SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 216

AUTOMATSKO OBLIKOVANJE KOMBINACIJSKIH MREŽA

Iva Brajer

Zagreb, lipanj 2011.

Zahvala mentoru i GP guruu, doc.dr.sc. Domagoju Jakoboviću, na prenesenom iskustvu, ukazanoj pomoći i ponajviše strpljenju čitavih tri godine mentorstva zbog predugih tekstova koje je trebalo pročitati (uključivo i ovaj) i ponekad suviše bizarnih ideja „inženjerskog pristupa problemu“ s moje strane.

Sadržaj

Uvod

Ljudi većinu vremena nisu niti svjesni koliko su zapravo okruženi sustavima koji se sastoje od neke vrste elektroničkih mreža. Budilice koje nam zvone svako jutro, mikrovalne pećnice na kojima si namještamo trajanje podgrijavanja jutarnje kave, daljinski upravljači s kojima provjeravamo vremensku prognozu na teletekstu, ključevi automobila s kojima si na određeni pritisak automatski otključavamo automobile pri polasku na posao. Cijeli život smo okruženi takvim sustavima, a da smo još manje svjesni koliko je truda, vremena i novaca bilo potrebno uložiti da se jedan takav sustav razvije kako bi bio funkcionalan za korištenje.

Proces razvijanja jednog sustava nije tako jednostavan. Pogotovo ako se uzme u obzir napredak kojeg su elektronički sustavi doživjeli kroz povijest. I sami smo svjedoci sve složenijim i funkcionalnijim sustavima koji se prodaju za sve manje novaca, a lansiraju na tržište u sve kraćem vremenu. Odgovor na pitanje kako je bilo moguće uz naglo povećanje složenosti takvih sustava izraditi ih u najkraćem mogućem roku i ponuditi ih potrošačima za manje novaca je automatizacijom samog procesa oblikovanja takvih sustava. Upravo će se u ovom radu dotaknuti problem oblikovanja mreža te proces automatizacije njihovog oblikovanja.

U prvom poglavlju predstavljen je opis problema oblikovanja kombinacijskih mreža koji je tema ovog znanstvenog rada. Bit će nešto riječi o primjerima rješavanja sličnih problema pronađenim u literaturi te uvod u kartezijsko genetsko programiranje koje će se koristiti za rješavanje problema. U drugom poglavlju je opisana implementacija rješenja kojoj je zadatak automatizirano pronalaziti strukture kombinacijskih mreža bez potrebe da se ljudski ekspert uključi u proces oblikovanja. U trećem poglavlju je opisana analiza tog rješenja nad ispitana četiri ispitna primjera oblikovanja kombinacijskih mreža te izvedeni zaključci. U četvrtom poglavlju su dane upute za uporabu implementacijskog dijela rješenja.

1. Opis problema

1.1. Mreže oblikovane evolucijskim metodama

Proizvodnja elektroničkih mreža je važna grana industrije koja neprestano napreduje u pogledu zahtjeva za sve složenijim ponašanjima mreža i stalnim poboljšanjima osnovnih komponenata. Tako Gordon i Bentley u [1] potvrđuju kako tradicionalne metode koje su se koristile kod oblikovanja elektroničkih mreža postaju financijski neodržive i vremenski zahtjevne. Takve metode su podrazumijevale zapošljavanje ljudskih eksperata koji su samostalno oblikovali mreže, međutim kako je složenost mreža rasla, rasla je i potreba za brojem eksperata i produljivali se vremenski rokovi, što je postalo financijski i vremenski neisplativo. Pokušavalo se i uvesti apstrakciju u proizvodnju elektroničkih mreža kao što su formalni jezici za opis sklopovlja, ali takve apstraktne metode su skrivale ono što je bilo bitno u proizvodnji, a to je unutarnja struktura mreža.

Miller, Thomson i Fogarty u [2] navode kako je oblikovanje elektroničkih mreža zapravo kompleksan zadatak jer podrazumijeva specifikaciju mreže pretvoriti u oblik prikladan za ciljnu tehnologiju (npr. odabir logičkih vrata), minimizaciju dobivene reprezentacije, optimizaciju s obzirom na definirana ograničenja (npr. vremenske karakteristike, ispravnost ulaza i izlaza) te preslikavanje dobivenog na ciljni uređaj.

Pojavila se potreba za drugačijim pristupom u industrijskom procesu proizvodnje elektroničkih mreža, a jedan takav pristup predstavljaju mreže oblikovane evolucijskim metodama (eng. *Evolvable Hardware*) kao što ih opisuju Gordon i Bentley u [1] te Torresen u [3]. Mreže oblikovane evolucijskim metodama predstavljaju koncept automatskog oblikovanja mreža inspiriran procesima iz prirode kao što je proces evolucije.

Evolucijske metode se temelje na slučajnosti koja prevladava u procesima iz prirode, a ne na egzaktnim matematičkim algoritmima što znači da takve metode ne garantiraju pronalazak maksimalno optimalnog rješenja nego optimalnog s obzirom na neke

postavljene kriterije (npr. ispravnost rada elektroničke mreže za sve ulazno-izlazne kombinacije vrijednosti). Obzirom da nije riječ o egzaktnim metodama, rješenja koja se dobivaju ne moraju uvijek biti ista što može biti prednost jer se iz skupa optimalno dobivenih rješenja može odabrati ono najprikladnije. S druge strane, to može biti i mana takvih metoda zbog nemogućnosti provjere postoji li bolje rješenje od pronađenog. Temelj tih metoda je na načinu pretraživanja prostora rješenja. Uvođenjem pristranosti u slučajni evolucijski proces dogodit će se preferiranje boljih rješenja (u nekom trenutku) po određenim kriterijima jer se rješenja tokom evolucijskog procesa vrednuju (na temelju postavljenih kriterija). Na taj način će evolucijske metode nastojati zanemariti prostor rješenja gdje se nalaze lošija rješenja.

Gordon i Bentley u [1] još navode kako mreže oblikovane evolucijskim metodama uzimaju u obzir definiciju mreže kao crne kutije (eng. *black box*) za koju je bitno opisati što radi, a ne kako radi. Drugim riječima, unutrašnja struktura je skrivena i opisano je samo njezino ponašanje, a komunikacija s crnom kutijom se izvodi na temelju očitanja glavnih ulaza i izlaza iz crne kutije. Unutrašnja struktura predstavlja izgled elektroničke mreže kojeg treba oblikovati na temelju komunikacije s ulazima i izlazima. Sav posao oblikovanja obavlja evolucijski proces, a verifikacija ispravnosti rada dobivene elektroničke mreže se provodi očitavanjem parova ulaza i izlaza. Ukoliko se za svaku kombinaciju ulaza na izlazu iz crne kutije dobivaju iste izlazne vrijednosti kao one tražene (iz same specifikacije ponašanja crne kutije) tada crna kutija radi korektno tj. uspješno simulira traženu elektroničku mrežu.

Čak i ako je jedini uvjet prilikom oblikovanja elektroničke mreže da ona radi korektno, ponekad je prostor rješenja kojeg treba pretražiti neka evolucijska metoda prevelik, s čime se slažu oba izvora (Gordon i Bentley u [1] te Torresen u [3]). U prostoru rješenja svako pojedino rješenje predstavlja jednu varijantu izgleda elektroničke mreže (raspored unutarnjih elemenata tj. način na koji su spojeni ulazi i izlazi). Veličina prostora rješenja eksponencijalno ovisi o broju ulaza. Za svaki novi ulaz (pod pretpostavkom da na pojedini ulaz dolazi jedan signal ili u digitalnoj elektronici jedan bit) broj ulazno-izlaznih

2 2

kombinacija se povećava s na , gdje n označava broj ulaza prije dodavanja novog ulaza. Ukoliko je ulaza previše, nastaje prava kombinatorna eksplozija te je prostor pretraživanja svih mogućih rješenja toliko velik da šanse pronalaska samo korektnog rješenja uvelike opadaju.

Uz to, oba izvora navode kako je problemu pretraživanja prevelikog prostora rješenja moguće doskočiti uvođenjem apstraktnih struktura u reprezentaciju rješenja što može smanjiti prostor pretraživanja (npr. umjesto osnovnih logičkih vrata uvesti multipleksore). Međutim, tada se postavlja pitanje kakve apstraktne strukture odabrati.

Gordon i Bentley u [1] predlažu uvesti varijabilnu veličinu reprezentacije rješenja gdje se veličina rješenja dinamički mijenja tokom evolucije prilikom čega se pojedini dijelovi odbacuju ili dodaju. To naravno podrazumijeva i činjenicu da se samim time i prostor rješenja dinamički mijenja.

Osim toga, autori navode kao zanimljivost uvođenje neutralnih mreža (eng. *neutral networks*) koje mogu pomoći u lakšem pretraživanju velikog prostora rješenja. Riječ je o posebnom skupu rješenja koje se jednako vrednuju u procesu evolucije, ali u sebi sadržavaju razlike u genetskom materijalu koje su nevidljive tokom vrednovanja. Uočeno je kako upravo te male razlike u genetskom materijalu u rješenjima koje se prilikom vrednovanja pokazuju istima (ali zbog genetske razlike zapravo nisu) mogu pomoći da se izbjegne zaglavljivanje u lokalnim optimumima tokom procesa pretraživanja prostora rješenja.

Mreže oblikovane evolucijskim metodama pokrivaju veliko područje načina oblikovanja različitih elektroničkih mreža, ali se u ovom znanstvenom radu koncentrira samo na digitalne kombinacijske mreže.

1.2. Problem automatskog oblikovanja kombinacijskih mreža

Kombinacijske mreže su vrsta digitalnih mreža u kojima se nad ulaznim signalom provode logičke operacije te izlaz ovisi isključivo o trenutnom ulazu [4]. Drugim riječima, prateći signal od ulaza preko unutarnjih elemenata koji su nad digitalnim signalima provodili logičke operacija do krajnjeg izlaza, moguće je sastaviti logički izraz (ili izraze ako ima više izlaza) koji predstavlja tu kombinacijsku mrežu. To je bitno svojstvo jer je poznato da se svaki logički izraz može optimizirati različitim tehnikama od kojih su možda najpoznatije primjena osnovnih Booleovih teorema i aksioma [4], Karnaughove tablice [4] i Quine-McCluskeyeva metoda [4]. Jednom kada se iz kombinacijske mreže dobiju logički izrazi po pojedinom izlazu, optimizacija djeluje kao „uhodana“ procedura koja će doprinijeti pojednostavljenju kombinacijske mreže.

Problem nastaje kada je izgled kombinacijske mreže nepoznat, odnosno kada je upravo sam izgled mreže ono što se treba oblikovati, a da ta mreža pritom bude optimalna po pitanju broja sklopova i razina (minimizirana). To navodi na problem automatskog oblikovanja kombinacijskih mreža koji je podvrsta problema oblikovanja elektroničkih mreža. Obzirom da je riječ o automatskom oblikovanju, potrebno je u oblikovnom procesu koristiti metode neovisne o ljudskom faktoru.

Metode koje će se koristiti prilikom rješavanja problema automatskog oblikovanja kombinacijskih mreža su evolucijske metode, odnosno kombinacijske mreže će se promatrati kao mreže oblikovane evolucijskim metodama. Evolucijske metode o kojima će biti riječ spadaju u područje genetskih algoritama [5] i genetskog programiranja [6].

Obje metode kroz proces evolucije prolaze na način da se određene jedinke odnosno njihovi genotipovi (koje predstavljaju rješenja nekog problema) iz populacije odabiru (npr. slučajno) za mutaciju i(li) križanje te se čitav proces odvija kroz određeni broj generacija. Tokom procesa evolucije jedinke odumiru te se nove jedinke stvaraju križanjem. Odumiranje jedinki određeno je dobrotom jedinki, odnosno jedinke se kroz proces evolucije vrednuju te one s lošijom dobrotom tokom vremena odumiru (brišu se iz

populacije). S druge strane, jedinke koje imaju bolju dobrotu preživljavaju te križanjem dobra svojstva prenose na svoje potomke. Iz opisanog vidljivo je zašto je rečeno da su evolucijske metode inspirirane prirodnim procesom evolucije koji se odvija na jednak način.

Genetski algoritam koristi stalnu veličinu genotipa[1], a u njegovoj osnovnoj varijanti koristi se binarna reprezentacija genotipa kao što je opisano u [5]. U takvoj reprezentaciji genotip izgleda kao niz nula i jedinica koji poprimaju određeno značenje ovisno o rješenju koje se njima predstavlja (npr. to može biti binarni zapis nekog broja ako je problem koji se rješava traženje ekstrema neke složene funkcije).

Za razliku od genetskog algoritma, genetsko programiranje koristi promjenjivu veličinu genotipa[2] koja se mijenja tokom evolucije. Kao što je opisano u [6], u njegovoj osnovnoj varijanti koristi se reprezentacija genotipa stablom (stablo kao usmjereni graf) pri čemu se genotip prikazuje kao hijerarhijska struktura gdje su svi čvorovi međusobno povezani. Listovi stabla su čvorovi terminali gdje se nalaze varijable ili konstante, a unutarnji čvorovi su funkcijski čvorovi koji provode neku operaciju nad terminalima. Funkcijski čvorovi i čvorovi terminali se zajedničkim nazivom zovu čvorovi primitivi. Stablo se također uređuje na način da vjerno opisuje rješenje problema kojeg rješava (npr. može se stablom prikazati logička funkcija ako se rješava problem automatskog oblikovanja kombinacijskih mreža [7]).

1.3. Primjeri rješavanja sličnih problema

Problem mreža oblikovanih procesom evolucije se spominje kao naročito izazovni te složeni problem i u znanstvenoj literaturi.

Coello Coello, Christiansen i Hernández Aguirre su u [8] predložili genetski algoritam koji rješava problem automatskog oblikovanja kombinacijskih mreža. Predložili su matrični

[1] Stalna veličina genotipa podrazumijeva da svaki genotip u populaciji ima istu veličinu, tj. za binarnu reprezentaciju to znači da svaki genotip ima jednako dugačak niz nula i jedinica.
[2] Promjenjiva veličina genotipa podrazumijeva da neki genotip u populaciji ne mora imati istu veličinu kao preostali genotipovi tj. za reprezentaciju stablom to znači da broj čvorova u pojedinom genotipu može varirati.

prikaz genotipa koji izgleda kao niz nabrojanih ulaza u pojedine ćelije matrice i funkcija koju obavljaju te ćelije. Zanimljivo je kako taj matrični prikaz nalikuje na izgled kartezijskog genotipa, o kojemu će biti riječ kasnije. Također, pokazali su kako njihova matrična reprezentacija genotipa pokazuje bolje performanse nad ispitnim primjerima od standardne binarne reprezentacije koja se koristi u genetskom algoritmu. Binarna reprezentacija davala je lošije rezultate od rješenja koje je predložio ljudski ekspert za razliku od matrične reprezentacije koja je davala zadovoljavajuće rezultate.

Sličnu matričnu reprezentaciju genotipa koristili su Reis i Machado u [9]. Oni su pokazali da je glavni problem u evolucijskim metodama oblikovanja kombinacijskih mreža uz veliki prostor pretraživanja rješenja (koji raste s brojem ulaza) i dugotrajno vrednovanje rješenja (pod uvjetom da su rješenja velika, a jesu ako je veliki broj ulaza u pitanju). Postavili su zahtjev za upotrebu složenih gradivnih blokova za razliku od jednostavnih logičkih vrata jer kao što je već rečeno u prethodnom poglavlju, to može smanjiti prostor pretraživanja. To u neku ruku pokazuje kako postoji zahtjev za prilagodbom reprezentacije rješenja (genotipa) zadanom problemu kao što je automatsko oblikovanje kombinacijskih mreža.

Koza je u [10] koristio genetsko programiranje za oblikovanje kombinacijskih mreža. Oblikovao je jednostavne mreže koristeći mali skup logičkih funkcijskih čvorova (AND, OR, NOT), ali je naglasak njegovog istraživanja bio na izradi funkcionalnih mreža, a ne na njihovoj optimizaciji. Coello Coello, Hernández Aguirre i Buckles su u [11] odlučili odbaciti svoju matričnu reprezentaciju genotipa te također pokušati s genetskim programiranjem pri čemu je osim funkcionalne mreže bilo potrebno dobiti mrežu s najmanjim brojem multipleksora (korišten je kao funkcijski čvor u genotipu). Naglasili su kako osim potpune funkcionalnosti može biti bitna i cijena implementacije koja podrazumijeva ograničenu veličinu tiskane silikonske pločice na koju će biti otisnuta dobivena mreža. Dobiveni rezultati su pokazali kako genetsko programiranje uspijeva uskladiti oba zahtjeva. U prijašnjem znanstvenom radu [7] problem oblikovanja kombinacijskih mreža se također nastojao riješiti genetskim programiranjem. Uz potpunu funkcionalnost traženo je bilo da se zadovolji i ograničenost veličine (broj funkcijskih čvorova treba biti što manji) te smanji vremensko kašnjenje (dubina stabla treba biti što manja). Pokazano je kako je genetsko programiranje uspjelo savladati sva tri postavljena kriterija, ali se oni nisu tokom evolucije

zasebno razmatrali nego su bili objedinjeni u jednu funkciju dobrote (eng. *fitness function*).

Korištenjem genetskog algoritma potrebno je mijenjati reprezentaciju genotipa u nešto što će biti efikasnije od binarne reprezentacije jer dobivena rješenja nisu dovoljno dobra. Potrebno je u genotip uvesti apstraktnije komponente tj. gradivne blokove, kako je navedeno u prethodnom poglavlju, kako bi se povećala učinkovitost pretraživanja smanjivanjem prostora pretraživanja.

Korištenjem genetskog programiranja nedostaci možda nisu toliko uočljivi jer se genotipom lako upravlja zbog njegove varijabilne duljine (navedeno kao prednost u prethodnom poglavlju) što uzrokuje dinamičko mijenjanje prostora pretraživanja rješenja. Međutim, neki nedostaci ovog pristupa su uočeni u navedenom prijašnjem znanstvenom radu i ostaloj literaturi te će biti posebno naglašeni kroz naredna poglavlja.

Za razliku od prijašnjeg znanstvenog rada, u ovom znanstvenom radu implementacija rješenja problema automatskog oblikovanja kombinacijskih mreža će se temeljiti na drugačijoj reprezentaciji genotipa od standardne reprezentacije stablom koja se koristi u genetskom programiranju (i prijašnjem znanstvenom radu). Riječ je o kartezijskom genotipu koji će biti opisan u sljedećem poglavlju te će na temelju drugih izvora biti naznačeno zašto je odabran upravo taj genotip za rješavanje navedenog problema.

1.4. Kartezijsko genetsko programiranje

Miller, Thomson i Fogarty su u [2] objavili studiju o problemu oblikovanja mreža uporabom evolucijskih algoritama. Koncentrirali su se na problem oblikovanja mreža uporabom procesa evolucije te ustvrdili kako se za oblikovanje može poslužiti programibilnim elektroničkim komponentama čiji je glavni predstavnik FPGA [12] (eng. *Field-Programmable Gate Array*). U svojoj studiji koristili su Xilinx XC6216 FPGA komponentu jer je napravljena kao programibilno polje logičkih ćelija s dva ulaza i jednim izlazom koje nemaju zadanu fiksnu operaciju koju obavljaju niti način na koje su međusobno povezane. Sve se to može konfigurirati po potrebi te su takve elektroničke

komponente izrazito prikladne za oblikovanje evolucijskim metodama. Istaknuli su način oblikovanja kombinacijskih mreža za jednu takvu FPGA komponentu s dva aspekta, a to su povezanost ćelija i ostvarena funkcionalnost unutar ćelija. Zaključeno je kako je u oblikovanju kombinacijskih mreža evolucijskim metodama potrebno uvesti takvu reprezentaciju genotipa koja omogućava više načina dolaska do rješenja što se može postići uvođenjem kompleksnijih reprezentacija koja pružaju određenu redundantnost genetskog materijala.

Struktura FPGA komponente poslužila je ujedno i kao inspiracija za novu reprezentaciju genotipa. Pristup reprezentaciji genotipa pod nazivom *kartezijsko genetsko programiranje* (eng. *Cartesian genetic programming*; kraće samo CGP) opisuje genotip kao niz stalne veličine sastavljen od prirodnih brojeva koji se na određen način preslikavaju u usmjerene grafove. Usmjereni grafovi nalikuju na strukturu FPGA komponente gdje su ćelije vrhovi grafa, a žice poveznice lukovi grafa. Miller je predstavio kartezijsko genetsko programiranje u [13] te pokazao kako se ono može učinkovito primijeniti na probleme koji obuhvaćaju učenje logičkih funkcija. Dodatno je razradio ideju zajedno s Thompsonom u [14] te opisao zašto kartezijsko genetsko programiranje pokazuje dobre performanse. Poglavlje koje slijedi opisuje detaljnije kartezijski genotip, a tekst za to poglavlje napisan je kao uvod u znanstveni rad i može ga se u proširenoj inačici pronaći u [15].

1.4.1. Kartezijski genotip

Kartezijsko genetsko programiranje nosi takav naziv iz razloga što se grafička interpretacija njegovog fenotipa može predstaviti kao dvodimenzionalna mreža čvorova preslikana u kartezijski koordinatni sustav (vidi Slika 1).

Genotip je predstavljen nizom prirodnih brojeva i stalne je duljine. Inicijalni ulazi[3] i izlazi iz čvorova se slijedno numeriraju. Funkcije koje se stavljaju u čvorove[4] se također (zasebno) slijedno numeriraju. Čitav genotip je samo niz ulaza u čvorove i funkcija s dodanim izlazima na kraju genotipa. Korisno je zamijetiti kako izlazi iz čvorova nisu dio genotipa niti

[3] Inicijalni ulazi su zapravo čvorovi terminali iz genetskog programiranja.
[4] Funkcije u čvorovima su funkcijski čvorovi iz genetskog programiranja.

je to potrebno jer se slijedno numeriraju po čvorovima (vidi Slika 1). Također, u ovom prikazu genotipa svaki čvor ima isti broj ulaza i jedan izlaz. To je opće prihvaćena konvencija (koja će se ovdje pridržavati), ali ne mora biti tako i čvorovi u genotipu mogu imati varijabilan broj ulaza i izlaza.

Genotip se na određen način preslikava u fenotip koji zapravo predstavlja usmjeren graf. Premda je genotip stalne duljine, fenotip je promjenjive duljine ovisno o broju neizraženih gena[5].

Važnost preslikavanja genotipa u fenotip je činjenica da to omogućava da se više genotipa preslika u isti fenotip i time je eksplicitno zadovoljena neutralnost. Neutralnost (eng. *neutrality*) označava prisutnost različitih genotipova s istom vrijednosti dobrote (eng. *fitness*).

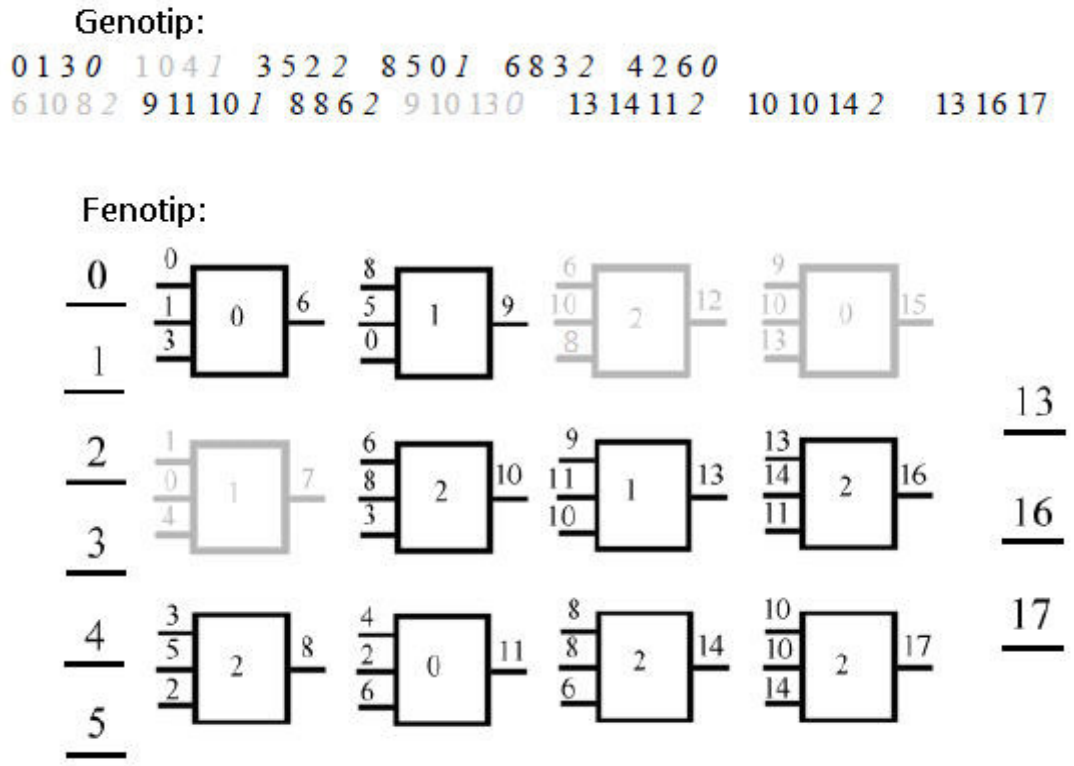
U kartezijskom genetskom programiranju zapravo postoji velik broj genotipova koji se preslikavaju u isti fenotip, a to je moguće zahvaljujući velikoj količini zalihosti (eng. *redundancy*). Postoje tri vrste zalihosti:

1. zalihost čvorova (eng. *node redundancy*) - nastaje zbog čvorova koji nisu dio usmjerenog grafa jer nisu direktno ili indirektno (preko drugih čvorova) spojeni s krajnjim izlazima (sivi blokovi na Slika 1)

2. funkcionalna zalihost (eng. *functional redundancy*) - nastaje zbog određenog broja čvorova koji zajednički tvore neku funkciju koja se zapravo može prikazati s puno manjim brojem čvorova

3. zalihost ulaza (eng. *input redundancy*) - nastaje ako neki od inicijalnih ulaza nisu povezani ni s jednim ulazom u čvorove

[5] Pod neizraženim genima misli se na nespojene izlaze nekih čvorova što označava nepovezan čvor u grafu npr. sivi kvadrati na Slika 1.



