SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 5394

# PRIMJENA OPTIMIZACIJSKIH ALGORITAMA NA PROBLEM USMJERAVANJA VOZILA

Matej Pipalović

Zagreb, lipanj 2018.

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 15. ožujka 2018.

# ZAVRŠNI ZADATAK br. 5394

Pristupnik: **Matej Pipalović (0036493046)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Primjena optimizacijskih algoritama na problem usmjeravanja vozila**

Opis zadatka:

Opisati problem usmjeravanja vozila i postojeće metode rješavanja toga problema. Definirati inačice problema s obzirom na zadana ograničenja i dati pregled uobičajenih pristupa rješavanju. Ostvariti programski sustav za rješavanje problema usmjeravanja vozila uz pomoć operatora lokalne pretrage i evolucijskih algoritama. Ispitati učinkovitost različitih operatora primijenjenih na navedeni problem. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 15. lipnja 2018.

Mentor:
Prof. dr. sc. Domagoj Jakobović

Predsjednik odbora za završni rad modula:
Prof. dr. sc. Siniša Srbljić

Dielovođa:
Doc. dr. sc. Tomislav Hrkać

# Sadržaj

# 1. Uvod

Problem raspoređivanja vozila (eng. “vehicle routing problem”, VRP) je jedan od najpoznatijih, najbitnijih i najviše proučavanih kombinatornih optimizacijskih problema. Generalno pitanje na koje VRP traži odgovor jest koje su to optimalne rute flote vozila, takve da posjećuju sve gradove, a pri tome zadovoljavaju određene kriterije.

Problem usmjeravanja vozila ima očite primjene u industriji. Programi kojima je svrha rješavati ovakve probleme mogu nerijetko značajno smanjiti troškove poslovanja pojedinih firmi. Dapače, transportni sektor čini 10% ukupnog BDP-a Europske unije, te je kao posljedica toga, svaka, pa tako i najmanja ušteda koja se može postići značajna.

Cilj problema jest minimizirati ukupnu cijenu rute, gdje cijena može biti ovisna o više parametara, poput duljine puta, vremenskog trajanja rute, mase vozila i sl. No najčešće je najvažniji kriterij minimizacija broja vozila. Određivanje optimalne rute svih vozila jest nepolinomijalno (NP) težak problem, a broj problema koji se može riješiti kombinatornom optimizacijom ili matematičkim pristupima je ograničen, te je iz tog razloga dosta često neophodno koristiti heurističke pristupe.

Varijanta problema koja je obrađena u ovom radu jest ona s ograničenim kapacitetom vozila (engl. Capacitated vehicle routing problem, CVRP). U svrhu rješavanja tog problema implementirani su neki od popularnih algoritama koji su se kroz povijest pokazali uspješnima za rješavanje ovog problema, a to su:

Genetski algoritam,
Tabu pretraživanje,
Algoritam simuliranog kaljenja,
Algoritam uštede.

Rad je podijeljen u šest poglavlja, prvo je uvodno. U drugo poglavlju rada je opisan problem koji se obrađuje, potom su u trećem poglavlju opisane metode računskoga rješavanja ovog i drugih problema, poput egzaktnih, heurističkih i metaheurističkih pristupa. U četvrtom poglavlju su detaljnije objašnjeni neki evolucijski algoritmi, koji spadaju u metaheurističke pristupe, te koji su obrađeni u ovom radu. Uz same algoritme opisani su i pripadni operatori, funkcije i ostale komponente algoritama. Peto poglavlje sadrži procese optimizacije parametara algoritama, kao i samo ispitivanje njihove funkcionalnosti, na skupu problema različitih složenosti. U šestom se nalaze neki zaključci koji su doneseni na temelju testiranja algoritama te istraživanja same problematike problema.

# 2. Problem usmjeravanja vozila (VRP)

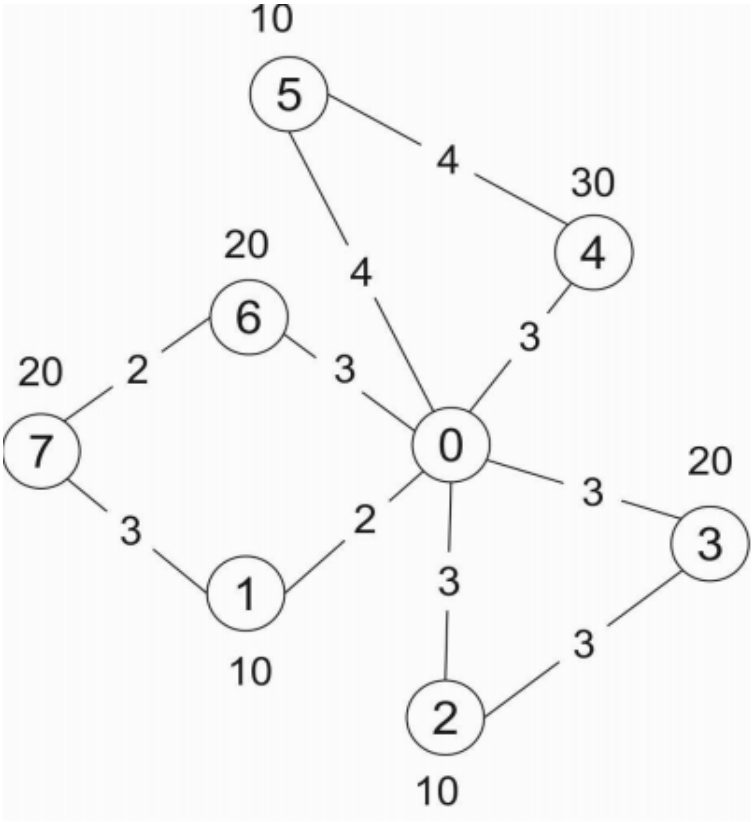
Problem usmjeravanja vozila se prvotno pojavljuje 1959. godine u radu George Dantzig i John Ramser[1] , a problem je predstavljen kao generalizacija problema trgovačkog putnika (travel salesman problem, TSP). Problem su opisali na stvarnom problemu – dostave nafte na benzinske postaje. U tu svrhu su predložili prvi matematički formuliran algoritam za rješavanje ovog problema.

1964. godine su Clarke and Wright unaprijedili algoritam Dantziga i Ramsera koristeći pohlepnu heuristiku, a algoritam je nazvan algoritam uštede (engl. the saving algorithm)[2]. Po uzoru na ova dva rada, ubrzo su napisane stotine modela i prijedloga kako pronaći optimalno, ili aproksimativno rješenje za razne inačice problema. Interes za ovim problemom je motiviran s korisnošću u stvarnim okolnostima, te samom složenošću problema. Najveće instance problema koje su konzistentno rješive najefikasnijim egzaktnim algoritmima su najčešće do 50 lokacija, dok se instance problema s više lokacija mogu riješiti optimalno samo u specifičnim slučajevima. Rješavanje za veće instance problema bi ili trajale predugo ili ne proizvode dovoljno dobra rješenja. Upravo zbog toga je potrebno primijeniti strategije rješavanja problema koje će naći dovoljno dobro rješenje u prihvatljivom vremenu.

Rad je fokusiran na varijantu problema s ograničenim kapacitetom vozila, a problem raspoređivanja vozila s ograničenim kapacitetom formalno je definiran kao usmjereni graf G, za koji vrijedi: $G=(V,H,c)$ , gdje je $V=\{0,1,2,...,n\}$ skup vrhova grafa, a $H=\{(i,j):i,j\in V,\ i\neq j\}$ skup bridova.
Vrh ***0*** predstavlja skladište flote od ***p*** vozila s jednakim kapacitetom od ***Q***, a ostalih ***n*** predstavlja gradove koje je potrebno posjetiti. Svaki grad $i\in V-\{0\}$ ima neku, pozitivnu potražnju $d_i\leq Q$ . Ne negativna cijena puta $c_{i,j}$ je definirana nad svakim lukom $(i,j)\in H$ . Cijena puta je predstavljena matricom cijene, koja je

simetrična, odnosno vrijedi $c_{i,j}=c_{j,i}$ za svaki $i,j \in V, i \neq j$ , te je zadovoljeno svojstvo trokuta: $c_{i,j} + c_{j,k} \geq c_{i,k}$ za $i,j,k \in V$. .

Slika 1 prikazuje primjer opisanog problema za 7 gradova i 3 vozila kapaciteta 50. Potražnja je definirana pored svake od lokacija.




*Slika 1: n = 7, Q = 50*

Specijalizirani oblik ovog problema je dobro poznati *problem trgovačkog putnika* (engl. *Travelling salesman person*, TSP), gdje umjesto flote vozila, imamo samo jedno vozilo (trgovca), koje posjećuje sve gradove, te nema ograničen kapacitet, niti ima neku drugu restrikciju.

# 3. Postupci rješavanja

Metode rješavanja optimizacijskih problema se najčešće dijele u tri skupine. To su egzaktne, heurističke i metaheurističke metode.
Egzaktne metode nisu uvijek pogodne, jer zbog činjenice da je značajan broj optimizacijskih problema po prirodi NP-složen, dimenzija problema mora biti vrlo niska kako kombinatorna složenost ne bi iscrpila računalne resurse.
Zbog toga je u svrhu rješavanja praktičnih problema gotovo neophodno koristiti heurističke i metaheurističke algoritme.
Heuristički i metaheuristički algoritmi ne garantiraju pronalazak optimalnog rješenja, no odlikuje ih mogućnost brzog pronalaska rješenja zadovoljavajuće kvalitete (približno optimalna rješenja).

