SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2583

METAJEZIK ZA OPIS EVOLUCIJSKIH ALGORITAMA

Marko Đurasević

Zagreb, lipanj 2012.

ZAHVALA

Zahvaljujem svojim roditeljima na svemu što su ikad napravili za mene, što su mi omogućili sve što je bilo u njihovoj moći i što su mi uvijek bili najveći oslonac u životu.

Također zahvaljujem svim prijateljima koji su prošle tri godine bili uz mene, i što su mi pomagali kad mi je bilo najpotrebnije.

I za kraj, zahvaljujem svom mentoru Domagoju Jakoboviću, na tome što je probudio interes u meni za ovo područje, za sva odgovorena pitanja, za vrijeme koje je utrošio na mene, za sve savjete koje mi je pružio tijekom izrade ovog rada i za razumijevanje koje je uvijek ukazao.

Svima vam od srca hvala, bez vas ovaj rad ne bi bio moguć.

Marko Đurasević

Sadržaj

1. Uvod

Ljudi su se tijekom svoje povijesti konstantno susretali s raznovrsnim problemima koje su pokušavali riješiti. Nažalost, ti su problemi vrlo često bili preteški ili nerješivi. Iako su ljudi razvojem civilizacije i tehnologije pronašli nove metode za rješavanje problema, u isto vrijeme pojavljivali su se problemi koju su bili još teži nego što su bili prijašnji. Bez obzira na to što se trenutačno nalazimo u razdoblju veoma brzih računala, broj problema koji se niti njima ne mogu u potpunosti riješiti, u nekom razumnom vremenskom razdoblju, je pozamašan. Upravo zbog toga razvijene su tzv. heurističke metode (grč. *Heuriskein* – umijeće pronalaska novih strategija (pravila) za rješavanje problema), odnosno algoritmi koji ne garantiraju pronalaženje optimalnog rješenja nekog problema, ali omogućuju pronalaženje rješenja koje je po nekim kriterijima dovoljno dobro i to u puno kraćem vremenskom okviru, nego što bi bilo potrebno za pronalaženje optimalnog rješenja. Upravo korištenjem takvih metoda otvorila se mogućnost rješavanja problema, čije bi rješavanje u normalnim okolnostima trajalo izrazito dugo.

No, problem opet nastaje u tome što se računalu na neki način mora opisati kako da riješi određeni problem. Upravo zbog toga dolazi do potrebe za razvojem načina kako bi se računalu opisalo što točno treba raditi. Rezultat te potrebe su upravo programski jezici, koji uvode apstrakciju i olakšavaju programeru da određuje računalu što da radi. Tijekom godina su se pojavili mnogi programski jezici opće uporabe koji programeru omogućuju pisanje različitih programa za računala (C, Java, C#, Python, Ruby), no također i jezici koji su usko specijalizirani za neko područje (ESDL [1], StreamIt [2]). Upravo takvi jezici olakšavaju pisanje programa koji su usko vezani za neko područje. Razlog tome je to što programera oslobađaju brige vezana uz mnogobrojne detalje koji njemu nisu izravno bitni. Na taj način povećavaju produktivnost samog programera, no također smanjuju i broj pogrešaka koje se mogu dogoditi.

Zbog sve veće popularnosti metaheuristika i njihove sve šire uporabe, nameće se potreba za razvojem jezika koji bi omogućavao jednostavan opis takvih metoda i algoritama. Predmet proučavanja ovog rada je upravo metajezik za opis

evolucijskih algoritama (evolucijski algoritmi su jedna grana metaheuristika). Metajezik opisan u ovom radu otvara mogućnost programerima da na jednostavan, razumljiv i brz način mogu opisati evolucijske algoritme. Prilikom korištenja metajezika programer ne mora nužno imati nikakva predznanja o okolini unutar koje se taj evolucijski algoritam izvodi.

U ovom radu opisan je metajezik za opis evolucijskih algoritama pod nazivom ECDL (engl. *Evolutionary Computation Description Language*). U drugom poglavlju dan je vrlo kratak uvod u osnovne oblike evolucijskih algoritama (genetski algoritam, genetsko programiranje, evolucijska strategija i evolucijsko programiranje). Nakon toga, u trećem poglavlju, dan je detaljan opis jezika ECDL, kao i programskog prevoditelja za navedeni jezik, dok su u četvrtom poglavlju dani primjeri evolucijskih algoritama napisani u jeziku ECDL.

2. Evolucijski algoritmi

Evolucijski algoritmi su podgrana metaheurističkih metoda optimizacije, koji koriste mehanizme inspirirane biološkom evolucijom, kao što su selekcija, mutacija, križanje i reprodukcija. Oni rade na principu da navedene mehanizme primjenjuju nad populacijom jedinki, pri čemu svaka jedinka populacije predstavlja moguće rješenje problema koji se rješava evolucijskim algoritmom. Kako bi se odredilo koje jedinke predstavljaju dobra, odnosno loša rješenja, one se ocjenjuju funkcijom dobrote, koja određuje koliko zapravo neka jedinka predstavlja dobro i kvalitetno rješenje. Evolucijski algoritmi su zbog svojih svojstava primjenjivi na veoma širok raspon problema, primjerice na problem trgovačkog putnika, raspoređivanja, pronalaženja minimuma funkcije i mnoge druge. Osim toga ti su algoritmi primjenjivi i u neke nekonvencionalne svrhe, tako se primjerice u zadnje vrijeme evolucijski algoritmi koriste i za stvaranje raznih oblika umjetnosti, bilo likovnih ili glazbenih djela. Postoje različiti oblici evolucijskih algoritama, no četiri najčešće korištenih su: genetski algoritam, genetsko programiranje, evolucijska strategija i evolucijsko programiranje.

2.1. Genetski algoritmi

Genetski algoritmi razvijeni su 1970-ih godina od J. Hollanda, te predstavljaju jedan od najpopularnijih tipova evolucijskih algoritama [3]. Najviše se primjenjuju na probleme optimizacije i pretraživanja. Tradicionalno su se rješenja u genetskim algoritmima zapisivala koristeći binarnu reprezentaciju problema, no danas se mogu naći i algoritmi koji koriste i mnoge druge oblike reprezentacije problema. Genetski algoritmi primjenjuju operator križanja nad dvjema jedinkama roditelja, te na taj način dolazi do stvaranja nove jedinke, nad kojom se dodatno još primjenjuje operator mutacije koji uvodi dodatne promjene nad novonastalom jedinkom, kako bi se još više povećala raznovrsnost. Raznovrsnost rješenja je veoma važna u genetskim algoritmima, kako bi se spriječilo to da algoritam zapne u nekom od lokalnih optimuma, i da se na taj način onemogući pronalaženje nekog novog i boljeg rješenja. Operator selekcije koristi se kako bi se osiguralo da bolja rješenja „opstanu“, dok ona lošija rješenja „odumiru“ i „nestaju“. Na taj način nastoji se stvoriti sve bolja i bolja populacija jedinki kako bi se na kraju u toj populaciji pojavilo neko rješenje koje je dovoljno dobro. Operatori selekcije,

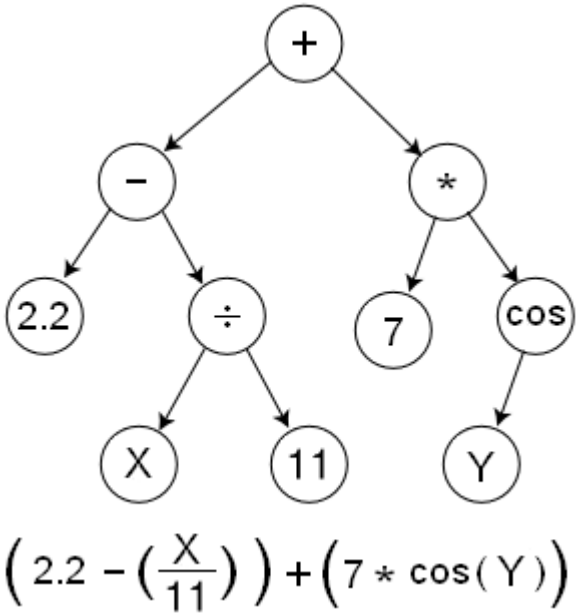
križanja i mutacije primjenjuju se nad populacijom, sve dok nije zadovoljen neki kriterij zaustavljanja, koji može biti: pojavljivanje dovoljno dobre jedinke, prolazak određenog broja iteracija, vremensko ograničenje, broj iteracija u kojima se nije ostvario napredak u populaciji i slično. Primjer pseudokoda vrlo jednostavnog genetskog algoritma prikazan je na slici 2.1.

```
stvori inicijalnu populaciju jedinki
evaluiraj inicijalnu populaciju jedinki
ponavljaj do zadovoljenja kriterija zaustavljanja
    odaberi nasumično 3 jedinke iz populacije
    od odabrane tri, najbolje dvije kriaj
    treću jedinku zamijeni novodobivenom jedinkom
    mutiraj novodobivenu jedinku
    evaluiraj novodobivenu jedinku
```

Slika 2.1. Pseudokod jednog genetskog algoritma

2.2. Genetsko programiranje

Genetsko programiranje razvijeno je od Johna R. Koze [3]. Ono se prvenstveno razlikuje od ostalih evolucijskih algoritama po tome što jedinke ne predstavljaju rješenja nekog problema, već program za rješavanje problema. Programi su pri tome najčešće prikazani u obliku strukture stabla, koja je idealna za rješavanje matematičkih problema (pri čemu unutarnji čvorovi predstavljaju operatore, dok listovi predstavljaju operande,). Slika 2.2. prikazuje primjer jednostavnog matematičkog izraza zapisanog u obliku stabla. Genetsko programiranje koristi iste genetske operatore koji su korišteni i u genetskim algoritmima (križanje i mutacija), samo što u ovom slučaju ti operatori moraju biti prilagođeni za rad nad stablima. Ovaj oblik evolucijskog algoritma se vrlo često koristi prilikom strojnog učenja.



Slika 2.2. Primjer matematičkog izraza zapisanog u obliku stabla

2.3. Evolucijska strategija

Evolucijska strategija je originalno razvijena od Rechenberga i Schewefela 1964. godine na Tehničkom sveučilištu Berlin [3]. Ovaj oblik evolucijskih algoritama uglavnom koristi operatore mutacije i selekcije, dok se križanje u ovom obliku evolucijskih algoritama veoma rijetko koristi. Evolucijska strategija koristi se za kontinuirane optimizacije, pri čemu su jedinke zapisane kao vektori realnih brojeva. Osim zapisa o rješenju problema, jedinke sadrže i neke druge parametre koji utječu na sam algoritam. Na taj način evolucijska strategija proizvodi i svojevrstan oblik samoadaptacije na način da se evolucijom rješenja također evoluiraju i parametri strategije u isto vrijeme. Postoje mnogobrojne varijante evolucijskih strategija, koje određuju kolika će biti veličina populacije dobivene primjenom genetskih operatora nad populacijom roditelja, no također određuju i na koji će se način između tih dviju populacija (populacije roditelja i populacije djece dobivene genetskim operatorima) odabrati jedinke za stvaranje nove generacije roditelja.

2.4. Evolucijsko programiranje

Evolucijsko programiranje razvijeno je od 1960. godine od Lawrencea J. Fogela [3]. Ovaj oblik evolucijskih algoritama uopće ne koristi operator križanja, nego se u

potpunosti oslanja na operator mutacije. U početku se evolucijsko programiranje koristilo za evoluciju konačnih automata kako bi se riješili problemi predviđanja, dok se kasnije evolucijsko programiranje koristi za rješavanje kontinuiranih optimizacijskih problema koristeći reprezentaciju jedinki realnim brojevima. Evolucijsko programiranje se danas sve rjeđe koristi od ostalih oblika evolucijskih algoritama upravo zbog svoje velike sličnosti s evolucijskom strategijom.

