SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 213

Rješavanje problema trgovačkog putnika uz pomoć genetskih algoritama

Marko Pielić

Zagreb, lipanj 2008.

Sadržaj

1. Uvod

Problem trgovačkog putnika jedan je od najpoznatijih problema algoritamske teorije brojeva. Trgovački putnik ima definirane gradove koje mora posjetiti. Svaki grad smije posjetiti samo jednom, a na kraju se mora vratiti u početni grad. Redoslijed kojim obilazi gradove mora biti takav da ukupni put bude minimalan. Ovaj problem spada u kategoriju NP-teških problema i ima faktorijelnu složenost, pa predstavlja pravi izazov i za ljude i za računala. Zbog njegove složenosti tradicionalne metode rješavanja nisu se pokazale pretjerano uspješnima.

Tu dolaze na red genetski algoritmi, stohastičke metode pretraživanja koje oponašaju prirodni tijek biološke evolucije. Prednost genetskih algoritama prilikom rješavanja ovog i sličnih problema je da ne započinju pretragu od jedne točke, nego od niza točaka koje predstavljaju početnu populaciju ([9]). Populacija se iterira kroz generacije te nastaju sve bolja rješenja. Genetski algoritam se zaustavlja kada nađe rješenje koje zadovoljava kriterij optimizacije, a to rješenje ne mora nužno biti optimalno.

Problem trgovačkog putnika vrlo je važan jer se na njega svode mnogi drugi problemi, uključujući problem postavljanja sklopovskih pločica (u kojem treba osigurati konstantne intervale između slanja signala), proces dobivanja sirovina (npr. papira iz drva), grupiranje podataka iz velikih polja, analiziranje kristalnih struktura, raspoređivanje redoslijeda poslova itd.

U ovom je radu prvo dan uvod u genetske algoritme, a zatim su opisani problem trgovačkog putnika i problemi koji se svode na njega. Na kraju je prikazano rješavanje problema trgovačkog putnika pomoću genetskog algoritma, te su analizirani dobiveni rezultati.

2. Genetski algoritmi

2.1 Općenito o genetskim algoritmima

U prirodi preživljavaju jedinke koje se prilagode novim okolnostima. Jedinke sa vanjskim karakteristikama bolje prilagođenim okolišu imaju veće šanse za preživljavanje i povećanje svoje vrste, dok manje prilagođene jedinke izumiru, uzrokujući tako izumiranje čitavih vrsta. Na taj način priroda provodi prirodnu selekciju i osigurava da geni bolje prilagođenih jedinki imaju veće šanse za preživljavanje od onih koji su lošije prilagođeni. Upotrebom tog osnovnog biološkog modela, nastali su evolucijski algoritmi (*Evolutionary Algorithms*) čiji su dio genetski algoritmi.

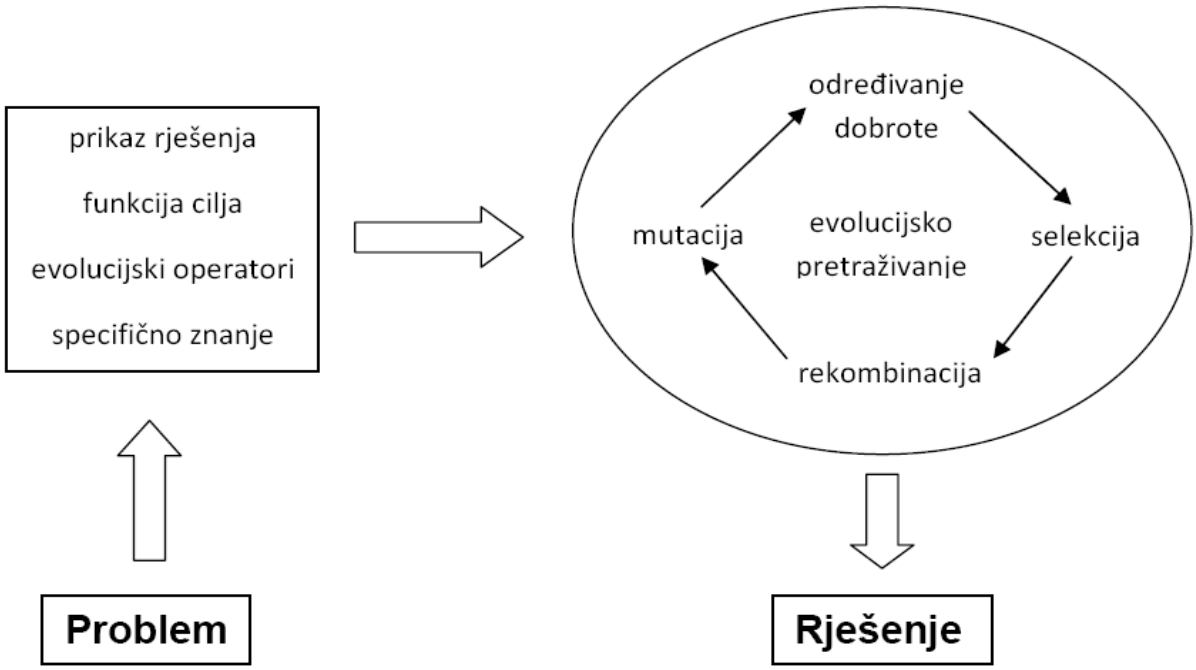
Genetski algoritmi (*Genetic Algorithms, GA*) su najpopularnija vrsta evolucijskih algoritama, a predstavljaju model optimiranja čije ponašanje potječe iz procesa evolucije koji se odvija u prirodi. Osnovni cilj genetskih algoritama je pronalaženje rješenja nekog problema koje tradicionalne determinističke metode ne mogu riješiti ([2]). Za razliku od većine determinističkih algoritama, genetski algoritmi pretragu ne započinju od jedne točke nego od cijelog niza potencijalnih rješenja. Ta potencijalna rješenja su najčešće generirana slučajnim odabirom, a predstavljaju početnu populaciju genetskog algoritma.

Početnoj se populaciji određuje dobrota (*fitness*). Dobrota je mjera kvalitete i govori koliko je neko rješenje dobro – bolja rješenja imat će veću dobrotu i obrnuto ([1]). Dobrota se određuje funkcijom cilja (funkcijom dobrote), koja ovisi o problemu koji se rješava. Kada se odredi dobrota, može započeti genetski proces. Početna populacija rješenja se iterira kroz generacije, a primjenjuje se princip preživljavanja najspremnijeg (*survival of the fittest*). U svakoj se generaciji odabiru bolja rješenja i odbacuju ona lošija. Odabrana rješenja su podvrgnuta genetskim operatorima. Prvo se križaju, a zatim s određenom vjerojatnošću i mutiraju. Tako nastaju nova potencijalna rješenja koja predstavljaju novu generaciju genetskog algoritma. Ta nova rješenja su u pravilu bolja nego stara rješenja od kojih su nastala. Cijeli se proces temelji na prirodnom procesu evolucije (Slika 2.1).

Stvaranje novih rješenja se nastavlja dok se ne zadovolji kriterij završetka genetskog procesa koji je definirao korisnik (Slika 2.2). Kada se kriterij zadovolji,

genetski algoritam je završio s radom. Međutim, to ne znači da je pronađeno optimalno rješenje. Završetak rada genetskog algoritma znači samo da je nađeno dovoljno dobro rješenje, koje može a i ne mora biti najbolje.

Kod genetskih algoritama često se upotrebljava pojam elitizam. Ako se koristi elitizam u genetskom algoritmu, najbolja jedinka iz trenutne generacije ide direktno u sljedeću, bez križanja i mutiranja. Unatoč tome, ta jedinka može dodatno sudjelovati i u stvaranju djece. Može se koristiti elitizam jedne jedinke ili elitizam više jedinki.



Slika 2.1. Rješavanje problema genetskim algoritmom

```
Genetski algoritam
{
t = 0;
generiraj početnu populaciju potencijalnih rješenja P(0);
odredi dobrotu P(0);
dok nije zadovoljen uvjet završetka evolucijskog procesa{
   t = t + 1;
   izaberi roditelje P'(t) iz P(t-1);
   rekombiniraj jedinke P'(t) i djecu spremi u P(t);
   mutiraj jedinke P(t);
   odredi dobrotu P(t);
}
ispiši rješenje;
}
```

Slika 2.2. Pseudokod genetskog algoritma

2.2 Prikaz rješenja

Svojstva živih bića zapisana su u njihovim kromosomima. Kromosomi su lančaste tvorevine koje se nalaze u jezgri svake stanice, a sastoje se od molekula DNK. Molekula DNK je nositelj informacija. Skup informacija koje karakteriziraju jedno svojstvo naziva se gen. Živa bića dobivaju jedan kromosom od oca, a jedan od majke što znači da za svako svojstvo postoje dva gena. Ti geni mogu biti ravnopravni ili jedan može biti dominantan, a drugi recesivan. Ako su geni ravnopravni, svojstvo potomka je negdje između svojstava oca i majke, a ako geni nisu ravnopravni onda potomak zadržava svojstvo dominantnog gena.

Postavlja se pitanje kako preslikati ove prirodne operacije na model strojnog učenja, tj. kako prikazati informaciju (rješenje) u genetskom algoritmu. Rješenja se najčešće prikazuju nizom bitova i binarnim brojevima. Glavna prednost takvog prikaza je u tome što se njegovi dijelovi zbog fiksne veličine mogu lako poravnati i uskladiti što omogućuje jednostavnu operaciju križanja. Na isti način se mogu koristiti i druge, složenije strukture poput matrica ili stabala. Korištenje rješenja

različite duljine otežava operaciju križanja. Kod problema određivanja rasporeda, rješenja se najčešće prikazuju nizovima cijelih brojeva.

2.3 Genetski operatori

2.3.1 Selekcija

Selekcija (*selection*) je odabir jedinki koje stvaraju potomstvo. Na taj se način čuvaju i prenose dobra svojstva (dobri geni) na sljedeću generaciju jedinki, dok se lošija svojstva odbacuju. Selekcija se vrši prema dobroti – jedinke s većom dobrotom imaju veće šanse da budu odabrane za roditelje u sljedećoj generaciji. Vjerojatnost odabira jedinke ovisi o njenoj dobroti, ali i o dobroti ostalih jedinki. Što je jedinka bliže ciljanom rješenju u odnosu na druge jedinke, njezina vjerojatnost razmnožavanja se povećava. Analogno tome, ako je iznos dobrote jedinke manji u odnosu na onaj drugih jedinki, njezine mogućnosti preživljavanja se smanjuju.

Postupak selekcije mogao bi se ostvariti jednostavnim odabirom N najboljih jedinki (gdje je N veličina populacije), ali takav odabir bi doveo do prerane konvergencije rješenja. Problem je u tome što se takvim odabirom gubi dobar genetski materijal koji mogu sadržavati loše jedinke. Zato i loše jedinke moraju imati određenu vjerojatnost preživljavanja (veću od nule).

Postoji mnogo načina odabira roditelja (*parent selection shemes*) za sljedeću generaciju, a najpoznatiji od njih su:

1. **Proporcionalni odabir** (*Proportionate Selection*). Jedinke se biraju proporcionalno prema dobroti. Jedinka sa najvećom dobrotom ima najveće šanse da bude izabrana, dok najlošija jedinka ima najmanje šanse preživljavanja. Vjerojatnost preživljavanja ostalih jedinki je negdje između.
2. **Odabir po rangu** (*Ranking Selection*). Populacija se sortira od najboljih jedinki prema najgorima. Broj kopija koje jedinka dobije određuje se funkcijom (*assignment function*) i proporcionalan je rangu jedinke. Rang jedinke je pozicija u odnosu na generaciju. Najbolja jedinka ima rang 1, dok najlošija jedinka ima rang N (gdje je N veličina populacije).

