dva manja problema koji će sadržavati 100 gradova te 1 predmet u svakome gradu i problem koji će sadržavati 100 gradova i 5 predmeta u svakome gradu. Svaki problem je imao drukčije definiranu cijenu iznajmljivanje i veličinu ruksaka kao što je vidljivo na slikama 11 i 12. [11]

```
PROBLEM NAME:    a280-TTP
KNAPSACK DATA TYPE: bounded strongly corr
DIMENSION:       280
NUMBER OF ITEMS:         279
CAPACITY OF KNAPSACK:    25936
MIN SPEED:       0.1
MAX SPEED:       1
RENTING RATIO:   5.61
```

Slika 11: Opis problema za 280 gradova i 1 predmet u gradu

(Izvor: https://cs.adelaide.edu.au/)

```
PROBLEM NAME:    a280-TTP
KNAPSACK DATA TYPE: uncorrelated, similar weights
DIMENSION:       280
NUMBER OF ITEMS:         1395
CAPACITY OF KNAPSACK:    637010
MIN SPEED:       0.1
MAX SPEED:       1
RENTING RATIO:   72.70
```

Slika 12: opis problema za 280 gradova i 5 predmeta u gradu
(Izvor: https://cs.adelaide.edu.au/)

6. Programski kod

U uvodu je rečeno da je programski kod pisan u C++ programskom jeziku u okruženju Microsoft Visual Studio 2019. Programski kod je pisan pomoću funkcija, a u kasnijem istraživanju svakako ima u cilju prebaciti se na objektno-orijentirani način pisanje programskog koda. Programski kod sadrži 3 *headera*: datoteke.h, matrice.h i random.h. Glavni programski kod se nalazi u C++ datoteci pod nazivom Glavni_zavrsni.cpp. Korištenje *headera* je učinjeno kako bi glavni kod bio čitljiviji te sklon lakšim doradama, a svi pomoćni dijelovi koji nisu vezani za glavno izvođenje algoritma su napisani u gore navedenim *headerima*.

Struktura podataka

Korištene su različite strukture podataka za implementaciju mravljeg algoritma za problem putujućeg lopova. U nastavku ovog odlomka će biti objašnjenje one najvažnije.

```
class rjesenje {
private:
    unsigned int id;
public:
    int grad;
    int broj_predmeta;
    double profit;
    double tezina;
};

class pomoc {
public:
    int id_predmeta;
    double koeficijent;
};
```

Slika 13: Prikaz strukture podataka klase `rjesenje` i klase `pomoc` (Izvor: vlastita izrada)

Na slici 13 možemo vidjeti prikaz dvije klase: `rjesenje` i `pomoc`. Klasa `rjesenje` sadrži karakteristike pojedinog predmeta, dok klasa `pomoc` sadrži karakteristike feromonskog traga. To bi značilo da klasa `rjesenje` sadrži tip integer koji označava broj grada u kojemu se predmet nalazi, tip integer koji označava id predmeta, tip double koji označava profit ili vrijednost predmeta te tip double koji predstavlja težinu predmeta. Klasa rješenje sadrži tip integer koji označava id predmeta i tip double koji predstavlja feromonski trag za taj predmet.

```
Matrix<double> TauTragovi;
Matrix<double> MatricaProblema;
Matrix<double> Predmeti;
Matrix<double> Predmeti_izracun;
vector<rjesenje> novi;
vector<rjesenje> shema_predmeta;
vector<rjesenje> krajnja_shema_krade;
vector<rjesenje> shema_kolonije;
vector<rjesenje> kopija_sheme_trenutnog_mrava;
int id_predmeta = 1;
vector<pomoc> random_racunanje;
vector<pomoc> pomoc_predmeti;
vector<int> put;
vector<bool> gradovi;
vector<bool> pokupio_predmet;

vector<double> put_koef;
double glavna_suma = INT_MIN;
double suma_kolonije = INT_MIN;
double kapacitet_ruksaka= 25936; //1 item per city
//double kapacitet_ruksaka = 637010; //5 item per city
//double kapacitet_ruksaka = 1262022; //10 item per city
double cijena_iznajmljivanja= 5.61; //1 item per city
//double cijena_iznajmljivanja = 72.70; //5 item per city
//double cijena_iznajmljivanja = 208.53; //10 item per city
double max_brzina = 1;
double min_brzina = 0.1;
int broj_iteracija = 1000;
int broj_mravi_u_koloniji = 20;
double rho = 0.02;
double delta_tau = 1;
double tau_max = 50;
double tau_min = 0.005;
Matrix<double> Put1;
```

Slika 14: Prikaz strukture podataka glavnog dijela programa (Izvor: vlastita izrada)

U glavnom dijelu programa koristim sljedeće strukture podataka vidljive na slici 14. `Glavna_suma` je varijabla koja označava trenutno najbolju sumu, odnosno najbolje rješenje, a `suma_kolonije` predstavlja varijablu koja označava najbolju sumu u trenutnoj iteraciji.

Kapacitet ruksaka označava maksimalnu težinu koju je moguće spremiti u ruksak. `Cijena iznajmljivanja` označava cijenu koju lopov plaća po jedinici vremena dok obilazi gradove jer je ruksak cijelo vrijeme iznajmljen. Maksimalna i minimalna brzina označavaju kretanje lopova sa punim i praznim kapacitetom ruksaka kroz jedinicu vremena. Broj iteracija predstavlja koliko će jedna kolonija istraživati put, a broj mrava u koloniji označava broj mrava koji će biti pušteni u jednoj iteraciji. Nakon jedne iteracije slijedi isparavanje i pojačavanje feromonskog traga. Varijabla `rho` označava postotak kojim feromonski trag isparava, a `delta_tau` označava vrijednost za koju se nagrađuje najbolje rješenje. `Tau_max` označava maksimalnu vrijednost feromonskog traga, a `Tau_min` označava minimalnu vrijednost feromonskog traga. Od bitnijih definiranih vektora bi spomenio `gradovi` koji je tipa bool i označava binarnu vrijednost posjećenih gradova. 1 ako je posjećen, a 0 ako nije. `Shema_predmeta` predstavlja rješenje trenutnog mrava, `Shema_kolonije` predstavlja najbolje rješenje u trenutnoj iteraciji, a `krajnja_shema_krade` sadrži najbolje rješenje svih iteracija. `MatricaProblema` sadrži udaljenosti među gradovima, a `pomoc_predmeti` sadrži trenutne feromonske tragove za sve predmete.

Mravlji algoritam za TTP

```
ofstream novifile("sume.csv");
ofstream koeficijenti("koef.csv");
koeficijenti << "Predmet;Koeficijent" << endl;
for (int i = 0; i < broj_iteracija; i++)
{
    dealokacija(put);
    dealokacija(pomoc_predmeti);
    dealokacija(put_koef);
    kopiranje_puta();
    if(i==99 || i==broj_iteracija-1)
        zapis_koeficijenata(i, koeficijenti);
    for (int j = 0; j < broj_mravi_u_koloniji; j++) {
        cout << "Mrav broj: " << j + 1 << " Kolonija: " << i + 1 << endl;
        if (j != 0)
            kopiranje_puta();
        //pocetak_algoritma();
        odredivanje_puta_slijed();
        provjera_predmeta_naprijed();
        provjera_predmeta_nazad();
        if (i == 0 && j==0)
            Lokalna_optimizacija_a(shema_kolonije, 200);
        if (i == broj_iteracija - 1 && j==0)
            Lokalna_optimizacija_b(krajnja_shema_krade, 200);
        dealokacija(put);
    }
    std::stringstream suma_iteracije;
    suma_iteracije.imbue(std::locale(std::cout.getloc(), new punct_facet<char, ','>));
    suma_iteracije << std::setprecision(2) << std::fixed << suma_kolonije;
    std::string s = suma_iteracije.str();
    novifile << s << endl;
    suma_kolonije = INT_MIN;
    mnozenje_koef();
    dealokacija(shema_kolonije);
}
novifile.close();
```

Slika 15: Glavni dio mravljeg algoritma u kodu (Izvor: vlastita izrada)

Na slici 15 možemo vidjeti glavni dio mravljeg algoritma za problem koji želimo riješiti, odnosno što se događa tijekom jednog pokretanja programa. Sastoji se od dvije for petlje, od kojih prva predstavlja broj iteracija, a druga služi za puštanje određenog broja mrava po iteraciji. Unutarnja for petlja sadrži funkcije:

- `kopiranja_puta()`,
- `odredivanje_puta_slijed()`,
- `provjera_predmeta_naprijed()`,
- `provjera_predmeta_nazad()`
- `delokacija(put)`.

Te funkcije su funkcije koje svaki pojedini mrav prođe prilikom konstrukcije rješenja, a sve ostale funkcije poput:

- `mnozenje_koef()`
- `zapis_koeficijenata(i, koeficijenti)`

su funkcije koje zahvaćaju cijele iteracije. Na slici možemo vidjeti funkcije:

- `Lokalna_optimizacija_a(shema_kolonije,200)`
- `Lokalna_optimizacija_b(krajnja_shema_krade,200)`

koje služe kao eksperimentalne funkcije za nužnost lokalne optimizacije u algoritmu. Rezultati i doneseni zaključci nakon izvođenja tih funkcija će biti prikazani u sljedećem poglavlju među rezultatima eksperimenata.

```
void kopiranje_puta() {
    pomoc* s2 = new pomoc;
    for (int i = 0; i < id_predmeta; i++) {
        s2->id_predmeta = i;
        s2->koeficijent = Put1(0, i);
        pomoc_predmeti.push_back(*s2);
        put_koef.push_back(Put1(0, i));
    }
}
```

Slika 16: `Funkcija kopiranja_puta()` (Izvor: vlastita izrada)

Slika 16 prikazuje definiciju funkcije `kopiranje_puta()`. Prikazana funkcija nam služi za ažuriranje feromonskih tragova koji se spremaju u vektore `pomoc_predmeti` i `put_koef`, a kasnije pomoću tih vektora i zapisanih koeficijenata (feromonskih tragova) određujemo slijed predmeta pri konstrukciji rješenja.

```
void odredivanje_puta_slijed() {
    while (put_koef.size() != 0) {
        double suma = 0;
        int p = 0;
        int interval = 0;
        double brojac = 0;
        double br = 0;
        for (int i = 0; i < put_koef.size(); i++) {
            suma = suma + put_koef[i];
        }
        double slucajan = randombroj(0, suma);
        interval = int(slucajan);
        double suma1 = 0;
        int j = 0;
        while (suma1 < slucajan) {
            suma1 = suma1 + put_koef[j];
            j = j + 1;
        }
        interval = j - 1;
        put.push_back(pomoc_predmeti[interval].id_predmeta + 1);

        pomoc_predmeti.erase(pomoc_predmeti.begin() + interval);
        put_koef.erase(put_koef.begin() + interval);
    }
}
```

Slika 17: Funkcija `odredivanje_puta_slijed()` (Izvor: vlastita izrada)

Slika 17 nam prikazuje definiciju funkcije `odredivanje_puta_slijeda()` koja radi slučajan odabir id-a predmeta na osnovi njihovih koeficijenata (feromonskih tragova) na način da se odabire slučajan broj između 0 i sume svih koeficijenata (feromonskih tragova), a nakon toga se gleda interval u kojemu se nalazi taj broj te se uzima taj id predmeta kao izabrani. To se ponavlja skroz dok u vektoru `pomoc_predmeti` ima zapisanih predmeta. Na ovaj način predmeti koji imaju veći feromonski trag, odnosno numerički koeficijent koji ga predstavlja u algoritmu ima veće šanse da će biti odabran prije drugih predmeta te na taj način ima veću šansu biti redoslijedno prije u vektoru put iz kojega se kasnije određuje realan put pomoću funkcija:

- `Provjera_predmeta_nazad()`
- `Provjera_predmeta_naprijed()`

U početku će odabir biti posve slučajan, ali bi se već nakon nekoliko iteracija situacija mogla promijeniti te će određeni koeficijenti imati veći feromonski trag u odnosu na ostale. Nakon 100 i 1000 iteracija gotovo je nemoguće da predmeti koji imaju koeficijent (feromonski trag) približno τ_{min} budu izabrani prije onih koji imaju koeficijent (feromonski trag) približno τ_{max}. Uvijek postoji vjerojatnost za to, ali se ona smanjuje puštanjem većeg broja mravi u iteraciji.

```
void provjera_predmeta_nazad() {
    rjesenje* s3 = new rjesenje;
    int b = -1;
    double tezina_ruksaka = kapacitet_ruksaka;
    for (int i = put.size() - 1; i > 0; i--)
    {
        int a = put[i] - 1;
        if (gradovi[novi[a].grad - 1] == 0)
        {

            if (b != -1) {
                if ((novi[a].grad != novi[b].grad) && (pokupio_predmet[novi[b].grad - 1] == 1)) {
                    gradovi[novi[b].grad - 1] = 1;
                }
            }
            if ((novi[a].tezina == 0) && (gradovi[novi[a].grad - 1] == 0)) {
                s3->broj_predmeta = novi[a].broj_predmeta;
                s3->grad = novi[a].grad;
                s3->profit = novi[a].profit;
                s3->tezina = novi[a].tezina;
                shema_predmeta.push_back(*s3);
                gradovi[novi[a].grad - 1] = 1;
                pokupio_predmet[novi[a].grad - 1] = 1;
                continue;
            }
            if ((novi[a].tezina <= tezina_ruksaka) && (novi[a].tezina != 0)) {
                s3->broj_predmeta = novi[a].broj_predmeta;
                s3->grad = novi[a].grad;
                s3->profit = novi[a].profit;
                s3->tezina = novi[a].tezina;
                shema_predmeta.push_back(*s3);
                tezina_ruksaka = tezina_ruksaka - novi[a].tezina;
                pokupio_predmet[novi[a].grad - 1] = 1;
            }
            b = put[i] - 1;
        }
    }
```

