SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 108

# Rješavanje problema smještanja i povezivanja kod sklopa FPGA

Andi Škrgat

Zagreb, svibanj 2021.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Da bi ste uklonili ovu stranicu obrišite naredbu* `\izvornik`.

# SADRŽAJ

# 1. Uvod

Evolucijsko računarstvo (eng. *Evolutionary computing*) je područje računarske znanosti koje se bavi problemom optimizacije složenih funkcija, a nastali su proučavanjem procesa u prirodi. Preciznije bismo mogli evolucijsko računarstvo svrstati u područje umjetne inteligencije (eng. *Artificial intelligence*) i mekog računarstva (eng. *Soft computing*).

Roditeljski koncept na kojem se temelji jest pojam metaheuristika (eng. *Metaheuristic*), a to je pak vrlo općenita heuristika čija je zadaća pomoći heuristiki specifičnoj problemu, pri usmjeravanju prema prostoru u kojem se nalaze potencijalno dobra rješenja našeg konkretnog problema. Evolucijsko računarstvo možemo podijeliti na evolucijske algoritme u koje spadaju genetski algoritmi, genetsko programiranje, evolucijske strategije, evolucijsko programiranje zatim na kategoriju u koju spadaju algoritmi rojeva, a njih pak čine mravlji algoritmi, algoritmi roja čestica, algoritmi pčela i dr. Zadnju skupinu algoritama čine umjetni imunološki algoritmi, algoritam diferencijske evolucije te algoritam harmonijske pretrage.

Evolucijski algoritmi se koriste kako bi dobro aproksimirali rješenja problema koji se ne mogu jednostavno rješiti primjenom nekih drugih koncepata. Čest problem koji se rješava evolucijskim računarstvom je problem raspoređivanja. Ovaj rad se bavi jednim takvim problemom u FPGA sklopu (eng. *Field- programmable gate array*). Programiranje FPGA sklopa podrazumijeva nekoliko koraka, a ovaj rad se bavi razmještanjem i konfiguriranjem logičkih blokova, ulaza i izlaza. Za raspoređivanje koristimo genetski algoritam koji je jedan od najčešće korištenih algoritama iz područja evolucijskog računarstva u problemima raspoređivanja.

Najprije ću opisati princip rada FPGA sklopa i samih genetskih algoritama. Nakon toga prezentirat ću implementaciju sustava koji je sposoban rješiti navedeni problem, navesti njegove prednosti, nedostatke i ograničenja u uporabi. Rezultati su prezentirani kroz tablice, grafove i slike generiranih rješenja.

## 1.1. FPGA sklop

### 1.1.1. Arhitektura FPGA sklopa

FPGA je digitalni sklop koji se koristi za realiziranje funkcija specificiranih od strane programera uporabom sklopovski specifičnih jezika za programiranje. Primjer jednog takvog jezika je VHDL.

Najosnovnija arhitektura FPGA sklopa podrazumijeva ulazno-izlazne spojeve (eng. *Pin*) na koje se dovode ulazi željenih funkcija i odvode izlazi. Središnji dio sklopa čine LUT tablice (eng. *Lookup table*). One imaju $2^n$ zapisa, a svaki zapis je zadužen za pojedinu kombinaciju ulaznih varijabli. LUT tablice su dio CLB-ova eng. *configurable logic blocks*) na koje se dovode žice. Napomenimo kako je ostvarenje CLB bloka pomoću LUT tablica samo jedan od korištenih izvedbi, druge često korištene tehnike podrazumijevaju korištenje multipleksera i jednostavnih flip-flop bistabila. Žice najčešće ne dolaze samostalno već u snopovima i one su zadužene za provođenje signala dobivenog od ulaznih varijabla. Prospojne kutije (eng. *Switch box*) spajaju žice koje se nalaze u susjednim snopovima. Ovisno o poziciji prospojne kutije moguće jest da jedna prospojna kutija spaja dva, tri ili četiri susjedna snopa žica.

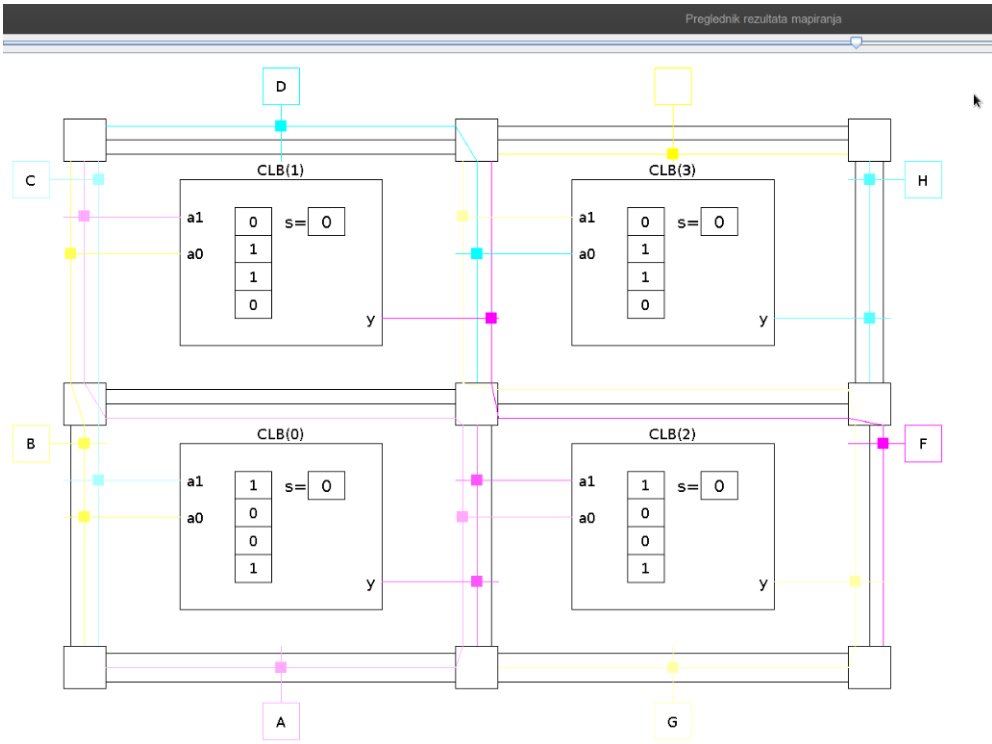
Primijetimo na temelju slike 1.1 kako u najopćenitijom slučaju dvije žice možemo spojiti na jako puno načina pa ne postoji nužno samo jedna ispravna konfiguracija konkretne prospojne kutije za ostvarenje željenog spoja.

### 1.1.2. Programski model FPGA sklopa

Implementacija cijelog sustava pisana je u programskom jeziku Java, a korištene su tehnologije Java 15 SDK za programiranje, Maven za upravljanje projektom, Github za verzioniziranje sustava i Java Swing za stvaranje grafičkih korisničkih sučelja (eng. *GUI*).

Model FPGA sklopa možemo definirati na dva načina: prvi ćemo zvati fizički model jer promatramo FPGA sklop kao stvarni fizički sustav i za njega pamtimo ono što imamo na raspologanju. Kod njega je specificirano koliko CLB-ova imamo na raspologanju, broj ulaza koji se dovode na pojedini CLB, broj žica koje se nalaze u jednom snopu, broj UI spojeva(eng. *Pin*) po svakom CLB-u, konfiguracija pojedine prospojne kutije te signali koji se nalaze na pojedinoj žici.

Drugi način jest da opisujemo FPGA kroz ono što želimo ostvariti i na koji način želimo to dobiti. Taj ćemo način opisivanja zvati logički model. Logički model zadajemo pomoću tekstualne datoteke u kojoj su navedeni potrebni parametri.




**Slika 1.1:** Arhitektura FPGA sklopa s četiri CLB-a, dva ulaza koji se dovode na pojedini CLB i tri žice u snopu. Fizički pogled na sklop.

Nakon opisa fizičkog i logičkog modela FPGA sklopa sada nam je lakše i razmišljati i vidjeti što je programski cilj ovog rada: uporabom genetskog algoritma logički model mapirati u fizički sustav. Možemo izvesti analogiju sa stvarnim životom u kojem je fizički model kutija u koju pokušavamo ubaciti specifični objekt na način da zadovoljava osnovne zahtjeve kutije.

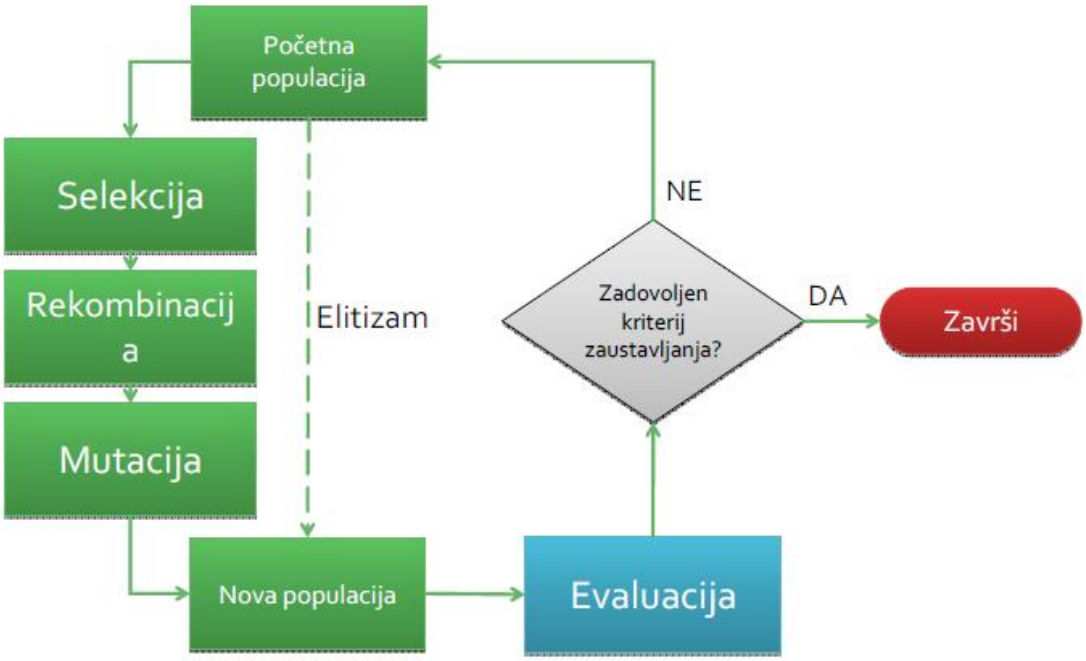
```
#f = a xor b; g = f xor c; h = g xor d
Architecture: 2, 4
Variables: [A,B,C,D]
Function(s) [F,G,H] is/are at outputs: [1, 2, 3]
CLB(0): C B
1001
CLB(1): A B
0110
CLB(2): CLB(0) A
1001
CLB(3): CLB(2) D
0110
```

**Slika 1.2:** Primjer zadavanja logičkog modela FPGA sklopa. Za svaki CLB piše koje točno varijable dovodimo, programiranje LUT tablice svakog CLB-a i koje funkcije želimo ostvariti na kojem izlazu.

