SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4020

OSTVARENJE METAJEZIKA ZA EVOLUCIJSKE ALGORITME U OKRUŽENJU ZA EVOLUCIJSKO RAČUNANJE

Luka Križan

Zagreb, svibanj, 2015.

ZAHVALA

Zahvaljujem se svojim roditeljima na stalnoj podršci u svemu u životu i stalnim poticajima da do maksimuma pokušam iskoristiti svoje potencijale.

Također se zahvaljujem svim svojim prijateljima (posebno Belli, Eni i Petri) koji su uvijek bili uz mene te mi pomagali kad je bilo najpotrebnije.

Na kraju, želim se zahvaliti svom mentoru Domagoju Jakoboviću jer je u meni probudio izrazito zanimanje za ovo područje te za sve savjete i pruženu pomoć prilikom izrade ovog rada.

Svima vam od srca hvala.

Luka Križan

Sadržaj

1. Uvod

Još od prvih civilizacija, ljudi su izrađivali pomagala za rješavanje problema, prvenstveno za računanje, što je bila i prva motivacija za izum i razvoj računala. Napretkom znanosti i tehnologije povećao se broj i raznolikost problema koje je moguće riješiti uporabom računala. Međutim, i dalje postoji velik broj problema koji je moguće u potpunosti točno riješiti isključivo pretragom cijelog prostora rješenja (npr. problem trgovačkog putnika, problem sastavljanja rasporeda, traženje ekstrema nederivabilnih funkcija...), što je unatoč velikoj brzini današnjih računala teško rješivo u razumnom vremenu. Kao alternativa, pojavile su se heurističke metode koje ne garantiraju pronalaženje optimalnog rješenja, ali omogućuju pronalazak dovoljno dobrog rješenja za praktičnu primjenu uz znatno kraće vrijeme rješavanja. Evolucijski algoritmi, kao grana metaheuristike, pokazali su se učinkovitim u rješavanju navedenih i sličnih problema. S ciljem pojednostavljenja programiranja evolucijskih algoritama razvijaju se okruženja (npr. ECF [1]) u kojima postoje programske izvedbe potrebnih funkcija i operatora. S ciljem dodatnog pojednostavljenja programiranja i oslobađanje programera od predznanja o okruženju za izvođenje evolucijskih algoritama, nastao je ECDL [2], metajezik za opis evolucijskih algoritama, kojim je algoritme moguće opisati na način vrlo sličan pisanju pseudokoda.

S ciljem integracije ECDL-a i C++ verzije ECF-a, u sklopu ovog rada nastao je međujezik za opis evolucijskih algoritama - EIL (engl. *Evolutionary Intermediate Language*) te potrebna programska potpora koja omogućuje interpretaciju algoritama napisanih u EIL-u u vremenu usporedivom vremenu izvođenja algoritama izravno napisanih u jeziku C++ za ECF.

U drugom poglavlju ovog rada dan je kratak uvod u evolucijske algoritme. U trećem poglavlju dan je detaljan opis jezika EIL s primjerima prevođenja izraza iz ECDL-a u EIL. U četvrtom poglavlju dan je opis programskog prevoditelja iz ECDL-a u EIL te interpretera EIL-a. U petom poglavlju dani su primjeri evolucijskih algoritama napisanih u ECDL-u i EIL-u te vremenska usporedba izvođenja s algoritmima izravno napisanim za ECF na različitim problemima.

2. Evolucijski algoritmi

Evolucijske algoritme (EA) definiramo kao podskup metoda evolucijskog računanja, odnosno, metaheurističkih optimizacijskih metoda. Koriste mehanizme inspirirane prirodom, poput selekcije, reprodukcije, križanja i mutacije. Evolucijski algoritmi rade nad populacijom rješenja – svako potencijalno rješenje predstavlja jednu jedinku u populaciji nad kojom se obavljaju prethodno navedeni mehanizmi. Bolje jedinke preživljavaju i postaju dio iduće generacije, a lošije jedinke odumiru, odnosno, prostor oko boljih rješenja se više pretražuje. Da bi se moglo razlikovati koje jedinke predstavljaju bolja, a koje lošija rješenja, nad njima se provodi funkcija dobrote (engl. *fitness function*). Zbog visoke razine apstrakcije, evolucijski algoritmi primjenjivi su na razne vrste problema (problem trgovačkog putnika, izrada rasporeda, traženje ekstrema funkcija...) te na razna područja (biologija, ekonomija, marketing, umjetnost,...). Postoji mnogo oblika evolucijskih algoritama, no najčešće korišteni su genetski algoritmi, genetsko programiranje, evolucijsko programiranje i evolucijska strategija.

2.1 Genetski algoritmi

Genetske algoritme 1970-ih godina razvio je John H. Holland. [3]. Oni su najčešće korišten tip EA. Struktura podataka jedinki oblikuje se prema problemu koji se rješava, a najčešće ju čini niz brojeva (tradicionalno, niz binarnih znamenki). Genetski algoritmi najčešće se primjenjuju na probleme optimizacije. Struktura genetskog algoritma prikazana je na slici 2.1 [4]. Kako bi bolje jedinke preživjele i postale dio sljedeće generacije, provodi se selekcija. Time se prostor pretraživanja usmjeruje na bolja rješenja. Križanje se provodi nad dvije jedinke (roditelji), čime nastaje nova jedinka slična roditeljima (dijete). Mutacijom se unose slučajne promjene u populaciju, čime se osigurava raznolikost jedinki i sprječava zapinjanje algoritma u nekom od lokalnih optimuma. Najčešći uvjeti završetka evolucijskog procesa su prolazak određenog broja iteracija, pojavljivanje jedinke s dovoljnom vrijednosti funkcije dobrote, vremensko ograničenje i sl.

```
t = 0;
generiraj početnu populaciju potencijalnih rješenja P(0);
sve do zadovoljenja uvjeta završetka evolucijskog procesa
{
      t = t + 1;
      selektiraj P'(t) iz P(t-1);
      križaj jedinke iz P'(t) i spremi djecu u P(t);
      mutiraj jedinke iz P(t);
}
ispiši rješenje;
```

Slika 2.1 – struktura genetskog algoritma [4]

2.2 Genetsko programiranje

Genetsko programiranje razvio je John R. Koza [3]. Specifičnost genetskog programiranja leži u činjenici da jedinke ne predstavljaju rješenja danog problema, nego programe za rješavanje problema. Programi se prikazuju u obliku strukture stabla, gdje unutarnji čvorovi predstavljaju operatore, a listovi predstavljaju operande. Genetsko programiranje najčešće se koristi prilikom strojnog učenja.

2.3 Evolucijska strategija

Evolucijsko strategiju razvili su Ingo Rechenberg i Hans-Paul Schwefel 1960-ih godina [3]. Navedena metoda najčešće koristi samo operatore mutacije i selekcije. Budući da se jedinke u evolucijskoj strategiji najčešće izvode kao vektori realnih brojeva, mutacija se izvodi pridodavanjem nasumične vrijednosti normalne distribucije svakoj komponenti vektora. Osim vektora realnih brojeva, jedinke sadrže i druge parametre koje utječu na sam algoritam (primjerice, vjerojatnost mutacije), čime evolucijom rješenja dolazi i do evolucije samih parametara.

2.4 Evolucijsko programiranje

Evolucijsko programiranje je 1960-ih razvio Lawrence J. Fogel [3]. Slično je genetskom programiranju – jedinke su programi nad kojima se provodi samo mutacija, ali se time mijenjaju isključivo njihovi numerički parametri.

3. Međujezik za opis evolucijskih algoritama

ECDL, metajezik za opis evolucijskih algoritama, razvijen je s ciljem pojednostavljenja programiranja evolucijskih algoritama. Vrlo je jednostavan i razumljiv te za njegovo korištenje programer ne mora imati nikakva predznanja o okolini u kojoj se evolucijski algoritmi zapravo izvode. Originalno, programski prevoditelj za ECDL podržavao je isključivo prevođenje u tekst programa jezika Java za razvojno okruženje ECFJ (engl. *Evolutionary Computation Framework in Java*). [2] Budući da postoji potreba za integraciju ECDL-a sa C++ verzijom ECF-a, u sklopu ovog rada razvijen je međujezik u koji se prevode izvorni programi napisani u ECDL-u te se time omogućuje njihova interpretacija unutar razvojnog okruženja ECF. U nastavku je dan kratak opis ECDL-a te detaljan opis međujezika s primjerima prevođenja iz ECDL-a.