Slika 1. Preslikavanje genotipa u fenotip za šest ulaza, tri izlaza i tri funkcije (0, 1, 2 u kvadratima, naznačene kurzivom u genotipu). Sivi kvadrati označavaju nepovezane čvorove.

Zalihost čvorova i zalihost ulaza zajednički tvore potencijal za dodavanje korisne neutralnosti jer nepovezani čvorovi mogu proći kroz niz neutralnih promjena i kasnije u procesu evolucije postati povezani s ostatkom grafa. Time se u konačnici postiže dobivanje veće vrijednosti dobrote.

Funkcionalna zalihost bi se s druge strane trebala izbjegavati jer uzrokuje nepotrebno povećavanje genotipa (a što je veći genotip to je i prostor pretraživanja rješenja veći). Takva vrsta zalihosti podrazumijeva uvođenje apstraktnijih struktura tj. gradivnih blokova u genotip kojima se skup osnovnih operacija zamjenjuje za jednu funkciju (npr. skup logičkih vrata povezan na određen način se zamijeni za multipleksor).

Ako postoji više genotipova koji u populaciji imaju najbolju vrijednost dobrote (jer imaju isti fenotip) tada se za najbolji genotip bira neki od njih slučajnim odabirom. To se podrazumijeva pod pojmom neutralne pretrage (eng. *neutral search*). Neutralna pretraga

ima čvrstu poveznicu s pojmom neutralnih mreža opisanih u prethodnim poglavljima za koje je naznačeno da mogu doprinijeti boljoj pretrazi prostora rješenja.

Kartezijski program (eng. *Cartesian program*; skraćeno *CP*) koji se označava s *P* je definiran kao skup

$$P = \{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$$

gdje je

- G - genotip kao niz prirodnih brojeva gdje je
  - n_i - broj ulaza u program
  - n_n - broj ulaza (ulaznih veza) u čvor (eng. *node input connections*)
  - n_o - broj izlaza iz programa
- F - skup funkcija u čvorovima gdje je
  - n_f - broj mogućih funkcija
- n_r - broj redaka
- n_c - broj stupaca
- l - parametar stupnja povratka (eng. *levels back parameter*) koji određuje koliko će prijašnjih stupaca čvorova moći spojiti svoje izlaze na čvor u trenutnom stupcu[6]

Kartezijsko genetsko programiranje u svojoj najjednostavnijoj inačici razmatra samo povezanost čvorova s jednosmjernim prolazom unaprijed (eng. *feed forward*

[6] Pritom se na primarne ulaze odnosi isto kao i na izlaze čvorova.

connectivity), međutim u njegov genotip se mogu lako uključiti i petlje, ali ti slučajevi se neće razmatrati.

Inicijalni ulazi programa i izlazi iz čvorova se smiju povezati s bilo kojim ulazom u čvorove u idućem stupcu, odnosno stupcima (ovisno o parametru stupnja povratka). Čvorovi u istom stupcu ne smiju biti međusobno spojeni jedni s drugim, ali zato bilo koji čvor može biti ili povezan ili nepovezan.

Primjer kartezijskog programa je na Slika 1. Uzete su vrijednosti:

- $n_i = 6$ - ulazi su označeni redom 0-5
- $n_n = 3$ - npr. za čvor koji se nalazi u prvom retku i prvom stupcu ulazne veze su 0, 1 i 3
- $n_o = 3$ - izlazi su redom 13, 16 i 17
- $n_f = 3$ - funkcije su redom numerirane kao 0, 1 i 2, dok u stvarnosti to mogu biti zbrajanje, oduzimanje i množenje
- $n_r = 3$ - tri su retka čvorova
- $n_c = 4$ - četiri su stupca čvorova
- $l = 2$ - moguće je dohvatiti izlaze čvorova udaljene za najviše dva stupca unazad u odnosu na ulaz čvora u trenutnom stupcu (npr. ulazi čvorova u zadnjem stupcu se zbog toga nikako ne mogu spojiti sa izlazima čvorova u prvom stupcu)

Dužina genotipa je stalna i iznosi $n_r \cdot n_c \cdot (n_n + 1) + n_o$. Međutim, to je samo maksimalna dužina kartezijskog programa koja jest stalna dok stvarna veličina (gledano kao usmjereni graf) može biti varijabilna, ali ograničena maksimumom. Na primjer, na Slika 1 je samo 9 čvorova povezano, nepovezani čvorovi su označeni sivom bojom i označavaju čvorove s kojima se nitko od čvorova u narednim stupcima nije povezao.

Pri inicijalizaciji genotipa ili primjenom nekog od operatora mutacije, promijenjeni geni se moraju pridržavati određenih ograničenja (pravila) koja osiguravaju da je genotip valjan, odnosno da je preslikavanje genotipa u fenotip smisleno.

Neka c_{kj}^{n} označava gen (jedan od n_n ulaznih veza u čvor) koji je k -ti ulaz (ulazna veza) u čvor u j -tom stupcu[7]. Pritom se treba pridržavati pravila:

$$e_{min} = n_i + (j-l) \cdot n_r$$

$$e_{max} = n_i + j \cdot n_r$$

$$c_{kj}^{n} < e_{max}[\,], \qquad ako\ j < l$$

$$e_{min}[\,] \leq c_{kj}^{n} < e_{max}[\,], \qquad ako\ j \geq l$$

Neka c_k^o označava gen (jedan od n_o izlaza) koji je k -ti izlaz iz programa. Pritom se treba pridržavati pravila:

$$h_{min} = n_i + (n_c - l) \cdot n_r$$

$$h_{max} = n_i + n_c \cdot n_r$$

$$h_{min}[\,] \leq c_k^o < h_{max}$$

Neka c_k^f označava gen (jedan od n_f funkcija) koji predstavlja funkciju u k -tom čvoru. Pritom treba vrijediti:

$$0 \leq c_k^f < n_f$$

Ukoliko se pridržava ovih pravila, mutacija i inicijalizacija genotipa se mogu slobodno primijeniti tako da daju valjano genotip.

[7] Prvi krajnje lijevi stupac je označen kao nulti stupac.

Poboljšanje verzije kartezijskog genetskog programiranja osim jednostavnih funkcija koje se ugrađuju u čvorove (npr. osnovne operacije, trigonometrijske, logičke) dozvoljavaju i upotrebu potprograma kao funkcija što se zapravo može shvatiti kao manji genotip unutar većeg. U genetskom programiranju je to poznato kao automatski definirane funkcije [10] (eng. *Automatically Defined Functions*; skraćeno *ADF*), dok se u kartezijskom genetskom programiranju više spominju automatski iskoristivi izlazi (eng. *Automatic Re-used Outputs*; skraćeno *ARO*) premda ne funkcioniraju na isti način kao *ADF*. Promatrat će se samo jednostavne funkcije ugrađene u čvorove.

1.4.2. Usporedba s genetskim programiranjem (prednosti i mane)

Prikaz kartezijskog genotipa je neovisan o tipu podataka i pri vrednovanju dobrote nije potrebno unutar samog genotipa pamtiti privremene međurezultate kao što je utvrdio Miller u [13]. Međutim, to nije slučaj kod genetskog programiranja gdje se u genotipu stabla terminali zamjenjuju za konstante, a unutar funkcijskih čvorova se pamti međurezultat nastao propagacijom vrijednosti od njihove djece (taj proces se još naziva simbolička regresija kojeg je objasnio Koza u [10]).

Uz to, Miller je naveo kako je križanje u kartezijskom genetskom programiranju smisleno i funkcionalno, dok je kod genetskog programiranja križanje vrlo složen proces. S druge strane mutacija kod kartezijskog genetskog programiranja je naizgled jednostavna, ali se mora pridržavati predefiniranih ograničenja (pravila iz prethodnog poglavlja). Genetsko programiranje s druge strane koristi intuitivnu i jednostavnu mutaciju.

Miller je ustvrdio kako kartezijsko genetsko programiranje zahtjeva manju populaciju kako bi se došlo do optimalnog rješenja za razliku od genetskog programiranja, barem ako se promatra slučaj rješavanja jednostavnih kombinacijskih mreža.

Gledano samo strukturu genotipa, čvorovi u genetskom programiranju su svi međusobno povezani (između bilo koja dva čvora postoji bar jedan put između njih) dok to u kartezijskom genetskom programiranju ne mora biti slučaj. S jedne strane to je vrlo prikladno za rješavanje problema vezanih uz oblikovanje kombinacijskih mreža jer se

kombinacijske mreže (s više izlaza) znaju sastojati od više odvojenih podmreža po pojedinom izlazu čiji se unutarnji čvorovi nigdje ne preklapaju ili samo djelomično preklapaju, a polaze od istih ulaza. Može se reći kako je prikaz korišten u kartezijskom genetskom programiranju bliži stvarnosti nego što je to slučaj kod genetskog programiranja.

U prethodnom znanstvenom radu [7] uočen je jedan tehnički nedostatak genetskog programiranja, a to je da je nemoguće s jednim genotipom prikazati kombinacijsku mrežu koja bi imala više izlaza. Moguće rješenje je predstaviti jednu jedinku s više genotipova pri čemu svaki genotip evaluira jedan izlaz, ali to bi bilo učinkovito samo ako za svaki izlaz postoji potpuno odvojena podmreža. U suprotnom, ako bi postojala preklapanja u unutarnjim čvorovima pojedinih genotipova (a često postoje), to bi značilo da svaki takav genotip u sebi gomila višak informacije.

Genetsko programiranje koristi varijabilnu veličinu genotipa za razliku od kartezijskog genotipa sa stalnom veličinom. Ta činjenica omogućuje puno veću ekspresivnost prostora rješenja jer se prostor rješenja dinamički mijenja tokom evolucije. Drugim riječima, genetsko programiranje pretražuje prostor globalno, pri čemu se prostor postepeno proširuje ili sužava pa se pritom ne koncentrira na određeno lokalno područje. Međutim, dinamička globalna pretraga ima i svoju negativnu stranu. Zbog učestalih mutacija i križanja pomoću kojih se genotip mijenja, može doći do naglog rasta genotipa tokom evolucije (eng. *bloat*) što uzrokuje gomilanje nepotrebnog genetskog materijala u genotipu kao što je objašnjeno u [6].

Kartezijsko genetsko programiranje je ograničeno u pretrazi prostora rješenja sa stalnom duljinom genotipa, ali je time izbjegnut nagli rast genotipa. S druge strane, na taj način kartezijsko genetsko programiranje pruža dovoljnu količinu lokalne pretrage za razliku od genetskog programiranja, što može biti korisno u problemu oblikovanja kombinacijskih mreža, kako tvrde Miller i Thompson u [14].

Kao što je navedeno u prethodnom poglavlju, za određivanje stalne veličine kartezijskog genotipa potrebno je odrediti veliki broj parametara (broj inicijalnih ulaza, ulaza u

čvorove, izlaza, funkcija itd.) što predstavlja nedostatak prema genetskom programiranju gdje je dovoljno samo odrediti maksimalnu dubinu rasta stabla te korišteni funkcijski skup i skup terminala.

Važno svojstvo kartezijskog genetskog programiranja je i spomenuta neutralnost. U populaciji može postojati veći skup genotipova koji nemaju isti genetski materijal, ali imaju isti fenotip pa imaju i istu vrijednost dobrote. Kartezijsko genetsko programiranje to postiže na način da se određeni čvorovi mogu aktivirati ili deaktivirati tokom evolucije, što znači da u konačnici nisu svi čvorovi uključeni u prenošenje ulaznih signala do krajnjih izlaza. tj. takvi čvorovi su tada višak ili redundantni. Takvo svojstvo ne postoji kod genetskog programiranja jer tamo uopće ne postoji fenotip, odnosno preslikavanje genotipa u fenotip se odvija po principu 1:1 dok je kod kartezijskog genetskog programiranja to N:1 pri čemu N može biti bilo koji broj.

Garmendia-Doval, Miller i Morley su u [16] dodatno pokazali jasnu odsutnost naglog rasta genotipa i korist neutralnosti u kartezijskom genetskom programiranju. Naime, činjenica da se bilo koji čvor može aktivirati ili deaktivirati u bilo kojem trenutku evolucije podrazumijeva da kada je dobrota genotipova visoka tada postoje veće šanse da će podjednako dobri genotipovi biti izabrani (bez obzira na razliku u genetskom materijalu fenotip im je isti pa su im i dobrote jednake). U genetskom programiranju uporabom genotipa stabla to nije moguće postići jer se većina dobrih fenotipova razlikuje zbog nepotrebno nagomilanog genetskog materijala. Gomilanje genetskog materijala nažalost uzrokuje prenošenje nepotrebnih gena tokom evolucije što paradoksalno ugrožava korisni genetski materijal. Velika količina genetički različitih, ali po fenotipu istih dijelova može postojati bez ikakve štete unutar kartezijskog genotipa (jer se taj dio ne mora vrednovati ako nije dio fenotipa).

Autori naglašavaju kako nagli rast genotipa postoji jedino u redundantnim dijelovima genotipa (bez obzira što je on stalne veličine), ali kako ta redundancija nije uočljiva u fenotipu, posljedica toga je da se prilikom očitanja konačnog rješenja redundantni dijelovi zanemaruju kao da ne postoje. Neutralnost genotipa stoga u sebi nosi i golemu količinu robusnosti.

Fišer, Schmidt, Vašíček i Sekanina su u [17] problem sinteze kombinacijskih mreža rješavali pomoću kartezijskog genetskog programiranja te zaključili kako ono efikasno implicitno otkriva nove strukture mreža i prema tome optimizira mreže univerzalno i neovisno o njihovoj strukturi (kartezijski genotip nije pristran prema strukturi koju oblikuje). Kartezijsko genetsko programiranje oblikuje učinkovite mreže samo na temelju ponašajnih specifikacija, međutim vrednovanje genotipova traje predugo.

Do sličnih zaključaka su došli Vassilev i Miller u [18] gdje su predstavili kartezijsko genetsko programiranje kao evolucijsku metodu koja garantirano pronalazi funkcionalno korektna rješenja, ali je vremenski ovisna.

Kao dodatan „vjetar u leđa“ za odabir upravo kartezijskog genetskog programiranja za rješavanje problema automatskog oblikovanja kombinacijskih mreža u ovom znanstvenom radu pridala je studija koja je primijenila taj genotip u praksi. Drábek i Sekakina su se u [19] bavili problemom automatskog oblikovanja filtara za slike jer su sustavi za raspoznavanje slika ovisni o promjenama u okolini što znači da trebaju biti adaptivni. Koristili su kartezijsko genetsko programiranje u izradi evolucijskih filtara za slike na način da su se filtri implementirali na FPGA komponenti mijenjajući njezinu početnu konfiguraciju. Pokazali su kako je moguće evoluirati filtre veće kvalitete od tradicionalnih filtara pri čemu neki od njih čak zahtijevaju manje logičkih vrata od tradicionalnih. Martínek i Sekanina su nastavili prethodni rad te u [20] predstavili evolucijski filtar za slike u potpunosti implementiran na FPGA komponenti koji minimizira razliku između slike s greškom i slike iz skupa za učenje (koja se nalazi u memoriji na FPGA). Koristeći kartezijsko genetsko programiranje za podešavanje konfiguracije na FPGA za taj filtar, dobiveni rezultati su slične kvalitete kao i drugi filtri navedeni u literaturama. Sva trojica navedenih autora se bave izradom evolucijskih filtara za slike s naglaskom na kartezijsko genetsko programiranje, a njihove publikacije na tu temu se mogu pronaći na [21].

2. Implementacija rješenja

2.1. Evolutionary Computation Framework

Evolutionary Computation Framework (kraće: ECF) je programski radni okvir za evolucijsko računanje implementiran u C++ programskom jeziku, a služi za rješavanje problema metodama evolucijskog računanja (detalje pronaći u [22]). Među evolucijskim metodama ugrađenim u radni okvir nalaze se genetski algoritam (podržana reprezentacija genotipa kao niz bitova, binarno kodirani realni brojevi, vektori s pomičnim zarezom, permutacijski vektori) te genetsko programiranje (podržana reprezentacija genotipa kao stablo). Radni okvir ima ugrađene i neke algoritme kao što su k-turnirski algoritam [23] (eng. *steady-state tournament*), algoritam proporcionalne selekcije [24] (eng. *roulette-wheel*) i algoritam optimizacije rojem čestica [25] (eng. *particle swarm optimization*) koji se mogu koristiti neovisno o odabranom genotipu.

U postojeći radni okvir ugrađen je kartezijski genotip te njegov pripadni operator evaluacije (za vrednovanje dobrote genotipa) i operatori križanja i mutacije. S radnim okvirom je spojen još jedan dio koji će poslužiti za vrednovanje genotipa (opisan u nastavku). Preporučljivo je tokom čitanja narednih poglavlja povremeno pogledati poglavlje o uputama za korištenje (vidi poglavlje 4) gdje se ukratko opisuje proces pokretanja ECF-a te sekundarnih dijelova da bi se oblikovala neka kombinacijska mreža.

2.2. Odabrani oblik opisa kombinacijskih mreža

Nakon odabira prikladne reprezentacije genotipa za rješavanje problema automatskog oblikovanja kombinacijskih mreža, potrebno je bilo odrediti kako će se opisivati problemi (kombinacijske mreže koje treba oblikovati). U raznoj literaturi se za opis kombinacijskih problema koristi jednostavna tablica istinitosti (npr. Coello Coello, Christiansen i Hernández Aguirre u [8] ili Reis i Machado u [9]) što je između ostalog korišteno i u prethodnom znanstvenom radu [7]. Tablica istinitosti može biti prikladan odabir ako se kombinacijske mreže koje je potrebno oblikovati sastoje od malog broja ulaza i izlaza te je

ljudski ekspert u stanju sam oblikovati takvu kombinacijsku mrežu i sastaviti tablicu istinitosti za nju.

S druge strane, ako je kombinacijska mreža zahtjevnijeg ponašanja, potrebno je uvesti apstraktnije opise mreža, što se odlučilo napraviti i u ovom znanstvenom radu. Ponašanje kombinacijske mreže opisano je formalnim jezikom za opis sklopovlja. Kako je takvih formalnih jezika više, odlučeno je koncentrirati se samo na Verilog (detaljni uvid u sam jezik se može pronaći u [26]).

Idući problem kojeg je trebalo riješiti je kako ukomponirati opis kombinacijske mreže Verilog programom u ECF. Znači, potrebno je bilo pronaći prikladni oblik pretvorbe Verilog programa kako bi se ponašanje neke kombinacijske mreže opisane na taj način moglo simulirati unutar ECF-a za potrebe vrednovanja genotipa (procjene dobrote).

Razmatrani oblici pretvorbe mogu se pronaći u tehničkom izvješću [27], a ovdje će biti opisan samo odabrani oblik pretvorbe, a to je pretvorba Verilog programa u C++ program u kojem je pisan i sam ECF kao programski radni okvir.

2.3. Verilator

Pretvorba Verilog programa u C++ program se pokazala najprikladnijom jer se tom pretvorbom dobiva C++ program kojeg je moguće jednostavno ukomponirati u sam ECF (također pisan u C++ programskom jeziku).

Od razmatranih alata odabran je *Verilator* koji djeluje kao prevodioc i simulator za pretvorbu Verilog programa u C++. Više informacija o njemu (upute za instalaciju, dokumentacija) moguće je pronaći na [28].

Kako je moguće vidjeti iz tehničke dokumentacije [29] *Verilator* ne služi toliko kao simulator nego više kao prevodioc Verilog programa u C++ program i to je upravo ono što je potrebno. *Verilator* pretvara Verilog program pogodan za sintezu (eng. *synthesizable*) u C++ program, odnosno stvara .cpp i .h datoteke kao međurezultat. Poziva se s raznim parametrima koji omogućuju i detaljnu simulaciju praćenja rada, ali to nije potrebno

uključivati za osnovni način rada. Tokom svog rada *Verilator* čita datoteku gdje se nalazi opis sklopa (tzv. glavnog modula) u Verilogu, povezuje ga te daje međurezultat u .ccp i .h datotekama. Potrebno je i napisati kratku C++ datoteku poveznicu (eng. *wrapper file*) koja instancira glavni modul iz Verilog datoteke te prenosi ime datoteke komandnoj liniji. Uz to, moguće je dobiti i izvršnu datoteku iz stvorenih .cpp i .h datoteka koja izvodi simulaciju.

Problem predstavlja činjenica da je *Verilator* pisan za Linux/Unix operacijske sustave i ne postoji podrška za Windows operacijski sustav. *Verilator* je potrebno prije svega konfigurirati prevođenjem putem *makefile* i *configure* naredbi unutar Linux/Unix okoline, a za to je najpogodniji *Cygwin* [30] kojeg preporučaju i autori *Verilator*-a (pod uputama za instalaciju na [28]). Prevođenje pod čistim Windowsima bi još bilo moguće uz pomoć *MinGW* [31] prevodioca, ali uz sam izvorni program *Verilator*-a su još potrebni *Perl* [32], *Flex* [33] i *Bison* [34]. *Cygwin* je puno pogodniji jer nudi *Perl*, *Flex* i *Bison* za instalaciju unutar svoje okoline. Unutar *Cygwin*-a potrebno se pozicionirati na mjesto gdje se nalazi izvorni program *Verilatora* (uz napomenu da datotečni put do izvornog programa ne smije sadržavati praznine). Nakon toga potrebno je izvršiti sljedeće kao što autori navode pod uputama za instalaciju[8] [28] :

```
unset VERILATOR_ROOT  # Za bash ljusku

./configure

make

make install
```

Nakon toga, prevedene su izvorne *Verilator*-ove datoteke te je moguće koristiti *Verilator* naredbe unutar *Cygwin*-a. Način na koji se pomoću *Verilator*-a prevodi jednostavna Verilog datoteka u C++ program jest:

```
$ verilator --cc module.v
```

[8] Uz napomenu da se unutar nekih inačica *Cygwin*-a koristi naredba *gmake* umjesto *make*, ali konfiguracija će sama ispisati koju je naredbu potrebno iduće izvesti.

Pritom se u *module.v* nalazi glavni Verilog modul, a *--cc* opcija označava da se Verilog program prevodi u C++ program[9]. Ovime se unutar trenutnog direktorija stvara novi direktorij *obj_dir* gdje su stvorene sljedeće datoteke*: Vmodule.cpp, Vmodule.h, Vmodule__Syms.cpp, Vmodule__.h, Vmodule__ver.d, Vmodule__verFiles.dat, Vmodule.mk, Vmodule_classes.mk.* Ono što je posebno zanimljivo jesu sve stvorene izlazne .cpp i .h datoteke jer se u njima nalazi prevedeni Verilog program u C++.

Kao što autori navode u dokumentaciji za *Verilator* [29] moguće je te izlazne datoteke prevesti s prevodiocem ugrađenim u *Microsoft Visual Studio*. Naime, potrebno je unutar *Microsoft Visual Studio* okoline novom projektu postaviti C++ datoteku poveznicu gdje je opisano instanciranje glavnog Verilog modula i povezivanje s komandnom linijom (ako je to potrebno) te sve .cpp i .h datoteke koje se nalaze u *obj_dir*. Uz to, potrebno je uključiti u postavkama projekta datotečni put do *Verilator*-ovog „include" direktorija tj. na datoteke iz „include" direktorija. Zbog različitih grešaka koje su se javljale prilikom povezivanja datoteka, preporučljivo je još iz „include" direktorija ručno dodati u sam projekt (među ostale datoteke) *verilated.cpp* te *verilated.h*. *Microsoft Visual Studio 2008* tada uspijeva prevesti i povezati datoteke te ih pokrenuti kao simulaciju.

