SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD
br. 1223

Marko Božiković

Zagreb, 2000.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD
br. 1223

GLOBALNI PARALELNI GENETSKI ALGORITAM

Marko Božiković

Zagreb, 2000.

SADRŽAJ

1. UVOD

Genetski algortmi su heurističke metode optimiranja zamišljeni kao imitacija prirodne evolucije. Prirodna evolucija neke vrste se može promatrati kao proces optimiranja, potraga za jedinkom koja je najbolje prilagođena uvjetima koji vladaju u okolini. Prirodnom selekcijom se biraju jedinke koje će preživjeti i stvoriti potomstvo. Na taj način populacija napreduje i sve se bolje prilagođava okolini. Genetski algoritmi imitiraju prirodnu evoluciju tako da proces koji se optimira predstavlja okolinu u kojoj žive jedinke – ulazni podaci za proces. 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, repodukcije i manipulacije genetskim materijalom se ponavlja sve dok nije zadovoljen uvjet zaustavljanja genetskog algoritma.

Rad na genetskim algoritmima je započeo John H. Holland krajem 60-ih godina u sklopu svojih istraživanja adaptivnih umjetnih sustava. Tijekom posljednja tri desetljeća su se genetski algoritmi pokazali kao moćne i općenite metode za rješavanje raznolikih problema na područjima inžinjerske prakse. Razlog tome leži u njihovoj jednostavnosti implementacije i prilagodljivosti velikom broju problema. Usporedno sa širenjem upotrebe genetskih algoritama rastu i napori da se njihov rad pokuša svesti na teoretske osnove. No, kako su u osnovi heurističke metode, analiza rada genetskih algoritama sadrži složene proračune iz područja teorije vjerojatnosti i statistike, pa veliki dio načina rada genetskih algoritama još uvijek nije dovoljno kvalitetno teorijski opisan.

Prema načinu rada genetski algoritmi spadaju u metode *usmjerenog slučajnog pretraživanja prostora rješenja* (eng. *guided random search techniques*). U tu grupu spadaju još i metode koje su temeljene na sličnim principima: *evolucijeske strategije* (eng. *evolutionary strategies*) i *simulirano kaljenje* (eng. *simulated annealing*). Osnovna snaga tih metoda u odnosu na razne deterministčke postupke optimiranja je mogućnost određivanja globalnog optimuma u višemodalnom prostoru[1]. Ali, za razliku od deterministčkih metoda, gdje je uvijek moguće dobiti rješenje sa željenom točnošću, stohastičke metode ne garantiraju pronalaženje globalnog optimuma, kao ni traženu točnost.

U ovom radu se obrađuje *globalni paraelni genetski algoritam* i njegova primjena na optimiranje funkcija realne varijable i rješavanje problema *N* kraljica[2].

Globalni paralelni genetski algoritam je jedna od strategija paralelizacije genetskih algoritama koje su se razvojem višeprocesorskih računala počele intenzivno istraživati u svrhu ubrzavanja rada genetskog algoritma. Zbog velikog broja operacija koje izvršava, genetski algoritam je spor u usporedbi sa specijaliziranim tehnikama rješavanja problema, pa je mogućnost paralelizacije vrlo važna za napredak genetskih algoritama. Osnovna ideja globalnog paralelnog genetskog algoritma je da se poslovi genetskog algoritma koji se

[1] Prostor sa više lokalnih optimuma.

[2] Kako rasporediti n kraljica na šahovsku ploču dimenzija $n \times n$ tako da se međusobno ne napadaju.

mogu izvoditi usporedno, rasporede na nekoliko procesora, koje kontrolira i zadaje im zadatke jedan glavni procesor, koji ujedno izvršava dijelove genetskog algoritma koje nije moguće paralelizirati. Time je moguće postići značajna ubrzanja uz zadržavanje jednostavnosti izvedbe genetskog algoritma.

Zbog svoje jednostavnosti i robusnosti genetski su algoritmi vrlo pogodni za optimiranje funkcija realne varijable. Budući da ne postavljaju zahtjeve na oblik funkcije (neprekinutost, derivabilnost, itd.), nije potrebno vršiti prilagodbu algoritma za pojedinu funkciju, čime je postignuta visoka robusnost.

Kako u mnogim kombinatoričkim problemima prostori rješenja rastu vrlo brzo sa dimenzijom problema, za mnoge takve probleme ne postoje determinističke metode koje su ih sposobne rješavati u realističnom vremenskom intervalu. I u ovom području genetski algoritmi pokazuju svoju snagu zbog svoje sposobnosti pretrage velikih prostora rješenja.

2. PRIRODNA EVOLUCIJA

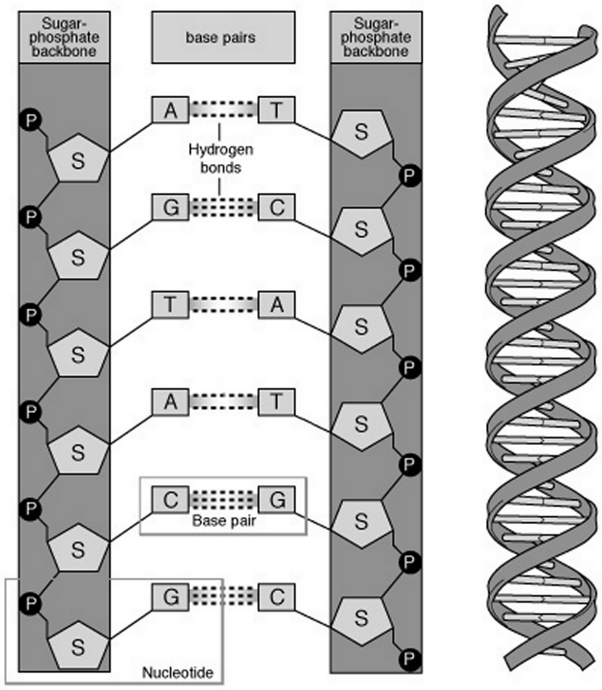
Iako se teorija evolucije u biologiji znanstveno formira tek u 19. st., već se sredinom 18. st. javljaju ideje koje se protive tadašnjem kreacionističkom shvaćanju svijeta. Tako s jedne strane C. Linné u svom djelu *Sustav prirode* (*Systema naturae,* 1735) strogo zastupa načelo: "Vrsta ima onoliko koliko ih je od početka stvoreno", dok G. Buffon u djelu *Prirodopis* (*Histoire naturelle*, 1749) iznosi ideju o postupnoj promjeni vrsta. Njegova teorija, temeljena na izučavanju velikog broja suvemenih biljaka i životinja, te opsežnog paleontološkog materijala, tvrdi da su "sve životinje proizašle od jedne životinje koja se tjekom vremena mijenjala i usavršavala, te proizvela sve životinjske rodove". Zoolog B. Lacépède[3] sličnosti među organizmima tumači zajedničkim pretkom, dok su međusobne razlike rezultat promijenjivih utjecaja okoline. On također spominje i nadživljavanje "bolje obdarenih" vrsta, čime se približava Darwinovoj teoriji evolucije. Na prirodoslovce 18. st. je utjecala i Leibnitzova filozofija neprekinutosti ("zakon kontinuiteta"), prema kojoj priroda ne pravi skokove, već je sve povezano nizom postupnih prijelaza.

Prvu cjelovitu evolucijsku teoriju je početkom 19. st. razradio J.B. Lamarck. Prvi put svoju ideju o zajedničkom podrijetlu organizama i njihovom postupnom razvoju iznosi u djelu *Sustav životinja bez kralježnice* (*Système des animaux sans vertèbres*, 1801), dok je u svom poznatom djelu *Filozofija zoologije* (*Philosophie zoologique*, 1809) razrađuje u sveobuhvatnu evolucijsku teoriju. Osnova Lamarckove teorije je polagan i kontinuiran proces preobrazbe vrsta u prirodi. Proces promjene vrste je uvjetovan promjenama okoline, čime se mijenjaju potrebe životinje, pa ona stiče nove navike. U skladu sa stečenim navikama, životinja više koristi određene organe, koji se stoga jače razvijaju, dok s druge strane neupotrebljavani organi slabe i iščezavaju (npr, oči u krtice). Promjene nastaju zbog volje životinje, te njenog stremljenja za zadovoljenjem svojih potreba i navike, te su uvijek u skladu sa okolinom, a životinje ih prenose na potomstvo.

Ch. Darwin je za razvoj svoje (povijesno najpoznatije) teorije evolucije sredinom 19. st. imao mnogo povoljniji teren, jer u to vrijeme biologija raspolaže velikim brojem činjenica koje potkrepljuju teoriju evolucije živoga svijeta. Tako su postignuti značajni rezultati na područjima komparativne anatomije, komparativne embriologije (K.E. Baer ustanovio sličnosti zametaka u kralježnjaka). T. Schwann 1839. otkriva jedinstvo stanične građe svih živih bića, i osniva nauku o stanici – citologiju. Ch. Lyell u svom djelu *Načela geologije* (*Principles of Geology*, 1831) postupne promjene Zemlje tumači polaganim i kontinuiranim djelovanjem prirodnih faktora (sunca, vjetra, vode, itd.). Darwin je teoriju evolucije predstavio u svom glavnom djelu, *O podrijetlu vrsta posredstvom prirodne selekcije ili o održavanju povlaštenh rasa u borbi za opstanak* (*On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle of Life*, 1859). Njegova se teorija evolucije prvenstveno oslanja na prirodni odabir, selekciju, koja podupire jedinke bolje prilagođene uvjetima okoline, dok slabije prilagođene jedinke odumiru. Faktore koji utječu na proces selekcije Darwin dijeli na vanjske (okolina) i unutarnje (kasnije nazvane genetičkima). Darwin je također dokazao da evolucija ima adaptivni karakter, te da su karakteristike organizama uvjetovane razvojem uvjeta u okolini kroz povijest. Suvremene evolucijske teorije se najvećim dijelom oslanjaju na darvinizam, uz manje izmjene koje su rezultat recentnih istraživanja.

[3] Autor poglavlja o ribama u Buffonovom *Prirodopisu*.

Osnovni evolucijski procesi se odvijaju unutar populacije određene vrste. Proces evolucije započinje kada se poremti genetska ravnoteža unutar populacije, zbog promjena u okolini, ili unutar same populacije. Osnovne sile evolucije danas se dijele u tri skupine: *mutacija*, *prirodno odabiranje* i *genetski drift*. *Mutacijom* se mijenjaju nasljedni faktori u populaciji, te se razlikuju četiri osnovna izvora: *mutacija gena*, *mutacija kromosoma*, *rekombinacija gena* i *rekombinacija kromosoma*. Mutacijom gena i mutacijom kromosoma (promjena broja ili građe kromosoma) se proširuje genetska raznolikost populacije. Rekombinacija gena je križanje među dvjema vrstama homozigotnih roditelja i stvaranje heterozigotnog potomstva, dok se rekombinacija kromosoma odvija tijekom mejoze i poznata je pod nazivom *crossing-over*. Druga sila evolucije, selekcija, predstavlje prirodni izbor između jedinki – nosilaca različito vrijednih nasljednih informacija u okvirima zatjeva okoline. Upravo je princip selekcije najveći Darwinov doprinos teoriji evolucije, koji je koristio pojam *preživljavanje najsposobnijih* (eng. *survival of the fittest*) za borbu za opstanak među pojedinim organizmima unutar populacije. Suvremene teorije promatraju prirodnu selekciju kao silu koja djeluje na cijele populacije (Darwin je selekciju primjenjivao na jedinke), te je smatraju osnovnom snagom koja uzrokuje promjene u genetskoj ravnoteži, a vrši se kroz utjecaj faktora iz okoline (temperature, oborine, prirodni neprijatelji, natjecanje oko hrane i životnog prostora, itd.). Genetski drift, kao treća osnovna sila evolucije djeluje na malim populacijama i uzrokuje ustaljivanje neutralnih ili neadaptivnih svojstava. Kao primjer ove pojave se uzimaju Darwinove zebe na otočju Galápagos: na malom području živi 13 različitih vrsta zeba, sa velikom raznolikošću oblika i veličina kljunova, ovisno o prehrambenim navikama pojedine vrste.



Slika 2.1 Struktura molekule DNA

Osnovni nositelj genetske informacije je molekula deoksiribonukleinske kiseline (DNK). Njezinu kemijsku strukturu su 1953. godine otkrili J.D. Watson i F.H.C. Crick. Molekula DNK ima oblik dvostruke spirale građene od fosforne kiseline i šećera (slika 2.1), a okomito na smjer protezanja lanca uz deoksiribozu svakog nukleotida vezana je dušična baza, koja ostvaruje vodikovu vezu sa dušičnom bazom drugoga lanca.

Dušične baze su: adenin (A), gvanin (G), timin (T) i citozin (C). Adeninske baze se vežu sa timiniskim, a gvaninske sa citozinskim bazama. Te četiri dušične baze su nosioci jedinične informacije u prirodi, slično bitovima (0 ili 1) u digitalnim računalima. Molekule DNK su organizirane u *kromosome*. Svaki kromosom sadrži dvije komplementarne niti DNK, što omogućuje prijenos genetske informacije pri diobi stanice. Kada se stanica treba podijeliti, niti DNK u kromosomima se razmataju i razdvajaju, a za baze na rastavljenim nitima DNK se vežu slobodne baze koje se nalaze u jezgri stanice. Na taj je način genetska informacija sačuvana i udvostručena, čime se dobijaju dvije stanice sa jednakim genetskim sadržajem.

Tijekom šezdesetih godina 20. st. otkriven je način na koji su genetske informacije kodirane unutar molekule DNK. Naime, danas je poznato dvadeset aminokiselina, koje su osnovni građevni dijelovi bjelančevina (grade stanične strukture, enzime i neke hormone). Kako bi se izgradila određena bjelančevina, potrebno je kreirati lanac aminokiselina točno određenim redoslijedom. Upravo su ti redoslijedi stvaranja lanaca aminokiselina kodirani u molekulama DNK. Budući da jedan nukleotid ima 4 moguće kombinacije, potrebna su tri nukelotida da bi se dobio dovoljan broj mogućih kombinacija za kodiranje 20 različitih aminokiselina. Tablica 2.1 prikazuje genetsku šifru.

Tablica 2.1 Tablica genetske šifre

Prvo slovo	Drugo slovo				Treće slovo
	U	C	A	G	
U	Phe	Ser	Tyr	Cys	U
	Phe	Ser	Tyr	Cys	C
	Leu	Ser	STOP	STOP	A
	Leu	Ser	STOP	Trp	G
C	Leu	Pro	His	Arg	U
	Leu	Pro	His	Arg	C
	Leu	Pro	Gln	Arg	A
	Leu	Pro	Gln	Arg	G
A	Ile	Thr	Asn	Ser	U
	Ile	Thr	Asn	Ser	C
	Ile	Thr	Lys	Arg	A
	Met ili START	Thr	Lys	Arg	G
G	Val	Ala	Asp	Gly	U
	Val	Ala	Asp	Gly	C
	Val	Ala	Glu	Gly	A
	Val	Ala	Glu	Gly	G

Ala-alanin
Arg-arginin
Asp-asparaginska kiselina
Asn-asparagin
Cys-cistein
Gln-glutamin
Glu-glutaminska kiselina
Gly-glicin
His-histidin
Ile-izoleucin
Leu-leucin
Lys-lizin
Met-metionin
Phe-fenilanin
Pro-prolin
Ser-serin
Thr-treonin
Trp-triptofan
Tyr-tirozin
Val-valin

Tako, na primjer, niz AUG CUU GCU AGU UUC UAA predstavlja lanac aminokiselina Leu – Ala – Ser – Phe. AUG označava početak, a UAA kraj lanca aminokiselina. Na taj su način u molekulama DNK zapisane sve informacije o jedinki određene vrste (npr, boja očiju, pozicije i veličine organa i dr.) Jedan gen nosi informaciju za jedno svojstvo, i smatra se da svaki gen sadrži oko 1000 nuklotida. Kao primjer, cjelokupni genetski materijal jedne žabe se sastoji od oko $3.2*10^9$ nukleotida, što iznosi približno 0.8GB podataka u binarnom zapisu.

