SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 48

Analiza učinkovitosti prikaza rješenja u okruženju za evolucijsko računanje

Rene Huić

Zagreb, srpanj 2010.

Sadržaj

Uvod

Genetski algoritmi su se već odavno dokazali, zbog čega se i danas koriste u raznim granama znanosti. Kako bi se olakšalo pisanje genetskih algoritama za razne optimizacijske probleme, izrađuje se programsko okruženje za evolucijske algoritme, *Evolutionary computation framework* (ECF), u programskom jeziku Java. Okruženje će sadržavati razne razrede i metode implementirane sa ciljem olakšavanja i smanjivanja pisanja koda prilikom programiranja evolucijskog algoritma.

U sklopu ovog rada su, između ostaloga, implementirana tri prikaza jedinke: binarni niz, permutirani niz i prikaz broja s pomičnom točkom. Uz to su implementirani i razni operatori križanja i mutacija, prilagođeni pojedinom prikazu. Glavni cilj ovog rada je ispitati i analizirati učinkovitost implementiranih funkcionalnosti na različitim kombinatornim i kontinuiranim optimizacijskim problemima.

1. Genetski algoritam

1.1. Osnove genetskog algoritma

Genetski algoritam je heuristička metoda koja služi za rješavanje optimizacijskih problema. Kombinira slučajno i usmjereno pretraživanje prostora rješenja. Princip rada genetskog algoritma analogan je evolucijskom procesu u prirodi. Prirodnom selekcijom se biraju jedinke koje će preživjeti i stvoriti potomstvo, tj. preživljavaju bolje, jače i sposobnije jedinke, te je veća vjerojatnost da će se te jedinke pariti kako bi nastali potomci koji će tvoriti slijedeću generaciju. Isto tako, potomci bi trebali biti bolji od roditelja baš zbog toga jer su nastali od sposobnijih jedinki. Na taj način populacija napreduje i sve se bolje prilagođava okolini. Preživljavanjem najboljih jedinki, genetski algoritam se usmjerava prema optimumu i pretražuje samo one dijelove prostora rješenja koji sadržavaju dobra rješenja. [1]

Prva generacija se slučajno odabire. Svaka slijedeća generacija se popunjava križanjem najboljih jedinki iz trenutne populacije. Svaka jedinka predstavlja jednu kombinaciju ulaznih parametara, kodiranih na primjereni način. Podaci koje sadrži jedinka predstavljaju njen genetski materijal. Jednako kao i u prirodnoj selekciji, selekcijom u genetskom algoritmu se biraju jedinke prema svom genetskom materijalu: one sa kvalitetnijim genima (tj., one koje daju bolje rezultate) će imati veću šansu za preživljavanje, te će dobiti priliku da prenesu svoj genetski materijal na potomstvo. Na taj način populacija u genetskom algoritmu napreduje, dajući sve bolja rješenja za problem koji se optimira. Proces selekcije, reprodukcije i manipulacije genetskim materijalom se ponavlja sve dok nije zadovoljen uvjet zaustavljanja genetskog algoritma, tj. sve dok se ne dođe do zadovoljavajućeg rezultata. Kako se ne bi dogodilo da se završi u nekom lokalnom optimumu, genetski algoritam koristi i operator mutacije koji simulira „nezgodu“ u stvarnoj evoluciji, jedinku koja se ne uklapa u populaciju. Razlog ovome je slučajan skok

na neko još neistraženo područje prostora rješenja u kojem se možda nalazi bolje rješenje od onih trenutno sadržanih u populaciji. Ako se to ne dogodi, već se dođe do lošijeg rješenja, selekcija će se pobrinuti da se to rješenje brzo zamijeni s onim boljima.

1.2. Operatori genetskog algoritma

Jedinka (ili kromosom) predstavlja jedno potencijalno rješenje problema koji se rješava, kodirano na određeni način i u njemu su sadržani svi podaci koji obilježavaju jednu jedinku. Budući da način kodiranja rješenja može znatno utjecati na učinkovitost genetskog algoritma, pravilan izbor strukture podataka koja će se koristiti predstavlja vrlo važan korak u implementaciji genetskog algoritma. Za odabrani način prikaza rješenja potrebno je definirati i genetske operatore, pri čemu je bitno da operatori ne stvaraju nove jedinke koje predstavljaju nemoguća rješenja, jer se time mnogo gubi na učinkovitosti algoritma. Postoji mnogo različitih prikaza koji se koriste u genetskim algoritmima za rješavanje različitih problema. U optimizaciji funkcija se obično koriste binarni (kromosom je predstavljen nizom bitova) i prikaz realnim brojem, u raznim kombinatoričkim problemima se koriste nizovi brojeva, polja, binarna stabla, i sl. [2]

Rad genetskog algoritma može se najbolje opisati sa slijedećih pet koraka[4]:

1. kodiranje jedinki
2. evaluacija dobrote jedinki
3. selekcija
4. križanje (Slika 1.1)
5. mutacija
6. dekodiranje rješenja

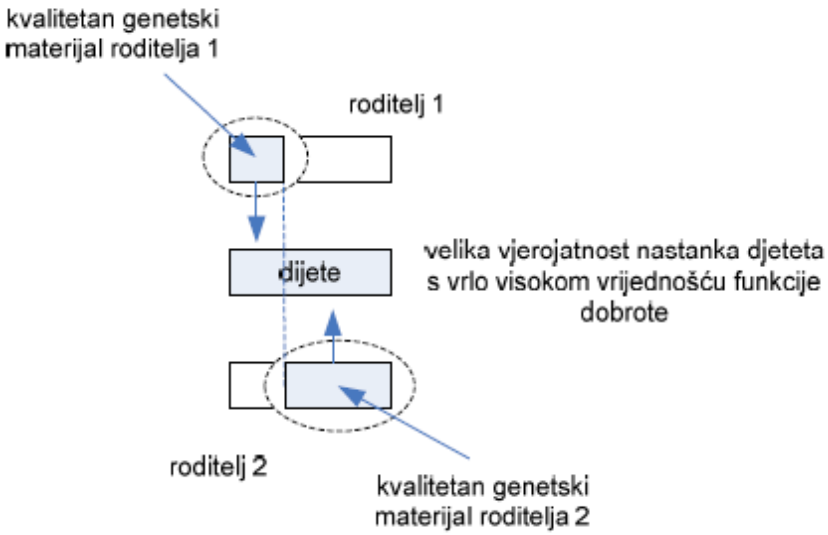
Kodiranje jedinki je početni korak u dizajnu genetskog algoritma. Svaka jedinka mora biti jedinstveno predstavljena u populaciji. Nakon što se odredi kako jedinke izgledaju, najčešće se slučajnom metodom generira početna populacija u kojoj se za svaku jedinku računa (evaluira) dobrota.

Dobrota je mjera koja genetskom algoritmu govori je li neko rješenje bliže ili dalje od optimuma u usporedbi s ostalim članovima populacije. Oni članovi koji imaju bolju dobrotu su bliži optimumu i za njih je veća vjerojatnost križanja. Genetski algoritmi ne očekuju nikakve dopunske informacije o problemu optimiranja osim funkcije dobrote. Ona ne mora biti ni neprekidna, ni derivabilna, pa čak niti egzaktno definirana, isto tako ne mora biti jednaka funkciji cilja. Kvaliteta jedinki mjeri se pomoću funkcije cilja f. Neki postupci selekcije zahtijevaju da funkcija cilja ne može biti negativna te da ne poprima samo velike, približno jednake, vrijednosti. Zbog toga se obično u svakoj iteraciji obavlja translacija funkcije cilja, tj. vrijednosti funkcije cilja pojedine jedinke oduzima se najmanja vrijednost funkcije cilja u cijeloj populaciji. Rezultat je funkcija dobrote d koja se računa prema jednadžbi (1) [5]:

$$D = f - f_{min}(t), \tag{1}$$

gdje je $f_{min}(t)$ najmanja vrijednost funkcije cilja u generaciji t.

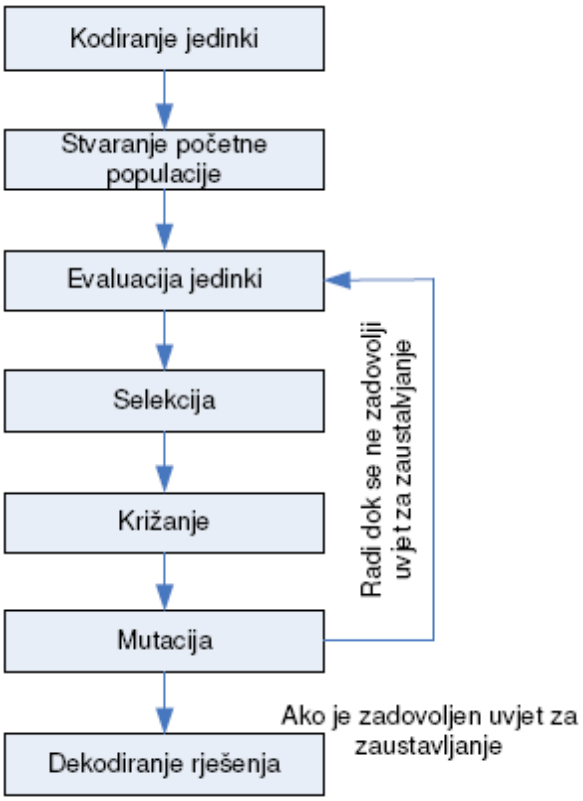
Križanje je postupak u kojem se odabiru dvije jedinke (roditelji) iz populacije te se njihovi dijelovi međusobno kombiniraju kako bi nastale nove jedinke (djeca). Novonastale jedinke se umeću u slijedeću generaciju umjesto onih lošijih.



Slika 1.1 *Križanje*

Neke od novonastalih jedinki se još dodatno i mutiraju kako bi se proširio prostor rješenja koji se pretražuje, i da se izbjegne slučaj zaglavljenja u lokalnom optimumu. Na primjer, ako se uzme da je jedna jedinka kodirana kao niz znakova, mutacija može biti zamjena nekog slučajno odabranog znaka drugim slučajno odabranim znakom.

Ovaj postupak se ponavlja sve dok se ne dostigne neki unaprijed definirani kriterij zaustavljanja nakon kojeg se najbolje postignuto rješenje dekodira. To rješenje je najbliže optimumu ili je optimum (Slika 1.2).



Slika 1.2 *Prikaz rada genetskog algoritma*

Za svaku vrstu problema potrebno je dizajnirati posebni genetski algoritam ili je potrebno prilagoditi problem nekom već postojećem genetskom algoritmu. Evoluciju rješenja korištenjem genetskog algoritma moguće je usmjeravati podešavanjem parametara koje neki genetski algoritam može imati, a može se i dizajnirati algoritam tako da se tokom rada algoritma parametri mogu sami podešavati, bez uplitanja korisnika, kako bi se promijenilo ponašanje algoritma. O parametrima ovisi koliko će se vremena potrošiti na evoluciju rješenja, kojom će se brzinom algoritam usmjeravati prema potencijalnom optimumu, hoće li pretraživati veći ili manji dio prostora rješenja, itd. Skup parametara koji za jedan genetski algoritam i jedan problem daje kvalitetne rezultate, za neki drugi problem ne mora dati jednako kvalitetne rezultate.

Veličina populacije, broj generacija (iteracija) i vjerojatnost mutacije su tri standardna parametra koje koriste svi genetski algoritmi prilikom evolucije rješenja

i o njima najviše ovisi način ponašanja genetskog algoritma. Postoji još različitih parametara koje koriste genetski algoritmi, ali tu o korištenju takvog pojedinog parametra, ovisi problem i način na koji se taj problem rješava genetskim algoritmom. Neki od tih „nestandardnih“ operatora može biti duljina kromosoma ili vjerojatnost križanja[5].

Veličina populacije – parametar koji direktno utječe na kvalitetu dobivenih rješenja, ali isto tako i ovisno o selekciji može imati velik utjecaj na duljinu izvođenja genetskog algoritma. Npr. kod selekcija koje će ovdje biti kasnije objašnjene, veličina populacije uvelike utječe na duljinu izvođenja, dok kod npr. turnirske selekcije, veličina populacije ne utječe direktno na vrijeme izvođenja genetskog algoritma. Pokazalo se da manja veličina populacije je učinkovita ako se uzme manji broj generacija, dok bi se za veću veličinu populacije trebalo povećati broj generacija.

Broj generacija (iteracija) – parametar koji direktno utječe i na kvalitetu dobivenih rješenja i na duljinu izvođenja genetskog algoritma. Logički je da veći broj generacija, daje više vremena genetskog algoritmu da pronađe optimum zadanog problema, no kod dizajniranja genetskog algoritma za teže probleme, vrijeme se uzima kao jedan od važnih faktora. Naravno da se uvijek pokušava naći optimum rješenja za problem, ali je nekada cijena za vrijeme koje bi trebalo genetskom algoritmu jednostavno prevelika, pa se zato pokušava pronaći zadovoljavajuće rješenje u što kraćem vremenu. Zadovoljavajuće rješenje je ono rješenje koje je dovoljno blizu optimumu, a za koje je potrebno mnogo manje vremena kako bi se do njega došlo.

Vjerojatnost mutacije – jedan od najvažnijih parametara kod genetskih algoritama, jer direktno utječe na to da li će doći kod određene jedinke ili kod određenog bita unutar jedinke do mutacije, a sama mutacija, kao što je već spomenuto, radi slučajne skokove algoritma po prostoru rješenja, što omogućuje izbjegavanje zaglavljivanja algoritma u lokalnim optimumima i proširivanje

područja pretrage na još neistražene dijelove prostora rješenja. Moguće je da će dobivena rješenja biti lošija od originalnih, ali zbog toga ta lošija rješenja imaju manju vjerojatnost ulaska u daljnje reprodukcije čime se ostavlja prostor za napredovanje u pravom smjeru. Genetski algoritam je dosta osjetljiv na promjene ovog parametra, pa je dosta bitno pažljivo odabrati njegove vrijednosti.

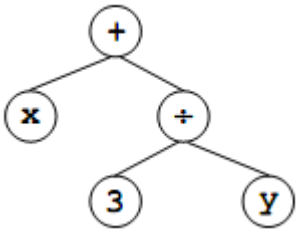
Parametri genetskog algoritma predstavljaju važnu komponentu rada algoritma. Mnogi su znanstvenici pokušavali odrediti skupove parametara koji bi bili optimalni za velik broj problema. No, nažalost se pokazalo da svaki problem ima svoj skup optimalnih parametara te da nije moguće odrediti parametre koji bi bili dobri za širok spektar problema. Ali se uspjelo doći do zaključaka o tome koji su parametri dobri za početak pretraživanja genetskog algoritma i od kuda treba krenuti kako bi se odredili optimalni parametri za pojedini genetski algoritam. Postojali su i pokušaji da se odrede relacije pomoću kojih bi se mogla izračunati koja je poželjna vrijednost trećeg parametra ako su poznata prva dva. Nemoguće je odrediti skupove parametara koji bi bili optimalni za rješavanje velikog skupa problema, ali postoje određene međuzavisnosti i pravilnosti između pojedinih parametara:

1) za veće populacije je dobro povećati broj generacija i smanjiti vjerojatnost mutacije
2) veći broj generacija daje najčešće bolja rješenja
3) jednako kvalitetna rješenja se mogu dobiti sa različitim skupovima parametara

Osim definiranja fiksnih parametara na početku genetskog algoritma, tj. parametara koji se tijekom rada genetskog algoritma ne mijenjaju, moguće je dizajnirati genetski algoritam s dinamičkim parametrima, tj. s parametrima koji se sami podešavaju sa svrhom da promijene ponašanje genetskog algoritma. Npr. ako se najbolja jedinka ne promijeni kroz određen broj generacija, tada sam algoritam mijenja parametre kao npr. veličinu populacije i vjerojatnost mutacije kako bi se izbjegla mogućnost zaglavljivanja u nekom lokalnom optimumu.