3.1 Opis metajezika

ECDL je specijalizirani jezik za pisanje programa evolucijskih algoritama. Sintaksno je vrlo sličan višim programskim jezicima poput C-a ili Jave – svaka naredba završava se oznakom kraja naredbe „ ; “, između jezičnih elementa slobodno se mogu dodavati praznine te postoji mogućnost pisanja jednolinijskih i više linijskih komentara. Budući da je usko specijaliziran, sadrži samo funkcionalnosti potrebne za pisanje evolucijskih algoritama. Zbog toga, podržava samo jedan tip varijabli – skupove jedinki, te dva tipa konstanti – cjelobrojne i realne konstante. Varijable imaju dva svojstva – *size* (kojim se dobiva broj jedinki u varijabli) i *clear* (kojim se brišu sve jedinke iz varijable). Pojedinoj jedinki unutar varijable pristupa se preko indeksa. Unutar jezika postoji posebna varijabla – *population*, koja predstavlja inicijalni skup jedinki u evolucijskom algoritmu. Naredbe grananja (*if*, *else*) i naredbe petlji (*for, while*) izvedene su na isti način kao i u višim programskim jezicima, uz iznimku naredbe *repeat*, koja je zamišljena kao naredba petlje za točno određeni broj ponavljanja. Unutar jezika postoje posebni operatori pridruživanja („=“), dodavanja („+=“) te brisanja („-=“) kojima je moguće sve jedinke iz neke varijable ili jedinke točno određene indeksom dodati ili ukloniti iz druge varijable. Aritmetički operatori (+, -, *, /, %), logički operatori (||, &&, !) i operatori usporedbe (<, >, >=, <=, ==, !=) imaju istu funkciju kao i u višim

programskim jezicima. Jedna od posebnosti jezika je novi operator usporedbe „?“ – operator sadržavanja. Njime je moguće provjeriti jesu li sve jedinke ili točno određena jedinka neke varijable sadržane u drugoj varijabli. Drugu posebnost jezika čine evolucijski operatori – operatori selekcije (*select*), križanja (*crossover*), mutacije (*mutate*) i evaluacije (*evaluate*). Navedeni evolucijski operatori predstavljaju jezgru svakog evolucijskog algoritma.

Također, vrlo važno svojstvo jezika je da se parametri algoritma ne navode unutar zapisa algoritma – u potpunosti su odvojeni od implementacije algoritma te se pohranjuju u posebnoj parametarskoj datoteci koja se parsira tek tijekom izvođenja algoritma [2].

Na slici 3.1. dan je primjer evolucijskog algoritma napisanog u ECDL-u.

```
repeat(population.size) {
      pool+=select.proportional(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 3.1 – evolucijski algoritam napisan u ECDL-u [2]

3.2 Osnove međujezika

Međujezik razvijen u sklopu ovog rada naziva se EIL (engl. *Evolutionary Intermediate Language*). Budući da je namijenjen interpretaciji, sintaksa međujezika morala je omogućiti lako prepoznavanje operacija i varijabli koje se koriste u naredbama. Zbog toga, međujezik je sintaksno vrlo sličan asemblerskim jezicima. Sintaksa jezika ne dozvoljava da se naredbe protežu kroz više linija – svaka naredba mora biti u posebnoj liniji. Većina naredbi međujezika sastoji se od dva dijela – identifikatora operacije i polja operanada. Identifikator operacije sastoji se od jedne riječi koja opisuje operaciju. Polje operanada sastoji se od niza varijabli ili konstanti odvojenih zarezom, a samo polje operanada je od identifikatora operacije odvojeno jednim razmakom.

Budući da u ECDL-u ne postoji mogućnost definicije korisničkih funkcija ili klasa, ona ne postoji ni u EIL-u. Također, isto kao i za ECDL, parametri algoritma u potpunosti su odvojeni od implementacije algoritma i pohranjuju se u posebnoj parametarskoj datoteci, što je detaljnije objašnjeno u poglavlju vezanom za prevođenje jezika [2].

EIL također podržava pisanje jednolinijskih komentara. Komentari započinju znakom „ ; “ iza kojeg mora slijediti jedan razmak. Komentari se smiju pisati samo u vlastitim linijama, odnosno, komentar ne smije stajati u istoj liniji s naredbom.

3.3 Varijable i konstante

Iako ECDL podržava samo jedan tip varijabli – skupove jedinki, zbog jednostavnije sintakse u EIL-u postoji potreba za dva tipa varijabli. Prvi tip čine skupovi jedinki koji su ekvivalentni skupovima jedinki iz ECDL-a. Skupovi zapravo ne sadrže same jedinke, već reference na njih [2]. Drugi tip varijabli obuhvaća cjelobrojne varijable i realne varijable. Njegova glavna namjena je pohrana međurezultata aritmetičkih operacija i veličine pojedinog skupa jedinki. Budući da je EIL sintaksno sličan asemblerskim jezicima, varijable u EIL-u funkcionalno se

mogu promatrati kao procesorski registri. Imenovanje varijabli nije izvedeno kao u višim programskim jezicima, nego se ime svake varijable sastoji od oznake tipa i indeksa. Oznaka *d* [1] predstavlja varijable skupova jedinki, a oznaka *i* [2] cjelobrojne i realne varijable. Varijabla s imenom *d0* ima posebnu namjenu; ona predstavlja cijelu populaciju, odnosno inicijalni skup jedinki u evolucijskom algoritmu. Od ostalih varijabli tipa *d* razlikuje se isključivo po tome što odmah pri pokretanju sadrži broj jedinki koji je određen u parametarskoj datoteci. Varijabla s imenom *d1* se uobičajeno koristi kao međurezultat izvođenja naredbi. Prije prvog dodjeljivanja vrijednosti, varijable tipa *d* ne sadrže niti jednu jedinku, a varijable tipa *i* imaju vrijednost 0. Ukupan broj varijabli koji se koristi u programu za oba tipa mora biti eksplicitno naveden unutar programa, što je detaljnije objašnjeno u idućem potpoglavlju.

Osim varijabli, kao i ECDL, EIL podržava dva tipa konstanti – cjelobrojne konstante i realne konstante. Iako se u ECDL-u vrijednosti konstanti ne mogu pridjeljivati varijablama, u EIL-u je moguće njihovu vrijednost pridjeljivati varijablama tipa *i* [2].

U daljnjem tekstu u opisu sintakse naredbi, za varijable tipa *d* koristi se notacija <d_var>, a za varijable tipa *i* <i_var>. Za dozvoljene tipove konstanti se koristi notacija <const>. Notacija <i_var>|<const> označava da se na danom mjestu unutar naredbe smiju koristiti varijable tipa *i* ili konstante.

[1] oznaka *d* dolazi od engl. *deme* : podrazred populacije, odnosno grupa blisko povezanih jedinki

[2] oznaka *i* dolazi od engl. *integer*: cijeli broj

3.4 Kontrolne naredbe

U jeziku EIL postoje tri kontrolne naredbe. Njihova namjena nije vezana za rad evolucijskih algoritama, nego je vezana za davanje informacija interpreteru. Prve dvije naredbe su #Vd i #Vi, kojima se interpreteru daje informacija koliko se varijabli tipa *d* i tipa *i* koristi u programu zbog alokacije memorije. Zadnja kontrolna naredba je naredba END, koja služi za označavanje kraja bloka naredbi evolucijskog algoritma. Naredbe #Vd i #Vi uobičajeno se koriste samo jednom na samom početku programa. Korištenje spomenutih naredbi na drugim mjestima ili više puta u programu nije preporučljivo, ali neće izazvati pogrešku; interpreter će koristiti zadnje navedene vrijednosti. Naredba END uobičajeno je zadnja naredba u programu. Ako u programu postoje naredbe navedene nakon naredbe END, one nikad neće biti izvršene. Izvršavanjem naredbe END interpreter prelazi u novu iteraciju evolucijskog algoritma. Sintaksa kontrolnih naredbi prikazana je na slici 3.2, a primjeri korištenja prikazani su na slici 3.3.