3. UVOD U GENETSKE ALGORITME

3.1 GENETSKI ALGORITAM: IMITACIJA PRIRODE

Evolucija neke populacije se može promatrati kao proces optimizacije: potragu za jedinkom koja je najbolje prilagođena uvjetima koji vladaju u okolini. Jedinke sa kvalitetnijim nasljednim svojstvima (koja se prenose genima) u borbi za opstanak imaju veće šanse za stvaranje potomstva od slabijih jedinki. Na taj način kvalitetniji genetski materijal dominira u populaciji, dok lošiji odumire. Nadalje, pri reprodukciji dolazi do miješanja genetskih materijala roditelja, pa djeca imaju dijeljene karakteristike svojih roditelja. Na taj se način u svakoj generaciji dobija novi skup genetskog materijala, gdje su neke jedinke bolje, a neke lošije od onih iz prethodne generacije, ali ukupna kvaliteta genetskog materijala populacije, kao i kvaliteta najbolje jedinke, rastu. Genetske algoritme je razvio J. Holland kao pojednostavljenu imitaciju prirodne evolucije koja se odvija na apstraktnim jedinkama, sa glavnom idejom istraživanja *robusnosti* umjetnih sustava. Robusnost umjetnih sustava je višestruko značajna: skupa prilagođavanja i promjene takvih sustava je potrebno rjeđe obavljati ili ih se može potpuno eliminirati; adaptivni sustavi svoj posao mogu obavljati dulje i pouzdanije. A kao najbolji primjer robusnosti se mogu uzeti biološki sustavi: samopopravljivost, samonavodivost, zalihnost i reprodukcija su samo neke od odlika koje postoje u prirodi, dok se iste nalaze samo u tragovima kod umjetnih sustava.

Genetski algoritmi imitiraju osovne procese prirodne evolucije: problem[4] koji se rješava predstavlja okolinu u kojem živi populacija jedinki. Sve jedinke (obično se nazivaju kromosomima) neke populacije su predstavljene određenom podatkovnom strukturom (broj, niz, matrica, stablo, itd.), koja predstavlja kodirano potencijalno rješenje zadanog problema, što je analogno genetskoj informaciji živog organizma. Svakom kromosomu se pridjeljuje mjera kvalitete, kojom se iskazuje koliko je taj kromosom blizu traženog rješenja. Ta se mjera kvalitete naziva *dobrota* (eng. *fitness*), a funkcija za određivanje dobrote se naziva *funkcija dobrote*[5]. Nakon toga se iz stare populacije stvara nova tako da se postupkom kojim se imitira prirodna selekcija odabiru jedinke koje će preživjeti. Na pripadnicima nove generacije se zatim primjenjuju genetski operatori, koji se dijele na unarne, koji djeluju na genetskom materijalu jednog kromosoma i na operatore višeg reda, koji stvaraju nove kromosome kombiniranjem nekoliko starih kromosoma ("roditelja"). Cijeli taj postupak (ocjena kvalitete kromosoma, selekcija i primjena genetskih operatora) se ponavlja sve dok nije zadovoljen uvijet zaustavljanja algoritma, pa se osnovna struktura genetskog algoritma može prikazati slikom 3.1.

[4] Minimizacija/maksimizacija funkcije, optimiranje radnog plana, izrada rasporeda, i dr.
[5] Funkcija dobrote može biti vrijednost funkcije koja se optimira, profit kod optimiranja radnog plana, itd.

```
Genetski_algoritam
početak
      g = 0
      kreiraj početnu populaciju P(0)
      dok nije zadvoljen uvjet zaustavljanja
      početak
            g = g + 1
            selektiraj generaciju P'(g) iz P(g-1)
            generiraj generaciju P(g) iz P'(g)
                  koristeći genetske operatore
      kraj
kraj
```

Slika 3.1 Osnovna struktura genetskog algoritma

Pri inicijalizaciji genetskog algoritma se kreira i početna populacija jedinki. Kromosomi se obično kreiraju slučajnim odabirom iz domene problema. Kao što je već spomenuto, genetski algoritam se izvršava sve dok nije zadovoljen uvjet zaustavljanja. Uvjet zaustavljanja se određuje ovisno o problemu koji se rješava i o uvjetima u kojima se problem rješava. Tako uvjet zaustavljanja može biti da dobrota 95% jedinki ne odstupa od nekog ε, istek zadanog vremenskog intervala, određeni broj generacija ili pronalaženje točnog rješenja[6].

Kod implementacije genetskog algoritma osnovni je problem pravilni odabir načina predstavljanja kromosoma, načina dekodiranja kromosoma, čime se dobivaju ulazni podaci za funkciju dobrote, te određivanje same funkcije dobrote.

3.2 PRIKAZ RJEŠENJA

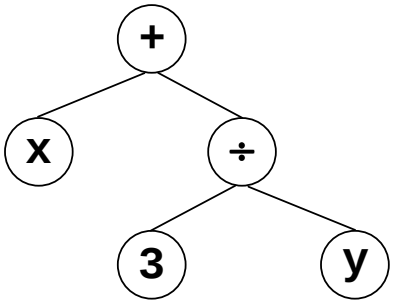
Kromosom predstavlja jedno potencijalno rješenje problema koji se rješava, kodirano na određeni način i u njemu su sadržani svi podaci koji obilježavaju jednu jedinku. Budući da način kodiranja rješenja može znatno utjecati na učinkovitost genetskog algoritma, pravilan izbor strukture podataka koja će 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 efikasnosti 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[7], u raznim kombinatoričkim problemima se koriste nizovi brojeva, polja, binarna stabla, i sl.

```
(3 7 2 6 1 4 5 8)
```

Slika 3.2 Primjer prikaza permutiranim nizom

[6] Obično se primjenjuje kod kombinatoričkih problema.
[7] U implementaciji je to obično broj s pomičnim zarezom jednostruke ili dvostruke preciznosti.



Slika 3.3 Primjer prikaza binarnim stablom

3.2.1 PRIKAZ RJEŠENJA BINARNIM KODOM

Prikaz binarnim kodom će biti pobliže opisan, budući da je uz njega vezana većina teorije o genetskim algoritmima. Kromosom prikazan binarnim kodom obično predstavlja binarno kodiranu vrijednost broja $x \in [dg, gg]$. Duljina kromosoma, odnosno broj bitova u binarnom vektoru određuje preciznost. Tako je u kromosom duljine n bitova moguće zapisati binarni broj u intervalu $[0, 2^n-1]$, gdje binarni vektor $v(0)=[00...0]$ predstavlja vrijednost $x=dg$, a binarani vektor $v(2^n-1)=[11...1]$ predstavlja vrijednost $x=gg$.U općenitom slučaju, binarni vektor $v(b)=[B_{n-1}B_{n-2}...B_1B_0]$ predstavljen je binarnim brojem b:

$$b = \sum_{i=0}^{n-1} B_i 2^i \tag{3.1}$$

Dekodiranjem kromosoma dobija se vrijednost od x iz intervala $[dg, gg]$, koje predstavlja potencijalno rješenje problema:

$$x = dg + \frac{b}{2^n - 1}(gg - dg) \tag{3.2}$$

S druge strane, kodiranje danog broja x iz intervala $[dg, gg]$ se obavlja prema izrazu:

$$b = \frac{x - dg}{gg - dg}(2^n - 1) \tag{3.3}$$

Pri optimiranju funkcija važno je odrediti preciznost rješenja. Neka je za neki interval $[dg, gg]$ u kojem se traži optimum zadana preciznost p, $p \in N$ (što znači da neko rješenje $x \in [dg, gg]$ odstupa od točnog rješenja za maksimalno 10^{-p}, odnosno da je rješenje x točno na p decimala). Tada za zadanu preciznost p i duljinu kromosoma n mora biti zadovoljena nejednakost:

$$(gg - dg) \cdot 10^p \leq 2^n - 1 \tag{3.4}$$

Uz malo sređivanja dobije se nejednakost za minimalnu duljinu kromosoma *n* za zadanu preciznost *p*:

$$n \geq \frac{\log\left[(gg - dg)\cdot 10^{p} + 1\right]}{\log 2} \tag{3.5}$$

Tablica 3.1 daje usporedni prikaz potrebne duljine kromosoma ako su zadane preciznost *p* i duljina intervala *[dg, gg]*.

Tablica 3.1 Potrebna duljina kromosoma
za zadane preciznost *p* i duljinu intervala *[dg, gg]*

preciznost na p decimala	Δx	gg - dg			
		1	10	100	1000
1	10^{-1}	4	7	10	14
2	10^{-2}	7	10	14	17
3	10^{-3}	10	14	17	20
4	10^{-4}	14	17	20	24
6	10^{-6}	20	24	27	30
9	10^{-9}	30	34	37	40
12	10^{-12}	40	44	47	50
15	10^{-15}	50	54	57	60
30	10^{-30}	100	103	107	110

3.3 FUNKCIJA DOBROTE

Uloga funkcije dobrote[8] u optimizaciji je ocjena kvalitete pojedine jedinke. U najjednostavnijem obliku, funkcija dobrote je ekvivalentna funkciji *f* koju se optimira. Tako je funkcija dobrote za binarni vektor *v* koji predstavlja realni broj $x \in [dg, gg]$ definirana kao:

$$dobrota(v) = f(x) \tag{3.6}$$

Kao što je već spomenuto, dobrota jedinke predstavlja njezinu kvalitetu. Što je jedinka kvalitetnija, to je vjerojatnije da će preživjeti, te križanjem stvoriti potomstvo, tako da funkcija dobrote igra ključnu ulogu u procesu selekcije jedinki. U općenitom slučaju se na funkciju dobrote ne postavljaju ograničenja, mada je u praksi ponekad potrebno primjeniti određene transformacije, kako bi se zadovoljili neki uvjeti koje može postavljati proces selekcije, kao što je prikazano u poglavlju 3.4. U kreiranju efikasnog genetskog algoritma,

[8] U literaturi se još naziva i *fitness* funkcija, funkcija sposobnosti, funkcija cilja, i sl.

definiranje funkcije dobrote predstavlja ključni (a često i najteži) korak, kako bi funkcija vjerno odražvala problem koji se rješava.

U dobrom genetskom algoritmu, tijekom procesa evolucije, kroz generacije, ukupna i prosječna dobrota populacije rastu. Ukupna (D) i prosječna ($\overline{D}$) dobrota populacije, za populaciju veličine N su definirani izrazima:

$$D = \sum_{i=1}^{N} dobrota(v_i) \tag{3.7}$$

$$\overline{D} = \frac{D}{N} \tag{3.8}$$

3.4 SELEKCIJA

Uloga procesa selekcije u genetskom algoritmu je očuvanje i prenošenje dobrih svojstava jedinki u novu populaciju. Selekcijom se (imitirajući prirodnu selekciju) odabiru dobre jedinke koje će u novoj generaciji sudjelovati u reprodukciji, čime se čuva dobar genetski materijal, prenosi se na novu populaciju, a loš genetski materijal odumire. Prvi, očiti, način na koji bi se selekcija mogla realizirati bio bi da se jedinke u trenutnoj populaciji sortiraju po dobroti, da se odabere $N - M$ [9] najboljih jedinki koje će se prenijeti u novu populaciju, te koje će reprodukcijom stvoriti M potomaka. U praksi se pokazalo da takav postupak selekcije rezultira prebrzom konvergencijom algoritma – proces optimizacije završi u svega nekoliko iteracija, te postoji opasnost od "zaglavljivanja" procesa u nekom lokalnom optimumu. Razlog tome leži u tome što i loše jedinke *mogu* posjedovati kvalitetne dijelove genetske informacije. Stoga je potrebno osigurati proces selekcije koji će i lošim jedinkama dati šansu (naravno, manju u odnosu na dobre jedinke) da prežive i sudjeluju u reprodukciji.

Postupci selekcije se dijele na dvije osnovne grupe: *generacijska* selekcija i *eliminacijska* selekcija. Prema vrsti ugrađene selekcije se i genetski algoritmi dijele na generacijske i eliminacijske. Kod generacijskog genetskog algoritma se procesom selekcije iz stare generacije stvara nova, a u slučaju eliminacijskog genetskog algoritma se "rupa" u populaciji nastala eliminacijom loših jedinki popunjava novim jedinkama.

[9] N je veličina populacije, a M je broj jedinki koje će odumrijeti.

3.4.1 JEDNOSTAVNA SELEKCIJA

Jednostavna selekcija je osnovni primjer generacijske selekcije. Procesom selekcije se iz stare populacije *P(t-1)* stvara nova populacija jedinki *P(t)* sa jednakim brojem jedinki, a odabir roditelja se provodi tako da je vjerojatnost odabira proporcionalna njihovoj dobroti. Postupak se provodi u 4 osnovna koraka:

1. izračunaju se vrijednosti *dobrota(v_i)* sa sve jedinke u populaciji

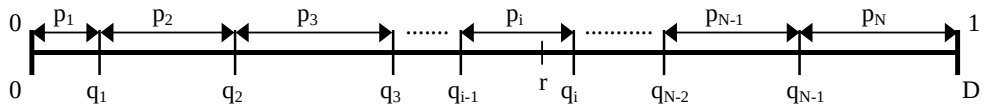
2. izračuna se ukupna dobrota prema izrazu (3.7)

3. za svaki kromosom se izračuna kumulativna dobrota q_k prema izrazu (3.9) tako da vjerojatnost selekcije za pojedini kromosom v_k iznosi p_k (izraz (3.10)), iz čega slijedi da je vjerojatnost selekcije proporcionalna njegovoj dobroti:

$$q_k = \sum_{i=1}^{k} dobrota(v_i) \tag{3.9}$$

$$p_k = \frac{dobrota(v_k)}{D} \tag{3.10}$$

4. generira se slučajni realni broj $r \in [0, D]$ i potraži se *i*-ti kromosom za koji vrijedi da je $r \in \langle q_{i-1}, q_i \rangle$, te se taj kromosom prenosi u novu generaciju



Slika 3.4 Kumulativna dobrota q_i i vjerojatnost selekcije jedinke p_i

Ovakvim se postupkom selekcije dobri kromosomi mogu pojaviti i po nekoliko puta u novoj populaciji.

Jednostavna selekcija ima nekoliko nedostataka: budući da je vjerojatnost odabira jedinke proporcionalna njenoj dobroti, funkcija dobrote *dobrota(v_i)* ne smije poprimati negativne vrijednosti. Ovaj nedostatak je moguće riješiti *translacijom* funkcije dobrote. Kreira se nova funkcija dobrote koja od vrijednosti dobrote kromosoma oduzima minimalnu vrijednost dobrote u čitavoj populaciji:

$$dobrota'(v_i) = dobrota(v_i) - \min_k \{dobrota(v_k)\}, \text{ gdje je } k=1, 2, ..., N \tag{3.11}$$

Vrijednost min*{dobrota(v_k)}* se za svaku generaciju mora posebno pronaći, što se obično obavlja usporedno sa potragom za najboljom jedinkom nakon izračunavanja kumulativnih vrijednosti dobrota. Na se ovaj način efikasno izbjegavaju negativne vrijednosti funkcije dobrote.

Drugi nedostatak jednostavne selekcije je i pojavljivanje dupliciranih kromosoma. Eksperimenti su pokazali da i do 50% populacije mogu biti duplikati kromosoma, čime se značajno smanjuje efikasnost genetskog algoritma.

3.4.2 ELIMINACIJSKA SELEKCIJA

Još jedan veliki nedostatak generacijske selekcije je i potreba za održavajem dviju populacija za vrijeme trajanja selekcije: stara populacija i nova, dobivena odabirom jedinki iz stare. Tek kada je nova populacija popunjena, stara se briše. Udvostručavanje memorijskog prostora potrebnog za generacijsku selekciju je moguće izbjeći eliminacijskom selekcijom.