Slika 18: 1 dio funkcije `provjera_predmeta_nazad()` (Izvor: vlastita izrada)

```
    vector<rjesenje>::iterator it;
    vector<rjesenje>::iterator mi;
    double suma = 0;
    double masa = 0;
    vector<rjesenje>::iterator kraj;
    int brojac = 0;
    for (it = shema_predmeta.begin(); it != shema_predmeta.end(); ++it) {
        brojac++;
        if (it != shema_predmeta.end() - 1)
            mi = it + 1;
        else {
            kraj = it;
            break;
        }
        masa = masa + it->tezina;
        suma = suma + it->profit;
        int prvi_grad = it->grad ;
        int drugi_grad = mi->grad ;
        suma = suma - (cijena_iznajmljivanja * (double(MatricaProblema(prvi_grad, drugi_grad) /
      (max_brzina - ((max_brzina - min_brzina) / kapacitet_ruksaka) * masa))));
    }

    int kraj_obilaska = kraj->grad;
    int pocetak_obilaska = shema_predmeta[0].grad ;
    suma = suma - (cijena_iznajmljivanja * MatricaProblema(1, pocetak_obilaska)) -
    (cijena_iznajmljivanja * (double(MatricaProblema(kraj_obilaska, 1) /
    (max_brzina - ((max_brzina - min_brzina) / kapacitet_ruksaka) * masa))));
    dealokacija(kopija_sheme_trenutnog_mrava);
    kopija_sheme_trenutnog_mrava = shema_predmeta;
    provjera_sume(suma);
    Lokalna_optimizacija(kopija_sheme_trenutnog_mrava, 10);

}
```

Slika 19: 2 dio funkcije `provjera_predmeta_nazad()` (Izvor: vlastita izrada)

Nakon što smo funkcijom `odredivanje_puta_slijeda()` odredili mogući slijed kupljenja predmeta, onda koristimo sljedeću funkciju kako bi odredili realan put. Ova funkcija provjera radi li se o predmetu koji se nalazi u gradu u kojemu smo već bili te ako se ne nalazi tamo, provjerava da li je težina predmeta manja od trenutnog kapaciteta ruksaka te ako je manja onda je moguće staviti predmet u ruksak. Nakon svega toga, određuje se suma predmeta te se ta suma kasnije uspoređuje s najboljom sumom od svih i sa sumom najboljom u iteraciji. Suma se zaboravlja ako nije bolja od bilo koje nabrojane sume. `Provjera_predmeta_nazad()` uzima isti vektor slučajno odabranih id-a predmeta, ali počinje od kraja te radi na istom principu kao i funkcija `provjera_predmeta_naprijed()`. Delokacija služi za brisanje sadržaja varijable, odnosno onoga što se nalazi u varijabli put, a to su koeficijenti koji predstavljaju feromonske tragove. Na slikama 18 i 19 je prikazana funkcija `provjera_predmeta_nazad()` kojoj je glavni cilj izračun realnog puta te kreiranje sheme plana krađe predmeta. Nakon što je kreirana shema plana krađe predmeta, kreće se u izračun krajnjeg rješenja. Izračun se radi kako bi mogli ocijeniti rješenje, odnosno usporediti ga s ostalim konstruiranim rješenjima u iteraciji. Nakon što se izračuna suma po formuli navedenoj na slici 19 onda se poziva funkcija `provjera_sume(suma)` koja šalje izračunatu sumu. Funkcija `Lokalna_optimizacije(kopija_sheme_trenutnog_mrava,10)` će biti pojašnjena u idućem odlomku.

```
void provjera_sume(double suma) {
    rjesenje* s4 = new rjesenje;
    rjesenje* kolonija = new rjesenje;
    if (suma > glavna_suma) {
        dealokacija(krajnja_shema_krade);
        vector<rjesenje>::iterator it;
        for (it = shema_predmeta.begin(); it != shema_predmeta.end(); ++it) {
            s4->broj_predmeta = it->broj_predmeta;
            s4->grad = it->grad;
            s4->profit = it->profit;
            s4->tezina = it->tezina;
            krajnja_shema_krade.push_back(*s4);
            glavna_suma = suma;
        }
    }
    if (suma > suma_kolonije) {
        dealokacija(shema_kolonije);
        vector<rjesenje>::iterator it;
        for (it = shema_predmeta.begin(); it != shema_predmeta.end(); ++it) {
            kolonija->broj_predmeta = it->broj_predmeta;
            kolonija->grad = it->grad;
            kolonija->profit = it->profit;
            kolonija->tezina = it->tezina;
            shema_kolonije.push_back(*kolonija);
            suma_kolonije = suma;
        }
    }
    dealokacija(shema_predmeta);
    dealokacija(pokupio_predmet);
    dealokacija(gradovi);
    reset_gradova();
}
```

Slika 20: Funkcija `provjera_sume(double suma)` (Izvor: vlastita izrada)

Funkcija `provjera_sume(double suma)` ima za cilj usporediti dobivenu sumu mrava sa sumom najboljeg rješenja od početka algoritma i sa sumom najboljeg rješenja u iteraciji. Ako se radi o boljem rješenju, onda dobivena suma postaje `suma_kolonije` ili `glavna_suma` ovisno o uvjetu kojeg ispunjava. Slika 20 nam prikazuje cijelu definiciju funkcije `provjera_sume(double suma)` te uvjete koji se provjeravaju u njoj. Nakon što završi jedna iteracija, pokreće se funkcija `mnozenje_koef()`.

```
void mnozenje_koef() {
    for (int i = 0; i < id_predmeta; i++)
    {
        //Put1(0, i) = Put1(0, i) * kazna;
        Put1(0, i) =(1-rho)* Put1(0, i);
        if (Put1(0, i) <= tau_min)
            Put1(0, i) = tau_min;
    }
    //NAGRA♦IVANJE NAJBOLJEG MRAVA U KOLONIJI
    /*
    vector<rjesenje>::iterator it;
    for (it = shema_kolonije.begin(); it != shema_kolonije.end(); ++it) {
        //Put1(0, it->broj_predmeta - 1) = Put1(0, it->broj_predmeta - 1) * nagrada;
        Put1(0, it->broj_predmeta - 1) = Put1(0, it->broj_predmeta - 1) + delta_tau;
        if (Put1(0, it->broj_predmeta - 1) <= tau_min)
            Put1(0, it->broj_predmeta - 1) = tau_min;
        if (Put1(0, it->broj_predmeta - 1) >= tau_max)
            Put1(0, it->broj_predmeta - 1) = tau_max;
    }
    */
    //NAGRA♦IVANJE NAJBOLJEG RJEŠENJA SVIH KOLONIJA

     vector<rjesenje>::iterator it;
    for (it = krajnja_shema_krade.begin(); it != krajnja_shema_krade.end(); ++it) {
        //Put1(0, it->broj_predmeta - 1) = Put1(0, it->broj_predmeta - 1) * nagrada;
        Put1(0, it->broj_predmeta - 1) = Put1(0, it->broj_predmeta - 1) + delta_tau;
        if (Put1(0, it->broj_predmeta - 1) >= tau_max)
            Put1(0, it->broj_predmeta - 1) = tau_max;
        //cout << Put1(0,it->broj_predmeta - 1) << endl;
    }

}
```

Slika 21: Funkcija `mnozenje_koef()` (Izvor: vlastita izrada)

Na slici 21 možemo vidjeti funkciju `mnozenje_koef()` koja se pokreće nakon svake iteracije mrava, a u njoj se događaju ispravanja feromonskih tragova te nagrađivanja onih najboljih. U literaturi možemo pronaći da se najčešće koriste dvije vrste nagrađivanja, a to su: nagrađivanje najboljeg trenutnog rješenja i nagrađivanje najboljeg rješenja u iteraciji. Osim toga postoje i općenitije strategije koje uzimaju najbolje rješenje iz zadnjih kapa iteracija. [13, 14, 15]. Na početku funkcije se prvo pokreće prva for petlja u kojoj se smanjuju feromonski tragovi za rho te se potom u drugoj for petlji nagrađuje najbolje rješenje svih iteracija. U

komentarima postoji programski dio za nagrađivanje najboljeg rješenja u iteraciji koje nismo koristili prilikom vršenja eksperimenata.

Nakon što se ove dvije petlje završe, pokreće se dio programskog koda koji zapisuje najbolje rješenje u excel file u csv datoteku uz zapis sume toga rješenja te gornju granicu kapaciteta ruksaka. Na slici 22 možemo vidjeti taj dio koda koji odrađuje zapis najboljeg rješenja.

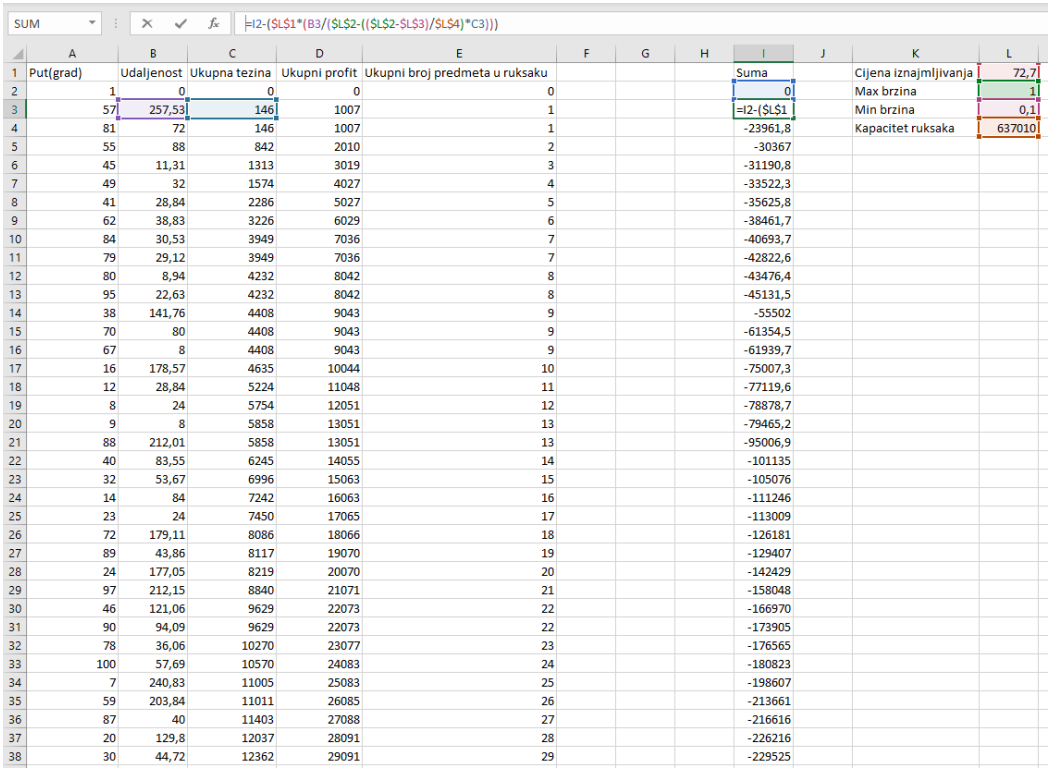
```
ofstream outputfile("najboljeRjesenje.csv");
outputfile << "Pocetak u gradu 1" << endl;
outputfile << "Broj predmeta;Grad;Profit;Tezina;" << endl;
for (int i = krajnja_shema_krade.size() - 1; i >= 0; i--)
{
    outputfile << krajnja_shema_krade[i].broj_predmeta << ";" << krajnja_shema_krade[i].grad + 1
    << "; " << krajnja_shema_krade[i].profit << "; " << krajnja_shema_krade[i].tezina;
    outputfile << endl;
}
outputfile << "Max broj predmeta u ruksaku: " << gornja_granica_predmeta() << endl;
std::stringstream zadnja_suma;
zadnja_suma.imbue(std::locale(std::cout.getloc(), new punct_facet<char, ','>));
zadnja_suma << std::setprecision(2) << std::fixed << glavna_suma;
std::string s = zadnja_suma.str();
outputfile << "Suma rješenja:; " << s << endl;
```