3. Metajezik za opis evolucijskih algoritama

Zbog sve veće popularnosti i sve šireg korištenja evolucijskih algoritama javlja se rastuća potreba za razvojem jezika koji bi služio za opis navedenih algoritama. Upravo je cilj ovog rada bio razviti i implementirati metajezik koji bi se koristio za opis evolucijskih algoritama. Cilj tog metajezika je prije svega da bude jednostavan i razumljiv, pa čak i onim korisnicima koji se prvi put susreću s jezikom. Osim toga korisnici, odnosno programeri su oslobođeni od brige za sve njima nevažne stvari, te se u potpunosti mogu posvetiti pisanju algoritama, na mnogo višoj razini nego što bi to bio slučaj da isti algoritam pišu u nekom od viših programskih jezika.

3.1. Postojeća rješenja

Naravno da ovaj metajezik nije prvi pokušaj osmišljavanja jezika za opis evolucijskih algoritama. Takvih pokušaja je već bilo nekoliko, no nažalost razvoj većina tih rješenja je odavno napušten (Open BEAGLE [2] i Evolutionary Algorithm Modeling Language [3]). Jedan novi metajezik zvan ESDL (Evolutionary System Definition Language) razvio je Steve Dower [1] [6] [7] [8] [9] [10] [11]. Ovaj jezik predstavlja puno razvijenije i bolje rješenje od prethodno navedenih. Sintaksa ovog jezika inspiraciju djelomično vuče iz jezika SQL, pa se tako mogu prepoznati pojedine ključne riječi koje dolaze iz tog jezika (*SELECT*, *FROM*) i djelomično iz Pascala, odakle je moguće prepoznati ključne riječi poput *BEGIN* i *END*. Upravo to čini jezik veoma lako razumljivim čak i bez ikakvog predznanja o samom jeziku. Bez obzira na sve navedene prednosti, ovaj jezik ima jednu veliku manu. Prilikom samog opisa evolucijskog algoritma potrebno je specificirati i parametre s kojima se dani algoritam izvodi, što je moguće vidjeti na slici 3.1. Na taj način se uvodi nefleksibilnost u jezik, jer se prilikom promjene parametara zahtijeva ponovno prevođenje cijelog algoritma, što dodatno oduzima vrijeme i smanjuje produktivnost programera, što naravno predstavlja veliki nedostatak. Još jedan, ali ne toliko izraženi nedostatak, jest da se ESDL program prevodi u tekst programa višeg programskog jezika Python. Zbog svojih karakteristika, program napisan u Pythonu je obično nekoliko puta sporiji od istog programa napisanog u nekom drugom višem programskom jeziku (C/C++, Java, C#). To posebice predstavlja problem prilikom izvođenja ovako zahtjevnih algoritama, što može potrajati veoma

dugo (ovisno o problemu, nekoliko sati, pa čak i nekoliko dana). Upravo zato usporenje od strane programskog jezika predstavlja dodatan nedostatak.

```
FROM random_real SELECT 500 population
YIELD population

BEGIN generation_equivalent
REPEAT 500
FROM population SELECT 2 parents USING binary_tournament
FROM parents SELECT offspring USING crossover ( per_pair_rate
=0.9), \
mutate ( per_gene_rate =0.01)

FROM offspring SELECT 1 replacer USING best
FROM population SELECT 1 replacee , rest USING uniform_shuffle

YIELD offspring , replacee

FROM replacer , rest SELECT population
END REPEAT

YIELD population
END
```

Slika 3.1 Primjer genetskog algoritma zapisanog u ESDL-u

3.2. Osnove jezika

Metajezik razvijen u sklopu ovog rada naziva se ECDL (Evolutionary Computation Description Language). Svoju sintaksu vuče iz viših programskih jezika poput C-a, Jave i C#-a. Tako se primjerice od programera zahtjeva da svaku naredbu završi s oznakom kraja naredbe „ ; “, kao što se radi i u navedenim višim programskim jezicima. Također kao i u višim programskim jezicima postoji mogućnost komentiranja pojedinih linija teksta programa. Tako se znakovi „ # “ i „ // “ koriste za jednolinijske komentare, dok se pomoću znakova „ /* “ otvara komentar koji se može protezati kroz više linija sve do znaka zatvaranja komentara „ */ “, što je vidljivo na slici 3.2. Praznine tabulatori se mogu slobodno dodavati između pojedinih elemenata jezika, budući da se oni ne uzimaju u obzir tijekom parsiranja, te tako nemaju nikakvog utjecaja na program. Također je jednu naredbu moguće i razlomiti u više redaka, bez ikakve posebne oznake. U jeziku

ne postoji mogućnost definicije korisničkih funkcija ili klasa budući da za time efektivno nema potrebe, te bi se samo uvela nepotrebna kompleksnost u jezik.

```
#jednolinijski komentar

//drugi oblik jednolinijskog komentara

/*
višelinijski
komentar
*/
```

Slika 3.2. Primjer podržanih komentara

Jedno vrlo važno svojstvo jezika, koje je potrebno ovdje napomenuti jest da se parametri algoritma ne navode unutar zapisa algoritma. Parametri algoritma su u potpunosti odvojeni od implementacije algoritma i s njom ni na koji način nisu povezani. Parametri se pohranjuju u posebnoj datoteci (objašnjeno kasnije), koja se parsira tek tijekom izvođenja algoritma. Upravo zbog toga u jezik nisu niti ugrađeni nikakvi mehanizmi koji služe za specificiranje parametara kroz sam jezik. Programer će kasnije kroz tu datoteku specificirati parametre koje će zadani algoritam koristiti.

3.3. Programski prevoditelj

Kako bi metajezik bio i praktično upotrebljiv, u sklopu rada razvijen je ECDLC (engl. *Evolutionary Computation Description Language Compiler*), programski prevoditelj koji datoteku u kojoj se nalazi program zapisan u ECDL-u prevodi u program nekog višeg programskog jezika. Trenutačno je podržano prevođenje samo u tekst programa jezika Java, iako se kasnije planira proširenje koje će omogućiti i dobivanje C++ teksta programa, odnosno međukoda koji bi omogućavao just-in-time prevođenje. Sam programski prevoditelj u potpunosti je napisan u programskom jeziku C#.

Programski prevoditelj prima jedan argument, a to je ime datoteke koju treba prevesti iz ECDL-a u Java program. Važno je naglasiti da navedena datoteka koja se prevodi obavezno mora sadržavati nastavak (ekstenziju) „.ecdl“, u protivnom se prevođenje neće moći izvršiti. Potrebno je također naglasiti da će se dobivena klasa u tekstu programa dobivenom nakon prevođenja i pripadajuća Java datoteka

zvati isto onako kako se zvala datoteka koju je prevoditelj primio kao argument (naravno bez pripadajuće ekstenzije). Upravo zbog te činjenice poželjno je da programer smisleno imenuje datoteku u kojoj se nalazi izvorni ECDL tekst programa, kako bi se spriječilo da nakon prevođenja ne dođe do neželjenih posljedica. Ime datoteke trebalo bi započinjat velikim slovom i ono ne bi smjelo sadržavati ikakve dijakritičke znakove, specijalne znakove (osim znaka „ _ “) ili praznine. Točnije rečeno ime bi trebalo zadovoljavati pravila imenovanja varijabli programskog jezika C (osim napomene vezane uz početno slovo imena datoteke). Također programer treba paziti da datoteci ne dodijeli ime koje je već definirano unutar programskog jezika Java, kako ne bi došlo do konflikta imena. Naravno ako se programer pridržava napomene da se imena datoteka s ECDL tekstom programa imenuju smisleno, do problema ne bi smjelo doći.

Ukoliko programer ne želi da mu programski prevoditelj izgenerira datoteku s istim imenom koju je imala i ulazna datoteka, postoji način kako to može i napraviti. Naime, jedina opcija koji programski prevoditelj trenutno ima ugrađenu u sebe je „–o“, koja omogućuje korisniku da specificira ime datoteke koja će se generirati. Ova opcija dolazi nakon što se navede ime ulazne datoteke. Iza navedene opcije potrebno je navesti ime koje će se koristiti kao naziv izlazne datoteke, ali kao i ime klase koja će biti generirana unutar datoteke. Ime mora biti navedeno bez ikakvih ekstenzija, jer će programski prevoditelj sam generirati datoteku s ekstenzijom „.java“.

Datoteku dobivenu prevođenjem datoteke s ECDL tekstom programa naravno nije moguće samostalno prevesti i izvesti. U toj datoteci će se nalaziti samo tekst programa koji će opisivati algoritam napisan od strane korisnika, no u njemu se neće nalaziti definicije operatora, jedinki i ostalih potrebnih struktura. To je zbog toga što je ovaj prevoditelj napravljen tako da se dobiveni tekst programa oslanja na već postojeće razvojno okruženje u kojem su definirane sve potrebne strukture podataka, metode i slične stvari važne za prevođenje i izvođenje dobivenog teksta programa. Razvojno okruženje na koje se oslanja ovaj programski prevoditelj je ECF (Evolutionary Computation Framework) [12], odnosno Java implementacija tog okruženja pod nazivom ECFJ (Evolutionary Computation Framework in Java). ECFJ je razvojno okruženje namijenjeno upravo razvoju različitih oblika evolucijskog računanja. Tako se nakon prevođenja dobiva tekst programa u Javi

koji koristi mnogobrojne i različite strukture, metode i operatore iz navedenog okruženja.

Kao što je već ranije navedeno, jezik ne podržava mehanizme za specifikaciju parametara algoritma, kao što su primjerice faktor mutacije, faktor križanja, uvjet zaustavljanja algoritma i slično. To svojstvo jezika proizlazi izravno iz ECF-a. Naime u ECF-u je specificirano da se ti parametri algoritma odvoje od opisa samog algoritma. Iako se to na prvi pogled možda može činiti nelogično ili čudno, takvim postupkom se zapravo dobiva na velikoj fleksibilnosti. Svi parametri algoritma izdvojeni su u zasebnu XML (Extended Markup Language) datoteku. Kada se evolucijski algoritam izvodi, on parsira navedenu XML datoteku, te iz nje čita parametre koji su potrebni za algoritam. Na taj način se programera oslobađa brige o ikakvim parametrima algoritma, te se on može u potpunosti posvetiti samo na dizajniranje samog algoritma. Osim toga odvajanjem parametara algoritma u posebnu datoteku i njezinim parsiranjem tijekom izvođenja samog algoritma nestaje potreba za ponovnim prevođenjem teksta programa algoritma ukoliko dođe do primjene parametara, nego je samo potrebno ponovo pokrenuti program, za razliku od slučaja kada bi parametri bili direktno definirani u tekstu programa algoritma što bi onda zahtijevalo ponovno prevođenje teksta programa, što svakako predstavlja nepotreban gubitak vremena.

3.4. Varijable i konstante

Kao i mnogobrojni drugi programski jezici ECDL također podržava varijable, no njihovo značenje je ponešto drugačije nego što je to u višim programskim jezicima. U ECDL-u varijabla predstavlja jedan multiskup jedinki, dok je ona u tekstu programa jezika Java ostvarena kao vektor jedinki. Varijable mogu predstavljati multiskup proizvoljnih veličina, pa se tako one mogu sastojati od jedne jedinke, više tisuća jedinki ili čak niti jedne jedinke. Svaka varijabla može u sebi sadržavati više referenci na istu jedinku. U ECDL-u ne postoje varijable nekog drugog tipa podatka (cjelobrojnih, realnih, znakovnih) kao što postoji u višim programskim jezicima, jer za njima jednostavno nema potrebe. Iz razloga što varijable predstavljaju jednu i samo jednu stvar, a to je multiskup jedinki, nestaje potreba za njihovom eksplicitnom deklaracijom (slično kao u mnogim skriptnim jezicima). Zbog toga se prvo pojavljivanje neke varijable može smatrati kao i neka vrsta

deklaracija, no pritom se treba paziti na činjenicu da varijabla prilikom svoje prve pojave predstavlja prazan multiskup, i tako sve dok joj se ne pridijele neke jedinke (izuzetak je varijabla *population* koja je objašnjena kasnije).