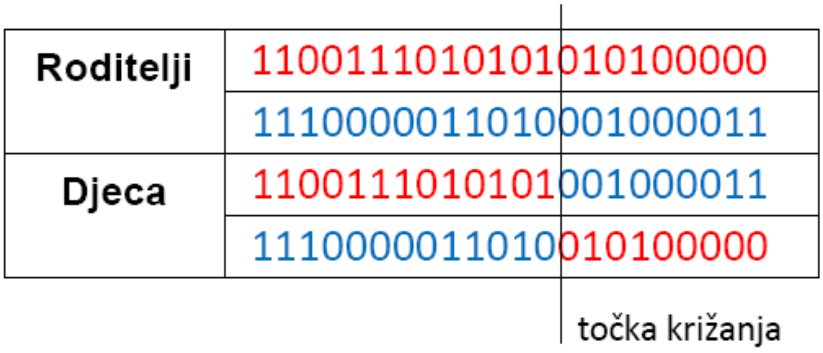
3. **Turnirski odabir** (*Tournament Selection*). Odabire se slučajan broj jedinki iz populacije (sa ili bez zamjena, ovisno o implementaciji), a najbolje od njih postaju roditelji sljedeće generacije. Postupak se ponavlja dok se ne popuni cijela sljedeća generacija, tj. dok broj odabranih jedinki ne postane jednak veličini populacije. Ovaj se odabir često naziva i k-turnirski odabir, gdje k označava broj jedinki čije se dobrote uspoređuju. Taj se broj naziva veličina prozora.

2.3.2 Križanje

Križanjem u prirodi genetski se materijal prenosi s roditelja na djecu. Novonastalo dijete posjeduje genetski materijal oba roditelja i ima jednak broj gena kao i oni, što znači da svaki roditelj daje samo pola svojih gena djetetu.

Isti se princip primjenjuje i u genetskim algoritmima. Procesom križanja (*crossover*) nastaju nove jedinke koje imaju kombinirane informacije sadržane u dvoje ili više roditelja. Križanje se često naziva i rekombinacija (*recombination*). Odabir križanja za genetski algoritam ovisi o odabranom načinu prikaza rješenja. [5] opisuje najpoznatije izvedbe križanja za jednostavni prikaz rješenja binarnim brojevima:

1. **Križanje u jednoj točki** (*one point crossover*). Odabire se jedna točka na roditeljima (jednako udaljena od početka za oba roditelja) i mijenja se desni dio kromosoma prvog roditelja s desnim dijelom kromosoma drugog roditelja.

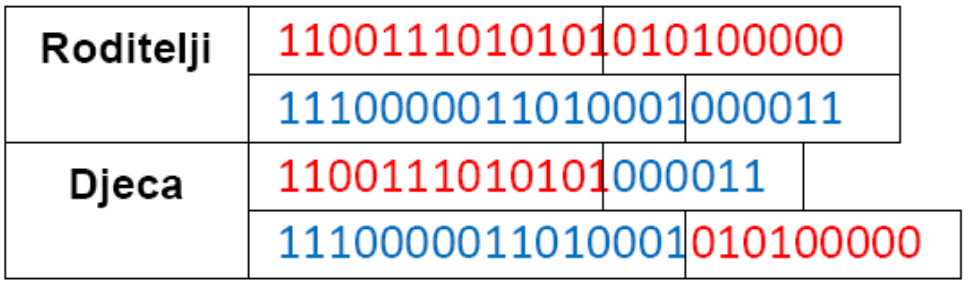


Slika 2.3. Križanje u jednoj točki

2. **Križanje u dvije točke** (*two point crossover*). Odabiru se dvije točke na roditeljima, pazeći da je pritom lijeva točka jednako udaljena od lijevog kraja i desna točka od desnog kraja kromosoma za oba roditelja. Roditelji izmjenjuju dijelove kromosoma između tih točaka.

Slika 2.4. Križanje u dvije točke

3. **Križanje rezanjem i spajanjem** (*cut and splice crossover*). Na roditeljima se odabire točka koja nije jednako udaljena od početka roditeljskih kromosoma. Mijenja se desni dio kromosoma jednog roditelja s desnim dijelom kromosoma drugog. Na taj način nastaju djeca koja imaju različite duljine kromosoma.



Slika 2.5. Križanje rezanjem i spajanjem

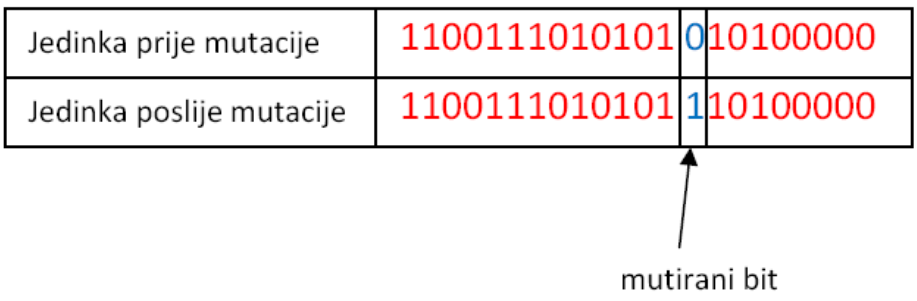
4. **Uniformno križanje** (*uniform crossover*). Uspoređuju se bit po bit oba roditelja i mijenjaju se sa fiksnom vjerojatnošću od 50%

5. **Polu-uniformno križanje** (*half-uniform crossover*). Uspoređuje se bit po bit oba roditelja i računa koliko ih se ne poklapa. Polovica od tih koji se ne poklapaju se mijenja.

2.3.3 Mutacija

U prirodi se mutacija definira kao slučajna promjena gena. Vjerojatnosti promjene za gene su različite, pa geni mogu biti stabilni i nestabilni. Vjerojatnost da gen A postane gen B nije ista kao da gen B postane gen A.

U genetskim algoritmima mutacija (*mutation*) je genetski operator koji se upotrebljava za dobivanje genetske raznolikosti sljedeće generacije rješenja od one postojeće. Najjednostavniji primjer mutacije je vjerojatnost da se neki bit u genetskom kodu (rješenju) promijeni iz svog originalnog stanja u novo stanje. To se postiže uvođenjem neke varijable za svaki bit u nizu. Ta varijabla govori hoće li bit biti modificiran ili neće. Ako se koristi binaran prikaz rješenja, mutacija je vjerojatnost da neki bit iz nule prijeđe u jedinicu ili iz jedinice u nulu (Slika 2.6). Mutacija sprječava rješenja da postanu preslična i na taj način uspore ili zaustave evoluciju. Kada se mutacija ne bi koristila, rješenja genetskog algoritma najčešće bi konvergirala prema nekom lokalnom optimumu.



Slika 2.6. Mutacija

2.4 Kriterij završetka

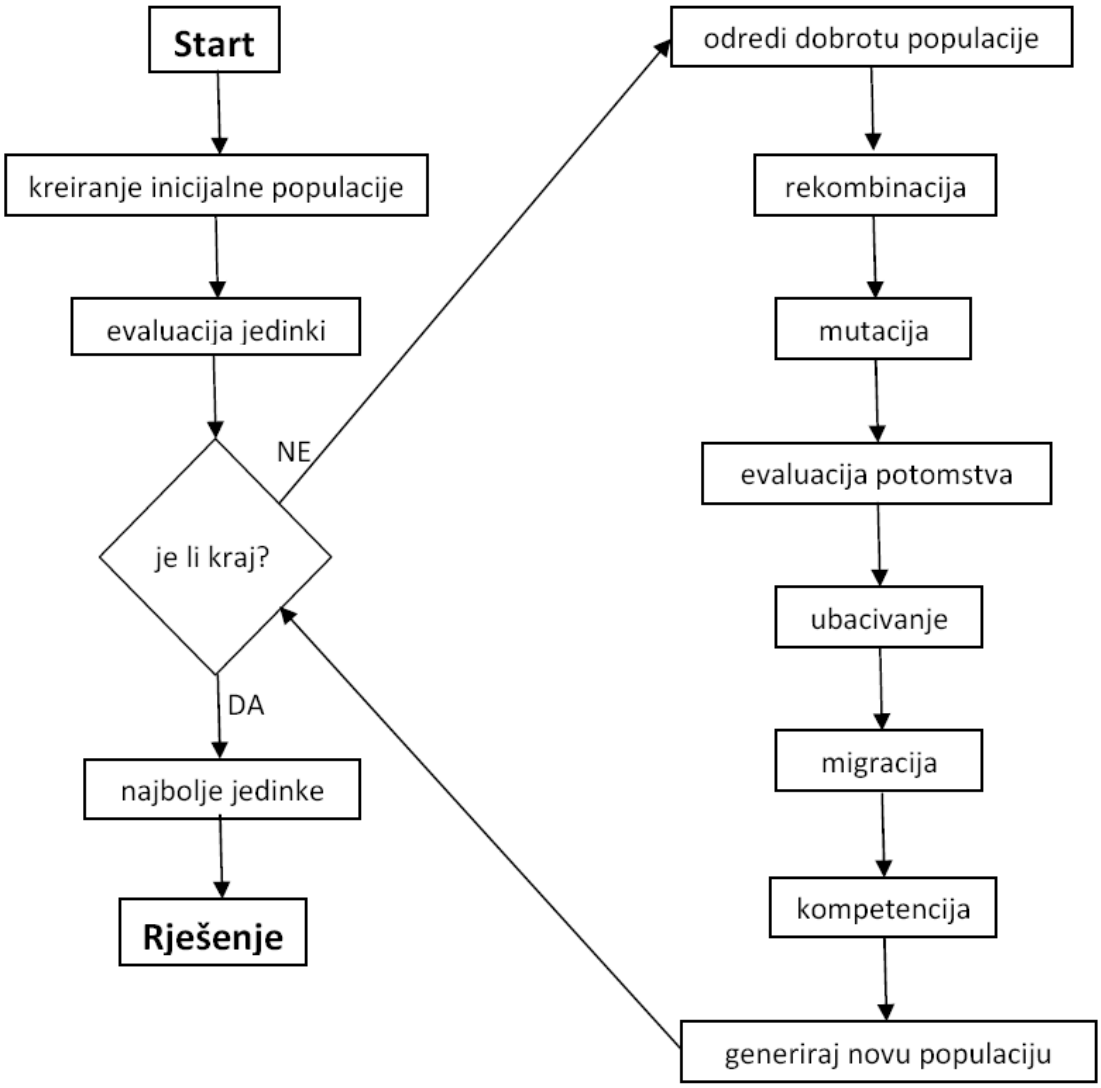
Genetski algoritam ponavlja proces generiranja sljedeće generacije sve dok se ne zadovolji kriterij završetka evolucijskog procesa, koji se još naziva i uvjet kraja ili kriterij kraja. Kriterij završetka je najčešće jedan od sljedećih:

- Pronađeno je rješenje koje zadovoljava minimalne kriterije
- Dosegnut je unaprijed određeni broj generacija
- Potrošen je budžet (vrijeme ili novac)
- Rješenje sa najvećim stupnjem dobrote dostiže ili je već dostiglo takvu granicu da daljnje iteracije ne daju bolje rezultate
- Ručno ispitivanje
- Kombinacije gore navedenih

2.5 Višepopulacijski genetski algoritmi

Genetski algoritmi mogu biti jednopopulacijski i višepopulacijski (*multi – population*). Višepopulacijski genetski algoritmi imaju više populacija pa se u njima javljaju neki novi genetski operatori, kao što su ubacivanje (*reinsertion*), migracija (*migration*) i natjecanje (*competition*). Svaka se populacija razvija izolirano nekoliko generacija prije nego se pojedine jedinke među populacijama zamijene. Postupak zamjene se naziva migracija. Pri natjecanju se dostupna sredstva dijele tako da uspješnije populacije dobivaju više, a one manje uspješne manje sredstava. Ubacivanje je ubacivanje jedinki iz jedne populacije u drugu.

U [4] je navedeno da su višepopulacijski genetski algoritmi sličniji prirodnom tijeku evolucije (Slika 2.7), pa samim time daju bolje rezultate nego jednopopulacijski. Obzirom da imaju više populacija u svakoj generaciji, zahtijevaju više memorije i usporavaju brzinu iteriranja po generacijama.



Slika 2.7. Višepopulacijski genetski algoritam

3. Problem trgovačkog putnika

3.1 Uvod u problem trgovačkog putnika

Problem trgovačkog putnika (*Traveling Salesman Problem, TSP*) je problem diskretne i kombinatorne optimizacije. Spada u skupinu NP-teških problema, a složenost mu je *O(n!)*. Matematičke probleme slične problemu trgovačkog putnika počeo je razmatrati Euler, kojeg je zanimalo kako bi skakač na šahovskoj ploči posjetio sva 64 mjesta samo jednom. Početkom 20.st. matematičari William Rowan Hamilton i Thomas Kirkman su razmatrali probleme koji se svode na problem trgovačkog putnika, a njegova opća forma se pojavljuje 30-ih godina 20. stoljeća. Pojam "trgovački putnik" prvi put je upotrijebljen 1932. godine.