```
#Vd <const>
#Vi <const>
END
```

Slika 3.2 – dozvoljena sintaksa kontrolnih naredbi

```
#Vd 5
#Vi 4
; program koristi 5 varijabli tipa d
; program koristi 4 varijable tipa i
; blok naredbi
END
; kraj trenutne iteracije evolucijskog algoritma
```

Slika 3.3 – primjer korištenja kontrolnih naredbi

3.5 Naredbe svojstava

U ECDL-u, varijable imaju dva dostupna svojstva: *size* i *clear* [2], stoga u EIL-u postoje naredbe SZ i CLR koje se izvršavaju nad varijablama tipa *d*. Naredba SZ vraća broj jedinki koji se nalazi u danoj varijabli. U slučaju da se unutar varijable više puta pojavljuje ista jedinka, prebrojat će se svaka njena pojava. Naredba CLR koristi se za brisanje sadržaja varijable, čime varijabla više neće sadržavati niti jednu jedinku. Na slici 3.4. prikazana je sintaksa korištenja naredbi svojstava, a na slici 3.5. nalaze se primjeri korištenja naredbi svojstava.

```
SZ <d_var>,<i_var>
CLR <d_var>
```

Slika 3.4 – sintaksa naredbi svojstava

```
; 1. primjer
SZ d0,i1
; u varijablu i1 upisuje se ukupan broj jedinki u populaciji
CLR d1
; briše se sav sadržaj varijable d1

; 2. primjer
CLR d2
SZ d2,i1
; vrijednost upisana u varijablu i1 će biti 0
```

Slika 3.5 – primjeri korištenja naredbi svojstava

3.6 Naredbe pridruživanja, dodavanja i brisanja

3.6.1 Naredbe pridruživanja

U EIL-u postoje dvije naredbe pridruživanja: ASG i MOV. Naredba ASG koristi se za dodjeljivanje vrijednosti varijablama tipa *d*, a naredba MOV za dodjeljivanje vrijednosti varijablama tipa *i*. Naredba ASG ima tri dozvoljena oblika s dva, tri ili četiri operanda zbog mogućnosti indeksiranja jedinki, što je detaljnije objašnjeno u primjerima na slici 3.7., zajedno s primjerom korištenja naredbe MOV i ekvivalentnim naredbama u ECDL-u. Na slici 3.6. prikazana je sintaksa naredbi pridruživanja.

```
ASG <d_var>,<d_var>
ASG <d_var>,<i_var>|<const>,<d_var>
ASG <d_var>,<i_var>|<const>,<d_var>,<i_var>|<const>
MOV <const>,<i_var>
```

Slika 3.6 – sintaksa naredbi pridruživanja

```
ASG d2,d3
; u varijablu d3 se upisuje sav sadržaj varijable d2
; pri tome se briše sav prethodni sadržaj varijable d3
; ekvivalentno ECDL izrazu d3=d2
ASG d2,1,d3
; u varijablu d3 se upisuje jedinka iz d2 s indeksom 1
; pri tome se briše sav prethodni sadržaj varijable d3
; upisana jedinka ima indeks 0
; ekvivalentno ECDL izrazu d3=d2[1]
ASG d2,1,d3,2
; ekvivalentno ECDL izrazu d3[2]=d2[1]
; sve ostale jedinke u varijabli d3 ostaju nepromijenjene
MOV 2,i1
; varijabla i1 poprima vrijednost 2
ASG d2,i1,d3
; brišu se sve jedinke iz d3 i d3[0] postaje d2[2]
```

Slika 3.7 – primjeri korištenja naredbi pridruživanja

3.6.2 Naredbe dodavanja i brisanja

Naredbe dodavanja i brisanja koriste se kako bi se pojedina jedinka ili više njih moglo ukloniti iz neke varijable ili dodati u neku varijablu. Naredba APP koristi se za dodavanje, a naredba RMV se koristi za uklanjanje. Obje naredbe imaju dva dozvoljena oblika koji se koriste ovisno o tome želi li se dodati/obrisati cijeli skup jedinki iz drugog skupa ili samo jedna jedinka (koja se tada indeksira). Sintaksa naredbi prikazana je na slici 3.8, a primjeri korištenja s objašnjenjima i ekvivalentnim izrazima u ECDL-u nalaze se na slici 3.9.

```
APP <d_var>,<d_var>
APP <d_var>,<i_var>|<const>,<d_var>
RMV <d_var>,<d_var>
RMV <d_var>,<i_var>|<const>,<d_var>
```

Slika 3.8 – sintaksa naredbi dodavanja i brisanja

```
APP d2,d3
; sve jedinke iz varijable d2 pridodaju se varijabli d3
; ekvivalentno ECDL izrazu d3+=d2
APP d2,2,d3
; varijabli d3 pridodaje se jedinka s indeksom 2 varijable d2
; ekvivalentno ECDL izrazu d3+=d2[2]
RMV d2,d3
; sve jedinke iz varijable d2 uklanjaju se iz varijable d3
; ekvivalentno ECDL izrazu d3-=d2
RMV d2,3,d3
; jedinka iz varijable d2 s indeksom 3 se uklanja iz d3
; ekvivalentno ECDL izrazu d3-=d2[3]
```

Slika 3.9 – primjeri korištenja naredbi dodavanja i brisanja

3.7 Aritmetičke naredbe

S obzirom da se u ECDL-u mogu vršiti aritmetičke operacije nad konstantama unutar uvjeta naredbi grananja i petlji [2], međujezik mora zadovoljiti istu funkcionalnost. Budući da namjena i struktura međujezika nisu prikladni za korištenja aritmetičkih operatora, aritmetičke operacije izvedene su na sličan način kao u asemblerskim jezicima – svaka aritmetička operacija između dvije varijable prevodi se u jednu naredbu kojom se pamti međurezultat i prosljeđuje idućim naredbama. Unutar EIL-a postoji 5 aritmetičkih naredbi i one se mogu koristiti samo nad konstantama i varijablama tipa *i*. Naredba ADD koristi se za zbrajanje, naredba SUB za oduzimanje, naredba MUL za množenje, naredba DIV za dijeljenje, a naredba MOD kao operator modulo. Na slici 3.10 prikazana je sintaksa aritmetičkih naredbi, a na slici 3.11 prikazani su primjeri korištenja aritmetičkih naredbi.

```
ADD <i_var>|<const>,<i_var>|<const>,<i_var>
SUB <i_var>|<const>,<i_var>|<const>,<i_var>
MUL <i_var>|<const>,<i_var>|<const>,<i_var>
DIV <i_var>|<const>,<i_var>|<const>,<i_var>
MOD <i_var>|<const>,<i_var>|<const>,<i_var>
```

Slika 3.10 – sintaksa aritmetičkih naredbi

