SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4332

Genetsko programiranje sa semantičkom analizom

Vedran Pintarić

Zagreb, lipanj 2016.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 9. ožujka 2016.

ZAVRŠNI ZADATAK br. 4332

Pristupnik: **Vedran Pintarić (0036476793)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Genetsko programiranje sa semantičkom analizom**

Opis zadatka:

Opisati načine uporabe genetskog programiranja u simboličkoj regresiji i evoluciji programa. Istražiti postojeće operatore i načine poboljšanja učinkovitosti evolucije. Posebnu pažnju posvetiti konceptu semantičkog genetskog programiranja i načina implementacije. Razviti algoritam genetskog programiranja koji koristi semantičku informaciju. Usporediti učinkovitost ostvarenog algoritma s obzirom na izvorno genetsko programiranje. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 17. lipnja 2016.

Mentor:

Izv. prof. dr. sc. Domagoj Jakobović

Predsjednik odbora za završni rad modula:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Zahvala mentoru prof. dr. sc. Domagoju Jakobovi\'cu za savjete, pomoć i povratne informacije pri izradi ovog završnog rada.

SADRŽAJ

1. Uvod

Genetsko programiranje jedna je od grana evolucijskog računarstva koja se bavi pronalaženjem i optimiranjem računalnih programa. Genetsko programiranje je optimizacijska tehnika u kojoj su programi prikazani kao geni, te se nad njima obavljaju operacije pomoću evolucijskog algoritma u svrhu optimizacije tog programa. Predviđen rezultat genetskog programiranja je ispravan program za zadani problem.

Doduše, genetsko programiranje nam ne omogućuje samo evoluciju programa. Neke primjene genetskog programiranja su:

- evolucija formalnih programa,
- generiranje algebarskih funkcija na temelju vrijednosti domena i kodomena,
- generiranje *booleovih* funkcija na temelju istinitosnih tablica.

Najviše ćemo se usredotočiti na probleme koje uzrokuje čisto sintaksno genetsko programiranje i kako semantička analiza rješava te probleme. Također ćemo opisati ideju semantičke analize, neku od mogućih implementacija i izazove u implementaciji.

Nakon opisivanja svih do sad poznatih mogućih implementacija, opisat ćemo konkretnu implementaciju te predstaviti rezultate ispitivanja.

2. Evolucijsko računarstvo

Evolucijsko računarstvo je jedno od područja koje pokriva umjetna inteligencija. U sklopu evolucijskog računarstva istražuju se razni algoritmi i principi koji imitiraju ponašanje života u biologiji, preciznije, koriste se Darwinovi principi evolucije sa ciljem provođenja skupe optimizacije. Pod Darwinove principe evolucije podrazumijevamo

- prirodnu selekciju,
- seksualnu reprodukciju,
- mutaciju.

Generalna ideja algoritama u evolucijskom računarstvu jest da se oni provode iterativno, generacija po generacija, kao što je slučaj i u biološkoj evoluciji. Najbitniji i osnovni element evolucijskog računarstva jest jedinka. Jedinka je nedjeljiv element u evoluciji na koju djeluju mutacije i selekcija (izbor hoće li jedinka "preživjeti"). Jedinka, također, može sudjelovati u seksualnoj reprodukciji s ostalim jedinkama. U biološkoj evoluciji pod pojmom "jedinka" podrazumijevamo, laički rečeno, jednu životinju dok u evolucijskom računarstvu pod tim pojmom mislimo na jedno potencijalno rješenje nekog problema, npr. jedinka može biti program kao što je to slučaj u genetskom programiranju o kojem će više biti riječ kasnije. Populacija je naziv za skup svih jedinki koje međusobno mogu biti u interakciji. Populacija će se iz generacije u generaciju mijenjati, te je cilj evolucijskih algoritama usmjeriti tu evoluciju u smjeru koji će jedinke mijenjati tako da bolje rješavaju neki predefinirani problem.

U biologiji svaka jedinka se sastoji od genotipa i fenotipa. Genotip jedinke je genetska informacija koju ta jedinka nosi, njen DNK, dok je fenotip posljedica genotipa, tj. manifestacija genotipa u izgledu, ponašanju i ostalim svojstvima jedinke. Genotip i fenotip imaju ista značenja u računarstvu. Genotip je zapis jedinke u memoriji računala, dok je fenotip samo ponašanje te jedinke. Za razliku od genotipa u biologiji koji se u velikoj većini zapisuje kao DNK neke jedinke, genotip u računarstvu može biti u raznim zapisima, kao što su *floating point* brojevi, *bitstringovi*, stabala i ostalo.