## 3.1 Egzaktni algoritmi

Egzaktni algoritmi nisu uvijek sposobni pronaći rješenje u razumnom vremenu, pogotovo ako je broj gradova koje je potrebno posjetiti visok. No bez obzira na taj, dosta često dovoljan razlog da se ovi algoritmi ne koriste, važno je spomenuti neke od najpopularnijih, te su u nastavku navedena dva algoritma koja imaju najraširenuju primjenu na ovom problemu.

### 3.1.1. Branch and bound

Ovaj se algoritam koristi strategijom *podijeli i vladaj*, te prostor rješenja dijeli na više manjih podskupova koje zatim rješava zasebno. Taj se proces često vizualizira stablom. Algoritam pretražuje grane stabla, koje predstavljaju podskup skupa rješenja. Prije pretraživanja potencijalnih rješenja, grana se uspoređuje s donjom i gornjom granicom koje su prethodno definirane, te se, ako u trenutnoj grani nije moguće pronaći bolje rješenje od trenutnog najboljeg, proces ne nastavlja. Algoritam bitno ovisi o kvaliteti procjene gornje i donje granice optimalnog rješenja, inače se degenerira u iscrpnu pretragu.

### 3.1.2. Cut and branch

Ovaj algoritam uključuje korištenja Branch and bound algoritma, ali koristi mehanizam podrezivanja kojim smanjuje skup pretraživanja početnog algoritma. To podrezivanje se obavlja dodavanjem valjanih nejednakosti koje rješenja moraju zadovoljiti. Odnosno, pokušava predvidjeti koje će grane stabla tvoriti neispravno rješenje, te takve grane odbacuju.

## 3.2. Heuristički algoritmi

Heuristički algoritmi pretražuju ograničen prostor rješenja, te zbog toga rješenja pronalaze u razumnom vremenu i za probleme viših dimenzija. Odnosno, mijenjaju potpunost za brzinu, što je u praktičnim primjenama gotovo uvijek nužno. Ovi algoritmi u procesu konstrukcije rješenja donose odluke na temelju heurističke funkcije, koja rangira grane u prostoru pretraživanja, te na temelju toga odabiru granu, odnosno smjer u kojem se nastavlja pretraživanje. Algoritam uštede je jedan od takvih algoritama koji se ističe kvalitetom rješenja koja je sposoban pronaći na jednostavnijim problemima.

### 3.2.1. Algoritam uštede

Ovaj se algoritam temelji na postepenom stvaranju rute. Algoritam počinje s n nezavisnih ruta, gdje je n broj gradova koje je potrebno posjetiti, te se za svaku kombinaciju gradova računa ušteda, odnosno udaljenost za koju će se skratiti ukupna udaljenost pređena od strane flote vozila ukoliko se gradovi iz više ruta spoje u jednu. Uštede se sortiraju prema padajućem poretku, te se u svakoj iteraciji spajaju one rute s najvećom vrijednošću uštede.

Opisani proces može se formalno prikazati kao:

1. Stvoriti $n$ ruta: $v_0 \rightarrow v_i \rightarrow v_0$ , za svaki $i \geq 1$ , gdje $v_0$ predstavlja početnu lokaciju, a $v_i$ lokaciju koju je potrebno posjetiti.
2. Izračunati uštedu za svaku spojenu dostavu lokacija *i* i *j*, koja je definirana kao $s_{i,j} = d_{i,0} + d_{0,j} - d_{i,j}$ , za svaki $i, j \geq 1,\ i \neq j$
3. Poredati uštede po padajućem poretku

4. Počevši od vrha liste ušteda spoji one dvije rute koje će stvoriti najveću (preostalu) uštedu, pod uvjetom da je zadovoljeno:
    1. Dvije lokacije već nisu u istoj ruti
    2. Niti jedna od lokacija nije u unutrašnjosti rute, odnosno, neposredno je do početne lokacije
    3. Nisu narušena zadana ograničenja problema (kapacitet vozila)
5. Ponavljaj korake 3. i 4. sve dok je moguće ostvariti dodatne uštede

## 3.3. Metaheurističke metode

Za razliku od tradicionalnih pristupa, metaheuristički algoritmi pružaju novi pristup rješavanju kompleksnih problema, koji bi inače bili previše zahtjevni ili vremenski složeni za rješavanje. Za razliku od ostalih pristupa, ovi su algoritmi sposobni naći prihvatljivo rješenje u pristojnom vremenu, a to je rješenje dosta često značajno bolje od heurističkih metoda. Jednostavni heuristički konstrukcijski algoritmi su dosta često idealan izbor za početno rješenje metaheurističkih algoritama. Neki od algoritama koji su se pokazali izuzetno uspješnima za rješavanje ovog i sličnih NP složenih problema su najčešće prirodom inspirirani algoritmi.

U svrhu rješavanja VRP-a, u praksi su najčešće korišteni genetski algoritmi, algoritam simuliranog kaljenja, algoritam kolonije mrava, tabu pretraživanje, te drugi. Spomenuti algoritmi su se u praksi pokazali kao moćan alat za rješavanje složenih kombinatornih problema.

# 4. Rješavanje VRP-a evolucijskim algoritmima

U nastavku će biti opisano koji su sve evolucijski algoritmi, pripadni operatori, te ideje korištene za rješavanje problema usmjeravanja vozila s ograničenim kapacitetom (CVRP).

Problemski model koji se rješava za mrežu prometnica uzima potpun usmjeren graf definiran matricom cijene. Svaki grad (korisnik) također ima unaprijed definirani zahtjev za količinu robe koju je potrebno dostaviti. Vozila su jednaka, odnosno, kapacitet svakog vozila je identičan. Težine između gradova mogu biti definirane preko matrice udaljenosti ili euklidski.

Algoritmi koji su implementirani su sljedeći:

i. Genetski algoritam
ii. Simulirano kaljenje
iii. Tabu pretraživanje

Algoritmi nisu sami sebi dovoljni, već su nam potrebni pripadni operatori, o kojima će ovisiti i kvaliteta dobivenog rješenja.

Svako rješenje predstavlja skup ruta cijele flote vozila. To je jednostavno izraziti kao listu lista posjećenih gradova, Odnosno, ako listu posjećenih gradova gledamo kao vozilo, onda je jedno rješenje lista vozila. Osim ovog zapisa, možemo koristiti i onaj gdje su sva vozila reprezentirana jednom listom, tako da su međusobno odvojena s gradom koji predstavlja lokaciju skladišta.

## 4.1. Metodologija rješavanja

Prvi korak u procesu pronalaska rješenja metaheurističkim algoritmima je generiranje početnog rješenja pomoću konstrukcijskog algoritma. Zatim, iterativno poboljšavati svojstva jednog ili više početno generiranih rješenja. Ovaj proces,

iako dosta jednostavan, se pokazuje kao prilično uspješan. Iterativno poboljšavanje rješenja specifično je za svaki algoritam, no sama ideja je ista, želimo raditi malene, inkrementalne promjene, koje potencijalno vode u optimalno rješenje.

## 4.2. Početna rješenja

Početno rješenje, odnosno početna populacija predstavlja rješenje s kojim počinjemo proces evolucije. Što je to rješenje bolje, to je posao algoritma naizgled lakši. Ako algoritam radi sa slučajnim početnim rješenjem, onda je broj promjena koje je potrebno obaviti u procesu evolucije dosta veći nego kada rješenje već ima dosta dobrih svojstava. Stoga, kako bismo kao rezultat algoritma dobili bolje rješenje, ili ga dobili u manjem broju generacija, poželjno je imati što bolja početna rješenja.

Upravo se ovdje u priču ulaze heuristički algoritmi. Naime, kako heuristički algoritmi generiraju dobra rješenja u malo vremena, s dobrim svojstvima, ovi su algoritmi idealan način za dobivanje početnih rješenja.

## 4.3. Funkcija dobrote

Funkcija dobrote mjeri koliko dobro neka jedinka ispunjava traženu svrhu. Kako u dosta optimizacijskih problema ne postoji jedno najbolje rješenje, moguće je definirati različite funkcije dobrote koje će cijeniti neka svojstva rješenja više od drugih, te tako potaknuti istraživanje prostora rješenja u tom smjeru. Tako primjerice, kroz funkciju dobrote definiranu na odgovarajući način možemo odrediti kakvo rješenje će algoritam cijeniti najviše, što u konkretnom slučaju može biti rješenje s minimalnim brojem vozila, ili ono s najmanjom pređenom udaljenošću, neka kombinacija ili negativne vrijednosti spomenutih parametara. Ako bi funkcija dobrote – funkcija koju želimo maksimizirati, bila definirana kao prijeđena udaljenost tada bi optimalno rješenje bilo ono koje je sa stajališta problema najgore. Dakle vidimo da ukoliko želimo imati dobra svojstva rješenja, moramo definirati funkciju dobrote koja će ih naglasiti.

Ovisno kakvo rješenje tražimo, definirane su različite funkcije računanja dobrote. Važno je istaknuti da se kroz ovu funkciju zapravo mogu odrediti bitna svojstva konačnog rješenja.