Ono što je zanimljivo jest C++ datoteka poveznica koja pruža instanciranje glavnog modula. Ona je posebno pogodna jer se u njoj može upravljati ulazima i izlazima instanciranog Verilog modula. Jednostavan primjer kako ona izgleda:

```
#include "Vour.h"   //prevedeni glavni modul iz Veriloga
#include "verilated.h" //glavne procedure iz „include" direktorija
int main(int argc, char **argv, char **env) {
    Verilated::commandArgs(argc, argv);   //pamti argumente
    Vour* top = new Vour;  //instanca glavnog modula
    top->reset = 0;        //namjesti ulaze
    top->clk = 1;          //namjesti ulaze
    while (!Verilated::gotFinish()) { //sve dok simulacija modula
```

[9] Zato jer u *Verilator*-u postoji još i opcija da ga se prevodi u *SystemC* i *SystemPerl* pa to *Verilator*-u treba posebno naglasiti.

```
                                        //nije gotova prolazi modulom
            top->eval(); //evaluiraj modul
    }
    cout << top->out_l << endl;    //pročitaj izlaze
    cout << top->out_r << endl;    //pročitaj izlaze
    top->final();  //završi simulaciju
    exit(0);
}
```

Verilog modul kojeg je moguće simulirati prethodno navedenom datotekom poveznicom mora izgledati na sljedeći način, odnosno, ulazi i izlazi moraju biti navedeni ključnim riječima *input* i *output*:

```
module t (out_l, out_r, reset, clk); //navesti ulaze i izlaze modula
    input reset;
    input clk;
    output out_l;
    output out_r;
    // ... ostatak Verilog programa ...
endmodule
```

Ulazi i izlazi se mogu u Verilogu pisati na dva načina, kao skalari i kao vektori (vidjeti u [26]) što je ilustrirano na sljedećem primjeru za ulaze (ekvivalentno vrijedi i za izlaze):

```
    input bit; //skalar veličine jednog bita
    input [7:0] byte; //vektor veličine 8 bitova
    input [31:0] integer; //vektor veličine 32 bita
```

Na temelju toga se u datoteci *Vtop.cpp* definiraju ekvivalentne varijable za Verilogove ulaze i izlaze. Prema tome, potrebno je prije svega pronaći u Verilogovoj datoteci sve ulaze i izlaze i vidjeti jesu li skalari ili vektori (i ako su vektori koje su veličine). Taj proces je automatiziran pisanjem jednostavnog parsera koji kao argument prima put do Verilog

datoteke, a vraća strukturu u kojoj je evidentirano ime svake varijable, je li ulaz ili izlaz i od koliko se bitova sastoji.

Na taj način će biti moguće automatizirati komunikaciju s C++ datotekom poveznicom u koju će se proizvoljno postavljati vrijednosti ulaza i čitati izlaze na način da će datoteka poveznica komunicirati s ostalim dijelovima programa iz ECF-a putem jedinstvenog sučelja. Ono što će to sučelje primati bit će struktura u koju će se spremati vrijednosti ulaza, a preko sučelja će vraćati strukturu gdje će biti spremljene vrijednosti izlaza po završetku rada simulacije Verilog modula. Preko tog sučelja će biti ostvarena komunikacija s evaluatorom dobrote kartezijskog genotipa iz ECF-a.

U nastavku je opisano spajanje rezultata koje daje *Verilator* s ECF-om.

2.4. Povezivanje ECF-a s Verilator-om

2.4.1. Detekcija ulaza i izlaza iz Verilog datoteke

Kako bi se ulazi i izlazi iz Verilog datoteke automatski detektirali, napisan je kratki parser koji ih pronalazi u Verilog datoteci te bilježi njihova imena i veličine. Parser je implementiran u okviru C++ klase `VerilogIODetect`.

Klasa koristi jednu javnu metodu `bool detectVerilogIO(string fileName)` koja prima datotečni put do Verilog datoteke te indicira pozivatelju uspješnost u slučaju da su ulazi i izlazi uspješno detektirani. Sama klasa koristi pomoćno zaglavlje `Regex` koje dolazi u sklopu paketa Boost C++ zaglavlja [35] i omogućava rad s regularnim izrazima. Pomoću regularnih izraza klasa pretražuje Verilog datoteku u potrazi za izrazima oblika:

```
input A, B;
input C, D;
input [4:0] in1, in2;
output E;
output [3:0] out1, out2;
```

```
output [3:0] out3;
```

Kako stoji u [26] ulaze i izlaze je moguće navesti i u samom prototipu modula na sljedeći način:

```
module imeModula (output E, input A, B, C, D);
/* … neki smisleni program …*/
endmodule
```

U literaturi [26] se navodi još jedan način pisanja vektorskih ulaza i izlaza gdje se umjesto konstanti koriste varijable prilikom definiranja veličine vektora:

```
module nekiModul(out, in);
parameter WIDTH = 7;
input [WIDTH:0] in;
output [WIDTH:0] out;
/* … neki smisleni program …*/
endmodule
```

Još jedan način pisanja vektorskih vrijednosti kakav se može pronaći u literaturi [26] je:

```
module xor8 (xout, xin1, xin2);
output [1:8] xout;
input [1:8] xin1, xin2;
/* … neki smisleni program … */
endmodule
```

Zadnja tri navedena načina pisanja ulaza i izlaza (te možda još neki koji nisu navedeni, a mogu se isto tako pronaći u literaturi) nisu podržana implementiranim detektorom ulaza i izlaza. Osim toga, imena ulaza i izlaza mogu biti samo alfanumerički znakovi (npr. ime `in_1` se neće moći detektirati). To bi značilo da je originalnu Verilog datoteku možda potrebno samo malo prilagoditi (srećom, u pitanju nisu neke velike promjene).

Nakon detekcije ulaza i izlaza, oni se posebno odvajaju u dvije strukture, u jednom se nalaze popisana imena ulaza i njihovih veličina, a u drugoj popisana imena izlaza te također njihovih veličina. Ukoliko je riječ o skalarnoj vrijednosti (npr. ulaz `A` je skalar), veličina tog ulaza ili izlaza će se zapisati kao 0. Ukoliko je riječ o vektorskoj vrijednosti, tada će se veličina zapisati kao `N` ako je vektor u Verilog datoteci naznačen s `[N:0]`(npr. veličina ulaza `in1` će biti 4). Zapravo je veličina izražena u broju bitova veća za jedan od očitane veličine pa će tako ulazni skalar `A` biti veličine jednog bita, a ulazni vektor `in1` veličine pet bitova.

Rezultate detekcije ulaza i izlaza iz Verilog datoteke koristit će generator datoteke poveznice i generator ulaznih vrijednosti koji će biti opisani u nastavku.

2.4.2. Generiranje datoteke poveznice

Na temelju očitanja ulaza i izlaza koje je pripremila klasa `VerilogIODetect` generira se datoteka poveznica (eng. *wrapper file*) koja će poslužiti za komunikaciju s datotekama koje je stvorio *Verilator*. Generiranje datoteke poveznice omogućuje klasa `VerilogWrapperGenerator` koja prvo pomoću klase `VerilogIODetect` detektira ulaze i izlaze te pomoću svoje javne metode `bool createWrapperFile(string filePath)`, koja kao parametar prima datotečni put do Verilog datoteke, stvara datoteku poveznicu.

Datoteka poveznica je zapravo stvorena kao klasa `Wrapper`. U njoj se instancira glavni modul iz Verilog datoteke (koji je sad pretvoren u C++ program) na način da se u datoteci poveznici uključi glavno zaglavlje koje je *Verilator* generirao za taj modul (nosi naziv *Vmodule.h* gdje je *module* ime Verilog datoteke) te glavno zaglavlje iz *Verilator*-ovog „include“ direktorija (*verilated.h*). U tom slučaju je s javnom metodom klase `void evaluate(map<string, uint> inputs, vector<uint> &outputs)` omogućena evaluacija modula na temelju ulaza koji su postavljeni u `inputs` parametar te očitavanje izlaza iz modula koji će se postaviti u `outputs` parametar. Tu metodu koristi generator ulaznih vrijednosti koji stvara određeni broj ulazno-izlaznih parova vrijednosti kako bi se oni iskoristili prilikom vrednovanja genotipa (za provjeru

funkcionalne ispravnosti kombinacijske mreže koja je predstavljena tim genotipom). Generator ulaznih vrijednosti opisan je u nastavku.

2.4.3. Generator ulaznih vrijednosti

Generator ulaznih vrijednosti implementiran je u klasi `InputGenerator` , a služi kako bi na temelju očitanja ulaza i izlaza (od klase `VerilogIODetect`) slučajno generirao ulazne vrijednosti te ih slao na evaluaciju modula datoteci poveznici (klasa `Wrapper`). Nakon toga sprema izlazne vrijednosti koje je dobio nakon evaluacije modula i sastavlja skup ulazno-izlaznih vrijednosti određene veličine. Generator će služiti kako bi pomogao u evaluaciji kombinacijske mreže koju predstavlja određeni Verilog modul jer su u njemu spremljeni ulazi i izlazi koji se trebaju dobiti.

Ono što zapravo generator ulaznih vrijednosti radi jest iz ponašanja kombinacijske mreže opisane Verilog programom sastavlja jednu internu tablicu istinitosti. Taj proces je dakako nevidljiv izvana jer podrazumijeva da se kombinacijska mreža opisuje na način da se specificira njezino ponašanje u Verilogu.

Konstruktor klase `InputGenerator(string filename, uint percentage)` kao parametre prima datotečni put do Verilog datoteke (služi da bi se proslijedio instanci klase `VerilogIODetect` da se detektiraju ulazi i izlazi) te postotak domene koju je potrebno generirati (dozvoljene vrijednosti su bilo koji prirodni broj u intervalu $\in [0,100]$). Ukoliko je postotak domene 100% tada se sastavlja cjelovita tablica istinitosti, a ako je postotak domene manji npr. 50% tada se slučajno generiraju ulazi iz domene sve dok se ne skupi dovoljno ulazno-izlaznih parova da se tablica istinitosti napuni do tog određenog postotka. Veličina domene je poznata, ovisi o broju ulaza te veličini svakog ulaza (u broju bitova). Tako se za veličinu domene može općenito reći da iznosi

pri

čemu l_k označava broj bitova k -tog ulaza, ako je tih ulaza ukupno n .

Javna metoda klase `map<string, uint> setInputs()` služi kako bi se slučajno generirao jedan skup ulaznih vrijednosti. Metoda `void storeResults(vector<uint> outputs)` koristi se da bi se spremile izlazne vrijednosti koje se dobivaju od `Wrapper` klase kada se pak njezinoj glavnoj metodi `evaluate` proslijede ulazne vrijednosti generirane u metodi `setInputs`. Izlazne vrijednosti se neće spremiti u slučaju da se slučajno generira kopija ulaznih vrijednosti čiji je original već spremljen u internu tablicu istinitiosti (inače bi se stvarale nepotrebne kopije u tablici istinitosti). Metoda `bool stopGenerator()` služi jedino za provjeru je li potrebno generirati još ulazno-izlaznih parova ili je tablica istinitosti (koju si `InputGenerator` interno sprema) dovoljno popunjena (jer zna koliko je velika domena i koliki postotak treba popuniti).

Mogućnost odabira postotka pokrivenosti domene je uvedena jer je uočeno kako bi se za velike domene predugo punila tablica istinitosti. Obzirom da se ulazne vrijednosti ne smiju ponavljati, to bi značilo da se svako ponavljanje treba odbaciti. Ukoliko je domena velika, a tablicu istinitosti treba u potpunosti popuniti, postavlja se pitanje koliko bi se ona učinkovita punila jer je riječ o jednostavnom slučajnom odabiru ulaznih vrijednosti. Što je veći broj vrijednosti u domeni te kako se tablica istinitosti s vremenom sve više popunjava, to će biti teže „pogoditi“ upravo one vrijednosti iz domene koje nedostaju da bi tablica istinitosti bila popunjena do kraja. Razlog tome je što se koristi jednostavan slučajni generator brojeva koji ne ignorira vrijednosti koje su bile prethodno odabrane nego odabire bilo koju (naravno, slučajno) iz prethodno postavljenog intervala.

Kada je interna tablica istinitosti dovoljno popunjena, ulazno-izlazni parovi se mogu iščitati iz strukture `vector<vector<uint> > domain` gdje se nalaze parovi ulaznih vrijednosti i strukture `vector<vector<uint> > codomain` gdje se nalaze njihovi korespondentni parovi izlaznih vrijednosti.

Klasa `InputGenerator` se koristi zajedno s klasom `Wrapper` u operatoru evaluacije kartezijskog genotipa koji koristi internu tablicu istinitosti klase `InputGenerator` da bi procijenio funkcijsku ispravnost genotipa (dobiva li genotip evaluacijom ulaza iste izlazne

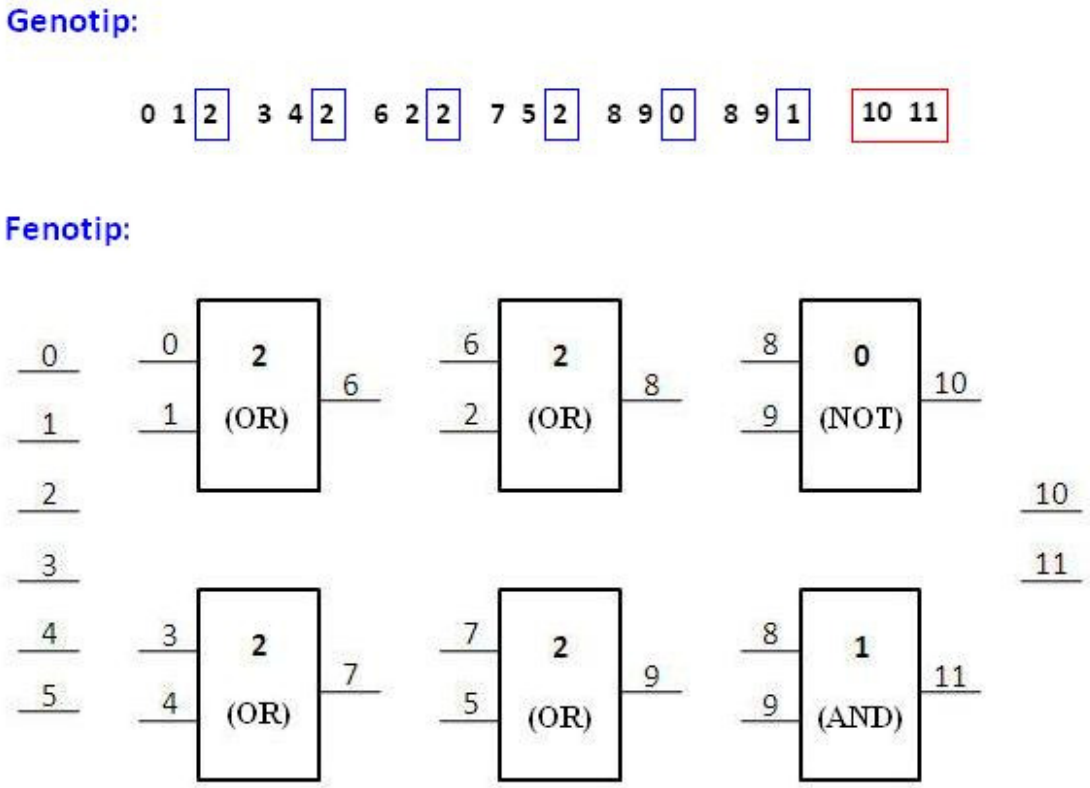
vrijednosti kao i generator). O tom operatoru će biti više riječi u dijelu u implementaciji kartezijskog genotipa (vidi poglavlje 2.5.5).

2.5. Implementacija kartezijskog genotipa unutar ECF-a

2.5.1. Kartezijski genotip

Kartezijski genotip je unutar ECF-a implementiran kao klasa `Cartesian` koja je zapravo niz prirodnih brojeva implementacijski predstavljen kao `vector<uint>`. Neke od važnijih metoda same klase objašnjene su u nastavku teksta.

Metoda koja inicijalizira kartezijski genotip je `bool initialize(StateP state)` i u njoj se odvija provjera parametara koji su postavljeni u konfiguracijskoj datoteci (najvažniji parametri su detaljnije opisani u poglavlju 2.6, a konfiguracijska datoteka u poglavlju 4). Primjer inicijalizacije jednog kartezijskog genotipa je prikazan na Slika 2.



Slika 2. Primjer kartezijskog genotipa za problem oblikovanja kombinacijskih mreža. Prikazano je preslikavanje genotipa u fenotip. U genotipu su plavim pravokutnicima označeni funkcijski čvorovi, a u crvenom pravokutniku su krajnji izlazi. Pretpostavlja se da su funkcijski čvorovi redom definirani kao NOT (0), AND (1) i OR(2).

U inicijalizacijskoj metodi se provjerava tip podatka nad kojim se žele izvoditi operacije u genotipu (u ovom slučaju je to *unsigned int*) te se instancira funkcijski skup (klasa `FunctionSet`) zajedno s definicijom tipa podatka. Čitaju se imena funkcija koja trebaju predstavljati funkcijske čvorove te je uz neke od njih moguće navesti i broj parametara koje će te funkcije primati (npr. za funkciju zbrajanja moguće je navesti da se koriste tri pribrojnika). U slučaju da se broj parametara funkcije ne navede, koristi se podrazumijevani broj parametara (npr. za već spomenutu funkciju zbrajanja podrazumijevana su naravno dva pribrojnika). Tada se u funkcijski skup dodaju funkcije tj. instance klasa tih funkcija (funkcija će biti instancirana samo ako je implementirana za taj genotip).

U samom genotipu funkcija je predstavljena prirodnim brojem (vidi na Slika 2). Prema tome, poredak u kojem su navedene funkcije u konfiguracijskoj datoteci ujedno označava i njihove indekse (tj. redne brojeve) u funkcijskom skupu. Pojedini indeks jedinstveno označava neku funkciju pa kada se iz genotipa pročita prirodni broj koji predstavlja funkciju, taj broj zapravo predstavlja njezin indeks u funkcijskom skupu ili poredak navođenja u konfiguracijskoj datoteci.

Nakon određivanja funkcijskog skupa u inicijalizacijskoj metodi se čita i provjerava ispravan unos za broj ulaznih veza u funkcijske čvorove, konstante koje će stajati na inicijalnim ulazima (ukoliko one postoje, a ovdje nisu bile potrebne), broj varijabli koje će biti na inicijalnim ulazima (broj ulaza u kombinacijsku mrežu), broj izlaza, broj redaka, broj stupaca i parametar stupnja povratka.

Inicijalizacija kartezijskog genotipa na kraju provjere koristi metodu `void makeGenotype()` koja slučajno stvara niz prirodnih brojeva koji predstavlja genotip, ali u okviru s postavljenim pravilima iz poglavlja o kartezijskom genotipu (vidi poglavlje 1.4.1). U tome joj pomažu dodatne metode `uint randInputConn(uint currCol)` koja odabire slučajni broj koji predstavlja neku ulaznu vezu u funkcijski čvor, `uint randFunction()` koja odabire slučajni broj koji predstavlja redni broj funkcije te `uint randOutput()` koja odabire slučajni broj koji predstavlja krajnji izlaz iz genotipa.

Metoda koja se koristi prilikom vrednovanja genotipa je `void evaluate(voidP inputs, void* result, uint funcNum)`. Pritom parametar `inputs` označava vrijednosti ulaza koji ulaze u neki čvor genotipa, parametar `result` pokazivač u kojeg treba upisati rezultat izvršene operacije nad dobivenim ulazima u tom čvoru te parametar `funcNum` koji označava redni broj funkcije u funkcijskom skupu (tj. operaciju koja će se izvršiti). Ta metoda interno poziva sličnu metodu evaluacije, ali za konkretni funkcijski čvor u funkcijskom skupu.

2.5.2. Funkcijski skup

Funkcijski skup označava niz funkcijskih čvorova koji se mogu pojaviti u kartezijskom genotipu. Funkcijski skup implementiran je kao klasa `FunctionSet`.

Klasa sa svojim konstruktorom `FunctionSet(string gettype)` registrira tip podatka nad kojim će funkcije u funkcijskom skupu provoditi operacije te registrira popis implementiranih funkcijskih čvorova (zasad su to: zbrajanje, oduzimanje, množenje, dijeljenje, sinus, kosinus, logičko I, logičko ILI, logičko NE, logički isključivi ILI, logička negacija isključivog ILI).

Za dodavanje funkcija u funkcijski skup koriste se dvije metode. Jedna je `bool addFunction(string name)`, a druga je `bool addFunction(string name, uint numArgs)`. Prva metoda kao svoj parametar prima samo ime (`name`) funkcije koju treba instancirati, a kao broj njezinih parametara se uzima podrazumijevana vrijednost. Druga metoda osim parametra imena funkcije prima i parametar `numArgs` koji označava broj odabranih parametara te funkcije. Funkcija (funkcijski čvor) koja će se instancirati ovisi i o odabranom tipu podataka koji se prenio u konstruktoru već pri samom instanciranju klase `FunctionSet`.

Metoda `void evaluate(voidP inputs, void* result, uint funcNum)` se poziva unutar istoimene metode u klasi samog genotipa (`Cartesian`), a ovdje ona poziva također metodu istog imena, ali za konkretnu funkciju (funkcijski čvor) koja treba evaluirati ulazne vrijednosti i spremiti rezultat.

Svaka od implementiranih funkcija polazi od iste apstraktne klase `Function` koja ima dva konstruktora. Jedan koji će instancirati funkciju s podrazumijevanim brojem parametara (`Function()`), a drugi koji će instancirati funkciju s proizvoljnim brojem parametara (`Function(uint numArgs)`). Važna metoda klase `Function` je `virtual void evaluate(voidP inputs, void* result)` koju treba implementirati svaka izvedena klasa apstraktne klase `Function`. Ta metoda evaluira

ulazne vrijednosti te vraća izlazne vrijednosti ovisno o funkcionalnosti koju implementira neka funkcija (funkcijski čvor).

Implementirane izvedene klase apstraktne klase `Function` su: `Add` (zbrajanje), `Sub` (oduzimanje), `Mul` (množenje), `Div` (dijeljenje), `Sin` (sinus), `Cos` (korisnus), `And` (logičko I), `Or` (logičko ILI), `Not` (logičko NE), `Xor` (logičko isključivo ILI) i `Xnor` (negacija logičkog isključivog ILI).

Trenutno su važne samo one koje se vežu uz logičke operacije[10] jer se vežu uz trenutno obrađivani problem, a i metoda `evaluate` im je zbog specifičnog evaluacijskog operatora drugačija od ostalih implementiranih funkcija. U njihovoj metodi `evaluate` iz parametra `inputs` se osim ulaznih vrijednosti čitaju i veličine tih ulaznih vrijednosti u bitovima. Iz niza koji se primio u parametru `inputs` se pročita prva ulazna vrijednost pa njezina veličina u bitovima, zatim druga ulazna vrijednost pa njezina veličina u bitovima itd. Razlog tome je što se htjelo dozvoliti logičke operacije s ulazima različitog broja bitova jer takvo nešto podržava Verilog tj. dozvoljeno je pisati operacije koje se izvode nad ulazima različitog broja bitova, koliko god to možda nesmisleno bilo.

Međutim, to se može promatrati na sljedeći način. Ako se prime npr. dva parametra i jedan ima tri bita, a drugi šest bitova, tada se nad tri najniža bita drugog parametra i prvim parametrom provede tražena logička operacija kao da se provodi bit po bit. Tri najviša bita drugog parametra izvodu tu istu logičku operaciju s nulama jer se prvi parametar može gledati i kao broj sa šest bitova, ali kojemu su prva tri najviša mjesta popunjena nulama.

Prema tome, pronalazi se ulazna vrijednost s najvećim brojem bitova te se logička operacija provodi kao da sve ulazne vrijednosti imaju taj broj bitova, ali su im najviši bitovi nadopunjeni nulama.

[10] One koje se ne vežu uz logičke operacije su implementirane za jedan drugi evaluacijski operator kartezijskog genotipa koji radi s *int* i *double* vrijednostima, više o tome je moguće pročitati u [15].

2.5.3. Operator mutacije

Operator mutacije kartezijskog genotipa implementiran je u klasi `CartesianMutOnePoint` koja mutira kartezijski genotip na način da slučajno odabire mjesto mutacije (jedan prirodni broj u nizu prirodnih brojeva koji predstavlja genotip) te na to mjesto stavlja neku drugu slučajno odabranu vrijednost.

Metoda `bool mutate(GenotypeP gene)` radi upravo ono što je opisano, s time da se prvo gleda postoji li možda definirana vjerojatnost mutacije[11] pa se prije svega prvo gleda hoće li se taj genotip uopće mutirati, a zatim se tek mutira na nekom slučajno odabranom mjestu. Ukoliko ne postoji definirana vjerojatnost mutacije tada se odmah bira slučajno mjesto mutacije.

Na temelju metode `mutOneValue(Cartesian* mut, int mutPoint)` se mutira genotip na način da se promijeni vrijednost na mjestu `mutPoint` u samom genotipu. Pritom se treba vidjeti nalazi li se na slučajno odabranom mjestu mutacije ulazna veza u čvor, indeks funkcije ili krajnji izlaz te pomoću pomoćnih funkcija iz samog genotipa (`randInputConn(uint currCol)`, `randFunction()`, `randOutput()`) odabrati valjanu zamjensku vrijednost pridržavajući se postavljenih pravila (iz poglavlja 1.4.1 o kartezijskom genotipu).

2.5.4. Operator križanja

Operator križanja kartezijskog genotipa implementiran je u klasi `CartesianCrsOnePoint`. Operator križanja radi na način da se uzimaju dva genotipa roditelja te se odredi slučajno mjesto križanja u genotipu. Nakon toga se s obzirom na to mjesto prijeloma uzima jedan dio genotipa od jednog roditelja te drugi dio genotipa od drugog roditelja i od njihovih genetskih materijala se stvori genotip djeteta.

[11] To se može definirati u konfiguracijskoj datoteci te se pročita u inicijalizacijskoj metodi operatora mutacije.

Križanje je izvedeno u metodi `bool mate(GenotypeP gen1, GenotypeP gen2, GenotypeP child)`. Prvo se slučajno bira mjesto križanja, a zatim se slučajno bira roditelj čija će se lijeva strana (lijevo gledano od mjesta križanja) genetskog materijala uzeti za genotip dijete. Od drugog roditelja se tada uzima desna strana (opet, desno gledano od mjesta križanja). Za razliku od mutacije, križanje u kartezijskom genotipu je vrlo jednostavno jer se ne treba pridržavati postavljenih pravila nego se bez obzira na odabranu točku križanja križanjem dobiva valjani genotip djeteta.