1.3. Prikaz rješenja

Uz same parametre genetskog algoritma, prikaz same jedinke, tj. prikaz rješenja genetskog algoritma je isto vrlo važna komponenta kod izrade genetskog algoritma. Budući da je jedinka zapravo potencijalno rješenje problema, koje je ovisno o prikazu same jedinke kodirano na određeni način, vrlo je važno odrediti pravilnu strukturu prikaza rješenja, jer uvelike utječe na samu učinkovitost genetskog algoritma. Jednako tako je potrebno odrediti pravilne genetske operatore za određeni prikaz rješenja, te ih prilagoditi da ne stvaraju jedinke koje bi predstavljale nemoguća rješenja, jer inače genetski algoritam uvelike gubi na učinkovitosti.



Slika 1.3 ***Prikaz rješenja binarnim stablom***

Postoji mnogo različitih načina prikaza rješenja, pri čemu prikaz rješenja uvelike ovisi o problemu koji se rješava genetskim algoritmom. U raznim kombinatoričkim problemima se koriste binarna stabla, nizovi brojeva ili polja kao prikaz rješenja, dok se kod optimiziranja funkcija najčešće koristi binarni prikaz (jedinka je prikazana nizom bitova).

(0 0 1 0 1 1 1 0 0)

Slika 1.4 *Binarni prikaz rješenja*

2. Ostvarenje programskog okruženja za evolucijske algoritme

Prema uzoru na C++ implementaciju okruženja ECF, implementirana je jednostavnija inačica u Java programskom jeziku. Za razliku od C++ verzije, kod Java programskog jezika se ne koriste direktno pokazivači, te ne postoji direktni pristup memoriji. Rukuje se referencama, koje se mogu zamisliti kao „pametni pokazivači“ (engl. *smart pointers*) jer pružaju dodatnu funkcionalnost, npr. automatsko prikupljanje smeća (engl. *automatic garbage collection*), granične provjere (engl. *bounds checking*) i sl. [8]. Korištenjem pametnih pokazivača se smanjuje nerazumijevanje pravog korištenja pokazivača kod neiskusnih programera, gubitka memorije (engl. *memory leak*) i općenito se programer koji radi u Java programskom jeziku mora manje zamarati direktnim rukovođenjem memorijom, jer se za to brine Java virtualni stroj.

ECF u Javi je dodatno nadgrađen implementacijom tri nova prikaza rješenja (genotipa): kao binarni niz, permutacijski niz i prikaz broja s pomičnom točkom. Za svaki od ta tri genotipa je implementiran i niz operatora mutacija i križanja, koji će biti navedeni i opisani kasnije u radu. Implementirane su i tri vrste operatora zaustavljanja:

1. Operator koji zaustavlja algoritam nakon određenog broja generacija (*term.maxgen*)
2. Operator koji zaustavlja algoritam nakon određenog broja sekundi (*term.maxtime*)
3. Operator koji zaustavlja algoritam ako se dobrota najbolje jedinke nije promijenila određen broj generacija (*term.stagnation*)

Postavke operatora se postavljaju u konfiguracijskoj datoteci, kao što se može vidjeti na slici Slika 2.1. Postavke u konfiguracijskoj datoteci vezane uz operatore križanja i mutacije predstavljaju vjerojatnost izvršavanja nekog operatora na zadanom genotipu, osim u par iznimaka gdje neki operator koristi dodatni parametar.

```
<Entry key="term.maxgen">10000</Entry>
<Entry key="term.maxtime">60</Entry>
<Entry key="term.stagnation">250</Entry>
```

Slika 2.1 Parametri operatora za zaustavljanje u konfiguracijskoj datoteci

2.1. Binarni niz

Prikaz jedinke kao binarnog niza se sastoji od jednog niza bitova, pri čemu svaki bit poprima vrijednost 0 ili 1, kao što se vidi na slici Slika 1.4. Binarni niz ima parametar dimenzionalnost, što znači da je moguće prikazati više od jednog broja u jednom binarnom nizu. Dimenzionalnost (*dimension*) određuje koliko će tih brojeva prikazivati jedan binarni niz. Parametar preciznost (*precision*) se odnosi na koju decimalu se žele prikazati brojevi. Maksimalna preciznost je 16. Postoje dva parametra koja se odnose na gornju (*ubound*) i donju (*lbound*) granicu intervala u kojem se nalazi svaki od brojeva prikazanim unutar binarnog niza. Parametar koji se odnosi na zaokruživanje realne vrijednosti prikazanih brojeva unutar binarnog niza je *rounding*. Na slici Slika 2.2 se može vidjeti isječak iz konfiguracijske datoteke gdje su definirani prije navedeni parametri.

```
<Entry key="lbound">-100</Entry>
<Entry key="ubound">100</Entry>
<Entry key="precision">15</Entry>
<Entry key="dimension">20</Entry>
<Entry key="rounding">0</Entry>
```

Slika 2.2 Isječak iz konfiguracijske datoteke za binarni prikaz

Parametri iz konfiguracijske datoteke koji se odnose na pojedini operator križanja ili mutacije se mogu vidjeti na slici Slika 2.3. Na koji se točno operator križanja ili mutacije odnosi parametar se može vidjeti u tablici Tablica 2.1.

```
<Entry key="crx.onepoint">1</Entry>
<Entry key="crx.uniform">1</Entry>

<Entry key="mut.mix">0.1</Entry>
<Entry key="mut.simple">0.1</Entry>
<Entry key="simple.bitprob">0.5</Entry>
```

Slika 2.3 Parametri iz konfiguracijske datoteke za operatore križanja i mutacije

Tablica 2.1 Prikaz parametra i pripadajućeg operatora za binarni genotip

crx.onepoint	Križanje s jednom točkom prekida
crx.uniform	Uniformno križanje
mut.mix	Miješajuća mutacija
mut.simple, simple.bitprob	Jednostavna mutacija

2.1.1. Operatori križanja

Implementirana su dva operatora križanja specifična za binarni prikaz rješenja:

a) **Križanje s jednom točkom prekida** – gradi dijete tako da se nasumično odabere jedna točka u binarnom nizu, koja se naziva točkom prekida. Zatim se nasumično odabere jedan od roditelja iz kojeg se kopiraju svi bitovi do točke prekida, dok se od drugog roditelja kopiraju svi bitovi od točke prekida. Npr. dva roditelja (pri čemu je točka prekida označena sa „|“)

r1 = (0 1 0 | 1 1 1 0 0)

r2 = (1 1 1 | 0 1 1 1 1)

bi proizvela dijete tako da se prvo nasumično odredi iz kojeg će se roditelja prvo kopirati bitovi. Uzmimo da je odabran roditelj 2, tada će dijete biti sljedeće:

d1 = (1 1 1 | 1 1 1 0 0)

b) **Jednoliko (uniformno) križanje** – gradi dijete tako da se svi bitovi jednaki kod oba roditelja kopiraju dok se za različite bitove, nasumično kreiraju novi. Npr. dva roditelja:

r1 = (0 1 0 1 1 1 0 0)

r2 = (1 1 1 0 1 1 1 1)

bi proizvela dijete tako da se prvo odrede koji su jednaki bitovi kod roditelja i onda kopiraju u dijete:

d1 = (x 1 x x 1 1 x x)

a zatim se nasumično odrede preostali bitovi kod djeteta:

d1 = (0 1 0 0 1 1 1 1)

2.1.2. Operatori mutacija

Za operatore mutacija su implementirane dvije verzije:

a) **Jednostavna mutacija** – s obzirom na vjerojatnost mutacije se gleda svaki bit binarnog niza, te se mijenja (1 u 0, 0 u 1) s obzirom na vjerojatnost mutacije bita. Npr. ako imamo jedinku:

j = (0 1 0 1 1 1 0 0)

bi u slučaju zadovoljavanja vjerojatnosti (to znači da je nasumično generiran broj manji od zadanog broja koji predstavlja vjerojatnost) mutacije i ako bitovi 3, 4 i 7 zadovoljavaju vjerojatnost mutacije bita tada će mutirana jedinka izgledati:

mj = (0 1 1 0 1 1 1 0)

b) **Miješajuća mutacija** – određuje dvije točke na binarnom nizu i bitove između tih točaka preslaguje slučajnim odabirom, ali tako da ukupan broj jedinica i ukupan broj nula između tih točaka ostanu isti kao u roditeljskom binarnom nizu. Npr. ako imamo jedinku:

j = (0 1 | 0 1 1 1 0 | 0)

bi u slučaju zadovoljavanja vjerojatnosti mutacije mutirana jedinka izgledala kao:

mj = (0 1 | 1 1 1 0 0 | 0)

2.2. Prikaz broja s pomičnom točkom

Prikaz jedinke kao prikaz broja s pomičnom točkom (engl. *Floating Point - FP*) se sastoji od jednog broja s pomičnom točkom. Ovaj prikaz ima mogućnost višedimenzionalnosti koja se može podesiti u konfiguracijskog datoteci. Osim dimenzionalnosti, moguće je podesiti interval u kojem se nalazi broj s pomičnom točkom, koji se podešava preko dva parametara, *ubound* i *lbound*, na isti način kao i kod binarnog prikaza. Primjer parametara iz konfiguracijske datoteke se može vidjeti na slici Slika 2.4

```
<Entry key="lbound">-100</Entry>
<Entry key="ubound">100</Entry>
<Entry key="dimension">20</Entry>
```

Slika 2.4 Isječak iz konfiguracijske datoteke a prikaz broja s pomičnom točkom

Parametri iz konfiguracijske datoteke koji se odnose na pojedini operator križanja ili mutacije se mogu vidjeti na slici Slika 2.5. Na koji se točno operator križanja ili mutacije odnosi parametar se može vidjeti u tablici Tablica 2.2.

```
<Entry key="crx.arithmeticaluniform">1</Entry>
<Entry key="arithmeticaluniform.a">0.3</Entry>
<Entry key="crx.onepoint">1</Entry>
<Entry key="crx.uniform">1</Entry>

<Entry key="mut.simple">1.0</Entry>
<Entry key="mut.nonuniform">0.1</Entry>
<Entry key="nonuniform.b">5</Entry>
```

Slika 2.5 Parametri iz konfiguracijske datoteke za operatore križanja i mutacije

Tablica 2.2 Prikaz parametra i pripadajućeg operatora za FP genotip

crx.arithmeticaluniform, arithmeticaluniform.a	Aritmetičko uniformno križanje
crx.onepoint	Križanje s jednom točkom prekida
crx.uniform	Uniformno križanje
mut.simple	Jednostavna mutacija
mut.nonuniform, nonuniform.b	Neuniformna mutacija

2.2.1. Operatori križanja

Implementirana su tri operatora križanja specifična za prikaz broja s pomičnom točkom:

a) **Aritmetičko uniformno križanje** – gradi dijete kao linearnu kombinaciju njegovih roditelja. S obzirom da je parametar a konstantan (definira se u konfiguracijskoj datoteci), križanje je uniformno. Npr. ako imamo roditelje:

r1 = (1.23 4.65 9.33)

r2 = (-3.43 7.35 1.11)

tada se dijete gradi kao linearna kombinacija preko formule: $s_d = a \cdot s_{r1} + (1-a) \cdot s_{r2}$, pri čemu je a = 0.3, te bi dijete za zadani primjer, izgledalo kao:

d = (-2.03 6.54 3.57)

b) **Križanje s jednom točkom prekida –** križanje za prikaz broja s pomičnom točkom funkcionira jednako kao kod binarnog prikaza. Jedina je razlika da se ne kopiraju bitovi već brojevi s pomičnom točkom. Za detalje pogledati a)

c) **Uniformno križanje** – jednako kao križanje za binarni prikaz, samo što se umjesto bitova koriste brojevi s pomičnom točkom. Za detalje pogledati uniformno križanje za binarni niz (b).

2.2.2. Operatori mutacija

Dvije vrste operatora mutacija su implementirane za prikaz broja s pomičnom točkom:

a) **Jednostavna mutacija** – nasumično se odabere element (u slučaju više dimenzionalnosti) te se promijeni na neku nasumičnu vrijednost, koja se mora nalaziti unutar intervala zadanog preko parametara u konfiguracijskoj datoteci.

b) **Neuniformna mutacija** – ako pretpostavimo da je jedinka definirana kao $s_r =< v_1,...,v_n >$, pri čemu je n dimenzionalnost jedinke, i ako je nasumično odabran element iz jedinke v_k tada će mutirana jedinka biti definirana kao $s_r =< v_1,...,v_k',...,v_n >$, dok vrijedi $v_k' = \begin{cases} v_k + \Delta(t, u_k - v_k) \\ v_k - \Delta(t, v_k - l_k) \end{cases}$, $\Delta(t, y) = y \cdot (1 - r^{(1-\frac{t}{T})^b})$, r predstavlja nasumičan broj iz intervala [0..1], T je maksimalan broj generacija, t je trenutna generacija, b = 5 (može se podesiti u konfiguracijskoj datoteci). Na koji od dva načina će biti v_k' izračunat se odabire nasumično.

2.3. Permutacijski niz

Prikaz jedinke permutacijskim nizom se sastoji od N cijelih brojeva, pri čemu se nijedan broj ne ponavlja. N predstavlja dimenzionalnost permutacijskog niza, i može se postaviti u konfiguracijskoj datoteci preko parametra size.

Parametri iz konfiguracijske datoteke koji se odnose na pojedini operator križanja ili mutacije se mogu vidjeti na slici Slika 2.6. Na koji se točno operator križanja ili mutacije odnosi parametar se može vidjeti u tablici Tablica 2.3.

```
<Entry key="crx.OX">1</Entry>
<Entry key="crx.PBX">1</Entry>
<Entry key="crx.PMX">1</Entry>

<Entry key="mut.inv">0.7</Entry>
<Entry key="mut.ins">0.1</Entry>
<Entry key="mut.toggle">0.1</Entry>
```

Slika 2.6 Parametri iz konfiguracijske datoteke za operatore križanja i mutacije

Tablica 2.3 Prikaz parametra i pripadajućeg operatora za permutacijski niz

crx.OX	OX križanje
crx.PBX	PBX križanje
crx.PMX	PMX križanje
mut.inv	Inverzna mutacija
mut.ins	Posmačna mutacija
mut.toggle	Jednostavna mutacija

2.3.1. Operatori križanja

Implementirana su tri operatora križanja specifična za prikaz rješenja preko permutacijskog niza:

a) ***PMX*** *(engl. Partially matched crossover)* – Goldberg i Lingle predložili [6] – gradi dijete tako da odabire podniz od jednog roditelja i očuva poziciju i red što je više moguće kraljica drugog roditelja. Podniz roditelja se selektira nasumičnim odabirom dva mjesta rezanja, koja služe kao granica kod zamjene. Npr. dva roditelja (pri čemu su mjesta rezanja označena sa „|“)

r1 = (1 2 3 | 4 5 6 7 | 8 9)

r2 = (4 5 2 | 1 8 7 6 | 9 3)

bi proizvela dvoje djece na sljedeći način. Prvo, dijelovi između mjesta rezanja se zamijene (simbol „x“ će se u sljedećih par primjera smatrati kao „trenutačno nepoznato“):

d = (x x x | 1 8 7 6 | x x)

Ova zamjena isto tako definira i seriju mapiranja:

1 <--> 4, 8 <--> 5, 7 <--> 6, 6 <--> 7

Tada se mogu popuniti nepoznata mjesta (od originalnih roditelja), ali ona za koja ne postoji konflikt (ne postoji već isti broj u nizu):

d = (x 2 3 | 1 8 7 6 | x 9)