# 2. Genetski algoritmi

## 2.1. Princip rada

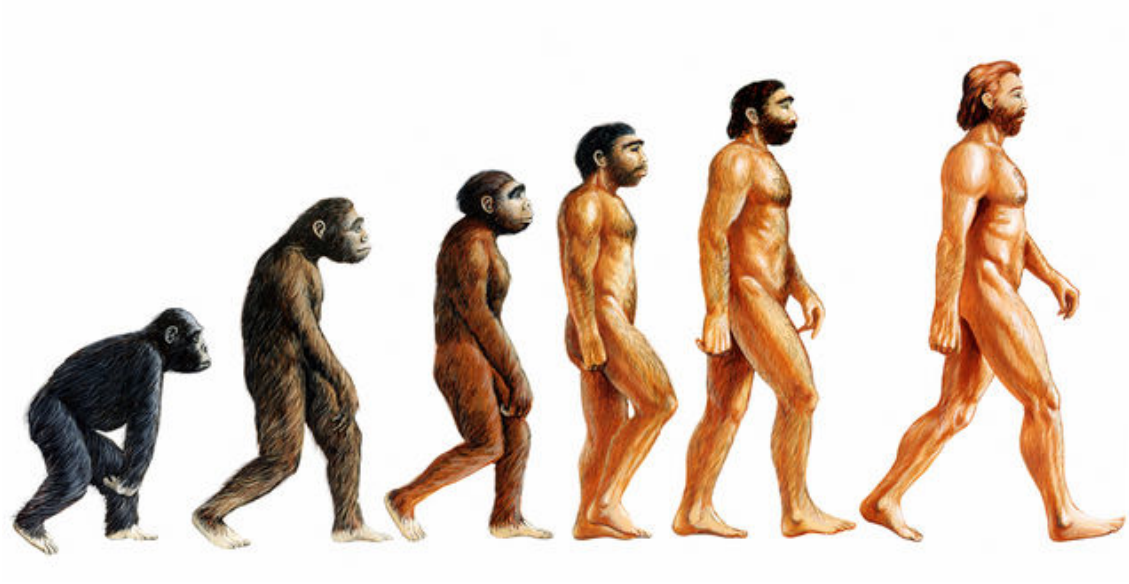
Genetski se algoritam zasniva na populaciji rješenja i najčešće se slučajnim procesima nasumično stvaraju početne jedinke. U složenijim problemima moguće je, prije pokretanja evolucijskog računanja, na temelju neke jednostavne heuristike izgraditi startnu populaciju. Takav pristup u većini slučajeva ubrzava konvergiranje algoritma. Populacija se s vremenom mijenja, a za to je zadužen proces selekcije, ali više o njemu u poglavlju **??**. Ono što ćemo za sada samo spomenuti jest to da selekcija ne mijenja nužno loše jedinke. Dapače, moguće jest da proces selekcije potpuno nasumično odabire roditeljske jedinke i da jedinku dijete zamijeni s najboljom jedinkom iz roditeljske populacije. U poglavlju **??** vidjet ćemo da elitistički pristup rješava taj problem.




**Slika 2.1:** Koraci u radu genetskog algoritma.

Genetski se algoritam može definirati kao heuristika pretraživanja koja inspiraciju temelji na Charles Darwinovoj teoriji evolucije. Ta se teorija temelji na 5 osnovnih pretpostavki:

1. plodnost vrsta - potomaka ima uvijek više, no što je potrebno,
2. veličina populacije je približno stalna,
3. količina hrane je ograničena,
4. kod vrsta koje se seksualno razmnožavaju, nema identičnih jedinki već postoje varijacije te
5. najveći dio varijacija prenosi se nasljeđem. [(Čupić, 2012)]



**Slika 2.2:** Darwinova teorija evolucije. Iz generacije u generaciju vrijednost funkcije dobrote prosječno raste.

Algoritam preslikava proces prirodne evolucije gdje su neke jedinke primjenom evolucijskog operatora selekcije izabrane kao roditelji koji će sudjelovati u procesu križanja kako bi stvorili genetski materijal jedinke djeteta. Postoji nebrojeno puno inačica genetskog algoritma, ali osnovna ideja je sljedeća: algoritam radi s populacijom jedinki pri čemu je svaka jedinka jedno potencijalno rješenje problema i nju nazivamo kromosomom. Za svaku jedinku računamo funkciju dobrote (eng. *fitness*), a ona može

biti izvedena kao funkcija nagrade, funkcija kazne ili pak hibrid. Cilj je da funkcija dobrote vjerno oslikava kvalitetu pojedine jedinke pa je tako, ovisno o vrsti problema, cilj minimizirati ili maksimizirati funkciju dobrote. Korištenjem evolucijskog operatora selekcije odabiru se kromosomi koji će sudjelovati u rekombinaciji gena - dio kromosoma s ciljem stvaranja bolje jedinke. Nekad međutim, to neće biti slučaj, odnosno moguće je da nova jedinka gora od roditeljskih. Taj postupak nazivamo križanje. Mutacijom mijenjamo pojedine gene kromosoma kako bismo dodatno utjecali na prostor pretraživanja stanja. Operatorom zamjene mijenjamo jedinku iz roditeljske populacije s novom jedinkom kako bismo očuvali veličinu populacije.

## 2.2. Traženje rješenja

Genetski algoritam služi za rješavanje težih optimizacijskih problema, za koje ne postoji egzaktna matematička metoda rješavanja ili su NP teški problemi pa se za veći broj nepoznanica ne mogu riješiti u razumnom vremenu [(Čupić et al., 2013)]. To radi na način da pretražuje prostor stanja pa možemo formalno predstaviti genetski algoritam kao složenu funkciju $g : J^N => J^N$ koja uz pomoć genetskih operatora preslikava roditeljsku populaciju u novu populaciju.

```
1 Genetski_algoritam(){
2     generiraj_pocetnu_populaciju();
3     dok (nije_zadovoljen_uvjet_zavrsetka_evolucijskog_procesa){
4         selektiraj_bolje_jedinke_za_reprodukciju();
5         reprodukcijom_generiraj_novu_populaciju();
6     }
7 }
```

**Slika 2.3:** Ciklički proces rada genetskog algoritma.

Sada smo sposobni definirati proces stvaranja genetskog algoritma. Prvi korak jest definirati prikaz jedinke. Tu možemo genetski algoritam podijeliti na one koji rade s bitovima i na one koji rade s nekakvom drugom strukturom podataka. U većini je slučajeva najpogodnije raditi sa strukturom polje (eng. *Array*) jer algoritam radi sa fiksnom veličinom populacije pa operacije dohvaćanja, postavljanja i zamjene možemo raditi u O(1) vremenskoj složenosti (eng. *Big O notation*). Nakon toga slijedi fino podešavanje parametara genetskog algoritma kako bismo dobili što je moguće bolje rješenje. Nerijetko se u praksi parametri najprije odrede na temelju nekog jednostavnog problema, nakon čega se ide na glavni slučaj. Za ubrzavanje vremena izvođenja moguće je, ovisno o izvedbi, paralelizirati dijelove algoritma kako bi se maksimalno

ubrzalo izvođenje. Spomenimo za kraj ovog dijela kako genetski algoritmi nisu deterministički algoritam koji će uvijek dati točno rješenje, već postupak kojim se u zadanom vremenu pokušava pronaći što je moguće bolje rješenje.

## 2.3. Evolucijski operatori

### 2.3.1. Selekcija

Selekcija je evolucijski operator čiji je zadatak osigurati da bolje jedinke češće sudjeluju u produkciji novih rješenja kako bismo usmjerenim pretraživanjem došli u prostor koji nam može ponuditi optimalno rješenje. Takav mehanizam možemo opisivati kroz više parametara, a prvi od njih je vrijeme preuzimanja (eng. *Takeover time*). To je vrijeme koje je potrebno operatoru selekcije da generira populaciju u kojoj se nalaze samo najbolja rješenje [Deb. i Goldberg (1991)]. Radi se o tome da kada bi novu populaciju stvarali samo operatorom selekcije koja favorizira bolje jedinke, tada je logično očekivati da će u svakoj novoj generaciji biti sve više i više kopija najboljeg rješenja. Vrijeme preuzimanja se najčešće mjeri u generacijama, a za k-turnirsku selekciju aproksimativan izraz je odredio Bäck (1996)

$$\rho = \frac{1}{\ln n}(\ln k + \ln(\ln k))) \tag{2.1}$$

gdje je n veličina populacije a k broj jedinki koje ulaze u turnir. Mehanizam selekcije može se promatrati i kroz selekcijski intenzitet koji se računa kao povećanje srednje dobrote populacije prije i nakon primjene operatora selekcije, podijeljenog sa standardnom devijacijom populacije: [Bäck (1995); Miller i Goldberg. (1995); Mühlenbein i Schlierkamp-Voosen (1993); Thierens (1997)]

$$I = \frac{\bar{f}_s - \bar{f}}{\sigma} \tag{2.2}$$

gdje je $\bar{f}$ prosječna dobrota rješenja u roditeljskoj populaciji, $\bar{f}_s$ prosječna dobrota rješenja u novoj populaciji, a $\sigma$ standardna devijacija dobrote rješenja u roditeljskoj populaciji. Iz izraza vidimo kako je selekcijski intenzitet obrnuto proporcionalan standardnoj devijaciji, ali se može pokazati i puno jača tvrdnja, kako je selekcijski intenzitet obrnuto proporcionalan brzini konvergencije.

$$\sigma = \frac{1}{n}\sum_{1}^{n}(f_i - \bar{f})^2 \tag{2.3}$$

Drugi parametri koji se koriste u okviru selekcije nisu toliko česti pa ćemo ih preskočiti.

Postoji velik broj selekcija, a mi ćemo se u ovom radu dotaknuti dvije najpoznatije, a to su k-turnirska selekcija i proporcionalna selekcija (eng. *Roulette-wheel selection*).

### K-turnirska selekcija

K-turnirska selekcija spada u turnirske selekcije, a neki od ostalih predstavnika te grupe su binarna turnirska selekcija i Boltzmannova turnirska selekcija. Radi se o postupku u kojem se nasumičnim odabirom izabere k kromosoma i vrati najbolji od njih. Klasični genetski algoritam najčešće pretpostavlja selektiranje dva kromosoma koji će služiti kao roditelji pa je moguće selekciju izvesti na način da roditelji budu i duplikati i različiti. Selekcijski intenzitet kod turnirske selekcije je određen izrazom

$$\sqrt{\sqrt{2}\ln k} \tag{2.4}$$

gdje je k broj jedinki koji ulazi u turnir [(Bäck, 1995)].