Što se imenovanja varijabli tiče, ono se podvrgava istim pravilima imenovanja koja koristi i programski jezik C za imenovanje istih. To znači da ime varijable mora početi s velikim, odnosno malim slovom engleske abecede ili znakom „_“, dok se ostatak imena može sastojati također od navedenih znakova i brojeva, što je vidljivo na slici 3.3. Imena varijabli osjetljiva su na velika i mala slova tako da primjerice imena *jedinka*, *JEDINKA* i *JeDiNkA* predstavljaju potpuno različite varijable. Također varijable moraju imati imena koja su različita od ključnih riječi ECDL-a. Tako se primjerice jedinke ne mogu nazvati *if*, *else*, *repeat* ili slično. Osim navedenih ograničenja postoji još jedno dodatno ograničenje prilikom imenovanja varijabli. Budući da se ECDL tekst programa prevodi u tekst programa jezika Java potrebno je paziti da se varijablama ne dodijeli neko od imena koje je u Javi zauzeto (int, *String*, *float*, *catch* i slično) i da se tako prilikom generiranja teksta programa ne bi generirao nejednoznačan tekst programa Java jezika koji se kasnije ne bi mogao ispravno prevesti.

```
INDIVIDUAL //ispravno

indiviual2 //ispravno

INDIVIDUAL_3 //ispravno

_INDIVIDUAL //ispravno

3INDIVIDUAL //neispravno, broj ne smije biti na početku naziva
varijable

INDIVUDUAL! //neispravno, naziv varijable sadrži nedozvoljen
znak
```

Slika 3.3. Primjeri naziva varijabli

Unutar jezika postoji jedna specijalna varijabla koja se naziva *population*. Ta varijabla predstavlja inicijalni multiskup jedinki nad kojom se mogu obavljati

operacije u evolucijskom algoritmu. Ona se po ničemu drugom ne razlikuje od ostali varijabli koje se koriste unutar ECDL-a, osim po tome što ona prilikom svoje prve upotrebe već u sebi sadrži neki broj jedinki koji je određen u konfiguracijskoj datoteci algoritma.

Sve varijable imaju dva osnovna svojstva koja se mogu iskoristiti. Svojstvima varijabli može se pristupiti tako da se nakon imena varijable navede točka i nakon toga ime jednog od dvaju dostupnih svojstava: *size* ili *clear*. Svojstvo *size* nam vraća broj jedinki koje se nalaze u multiskupu koju predstavlja varijabla. Pri tome ukoliko se ista jedinka unutar multiskupa ponavlja više puta, ona će se prebrojati za svaku njenu pojavu. Drugo svojstvo je *clear*. Budući da se ponekad javlja potreba za pražnjenjem varijable bilo je potrebno unutar jezika ostvariti mehanizam pomoću kojeg se ovakva funkcionalnost da vrlo lako ostvariti (primjerice u uzastopnim iteracijama kroz populaciju, prilikom svakog prolaska potrebno je očistiti varijablu koja predstavlja turnir, inače bi se u toj varijabli prikupljale jedinke iz prošlih iteracija). Primjer uporabe svojstava *clear* i *size* je vidljiv na slici 3.4.

Budući da varijable predstavljaju multiskupove koji se mogu sastojati od mnogo jedinki potreban je mehanizam kako bi se moglo pristupiti pojedinim jedinkama unutar varijable. Taj mehanizam ostvaren je kao indeksiranje, odnosno navođenjem uglatih zagrada iza imena varijable, te broja jedinke kojoj se želi pristupiti umanjenog za jedan (dakle prva jedinka ima indeks nula, druga indeks jedan i tako dalje). Na taj način pristupa se jedinci unutar varijable s navedenim indeksom. Pri tome se mora paziti da se kao indeks ne navede negativan broj ili broj koji premašuje broj jedinki unutar varijable koja se indeksira. Indeksiranje bi se trebalo koristiti s oprezom i samo onda kada je programer siguran da se na zadanom indeksu sigurno nalazi jedinka. Primjer indeksiranja zadnje jedinke vidljiv je na slici 3.4.

```
individual.clear;

individual = population[population.size-1];
```

Slika 3.4. Primjer svojstava i indeksiranja varijabli

Osim varijabli u ECDL-u postoje i dvije vrste konstanti. To su cjelobrojne i realne konstante. Za ostalim konstantama, primjerice znakovnim, nema potrebe i stoga nisu ugrađene u jezik, kao što je i vidljivo na slici 3.5. Konstante se ne mogu pridjeljivati varijablama, budući da ne postoje tipovi varijabli kojima bi se te konstante mogle pridijeliti, nego se one mogu koristiti samo unutar uvjeta naredbi grananja, uvjeta programskih petlji i prilikom indeksiranja jedinki unutar varijabli.

```
3.14

4

"string" // nije podržano

'c' // nije podržano
```

Slika 3.5. Primjeri konstanti

3.5. Naredbe grananja i petlje

Kao i mnogim drugim programskim jezicima, i u ECDL-u je podržana naredba grananja *if else*, dok naredba grananja *switch case* nije podržana u jeziku. Oblici *if else* naredbi koji su podržani isti su onima viših programskih jezika poput C#-a ili Jave, upravo zbog toga što se izrazi unutar zagrada moraju moći evaluirati u istinu ili laž. Nije dozvoljeno, primjerice, unutar uvjeta staviti samo broj tri i očekivati da će se on evaluirati kao istina, ili obrnuto staviti broj nula i očekivati da će se on evaluirati u laž (kao što bi bio slučaj u programskom jeziku C). Uz *if* dio u zagradama dolazi uvjet koji se ispituje i prema čijem ishodu se odlučuje koji će se programski dio izvršiti. Ukoliko je uvjet evaluiran kao istina izvršava se samo dio programa koji je naveden u *if* dijelu, a ukoliko se evaluira kao laž, preskače se dio u *if* dijelu, te se izvršava dio unutar else dijela naredbe grananja, ukoliko je on prisutan. Uz else dio se može navesti nova *if* naredbe i tako se može dobiti ulančavanje od nekoliko *if* naredbi. Primjer jedne *if else* naredbe prikazan je na slici 3.6.

```
if(population.size==0){
    //blok naredbi
}
else if(individual.size==population.size){
    //blok naredbi
}
else{
    //blok naredbi
}
```

Slika 3.6. Primjer uvjetne naredbe

Osim naredbe grananja ECDL podržava još i tri naredbe petlje. Svaka od tih naredbi prilagođena je za izradu petlji koje su specijalizirane za točno određeni problem. Potrebno je napomenuti da ECDL ne podržava naredbe kojima se može modificirati izvođenje petlje, kao što su *continue* i *break*.

Ukoliko se određeni dio teksta programa želi izvršiti točno određeni broj puta, koristi se programska petlja *repeat*. U uvjet *repeat* petlje se stavi broj ponavljanja koji se treba izvršiti. To ne mora nužno biti konstanta, nego se unutar tog uvjeta može staviti i svojstvo size varijable, što znači da će se zadani programski odsječak izvršiti onoliko puta koliko je jedinki unutar multiskupa koju zadana varijabla predstavlja. Na slici 3.7. prikazani su primjeri korištenja naredbe *repeat*.

```
repeat(100){
    //blok naredbi
}

repeat(population.size){
    //blok naredbi
}
```

Slika 3.7. Primjer repeat naredbe

Ukoliko se neki programski odsječak mora izvršavati sve do zadovoljavanja nekog uvjeta, koristi se programska petlja *while*. Unutar uvjeta *while* naredbe zadaje se uvjet koji se evaluira, kao što je vidljivo na slici 3.8, isto kao i kod *if* naredbe grananja. Izraz se evaluira prilikom svakog ulaska u petlju, ukoliko je izraz evaluiran kao istina, onda se izvršava tijelo petlje, te se ponovo ispituje uvjet i

tako sve dok uvjet nije zadovoljen. Kada uvjet nije zadovoljen preskače se tijelo petlje te se nastavlja izvođenje ostatka programa. ECDL ne podržava *do while* petlju ili neke druge egzotične oblike while petlje (*while else*).

```
while(parents.size<100){
    //blok naredbi
}
```

Slika 3.8. Primjer while naredbe

Ponekad je potrebno pojedini odsječak teksta programa izvršiti za svaku jedinku unutar varijable. Za takav zahtjev potrebno je koristiti *for* petlju.Oblik ove petlje prikazan je na slici 3.9. *Population* predstavlja varijablu odnosno multiskup kroz čije je jedinke potrebno iterirati, dok *individual* predstavlja varijablu koja će služiti za pohranu jedinke kojoj se u pojedinoj iteraciji for petlje pristupa. Sve promjene nad jedinkom individual reflektirat će se i promjenama nad tom istom jedinkom u varijabli population.

```
for individual in population {
    //blok naredbi
}
```

Slika 3.9. Primjer for petlje

3.6. Operatori

Kako bi se mogle obavljati pojedine akcije nad varijablama i konstantama potrebni su nam operatori. Operatori se klasično mogu podijeliti na logičke operatore, aritmetičke operatore i slično, no također se mogu podijeliti po tome upotrebljavaju li se nad varijablama, konstantnim izrazima ili logičkim izrazima. U uvjetima petlji i naredbama grananja moguće je koristiti oble zagrade, koje služe da bi se njima povećala čitljivost izraza ili promijenio redoslijed izvršavanja pojedinih operatora.

3.6.1. Operatori pridruživanja, dodavanja i brisanja

Ovi operatori su jedini unutar jezika ECDL koji se upotrebljavaju izvan uvjeta naredbi grananja i petlji. Mogu se koristiti isključivo nad varijablama ili indeksiranim varijablama, ovisno o operatoru. Ovi operatori su svi binarni. Iako se možda na prvi pogled čini kao siromašan skup operatora, treba se podsjetiti da su varijable samo vektori jedinki, i da se pomoću navedenih operatora mogu ostvariti sve bitne operacije nad varijablama koje su programeru potrebne.

Operator pridruživanja („=“) koristi se kako bi se jednoj varijabli pridružila druga varijabla. Vrlo je bitno naglasiti da se varijabli kojoj se pridružuje neka nova vrijednost pridružuje referenca na tu vrijednost. To znači da će obje varijable prikazivati na jedan te isti multiskup i promjene nad jednom varijablom će se reflektirati i na drugu. To se na prvi mah može činiti kao dosta veliki problem, no kada se pogleda alternativa koja je dostupna, ipak postaje jasnije zašto je odabran upravo ovakav pristup. Naime, populacije s kojima rade evolucijski algoritmu mogu biti jako velike, a algoritmi se izvode i s velikim brojem iteracija. Ukoliko bi se prilikom pridruživanja koristilo kopiranje koje bi iz jedne varijable kopiralo sve jedinke u drugu varijablu, tako da one budu međusobno neovisne, to bi svakako uzrokovalo usporenje, koje bi ovisno o broju jedinki koje se kopiraju, bilo minimalno i jedva osjetljivo ili bi uvelike usporilo rad algoritma, što je brojčano potkrijepljeno u poglavlju 4.2. Evolucijski algoritmi su ionako zahtjevni algoritmi čije izvođenje traje dosta dugo, tako da je uvođenje dodatnog kašnjenja kroz jezik ipak nešto što bi se trebalo pokušati izbjeći što je više moguće.