## 4.4. Ograničenja

Ograničenja definiraju svojstva rješenja koja moraju biti zadovoljena da bi rješenje bilo prihvatljivo. Primjer jednostavnog ograničenja jest upravo kapacitet vozila. Ukoliko je vozilo popunjeno do kraja, tada bi posjećivanje dodatnog grada narušilo ograničenje te tako i samo rješenje više ne bi bilo u skupu prihvatljivih rješenja. Uz mehanizam ograničenja moguće je jednostavno implementirati dodatne zahtjeve nad rješenjima. No, ukoliko je definirano mnogo ograničenja, odnosno, ukoliko postoji puno zahtjeva nad rješenjem, tada računanje može trajati dugo, jer se dodavanjem ograničenja smanjuje prostor zadovoljivih rješenja, te je potraga za njima značajno teža. Problem zadovoljivosti se u takvim situacijama često rješava tako da se dozvoli da rješenje bude u nedozvoljenom prostoru, no ta se činjenica kažnjava dodatnom funkcijom koju uračunamo u funkciju dobrote u jednom od sljedeća dva oblika:

1. aditivni oblik: $f(x)=f(x)+p(x)$ , gdje je $p(x)=0$ ako su svi zahtjevi zadovoljeni, odnosno $p(x)>0$ inače
2. multiplikativni oblik: $f(x)=f(x)*p(x)$ , gdje je $p(x)=1$ ako su svi zahtjevi zadovoljeni, odnosno $p(x)>1$ inače

Zato što definirani problem nema mnogo čvrstih ograničenja, taj mehanizam nije bilo potrebno implementirati. Rješenja koja izlaze iz skupa zadovoljivih moguće je i ne stavljati u novu populaciju, odnosno ponoviti stvaranje novog rješenja.

## 4.5. Evolucijsko računanje

Evolucijsko računanje je područje umjetne inteligencije koje je često sposobno pronaći visoko optimizirana rješenja na velikom skupu problema. Zbog toga su ovakvi pristupi iznimno popularni u računarstvu. Karakteristično je po tome što je napredak iterativan, odnosno algoritmi se izvode u generacijama, što je samo jedan od koncepata preuzetih iz prirodnih procesa evolucije. Iako ne svi, velika

većina popularnih i uspješnih algoritama iz ove porodice je inspirirana prirodom, te pokušaji imitiranja i drugih koncepata nerijetko daju dobre rezultate. Primjerice, najpopularniji evolucijski algoritam, genetski algoritam, se temelji na principima evolucije u prirodi koje je utemeljio Charles Darwin, a njihova primjena u računarskoj znanosti počela ranih 1950-ih godina. Osim genetskog algoritma postoje mnogi drugi koji su također inspirirani prirodom i prirodnim procesima, poput algoritma simuliranog kaljenja, kolonije mrava, roja čestica, imunološkog algoritma i mnogih drugih. Evolucijski algoritmi se ponašaju dosta dobro u aproksimiranju optimalnih rješenja zato što ne donose pretpostavke o obliku funkcije dobrote, te modeliraju rješenje isključivo slučajnim promjenama. Ovaj pristup, iako dosta jednostavan, je prilično robustan, te pokazuje jako dobre rezultate i na prilično kompleksnim problemima, što nam govori da ne postoji nužna veza između kompleksnosti dobrog algoritma i samog problema koji rješavamo.

### 4.5.1. Genetski algoritam

Genetski algoritmi su temeljeni na konceptima koje susrećemo u prirodi, tj. prirodna selekcija i genetika. U iterativnom procesu iz prethodne generacije nastaju novi potomci primjenom tranzicijskih operacija. *Opstanak najjačih* je temeljni princip genetskih algoritama. Glavni koraci izvođenja algoritma uključuju: inicijalizaciju, evaluiranje funkcije dobrote, odabir jedinki za novu generaciju, križanje i mutaciju. Taj se iterativni proces ponavlja sve dok nije ispunjen neki od kriterija zaustavljanja. Kriterij zaustavljanja može biti ili dosezanje maksimalnog vremena računanja ili ne napredovanje najboljeg rješenja određeni broj iteracija.
Evaluacija funkcije dobrote nam omogućuje definiranje vrijednosti nad svakim rješenjem. Ono rješenje s najvećom funkcijom dobrote je najbolje rješenje u populaciji.

Operator selekcije odabire rješenje iz populacije koje će se koristiti u stvaranju novih jedinki (reprodukcija) na temelju njihovih dobrota. Reprodukcija se odvija u dva koraka, križanje i mutacija.

Križanje je proces gdje se iz dva odabrana rješenja stvara potomak, posljedica toga jest da dobiveni potomak zadržava neka svojstva oba svoja roditelja. Ideja je upravo da se na temelju starih rješenja generiraju nova, takva da posjeduju elemente rješenja koja su pokazala dobra svojstva. To nije uvijek slučaj, te zbog toga postoji selekcija. U svrhu križanja je moguće koristiti mnoge operatore, ili ih ne koristiti uopće ako je proces ujedno degenerativan.

Operator mutacije iz jednog rješenja stvara jedno novo rješenje. Mutacija za zadaću ima i traženje boljeg rješenja, te očuvanje genetske raznolikosti. Križanje i mutacija su generalno izvedeni tako da imaju neki parametar koji određuje koliko će sami operatori biti "agresivni", odnosno, kolike i kakve promijene bi oni trebali činiti.

Oba operatora igraju vrlo bitnu ulogu u genetskim algoritmima zbog razine uspješnosti rekombinacije već postojećih rješenja u nova. Ako neki od spomenutih operatora ne koristimo, tada se zapravo algoritam degenerira u jednostavniju evolucijsku strategiju.

### 4.5.2. Algoritam simuliranog kaljenja

Algoritam simuliranog kaljenja je metaheuristički algoritam koji je također inspiriran prirodnim procesima. U metalurgiji, kaljenjem se nastoji postići stanje minimalne unutarnje energije metala koje se dobije formiranjem pravilne kristalne rešetke. Metal se zagrijava na visoku temperaturu gdje se metal tali, nakon čega dolazi do hlađenja. Na visokim temperaturama prijelazi atoma u energetskim stanjima unutar kristalne rešetke se događaju stohastički. Što je temperatura metala veća, to je veće i gibanje atoma, pa tako su i sami prijelazi češći. Brzim spuštanjem temperature, atomi ne stignu formirati stabilne kristalne strukture. Svaka struktura kristalne rešetke predstavlja lokalni optimum, ali nije pošto tražimo globalni optimum, želimo imati kristalnu što stabilniju rešetku. Kako bi se postigao globalni optimum – pravilna struktura rešetke, koristi se sporo hlađenje (engl. Annealing) koje ostavlja dovoljno vremena atomima da formiraju takve strukture rešetke. Slična se ideja može primijeniti i na rješavanje optimizacijskih problema, tako što će temperatura biti parametar koji određuje koliko široki prostor rješenja se istražuje. Kada je temperatura veća, onda želimo imati veću pokretnost rješenja,

odnosno jedinka će se kretati kroz cjelokupni prostor rješenja, dok pri nižim temperaturama ona "istraživati" samo uži prostor, što predstavlja manjak kinetičke energije atoma. Pri većim temperaturama, to znači i veću vjerojatnost prihvaćanja lošijeg rješenja (onog s manjom dobrotom). Ukoliko novo rješenje ima veću dobrotu, tada ga prihvaćamo bezuvjetno. Kako se temperatura postepeno smanjuje, tako algoritam sve više nalikuje algoritmu gradijentnog spusta. Važno je spomenuti da se vjerojatnost prijelaza u fizici računa prema Boltzmanovoj distribuciji, a ona ovisi o trenutnoj temperaturi, razlici energetskih razina atoma, te Boltzmanovoj konstanti. U slučaju ovog algoritma, možemo zanemariti spomenutu konstantu, te prema sljedećem izrazu možemo računati vjerojatnost prijelaza u novo, "lošije" stanje: $p = e^{-\frac{\Delta E}{T_i}}$, gdje $\Delta E$ predstavlja razliku funkcije cilja. Implementacijski to znači da ćemo bacati težinski novčić (onaj koji nema jednaku vjerojatnost pojavljivanja obje strane), te ovisno o ishodu odabrati akciju.

Promjena temperature je ključna, jer želimo algoritmu dati dovoljno vremena da istražuje prostor rješenja, te da konvergira u lokalni minimum blizu kojeg se nalazi nakon hlađenja. Stoga je definirana funkcija temperature, koja novu temperaturu definira kao $T_{i+1} = \alpha T_i$, gdje je $\alpha$ konstanta koja je najčešće u intervalu [0.8, 0.99].

Za razliku od genetskog, ovaj algoritam ne radi sa populacijom rješenja, već samo sa jednim kojeg uzastopno, iterativno izmjenjuje.

### 4.5.3. Tabu pretraživanje

Tabu pretraživanje (engl. Tabu search) jest metaheuristički algoritam koji koristi metode lokalne pretrage za rješavanje optimizacijskih problema. Lokalne pretrage (pretrage susjedstva) za potencijalno rješenje traže susjedstvo u kojem se nadaju pronaći novo rješenje sa boljim karakteristikama. Susjedstvo rješenja je skup rješenja koja su slična početnom, no imaju lagane devijacije u nekim svojstvima. Zbog toga, algoritmi lokalne pretrage imaju tendenciju zapinjati u lokalnim optimumima. Za razliku od tradicionalnih algoritama lokalne pretrage, tabu pretraživanje postiže bolje rezultate tako što opušta glavno pravilo (postizanje napretka u svakom koraku), te dopušta prihvaćanje lošijeg rješenja ako ne postoji

bolje u susjedstvu. Dodatno, algoritam obeshrabruje posjećivanje već posjećenih stanja. Implementacija tabu pretraživanja sadrži strukturu podataka koja označuje koja su rješenja već posjećena, a označena su kao *tabu,* tako da algoritam ne razmatra ista rješenja uzastopno. Isto kao i algoritam simuliranog kaljenja, ovaj algoritam ne radi sa populacijom već radi sa susjedstvom samo jednog rješenja, koje se nanovo traži u svakoj iteraciji.