Sam problem je vrlo jednostavan – trgovački putnik ima unaprijed definirane gradove i sve međusobne udaljenosti među njima, te mora posjetiti svaki grad samo jednom i vratiti se u početni grad. Pitanje je kojim bi redoslijedom trgovački putnik morao obilaziti gradove, a da ukupna duljina puta bude minimalna. Na prvi pogled ovaj se problem i ne čini preteškim, ali ako se uzme u obzir da ima faktorijelnu složenost (Slika 3.1) računanjem se dobije da već za 11 gradova postoji 1 814 400 mogućih redoslijeda obilazaka.

Postoji mnogo problema koji su povezani s problemom trgovačkog putnika ili se svode na njega. Neki od sličnih problema su:

1. Pronaći Hamiltonov ciklus najmanje težine u težinskom grafu.

   Hamiltonov ciklus (*Hamiltonian cycle*) je ciklus koji u težinskom grafu posjećuje svaki čvor samo jednom i vraća se u početni čvor. Razlika od problema trgovačkog putnika je u tome da svi čvorovi ne moraju biti međusobno povezani, dok su kod trgovačkog putnika svi gradovi povezani. Ovaj problem je također NP-težak problem.

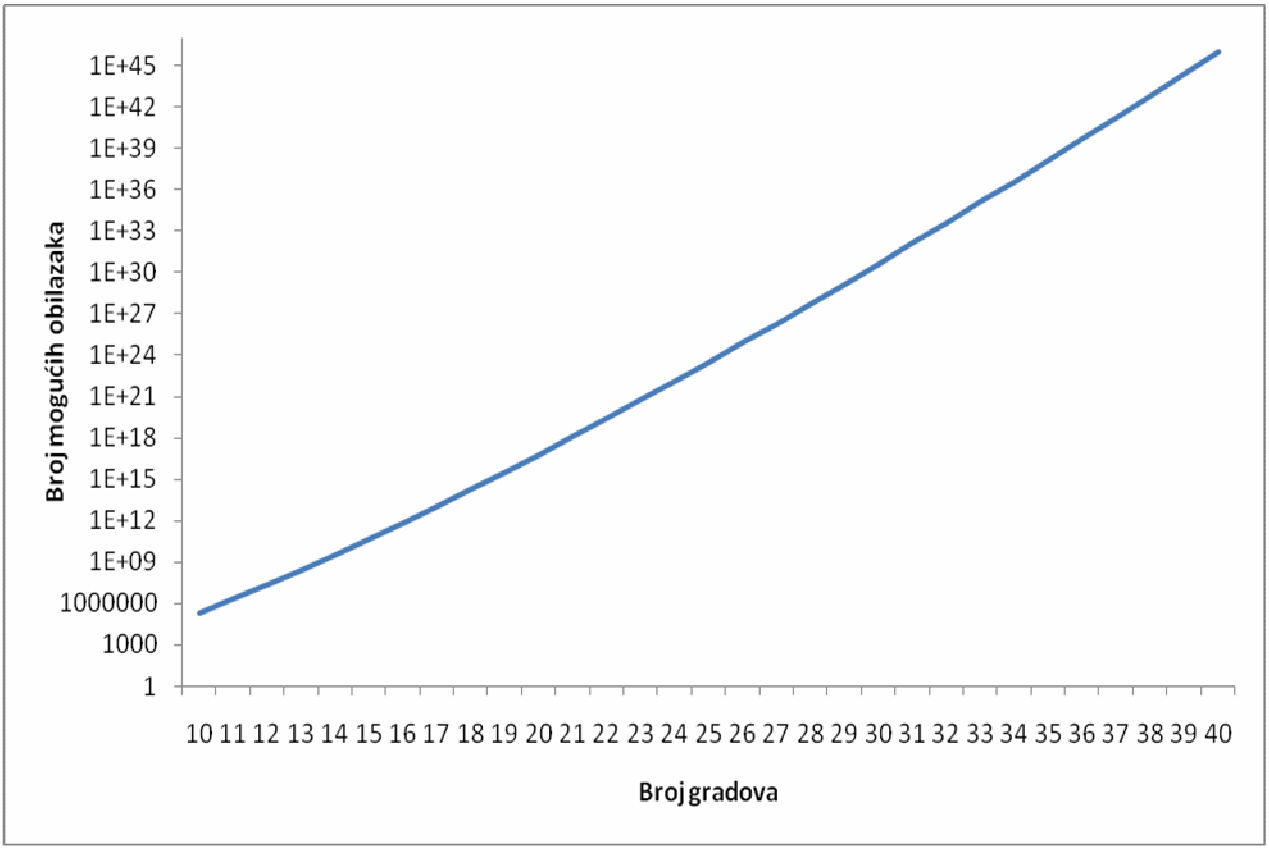
2. Problem trgovačkog putnika s uskim grlom (*Bottleneck Traveling Salesman Problem*).

   Potrebno je pronaći Hamiltonov ciklus u težinskom grafu, ali takav da udaljenost na najtežem bridu bude minimalna. Ovaj problem ima veliku

praktičnu važnost za prijevoznike i logističare, a javlja se i prilikom bušenja i određivanja redoslijeda poslova.

3. Opći problem trgovačkog putnika (*Generalized Traveling Salesman Problem*).

   Trgovački putnik ima na raspolaganju gradove, ali i države, te mora posjetiti točno jedan grad iz svake države, a da prevaljeni put bude minimalan. Ovako definiran problem često se naziva i problem putujućeg političara (*Traveling Politician Problem*), a može se svesti na obični problem trgovačkog putnika sa istim brojem gradova, ali modificiranim udaljenostima.



Slika 3.1. Složenost problema trgovačkog putnika

3.2 Metode rješavanja

Zbog svoje faktorijelne složenosti, problem trgovačkog putnika predstavlja pravi izazov za rješavanje. Pojavilo se mnogo algoritama za rješavanje, a izumom računala uspješnost rješavanja se uvelike popravila. Neki algoritmi za rješavanje navedeni su u nastavku.

3.2.1 Pretraga grubom silom

Pretraga grubom silom (*brute force search*) je najjednostavniji način rješavanja problema trgovačkog putnika. Međutim, zbog svoje jednostavnosti ovo je ujedno i najlošiji (najsporiji) način. Ideja je da se jednostavno isprobaju sve permutacije i pamti najbolja kombinacija. Obzirom da je složenost problema faktorijelna, ovakva pretraga gubi praktično značenje već kod malog broja gradova. Može se koristiti za probleme do 10-ak gradova. Dobiveno konačno rješenje je uvijek optimalno.

3.2.2 Pohlepni algoritam

Rješavanje problema trgovačkog putnika pohlepnim algoritmom (*Greedy Algorithm*) obično ljudima "prvo padne na pamet". Pohlepni algoritam pretragu započinje od proizvoljnog početnog grada. Svaki sljedeći grad je onaj koji je najbliži trenutnom gradu, a da još nije posjećen. Pohlepni algoritam obično (ali ne uvijek) ne daje optimalno rješenje, jer ne koristi sve dostupne podatke. Tako se primjerice često posjećuje neki grad prerano što onemogućuje pronalazak boljeg rješenja kasnije. Važnost pohlepnog algoritma je u tome što se može brzo provesti, a daje odličnu aproksimaciju optimuma. Većinom se koristi za usporedbu i testiranje drugih algoritama.

Pohlepni se algoritam često naziva i algoritam najbližeg susjeda (*Nearest Neighbour Algorithm*), jer se u svakom koraku posjećuje grad koji je najbliži trenutnom gradu.

3.2.3 Grananje i ograničavanje

Grananje i ograničavanje (*Branch and Bound*) je opći algoritam za traženje optimalnih rješenja mnogih optimizacijskih problema. Sastoji se od brojanja svih potencijalnih rješenja, pri čemu se veliki podskupovi loših rješenja odbacuju. Loša rješenja su ona koja su unutar nekog intervala, tj. između predefinirane gornje i donje granice.

Redoslijed obilaska gradova trgovačkog putnika se prvo rekurzivno grana na dva dijela, te nastaje struktura stabla. Nakon toga se računaju gornja i donja granica za svaki dobiveni podskup. Ako je donja granica nekog čvora bolja od gornje granice nekog drugog čvora, taj drugi čvor se izbacuje iz pretrage. Rekurzija se zaustavlja kad se skup reducira na jedan element ili kad njegova gornja granica postane jednaka njegovoj donjoj granici.

3.2.4 Linearno programiranje

Linearno programiranje (*Linear Programming*) je optimiziranje linearne funkcije cilja, koja je podvrgnuta ograničenjima linearne jednakosti i nejednakosti. Ono traži način kako postići najbolji rezultat upotrebljavajući listu zahtjeva koja je opisana linearnim jednadžbama. Omogućuje uspješno rješavanje problema veličine do 200 gradova.

3.2.5 Najbliže ubacivanje

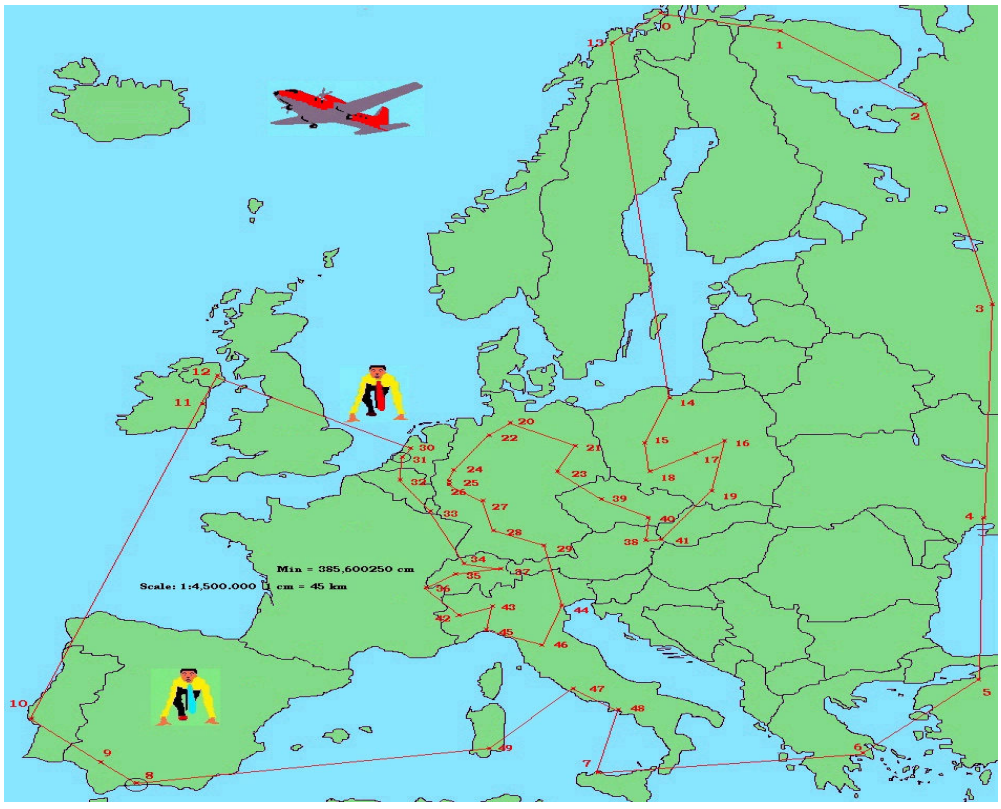
Najbliže ubacivanje (*Nearest Insertion*) uzima neki početni grad koji predstavlja početnu rutu. Zatim traži drugi grad koji je najbliži početnome te ga ubacuje u početnu rutu. Ruta sada sadrži dva grada, a sljedeći grad koji se dodaje je onaj grad koji je najbliži bilo kojem gradu u ruti, a da već nije sadržan u njoj. Taj se grad ubacuje u rutu na optimalno mjesto, tako da dobivena udaljenost bude minimalna. Postupak se ponavlja dok se ne ubace svi gradovi.

3.2.6 Najdalje ubacivanje

Najdalje ubacivanje (*Farthest Insertion*) je slično najbližem ubacivanju. Razlika je u tome što se u kod najdaljeg ubacivanja u rutu dodaje grad koji je najudaljeniji od bilo kojeg grada u ruti, a ne onaj koji je najbliži. Najudaljeniji grad se dodaje na optimalno mjesto, tako da nova uvaljenost bude minimalna.