Na kraju, prvi „x“ kod djeteta (koji bi trebao biti 1, ali postoji konflikt, jer već postoji broj 1 u nizu) se zamjenjuje sa 4, jer postoji mapiranje 1 <--> 4. Jednako tako, drugi „x“ kod djeteta se zamjenjuje sa 5. Nakon toga dijete izgleda kao:

d = (4 2 3 | 1 8 7 6 | 5 9)

b) ***OX*** *(poredano križanje)* – Davis predložio [6] – gradi dijete tako da odabire podniz od jednog roditelja i očuva poziciju i red što je više moguće elemenata drugog roditelja. Npr. dva roditelja (pri čemu su mjesta rezanja označena sa „|“)

r1 = (1 2 3 | 4 5 6 7 | 8 9)

r2 = (4 5 2 | 1 8 7 6 | 9 3)

bi proizvela dijete na sljedeći način. Prvo, dijelovi između mjesta rezanja se kopiraju u dijete:

d = (x x x | 4 5 6 7 | x x)

Dalje, počevši od drugog mjesta rezanja jednog roditelja, elementi iz drugog roditelja su kopirani u istom poretku, pri tome izostavljajući one elemente koji su već zauzeti. Pri dolasku do kraja niza, kreće se od

početne pozicije u nizu. Poredak elemenata kod drugog roditelja (s početkom od drugog mjesta rezanja) je:

9 – 3 – 4 – 5 – 2 – 1 – 8 – 7 – 6

nakon micanja elemenata 4, 5, 6 i 7, koja već postoje u djetetu, dobije se

9 – 3 – 2 – 1 – 8

Ovaj poredak elemenata se stavlja u dijete (s početkom od drugog mjesta rezanja):

d = (2 1 8 | 4 5 6 7 | 9 3)

c) **PBX** (engl. *Position based crossover*) – gradi dijete tako da se prvo nasumične vrijednosti iz prvog roditelja kopiraju na jednaka mjesta u djetetu, a nakon toga se na ostala mjesta kopiraju vrijednosti od drugog roditelja. Npr. ako imamo dva roditelja:

r1 = (1 2 3 4 5 6 7 8 9)

r2 = (4 5 2 1 8 7 6 9 3)

i ako su nasumično odabrana mjesta, npr. 1, 2, 5 i 9, tada se prvo kopiraju vrijednosti sa tih mjesta iz prvog roditelja:

d = (1 2 x x 5 x x x 9)

te se nakon toga kopiraju vrijednosti iz drugog roditelja, ali jednakim redoslijedom kao i kod tog roditelja:

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

2.3.2. Operatori mutacija

Za permutacijski niz su implementirane tri različite mutacije:

a) **Posmačna mutacija** – prvo se nasumično odredi podniz jedinke, te se tada elementi podniza pomaknu u desno, pri čemu se najdesniji element podniza postavi na prvo mjesto. Npr. imamo jedinku:

j = (1 2 3 4 5)

i uzmimo da se nasumično odabrao podniz od drugog do četvrtog elementa jedinke. Nakon prebacivanja četvrtog (najdesnijeg) elementa na prvo mjesto podniza (drugo mjesto jedinke), mutirana jedinka izgleda:

mj = (1 4 2 3 5)

b) **Inverzna mutacija** – prvo se, kao i kod prošle mutacije, nasumično odabere podniz jedinke, te se tada zamijene mjesta elemenata, tako da se prvo zamijeni prvi i zadnji element podniza, zatim drugi i predzadnji, i td. Npr. ako imamo jedinku:

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

i uzmimo da se nasumično odabrao podniz od trećeg do sedmog elementa jedinke. Prvo se zamijene mjesta prvog i zadnjeg elementa podniza pa dobivamo:

j = (1 2 7 x x x 3 8 9)

zatim se zamijene mjesta drugog i predzadnjeg elementa podniza, te ostaje srednji element, koji se ne mijenja s obzirom da je jedini u podnizu ostao nezamijenjen pa se nema ni s kojim drugim elementom za zamijeniti. Nakon toga dobivamo mutiranu jedinku koja izgleda kao:

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

c) **Jednostavna mutacija** – prvo se nasumično odrede dva elementa jedinke, te se ona zamijene.

3. Ispitni primjeri

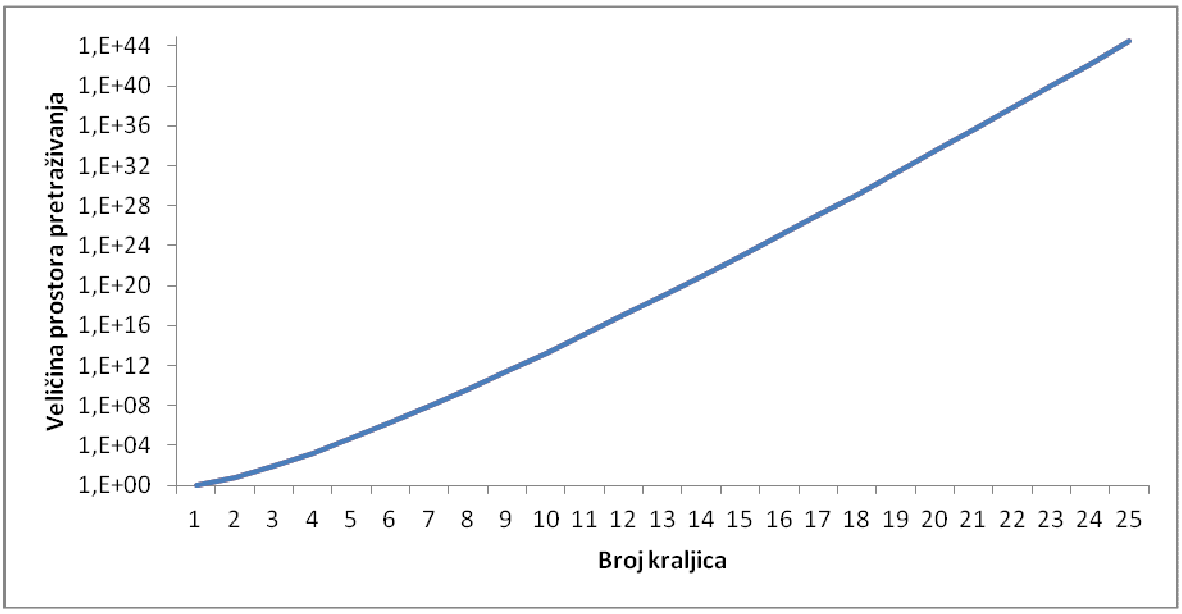
Zbog više različitih implementacija genotipa u ECF-u, implementirani su i različiti primjeri s kojima je ispitana učinkovitost genotipa. Za implementaciju genotipa kao permutiranog niza je implementiran problem N kraljica i problem trgovačkog putnika. Za usporedbu binarnog niza i prikaz broja s pomičnom točkom je implementirano sedam funkcija, gdje je cilj pronaći globalni minimum tih funkcija.

3.1. Problem N kraljica

Problem n-kraljica je klasičan kombinatorički problem u području umjetne inteligencije. Budući da problem ima jednostavnu, regularnu strukturu, i inherentne je kompleksnosti, korišten je za stvaranje mjernih programa za algoritme pretraživanja umjetne inteligencije. Problem je polinomne složenosti (P) ako koristi manje od O(n^t) koraka, gdje je t neki konačni broj. Ako se pogođeno rješenje problema može provjeriti u polinomnom vremenu (tj. mogu se ponoviti rezultati testiranja) onda se kaže da je NP, baš kao što je i problem N kraljica. Skup problema koji leže u NP je jako velik (uključuje i problem faktorizacije brojeva), pri čemu složenosti mogu dosezati $O(2^n)$ ili $O(n!)$.

Za rješavanje problema je potreban algoritam, koji je efikasan u pretrazi i optimizacijskim okolinama, a isto tako mora biti sposoban zadovoljiti ograničenja problema. Kod takvih problema je cilj pronaći sve dodijeljene vrijednosti takve da zadovoljavaju sva ograničenja. Ne postoji ni jedan algoritam polinomne složenosti koji može riješiti NP-težak problem, ali postoje neke nedeterminističke metode koje omogućuju rješavanje NP problema u polinomijalnom vremenu. Jedna od takvih metoda je genetski algoritam, no mora se upamtiti da je genetski algoritam samo aproksimativan, tj. ne jamči pronalaženje optimalnog rješenja zadanog problema.[3]

Problem 4-kraljica je najjednostavniji primjer problema n-kraljica za koji postoji rješenje. Potrebno je postaviti četiri kraljica na šahovsku ploču veličine 4x4, tako da se nijedan par kraljica ne može međusobno napasti. To znači da nijedan par kraljica se ne može nalaziti u istom redu ili stupcu ili na istoj dijagonali. U općenitom problemu se treba postaviti N kraljica na šahovsku ploču veličine NxN, tako da se nijedan par kraljica ne može napadati. Postoji $\binom{n^2}{n}$ različitih načina da se postavi N kraljica na ploču veličine NxN. Za relativno malen problem od 10 kraljica, postoji više od $1.73 * 10^{13}$ načina postavljanja na ploču. To je ogroman prostor rješenja za pretraživanje, kod tako malog problema. Na slici Slika 3.1 se može vidjeti koliko je zapravo problem N kraljica zahtjevan za rješavanje.



Slika 3.1 Složenost problema N kraljica

Zbog velike kompleksnosti problema N kraljica, poželjno je prilagoditi i sami prikaz rješenja kod genetskog algoritma, da bi se ubrzao njegov rad. U slučaju permutiranog niza se dobiva ubrzanje na temelju toga što se ne treba provjeravati za konfliktima u istom retku ili stupcu.

Poteškoća s određivanjem funkcije dobrote kod mnogih problema je u stvaranju funkcije koja će što bolje odrediti koliko je neko rješenje dobro, tj. koliko je blizu optimumu. Za razliku kod problema pronalaska minimuma neke funkcije, kod problema N kraljica nemamo djelomično točno rješenje, jer ako nemamo konflikata između kraljica (tj. nijedna kraljica ne može napasti neku drugu), onda smo pronašli rješenje, a ako imamo, onda to nije rješenje problema. Zbog toga je cilj funkcije dobrote odrediti koliko je neko pogrešno rješenje blisko točnom. Prikazi rješenja su odabrani tako da se kod oba prikaza eliminiraju konflikti po stupcima i recima. U tom slučaju je jedini problem koji se javlja kod problema N kraljica zapravo taj da dolazi do konflikata kraljica po dijagonalama. Dobrota najbolje jedinke definirana je kao ukupan broj kraljica, tj. N. Za svaku kraljicu koja je u konfliktu sa rješenjem, dobrota će se smanjivati za 1.

Za određivanje konflikata na dijagonalama unutar nekog rješenja kod permutiranog prikaza $(k_1, k_2, ..., k_n)$, koriste se dvije formule:

$$k_i = k_j + (i - j) \tag{2}$$

$$k_i = k_j - (i - j) \tag{3}$$

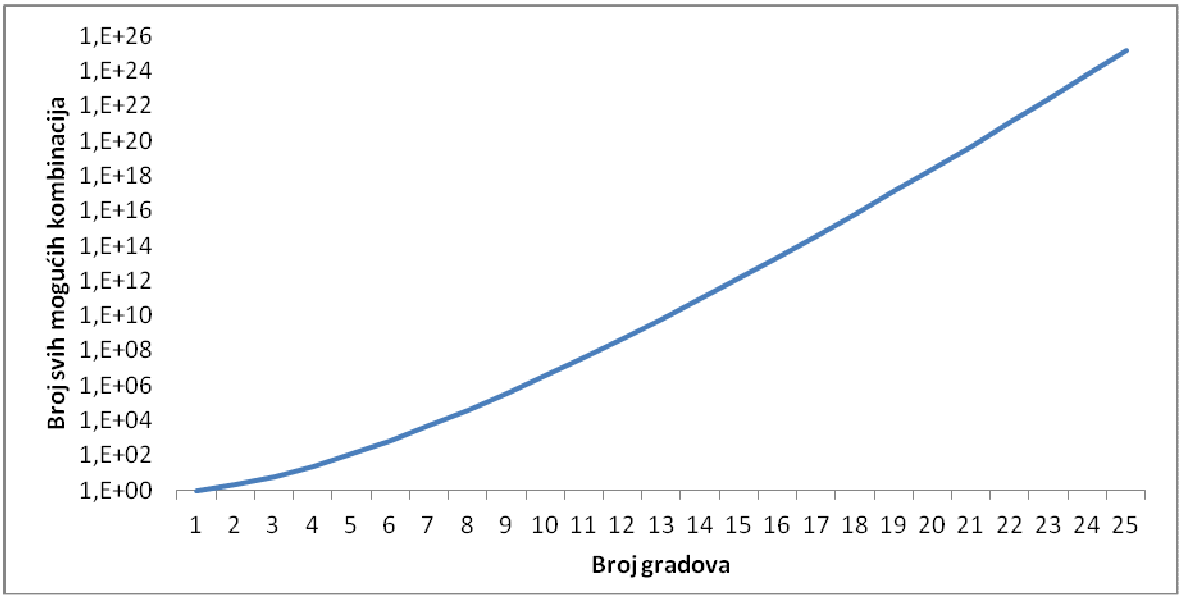
Glavni algoritam pronalaska konflikata je takav, da na početku počne od trenutne pozicije kraljice u dva smjera, budući da postoje dvije dijagonale, gornja i donja. Za svaku sljedeću poziciju ispituje se je li u konfliktu na temelju dvije prethodno napisane formule. S prvom se provjerava da li se trenutno ispitivana pozicija kraljice nalazi na gornjoj dijagonali početne kraljice, dok se s drugom formulom ispituje da li se trenutno ispitivana pozicija kraljice nalazi na donjoj dijagonali početne kraljice. Kada se nađe konflikt na nekoj od dijagonala, tada se prekida potraga na toj dijagonali. Na taj način se izbjegava pogrešno računanje konflikata u slučaju više kraljica na istoj dijagonali. Složenost algoritma je u najgorem slučaju, tj. kada nema konflikata, jednaka $O(n^2)$, što je zapravo relativno sporo, dok u najboljem slučaju, tj. kada je svaka kraljica u konfliktu sa onom

sljedećom, se složenost smanjuje, jer se ne mora kod svakog bita jedinke prolaziti do kraja niza, već se traženi konflikti pronalaze nakon prolaska dva susjedna bita.

3.2. Problem trgovačkog putnika

Problem trgovačkog putnika (engl. *Travelling salesman problem*) je NP težak problem, koji spada u kombinatorne optimizacije, pri čemu mu je složenost *O(n!)*. Problemi slični trgovačkom putniku su zaokupljali razne matematičare tokom povijesti, pa je tako Euler prvi počeo razmatrati slične probleme, pri čemu je njega zanimalo kako bi skakač na šahovskoj ploči posjetio svih 64 mjesta, a da ne prođe istim mjestom dva puta. Opća forma problema se pojavljuje početkom 20. stoljeća, dok je sam pojam „trgovački putnik“ prvi put upotrijebljen 1932. godine.

Problem trgovačkog putnika se svodi na pronalazak puta koji prolazi svakim gradom točnom jednom, a da je on najkraći mogući. Na prvi pogled se čini jednostavan problem, ali nije teško pronaći neku kombinaciju gradova kojom prolazi put, već je problem iz ogromne složenosti tj. ogromnog broja mogućih kombinacija, pronaći onaj optimalan. Na slici Slika 3.2 **Složenost problema trgovačkog putnika** se može vidjeti koliko je to zapravo složen i zahtjevan problem.



Slika 3.2 Složenost problema trgovačkog putnika

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 (engl. *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 (engl. *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 (engl. *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 (engl. *Traveling Politician Problem*), a može se svesti na obični problem trgovačkog putnika sa istim brojem gradova, ali modificiranim udaljenostima.