Eliminacijska selekcija se obavlja upravo suprotno generacijskoj: umjesto biranja dobrih jedinki koje će preživjeti, biraju se *loše* jedinke koje će odumrijeti. Stoga se umjesto funkcije dobrote deinira *funkcija kazne*, pa je vjerojatnost odabira pojedine jedinke *obrnuto proporcionalna* njenoj dobroti:

$$kazna(v_i) = \max_k \{dobrota(v_k)\} - dobrota(v_i) \tag{3.12}$$

Ovakvim načinom selekcije se rješava još jedan problem generacijske selekcije: očuvanje najbolje jedinke (*elitizam*), zato što je kod eliminacijske selekcije vjerojatnost odabira najbolje jednike jednaka nuli, pa u proces selekcije nije potrebno ugrađivati dodatne mehanizme za zaštitu najbolje jedinke. Ovdje još valja napomenuti da je izračunavanje kumulativne dobrote i funkcije kazne potrebno izvršiti prije svakog odabira jedinke za eliminaciju, jer se jednom eliminirani kromosom ne može ponovo odabrati.

3.4.3 TURNIRSKA SELEKCIJA

Najjednostavniji oblik turnirske selekcije kreira novu populaciju tako da sa jednakom vjerojatnošću bira dvije jedinke iz populacije i bolju od njih kopira u novu populaciju, što se ponvlja *N* puta. Ovakav oblik turnirske selekcije spada u generacijske selekcije.

Međutim, turnirska se selekcija obično primjenjuje na drugačiji način, kako bi se eliminirao najveći nedostatak generacijske i eliminacijske selekcije: računanje kumulativnih dobrota (kazni) i njihovo sortiranje u svakoj generaciji (a kod eliminacijske selekcije prije svake eliminacije).

S jednakom se vjerojatnošću odabiru 3 (ili više) jedinke iz populacije. Na dvije najbolje jedinke se primjenjuje operator križanja, a njihov se potomak stavlja na mjesto najlošije trenutno odabrane jedinke. Takva se selekcija naziva *3-turnirska*[10] *selekcija* (eng. *3-way*

[10] Odnosno n-turnirska ako se bira n jedinki.

tournament selection), i u njoj nema vidljive granice između generacija, jer se selekcija i reprodukcija obavljaju kontinuirano.

Prednosti ovakve vrste selekcije su: dobrota jedinke se računa samo za nove jedinke, nema potrebe za računanjem kumulativnih dobrota, kao ni njihovog sortiranja, čime se znatno dobija na brzini izvođenja algoritma. Svojstvo elitizma je također zadovoljeno, zato što je selekcijom i reprodukcijom nemoguće obrisati najbolju jedinku (naravno, još uvijek se mora paziti prilikom mutacije). Još jedna prednost turnirske selekcije je inherentna mogućnost paraleliziranja procesa selekcije i reprodukcije, jer je moguće nezavisno izvoditi nekoliko usporednih turnirskih selekcija nad populacijom jedinki.

3.5 OSTALI GENETSKI OPERATORI

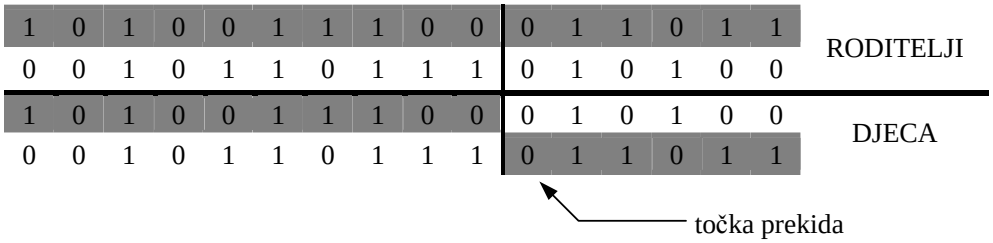
Uz prikaz rješenja i selekciju, genetski operatori *križanja* i *mutacije* su još jedna važna karakteristika genetskog algoritma. Djelovanjem tih genetskih operatora se mijenja genetski materijal jedinki, te se vrši njegova razmjena među jedinkama, a njihova je zadaća da stvore potomstvo iz odabrane populacije roditelja[11]. Stoga genetski operatori u velikoj mjeri određuju kvalitetu rada genetskog algoritma. Operatori djeluju tijekom reprodukcije, u kojoj sudjeluju jedinke koje su "preživjele" selekciju: križanjem dolazi do razmjene genetskih materijala roditelja kako bi se stvorilo "potomstvo", a mutacijom dolazi do slučajne izmjene gena.

3.5.1 KRIŽANJE

Križanje je proces u kojem dolazi do razmjene genetskog materijala između dvije jedinke – *roditelja*. Križanjem nastaju jedna ili dvije nove jedinke koje se nazivaju *djeca* (eng. *offsprings*). Budući da dolazi do miješanja genetskog materijala, djeca *nasljeđuju* svojstva svojih roditelja, pa ako su roditelji *dobri*, velika je vjerojatnost da će i djeca biti *dobra*, ako ne i bolja od svojih roditelja. Postupak križanja je potrebno posebno definirati za svaki prikaz kromosoma. Ovdje će detaljnije biti prikazani neki operatori križanja za binarni prikaz.

Najjednostavniji operator križanja, križanje s jednom točkom prekida, je prikazan na slici 3.5: slučajno se odabere jedna točka prekida unutar kromosoma, te se sadržaji roditeljskih kromosoma zamijene. Time se dobijaju dva potomka.

[11] S nadom da će djeca biti bolja od svojih roditelja.



Slika 3.5 Križanje s jednom točkom prekida

U općenitom slučaju križanje može biti definirano s proizvoljnom brojem točaka. Krajnji slučaj, križanje sa *n*-1 prekidnih točaka (gdje je *n* broj bitova u kromosomu) se naziva *uniformno križanje*. Pri uniformnom križanju nastaje jedno dijete, i to tako da je vjerojatnost nasljeđivanja pojedinog bita od određenog roditelja 0.5, tj. vjerojatnost nasljeđivanja pojedinog gena je jednaka za oba roditelja. Pseudokod uniformnog križanja dan je na slici 3.6. Ako se vjerojatnost nasljeđivanja razlikuje, tada se križanje naziva *p-uniformno križanje*. Npr.: ako je *p*=0.4, tada je vjerojatnost nasljeđivanja pojedinog gena od prvog roditelja jednaka 0.4, a od drugog 0.6. Vjerojatnosti nasljeđivanja se mogu razlikovati i za pojedine gene unutar samog kromosoma. Tada se definira maska sa vjerojatnostima nasljeđivanja gena. Tako je u primjeru na slici 3.7 vjerojatnost da će prvi gen biti uzet od prvog roditelja jednaka 0.2, a od drugog 0.8. Treći gen će biti sigurno uzet od prvog, a gen *n*-2 će biti uzet od drugog roditelja.

```
uniformno_križanje
početak
      za i=0 do n-1
            dijete[i]=bilo_koji(roditelj1[i], roditelj2[i])
kraj
```

Slika 3.6 Pseudokod uniformnog križanja

gen	1	2	3	4	5	...	n-2	n-1	n
p	0.2	0.5	1	0.6	0.7	...	0	0.3	0.5

Slika 3.7 Maska sa vjerojatnostima odabira gena.

Na računalu je uniformno križanje najlakše i najbrže implementirati kao logičku operaciju, jer su logičke operacije sastavni dio strojnog jezika računala. Prvi korak je kreiranje slučajne binarne maske *M* koja je jednake duljine kao i kromosomi koje se križa. Bitovi u maski određuju od kojega se roditelja uzima pojedini gen: 1 znači da će gen biti uzet od prvog roditelja, a 0 da će gen biti uzet od drugog roditelja. Prvo se pomoću logičke operacije *I* maskiraju bitovi prvog roditelja: $A \wedge M$. Bitovi drugog roditelja se maskiraju pomoću invertirane maske: $B \wedge \neg M$.

Tako maskirani bitovi se zatim pomoću logičke operacije *ILI* kombiniraju u djetetov kromosom (izraz (3.13)) Primjer uniformnog križanja dan je na slici 3.8.

$$DIJETE = (A \wedge M) \vee (B \wedge \neg M) \tag{3.13}$$

1	0	0	1	0	1	1	1	0	1	1	0	0	1	0	0	RODITELJI
0	1	1	1	1	0	0	1	0	0	0	1	1	1	0	1	
1	1	0	1	1	0	0	0	1	1	0	1	0	0	1	0	MASKA
1	0	1	1	0	0	0	1	0	1	0	0	1	1	0	1	DIJETE

Slika 3.8 Primjer uniformnog križanja

Joše neke od varijanti križanja su: *segmentno križanje*, koje je u biti križanje s više točaka prekida, samo što se za svako križanje posebno određuje broj i pozicija prekida. *Miješajuće križanje* se obavlja u tri koraka: u prvom se koraku bitovi u svakom od roditelja izmiješaju. Zatim se provede klasično križanje, s jednom ili više točaka prekida. U trećem se koraku bitovi roditelja vraćaju na svoja mjesta.

U procesu križanja je korisno vršiti provjeru da li su kromosomi roditelja identični. Ako jesu, obično se kreira slučajni kromosom za jednoga od roditelja (npr, mutacijom), pa se onda provodi križanje. Tim se postupkom kontrolira pojavljivanje duplikata, jer oni samo usporavaju rad genetskog algoritma, a i proširuje se prostor pretraživanja.

3.5.2 MUTACIJA

Drugi operator koji djeluje na jedinke tijekom reprodukcije je *mutacija*. Mutacija djeluje na jednu jedinku i sastoji se od slučajne promjene jednog ili više gena.

Dva parametra određuju mutaciju u genetskom algoritmu: vrsta mutacije i vjerojatnost mutacije jednog bita p_m. Vjerojatnost mutacije mora biti u intervalu *<0, 1>*. Ako vjerojatnost mutacije teži k 1, tada se genetski algoritam pretvara u postupak slučajne pretrage prostora rješenja, a ako vjerojatnost mutacije teži k nuli, manje se unosi "svježe" genetske informacije u populaciju, tako da postoji velika vjerojatnost da će genetski algoritam stati u nekom lokalnom optimumu.

Jednostavna mutacija je najjednostavniji oblik mutacije: u slučajno odabranom kromosomu se invertira jedan slučajno odabrani bit.

1	1	0	1	0	1	1	0	1	0	1	1	1	0	1	0	prije mutacije
1	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	poslije mutacije

Slika 3.9 Jednostavna mutacija

Miješajuća mutacija odabire kromosom i dvije pozicije između kojih izmiješa ili invertira gene (invertirajuća miješajuća mutacija). Potupna miješajuća mutacija je krajnji slučaj gdje se miješaju svi geni u kromosomu. Tablica 3.2 prikazuje različite vrste mutacija.

Tablica 3.2 Različite vrste mutacija nad kromosomom 1011010011011101

vrsta mutacije	slučajno odabran gen, npr. sedmi	slučajno generirana maska 0001011110100010	svi geni
jednostavna	10110100**1**011101	101**0**0**1**10111111	**0100101100100010** (potpuna inverzija)
miješajuća	–	101**1**0**1**01**1**0111**0**1	**1011011110110100** (broj jedinica i nula ostaje isti)
potpuna miješajuća	–	101**0**0**0**10**1**0111**1**1	**0110011101000110** (potpuno slučajni kromosom)
invertirajuća miješajuća	–	101**0**0**1**10**1**111**1**1	**0100101100100010** (potpuna inverzija)

Kada geneski algoritam ne koristi binarni prikaz kromosoma, vjerojatnost mutacije jednog bita p_m nema smisla, pa je potrebno definirati vjerojatnost mutacije jednog kromosoma p_M (n predstavlja broj bitova u kromosomu):

$$p_M = 1 - (1 - p_m)^n \tag{3.14}$$

Na primjer, ako je p_m=0.01, i neka je n=40. Tada je p_M=0.331, što znači da će na svakih 100 kromosoma, 33 biti mutirana.

Zadaća operatora mutacije je da unese novi, odnosno obnovi izgubljeni genetski materijal. Time se proširuje prostor rješenja koji se pretražuje. Kao primjer opet može poslužiti binarni prikaz: ukoliko se dogodi da svi kromosomi u populaciji na određenom mjestu u kromosomu imaju istu vijednost gena, taj se gen neće moći promijeniti (iz 0 u 1 ili obrnuto) kroz križanje, tako da je efektivno izgubljeno pola prostora pretraživanja. Isto tako, ukoliko populacija završi u nekom lokalnom optimumu, samo pomoću mutacije postoji mogućnost da neka od jedinki “iskoči” iz prostora lokalnog optimuma. U tom je slučaju dovoljno da samo jedna jedinka kroz mutaciju postane bolja od ostalih, pa da se cijela populacija brzo preseli u područje gdje bi se moglo nalaziti bolje rješenje.

3.6 STRUKTURA GENETSKOG ALGORITMA

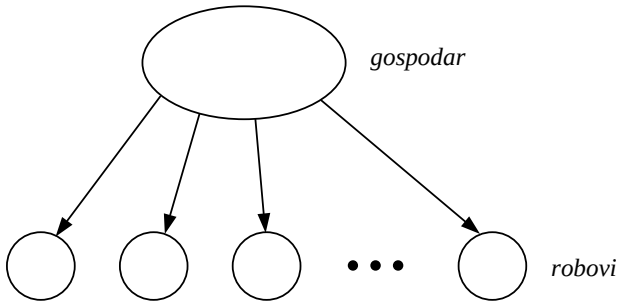
Osnovna struktura genetskog algoritma prikazana je na slici 3.1. No, kako genetski algoritam izvodi veliku količinu operacija, ovakav sekvencijalni pristup se pokazao previše sporim u primjeni na zahtjevnijim problemima. U osnovnoj strukturi genetskog algoritma moguće je izdvojiti pojedine dijelove koji se mogu izvršavati neovisno, pa je razvojem višeprocesorskih računala stvorena mogućnost paralelizacije, čime genetski algoritmi značajno dobijaju na brzini.

Jedan od dijelova genetskog algoritma koji može imati koristi od paralelizacije je evaluacija jedinki. Naime, evaluacija određene jedinke je potpuno neovisna o evaluaciji neke druge jedinke, tako da je moguće rasporediti evaluaciju cijele populacije na nekoliko raspoloživih procesora. Time se može znatno dobiti na brzini izvođenja algoritma, naročito kada je funkcija dobrote relativno složena. 3-turnirska selekcija je također pogodna za paralelizaciju zbog svojeg načina rada: odaberu se slučajno tri jedinke, najbolje dvije se križaju i dijete zamijeni najlošiju jedinku, koja se zatim još može i evaluirati. Izvođenjem nekoliko paralelnih turnirskih selekcija nad jednom populacijom također se znatno dobija na brzini konvergencije algoritma.

Postoji mnogo shema za paralelizaciju rada genetskog algoritma, ovisno o različitim procesorskim i memorijskim konfiguracijama višeprocesorskih računala. Ovdje će detaljnije biti opisan *globalni paralelni genetski algoritam*.

3.6.1 GLOBALNI PARALELNI GENETSKI ALGORITAM

Osnovna struktura globalnog paralelnog genetskog algoritma[12] (GPGA) dana je na slici 3.10.



Slika 3.10 Osnovna struktura GPGA

Ideja GPGA je da se jedan dio rada genetskog algoritma paralelizira stvaranjem *slugu* kojima *gospodar* daje zadatke na izvršavanje. U klasičnoj konfiguraciji gospodar je zadužen za održavanje populacije jedinki i izvršavanje genetskih operatora (selekcija, križanje i mutacija), dok su sluge zaduženi za evaluaciju jedinki, i to tako da gospodar

[12] Još se naziva i *master-slave* genetski algoritam.

svakom slugi dodjeljuje dio populacije nad kojim će ovaj izvršiti evaluaciju, te čeka da svi sluge jave da su završili s poslom[13]. Time se može postići značajno ubrzanje u odnosu na sekvencijalni genetski algoritam, naročito ako je funkcija dobrote složena, pa evaluacija jedinki traje relativno dugo, te ako se koriste velike populacije, pa je potrebno izvršiti mnogo evaluacija u svakoj generaciji.