2.1. Selekcija

U prirodnoj evoluciji selekcija se odvija tako što preživljavaju samo one vrste koje uspiju preživjeti dovoljno dugo da kroz reprodukciju prenesu svoj genetski materijal na sljedeću generaciju. U računarstvu selekcija se najčešće obavlja pomoću funkcije koja računa dobrotu (eng. *fitness*) jedinke. Dobrota jedinke je, najčešće, broj koji predstavlja sposobnost jedinke u rješavanju zadanog problema te u ovisnosti o tome i korištenom algoritmu se bira hoće li jedinka "preživjeti" da prenese svoj genetski materijal na sljedeću generaciju.

2.2. Reprodukcija

Pod pojmom "reprodukcija" najčešće mislimo na seksualnu reprodukciju kojom dvije jedinke roditelji kombiniraju svoj genetski materijal s ciljem stvaranja genetskog materijala jedinke djeteta. U računarstvu smo fleksibilniji s načinima reprodukcije nego što je to priroda pa možemo i kombinirati razne načine reprodukcije kao što su, maloprije spomenuta, seksualna reprodukcija dva roditelja ili kloniranje, a evoluciju možemo provesti i bez reprodukcije i samo se oslanjati na mutaciju jedinki.

2.3. Mutacija

U biologiji, mutacija je trajna promjena genetske poruke neke jedinke. Mutacija se može dogoditi spontano ili u slučaju neke nepopravljive štete na DNK. Mutacije najčešće rezultiraju u nepovoljnim svojstvima za preživljavanje jedinke, ali ponekad dolazi i do korisnih mutacija. Upravo iz tog razloga što su neke mutacije korisne u evolucijskim algoritmima koristimo i operator mutacije. Pomoću operatora mutacije unosimo sitne promjene u genotip jedinki te postoji mogućnost pojavljivanja novih povoljnih svojstava u fenotipu jedinki, te će se to svojstvo u daljnjim generacijama pojavljivati u sve većem udjelu populacije zbog reprodukcije i selekcije.

3. Genetsko programiranje

Kao i u općenitijim genetskim algoritmima, genetsko programiranje evoluira populaciju jedinki s ciljem optimizacije rješenja za neki predefinirani problem. Konkretno, u genetskom programiranju jedinke koje se evoluiraju su najčešće računalni programi. S evolucijom računalnih programa dobivamo i nove probleme. Kasnije ćemo opisati problem operatora reprodukcije i mutacije u genetskom programiranju koji ne uzimaju u obzir semantiku programa već samo njegovu sintaksu te tako i najmanje promjene koje naprave ti operatori drastično mijenjaju semantiku programa i potencijalno i njegovu dobrotu, što nam često nije od koristi jer želimo kontroliranu evoluciju.

3.1. Sastav jedinke

U genetskom programiranju, jedinka se sastoji od primitiva. Postoje dva tipa primitiva - terminali i funkcije.

Funkcije su primitivi koji primaju neke druge primitive (argumente) na temelju čijih vrijednosti vraća vrijednost razinu iznad. Broj argumenata koje neka funkcija prima (eng. *arity*) varira od funkcije do funkcije. Primjer funkcija su algebarske funkcije za zbrajanje i oduzimanje, logički *I* i *ILI*, u formalnim programima neka korisnički definirana funkcija, i slično.

Terminali su primitivi koji sami po sebi imaju neku definiranu vrijednost. Možemo ih gledati kao funkcije bez argumenata. Primjer terminala su varijable i konstante.

3.2. Vrste jedinki

Kao što je već spomenuto jedinke u genetskom programiranju mogu biti računalni programi, međutim, pokraj takvog načina korištenja genetsko programiranje možemo iskoristiti za optimizaciju više konstrukata:

– računalnih programa,

– *booleovih* funkcija,

– algebarskih funkcija.

U nastavku rada fokus će nam biti na optimizaciji *booleovih* i algebarskih funkcija iz razloga što je semantiku lakše izvesti za takve primjere.

3.2.1. Računalni programi

Pri optimizaciji računalnih programa prvo definiramo problem za koji želimo dobiti optimalan program. Tada pokrećemo rad algoritma i provodimo evoluciju za navedeni problem, svakoj jedinci se dodjeluje dobrota u skladu s njezinim sposobnošću rješavanja definiranog problema.

Na primjer, možemo definirati funkcije koje će program koristiti za kretanje robota-usisavača te mu kao problem postaviti neku prostoriju. Programi s većom dobrotom će biti oni koji prijeđu cijelu površinu sobe u najmanjem mogućem putu.

3.2.2. *Booleove* funkcije

Pri problemima gdje na temelju istinitosne tablice neke *booleove* funkcije trebamo izvesti samu *booleovu* funkciju možemo koristiti genetsko programiranje. Jedinke će u tom slučaju biti *booleove* logičke funkcije te će se sastojati od varijabli i logičkih funkcija.