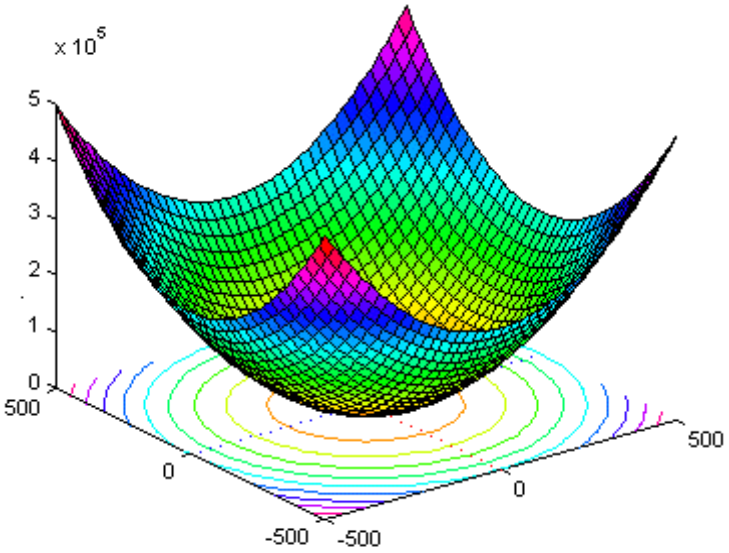
3.3. Funkcije

Za usporedbu binarnog niza i prikaza broja s pomičnom točkom implementirano je 7 funkcija koje će u detalje biti objašnjene u sljedećim odlomcima. Prvih 6 funkcija, sa još nekim dodatnim optimizacijskim funkcijama i njihovim opisima, se može vidjeti na [7].

3.3.1. De Jongova funkcija 1

De Jongova funkcija 1 je jedna od najjednostavnijih optimizacijskih funkcija. Još je poznata kao sferni model. Kontinuirana je, konveksna i unimodalna. Definirana je kao: $f(x) = \sum_{i=1}^{n} x_i^2$, $-5.12 \le x_i \le 5.12$

Globalni minimum je f(x)=0, x_i=0, i=1..n. Na slici Slika 3.3 **Graf De Jongove funkcije 1** se može vidjeti 2D prikaz funkcije u prostoru [-500, 500].



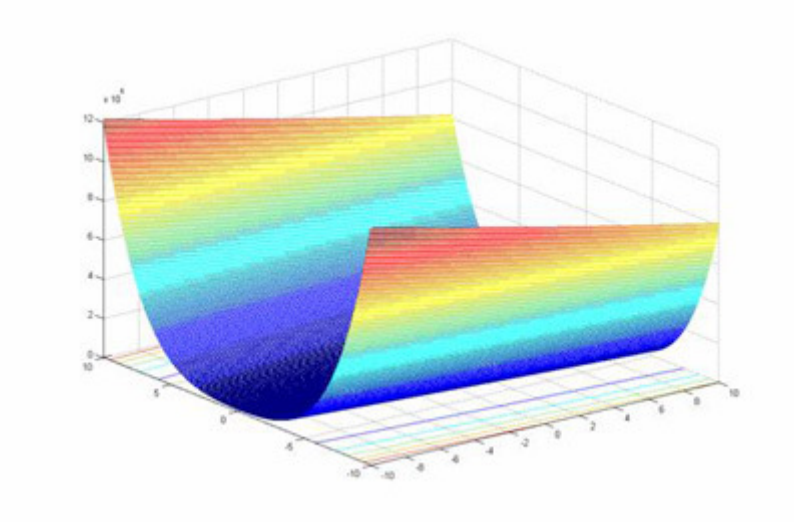
Slika 3.3 Graf De Jongove funkcije 1

3.3.2. Rosenbrockova dolina

Rosenbrockova dolina (ili De Jongova funkcija 2 ili još poznata kao banana funkcija) predstavlja klasičan optimizacijski problem. Globalni minimum se nalazi unutar duge, uske, paraboličko oblikovane ravnine. Problem kod ove funkcije je dostići globalni minimum, zbog čega je funkcija često korištena za procjenu učinkovitosti algoritma. Funkcija je definirana kao:

$$f(x) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (1 - x_i)^2), \quad -2.048 \le x_i \le 2.048$$

Globalni minimum je f(x)=0, x_i=0, i=1..n. 2D prikaz funkcije se može vidjeti na slici Slika 3.4.



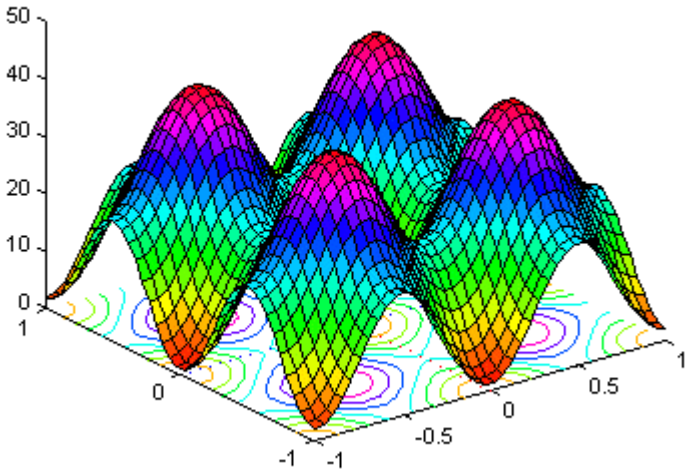
Slika 3.4 Graf Rosenbrockove funkcije

3.3.3. Rastriginova funkcija

Rastriginova funkcija je varijanta De Jongove funkcije 1 uz kosinus modulaciju da bi se dobilo više lokalnih minimuma. Funkcija je definirana kao:

$$f(x) = 10 \cdot n + \sum_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2 \cdot \pi \cdot x_i)), \quad -5.12 \le x_i \le 5.12$$

Globalni minimum je f(x)=0, x_i=0, i=1..n. Prikaz funkcije u 2D-u se može vidjeti na slici Slika 3.5.



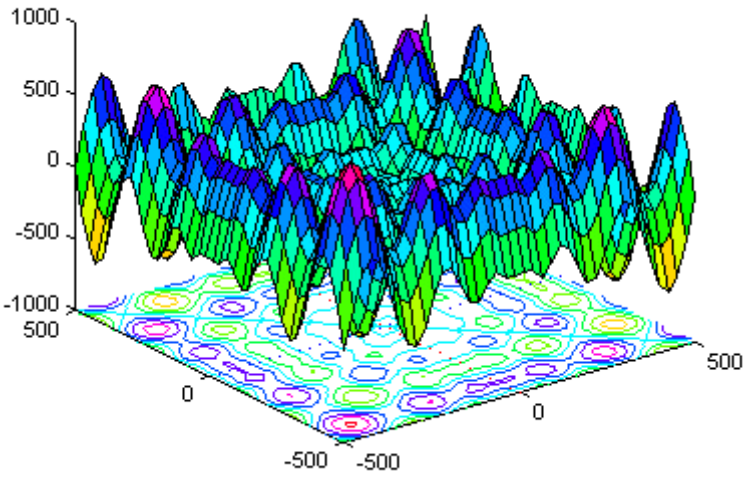
Slika 3.5 Graf Rastriginove funkcije

3.3.4. Schwefelova funkcija

Problem pronalaska minimuma kod Schwefelove funkcije je taj što je globalni minimum dosta udaljen od sljedećeg najboljeg lokalnog minimuma, pa se algoritmi često zaglibe tj. konvergiraju u krivi smjer. Funkcija je definirana kao:

$$f(x) = \sum_{i=1}^{n} -x_i \cdot \sin(\sqrt{|x_i|}), \quad -500 \le x_i \le 500$$

Globalni minimum je f(x)=-n*418.9829, x_i=420.9687, i=1..n. 2D graf se može vidjeti na slici Slika 3.6.



Slika 3.6 Graf Schwefelove funkcije

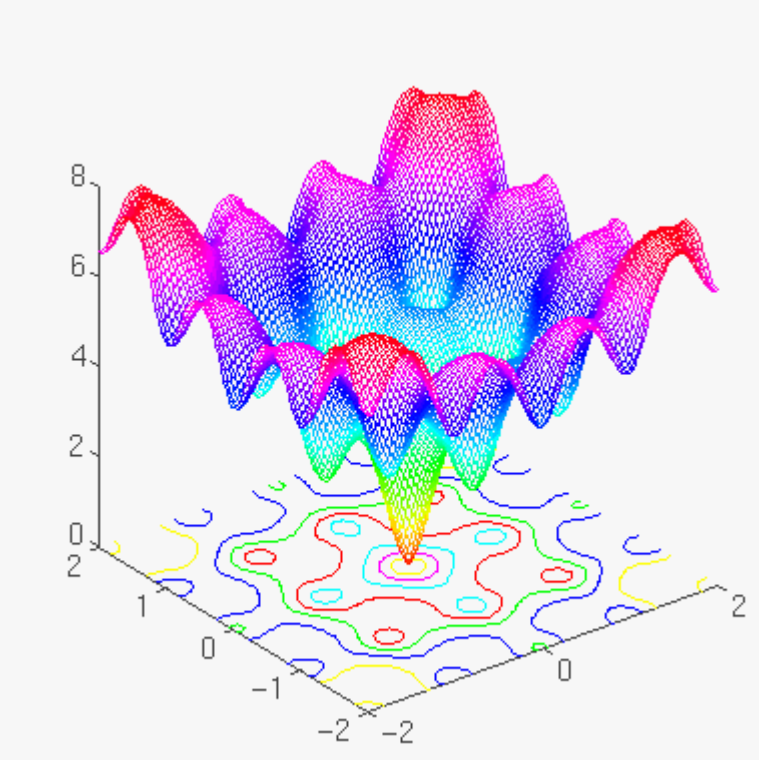
3.3.5. Ackleyeva staza

Ackleyeva staza (engl. *Ackley's Path*) je često korištena multimodalna optimizacijska funkcija. Definirana je kao:

$$f(x) = -a \cdot e^{-b\cdot\sqrt{\frac{\sum_{i=1}^{n} x^2}{n}}} - e^{\frac{\sum_{i=1}^{n}\cos(c\cdot x_i)}{n}} + a + e,$$

$$-1 \le x_i \le 1, a = 20, b = 0.2, c = 2\cdot\pi$$

Globalni minimum je f(x)=0, x_i=0, i=1..n. Prikaz funkcije u 2D-u se može vidjeti na slici Slika 3.7.



Slika 3.7 Graf Ackleyeve staze

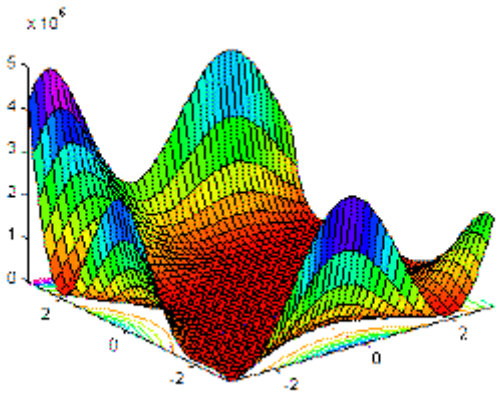
3.3.6. Goldstein-Price funkcija

Goldstein-Price je tipična optimizacijska 2D funkcija. Definirana je kao:

$$f(x_1,x_2)=(1+(x_1+x_2+1)^2\cdot(19-14x_1+3x_1^2-14x_2+6x_1x_2+3x_2^2))\cdot$$
$$(30+(2x_1-3x_2)^2\cdot(18-32x_1+12x_1^2+48x_2-36x_1x_2+27x_2^2))$$

$$-2\le x_1,x_2\le 2$$

Globalni minimum je $f(x_1,x_2)=3$, $(x_1,x_2)=(0,-1)$. Graf funkcije se može vidjeti na slici Slika 3.8.



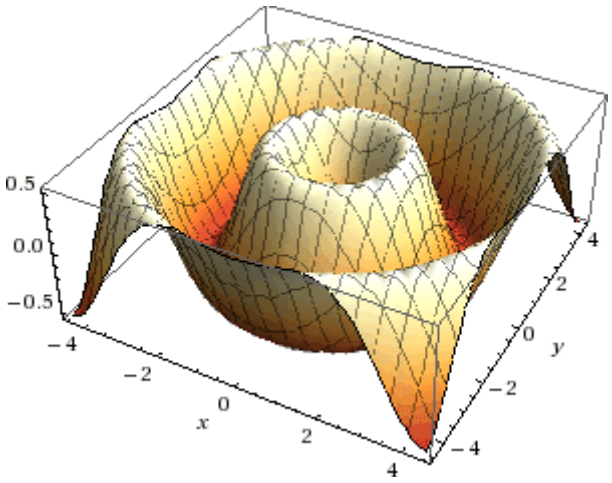
Slika 3.8 Graf Goldstein-Price funkcije

3.3.7. Funkcija 7

Sedma testna funkcija je definirana kao: $f(x) = 0.5 - \frac{\sin^2 \sqrt{\sum_{i=1}^{N} x_i^2} - 0.5}{(1 + 0.001 \sum_{i=1}^{N} x_i^2)^2}$,

$-100 \le x_i \le 100$.

Globalni minimum je f(x)=1, x_i=0, i=1..n. Prikaz funkcije u 2D-u se može vidjeti na slici Slika 3.9.



Slika 3.9 2D graf sedme testne funkcije

4. Rezultati ispitivanja

Svi testovi navedeni u ovom dijelu rada su testirani na dvojezgrenom Core2Duo 2.4GHz procesoru. Za svaki prikaz je korišten genetski algoritam sa 3-turnirskim odabirom.

4.1. Permutirani niz

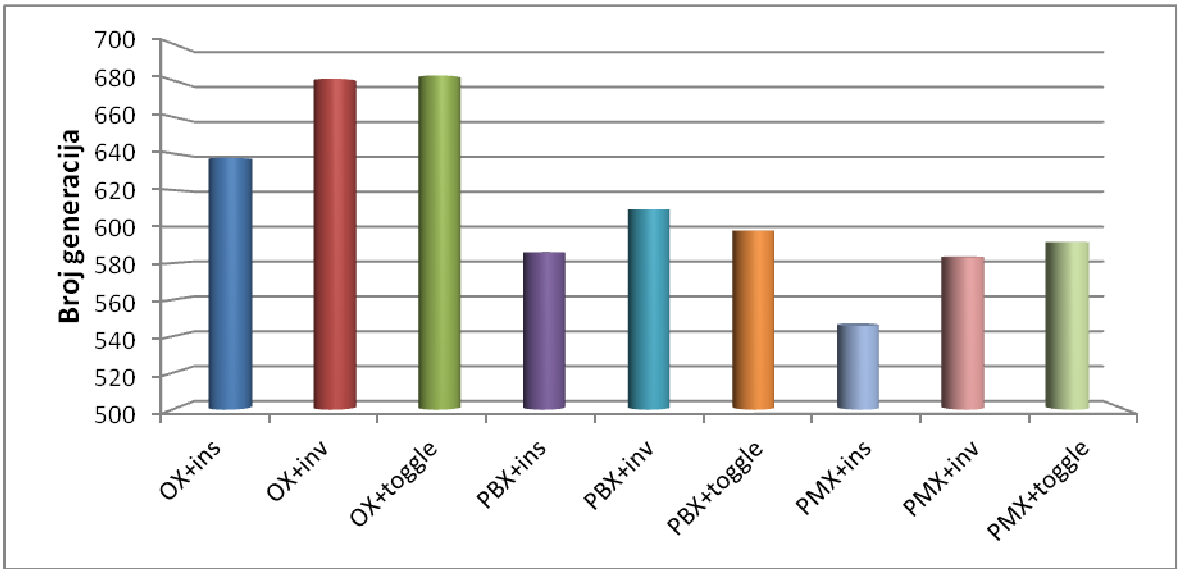
Implementacija genotipa permutirani niz je ispitana na dva primjera kombinatoričkog problema, problem N kraljica i TSP. Ispitane su kombinacije operatora križanja i mutacije kako utječu u prosjeku na dobrotu, te koja kombinacija je najbolja za određen problem. Postoji tri vrste križanja i mutacije za permutirani niz, što znači da je ispitano devet kombinacija. Uz to je ispitan i utjecaj dimenzije problema na vrijeme izvođenja algoritma.