## 4.6. Operatori

Operatori su komponente evolucijskih algoritma koji obavljaju ključnu ulogu algoritma, stvaranje novog rješenja iz trenutnog/trenutnih. Bez njih, rješenja ne bi mogla napredovati tako da ih je važno dobro odabrati kako bismo dobro istraživali prostor rješenja. Različiti će operatori stvoriti različita nova rješenja, te tako utjecati na performansu evolucijskog algoritma. Zato što efikasnost operatora ne ovisi samo od problema do problema, već i o fazi u kojoj se algoritam nalazi, važno je također definirati i mehanizam pomoću kojega će se dinamički mijenjati svojstva operatora kako bismo maksimizirali njihovu efikasnost u svakom stadiju algoritma. To je posljedica toga što ćemo u početku izvođenja algoritma biti daleko od optimalnog rješenja, te će veće promjene rješenja, koje značajno mijenjaju njegova svojstva biti poželjne, no kako je algoritam bliže kraju izvođenja, tako su manje promjene poželjnije, jer bi velika promjena rješenja značajno degenerirala već stečena dobra svojstva rješenja. Taj se problem može dinamički riješiti preko operatora kompozita[3] koji sadrži više instanci operatora, te bilježi njihovu uspješnost u stvaranju novog, boljeg rješenja. Što neki od operatora uspješniji, to mu se dodjeljuje veća vjerojatnost odabira u sljedećem pozivu. No, pošto želimo dinamičko ponašanje, te ne želimo da operatori koji su bili uspješni na početku izvođenja budu čest izbor i pri kraju, potrebno je periodički resetirati statistike operatora. Ova je ideja primjenjiva na sve od operatora koji su korišteni u ovom radu (križanje, mutaciju, susjedstvo).

### 4.6.1. Križanje

Zadaća operatora križanja jest iz dvije jedinke stvoriti novu, takvo da nakon izvođenja jedinka ne krši ograničenja. Ukoliko ona ipak ne zadovoljava to svojstvo,

postupak križanja se pokušava ponovno. U nekim situacijama vjerojatnost da će dva roditelja stvoriti novu jedinku u dozvoljenom prostoru može biti vrlo niska. Iz tog se razloga definira maksimalni broj pokušaja križanja. Ukoliko nakon definiranog broja pokušaja operator ipak ne uspije stvoriti novu jedinku, tada dojavljuje da križanje nije uspjelo, što algoritam može prepoznati te primjerice pokušati stvoriti novu jedinku s drugim roditeljima. Moguće je također ne izvršavati funkciju ovog operatora, te kao rezultat križanja svjesno vratiti jednog od roditelja. Tako se preskače ovaj korak, koji ovisno o ostalim operatorima nije nužno niti potreban, iako je poželjan kako bismo imali utjecaj populacije na jedinke.

Neki od korištenih operatora križanja koji su se u ovom radu pokazali najboljima su *BestRouteCrossOver* i *OrderCrossOver.*

*BestRouteCrossOver* je operator koji od dva rješenja kombinira rute tako što stvori listu u kojoj se nalaze rute oba roditelja, te potom postepeno gradi novo rješenje pomoću ruta koje se nalaze u toj listi. Nakon svakog odabira rute iz liste, potrebno je izbaciti sve rute koje sadrže neki od gradova posjećenih u toj ruti. Prvi korak algoritma je gotov onda kada se lista isprazni. U tom trenutku, rješenje nije gotovo, jer neće svi gradovi biti posjećeni, zbog toga se stvaraju nove rute od preostalih gradova. Za to koristimo heuristiku koja za traženje što boljih ruta s preostalim gradova koristi ideju sličnu onoj iz algoritma uštede koju smo vidjeli u algoritmu uštede.

Demonstrirajmo taj operator na primjeru.

Imamo 2 roditelja:

1. prvi, s rutama: $1 \rightarrow 2 \rightarrow 3$ , $4 \rightarrow 5$ i $6 \rightarrow 7 \rightarrow 8$
2. drugi, s rutama: $2 \rightarrow 3$ , $4 \rightarrow 5 \rightarrow 6$ i $7 \rightarrow 8 \rightarrow 1$

kombiniramo sve rute od oba roditelja u zajedničku listu ruta : $1 \rightarrow 2 \rightarrow 3$ , $4 \rightarrow 5$ , $6 \rightarrow 7 \rightarrow 8$ , $2 \rightarrow 3$ , $4 \rightarrow 5 \rightarrow 6$ i $7 \rightarrow 8 \rightarrow 1$

Iz ove liste odaberemo rutu na slučajan način te ju dodajemo u novo rješenje, na primjer $4 \rightarrow 5$ . Kako postoji još jedna ruta s gradovima 4 i 5, izbacujemo ju iz liste. Ovaj proces ponavljamo sve dok lista nije prazna. Neka je sljedeća odabrana ruta $7 \rightarrow 8 \rightarrow 1$ . Dodajemo i nju u novo rješenje, te iz liste izbacujemo sve rute koje sadrže gradove 7, 8 i 1, nakon čega ostaje samo jedna ruta u listi dijeljenih ruta, a to je $2 \rightarrow 3$ te je ona odabrana u sljedećem koraku.

Rješenje koje imamo nakon ovog procesa je parcijalno, te se sastoji od ruta $2 \rightarrow 3$ , $4 \rightarrow 5$ , $7 \rightarrow 8 \rightarrow 1$ . Kao što vidimo, u rješenju nedostaje grad 6, a njega ubacujemo u jednu od 3 rute ako je to moguće (ima dovoljno mjesta u vozilu). Ruta, te položaj unutar rute se odabire tako da pređena udaljenost upravo te kombinacije (ruta, položaj unutar rute) bude najmanja.

Tako da na primjer za rješenje možemo kombinaciju sljedećih ruta: $2 \rightarrow 6 \rightarrow 3$ , $7 \rightarrow 8 \rightarrow 1$ , te $4 \rightarrow 5$ .

*OrderCrossOver operator* iz dva rješenja zapisana u obliku [1,2,3,d,4,5,6,d,7,8] zamijeni podskup između dva roditelja, nakon što se uklone skladišta vozila *d.* Ostatak gradova se popuni po redu u kojem se pojavljuju u drugom rješenju, te se radi rekonstrukcija rješenja, koja će dodavati gradove u rute sve dok su zadovoljena sva ograničenja.

Na primjer, ako imamo roditelje:

1. [1,2,3,4,5,6,7,8]
2. [1,7,8,2,6,4,3,5]

te odaberemo podskup [3,4,5] iz prvog roditelja, stvorit ćemo novo rješenje koje se trenutno sastoji od [_,_,3,4,5,_,_,_]. Na preostala mjesta ćemo ubaciti preostale gradove iz drugog rješenja po redu u kojem se pojavljuju(1, 7, 8, 2, 6).

Konačni redoslijed će biti [1,7,3,4,5,8,2,6]. Iz ovog oblika je još samo potrebno ponovno stvoriti rute, a one primjerice mogu biti [1,7,3], [4,5,8] i [2,6].

### 4.6.2. Mutacija

Mutacije su vrlo važan, ako ne i najvažniji element svakog evolucijskog algoritma, a predstavljaju slučajne promjene u genomu jedinke. Iako najčešće ne doprinose stvaranju boljeg rješenja izravno (jedinka često završi s lošijom dobrotom), doprinose raznovrsnosti genetskog materijala u populaciji, odnosno sprječavaju zasićenje populacije s genetski sličnim jedinkama.

Za rješavanje specifičnog problema (VRP-a), mutacije se mogu podijeliti u dvije skupine: one koje mutiraju samo odabranu rutu, te one koje kombiniraju više ruta.

Mutacije koje mutiraju samo jednu rutu se dosta efikasno mogu koristiti i u svrhu traženja lokalnog optimuma za trenutni raspored gradova po rutama, gdje je onda

lokalni optimum zapravo optimalna permutacija gradova u svakoj od ruta. Uzastopno ponavljanje mutacija nad pojedinačnim rutama rješenja zapravo usporedno rješava nekoliko TSP problema.

Mutacije koje mutiraju samo jednu rutu, koliko god uspješne bile, ne mogu postići dobrotu rješenja veću od one na koju su ograničene s početnim rasporedom gradova po rutama, te je svaki napredak minimalan. Zbog toga su dosta važne mutacije koje će mutirati više ruta istodobno i međusobno.

Te mutacije su zapravo neophodne ako za početno rješenje imamo neoptimalan raspored, te se pokazuje da su one zapravo teoretski same sebi dostatne za traženje optimalnog rješenja, no za to bi bio potreban veći broj iteracija algoritma nego kada se koriste zajedno u nekom obliku.

Neki od operatora mutacije korištenih u ovom radu su:

1. *Scramble Mutation* – nad skupom vozila (jednim ili više) permutira podskup uzastopno posjećenih gradova. Primjerice $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ bi se moglo mutirati u $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3$ , ako se mutira dio rute od trećeg do posljednjeg grada
2. *Inversion Mutation* – nad skupom vozila reverzno zamijeni redoslijed posjećivanja nekih gradova. Primjerice $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ bi se moglo mutirati u $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 6$
3. *Slide Mutation* – nad skupom vozila promijeni redoslijed posjećivanja gradova tako da se podskup gradova posjeti prije ili poslije jednog ili više gradova. Tako na primjer možemo iz $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ dobiti $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3$
4. *String Cross* – zamijeni dva luka između dvije rute. Na primjer, iz ruta: $1 \rightarrow 3 \rightarrow 5 \rightarrow 7$ i $2 \rightarrow 4 \rightarrow 6 \rightarrow 8$ bismo mogli dobiti nove rute $1 \rightarrow 3 \rightarrow 4 \rightarrow 8$ i $2 \rightarrow 4 \rightarrow 5 - 7$
5. *String Exchange* – isto kao i *String Cross,* samo sada umjesto jedne zamjene imamo njih *k. String Cross* je specifični oblik ovog operatora, gdje je $k = 1$
6. *String Relocation* – skup od najviše *k* gradova se prebacuje iz jedne rute u drugu. Parametar *k* je tipično 1 ili 2.