**Pseudokod 2.4.1 — $k$-turnirska selekcija.**
Ulaz: populacija $P$ jedinki veličine VEL_POP; $k \leq$VEL_POP. Izlaz: jedna jedinka

- Inicijaliziraj pomoćnu populaciju $P'$ na praznu
- Ponavljaj $k$ puta:
  - Iz populacije $P$ slučajnim postupkom uz jednoliku distribuciju vjerojatnosti odaberi jednu jedinku koja još nije u populaciji $P'$
  - Ubaci je u populaciju $P'$
- Vrati najbolju (ili najgoru - ovisno što tražimo) jedinku iz populacije $P'$

**Slika 2.4:** Pseudokod k-turnirske selekcije

### Proporcionalna selekcija

Proporcionalna selekcija je najpoznatiji predstavnik grupe mehanizama selekcija koje svoj odabir temelje isključivo na dobroti jedinka. Kod ovog se postupka svakoj jedinki pridruži vjerojatnost odabira koja je proporcionalna njezinoj dobroti:

$$p_i = \frac{f_i}{\sum_1^n f_j} \tag{2.5}$$

Ovakav mehanizam podrazumijeva da su dobrote svih jedinki nenegativne pa je nužno dobrote translatirati u prostor $R^+$. Translaciju je dobro napraviti i zbog poznatog problema skale. Naime, ako su dobrote jedinki približno jednaki veliki brojevi,

onda su i vjerojatnosti odabira svake od jedinke gotovo iste. Naprimjer, uzmimo da se populacija sastoji od 5 jedinki i da njihove dobrote redom iznose 10000001, 10000002, 10000003, 10000004 i 10000005. Zadnja jedinka ima najveću vrijednost dobrote pa želimo sa što većom vjerojatnošću nju odabrati. Međutim, ako pustimo ovakve vrijednosti, vjerojatnosti njihovog odabira su identične do na 6. decimalu i iznose približno $\approx 20\%$. Translacijom za minimalnu vrijednost, u ovom slučaju 10000001, nove vrijednosti dobrota su 0, 1, 2, 3 i 4 pa je primjerice vjerojatnost odabira najbolje jedinke sada jednaka 40%.
Selekcijski intenzitet proporcionalne selekcije možemo izračunati izrazom:

$$I = \frac{\sigma}{\bar{f}} \tag{2.6}$$

gdje obje varijable se odnose na trenutačnu populaciju s kojom radimo.

**Pseudokod 2.4.2 — Implementacija proporcionalne selekcije.**

Ulaz: P - populacija
Izlaz: odabrana jedinka

```
suma = 0
za svaku jedinku J iz P: suma += J.dobrota

trokutic = random(0, suma)

podrucje = 0
za svaku jedinku J iz P:
  podrucje += J.dobrota
  ako je trokutic <= podrucje
    vrati J
  kraj ako
kraj za
```

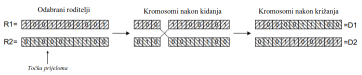
**Slika 2.5:** Pseudokod proporcionalne selekcije

### 2.3.2. Reprodukcija

#### Križanje

Križanje je evolucijski operator koji stvara djecu. Izvedeno je iznimno puno načina križanja, ali svi imaju isti cilj: kombinirati gene roditelja kako bi se stvorilo što je moguće bolje dijete.
Ako se vratimo na našu pretpostavku kako genetski algoritam pretražuje prostor onda

je križanje fina pretraga u okolini jedinka roditelja. Takav način rada ima svoju prednost upravo zbog toga jer na jako precizan način prolazi po jednom dijelu prostora rješenja, ali s druge strane moguće je da zbog toga izgubimo optimalnost. Uzmimo za primjer optimizacijski problem pronalaska maksimuma funkcije na nekom intervalu gdje se za operator križanja koristi aritmetička sredina. Ako selekcija u početku izbaci jedinku koja ima veliku vrijednost funkcije na nekom intervalu, ali ne na cijelom (lokalni optimum) tada će takvo križanje u tom skraćenom intervalu najvjerojatnije stvarno naći maksimum, ali to neće biti globalni optimum.
Osim križanja aritmetičkom sredinom često se koristi križanje s jednom točkom prekida, križanje s više točka prekida i uniformno križanje. Kod križanja s jednom točkom prekida nasumičnim odabirom se odredi jedna točka do koje se u dijete ubacuju svi geni iz prvog roditelja, a od točke naprijed svi geni drugog roditelja. Uniformnim križanjem za svaki gen biramo roditelja te je takvo križanje neprikladno ukoliko nam je važna sekvenca gena, a ne samo jedan gen. Križanje s više točka prekida je analogno križanju s jednom točkom prekida samo što se geni prekidaju na više mjesta.




**Slika 2.6:** Križanje s jednom točkom prijeloma


**Slika 2.7:** Križanje s više točka prijeloma

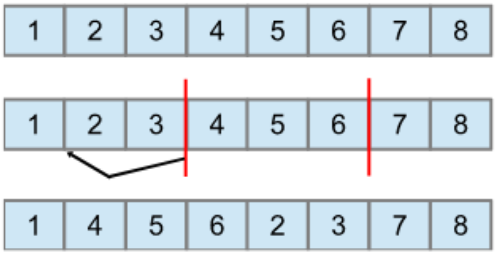
### Mutacija

Mutacija ima upravo suprotnu ulogu od križanja. Ona radi grubu pretragu prostora pretraživanja na način da promijeni određene gene i time izbaci jedinku iz potencijalno lokalnog optimuma i vrati je u širi prostor u kojem je moguće pronaći globalni. Upravo iz tog razloga mutacija povećava volumen populacije, a križanje smanjuje.

**Mutacija zamjenom (eng. *Swap mutation*)** Kod ove vrste mutacije nasumičnim odabirom se izaberu dvije pozicije i geni se na tim pozicijama jednostavno zamjene.

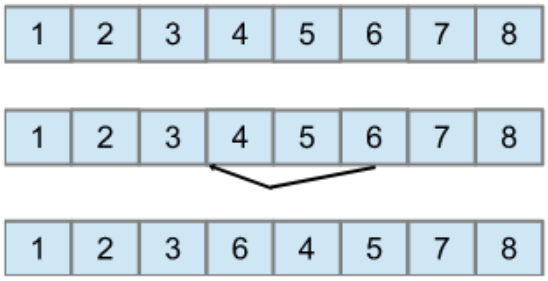
**Mutacija premještanjem (eng. *Cut mutation*)** Nasumičnim odabirom izaberemo dvije pozicije te sekvencu gena između te dvije pozicije stavimo iza treće nasumično

odabrane pozicije. Geni koji su bili iza treće nasumično odabrane pozicije stavljamo na prvu slobodnu poziciju iza.



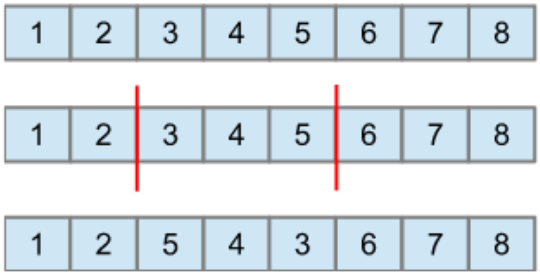

**Slika 2.8:** Mutacija premještanjem

**Mutacija umetanjem (eng. *Insertion mutation*)** Slučajno se odabere jedna pozicija i iz te pozicije se uzme jedan gen koji se stavlja iza druge nasumično odabrane pozicije.




**Slika 2.9:** Mutacija umetanjem

**Jednostavna mutacija inverzijom (eng. *Simple inversion mutation*)** [(Holland, 1975)] Nad kromosomom se odaberu dvije slučajne pozicije te se tako selektirani podniz prepiše obrnutim redoslijedom.




**Slika 2.10:** Jednostavna mutacija inverzijom

**Vjerojatnosna mutacija** Ovakva mutacija zahtijeva postojanje parametra koji određuje s kojom vjerojatnošću ćemo promijeniti svaki gen. Taj parametar zvati ćemo $\alpha$. Zatim se za svaki gen u kromosomu računa slučajna vrijednost i ako je manja od vrijednosti parametra onda se taj gen mijenja.

### 2.3.3. Parametri genetskog algoritma

U svakom genetskom algoritmu vrlo je važno pronaći dobar omjer između svih parametara koji se koriste, a pri tome mislimo na:

1. veličinu populacije
2. broj generacija
3. selekcija
4. mutiranje
5. križanje

Također, važno je reći da ne postoje točne vrijednosti parametara, nego se do njih dolazi eksperimentiranjem i iskustvom. Da bi genetski algoritam funkcionirao na način na koji mi to želimo jako je važno pronaći optimalne vrijednosti gore navedenih parametara. Posebice su nam bitni mutacija i križanje s kojima pretražujemo prostor rješenja. Ako je kontrakcijsko djelovanje mutacije prejako, ono će konzistentno uništavati sve pozitivno što je postupak pretraživanja do tada pronašao i pretragu će svesti na slučajnu [(Čupić, 2019)]. Treba težiti tome da su mutacija i križanje u balansu kako bismo osigurali napredak u pretraživanju.

| Parameter | Value |
|---|---|
| Elitism rate | 0.10 |
| Crossover rate | 0.90 |
| Mutation rate | 0.01 |
| Number of individuals | 37 |
| Number of variables | 864 |
| Optimization minutes | 30 |

**Slika 2.11:** Primjer parametara genetskog algoritma

### 2.3.4. Eliminacijski genetski algoritam

Eliminacijski genetski algoritam je takav algoritam kod kojeg u svakoj generaciji stvaramo samo jedno novo dijete te ga ubacujemo na mjesto neke od roditeljske jedinke, najčešće je to najlošija jedinka. Najjednostavnija verzija tog algoritma je ona u kojoj se nasumično odabiru tri jedinke križaju bolje dvije i stavljaju na mjesto treće ako je bolja od nje.

**Pseudokod 2.1.1 — Eliminacijski genetski algoritam.**

- Generiramo slučajnu populaciju mozgova od VEL_POP jedinki; evaluiramo svaki.
- Ponavljamo dok nije kraj:
  - Biramo slučajno tri jedinke
  - Dijete = Križamo bolje dvije + Mutacija
  - Vrednujemo dijete i ubacimo ga na mjesto treće ako nije lošije od nje

**Slika 2.12:** Pseudokod eliminacijskog genetskog algoritma

### 2.3.5. Generacijski genetski algoritam

Generacijski genetski algoritmi u svakoj generaciji stvaraju potpuno novu populaciju te roditeljsku zamjenjuju populacijom djece.