4.1.1. Problem N kraljica

Za svaku od devet kombinacija parova križanja i mutacija koja je ispitana na problemu fiksne dimenzije (20), vjerojatnost križanja je bila 1, a vjerojatnost mutacije je varirana od 0.1, 0.2 … 1. Za svaku vrijednost mutacije ispitivanje je ponovljeno 70 puta, te je uzeta prosječna vrijednost generacije, dobrote najbolje jedinke te postotka pronalaska rješenja na veličini populacije od 100 jedinki. Važno je napomenuti da su bili postavljeni sljedeći parametri zaustavljanja algoritma: maksimalan broj generacija je 4000, vrijeme izvođenja algoritma je 60sec te maksimalan broj generacija bez promjene dobrote najbolje jedinke je 500.

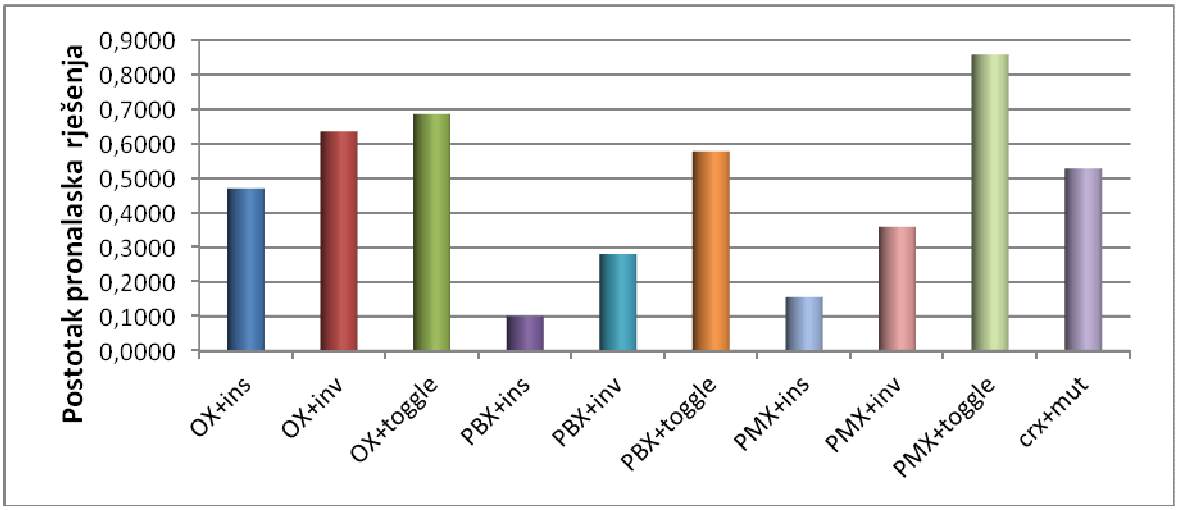
Na slici Slika 4.1 je prikazan prosječan broj generacija potreban određenoj kombinaciji operatora križanja i mutacije. Kada bi se uzeli u obzir samo ti rezultati, moglo bi se zaključiti da je najbolja kombinacija za problem N kraljica PMX križanje i posmačna mutacija, no zbog postavljenih operatora zaustavljanja to nije tako. Dok su kod algoritma postavljeni operatori zaustavljanja nije moguće

zaključivati o tome koja je najbolja kombinacija križanja i mutacije na temelju broja generacija, zbog njihovih utjecaja.



Slika 4.1 Prosječan broj generacija potreban svakoj kombinaciji operatora

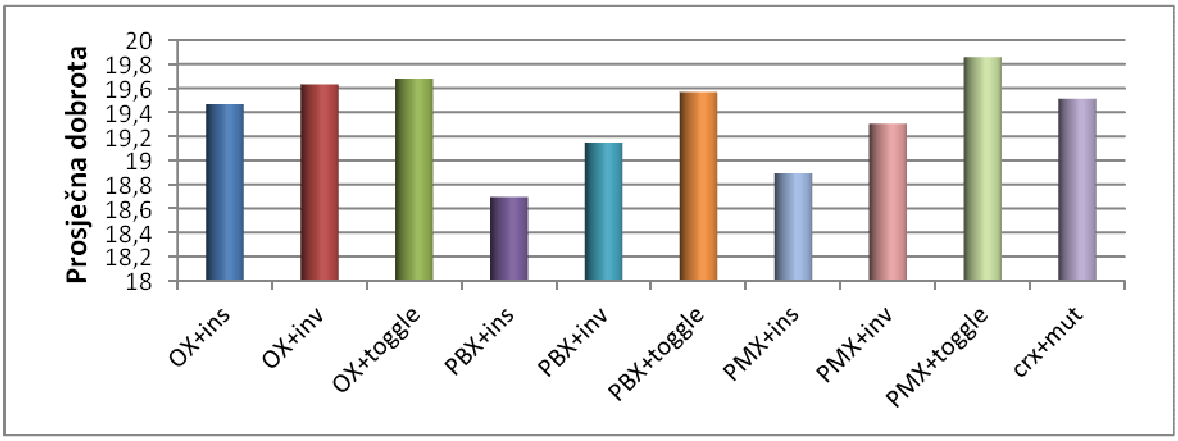
Npr. ako se pogleda slika Slika 4.2 koja prikazuje prosječan postotak pronalaska rješenja za pojedinu kombinaciju operatora, jasno se može vidjeti da kombinacija operatora PMX i posmačna mutacija zapravo uopće nije dobar izbor jer joj je uspješnost pronalaska rješenja tek nešto iznad 15%, što je druga najlošija kombinacija od svih 9. Rezultati su takvi jer ovdje dolazi do utjecaja operatora zaustavljanja kada se dobrota najbolje jedinke nije promijenila određen broj generacija, u našem slučaju barem 500. Što znači da se kod te kombinacije vrlo brzo konvergiralo lokalnom minimumu i najbolja jedinka se više nije mijenjala, zbog čega se algoritam brzo zaustavljao.



Slika 4.2 Postotak pronalaska rješenja svake kombinacije operatora

Zadnja kombinacija parametara označena kao „*crx+mut*“ (Slika 4.2), predstavlja kombinaciju kada se u konfiguracijsku datoteku postave parametri za sve operatore križanja i mutacije vezane za definirani prikaz rješenja. Vjerojatnosti operatora su jednake za sve operatore, npr. ako imamo dva operatora križanja i tri operatora mutacija, tada je vjerojatnost svakog križanja 50% a vjerojatnost svake mutacije 33.33%. To znači da je odabir određenog operatora križanja ili mutacije nasumičan, ali sa jednakom vjerojatnošću.

Prema rezultatima sa slike Slika 4.2 najbolja kombinacija operatora križanja i mutacije za problem N kraljica je PMX križanje sa jednostavnom mutacijom, koji u prosjeku u 85.86% slučaja pronalazi traženo rješenje. Taj zaključak potvrđuju i rezultati sa slike Slika 4.3 koji predstavljaju prosječnu dobrotu najbolje jedinke. Razlog zbog čega je manja razlika između najboljih kombinacija kod prosječne dobrote za razliku od prosječnog postotka pronalaska rješenja je taj što se često pronađe jedinka jako blizu optimalne dobrote (optimalna je u našem slučaju 20, jer imamo 20 kraljica), npr. 19, ali za razliku od najbolje kombinacije puno rjeđe pronađe onu optimalnu jedinku, tj. rješenje. Rezultati za kombinaciju svih operatora križanja i mutacije su prosječni, pri čemu je postotak pronalaska rješenja iznad 50%. Tablica 4.1 prikazuje i postotke pronalaska rješenja ostalih kombinacija.

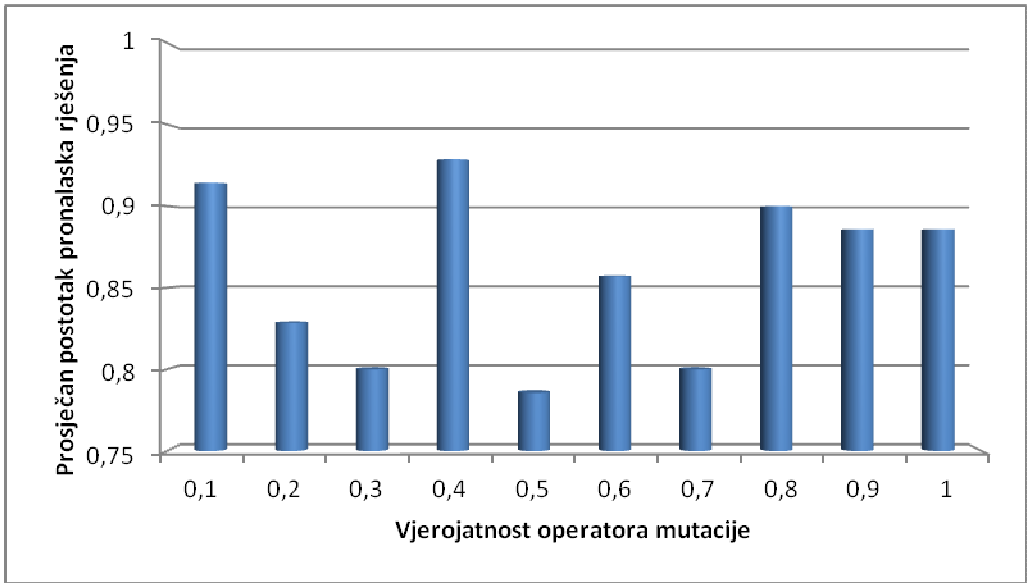


Slika 4.3 Prosječna dobrota najbolje jedinke za određenu kombinaciju

Tablica 4.1 Prosječan postotak pronalaska rješenja

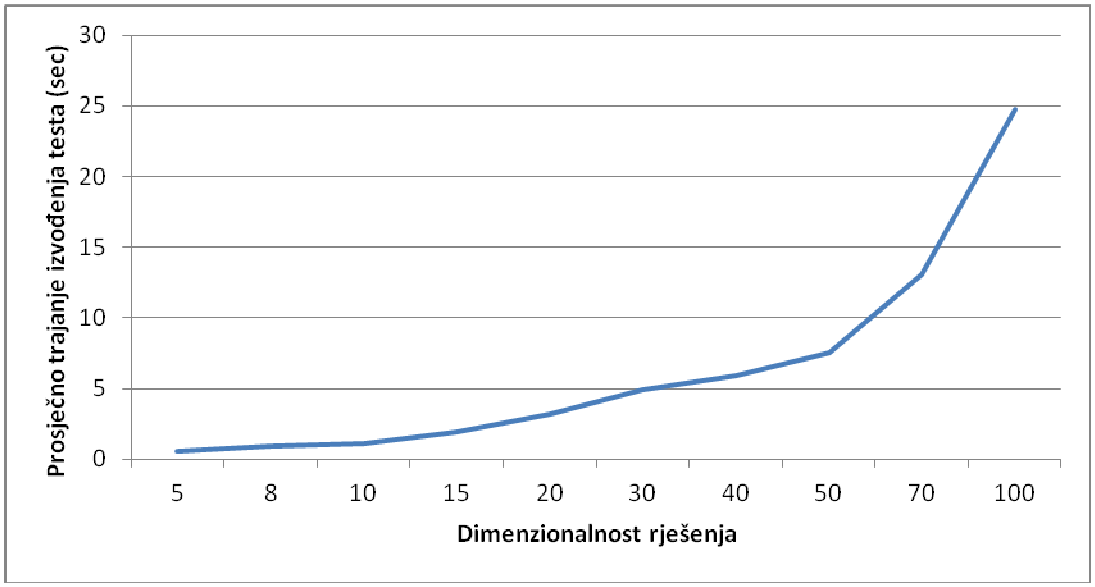
Vrsta kombinacije operatora	Prosječni postotak pronalaska rješenja
OX+ins	47,14%
OX+inv	63,57%
OX+toggle	68,57%
PBX+ins	10,14%
PBX+inv	28,14%
PBX+toggle	57,86%
PMX+ins	15,57%
PMX+inv	36%
PMX+toggle	85,86%
Crx+mut	52,86%

Rezultati testiranja vrijednosti mutacije najbolje kombinacije operatora se mogu vidjeti na slici Slika 4.4. Zanimljivo je za uočiti kako se i za niske vjerojatnosti mutacije (0.1) i za visoke (0.8) dobivaju slični rezultati, razlika je svega 1.5%, dok se najbolji rezultati dobivaju pri vjerojatnosti mutacije od 0.4, koja u 92.8% slučaja pronalazi rješenje.



Slika 4.4 Vjerojatnost pronalaska rješenja u ovisnosti o vjerojatnosti operatora mutacije

Za utjecaj dimenzionalnosti rješenja na vrijeme izvođenja algoritma korištena je kombinacija PMX križanje sa jednostavnom mutacijom, pri čemu je vjerojatnost mutacije fiksno postavljena na 0.4, na populaciji od 100 jedinki. Za svaku dimenzionalnost je ispitivanje ponovljeno 30 puta, te je za analizu uzet prosjek testova. Parametri operatora zaustavljanja su jednako postavljeni kao i za prije navedene testove. Rezultati testiranja je moguće vidjeti na slici Slika 4.5. Kao što se i može vidjeti na slici, prosječno trajanje izvođenja jednog testa se eksponencijalno povećava, zbog toga što se i sam prostor rješenja eksponencijalno povećava s porastom dimenzionalnosti. Npr. prosječno trajanje izvođenja testa sa dimenzionalnošću od 10 je 1.14sec, dok je za 100 24.75sec, što znači da se za porast dimenzionalnosti za faktor 10, trajanje povećalo za faktor 21.68.



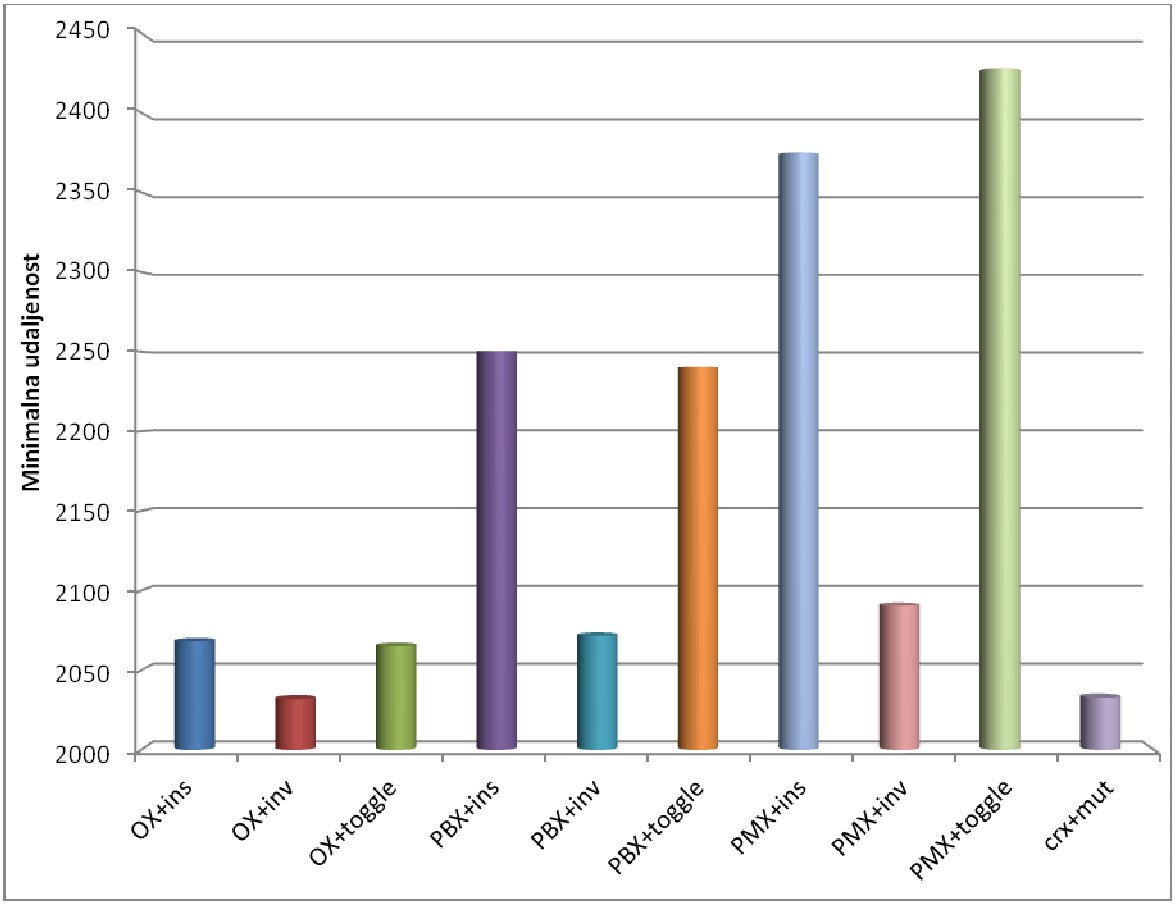
Slika 4.5 Utjecaj dimenzionalnosti rješenja na trajanje izvođenja algoritma

4.1.2. Problem trgovačkog putnika

Problem trgovačkog putnika je ispitan na jednak način kao i problem N kraljica, sa istim kombinacijama operatora križanja i mutacije, kada se koristio za problem isti genotip, tj. permutirani niz, te je ispitivanje ponavljano 210 puta, za razliku od 70 puta koliko je ponavljano ispitivanje kod problema N kraljica. Postavke operatora zaustavljanja su također jednake kao i kod problema N kraljica.