```
ADD 2,3,i0
; i0 = 2 + 3 = 5
SUB i0,2.5,i1
; i1 = i0 - 2.5 = 2.5
MUL i0,i0,i0
; i0 = i0 * i0 = 25
DIV i0,2,i1
; i1 = i0 / 2 = 12.5
MOD i0,4,i2
; i2 = i0 % 4 = 1
```

Slika 3.11 – primjeri korištenja aritmetičkih naredbi

3.8 Naredbe usporedbe, logičke naredbe i naredbe grananja

3.8.1 Naredbe usporedbe

Operatori usporedbe u EIL-u su izvedeni na sličan način kao i aritmetički operatori. U tablici 3.1 prikazan je odnos prevođenja operatora usporedbe u odgovarajuću naredbu usporedbe.

Tablica 3.1 – odnos operatora i naredbi usporedbe

Operator usporedbe	Naredba usporedbe
==	EQ
!=	NEQ
>	GT
>=	GTE
<	LT
<=	LTE
?	CON
!?	NCON

Operator označen s „?“ naziva se operatorom sadržavanja. Njime se provjerava je li multiskup s lijeve strane operatora podskup multiskupa s desne strane operatora. Operator ne provjerava poredak jedinki, već samo provjerava sadrži li jedan multiskup drugi multiskup, što je i glavni razlog potrebe za operatorom sadržavanja. Ako dvije varijable sadrže iste jedinke, ali različitim redoslijedom, operator jednakosti (==) kao rezultat će vratiti logičku neistinu, dok će operator sadržavanja vratiti logičku istinu [2]. Operator „!?“ je negacija operatora sadržavanja. On nije dio ECDL-a, ali je dodan u EIL zbog pojednostavljenja procesa prevođenja logičkih izraza. Naredba sadržavanja i njena negacija se

mogu upotrebljavati isključivo na varijablama tipa *d*, dok se naredba jednakosti može upotrebljavati na oba tipa varijabli. Ostale naredbe usporedbe mogu se upotrebljavati isključivo na varijablama tipa *i*. Na slici 3.12 prikazana je cijela sintaksa naredbi usporedbe, a na slici 3.13 primjeri korištenja naredbi usporedbe u EIL-u.

```
EQ <d_var>,<d_var>
EQ <d_var>,<i_var>|<const>,<d_var>,<i_var>|<const>
EQ <i_var>|<const>,<i_var>|<const>
NEQ <d_var>,<d_var>
NEQ <d_var>,<i_var>|<const>,<d_var>,<i_var>|<const>
NEQ <i_var>|<const>,<i_var>|<const>
GT <i_var>|<const>,<i_var>|<const>
GTE <i_var>|<const>,<i_var>|<const>
LT <i_var>|<const>,<i_var>|<const>
LTE <i_var>|<const>,<i_var>|<const>
CON <d_var>,<d_var>
CON <d_var>,<i_var>|<const>,<d_var>
NCON <d_var>,<d_var>
NCON <d_var>,<i_var>|<const>,<d_var>
```

Slika 3.12 – sintaksa naredbi usporedbe

```
EQ d2,d3
; usporedba: d2==d3
EQ d2,0,d3,i1
; usporedba: d2[0]==d3[i1]
EQ i2,0
; usporedba i2 == 0
CON d2,d3
; provjera nalaze li se sve jedinke iz d2 u d3
CON d2,1,d3
; provjera nalazi li se jedinka d2[1] u d3
LT i3,2
; usporedba i3 < 2
```

Slika 3.13 – primjeri korištenja naredbi usporedbe

3.8.2 Logičke naredbe

Logički operatori I i ILI u EIL-u su izvedeni kao naredbe AND i OR. Nijedna od navedenih naredbi nema polje operanada. U programu, dolaze između dvije naredbe usporedbe. Operator negacije ne postoji u EIL-u, nego se izravno u procesu prevođenja iz ECDL-a u EIL operatori usporedbe zamjenjuju svojim ekvivalentom za negaciju te se operator I zamjenjuje operatorom ILI i obrnuto. Od ostalih naredbi vezanih za logičke operatore postoje naredbe BRC i CRB koje predstavljaju otvorenu i zatvorenu zagradu. U programu, između naredbi BRC i CRB mora se nalaziti barem jedna naredba usporedbe ili više njih povezanih naredbama AND ili OR. Na slici 3.14 prikazana je usporedba logičkih izraza u ECDL-u i EIL-u.

```
//logički izrazi u ECDL-u


// 1. primjer
population.size>3 &&
population.size<10




// 2. primjer
!(individual[0] ? tournament
|| individual[1] ? tournament)
```

```
; logički izrazi u EIL-u
;
; 1. primjer
SZ d0,i0
GT i0,3
AND
SZ d0,i0
LT i0,10
;
; 2. primjer
BRC
NCON d3,0,d4
AND
NCON d3,1,d4
CRB
```

Slika 3.14 – usporedba logičkih izraza u ECDL-u i EIL-u

3.8.3 Naredbe grananja

EIL podržava if – else grananja. Struktura grananja započinje naredbom IF iza koje u programu dolaze logičke naredbe i naredbe usporedbe kojima je opisan prevedeni logički uvjet. Nakon uvjeta dolazi naredba THEN, kojom se interpreteru daje informacija da završava niz logičkih naredbi i naredbi usporedbe. Nakon naredbe THEN slijedi niz naredbi koje se izvršavaju u slučaju istinite evaluacije logičkog uvjeta. Niz naredbi završava se naredbom END_IF, kojom se interpreteru daje informacija gdje nastaviti izvođenje u slučaju neistinite evaluacije logičkog uvjeta. Else blok započinje naredbom ELS, a završava naredbom END_ELS, kojom se interpreteru daje informacija gdje nastaviti izvođenje u slučaju da je evaluacija logičkog uvjeta u IF dijelu bila istinita. Na slici 3.15 prikazana je usporedba naredbi grananja u ECDL-u u EIL-u.

```
//grananje u ECDL-u
if(log_uvjet_1) {
      //naredbe_1
}
else if(log_uvjet_2) {
      //naredbe_2
}
else {
      //naredbe_3
}
```

```
; grananje u EIL-u
IF
; opis logičkog uvjeta 1
THEN
; naredbe_1
END_IF
ELS
IF
; opis logičkog uvjeta 2
THEN
; naredbe_2
END_IF
ELS
; naredbe_3
END_ELS
END_ELS
```

Slika 3.15 – usporedba naredbi grananja u ECDL-u i EIL-u

3.9 Petlje

U EIL-u postoje dvije vrste petlji – petlja za unaprijed znan broj ponavljanja (naredba LOOP) i beskonačna petlja (naredba LOOP_INF). Broj ponavljanja u naredbi LOOP može biti zadan isključivo preko varijable tipa *i*. Blok naredbi unutar obje petlje zaključuje se naredbom END_LP. Naredba LOOP_INF zamjenjuje while petlju i for petlju iz ECDL-a, te uz naredbu BRK uz ispunjenje uvjeta za izlaz iz petlje nastavlja izvođenje naredbi nakon pripadajuće END_LP naredbe. Na slici 3.16 prikazana je usporedba petlji u ECDL-u i EIL-u.

`//petlje u ECDL-u`
`//1. primjer`
`repeat(100)`
`{`
`  //blok naredbi`
`}`
 	
`//2.primjer`
`while(logički uvjet)`
`{`
`  //blok naredbi`
`}`
	`; petlje u EIL-u`
	`; 1. primjer`
	`MOV 100,i0`
	`LOOP i0`
	`; blok naredbi`
	`END_LP`
	`;`
	`; 2.primjer`
	`LOOP_INF`
	`IF`
	`; negacija logičkog uvjeta`
	`THEN`
	`BRK`
	`END_IF`
	`; blok naredbi`
	`END_LP`

-

Slika 3.16 – usporedba petlji u ECDL-u i EIL-u

3.10 Evolucijske naredbe

3.10.1 Naredbe selekcije

U tablici 3.2 prikazan je odnos prevođenja naredbi selekcije iz ECDL-a u naredbe selekcije u EIL-u te kriterij po kojem se jedinke selektiraju u navedenoj naredbi.

Tablica 3.2 – odnos naredbi selekcije u ECDL-u i EIL-u i kriterij selekcije

Naredba u ECDL-u	Naredba u EIL-u	Kriterij
select.random	SLRND	slučajan odabir jedinke
select.best	SLBST	odabir jedinke s najboljom dobrotom
select.worst	SLWRST	odabir jedinke s najlošijom dobrotom
select.proportional	SLPRP	slučajan odabir jedinke, ali vjerojatnost odabira ovisi o dobroti jedinke

Za posljednje navedenu naredbu, SLPRP, definira se dodatni parametar algoritma – selekcijski pritisak. Njime se definira koliko se najbolja jedinka smatra boljom od najlošije jedinke te se prema tome oblikuju vjerojatnosti odabira pojedinih jedinki u ovisnosti o njihovoj dobroti [2]. Parametri algoritma detaljnije su objašnjeni u poglavlju 4. Na slici 3.17 prikazana je sintaksa naredbi selekcije u EIL-u, a na slici 3.18 primjeri korištenja naredbi selekcije u EIL-u.