**Pseudokod 2.3.1 — Generacijski genetski algoritam.**

- Generiraj slučajnu populaciju mozgova od VEL_POP jedinki; evaluiraj svaki.
- Ponavljaj dok nije kraj:
  - Inicijaliziraj pomoćnu populaciju na praznu.
  - Ponavljaj dok veličina pomoćne populacije ne postane jednaka veličini populacije roditelja
    - Odaberi dva roditelja iz populacije roditelja

**Slika 2.13:** Pseudokod generacijskog genetskog algoritma

### 2.3.6. Elitizam

Elitizam je svojstvo algoritma da ne može izgubiti najbolje pronađeno rješenje [(Čupić, 2019)]. Kod eliminacijske verzije ono je najčešće očuvano dok se kod generacijskog npr. to može jednostavno postići umetanjem jedinke na početak nove populacije.

## 2.4. Implementacija sustava

### 2.4.1. Opis modela

Programski dio završnog rada obuhvaćao je izradu sustava koji je sposoban logički model mapirati u fizički model FPGA sklopa. Taj se sustav ne koristi sam za sebe već leži između dijela koji radi dekompoziciju funkcija koje želimo ostvariti i dijela koji je zadužen za prikaz i simulaciju rada FPGA sklopa. Oba dijela su izrađena od strane izv. prof. dr. sc Marka Čupića.


**Slika 2.14:** Arhitektura cijelog sustava koji se koristi

Kao što je već i napisano za rad je korišten genetski algoritam. Implementirano je sveukupno 70 inačica algoritma, n načina križanja i n načina mutiranja koji će biti objašnjeni kasnije u zasebnim poglavljima. U dodatku se nalazi detaljniji opis svih verzija algoritma, a u zasebnim poglavljima detaljnije su opisani oni koji su davali najsmislenije rezultate

Svaki optimizacijski algoritam radi s populacijom jedinki pa je prvo što je trebalo napraviti izgraditi model jedinke. Jedinka u ovom sustavu je predstavljena klasom ***AIFPGAConfiguration***.

Ona sadrži polja cijelih brojeva *clbIndexes* i *pinIndexes*.

*ClbIndexes* je polje određeno brojem CLB-ova koji su nam na raspolaganju u FPGA modelu. U ovom radu koristio sam dva pristupa, jedan u kojem sam htio očuvati različitosti svih brojeva koji se pojavljuju u CLB-ovima te drugi slobodniji pristup kod koje su se mogli naći duplikati u konfiguraciji pa su se kažnjavali preko funkcije dobrote.

#### Clb pozicije

Jedna validna postavka polja *clbIndexes* izgleda ovako:

**Tablica 2.1:** *clbIndexes*

| | | | |
|---|---|---|---|
| 1 | 2 | 0 | 3 |

```java
package hr.fer.zemris.bachelor.thesis.mapping.configuration;


import hr.fer.zemris.fpga.FPGAModel.FPGAModelConfiguration;

/**
 * Class that will serve as individual in genetic algorithm application.
 * @author andi
 *
 */
public class AIFPGAConfiguration {

	/**
	 * Saves configuration
	 */
	public FPGAModelConfiguration configuration;

	/**
	 * clb indexes for saving clb positions
	 */
	public int[] clbIndexes;

	/**
	 * Saves pin positions
	 */
	public int[] pinIndexes;

	public static final Object MULTIPLE_LABELS = new Object();


	public AIFPGAConfiguration(FPGAModelConfiguration configuration, int[] clbIndexes, int[] pinIndexes) {
		super();
		this.configuration = configuration;
		this.clbIndexes = clbIndexes;
		this.pinIndexes = pinIndexes;
	}

}
```

**Slika 2.15:** AIFPGAConfiguration jedinka genetskog algoritma

Ovakvom postavkom dogodilo bi se sljedeće: na poziciji 1 u stvarnom fizičkom modelu FPGA sklopa nalazio bi se onaj CLB koji se dekompozicijom našao na poziciji 0 u logičkom modelu, na poziciji 2 bio bi CLB koji se u logičkom modelu našao na poziciji 1 itd.

### Pin pozicije

Slično izgleda i polje *pinIndexes*. Kod njega se prvih m indexa koristi za ulaze i redom se postavljaju svaka od M ulaznih varijabli, a zatim sljedećih n za izlaze. Moguće je da neki od pinova ostane neiskorišten, npr. ako imamo četiri ulazne varijable, tri izlazne i osam pinova ukupno, onda će jedan pin ostati neiskorišten i na njemu se neće generirati nikakav signal.

**Tablica 2.2:** *pinIndexes*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 0 | 2 | 1 | 6 | 3 |

Ako iskoristimo prijašnji primjer s četiri ulaza, tri izlaza i jednim neiskorištenim pinom onda će ovakva konfiguracija *pinIndexes* na pinove redom postavljati ove labele (pretpostavka je da su ulazne varijable redom A, B, C, D te izlazne F, G, H):

**Tablica 2.3:** *pinLabels*

| | |
|---|---|
| 0 | D |
| 1 | G |
| 2 | F |
| 3 | - |
| 4 | A |
| 5 | B |
| 6 | H |
| 7 | C |

Osim ova dva polja ***AIFPGAConfiguration*** razred agregira ***FPGAModelConfiguration*** razred.

### Prospojna kutija

Prvo što ćemo opisati kod ovog razreda jest konfiguracija prospojne kutije. One su definirane kao 3D polje gdje je jedna dimenzija određena brojem prospojnih kutija, a druge dvije dimenzije su veličine 4*broj žica u jednom snopu. Razlog zašto broj množimo s 4 je zato što prospojna kutija može spajati do 4 susjedna snopa žica, a 2D je polje zato što su veze usmjerene, a ne dvosmjerne pa je primjerice vezu od žice 2 jednog snop do žice 1 drugog snopa potrebno zapisati na dva mjesta. U svakom od *n* prospojnih kutija nalazi se 2D polje bajtova, a u svako od polja može biti upisano 0, 1 ili 2. Ako je upisana 0, onda spoj ne postoji. Ako je upisana 1 to znači da spoj prima signal na toj žici i moguće je za svaku žicu primiti najviše jedan signal iz prospojne kutije. Ako je upisana 2 onda spoj šalje signal iz te žice odnosno možemo reći da je žica izlazna. Iz ovdje navedenih svojstava možemo izvesti zaključak kako je validna konfiguracija prospojne kutije ona koja u svakom retku ima najviše jednu 1 ili više 2 te ako je na mjestu [j][k] upisana 1 onda na poziciji [k][j] mora biti upisana 2 te ako je na [j][k] = 0 onda je i na [k][j] = 0. Također, žice iz istog snopa ne smiju biti povezane na nikakav način.

```
243         public class FPGAModelConfiguration {
244                 public byte[][][] switchBoxes;
245                 public byte[][] clbInIndexes;
246                 public byte[] clbOutIndexes;
247                 public byte[] pinIndexes;
248                 public boolean[] pinInput;
249
250                 public void copyFrom(FPGAModelConfiguration c) {
251                         for(int i = 0; i < pinInput.length; i++) {
252                                 pinInput[i] = c.pinInput[i];
253                         }
254                         for(int i = 0; i < pinIndexes.length; i++) {
255                                 pinIndexes[i] = c.pinIndexes[i];
256                         }
257                         for(int i = 0; i < clbOutIndexes.length; i++) {
258                                 clbOutIndexes[i] = c.clbOutIndexes[i];
259                         }
260                         for(int i = 0; i < clbInIndexes.length; i++) {
261                                 for(int j = 0; j < clbInIndexes[i].length; j++) {
262                                         clbInIndexes[i][j] = c.clbInIndexes[i][j];
263                                 }
264                         }
265                         for(int i = 0; i < switchBoxes.length; i++) {
266                                 for(int j = 0; j < switchBoxes[i].length; j++) {
267                                         for(int k = 0; k < switchBoxes[i][j].length; k++) {
268                                                 switchBoxes[i][j][k] = c.switchBoxes[i][j][k];
269                                         }
270                                 }
271                         }
272                 }
273         }
274
```

**Slika 2.16:** FPGAModelConfiguration razred, dio jedinke

### Ostali podatkovni članovi

***FPGAModelConfiguration*** sadržava još i *clbInIndexes*, *clbOutIndexes* i *pinIndexes*, a svi oni pamte istu stvar: s koje od n žica, ako je n broj žica u snopu, prihvaćaju signal i ako je riječ o CLB-u - prosljeđuju dalje.

## 2.5. Algoritmi

Slijedi opis algoritama koji se spominju u završnom radu. Čitatelja se upućuje na DODATAK KOJI NE POSTOJI za ostale verzije algoritma. Najprije opišimo dijelove

**Tablica 2.4:** *Validna konfiguracija prospojne kutije u kojoj se snop žica sastoji od samo jedne žice.*

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 2 |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |

**Tablica 2.5:** *Neispravna konfiguracija prospojne kutije u kojoj se snop žica sastoji od samo jedne žice. Ne može se dogoditi da jedna žica prima više signala (redak 0) te da jedna žica istovremeno šalje i prima signal (redak 3)*

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 2 |
| 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 |

koji koriste svi algoritmi.

### Čistači

Svaki algoritam koji koristi takvo križanje ili takvu mutaciju koja može stvoriti neispravnu verziju neke prospojne kutije, koristi čistače.

**Jednostavni čistač** Postoji jednostavni čistač ***SimpleSwitchBoxCleaner*** koji ako spoj nije validan, postavlja ga na 0. Takva se situacija može dogoditi ako je konfiguracija prospojne kutije takva da je pozicija [j][k] = 1, a pozicija [k][j] != 2 i obrnuto. Druga mogućnost jest da je spoj već definiran kao izlazni pa da se pokušava redefinirati kao ulazni i obrnuto te zadnja mogućnost da spoj već prima signal s neke žice pa onda po definiciji ne smije primati signal s neke druge žice.

**Napredniji čistač** Napredniji čistač ***AdvancedSwitchBoxCleaner*** će pokušati zamijeniti pozicije [j][k] i [k][j] te time barem eliminirati potencijalne greške navedene kao mogućnost 2.

### 2.5.1. Inicijalizatori i nasumični kreatori

Inicijalizator koristi ***AIFPGAConfigurationRandomizer*** koji je zadužen za početno generiranje populacije. On to radi tako da generira jednu po jednu jedinku koristeći pritom postavljene startne postavke:

1. U svakoj prospojnoj kutiji napravi se m validnih spojeva gdje je m nasumični broj iz intervala $[1, dohvatiMaxBrojVezaZaProspKutiju()]$. Maksimalan broj veza koji se može napraviti u svakoj prospojnoj kutiji je:

$$maxVeze = 2 * brojUlaza - 1 \tag{2.7}$$

2. ClbInIndexes, clbOutIndexes i pinIndexes iz FPGAModelConfiguration se generiraju potpuno nasumično

3. clbIndexes i pinIndexes su nasumični različiti brojevi

### 2.5.2. SV66

Ovaj algoritam koristi napredni čistač prospojne kutije, križanje *ValidCrosser* koje je opisano u poglavlju: te proporcionalnu selekciju. Koristi se eliminacijski algoritam na način da je omogućen elitizam, a nova jedinka se stavlja na mjesto najlošije u roditeljskoj populaciji.