Slika 22: Zapis krajnjeg rješenja u csv datoteku (Izvor: vlastita izrada)

```
ofstream ttp("ttp.csv");
ttp << "Put(grad);Udaljenost;Ukupna tezina; Ukupni profit;Ukupni broj predmeta u ruksaku" << endl;
int prethodniGrad = 1;
ttp << "1;" <<"0;"<<"0;"<<"0;"<<"0;"<< endl;
double tezina = 0;
double profit = 0;
double brojpredmeta_u_ruksaku = 0;
vector<rjesenje>::iterator it;
for (it=krajnja_shema_krade.begin();it!=krajnja_shema_krade.end();++it)
{
    tezina = tezina + it->tezina;
    profit = profit + it->profit;
    if (it->tezina != 0)
        brojpredmeta_u_ruksaku++;
    if (it->grad != prethodniGrad) {
        std::stringstream udaljenost_gradova;
        udaljenost_gradova.imbue(std::locale(std::cout.getloc(), new punct_facet<char, ','>));
        udaljenost_gradova << std::setprecision(2) << std::fixed << double(MatricaProblema(prethodniGrad, it->grad));
        std::string duljina = udaljenost_gradova.str();
        ttp << it->grad << ";" << duljina << ";" << tezina << ";" << profit << ";" << brojpredmeta_u_ruksaku << endl;
        prethodniGrad = it->grad;
    }
}
std::stringstream udaljenost_gradova1;
udaljenost_gradova1.imbue(std::locale(std::cout.getloc(), new punct_facet<char, ','>));
udaljenost_gradova1 << std::setprecision(2) << std::fixed << double(MatricaProblema(prethodniGrad, 1));
std::string duljina1 = udaljenost_gradova1.str();
ttp << 1 << ";" << duljina1 << ";" << tezina << ";" << profit << ";" << brojpredmeta_u_ruksaku << endl;
std::stringstream ttp_suma;
ttp_suma.imbue(std::locale(std::cout.getloc(), new punct_facet<char, ','>));
ttp_suma << std::setprecision(2) << std::fixed << glavna_suma;
std::string k = ttp_suma.str();
ttp << "Suma rjesenja:; " << k << endl;
return 0;
}
```

Slika 23: Zapis rješenja u csv datoteku za uređivanje rješenja (Izvor: vlastita izrada)

Na slici 23 vidimo zapis datoteke u ttp.csv. Izgled datoteke je vidljiv na slici 24. U toj datoteci možemo mijenjati određene početne parametre te istraživati kako se konačna suma mijenja ovisno o početnim postavkama. Osim promjene početnih postavki možemo promijeniti težinu ruksaka te vrijednost predmeta.



SUM | =I2-(L1*(B3/(L2-(L2-L3)/L4)*C3)))

	A	B	C	D	E	F	G	H	I	J	K	L
1	Put(grad)	Udaljenost	Ukupna tezina	Ukupni profit	Ukupni broj predmeta u ruksaku				Suma		Cijena iznajmljivanja	72,7
2	1	0	0	0	0				0		Max brzina	1
3	57	257,53	146	1007	1				=I2-(L1		Min brzina	0,1
4	81	72	146	1007	1				-23961,8		Kapacitet ruksaka	637010
5	55	88	842	2010	2				-30367			
6	45	11,31	1313	3019	3				-31190,8			
7	49	32	1574	4027	4				-33522,3			
8	41	28,84	2286	5027	5				-35625,8			
9	62	38,83	3226	6029	6				-38461,7			
10	84	30,53	3949	7036	7				-40693,7			
11	79	29,12	3949	7036	7				-42822,6			
12	80	8,94	4232	8042	8				-43476,4			
13	95	22,63	4232	8042	8				-45131,5			
14	38	141,76	4408	9043	9				-55502			
15	70	80	4408	9043	9				-61354,5			
16	67	8	4408	9043	9				-61939,7			
17	16	178,57	4635	10044	10				-75007,3			
18	12	28,84	5224	11048	11				-77119,6			
19	8	24	5754	12051	12				-78878,7			
20	9	8	5858	13051	13				-79465,2			
21	88	212,01	5858	13051	13				-95006,9			
22	40	83,55	6245	14055	14				-101135			
23	32	53,67	6996	15063	15				-105076			
24	14	84	7242	16063	16				-111246			
25	23	24	7450	17065	17				-113009			
26	72	179,11	8086	18066	18				-126181			
27	89	43,86	8117	19070	19				-129407			
28	24	177,05	8219	20070	20				-142429			
29	97	212,15	8840	21071	21				-158048			
30	46	121,06	9629	22073	22				-166970			
31	90	94,09	9629	22073	22				-173905			
32	78	36,06	10270	23077	23				-176565			
33	100	57,69	10570	24083	24				-180823			
34	7	240,83	11005	25083	25				-198607			
35	59	203,84	11011	26085	26				-213661			
36	87	40	11403	27088	27				-216616			
37	20	129,8	12037	28091	28				-226216			
38	30	44,72	12362	29091	29				-229525			

Slika 24: Izgled datoteke ttp.csv nakon zapisa (Izvor: vlastita izrada)

Slika 25: Kontrolni ispis u konzoli (Izvor: vlastita izrada)

Slika 25 pokazuje ispis u konzoli koji se ispisuje prije nego bilo koji mrav krene u konstrukciju rješenja, a služi za određivanje određenih zastoja koji su nastajali prilikom implementacije.

Lokalna optimizacija

Lokalna optimizacija služi za poboljšanje već konstruiranog rješenja. U našem slučaju nakon što mrav konstruira rješenje onda bi mogao svoje rješenje lokalno optimizirati kako bi ono postalo bolje. U algoritmu ćemo implementirati jednostavnu lokalnu optimizaciju te ćemo kasnije u poglavlju provedenih eksperimenata prokomentirati rezultate dobivene bez i sa lokalnom optimizacijom. Ovdje ćemo prokomentirati kod te ideju jednostavne lokalne optimizacije.

```
void Lokalna_optimizacija(vector<rjesenje> pocetno_rjesenje, int broj_koraka) {
    for (int i = 0; i < broj_koraka; i++)
        Lokalna_jedan_korak(pocetno_rjesenje);
}
```

Slika 26: Funkcija `Lokalna_optimizacija(rjesenje, broj_koraka)` (Izvor: vlastita izrada)

Na slici 26 možemo vidjeti početnu funkciju `Lokalne_optimizacije(pocetno_rjesenje, broj_koraka)`. Početno rješenje predstavlja konstruirano rješenje mrava poslanu u lokalnu optimizaciju, dok broj koraka označava u koliko će se koraka pokušati doći do boljeg rješenja. U svakom koraku pokreće se funkcija `Lokalna_jedan_korak(pocetno_rjesenje)` koju možemo vidjeti na slici 27.

```
void Lokalna_jedan_korak(vector<rjesenje> pocetno_rjesenje) {
    vector<rjesenje> r1 = izbaci_jedan_predmet(pocetno_rjesenje);  //izbaci prvi nasumični predmet

    if (odrediKonacnuSumuRjesenja(r1) > odrediKonacnuSumuRjesenja(pocetno_rjesenje))
    {
        dealokacija(shema_predmeta);
        shema_predmeta = r1;
        provjera_sume(odrediKonacnuSumuRjesenja(r1));
        pocetno_rjesenje = r1;
    }
    vector<rjesenje> r2 = izbaci_jedan_predmet(r1); //izbaci drugi nasumični predmet
    if (odrediKonacnuSumuRjesenja(r2) > odrediKonacnuSumuRjesenja(pocetno_rjesenje)) {
        dealokacija(shema_predmeta);
        shema_predmeta = r2;
        provjera_sume(odrediKonacnuSumuRjesenja(r2));
        pocetno_rjesenje = r2;
    }
    vector<rjesenje> r3 = nasumično_dodaj_predmet(r2); //razmatraju se samo predmeti lakši od preostalog kapaciteta naprtnjače
    if (odrediKonacnuSumuRjesenja(r3) > odrediKonacnuSumuRjesenja(pocetno_rjesenje))
    {
        dealokacija(shema_predmeta);
        shema_predmeta = r3;
        provjera_sume(odrediKonacnuSumuRjesenja(r3));
        pocetno_rjesenje = r3;
    }
    vector<rjesenje> r4 = nasumično_dodaj_predmet(r3); //razmatraju se samo predmeti lakši od preostalog kapaciteta naprtnjače
    if (odrediKonacnuSumuRjesenja(r4) > odrediKonacnuSumuRjesenja(pocetno_rjesenje)) {
        dealokacija(shema_predmeta);
        shema_predmeta = r4;
        provjera_sume(odrediKonacnuSumuRjesenja(r4));
        pocetno_rjesenje = r4;
    }
}
```

Slika 27: Funkcija `Lokalna_jedan_korak(pocetno_rjesenje)` (Izvor: vlastita izrada)

U svakom koraku lokalne optimizacije kao što je vidljivo na slici 27 događaju se 4 modifikacije. Prva modifikacija je izbacivanje jednog slučajnog vrijednosnog predmeta iz ruksaka. Nakon toga se provjerava rješenje, a ako je ono bolje od početnog onda se zadržava inače se odbacuje. Druga modifikacija je izbacivanje još jednog slučajnog vrijednosnog predmeta iz ruksaka te provjera rješenja nakon toga. Kao što je bilo u prethodnom slučaju, ako je rješenje bolje od prethodnog onda se zadržava inače se odbacuje.

```
vector<rjesenje> izbaci_jedan_predmet(vector<rjesenje> rješenje) {
    vector<rjesenje>::iterator it;
    int brojacVrijednihPredmeta = 0;
    vector<rjesenje> izbaceniPredmet = rješenje;
    for (it = izbaceniPredmet.begin(); it != izbaceniPredmet.end(); ++it)
    {
        if (it->profit != 0)
            brojacVrijednihPredmeta++;
    }
    int slucajanPredmet = (int)randombroj(0, brojacVrijednihPredmeta);
    brojacVrijednihPredmeta = 0;
    for (it = izbaceniPredmet.begin(); it != izbaceniPredmet.end(); ++it)
    {
        if (it->profit != 0)
            brojacVrijednihPredmeta++;
        if (brojacVrijednihPredmeta == slucajanPredmet) {
            it->profit = 0;
            it->tezina = 0;
            break;
        }
    }
    return izbaceniPredmet;
}
```

Slika 28: Funkcija `izbaci_jedan_predmet(rjesenje)` (Izvor: vlastita izrada)

Na slici 28 možemo vidjeti na koji način se događa nasumično izbacivanje predmeta. Prvo se prebroje svi vrijednosni predmeti u ruksaku, a potom se odredi slučajan broj između 0 i ukupnog broja tih predmeta. Predmetu kojeg se treba izbaciti se potom dodijeli vrijednost 0 za profit i vrijednost 0 za težinu čime on postaje bezvrijedan predmet u tome gradu. Treća modifikacija dodavanje jednog novog predmeta (u obzir dolaze samo predmeti čija je težina manje od trenutnog kapaciteta ruksaka. Nakon što se doda jedan slučajan predmet u ruksak onda se ponovno provjerava rješenje i ako se radi o boljem rješenju onda se zadržava inače se odbacuje. To isto se događa s četvrtom modifikacijom koja dodaje još jedan slučajan predmet u ruksak, a onda uspoređuje to rješenje s rješenjem početno konstruiranog rješenja. Ovo što je ovdje opisano je samo jedan korak lokalne optimizacije, a ako se postavi veći broj onda je moguće da se izbacivanje i dodavanje predmeta događa više puta čime se može uvelike utjecati na poboljšanje samog rješenja.

```
vector<rjesenje> nasumično_dodaj_predmet(vector<rjesenje> rješenje) {
    vector<rjesenje>::iterator it;
    vector<rjesenje> predmeti = novi;
    double tezinaRuksaka = kapacitet_ruksaka;
    rjesenje* noviPredmet = new rjesenje();
    for (it = rješenje.begin(); it != rješenje.end(); ++it)
    {
        tezinaRuksaka = tezinaRuksaka - it->tezina;
    }
    bool pronaden = true;
    for (int i = 0; i < 10; i++)
    {
        int slucajanGrad = (int)randombroj(0, gradovi.size());
        for (it = novi.begin(); it != novi.end(); ++it) {
            if (it->grad == slucajanGrad && tezinaRuksaka > it->tezina&& it->tezina != 0)
            {
                noviPredmet->broj_predmeta = it->broj_predmeta;
                noviPredmet->grad = it->grad;
                noviPredmet->profit = it->profit;
                noviPredmet->tezina = it->tezina;
                std::vector<rjesenje>::iterator mi;
                for (mi = rješenje.begin(); mi != rješenje.end(); ++mi) {
                    if (mi->broj_predmeta == noviPredmet->broj_predmeta) {
                        pronaden = true;
                        break;
                    }
                    if (mi == rješenje.end() - 1)
                        pronaden = false;
                }
                if (!pronaden)
                    break;
            }
        }
    }
    if (!pronaden) {
        int brojacPozicija = 0;
        for (it = rješenje.begin(); it != rješenje.end(); ++it) {
            brojacPozicija++;
            if (it->grad == noviPredmet->grad) {
                break;
            }
        }
        rješenje.insert(rješenje.begin() + brojacPozicija, *noviPredmet);
    }
    return rješenje;
```

Slika 29: Funkcija `nasumično_dodaj_predmet(rjesenje)` (Izvor: vlastita izrada)

Slika 29 nam prikazuje funkciju nasumično_dodaj_predmet(rjesenje) koja funkcionira na način da prvo odredi trenutni kapacitet ruksaka, a potom slučajno odabere grad. Nakon slučajnog odabira grada, funkcija ide kroz vektor predmeta i ispituje da li se taj predmet nalazi u tome gradu i da li je njegova težina veća od 0 i manja od trenutnog kapaciteta ruksaka. Ako je uvjet prošao onda se moramo osigurati da se taj predmet već ne nalazi u rješenju. Ako se ne nalazi u rješenju onda se izlazi iz petlje i dodaje se taj predmet na mjesto u vektoru koje redoslijedno ide za taj grad. Ako se kojim slučajem u deset pokušaja ne uspije dodati novi predmet, onda se vraća početno rješenje te se odustaje od lokalne optimizacije. Ovaj uvjet je dodan ne bi došlo do beskonačne petlje prilikom koje nikad ne bi našli predmet koji nam odgovara.