2.5.5. Evaluacijski operator

Evaluacijski operator kartezijskog genotipa implementiran je u klasi `CircuitEvalOp`, a genotipove vrednuje samo s funkcionalnog aspekta (važno je da kombinacijska mreža radi ispravno). U vrednovanju mu pomažu prethodno opisana klasa `InputGenerator` koja generira slučajne ulazne vrijednosti za kombinacijsku mrežu i klasa `Wrapper` koja evaluira Verilog modul u kojem je opisano ponašanje te kombinacijske mreže i vraća izlaze koji se trebaju dobiti. Ti izlazi se zajedno s korespondentnim ulazima upisuju u internu tablicu istinitosti u klasi `InputGenerator`.

Prilikom inicijalizacije u metodi `bool initialize(StateP)` iz konfiguracijske datoteke se čita datotečni put do Verilog datoteke gdje se nalazi glavni modul (opisano ponašanje kombinacijske mreže) i postotak pokrivenosti domene ulaznih vrijednosti koje treba generirati generator ulaznih vrijednosti. Generatoru ulaznih vrijednosti (`InputGenerator`) se tada prosljeđuju oba parametra te on detektira ulaze i izlaze u Verilog datoteci (pomoću klase `VerilogIODetect`). Ukoliko je za pokrivenost domene naznačeno da je ona 100%, tada se odmah prilikom inicijalizacije dopunjuje interna tablica istinitosti generatora ulaznih vrijednosti.

Metoda `FitnessP evaluate(IndividualP individual)` vrednuje jedinku tj. njezin genotip. Prvo se provjerava je li možda proces evolucije ušao u iduću generaciju te se u tom slučaju iznova dopunjuje interna tablica istinitosti generatora ulaznih vrijednosti (osim ako je bilo naznačeno da pokrivenost domene mora biti 100%, tada to nije

potrebno). Drugim riječima, u svakoj novoj generaciji se genotipovi vrednuju s obzirom na drugačiji (slučajno izabrani) skup ulazno-izlaznih vrijednosti.

Postupno se prolazi kroz genotip, pamteći vrijednosti, prvo s inicijalnih ulaza (njih je postavio generator ulaznih vrijednosti), a zatim postupno s izlaza funkcijskih čvorova, s time da se s evaluacijom po funkcijskim čvorovima kreće od krajnje lijevog stupca prema krajnje desnom (prema krajnjim izlazima). Stupac s funkcijskim čvorovima se pak evaluira tako da se krene od krajne gornjeg retka pa se ide prema krajnjem donjem retku. Prema tome, smjer kretanja evaluacije je nadesno i prema dolje. Na kraju se rezultati dobiveni na krajnjim izlazima genotipa uspoređuju s izlaznim vrijednostima iz interne tablice istinitosti generatora ulaznih vrijednosti koji se trebaju dobiti. Ukoliko su oba rezultata jednaka, dobrota genotipa se uvećava za jedan.

Dobrota genotipa je dakle prirodni broj koji se kreće od 0 do N gdje N označava broj parova ulaznih vrijednosti u domeni koje je potrebno ispravno preslikati u korespondentne parove izlaznih vrijednosti u kodomeni.

2.6. Parametri kartezijskog genotipa

Parametri kartezijskog genotipa koje je potrebno definirati da bi se genotip mogao biti ispravno definiran jesu (detaljno su objašnjeni u poglavlju 1.4.1):

1. Broj ulaznih varijabli – podrazumijeva inicijalne ulaze koje se zamjenjuju nekim skupom ulaznih vrijednosti iz domene (prilikom rješavanja problema automatskog oblikovanja kombinacijskih mreža to je jedan skup ulaznih vrijednosti iz domene)

2. Skup konstanti – podrazumijeva slučajeve gdje je potrebno na inicijalne ulaze postaviti konstantne vrijednosti koje tu ostaju neovisno o postojanju inicijalnih ulaza koji su varijable i čije se vrijednosti mijenjaju tokom evaluacije genotipa (skup konstanti se prilikom rješavanja problema automatskog oblikovanja kombinacijskih mreža ne koristi)

3. Broj izlaza – podrazumijeva krajnje izlaze u genotipu (prilikom rješavanja problema automatskog oblikovanja kombinacijskih mreža to je jedan skup izlaznih vrijednosti iz kodomene)

4. Broj ulaznih veza – označava broj ulaza svakog pojedinog funkcijskog čvora u kartezijskom genotipu

5. Broj redaka – broj redaka funkcijskih čvorova u genotipu

6. Broj stupaca – broj stupaca funkcijskih čvorova u genotipu

7. Parametar stupnja povratka – broj koji označava spojenost funkcijskih čvorova iz prethodnih stupaca s funkcijskim čvorom u trenutnom stupcu

8. Funkcijski skup – skup funkcijskih čvorova koji se koriste u genotipu (prilikom rješavanja problema automatskog oblikovanja kombinacijskih mreža to su samo logičke funkcije)

Način na koji se ti parametri navode u konfiguracijskoj datoteci zajedno s ostalim parametrima je objašnjen u zasebnom poglavlju (vidi poglavlje 4).

3. Analiza rezultata

3.1. Početna zapažanja

Prilikom odabira ispitnih primjera za ispitivanje učinkovitosti predloženog rješenja za problem oblikovanja kombinacijskih mreža, uočena su neka ograničenja koja postavlja opis programa u Verilogu.

Naime, kartezijski genotip predstavlja jednu kombinacijsku mrežu. U Verilog programu trebalo bi biti specificirano ponašanje kombinacijske mreže. Kao što je navedeno u [26], Verilog dopušta dva načina opisa nekog općenitog sklopovlja, jedan je onaj pogodan za sintezu (eng. *synthesizable*) gdje se opisuje kako radi neko sklopovlje, a drugi je ponašajni opis (eng. *behavioral*) gdje se opisuje što radi neko sklopovlje. Moguće je i miješati oba oblika opisa. U čistom opisu pogodnom za sintezu specificira se povezanost osnovnih komponenata sklopovlja te taj opis nije trenutno zanimljiv jer je potrebno znati unutrašnju strukturu sklopovlja (a to se želi oblikovati). S druge strane, čisti ponašajni opis definira ulaze i izlaze te na puno apstraktniji način (bliži načinu kako ljudi razmišljaju) opisuje samo ponašanje sklopa. Jasno je kako je isplativije koristiti ponašajni opis, međutim, treba biti pažljiv i na vrijeme uvidjeti što neki opis predstavlja te može li se iz toga oblikovati neka kombinacijska mreža.

Jedan takav primjer je klasičan primjer automata stanja iz literature [26]:

```
module fsmNB (out, in, clock, reset);
    output                          out;
    input                           in, clock, reset;
                                    // kako očitati cS1 i cS0?
    reg                             out, cS1, cS0;
    always @(cS1 or cS0)     // kombinacijski dio mreže
            // OPREZ: očitanje glavnog izlaza ovisi o memorijskim
            //elementima cS0 i cS1!
```

```
            out = ~cS1 & cS0;
    always @(posedge clock or negedge reset) begin
            // sekvencijski dio mreže
            // cS0 i cS1 se ovdje ponašaju kao memorijski elementi
            if (~reset) begin
                    cS1 <= 0;
                    cS0 <= 0;
            end
            else begin
                    cS1 <= in & cS0;
                    cS0 <= in | cS1;
            end
    end
endmodule
```

Zahtjevi za ispravnim oblikom pisanja ulaza i izlaza iz poglavlja o detekciji ulaza i izlaza (vidi poglavlje 2.4.1) su zadovoljeni, ali postoji nešto što treba zamijetiti, a to je definicija `req` varijabli `cS1` i `cS0`. Nijedna od tih varijabli nije definirana kao ulaz niti kao izlaz, a kada se nešto definira kao `req` varijabla time se želi reći da se ona ponaša kao registar [26], a kada je nešto registar tada je to memorijski element [4]. Blok `always` se u Verilogu izvršava konstantno [26] (kao beskonačna petlja), prema tome u Verilogu je jednostavno predstaviti sekvencijske mreže bez potrebe za eksplicitnim uvođenjem beskonačne petlje. Sekvencijske mreže su kombinacijske mreže u kojima postoje petlje, odnosno postoje memorijski elementi [4]. Jasno je da se takvo nešto ne može predstaviti običnom kombinacijskom mrežom, odnosno genotipom koji predstavlja kombinacijsku mrežu. Prema tome, treba u potpunosti odbaciti Verilog opise u kojima stoje `req` varijable u kojima se nešto pamti pri svakom prolazu beskonačne petlje jer je vrlo vjerojatno riječ o sekvencijskoj mreži.

S druge strane, ne treba ni pretjerivati s apstraktnim ponašajnim opisima. Jedan takav jednostavan primjer je opis zbrajala [26]:

```
module adder (sum, a, b);

    input   [3:0]   a, b;

    output  [3:0]   sum;


    assign sum = a + b; // OPREZ: ovo je modulo zbrajanje!

endmodule
```

Kada se takav primjer pokrene, dobivaju se rješenja koja navode na djelomično zbrajalo [4] (eng. *half adder*) i takva rješenja su korektna. Međutim, dobrota rješenja pri svakom pokretanju stane na jednoj vrijednosti i ne poboljšava se s obzirom na maksimalnu moguću vrijednost dobrote. Razlog tome je što se prilikom evaluacije ovog Verilog modula zbrajanje provodi kao modulo operacija s brojem 16 (jer su vektori `a` i `b` veličine 4 bita). Dakle, kada se npr. za `a` postavi vrijednost 8, a za `b` vrijednost 10, tada Verilog daje kao izlazni rezultat u `sum` vrijednost 2. To je ispravan rezultat, međutim, kako da kombinacijska mreža zna za modulo operaciju? Verilog takvu operaciju provodi u pozadini i ona nije jasno vidljiva iz samog Verilog programa, međutim treba biti pažljiv i koristiti samo one operacije koje ne zahtijevaju neki pozadinski proces jer to genotip koji predstavlja kombinacijsku mrežu neće biti u stanju uočiti.

Prema tome, ako rezultati djeluju sumnjivo, možda bi bilo potrebno prije namještanja parametara u konfiguracijskoj datoteci otvoriti Verilog datoteku i pokušati vidjeti je li se nešto slučajno predvidjelo (pogotovo ako se sadržaj u Verilog datoteci nije vlastoručno pisao). Još primjera ograničenja može se pronaći u dodatku radu.

Osim obraćanja malo pažnje na opis u Verilog datoteci, potrebno se još osvrnuti na generator ulaznih vrijednosti koji može pokriti jedan dio (postotak) domene ulaznih vrijednosti tj. ne mora u potpunosti ispuniti svoju internu tablicu istinitosti. Ukoliko se ne odluči puniti cijela tablica istinitosti (npr. u slučaju da se želi ubrzati izvođenje ECF-a), slučajno će se puniti svaki puta drugačija djelomična tablica istinitosti pri ulasku u novu generaciju tokom evolucije. Dobrota genotipa u tom slučaju će biti jednaka maksimalnoj dobroti pomnoženoj s postotkom domene koji se pokriva. Npr. ako se odluči ispitati primjer gdje se nalaze 4 jednobitna ulaza te se za postotak popunjenosti domene uzme

50%, maksimalna dobrota bi bila jednaka 16 da je domena popunjena u potpunosti, ali zbog smanjene tablice istinitosti, maksimalno moguća dobrota u tom slučaju će biti 8. Ništa ne garantira da ako je dobiveno rješenje s dobrotom 8 da je njegova dobrota gledano na cijelu domenu jednaka 16. Smanjena popunjenost domene podrazumijeva samo brže izvršavanje ECF-a. Dakle, ako se želi biti apsolutno uvjeren u ispravnost rezultata, preporučljivo je za popunjenost domene uvijek koristiti 100%, što će se koristiti i prilikom ispitivanja ispitnih primjera u nastavku teksta. S druge strane, moglo bi se i dobrotu definirati s obzirom na postotak tablice istinitosti (uz normiranje vrijednosti).

3.2. Ispitni primjeri i ispitna pitanja

Prilikom odabira ispitnih primjera postavljeno je pitanje što se želi ispitivati tj. kakva pitanja će se razmatrati tokom ispitivanja. Odlučeno je da će se ispitna pitanja podijeliti u tri kategorije:

1. Međusobni utjecaj parametara genotipa – kakav je međusobni utjecaj parametara broja redaka, stupaca i stupnja povratka, ovise li o strukturi ispitnog primjera ili ih je dovoljno uvijek postaviti u isti odnos da se dobiju zadovoljavajuća rješenja

2. Kvantitativna uspješnost rezultata – za koje parametre je dobivena najveća dobrota genotipa i je li dobivena kombinacijska mreža u potpunosti ispravna

3. Kvalitativna uspješnost rezultata – kako strukturalno izgleda kombinacijska mreža za koju je dobivena najveća dobrota genotipa te kakva je njezina struktura u odnosu na strukturu rješenja kojeg je predložio ljudski ekspert

Ispitni primjeri su podijeljeni u dvije kategorije:

1. Jednostavni primjeri – primjeri sastavljeni od jednobitnih ulaza te jednog jednobitnog izlaza (*binaryToESeg_Behavioral* koji opisuje pretvorbu binarnog koda u 8-segmentni prikaz, *synCaseWithDefault* koji opisuje jednostavnu upotrebu `case` konstrukta)

2. Primjeri s više izlaza – primjeri u kojima postoji više izlaza od jednog (*oneBitFullAdder* koji opisuje jednobitno zbrajanje s prijenosom, *decoder2To4* koji opisuje dekodiranje dvije ulazne na četiri izlazne vrijednosti)

Parametri koji su se odabrali prilikom ispitivanja određeni su na temelju relativne usporedbe s rješenjima koje je predložio ljudski ekspert. Prema tome, nisu se uzimale niti premale niti prevelike vrijednosti za broj redaka i stupaca tako da je kao uporište rješenju koje se dobije kartezijskim genotipom poslužilo rješenje kojeg predlaže ljudski ekspert.

Algoritam koji se koristio je bio algoritam proporcionalne selekcije [24] (eng. *roulette-wheel*) koji je ugrađen u sam ECF i važno je napomenuti kako je kod njega bitan način definiranja dobrote. Za taj algoritam je moguće u konfiguracijskoj datoteci podesiti neke parametre (vidi [22]) pa se tako za vjerojatnost križanja postavila vrijednost od 0.5 , a za selekcijski pritisak (koliko je najbolja jedinka „bolja“ od najgore) vrijednost 1.

Ispitivanje se provodilo na vlastitom prijenosnom računalu[12] unutar *Microsoft Visual Studio 2008* razvojne okoline.

3.3. Prva ispitna skupina: jednostavni primjeri

3.3.1. Ispitni primjer: *binaryToESeg_Behavioral*

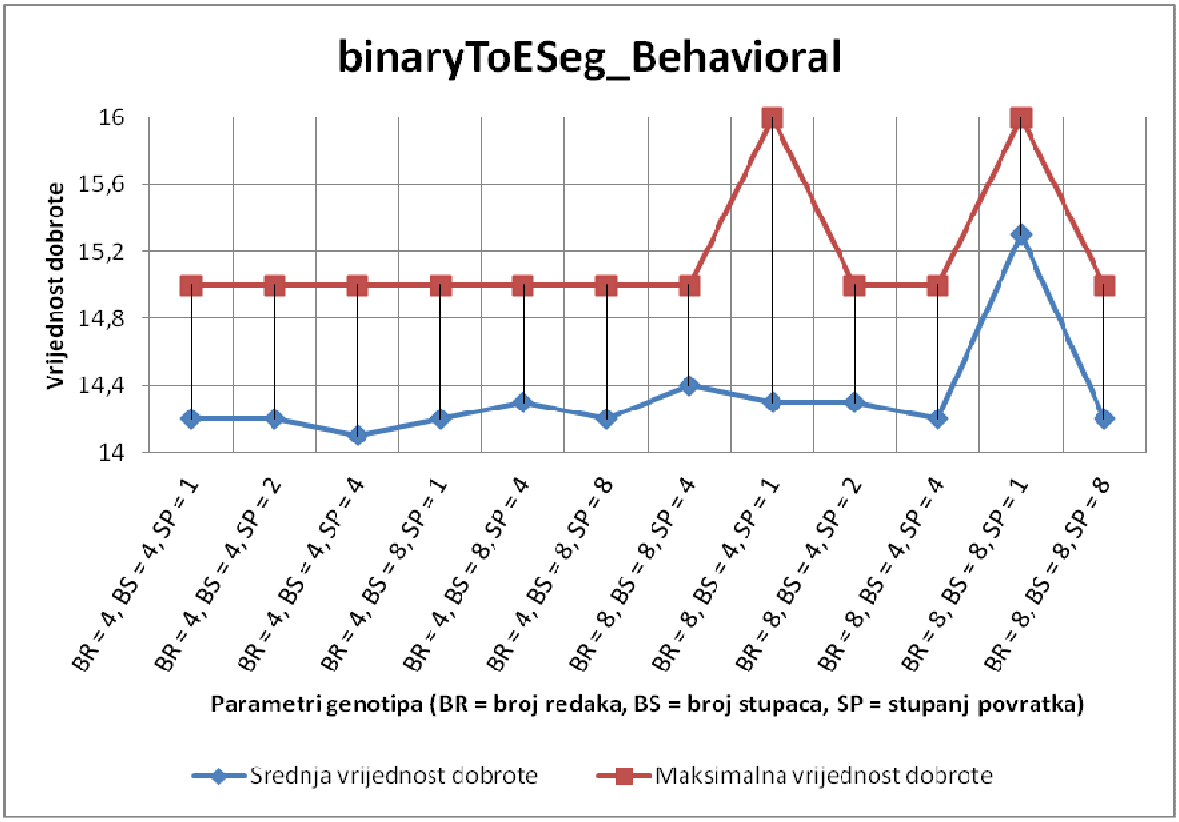
Ispitni primjer *binaryToESeg_Behavioral* (preuzet iz [26]) opisuje pretvorbu binarnog koda u 8-segmentni prikaz na ekranu s time da ovaj opis pokriva samo „E“ segment 8-segmentnog dijela:

```
module binaryToESeg_Behavioral (eSeg, A, B, C, D);
    output          eSeg;
    input           A, B, C, D;
    reg             eSeg;
    always @(A or B or C or D) begin
```

[12] Lenovo R60e, Intel Centrino Duo T5500 @ 1.66GHz, 1GB RAM, OS: WinXP Proffesional

```
            eSeg = 1;
            if (~A & D) // A = 0, D = 1
                    eSeg = 0;
            if (~A & B & ~C) // A = 0, B = 1, C = 0
                    eSeg = 0;
            if (~B & ~C & D) // B = 0, C = 0, D = 1
                    eSeg = 0;
    end
endmodule
```

Iz priloženog Verilog programa je vidljivo kako postoje četiri jednobitna ulaza te jedan jednobitni izlaz. To znači da će tablica istinitosti za ovaj primjer biti veličine 16 kombinacija ulazno-izlaznih vrijednosti (što će ujedno biti i maksimalni iznos dobrote). Ispitni primjer je testiran s populacijom od 200 jedinki te brojem generacija 100 s ponavljanjem od deset puta (deset pokretanja s istim parametrima). Od funkcijskih čvorova izabrani su AND, NOT, OR, XOR i XNOR. Tokom ispitivanja su mijenjani razni parametri kako bi se uočio utjecaj pojedinih parametara na dobivanje rješenja. Upotrijebljeni parametri te dobiveni rezultati prikazani su na Slika 3.



Slika 3. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *binaryToESeg_Behavioral*

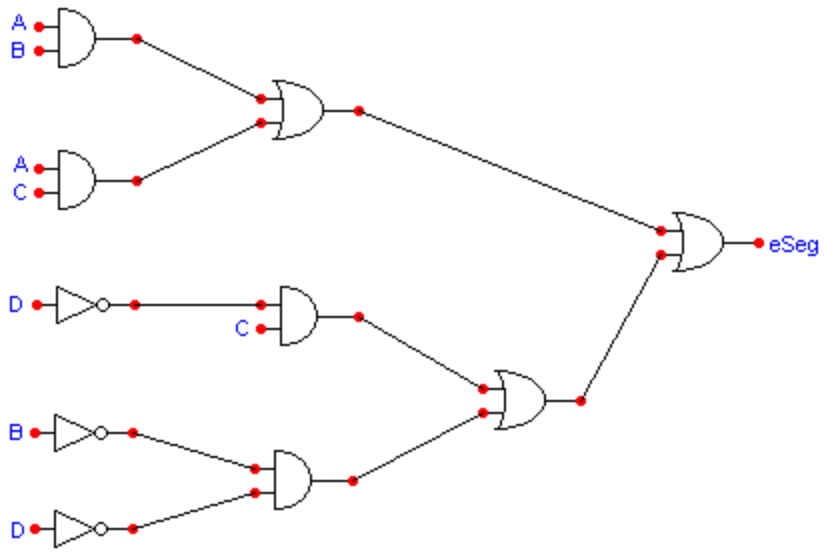
Kako se iz Slika 3 može vidjeti, korišteni parametri broja redaka i stupaca su u rasponu od 4 do 8, a parametar stupnja povratka ovisi naravno o broju stupaca (ne smije premašiti tu vrijednost). Ono što se htjelo ispitati jest kako se ponaša kartezijski genotip pri mijenjanju izgleda fenotipa na kvadratni oblik (broj redaka i broj stupaca su isti), oblik pravokutnika (broj redaka je manji od broja stupaca) te oblik inverznog pravokutnika (broj stupaca je manji od broja redaka). Također, htjelo se ispitati kako se u tim slučajevima ponaša parametar stupnja povratka ako se on postavi na najmanju vrijednost 1[13], na vrijednost polovine broja stupaca što iznosi $\frac{N}{2}$ ako je broj stupaca N ili na najveću vrijednost broja stupaca N. Ovaj način ispitivanja parametara će se odsad primijeniti i na sve ostale ispitne primjere.

Rezultati pokazuju da se maksimalna vrijednost dobrote dobija za dvije kombinacije parametara, jednom za kombinaciju (BR = 8, BS = 4, SP = 1) te čak tri puta za kombinaciju

[13] To znači da funkcijski čvorovi mogu biti povezani samo s čvorovima iz prethodnog stupca.

(BR = 8, BS = 8, SP = 1). Međutim, kombinacija parametara (BR = 8, BS = 4, SP = 1) pokazuje veliki odmak od svoje srednje vrijednosti za razliku od kombinacije parametara (BR = 8, BS =8, SP = 1). Razlog tome je što je prva kombinacija parametara svega jednom dobila rješenje s maksimalnom vrijednosti dobrote, a ostala dobivena rješenja su većinom dobrote iznosa 14, dok druga kombinacija tri puta dobiva rješenje s maksimalnom dobrotom, ali su joj sva ostala dobivena rješenja dobrote 15. Prema tome, druga kombinacija parametara pokazuje upola manje odstupanje i relativnu pogrešku rezultata u skupu od deset mogućih ispitivanja nad istim parametrima. Ovo je važno zapamtiti jer kada god se u daljnjem tekstu spomene veliki ili mali odmak maksimalne od srednje vrijednosti, podrazumijevat će se nešto slično kao što se dogodilo i u ovom slučaju (što se tiče pojedinačnih rezultata ispitivanja).

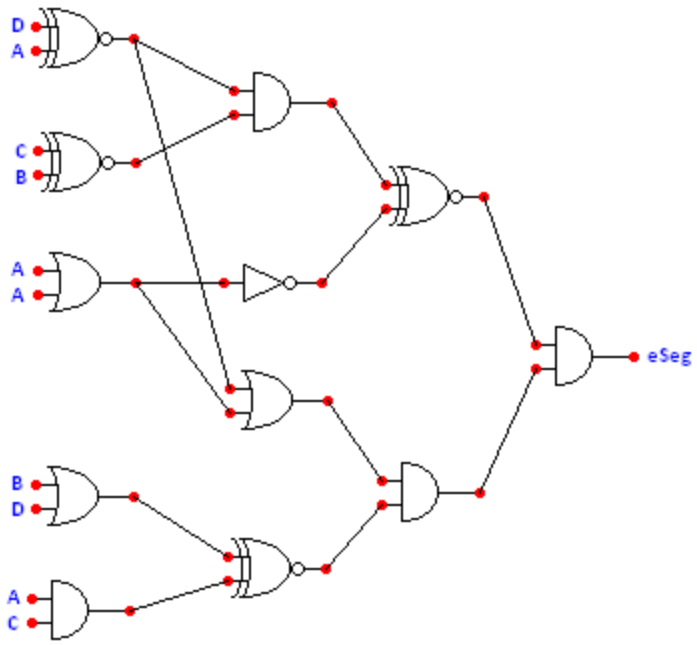
Neka se sada promotre strukture kombinacijskih mreža koje su dobivene od strane kartezijskog genotipa. Kako bih struktura kombinacijske mreže trebala izgledati za ovaj ispitni primjer predloženo od strane ljudskog eksperta na Slika 4.



Slika 4. Prijedlog strukture kombinacijske mreže za ispitni primjer *binaryToESeg_Behavioral* od strane ljudskog eksperta (preuzeto iz [26])

Kombinacija parametara (BR = 8, BS = 4, SP = 1) dobiva genotip veličine 32 funkcijska čvora od kojih je samo 12 aktivnih[14] (37.5% ih je aktivnih). S druge strane, kombinacija parametara (BR = 8, BS = 8, SP = 1) s genotipom veličine 64 funkcijska čvora za najmanju vrijednost aktivnih funkcijskih čvorova doseže 23 (35.9375% ih je aktivnih). Postotak aktivnosti je otprilike jednak, ali treba uzeti u obzir da druga kombinacija parametara daje puno veću veličinu genotipa.