### 2.5.3. SV70

Koristi sve iste dijelove kao i 2.5.2, samo što je ova verzija generacijski elitistički algoritam.

### 2.5.4. SV16

Ova verzija algoritma koristi jednostavni čistač prospojnih kutija, vjerojatnosnu mutaciju kojoj se parametar $\alpha$ može podesiti te križanje s jednom točkom prijeloma. Primijetimo da ova verzija algoritma ne radi stalno s validnim konfiguracijama, nego je primjerice moguće da postoje duplikati u polju *clbIndexes* koje bi, po definiciji, trebalo sadržavati sve različite vrijednosti. Po načinu rada ovo je eliminacijski genetski algoritam s jednostavnom troturnirskom selekcijom u koji je ugrađen elitizam.

### 2.5.5. SV31

Eksperimentalno, iz nekih algoritama sam maknuo mutaciju. SV31 algoritam je jedan od takvih. Koristi jednostavni čistač prospojnih kutija i uniformno križanje. Možemo ga kategorizirati kao generacijski algoritam u koji nije ugrađen elitizam.

### 2.5.6. Funkcija dobrote

Svaki genetski algoritam ima svoja nužna ograničenja koja se brinu o tome da zadovoljavaju svojstva sustava. U našem problemu kojeg rješavamo postoje dva nužna ograničenja koja svako rješenje mora zadovoljiti da bi bilo iskoristivo:

1. Ulazi na svaki CLB sklop moraju biti točno oni koji su dobiveni dekompozicijom logičkih funkcija.

2. Izlazi svakog CLB sklopa moraju biti odvedeni na točno onaj UI spoj (eng. *Pin*) koji je specificiran u tekstualnoj datoteci kojom zadajemo logički model.

#### Hibridna funkcija dobrote

Prva verzija algoritma radila je s hibridnom funkcijom dobrote u kojoj su se dobre stvari nagrađivale, a loše kažnjavale. Takav model se nije pokazao dobrim iz dva razloga:

1. Nijedan od 70 inačica algoritma ni nakon $\approx 100000$ pokretanja nije pronašao rješenje ni za najjednostavniji primjer.

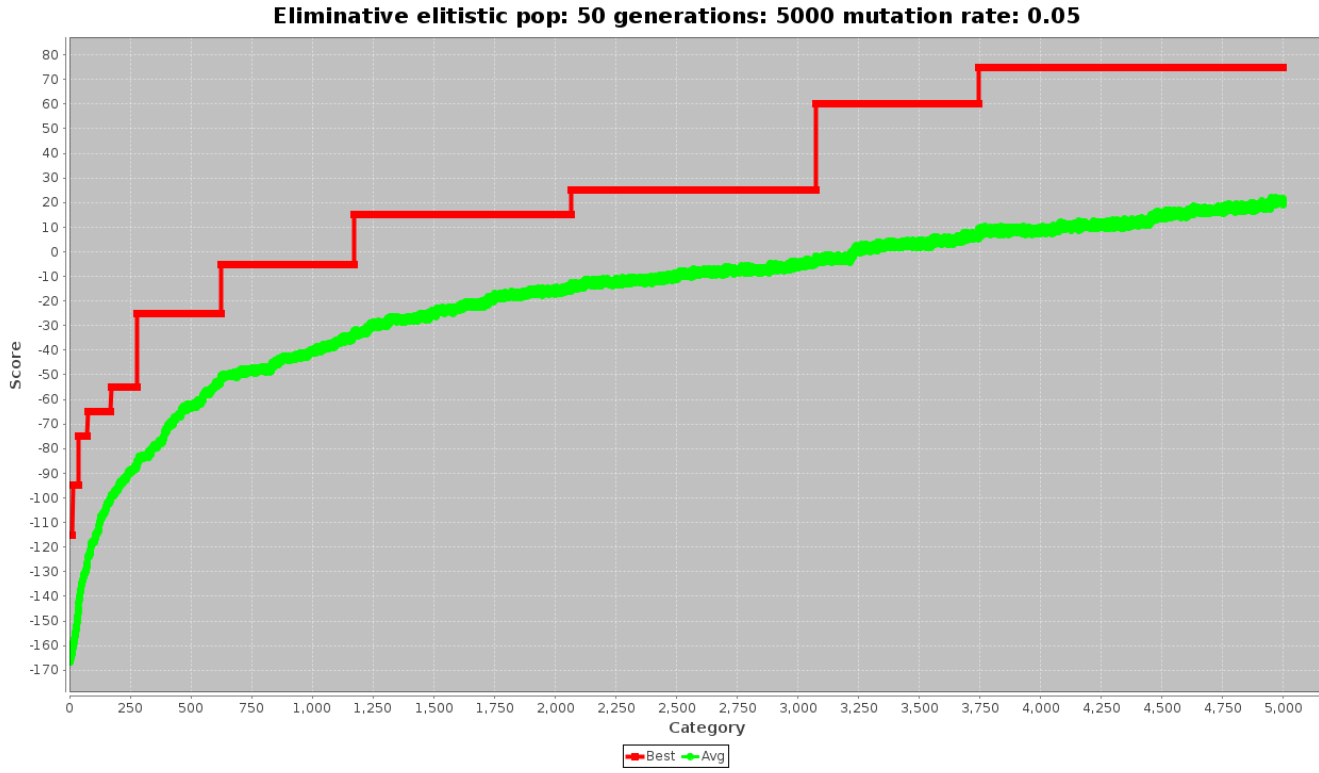
2. Vrlo je teško bilo pratiti da li je najbolja jedinka kvalitetno rješenje ili ne zbog velikog broja mogućnosti s kojim se moglo doći do tog rezultata.

Način vrjednovanja može se pročitati iz tablice 2.6

Iz slike 2.17 vidimo kako prosječna vrijednost funkcije dobrote raste što upućuje na to da model hibridne funkcije kao takav ima smisla. Crvena linija prati vrijednost funkcije dobrote najbolje jedinke i taj je graf rastući zbog ugrađenog svojstva elitizma. Primijetimo da svaki skok (stepenica) na crvenoj liniji znači da je pronađeno nešto što naš model nagrađuje ili kažnjava pa nije kaznio. Vrijednost skoka možemo tumačiti iz tablice 2.6. Tako možemo da je do 250. generacije sustav uspio pronaći 3 točna ulaza i/ili izlaza te 2 takva ulaza i/ili izlaza koji imaju točne tipove dovedene, ali krive signale (npr. doveden je ulaz A, a trebao je biti B). Primijetimo da, iako je to nešto što je krivo i ne odgovara našoj definiciji sustava, to ne bi trebalo jako kažnjavati jer u principu

**Tablica 2.6:** *Hibridna funkcija*

| | |
|---|---|
| Točan ulaz na CLB sklopu | +30 |
| Krivi tip na ulazu CLB sklopa,<br>npr. ništa nije dovedeno ili doveden je izlaz CLB sklopa, a treba nam pin | -20 |
| Krivi ulaz na CLB sklopu, ali dobar tip | -10 |
| Izlaz iz CLB sklopa se odvodi na traženi pin | +30 |
| Izlaz iz CLB sklopa se ne odvodi na traženi pin | -10 |
| Izlaz iz CLB sklopa se odvodi na traženi pin i na još neke | -5 |

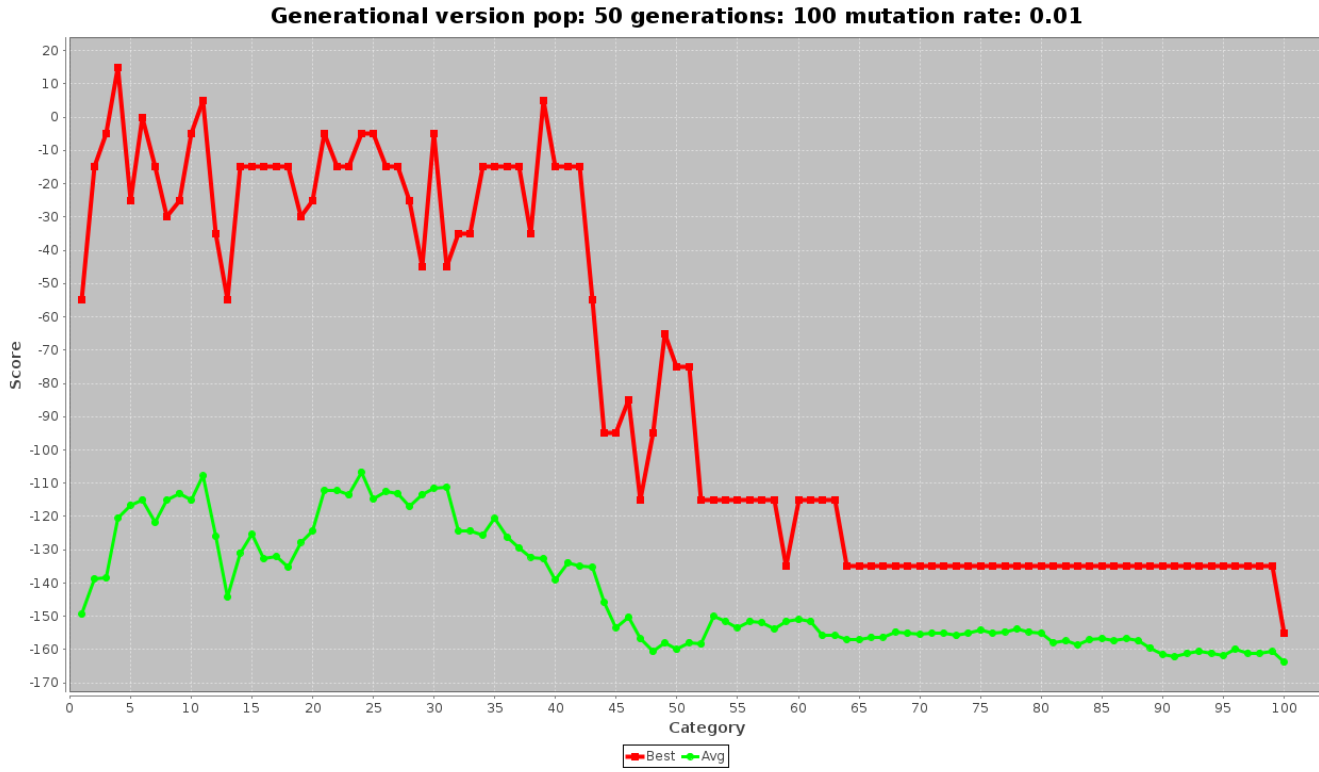



**Slika 2.17:** Primjer funkcije dobrote za algoritam SV66

mi želimo iskoristiti dio genetskog materijala te jedinke, a samo moramo promijeniti vrijednost te ulazne varijable. No, to možemo i samo premetanjem u *pinIndexes* pa bi takvu jedinku genetski algoritam trebao puno lakše pronaći, nego mjenjati totalno konfiguraciju u prospojnoj kutiji.