3.2.7 Evolucijski algoritmi

Rješavanje problema trgovačkog putnika evolucijskim algoritmima je vrlo često ([3]). Iako evolucijski algoritmi ne nalaze uvijek optimalno rješenje, njihova prednost je što mogu u razumnom vremenu naći rješenje koje je svega 2 ili 3% lošije od optimalnog. Mogu se primijeniti na probleme različite veličine – od onih malih do onih ogromnih koji se sastoje od nekoliko stotina tisuća gradova ([6]). Dio evolucijskih algoritama su genetski algoritmi. Više o rješavanju problema trgovačkog putnika genetskim algoritmima nalazi se u poglavlju 5.



Slika 3.2. Problem trgovačkog putnika na području Europe

4. Problemi raspoređivanja poslova

4.1 Opis problema

Problemi raspoređivanja poslova (*Scheduling Problems*) su problemi u kojima treba alocirati ograničena sredstva za obavljanje nekih aktivnosti tijekom vremena. Oni predstavljaju kompliciran zadatak koji može biti definiran pomoću niza ograničenja koja se trebaju ispuniti kako bi zadatak bio uspješno obavljen. Konačno rješenje mora zadovoljavati sve postavljene uvjete. Kada su svi uvjeti zadovoljeni, raspored se može optimizirati prema nekom kriteriju. Kriterij može biti cijena, kašnjenje, vrijeme obrade dijela proizvoda, vrijeme obrade cijelog proizvoda itd.

Na raspolaganju je konačan skup od *n* poslova i *m* strojeva. Svaki se posao sastoji od skupa operacija, a svaki stroj može obavljati najviše jedan posao odjednom. Svaki posao se mora obavljati neko vrijeme na nekom stroju i za to vrijeme ne smije biti prekidan. Potrebno je pronaći optimalan raspored, tj. alocirati takav redoslijed obavljanja zadanih poslova na zadanim strojevima da ukupna duljina obavljanja posla bude minimalna.

Neki poslovi imaju velika ograničenja zbog nedobavljivosti sredstava (bilo novca ili opreme) i/ili malog vremenskog roka u kojem se moraju obaviti. Poslovi su obično disjunktni, što znači da dva zadatka ne mogu upotrebljavati isto sredstvo u isto vrijeme. Problem raspoređivanja poslova je NP-težak problem, a [8] navodi da je za određivanje mogućeg i učinkovitog (*feasible*) rješenja potrebno definirati postupke kako bi se složenost problema smanjila.

Problemi raspoređivanja poslova uz vremena postavljanja strojeva (*Job Scheduling Problems with setup times*) formalno su prikazani u [7] kao niz poslova $J=\{J_1, J_2,\ldots, J_n\}$ i niz sredstava $R=\{R_1, R_2,\ldots, R_n\}$. Svaki se posao J_i sastoji od skupa operacija $O_i=\{O_{i1}, O_{i2}, \ldots, O_{in}\}$ koje moraju biti obavljene između početnog (rt_i) i krajnjeg trenutka (dt_i). Obavljanje svake operacije O_i zahtjeva upotrebu skupa sredstava $R_{ik} \subseteq R$ tijekom nekog vremenskog intervala.

Postoji mnogo varijacija na temu problema raspoređivanja poslova uz vremena postavljanja strojeva. Tako primjerice strojevi mogu biti povezani ili

neovisni, jednaki ili različiti, mogu ili ne moraju zahtijevati određen razmak između obavljanja dvaju poslova. Poslovi često imaju takva ograničenja da jedan posao mora završiti prije nego se drugi počne izvršavati. Neki poslovi ovise o strojevima na kojima se izvode, pa se ne mogu izvoditi na svima nego na točno određenim strojevima. Drugi se pak brže izvode na jednoj vrsti strojeva, a sporije na nekoj drugoj.

4.2 Metode rješavanja

Prilikom rješavanja problema raspoređivanja poslova, postavljaju se pitanja kada maknuti trenutni posao sa stroja, koji posao staviti sljedeći i kada ga staviti. Posao se miče sa stroja kada je gotov, a može se maknuti i ako je dozvoljeno prekidanje, uz uvjet da je posao višeg prioriteta spreman za izvođenje.

Kad tvornica proizvodi nekoliko vrsta proizvoda ili kad stroj ima više poslova za obaviti, treba se odrediti redoslijed izvođenja poslova. Upotreba jedinstvenog sustava za proizvodnju različitih komponenti najčešće traži neko vrijeme pripreme stroja između obavljanja dvaju poslova. Vrijeme pripreme stroja je neproduktivno vrijeme jer se za njegovog trajanja na stroju ne obavlja niti jedan posao, pa ga je potrebno minimizirati. Ono ovisi o proizvodu koji se netom proizvodio i proizvodu koji će se sljedeći proizvoditi. Ako tvornica ima jedan stroj koji obavlja četiri posla, moguća situacija prikazana je u tablici (Tablica 4.1) .

Tablica 4.1. Vrijeme postavljanja stroja

	Posao 1	Posao 2	Posao 3	Posao 4
Posao 1	0	30	40	40
Posao 2	55	0	25	30
Posao 3	20	10	0	10
Posao 4	10	50	15	0

U tablici su prikazana vremena (u minutama) potrebna da se stroj pripremi za obavljanje novog posla. Primjerice, ako je stroj obavljao posao 3, tada mu je potrebno 25 minuta da se pripremi za obavljanje posla 2. Ako se isti posao obavlja dva puta zaredom, vrijeme pripreme stroja jednako je nuli. Ukoliko stroj mora obaviti sva 4 posla, a početni posao je posao 1, moguće kombinacije i potrebna vremena postavljanja su:

- 1-2-3-4 55 + 10 + 15 = 80
- 1-2-4-3 55 + 50 + 10 = 115
- 1-3-2-4 20 + 25 + 50 = 95
- 1-3-4-2 20 + 15 + 30 = 65
- 1-4-2-3 10 + 30 + 10 = 50
- 1-4-3-2 10 + 10 + 25 = 45

Očito je da je optimalna kombinacija 1-4-3-2, čije ukupno utrošeno vrijeme predstavlja minimum mogućeg utrošenog vremena. U praksi se obično poslovi proizvode ciklično, što znači da bi u prethodnom primjeru trebalo završiti s poslom s kojim se i počelo. U tom slučaju rezultati bi bili:

- 1-2-3-4-1 55 + 10 + 15 + 40 = 120
- 1-2-4-3-1 55 + 50 + 10 + 40 = 155
- 1-3-2-4-1 20 + 25 + 50 + 40 = 135
- 1-3-4-2-1 20 + 15 + 30 + 30 = 95
- 1-4-2-3-1 10 + 30 + 10 + 40 = 90
- 1-4-3-2-1 10 + 10 + 25 + 30 = 75

Potrebno vrijeme se povećalo, ali optimalna kombinacija je ostala ista (iako se i ona mogla promijeniti). Ako se malo bolje pogledaju sve kombinacije i način izračunavanja optimuma, može se zaključiti da se problem raspoređivanja poslova na jednom stroju uz vremena postavljanja svodi na problem trgovačkog putnika. Zato se on rješava istim metodama kao i problem trgovačkog putnika.

5. Rješavanje problema trgovačkog putnika uz pomoć genetskih algoritama

5.1 Prikaz rješenja

Funkcija cilja za problem trgovačkog putnika je vrlo jednostavna i prirodno se nameće – dva se rješenja mogu uspoređivati prema duljini obilaska svih gradova te se lako može zaključiti koje od njih je bolje. Međutim, stvar sa izborom prikaza rješenja je sasvim suprotna. Binarni prikaz puta je potpuno neprikladan, jer je potrebno permutirati cijele gradove, pa njihov binarni kod nema nikakvo značenje. Binarni bi kod uzrokovao i kreiranje nekih ilegalnih rješenja, pa bi potrebni bili algoritmi za popravljanje (*Repair Algorithm*). Zato se rješenja problema trgovačkog putnika najčešće prikazuju vektorski i matrično. Pri vektorskom prikazu rješenja su prikazana kao lista gradova. Vrste vektorskog prikaza su zbijeni prikaz, redni prikaz i prikaz po redoslijedu obilaska.

5.1.1 Zbijeni prikaz

Pri zbijenom prikazu (*Adjacency Representation*) gradovi su prikazani listom, a grad *j* se nalazi na poziciji *i* ako i samo ako put vodi iz grada *i* u grad *j*. Primjerice, vektor

(2 4 8 3 9 7 1 5 6)

predstavlja redoslijed obilaska

1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7.

Svaki redoslijed obilaska ima jedinstven zbijeni prikaz, ali neki vektori predstavljaju ilegalne rute. Tako vektor

(2 4 8 1 9 3 5 7 6)

predstavlja ilegalnu rutu

1 – 2 – 4 – 1.

Ta je ruta dovela do preranog zatvaranja ciklusa, što znači da bi putnik posjetio jedan grad dvaput (a to nije u skladu s uvjetom u opisu problema).

5.1.2 Redni prikaz

Kod rednog prikaza (O*rdinal Representation*) postoji uređena lista gradova koja služi kao referentna točka za gradove u rednom prikazu. Primjer takve liste je

C = (1 2 3 4 5 6 7 8 9).

Put je prikazan listom gradova, a *i*-ti element liste je broj iz intervala [1, n-i+1], gdje je n broj elemenata. Primjer takvog prikaza je

L = (1 1 2 1 4 1 3 1 1).

Prvi broj u prikazu je 1, pa se uzima prvi grad iz liste C i briše se iz nje. Tako se dobiva da je prvi grad 1. Zatim se čita sljedeći broj iz liste L, koji je opet 1 pa se uzima prvi broj iz promijenjene liste C. Prvi grad je sada broj 2, što znači da se on briše iz liste C i da postaje drugi grad traženog obilaska. Sljedeći broj u listi L je 2, pa se čita drugi grad iz liste C i dobiva 4. Postupak se ponavlja za sve gradove i dobiva se obilazak

1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7.

Glavna prednost rednog prikaza je što se s njim može koristiti klasično križanje. Bilo koje dvije ture rednog prikaza klasičnim križanjem u jednoj točki daju djecu koja su legalne ture.

5.1.3 Prikaz po redoslijedu obilaska

Prikaz po redoslijedu obilaska (*Path Representation*) je najprirodniji prikaz problema trgovačkog putnika. U njemu prikaz

(4 5 1 2 3 9 8 6 7)

predstavlja redoslijed obilaska

4 – 5 – 1 – 2 – 3 – 9 – 8 – 6 – 7.

Ovakav je prikaz korišten u ovom radu.

5.2 Opis ideje

Prije pokretanja genetskog algoritma, stvara se instanca razreda *Gradovi*. Taj razred sadrži sve potrebne podatke o gradovima. Imena gradova se slučajnim odabirom čitaju iz datoteke *Gradovi.txt* koja sadrži popis potencijalnih imena. Ako je broj gradova manji od 9, udaljenosti među njima se generiraju dinamički, a optimalno rješenje se pretražuje grubom silom (*brute force attack*). Takva je pretraga neučinkovita već za mali broj gradova, pa ako je broj gradova veći ili jednak 9 potrebno je imati datoteku koja sadrži definirane udaljenosti među gradovima i unaprijed poznat optimalni put. Naziv te datoteke je *N Gradova.txt*, gdje N predstavlja broj gradova. Udaljenosti među gradovima su predstavljene cijelim brojem u intervalu [1, 10000]. Razmatrani problem trgovačkog putnika je simetričan, što znači da je udaljenost od grada A do grada B jednaka udaljenosti od grada B do grada A.

Jedno rješenje genetskog algoritma predstavljeno je razredom *Jedinka*. On sadrži podatke o rješenju koji su potrebni za rad algoritma i genetskih operatora (redoslijed obilaska, ukupna udaljenost, dobrota, jedinstveni broj u generaciji).

Sam genetski proces je opisan u razredu *GenetskiAlgoritam*. Korišten je jednopopulacijski genetski algoritam. Početkom izvođenja algoritma stvara se lista jedinki koja predstavlja populaciju. Nakon toga se svim jedinkama određuje dobrota. Formula za određivanje dobrote prikazana je jednadžbom (5.1). Dobrota se određuje tako da se prvo izračuna najveća moguća udaljenost obilaska, a to je ona koja bi se dobila kada bi svi gradovi bili maksimalno udaljeni jedan od drugoga. Dobrota jedinke je razlika između te najveće duljine obilaska i duljine obilaska jedinke. Ovakvo određivanje dobrote je apsolutno, što znači da dobrota jedinke u prvoj i u npr. petoj generaciji ima isto značenje.

$$\mathrm{d} = 10000 * (\mathrm{n} + 1) - p \tag{5.1}$$

d označava dobrotu jedinke, *n* je broj gradova, a *p* je duljina obilaska jedinke čija se dobrota određuje.