Mutacije koje mutiraju samo jedno vozilo se mogu modificirati tako da mutiraju više njih, no tada značajno smanjujemo izglednost generiranja ispravnog rješenja. Kako se ne bi trošilo procesorsko vrijeme na mutacije koje vrlo vjerojatno neće stvoriti ispravno rješenje, često su potrebne dodatne heuristike za popravak rješenja, koje će ga vratiti u dozvoljeno područje, ukoliko je to moguće.

### 4.6.3. Susjedstvo

Operator susjedstva je efektivno skup rješenja nastalih kao različite mutacije istog rješenja, te kao takvo nasljeđuje sva svojstva mutacije. Važno je samo nadodati kako si u algoritmima gdje koristimo susjedstvo možemo dozvoliti malo agresivnije mutacije, jer ćemo dobiti skup mutanata umjesto jednoga, no opet je poželjno dinamički odrediti operator susjedstva, koji u obzir uzima prethodnu uspješnost.

### 4.6.4. Selekcija

Selekcija je operator koji u populacijskim metaheurističkim algoritmima igra ključnu ulogu u stvaranju nove populacije. Naime, njegova zadaća jest odabrati rješenje koje će se kombinirati u novo rješenje, za novu populaciju. To je važno zato što želimo da se rješenja s boljim značajkama više razmnožavaju, s obzirom da je veća vjerojatnost da će onda i njihov rezultat biti bolji. No, kada bismo kombinirali samo najbolja rješenja, vrlo brzo bismo izgubili genetsku raznolikost, te bi križanje bilo ne efikasno. Stoga je važno imati dobar operator selekcije, koji će nam dati što je bolju kombinaciju te dvije krajnosti.

Najpopularniji operatori selekcije su turnirska selekcija (engl. tournament selection) i proporcionalna selekcija (engl. *RouletteWheel*). Turnirska selekcija u obzir uzima nekoliko jedinki iz populacije, te odabire ono s najvećom dobrotom. S druge strane, *RoulleteWheel* radi tako da svakom rješenju populacije dodijeli vjerojatnost odabira proporcionalnu njihovoj dobroti, te slučajno odabere neko rješenje.

## 4.7. Funkcija lokalnog optimuma

Ponekad se može dogoditi, da u procesu traženja optimalnog rješenja odbacimo rješenje s dobrim svojstvima i velikim potencijalom, odnosno da ga vrednujemo

manje od nekog rješenja koje je u lokalnom optimumu, iako takvo rješenje ne može jednostavno napredovati. Stoga, želimo imati mehanizam koji će uz pomoć minimalnih promjena pokušati ostvariti dio potencijala kojeg imaju, te povećati vlastitu šansu za opstankom u populaciji, ili jednostavno bržim napretkom u željenom smjeru.

Pokazuje se da je ovakvu funkcionalnost jednostavno implementirati, a moguće ju je ostvariti na mnogo načina. Jedan od najjednostavnijih je jednostavno pokrenuti neki optimizacijski algoritam, ali s malim brojem iteracija, kako velike promjene nisu potrebne. Takvi algoritmi iz tog razloga koriste operatore koji rade male promjene nad jedinkom, a to najčešće uključuje uglavnom mutacije koje rade nad samo jednom rutom, ili više njih neovisno. Moguće je također implementirati neki heuristički algoritam pronalaska lokalnog optimuma, koji će u obzir uzeti i konkretni problem koji rješavamo. Konkretno, to bi značilo da bi takav algoritam pokušao naći optimalnu permutaciju gradova unutar rute, za razliku od slučajnih promjena.

# 5. Rezultati

Eksperimentiranjem se nastojalo ustanoviti hoće li, odnosno koliki će utjecaj promjene parametara algoritama imati na kvalitetu konačnog rezultata, te kako je moguće ostvariti optimalne rezultate. Kako većina algoritama ovisi o mnogo parametara koji su dosta često i međuovisni, testiranje se ne provodi sa svim podskupovima parametara. Stoga će se pokušati pronaći optimalni parametri tako da se promjena izolira na samo jedan od parametara, te tako smanji skup parametara koji testiramo, no pri promjeni jednog od parametara može se smanjiti dobar utjecaj dobro odabranog drugog parametra, zbog međusobne ovisnosti tih parametara.

## 5.1. Ulazni podaci

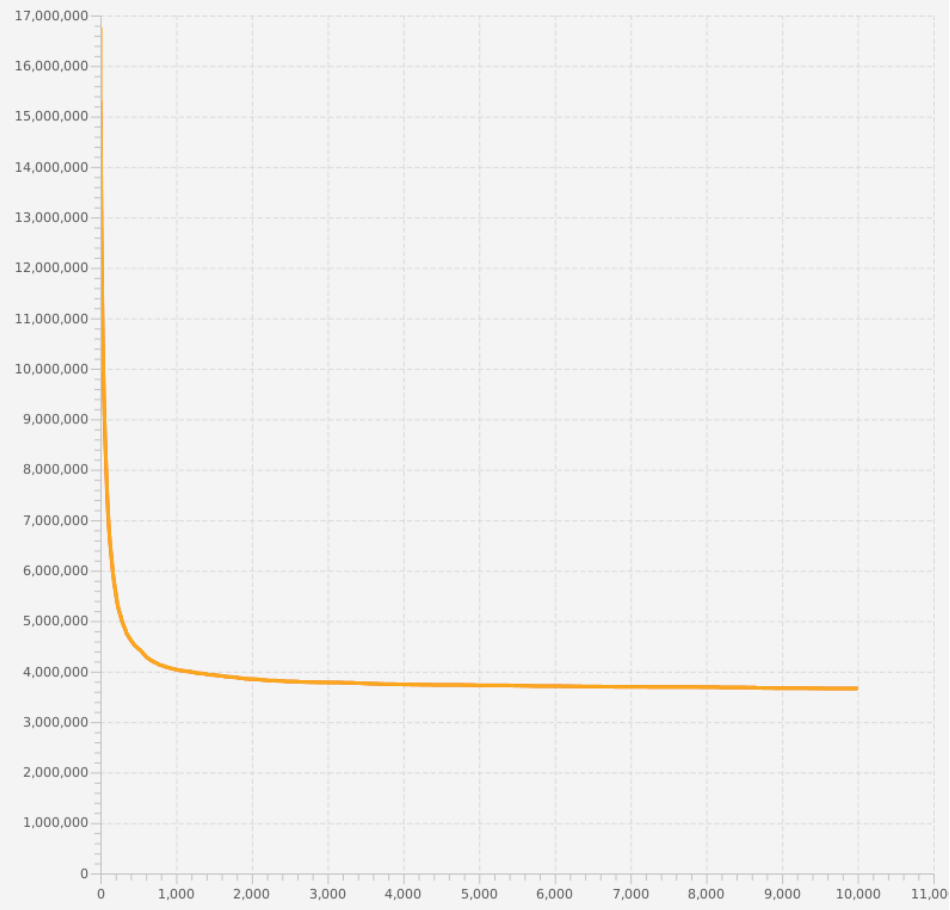
Ulazni podaci predstavljaju matricu udaljenosti između varijabilnog broja gradova. Kako veličina tog skupa gradova određuje složenost cijelog problema, tako i utjecaj operatora može biti veći ili manji. Primjerice, ako imamo skup s manje od 50 gradova, broj iteracija potrebnih da se dođe u lokalni optimum jest značajno manji nego li je za problem s 200 gradova. Zbog toga je važno napomenuti da se svi rezultati za implementirani sustav temelje na skupu od 216 gradova, od kojih jedan predstavlja skladište, te je potrebno posjetiti ostalih 215. (povremeno se referenciraju manji problemi zbog dostupnosti koordinata gradova te mogućnosti vizualne reprezentacije rješenja.)

## 5.2. Uvjet zaustavljanja

Kako bi se algoritam ubrzao, odnosno kako bi se smanjio broj iteracija algoritama koje je potrebno izvoditi kako bismo dobili prihvatljive rezultate, napravljen je ovaj test gdje je nad većim brojem pokretanja algoritma s velikim brojem iteracija mjeren napredak. Kada taj napredak (promjena funkcije dobrote) postane mali, onda se algoritam može prekinuti. Naravno, kada bi se algoritam izvodio više od odabranog broja generacija, tada bi rezultati bili bolji, no za potrebe testiranja

operatora nije nužno potrebno imati optimalne rezultate, već je važno uočiti trend funkcije dobrote. Ovako se također može bolje procijeniti efikasnost određenih operatora, jer bi kroz više vremena nego je objektivno potrebno i oni lošiji operatori (za specifični problem) postigli bolje rezultate.

Na slici 2 se vide rezultati jednog takvog ispitivanja izvršenog za genetski algoritam. Rezultati tu dobiveni kao prosječna vrijednost funkcije dobrote u 15 pokretanja algoritma GA. Pokretanja su izvršena nad populacijom od 100 jedinki koja se evoluirala kroz 10000 iteracija. Početno rješenje je slučajno generirano.