```
SLRND <d_var>,<d_var>
SLRND <d_var>,<d_var>,<i_var>|<const>
SLBST <d_var>,<d_var>
SLBST <d_var>,<d_var>,<i_var>|<const>
SLWRST <d_var>,<d_var>
SLWRST <d_var>,<d_var>,<i_var>|<const>
SLPRP <d_var>,<d_var>
SLPRP <d_var>,<d_var>,<i_var>|<const>
```

Slika 3.17 – sintaksa naredbi selekcije u EIL-u

```
SLRND d0,d2
; briše se sav prethodni sadržaj varijable d2
; u d2[0] upisuje se slučajno odabrana jedinka iz populacije
SLBST d3,d2,1
; odabire se najbolja jedinka iz varijable d3
; odabrana jedinka se upisuje u d2[1]
```

Slika 3.18 – primjeri korištenja naredbi selekcije u EIL-u

3.10.2 Naredbe križanja

Naredbom križanja stvara se jedinka slična roditeljima. Odabir vrste križanja koje će se provesti nad jedinkama navodi se kao parametar u parametarskoj datoteci. U ECDL-u, pa tako i u EIL-u postoje dva oblika naredbi križanja – križanje bez kopiranja (naredba CRS) i križanje s kopiranjem (naredba CRSC). Kod križanja bez kopiranja, jedinka koja je rezultat križanja zamjenjuje već postojeću jedinku. Kod križanja s kopiranjem, rezultat križanja je potpuno nova jedinka te se križanjem ne uzrokuju promjene nad već postojećim jedinkama. Sintaksa naredbi križanja prikazana je na slici 3.19, a primjeri korištenja prikazani su na slici 3.20.

```
CRS <d_var>,<i_var>|<const>,<d_var>,<i_var>|const>,<d_var>,
     <i_var>|<const>
CRSC <d_var>,<i_var>|<const>,<d_var>,<i_var>|<const>,<d_var>
CRSC <d_var>,<i_var>|<const>,<d_var>,<i_var>|const>,<d_var>,
     <i var>|<const>
```

Slika 3.19 – sintaksa naredbi križanja u EIL-u

```
CRS d1,0,d2,1,d3,1
; ekvivalentno ECDL naredbi crossover(d1[0],d2[1],d3[1])
; d1[0] i d2[1] su roditelji
; rezultat križanja se pohranjuje u d3[1]
CRSC d1,0,d2,1,d3
; ekvivalentno ECDL naredbi d3=crossover(d1[0],d2[1])
; rezultat križanja je nova jedinka upisana u d3[0]
; ne mijenja se nijedna već postojeća jedinka
; prijašnji sadržaj varijable d3 se briše
CRSC d1,0,d2,1,d3,1
; ekvivalentno ECDL naredbi d3[1]=crossover(d1[0],d2[1])
; rezultat križanja je nova jedinka
; ne mijenja se nijedna već postojeća jedinka
; rezultat se upisuje u d3[1]
```

Slika 3.20 – primjeri korištenja naredbi križanja

3.10.3 Naredba mutacije

Mutacijom se nad jedinkom ili skupom jedinki obavlja slučajna promjena nad njihovom strukturom. U EIL-u, vrsta mutacije i njena vjerojatnost navode se odvojeno, kao parametri u parametarskoj datoteci. Kao i kod naredbi križanja, u EIL-u također postoje dva osnovna oblika naredbe mutacije – mutacija bez kopiranja i mutacija s kopiranjem. Mutacijom bez kopiranja promjene se vrše nad danom varijablom, dok se pri mutaciji s kopiranjem stvara kopija originalne varijable te se promjene vrše nad kopijom, čime originalna varijabla ostaje

nepromijenjena. Na slici 3.21 prikazana je sintaksa naredbe mutacije, a na slici 3.22 primjeri korištenja naredbe mutacije.

```
MTT <d_var>
MTT <d_var>,<i_var>|<const>
MTT <d_var>,<d_var>
MTT <d_var>,<i_var>|<const>,<d_var>
```

Slika 3.21 – sintaksa naredbe mutacije

```
MTT d2
; mutacija se provodi nad svim jedinkama iz varijable d2
; ekvivalentno s ECDL izrazom mutate(d2)
MTT d2,1
; mutacija se provodi nad jedinkom upisanom u d1[1]
; ekvivalentno s ECDL izrazom mutate(d2[1])
MTT d2,d3
; briše se sav sadržaj varijable d3
; u d3 se kopira sadržaj varijable d2
; mutacija se provodi nad svim jedinkama sadržanim u d3
MTT d2,1,d3
; briše se sav sadržaj varijable d3
; u varijablu d3 se kopira jedinka d2[1]
; mutacija se provodi nad varijablom d3
```

Slika 3.22 – primjeri i objašnjenja korištenja naredbe mutacije

3.10.4 Naredba evaluacije

Evaluacija nije pravi evolucijski operator, ali je vezana za evolucijske algoritme. Kada dođe do stvaranja nove jedinke ili obavljanja promjena nad pojedinim jedinkama, evaluacijom se određuje koliko nova ili promijenjena jedinka predstavlja dobro rješenje [2]. Sintaksa naredbe evaluacije prikazana je na slici 3.23, a primjeri i objašnjenje korištenja prikazani su na slici 3.24.

```
EVL <d_var>,<i_var>|<const>
EVL <d_var>
```

Slika 3.23 – sintaksa naredbe evaluacije