Ukoliko se s lijeve strane operatora pridruživanja nalazi jedinka, s desne strane operatora može se nalaziti neka druga varijabla, pri čemu će onda varijabla na lijevoj strani pokazivati na multiskup na koji pokazuje varijabla na desnoj strani. Osim toga s desne strane se može nalaziti i indeksirana varijabla, pri čemu će onda varijabla na desnoj strani pokazivati na jedinku koja je bila indeksirana, no također se može nalaziti jedan od evolucijskih operatora selekcije, mutacije ili križanja. Ukoliko se na lijevoj strani nalazi indeksirana varijabla, s desne strane se nikako ne može nalaziti obična varijabla budući da bi to izazvalo sintaksnu pogrešku. Ono što se smije nalaziti jest druga indeksirana varijabla, pri čemu se onda indeksu s lijeve strane dodaje jedinka indeksirana na desnoj strani izraza. Primjeri ovih indeksiranja su vidljivi na slici 3.10. Na desnoj strani se smije nalaziti

jedan od operatora selekcije ili križanja, ali ne i mutacije, iz razloga što operator mutacije, u ovisnosti o argumentu kojeg prima može vratiti jednu ili više jedinki. Kako bi se spriječila mogućnost da se indeksiranoj jedinci pokuša dodati više od jedne varijable, onemogućeno je unutar jezika da se na desnoj strani smije pojaviti evolucijski operator mutacije. Ukoliko ga korisnik ipak navede, slučajno ili namjerno, to će jednostavno izazvati sintaksnu pogrešku. Kod selekcija i križanja ovaj problem ne postoji Budući da će ti evolucijski operatori uvijek vratiti točno jednu jedinku.

```
individual=population;

individual[0]=population; // neispravno

individual[0]=population[1];

individual=population[0];
```

Slika 3.10. Primjeri izraza pridruživanja

Operator dodavanja („+=“) koristi se kako bi se u varijablu na lijevoj strani operatora dodale sve jedinke koje se nalaze u varijabli na desnoj strani operatora. Osim varijable na desnoj strani može između ostalog stajati i indeksirana varijabla, pri čemu se onda u varijablu s lijeve strane dodaje jedinka koju se indeksira, ili se može također nalaziti neki od genetskih operatora selekcije, mutacije ili križanja, pri čemu će se u varijablu dodati ona jedinka (odnosno, ako se radi o operatoru mutacije, može biti i više jedinki) koju je zadani genetski operator vratio kao rezultat. Na lijevoj strani uvijek mora stajati varijabla koja ni u kojem slučaju ne smije biti indeksirana. Bitno je za napomenuti da, ukoliko se u jednu varijablu pokuša dodati jedinka koja se već nalazi u toj varijabli, ona će biti ponovno dodana, te će se tako nalaziti dvije reference na istu jedinke unutar multiskupa. Broj istovrsnih jedinki koje se mogu nalaziti unutar iste varijable je ograničen samo memorijom računala. Na slici 3.11. dani su primjeri korištenja operatora dodavanja.

```
individual+=population; //ispravno, dodaje se sve jedinke iz
varijable na desnoj strani

individual+=population[0]; // ispravno, dodaje se samo
indeksirana jedinka

individual[0]+=population; // neispravno, ne lijevoj strani ne
smije se nalaziti indeksirana varijabla

individual[0]+=population[1]; // neispravno, isto kao i proli
primjer
```

Slika 3.11. Primjeri operatora dodavanja

Operator brisanja („-=“) koristi se kako bi se određene jedinke obrisale, odnosno bolje rečeno izbacile iz multiskupa. Sintaksa je ista kao i kod operatora dodavanja, naime na lijevoj strani se nalazi se varijabla iz koje se brišu određene jedinke, a na desnoj strani se nalazi varijabla koja sadrži jedinke koje se brišu iz lijeve jedinke. Ukoliko se u varijabli iz koje se brišu jedinke ne nalazi neka od jedinki koja se nalazi unutar varijable koja sadrži one jedinke koje treba obrisati, te jedinke se jednostavno preskaču i brišu se one koje se mogu pronaći. Naravno na desnoj strani se može nalaziti i indeksirana varijabla, pri čemu se onda samo ona briše iz varijable na lijevoj strani operatora, ili se također može nalaziti genetski operator selekcije, mutacije ili križanja, pa se briše jedinka koju vrati određeni genetski operator. Primjeri operatora brisanja prikazani su na slici 3.12. Operator mutacije i križanja nema baš puno smisla stavljati s desne strane operatora brisanja, budući da će ta dva genetska operatora, u najvećem broju slučajeva, vratiti novu jedinku koja se sigurno neće nalaziti u multiskupa iz koje bi se je trebalo obrisati, i stoga na kraju ta naredba neće imati nikakvih posljedica. Bitno je napomenuti još jedno svojstvo koje je specifično za operator brisanja. To je svojstvo da ukoliko se s desne strane nalazi varijabla koja je indeksirana, ili se nalazi varijabla kojoj je veličina jednaka jedan, onda se iz varijable iz koje se briše, briše samo jedna pojave te jedinke, premda ih u tom multiskupa bilo više. Nasuprot tome, ukoliko je veličina varijable na desnoj strani operatora veća od jedan, onda se brišu sve pojave jedinki u toj varijable iz one varijable koja se nalazi na lijevoj strani operatora brisanja.

```
individual-=population; //ispravno, iz varijable individual
brišu se sve jedinke koje se nalaze i u varijabli population

individual-=population[0]; // ispravno, briše se samo
indeksirana jedinka, ukoliko se ona nalazi u varijabli
individual

individual[0]-=population; // neispravno, ne lijevoj strani ne
smije se nalaziti indeksirana varijabla

individual[0]-=population[1]; // neispravno, isto kao i prošli
primjer
```

Slika 3.12. Primjeri operatora brisanja

3.6.2. Aritmetički operatori

ECDL podržava aritmetičke operatore +, -, *, /, %, pri čemu su im značenja ista kao u programskom jeziku C. Aritmetički operatori se mogu primjenjivati samo nad konstantnim izrazima u koje spadaju cjelobrojni brojevi, brojevi s pomičnim zarezom i svojstvo varijable size. Rezultat aritmetičkih operatora je konstantni izraz. Specifičnost aritmetičkih operatora u ECDL-u je da se oni mogu upotrebljavati samo unutar uvjeta naredbi grananja i programskih petlji. Njihova pojava u bilo kojem drugom dijelu ECDL programa predstavlja sintaksnu pogrešku.

3.6.3. Unarni operatori

Unarni operatori predstavljaju pravu rijetkost u ECDL-u, te u trenutnoj implementaciji postoje samo tri unarna operatora. Od unarnih operatora koji se mogu primijeniti na brojeve razlikujemo dva, a to su: „+“ i „-“. Ta dva operatora imaju značenje predznaka broja ispred kojeg stoje. Postoji mogućnost da se u budućim verzijama ECDL-a izbaci unarni operator „+“, jer se radi o redundantnom operatoru, koji uopće nema neku veliku ulogu unutar samog jezika. Osim operatora koji se mogu primijeniti na brojeve, postoji još jedan unarni operator koji se može primijeniti samo na logičke izraze, a to je operator logičke negacije „ ! “. Njegovo značenje je potpuno analogno onome koje ima u svim višim programskim jezicima, a to je da okreće rezultat evaluacije nekog logičkog izraza, to jest, ako je

izraz bio evaluiran kao istinit, onda će ovaj operator vratiti logičku neistinu i obrnuto, ako je bio evaluiran kao neistinit, onda će vratiti logičku istinu.

3.6.4. Operatori usporedbi i logički operatori

Operatori usporedbe koji su podržani od strane ECDL-a su <, >, <=, >=, ==, != i ?. Svi navedeni operatori su binarni, a o svakom operatoru pojedinačno ovisi može li se primijeniti nad varijablama, konstantnim izrazima ili nad obojem. Rezultat svih navedenih operatora, bez obzira nad čime se primjenjivali, je uvijek logička istina ili laž.

Operatori manje („<“), veće („>“), manje ili jednako („<=“) i veće ili jednako („>=“) mogu se upotrebljavati samo nad konstantnim izrazima. Njihova značenja analogna su njihovim imenima.

Operatori jednakosti („==“) i nejednakosti („!=“) mogu se upotrebljavati nad konstantnim izrazima ali i nad varijablama. Bitno je da su oba operanda istog tipa. Primjerice, ne može jedan operand biti realan broj a drugi varijabla, to će izazvati sintaksnu pogrešku. Ukoliko se operatori primjenjuju nad konstantnim izrazima ili izrazima koji se evaluiraju u njih, uspoređuje se jesu li operandi na lijevoj i desnoj strani jednaki odnosno različiti, ovisno o operatoru koji je upotrijebljen. Pri tome je potrebno paziti pri usporedbi realnih brojeva, zbog toga što računalo ne može uvijek točno pohraniti realne brojeve pa se prilikom njihove usporedbe može dogoditi pogreška, no budući da ECDL nije jezik u kojem bi usporedba realnih brojeva trebala biti učestala, ovaj problem i nije toliko važan niti izražen. Osim što se mogu primijeniti nad konstantnim izrazima, operatori jednakosti odnosno nejednakosti se također mogu upotrijebiti i nad varijablama, kao što je vidljivo na slici 3.8. Pri njihovoj usporedbi uspoređuju se sve jedinke koje su u zadanim varijablama. Provjerava se jesu li sve jedinke na istim pozicijama međusobno jednake, odnosno radi li se o referencama na istu jedinku. Ukoliko se radi o referencama na iste jedinke, ovisno o primijenjenom operatoru, vraća se ili logička istina ili logička laž. Iz navedenog se može zaključiti da ukoliko dvije varijable sadrže potpuno iste jedinke, ali se njihov redoslijed međusobno razlikuje da će operator jednakosti vratiti logičku neistinu, jer taj operator izričito zahtjeva da su jedinke i jednoj i drugoj varijabli poredane istim redoslijedom. Osim toga mogu se uspoređivati i jedinke na točno određenim indeksima, a ne cijelog multiskupa jedinki. Iako sintaksa jezika dozvoljava usporedbu u kojoj je jedan operand

varijabla, a drugi indeksirana varijabla, takva usporedba ima malo smisla, odnosno bolje rečeno uopće ga nema. Naime taj izraz će se uvijek evaluirati kao logička neistina, čak i ako se varijabla koja nije indeksirana sastoji od samo jedne jedinke i ona je u potpunosti jednaka indeksiranoj jedinci druge varijable.

Posljednji operator u navedenoj grupi operatora usporedbi jest operator sadržavanja „?“. Ovaj operator nije preuzet iz nekih drugih programskih jezika, pa stoga može u početku djelovati malo čudno, no zapravo se radi o vrlo jednostavnom operatoru. On jedini spada u skupinu operatora koji se mogu primijeniti samo nad varijablama. Operator provjerava da li jedinke koje sadrži operand na lijevoj strani sadrži i operand koji se nalazi na desnoj strani. Formalnije, provjerava se dali je multiskup na lijevoj strani podskup (ne treba biti pravi podskup) od multiskupa na desnoj strani. Ovome operatoru nije važan poredak jedinki u multiskupovima, već će samo provjeriti sadrži li jedan multiskup drugi, bez obzira na raspored jedinki u njima. Prisjetimo li se kako radi operator usporedbe, možemo vidjeti jedan njegov veliki nedostatak, a to je da on prilikom ispitivanja jednakosti, odnosno nejednakosti zahtijeva da su jedinke poredane u istom redoslijedu u oba operanda. To predstavlja veliki problem ukoliko programer želi samo provjeriti sadrže li dvije varijable iste jedinke, no njihov redoslijed u varijablama im nije bitan. Taj problem se može riješiti primjenom operatora sadržavanja, upravo zbog njegovog svojstva da jedna jedinka ne mora biti pravi podskup druge, što znači da će on vratiti istinu ako obje varijable sadrže iste jedinke. Na taj način riješen je i problem koji se javlja u operatorima usporedbe. Primjer upotrebe operatora sadržavanja je prikazan na slici 3.13.

```
individual==population

individual==population[0] //ispravno, ali besmisleno jer će
uvijek vratiti logičku neistinu

individual[0]!=population[0]

individual?population

individual[0]?population

individual?population[1] // neispravno, na desnoj strani ne
smije se nalaziti indeksirana jedinka
```

Slika 3.13. Primjer logičkih izraza nad varijablama

Logički operatori koje ECDL podržava su logičko I („&&“) i logičko ILI („||“). Logički operatori nad bitovima nisu podržani, niti su oni potrebni. Operandi logičkih operatora moraju biti izrazi koji se na kraju evaluiraju u logičku istinu ili neistinu. To znači da se kao operand logičkog operatora ne može samo staviti jedan broj, jer se brojevi ne evaluiraju u logičku istinu odnosno logičku neistinu, već je potrebno staviti cijeli logički izraz koji će se moći evaluirati. Sami logički operatori se također na kraju evaluiraju u logičku istinu ili neistinu.