*Slika 2: Promjena udaljenosti sa brojem generacija algoritma[GA]*

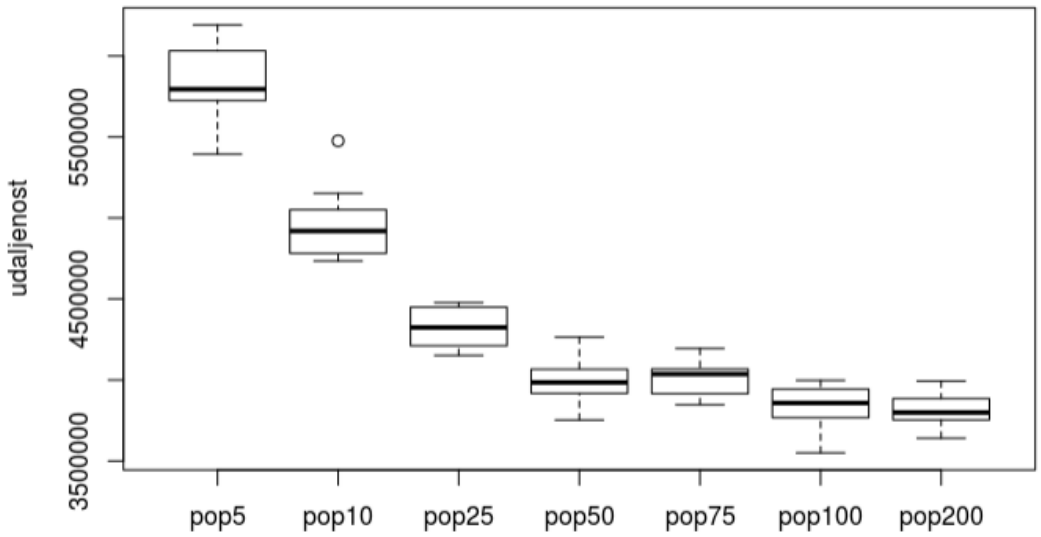
Na slici se može uočiti, da, iako postoji napredak tijekom cijelog izvođenja algoritma, on nakon primjerice 2000. iteracije ne daje rješenje s toliko manjom

funkcijom dobrote, odnosno ta razlika je oko 5%. Stoga je upravo taj broj iteracija odabran za generiranje ostalih brojeva. Spomenuo bi kako je za alternativni testni primjer od 101 grada, broj generacija potreban za konzistentno postizanje prihvatljivog rješenja oko 400. Dakle, primjetan je nelinearni porast složenosti.
Isti proces je proveden i za tabu pretraživanje, te možemo zaključiti da je kod tabu algoritma 2000 generacija također granica iza koje povećanje dobrote nije značajno.

## 5.3. Veličina populacije (GA)

Veličina populacije je kod populacijskih evolucijskih algoritama jedan od najznačajnijih parametara. Veličinu populacije možemo podijeliti u manje i veće, a svaka ima svoje prednosti. Algoritmi s manjim populacijama se brže izvode, jer nije potrebno generirati veliki broj novih rješenja, a upravo to generiranje je dosta često najskuplji dio algoritma. Algoritmi s većim populacijama čuvaju raznolikost jedinki, odnosno raspršenost rješenja u skupu pretraživanja zbog toga što se teže odbacuju lošije jedinke.

Rezultat ovog ispitivanja je prikazan u *boxplotu* na slici 3. Kao i u prethodnom testiranju, za generiranje ove slike bilo je potrebno više puta pokrenuti algoritam s različitim parametrom veličine populacije. Dobiveni grafovi predstavljaju distribucije najboljih rješenja dobivenih pokretanjem genetskog algoritma 10 puta. Korištena je turnirska selekcija veličine 0.2 * (veličina populacije).



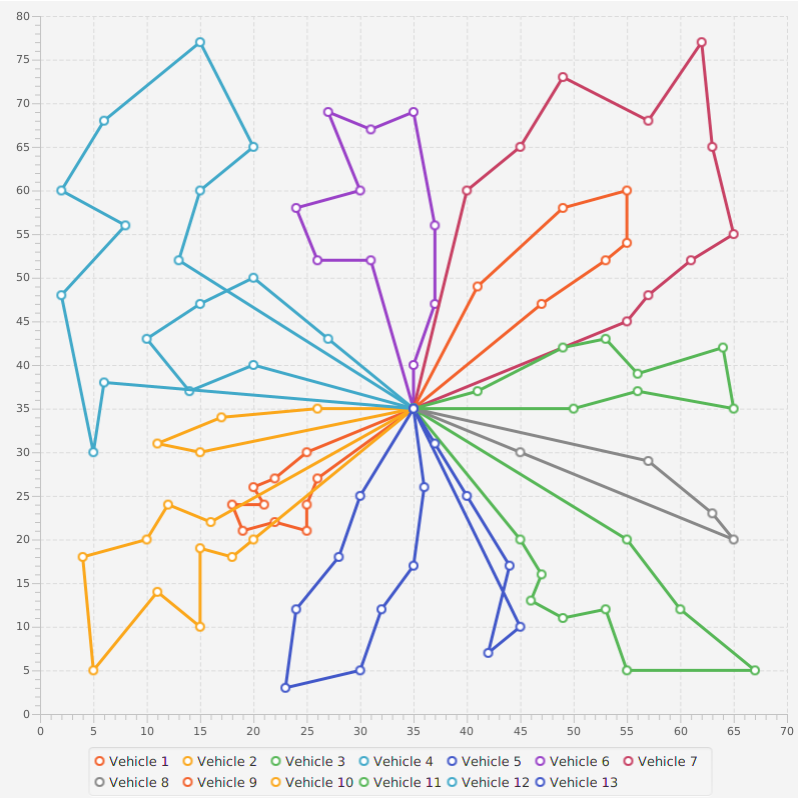

*Slika 3: distribucija najboljih rješenja iz 10 pokretanja ovisno o veličini populacije[GA]*

Na slici možemo vidjeti kako za slučaj malih populacija rješenja koja dobijemo zaostaju za onima s većim brojem iteracija, te kako zapravo sa povećanjem populacije taj skup rješenja konvergira oko jedne funkcije dobrote. Na temelju generiranih *boxplotova* možemo zaključiti kako nema smisla povećavati populaciju iznad 100, odnosno, da iako veće populacije generiraju bolja rješenja, ona nisu značajno bolja, te pošto veća populacija povlači veću složenost, pa tako i vrijeme potrebno za izvođenje algoritma, veličine populacije koje su vrijedne razmatranja su 50 i 100.

## 5.4. Početno rješenje

Početno rješenje, očekivano, igra dosta bitnu ulogu. Što je početno rješenje bolje dobrote, to algoritam ima više vremena da ga evoluira, te tako postiže bolje konačno rješenje. Osim toga, heuristički generirana početna rješenja mogu imati dobra svojstva, koja mogu dijeliti neke komponente onih optimalnih. Primjerice na slici 4, možemo vidjeti početno rješenje generirano pomoću algoritma uštede

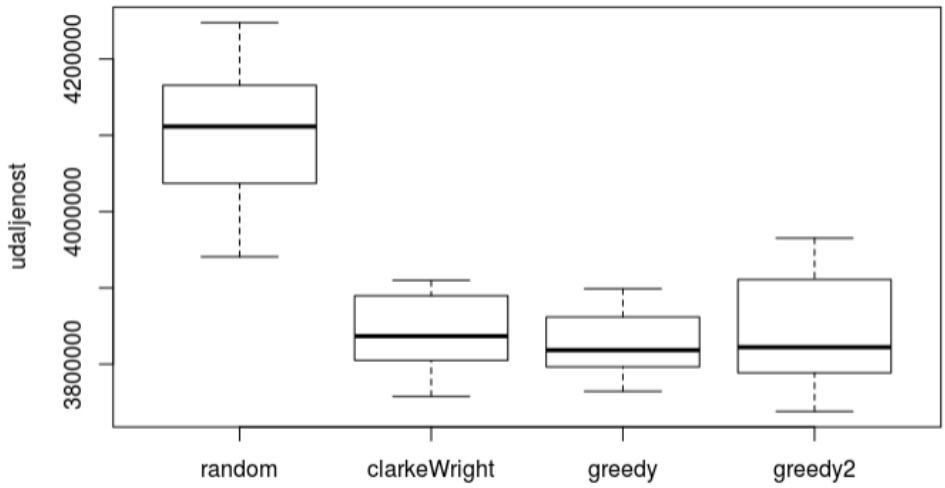
(Clarke & Wright). Dosta često, na jednostavnijim primjerima, ovaj heuristički algoritam je sposoban generirati i optimalno rješenje, no u ovom primjeru koristi veći nego potreban broj vozila, te bi se neke od ruta mogle dodatno optimirati.




*Slika 4: Početno rješenje generirano koristeći algoritam uštede*

No, pošto ovakva rješenja mogu biti gotovo optimalna, te ne bi imalo smisla razmatrati napredak algoritma počevši sa spomenutim rješenjem, korišteno je slučajno generirano početno rješenje.

Na *boxplotu* na slici 5 se nalazi usporedba dobrota konačnih rješenja u ovisnosti o načinu na koji generiramo početno rješenje.




*Slika 5: distribucija konačnih rješenja u ovisnosti o načinu generiranja početnog rješenja*

Osim savings algoritma u obzir su uzeta i 2 algoritma pohlepnog generiranja rješenja. Prvi ("greedy") gradi rutu trenutnog vozila tako da posjeti najbliži grad ako je moguće, tj. nisu prekršena ograničenja, te kada više ne može posjetiti grad, nastavi postupak s novim vozilom. Drugi ("greedy2") posjećuje slučajni grad od pet najbližih, po istom principu. Iako su početne dobrote pohlepnih algoritama manje od algoritma uštede, zbog ne tako loših početnih rješenja, dobar su izbor za generiranje početnog rješenja.