Za određivanje najbolje kombinacije operatora križanja i mutacije ispitivanje je vršeno na problemu s 29 gradova, gdje je optimalna udaljenost iznosila 2020.

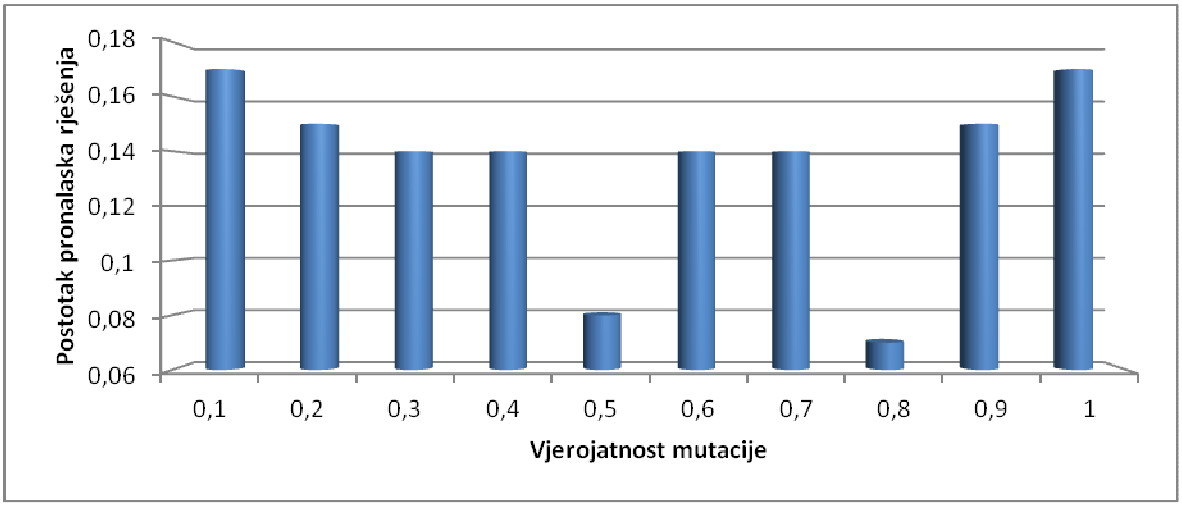
Rezultati testiranja se mogu vidjeti na slici Slika 4.6. Oni ukazuju da je najbolja kombinacija operatora križanja i mutacije OX križanje i inverzna mutacija, dok je kombinacija PMX križanja i jednostavne mutacije, koja je bila najbolja kod problema N kraljica, ovdje zapravo najgora. U ovom slučaju kombinacija svih operatora križanja i mutacije daje druge najbolje rezultate, što je mnogo bolje nego kod problema N kraljica.



Slika 4.6 Pronađena minimalna udaljenost s obzirom na kombinaciju operatora

Nakon što je pronađena najbolja kombinacija operatora križanja i mutacije za problem s 29 gradova, ispitano je koja je najbolja vrijednost vjerojatnosti mutacije. Ispitivalo se za vrijednosti 0.1, 0.2, …, 1.0, te je za svaku vrijednost vjerojatnosti test ponovljen 100 puta i uzete su prosječne vrijednosti. Postotak pronalaska rješenja se može vidjeti na slici Slika 4.7 i tablici Tablica 4.2. Zanimljivo je uočiti da se najbolji rezultati dobiju za najmanju i najveću vjerojatnost, tj. kada je vjerojatnost mutacije tek 0.1 i kada je 1.0, koje u 17% slučaja pronađu rješenje. Kada se usporedi sa ostalim vjerojatnostima mutacije, vidljivo je da je sa većinom vrijednosti jako mala razlika, tek do 3%, ali opet postoje dvije „kritične“ vrijednosti, 0.5 i 0.8, koje daju najslabije rezultate, jer tek u 7-8% slučaja pronađu rješenje. Razlika između vjerojatnosti mutacije od 0.1 i 1.0 je uočljiva kada se gleda prosječna vrijednost dobrote, gdje ima bolju vrijednost 1.0 vjerojatnost, jer u prosjeku najbolja jedinka ima manju dobrotu. Ali, kada se gledaju prosječne

vrijednosti dobrote, onda ispada da je vjerojatnost mutacije od 0.5 jedna od boljih, jer ima jedan od najmanjih prosječnih dobrota. Jedino što je moguće zaključiti iz toga je da ishod algoritma uvelike ovisi o početnoj populaciji i stohastici kod mutacije.



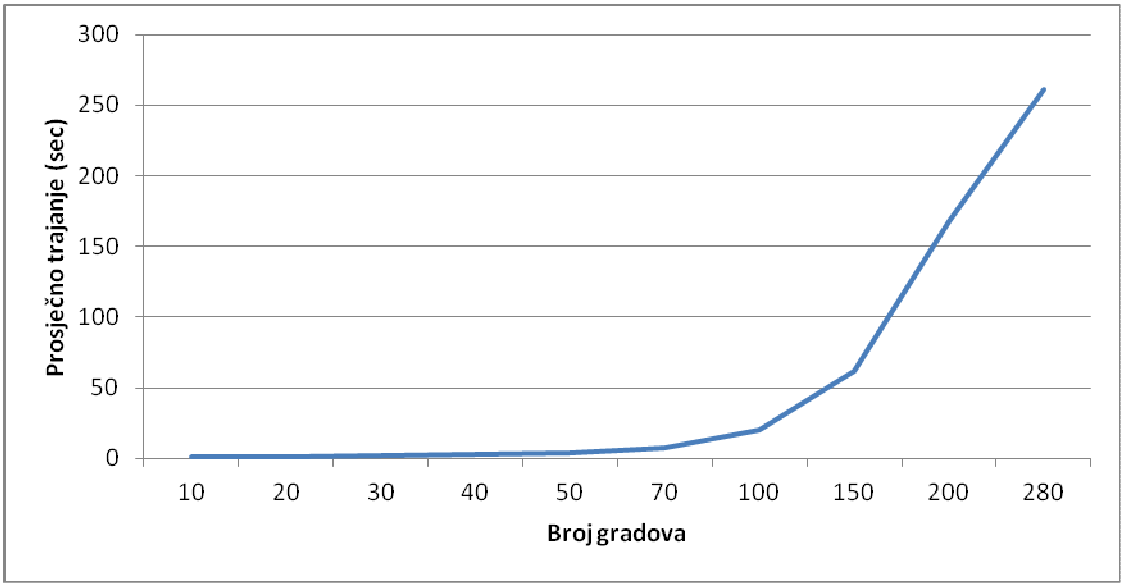
Slika 4.7 Postotak pronalaska rješenja u ovisnosti o vjerojatnosti inverzne mutacije

Tablica 4.2 Prosječni rezultati ovisni o vjerojatnosti mutacije

Vjerojatnost mutacije	Postotak pronalaska rješenja	Prosječna vrijednost dobrote
0,1	17%	2032,02
0,2	15%	2032,98
0,3	14%	2030,11
0,4	14%	2030,66
0,5	8%	2030,25
0,6	14%	2032,21
0,7	14%	2030,48

0,8	7%	2032,01
0,9	15%	2030,03
1	17%	2030,04

Kod određivanja koliki utjecaj ima dimenzionalnost rješenja na vrijeme izvođenja korištena je kombinacija OX križanja i inverzne mutacije, pri čemu je vjerojatnost mutacije iznosila 1.0. Veličina populacije je 100, maksimalan broj generacija izvođenja testa 6000, maksimalno 500 generacija bez promjene dobrote najbolje jedinke, te uz maksimalno trajanje testa od 600sec. Svaki test je ponovljen 30 puta te je uzet prosjek. Za svaku testiranu dimenzionalnost je korišten jednak problem, a to je problem sa 280 gradova, samo kod dimenzionalnosti k, k < 280, se uzimao u obzir podproblem veličine k od glavnog problema. Na slici Slika 4.8 se mogu vidjeti rezultati testiranja. Slično kao i kod rezultata kod problem N kraljica, i ovdje se trajanje testa eksponencijalno povećava zbog samog povećavanja prostora pretraživanja rješenja.

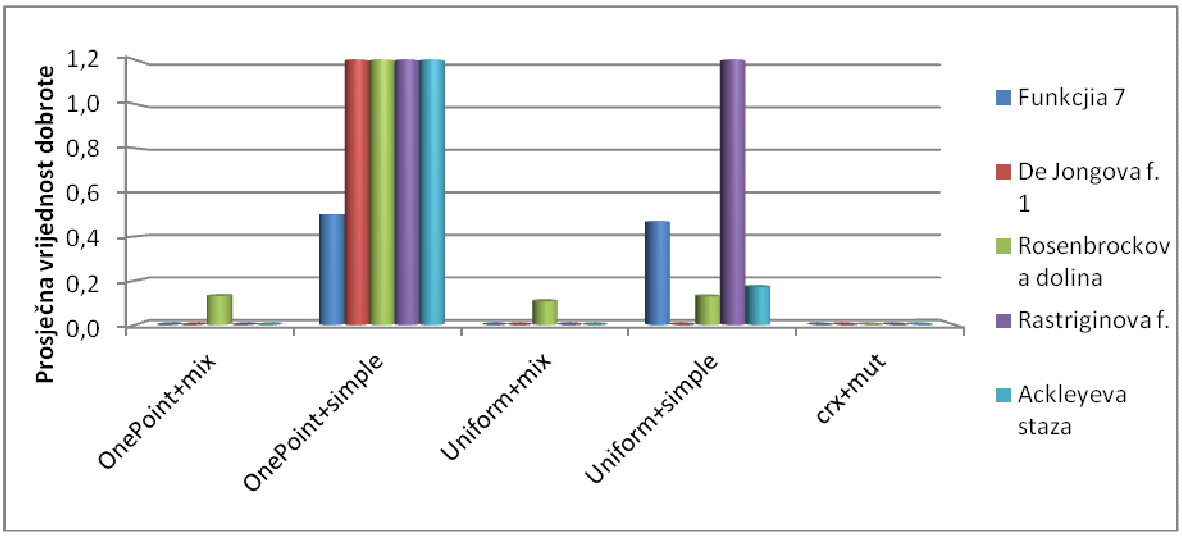


Slika 4.8 Prosječna ovisnost trajanja testa o dimenzionalnosti rješenja

4.2. Binarni niz

Implementacija binarnog niza je ispitana na 7 prije navedenih i opisanih funkcija. Ispitane su četiri kombinacije operatora križanja i mutacija da bi se odredila najbolja kombinacija za pojedinu funkciju. Funkcija dobrote, se određuje na način da je cilj algoritma postizanje dobrote 0, te je posebno prilagođen svakoj od 7 testnih funkcija. Test za svaku kombinaciju parametara je ponovljen 10 puta, te je uzet prosjek za analiziranje rezultata.

Zbog velike razlike u rezultatima između Schwefelove i Goldstein-Priceove funkciju i ostalih testnih funkcija, rezultati su prikazani na slikama Slika 4.9 i Slika 4.10.



Slika 4.9 Rezultati testiranja kombinacija operatora križanja i mutacije

Kao što se i može vidjeti sa rezultata na slici Slika 4.9, barem je jedna kombinacija operatora uspjela pronaći ili se približiti globalnom minimumu. Ako gledamo samo tih 5 funkcija, najbolja kombinacija operatora je kombinacija svih operatora križanja i mutacija koja je uspjela pronaći minimum svih funkcija ako gledamo prvih 14 decimala. Detaljni rezultati se mogu vidjeti u tablici Tablica 4.3.

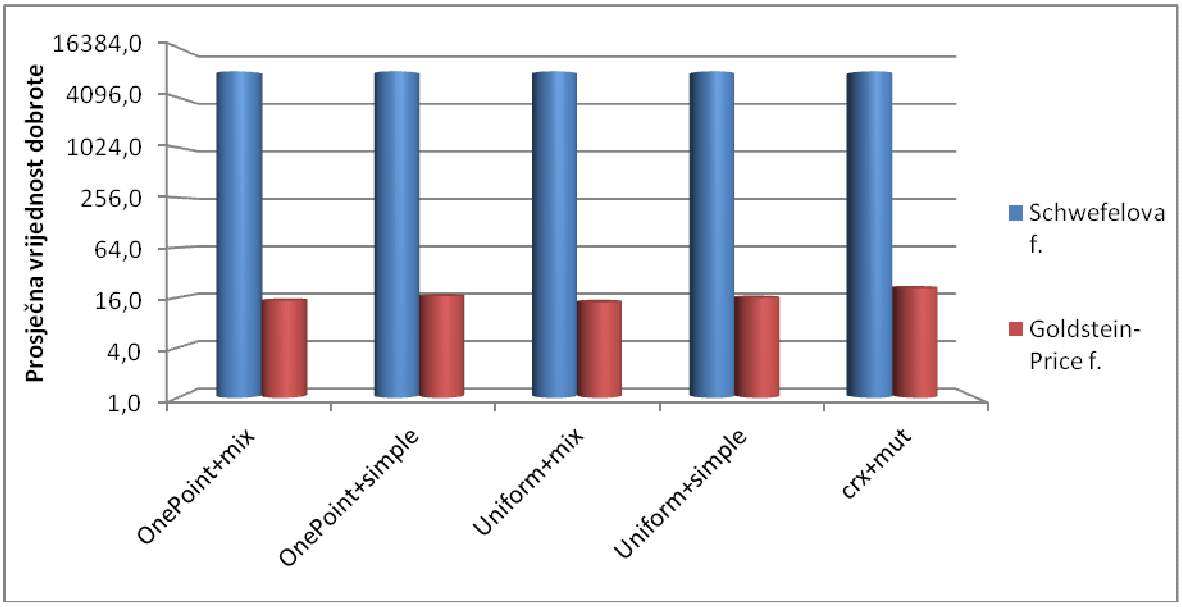
Tablica 4.3 Rezultati testiranja vezani uz sliku 4.9

	Funkcija 7	De Jongova f.	Rosenbrockova dolina	Rastriginova f.	Ackleyeva staza

OnePoint+ mix	1,000E-04	0,000E+00	1,312E-01	0,000E+00	0,000E+00
OnePoint+ simple	4,997E-01	1,331E+00	3,714E+00	2,439E+01	9,573E+00
Uniform+m ix	2,000E-04	0,000E+00	1,073E-01	0,000E+00	0,000E+00
Uniform+m ix	4,678E-01	1,000E-04	1,301E-01	1,450E+00	1,725E-01
Crx+mut	0	7,10E-30	1,56E-27	0	4,00E-15

Za razliku od gore navedenih rezultata, Schwefelova i Goldstein-Priceova funkcija su se pokazale nemogućima za makar približno rješavanje problema ispitanim kombinacijama operatora te zadanim parametrima. Kao što se i može vidjeti u tablici Tablica 4.4, kod Schwefelove funkcije se prosječna vrijednost dobrote spustila tek do 8379.658 kod uniformnog i križanja s jednom točkom prekida te miješajućom mutacijom u oba slučaja, dok kod Goldstein-Priceove funkcije se spustila na 9.46 kod uniformnog križanja i jednostavne mutacije.