Budući da struktura GPGA ne postavlja specifične zahtjeve na konfiguraciju višeprocesorskog računala, učinkovita implementacija je moguća kako na računalima sa dijeljenom memorijom, tako i na računalima sa distribuiranom memorijom. Na sustavima sa distribuiranom memorijom, cijela populacija je pohranjena u memoriju procesora koji je gospodar, te je taj procesor zadužen za slanje dijelova populacije ostalim procesorima (slugama). U praksi se pokazalo da GPGA daje gotovo linearan porast brzine u odnosu na sekvencijalni genetski algoritam, ali samo do određenog broja procesora. Razlog tome je dodatno vrijeme koje glavni procesor troši na raspoređivanje poslova i komunikaciju sa ostalim pocesorima (naročito izraženo kod sustava sa distribuiranom memorijom, gdje je potrebno slati dijelove populacije ostalim procesorima), pa se nakon određene granice više ne isplati povećavati broj procesora.

Još jedna karakteristika klasičnog sinkronog GPGA jest da se po svojstvima (osim po brzini) ne razlikuje od sekvencijalnog genetskog algoritma, pa je na njega primjenjiva teorija koja važi za sekvencijalni genetski algoritam.

[13] Takav GPGA se naziva *sinkroni* GPGA, za razliku od *asinkronog*, gdje se algoritam ne zaustavlja dok sluge rade svoj posao.

4. NEKE OD PRIMJENA GENETSKIH ALGORITAMA

Zbog svoje jednostavnosti i robusnosti, genetski su algoritmi našli primjenu u mnogim poljima znanosti i tehnologije. Samo neke od primjena su: optimiranje funkcija, rješavanje kombinatoričkih problema, teorija igara, raspoznavanje uzoraka, strojno učenje, problem rasporeda, itd.

4.1 OPTIMIRANJE FUNKCIJA REALNE VARIJABLE

Zbog svoje strukture genetski su se algoritmi pokazali izuzetno pogodnima za optimiranje funkcija realnih varijabli. Njihova robusnost na tom području proizlazi iz ugrađenih karakteristika. Kodiranje i dekodiranje kromosoma je krajnje jednostavno: najjednostavnije je koristiti binarni prikaz, pa ako se za svaku dimenziju funkcije uzme n bitova u kromosomu, tada je potrebno podijeliti interval pretraživanja $[dg, gg]$ na 2^n dijelova, a dekodiranje kromosoma se obavlja prema izrazu (3.2). Funkciju dobrote također nije teško odrediti: to je vrijednost funkcije koja se optimira za točku koju predstavlja određeni kromosom, tako da što je vrijednost funkcije veća, to je i dobrota jedinke veća[14]. Genetski operatori za binarni prikaz su vrlo jednostavni i lako ih je implementirati u bilo kojem proceduralnom programskom jeziku.

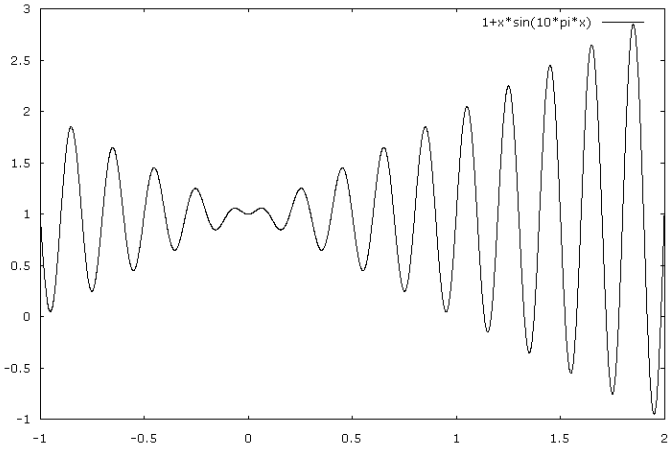
Genetski algoritam ne vrši pretragu prostora rješenja pomoću samo jedne točke, već se koristi čitava populacija točaka, čime se postiže neosjetljivost na lokalne optimume, a na kraju se ne dobija samo jedno rješenje, već čitav skup rješenja bliskih globalnom optimumu (što je vrlo pogodno ukoliko funkcija ima nekoliko globalnih optimuma). Budući da se funkcija koja se optimira tretira kao *crna kutija*, ne postavljaju se nikakvi posebni zahtjevi na oblik funkcije, kao što su derivabilnost, neprekinutost i sl. Iz istog razloga nije potrebno vršiti prilagodbu algoritma pri promjeni funkcije koja se optimira.

Naravno, genetski algoritmi nisu savršeni. Jedan od osnovnih problema je taj što genetski algoritam ne garantira pronalaženje globalnog optimuma, on samo daje rješenje koje je *blisko* globalnom optimumu, tako da je nemoguće postići potpunu pouzdanost rješenja[15]. Još jedan veliki problem je i određivanje početnih parametara genetskog algoritma, jer ne postoji sistematični postupak za određivanje dobrih početnih parametara za dani problem, pa se određivanje parametara često svodi na isprobavanje različitih vrijednosti. Genetski algoritmi zbog velike količine operacija koje se moraju provesti konvergiraju puno sporije od specijaliziranih metoda, pa zahtjevaju veliku računalnu snagu.

Na slikama su dane neke od ispitnih funkcija korištene za testiranje rada genetskog algoritma.

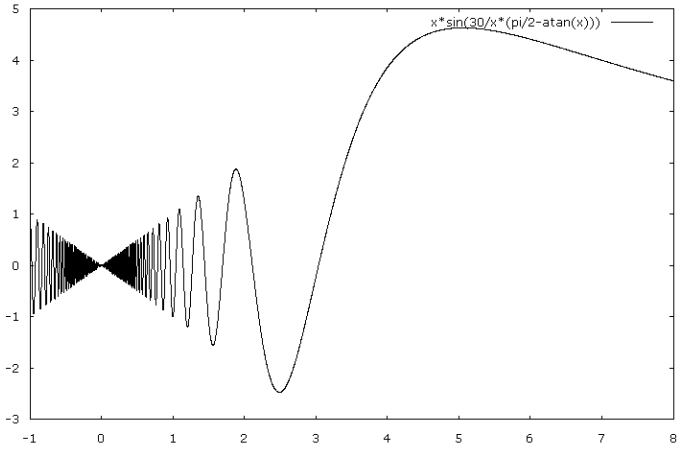
[14] U slučaju minimizacije funkcije, funkcija dobrote se uzima kao vrijednost funkcije pomnožena sa -1.
[15] Točnost rješenja se jednostavno može povećati pokretanjem genetskog algoritma sa smanjenim prostorom pretraživanja.

$$f(x) = 1 + x\sin(10x\pi)$$
$$\max\{f(x)\} = f(1.85047) = 2.850274$$



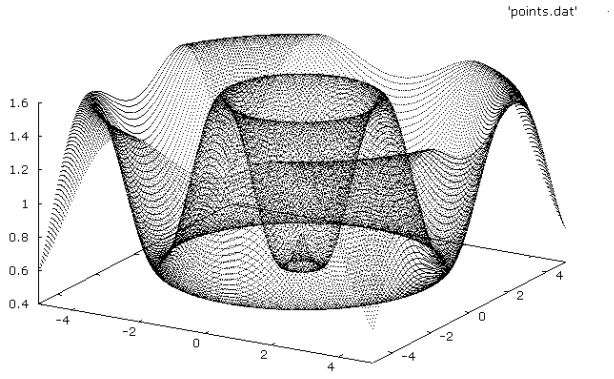
Slika 4.1 Primjer ispitne funkcije

$$f(x) = x\sin\left\{\frac{30}{x}\left[\frac{\pi}{2} - \arctan(x)\right]\right\}$$
$$\max\{f(x)\} = f(5.05567) = 4.632549$$



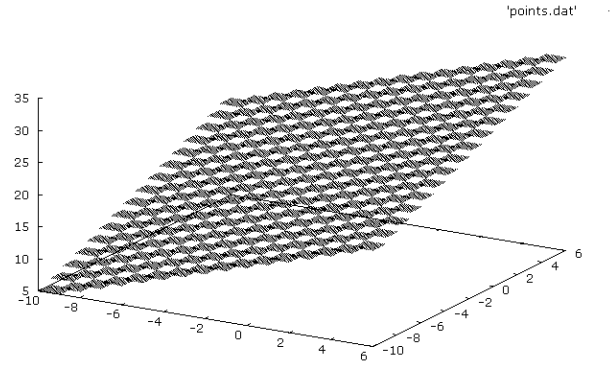
Slika 4.2 Primjer ispitne funkcije

$$f(x,y)=0.5+\sin^2\left(\frac{\sqrt{x^2+y^2}-0.5}{\left[1+0.001\left(x^2+y^2\right)\right]^2}\right)$$
$$\min\{f(x,y)\}=0.5$$



Slika 4.3 Primjer ispitne funkcije

$$f(x,y)=25+\lfloor x\rfloor+\lfloor y\rfloor$$
$$\max\{f(x,y)\}=f([5,6],[5,6])=35$$



Slika 4.4 Primjer ispitne funkcije

4.2 RJEŠAVANJE NP PROBLEMA

Razred problema koje nije moguće riješiti u polinomijalnom vremenu pomoću deterministič kih metoda naziva se razred NP (ne-polinomijalnih) problema[16]. NP problemi su vrlo teški, sa složenostima koje mogu biti reda $O(2^n)$ ili čak $O(n!)$. Za rješavanje NP problema razvijene su nedeterminističke metode kojima je moguće rješavati NP probleme u polinomijalnom vremenu. Ali, budući da su te metode aproksimativne, one ne garantiraju pronalaženje optimalnog rješenja.

Najopćenitiji oblik NP problema je zadovoljivost logičke funkcije: za danu logičku funkciju od *n* varijabli, da li je moguće naći takvu kombinaciju vrijednosti varijabli da funkcija daje rezultat 1 ("istinito"). Složenost ovoga problema je $O(2^n)$ – za funkciju sa *n* logičkih varijabli postoji 2^n mogućih kombinacija (tablica 4.1).

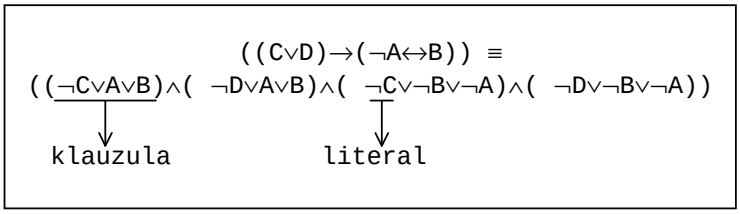
Tablica 4.1 Zadovoljivost logičke funkcije $A \wedge B$

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Već za relativno malen broj varijabli rješavanje ovog problema deterministič kim metodama u nekom realističnom vremenu postaje nemoguće: računalu koje može ispitati 10^{12} mogućih kombinacija u jednoj sekundi trebalo bi 18 sekundi za rješavanje funkcije sa 50 logičkih varijabli, a već za funkciju od 100 varijabli bi bilo potrebno približno $4 \cdot 10^{10}$ godina! Stoga je nužno korištenje nedeterminističkih metoda, a genetski su se algoritmi pokazali pogodnima zbog svoje mogućnosti efikasnog pretraživanja velikih prostora rješenja.

Budući da su vrijednosti logičkih varijabli binarne (0 ili 1), za prikaz rješenja se može odabrati binarni prikaz, gdje svaki bit odgovara pojedinoj logičkoj varijabli. Određivanje funkcije dobrote je nešto kompliciranije, zato što logičke funkcije kao izlaz daju samo 0 ili 1 ("laž" ili "istina"), pa samo promatranjem izlaza logičke funkcije ne možemo zaključiti koliko je neko rješenje dobro ili loše. Stoga je potrebno razviti heuristiku koja će pokazati u kojoj su mjeri loša rješenja doista loša. Jedan mogući način je pretvaranje rješavane logičke funkcije u *klauzalni* oblik. Klauzalni oblik funkcije se sastoji od konjunkcije *klauzula*, gdje je klauzula disjunkcija od n *literala* (literal je logička varijabla ili negacija logičke varijable). Na slici 4.5 je prikazana jedna logička funkcija i njen klauzalni oblik.

[16] Problemi koje je moguće riješiti u polinomijalnom vremenu pomoću deterministič kih metoda nazivaju se P problemima.



Slika 4.5 Klauzalni oblik logičke funkcije

Za ovako prikazanu logičku funkciju je očito da će rezultat biti jednak 1 ako i samo ako svaka pojedina klauzula daje rezultat 1, pa se za funkciju dobrote može koristiti broj klauzula koje daju rezultat 1. Označimo li klauzule sa K_i, funkcija dobrote se može napisati kao:

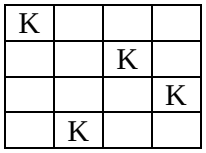
$$dobrota(v) = \frac{\sum_{i=1}^{n} K_i}{n} \tag{4.1}$$

Ovako definirana funkcija dobrote vraća vrijednosti iz intervala *[0, 1]*, gdje vrijednost 1 predstavlja rješenje problema.

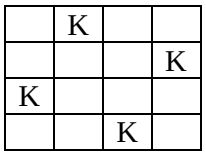
Za većinu NP problema ne postoji dovoljno dobar prikaz rješenja za rješavanje pomoću genetskog agoritma. Ali, kako je moguće sve NP probleme mapirati na rješavanje zadovoljivosti logičke funkcije, koje ima dobar prikaz rješenja, proizlazi da bi se mnogi NP problemi mogli rješavati pomoću genetskih algoritama.

4.2.1 PROBLEM *N* KRALJICA

Kraljica, kao najjača figura u šahu se može pomicati u svim smjerovima na šahovskoj ploči za proizvoljan broj polja. Problem *N* kraljica se zahtijeva raspoređivanje *n* kraljica na šahovsku ploču dimenzija $n \times n$ na takav način da se kraljice međusobno ne napadaju. Ovaj problem spada u NP razred problema zato što postoji $\binom{n^2}{n}$ mogućih načina da se kraljice rasporede na ploču. Međutim, u rješavanju tog problema se obično koristi prikaz permutiranim nizom brojeva, čime se eliminiraju nedozvoljene pozicije kraljica po recima i stupcima (tj, eliminiraju se kombinacije u kojima se dvije ili više kraljica nalaze u istom retku ili stupcu), pa je prostor rješenja reduciran na složenost *O(n!)*. Na slici 4.6 su prikazana dva primjera za slučaj 4 kraljice. Pozicija broja u nizu označava stupac, a broj predstavlja redak u kojem se kraljica nalazi.



(4, 1, 3, 2)



(2, 4, 1, 3)

Slika 4.6 Primjeri prikaza za 4 kraljice

Poteškoća kod određivanja funkcije dobrote za problem *N* kraljica je što je određeno rješenje točno ili pogrešno. Kao i za većinu kombinatoričkih problema, potrebno je odrediti funkciju dobrote koja će ocijeniti koliko je pojedino pogrešno rješenje blisko točnom. Zapisom rješenja pomoću permutiranog niza eliminirani su konflikti po stupcima i recima. Stoga pogrešna rješenja imaju neke kraljice raspoređene tako da se međusobno napadaju po dijagonalama, pa se funkcija dobrote može napisati tako da broji upravo konflikte na dijagonalama – što je broj konflikata veći, to je rješenje lošije. Za točno rješenje će funkcija dobrote vratiti nulu.