Pogledajmo kako se ponaša algoritam SV31.

Odmah se vidi na prvu da algoritam nije dobar. Prvu stvar koju primjećujemo je skokovitost crvene linije koja nam prati najbolju jedinku kroz generacije i koja pri kraju rada algoritma doseže pad. No, s obzirom da u ovaj algoritam nije ugrađen elitizam, skokovitost nas ne čudi. Ono što stvarno pokazuje kako konvergencija algoritma




**Slika 2.18:** Primjer funkcije dobrote za algoritam SV31

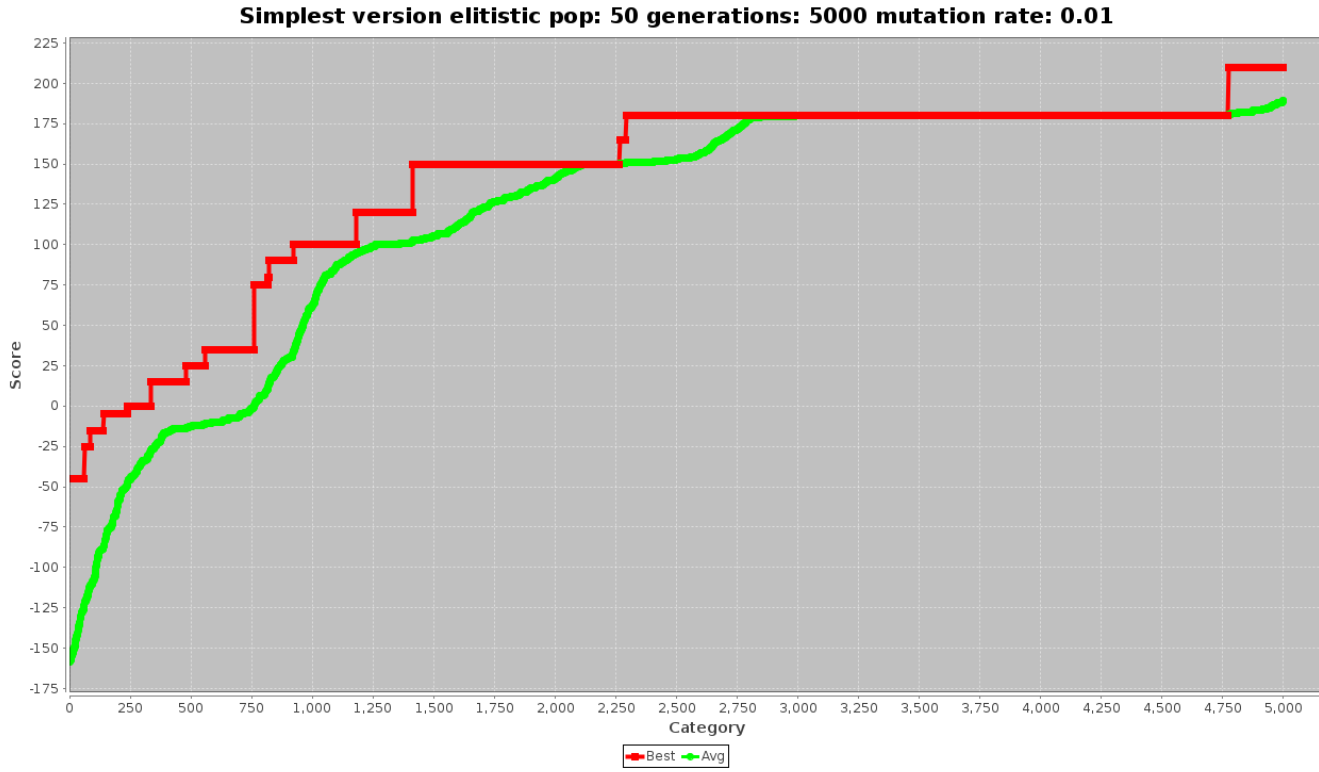
zapravo i ne postoji je zelena linija koja prati prosječnu vrijednost dobrote kroz generacije. Primjećujemo da postoje intervali generacija na kojem ona raste, ali na globalnom intervalu ona pada što upućuje na to da genetski algoritam zapravo nema pojma radi li nešto dobro ili loše.

S obzirom da ne postoji nijedno istraživanje koje bi nam dalo za pravo zaključiti da je generacijski algoritam lošiji od eliminacijskog, zaključujemo da je sustav u kojem ne postoji mutacija loš te ga se nije koristilo u daljnjem radu.

**Napomena:** Svi slučajevi koji se opisuju ovdje dobiveni su pokretanjem algoritama više stotina puta, a ovdje su uzeti primjeri koji samo najbolje oslikavaju ponašanje konkretnog algoritma.

Algoritam koji je najviše obećavao na početku bio je zapravo i najjednostavniji: SV16.

Iako je po samom principu rada vrlo sličan algoritmu SV66, pri gotovo svakom pokretanju algoritma, možemo uvidjeti kako postoji puno više manjih skokova kod najbolje dobrote jedinke, a prosječna dobrota raste puno brže, nego kod SV66. Zanimljivo, je da SV16 ne koristi nikakvu posebnu selekciju već, kao što je navedeno u 2.5.4, nasumično odabere tri jedinke, križa dvije od tri te stavi na mjesto treće ako ima veću dobrotu.




**Slika 2.19:** Primjer funkcije dobrote za algoritam SV16

### Neuspješna nadogradnja hibridne funkcije dobrote

Prije nego što zagazimo u kompliciraniji model kreiranja funkcije dobrote spomenimo još jedan korišteni pristup. Za vrijeme rada algoritma primijećeno je kako algoritam u puno slučajeva uspije na ispravan način dovesti izlaze CLB sklopa na pinove, ali nikada nije u stanju više od 5 ulaza ispravno posložiti. Pokušao sam kreirati takvu funkciju dobrote u kojoj svaki pronađeni ulaz donosi puno više, nego što krivi odnosi. Nagrada se tada za svaki točan ulaz povećavala s kvadratnim članom, a svaki krivi ulaz smanjivala s linearnim članom. Formula za n točnih i m krivih ulaza izgleda ovako:

$$value = \sum_{i=1}^{n} i^2 - 5 * \sum_{k=1}^{m} k \tag{2.8}$$

Sve moguće kombinacije točnih/krivih ulaza zapisani su u tablici 2.7:

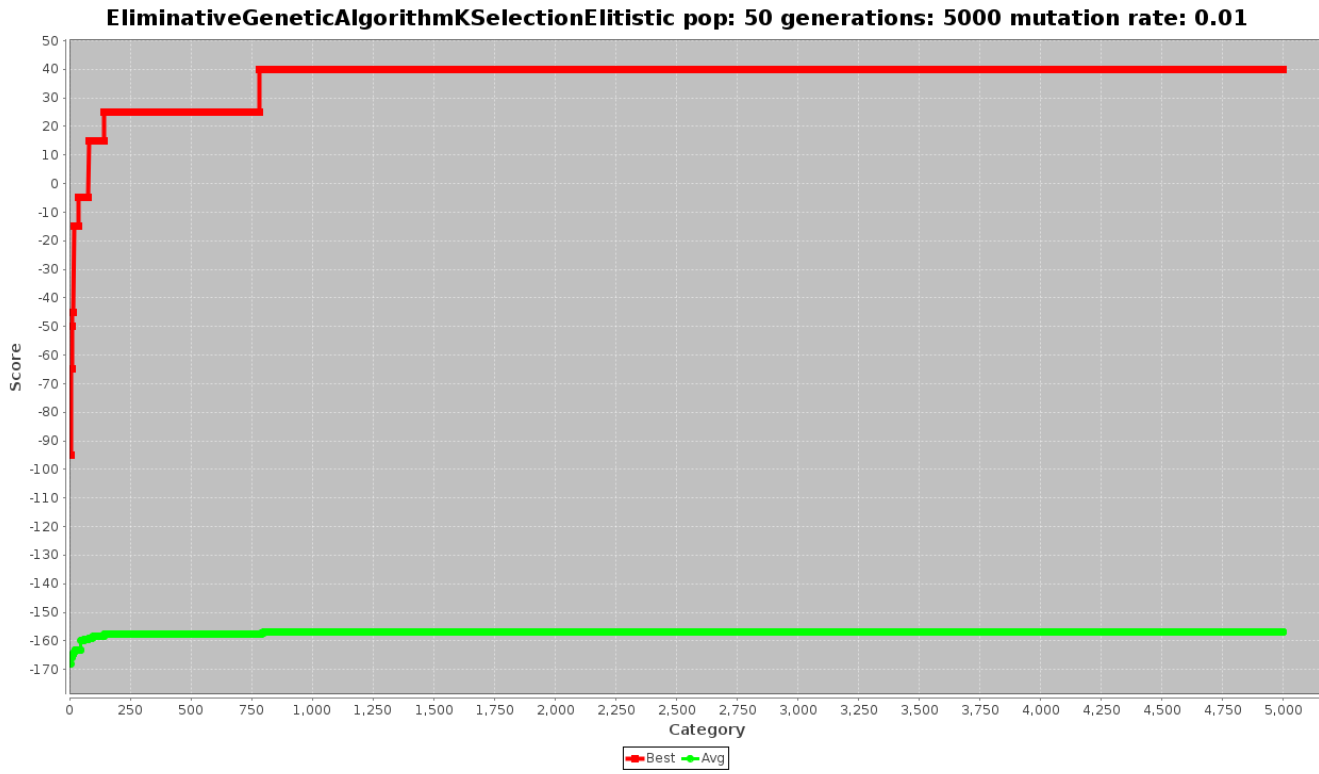
Funkciju dobrote za izlaze sam modelirao na isti način kao za ulaze samo što sam njihove vrijednosti skalirao s parametrom 0.5. S time sam želio postići da se jedinke s dobrim ulazima više koristiti u reprodukciji novih, nego one s dobro postavljenim izlazima. Nažalost, ni ovo nije dovelo do pronalaska nikakvih potpunih rješenja.