3.7. Evolucijski operatori

Evolucijski operatori su operatori koji proizlaze izravno iz evolucijskih algoritama i čiji su mehanizmi sami po sebi inspirirani evolucijskim procesom. Unutar ECDL-a trenutno postoje četiri evolucijska operatora, a to su selekcija (select), križanje (crossover), mutacija (mutate) i evaluacija (evaluate). Svi oni se mogu primjenjivati samo nad jedinkama, ili indeksiranim jedinkama, ovisno o kojem se evolucijskom operatoru radi. Njihov oblik i ponašanje uvelike sliče funkcijama iz programskog jezika C. Naime svaki evolucijski parametar, nakon svog imena, unutar oblih zagrada prima listu parametara (broj parametara je ovisan o evolucijskom operatoru koji se primjenjuje), te nad njima obavlja određenu funkcionalnost. Ovisno o vrsti evolucijskog operatora, on kao rezultat izvođenja može vratiti novu jedinku, ili može napraviti određene promjene nad nekom od jedinka koje je primio

kao parametar. Ovi evolucijski operatori predstavljaju jezgru svakog evolucijskog algoritma.

3.7.1. Operator selekcije

Operatorom selekcije može se odabrati jedinka iz varijable po nekom kriteriju. Vrste selekcija koja se mogu koristiti unutar ECDL-a su proporcionalna selekcija, selekcija najboljeg, selekcija najlošijeg ili slučajna selekcija. Podržani operatori selekcije će se u budućim verzijama ECDL-a moći proširiti proizvoljnim, korisnički definiranim funkcijama. Kao što im i samo ime govori slučajna selekcija odabire i vraća jedinku iz nekog multiskupa slučajnim odabirom, selekcija najboljeg unutar zadanog multiskupa traži jedinku s najboljom dobrotom i vraća nju, dok selekcija najlošijeg unutar zadanog multiskupa traži jedinku s najmanjom dobrotom i vraća upravo nju. Proporcionalna selekcija je malo složeniji oblik selekcije od dosad navedenih oblika. Ona je po načinu odabira najsličnija slučajnoj selekciji, no za razliku od slučajne selekcije gdje sve jedinke imaju istu vjerojatnost da budu odabrane, ovdje vjerojatnost da jedinka bude odabrana izravno ovisi o dobrotama samih jedinki. Osim toga vjerojatnost odabira ovisi i o jednom parametru algoritma koji se naziva selekcijski pritisak, koji govori koliko je najbolja jedinka „bolja“ od najlošije jedinke. Ovisno o vrijednosti selekcijskog pritiska, mijenjat će se i vjerojatnosti pojedinih jedinki da upravo one budu odabrane, tako se može s određenim vrijednostima selekcijskog pritiska može postići da i najlošije jedinke imaju veću vjerojatnost da upravo one budu odabrane. No taj selekcijski pritisak ne navodi se u algoritmu, nego se kao i svi ostali parametri algoritma, navodi u XML datoteci koja sadrži i sve ostale parametre. Oblik svih operatora selekcije je takav da iza riječi select slijedi točka, a odmah iza toga tip selekcije i u zagradama varijabla nad kojom će se izvršiti selekcija.

Operatori selekcije u ECDL-u su: select.best(population), select.worst (population), select.random(population), select.fitnessproportional(population), kao što je prikazano na slici 3.14. Potrebno je na kraju napomenuti da operatori selekcije samo pronalaze određenu jedinku unutar varijable i vraćaju referencu na nju. Selektirana jedinka i dalje ostane sadržana unutar varijable iz koje je selektirana. Ukoliko korisnik želi izbaciti selektiranu jedinku iz tog multiskupa iz koje je ona selektirana, to će morati napraviti s operatorom brisanja.

```
individual=select.random(population); //slučajni odabir
jedinke

individual=select.best(population); //odabir jedinke s
najboljom dobrotom

individual=select.worst(population); //odabir jedinke s
najlošijom dobrotom

individual=select.fitnessproportional(population); // slučajni
odabir s vjerojatnošću ovisnom o iznosu dobrote
```

Slika 3.14. Primjer vrsta selekcije

3.7.2. Operator križanja

Operatorom križanja uzimaju se dvije jedinke koje se nazivaju roditeljima, te se na temelju njih stvara potpuno nova jedinka (nazivamo ju djetetom), koja je nastala kombinacijom svojstava obaju roditelja. Postoji mnogo vrsta križanja, kao križanje s jednom točkom prekida, križanje s n točaka prekida, uniformno križanje i slično. No programer se u trenutku pisanja algoritma u ECDL-u ne mora zamarati detaljima oko toga koje će se križanje koristiti. U trenutačnoj implementaciji jezika operator križanja je generičan i jedinstven, dakle ne postoji mogućnost, kao kod selekcija, da programer detaljnije odabere koje bi se križanje koristilo. No, u budućim verzijama ECDL-a nastojat će se programeru omogućiti definiranje vlastitih i specijaliziranih operatora. Koje će se križanje koristiti bit će specificirano u XML datoteci s parametrima. Isto to vrijedi i za vjerojatnost križanja (ukoliko se navedeni parametar uopće koristi). Taj parametar će biti zapisan u XML datoteci s parametrima, te je programer prilikom pisanja algoritma lišen brige oko te vjerojatnosti. On treba samo specificirati na kojem mjestu u algoritmu bi se trebalo dogoditi križanje, i nad kojim jedinkama bi se ono trebalo provesti.

```
crossover(population[1],population[2],population[0]);

individual=crossover(population[0],population[1]);
```

Slika 3.15. Načini poziva operatora križanja

Postoje dva načina na koja je moguće obaviti križanje, ovisno o zahtjevima programera. Prvi oblik operatora križanja je *crossover(parent[constant1], parent[constant2], child[constant3])*, kao što je prikazano na slici 3.15. Važno je da su svi parametri indeksirane varijable, niti slučajno se ne smije ovdje pojaviti obična varijabla, jer će to izazvati pogrešku. Parametri su redom, prva jedinka koja se uzima kao roditelj, druga jedinka koja se uzima za roditelja i jedinka koja će se zamijeniti s jedinkom djeteta koja je nastala križanjem obaju roditelja. Iako će se posljednja jedinka zamijeniti novom jedinkom koja će nastati kao produkt križanja, neovisno od toga vrlo je bitno da ta indeksirana jedinka pokazuje na neku postojeću jedinku, inače će doći do pogreške. Ovaj oblik operatora križanja mora ne smije se nalaziti unutar nekog izraza, već se mora nalaziti samostalno u retku kao jedna naredba, u suprotnome će se izazvati sintaksna greška, koju će programski prevoditelj prijaviti. Ovo križanje je dobro koristiti ako se želi brže izvođenje algoritma. Brzina se postiže time što će se izmijeniti već postojeća jedinka, te nije potrebno stvarati novu jedinku.

Drugi oblik križanja je *crossover(parent[constant1], parent[constant2])*. Ovo križanje prima dva parametra, a to su jedinke roditelja koji će se iskoristiti za križanje. U ovom obliku križanja je, kao i u prošlom, veoma važno da parametri budu indeksirane varijable, inače će doći do sintaksne greške. Za razliku od prošlog oblika križanja, u ovom križanju se kao rezultat operatora križanja vraća nova jedinka. Upravo zbog toga ovaj oblik operatora križanja ne može stajati samostalno, već se mora nalaziti unutar nekog izraza, i to izraza pridruživanja, izraza dodavanja ili izraza brisanja. Ovaj oblik operatora križanja je sporiji nego što je prethodno navedeni, upravo zbog toga što se promjene neće vršiti nad postojećom jedinkom, već će se vratiti nova jedinka. Stoga je ovaj oblik operatora križanja preporučljivo koristiti onda i samo onda kada programer želi kao rezultat dobiti potpuno novu jedinku, a da pritom sačuva sve stare jedinke.

3.7.3. Operator mutacije

Operatorom mutacije se nad određenom jedinkom ili jedinkama obavljaju slučajne promjene nad njihovom strukturom. Kao što je bio slučaj i kod križanja, ovdje također postoji mnogo vrsta mutacija koje se mogu primijeniti nad jedinkama, no oko toga se programer u trenutku pisanja programa ne mora brinuti,

budući da se vrsta mutacije navodi u XML datoteci s ostalim parametrima. Isto vrijedi i za faktor mutacije koji govori koliko je vjerojatno da će se mutacija dogoditi. Kao što je bilo i kod operatora križanja ovdje postoje također dva oblika operatora mutacije koje korisnik može koristiti, kao što je prikazano na slici 3.16.

```
mutate(individual); // mutacija se obavlja direktno nad
varijablom predanom kao parametar


newIndividual=mutate(individual); //vraća se nova mutirana
varijabla
```

Slika 3.16. Vrste poziva operatora mutacije

Oba oblika mutacije imaju isti izgled, a to je *mutate(population)*. Jedini argument koji ovaj evolucijski operator prima je varijabla čije je jedinke potrebno mutirani. Pri tome se može kao parametar navesti varijabla, pri čemu će se mutirati apsolutno sve jedinke sadržane unutar te varijable ili se može navesti indeksirana jedinka, pri čemu će se mutirati samo ta jedinka koja je bila indeksirana. Oblici mutacije se razlikuju po načinu uporabe ovog operatora, a ne po njegovom izgledu. Naime ako se operator mutacije navede samostalno kao naredba, ne nalazeći se unutar nekog izraza, tada će on mutaciju obaviti eksplicitno nad varijablom, odnosno jedinkama koje su mu predane kao parametar. Nakon mutacija, varijabla predana kao parametar će sadržavati mutirane jedinke. Ukoliko se ovaj operator mutacije navede u nekom izrazu kao izrazu pridruživanja, izrazu brisanja, ili izrazu dodavanja, tada se mutacija neće eksplicitno obavljati nad varijablom koja je predana kao parametar, već će se stvoriti kopija te varijable i tada će se nad njom obaviti mutacija, a kao rezultat tog operatora mutacije vratiti će se kopija te varijable nad kojom je obavljena mutacija. Prema navedenom može se zaključiti kako je prvi oblik operatora mutacije puno brži, jer ne zahtjeva kopiranje multiskupa prije njenog mutiranja, kao što je to slučaj u drugom obliku. Stoga se preporučuje korištenje prvog oblika mutacija, osim kad korisniku nije bitno da sačuva postojeće jedinke nad kojima bi bilo potrebno izvršiti mutaciju.

3.7.4. Operator evaluacije

Iako nije pravi evolucijski operator kao što su bila prethodna tri, evaluacija je navedena ovdje budući da je ipak uvelike vezana uz evolucijske algoritme. Kada dođe do promjena nad pojedinom jedinkom, ili se stvori nova jedinka potrebno je odrediti koliko ta nova ili promijenjena jedinka predstavlja dobro rješenje. Upravo u tu svrhu služi operator evaluacije. Oblik mu je *evaluate(population)* pri čemu se kao parametar može primiti obična varijabla, što povlači za posljedicu da će doći do evaluacije svake pojedine jedinke unutar navedene varijable, ili ukoliko korisnik želi da se evaluira samo jedna jedinka u multiskupu, on može navesti i indeksiranu varijablu s odgovarajućim indeksom jedinke koju je potrebno evaluirati. Po izgledu je dosta sličan operatoru mutacije. Kao jedini argument prima varijablu ili indeksiranu varijablu koju je potrebno evaluirati. Ukoliko se radi o varijabli evaluirat će se sve jedinke unutar varijable, u suprotnom, ako se radi o indeksiranoj varijabli evaluirat će se samo ona indeksirana varijabla. Ovaj operator mora se koristiti kao samostalna naredba, to jest ne smije biti dio nekog većeg izraza. Primjer operatora *evaluate* prikazan je na slici 3.17.

```
evaluate(individual);

evaluate(individual[0]);
```

Slika 3.17. Primjer korištenja operatora evaluacije

3.8. Prednosti i nedostaci ECDL-a

Najveća prednost ECDL-a je svakako to što ubrzava pisanje evolucijskih algoritama. Naime, budući da je ECDL u potpunosti specijaliziran i namijenjen za opis evolucijskih algoritama, njih je unutar ovog metajezika moguće vrlo lako i brzo opisati (prikazano kasnije u primjeru). Također programera se oslobađa od svih nebitnih detalja, odnosno od onih detalja koji nisu bitni za opis evolucijskih algoritama. Tijekom razvoja jezika sve stvari koje se inače ne bi koristile tijekom pisanja evolucijskih algoritama su u potpunosti izbačene iz jezika.