Obzirom da cilj nije bio dobiti optimalne kombinacijske mreže nego ispravne (jer to omogućuje funkcija dobrote), kada bih se sastavile strukture kombinacijskih mreža iz obje kombinacije parametara te dodatno optimizirale, dobila bi se ista kombinacijska mreža kao na Slika 4. Ipak, prikaz strukture koje dobiva kombinacija parametara (BR = 8, BS = 4, SP = 1) čisto radi usporedbe prikazana je na Slika 5.



Slika 5. Struktura kombinacijske mreže za ispitni primjer *binaryToESeg_Behavioral* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 8, BS = 4, SP = 1)

Usporedbom kombinacijskih mreža na Slika 4 i Slika 5, zaključuje se kako je broj razina isti, a broj sklopova na Slika 5 za dva sklopa veći. Drugim riječima, ne postoji nekakva velika

[14] To znači da ostali funkcijski čvorovi nisu povezani s krajnjim izlazom i globalno ne utječu na izlaz iz genotipa.

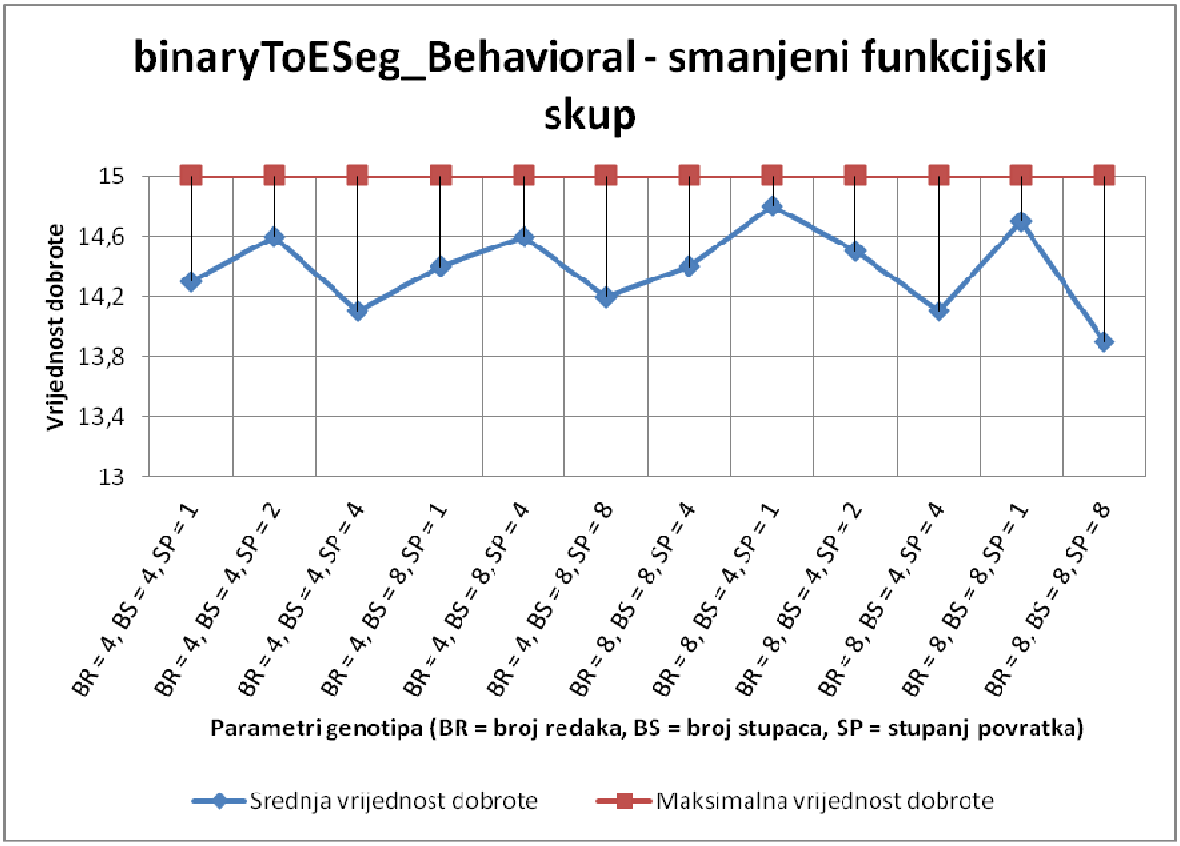
razlika, ali to možda daje naslutiti kako ne postoji bolje rješenje od onoga kojeg predlaže ljudski ekspert na Slika 4.

Ali, bolje rješenje možda ne postoji s trenutno odabranim parametrima. Važno je napomenuti kako su se u funkcijskom skupu koristili XOR i XNOR. Prema tome, možda bolje rješenje postoji, ali u prostoru rješenja gdje se u funkcijskom skupu nalaze samo oni sklopovi koje koristi i ljudski ekspert (vidi Slika 4), a to su AND, OR i NOT.

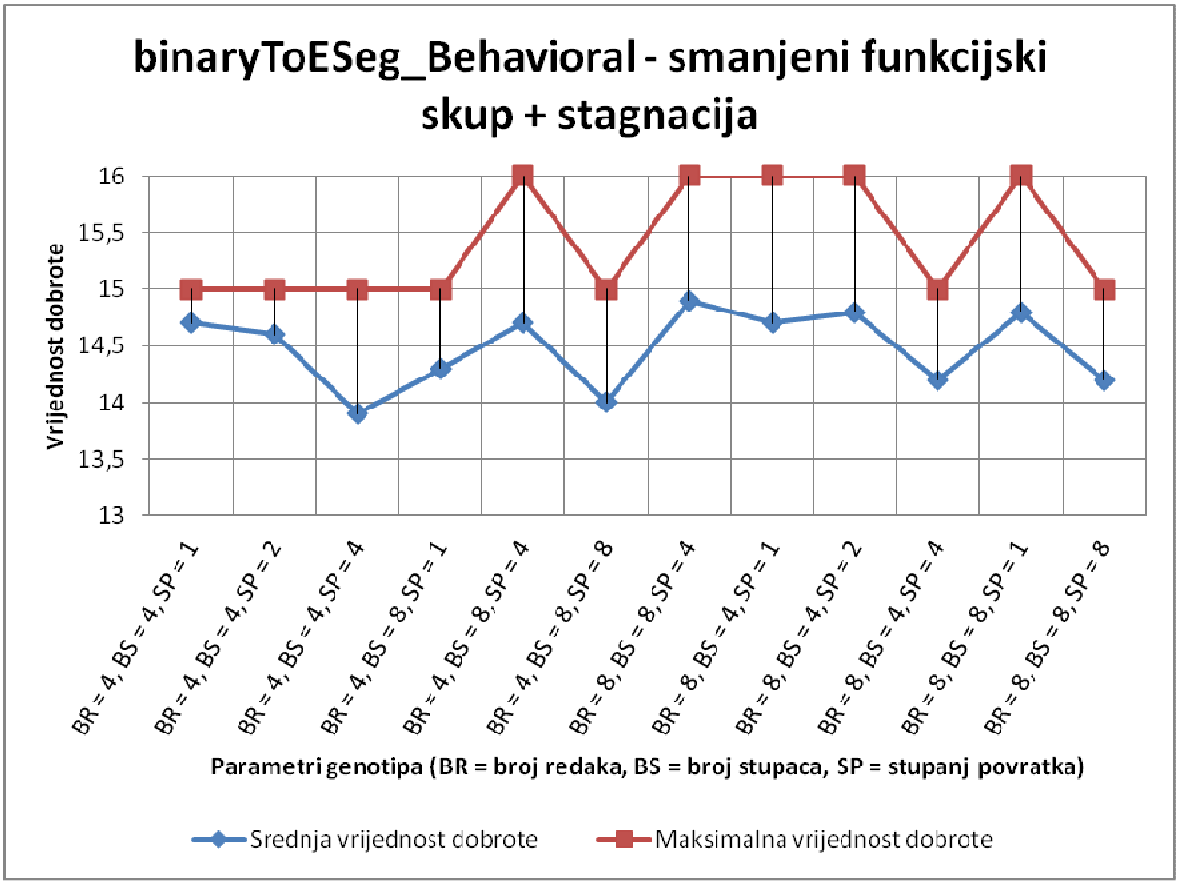
Na Slika 6 prikazani su rezultati ispitivanja gdje su se s obzirom na prethodno ispitivanje kao jedini parametri mijenjali funkcijski skup (sada je smanjen na AND, OR i NOT) te veličina populacije (sada je 100). Rezultati su dali naslutiti kako 100 generacija neće biti dovoljno da se pokuša naći rješenje. S obzirom na to, u idućem ispitivanju je umjesto broja generacija uvedena mogućnost stagnacije[15] od 200 generacija (ostali parametri ostaju isti kao i u prethodnom ispitivanju). Dobiveni rezultati su ovaj puta bili zadovoljavajući.

Za pet kombinacija parametara dobila su se rješenja maksimalne dobrote. To su redom (BR = 4, BS = 8, SP = 4), (BR = 8, BS = 8, SP = 4), (BR = 8, BS = 4, SP = 1), (BR = 8, BS = 4, SP = 2) i (BR = 8, BS = 8, SP = 1), kao što je uostalom i vidljivo na Slika 7. Od tih pet kombinacija parametara, tri su dale rješenja koja predstavljana strukturom mreža daju manje sklopova nego što to predlaže ljudski ekspert (vidi Slika 4). Kombinacija parametara (BR = 4, BS = 8, SP = 4) dobiva rješenje s 10 sklopova, zatim kombinacija (BR = 8, BS = 8, SP = 1) dobiva rješenje s 11 sklopova te kombinacija (BR = 8, BS = 4, SP = 2) dobiva rješenje s najmanjim brojem od 9 sklopova. Struktura mreže za zadnju kombinaciju prikazana je na Slika 8. Iz nje je vidljivo kako je u odnosu na rješenje kojeg predlaže ekspert (vidi Slika 4) broj sklopova smanjen za tri, a broj razina je ostao isti. Drugim riječima, dobiveno je bolje rješenje od ekspertovog (točnije, tri bolja rješenja, ali samo je ovo prikazano).

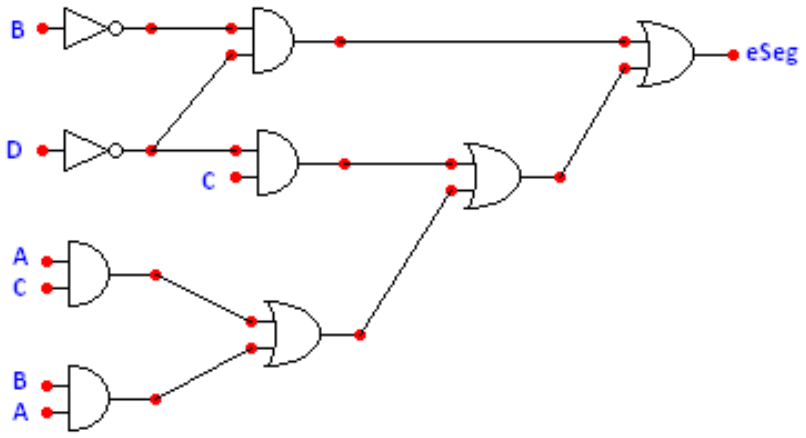
[15] Stagnacija podrazumijeva da se rješenje traži sve dok dobrota jedinki u populaciji ne počne stagnirati kroz određeni broj generacija (taj broj generacija se može zadati kao parametar) [22].



Slika 6. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *binaryToESeg_Behavioral* u slučaju smanjenog funkcijskog skupa (AND, OR, NOT)



Slika 7. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *binaryToESeg_Behavioral* u slučaju smanjenog funkcijskog skupa (AND, OR, NOT) i praćenja stagnacije



Slika 8. Struktura kombinacijske mreže za ispitni primjer *binaryToESeg_Behavioral* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 8, BS = 4, SP = 2) i smanjeni funkcijski skup

Iz kombinacija parametara koje dobivaju maksimalna moguća rješenja (za sva tri provedena ispitivanja nad ovim ispitnim primjerom) moguće je zaključiti kako je poželjno

ostaviti veći broj redaka od broja stupaca (oblik inverznog pravokutnika) te sa stupnjem povratka omogućiti da čvorovi budu povezani samo s prethodnicima ili do veličine polovine broja stupaca u genotipu. Ako se promotri rješenje koje predlaže ljudski ekspert na Slika 4, broj sklopova u prvoj razini jest veći od ukupnog broja razina te su svi sklopovi međusobno povezani na način da postoji povezanost samo između susjednih razina. Prema tome, možda je struktura rješenja uistinu utjecala na odabir parametara.

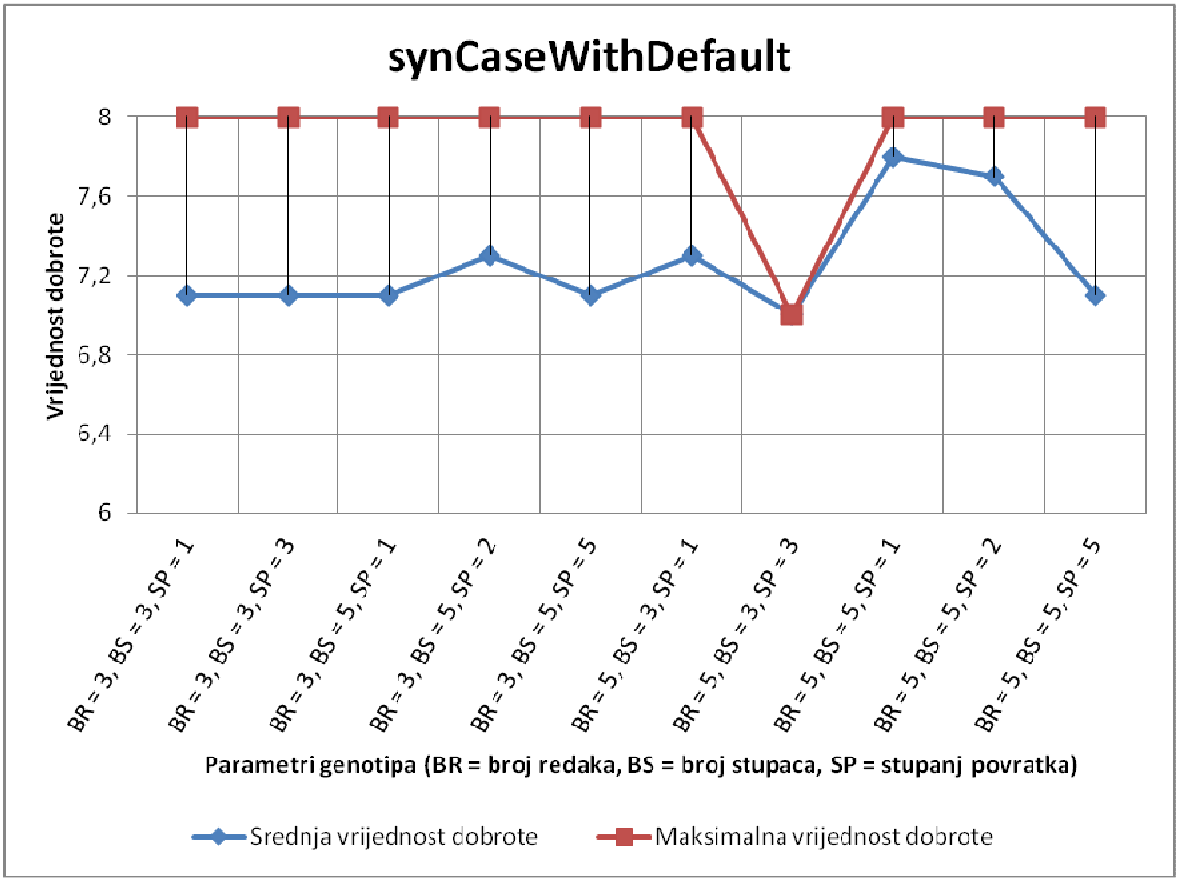
3.3.2. Ispitni primjer: *synCaseWithDefault*

Ispitni primjer *synCaseWithDefault* (preuzet iz [26]) opisuje jednostavnu uporabu `case` konstrukta iz Veriloga:

```
module synCaseWithDefault (f, a, b, c);

    output          f;

    input           a, b, c;

    reg             f;

    always @(a or b or c)

            // jednobitni ulazi se zajednički mogu promatrati kao

            // trobitna kombinacija ulaza

            case ({a, b, c})

                    3'b000:         f = 1'b0;

                    3'b101:         f = 1'b0;

                    3'b110:         f = 1'b0;

                    default:        f = 1'b1;

            endcase

endmodule
```

Iz priloženog je vidljivo kako se koriste tri jednobitna ulaza te jedan jednobitni izlaz. Tablica istinitosti će za ovaj ispitni primjer biti veličine 8 pa će i maksimalno moguća vrijednost dobrote biti 8. Prilikom ispitivanja, odabrana je veličina populacije od 50 jedinki te broj generacija 100. Korišteni funkcijski čvorovi su bili AND, NOT, OR, XOR i XNOR. Slično kao i za prvi ispitni primjer, ispitivao se utjecaj različitih parametara (broj redaka,

broj stupaca i parametar stupnja povratka) na dobivanje rješenja. Sa svakom kombinacijom parametara prošlo se kroz ispitni primjer deset puta (deset pokretanja po kombinaciji parametara). Upotrijebljeni parametri te dobiveni rezultati prikazani su na Slika 9.



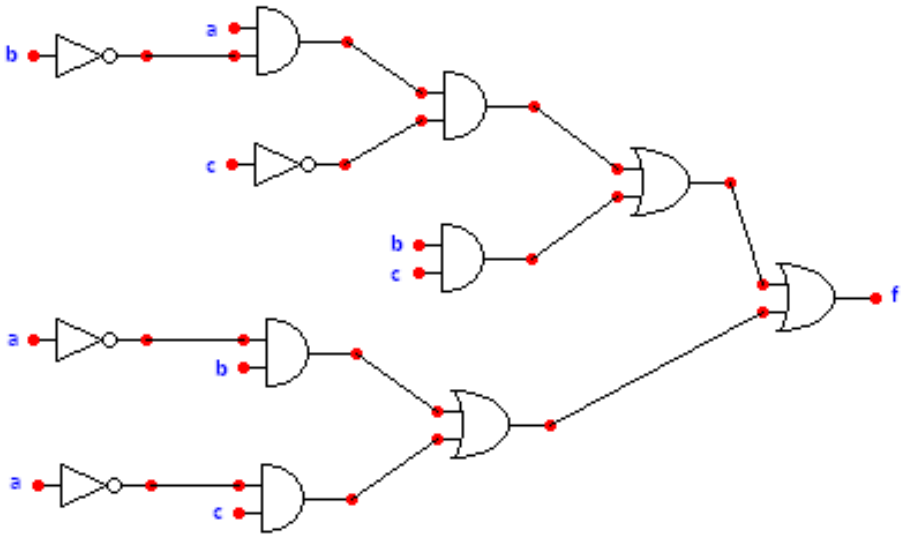
Slika 9. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *synCaseWithDefault*

Kako je vidljivo na Slika 9, postoji niz kombinacija parametara s kojima se može doći do ispravnih rješenja (rješenja koja imaju maksimalnu vrijednost dobrote). Međutim, iz toga se ne može zaključiti koja je kombinacija parametara najbolji izbor.

Kombinacija parametara	Broj ispravnih rješenja	Broj logičkih sklopova
BR = 3, BS = 3, SP = 1	1	6
BR = 3, BS = 3, SP = 3	1	4
BR = 3, BS = 5, SP = 1	3	10, 11, (> 12)
BR = 3, BS = 5, SP = 2	3	5, 2x8
BR = 3, BS = 5, SP = 5	1	4
BR = 5, BS = 3, SP = 1	3	6, 2x7
BR = 5, BS = 5, SP = 1	8	3x10, 2x12, 3x(> 12)
BR = 5, BS = 5, SP = 2	8	4, 2x5, 2x7, 8, 2x9
BR = 5, BS = 5, SP = 5	1	5

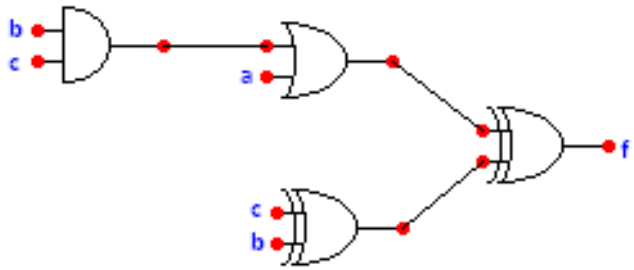
Tablica 1. Dobiven broj ispravnih rješenja za pojedine kombinacije parametara ispitnog primjera *synCaseWithDefault* te broj logičkih sklopova u samom prikazu tih rješenja kombinacijskom mrežom

Iz Tablica 1 vidljivo je kako se genotip najbolje snalazi s kombinacijama parametara (BR = 5, BS = 5, SP = 1) te (BR = 5, BS = 5, SP = 2) jer za njih daje najveći broj ispravnih rješenja (čak 8 od mogućih 10 pokretanja ispitnog primjera), a te kombinacije parametara s druge strane dobivaju najmanje odstupanje od srednje vrijednosti dobrote (vidi Slika 9). Međutim, neka se promotri struktura kombinacijske mreže (vidi Slika 10) koju predlaže ljudski ekspert za dotični ispitni primjer. Vidljivo je kako je kombinacijska mreža sastavljena od 12 logičkih sklopova te ima 5 razina.

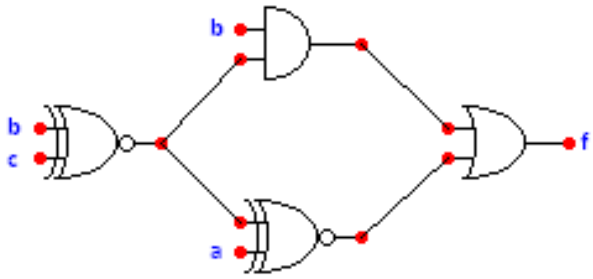


Slika 10. Prijedlog strukture kombinacijske mreže za ispitni primjer *synCaseWithDefault* od strane ljudskog eksperta (samostalno sastavljeno na temelju minimizacije tablice istinitosti)

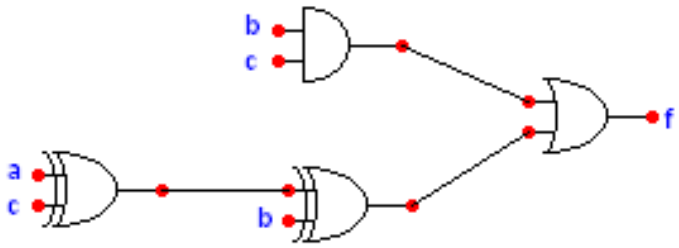
Promatranjem rješenja za strukturu kombinacijske mreže za različite kombinacije parametara (vidi Tablica 1), rezultati pokazuju kako kombinacija parametara (BR = 5, BS = 5, SP = 1) daje rješenja s više od 12 sklopova dok kombinacija parametara (BR = 5, BS = 5, SP = 2) daje rješenja s manje od 12 sklopova. Međutim, kombinacije parametara (BR = 3, BS = 3, SP = 3), (BR = 3, BS = 5, BS = 5) te već spomenuta (BR = 5, BS = 5, SP = 2) daju rješenja s najmanje upotrijebljenih sklopova da bi se dobilo ispravno rješenje tj. koriste samo 4 sklopa. Ta rješenja se mogu vidjeti na Slika 11, Slika 12 i Slika 13.



Slika 11. Struktura kombinacijske mreže za ispitni primjer *synCaseWithDefault* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 3, BS = 3, SP = 3)



Slika 12. Struktura kombinacijske mreže za ispitni primjer *synCaseWithDefault* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 3, BS = 5, SP = 5)



Slika 13. Struktura kombinacijske mreže za ispitni primjer *synCaseWithDefault* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 5, BS = 5, SP = 2)

Prema tome, moguće je dobiti bolja rješenja od rješenja koje predlaže ljudski ekspert. Rješenja na Slika 11, Slika 12 i Slika 13 imaju 8 sklopova manje te 2 razine manje od rješenja kojeg predlaže ljudski ekspert (vidi Slika 10) što predstavlja značajno poboljšanje. Zapravo, čak 25 od 29 dobivenih ispravnih rješenja (vidi Tablica 1) ima manje sklopova (a vjerojatno time i nešto manje razina) od rješenja kojeg predlaže ljudski ekspert.

U ovom ispitnom primjeru se podešavanjem parametara dobilo bolje rješenje od predloženog pa je očito postojala bolja struktura kombinacijske mreže ukoliko se uvelo korištenje XOR i XNOR sklopova.

Ipak, bilo bi korisno promotriti što će se dogoditi ukoliko se funkcijski skup smanji samo na AND, OR i NOT sklopove (isto kao i u prethodnom ispitnom primjeru). U skladu s time, provela su se dodatna dva ispitivanja gdje su se malo promijenili parametri prethodnog ispitivanja. U prvom se smanjio samo funkcijski skup i njegovi rezultati prikazani su na Slika 14. U drugom se osim smanjivanja funkcijskog skupa umjesto broja generacija postavila stagnacija od 200 generacija. Njegovi rezultati prikazani su na Slika 15.