## 5.5. Operatori

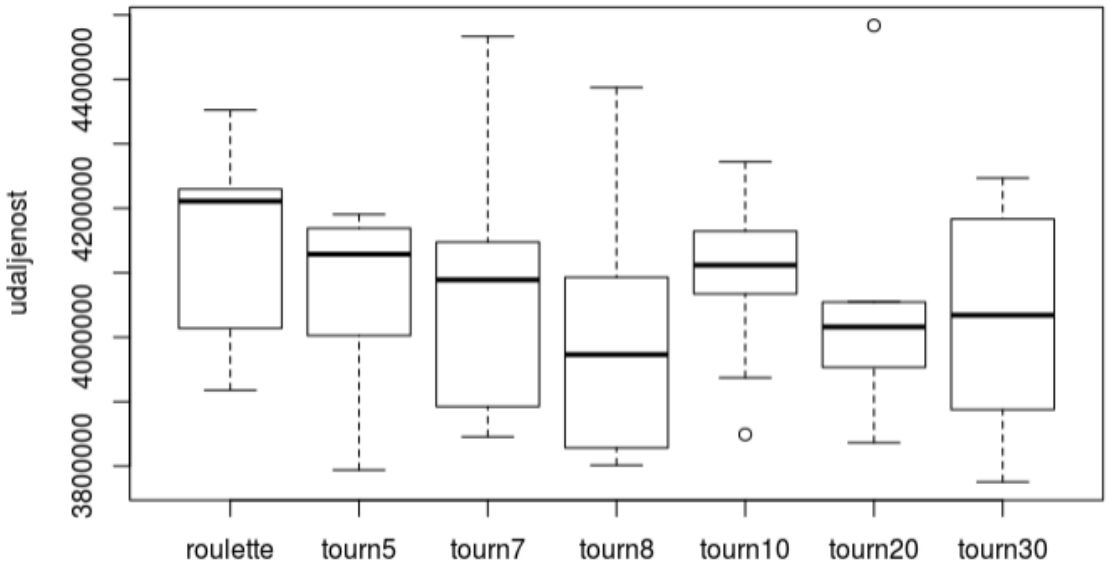
Kako su operatori zapravo dio algoritma koji zapravo evoluira rješenje – nabolje ili nagore, oni su najbitniji dio samog procesa evolucije. Stoga ih je važno optimirati, kako bi se njihov utjecaj što bolje osjetio i na krajnjim rezultatima.

### 5.5.1. Selekcija (GA)

Operator selekcije, odnosno operator zadužen za odabir roditelja u procesu stvaranja nove populacije genetskog algoritma je dosta važan operator. Njegova uloga je odabrati genetski materijal koji se križa, mutira, te prenosi u sljedeću generaciju. Najjednostavniji operator koji možemo zamisliti, jest da odaberemo

najbolje rješenje, no na taj bi način poništili sve pozitivne efekte populacije, odnosno degenerirali algoritam tako da efektivno ne bude populacijski, već nalik Tabu algoritmu, koji radi na sličnom principu. Stoga su korišteni operatori selekcije koji zadržavaju dobra svojstva algoritma, odnosno *RouletteWheelSelection* i turnirska selekcija (*TournamentSelection*). Pokazalo se da je za veličinu populacije od 50 najbolji operator selekcije turnirska selekcija, veličine 8.

---

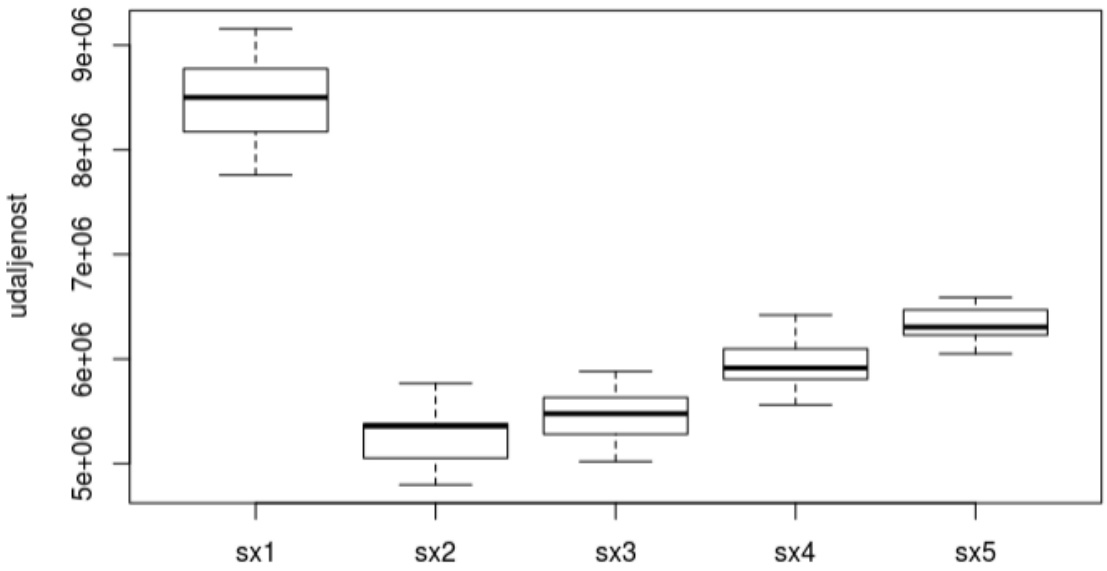



### 5.5.2. Mutacija

Od svih implementiranih algoritama ovaj su operator koristili svi algoritmi, te se pokazalo da se utjecaji pojedinih mutacija podjednako odražavaju na konačni rezultat neovisno o korištenom algoritmu. Rezultati su stoga generirani s genetskim algoritmom, pri tome ne koristeći operator križanja.

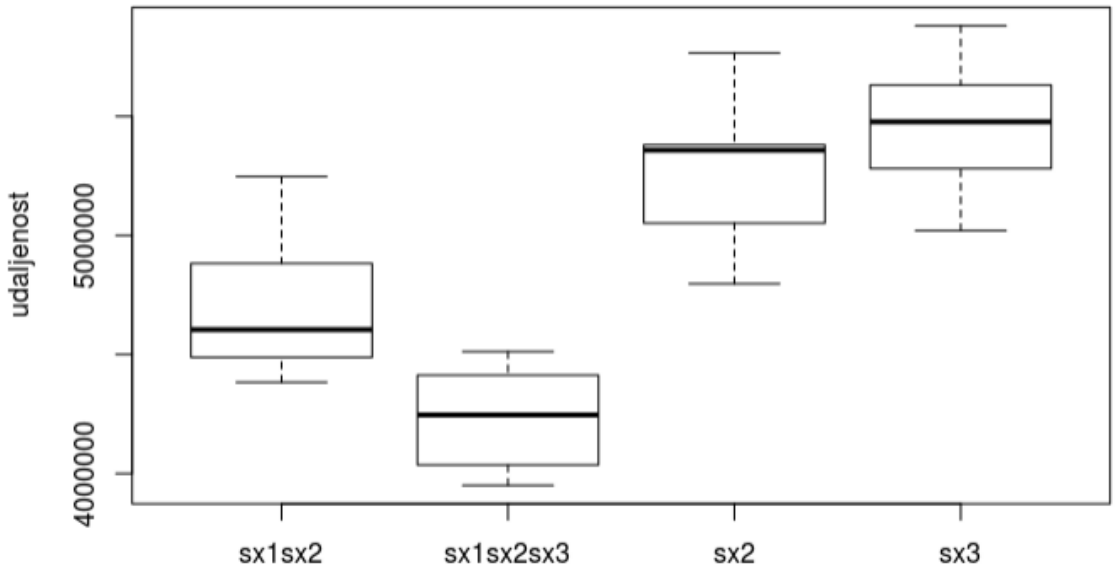
Nisu sve mutacije neovisne, te postoje one kojima je cilj primjerice samo permutacija gradova unutar jedne rute. Takve mutacije bi ovisile gotovo samo o početnom rješenju, stoga ih nema smisla razmatrati samostalno.

Od "nezavisnih" mutacija se *StringExchange* pokazala kao najbolja, a rezultati ovisno o parametru *k* se nalaze sa slici 6.




*Slika 6: ovisnost dobrote rješenja o parametru k operatora StringExchange*

Kao što vidimo, algoritam, iako jest samostalan, postiže rješenja koja nisu tako blizu optimalnih, a to vidimo tako što kada kombiniramo više operatora mutacije dobijamo bolje rezultate. Tako primjerice, na slici 7 imamo prethodnu usporedbu ali sa dodatkom nekoliko kombinacija mutacija. Kao što vidimo, iako *StringExchange* pokazuje bolje rezultate od ostalih mutacija kad je samostalan, ti rezultati mogu biti dosta bolji koristeći one operatore koji nisu samostalni, ili čak i jesu ali ne daju tako dobre rezultate kada ih ispitujemo samostalno.