Nakon određivanja dobrote ulazi se u petlju koja se izvršava dok se ne postigne definirani broj generacija ili dok ne stigne zahtjev za zaustavljanje algoritma. U petlji se selektiraju jedinke koje će biti roditelji u sljedećoj generaciji, a

zatim se križaju i mutiraju. Nakon toga se stvara nova generacija i određuje njena dobrota. Kada određivanje dobrote završi, prelazi se na sljedeću iteraciju petlje. Vrsta križanja i mutacije se odabire u grafičkom sučelju. Svaka mutacija implementira sučelje (*interface*) *Mutacija*, a svako križanje implementira sučelje *Krizanje*. Tijekom rada programa, ovisno o izboru korisnika, stvaraju se odgovarajući objekti.

U algoritmu se koristi elitizam najbolje jedinke, što znači da najbolja jedinka uvijek ide u sljedeću populaciju bez podvrgavanja križanju i mutaciji. Bez obzira na elitizam, ta najbolja jedinka ima podjednake šanse da bude izabrana u selekciji kao i svaka druga jedinka.

Prije početka rada algoritma, moguće je definirati što će se ispisivati i koliko često. Ispisi se mogu mijenjati i kasnije, tijekom rada algoritma, ali o tome više u poglavlju o grafičkom sučelju (5.4).

5.3 Korišteni genetski operatori

5.3.1 Selekcija

U ovom je radu korištena turnirska selekcija. Veličina prozora za selekciju (broj jedinki koji se promatra u tom trenutku) se može definirati prije pokretanja programa, a pretpostavljena vrijednost se dobije slučajnim odabirom. Veličina prozora mora biti veća od dva i manja od veličine populacije. Od svih jedinki obuhvaćenih prozorom, dvije najbolje postaju roditelji jednike u sljedećoj generaciji. Postupak se ponavlja dok se ne popuni cijela sljedeća generacija.

5.3.2 Križanje

Napravljene su četiri implementacije križanja. Prva implementacija je pohlepno križanje (*Greedy Crossover*). Pri pohlepnom križanju uzima se prvi grad od jednog roditelja kao početni grad djeteta. Zatim se gleda koji su gradovi sljedeći iza tog grada kod oba roditelja. U dijete se dodaje onaj sljedeći grad koji je bliži (za kojeg je prevaljeni put iz posljednjeg grada djeteta manji). Ako se taj bliži grad ne može

dodati jer se već nalazi u djetetovoj ruti, onda se dodaje udaljeniji grad. Ako dijete već sadrži i taj grad, tada se sljedeći grad bira slučajnim odabirom. Postupak se ponavlja dok se ne popuni cijela ruta djeteta. Drugo dijete se dobiva na isti način, uz razliku da se za početni grad uzima početni grad od drugog roditelja.

Druga implementacija križanja je gore navedeno pohlepno križanje u kombinaciji s operatorom inverzije (*inversion*). Dva roditelja prvo daju djecu po postupku pohlepnog križanja, a zatim se na svakom djetetu slučajnim odabirom biraju dvije točke, te se redoslijed obilaska između tih točaka invertira.

Treće križanje koje je implementirano je OX križanje. Kod ovog se križanja slučajnim odabirom izabiru dvije točke križanja na roditeljskim kromosomima. Dijelovi kromosoma između tih točaka ostaju isti. Zatim se, počevši od druge točke križanja jednog roditelja, ubacuju gradovi drugog roditelja. Gradovi koji su već prisutni u prvom roditelju se ne ubacuju, a redoslijed ubacivanja jednak je redoslijedu gradova drugog roditelja. Isti postupak vrijedi i za dobivanje drugog djeteta.

Posljednje implementirano križanje je CX križanje. Kod njega se prvi grad uzima iz prvog roditelja, a svaki sljedeći grad je onaj grad koji se nalazi u drugom roditelju na istom mjestu kao i trenutni grad u prvom roditelju. Taj se grad dodaje na poziciju koju je imao u prvom roditelju. Početni grad drugog djeteta je prvi grad od drugog roditelja, a postupak dobivanja ostalih gradova je isti kao i za prvo dijete.

5.3.3 Mutacija

Prilikom rješavanja problema trgovačkog putnika genetskim algoritmom, implementirane su četiri mutacije. Vjerojatnost mutacije može se definirati prije pokretanja programa.

Mutacija zamjenom dvaju gradova uzima dva grada dobivena slučajnim odabirom, te im mijenja mjesta u redoslijedu obilaska. Gradovi koji se mijenjaju ne moraju biti susjedni.

Mutacija ubacivanjem grada slučajnim odabirom izabire jedan grad, te ga ubacuje na neko novo mjesto u obilasku.

Preostale mutacije se svode na mutaciju ubacivanjem. Kod mutacije optimalnim ubacivanjem uzima se neki grad dobiven slučajnim odabirom te se ubacuje na optimalno mjesto. Optimalno mjesto je ono mjesto u obilasku koje rezultira minimalnom duljinom obilaska. Ako se izabrani grad već nalazi na optimalnom mjestu, do mutacije ne dolazi.

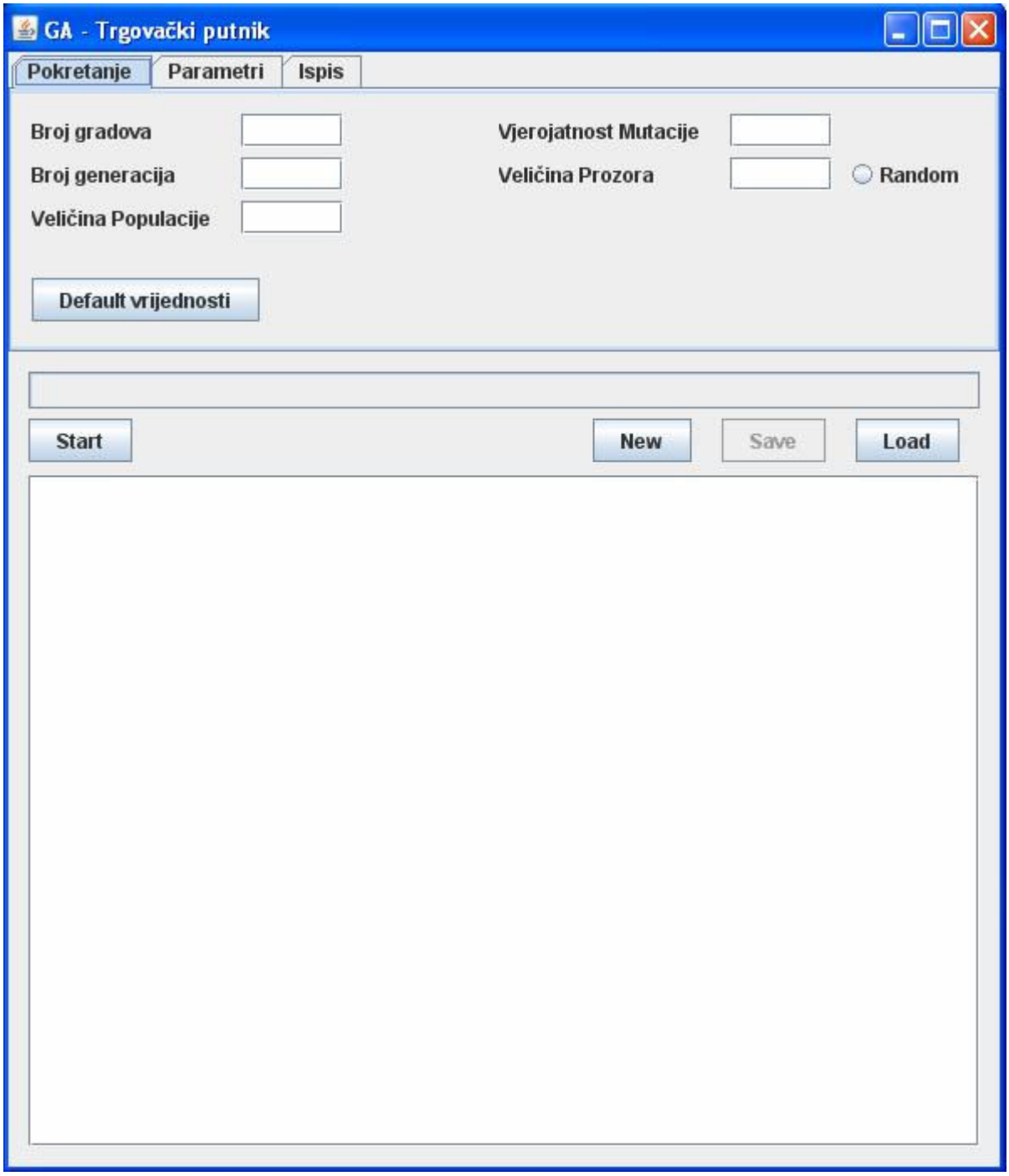
Mutacija pohlepnom zamjenom (*Greedy Swap Mutation*) slučajnim odabirom bira neki grad i mjesto na koje bi se on mogao ubaciti. Grad se ubacuje na to mjesto ako i samo ako je duljina novog obilaska kraća od duljine starog. U protivnome do mutacije ne dolazi.

5.4 Grafičko sučelje

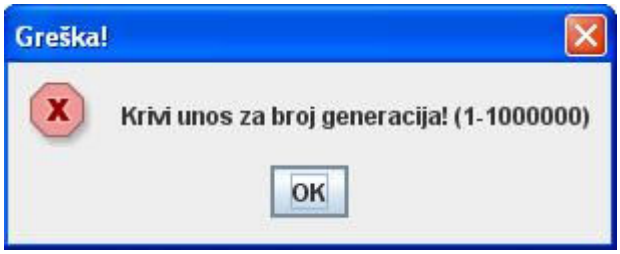
Genetski algoritam je napisan u javi, pa je za pokretanje potreban Java Virtual Machine. Izvršna jar datoteka se pokreće dvostrukim klikom, nakon čega se otvara grafičko sučelje (Slika 5.1). Programom se upravlja iz grafičkog sučelja.

Grafičko sučelje se sastoji od plohe s karticama (*tabovima*) na vrhu i bijele površine predviđene za ispis rezultata (*text area*). Tijekom rada program ispisuje rezultate na područje predviđeno za ispise. Napredak algoritma kroz generacije prikazuje se grafičkim indikatorom (*progress bar*). Grafički indikator se postavlja na novu vrijednost i prilikom svakog učitavanja sa diska, te prikazuje napredak netom učitanog stanja programa. Sam rad genetskog algoritma izvodi posebna dretva (*thread*), pa ne bi trebalo biti problema s korištenjem grafičkog sučelja tijekom rada samog algoritma.

Program se pokreće pritiskom na tipku *Start*. Kada je tipka *Start* pritisnuta, program čita podatke iz tekst polja *Broj gradova, Broj generacija, Veličina populacije, Vjerojatnost mutacije* i *Veličina prozora* te postavlja odgovarajuće parametre algoritma. Podaci se u tekst polja moraju upisati prije pritiska tipke *Start*. Ukoliko su upisane vrijednosti pogrešne ili nije upisano ništa, program prikazuje poruku o greški. Primjer poruke o greški prikazan je na slici (Slika 5.2). U poruci uvijek piše razlog greške, a u zagradama su navedene dozvoljene vrijednosti parametara. Dozvoljene vrijednosti parametara se pojavljuju i kao *tooltipovi*.



Slika 5.1. Izgled grafičkog sučelja



Slika 5.2. Poruka o greški

Ako tipka *Random* nije označena, podatak o veličini prozora se čita iz tekst polja. Označavanjem tipke *Random* veličina prozora za selekciju se određuje slučajnim odabirom, a tekst polje za unos veličine prozora postaje nedostupno. Analogno tome, micanjem označenosti te tipke tekst polje postaje dostupno.

Tipka *Default vrijednosti* postavlja sva tekst polja na pretpostavljene vrijednosti. Tako se broj generacija postavlja na 10000, broja gradova na 29, veličina populacije na 40 jedinki, vjerojatnost mutacije na 1.0, a veličina prozora na slučajan odabir. Ako se tipka pritisne dok tekst polja imaju neke vrijednosti, pritisak na nju će prepisati pretpostavljene preko trenutnih vrijednosti.