Za permutirani niz od *n* kraljica (k_1, ..., k_i, ..., k_j, ..., k_n), *i*-ta i *j*-ta kraljica su na istoj dijagonali ako vrijedi:

$$i - k_i = j - k_j \tag{4.2}$$

ili

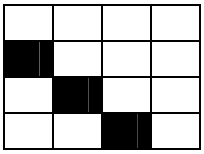
$$i + k_i = j + k_j \tag{4.3}$$

Što se može kraće zapisati kao:

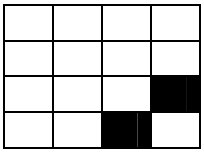
$$\|k_i - k_j\| = \|i - j\| \tag{4.4}$$

Na ovaj način se može odrediti funkcija dobrote koja će vraćati ukupan broj konflikata u nekom rješenju. Na taj način je moguće odrediti koliko je određeno rješenje “dobro”.

Ovakvim jednostavnim pristupom se dobija funkcija dobrote čija je složenost $O(n^2)$, što ju čini relativno sporom, naročito za veći broj kraljica. Uvođenjem brojača za svaku dijagonalu moguće je reducirati složenost funkcije dobrote na $O(n)$. Na ploči dimenzija $n \times n$ postoji $2n-1$ “lijevih” dijagonala (protežu se od vrha ka dnu ploče, slijeva nadesno) i $2n-1$ “desnih” dijagonala (od dna prema vrhu ploče , zdesna nalijevo)



Slika 4.7 Treća “lijeva” dijagonala



Slika 4.8 Druga “desna” dijagonala

Kraljica koja se nalazi u *i*-tom stupcu i k_i-tom retku se nalazi na $i-k_i$ “lijevoj” i $n-i+k_i$ “desnoj” dijagonali. Ako je nakon prebrojavanja na nekoj dijagonali 0 ili 1 kraljica, na toj dijagonali nema konflikata. Za dijagonale na kojima je broj kraljica veći od 1, broj konflikata jednak je broju kraljica na toj dijagonali umanjenom za 1. Pseudokod algoritma funkcije dobrote prikazan je na slici 4.9. Suma konflikata je normirana s obzirom na duljinu dijagonale.

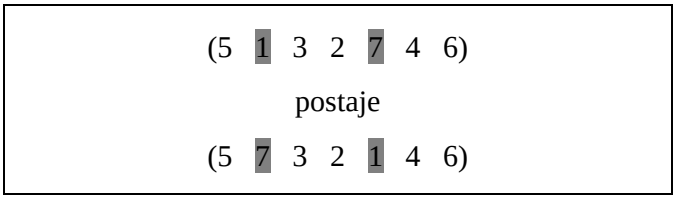
```
postavi brojače lijeve i desne dijagonale na 0

za i= 1 do n
      lijeva_dijag[i+kᵢ]++
      desna_dijag[n-i+kᵢ]++
suma = 0
za i = 1 do (2n-1)
      brojač = 0
      ako (lijeva_dijag[i] > 1)
            brojač += lijeva_dijag[i]-1
      ako (desna_dijag[i] > 1)
            brojač += desna_dijag[i]-1
      suma += brojač / (n-abs(i-n))
```

Slika 4.9 Funkcija dobrote za problem *N* kraljica

Osim određivanja funkcije dobrote, za problem *N* kraljica potrebno je odrediti i specijalizirane operatore križanja i mutacije.

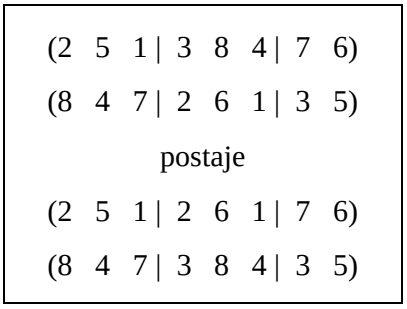
Za operator mutacije najjednostavnije je koristiti operator koji će u slučajno odabranom kromosomu zamijeniti dvije pozicije unutar niza. U primjeru na slici u kromosomu su slučajno odabrane pozicije 2 i 5:



Slika 4.10 Primjer mutcije

Operatora križanja postoji nekoliko vrsta. Najjednostavniji operator križanja je ekvivalentan operatoru mutacije, tj. izmjena dvije slučajno odabrane pozicije u kromosomu. Očiti nedostatak ovog operatora je taj što ne koristi i kombinira informacije dvaju roditelja kako bi se dobio potencijalno bolji potomak, pa se rješavanje praktički svodi na slučajnu pretragu prostora rješenja.

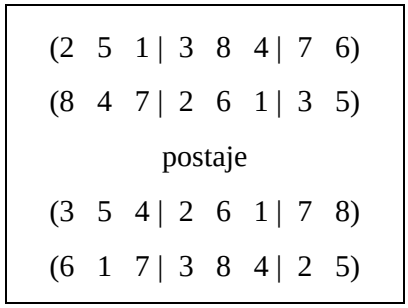
Slijedeći operator križanja se naziva PMX[17] operator. Po načinu rada je sličan križanju sa dvije točke prekida kod binarnog prikaza kromosoma. Prvi korak je odabir dvije točke prekida u roditeljskim kromosomima i izmjena genetskog materijala:



Slika 4.11 Prvi korak PMX križanja

Vidljivo je da niti jedan potomak nije pravilan, jer se u kromosomu određeni brojevi ponavljaju. Taj se problem rješava u drugo koraku PMX križanja: u prvom potomku se broj 2 pojavljuje na pozicijama 1 i 4. Kako je 2 na poziciji 4 noviji, 2 na poziciji 1 se zamjenjuje sa 3, koji se nalazi na poziciji 4 prije križanja. Analogno se rješavaju ostala ponavljanja unutar kromosoma.

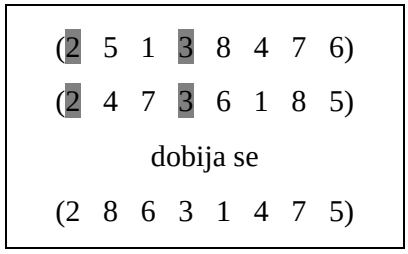
[17] eng. Partially Matched Crossover



Slika 4.12 Drugi korak PMX križanja

U praksi se pokazalo da se PMX križanje ne ponaša dovoljno dobro, jer relativno brzo dolazi do rješenja bliskih optimumu, ali postoji tendencija unifikacije populacije, pa se sporo dolazi do konačnog rješenja. Relativno velika složenost algoritma je još jedan problem PMX križanja.

Treći operator križanja je zamišljen kao svojevrsni kompromis između slučajnog mijenjanja kromosoma i izmjene genetskih materijala roditelja, te prilagođen 3-turnirskoj selekciji korištenoj u programu. Osnovna zamisao tog operatora je da se od roditelja na dijete prenesu zajedničke pozicije, a ostale se pozicije popune slučajnim odabirom, ali tako da je rezultirajući kromosom pravilan (tj. da je permutirani niz od n članova)



Slika 4.13 Treći operator križanja

U primjeru na slici 4.13 se podudaraju broj 2 na poziciji 1 i broj 3 na poziciji 4. Oni se prenose na potomka, a ostale pozicije u kromosomu se slučajno raspoređuju, pazeći da je rezultirajući kromosom pravilan. Iz primjera se također može primjetiti da ako kod roditelja nema podudaranja pozicija, potomak s kreira potpuno slučajno.

U praksi se pokazalo da i ovaj operator križanja može dovesti do brze unifikacije populacije (naročito kod većih problema), pa je uveden još jedan korak provjere: ako su roditelji jednaki, kreira se slučajni kromosom za drugog roditelja, pa se tek onda provodi križanje. Ovako modificiran operator se pokazao vrlo kvalitetnim, rezultirajući brzom konvergencijom algoritma.

5. OPIS PROGRAMA

U ovom poglavlju će biti opisana programska implementacija globalnog paralelnog algoritma korištenog za rješavanje problema N kraljica i optimiranje funkcija realne varijable.

Pri oblikovanju programa posebna je pažnja bila posvećena modularnosti izvedbe, kako bi se ostvarila mogućnost jednostavne nadogradnje i prilagodbe programa. Jezgru programa čine razredi potrebni za održavanje populacije (jedan razred za predstavljanje jedinki i jedan razred za održavanje populacije jedinki), zatim razred koji predstavlja gospodara u GPGA, te bazni razredi operatora[18].

Za svaki problem (ili skupinu problema) koji se rješavaju, potrebno je definirati razrede operatora koji će se izvršavati nad jedinkama. Na taj je način omogućeno stvaranje biblioteke različitih razreda operatora koje je moguće koristiti za određene prikaze jedinki, odnosno za određene vrste problema, te je moguća njihova jednstavna zamjena u svrhu eksperimentiranja.

5.1 JEZGRA

Kao što je već spomenuto, jezgru programa čine razredi za održavanje populacije, razred gospodar i bazni razredi operatora.

Na slici 5.1 je prikazana deklaracija razreda za predstavljanje jedinki, *CIndividual*. Razred *CIndividual* je *template* razred kome je moguće definirati tip podataka koji predstavljaju kromosom i tip podataka koji predstavljaju dobrotu jedinke. Na taj je način postignuta maksimalna prilagodljivost različitim vrstama prikaza kromosoma, kao i različitim vrstama prikaza dobrote jedinke. Moguće je, na primjer, za neki problem definirati kromosom kao niz cijelih brojeva, a dobrotu kao realni broj dvostruke preciznost, dok za neki drugi problem kromosom može biti prikazan razredom koji predstavlja stablo, a doborota kao cijeli broj. Jedini zahtjev za razred kojim se predstavlja dobrota jedinke je da za njega postoje definirani operatori usporedbe (>, < i =). U tablici 5.1 su prikazane metode definirane u razredu *CIndividual*.

[18] Ovdje se ne misli samo na genetske operatore, nego i na “operatore”, npr. inicijalizacije programa ili evaluacije jedinki.

```
template <class ChromClass, class FitClass>
class CIndividual {

public:
	// construction/destruction
	CIndividual();
	virtual ~CIndividual();

	// allocates chromosome
	virtual void AllocChromosome(DWORD dwChromDim);

	// chromosome handling functions
	virtual ChromClass* GetChromosomePtr();
	virtual DWORD	GetChromosomeDim();

	// fitness handling functions
	virtual const FitClass&	GetFitness();
	virtual void		SetFitness(const FitClass& Fitness);

private:
	// chromosome data
	ChromClass*		m_pChromosome;
	DWORD			m_dwChromosomeDimension;

	// individual fitness
	FitClass		m_Fitness;
};
```

Slika 5.1 Deklaracija razreda *CIndividual*

Tablica 5.1 Metode razreda *CIndividual*

AllocChromosome	Alocira memoriju potrebnu za spremanje kromosoma.
GetChromosomePtr	Vraća pokazivač na kromosom
GetChromosomeDim	Vraća dimeziju kromosoma (kromosom može biti i polje podataka)
GetFitness	Vraća dobrotu jedinke
SetFitness	Sprema dobrotu jedinke

Razred *CPopulation* predstavlja populaciju. *CPopulation* je također *template* razred, zadužen za stvaranje i održavanje populacije jedinki. Pri deklaraciji objekta tipa *CPopulation* potrebno je definirati tip kromosoma i dobrote jedinke. Konstrutror zatim kreira populaciju zadane veličine. Drugi konstruktor u razredu ne kreira novu populaciju, već služi stvaranju objekta koji pokazuje na dio postojeće populacije. Stvaranje takvog objekta je korisno kada, na primjer, gospodar želi slugi poslati dio populacije na evaluaciju. Slika 5.2 prikazuje deklaraciju razreda *CPopulation*, a tablica 5.2 metode definirane u razredu *CPopulation*.

```
template <class ChromClass, class FitClass>
class CPopulation {

public:
	// constructors
	CPopulation(DWORD dwPopulationSize, DWORD dwChromosomeDimension);
	CPopulation(CPopulation<ChromClass, FitClass>& Population,
		DWORD dwPopChunkStartIndex, DWORD dwPopChunkSize);

	// destructor
	~CPopulation();

	// operations

	// returns pointer to the individual at the given index (zero-based)
	CIndividual<ChromClass, FitClass>* GetIndividualPtr(DWORD dwIndex);

	// returns population size
	DWORD GetPopulationSize();

	// returns the best individual index
	DWORD GetBestIndividualIndex();

	// returns the pointer to the best individual
	CIndividual<ChromClass, FitClass>* GetBestIndividualPtr();

	// thread safe stuff
	BOOL TryLockIndividual(DWORD dwIndex);
	void UnlockIndividual(DWORD dwIndex);

private:
	CIndividual<ChromClass, FitClass>*	m_pPopulation;
	// size of the population
	DWORD  m_dwPopulationSize;

	// set to FALSE if a population object is a part of another
	// population (that way the population data is not deleted
	// when this object is destroyed)
	BOOL   m_bCleanup;

	// stuff needed for thread-safe access to the population individuals
	// 'individuals-in-use' flag array
	BOOL*  m_pbIndividualFlags;

	// critical section for accessing m_pbIndividualFlags array
	CRITICAL_SECTION    m_csFlags;
};
```

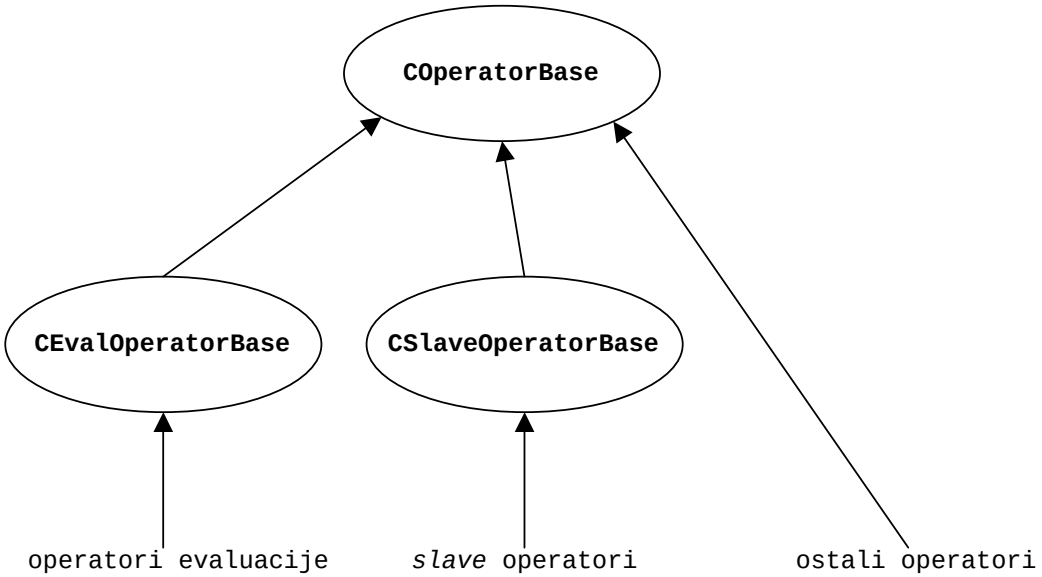
Slika 5.2 Deklaracija razreda *CPopulation*

Tablica 5.2 Metode razreda *CPopulation*

GetIndividualPtr	Vraća pokazivač na zadanu jedinku
GetPopulationSize	Vraća veličinu populacije
GetBestIndividualIndex	Vraća poziciju najbolje jedinke u populaciji
GetBestIndividualPtr	Vraća pokazivač na najbolju jedinku u populaciji
TryLockIndividual	Pokušava zaključati zadanu jedinku za ekskluzivan pristup u višedretvenom okruženju.
UnlockIndividual	Otključava jedinku zaključanu pomoću metode **TryLockIndividual**

Za razrede operatora su definirana tri čista virtualna razreda: *COperatorBase*, *CEvalOperatorBase* i *CSlaveOperatorBase*. Iz ta tri razreda se izvode svi ostali razredi operatora. Razredi operatora mogu biti jednostavni ili složeni. Primjeri jednostavnog razreda operatora su: operator inicijalizacije, operator selekcije, operator križanja, operator evaluacije, i sl. Složeni operatori u sebi sadrže nekoliko jednostavnih operatora. Primjer složenog operatora je *slave* operator koji u sebi sadrži operatore selekcije, križanja i evaluacije.