Na temelju rezultata svih funkcija može se zaključiti da je najbolja kombinacija operatora za binarni prikaz jedinke, uniformno križanje i miješajuća mutacija. U tablici Tablica 4.5 se mogu vidjeti detaljni prosječni rezultati po vjerojatnosti mutacije iz najbolje kombinacije operatora. Kada se uzmu prosječne vrijednosti svih funkcija, vjerojatnost mutacije koja daje najbolje rezultate je 1.0, bez obzira što za neke druge vrijednosti vjerojatnosti se dobiva manja dobrota za određene funkcije.



Slika 4.10 Rezultati testiranja kombinacija operatora križanja i mutacije

Tablica 4.4 Rezultati testiranja vezani uz sliku 4.10

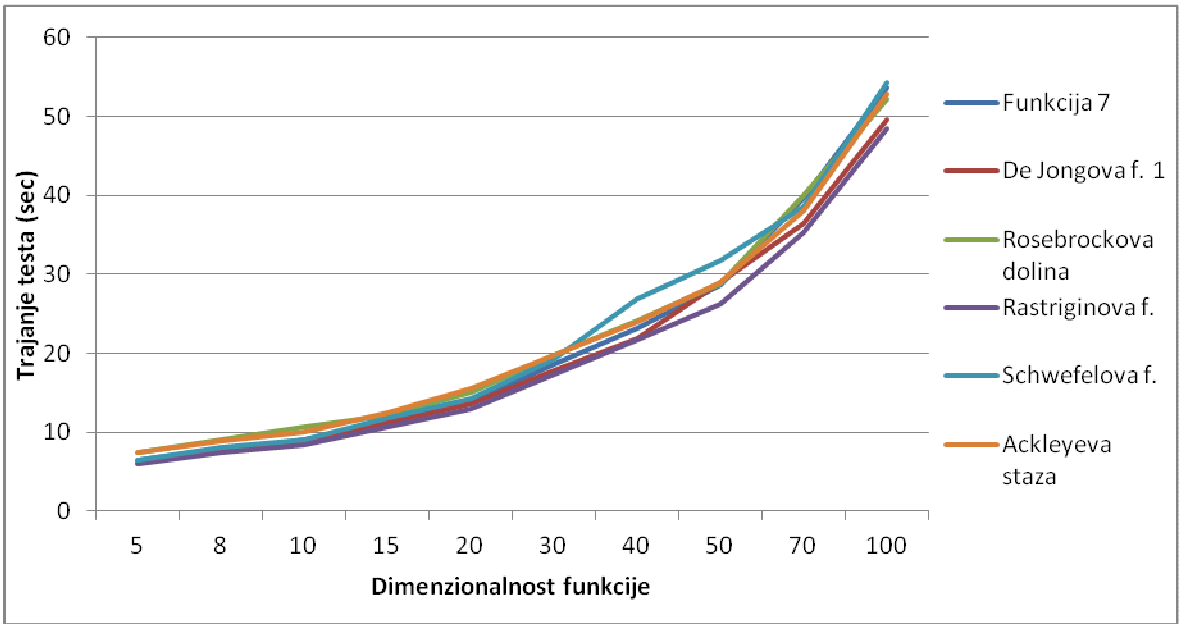
	Schwefelova f.	Goldstein-Price f.
OnePoint+mix	8379,65820	9,74910
OnePoint+simple	8392,24390	12,75090
Uniform+mix	8379,65810	10,45160
Uniform+mix	8389,65170	9,45960
Crx+mut	8379,658	14,38086

Tablica 4.5 Detaljni rezultati s obzirom na vjerojatnost mutacije iz najbolje kombinacije operatora

	Funkcija 7	De Jongova f. 1	Rosenbrockova dolina	Rastriginova f.	Schwefelova f.	Ackleyeva staza	Goldstein-Price f.
0,1	9.72E-04	3.16E-30	8.57E-28	0	8379.658	4.44E-16	10.45157

0,2	0	3.16E-30	0.119239	0	8379.658	6.13E-15	13.12594
0,3	0	6.31E-30	9.74E-28	0	8379.658	6.13E-15	13.40759
0,4	0	6.31E-30	1.19E-01	0	8379.658	4.44E-16	14.15824
0,5	0	4.73E-30	1.19E-01	0	8379.658	6.13E-15	15.90749
0,6	9.72E-04	6.31E-30	0.119239	0	8379.658	4.44E-16	12.37071
0,7	0	4.73E-30	2.38E-01	0	8379.659	3.29E-15	19.73652
0,8	0	7.89E-30	0.238479	0	8379.658	4.44E-16	14.73365
0,9	0	9.47E-30	1.19E-01	0	8379.658	6.13E-15	12.91153
1.0	0	7.89E-30	8.57E-28	0	8379.658	4.44E-16	14.54473

Utjecaj dimenzionalnosti na trajanje izvođenja testa je ispitivan na istim funkcijama i sa jednako postavljenim operatorima zaustavljanja, pri čemu je uzimana najbolja pronađena kombinacija operatora križanja i mutacije, uz najbolju vjerojatnost mutacije. Uzet je prosjek iz 20 pokretanja svakog testa. Nije se testiralo na Goldstein-Price funkciji jer je ona fiksna 2D funkcija, za razliku od ostalih testnih funkcija. Rezultati se mogu vidjeti na slici Slika 4.11. Kao što je i očekivano, vrijeme trajanja pojedinog testa se povećava s dimenzionalnošću funkcije, ali ne eksponencijalno kako smo to vidjeli na primjerima TSP problema i problema N kraljica. Zanimljivo je za uočiti da bez obzira na kompleksnost funkcije dobrote za pojedinu funkciju, vremena trajanja testa su vrlo slična. Kod dimenzionalnosti od 100, najveća je razlika u trajanju između funkcije 7 kojoj je trebalo najdulje, u prosjeku 53.675sec, i Rastriginove funkcije kojoj je trebalo 48.417sec.

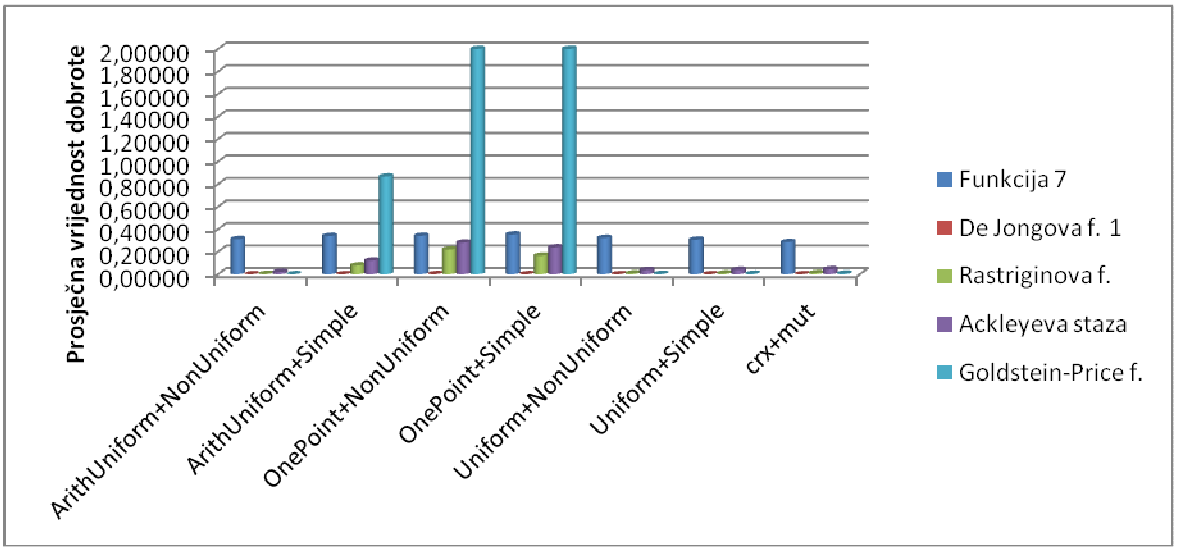


Slika 4.11 Utjecaj dimenzionalnost na trajanje testa kod binarnog prikaza

4.3. Prikaz broja s pomičnom točkom

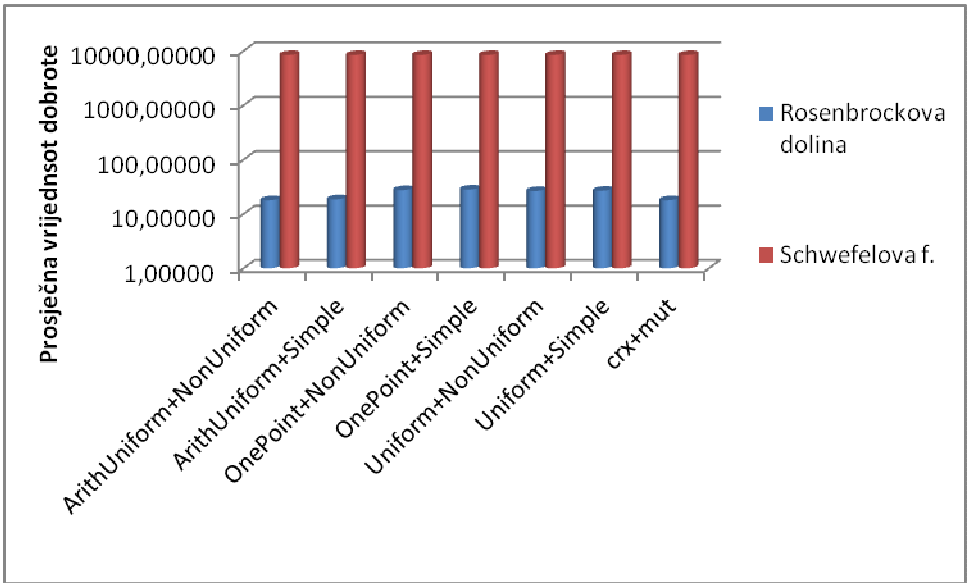
Kod FP implementacije genotipa ispitano je 6 kombinacija operatora križanja i mutacije na 7 prije navedenih funkcija. Funkcija dobrote je određena jednako kao i kod binarnog niza. Ispitivanje je provedeno na populaciji od 100 jedinki, te su operatori zaustavljanja postavljeni na maksimalno 8000 generacija, pri čemu se gleda maksimalno 500 generacije bez promjene dobrote najbolje jedinke, te je maksimalno trajanje testa postavljeno na 60sec. Svaki test je ponovljen 10 puta.

Rezultati testiranja se mogu vidjeti na slikama Slika 4.12 i Slika 4.13. Testovi su opet podijeljeni na dvije skupine zbog većih razlika u rezultatima između funkcija. U ovom slučaju je opet Schwefelova funkcija predstavljala problem kod pronalaska minimuma funkcije, te je najbolja kombinacija operatora uspjela spustiti dobrotu tek na 8379.659, što je skoro potpuno jednako kao i kod binarnog prikaza. Druga funkcija koja je predstavljala problem je bila Rosenbrockova dolina, koja ipak nije bila toliki problem kao i Schwefelova funkcija, ali joj ipak dobrota nije uspješno smanjena ispod 17.853. Kod ovog prikaza rješenja kombinacija svih operatora križanja i mutacije također daje dobre rezultate.



Slika 4.12 Rezultati testiranja kombinacija operatora križanja i mutacije

Uzimajući u obzir sve funkcije, u prosjeku je najbolje prošla kombinacija aritmetičkog uniformnog križanja i neuniformne mutacije. U tablici Tablica 4.6 se mogu vidjeti rezultati za funkcije s obzirom na vjerojatnost neuniformne mutacije iz najbolje kombinacije operatora. Iz rezultata se može vidjeti da u prosjeku vjerojatnost mutacije koja daje najbolje rezultate je 0.2.



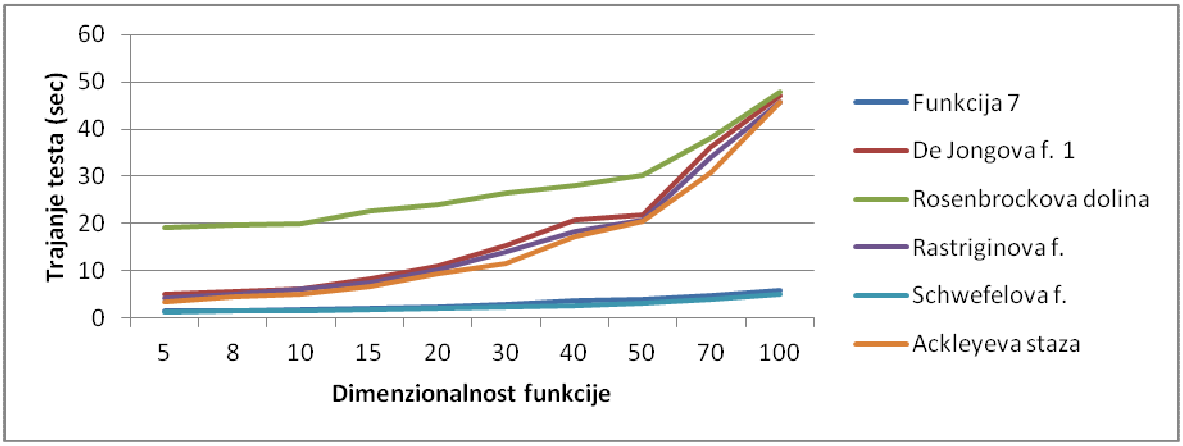
Slika 4.13 Rezultati testiranja kombinacija operatora križanja i mutacije

Tablica 4.6 Detaljni rezultati s obzirom na vjerojatnost mutacije iz najbolje kombinacije operatora

	Funkcija 7	De Jongova f. 1	Rosenbrockova dolina	Rastriginova f.	Schwefelova f.	Ackleyeva staza	Goldstein-Price f.
0,1	0,285587	1,14E-05	17,83297	0,002687	8379,66	0,027703	6,39E-04
0,2	0,292316	1,84E-05	17,78967	0,001934	8379,66	0,014424	1,84E-04
0,3	0,312981	1,01E-05	17,86179	0,003626	8379,659	0,017946	2,78E-04
0,4	0,31421	8,39E-06	17,87224	0,002241	8379,659	0,024845	0,002027
0,5	0,333766	1,09E-05	17,81453	0,002531	8379,658	0,023921	4,06E-04
0,6	0,302428	4,75E-06	17,8052	0,004311	8379,659	0,022645	2,82E-04
0,7	0,321998	3,35E-05	17,9244	0,002283	8379,659	0,016714	0,001052
0,8	0,323676	9,51E-06	17,96215	0,002193	8379,658	0,020957	1,66E-04
0,9	0,334278	2,28E-05	17,85534	0,001442	8379,658	0,015889	6,33E-04
1.0	0,297369	1,35E-05	17,95264	0,005911	8379,66	0,018968	9,98E-04

Kod utjecaja dimenzionalnosti na vrijeme trajanja testa, korištena je kombinacija aritmetičkog uniformnog križanja i neuniformne mutacije, uz vjerojatnost mutacije 0.2. Parametri zaustavljanja su postavljeni kao prije navedeno, te je svaki test ponovljen 20 puta. Niti ovdje nije provedeno ispitivanje na Goldstein-Price funkciji zbog prije navedenog razloga. Rezultati testiranja se mogu vidjeti na slici Slika 4.14. Zanimljivo je za uočiti kako funkciji 7 i Schwefelovoj funkciji treba tek oko 5-6sec kod dimenzije 100, dok svima ostalima treba 45-48sec. Razlog tome je što za te dvije funkcije algoritam vrlo brzo konvergira u minimum (neovisno o tome da li je globalni ili lokalni) te se ne promijeni nakon 500 generacija, nakon čega se prekida rad algoritma. Slično, kod Rosenbrockove doline se svako malo mijenja dobrota za male vrijednosti, zbog

čega ju u većini slučajeva zaustavi tek maksimalno postavljen broj generacija i zbog toga ovaj test traje dulje od svih ostalih.