Osim toga programer ne mora direktno poznavati arhitekturu unutar koje se izvode evolucijski algoritmi, dovoljno je da je upoznat s ECDL-om. Naravno da se potreba da programer savlada novi jezik može činiti kao dosta veliki nedostatak.

No ECDL je vrlo jednostavan metajezik, koji nije natrpan bezbrojnim i nepotrebnim opcijama, tako da je taj jezik vrlo lako za naučiti i svladati. Koliko je zapravo ovaj metajezik jednostavan može se vidjeti u kasnijem poglavlju, gdje je jedan genetski algoritam opisan ECDL-om u samo nekoliko linija teksta programa koje su sve, manje više, samoobjašnjavajuće. Osim toga, za pretpostaviti je da je programeru puno lakše naučiti i upoznati se s ovim metajezikom, nego upoznati se s cijelom strukturom okruženja unutar koje se generirani tekst programa izvodi.

Naravno, ovaj jezik kao i mnogi drugi jezici ima mnogo svojih nedostataka. Potrebno je napomenuti da se ovdje trenutno radi o prvoj funkcionalnoj inačici ovog jezika. Kada se pogledaju neki viši programski jezici i kako je tekao njihov razvoj, vidljivo je da su oni svi prošli kroz mnogo verzija, koje su sa sobom donosile mnoge promjene u jezik, ali naravno i mnoga poboljšanja. Stoga ne treba čuditi to što ova prva verzija ima možda dosta nedostataka koje je potrebno što prije riješiti. Naravno da je neke nedostatke nemoguće otkriti i pronaći bez toga da se ovaj jezik prvo ne upotrijebi u nekoj praktičnoj primjeni za neki problem, i da se upravo kroz tu uporabu otkriju još neke slabosti i nedostatci jezika, te da se kritički ocijeni što bi se i kako bi se trebalo poboljšati. Osim toga potrebna su i mišljenja drugih ljudi, posebice onih koji se i sami bave razvojem i primjenom evolucijskih algoritama. Upravo će oni moći najbolje prepoznati nedostatke i izraziti koje bi im dodatne opcije dobrodošle u jeziku.

Jedan nedostatak koji je ponekad više, a ponekad manje izražen je brzina izvođenja generiranog programa. Kao što je i često slučaj kod automatski generiranog teksta programa, brzina njegovog izvođenja je obično manja nego je brzina izvođenja ručno napisanog teksta programa. ECDLC svakako nastoji generirati tekst programa koji je što brži. Trenutno ne postoji neka osjetna razlika između brzine izvođenja ECDL generiranog teksta programa, i ručno napisanog teksta programa. Naravno da korisnik može nakon generiranja teksta programa taj tekst dodatno modificirati i time ubrzati, ukoliko mu je brzina izvođenja stvarno od iznimne važnosti.

Što se programskog prevoditelja tiče, kao jedan njegov nedostatak bi se moglo shvatiti to što trenutačno nema opcija kojim bi se moglo dodatno upravljati njegovim ponašanjem, iznimku jedino predstavlja već ranije navedena jednostavna opcija koja omogućuje korisniku specifikaciju imena izlazne datoteke.

No ovaj programski prevoditelj napravljen je da bude što je moguće jednostavniji, bez nekog velikog skupa opcija. Osim toga, iako programski prevoditelj pruža ispis poruka o leksičkim i sintaksnim greškama koje su pronađene prilikom prevođenja, opis tih pogrešaka bit će nerazumljiv običnom programeru, budući da su opisi tih grešaka bili namijenjeni lakšem pronalasku grešaka prilikom razvoja jezika. Jedna stvar koja bi može pomoći programerima prilikom pronalaženja grešaka je to, što osim opisa greške koja se je dogodila, programski prevoditelj ispisuje i broj retka u kojem se je ta greška dogodila, što omogućuje programerima da lakše pronađu postojeće greške, budući da točno znaju u kojim recima programskog teksta programa se one nalaze.

3.9. Moguća proširenja ECDL-a

Iako je dosad napravljena glavna struktura jezika i izrađen funkcionalni programski prevoditelj, postoji mnogo funkcionalnosti koje bi se mogle dodatno ugraditi u jezik, te također, dodatne funkcionalnosti kojima bi se programski prevoditelj mogao proširiti.

Što se tiče proširenja jezika, jedno od proširenja koje se nameće jest dodavanje novih svojstava jedinkama kao što bi primjerice bila copy (vraća kopiju varijable, a ne referencu) i fitness (vraća dobrotu jedinke). Naravno da uvođenjem novih svojstava dolazi do kompliciranja jezika, za koji se je nastojalo da bude što jednostavniji. Za svojstvo copy je pitanje koliko je ono uopće nužno, budući da postoje evolucijski operatori posebno specijalizirani za to da se vrate nove jedinke, a da originalne ostanu netaknute. Za svojstvo fitness javlja se problem ako se varijabla sastoji od više jedinki, pitanje je što vratiti u tom slučaju. Upravo zbog ovih nedoumica navedena svojstva nisu uključena u prvotnu verziju jezika, te se razmatraju kao moguća proširenja. Osim toga, kroz vrijeme i korištenje jezika pokazat će se postoji li potreba za uvođenjem nekih dodatnih svojstava za varijable ili ne.

Unutar jezika postoji i indirektna varijanta kako usporediti koja jedinka ima veću, odnosno manju dobrotu, a to je da se jedinke čije se dobrote žele usporediti stave u istu varijablu te se onda na toj varijabli primijeni operator selekcije koji će vratiti jedinku s većom, odnosno manjom dobrotom, ovisno o zahtjevu korisnika. Upravo zbog toga, ne uključivanjem svojstva fitness, jezik nije izgubio neku veliku dodatnu

funkcionalnost, jer ovo svojstvo bi zapravo omogućilo jednostavniju usporedbu dobrota pojedinih jedinki.

Dodatno proširenje koje će se sigurno ugraditi do iduće verzije ECDL-a je mogućnost generiranja teksta programa sličnog C++ programskom jeziku. Naime, to što će se generirati biti će jezik koji će se moći koristiti za just-in-time prevođenje algoritma. Na taj način biti će izbjegnuta potreba za ponovnim prevođenjem, nego će se program moći direktno izvesti. Upravo zbog toga će programskom prevoditelju biti potrebno dodati opciju kojom će programer moći odabrati koju vrstu teksta programa želi da programski prevoditelj generira. Ovo proširenje je najprioritetnije od svih planiranih proširenja, upravo zbog toga što postoji verzija ECF-a napisana u C++-u, koja predstavlja puno zrelije i bolje ostvarenje od ECF-a koji je implementiran u programskom jeziku Java.

Osim evolucijskih algoritama postoje još drugi mnogobrojni algoritmi koji spadaju pod metaheuristike. Neki od poznatijih algoritama su optimizacija kolonijom mrava (engl. *ant colony optimization*), optimizacija rojem čestica (engl. *particle swarm optimization*), umjetni imunološki sustav (engl. *artificial immune system*), diferencijska evolucija (engl. *differential evolution*), tabu pretraživanje (engl. *tabu search*). Kao moguće proširenje ECDL-a javlja se upravo mogućnost opisivanje i navedenih metaheuristika. Ovo proširenje predstavlja možda najveće moguće proširenje ECDL-a navedeno u ovom poglavlju. Naravno da zbog toga implementacija ovog proširenja za ECDL predstavlja poveći izazov. Naime navedeni algoritmi se podosta razlikuju od evolucijskih algoritama, tako da bi bilo potrebno uvelike izmijeniti ECDL kako bi se omogućilo njihovo opisivanje. Upravo zbog toga je potrebno dobro razmisliti koje bi se sve algoritme moglo uključiti, i kakav bi utjecaj njihovo uključivanje imalo na ECDL, jer uključivanjem mogućnosti opisa velikog broja algoritama i postupaka, uvelike se povećava kompleksnost samog metajezika.

Mogućnost korištenja korisnički definiranih operatora također bi predstavljalo proširenje koje bi dodatno obogatilo ECDL. Naime korisnik bi unutar ECF-a mogao razviti neki svoj operator, te ga uključiti i iskoristiti u jeziku. To bi korisniku omogućilo da, ukoliko nije zadovoljan pruženim operatorima unutar ECDL-a, može izraditi vlastiti operator i iskoristiti ga u samom jeziku. Naravno da to povlači nove probleme oko toga kako omogućiti programskom prevoditelju da ih ispravno

interpretira i da je u mogućnosti navedene korisnički definirane funkcije pravilno iskoristiti unutar ECDL-a.

Zbog velike zahtjevnosti i dugog izvođenja evolucijskih algoritama traže se mogućnosti njihovog ubrzanja. Jednu mogućnost njihovog ubrzavanja predstavlja upravo paralelizacija. Nažalost, paralelizacija nije nimalo jednostavan koncept i često ju je vrlo zahtjevno primijeniti na neki problem. Upravo zbog toga bilo bi dobro kada bi ECDL omogućavao programeru da na vrlo visokoj razini specificira paralelne evolucijske algoritme. Potrebno je još detaljno vidjeti kako bi se navedena funkcionalnost mogla uopće ugraditi u ECDL, i na koji način bi ona bila dostupna programeru.

Što se programskog prevoditelja tiče, jedno od mogućih proširenja je dodavanje pojedinih opcija. Potrebno je vidjeti koje bi opcije dobrodošle i koje od njih bi olakšale rad, te ih onda uključiti u programski prevoditelj. Naravno, treba izbjeći unošenje nepotrebnih i suvišnih opcija koje će biti vrlo rijetko korištene ili još gore, uopće neće biti korištene. Svakako će kasnije biti potrebno dodati dodatnu opciju programskom prevoditelju, kada će se osim teksta programa jezika Java moći generirati i tekst programa zapisan u nekom drugom programskom jeziku, tako da korisnik odabere u kojem će programskom jeziku biti zapisan izlaz iz programskog prevoditelja.

Kao jedan nedostatak programskog prevodioca navedeno je to da je ispis pogrešaka veoma siromašan i nerazumljiv. Upravo zbog toga bilo bi dobro razviti sustav koji bi bolje opisivao greške koje je korisnik napravio i koje bi mu omogućilo brže pronalaženje i ispravljanje samih grešaka.

Kako bi se programerima olakšalo pisanje programa u ECDL-u, dobro bi bilo razviti barem nekakav dodatak za neki popularni tekst uređivač (primjerice Notepad++), kako bi se programerima automatski označile ključne riječi, konstante, komentari i slične strukture jezika. Upravo jedan takav dodatak za popularni tekstualni uređivač Notepad++ je trenutno u razvoju. Svi programski primjeri pisani su u testnoj verziji tog dodatka, tako da je otprilike moguće dobiti dojam kako će bojanje sintakse jezika na kraju izgledati. Naravno, bolje od toga bilo bi da se razvije jednostavno razvojno okruženje za ECDL, koje bi onda programeri mogli koristiti za pisanje ECDL programa i za njihovo prevođenje u

neki drugi jezik. Naravno da to razvojno okruženje ne bi podržavalo sve napredne opcije koje sadržava sva moderna razvojna sučelja, već bi ono imalo samo najosnovnije opcije za pisanje algoritma u ECDL jeziku.