```
EVL d2,1
; ekvivalentno ECDL izrazu evaluate(d2[1])
; evaluira se jedinka iz varijable d2 s indeksom 1
EVL d3
; ekvivalentno ECDL izrazu evaluate(d3)
; evaluiraju se sve jedinke sadržane u varijabli d3
```

Slika 3.24 – primjeri i objašnjenja korištenja naredbe evaluacije u EIL-u

3.11 Moguća proširenja metajezika i međujezika

U trenutnoj verziji ECDL-a i EIL-a, ne postoji mogućnost izravnog upravljanja podacima sadržanih u jedinkama te je zbog toga u njima moguće pisati samo algoritme koji su neovisni o genotipu, odnosno, algoritme neovisne o strukturi i tipu podataka jedinki. Zbog toga nije moguće ECDL-om i EIL-om opisati algoritme poput optimizacije kolonijom mrava (engl. *ant colony optimization*), optimizacije rojem čestica (engl. *particle swarm optimization*) i slične algoritme koji u svom radu koriste jedinke prikazane realnim brojevima te izravno pristupaju i mijenjaju njihove podatke. Moguće proširenje kojim bi se riješio taj problem je uvođenje realnih varijabli u ECDL te dodavanje operatora kojim bi se moglo pristupiti podacima unutar jedinke te mijenjati njihov sadržaj. Navedeno proširenje izrazito bi povećalo kompleksnost ECDL-a te bi uzrokovalo promjene i nad ostalim operatorima; gramatika jezika morala bi doživjeti veće izmjene zbog dodatnog tipa varijabli te bi u prevođenje trebalo dodati i semantičku analizu (koja trenutno nije implementirana jer za njom nema potrebe zbog samo jednog tipa varijabli). Na EIL bi se navedeno proširenje odrazilo u znatno manjoj mjeri jer realne varijable već postoje te bi bilo potrebno samo dodati naredbe za pristup i izmjenu podataka jedinke.

Također, jezike bi bilo korisno proširiti naredbama kopiranja jedinki. U trenutnoj verziji, kopiranje jedinki moguće je izvesti isključivo preko naredbi križanja i mutacije. Iako taj koncept pokriva većinu slučajeva u kojima originalne jedinke moraju ostati nepromijenjene, njime se ipak donose znatna ograničenja. Primjerice, ako bi se željela stvoriti nova generacija jedinki iz trenutne generacije koristeći neki od operatora selekcije (npr. algoritam *Roulette Wheel* [1]), u novoj generaciji moguća je višestruka pojava nekih od jedinki. Budući da su varijable izvedene kao skupovi referenci na jedinke, a ne kao skupovi samih jedinki, promjena nad jednom jedinkom koja se više puta ponavlja unutar skupa uzrokovala bi istu promjenu nad svakom njenom pojavom unutar tog skupa kao i nad njenom pojavom unutar izvornog skupa nad kojim je vršena selekcija. Uvođenjem posebnih naredbi kopiranja vrlo jednostavno bi se mogao riješiti navedeni problem bez povećanja složenosti jezika.

4. Programski prevoditelj i interpreter

Proces prevođenja i interpretiranja izvornog programa napisanog u ECDL-u započinje prevođenjem programa iz ECDL-a u EIL. Nakon toga, pokreće se interpreter koji koristi generirani EIL program. Program je moguće napisati i izravno u EIL-u, ali se ne preporučuje jer interpreter u danom EIL programu ne provjerava postojanje mogućih pogrešaka. U trenutnoj implementaciji, prevoditelj i interpreter u potpunosti su odvojeni – potrebno je posebno pokrenuti prevođenje, a zatim posebno pokrenuti interpretiranje.

4.1 Prevoditelj

U sklopu ovog rada, razvijen je programski prevoditelj iz ECDL-a u EIL. Prevoditelj je zasnovan na originalnom prevoditelju ECDL-a za ECFJ [2] i u potpunosti napisan u programskom jeziku C#. Jedina razlika između originalnog prevoditelja i prevoditelja za EIL je u generatoru ciljnog koda. U planu je prevođenje prevoditelja iz jezika C# u C++ čime bi se omogućilo spajanje prevoditelja i interpretera u isti program. Prevođenje varijabli iz ECDL-a u EIL vrši se dodjeljivanjem jedinstvenog indeksa svakoj varijabli. Indeks 0 uvijek je dodijeljen varijabli *population*, a indeks 1 je rezerviran za varijablu koja se koristi kao međurezultat. Ostali se indeksi varijablama dodjeljuju slijedno prema redoslijedu njihovog pojavljivanja u programu. Izrazi iz ECDL-a u EIL prevode se prema pravilima opisanim u prethodnom poglavlju, osim složenijih izraza u kojima se mora koristiti međurezultat (prethodno spomenuta varijabla tipa *d* s indeksom 1). Na slici 4.1 prikazan je primjer prevođenja složenijih izraza iz ECDL-a s objašnjenjem.

//ECDL program
//1. primjer
tournament -=
  select.random(tournament);
 	
//2. primjer
tournament +=
  crossover(population[0],
  population[1]);
	; EIL program
	; 1. primjer
	SLRND d2,d1
	; d1 - međurezultat
	RMV d1,d2
	; d2 – varijabla tournament
	;
	; 2. primjer
	CRSC d0,0,d0,1,d1
	; d1 - međurezultat
	APP d1,d2

Slika 4.1 – prevođenje složenijih izraza iz ECDL-a u EIL

4.2 Interpreter

U sklopu rada razvijen je interpreter međujezika - ELI (engl. *Evolutionary intermediate Language Interpreter*). U potpunosti je napisan u C++ kao algoritam ECF-a te se u svojem radu najviše oslanja na funkcionalnosti sadržane u ECF-u [1]. Rad interpretera započinje njegovom inicijalizacijom, pri čemu se čita cijela datoteka s EIL programom i naredbe se upisuju u radnu memoriju. Budući da se evolucijski algoritmi izvode u velikom broju iteracija, time se omogućuje veća brzina rada jer se datoteka s programom čita samo jednom. Nakon inicijalizacije započinje rad interpretera. Gotovo sve naredbe međujezika preslikavaju se u omjeru 1:1 na naredbe jezika C++ te se kao takve i izvode. Izuzetci navedenog pravila su naredbe grananja i petlje. Prilikom prvog nailaska na naredbu petlje, interpreter privremeno zaustavlja izvođenje naredbi i traži odgovarajuću naredbu za završetak petlje te, ako postoji, naredbu za izlaz iz petlje. Razlog tome leži u mogućnosti ugnježđivanja petlji te na taj način interpreter može odrediti koja naredba za završetak i izlaz pripada kojoj petlji. Također, ako logički uvjet petlje nije zadovoljen, interpreter odmah ima informaciju gdje treba nastaviti izvođenje. Prilikom prvog nailaska na naredbu IF, interpreter također privremeno zaustavlja izvođenje i traži odgovarajuće naredbe THEN, END_IF, te ako postoje, i naredbe ELS i END_ELS. Razlog tome sličan je kao i za petlje; moguće je ugnijezditi više IF izraza te time interpreter određuje koje naredbe pripadaju kojem bloku IF,

odnosno ELS naredbi. Također, time interpreter dobiva informaciju gdje treba nastaviti izvođenje u ovisnosti o istinitosti zadanih logičkih uvjeta. Navedeni mehanizmi nemaju velikog utjecaja na brzinu interpretera, jer se izvode samo prilikom prvog nailaska na navedene naredbe u prvoj iteraciji evolucijskog algoritma.

Varijable tipa *d* unutar interpretera se pohranjuju u vektor vektora pokazivača na jedinke te im se pristupa preko indeksa sadržanog u imenu. Svaka varijabla tipa *d* zapravo ne sadrži same jedinke, već sadrži reference na jedinke. Varijable tipa *i* pohranjuju se u vektor realnih brojeva te im se također pristupa preko indeksa sadržanog u imenu.

Kao i svi algoritmi u ECF-u, interpreter prima parametre preko XML (engl. *Extended Markup Language*) datoteke, koja se parsira prilikom pokretanja algoritma, odnosno, prilikom pokretanja problema na kojem se algoritam koristi. Interpreter izravno koristi dva parametra – ime datoteke s EIL programom koju je potrebno interpretirati i vrijednost selekcijskog pritiska. Ostali parametri (npr. veličina populacije, vjerojatnost mutacije, uvjet zaustavljanja,...) također se specificiraju unutar parametarske XML datoteke, ali nemaju izravno veze sa samim interpreterom te je njihovo korištenje riješeno mehanizmima ECF-a (kao što je slučaj i u ostalim algoritmima ECF-a).

5. Primjeri evolucijskih algoritama i usporedba izvođenja

5.1 Primjeri evolucijskih algoritama u ECDL-u i EIL-u

U nastavku su opisana dva jednostavna evolucijska algoritma - *Steady State Tournament* (SST) i *Elimination*.

U svakoj iteraciji algoritma *Steady State Tournament* iz populacije se nasumično odabiru tri jedinke te se najlošija od njih zamjenjuje produktom križanja druge dvije odabrane jedinke. Nad zamijenjenom jedinkom tada se provodi i mutacija. Svaka iteracija algoritma ponavlja se *n* puta, gdje *n* predstavlja broj jedinki u populaciji.

U svakoj iteraciji algoritma *Elimination* iz populacije se uklanja 60% jedinki koristeći obrnuto proporcionalnu selekciju (lošije jedinke imaju veću vjerojatnost selekcije). Prazna mjesta u populaciji tada se popunjavaju produktima križanja slučajno odabranih jedinki. Nakon toga, nad cijelom populacijom se provodi mutacija. Najbolje rješenje iz prijašnje iteracije ostaje nepromijenjeno. [1]

Navedeni algoritmi odabrani su kao primjer jer unutar ECF-a već postoje njihove gotove implementacije pa je moguće napraviti usporedbu vremena izvođenja postojeće implementacije i interpretiranog programa. Navedeni algoritmi napisani kao programi u ECDL-u prikazani su na slikama 5.1 i 5.3, a programi dobiveni njihovim prevođenjem u EIL su prikazani na slikama 5.2 i 5.4.