Iz razreda *CEvalOperatorBase* se izvode razredi operatora evaluacije, a iz razreda *CSlaveOperatorBase* se izvode *slave* operatori koje će izvršavati sluge. Svi ostali jednostavni i složeni operatori se izvode iz razreda *COperatorBase*. Na slici 5.3 je prikazana hijerarhija operatorskih razreda, a na slici 5.4 su prikazane deklaracije baznih razreda operatora.



Slika 5.3 Hijerarhija razreda operatora

```
class COperatorBase {
public:
	COperatorBase(){};
	virtual ~COperatorBase(){};

	virtual DWORD Run() = 0;
};

//////////////////////////////////////////////////////////////////////

class CEvalOperatorBase : public COperatorBase {
public:
	CEvalOperatorBase(){};
	virtual ~CEvalOperatorBase(){};

	virtual DWORD Run() = 0;
	virtual DWORD Run(CIndividual
		<class ChromClass, class FitClass>* pIndividual) = 0;
};

//////////////////////////////////////////////////////////////////////

class CSlaveOperatorBase : public COperatorBase {
public:
	CSlaveOperatorBase(){};
	virtual ~CSlaveOperatorBase(){};

	virtual DWORD Run() = 0;
	virtual DWORD Run(DWORD dwThreadNo, DWORD dwTotalThreads) = 0;
};
```

Slika 5.4 Deklaracija baznih razreda operatora

Svi operatori vrše potrebne inicijalizacije u svojim konstruktorima, a objekt koji koristi određeni operator, izvršava operator pozivom metode **Run**.

Razred *CGpgaEngine* predstavlja gospodara u GPGA. Objekt gospodar vrši inicijalizaciju genetskog algoritma, pokreće dretve slugu, te izvršava *master* operator. Deklaracija razreda je prikazana na slici 5.5.

```
class CGpgaEngine {
public:
	// construction/destruction
	CGpgaEngine(COperatorBase& opInitialize, COperatorBase& opMaster,
			CSlaveOperatorBase& opSlave, DWORD dwSlaveThreads);

	~CGpgaEngine();

	// runs the engine for specified number of iterations (max)
	DWORD Run(DWORD dwIterations);

	// returns the statistics for the previous run
	DWORD* GetStatistics();

private:

	// slave thread(s) function
	static DWORD WINAPI SlaveThread(LPVOID lpThreadData);

	COperatorBase&	m_opInitialize;
	COperatorBase&	m_opMaster;
	CSlaveOperatorBase& m_opSlave;

	// total number of slave threads
	DWORD		m_dwSlaveThreads;
	// thread data (one for each :-)
	THREAD_DATA*	m_pThreadsData;
	// thread handles
	HANDLE*		m_phThreads;
	// handles used by the threads to indicate that they have finished
	// with running the slave operator
	HANDLE*		m_phThreadsFinished;

	// events used to signal to the slave threads to run the
	// slave operator
	HANDLE*		m_phThreadsStart;
	// semaphore used to indicate to the slave threads that they
	// should exit
	HANDLE		m_hThreadsExitSemaphore;
};
```

Slika 5.5 Deklaracija razreda *CGpgaEngine*

Pri konstrukciji, razred *CGpgaEngine* očekuje reference na operator inicijalizacije, operator koji će izvršavati sluge (može biti jednostavan ili složen), te operator koji će izvršavati gospodar (također može biti jednostavan ili složen). Zadnji parametar konstruktora je broj slugu. Ako se broj slugu postavi na 1, praktički se dobija sekvencijalni genetski algoritam, što je korisno za usporedbu brzine rada sekvencijalnog genetskog algoritma i GPGA.

Metodom **Run** se pokreće genetski algoritam za zadani broj iteracija. Metodu **Run** je moguće zvati nekoliko puta za redom bez potrebe za reinicijalizacijom algoritma[19]. Metoda **GetStatistics** vraća polje cijelih brojeva; svaki broj predstavlja ukupno vrijeme izvođenja pojedinog sluge.

[19] Što je korisno ako se želi, na primjer, pratiti statistika tijekom izvođenja algoritma, ili ako se želi povećati točnost rješenja.

5.2 RAZREDI ZA OPTIMIRANJE FUNKCIJA REALNIH VARIJABLI

Za implementaciju razreda za optimiranje funkcija realnih varijabli je odabrana slijedeća konfiguracija: prikaz kromosoma je binarni, sa 32 bita po dimenziji funkcije, a dobrota jedinke je realni broj dvostruke preciznosti. Kao funkcija dobrote je uzeta vrijednost koju vraća funkcija koju se optimira za dekodiranu vrijednost kromosoma[20].

Raspored genetskih operatora je slijedeći: gospodar izvršava jednostavnu mutaciju nad populacijom, a sluge vrše 3-turnirsku selekciju i uniformno križanje. Evaluacija pojedine jedinke se izvršava nakon obavljene mutacije, odnosno križanja. Razred evaluacije u sebi sadrži više ispitnih funkcija. Funkcija koja će biti optimirana se bira pri stvaranju objekta operatora evaluacije. Operator inicijalizacije inicijalizira generator slučajnih brojeva, stvara slučajnu populaciju i vrši početnu evaluaciju svih jedinki.

Na slikama 5.6 i 5.7 je prikazan izvorni kod 3-turnirske selekcije. Izvorni kod uniformnoga križanja je prikazan na slici 5.8. Ako su roditelji identični, kreira se novi, slučajni kromosom za drugog roditelja, pa se tek onda provodi križanje. Operator mutacije izvodi jednostavnu mutaciju nad populacijom, vodeći računa da ne mutira najbolju jedinku. Izvorni kod je prikazan na slici 5.9.

```
DWORD CRealFnSelectionOperator::Run()
{
	CIndividual<DWORD, double>* pIndividual1;
	CIndividual<DWORD, double>* pIndividual2;
	CIndividual<DWORD, double>* pIndividual3;
	CIndividual<DWORD, double>* pTmpInd;

	DWORD  dwChromosome1;
	DWORD  dwChromosome2;
	DWORD  dwChromosome3;

	try {
		// select first random chromosome index
		do {
			dwChromosome1 = (DWORD) (rand() %
				m_Population.GetPopulationSize());
		} while(!m_Population.TryLockIndividual(dwChromosome1));

		// select second random chromosome index
		// must be different from the first one - we ensure this by
		// thread-safe access to individuals
		do {
			dwChromosome2 = (DWORD) (rand() %
				m_Population.GetPopulationSize());
		} while(!m_Population.TryLockIndividual(dwChromosome2));
```

Slika 5.6 Izvorni kod operatora selekcije (nastavak na sljedećoj stranici)

[20] Ako se vrši minimizacija funkcije, vrijednost koju vraća funkcija se množi sa -1.

```
            // select third random chromosome index
            // must be different from the previous two - we ensure this by
            // thread-safe access to individuals
            do {
                  dwChromosome3 = (DWORD) (rand() %
                        m_Population.GetPopulationSize());
            } while(!m_Population.TryLockIndividual(dwChromosome3));

            // now, sort the chromosomes
            pIndividual1 = m_Population.GetIndividualPtr(dwChromosome1);
            pIndividual2 = m_Population.GetIndividualPtr(dwChromosome2);
            pIndividual3 = m_Population.GetIndividualPtr(dwChromosome3);

            if (pIndividual1->GetFitness() < pIndividual2->GetFitness()) {
                  pTmpInd = pIndividual1;
                  pIndividual1 = pIndividual2;
                  pIndividual2 = pTmpInd;
            }

            if (pIndividual2->GetFitness() < pIndividual3->GetFitness()) {
                  pTmpInd = pIndividual2;
                  pIndividual2 = pIndividual3;
                  pIndividual3 = pTmpInd;
            }

            if (pIndividual1->GetFitness() < pIndividual2->GetFitness()) {
                  pTmpInd = pIndividual1;
                  pIndividual1 = pIndividual2;
                  pIndividual2 = pTmpInd;
            }

            // perform the crossover
            CRealFnCrossoverOperator opCrossover
                  (pIndividual1, pIndividual2, pIndividual3);
            if (opCrossover.Run() == 2) {
                  // we must evaluate the second individual as well
                  m_EvalOperator.Run(reinterpret_cast<CIndividual<
                        class ChromClass, class FitClass>*>(pIndividual2));
            }
            m_EvalOperator.Run(reinterpret_cast<CIndividual<
                        class ChromClass, class FitClass>*>(pIndividual3));

            // unlock the individuals
            m_Population.UnlockIndividual(dwChromosome1);
            m_Population.UnlockIndividual(dwChromosome2);
            m_Population.UnlockIndividual(dwChromosome3);

            return 1;
      } catch (...) {

            // unlock the individuals
            m_Population.UnlockIndividual(dwChromosome1);
            m_Population.UnlockIndividual(dwChromosome2);
            m_Population.UnlockIndividual(dwChromosome3);
            return 0xFFFFFFFF;
      }
}
```

Slika 5.7 Izvorni kod operatora selekcije (nastavak)

```
DWORD CRealFnCrossoverOperator::Run()
{
	try {
		DWORD  dwMask;

		DWORD  dwDim        = m_pParent1->GetChromosomeDim();
		DWORD* pdwParent1   = m_pParent1->GetChromosomePtr();
		DWORD* pdwParent2   = m_pParent2->GetChromosomePtr();
		DWORD* pdwOffspring= m_pOffspring->GetChromosomePtr();

		DWORD  dwResult = 1;

		// if parents are equal, create a random second parent
		if (!memcmp(pdwParent1, pdwParent2, dwDim * sizeof(DWORD))) {
			for (DWORD i = 0; i < dwDim; ++i) {
				pdwParent2[i] =
					((rand() & 0x00FF) |
					((rand() & 0x00FF) << 8) |
					((rand() & 0x00FF) << 16) |
					((rand() & 0x00FF) << 24));
			}

			dwResult = 2;
		}

		for (DWORD i = 0; i < dwDim; ++i) {
			// for each DWORD in chromosome array,
			// create a random mask...
			dwMask = ((rand() & 0x00FF) |
					((rand() & 0x00FF) << 8) |
					((rand() & 0x00FF) << 16) |
					((rand() & 0x00FF) << 24));

			// ... and do a uniform crossover
			// offspring = (parent1 & mask) | (parent2 & ~mask)
			pdwOffspring[i] =   (pdwParent1[i] & dwMask) |
					    (pdwParent2[i] & ~dwMask);
		}

		return dwResult;
	} catch (...) {
		return 0xFFFFFFFF;
	}
}
```

Slika 5.8 Izvorni kod operatora križanja

```
DWORD CRealFnMutationOperator::Run()
{
	try {
		// determine the number of mutating chromosomes
		// we need this +0.5 in order to ensure correct truncation
		// of the float value
		DWORD dwMutatingChromosomes;
		dwMutatingChromosomes = (DWORD)(m_Population.GetPopulationSize()
			* m_Population.GetIndividualPtr(0)->GetChromosomeDim()
			* m_dBitMutationRate + 0.5);

		DWORD  dwBestIndividual = m_Population.GetBestIndividualIndex();

		// now, select dwMutatingChromosomes chromosomes to mutate,
		// and invert one bit chosen at random
		DWORD  dwRandChromosome;
		DWORD  dwRandDword;
		DWORD  dwRandomBit;
		DWORD  dwMutationMask;
		CIndividual<DWORD, double>* pMutatingInd;

		for (DWORD i = 0; i < dwMutatingChromosomes; ++i) {
			// select random chromosome
			// (we must not select the best individual)
			do {
				dwRandChromosome = rand() %
					m_Population.GetPopulationSize();
			} while (dwRandChromosome == dwBestIndividual);

			// select random bit in the chromosome
			dwRandomBit = rand() % (m_Population.GetIndividualPtr(0)
				->GetChromosomeDim() * 32);

			// now, determine in which DWORD within the chormosome the
			// mutating bit lies (divide by 32)...
			dwRandDword = dwRandomBit >> 5;
			// ... create the mutation mask
			dwMutationMask = 1 << (dwRandomBit % 32);
			// ... mutate the chromosome...
			pMutatingInd =
				m_Population.GetIndividualPtr(dwRandChromosome);
			(pMutatingInd->GetChromosomePtr())[dwRandDword] ^=
				dwMutationMask;

			// ... and evaluate the new individual
			m_EvalOperator.Run(reinterpret_cast<CIndividual<
				class ChromClass, class FitClass>*>(pMutatingInd));
		}

		// done! return.
		return 1;

	} catch (...) {
		return 0xFFFFFFFF;
	}
}
```

Slika 5.9 Izvorni kod operatora mutacije

5.3 RAZREDI ZA RJEŠEVANJE PROBLEMA *N* KRALJICA

Osnovni pristup rješavanju problema *N* kraljica je opisan u poglavlju 4.2.1. Za prikaz kromosoma je korišteno polje cijelih 16-bitnih brojeva, koje predstavlja permutirani niz pozicija. Dobrota jedinke je iskazana kao realni broj jednostruke preciznosti.

Genetski operatori su raspoređeni kao i pri optimiranju funkcija realnih varijabli: gospodar vrši mutaciju, a sluge obavljaju 3-turnirsku selekciju i križanje jedinki. Evaluacija jedinki se obavlja po izvršenoj mutaciji, odnosno križanju. Kako je korišten novi prikaz kromosoma, bilo je potrebno definirati i novu funkciju dobrote, kao i genetske operatore.

Kao funkcija dobrote korištena je funkcija opisana u poglavlju 4.2.1 i čiji je pseudokod prikazan na slici 4.9. Izvorni kod funkcije je prikazan na slici 5.10.

Korišteni operator mutacije je također opisan u poglavlju 4.2.1 i sastoji se od zamjene dvije slučajno odabrane pozicije unutar kromosoma. Pri tome se vodi računa o tome da se ne mutira najbolja jedinka u populaciji. Izvorni kod je prikazan na slici 5.11.