4. Primjeri evolucijskih algoritama zapisanih u ECDL-u

4.1. Primjer ECDL programa i generiranog Java programa

U ovom poglavlju prikazan je primjer genetskog algoritma zapisanog u ECDL-u, kao i primjer teksta programa u jeziku Java koji je dobiven nakon prevođenja tog algoritma programskim prevoditeljem. Radi se o jednostavnom genetskom algoritmu, koji koristi troturnirsku eliminacijsku selekciju. To znači da se iz populacije svih jedinki nasumično odabiru tri jedinke, od kojih se najbolje dvije jedinke (roditelji) odabiru za operaciju križanja. Jedinka dobivena križanjem (dijete) zamjenjuje treću, odnosnu najlošiju jedinku od onih koje su bile odabrane. Nadalje nad tom jedinkom se provodi operator mutacije, i dobivenoj jedinci se na kraju evaluira njezina dobrota. Ovaj postupak ponavlja se sve dok nije zadovoljen jedan od uvjeta zaustavljanja genetskog algoritma. Uvjet zaustavljanja, kao i veličina populacije i faktor mutacije zapisani su u vanjskoj XML datoteci sa svim ostalim parametrima algoritma.

```
repeat(population.size){
tournament.clear;

repeat(3){
tournament+=select.random(population);
}

worst=select.worst(tournament);
tournament-=worst;

crossover(tournament[0],tournament[1],worst[0]);
mutate(worst);
evaluate(worst);
}
```

Slika 4.1. Genetski algoritam zapisan u ECDL

Iz primjera na slici 4.1. vidljivo je kako je genetski algoritam zapisan u ECDL-u vrlo koncizan i jednostavan. Naime čitav algoritam opisan je samo u nekoliko linija ECDL teksta programa. Ovaj tekst programa je veoma lako čitljiv čak i ljudima koji

se nisu prethodno puno susretali s evolucijskim algoritmima, dok nešto iskusniji mogu iz ovog teksta programa gotovo trenutno vidjeti o kakvom se evolucijskom algoritmu tu radi. Moglo bi se reći da je ovaj tekst programa po razini apstrakcije gotovo na razini pseudokoda. Naime programer mora samo specificirati tijek evolucijskog algoritma, dok detalji kako se to sve odvija u pozadini programeru uopće nisu potrebni. Kao što je kroz ovaj rad već nekoliko puta naglašeno, a što je sad vidljivo i kroz ovaj isječak teksta programa, parametri nisu nigdje specificirani unutar samog algoritma. Također, vidljivo je u tekstu programa da su korišteni evolucijski operatori koji ne vraćaju nove jedinke, već oni koji promjene izvršavaju direktno na predanim im jedinkama. Naravno da su se u algoritmu mogli iskoristiti i oblici evolucijskih operatora koji vraćaju nove jedinke, no to bi za posljedicu imalo to da bi se algoritam izvodio nešto sporije nego što se izvodi ovako. Upravo zbog toga je preporučljivo koristiti evolucijske operatore križanja i mutacije koji rade na predanim im jedinkama kad god je to moguće i što je češće moguće. Na taj način neće doći do nepotrebnog usporavanja algoritma, do kojeg bi došlo da korisnik koristi druge oblike evolucijskih operatora križanja i mutacije. Između ostalog vidljivo je kako je sama sintaksa jezika vrlo slična sintaksi popularnih programskih jezika, upravo s tom namjerom da se programeri lakše i brže mogu prilagoditi ECDL-u.

Kao što je vidljivo iz slika 4.2. i 4.3., ECDL generira dobar dio teksta programa koji skoro pa i ne ovisi o samom evolucijskom algoritmu. Tekst programa koji je izravno ovisan o opisanom evolucijskom algoritmu u ECDL programu je tekst programa generiran unutar advanceGeneration metode. Naravno, nije to sad tekst programa koji programer samo ne bi mogao napisati, no tekst programa je na prvi pogled dosta nepregledniji od teksta programa napisanog u ECDL-u. Također pri pisanju teksta programa u Javi, veća je vjerojatnost da će se programeru potkrasti poneka greška u algoritam, nego da isti piše u ECDL-u. Osim toga, ukoliko programer piše program direktno u Javi, ipak bi trebao imati neka osnovna znanja o ECF okruženju, dok ga pisanje tog programa u ECDL-u oslobađa te brige.

```
import java.util.Vector;
import ecf.Deme;
import ecf.Individual;
import ecf.State;
import ecf.selection.SelectionOperator;
import ecf.selection.SelRandomOp;
import ecf.selection.SelWorstOp;

public class SteadyStateTournament extends Algorithm {

	private int nTournament;
	private SelectionOperator selRandomOp;
	private SelectionOperator selWorstOp;

	public SteadyStateTournament(State state) {
		super(state, "SteadyStateTournament");

		selRandomOp = new SelRandomOp();
		selectionOp.add(selRandomOp);

		selWorstOp = new SelWorstOp();
		selectionOp.add(selWorstOp);

	}

	public void initialize() {
		nTournament =
Integer.valueOf(getParameterValue("tsize"));
		if (nTournament < 3) {
			String poruka = "Error: SteadyStateTournament
algorithm requires minimum tournament size of 3!";
			state.getLogger().log(1, poruka);
			throw new IllegalArgumentException(poruka);
		}

		for (int i = 0; i < selectionOp.size(); i++) {
			selectionOp.get(i).initialize();
		}
	}

	public void registerParameters() {
		registerParameter("tsize", "3");
	}
```

Slika 4.2. Generirani Java tekst programa (1)

```
	public void advanceGeneration(Deme population) {

		Vector<Individual> helper = new Vector<Individual>();
		Vector<Individual> worst = new Vector<Individual>();
		Vector<Individual> tournament = new Vector<Individual>();


		for (int i0 = 0; i0 < population.size(); i0++) {
			tournament.clear();

			for (int i1 = 0; i1 < 3; i1++) {
				tournament.add(selRandomOp.select(population));
			}

			worst.clear();
			worst.add(selWorstOp.select(tournament));

			if (worst.size() == 1)
				removeFrom(worst.get(0), tournament);
			else
				tournament.removeAll(worst);

			mate(tournament.get(0), tournament.get(1),
worst.get(0));
			mutate(worst);

			for (int i2 = 0; i2 < worst.size(); i2++) {
				worst.get(i2).evaluate();
			}
		}

	}

}
```

Slika 4.3. Generirani Java tekst programa (2)

4.2. Usporedba algoritma s kopiranjem i bez kopiranja jedinki

Kao što je već nekoliko puta spomenuto kroz ovaj rad, postoji velika vremenska razlika između izvođenja algoritama s, i bez kopiranja jedinki. Budući da su evolucijski algoritmi veoma zahtjevni i njihovo izvođenje može trajati dosta dugo, uvođenje dodatnog usporenja nije poželjno. Stoga je potrebno minimizirati i otkloniti nepotrebna usporenja. Zbog toga je tijekom razvoja jezika bilo potrebno usporediti vrijeme izvođenja algoritama koji koriste kopiranje jedinki i algoritama koji rade s referencama i odlučiti hoće li operatori raditi nad referencama ili će raditi nad kopijama jedinki.

U ovom potpoglavlju demonstrirat će se vremenska razlika između genetskog algoritma u kojem se koriste evolucijski operatori koji rade sa referencama i genetskog algoritma koji koristi evolucijske operatore koji rade nad kopijama jedinki. Rezultati prikazani u ovom poglavlju nastojat će argumentirati odluku da svi operatori koji su ugrađeni u jezik ECDL rade nad referencama, umjesto nad kopijama. Osim toga prikazat će se koliko programeri moraju biti oprezni kada koriste operatore s kopiranjem jedinki, jer oni mogu nepotrebno usporiti program, ukoliko se koriste na mjestima gdje bi se mogli koristiti operatori koji rade nad referencama.

Programski tekst sa slike 4.1. iz prošlog potpoglavlja služit će u usporedbi kao algoritam koji koristi reference. Budući da je ovaj program već objašnjen, neće se ponovno objašnjavati u ovom potpoglavlju. Slika 4.4. predstavlja tekst programa koji koristi kopiranje jedinki. Usporedbom ova dva programa vidljivo je da postoje minimalne razlike između njih. Naime razlike se svode na to da se koriste evolucijski operatori koji rade nad kopijama varijabli. Osim toga potrebne su još dodatne dvije naredbe koje se koriste za otklanjanje jedinke iz multiskupa, odnosno za njeno dodavanje u multiskup.

```
repeat(population.size){
tournament.clear;

repeat(3){
tournament+=select.random(population);
}

worst=select.worst(tournament);
tournament-=worst;
population-=worst;

worst=crossover(tournament[0],tournament[1]);
worst=mutate(worst);
evaluate(worst);

population+=worst;
}
```

Slika 4.4. Algoritam sa kopiranjem jedinki

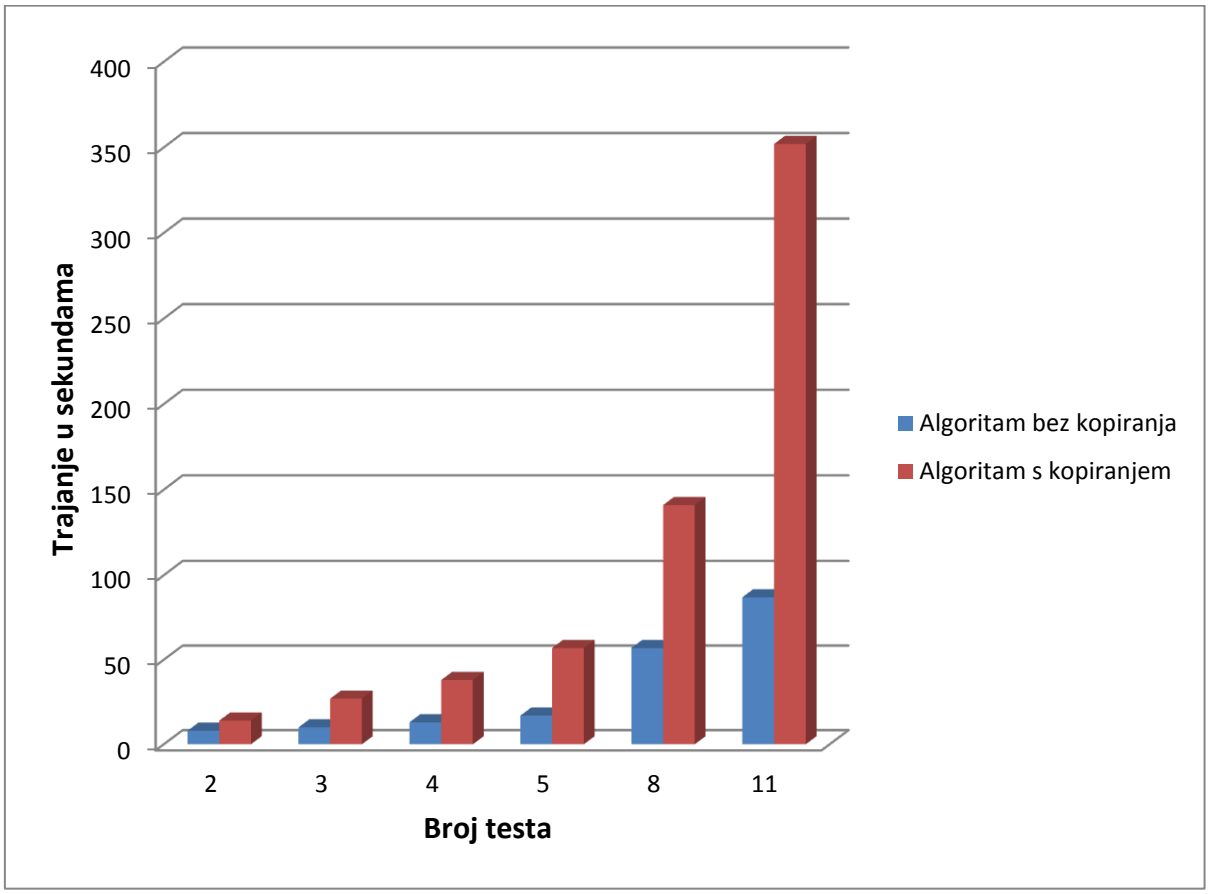
Algoritmi su se koristili za rješavanje problema pod nazivom „problem dvije kutije“ (engl. „*Two boxes problem*“). Jedinke su predstavljene kao stabla, minimalne dubine od dvije razine, a maksimalne dubine od 5 razina. Korišteno je uniformno križanje jedinki, s vjerojatnošću od 0.5 i mutacija zamjene čvorova, također s vjerojatnošću od 0.5. Kao uvjet zaustavljanja korišten je broj iteracija algoritma.