```
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[0]);
    evaluate(worst[0]);
}
```

Slika 5.1 – algoritam Steady State Tournament napisan u ECDL-u

```
#Vd 4
#Vi 2
SZ d0,i0
LOOP i0
CLR d2
MOV 3,i1
LOOP i1
SLRND d0,d1
APP d1,d2
END_LP
SLWRST d2,d3
RMV d3,d2
CRS d2,0,d2,1,d3,0
MTT d3,0
EVL d3,0
END_LP
END
```

Slika 5.2 – generirani EIL program dobiven prevođenjem algoritma Steady State Tournament napisanog u ECDL-u

```
best=select.best(population);
newGen=population;
repeat(0.6*newGen.size)
{
    victim.clear;
    victim=select.proportional(newGen);
    newGen-=victim[0];
}
repeat(0.6*newGen.size)
{
    parents.clear;
    parents+=select.random(newGen);
    parents+=select.random(newGen);
    children+=crossover(parents[0],parents[1]);
}
newGen+=children;
population=newGen;
population-=best;
mutate(population);
best_mutated=mutate(best);
population+=best_mutated;
evaluate(population);
random=select.random(population);
population-=random;
best+=random;
population+=select.best(best);
```

Slika 5.3 – algoritam Elimination napisan u ECDL-u