Jedan tipičan primjer kako je izgledala funkcija dobrote dok se koristio ovakav

**Tablica 2.7:** *Neuspješna nadogradnja hibridne funkcije dobrote*

| Točnih | Krivih | Vrijednost |
|---|---|---|
| 8 | 0 | 204 |
| 7 | 1 | 135 |
| 6 | 2 | 76 |
| 5 | 3 | 25 |
| 4 | 4 | -20 |
| 3 | 5 | 61 |
| 2 | 6 | -100 |
| 1 | 7 | -139 |
| 0 | 8 | -175 |

model(samo s drugim faktorima) prikazan je na slici 2.20.




**Slika 2.20:** Primjer preuranjene konvergencije

Osim što primjećujemo da genetski algoritam malo toga pronalazi (ima malo skokova), zanimljivo je da sve novo algoritam pronađe do 1000. generacije. Takvu pojavu nazivamo preuranjena konvergencija, a radi se o tome da algoritam zapne u lokalnom optimumu i zbog premalo mutacije, odnosno u ovom slučaju zbog ne baš najsretnije

funkcije dobrote, ne uspije pretražiti ostatak prostora rješenja.

### Smanjenje prostora pretraživanja stanja

Kako bismo olakšali posao genetskom algoritmu, sljedeći je korak bio smanjiti prostor pretraživanja. Pažljivi je čitatelj mogao primijetiti kako su se kod rubnih prospojnih kutija mogle naći veze koje su spajale dvije nepostojeće žice ili jednu nepostojeću s jednom postojećom žicom. Iz tog smo razloga nadogradili ***AIFPGAConfiguration-Randomizer*** te je od onda stvarao samo smislene veze u prospojnoj kutiji. Također su, mutacija i križanje doživjeli promjene te sada i oni prilikom reprodukcije djece, odnosno promjene genetskog materijala, stvaraju validne gene.

```
65          private void randomSwitchBoxConfiguration(byte[][][] swBoxes) {
66                  int maxConnections = Coefficients.getMaxConnections(model.clbs[0].inConnectionIndexes.length);
67
68                  int conns = nextIntBetween(1, maxConnections); // 1 to 5 for example
69
70                  for (int i = 0; i < swBoxes.length; i++) {
71
72                          int row = i / (model.columns + 1);
73                          int col = i % (model.columns + 1);
74                          int connections = 0;
75
76                          while (connections < conns) {
77                                  int j, k, group_j, group_k;
78
79                                  j = random.nextInt(model.wiresCount * 4); // get first index
80
81                                  if(row == 0 && j < model.wiresCount) continue;
82                                  if(row == model.rows && (j >= 2 * model.wiresCount && j < 3 * model.wiresCount)) continue;
83                                  if(col == 0 && (j >= 3 * model.wiresCount && j < 4 * model.wiresCount)) continue;
84                                  if(col == model.columns && (j >= model.wiresCount && j < 2 * model.wiresCount)) continue;
85
86                                  group_j = j / model.wiresCount; //get first group
87
88                                  while (true) {
89                                          k = random.nextInt(model.wiresCount * 4); // get second index
90                                          group_k = k / model.wiresCount;
91                                          if (group_k == group_j) continue;
92                                          //dont even think about creating stupid connections
93                                          if(row == 0 && k < model.wiresCount) continue;
94                                          if(row == model.rows && (k >= 2 * model.wiresCount && k < 3 * model.wiresCount)) continue;
95                                          if(col == 0 && (k >= 3 * model.wiresCount && k < 4 * model.wiresCount)) continue;
96                                          if(col == model.columns && (k >= model.wiresCount && k < 2 * model.wiresCount)) continue;
97                                          break;
98                                  }
99
100                                 int rnd = random.nextInt(3); // 0 1 or 2 for switch box
101                                 if (rnd == 1) {
102                                         swBoxes[i][j][k] = 1;
103                                         swBoxes[i][k][j] = 2;
104                                 } else { //rnd == 2
105                                         swBoxes[i][j][k] = 2;
106                                         swBoxes[i][k][j] = 1;
107                                 }
108                                 connections++;
109                         }
```

**Slika 2.21:** Unaprijeđena verzija stvaranja konfiguracija prospojnih kutija

Sustav je kroz cijelo vrijeme bio unaprijeđivan na temelju uviđenih stvari koje nisu nužno bile krive, ali su vrlo brzo dovodile do jedinki koje nisu rješenja. Prvo što sam odlučio dodati u model jest informacija ima li i koliko ima gaženja signala u modelu. Ako pogledamo 1.1 vidimo da je moguće da se više ulaza, više izlaza ili pak više ulaza i izlaza nađu na nekom CLB ulazu. Takva situacija nam je vrlo nepovoljna iz jednostavnog razloga što onda taj model ne može biti nikako ispravan i teško možemo bilo šta iskoristiti iz te jedinke za stvaranje nove. Odnosno preciznije je reći da bismo možda mogli i iskoristiti nešto, ali zbog nedeterminizma u mutaciji i križanju ne želimo riskirati i uzeti lošu stvar iz te jedinke u izgradnji nove. Upravo zato se taj slučaj kažnjava s vrlo velikim faktorom jer želimo da relativna proporcionalna selekcija s vrlo malom vjerojatnošću uzme takvu jedinku pri izgradnji djeteta.
Osim toga, počeli smo razlikovati situacije u kojima na ulaze/pinove nije ništa dovedeno u odnosu na prijašnje situacije u kojima je jedini bio mogući krivi tip odnosno krivi signal. I ovakvo stanje želimo kazniti s velikim faktorom jer moramo mijenjati konfiguraciju prospojne kutije kod djeteta što je puno teži zadatak, a ne samo ulaze/izlaze.

Kroz cijeli proces izgradnje funkcije dobrote fokus je bio na prospojnim kutijama. Naime, ako su one previše ili premalo toga spojile onda nije bilo šanse da druge strukture podataka "izvuču" algoritam iz zapinjanja. Stoga sam svaku prospojnu kutiju u kojoj su se nalazile više od tri veze počeli kažnjavati jer smo uvidjeli da previše veza znači najčešće i gaženje signala. To sam međutim kažnjavao s malim faktorom jer iako ima previše veza, moguće je da su neke od njih korisne pa takve jedinke ne želimo u potpunosti eliminirati kao one kod kojih je već primijećeno gaženje signala.

Još jedna stvar kod prospojnih kutija nije bila poželjna, a često je bila primjećena: kad god je jedna žica proslijedila signal na više od jedne žice u neki snop to je obično u startu značilo da ta jedinka ne sadržava kvalitetan genetski materijal. Objašnjenje možemo potražiti u sliki SLIKA ISPOD.

Stvar je u tome da na niti jedan CLB sklop nikad nećemo dovoditi više istih ulaza i kod jedinke kod koje je takav slučaj uočen, nju želimo kazniti.

Naravno, kao što je i problem ako imamo previše veza u nekoj prospojnoj kutiji, ako nemamo nijednu isto nije dobro pa svaki put kada je prospojna kutija prazna dobroti jedinke dodamo kaznu. Takvo kažnjavanje nije zato jer nju ne želimo koristiti, moguće je da ona ima dobrih veza u drugim prospojnim kutijama koje nam trebaju, ali će ipak u većini slučajeva prazna prospojna kutija biti znak loše kvalitete rješenja.

Dodavanjem ovih parametara u dobrotu funkcije došli smo do stanja u kojem prospojne kutije više manje imaju uvijek optimalan broj veza. Sljedeća situacija za koju

smo uvidjeli da nije dobra i da bi je trebalo popraviti je ona u kojoj se neki ulazni pin ne proslijeđuje ni na jednu od dvije susjednih prospojnih kutija. U tom slučaju jedinka zapravo zaboravlja na taj ulaz pa će sigurno gdje god bi se on trebao koristiti, tamo naći praznina ili nešto drugo.

### 2.5.7. Križanja i mutacije

#### Križanje aritmetičkom sredinom

Tijekom rada algoritama primijećeno je kako križanje u kojem se za svaki gen računa aritmetička sredina ne daje dobre rezultate. Ako malo razmislimo o tome što koji gen jedinke predstavlja onda shvaćamo zašto je to tako. Uzmimo za primjer polje *clbInIndexes*. To polje pamti pozicije žica s kojih se prima signal kod CLB sklopa. Pretpostavimo da su konfiguracije polja *clbInIndexes* dvije jedinke ovakve:

**Tablica 2.8:** *Konfiguracija* clbInIndexes *prve jedinke. Pretpostovljamo da postoji osam dvoulaznih CLB sklopova.*

| | |
|---|---|
| 0 | 1 |
| 2 | 1 |
| 2 | 1 |
| 2 | 2 |
| 0 | 0 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |

**Tablica 2.9:** *Konfiguracija* clbInIndexes *druge jedinke. Pretpostovljamo da postoji osam dvoulaznih CLB sklopova i da se snop sastoji od 3 žice.*

| | |
|---|---|
| 2 | 1 |
| 1 | 2 |
| 0 | 2 |
| 1 | 1 |
| 0 | 1 |
| 2 | 0 |
| 1 | 1 |
| 1 | 2 |

Nakon križanja, konfiguracija jedinke djeteta izgledati će ovako:

**Tablica 2.10:** *Konfiguracija* clbInIndexes *djeteta. Pretpostovljamo da postoji osam dvoulaznih CLB sklopova i da se snop sastoji od 3 žice.*

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| 0 | 0 |
| 1 | 1 |
| 1 | 1 |
| 1 | 2 |

Prva stvar koja upada u oči jest da imamo jako puno parova samih jedinica. Prokomentirajmo zašto je to tako.
Naime, u ovakvom FPGA modelu postoji 9 različitih parova roditelja pozicija i oni su redom: 00, 01, 02, 10, 11, 12, 20, 21 i 22. U 55.56% slučajeva rezultat aritmetičke sredine roditeljskog para biti će upravo 1, u 33.33% slučajeva rezultat će biti 0 dok je vjerojatnost da se očuva gen koji koristi pozicija 2 za dobivanje ulaznog signala jednaka 11.11%.
Osim matematike, razmišljanjem o problemu uviđamo kako genetskom algoritmu informacija da iz roditeljskog para pozicija npr. 02 nastaje gen 1 ne može puno pomoći.