Kao operatori križanja su isprobana dva operatora opisana u poglavlju 4.2.1: operator križanja ekvivalentan operatoru mutacije i vlastiti operator križanja (PMX operator nije korišten). Kako operator križanja ekvivalentan mutaciji nije davao zadovoljavajuće rezultate, u konačnoj verziji programa je korišten vlastiti operator križanja čiji je izvorni kod prikazan na slikama 5.12 i 5.13.

```
DWORD CNQeenEvalOperator::Run(
		CIndividual<class ChromClass, class FitClass>* pIndividual)
{
	CIndividual<WORD, float>* pInd =
		reinterpret_cast<CIndividual<WORD, float>*>(pIndividual);

	// these are grid diagonals
	DWORD* pLeftDiagonal  = NULL;
	DWORD* pRightDiagonal = NULL;
	DWORD  dwDiagonals;
	// chromosome pointer
	WORD*  pChromosome;
	DWORD  dwQueens;

	DWORD  i;
	float  fSum;
	DWORD  dwCounter;

	try {
		// get the number of diagonals, and allocate them
		dwQueens = pInd->GetChromosomeDim();
		dwDiagonals = 2 *  dwQueens - 1;
		pLeftDiagonal  = new DWORD[dwDiagonals];
		pRightDiagonal = new DWORD[dwDiagonals];
		::ZeroMemory(pLeftDiagonal, dwDiagonals * sizeof(DWORD));
		::ZeroMemory(pRightDiagonal, dwDiagonals * sizeof(DWORD));

		/// get chromosome data pointer
		pChromosome = pInd->GetChromosomePtr();

		// calculate the fitness
		for (i = 0; i < dwQueens; ++i) {
			++pLeftDiagonal[i + pChromosome[i]];
			++pRightDiagonal[dwQueens - i + pChromosome[i] - 1];
		}

		fSum = 0.0;

		for (i = 0; i < dwDiagonals; ++i) {
			dwCounter = 0;

			if (pLeftDiagonal[i] > 1)
				dwCounter += pLeftDiagonal[i] - 1;
			if (pRightDiagonal[i] > 1)
				dwCounter += pRightDiagonal[i] - 1;

			fSum += (float)dwCounter /
				(float)(dwQueens - abs(i + 1 - dwQueens));
		}
		pInd->SetFitness(-fSum);

		// cleanup
		delete [] pLeftDiagonal;
		delete [] pRightDiagonal;

		return 1;
	} catch (...) {
		// cleanup
		if (pLeftDiagonal)  delete [] pLeftDiagonal;
		if (pRightDiagonal) delete [] pRightDiagonal;
		return 0xFFFFFFFF;
	}
}
```

Slika 5.10 Izvorni kod funkcije dobrote za problem *N* kraljica

```
DWORD CNQueenMutationOperator::Run()
{
	try {
		// determine the number of mutating chromosomes
		// we need this +0.5 in order to ensure correct truncation
		// of the float value
		DWORD dwMutatingChromosomes;
		dwMutatingChromosomes = (DWORD)(m_Population.GetPopulationSize()
			* m_dChromosomeMutationRate + 0.5);

		DWORD  dwBestIndividual = m_Population.GetBestIndividualIndex();

		// now, select dwMutatingChromosomes chromosomes to mutate
		DWORD  dwRandChromosome;
		DWORD  dwPos1;
		DWORD  dwPos2;

		CIndividual<WORD, float>*  pMutatingInd;
		DWORD                      dwQueens;
		WORD*                      pMutatingIndChrom;
		WORD                       wTmp;

		for (DWORD i = 0; i < dwMutatingChromosomes; ++i) {
			// select random chromosome
			// (we must not select the best individual)
			do {
				dwRandChromosome = rand32() %
					m_Population.GetPopulationSize();
			} while (dwRandChromosome == dwBestIndividual);

			pMutatingInd =
				m_Population.GetIndividualPtr(dwRandChromosome);

			dwQueens = pMutatingInd->GetChromosomeDim();

			// select two different points in the tuple...
			dwPos1 = rand32() % dwQueens;
			while ((dwPos2 = rand32() % dwQueens) == dwPos1);

			// ... mutate the individual...
			pMutatingIndChrom = pMutatingInd->GetChromosomePtr();

			// swap positions in the tuple
			wTmp = pMutatingIndChrom[dwPos1];
			pMutatingIndChrom[dwPos1] = pMutatingIndChrom[dwPos2];
			pMutatingIndChrom[dwPos2] = wTmp;

			// ... and evaluate the new individual
			m_EvalOperator.Run(reinterpret_cast<
				CIndividual<class ChromClass,
				class FitClass>*>(pMutatingInd));

			if (pMutatingInd->GetFitness() == 0) return 0;
		}

		// done! return.
		return 1;

	} catch (...) {
		return 0xFFFFFFFF;
	}
}
```

Slika 5.11 Izvorni kod operatora mutacije za problem *N* kraljica

```
DWORD CNQueenCrossoverOperator::Run()
{
	try {
		DWORD  dwQueens     = m_pParent1->GetChromosomeDim();
		WORD*  pwChrom1     = m_pParent1->GetChromosomePtr();
		WORD*  pwChrom2     = m_pParent2->GetChromosomePtr();
		WORD*  pwChrom3     = m_pOffspring->GetChromosomePtr();
		WORD   wTmp;
		DWORD  dwPos1;
		DWORD  dwPos2;
		DWORD  dwTmpPos;
		DWORD i;
		DWORD j;

		DWORD dwResult      = 1;

		// ... first create default n-tuple (0, 1, 2, ...) ...
		for (i = 0; i < dwQueens; ++i) {
			pwChrom3[i] = (WORD)i;
		}

		// ... and then tumble it a little bit
		for (i = 0; i < dwQueens; ++i) {
			// select two different positions to swap...
			dwPos1 = rand32() % dwQueens;
			while ((dwPos2 = rand32() % dwQueens) == dwPos1);

			// ... and then swap them
			wTmp = pwChrom3[dwPos1];
			pwChrom3[dwPos1] = pwChrom3[dwPos2];
			pwChrom3[dwPos2] = wTmp;
		}

		// parents are equal?
		if (memcmp(pwChrom1, pwChrom2, dwQueens * sizeof(WORD)) == 0) {
			// yes, create a random chromosome
			// for the second individual
			// ... first create default n-tuple (0, 1, 2, ...) ...
			for (i = 0; i < dwQueens; ++i) {
				pwChrom2[i] = (WORD)i;
			}
			// ... and then tumble it a little bit
			for (i = 0; i < dwQueens; ++i) {
				// select two different positions to swap...
				dwPos1 = rand32() % dwQueens;
				while ((dwPos2 = rand32() % dwQueens) == dwPos1);

				// ... and then swap them
				wTmp = pwChrom2[dwPos1];
				pwChrom2[dwPos1] = pwChrom2[dwPos2];
				pwChrom2[dwPos2] = wTmp;
			}
			dwResult = 2;
		}
```

Slika 5.12 Izvorni kod operatora križanja za problem *N* kraljica (nastavak na sljedećoj stranici)

```
            // perform the crossover
            // see which positions in parents coincide
            for (i = 0; i < dwQueens; ++i) {
                  if (pwChrom1[i] == pwChrom2[i]) {
                        for (j = 0; j < dwQueens; ++j) {
                              if (pwChrom3[j] == pwChrom1[i]) break;
                        }

                        wTmp = pwChrom3[i];
                        pwChrom3[i] = pwChrom3[j];
                        pwChrom3[j] = wTmp;
                  }
            }
            return dwResult;

      } catch (...) {
            return 0xFFFFFFFF;
      }
}
```

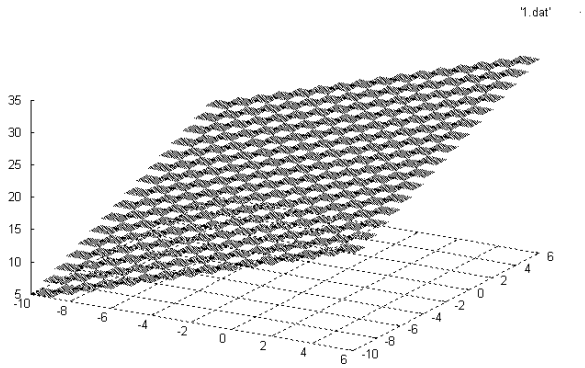
Slika 5.13 Izvorni kod operatora križanja za problem *N* kraljica (nastavak)

6. REZULTATI

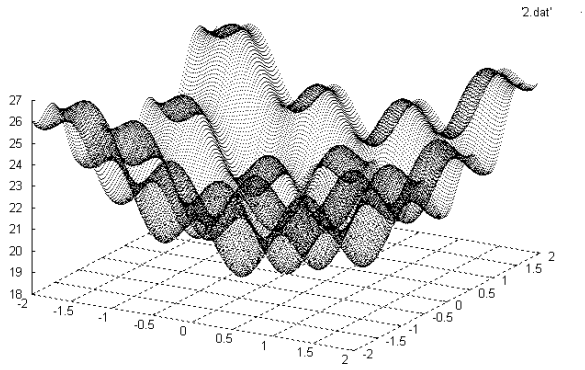
Sva testiranja opisana u ovom poglavlju su obavljena na jednoprocesorskom računalu Pentium II@300MHz, tako da je paralelni rad dretvi mogao biti samo simuliran, pa nije bilo moguće napraviti pouzdanu usporedbu ukupnih vremena izvođenja programa. Stoga su prikazani rezultati samo okvirni pokazatelji brzine izvođenja GPGA.

6.1 OPTIMIRANJE FUNKCIJA REALNIH VARIJABLI

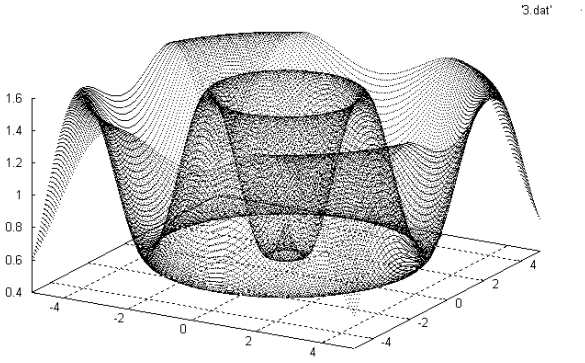
Za testiranje rada GPGA na optimiranju funkcija realnih varijabli korištene su ispitne funkcije koje posjeduju "nezgodna" svojstva: prva ispitna funkcija, prikazana na slici 6.1, ne zadovoljava uvjet neprekinutosti; druga funkcija (slika 6.2) ima nekoliko lokalnih optimuma u blizini lokalnog, dok treća ispitna funkcija (slika 6.3) posjeduje mnogo globalnih optimuma. Prva i druga funkcija su pri optimiranju korištene u svojim 40-dimenzijskim oblicima, dok su sva grafička rješenja predstavljena za 2-dimenzijske oblike.



Slika 6.1 Prva ispitna funkcija



Slika 6.2 Druga ispitna funkcija



Slika 6.3 Treća ispitna funkcija

Osnovni cilj testiranja je bio utvrditi kako broj slugu utječe na brzinu izvršavanja algoritma. Ispitivanje se vršilo sa 1, 2, 5 i 10 slugu. Budući da se u programu koristila 3-turnirska selekcija, vodilo se računa da se broj iteracija slugu prilagodi tako da ukupan broj izvršenih selekcija bude konstantan. Ostali parametri algoritma (veličina populacije, vjerojatnost mutacije i dr.) se nisu mijenjali. Interval pretraživanja je bio *[-10, 10]* za svaku dimenziju funkcije.

U tablicama su prikazani rezultati za ispitne funkcije. Za svaki eksperiment su prikazani: broj selekcija svakog sluge po iteraciji gospodara, najdulje trajanje izvođenja dretvi slugu (T_S), trajanje izvođenja gospodara (T_M), zbroj najduljeg trajanja izvođenja dretvi slugu i trajanja izvođenja gospodara, te konačna vrijednost optimirane funkcije

Tablica 6.1 Rezultati za prvu ispitnu funkciju

Dimenzija funkcije: 40 Veličina populacije: 50 Vjerojatnost mutacije: 0.01 Broj iteracija gospodara: 500 Smjer optimiranja: maksimizacija Vrijednost funkcje u optimumu: 385.000000					
Broj slugu	**Broj selekcija po iteraciji gospodara**	**T_S**	**T_M**	**$T_S + T_M$**	**Vrijednost funkcije**
1	400	11.886 s	0.308 s	12.194 s	385.000000
2	200	5.851 s	0.302 s	6.153 s	385.000000
5	80	2.692 s	0.303 s	2.995 s	385.000000
10	40	1.403 s	0.303 s	1.706 s	385.000000

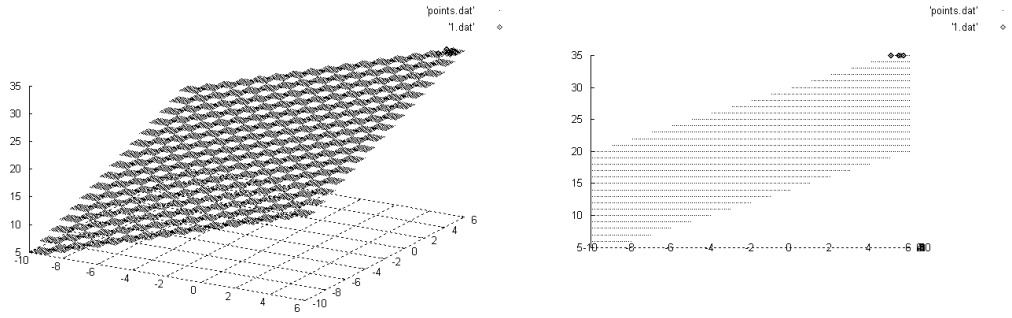
Tablica 6.2 Rezultati za drugu ispitnu funkciju

Dimenzija funkcije: 40 Veličina populacije: 50 Vjerojatnost mutacije: 0.01 Broj iteracija gospodara: 500 Smjer optimiranja: minimizacija Vrijednost funkcje u optimumu: 360.000000					
Broj slugu	**Broj selekcija po iteraciji gospodara**	**T_S**	**T_M**	**$T_S + T_M$**	**Vrijednost funkcije**
1	400	10.965 s	0.232 s	11.197 s	370.492862
2	200	5.490 s	0.215 s	5.705 s	372.381004
5	80	2.448 s	0.212 s	2.660 s	370.484683
10	40	1.131 s	0.215 s	1.346 s	372.388308

Tablica 6.3 Rezultati za treću ispitnu funkciju

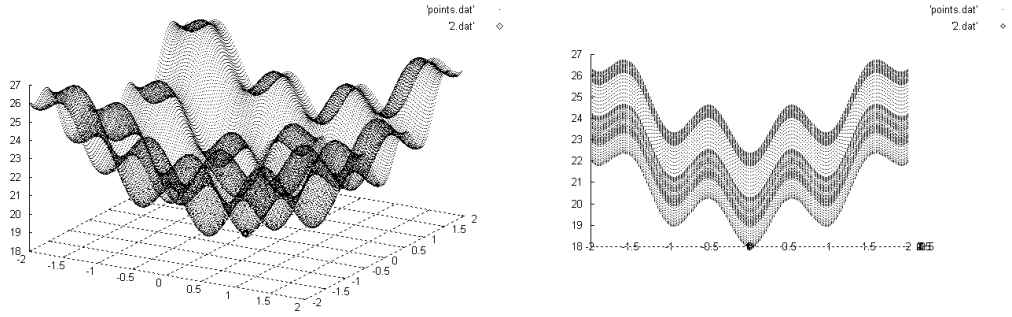
Dimenzija funkcije: 2 Veličina populacije: 50 Vjerojatnost mutacije: 0.01 Broj iteracija gospodara: 500 Smjer optimiranja: minimizacija Vrijednost funkcje u optimumu: 0.500000					
Broj slugu	**Broj selekcija po iteraciji gospodara**	**T_S**	**T_M**	**$T_S + T_M$**	**Vrijednost funkcije**
1	400	1.625 s	0.009 s	1.634 s	0.500000
2	200	0.578 s	0.010 s	0.588 s	0.500000
5	80	–*	–*	–*	0.500000
10	40	–*	–*	–*	0.500000

Na sljedećim slikama su grafički prikazani rezultati nekoliko optimizacija za 2-dimenijske primjere ispitnih funkcija.

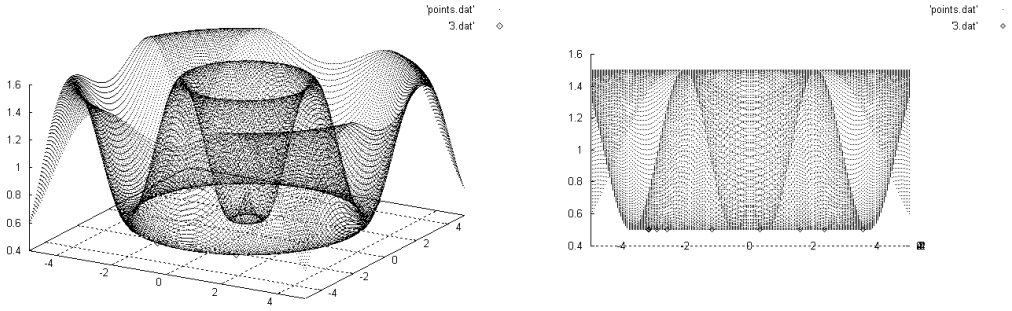


Slika 6.4 Rješenja za prvu ispitnu funkciju

* Prekratka vremena za pouzdano mjerenje.