Ako se pogledaju oba ispitivanja, vidljivo je kako se sa smanjenim funkcijskim skupom dobiva puno manje rješenja s maksimalnog dobrotom za razliku od prethodnog ispitivanja.

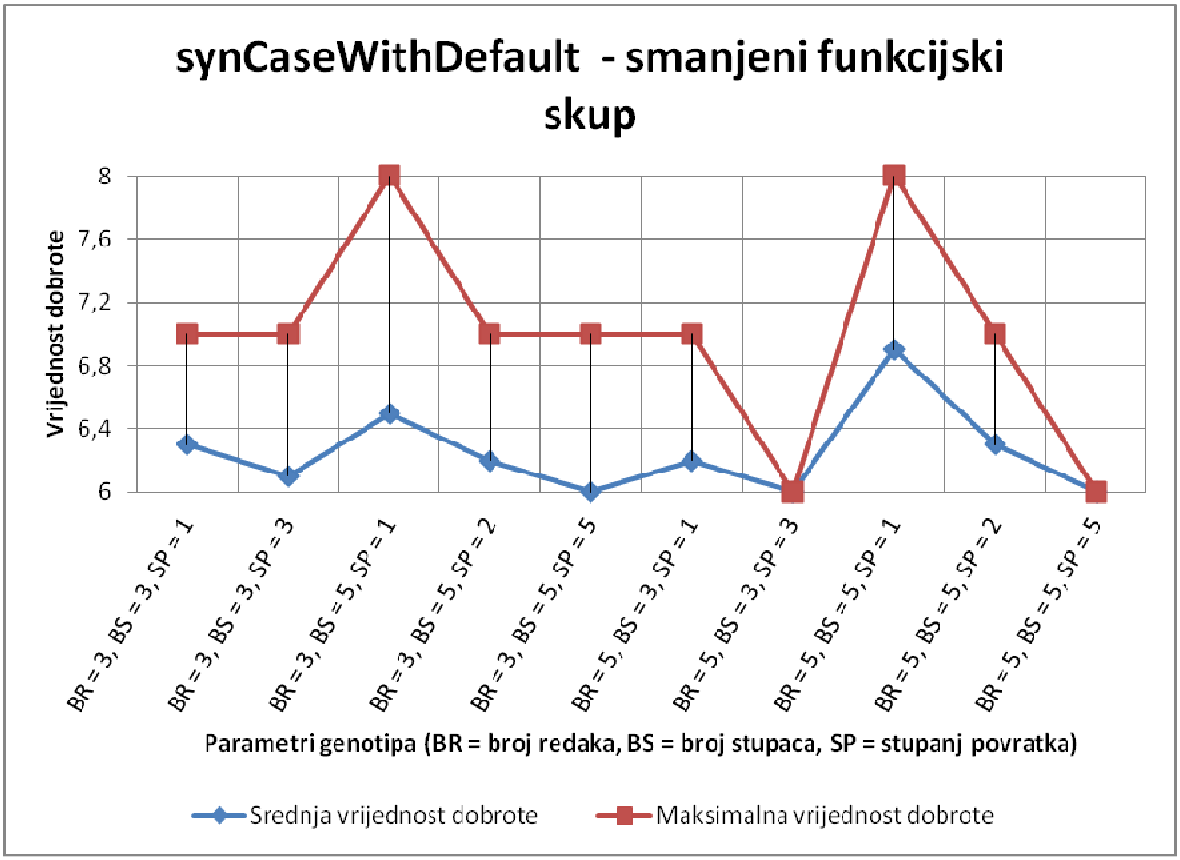
Prvo ispitivanje (vidi Slika 14) dobiva tri rješenja s maksimalnom dobrotom, jedno za kombinaciju parametara (BR = 3, BS = 5, SP = 1) koje se sastoji od 12 sklopova te dva za kombinaciju parametara (BR = 5, BS = 5, SP = 1) od kojih se jedno sastoji od 14, a drugo od

15 sklopova. Drugo ispitivanje (vidi Slika 15) dobiva dva rješenja s maksimalnom dobrotom, jedno za kombinaciju parametara (BR = 5, BS = 5, SP = 1) koje se sastoji od 11 sklopova te jedno za kombinaciju parametara (BR = 5, BS = 5, SP = 2) koje se sastoji od 9 sklopova.

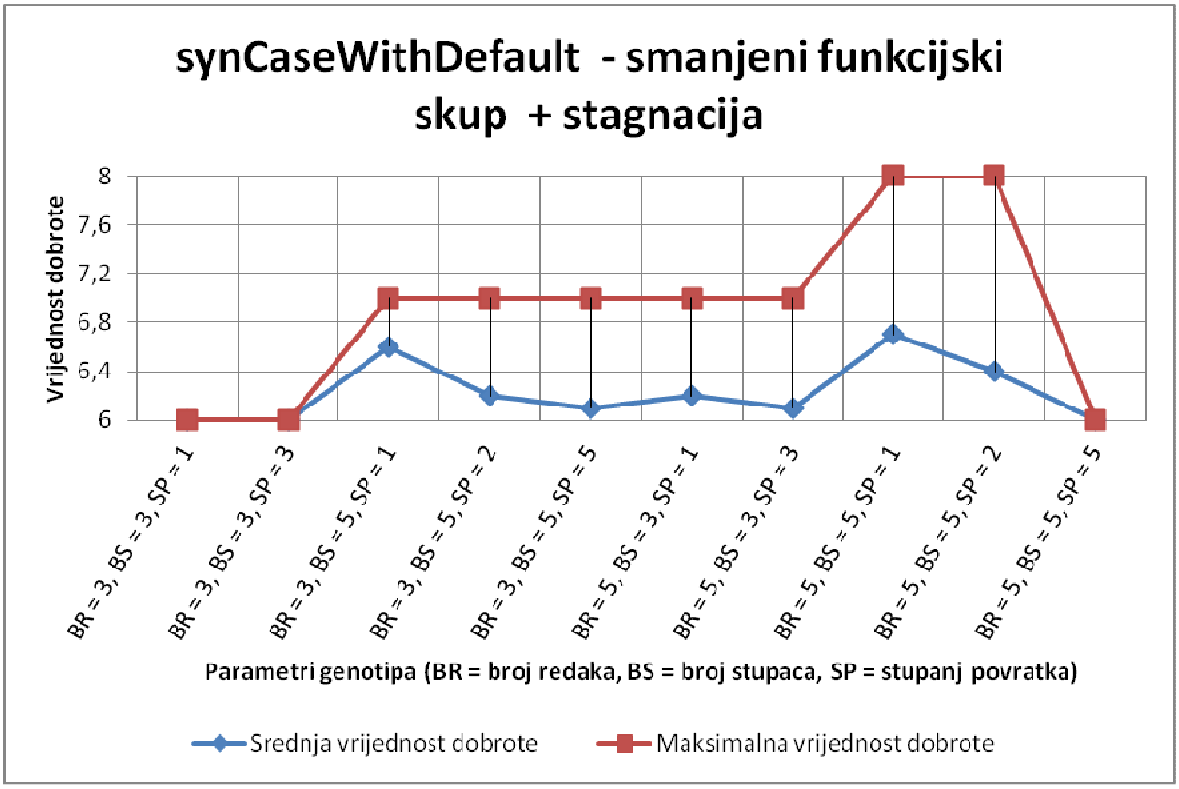
Sveukupno, dobiveno je pet rješenja od kojih jedno ima jednak broj, a dvoje manji broj sklopova od rješenja kojeg predlaže ljudski ekspert (vidi Slika 10). Odabrano najbolje rješenje (koje se sastoji od samo 9 sklopova) prikazano je na Slika 16.

Dobivena rješenja pokazuju kako je korištenjem XOR i XNOR sklopova u funkcijskom skupu moguće dobiti više reprezentacija rješenja koja su jednakovrijedne što se tiče vrijednosti dobrote.

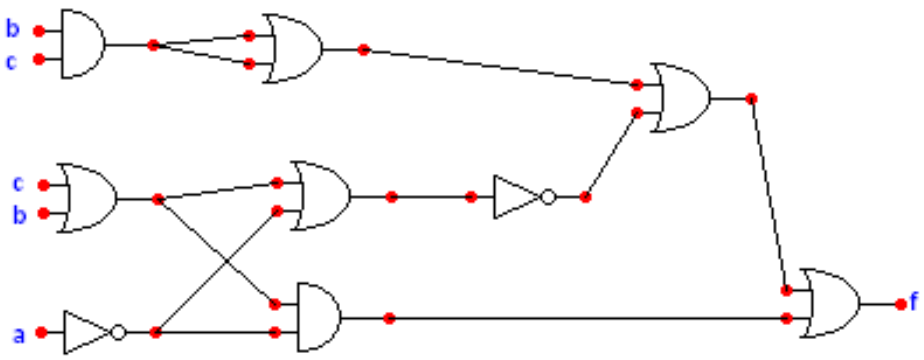
Što se pak tiče odabira parametara u sva tri ispitivanja, kombinacije parametara (BR = 5, BS = 5, SP = 1) te (BR = 5, BS = 5, SP = 1) pokazale najpovoljnijima po pitanju dobivanja ispravnih rješenja. Ako se pogleda rješenje koje predlaže ljudski ekspert (vidi Slika 10), kvadratni broj stupaca i broj redaka se može poistovjetiti sa strukturom rješenja, a isto tako i parametar stupnja povratka za kojeg je poželjno da bude što manji (jer su svi sklopovi u rješenju povezani samo s prethodnicima).



Slika 14. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *synCaseWithDefault* u slučaju smanjenog funkcijskog skupa (AND, OR, NOT)



Slika 15. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *synCaseWithDefault* u slučaju smanjenog funkcijskog skupa (AND, OR, NOT) i praćenja stagnacije



Slika 16. Struktura kombinacijske mreže za ispitni primjer *synCaseWithDefault* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 5, BS = 5, SP = 2) i smanjeni funkcijski skup

3.4. Druga ispitna skupina: primjeri s više izlaza

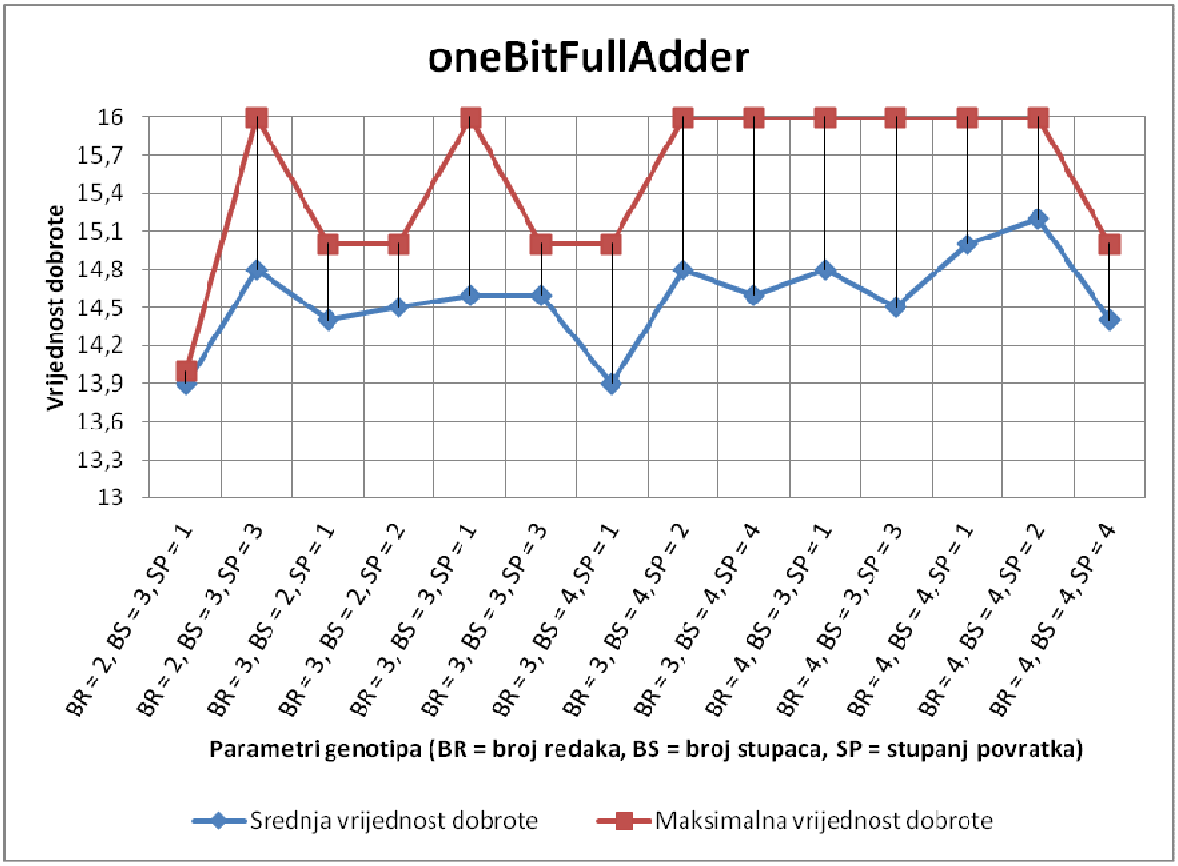
3.4.1. Treći ispitni primjer: *oneBitFullAdder*

Ispitni primjer *oneBitFullAdder* (preuzet iz [26]) opisuje jednobitno zbrajanje s prijenosom pri čemu se detektira rezultat zbrajanja i izlazni prijenos:

```
module oneBitFullAdder(cOut, sum, aIn, bIn, cIn);

    output          cOut, sum;

    input           aIn, bIn, cIn;

    assign          sum = aIn ^ bIn ^ cIn,

                    cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);

endmodule
```

Koriste se tri jednobitna ulaza te dva jednobitna izlaza. Tablica istinitosti će za dotični ispitni primjer biti veličine 16 ulazno-izlaznih kombinacija (s time će i maksimalna moguća dobrota biti 16). Ispitni primjer se pokretao s parametrima od 50 jedinki u populaciji te 100 generacija. Korišteni funkcijski čvorovi su AND, NOT, OR, XOR i XNOR. Za svaku kombinaciju parametara broja redaka, broja stupaca i parametra stupnja povratka koristilo se deset ispitivanja ispitnog primjera (pokretao se deset puta po kombinaciji). Slično kao i za prethodnu ispitnu skupinu, pokušalo se pronaći kombinaciju parametara koja učinkovito pronalazi ispravna rješenja. Odabrane kombinacije parametara te dobiveni rezultati prikazani su na Slika 17.

Postoji više kombinacija parametara za koje se dobivaju ispravna rješenje kao što je prikazano na Tablica 2. Uspoređujući rezultate s Tablica 2 i Slika 17, vidljivo je kako najmanje odstupanje od srednje vrijednosti pokazuju kombinacije parametara (BR = 4, BS = 4, SP = 1) i (BR = 4, BS = 4, SP = 2), a uz to za tri (za prvu kombinaciju), odnosno četiri (za drugu kombinaciju) od mogućih pet ispitivanja dobivaju ispravna rješenja.



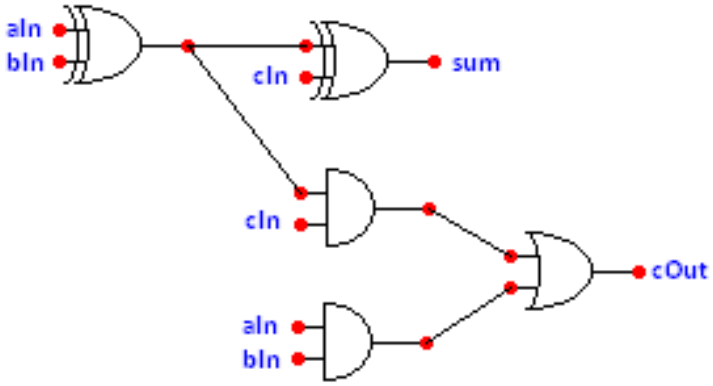
Slika 17. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *oneBitFullAdder*

Kombinacija parametara	Broj ispravnih rješenja	Broj logičkih sklopova
BR = 2, BS = 3, SP = 3	1	5
BR = 3, BS = 3, SP = 1	2	2x(> 5)
BR = 3, BS = 4, SP = 2	3	3x(> 5)
BR = 3, BS = 4, SP = 4	1	(> 5)
BR = 4, BS = 3, SP = 1	2	2x(> 5)
BR = 4, BS = 3, SP = 3	1	5
BR = 4, BS = 4, SP = 1	3	3x(> 5)
BR = 4, BS = 4, SP = 2	4	4x(> 5)

Tablica 2. Dobiven broj ispravnih rješenja za pojedine kombinacije parametara ispitnog primjera *oneBitFullAdder* te broj logičkih sklopova u samom prikazu tih rješenja kombinacijskom mrežom

Međutim, valja ponovno promotriti strukturu kombinacijske mreže kakvu predlaže ljudski ekspert (vidi Slika 18) te strukture koje se dobivaju kao rješenja. Ljudski ekspert predlaže rješenje koje se sastoji od 5 sklopova te ima 3 razine. Za većinu kombinacija parametara dobivena su rješenja koja imaju više od 5 sklopova (vidi Tablica 2), međutim za

kombinacije parametara (BR = 2, BS = 3, SP = 3) i (BR = 4, BS = 3, SP = 3) se dobivaju rješenja koja se također sastoje od 5 sklopova. Ako se iz tih rješenja sastavi struktura kombinacijske mreže, dobiva se ista struktura kao i na Slika 18, odnosno ista kakvu predlaže ljudski ekspert.



Slika 18. Prijedlog strukture kombinacijske mreže za ispitni primjer *oneBitFullAdder* od strane ljudskog eksperta (preuzeto iz [4])

Iz kombinacije parametara (BR = 2, BS = 3, SP = 3) i (BR = 4, BS = 3, SP = 3) koja dobivaju rješenje istovjetno rješenju kojeg predlaže ljudski ekspert te promatranjem strukture rješenja tog predloženog rješenja (vidi Slika 18), vidljivo je kako je struktura gotovo kvadratnog oblika (tri su razine i u drugoj razini je broj sklopova tri) i signal nije ravnomjerno raspoređen razinu po razinu kao što je to bio slučaj s prvim ispitnim primjerom (vidi Slika 4). Odabrani broj redaka i stupaca se uistinu kreće oko kvadratnog oblika, a zanimljivo je kako je za parametar stupnja povratka odabrani maksimalni broj (jednak broju stupaca) bio taj koji će dovesti do dobrog rješenja.

I u ovom slučaju dobivena su dobra rješenja, kako po iznosu dobrote, tako i po samoj strukturi dobivene kombinacijske mreže. Međutim, za razliku od prva dva ispitna primjera, nije pronađeno rješenje koje je bolje od rješenja koje predlaže ljudski ekspert pa sukladno tome ono možda i ne postoji[16].

[16] Uz korištene funkcijske čvorove (AND, NOT, OR, XOR, XNOR) ne postoji bolje rješenje od predloženog [4].

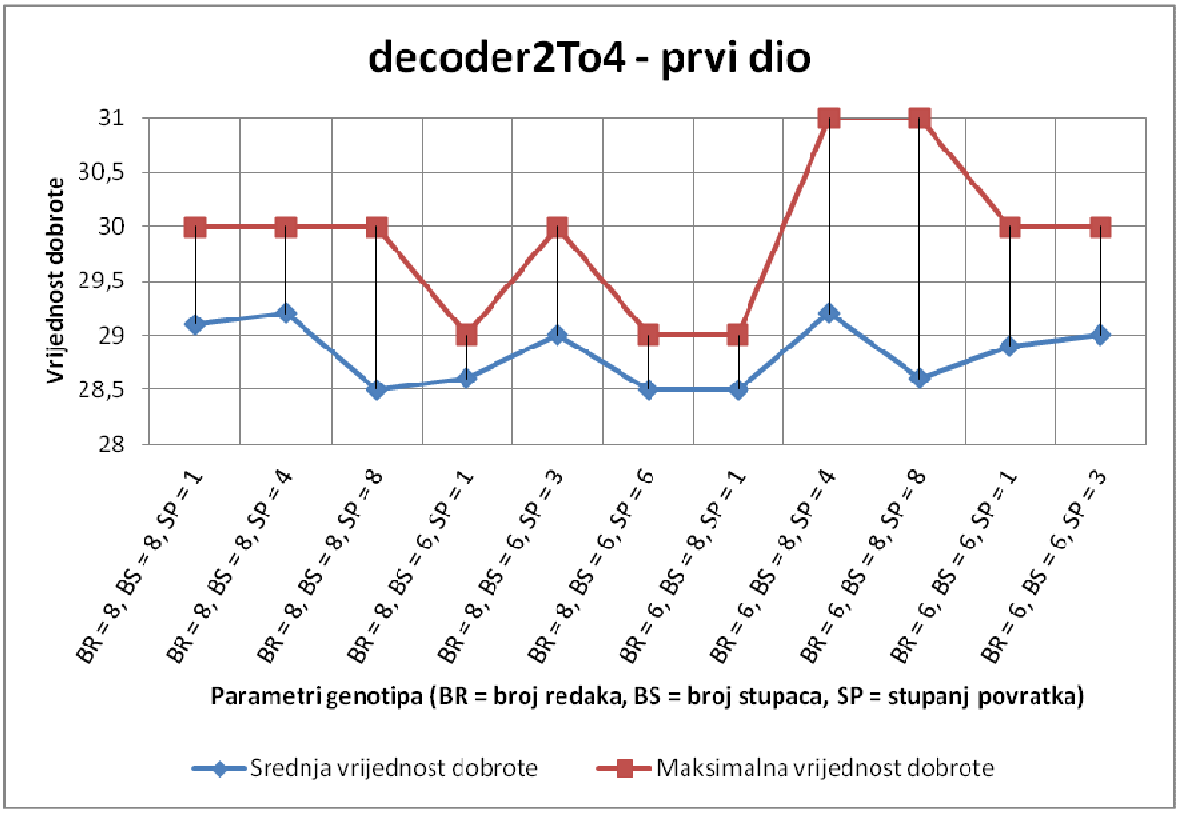
3.4.2. Četvrti ispitni primjer: *decoder2To4*

Ispitni primjer *decoder2To4* (preuzet iz [36]) opisuje dekodiranje dvije ulazne vrijednosti na četiri izlazne vrijednosti koristeći signal omogućivanja (eng. *enable*):

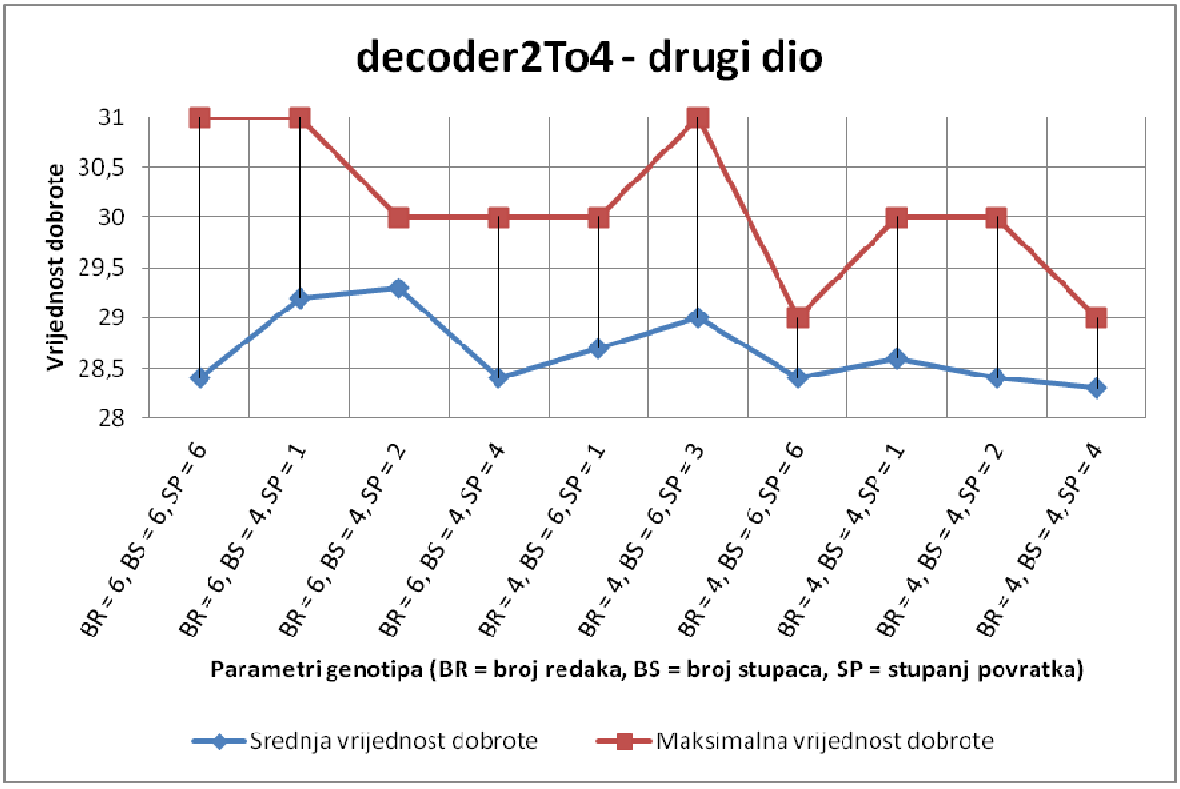
```
module decoder2To4(Y3, Y2, Y1, Y0, A, B, en);
     output Y3, Y2, Y1, Y0;
     input  A, B;
     input  en;
     reg    Y3, Y2, Y1, Y0;
    // dekoder s aktivnim visokim enable signalom te aktivnim
    // niskim izlazima
     always @(A or B or en) begin
       if (en == 1'b1) // omogućeno kad je enable = 1
         case ( {A,B} )
           // aktivan izlaz je onaj na kojem je nula
           2'b00:   {Y3,Y2,Y1,Y0} = 4'b1110;
           2'b01:   {Y3,Y2,Y1,Y0} = 4'b1101;
           2'b10:   {Y3,Y2,Y1,Y0} = 4'b1011;
           2'b11:   {Y3,Y2,Y1,Y0} = 4'b0111;
           default: {Y3,Y2,Y1,Y0} = 4'bxxxx;
         endcase
       if (en == 0)
         {Y3,Y2,Y1,Y0} = 4'b1111;
     end
endmodule
```

Ispitni primjer koristi dva jednobitna ulaza i četiri jednobitna izlaza. Tablica istinitosti će za ovaj ispitni primjer biti veličine 32 ulazno-izlaznih kombinacija što će ujedno biti i vrijednost maksimalne moguće dobrote. Ispitni primjer se pokretao s parametrima od 50 jedinki u populaciji te 100 generacija. Korišteni funkcijski čvorovi su AND, NOT, OR, XOR i

XNOR. Za svaku kombinaciju parametara broja redaka, broja stupaca i parametra stupnja povratka bilo je deset ispitivanja ispitnog primjera (pokretao se deset puta po kombinaciji). Slično kao i za prethodnu ispitnu skupinu, pokušalo se pronaći kombinaciju parametara koja učinkovito pronalazi ispravna rješenja. Odabrane kombinacije parametara te dobiveni rezultati prikazani su na Slika 19 i Slika 20.