### Završno križanje

Nakon dugog eksperimentiranja uočene su neke stvari koje bi mogle pomoći u stvaranju kvalitetne jedinke te su implementirane kroz križanje i mutaciju. Križanje *clbIndexes* i *pinIndexes* se obavlja na način da se sa 50% vjerojatnošću odabere jedinka od koje će se preuzeti gen, ali se on uzima samo ako ga je moguće uzeti odnosno ako taj gen već ne postoji. Inače se uzima gen druge jedinke.

```
134         private int[] crossClbAndPin(int[] arr1, int[] arr2) {
135                 Set<Integer> unused = new HashSet<>(); // preprocess
136                 for (int i = 0; i < arr1.length; i++) {
137                         unused.add(arr1[i]);
138                 }
139                 int[] solution = new int[arr1.length];
140 
141                 for (int i = 0; i < arr1.length; i++) {
142                         int rnd = random.nextInt(2); // we need 0 or 1 for choose array
143                         if (rnd == 0) {
144                                 if (unused.contains(arr1[i])) {
145                                         solution[i] = arr1[i];
146                                 } else if (unused.contains(arr2[i])) {
147                                         solution[i] = arr2[i];
148                                 } else {
149                                         solution[i] = (Integer) unused.toArray()[0];
150                                 }
151                         } else if (rnd == 1) {
152                                 if (unused.contains(arr2[i])) {
153                                         solution[i] = arr2[i];
154                                 } else if (unused.contains(arr1[i])) {
155                                         solution[i] = arr1[i];
156                                 } else {
157                                         solution[i] = (Integer) unused.toArray()[0];
158                                 }
159                         }
160                         unused.remove(solution[i]);
161                 }
162 
163                 if (unused.size() > 0) {
164                         throw new IllegalStateException("Invalid configuration in valid crossover!");
165                 }
166                 unused.clear();
167                 for (int i = 0; i < solution.length; i++) {
168                         if (!unused.add(solution[i]))
169                                 throw new IllegalStateException("Duplicates in crossovering!");
170                 }
171 
172                 return solution;
173 
174         }
175 
```

**Slika 2.22:** Križanje clb i pin pozicija.

Pozicije žica s kojima se uzima signal kod CLB sklopa i pinova, odnosno pozicija na koju se šalje križaju se običnim uniformnim križanjem.

```
123         private void crossRandomlyBytes(byte[] result, byte[] arr1, byte[] arr2) {
124                 for (int i = 0; i < result.length; i++) {
125                         int index = random.nextInt(2);
126                         if (index == 0) {
127                                 result[i] = arr1[i];
128                         } else {
129                                 result[i] = arr2[i];
130                         }
131                 }
132         }
133
```

**Slika 2.23:** Uniformno križanje pozicija veza.

S obzirom da su prospojne kutije najkompliciraniji dio našeg sustava, kod njih i imamo najsloženije križanje. Prije je spomenuto kako kažnjavamo previše veza u jednoj prospojnoj kutiji te kako početni konfigurator stvara ograničeni broj veza. Kada bismo koristili uniformno križanje i kod prospojnih kutija dogodilo bi se da kroz npr. 10000 generacija genetski materijal veza u prospojnoj kutiji se jednostavno izgubi. Zato se, u konfiguracijama u kojima algoritam bira između veze u jednoj jedinki i nepostojeće veze u drugoj, sa 70% vjerojatnošću bira veza. Ako obje konfiguracije sadrže veze onda koristimo obično uniformno križanje gena.

```
64          private void crossSwitchBoxes(byte[][][] swBoxes, byte[][][] conf1, byte[][][] conf2) {
65                  for (int i = 0; i < swBoxes.length; i++) {
66                          for (int j = 0; j < swBoxes[i].length; j++) {
67                                  for (int k = 0; k < swBoxes[i][j].length; k++) {
68                                          double p = random.nextDouble();
69
70                                          if (conf1[i][j][k] == 0) {
71                                                  if (p <= 0.5) { // choose with 70% the other one
72                                                          swBoxes[i][j][k] = conf2[i][j][k];
73                                                          swBoxes[i][k][j] = conf2[i][k][j];
74                                                  }
75                                          } else if (conf2[i][j][k] == 0) {
76                                                  if (p <= 0.5) { // choose with 70% the other one
77                                                          swBoxes[i][j][k] = conf1[i][j][k];
78                                                          swBoxes[i][k][j] = conf1[i][k][j];
79                                                  }
80                                          } else {
81                                                  if (p <= 0.5) {
82                                                          swBoxes[i][j][k] = conf1[i][j][k];
83                                                          swBoxes[i][k][j] = conf1[i][k][j];
84                                                  } else {
85                                                          swBoxes[i][j][k] = conf2[i][j][k];
86                                                          swBoxes[i][k][j] = conf2[i][k][j];
87                                                  }
88                                          }
89                                  }
90                          }
91                  }
92          }
93
```

**Slika 2.24:** Križanje prospojnih kutija.

### Završna mutacija

Mutacija je genetski operator kojemu je posvećeno najviše pažnje. Izvedena je na način da se pozicije s kojih se prima ulazni signal zarotiraju ulijevo pa tako ako jedinka ima pozicije 01, nakon mutacije će imati 10. S time se želi isprobati još jedna konfiguracija prije promjene genetskog materijala kod djeteta jer možda je konfiguracija prospojne kutije takva da ima dobre veze, ali su samo krivi signali.

```
62          /**
63           * Circular shift of all positions
64           *
65           * @param arr
66           */
67          private void mutateClbInputs(byte[] arr) {
68                  byte tmp = arr[0];
69                  for (int i = 0; i < arr.length - 1; i++) {
70                          arr[i] = arr[i + 1];
71                  }
72                  arr[arr.length - 1] = tmp;
73          }
74
```

**Slika 2.25:** Mutiranje ulaznih signala.

Pozicije CLB sklopova i ulazno/izlazne veze mutiraju na način da se kod njih primijeni postupak kao u poglavlju 2.3.2.

```
202         /**
203          * Simply swaps two elements
204          *
205          * @param arr
206          */
207         private void swapMutate(int[] arr) {
208                 if (arr.length == 1)
209                         return;
210                 int index1 = random.nextInt(arr.length);
211                 int index2;
212                 while ((index2 = random.nextInt(arr.length)) == index1)
213                         ; // don't allow same indices
214
215                 int temp = arr[index1];
216                 arr[index1] = arr[index2];
217                 arr[index2] = temp;
218         }
```

**Slika 2.26:** Vjerojatnosna mutacija pozicija.

Sljedeći pseudokod objašnjava postupak izmjene gena prospojnih kutija:

```
VezeZaPromijeniti=random(1, dohvatiMaxBrojVeza());

dokle trebaJošMijenjati {
  Odaberi jednu vezu i sa 75%-tnom vjerojatnošću odaberi da ćemo je izbrisati i stvoriti drugu ili sa 25% promijeni joj samo kraj veze.
  Ako je odabrano 25% onda:
    napravi_novi_kraj_veze
    Sa 50%-tnom vjerojatnošću izbrisi staru vezu.
}
```

**Slika 2.27:** Promjena genetskog materijala prospojnih kutija.

# 3. Eksperimentalni rezultati

## 3.1. Opis

Rad cijelog sustava opisivati ću kroz nekoliko primjera, a osim same implementacije sustava dodatno su implementirani sustavi koji omogućuju prikaz kako su se funkcija dobrote i selekcijski pritisak kretali tijekom rada evolucijskog algoritma. Zbog toga što je selekcijski pritisak kasnije dodat on neće biti komentiran u svakoj situaciji. Također, iako je implementirano 70 inačica algoritma, zbog sličnosti rada većine od njih princip rada i rezultati biti će komentirani samo za neke od njih.

```
40                for (int i = 0; i < generations; i++) {
41  //                    logger.log("Generation no " + i + "\n");
42                        int i1, i2, i3, index;
43                        i1 = randomizer.nextInt(populationSize);
44                        while ((i2 = randomizer.nextInt(populationSize)) == i1);
45                        do {
46                                i3 = randomizer.nextInt(populationSize);
47                        } while (i3 == i1 || i3 == i2);
48  //                    logger.log(i1 + " " + i2 + " " + i3 + "\n");
49                        index = ConfUtil.getWorstFromThree(fitnesses, i1, i2, i3);
50                        AIFPGAConfiguration conf1, conf2, newConf;
51                        if (index == i1) {
52                                conf1 = population[i2];
53                                conf2 = population[i3];
54                        } else if (index == i2) {
55                                conf1 = population[i1];
56                                conf2 = population[i3];
57                        } else {
58                                conf1 = population[i1];
59                                conf2 = population[i2];
60                        }
61                        newConf = crosser.crossover(conf1, conf2);
62                        if(mutator != null) mutator.mutate(newConf);
63                        cleaner.clean(newConf);
64                        FPGAModel model = FPGAEvaluator.EvaluatorArranger.prepareModelForEvaluation(ex, newConf, mapTask,
65                                        sfpga);
66                        //bestOverall = model;
67                        double value = evaluator.evaluate(newConf, model, mapTask);
68                        if(checkEvaluatorEnding(model)) return;
69                        if(value > fitnesses[index]) {
70                                population[index] = newConf;
71                                fitnesses[index] = value; //
72                        }
73                        putAverageFitnessForGen(i+1); // we don't want to start from zero
74                        putBestFitnessForGen(i+1);
75                }
76        }
77
78  }
```

**Slika 3.1:** Dio izvornog koda algoritma SV16.

# 4. Zaključak

Zaključak

# LITERATURA


T. Bäck. *Generalized convergence models for tournament- and (mu,lambda)-selection* , 1995.

T. Bäck. *Evolutionary Algorithms in Theory and Practice*, 1996.

K. Deb. i D. Goldberg. *A comparative analysis of selection schemes used in genetic algorithms. In Foundations of genetic algorithms (FOGA1).* FOGA, 1991.

J. H. Holland. *Adaptation in Natural and Artifcial Systems: An Introductory Analysis with Applications to Biology, Control, and Artifcial Intelligence*, 1975.

B. Miller i D. Goldberg. *Genetic algorithms, tournament selection, and the effects of noise*, 1995.

H. Mühlenbein i D. Schlierkamp-Voosen. *Predictive models for the breeder genetic algorithm, I: Continuous parameter optimization. EvolutionaryComputation*, 1993. URL `http://dx.doi.org/10.1162/evco.1993.1.1.25`.

D. Thierens. *Selection schemes, elitist recombination, and selection intensity*, 1997.

Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi*. FER, 2012. URL `http://java.zemris.fer.hr/nastava/pioa/knjiga-0.1.2013-12-30.pdf`.

Marko Čupić. *Evolucijsko računanje*. FER, 2019. URL `http://java.zemris.fer.hr/nastava/ui/evo/evo-20190604.pdf`.

Marko Čupić, Dalbelo-Bašić Bojana, i Golub Marin. *Neizrazito, evolucijsko i neuroračunarstvo*. FER, 2013. URL `http://java.zemris.fer.hr/nastava/nenr/knjiga-0.1.2013-08-12.pdf`.

**Rješavanje problema smještanja i povezivanja kod sklopa FPGA**

**Sažetak**

Sažetak na hrvatskom jeziku.

**Ključne riječi:** Ključne riječi, odvojene zarezima.

**Solving placement and routing problems in FPGA**

**Abstract**

Abstract.

**Keywords:** Keywords.