Oba algoritma testirana su na računalu s četverojezgrenim procesorom i7 920 @2.67 GHz, i 6 Gb RAM-a. Algoritmi su istestirani na 13 različitih kombinacija parametara. Svaki vremenski rezultat dobiven je kao prosjek od deset testiranja za pojedini parametar. Za testiranje je korištena razvojna okolina Eclipse, kao i okruženje ECF implementirano u programskom jeziku Java.

Tablica 4.1. Usporedba vemena izvođenja algoritama

Redni broj	Parametri	Vrijeme izvođenja algoritma s referencama	Vrijeme izvođenja algoritma s kopiranjem
1.	Populacija = 100 Broj iteracija = 300	3.077 sekundi	4.026 sekundi
2.	Populacija = 300 Broj iteracija = 300	8 sekundi	14. 865 sekundi
3.	Populacija = 500 Broj iteracija = 300	10.313 sekundi	27.098 sekundi
4.	Populacija = 700 Broj iteracija = 300	13.784 sekundi	38.685 sekundi
5.	Populacija = 1000 Broj iteracija = 300	17,178 sekundi	51.474 sekundi
6.	Populacija = 1000 Broj iteracija = 500	33.350 sekundi	83.479 sekundi
7.	Populacija = 1000 Broj iteracija = 1000	63.742 sekundi	129.226 sekundi
8.	Populacija = 3000 Broj iteracija = 300	57.413 sekundi	141.433 sekundi
9.	Populacija = 3000 Broj iteracija = 500	83.854 sekundi	234.478 sekundi
10.	Populacija = 3000 Broj iteracija = 1000	170.750 sekundi	10 minuta i 13 sekundi
11.	Populacija = 5000 Broj iteracija = 300	87.627 sekundi	5 minuta i 52 sekunde
12.	Populacija = 5000 Broj iteracija = 500	159.780 sekundi	11 minuta i 26 sekundi
13.	Populacija = 5000 Broj iteracija = 1000	307.984 sekundi	23 minute i 57 sekundi

Kao što je vidljivo iz tablice 4.1. postoji pozamašna razlika između vremenskog izvođenja algoritama s, i bez kopiranja jedinki. Ta razlika postaje veća kako se povećava populacija jedinki, što je i razumljivo. Povećanjem populacije povećava se broj kopiranja u algoritmu i tako se uvodi usporenje u sam program. Za zadnji slučaj vrijeme izvođenja, korištenjem kopiranja jedinki, je trajalo čak preko 20 minuta. Za ovakav jednostavan problem to je neprihvatljivo. Zbog toga odlučeno je da će se u jezik ugraditi operatori koji će raditi isključivo nad referencama, a ne nad kopijama. Jedina iznimka su evolucijski operatori mutacije i križanja, zbog razloga što je ponekad neophodno raditi nad kopijama jedinki. Programeru je ostavljeno na odabir koje će evolucijske operatore koristiti, no iz ovog primjera trebalo bi biti vidljivo da se korištenje operatora koji rade s kopijama ne preporučuje, jedino ako njihovo korištenje stvarno nije neophodno. Na slici 4.5. grafički je prikazana usporedba trajanja svih testova od 300 iteracija.



Slika 4.5. Usporedba vremena izvođenja algoritama

4.3. Primjer generacijskog algoritma s proporcionalnom selekcijom

U ovom poglavlju prikazan je primjer generacijskog evolucijskog algoritma u kojem se koristi proporcionalna selekcija. Na slici 4.6 prikazan je zapis tog algoritma u ECDL-u. Kao što je vidljivo iz primjera radi se o vrlo jednostavnom algoritmu koji najprije proporcionalnom selekcijom odabire jedinke i sprema ih u posebnu varijablu. Važno je napomenuti da je pri tome moguće da se u novoj varijabli pojavi više referenci na istu jedinku. Nakon odabira određenog broja jedinki, u novoj petlji se pomoću nasumične selekcije odabiru dvije jedinke iz varijable *pool*, koje se koriste za križanje. Bitno je primijetiti kako se koristi križanje koje vraća novu jedinku. Jedinke dobivene križanjem se nakon toga mutiraju, pri čemu je ovdje sasvim dovoljno koristiti mutaciju koja direktno mijenja originalnu jedinku. Mutirane jedinke se dodaju u novu varijablu, a jedinke koje su bile odabrane za križanje se brišu iz varijable *pool*. Kada druga petlja završi s radom u varijablu *nova* dodaju se jedinke iz varijable *pool* i nakon toga se u varijablu *population* stavljaju jedinke iz varijable *new*.

```
repeat(population.size) {
    pool += select.fitnessproportional(population);
}

repeat(population.size * 0.3) {
        r1 = select.random(pool);
        r2 = select.random(pool);

        d1 = crossover(r1[0], r2[0]);
        d2 = crossover(r1[0], r2[0]);

        mutate(d1);
        mutate(d2);

        nova += d1;
        nova += d2;

        pool -= r1;
        pool -= r2;
}

nova += pool;
population = nova;
```

Slika 4.6. Generacijski evolucijski algoritam sa proporcionalnom selekcijom

4.4. Primjer složenijeg algoritma

U ovom poglavlju dan je jedan malo duži evolucijski algoritam zapisan u jeziku ECDL. Iako algoritam možda nema previše smisla primijeniti u praksi, ipak služi kao dobar primjer kojim se nastoje pokazati mogućnosti koje pruža ECDL i da se jezik može primijeniti na izradu kompliciranijih evolucijskih algoritama od onih koji su do sada prikazani u ovom poglavlju. Tekst programa algoritma prikazan je na slici 4.7. U primjeru su vidljiva neka svojstva jezika koja su već u ranijim poglavljima objašnjena (postojanje naredbi grananja, neovisnost operatora mutacije i evaluacije o broju jedinki unutar varijable parametra).

Ovdje je dobro istaknuti jednu moguću pogrešku koja bi se vrlo često mogla javljati prilikom pisanja programa. Naime, programer mora paziti da sa svojstvom *clear* očisti varijable kada je to potrebno. Ukoliko programer to ne bi napravio, dogodilo bi se to da bi se u toj varijabli nakupljalo sve više i više jedinki, što može dovesti do usporenja programa, pa čak i beskonačnih petlji. U ovom primjeru to bi se dogodilo ako bi programer na kraju petlje zaboravio pozvati svojstvo *clear* nad varijablom *pop1* ili *pop2*.

Druga greška, koja možda nije toliko česta, je da programer u jednu varijablu stavi više referenci na istu jedinku, i obriše tu referencu iz originalne varijable. Kada programer želi dodati sve jedinke iz te nove varijable nazad u originalnu varijablu, dogodit će se to da će se u varijablu dodati sve reference na jedinku koja je izbrisana i da će se tako povećati veličina originalne varijable. To posljedično može prouzročiti neprestan rast veličine varijable i ulazak u beskonačnu petlju. Zbog toga programer mora biti oprezan kada radi s operatorima brisanja i dodavanja, kako ne bi napravio sličnu pogrešku.

```
repeat(2*population.size){

     repeat(population.size/10){
          pop1+=select.best(population);
          population-=pop1;
     }

     repeat(population.size/10){
          pop2+=select.random(population);
          population-=pop2;
     }

     if(pop1.size>3){

          repeat(pop1.size){

               tournament=select.random(pop1);
               tournament+=select.random(pop1);

               worst=select.worst(pop1);

               crossover(tournament[0], tournament[1],
               worst[0]);

               mutate(worst);
               evaluate(worst);

          }
     }

     mutate(pop2);
     evaluate(pop2);

     population+=pop2;
     population+=pop1;

     pop1.clear;
     pop2.clear;

}
```

Slika 4.7. Tekst programa algoritma

5. Zaključak

Programski jezici viših razina, odnosno programski jezici usko specijalizirani za rješavanje pojedinog problemskog područja, programeru uvelike olakšavaju pisanje programa koji spadaju u domenu tog problemskog područja, upravo svojom prilagođenošću rješavanju problema, kao i oslobađanjem samih programera gotovo čitave brige o detaljima koji zapravo njima prilikom rješavanja problema nisu bitni, i koji nisu povezani s problemom kojeg sami programeri rješavaju.

ECDL je jezik koji je razvijen s ciljem da programerima olakša razvoj evolucijskih algoritama. Operatori, varijable, konstante i sve ostalo unutar jezika prilagođeno je, što je više moguće, razvoju evolucijskih algoritama. Bez obzira na to, jezik pokušava programerima ostaviti što je više fleksibilnosti prilikom pisanja evolucijskih algoritama, upravo zbog toga da se programeri ne osjećaju ograničeni jezikom.

Iako je ECDL potpuno funkcionalan i spreman za korištenje, to naravno ne znači da je on u potpunosti razvijen. U samom jeziku i prevoditelju postoji nedostataka koje je potrebno ispraviti, no također postoji i mnogo mogućnosti za proširenja. Sve mogućnosti i dodatne opcije nije dobro uključiti u jezik, nego je kroz vrijeme potrebno vidjeti koji elementi nedostaju ECDL-u i programskom prevoditelju. Tako će se razaznati nedostaci koji su uistinu bitni prilikom razvoja evolucijskih algoritama u ECDL-u, te će se moći pristupiti rješavanju upravo tih nedostataka.

6. Literatura

[1] Evolutionary System Definition Language, stevedower.id.au/esdl, 7. 5. 2012.

[2] StreamIt, groups.csail.mit.edu/cag/streamit/, 3. 5. 2012.

[3] Tabli, E. G. Metaheuristics from design to implementation. WILLEY. 2009.

[4] Open BEAGLE, a versatile EC framework, http://beagle.gel.ulaval.ca/, 3. 5. 2012.

[5] Evolutionary Algorithm Modeling Language, sourceforge.net/projects/eaml/, 3. 5. 2012.

[6] Dower, S., Woodward, C. J. ESDL: A Simple Description Language for Population-Based Evolutionary Computation, 2011

[7] Dower, S., Woodward, C. J. Evolutionary System Definition Language, 2010

[8] Dower, S., Specifying Ant System in ESDL, 2011

[9] Dower, S., Woodward, C. J. Specifying Differential Evolution in ESDL, 2010

[10] Dower, S., Woodward, C. J. Specifying Particle Swarm Optimization in ESDL 2010

[11] Dower, S. ESDL Multiblock Extension Proposal, 2010.

[12] ECF – Evolutionary Computation Framework, http://gp.zemris.fer.hr/ecf/ , 3. 5. 2012.

7. Sažetak

Naslov: Metajezik za opis evolucijskih algoritama.

U ovom radu predstavljen je metajezik za opis evolucijskih algoritama (ECDL), te je dodatno implementiran programski prevoditelj koji program zapisan u ECDL-u prevodi u program zapisan u Javi. Dobiveni Java program moguće je izvesti u sklopu ECF-a u kojem su definirani svi operatori i sve ostale strukture koje su potrebne za izvođenje evolucijskih algoritama. Svrha jezika je uvođenje veće razine apstrakcije u opis evolucijskih algoritama, kako bi se na taj način olakšao i ubrzao njihov razvoj, te kako bi se programera oslobodilo brige oko nebitnih detalja i na taj način smanjila mogućnost pojave pogrešaka prilikom razvoja evolucijskih algoritama. Nadalje planiran je daljnji razvoj i uvođenje novih opcija i mogućnosti u sam ECDL i programski prevoditelj.

Ključne riječi: metajezik, evolucijski algoritmi, genetski algoritmi, genetsko programiranje, evolucijska strategija, evolucijsko programiranje, ECDL.

8. Abstract

Title: Evolutionary computatopn description language.

The Evolutionary computation description language (ECDL) and the implemented compiler for translating programs written in the ECDL to programs written in the Java programming language are described in this paper. The generated Java program can be run with the Evolutionary Computation Framework (ECF), in which all the required operators and data structures, for running the evolutionary algorithm, are defined. The goal of this language is to introduce a new level of abstraction into the description of evolutionary algorithms, making their design easier and much faster, and also relieving the programmer of the concerns accomapanied with unimportant implementation details, thus lowering the possibility of errors during the design of evolutionary algorithms. Furthermore, the development of the language and compiler will continue, and new features will be added to both the compiler and the language itself.

Keywords: metalanguage, evolutionary algorithms, genetic algorithms, genetic programming, evolutionary strategy, evolutionary programming, ECDL.