Slika 6.5 Rješenja za drugu ispitnu funkciju



Slika 6.6 Rješenja za treću ispitnu funkciju

6.2 RJEŠAVANJE PROBLEMA *N* KRALJICA

Rad genetskog algoritma na rješavanju problema *N* kraljica testiran je na problemima veličine 15, 20, 40, 50, 100 i 200 kraljica. Parametri za sve veličine problema su bili jednaki: veličina poplacije od 1000 jedinki, broj slugu 2 i vjerojatnost mutacije kromosoma 0.02.

U tablici 6.4 su dani rezultati: veličina problema, najduže trajanje izvršavanja dretvi slugu (T_S), trajanje izvođenja gospodara (T_M) i ukupan broj iteracija gospodara. U svakoj iteraciji gospodara sluge su izvršile ukupno 2000 selekcija i križanja (po 1000 svaki sluga). U tablici 6.5 su prikazane veličine prostora rješenja. Usporedba tih vrijednosti sa vremenima izvršavanja jasno pokazuje snagu genetskih algoritama u rješavanju kombinatoričkih problema visoke složenosti.

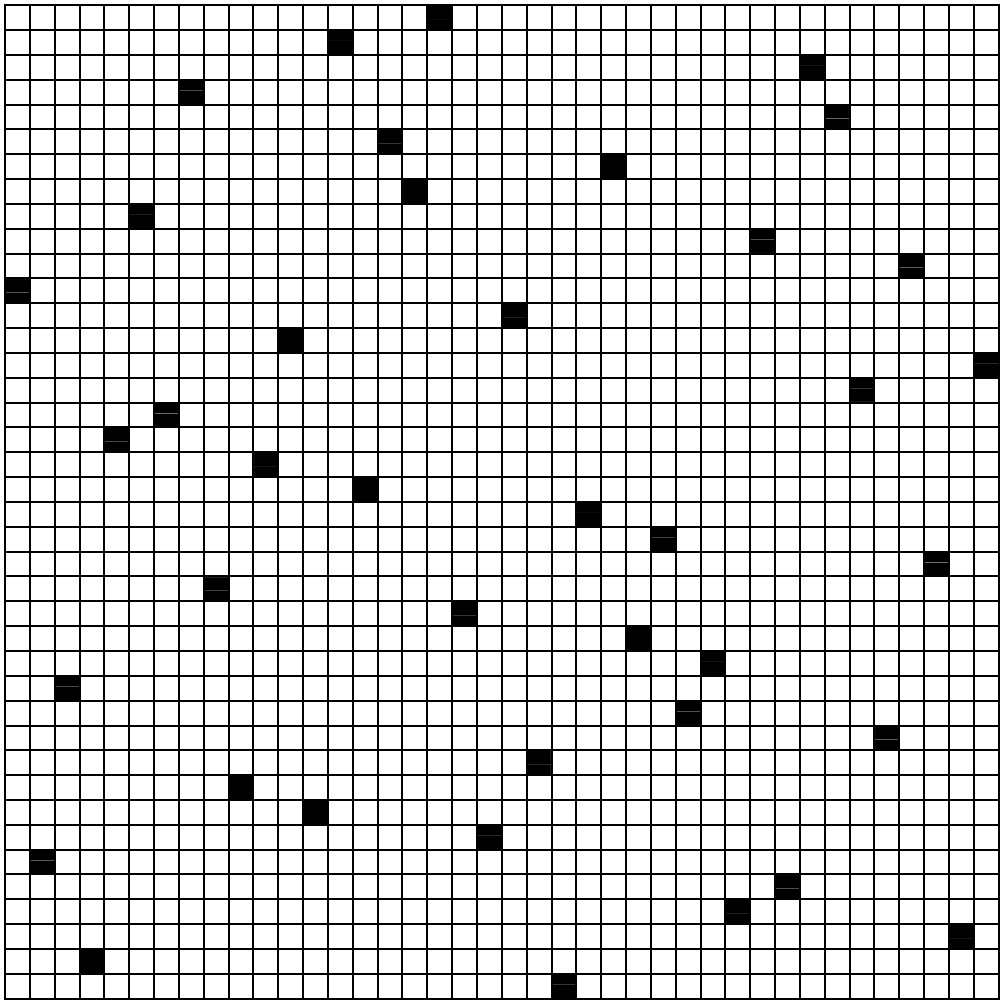
Na slikama 6.7 i 6.8 su dani grafički prikazi dva različita rješenja za problem 40 kraljica, iz čega je vidljivo da je genetski algoritam u stanju pronaći različita rješenja ukoliko ona postoje. Konačno, slike 6.9 i 6.10 pokazuju numerička rješenja za probleme 100 i 200 kraljica.

Tablica 6.4 Vremena izvođenja

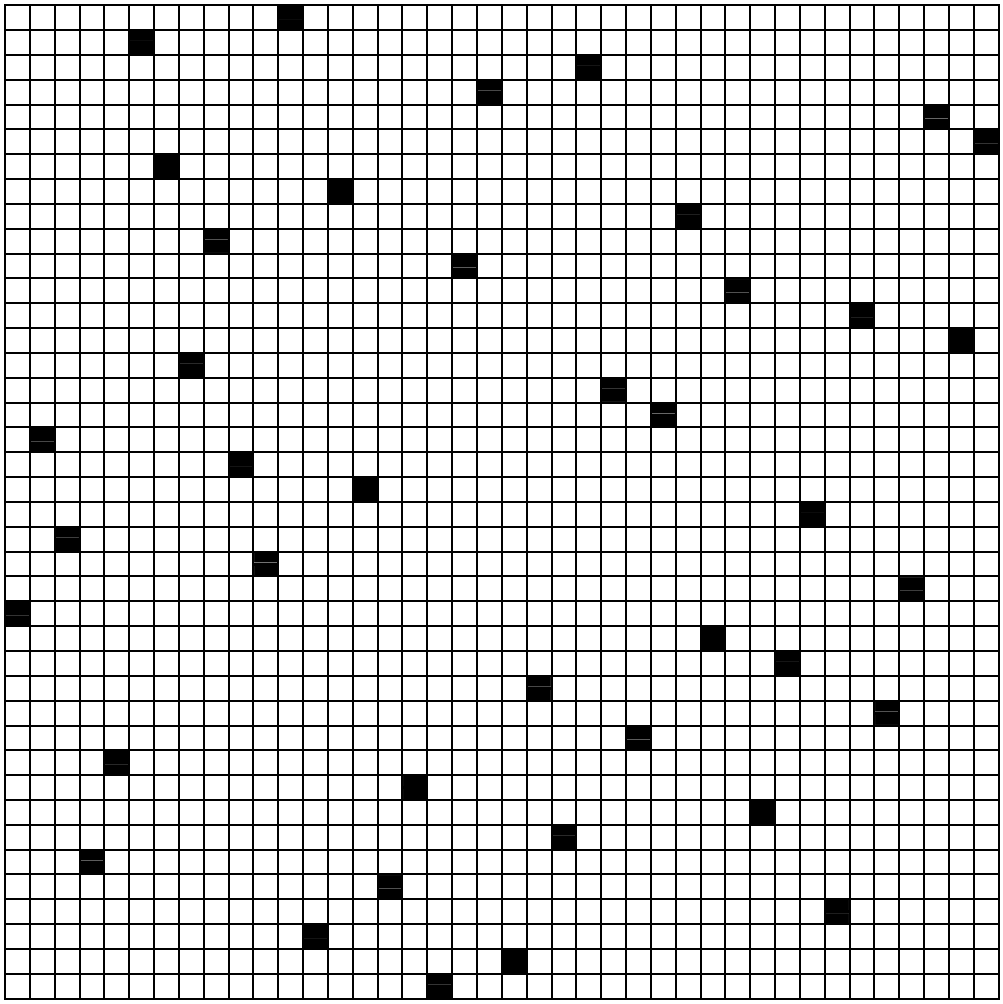
Broj kraljica	T_S	T_M	Broj iteracija
15	6.2 s	0.119 s	351
20	26.5 s	0.226 s	586
40	1.2 min	0.486 s	802
50	2.8 min	1.008 s	1482
100	7.8 min	1.241 s	1323
200	81 min	6.083 s	4205

Tablica 6.5 Veličina prostora rješenja

Broj kraljica	Prostor rješenja $n! \approx 10^x \approx 2^y$		
15	15!	$10^{12.12}$	$2^{40.25}$
20	20!	$10^{18.39}$	2^{61}
40	40!	$10^{47.91}$	2^{159}
50	50!	$10^{64.48}$	2^{214}
100	100!	10^{158}	2^{524}
200	200!	10^{375}	2^{1245}



Slika 6.7 Rješenje problema 40 kraljica



Slika 6.8 Još jedno rješenje problema 40 kraljica

```
(21, 52, 72, 60, 22, 81, 64,  9, 54, 88, 27, 78, 10, 28, 44,
 87, 76, 31, 11, 57, 94, 69, 51, 45, 32, 70, 13, 92,  6, 83,
 37, 75, 96, 38, 89, 46, 55, 50,  7, 20, 67, 84, 79, 77, 97,
 34, 19, 53,  2, 35,  5, 47, 26, 16, 93, 17,  8, 18, 82, 68,
 25, 65, 74, 42, 95, 23, 86, 43, 36, 85, 80, 71, 62, 98,  3,
 24, 48, 61,  4, 39,  0, 63, 12, 49, 59, 73, 30, 14, 29, 66,
 41, 58, 56, 40, 15,  1, 90, 33, 99, 91)
```

Slika 6.9 Rješenje problema 100 kraljica

```
( 29,  35,  99, 132,  83,   6, 141, 113, 152, 198,  28,  96,  69, 143,   8,
 195, 120, 158,  58, 161, 189,  62,   4,  20, 174, 140, 173,   1, 199,  51,
  79, 111,  52,  89,  33,  39,  75, 114,  65, 133, 104,  31,  22, 150, 147,
 183,  44, 119, 153,  12,  82,  11, 169,  64,  19, 179,  23,  80,  61, 159,
 118,  85, 125, 107, 112,  37, 184, 155, 136, 116, 196,  97, 145, 192,  59,
 188,  67, 172, 162, 178, 167,  24,  70,  17,  71, 177, 102,   9, 131,   0,
 123,  40,  16,  46,  73, 164,   2, 139, 194, 180, 100, 122,  38,  45, 171,
  13,  10,   5,  78,  36,  15, 170, 149,  41,  14,  91, 126,   7,  76, 157,
 106, 110,  90,  56,  25,  86,  77, 108, 163,  48, 175, 115, 151,  26,  98,
  27, 197, 190, 134, 117,  57, 187,  74,  81,  30,  60,  43,  94, 156,  84,
 165,  88,  68,  55, 168, 105, 129, 101, 160, 166,  42, 127, 193,  32,  18,
  50,   3,  87, 181,  95, 124, 130,  66, 185, 103, 146, 121, 186,  92, 135,
  63,  72, 137,  47, 109, 191, 148, 182,  49, 128, 142,  53, 138,  21,  93,
  34, 144, 154,  54, 176)
```

Slika 6.10 Rješenje problema 200 kraljica

7. ZAKLJUČAK

U ovom radu je bio prikazan globalni paralelni genetski algoritam (GPGA) i njegova primjena na optimiranje funkcija realnih varijabli i rješavanje problema *N* kraljica. Naglasak pri rješavanju tih problema je stavljen na promatranje povećanja brzine rada koju GPGA donosi u odnosu na sekvencijalni genetski algoritam.

Kroz prikaz nekih od genetskih operatora i postojećih struktura genetskih algoritama, dan je uvid u njihove prednosti i mane. Najveća prednost genetskih algoritama svakako je primjenjivost na velik broj različitih vrsta problema, pa je pomoću njih moguće rješavati sve probleme koji se mogu predstaviti kao problemi optimizacije, bez obzira na vrstu informacije koju je potrebno optimirati. Uzme li se pri tom u obzir jednostavnost implementacije i nadogradnje samog genetskog algoritma, zatim da za rješavanje nije potrebno detaljno poznavanje strukture problema, kao i neosjetljivost algoritma na višemodalne prostore rješenja, jasno je da su genetski algoritmi vrlo robusne i snažne metode optimiranja.

Ipak, kako su genetski algoritmi u osnovi heurističke metode, nije moguće garantirati pronalaženje globalnog optimuma. Genetski algoritam daje samo rješenje koje se nalazi *u blizini* globalnog optimuma, ali se preciznost rješenja se jednostavno može povećati ponavljanjem postupka sa, npr. suženim prostorom pretrage. Nadalje, genetski algoritam je često potrebno prilagoditi posebnim zahtjevima određenog problema, npr. definirati novu vrstu prikaza prikladnu za dani problem, te je za svaku vrstu prikaza je potrebno definirati kvalitetne genetske operatore. Definiranje *dobre* funkcije dobrote, koja kvalitetno prikazuje rješavani problem također može biti vrlo problematično, a nerijetko je potrebno prilagoditi funkciju dobrote zahtjevima genetskih operatora. Još jedan od velikih problema genetskih algoritama je i nepostojanje sustavnog postupka za određivanje dobrih početnih parametara, pa je određivanje parametara često svedeno na čisto eksperimentiranje. U odnosu na specijalizirane numeričke metode, genetski algoritmi konvergiraju puno sporije, a zbog izvođenja velikog broja računskih operacija, brzina rada je u usporedbi sa ostalim metodama relativno mala.

Potreba za ubrzavanjem rada genetskih algoritama u zadnja je dva desetljeća dovela do razvoja struktura genetskih algoritama koje ubrzavaju izvođenje algoritma paralelizacijom rada određenih dijelova. Zbog složenosti većine paralelnih genetskih algoritama, teorijska podloga njihova rada je još uvijek slabo razvijena.

Globalni paralelni genetski algoritam je svojom strukturom (osim paralelnog izvršavanja nekih operatora) identičan sekvencijalnom genetskom algoritmu, pa se na njega može u većem dijelu primijeniti teorija o radu sekvencijalnih genetskih algoritama.

Eksperimentima na optimiranju funkcija realnih varijabli pokazano je da je povećanje brzine rada GPGA gotovo linearno sa povećanjem broja slugu. Do odstupanja od linearnosti dolazi zbog izvršavanja genetskih operatora od strane gospodara (u danim primjerima je to mutacija), kao i trošenja vremena na komunikaciju gospodara sa slugama, čime je ujedno i ograničen maksimalan broj slugu za koji je izvršavanje GPGA efikasno.

Rješavanjem problema *N* kraljica pokazana je snaga genetskih algoritama na pretrazi velikih prostora rješenja koji se susreću pri rješavanju teških kombinatoričkih problema, za koje ne postoje determinističke metode rješavanja u realističnom vremenskom intervalu.

LITERATURA

[1] D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

[2] M. Golub, D. Jakobović, *Genetski Algoritam*, 1997.

[3] K.D. Crawford, *Solving n-Queens Problem Using Genetic Algorithms*, Tulsa University, -

[4] E. Cantú-Paz, *Designing Efficient Master-Slave Parallel Genetic Algorithms*, University of Illinois at Urbana-Champaign, 1997.

[5] E. Cantú-Paz, D.E. Goldberg, *Parallel Genetic Agorithms with Distributed Panmitic Populations*, IlliGAL Report No. 99006, January 1999.

[6] E. Cantú-Paz, *A Summary of Research on Parallel Genetic Algorithms*, IlliGAL Report No. 95007, July 1995.

[7] E. Cantú-Paz, *A Survey of Parallel Genetic Algorithms*, University of Illinois at Urbana-Champaign, -

[8] E.A. Williamsa, W.A. Crossley, *Empirically-Derived Population Size and Mutation Rate Guidelines for a Genetic Algorithm with Uniform Crossover*, School of Aeronautics and Astronautics, Purdue University, 1997.

[9] E.K. Burke, D.B. Varley, *A Genetic Algorithms Tutorial Tool for Numerical Function Optimisation*, Automated Scheduling and Planning Group, Department of Computer Science, University of Nottingham, -.

[10] *Opća enciklopedija*, Leksikografski zavod, 1982.