Kada se algoritam pokrene, tipka *Start* mijenja svoje ime i funkciju u *Stop*. Pritiskom na tipku *Stop* zaustavlja se algoritam. Treba napomenuti da se algoritam neće zaustaviti trenutno, nego će završiti preostale operacije križanja i mutiranja i tek onda zaustaviti svoj rad. Kada se stisne tipka *Stop*, ona mijenja svoje ime u *Continue*. Tipka *Continue* omogućava nastavljanje izvođenja genetskog algoritma, od trenutka kada je zaustavljen. Prije pritiska tipke *Continue*, mogu se mijenjati parametri o ispisu i ciljani broj generacija. Moguće je promijeniti i veličinu populacije, vjerojatnost mutacije i veličinu prozora, iako se to obično ne radi usred rada algoritma. Kada algoritam završi s radom, tipka *Stop* automatski mijenja svoje ime u *Continue*.

Tipka *Save* omogućuje spremanje trenutnog stanja algoritma na disk. Analogno tome, tipka *Load* čita podatke s diska i sprema ih u genetski algoritam, te tako omogućava nastavak izvođenja. Ako je učitavanje uspješno izvedeno, tipka *Start* mijenja svoje ime u *Continue*. Pritiskom na tipku *New* cijeli se program resetira, te se kreće s radom ispočetka. Treba napomenuti da su tipke *Load, Save* i *Continue* omogućene samo kada je algoritam zaustavljen. Prije njihova

korištenja, algoritam treba zaustaviti (ako već nije zaustavljen) pritiskom na tipku *Stop*.

Pritiskom na karticu *Parametri* može se izabrati jedna od četiri mogućnosti mutacije i križanja. Uz svaku opciju je navedeno ime i pomoćni opis koji se pojavljuje ako se miš zadrži neko vrijeme nad njom. Opcija se bira označavanjem kružića uz ime opcije. Za rad algoritma mora biti izabrana točno jedna mutacija i točno jedno križanje. Korištena križanja i mutacije su objašnjeni u poglavljima 5.3.2 i 5.3.3. Pretpostavljena mutacije je *Ubacivanje grada na slučajno mjesto*, a pretpostavljeno križanje je *Pohlepno križanje*. Tipka *Default vrijednosti* iz kartice *Pokretanje* ne postavlja mutaciju i križanje na njihove pretpostavljene vrijednosti. Pretpostavljene vrijednosti se postavljaju jedino pokretanjem programa ili pritiskom na tipku *New*.

Kartica *Ispis* omogućuje definiranje ispisa programa. Ispisivati se može najbolja jedinka, cijela populacija, križanje i mutacija. Pretpostavljene vrijednosti ispisuju najbolju jedinku svakih 20 generacija. Ispis se aktivira i deaktivira pritiskom na kvadratić za izbor (*checkbox*). Kada je neki ispis aktiviran, u ekvivalentnom tekst polju mora biti upisan broj koji označava svakih koliko će se generacija ispisivati navedena opcija. Valja napomenuti da ispisi uvelike usporavaju program, pa treba biti umjeren u njihovom korištenju. Pritisak na tipku *Start* provjerava i vrijednosti koje su navedene u tekst poljima za ispis, te ispisuje obavijest o greški ukoliko u nekom polju piše besmislena vrijednost. Bez obzira na odabrane opcije, cijela prva i posljednja generacija algoritma se uvijek ispisuju, da se ne desi da algoritam završi, a korisnik ne vidi rezultate. Isto vrijedi i za najbolju jedinku prve i posljednje generacije.

Pritiskom na tipku *Ispis gradova* ispisuju se gradovi u obliku tablice. Da bi se mogli ispisati, gradovi moraju biti kreirani, a da bi bili kreirani algoritam je morao početi raditi. Ukoliko gradovi nisu kreirani, ispisuje se poruka o greški.

6. Analiza rezultata

Genetski algoritam je testiran na dvojezgrenom AMD Turionu sa 2GB RAM memorije. Za probleme koji se mogu dinamički generirati (to su problemi do 8 gradova) algoritam uvijek nalazi optimum, za svega dvije ili tri generacije. Ovakvi su problemi vrlo jednostavni, pa bitnog utjecaja genetskih operatora i skupova parametara nema. Za složenije probleme potrebne su pripadne datoteke, kao što je objašnjeno u poglavlju 5.2. Algoritam je testiran sa datotekama koje sadrže problem veličine 29, 100 i 280 gradova. Najbolji rezultati dobiveni algoritmom prikazani su u tablici (Tablica 6.1).

Tablica 6.1. Najbolji rezultati dobiveni genetskim algoritmom

Broj gradova	Put dobiven algoritmom	Optimalni put	Odstupanje od optimalnog rješenja
29	2020	2020	0.00%
100	21431	20749	3.18%
280	2859	2579	9.79%

Iz tablice je vidljivo da je povećanje broja gradova obrnuto proporcionalno s uspješnosti genetskog algoritma, što je i očekivano. Dok se optimum kod problema sa 29 gradova može dobiti upotrebljavajući više različitih mutacija i križanja, kod problema sa 100 i 280 gradova najboljima su se pokazali mutacija koja ubacuje grad na slučajno mjesto u obilasku i pohlepno križanje.

Algoritam pronalazi optimalno rješenje problema sa 29 gradova u 30-ak posto slučajeva. U protivnome "zaglavi" na nekom lokalnom optimumu, a obzirom da su genetski algoritmi stohastičke metode pretraživanja, nikada se ne može predvidjeti hoće li algoritam "zaglaviti" ili će pronaći optimum. Naravno, ako se koriste loše definirani parametri algoritam će pronaći još lošije rješenje. Prosječni rezultati algoritma dobiveni višestrukim testiranjem optimalnim postavkama prikazani su u

tablici na sljedećoj stranici (Tablica 6.2). Optimalne postavke su pohlepno križanje i mutacija koja ubacuje gradove na slučajno mjesto sa vjerojatnošću od 100%. Veličinu populacije treba postaviti ovisno o broju gradova (za 29 gradova bar 40 jedinki, za 280 gradova bar 200), a veličinu prozora pri selekciji na slučajan odabir ili na oko 20% veličine populacije. Algoritam treba raditi barem nekoliko sati.

Kod problema sa 100 i 280 gradova situacija je složenija. Svako pokretanje algoritma daje slične, ali rijetko kada u potpunosti iste rezultate. Zato su genetski operatori i skupovi parametara testirani na tim problemima. Obzirom da su se pohlepno križanje i mutacija ubacivanjem grada na slučajno mjesto u obilasku pokazali najboljima, oni su korišteni u većini testova. U većini testova korištena je i veličina prozora za selekciju dobivena slučajnim odabirom, jer je kao takva potpuno neovisna o broju jedinki koje se testiraju. Testovi su trajali ili unaprijed definirani broj generacija, ili je postojalo vremensko ograničenje nakon kojeg se rad algoritma prekidao. Rezultati testiranja su prikazani grafovima, na čijoj se *y*-osi nalazi postotak optimalnog rješenja. On predstavlja omjer dobivenog i optimalnog rješenja izražen u postocima, kao što to prikazuje jednadžba (6.1).

$$x = op/dp * 100\% \tag{6.1}$$

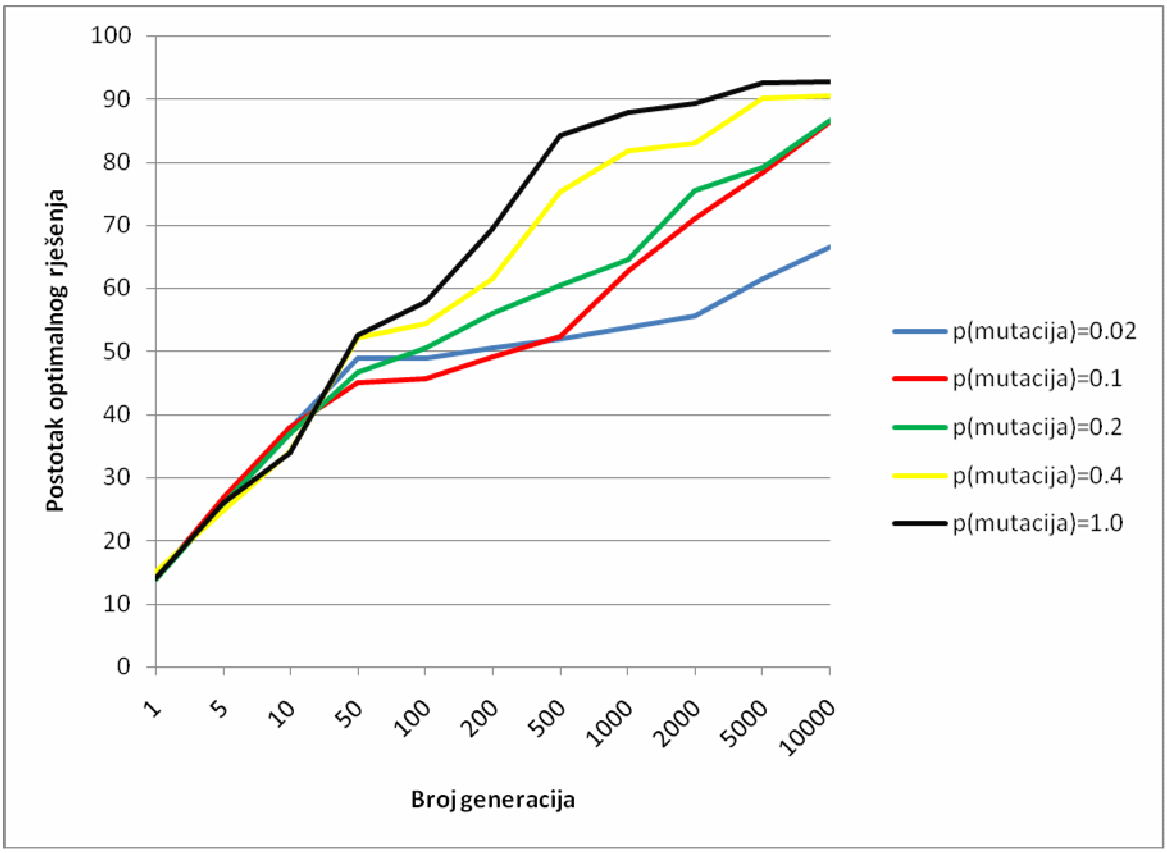
x označava postotak optimalnog rješenja, *op* je duljina optimalnog puta, a *dp* je duljina dobivenog puta.

Tablica 6.2. Prosječni rezultati dobiveni genetskim algoritmom

Broj gradova	Prosječni dobiveni put	Optimalni put	Odstupanje od optimalnog rješenja
29	2030	2020	0.49%
100	21782	20749	4.74%
280	3115	2579	17.21%

6.1 Utjecaj vjerojatnosti mutacije

Vjerojatnost mutacije testirana je na problemu od 100 gradova. Korišteno je pohlepno križanje i mutacija ubacivanjem grada na slučajno mjesto u obilasku. Populacija je sadržavala 100 jedinki, a test je trajao 15 000 generacija. Veličina prozora pri selekciji određivala se slučajnim odabirom. Testirane su vjerojatnosti mutacija od 2%, 10%, 20%, 40% i 100%. Rezultati su prikazani grafom (Slika 6.1).



Slika 6.1. Utjecaj vjerojatnosti mutacije na rad genetskog algoritma

Iz grafa je vidljivo da porastom vjerojatnosti mutacije genetski algoritam nalazi bolja rješenja. Optimalna vjerojatnost mutacije je 1.0, što znači da pri svakoj jedinki dolazi do mutacije. Ovakvo ponašanje je posljedica implementacije testirane mutacije. Mutacija je definirana kao vjerojatnost da se jedan grad ubaci na neko slučajno mjesto. Kod malog broja gradova, manja je vjerojatnost mutacije prilično učinkovita. Međutim, povećanjem broja gradova ubacivanje samo jednog grada je dosta sporo, tj. mijenja se samo mali dio kromosoma. Upotrebom male

vjerojatnosti mutacije jedinke postaju preslične pa je cijeli proces konvergencije algoritma prilično spor. Mutacije su testirane na primjeru od 100 gradova, koji je dosta velik pa veće vjerojatnosti mutacije puno brže pronalaze povoljna rješenja. Slično razmatranje vrijedi i za ostale tri implementirane mutacije.