*Slika 7: utjecaj kombiniranja operatora na distribuciju konačnih rješenja*

### 5.5.3. Operator hlađenja (SA)

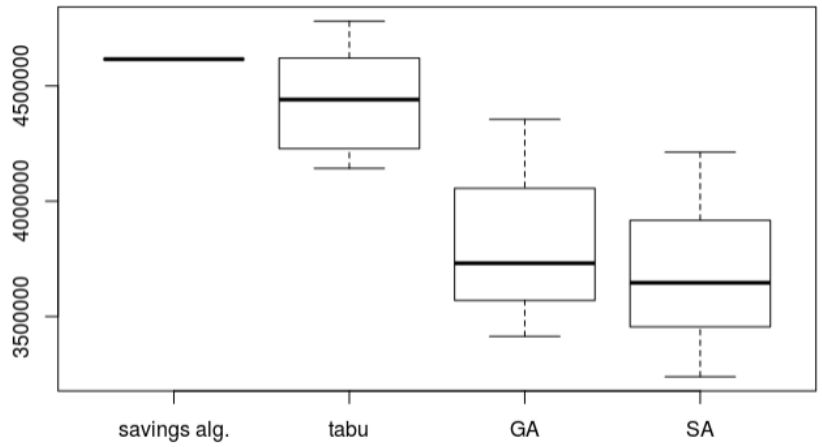
Ovaj operator efektivno određuje kada će se i koliko dugo algoritam simuliranog kaljenja fokusirati na traženje lokalnog optimuma, odnosno istraživanja prostora rješenja. Kako bismo algoritmu ostavili dovoljno vremena da konvergira u lokalni optimum, ali mu isto tako dozvolili da pretraži što veći prostor rješenja, potrebno je pronaći što bolju strategiju hlađenja. Korištena je geometrijski operator hlađenja, sa 4 parametra. Početnom temperaturom, koeficijentom $\alpha$ koji određuje koeficijent s kojim će se trenutna temperatura pomnožiti da bi se dobila ona za novu iteraciju. Broj iteracija vanjske, te unutarnje petlje. Kako je riječ o mnogo parametara, očito je da optimalni skup ne postoji, stoga je potrebno analizirati različite parametre.

Kroz testiranja je pokazano da je uz početnu temperaturu od 1000, parametar $\alpha$ od 0.99, te 100 unutarnjih i 500 vanjskih iteracija postižu dobri rezultati. Broj iteracija se pokazao kao bitniji parametar od same temperature i parametra $\alpha$, što je moguće lako objasniti kroz činjenicu da neke od mutacija sadrže određene heuristike za popravljanje rješenja, te se zbog toga dobroto rješenja neće

značajno narušiti, pa je samim time i vjerojatnost njegova prihvaćanja veća, čak i uz nižu temperaturu.

## 5.6. Algoritmi

Konačno, kada imamo parametre svakog operatora, odnosno algoritma koji generiraju najbolja rješenja, možemo usporediti algoritme na tom testnom primjeru. Nakon optimizacije parametara na glavnom skupu, od 216 gradova, algoritam simuliranog kaljenja se pokazao najboljim, što se može vidjeti na *boxplotu* na slici 8.




*Slika 8: usporedba algoritama*

No, kako kvaliteta rješenja koja je dao algoritam ovisi o mnogo značajki, moguće je da je bitnu ulogu igrao upravo skup na kojem je provedeno testiranje. Stoga su isti testovi provedeni na još nekoliko ulaznih skupova različitih dimenzija, a rezultati su dani u sljedećoj tablici.

| Skup | SA | GA | Tabu | Savings alg. |
|---|---|---|---|---|
| n25-k5 | 222.998 | 222.998 | 222.998 | 229.041 |
| n80-k10 | 789.845 | 781.225 | 789.434 | 1402.178 |
| n100-k10 | 823.849 | 825.779 | 851.701 | 974.086 |
| n194-k10 | 45901.38 | 45628.4 | 46458.323 | 47895.338 |
| n216-k10 | 3238762 | 3413809 | 4142221 | 4615867 |

Iz tablice možemo vidjeti kako algoritam simuliranog kaljenja nije konzistentno bolji od genetskog algoritma na svim testnim skupovima. Iako je ovaj ispitni skup problema dosta malen za donošenje zaključaka, možemo vidjeti da podupire „*no free lunch theorem*”[4], koji kaže da kada uspoređujemo dva algoritma na velikom skupu problema (ne samo VRP), onda ne postoji bolji od ta dva algoritma. Osim samog problema, veliki utjecaj na uspješnost ima i činjenica koji testni skup pokušavamo riješiti, što upravo vidimo u ovom eksperimentu.

# 6. Zaključak

U ovom radu istražen je problem usmjeravanja vozila s ograničenim kapacitetom. Njihova je zadaća dostaviti robu sa svih lokacija, te taj problem pokušati riješiti što je manjom flotom moguće, kojoj je cilj prijeći što manju ukupnu udaljenost. S ovim se optimizacijskim problemom susreće velik dio poduzeća u industriji, te je rješavanjem ovog problema moguće značajno uštedjeti na transportnim troškovima. Taj problem spada u kategoriju NP teških problema.

S obzirom na kompleksnost problema, te nepolinomijalni porast složenosti s povećanjem broja gradova, egzaktni algoritmi u praksi nisu u mogućnosti dati zadovoljavajuće karakteristike, odnosno, taj je pristup dosta često spor. Zbog toga, umjesto njih želimo koristiti algoritme koji će ubrzati proces računanja, iako oni ne garantiraju pronalazak optimalnog rješenja.

U radu su istraženi metaheuristički postupci koji se često koriste u praktičnim primjenama. Kako bi ti postupci bili uspješni, važno je držati dobar omjer lokalnog pretraživanja te očuvanja genetske raznolikosti rješenja, a u tu je svrhu prihvatljivo ponekad prihvatiti i lošije rješenje od trenutnog.

Istražena su dva operatora selekcije, turnirska i proporcionalna selekcija. Pokazalo se da operator selekcije nije bitan za kvalitetu populacije, te je iz tog razloga korišten manje računski zahtjevni operator turnirske selekcije.

Za križanje je korišteno nekoliko operatora, od kojih su se dva istaknula kao najbolja, no i bez njih je mutacija bila dovoljno robusna, te rješenje dobiveno bez križanja ne odstupa značajno od onoga dobivenog uz križanje.

Za operator mutacije je korišteno nekoliko različitih operatora, koji su bili podijeljeni u dvije skupine: na one koji mijenjaju pojedine rute, ili ih kombiniraju. Korištena je njihova kombinacija na dinamičan način, kako bi se operatori primijenili pravovremeno i ne isključivo.

Kako bismo dobili najbolje rezultate s određenim algoritmom, potrebno je njegove parametre prilagoditi specifičnom problemu koji se rješava. To može biti dosta složen problem, jer osim poznavanja samih algoritama i operatora, često zahtjeva poznavanje same problematike. Problem optimizacije parametara je u radu riješen

tako što se promjena vrijednosti parametara izolirala na jednu varijablu kako bi se utjecaj promjene te vrijednosti najbolje preslikao na kvalitetu rješenja. Potom smo isti postupak primijenili i na ostale parametre algoritama.

Provedena su istraživanja kojima je bio cilj utvrditi dali je neki od algoritama bolji od drugih. Pokazalo se da niti jedan od algoritama nije konzistentno bolji, te da kvaliteta rješenja može ovisiti i o samom ulaznom skupu. Stoga, na odgovor koji je algoritam najbolji za rješavanje problema usmjeravanja vozila ne možemo dati zaključni odgovor, već samo utvrditi da svaki od algoritama (operatora) ima svoje prednosti.

# Literatura:


[1] Dantzig, G. B., and J. H. Ramser. “The Truck Dispatching Problem.”Management Science, vol. 6, no. 1, 1959, pp. 80–91.

[2] Clarke, G.U., Wright, J.W., 1964. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research 12, 4, 568–581.

[3] Hong, TP., Wang, HS. & Chen, *Simultaneously Applying Multiple Mutation Operators in Genetic Algorithms,* WC. Journal of Heuristics (2000) 6: 439.

[4] Wolpert, D.H., Macready, W.G. (1997), "No Free Lunch Theorems for Optimization”

[5] Kurasova O. *Genetic Algorithm for VRP with Constraints Based on Feasible Insertion*, INFORMATICA, 2014, vol. 25, No. 1

[6] Korablev V., Makeev I., Kharitonov E., Tshukin B. Romanov I., *Approaches to solve the vehicle routing problem in the valuables delivery domain*, Procedia Computer Science, vol. 88, 2016

[7] Hertz A., Taillard E., Werra De.: A TUTORIAL ON TABU SEARCH

[8] C. Snoeys, *Comparison of Meta-heuristic Approaches for Large-scale Vehicle Routing Problems,* GHENT UNIVERSITY, 2013

[9] Puljić K., Manger R. *Comparison of eight evolutionary crossover operators for the vehicle routing problem*, Math. Commun. 18(2013), 359-375

# PRIMJENA OPTIMIZACIJSKIH ALGORITAMA NA PROBLEM USMJERAVANJA VOZILA

**SAŽETAK:**

Ovaj rad obrađuje problem usmjeravanja vozila, te je fokusiran na njegovo rješavanje pomoću metaheurističkih algoritama. Implementirani su i uspoređeni sljedeći algoritmi: genetski algoritam, simulirano kaljenje, tabu pretraživanje, te algoritam uštede. Pokušala se istražiti ovisnost parametara algoritama i operatora, te njihov utjecaj na kvalitetu rješenja.

**KLJUČNE RIJEČI:**

vrp, heuristika, metaheuristika, genetski algoritam, simulirano kaljenje, tabu pretraživanje, optimizacija, evolucijsko računanje

# APPLICATION OF OPTIMISATION ALGORITHMS ON VEHICLE ROUTING PROBLEM

**ABSTRACT:**

This paper deals with vehicle routing problem and is focused on it's solving by using metaheuristic approaches. The following algorithms have been implemented and compared: genetic algorithm, simulated annealing, tabu search and savings algorithm. It tried to explore the dependence of algorithms, operator parameters and their impact on quality of the solution.

**KEY WORDS:**

vrp, heuristics, metaheuristics, genetic algorithm, simulated annealing, tabu search, optimization, evolutionary computation