7. Eksperimenti i rezultati eksperimenata

Provođeni su različiti eksperimenti problema putujućeg lopova na prethodno objašnjenome algoritmu. U ovom poglavlju će biti prikazani te analizirani rezultati tih eksperimenata. Problem putujućeg lopova je moderan problem koji je zainteresirao znanstvenike zbog toga što je to višekomponentni problem koji je približen stvarnim problemima s kojima se velike i male tvrtke danas susreću. Do danas je kreiran veliki broj instanci ovoga problema koji se međusobno razliku po: veličini gradova, broju predmeta po pojedinom gradu, kapacitetu ruksaka te cijeni iznajmljivanja ruksaka. Eksperimenti u radu su provedeni na :

- 51 gradu
- 76 gradova
- 100 gradova i
- 280 gradova.

Po karakteristici broja predmeta po gradovima smo razlikovali:

- TTP s po jednim predmetom u gradu i
- TTP s po pet predmeta u gradu

Razlikovali smo eksperimente također po karakteristici korištenja lokalne optimizacije pa tako smo razlikovali:

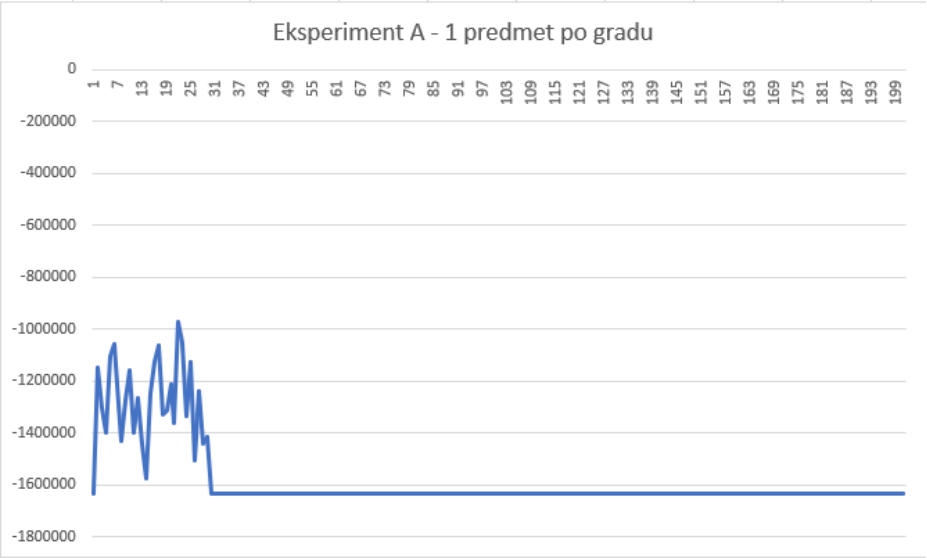
- Rješenje TTP-a koje ne koristi lokalnu optimizaciju i
- Rješenje TTP-a koje koristi lokalnu optimizaciju

Uz sve prethodno navedeno, provedeno je ukupno 25 eksperimenata koji su se razlikovali po prethodno navedenim karakteristikama. Za početak ćemo objasniti prve eksperimente koji su provedeni za ispitivanje o načinu implementacije lokalne optimizacije u algoritmu. U svim slikama grafova X-os predstavlja broj iteracije, a Y-os predstavlja vrijednost najbolje sume u toj iteraciji, odnosno vrijednost najbolje rješenja u iteraciji (Slika 30 – Slika 50).

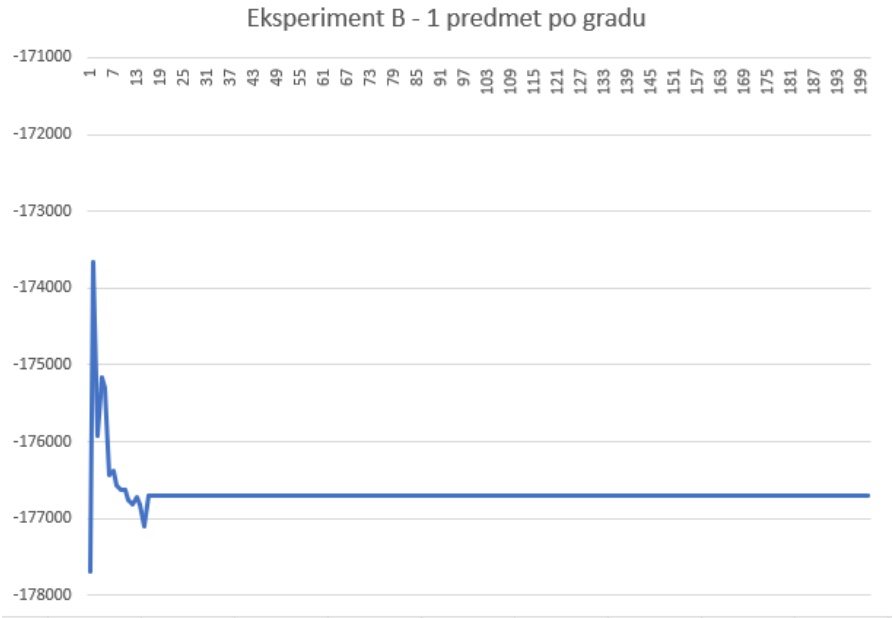
Podešavanja broja ponavljanja u lokalnoj optimizaciji

Kako bismo implementirali jednostavnu lokalnu optimizaciju prvo moramo provjeriti ima li smisla samo dodavati nasumične predmete u ruksak ili kvalitetu rješenja možemo povećati i izbacivanjem određenih predmeta iz ruksaka. Eksperiment će se vršiti po principu da uzmemo neko gotovo rješenje te slijedno izbacujemo po jedan predmet i računamo kvalitetu rješenja nakon svakog pojedinog izbačenog predmeta. Izbacujemo samo po jedan predmet u ruksaku,

a sve prethodne vraćamo u ruksak. Kako bi mogli analizirati rezultate, napravit ćemo dva takva eksperimenta od kojih će prvi biti proveden na način da se iz početnih iteracija, kada algoritam još uvijek uči, uzme rješenje te ga se provede kroz korake eksperimenta, a drugi će biti proveden na način da ćemo iz zadnjim iteracijama, kada je algoritam već dovoljno dobro naučio optimalni put, uzeti rješenje te ga provesti kroz korake eksperimenta. Svaki eksperiment će sadržavati broj pokušaja koji smo postavili na 200 i on će označavati broj vrijednosnih predmeta koji će biti slijedno izbačeni iz ruksaka. Nakon svakog izbacivanja će biti zapisana kvaliteta rješenja u obliku sume, a kvalitetu eksperimenata ćemo prikazati pomoću grafa vrijednosti tih suma. X os predstavlja broj iteracije, dok Y os predstavlja vrijednosti suma u pojedinim iteracijama.



Slika 30: Eksperiment A - 1 predmet po gradu (Izvor: vlastita izrada)



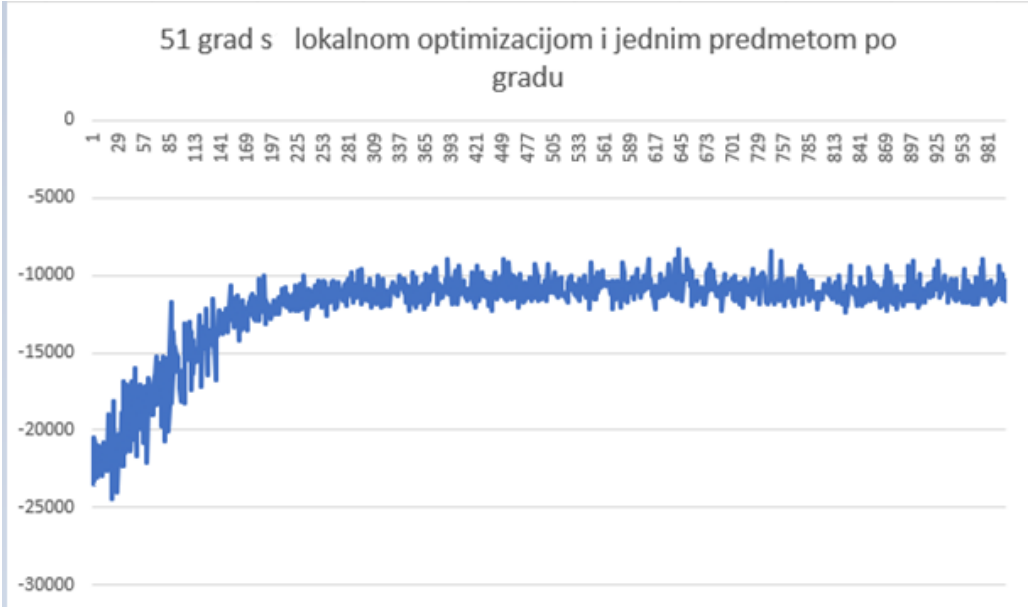
Slika 31: Eksperiment B - 1 predmet po gradu (Izvor: vlastita izrada)

Eksperiment A je eksperiment koji je proveden u početnim iteracijama, dok je eksperiment B proveden na rješenju u kasnijim iteracijama. Na slikama 30 i 31 su prikazani grafovi, a ono što možemo vidjeti su određene promjene. Početna točka 1 označava početno rješenje prije bilo kakvog izbacivanja, a sve ostale točke predstavljaju sumu rješenja kada je izbačen jedan predmet. Možemo vidjeti kako oba grafa imaju uspone i padove iz čega bi se dalo zaključiti da u implementaciji lokalne optimizacije valja uzeti u obzir i izbacivanje predmeta iz ruksaka kojim bi se poboljšalo samo rješenje. U KP problemu bi izbacivanje predmeta iz ruksaka rezultiralo pogoršanjem kvalitete rješenja dok ovdje moramo uzeti u obzir da je ruksak iznajmljen te se lopov kreće sporije ako mu je ruksak puniji predmetima. Na osnovi ovog eksperimenta je kreirana funkcija `Lokalna_optimizacija(rjesenje, broj_koraka)` koja je objašnjena u poglavlju programskog koda. Ovdje možemo primijetiti kako bi bilo dobro broj koraka smanjiti na približno 20 jer se u prvih 20 iteracija događaju određene promjene. U ovim eksperimentima zabilježeni su rezultati suma koji su utjecali na promjenu rješenja, dok u iteracijama kod kojim se nije dogodila značajna promjena kvaliteta rješenja vidljiva je stagnacija, odnosno ravna linije vidljiva na slikama 30 i 31.

Eksperimenti s 51 gradom

Nad problemom s 51 gradom napravili smo četiri eksperimenta i to su redom sljedeći: 51 grad bez lokalne optimizacije i jednim predmetom po gradu, 51 grad s lokalnom optimizacijom i jednim predmetom po gradu, 51 grad bez lokalne optimizacije i pet predmeta po gradu te eksperiment s 51 gradom s lokalnom optimizacijom i pet predmeta po gradu.