Slika 4.14 Utjecaj dimenzionalnosti na vrijeme trajanja testa kod FP prikaza

4.4. Binarni prikaz vs prikaz broja s pomičnom točkom

Direktna usporedba rezultata iz najbolje kombinacije operatora križanja i mutacije kod ova dva prikaza se može vidjeti u tablici Tablica 4.7. Rezultati iz tablice predstavljaju najbolje (tj. najmanje) vrijednosti dobivene za pojedinu funkciju. Prema tim podacima, binarni prikaz u prosjeku daje bolje rezultate od FP prikaza, s jednom iznimkom, Goldstein-Price funkcijom.

Tablica 4.7 Usporedba rezultata testiranja za FP i binarni prikaz

	FP prikaz	Binarni prikaz
Funkcija 7	0,285587	0,00E+00
De Jongova f. 1	4,75E-06	1,58E-30
Rosenbrockova dolina	17,7853	6,24E-28
Rastriginova f.	0,001442	0,00E+00
Schwefelova f.	8379,659	8379,658

Ackleyeva staza	0,014424	4,44E-16
Goldstein-Price f.	0,000166	9,749103

Jedna od većih razlika između ta dva prikaza je brzina konvergiranja k lokalnom ili globalnom minimumu. Prosječni brojevi generacija potrebni za pronalazak minimuma kod testnih funkcija, gdje je korištena najbolja kombinacija operatora za pojedini prikaz se može vidjeti u tablici Tablica 4.8. Binarni prikaz vrlo brzo konvergira te u prosjeku za niti 100 generacija pronađe svoj minimum i teško se mijenja, što većinom znači čekanje operatora zaustavljanja koji gleda da li se mijenja dobrota najbolje jedinke. FP prikaz je potpuno drugačiji jer se nerijetko događa da dolazi do maksimalnog broja dopuštenih generacija, te mu je u prosjeku potrebno nekoliko tisuća generacija dok se ne ustabili. Dobrota kod FP prikaza se u blizini minimuma još uvijek mijenja, ali je ta promjena mnogo sporija, zbog čega se ne aktivira operator zaustavljanja koji gleda dobrotu najbolje jedinke, te se izvrše sve ili veći dio predviđenih maksimalnih generacija.

Tablica 4.8 Usporedba prosječnog broja generacija za pojedini prikaz

	FP prikaz	Binarni prikaz
Funkcija 7	2467,8	541,5
De Jongova f. 1	8000	530,9
Rosenbrockova dolina	3150	541,8
Rastriginova f.	630,9	519,2
Schwefelova f.	3141,9	539,1
Ackleyeva staza	797,1	540,1
Goldstein-Price f.	2467,8	517

Utjecaj na brzinu konvergiranja kod binarnog i FP prikaza imaju uvelike implementirani operatori mutacije i križanja, ali i funkcija dobrote koja ovisi o optimizacijskoj funkciji. Npr. u prosjeku će promjena jednog bita kod binarnog

prikaza imati puno manji utjecaj na vrijednost dobrote i na raznolikost populacije, nego potpuna promjena jednog elementa kod FP prikaza na neku nasumičnu vrijednost koja može biti unutar cijelog zadanog intervala. Zbog toga je binarni prikaz stabilniji i brži kod konvergiranja, za razliku od FP prikaza. Kod Goldstein-Price funkcije se promatraju samo dva elementa, tj. samo dvije dimenzije, kod oba prikaza. To znači da ovdje postoji puno veći utjecaj tog jednog elementa na dobrotu, nego kod drugih, višedimenzionalnih, funkcija. Zbog tromosti binarnog niza se ne uspijeva poboljšati najbolja jedinka, te algoritam puno brže završava sa slabijom dobrotom, nego kod FP prikaza.

5. Primjer korištenja okruženja ECF

U ovom dijelu rada će se detaljno opisati jedan primjer korištenja okruženja ECF. Koristit će se prikaz preko permutacijskog niza za rješavanje TSP problema s 29 gradova. Za operatore križanja i mutacije će se koristiti najbolja kombinacija sa testiranja, OX križanje i inverzna mutacija. Ključno je da se u direktoriju sa programom nalazi konfiguracijska datoteka „Parametri.xml“.

5.1. Izgradnja konfiguracijske datoteke

Prije kodiranja samog programa, potrebno je definirati parametre unutar konfiguracijske datoteke. Parametri vezani uz prikaz i operatore križanja i mutacije su prethodno objašnjeni, pa će ovdje biti samo nabrojani. Za ovaj primjer je korišten problem 29 gradova, što znači da je dimenzija permutacijskog niza iznosila 29, jer dimenzija permutacijskog niza predstavlja broj gradova. Zatim su postavljene vrijednosti vjerojatnosti izvođenja operatora križanja i mutacije. Vjerojatnost mutacije je postavljena na 1, isto kao i vjerojatnost križanja.

Nakon toga su podešeni operatori zaustavljanja. Oni su standardno postavljeni na maksimalno trajanje algoritma od 60sec, na maksimalan broj generacija od 8000, i na maksimalan broj generacija bez promjene od 500.

Postoje još neki parametri u konfiguracijskoj datoteci, ali se oni neće objašnjavati ovdje jer izlaze van granica ovog rada. Oni se mogu samo prepisati.

Primjer konfiguracijske datoteke se može vidjeti na slici Slika 5.1.

```
<ECF>
    <Algorithm> <!-- moze biti samo jedan, ali ovdje su navedeni svi postojeci -->
        <SteadyStateTournament>
            <Entry key="tsize">3</Entry> <!-- tournament size -->
        </SteadyStateTournament>
    </Algorithm>
    <Genotype>
        <Permutation>
            <Entry key="size">29</Entry>

            <Entry key="mut.inv">1.0</Entry>
            <Entry key="crx.OX">1</Entry>
        </Permutation>
    </Genotype>
    <Registry>
        <Entry key="randomizer.seed">0</Entry> <!-- 0 uses time(NULL) (default: 0) -->
        <Entry key="population.size">100</Entry> <!-- number of individuals (default: 100) -->
        <Entry key="population.demes">1</Entry> <!-- number of demes (default:1)-->
        <Entry key="migration.freq">0</Entry> <!-- individuals are exchanged each 'freq' generations (default: none) -->
        <Entry key="migration.number">2</Entry> <!-- number of individuals to be sent to another deme (default: 1) -->
        <Entry key="mutation.indprob">0.3</Entry> <!-- individual mutation probability (regardless of the algorithm!) (default: 0.3) -->
        <Entry key="mutation.genotypes">all</Entry> <!-- if there are multiple genotypes, which to mutate? 'random': a random one, all: mutate all (default: random) -->
        <Entry key="crossover.genotypes">all</Entry> <!-- if there are multiple genotypes, which to cross? 'random': a random pair, 'all': all pairs (default: random) -->
        <Entry key="term.maxgen">8000</Entry> <!-- max number of generations (default: none) -->
        <Entry key="term.maxtime">60</Entry> <!-- max number of seconds to run (default: none) -->
        <Entry key="term.stagnation">500</Entry><!-- max number of consecutive generations without improvement (default: 5000 / pop_size) -->
        <Entry key="log.level">4</Entry> <!-- log level; valid values are 1 (minimal) to 5 (verbose) (default: 3) -->
        <Entry key="log.filename">log.txt</Entry> <!-- log filename (default: none) -->
        <Entry key="log.frequency">1</Entry> <!-- log only every 'frequency' generations (default: 1) -->
        <Entry key="milestone.filename">out.txt</Entry> <!-- milestone file (if stated) stores all the population (default: none) -->
        <Entry key="milestone.interval">0</Entry> <!-- milestone saving interval in generations; 0: save only at the end (default: 0) -->
    </Registry>
</ECF>
```

Slika 5.1 Primjer konfiguracijske datoteke

5.2. Kodiranje algoritma

Prvi korak je u *main* metodi napraviti objekt razreda koji će pokrenuti algoritam i proslijediti glavnoj metodi razreda parametre (*String[] args*) iz *main* metode. Taj razred mora naslijediti sučelje *IEvaluate*. Nasljeđivanjem navedenog sučelja potrebno je implementirati 4 metode:

1) *evaluate(Fitness fitness)* – ključna metoda u kojoj se definira algoritam računanja dobrote
2) *createFitness()* – metoda u kojoj se „stvara“ vrsta dobrote, tj. određuje se da li će cilj algoritma biti smanjiti dobrotu jedinke, ili povećati. Ako se kreira objekt *FitnessMin*, tada će se smanjivati, a ako se kreira *FitnessMax* tada će se povećavati dobrota.
3) *registerParameters()* i *initialize()* metode se ne koriste za navedene probleme u ovom radu, pa je dovoljno samo da ih se definira, konkretna implementacija može izostati.

Jedan primjer gore navedenih metoda se može vidjeti na slici Slika 5.2. Prva linija koda u metodi *evaluate()* znači da se uzima u obzir samo prvi genotip od svih navedenih u konfiguracijskog datoteci. Da se stavi cijeli kod iz metode *evaluate()* u petlju i pazi koji se genotip gleda u trenutnoj iteraciji, tada bi se izvodila metoda za sve navedene genotipe. Navedena implementacija se odnosi na FP prikaz rješenja za problem minimizacije De Jongove funkcije 1.

```
@Override
public void evaluate(Fitness fitness) {
    FloatingPoint fp = (FloatingPoint) fitness.getIndividual().getGenotype(0);
    double fitnes = 0;
    for (int i = 0; i < fp.getnDimension(); i++) {
        double value = fp.getNumber(i);
        fitnes += value * value;
    }
    fitness.setValue(Math.abs(fitnes));
}

@Override
public Fitness createFitness() {
    return new FitnessMin();
}

@Override
public void registerParameters() {
}

@Override
public void initialize() {
}
```

Slika 5.2 Primjer implementacija metoda za Jongovu funkciju 1

Nakon implementiranja gore navedenih metoda potrebno je implementirati metodu unutar navedenog razreda koja će pokrenuti algoritam. Ideja je da se prvo napravi objekt *State* te mu se preko konstruktora proslijedi trenutni razred. Nakon toga se pokrene metoda *initialize(args)* iz objekta state, te joj se proslijede argumenti iz *main* metode. Nakon toga se pokrene iz objekta state metoda *run()*, koja će zapravo upogoniti algoritam definiran preko konfiguracijske datoteke. U konzolu će se ispisivati trenutno stanje algoritma, onoliko detaljno koliko je definirano preko parametra „log.level“ u konfiguracijskoj datoteci.

Zaključak

U sklopu okruženja ECF implementirana su tri prikaza jedinke: binarni niz, permutirani niz te prikaz broja s pomičnom točkom. Njihove mogućnosti i učinkovitosti operatora križanja i mutacije specifičnih za pojedini prikaz ispitane su na raznim problemima.

Na temelju rezultata ispitivanja, nedvojbeno se može zaključiti da rad genetskog algoritma uvelike ovisi o problemu koji se rješava, budući da nijedna kombinacija operatora križanja i mutacije za pojedini prikaz nije uspjela pronaći optimalno rješenje na svim problemima na kojima su se testirali. Zbog toga je vrlo važno prije implementiranja neke vrste genetskog algoritma i njegovih operatora proučiti koje su najbolje kombinacije operatora i prikaza jedinke davale najbolja rješenja na sličnim primjerima. Ali u prosjeku se, prema dobivenim rezultatima, može zaključiti da binarni prikaz daje bolje rezultate od prikaza s pomičnom točkom.

U radu su prikazane najbolje kombinacije operatora za pojedini problem. Pokazalo se da se kod problema s eksponencijalnim širenjem prostora pretraživanja rješenja na sličan način povećava trajanje genetskog algoritma, jer veliki dio trajanja genetskog algoritma ovisi o funkciji dobrote, koja usmjerava genetski algoritam prema minimumu. Isto tako, u slučaju neeksponencijalnog povećavanja prostora za pretraživanje se produljuje trajanje genetskog algoritma, što potvrđuje koliki utjecaj funkcija dobrote ima na trajanje algoritma.

Literatura

[1] Marin Golub, „Genetski algoritam“, prvi dio, http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf, 12.03.2010.

[2] Wikipedia, „Genetic algorithm“, http://en.wikipedia.org/wiki/Genetic_algorithm, 12.03.2010.

[3] Abdollah Homaifar, Joseph Tumer, Samia Ali, A.H., J.T., S.A., „The N-queens problem and genetic algorithms“, Machine Intellifence Laboratory, Department of Electrical Engineering, North Carolina A&T State University, 1992

[4] Marin Golub, „Paralelni genetski algoritmi“, http://www.zemris.fer.hr/~golub/ga.html, 2004

[5] Vedran Lovrečić, diplomski rad, „Podešavanje parametara genetskog algoritma“, 2006

[6] Michalewicz, Z, „Genetic Algorithms + Data Structures = Evolutionary Programs“, Springer-Verlag, Berlin, 1992.

[7] GEATbx: Example Functions (single and multi-objective functions) 2 Parametric Optimization, http://www.geatbx.com/docu/fcnindex-01.html, 27.03.2010.

[8] Bruce Eckel, „Thinking in Java“, Prentice Hall, Boston, 2006.

Analiza učinkovitosti prikaza rješenja u okruženju za evolucijsko računanje

Sažetak

Da bi se olakšala izgradnja genetskog algoritma u programskom jeziku Java, izgrađuje se okruženje za evolucijsko računanje (ECF) u Javi. U sklopu ovog rada implementirana su tri prikaza jedinke: binarni niz, permutirani niz i prikaz broja s pomičnom točkom. Uz to su i implementirani razni operatori križanja i mutacija, prilagođeni pojedinoj implementaciji prikaza.

Implementacije su ispitane na različitim kombinatornim i kontinuiranim optimizacijskim problemima, prilagođenima za određenu implementaciju. Prikazani su rezultati ispitanih kombinacija operatora križanja i mutacija za pojedini problem te utjecaj dimenzionalnosti problema na trajanje izvođenja genetskog algoritma. Rezultati sugeriraju da u prosjeku binarni niz daje bolje rezultate od FP prikaza, ali isto tako da učinkovitost algoritma uvelike ovisi o problemu koji se rješava s njim.

Ključne riječi: umjetna inteligencija, genetski algoritam, genetski operatori, FP prikaz, binarni niz, permutirani niz, kombinatorički problem, problem N kraljica, TSP, kontinuirane optimizacijske funkcije

Analysis of genotypes in evolutionary computational framework

Summary

To simplify coding of genetic algorithm in programming language Java, we are building evolutionary computational framework in Java. As part of this paper there are three implemented genotypes: binary, permutation array and floating point. For every genotype there are few operators for crossing over and mutation implemented.

Implementations are tested on various combinatorial and continuous optimization problems, designed for a specific implementation. The results for combinations of operators for crossing over and mutation for different implemented problems are shown. For each problem the impact of *dimension*ality on duration of genetic algorithm is also tested. The results suggest that on average binary genotype gives better results than floating point, but also that the efficiency of algorithm is greatly influenced by the problem that algorithm is solving.

Keywords: artificial intelligence, genetic algorithm, genetic operators, floating point genotype, binary genotype, permutation array genotype, combinatorial problems, the N queens problem, TSP, single and multi-objective functions

Skraćenice

1. ECF *Evolutionary computation framework* okruženje za evolucijsko računanje
2. PMX *Partially matched crossover* vrsta operatora križanja
3. OX *Order crossover* vrsta operatora križanja
4. PBX *Position based crossover* vrsta operatora križanja
5. FP *Floating Point* prikaz broja s pomičnom točkom
6. TSP *Travelling salesman problem* problem trgovačkog putnika