Slika 19. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *decoder2To4* (prvi dio)

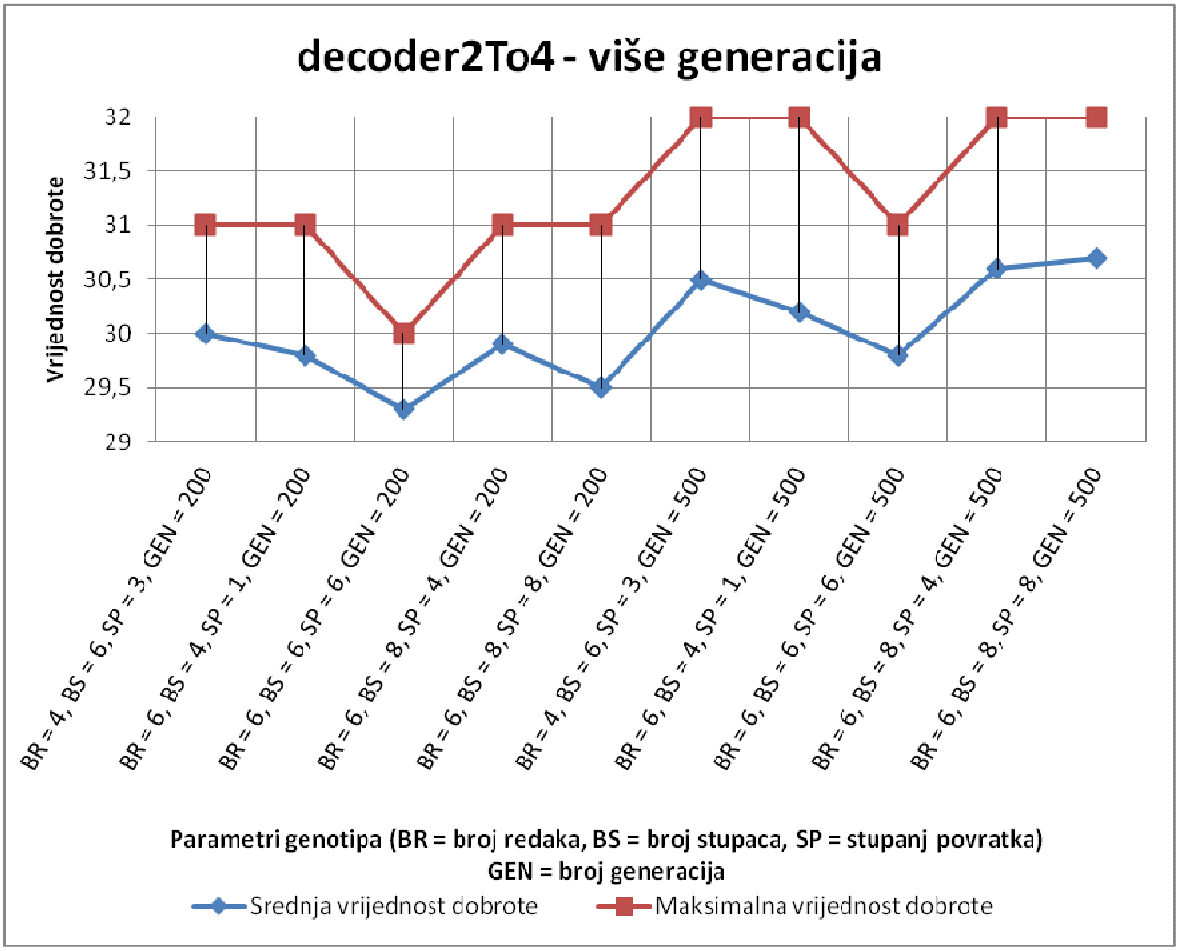


Slika 20. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *decoder2To4* (drugi dio)

Kao što je vidljivo sa Slika 19 i Slika 20, među dobivenim maksimalnim vrijednostima dobrote ne nalazi se nijedna vrijednost koja odgovara vrijednosti maksimalno moguće dobrote. Drugim riječima, ne dobiva se ispravno rješenje nego djelomično ispravno rješenje koje pokriva do 31 ulazno-izlaznih kombinacija tablice istinitosti, ali ne i sve 32 kombinacije. Među odabranim kombinacijama parametara, ne postoji nijedna kombinacija koja bi navela algoritam na dolazak do ispravnog rješenja.

Provedena su dodatna dva ispitivanja gdje se mijenjalo prvotne parametre na način da se u prvom ispitivanju za broj generacija uzelo vrijednost od 200 generacija, a za drugo ispitivanje vrijednost od 500 generacija. Nisu uzete sve kombinacije parametara u obzir, nego samo one s kojima su se u prethodnom ispitivanju (vidi Slika 19 i Slika 20) dobila rješenja s najvećom dobrotom (vrijednost 31). Dobiveni rezultati prikazani su na Slika 21 gdje je vidljivo kako je ispravno rješenje s maksimalno mogućom dobrotom ipak moguće dobiti, ali samo ako se za broj generacija uzme broj veći od 200. Ispravna rješenja dobivena su za kombinaciju parametara (BR = 4, BS = 6, SP = 3, GEN = 500) koja daje

rješenje od 14 sklopova, zatim kombinaciju parametara (BR = 6, BS = 4, SP = 1, GEN = 500) koja daje rješenje od 20 sklopova, kombinaciju parametara (BR = 6, BS = 8, SP = 4, GEN = 500) s jednim rješenjem od 24 sklopa i jednim od 21 sklop te kombinaciju parametara (BR = 6, BS = 8, SP = 8, GEN = 500) s rješenjem od 15 sklopova.



Slika 21. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *decoder2To4* uz povećan broj generacija

Neka se promotri struktura kombinacijske mreže koju predlaže ljudski ekspert (vidi Slika 22). Vidljivo je kako u njoj postoji skup nezavisnih sklopova te skup zavisnih sklopova, gledano s obzirom na utjecaj na krajnje izlaze. Sklopova ima sveukupno 11, a broj razina 3. Sudeći po prethodnim rezultatima, nijedno dobiveno rješenje nema manji ili jednak broj sklopova kao rješenje predloženo od strane eksperta.

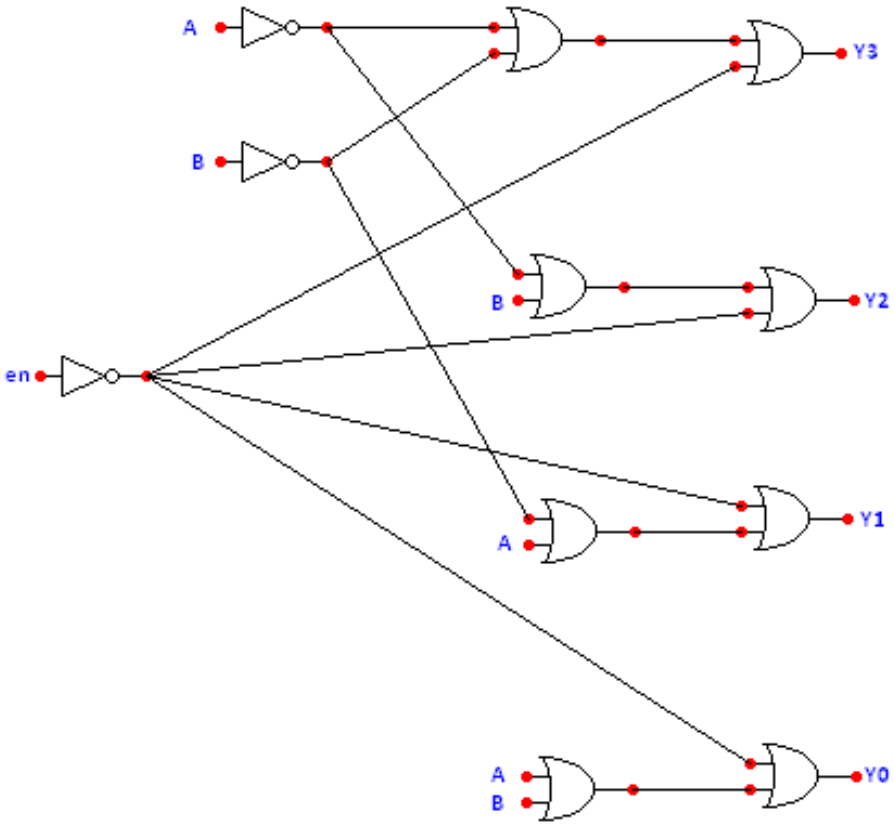
Provedena su naredna dva ispitivanja s istim kombinacijama parametara. U prvom se koristila stagnacija od 200 generacija, a u drugom osim stagnacije smanjeni funkcijski skup (AND, OR i NOT). Dobiveni rezultati prikazani su na Slika 24.

U prvom ispitivanju (samo stagnacija) dobivena su ispravna rješenja s maksimalnom dobrotom za kombinaciju parametara (BR = 4, BS = 6, SP = 3, SF) s jednim rješenjem od 16 sklopova i jednim od 21 sklop te za kombinaciju parametara (BR = 6, BS = 4, SP = 1, SF) s rješenjem od 21 sklop. Zatim, za kombinaciju parametara (BR = 6, BS = 8, SP = 4, SF) s jednim rješenjem od 21 sklop te drugim od 24 sklopa te za kombinaciju parametara (BR = 6, BS = 8, SP = 8, SF) s rješenjem od 14 sklopova.

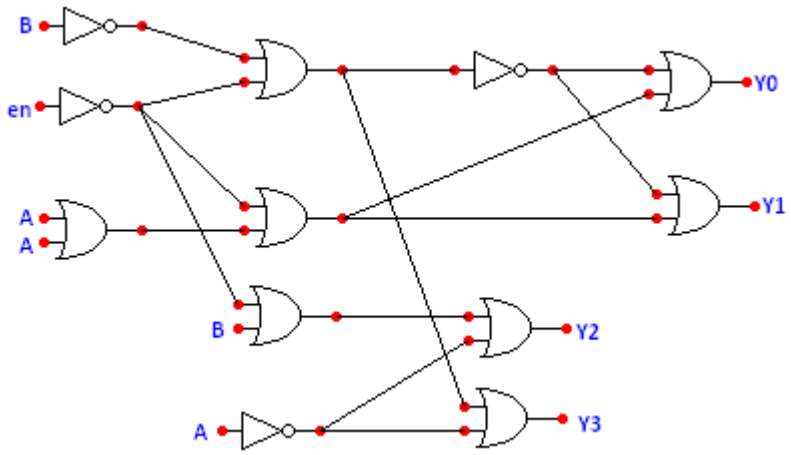
U drugom ispitivanju (stagnacija i smanjeni funkcijski skup) dobivena su ispravna rješenja s maksimalnom dobrotom za kombinaciju parametara (BR = 4, BS = 6, SP = 3, NF) s jednim rješenjem od 16 sklopova i jednim od 12 sklopova te kombinaciju parametara (BR = 6, BS = 4, SP = 1, NF) s rješenjem od 17 sklopova. Zatim, za kombinaciju parametara (BR = 6, BS = 6, SP = 6, NF) s rješenjem od 13 sklopova, kombinaciju parametara (BR = 6, BS = 8, SP = 4, NF) s jednim rješenjem od 18 , jednim od 30 i jednim od 20 sklopova te za kombinaciju parametara (BR = 6, BS = 8, SP = 8, NF) s rješenjem od 14 sklopova.

Uzevši u obzir ta dodatna četiri ispitivanja, rješenje koje je dovoljno blizu rješenju eksperta (po broju sklopova i razina) jest rješenje kojeg daje kombinacija parametara (BR = 4, BS = 6, SP = 3, NF) prikazano na Slika 23. Za razliku od rješenja kojeg predlaže ljudski ekspert (vidi Slika 22), ta struktura ima jedna sklop i jednu razinu više, ali je u potpunosti ispravna.

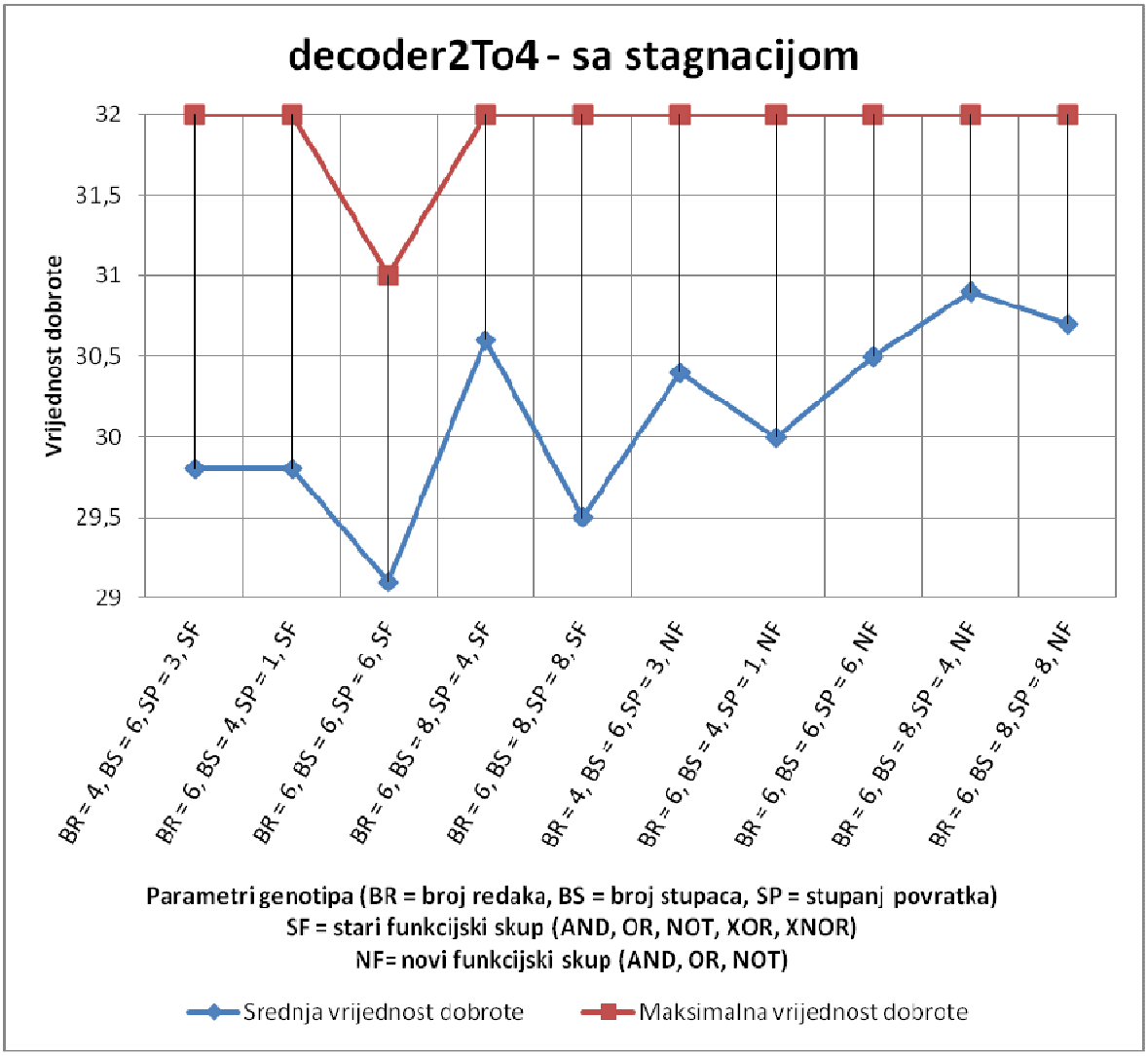
Prema tome, dodatnim ispitivanjima je utvrđeno kako je i za ovaj ispitni primjer moguće pronaći ispravna rješenja, ali i dovoljno bliska rješenju eksperta. Što se tiče kombinacija parametara koje su davale bolja rješenja (s većom dobrotom), u pravilu su parametri vrlo raznoliki i ne poklapaju se baš sa strukturom koju predlaže ekspert.



Slika 22. Prijedlog strukture kombinacijske mreže za ispitni primjer *decoder2To4* od strane ljudskog eksperta (samostalno sastavljeno na temelju minimizacije tablice istinitosti)



Slika 23. Struktura kombinacijske mreže za ispitni primjer *decoder2To4* dobivena kartezijskim genotipom za kombinaciju parametara (BR = 4, BS = 6, SP = 3, NF)



Slika 24. Dobiveni rezultati ispitivanja te upotrijebljeni parametri ispitivanja za ispitni primjer *decoder2To4* uz korištenje stagnacije i smanjenog funkcijskog skupa

3.5. Analiza rezultata

Na temelju ispitana četiri ispitna primjera podijeljena u dvije ispitne skupine moguće je djelomično odgovoriti na prvo ispitno pitanje. Naime, odabrani parametri uvelike ovise o strukturi kombinacijske mreže ukoliko je ljudski ekspert predložio kombinacijsku mrežu za koju se ne može pronaći bolja struktura u odnosu na odabrane funkcijske čvorove. Ukoliko se odabir parametara pridržava strukture kombinacijske mreže kakvu predlaže ljudski ekspert, u tom slučaju se mogu pronaći rješenja istovjetna rješenju kojeg je predložio ljudski ekspert (najbolji primjer toga je treći ispitni primjer, jednobitno zbrajalo s prijenosom). S druge strane, ukoliko postoji struktura bolja od strukture koju je predložio

ljudski ekspert, odabir parametara može s jedne strane navoditi kako bi ta struktura trebala izgledati (najbolji primjer toga je svakako drugi ispitni primjer, primjena `case` konstrukta) ukoliko se počnu pronalaziti ispravna rješenja s parametrima koji ukazuju na manju veličinu kombinacijske mreže u odnosu na predloženu mrežu ljudskog eksperta.

S druge strane, nije pronađen nikakav odnos između međusobnog utjecaja parametara broja redaka, stupaca i stupnja povratka koji bi općenito vrijedio za bilo koji ispitni primjer. To ne znači da takav odnos ne postoji, ali možda je zanemariv u odnosu na utjecaj kojeg na njega ima struktura same kombinacijske mreže koja se oblikuje.

Što se tiče kvantitativne uspješnosti rezultata, za sva četiri ispitna primjera dobivene su maksimalno moguće vrijednosti dobrote, odnosno potpuno ispravna rješenja. Prema tome, rješenja koja se dobivaju su dobra u odnosu na ispravnost.

S obzirom na kvalitativnu uspješnost rezultata, ako se usporedi struktura kombinacijske mreže koju predlaže ljudski ekspert te struktura koja je dobivena kao rješenje kartezijskim genotipom, dobivaju se zanimljivi zaključci. Prvi ispitni primjer (binarni u 8-segmentni prikaz) dobiva rješenja koja su po broju sklopova bliska broju sklopova rješenja kojeg predlaže ljudski ekspert ukoliko se smanji funkcijski skup. Drugi ispitni primjer (primjena `case` konstrukta) dobiva bolja rješenja u odnosu na rješenje kojeg predlaže ljudski ekspert ukoliko se uzme prošireni funkcijski skup (s XOR i XNOR čvorom). Treći ispitni primjer (jednobitno zbrajalo s prijenosom) dobiva u potpunosti isto rješenje kao rješenje kojeg predlaže ljudski ekspert. Za četvrti ispitni primjer (dekoder 2 u 4) se dobiva rješenje blisko rješenju eksperta po broju sklopova i razina.

Kako je cilj bio pronaći ispravna rješenja, a ne optimalnu strukturu rješenja, zapravo su rezultati viši od očekivanih rezultata. U dva slučaja dobivena su bolja rješenja od onog kojeg predlaže ljudski ekspert, u jednom slučaju isto rješenje kao i predloženo, i u jednom rješenje koje je dovoljno blisko (po broju sklopova i razina) predloženom. Prema tome, kartezijski genotip je u stanju implicitno obavljati optimizaciju pri čemu u prvi plan dolazi korištenje neutralnosti u samom genotipu. To se može vidjeti iz broja aktivnih funkcijskih

čvorova u genotipu[17] koji su spojeni s krajnjim izlazima, što je pogotovo vidljivo ako se uzme kao parametar stupnja povratka broj blizak broju stupaca jer se u tim slučajevima dobiva manji postotak aktivnih funkcijskih čvorova u genotipu s obzirom na ukupan broj funkcijskih čvorova. Razlog tome je što se tada može „preskakati" puno veći broj stupaca funkcijskih čvorova nego u slučajevima kada je parametar stupnja povratka između 1 i $\frac{N}{2}$, pri čemu je N broj stupaca. Naravno, broj aktivnih funkcijskih čvorova ovisi i o samom broju ukupnih funkcijskih čvorova pa tako treba paziti i na odabir broja redaka i stupaca koji određuju ukupan broj funkcijskih čvorova u genotipu. Sličan zaključak s implicitnim otkrivanjem novih struktura kombinacijskih mreža od strane kartezijskog genotipa donijeli su Fišer, Schmidt, Vašíček i Sekanina u [17].

Zalihost, naravno, postoji, ali ne uzrokuje nagli rast genotipa (eng. *bloat*) jer je s jedne strane spriječena stalnom veličinom genotipa, a s druge strane neutralnošću koja aktivira ili deaktivira pojedine funkcijske čvorove. Čak i uz prisutnu zalihost u genotipovima, kartezijski genotip je bio u stanju pronaći ispravna i optimalna rješenja. Slične zaključke što se tiče neutralnosti i naglog rasta genotipa donijeli su Garmendia-Doval, Miller i Morley u [16].

S druge strane, postoji nedostatak mogućnosti globalnog pretraživanja obzirom da se sa stalnom veličinom genotipa sve svodi na lokalno pretraživanje. To je posebno vidljivo u četvrtom ispitnom primjeru (dekoder 2 u 4) gdje je možda bilo potrebno i malo globalno pretraživati okolni prostor kako se ne bi mogućnost pronalaska boljeg rješenja svela na odabir ispravnih parametara (prvenstveno broj redaka, stupaca i stupanj povratka).

Uočen je još jedan nedostatak, a to je trajanje evaluacije genotipova tokom evolucije. To je naročito bilo zamijećeno u prvom ispitnom primjeru (binarni u 8-segmentni prikaz) jer se uzela populacija od 200 jedinki te postavilo potpuno pokrivanje ulazne domene (100% pokrivenost). Trajanje jednog izvođenja s brojem redaka i stupaca postavljenim na 8 trajalo je nekoliko mminuta. Djelomično iz tog razloga se za preostale ispitne primjere

[17] Naravno, kada bi se fenotipovi tih genotipova koji predstavljaju rješenja „nacrtali", ali zbog velike veličine nekih fenotipova, ti crteži se nisu priložili u dodatku ovog znanstvenog rada nego samo genotipovi (kao nizovi prirodnih brojeva).

uzela populacija od 50 jedinki. S druge strane, smanjeni broj veličine populacije nije utjecao na mogućnost pronalaženja ispravnih rješenja[18] što znači da kartezijski genotip uistinu nosi dobru osobinu genotipova stalne veličine koji ne zahtijevaju veliku populaciju kao što je to slučaj s genotipovima promjenjive veličine. Što se tiče predugog izvođenja tj. evaluacije, problem nije u implementaciji nego u samom genotipu, što su potvrdili Fišer, Schmidt, Vašíček i Sekanina u [17] te Vassilev i Miller u [18] izjavivši da je evaluacija kartezijskog genotipa iznimno vremenski ovisna, pogotovo o broju ulaza. Iz tog razloga su za ispitne primjere uzeti samo oni čiji su ulazi i izlazi bili skalari, a ne vektori.

Općeniti zaključak je da se kartezijski genotip ponaša u skladu s postavljenim pretpostavkama. Lokalno pretražuje zbog stalne veličine genotipa te izbjegava nagli rast genotipa zbog stalne veličine i neutralnosti. S druge stane, nije se očekivalo da implicitno pronalazi bolja ili bar slična rješenja predloženim rješenjima od strane ljudskih eksperata.