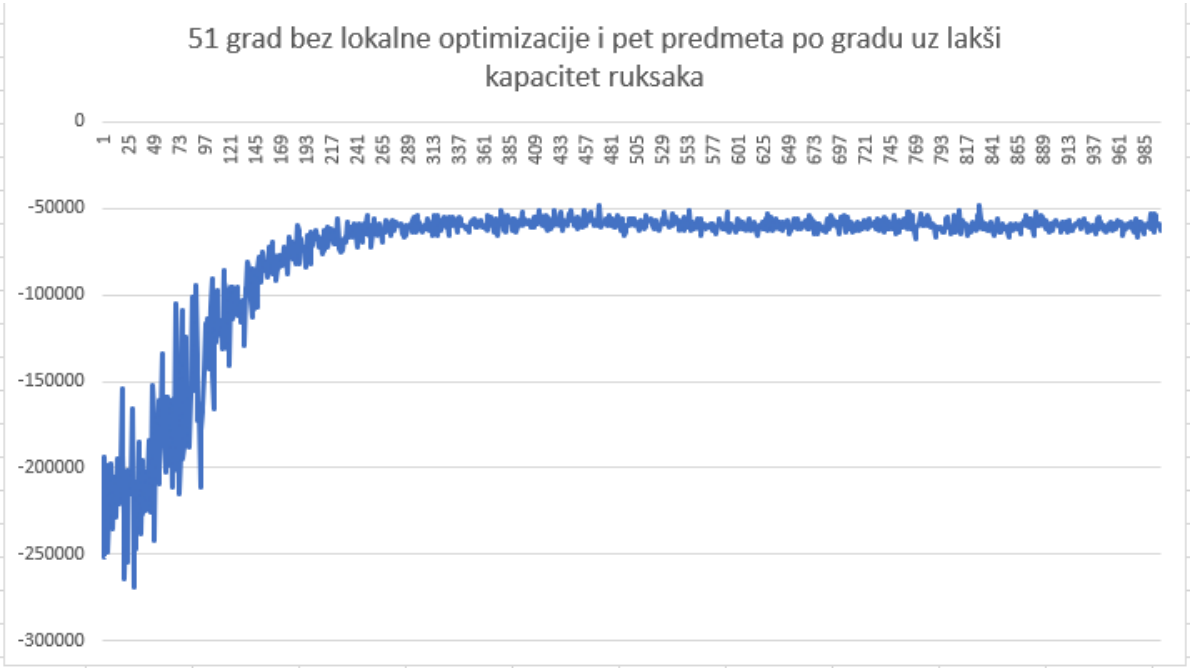
Slika 32: 51 grad bez lokalne optimizacije i jednim predmetom po gradu (Izvor: vlastita izrada)



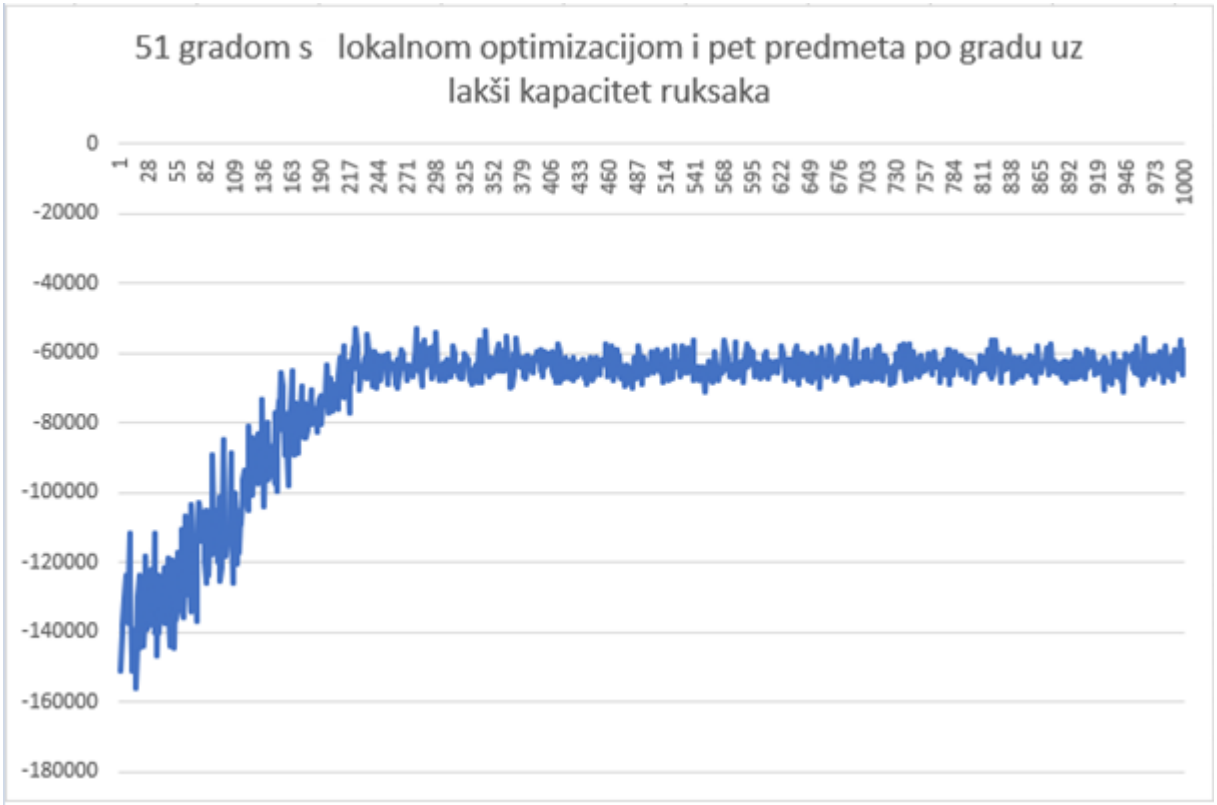
Slika 33: 51 grad s lokalnom optimizacijom i jednim predmetom po gradu (Izvor: vlastita izrada)

Na slikama 32 i 33 prikazani su rezultati za isti problem samo s različitim pristupom problemu. Na slici 32 problem su rezultati problema koji je rješavan mravljim algoritmom bez lokalne optimizacije, dok su na slici 33 prikazani rezultati istog problema uz korištenje lokalne optimizacije. Možemo vidjeti da su najbolja rješenja iteracije puno bolja u početku kod korištenja lokalne optimizacije. Također je vidljivo kako u kasnijem dijelu graf na slici 33 lagano prelazi -10 000 te se kreće oko te vrijednosti za razliku od prikaza na slici 32 gdje linija grafa ide ispod vrijednosti -10 000 te u nekim točkama dotiče tu vrijednost. Na ove dvije slike jasno možemo vidjeti razlike korištenja lokalne optimizacije te poboljšanja koja ona daje algoritmu. Prema ovome vidljivo je kako u početku, dok algoritam uči, lokalna optimizacije daje puno bolja rješenja za razliku od onih prikazanih na slici 32 bez lokalne optimizacije.

Na slikama 34 i 35 gdje se broj predmeta po gradu povećao možemo vidjeti sličan rezultat kao na prethodne dvije slike. U slučaju korištenja lokalne optimizacije kvaliteta rješenja u početku je znatno bolja nego u slučaju kada se ona ne koristi. Lokalna optimizacije uvelike utječe na brzinu izvođenja programskog koda pa je ona razlog zbog kojega se program duže izvodi, ali kako vidljivo na ovim grafovima poboljšava kvalitetu samog rješenja onda možemo zanemariti brzinu izvođenja jer se programski kod uspije izvršiti u realnom vremenu uz određeno zaostajanje za programskim kodom koji ju ne koristi. Ovdje možemo primijetiti da nema prevelike razlike u kasnijoj fazi kao što se mogla vidjeti na slikama 32 i 33, ali još uvijek postoji razlika kod početnog učenja kao što je vidljivo na priloženim grafovima.



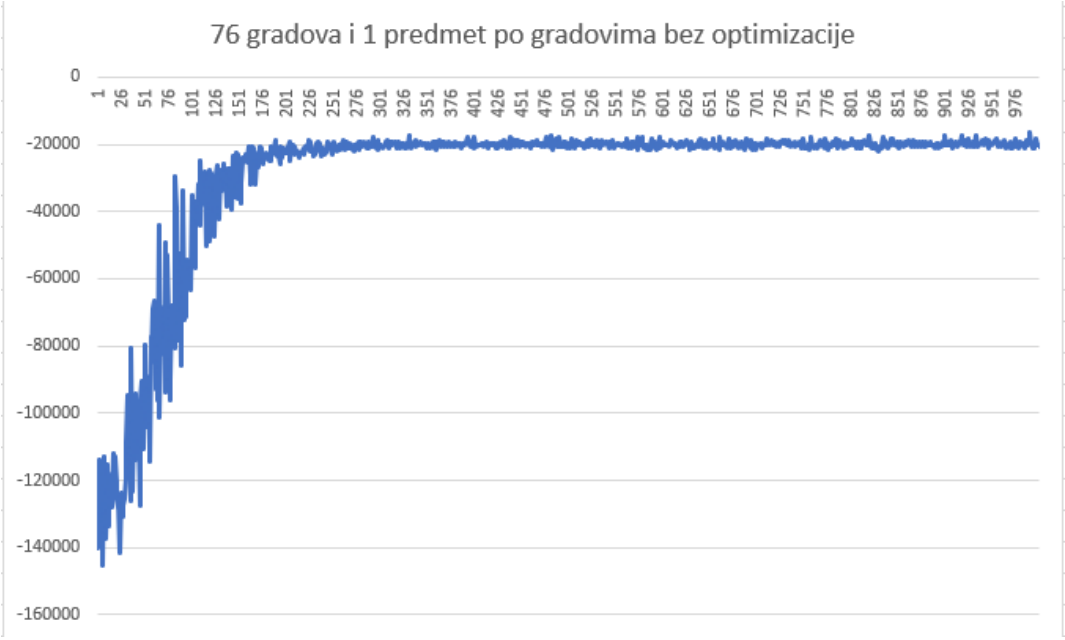
Slika 34: 51 grad bez lokalne optimizacije i pet predmeta po gradu uz lakši kapacitet ruksaka (Izvor: vlastita izrada)



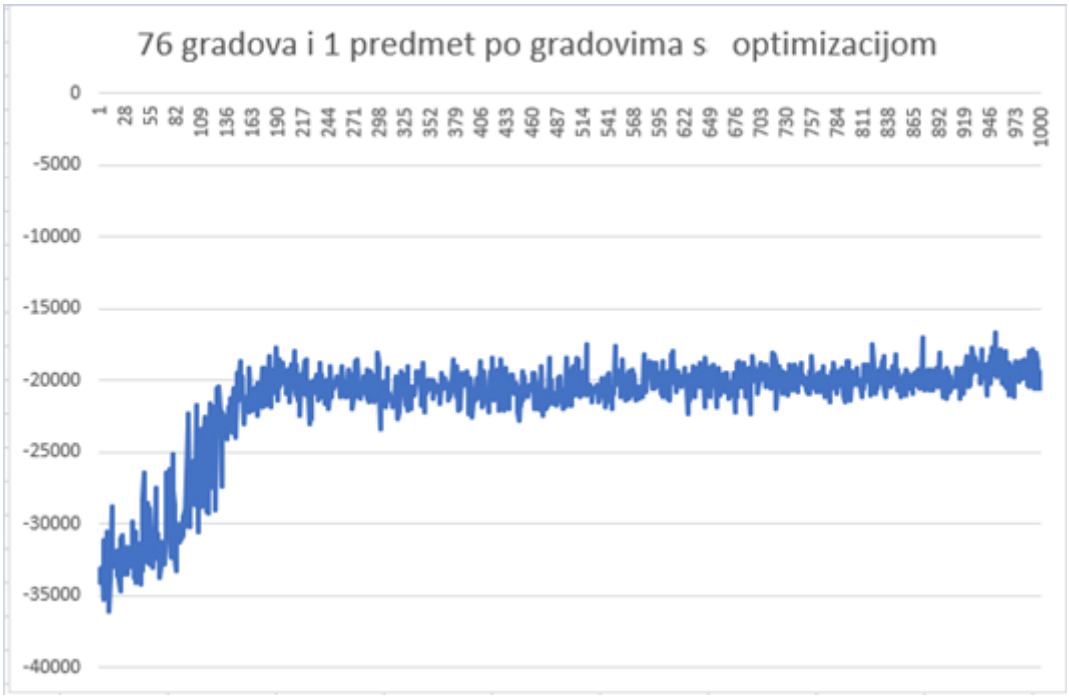
Slika 35: 51 gradom s lokalnom optimizacijom i pet predmeta po gradu uz lakši kapacitet ruksaka (Izvor: vlastita izrada)

Eksperimenti s 76 gradova

Kod problema s 76 gradova proveli smo iste eksperimente kao kod problema s 51 gradom. Ovaj problem se u literaturi može pronaći pod nazivom pr76-TTP. U odnosu na ostale probleme na kojima su provođeni eksperimenti, ovaj problem ima jeftiniju cijenu najma ruksaka te su sami predmeti u gradovima lakši u usporedbi s ostalim predmetima.

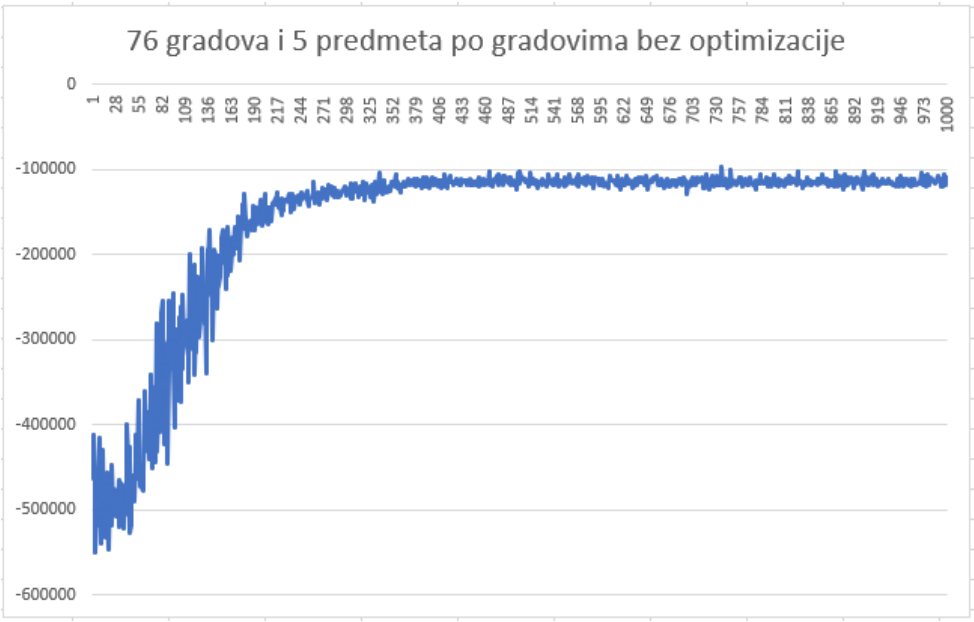


Slika 36: 76 gradova i 1 predmet po gradovima bez optimizacije (Izvor: vlastita izrada)



Slika 37: 76 gradova i 1 predmet po gradovima s optimizacijom (Izvor: vlastita izrada)

Na slikama 36 i 37 su vidljivi rezultati najboljih suma u svakoj iteraciji za isti problem s i bez korištenja lokalne optimizacije. Kao u problemu s 51 gradom imamo slične rezultate. Lokalna optimizacija značajno poboljšava početno konstruiranje rješenja pa zbog toga slika 36 ima puno veći uspon grafa odnosno učenje od grafa na slici 37. Možemo vidjeti također da su rezultati u fazi stagnacije rješenja malo bolji kod korištenja lokalne optimizacije jer na slici 37 graf u fazi stagnacije prelazi crtu od -20 000 dok na slici 36 graf se kreće uz crtu od -20 000 s tim da određene točke prelaze tu vrijednost. Lokalna optimizacija se i na ovom primjeru pokazala kao korisno poboljšanje algoritma te su njeni rezultati vidljivi u početcima, dok algoritam uči i istražuje optimalni put.