```
#Vd 9
#Vi 4
SLBST d0,d2
ASG d0,d3
SZ d3,i1
MUL 0.6,i1,i0
LOOP i0
CLR d4
SLPRP d3,d4
RMV d4,0,d3
END_LP
SZ d3,i3
MUL 0.6,i3,i2
LOOP i2
CLR d5
SLRND d3,d1
APP d1,d5
SLRND d3,d1
APP d1,d5
CRSC d5,0,d5,1,d1
APP d1,d6
END_LP
APP d6,d3
ASG d3,d0
RMV d2,d0
MTT d0
MTT d2,d7
APP d7,d0
EVL d0
SLRND d0,d8
RMV d8,d0
APP d8,d2
SLBST d2,d1
APP d1,d0
END
```

Slika 5.4 – generirani EIL program dobiven prevođenjem algoritma Elimination

5.2 Usporedba izvođenja

Usporedba vremena izvođenja direktne implementacije u ECF-u i interpretiranja prethodno navedenog algoritma SST provedena je na tri problema – problemu traženja minimuma funkcije (ECF primjer *FunctionMin*, funkcija 2, dimenzija 5), problemu trgovačkog putnika (ECF primjer *GATSP*, na problemu bays29) te na problemu simboličke regresije (postupak pronalaženja matematičkog izraza iz empirijskih podataka – ECF primjer *GPSymbReg*, uz maksimalnu dubinu stabla 13) na različitim veličinama populacije i broju iteracija. Ostali parametri unutar pojedinog problema bili su konstantni prilikom testiranja. Kao uvjet zaustavljanja korišten je broj iteracija algoritma. Navedeni problemi odabrani su zbog različite vremenske složenosti evolucijskih operatora. Pri rješavanju problema trgovačkog putnika i simboličke regresije, koristi se složeniji genotip (*Permutation* i *Tree* umjesto *FloatingPoint*) te su zbog toga evolucijski operatori, pogotovo evaluacija, znatno složeniji za izvođenje. Pretpostavke prije izvođenja bile su da će vrijeme izvođenja interpretiranjem i vrijeme izvođenja direktne implementacije biti u odnosu c : 1, gdje c predstavlja konstantni faktor veći od 1. Povećanjem složenosti, a time i vremena za izvođenje evolucijskih operatora (problemi *GATSP* i *GPSymbReg*) očekuje se smanjenje navedenog faktora, odnosno njegovo približavanje vrijednosti 1.

Izvođenje je testirano na računalu s dvojezgrenim procesorom Intel i3-2330M @ 2.20 GHz i 4 GiB radne memorije. Svaki prikazani rezultat je prosjek 100 mjerenja za dane parametre. Dobiveni podaci prikazani su u tablicama 5.1, 5.2 i 5.3, te na slikama 5.5, 5.6 i 5.7.

Iz dobivenih podataka vidi se da je u prvom slučaju interpreter otprilike 3,5 puta sporiji od direktno implementiranog programa, u drugom slučaju otprilike 1,9 puta sporiji, a u trećem slučaju otprilike 1,2 puta sporiji. Podaci dobiveni testiranjem potvrdili su početnu pretpostavku – povećanje složenosti genotipa će smanjiti utjecaj neefikasnosti interpretera pri izvođenju programa.

Tablica 5.1 – usporedba vremena izvođenja za algoritam SST na problemu FunctionMin

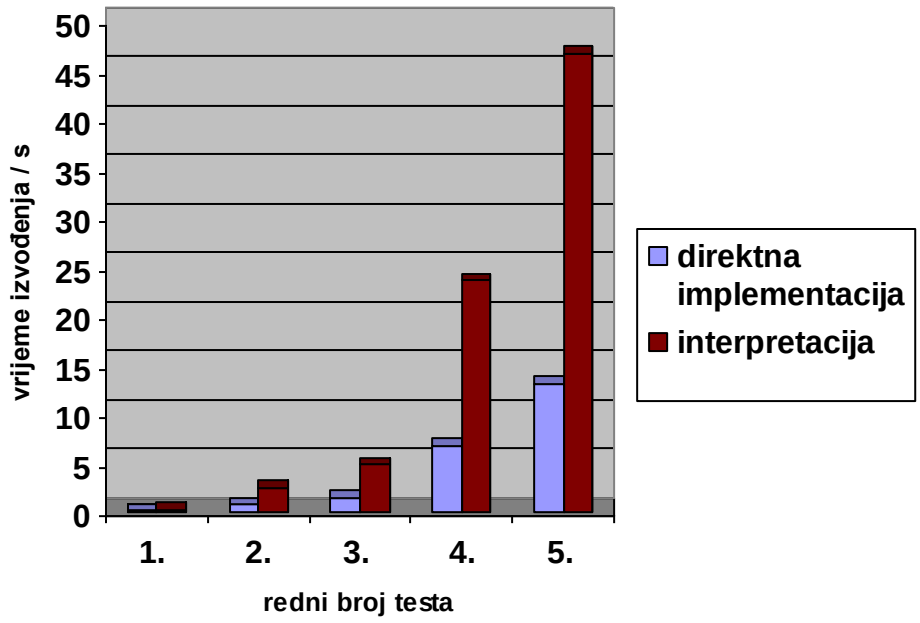
Redni broj	Parametri	Vrijeme izvođenja – direktna implementacija / s	Vrijeme izvođenja – interpretacija / s
1.	Populacija = 100 Broj iteracija = 100	0,029	0,098
2	Populacija = 500 Broj iteracija = 500	0,661	2,355
3.	Populacija = 500 Broj iteracija = 1000	1,315	4,724
4.	Populacija = 500 Broj iteracija = 5000	6,615	23,466
5.	Populacija = 1000 Broj iteracija = 5000	12,978	46,669

Tablica 5.2 – usporedba vremena izvođenja za algoritam SST na problemu GATSP

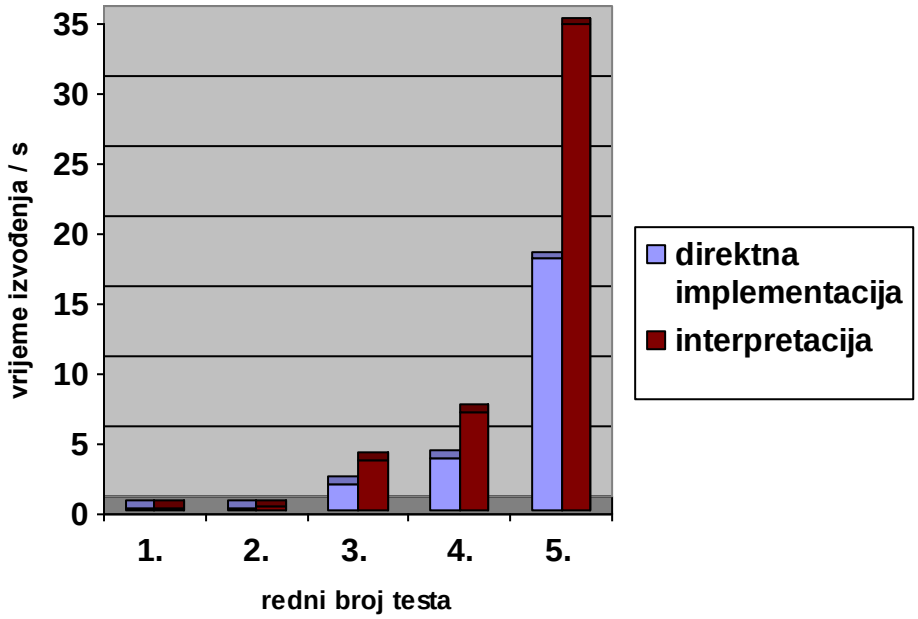
Redni broj	Parametri	Vrijeme izvođenja – direktna implementacija / s	Vrijeme izvođenja – interpretacija / s
1.	Populacija = 50 Broj iteracija = 100	0,043	0,077
2	Populacija = 100 Broj iteracija = 100	0,087	0,151
3.	Populacija = 500 Broj iteracija = 500	1,845	3,529
4.	Populacija = 500 Broj iteracija = 1000	3,679	6,957
5.	Populacija = 1000 Broj iteracija = 2500	17,868	34,561

Tablica 5.3 – usporedba vremena izvođenja za algoritam SST na problemu GPSymbReg

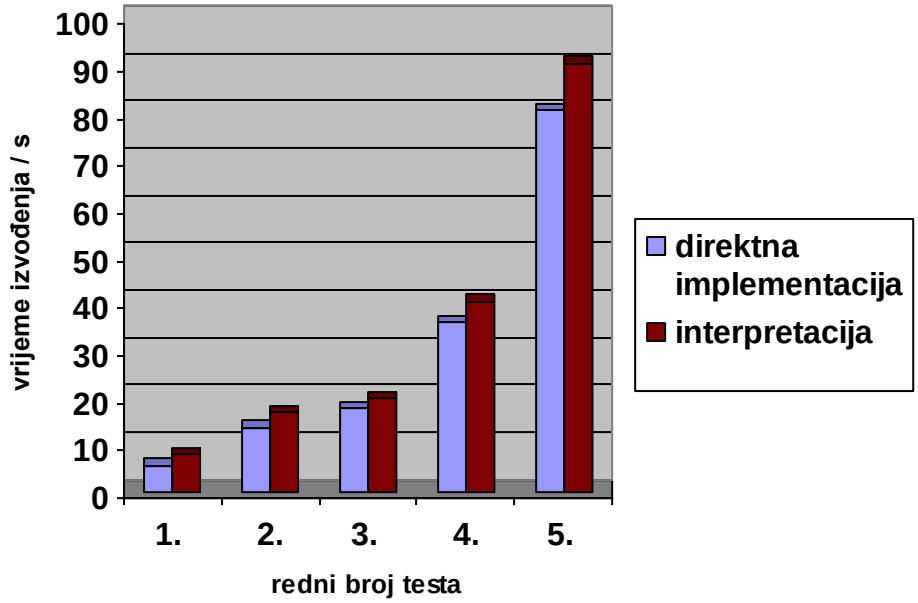
Redni broj	Parametri	Vrijeme izvođenja – direktna implementacija / s	Vrijeme izvođenja – interpretacija / s
1.	Populacija = 50 Broj iteracija = 100	5,546	7,908
2	Populacija = 100 Broj iteracija = 100	13,682	16,825
3.	Populacija = 500 Broj iteracija = 500	17,608	19,809
4.	Populacija = 500 Broj iteracija = 1000	35,871	40,273
5.	Populacija = 1000 Broj iteracija = 1000	80,661	90,500



Slika 5.5 – usporedba vremena izvođenja na problemu FunctionMin



Slika 5.6 – usporedba vremena izvođenja na problemu GATSP



Slika 5.7 – usporedba vremena izvođenja na problemu GPSymbReg

6. Zaključak

Specijaliziranim programskim jezikom za rješavanje problema određenog područja može se znatno olakšati proces pisanja programa te također smanjiti i potrebno vrijeme za pisanje programa vezanog za navedeno područje. Programer treba imati samo minimalnu količinu znanja o razvojnom sučelju za koji je specijalizirani jezik vezan. Takav je jezik znatno jednostavniji i apstraktniji te je time odmah i smanjena krivulja njegovog učenja.

ECDL i EIL stvoreni su upravo s tom svrhom, olakšanja pisanja programa vezanih za specifično područje – područje evolucijskih algoritama. Posredovanje međujezika uzrokuje neželjene posljedice poput usporavanja rada programa, ali se njime otvaraju mogućnosti korištenja drugih razvojnih okolina. Odbacivanjem izravnog prevođenja ECDL-a u drugi viši programski jezik te posredovanjem gotovog međujezika moguće je znatno jednostavnije ECDL primijeniti na druga razvojna okruženja za evolucijsko računanje neovisno o programskom jeziku u kojem su izvedena – potrebno je samo implementirati interpreter kojim bi se međujezik mogao integrirati s navedenim razvojnim okruženjem.

Iako ECDL trenutno i dalje pruža ograničene mogućnosti, njime je moguće ostvariti mnoge funkcionalnosti. Uvođenje EIL-a kao poveznice sa razvojnom okolinom tek je prva nadogradnja ECDL-a. Njegovim daljnjim korištenjem pojavit će se nove ideje i rješenja kako ugraditi nove dodatne mogućnosti te time proširiti područje njegove primjene i dodatno olakšati razvoj i korištenje evolucijskih algoritama.

7. Literatura

[1] ECF – Evolutionary Computation Framework, http://gp.zemris.fer.hr/ecf/, 16.5.2015.

[2] Đurasević, M., Metajezik za opis evolucijskih algoritama, završni rad – preddiplomski studij, Fakultet elektrotehnike i računarstva, Zagreb, 2012., http://bib.irb.hr/datoteka/586592.Marko_urasevic_Zavrsni.pdf

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

[4] Golub, M., Genetski algoritam, http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf, 2004.

8. Sažetak

Naslov: Ostvarenje metajezika za evolucijske algoritme u okruženju za evolucijsko računanje

U ovom radu predstavljeno je proširenje postojećeg metajezika za opis evolucijskih algoritama (ECDL). Proširenje se sastoji od međujezika za opis evolucijskih algoritama (EIL) te pripadajuće programske potpore kojom je ostvarena integracija navedenog metajezika i ECF-a (razvojnog okruženja za evolucijsko računanje). Navedena programska potpora sastoji se od programskog prevoditelja kojim se ECDL prevodi u EIL te interpretera kojim se omogućuje izvođenje programa napisanog u EIL-u unutar ECF-a. Također, korištenjem međujezika, omogućuje se i lakša buduća integracija sa mogućim drugim razvojnim okolinama. EIL je tek prva nadogradnja ECDL-a te se budućim nadogradnjama mogućnosti ECDL-a mogu dodatno proširiti.

Ključne riječi: metajezik, međujezik, evolucijsko računanje, evolucijski algoritmi, ECF, ECDL, EIL

9. Abstract

Title: Metalanguage implementation in Evolutionary Computation Framework

Main topic of this paper is an update of the Evolutionary Computation Description Language (ECDL). The update consists of a new intermediate language - Evolutionary Intermediate Language (EIL) and additional software – ECDL to EIL compiler and EIL interpreter (ELI), which are intended as a link between ECDL and ECF (Evolutionary Computation Framework). Additionally, using intermediate language instead of using direct compiling of ECDL to another high-level programming language has other benefits, for example, easier integration of ECDL with other frameworks. EIL and included software are only the first update of ECDL; additional functionalities will be added with future updates.

Keywords: metalanguage, intermediate language, evolutionary computing, evolutionary algorithms, ECF, ECDL, EIL