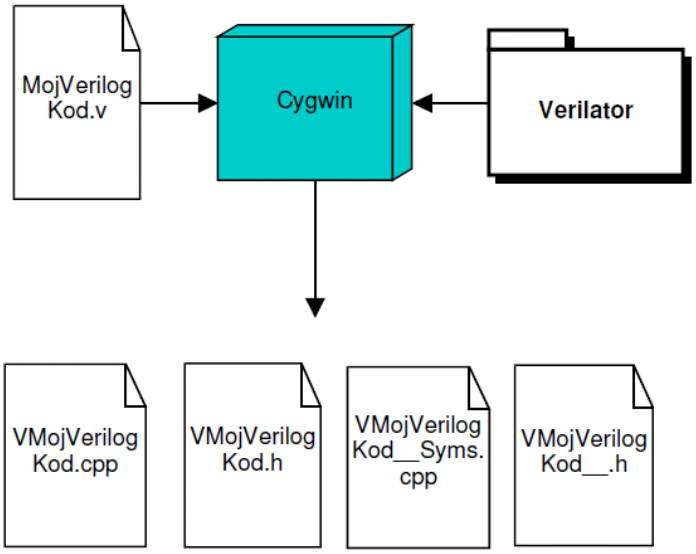
[18] Osim u četvrtom ispitnom primjeru (dekoder 2 na 4), ali tu čak i kada se pokušalo povećati populaciju na 100 jedinki i broj generacija na 200, i dalje se nije moglo pronaći ispravno rješenje.

4. Upute za korištenje

Neovisno o poglavlju o implementaciji rješenja (vidi poglavlje 2), ovdje je navedeno kako se priprema pokretanje jednog ispitnog primjera problema oblikovanja kombinacijske mreže s tehničkog aspekta.

Pretpostavlja se da je *Cygwin* pripremljen te da su se *Verilator*-ove izvorne datoteke u njemu prevele te da je moguće iz komandne linije u *Cygwin*-u upravljati *Verilator*-om. Također, pretpostavlja se da je ECF pokrenut unutar razvojne okoline *Microsoft Visual Studio*.

Prvi korak je prevođenje Verilog programa u C++ program uz pomoć *Verilator*-a koji se može pratiti i na Slika 25.



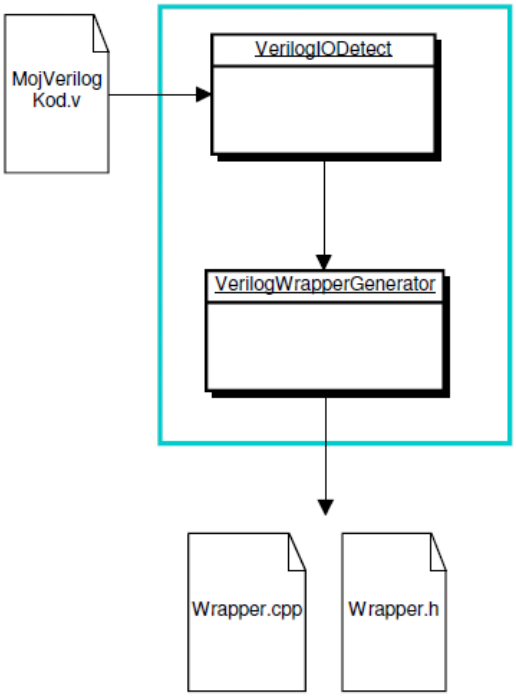
Slika 25. Tijek prevođenja Verilog programa s *Verilator*-om unutar *Cygwin* okoline

Kao što je opisano u poglavlju o *Verilator*-u (vidi poglavlje 2.3), potrebno je prvo pripremiti ulaznu Verilog datoteku (npr. neka se zove *MojVerilogKod.v*) te je prevesti unutar *Cygwin*-a na sljedeći način:

```
$ verilator --cc module.v
```

Stvara se novi direktorij *obj_dir* u aktivnom direktoriju (gdje se izvelo prevođenje) te se u njemu nalaze potrebne datoteke: *VMojVerilogKod.cpp, VMojVerilogKod.h, VMojVerilogKod__Syms.cp*p i *VMojVerilogKod__.h*. Time je završen prvi korak.

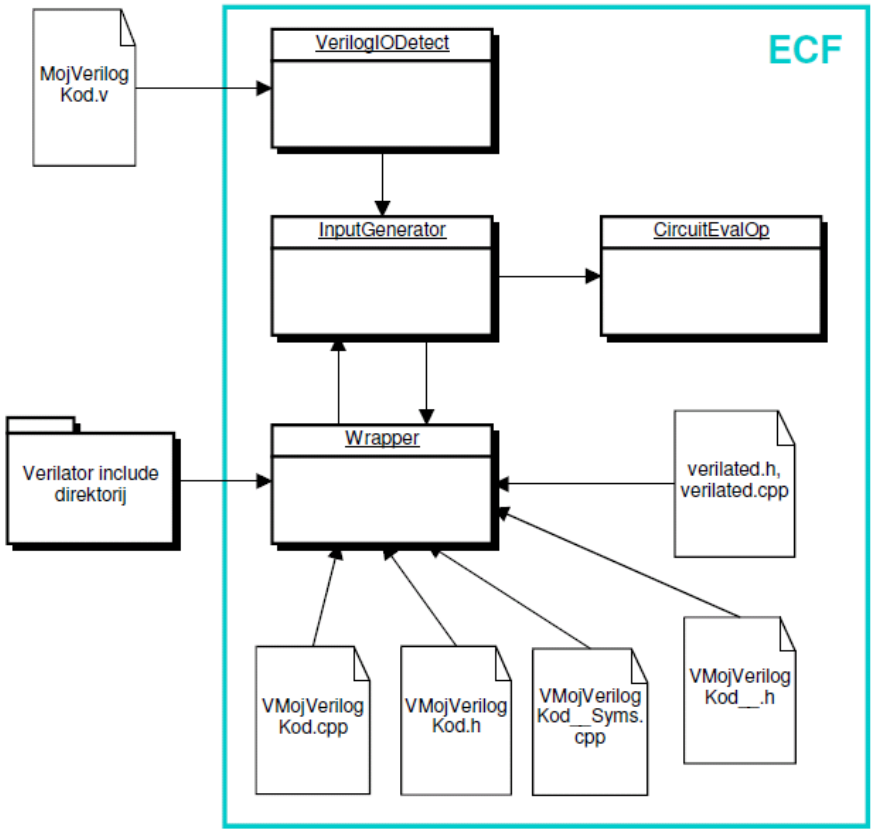
Drugi korak predstavlja stvaranje datoteke poveznice koje je opisano u poglavljima 2.4.1 i 2.4.2 (za skraćeni prikaz postupka vidjeti Slika 26).



Slika 26. Tijek stvaranja datoteke poveznice pomoću VerilogIODetect i VerilogWrapperGenerator klasa

Datotečni put do Verilog datoteke se predaje kao parametar komandne linije projektu unutar *Microsoft Visual Studio* okoline gdje se nalaze `VerilogIODetect` i `VerilogWrapperGenerator` klase. Iz Verilog datoteke se detektiraju ulazi i izlazi te se stvara datoteka poveznica kao `Wrapper` klasa podijeljena u dvije datoteke (*Wrapper.cpp* i *Wrapper.h*). Potrebno je još dodatno uključiti *include* direktorij od paketa Boost C++ zaglavlja jer se koristi njihovo zaglavlje s regularnim izrazima. Također, bit će potrebno i uključiti *.lib* datoteku vezanu uz Boost-ove regularne izraze *libboost_regex-vc90-mt-gd-1_42.lib* s time da *1_42* označava samo verziju paketa Boost-ovih zaglavlja koja se koristi.

Zadnji, odnosno treći korak jest pokretanje samog ECF-a s međurezultatima prikupljenim u prethodna dva koraka (grafički prikaz je na Slika 27).



Slika 27. Način komunikacije dijelova ECF-a s prethodno dobivenim međurezultatima (datoteka poveznica i datoteke koje je generirao *Verilator*)

Unutar ECF-a, u isti direktorij gdje se nalazi i klasa `CircuitEvalOp` (vidi poglavlje 2.5.5), postavljene su klase `VerilogIODetect` i `InputGenerator` (vidi poglavlje 2.4.3) koje se ne mijenjaju ovisno o strukturi problema koji se rješava. Promjenjivi elementi su datoteka poveznica (`Wrapper` klasa) i datoteke koje je generirao Verilator (one koje počinju nazivom *VMojVerilogKod*). Njih je potrebno svaki put kada se prođe kroz prva dva koraka kopirati u direktorij gdje se nalaze već spomenuti `CircuitEvalOp`, `VerilogIODetect` i `InputGenerator`. U sam projekt u *Microsoft Visual Studio* okolini (projekt u kojem je ECF) potrebno je još uključiti datotečni put do *Verilator*-ovog *include* direktorija (jer će se morati koristiti datoteke koje je generirao *Verilator*) te još iz tog *include* direktorija kopirati datoteke *verilated.cpp* i

verilated.h[19] i postaviti ih u već navedeni direktorij gdje su sve preostale datoteke. Zbog zahtjeva klase `VerilogIODetect` za Boost-ovim zaglavljima potrebno je još uključiti i *include* direktorij tih zaglavlja te već spomenutu *.lib* datoteku *libboost_regex-vc90-mt-gd-1_42.lib*.

Ono što na prethodnoj slici (vidi Slika 27) nije navedeno jest konfiguracijska datoteka koja vrijedi za cijeli ECF (kako detaljno podesiti konfiguracijsku datoteku može se pronaći na [22]). Ona izgleda na sljedeći način:

```
<ECF>
    <Genotype>
            <Cartesian>
                    <Entry key="type">uint</Entry>
                    <Entry key="numoutputs">1</Entry>
                    <Entry key="numinputconns">2</Entry>
                    <Entry key="numrows">8</Entry>
                    <Entry key="numcols">4</Entry>
                    <Entry key="levelsback">1</Entry>
                    <Entry key="functionset">AND NOT OR XOR
                    XNOR</Entry>
                    <Entry key="numvariables">4</Entry>
            </Cartesian>
    </Genotype>

    <Registry>
            <Entry key="population.size">200</Entry>
            <Entry key="vfile">C:/Negdje/MojVerilogKod.v</Entry>
            <Entry key="percentage">100</Entry>
    </Registry>
```

[19] Nije ih potrebno kopirati svaki puta, nego samo jednom kada se projekt prvi puta konfigurira.

```
</ECF>
```

Dijelovi konfiguracijske datoteke vezani uz kartezijski genotip su navedeni pod elementom `<Cartesian>`. Oni su redom:

1. `<type>` - tip podataka koji će se koristiti prilikom evaluacije vrijednosti u funkcijskim čvorovima (za problem oblikovanja kombinacijskih mreža potrebno je koristiti *unsigned int* što se u konfiguracijskoj datoteci naznačava kao „uint“, u suprotnom će biti uključen podrazumijevani tip, a to je *double*)

2. `<numoutputs>` - navodi se broj krajnjih izlaza iz kombinacijske mreže (treba odgovarati broju izlaza opisanih u Verilog datoteci koja se koristi)

3. `<numinputconns>` - navodi se broj ulaznih veza u funkcijske čvorove, u slučaju podrazumijevanih parametara za logičke funkcijske čvorove, ovdje se treba postaviti broj 2

4. `<numrows>` - broj redaka funkcijskih čvorova (u fenotipu)

5. `<numcols>` - broj stupaca funkcijskih čvorova (u fenotipu)

6. `<levelsback>` - parametar stupnja povratka

7. `<functionset>` - funkcijski skup korištenih funkcija u genotipu, logičke funkcije se navode imenima: AND, OR, NOT, XOR i XNOR (nije ih potrebno navesti sve, samo one koje se želi koristiti)

8. `<numvariables>` - broj korištenih inicijalnih ulaza (varijabli) u kombinacijsku mrežu (treba odgovarati broju ulaza opisanih u Verilog datoteci koja se koristi)

Ukoliko se u funkcijskom skupu žele navesti funkcije s drugačijim brojem parametara od pretpostavljenog ili se želi koristiti ista funkcija, ali za više različitih parametara tada se to u konfiguracijskoj datoteci može navesti na ovaj način:

```
<Entry key="functionset">NOT AND 2 3 OR 3 4 XOR</Entry>
```

U ovom slučaju će se, uz standardne NOT i XOR funkcijske čvorove, koristiti dva AND funkcijska čvora, jedan s 2 ulaza te jedan s 3 ulaza te dva OR funkcijska čvora, jedan s 3 ulaza i jedan s 4 ulaza. Iznimka je jedino NOT čvor koji može koristiti samo podrazumijevani broj parametara (prima samo jedan parametar).

Dijelovi konfiguracijske datoteke vezani uz evaluacijski operator kartezijskog genotipa su:

1. `<vfile>` - datotečni put do Verilog datoteke gdje je opisana kombinacijska mreža koja se želi oblikovati

2. `<percentage>` - postotak domene ulaznih vrijednosti koje generator ulaznih vrijednosti mora pokriti (ili postotak tablice istinitosti koji se uzima u obzir prilikom vrednovanja genotipa), ovdje može stajati samo prirodni broj u rasponu $<0,100]$

Vidljivo je kako postoji određena zalihost podataka u konfiguracijskoj datoteci jer čim se iz Verilog datoteke mogu pročitati ulazi i izlazi može se pročitati i broj ulaza i izlaza, a u konfiguracijskoj datoteci se svejedno traži da se navedu ti brojevi. Razlog zašto je to tako ostavljeno jest da kartezijski genotip bude neovisan o evaluacijskom operatoru[20] koji se koristi za rješavanje proizvoljnog problema.

[20] Kao što je možda već spomenuto, u ECF-u postoji implementiran još jedan evaluacijski operator koji ovisi o podacima iz konfiguracijske datoteke kao što je broj varijabli na ulazu i broj krajnjih izlaza (više informacija o tome u [15]).

Zaključak

U ovom znanstvenom radu dotakao se problem automatiziranog oblikovanja kombinacijskih mreža. Predloženo rješenje je bilo oblikovati mreže evolucijskim metodama, konkretno kartezijskim genetskim programiranjem.

Kartezijsko genetsko programiranje je u literaturi vezanoj uz tu varijantu genetskog programiranja navedeno kao prikladan odabir za probleme koji se vežu uz logičke operacija ili mreže. Nad četiri odabrana ispitna primjera pokazano je kako se kartezijsko genetsko programiranje dobro snalazi s takvim problemom. Premda je cilj bio dobivati ispravne kombinacijske mreže (što je izraženo u funkciji dobrote), a ne optimalne po pitanju broja sklopova i razina, kartezijsko genetsko programiranje je ipak bilo u mogućnosti implicitno oblikovati optimalne kombinacijske mreže. To je rezultat koji je bio sasvim neočekivan s obzirom na očekivane rezultate u ovom znanstvenom radu.

Kartezijsko genetsko programiranje je bilo u stanju osim pronalaženja ispravnih rješenja, koristiti neutralnost koja pruža mogućnost aktiviranja ili deaktiviranja pojedinih funkcijskih čvorova te na taj način dobivati rješenja koja u sebi nemaju preveliku zalihost. Naravno, to ne vrijedi za sva pronađena rješenja, nego za ona koja su subjektivno odabrana kao najbolja. Neka rješenja su uistinu bila djelomično ispravna te su u sebi sadržavala velik dio aktivnih funkcijskih čvorova koji su zahtijevali optimizaciju.

Pitanje je kako bi se kartezijsko genetsko programiranje ponašalo u slučaju kada bi mu se eksplicitno definirala optimizacija (npr. uvela pristranost prema što manjem broju aktivnih funkcijskih čvorova u funkciji dobrote) te bi li skup najboljih rješenja u tom slučaju bio bolji u vidu dobivanja optimalnih struktura kombinacijskih mreža. S jedne strane, neutralnost u kartezijskom genetskom programiranju djeluje kao implicitni optimizator kombinacijskih mreža, što znači da je eksplicitno uvođenje optimizacije možda i nepotrebno. S druge strane, uvođenje eksplicitne optimizacije u funkciju dobrote bi još dodatno usporilo evaluaciju za koju je utvrđeno da je veoma vremenski ovisna.

Za većinu ispitnih primjera uspjelo se dobiti ispravna rješenja, ali i optimalna te se time kartezijsko genetsko programiranje uistinu pokazalo kao dobar odabir za rješavanje problema automatskog oblikovanja kombinacijskih mreža.

Literatura

[1] Timothy G.W Gordon and Peter J Bentley, "On Evolable Hardware," in *Soft Computing in Industrial Electronics*.: Physica-Verlag, 2002, pp. 279-323.

[2] J.F Miller, P Thomson, and T Fogarty, "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," Deptartment of Computer Studies, Napier University, Edinburgh, Case Study 1997.

[3] Jim Torresen, "An Evolvable Hardware Tutorial," in *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'2004)*, Leuven, Belgium, 2004, pp. 821-830.

[4] Uroš Peruško and Vlado Glavinić, *Digitalni sustavi*. Zagreb: Školska knjiga, 2005.

[5] Melanie Mitchell, *An Introduction to Genetic Algorithms*. Massachusetts, USA: A Bradford Book (MIT Press), 1998.

[6] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. (2008, ožujak) A Field Guide to Genetic Programming.

[7] Iva Brajer, "Oblikovanje kombinacijskih mreža uporabom evolucijskih algoritama," Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb, Završni rad 951, 2009.

[8] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre, "Automated Design of Combinational Logic Circuits Using Genetic Algorithms," in *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, ICANNGA'97*, University of East Anglia, Norwich, England, 1997, pp. 335-338.

[9] Cecília Reis and J. A. Tenreiro Machado, "An Evolutionary Approach to the Synthesis of Combinational Circuits," in *Proceedings of ICCC 2003 IEEE International Conference on Computational Cybernetics*, Siofok, Hungary, 2003.

[10] John R Koza, *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. Cambridge, Massachusetts, USA: The MIT Press, 1992.

[11] Carlos A. Coello Coello, Arturo Hernández Aguirre, and Bill P Buckles, "A Genetic Programming Approach to Logic Function Synthesis by Means of Multiplexers," in *roceedings of the First NASA/DOD Workshop on Evolvable Hardware*, Los Alamitos, California, 1999, pp. 46-53.

[12] Wikipedia, the free encyclopedia. (2011, ožujak) Field-programmable gate array. [Online]. HYPERLINK "http://en.wikipedia.org/wiki/Field-programmable_gate_array" http://en.wikipedia.org/wiki/Field-programmable_gate_array

[13] J.F. Miller, "An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach," in *Proceedings of the Genetic and Evolutionary Conference (GECCO'99)*, San Francisco, 1999, pp. 1135-1142.

[14] J.F. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proceedings of the Third European Conference on Genetic Programming (EuroGP2000)*, Berlin, 2000, pp. 121-132.

[15] Iva Brajer, "Kartezijsko genetsko programiranje," Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb, Diplomski seminar 2010.

[16] A. Beatriz Garmendia-Doval, J. F. Miller, and S. David Morley, "Post Docking Filtering using Cartesian Genetic Programming," in *Genetic programming Theory and Practice*.

University of Michigan, Illinois, USA, 2004.

[17] Petr Fišer, Jan Schmidt, Zdeněk Vašíček, and Lukáš Sekanina, "On Logic Synthesis of Conventionally Hard to Synthesize Circuits Using Genetic Programming," in *On Proceedings of the 13th International IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, Vienna, 2010, pp. 346-351.

[18] V.K. Vassilev and J.F. Miller, "Towards the Automatic Design of More Efficient Digital Circuits," in *2nd NASA/DOD Workshop on Evolvable Hardware*, Palo Alto, 2000, pp. 151-160.

[19] L. Sekanina and V. Drábek, "Automatic Design of Image Operators Using Evolvable Hardware," in *Proceedings of 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, Brno, 2002, pp. 132-139.

[20] T. Martínek and L. Sekanina, "An Evolvable Image Filter: Experimental Evaluation of a Complete Hardware Implementation in FPGA," in *Evolvable Systems: From Biology to Hardware*, Berlin, 2005, pp. 76-85.

[21] Faculty of Information Technology Brno University of Technology. Evolvable Hardware Research Group. [Online]. HYPERLINK "http://www.fit.vutbr.cz/research/groups/ehw/pubs.php" http://www.fit.vutbr.cz/research/groups/ehw/pubs.php

[22] Domagoj Jakobović. (2011, travanj) ECF - Evolutionary Computation Framework. [Online]. HYPERLINK "http://gp.zemris.fer.hr/ecf/" http://gp.zemris.fer.hr/ecf/

[23] Wikipedia, the free encyclopedia. (2010, listopad) Tournament selection. [Online]. HYPERLINK "http://en.wikipedia.org/wiki/Tournament_selection" http://en.wikipedia.org/wiki/Tournament_selection

[24] Wikipedia, the free encyclopedia. (2011, siječanj) Fitness proportionate selection. [Online]. HYPERLINK "http://en.wikipedia.org/wiki/Fitness_proportionate_selection" http://en.wikipedia.org/wiki/Fitness_proportionate_selection

[25] Wikipedia, the free encyclopedia. (2011, travanj) Particle swarm optimization. [Online]. HYPERLINK "http://en.wikipedia.org/wiki/Particle_swarm_optimization" http://en.wikipedia.org/wiki/Particle_swarm_optimization

[26] Donald E Thomas and Philip R Moorby, *The Verilog Hardware Description Language, Fifth Edition*. Dordrecht: Kluwer Academic Publishers, 2002.

[27] Igor Bespaljko et al., "Napredna primjena evolucijskih algoritama," Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb, Projektna tehnička dokumentacija 2011.

[28] Wilson Snyder, Duane Galbi, and Paul Wasson. (2010, studeni) Introduction to Verilator. [Online]. HYPERLINK "http://www.veripool.org/wiki/verilator" http://www.veripool.org/wiki/verilator

[29] Wilson Snyder. (2011, travanj) Verilator - Documentation. [Online]. HYPERLINK "http://www.veripool.org/ftp/verilator_doc.pdf" http://www.veripool.org/ftp/verilator_doc.pdf

[30] (2011, travanj) Cygwin. [Online]. HYPERLINK "http://www.cygwin.com/" http://www.cygwin.com/

[31] (2011, travanj) MinGW | Minimalist GNU for Windows. [Online]. HYPERLINK "http://www.mingw.org/" http://www.mingw.org/

[32] (2011, travanj) The Perl Programming Language. [Online]. HYPERLINK "http://www.perl.org/" http://www.perl.org/

[33] (2011, travanj) flex: The Fast Lexical Analyzer. [Online]. HYPERLINK "http://flex.sourceforge.net/" http://flex.sourceforge.net/

[34] (2011, travanj) Bison - GNU parser generator. [Online]. HYPERLINK "http://www.gnu.org/software/bison/" http://www.gnu.org/software/bison/

[35] (2011, travanj) Boost C++ Libraries. [Online]. HYPERLINK "http://www.boost.org/" http://www.boost.org/

[36] Allison and Robert W. (2011, svibanj) Verilog Examples. [Online]. HYPERLINK "http://www.cecs.csulb.edu/~rallison/pdf/Decoder_2_to_4.pdf" http://www.cecs.csulb.edu/~rallison/pdf/Decoder_2_to_4.pdf

[37] Carlos A. Coello Coello, Arturo Hernández Aguirre, and Bill P Buckles, "Evolutionary Multiobjective Design of Combinational Logic Circuits," in *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware*, Los Alamitos, California, 2000, pp. 161-170.

[38] Wikipedia, the free encyclopedia. (travanj, 2011) Pareto Efficiency. [Online]. HYPERLINK "http://en.wikipedia.org/wiki/Pareto_efficiency" http://en.wikipedia.org/wiki/Pareto_efficiency

Naslov, sažetak i ključne riječi

Automatsko oblikovanje kombinacijskih mreža

Ovaj znanstveni rad obrađuje problematiku automatskog oblikovanja kombinacijskih mreža s gledišta evolucijskih metoda. Sama problematika je opisana u uvodnom dijelu uz navođenje činjenica iz literature. Opisano je kartezijsko genetsko programiranje koje će služiti kao evolucijska metoda rješavanja dotičnog problema te su navedene njegove prednosti i mane u odnosu na genetsko programiranje. Objašnjena je implementacija kartezijskog genetskog programiranja u programskom radnom okviru za evolucijsko računanje. Nad četiri ispitna primjera provelo se ispitivanje s obzirom na postavljena ispitna pitanja utjecaja parametara kartezijskog genotipa te kvalitativne i kvantitativne uspješnosti dobivenih rješenja. Dobivene rezultate se analiziralo te izvelo zaključke.

Ključne riječi: automatsko oblikovanje, kombinacijske mreže, kartezijsko genetsko programiranje

Automated design of combinatorial networks

This scientific work deals with automated design of combinatorial networks from the perspective of evolutionary methods. Issue by itself is described in introduction with adduction of facts from literature. Cartesian genetic programming which will be used as evolutionary method in solving questioned problem is described and all its advantages and disadvantages are mentioned. Implementation of Cartesian genetic programming within framework for evolutionary computation is described. Examination is conducted over four test examples relating to the test questions of Cartesian genetic parameter influence and quantitative and qualitative efficacy of given solutions. Given solutions are analyzed and conclusion is made.

Keywords: automated design, combinatorial networks, Cartesian genetic programming