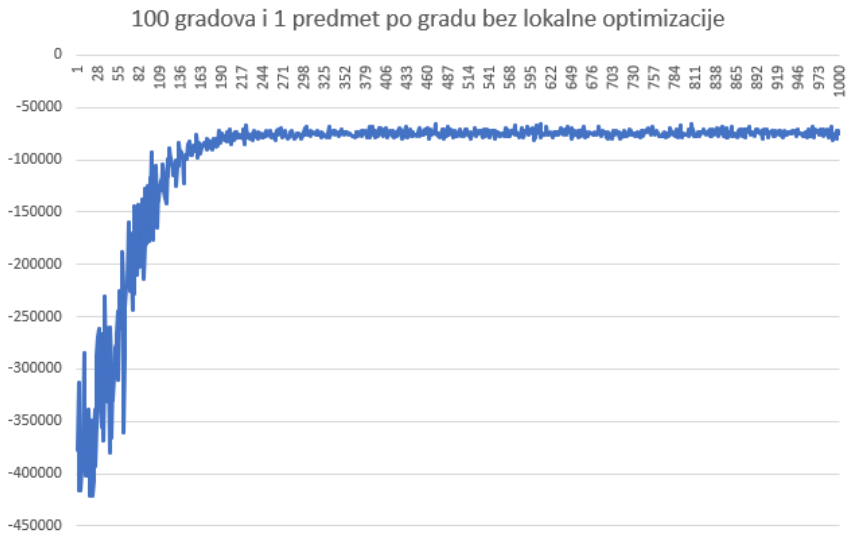
Slika 38: 76 gradova i 5 predmeta po gradovima bez optimizacije (Izvor: vlastita izrada)

Slika 39: 76 gradova i 5 predmeta po gradovima s optimizacijom (Izvor: vlastita izrada)

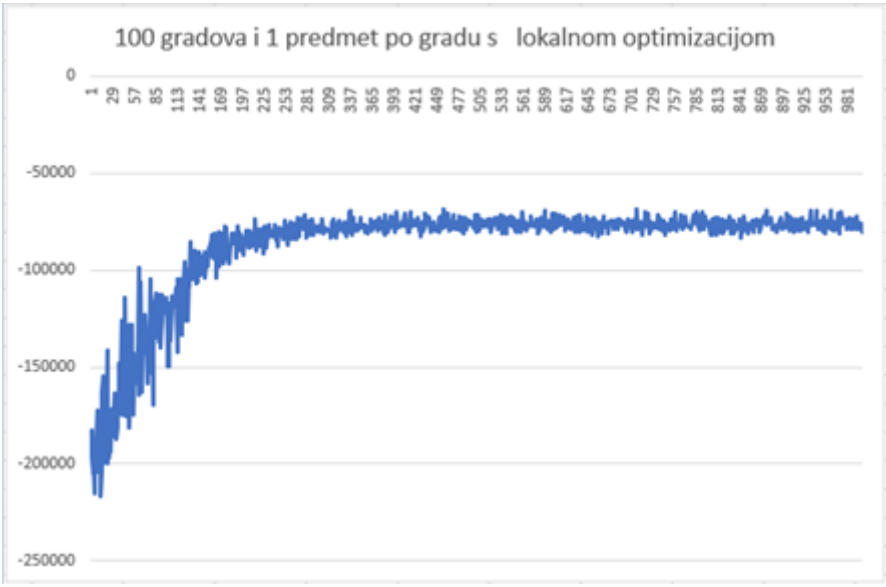
Na slikama 38 i 39 se nalaze grafovi rješenja za problem TTP-a s 76 gradova i 5 predmeta po gradovima s i bez korištenja lokalne optimizacije. Rezultati su slični s rezultatima problema s 51 gradom i 5 predmeta po gradovima na način da lokalna optimizacija poboljšava kvalitetu rješenja u početku, dok u fazi stagnacije i ne daje neko vidljivo poboljšanje algoritmu.

Eksperimenti s 100 gradova

Nad problemom s 100 gradova su provedeni eksperimenti gdje smo u glavni fokus također stavili uporabu lokalne optimizacije te broj predmeta po gradovima 1 i 5. Ovaj problem se u literaturi može pronaći pod nazivom kroA100-TTP.

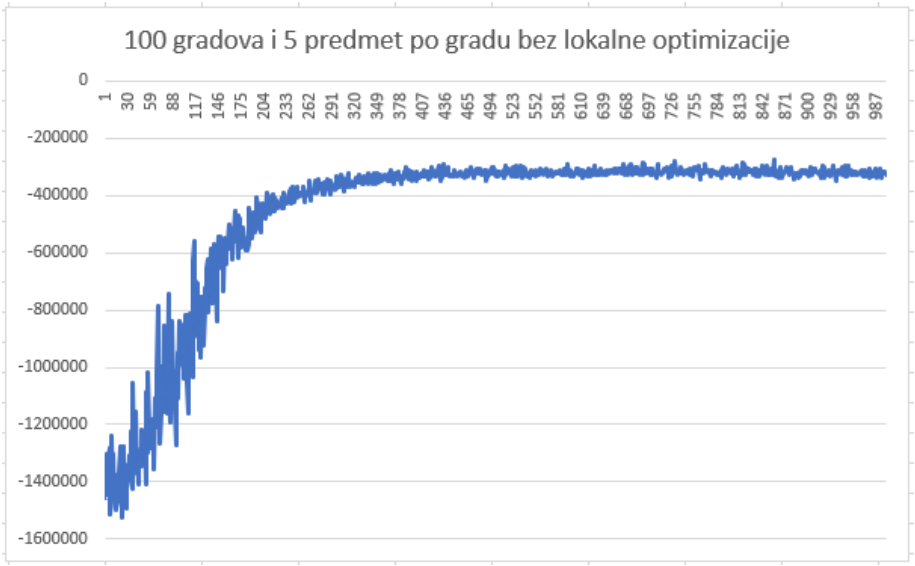


Slika 40: 100 gradova i 1 predmet po gradu bez lokalne optimizacije (Izvor: vlastita izrada)

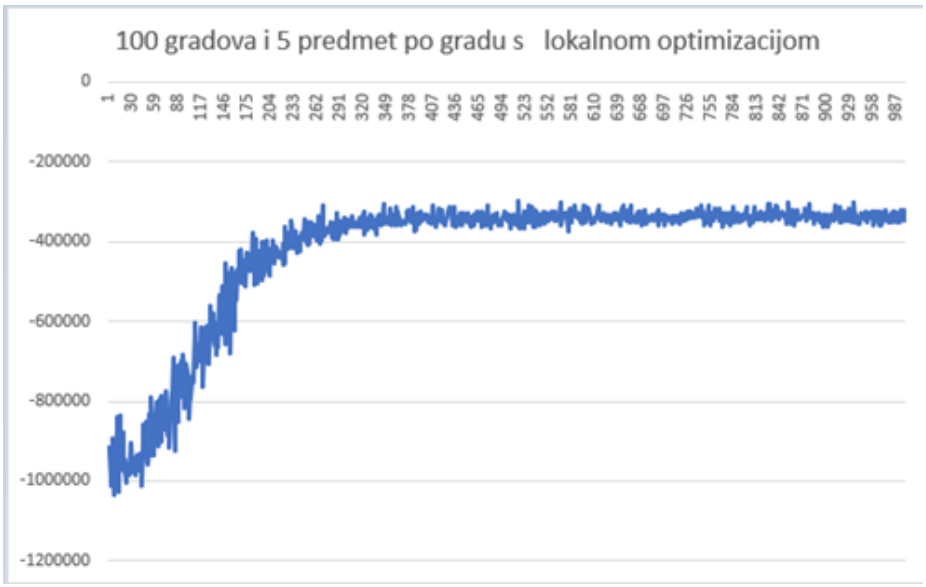


Slika 41: 100 gradova i 1 predmet po gradu s lokalnom optimizacijom (Izvor: vlastita izrada)

Na slikama 40 i 41 možemo vidjeti grafove za problem 100 gradova i jednim predmetom po gradu s i bez lokalne optimizacije. Kao i na prijašnjim primjerima grafova, vidimo znatno poboljšanje na grafu koji koristi lokalnu optimizaciju. Na slici 41 mravi puno prije dođu do konstrukcije optimalnog rješenja nego mravi na slici 40. Lokalna optimizacije omogućuje mravima da istražuju rješenja puno duže jer mravi uspiju konstruirati relativno optimalno rješenje nakon manjeg broja iteracija, a potom im ostaje još puno vremena prije nego li dođu u fazu stagnacije u kojoj su koeficijenti (feromonski tragovi) raspoređeni na one koji se nalaze oko τ_{min} i one koji se nalaze oko τ_{max} pa je zbog toga mravima puno teže istraživati konstrukciju novih rješenja.



Slika 42: 100 gradova i 5 predmet po gradu bez lokalne optimizacije (Izvor: vlastita izrada)

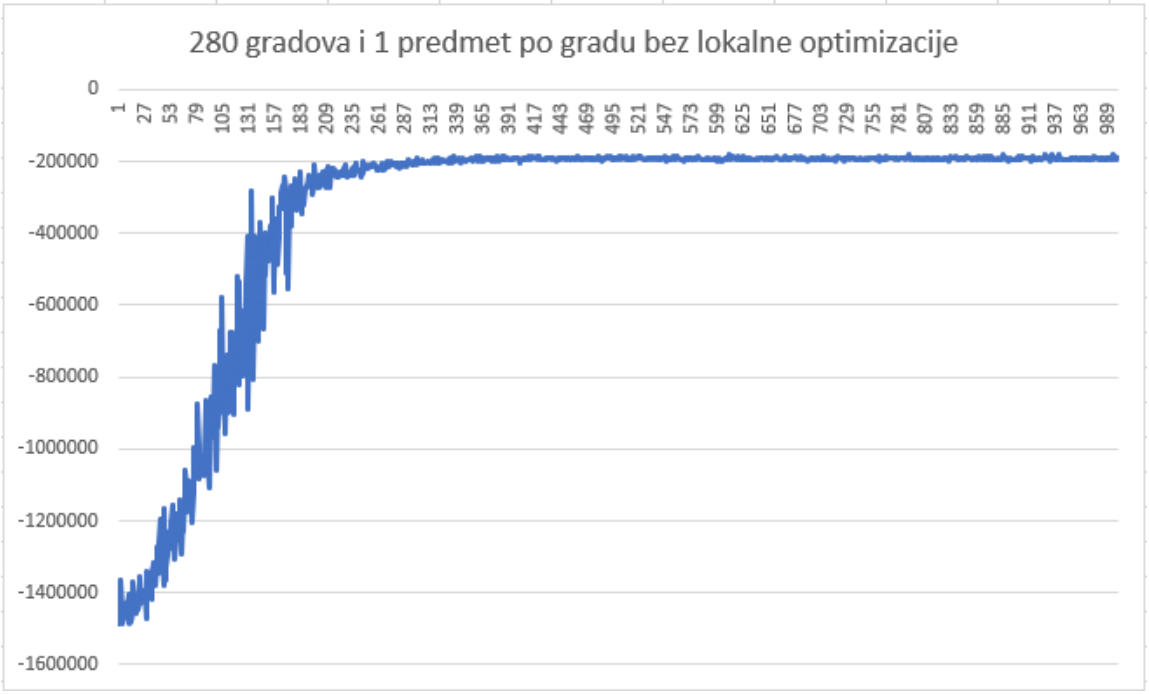


Slika 43: 100 gradova i 5 predmet po gradu s lokalnom optimizacijom (Izvor: vlastita izrada)

Na slikama 42 i 43 možemo vidjeti grafove koji predstavljaju proces konstrukcije optimalnog rješenja kroz određeni vremenski period sa i bez lokalne optimizacije te pet predmeta po gradu. Kao i svaki prethodni problem koji je sadržavao pet predmeta po gradu, ovdje također možemo povući neke sličnosti kao što je učenje sa i bez lokalne optimizacije. Ono je isto kao i u prethodnim slučajevima te je vidljivo na slici 43 da algoritam koji koristi lokalnu optimizaciju u početku konstruira kvalitetnija rješenja od onih koji ju ne koriste. Možemo zamijetiti kako se rješenja na slici 42 u početku kreću oko vrijednosti -140 000 dok se na slici 43 koji koristi lokalnu optimizaciju rješenja u početku kreću oko vrijednosti -90 000.