Izuzetak je početnih 20-ak generacija algoritma, kod kojih su jedinke vrlo različite pa je utjecaj križanja veći od utjecaja mutacije. Zato vjerojatnost mutacije na početku rada algoritma ima vrlo mali utjecaj na konvergenciju. Svakim stvaranjem nove populacije utjecaj križanja je sve manji, a utjecaj mutacije sve veći.

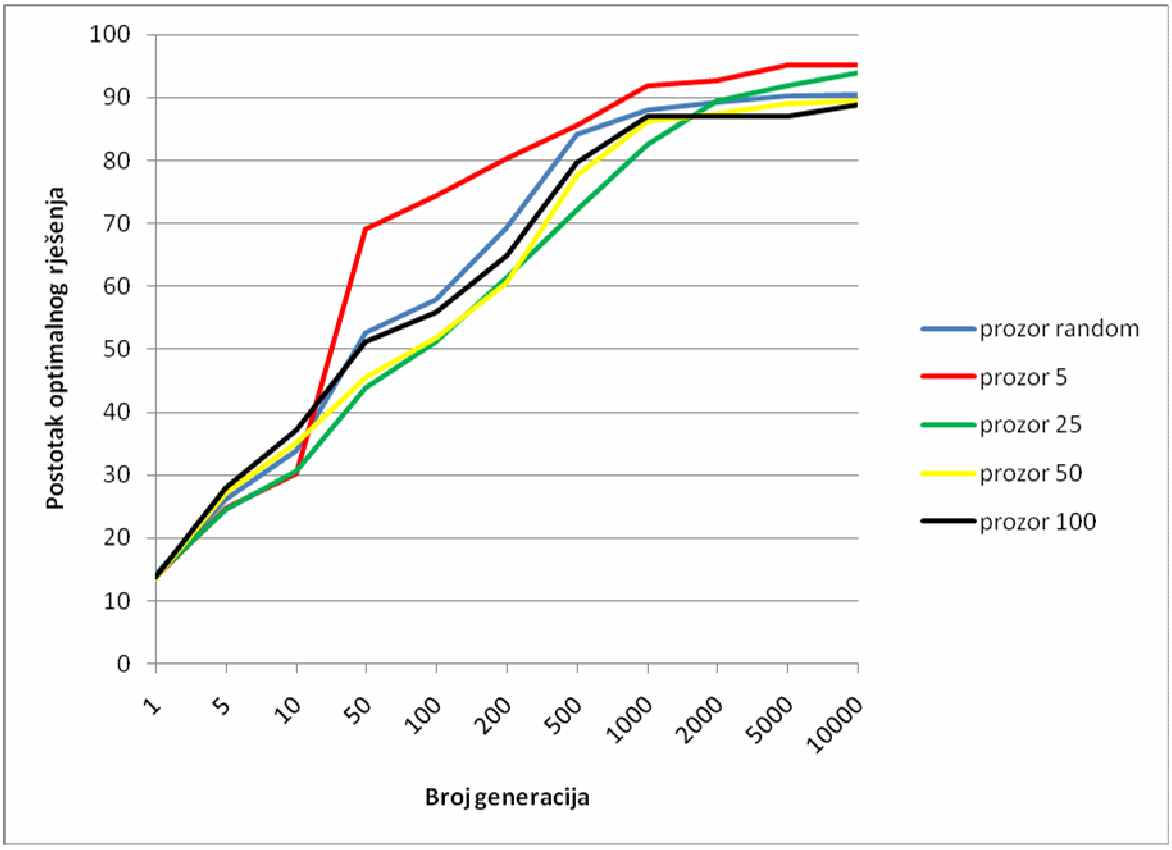
6.2 Utjecaj veličine prozora

Veličina prozora pri selekciji roditelja za sljedeću generaciju algoritma testirana je pohlepnim križanjem i mutacijom koja ubacuje neki grad na slučajno odabrano mjesto u obilasku. Svaki test je pušten kroz 10000 generacija na mapi od 100 gradova. Testirana je slučajna veličina prozora i veličine prozora od 5, 25, 50 i 100 jedinki. Te veličine prozora su 5%, 25%, 50% i 100% veličine populacije, respektivno. Dobiveni rezultati prikazani su slikom na sljedećoj stranici (Slika 6.2).

Iz slike se može zaključiti da pri većoj veličini prozora genetski algoritam počne brže konvergirati. Međutim, to brzo konvergiranje često vodi k lokalnim optimumima pa algoritam brzo prestaje biti učinkovit (već nakon 10-ak generacija). Upotrebom male veličine prozora algoritam sporije počne konvergirati, ali je otporniji na lokalne optimume. Brzina njegovog konvergiranja se smanjuje puno sporije od brzine konvergiranja kod upotrebe većih prozora. Mala veličina prozora znači da algoritam u nekom trenutku ima malen broj jedinki na izboru za roditelje iduće generacije, što objašnjava sporije početno konvergiranje, ali omogućuje nalaženje dobrog genetskog materijala kod lošijih jedinki.

Veličina prozora postavljena na slučajnu vrijednost (*random*) je kombinacija različitih veličina prozora, jer se pri svakoj selekciji određuje ispočetka. Ovakav odabir veličine prozora se pokazao vrlo učinkovitim, jer posjeduje dobre karakteristike i velikih i malih veličina prozora. Algoritam uspješno "izbjegava" lokalne optimume, a brzina konvergencije je prihvatljiva. Pri slučajno odabranoj

veličini prozora korisnik ne mora misliti koliko je veličina prozora velika u odnosu na veličinu populacije, već to algoritam obavlja za njega.

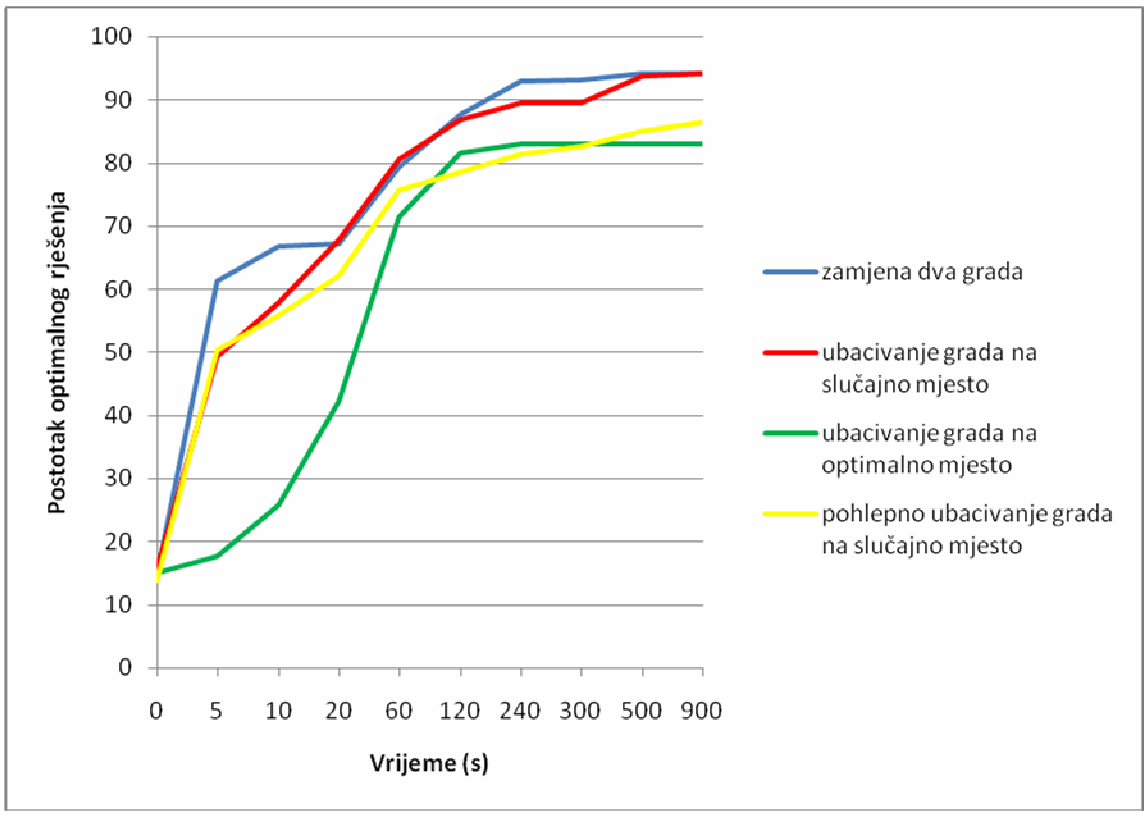


Slika 6.2. Utjecaj veličine prozora na rad genetskog algoritma

6.3 Utjecaj odabira mutacije

Za rješavanje problema trgovačkog putnika implementirane su četiri mutacije. One su testirane testom trajanja 15 minuta i uz korištenje pohlepnog križanja. Trajanje testa je određeno minutama, a ne brojem generacija zbog toga što različite mutacije rade vrlo različitim brzinama. Tako je mutacija koja ubacuje grad na optimalno mjesto 20-ak puta sporija od mutacije koja ubacuje grad na slučajno odabrano mjesto, a razlika bi bila još i veća kada bi se koristio veći broj gradova. Testirani problem se sastojao od 100 gradova, a veličina populacije je iznosila 100 jedinki. Vjerojatnost mutacije je postavljena na 100% da se dobije čim bolja veza

mutacije i napretka genetskog algoritma. Rezultati su prikazani na slici ispod (Slika 6.3).



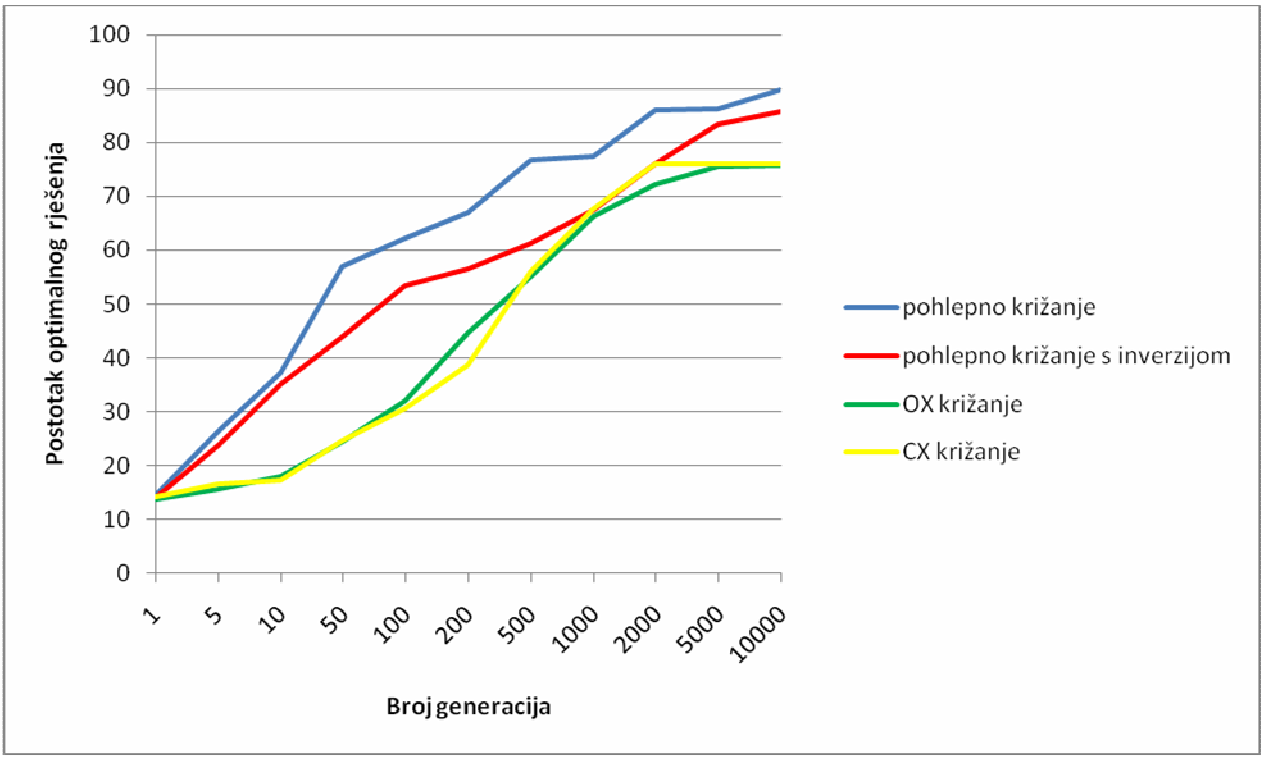
Slika 6.3. Utjecaj odabira mutacije na rad genetskog algoritma

Najboljim mutacijama pokazale su se mutacija koja mijenja mjesto dva grada u obilasku i mutacija koja ubacuje grad na slučajno mjesto u obilasku. Druge dvije mutacije imaju dosta problema s lokalnim optimumima. Mutacija koja pohlepno ubacuje grad na slučajno mjesto u obilasku konvergira uvijek, ali dosta često zastajkuje na lokalnim optimumima.