Eksperimenti s 280 gradova

Kod problema s 280 gradova proveli smo iste eksperimente kao kod ostalih problema. Ovaj problem se u literaturi može pronaći pod nazivom a280-TTP. Ovo je posljednji problem koji smo testirali na implementiranom algoritmu. Za veći broj gradova se zahtijevaju drukčiji pristupi u implementaciji algoritma koji bi znatno poboljšali samu konstrukciju rješenja. Ovaj problem s 280 gradova je bio jedan od testnih primjera na natjecanju CEC competition at IEEE WCCI 2014: Optimisation of Problems with Multiple interdependent components u Beijing-u.

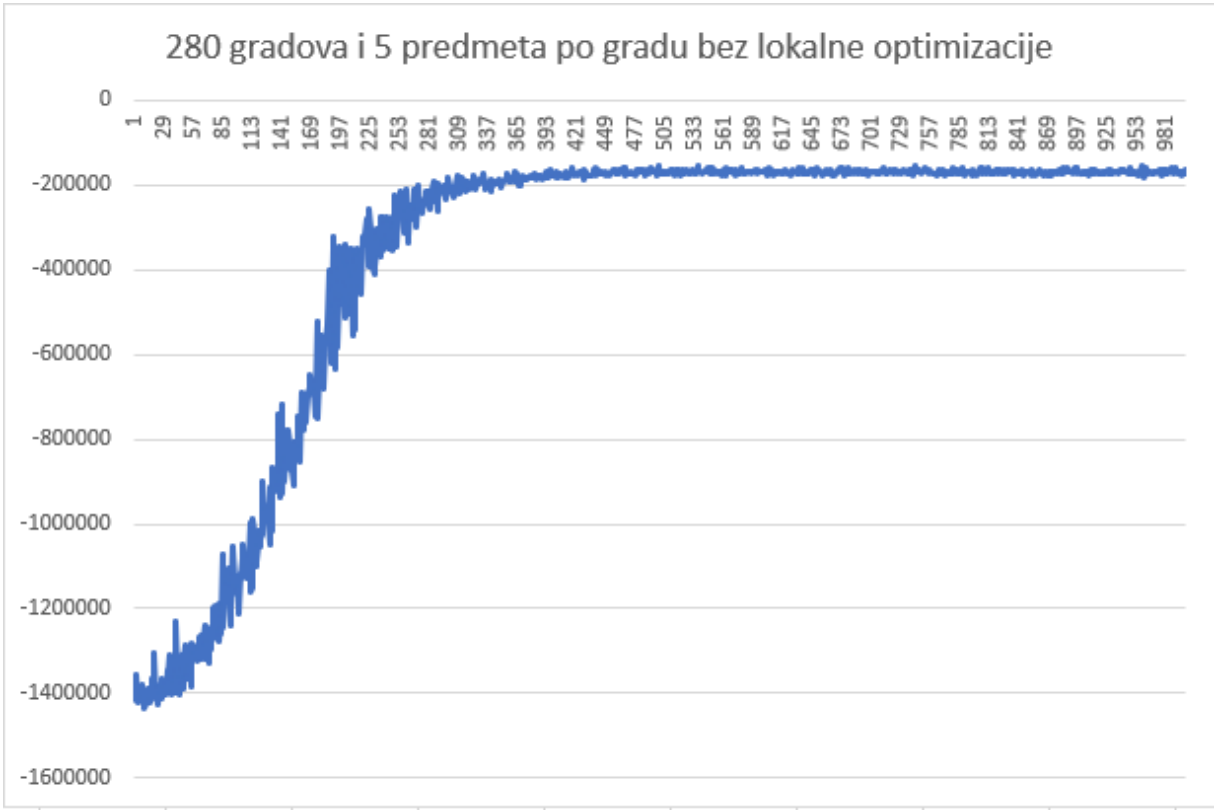


Slika 44: 280 gradova i 1 predmet po gradu bez lokalne optimizacije (Izvor: vlastita izrada)

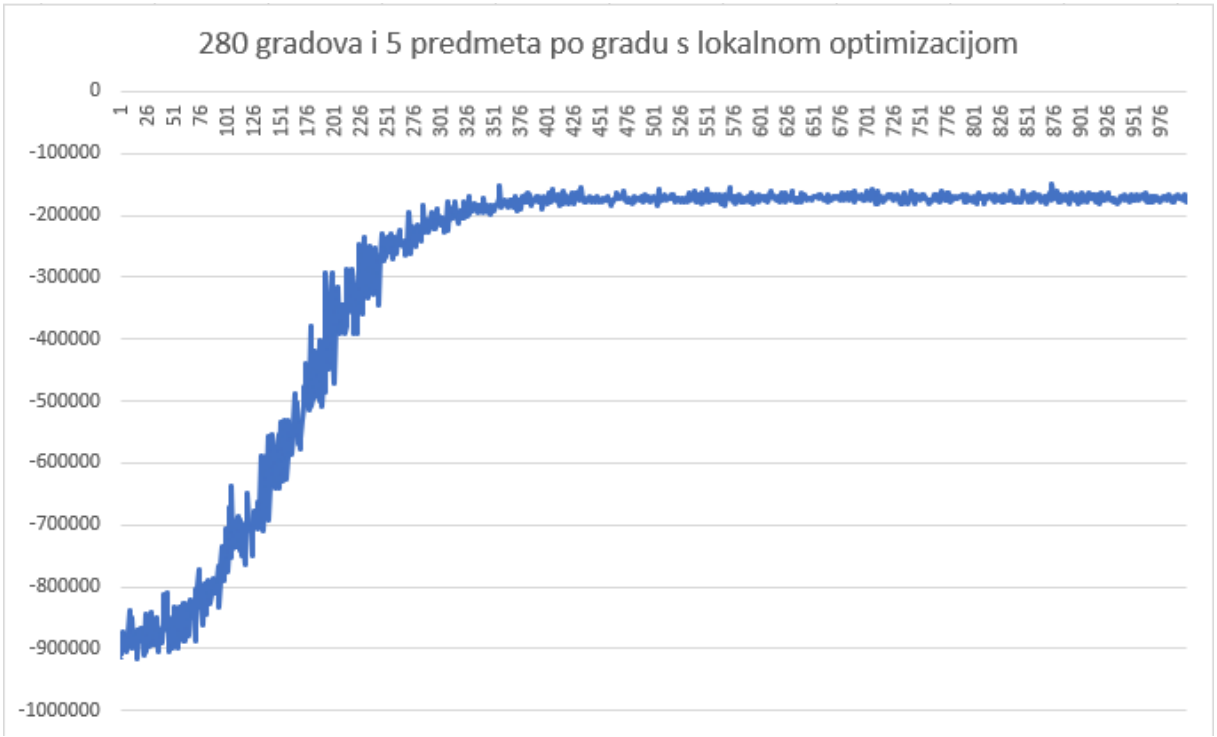
Slika 45: 280 gradova i 1 predmet po gradu s lokalnom optimizacijom (Izvor: vlastita izrada)

Na slikama 44 i 45 možemo vidjeti dva grafa koja predstavljaju konstrukciju rješenja po iteracijama za problem a280-TTP sa i bez lokalne optimizacije. Ovaj zadnji eksperiment s dosad najvećim brojem gradova dokazao je ono što smo zamijetili u posljednja tri eksperimenta, a to je da lokalna optimizacije uvelike doprinosi poboljšanju mravljeg algoritma za problem putujućeg lopova. Poboljšanje lokalne optimizacije vidljivo kada usporedimo početak grafa na slici 44 i početak grafa na slici 45 gdje jasno vidimo da su velike razlike u kvaliteti rješenja u početnim iteracijama. Faza u kojoj algoritam ne uči ništa novo, odnosno faza stagnacije na oba grafa dolazi oko 200-te iteracije samo što je razlika kvaliteta rješenja s i bez lokalne optimizacije. Slika 45 u fazi stagnacije poprima puno bolja rješenja, na grafu je vidljivo da rješenja idu ispod -200 000, dok na slici 44 u fazi stagnacije rješenja idu uz liniju -200 000.

Na slikama 46 i 47 su prikazani grafovi rješenja za problem a280-TTP gdje svaki grad sadrži po pet predmeta s i bez uporabe lokalne optimizacije. Kao i kod prethodnih grafova koji su opisivali rješenja problema koji sadrže po 5 predmeta u svakom gradu ovdje je vidljiv sličan rezultat. Uporaba lokalne optimizacije poboljšava početnu kvalitetu rješenja te omogućuje algoritmu da u početku bolje uči te kasnije kreira bolja rješenja. U fazi stagnacije se ne vidi znatno poboljšanje, ali ono postoji. Ovdje smo prilikom testiranja koristili lakši kapacitet, odnosno kapacitet ruksaka koji je namijenjen problemu 280 gradova i 1 predmetom po gradu.



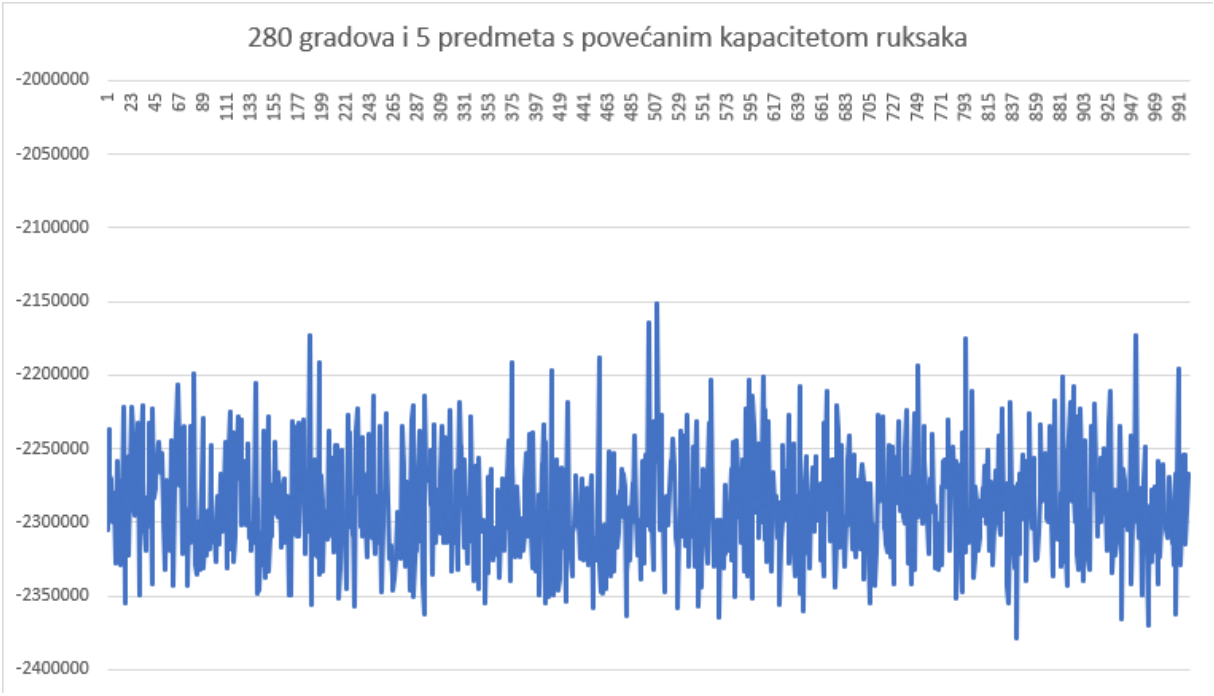
Slika 46: 280 gradova i 5 predmeta po gradu bez lokalne optimizacije (Izvor: vlastita izrada)



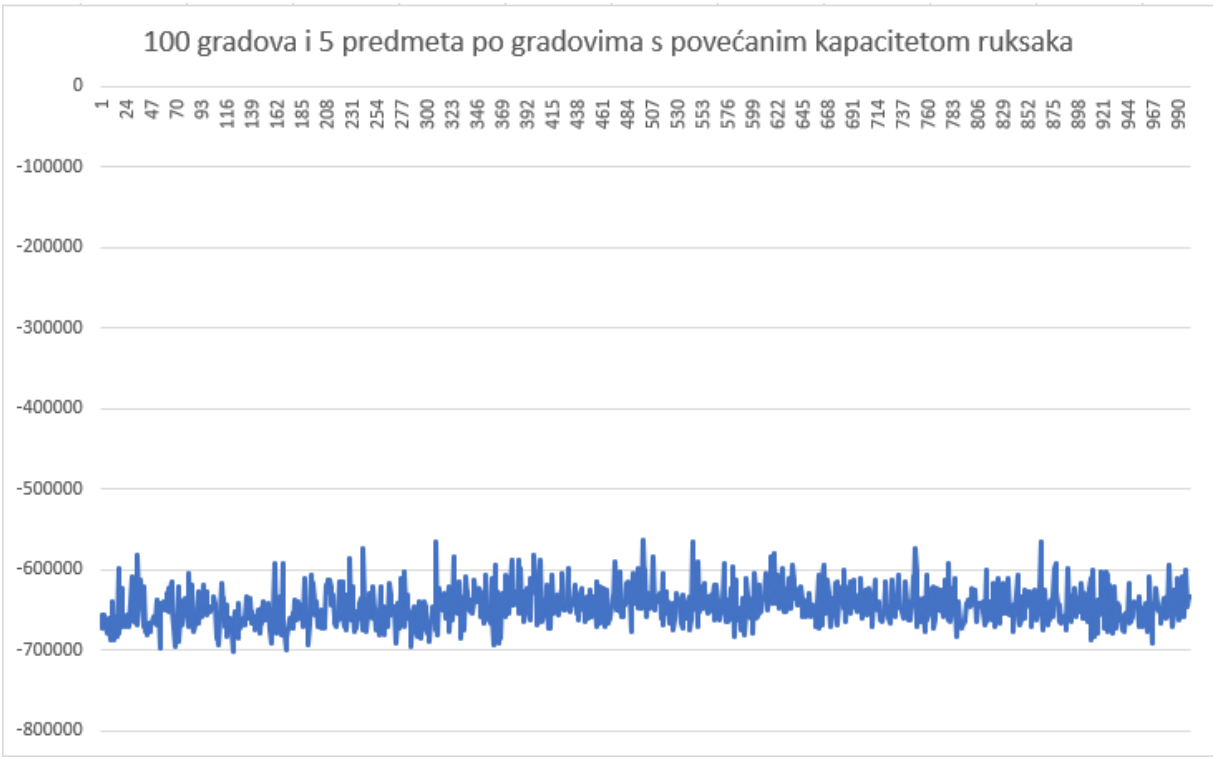
Slika 47: 280 gradova i 5 predmeta po gradu s lokalnom optimizacijom (Izvor: vlastita izrada)

Eksperimenti s povećanim kapacitetom ruksaka

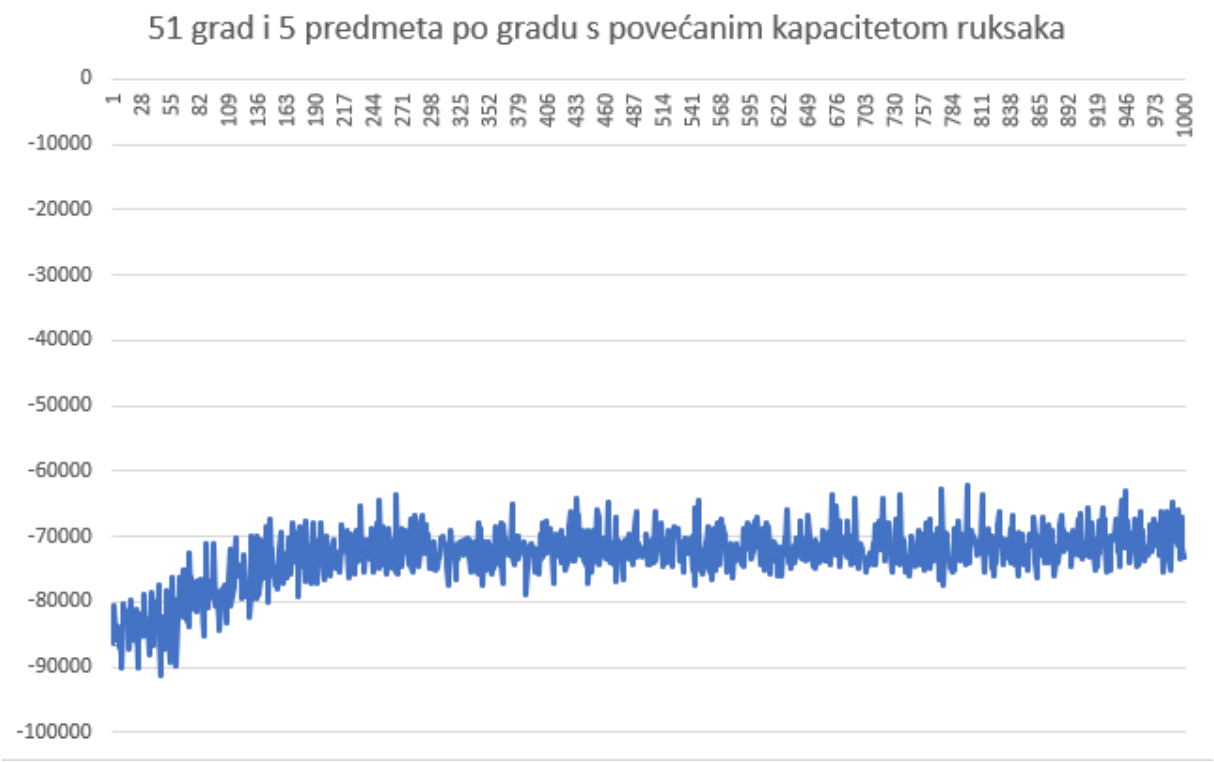
Prilikom eksperimentiranja, određeni testovi su pušteni s većim kapacitetom i naravnom većom cijenom najma, a u nastavku odlomka će biti predstavljeni rezultati.



Slika 48: 280 gradova i 5 predmeta s povećanim kapacitetom ruksaka (Izvor: vlastita izrada)



Slika 49: 100 gradova i 5 predmeta po gradovima s povećanim kapacitetom ruksaka (Izvor: vlastita izrada)



Slika 50: 51 grad i 5 predmeta po gradu s povećanim kapacitetom ruksaka (Izvor: vlastita izrada)

Na slikama 48, 49 i 50 su predstavljeni grafovi različitih tipova problema putujućeg lopova na kojima su kapacitet ruksaka i cijena najma ruksaka bili povećani. Ovi rezultati ne pokazuju napredak algoritma kroz vrijeme, odnosno u ovim primjerima algoritam ne uči najbolje. Možemo vidjeti kako se sva kreirana rješenja mogu staviti u jedan interval pa i to može biti razlog zbog kojeg algoritam ne uči najbolje kada mu se ove dvije postavke promijene na veće. Na slici 50 možemo vidjeti mali napredak, odnosno vidljivo je učenje algoritma kroz iteracije, ali ono je izrazito malo dok na slikama 48 i 49 ono ne postoji. U svakom slučaju potrebna je detaljna analiza koja bi pojasnila ovakve rezultate. Dvije su mogućnosti problema, a to su da algoritam u ovom slučaju nema dobra rješenja pa zbog toga ni ne može učiti i druga mogućnost je da je krivi pristup, odnosno da algoritam zahtjeva drukčiju implementaciju za problem kod kojega su povećani kapacitet ruksaka i cijena najama. Isti primjeri problema su pušteni s smanjenim kapacitetom te su rezultati prikazani u prijašnjim poglavljima, a na tim grafovima je vidljiv napredak kroz iteracije. Postoji mogućnost da zbog izrazito velike cijene najma i kapaciteta ruksaka nije moguće kreirati dobro rješenje koje će usmjeriti algoritam na kreiranje kvalitetnijih rješenja.

Utjecaj koeficijenta iznajmljivanja ruksaka na rješenje

Posljednji element koji je testiran u ovom radu je koeficijent iznajmljivanja ruksaka *R*. Uzet je određen broj konačnih rješenja na kojima su vršene promjene koeficijenta *R* uz praćenje konačne sume rješenja. Dobiveni rezultati na manjem broju primjera ukazuju na to da smanjenjem iznajmljivanja ruksaka za 90% i više, odnosno postavljenja *R* na probleme koji imaju manji kapacitet ruksaka pa zbog toga i manji *R* uvelike utječe na krajnju kvalitetu rješenja pa je tako rješenja koja su do tada bila u minusu, postala pozitivna, a rješenja koja su bila blizu nule su skočile u veliki plus. Samo postavljanje problema znatno utječe na kvalitetu rješenja. Koeficijent *R* je značajan za formulu po kojoj se računa kvaliteta rješenja za problem putujućeg lopova. Važnost koeficijenta *R*, odnosno cijene iznajmljivanja ruksaka vidimo ovdje te njezinu ulogu u opisu stvarnog problema ne smijemo zanemariti. Problem putujućeg lopova je višekomponentni problem koji je približen stvarnim problemima, a upravo svako smanjenje određenih varijabli u jednadžbi po kojoj se računa konačno rješenje treba biti pažljivo razmatrano i uzeto u obzir. Problem putujućeg lopova je veoma osjetljiv problem gdje određene varijable možemo promijenit za sitan iznos, a krajnji rezultat prije i poslije promjene će tvoriti ogromnu razliku.

8. Zaključak

Današnje tvrtke susreću se sa sve većim i kompliciranim zahtjevima kada su u pitanju optimizacija cijelog pogonskog procesa te minimizacija troškova i maksimizacija profita. Problem putujućeg lopova je prvi višekomponentni problem koji kombinira problem trgovačkog putnika (TSP) i problem naprtnjače (KP) koji su dobro poznati problemi u svijetu optimizacije. Sergey Polyakovskiy predstavlja prve primjerke problema putujućeg lopova koji dobivaju veliku pažnju znanstvenika.

Završni rad „Algoritam računalne inteligencije za rješavanje problema putujućeg lopova“ dotaknuo se osnova računale inteligencije te osnova mravljeg algoritma. Kroz uvodna poglavlja, objašnjeno je postajanje NP-teških problema te da takvi problemi nisu rješivi standardnim matematičkim metodama u realnom vremenu pa se tu javljaju metaheuristički algoritmi koji se temelje na iskustvu, a imaju inženjerski pristup u smislu da nije potrebno naći najbolje rješenje nego samo ono pristojno dobro odnosno ono optimalno ili približno optimalno rješenje koje će biti prihvaćeno. Jedan od takvih algoritama je upravo mravlji algoritam koji smo osmislili za rješavanje promatranog problema te smo objasnili njegove najznačajnije karakteristike. Potom smo implementirali sam algoritam u jeziku C++, u okruženju Microsoft Visual Studio 2019. Algoritam smo implementirali pomoću funkcija te korištenjem *headera* kao pomoćnih datoteka. Testiranje algoritma smo izvršili nad problemima koji su u literaturi poznati pod nazivom: eli51-TTP, pr76-TTP, kroA100-TTP i a280-TTP. To su problemi koji su opisivali 51, 76, 100 i 280 gradova, respektivno.

Kroz različite eksperimente došli smo do zaključka kako korištenje lokalne optimizacije uvelike doprinosi poboljšanju algoritma na njegovom početku. Lokalna optimizacija poboljšava kvalitetu samog rješenja na početku algoritma, dok algoritam intenzivno uči, te omogućuje kasnijim virtualnim mravima konstrukciju boljih rješenja. Do tih zaključaka smo došli uspoređivanjem grafova rješenja sa i bez korištenja lokalne optimizacije. Nadalje, korištenje lokalne optimizacije utječe na brzinu izvođenja samog algoritma. Algoritam koji koristi lokalnu optimizaciju je znatno sporiji od onoga koji ju ne koristi. S obzirom na poboljšanu kvalitetu rješenja i tip problema kojega algoritam rješava, brzina izvođenja je u ovim slučajevima bila zanemariva. Većina eksperimenata prikazuje kako algoritam dobro uči s vremenom, no naišli smo i na rezultate gdje su povećani kapacitet ruksaka te povećana cijena iznajmljivanja predstavljali problem za sam algoritam te on nije bio u mogućnosti konstruirati rješenja koja će mu omogućiti učenje kroz određen broj iteracija. Došli smo do zaključka kako takvi problemi zahtijevaju drukčiji pristup prilikom same implementacije te da ova jednostavna implementacija mravljeg algoritma nije dovoljno razvijena da omogući značajan napredak samog algoritma u učenju.

Popis literature

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

[2] Matiyasevich Y. (1993) *Hilbert's tenth problem.* London: MIT Press Cambridge

[3] Tus, A. (2012). Heurističke metode lokalne pretrage primijenjene na problem izrade rasporeda sati za škole (Diplomski rad), Fakultet elektrotehnike i računarstva, Zagreb, Sveučilište u Zagrebu

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

[5] Heuristic (computer science), (2020). U Wikipedia online. Preuzeto s http://en.wikipedia.org/wiki/Heuristic_algorithm

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

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

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

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

[10] Wagner, M., Lindauer, M., Mısır, M., i sur. (2017). A case study of algorithm selection for the traveling thief problem. *J Heuristics* 24**,** Her Majesty the Queen in Right of Australia

[11] CEC competition at IEEE WCCI (2014). Optimisation of Problems with Multiple interdependent components, Preuzeto 25.08.2020. s https://cs.adelaide.edu.au/~optlog/CEC2014Comp/

[12] Ivković, Nikola; Golub, Marin. A New Ant Colony Optimization Algorithm : Three Bound Ant System // Lecture Notes in Computer Science, 8667 (2014), 280-281

[13] Ivković, Nikola. Modeliranje, analiza i poboljšanje algoritama optimizacije kolonijom mrava, 2014., doktorska disertacija, FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA, Zagreb

[14] 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

[15] 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)

Popis slika