Mutacija koja ubacuje grad na optimalno mjesto je daleko najlošija mutacija, jer ne radi svoj osnovni posao – ne čini populaciju raznolikom. Ako se koristi ta mutacija, cijela populacija vrlo brzo postane preslična što uzrokuje gubitak dobrog genetskog materijala lošijih jedinki. Ova je mutacija za 15 minuta prošla tek 400 generacija, dok su ostale mutacije prošle kojih 8 ili 9 tisuća. Kada bi ova mutacija imala vremena za proći nekoliko tisuća generacija, situacija opet ne bi bila bolja jer bi algoritam svejedno ostao zaglavljen na lokalnom optimumu.

6.4 Utjecaj odabira križanja

Za razliku od mutacija, sva križanja rade podjednako brzo pa se njihovo testiranje ograničilo na 10 000 generacija. Koristila se mutacija koja ubacuje grad na slučajno mjesto u obilasku, a njena je vjerojatnost postavljena na 80% da se osigura dovoljna raznolikost jedinki. Populacija je imala 100 jedinki, a razmatrani problem je bio opsega 100 gradova. Dobiveni rezultati prikazani su grafom (Slika 6.4).



Slika 6.4. Utjecaj odabira križanja na rad genetskog algoritma

Najbolje križanje od implementiranih je pohlepno križanje. Konvergencija algoritma kada se upotrebljava pohlepno križanje je izrazito brza, pa se već nakon samo 500 generacija algoritam približio na 75% optimalnog rješenja. Sva ostala križanja za to su vrijeme tek na nekih 55-60%, koje pohlepno križanje dosegne za samo 50 generacija. Iako brzo konvergira, ovo križanje nije ništa manje otporno na lokalne optimume od ostalih križanja.

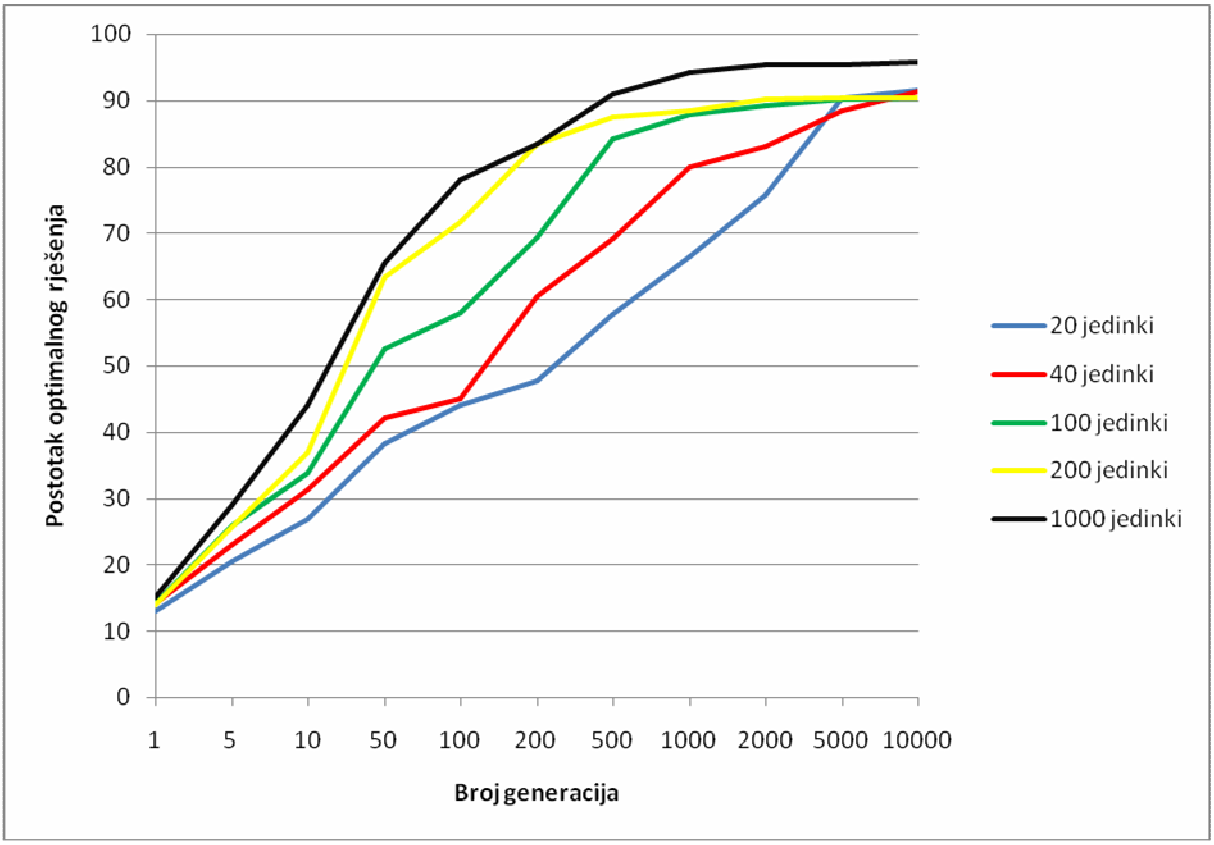
Modificirano pohlepno križanje (pohlepno križanje s inverzijom) se pokazalo kao drugo najbolje križanje. Iako konvergira dosta sporije nego čisto pohlepno križanje, ovo je križanje najotpornije na lokalne optimume (nagib pravca jedino kod ovog križanja niti u jednom trenutku nije jednak nuli), jer se u svakoj generaciji dio rute invertira. Invertiranje rute je ujedno i najveći nedostatak ovog križanja, jer uzrokuje gubitke nekih rješenja koja bi mogla biti zadržana da nema invertiranja.

OX i CX križanje konvergiraju dosta sporije i daju slabije rezultate od pohlepnih križanja. Lokalni optimumi im oduzimaju previše vremena (tako je CX križanje od generacije 2000 do kraja testa zaglavljeno na jednom od njih). Povećanjem broja generacija njihovi bi se rezultati popravili, ali ne bi dostigli rezultate dobivene pohlepnim križanjem. Izvode se nešto brže od pohlepnog križanja.

6.5 Utjecaj veličine populacije

Veličina populacije genetskog algoritma testirana je na problemu od 100 gradova, koristeći pohlepno križanje i mutaciju koja ubacuje grad na slučajno mjesto u obilasku. Vjerojatnost mutacije je postavljena na 100%, a veličina prozora na slučajan odabir. Testirane su veličine populacije od 20, 40, 100, 200 i 1000 jedinki, a svaki je test trajao 10000 generacija. Graf na sljedećoj stranici prikazuje rezultate dobivene testiranjem (Slika 6.5).

Pri testiranju najbolji rezultati su dobiveni sa najvećom populacijom, što je očekivano. Međutim, taj je test ujedno i trajao daleko najdulje pa je prikladniji za probleme sa većim brojem gradova. Povećanjem brojnosti populacije ubrzava se konvergencija algoritma. Populacija veličine 2000 je za 200 generacija bila na preko 80% optimalnog rješenja, dok je populacija od 20 jedinki bila tek na 45%. Povećanjem broja generacija rezultati manjih populacija dostižu rezultate većih. Tako su nakon 10000 generacija populacije od 20, 40, 100 i 200 jedinki imale gotovo jednaku najbolju jedinku. Povećanjem broja zadanih gradova, za uspješan rad algoritma potrebno je povećati veličinu populacije.



Slika 6.5. Utjecaj veličine populacije na rad genetskog algoritma

7. Zaključak

Problem trgovačkog putnika može se riješiti genetskim algoritmima. Genetski algoritmi stohastičkim pretraživanjem mogu naći rješenje svega 2 ili 3% lošije od optimalnog čak i kada broj gradova iznosi nekoliko stotina tisuća ([6]). Niti jedan standardni algoritam nije toliko uspješan na problemima takve veličine.

Problemi određivanja rasporeda uz vremena postavljanja strojeva se svode na problem trgovačkog putnika, pa se rješavaju istim metodama. Vrijeme potrebno da se stroj pripremi za obradu nekog posla može se prikazati na isti način kao i put koji trgovački putnik mora prijeći ako želi doći u neki grad.

Uspješnost rada genetskog algoritma ovisi o njegovim parametrima i o izboru mutacije i križanja. Manja populacija ima veće šanse da "zaglavi" na nekom lokalnom optimumu, ali zato iterira kroz generacije puno brže nego veća populacija. Veća populacija je pogodnija za probleme s većim brojem gradova. Veća vjerojatnost mutacije povećava raznolikost jedinki u populaciji, te sprječava problem prerane konvergencije rješenja. Veličina prozora za selekciju je najučinkovitija ako se određuje slučajnim odabirom ili ako iznosi 5-10% veličine populacije.

Implementirani algoritam bi se mogao poboljšati ako bi se dodao prikaz rješenja u obliku matrice. Taj je prikaz dobar jer se informacija ne prenosi samim imenom grada kao kod vektorskog prikaza, nego udaljenostima kojima su gradovi međusobno povezani.

8. Literatura

[1] Golub, M. Genetski algoritam, prvi dio, http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf, 10.05.2008.

[2] Wikipedia, Genetic algorithm, http://en.wikipedia.org/wiki/Genetic_algorithm, 14.05.2008.

[3] Wikipedia, Evolutionary algorithm, http://en.wikipedia.org/wiki/Evolutionary_algorithms, 14.05.2008.

[4] Evolutonary algorithm, http://www.geatbx.com/docu/algindex.html, 07.06.2008.

[5] Wikipedia, Crossover (genetic algorithm), http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm), 15.05.2008.

[6] Wikipedia, Traveling salesman problem, http://en.wikipedia.org/wiki/Traveling_salesman_problem, 07.06.2008.

[7] Xu, Q, Introduction to Job Shop Scheduling Problem, 30.08.2001, http://www.cs.umbc.edu/671/fall01/class-notes/jobshop.ppt, 26.05.2008.

[8] Garrido, A., Salido, M. A., Barber F., Lopez M. A., Heuristic Methods for Solving Job-Shop Scheduling Problems, http://citeseer.ist.psu.edu/rd/80924777%2C402791%2C1%2C0.25%2CDownload/http%3AqSqqSqwww-is.informatik.uni-oldenburg.deqSq~sauerqSqpuk2000qSqpapersqSqGarrido_Job-Shop.pdf, 26.05.2008.

[9] Bryant, K., Genetic Algorithms and the Traveling Salesman Problem, http://www.math.hmc.edu/seniorthesis/archives/2001/kbryant/kbryant-2001-thesis.pdf, 26.05.2008.

9. Sažetak

Problem trgovačkog putnika (TSP) je NP-težak problem diskretne i kombinatorne optimizacije. Na njega se svode mnogi problemi, uključujući problem raspoređivanja poslova uz vremena postavljanja strojeva.

U javi je napisan program koji rješava problem trgovačkog putnika genetskim algoritmom. Genetski algoritmi su stohastičke metode pretraživanja koje oponašaju prirodni tijek biološke evolucije. Algoritam je testiran na problemima različitih veličina, a koristili su se različiti parametri (vjerojatnost mutacije, veličina prozora za selekciju, veličina populacije, različita križanja i mutacije). Dobiveni rezultati su komentirani i izvedeni su zaključci .

9.1 Abstract

Traveling Salesman Problem (TSP) is NP-hard problem in computational complexity theory. Many problems can be reduced to it, including the Job Scheduling Problems with setup times.

Genetic algorithm in java was written to solve the TSP. Genetic algorithms are search techniques used in computing to find exact or approximate solutions to optimization and search problems. The algorithm was tested on different size problems and using different parameters (mutation probability, selection window size, population size, different mutations and crossovers). Obtained results are commented and conclusions are reached.

9.2 Ključne riječi

Problem trgovačkog putnika (*Traveling Salesman Problem, TSP)*, problem raspoređivanja poslova uz vremena postavljanja strojeva (*Job Scheduling Problems with setup times*), genetski algoritam (*Genetic Algorithm, GA*)