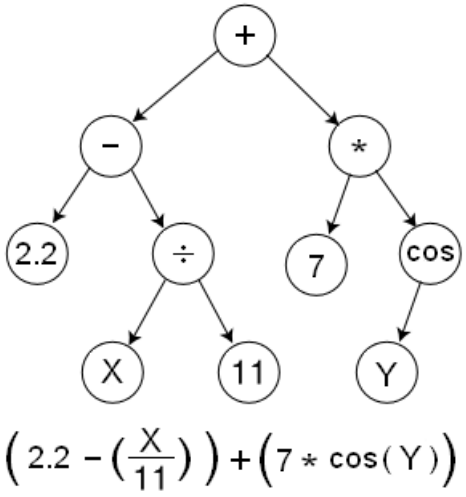
3.2.3. Algebarske funkcije

Kao i kod izvođenja *booleovih* funkcija, genetsko programiranje možemo iskoristiti i za simboličku regresiju algebarskih izraza. Ulazni problem nam je u tom slučaju vektor vrijednosti funkcije u nekim točkama, a jedinke su nam algebarski izrazi. Cilj algoritma je pronalazak algebarske funkcije koja će se sastojati od danih osnovnih funkcija (poput zbrajanja, oduzimanja, množenja...) i koja će najbolje aproksimirati dane vrijednosti.

Na primjer, za vrijednosti funkcije dvije varijabli dane u tablici 3.1, očekivana funkcija koju bi genetsko programiranje trebalo dati kao rješenje jest dana u obliku stabla na slici 3.1. Često nije moguće pomoću genetskog programiranja dobiti funkciju koja se točno poklapa kao u ovom slučaju. Glavni razlozi tome su ograničenje zadanog skupa primitiva i ograničenje same tehnike genetskog programiranja.

Tablica 3.1: Vrijednosti funkcije

x	1	2	3	4	5
y	1	2	3	4	5
f(x, y)	5.891	-0.895	-5.003	-2.739	3.731



Slika 3.1: Ciljna funkcija

3.3. Problemi s genetskim programiranjem

Do sada opisan način funkcioniranja genetskog programiranja dolazi s jednim specifičnim problemom - evolucija jedinki se svodi samo na sintaksne promjene. Problem u tome je što evolucija samo uz pomoć sintaksnih promjena potpuno ignorira semantiku programa (osim u računanju dobrote) te nam to onemogućuje kontroliranu i usmjerenu evoluciju populacije. Već male promjene u sintaksi programa drastično mijenjaju semantiku programa, odnosno njegov izlaz. Posljedice toga mogu biti različite, poput:

- djeca po semantici nemaju nikakve *razlike* s roditeljima,
- djeca po semantici nemaju nikakve *sličnosti* s roditeljima,
- povoljna svojstva mogu nasumično nestajati iz populacije.

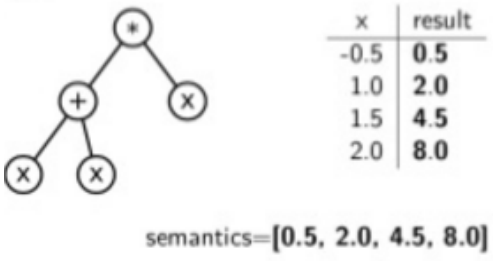
Navedeni problem možemo riješiti uvođenjem semantičke analize u evoluciju programa. Međutim, sada se postavlja problem definiranja semantike programa i kako iskoristiti tu semantiku da se usmjeri evolucija programa. O tome ćemo se baviti u narednim poglavljima.

4. Semantička analiza

Pri uvođenju semantičke analize u genetsko programiranje pred nas se postavljaju novi problemi:

- Kako definirati semantiku jedinke?
- Kako definirati operatore koji postepeno mijenjaju semantiku jedinki?
- Koji algoritmi i/ili operatori su vremenski i memorijski prihvatljivi?

Kroz rješavanje navedenih problema ćemo opisati načine definiranja semantike i njenog korištenja sa svrhom poboljšanja genetskog programiranja.

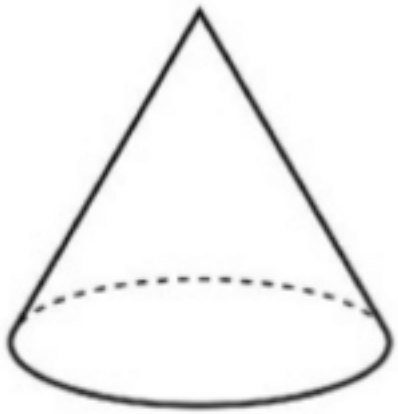


Slika 4.1: Primjer semantike algebarske funkcije [1]

4.1. Definicija semantike

Vjerojatno najlakši i najintuitivniji način za definiranje semantike programa jest definirati ju kao izlaz programa. To je najlakše prikazati na primjeru za *booleove* ili algebarske funkcije. Kao što se vidi na slici 4.1 postoji jedinka koja zapravo predstavlja funkciju $x * (x + x)$ i njen ulazni vektor se sastoji od vrijednosti -0.5, 1, 1.5, 2.0. Semantiku te jedinke predstavlja njen izlaz. U ovom slučaju semantika jedinke je vektor *semantics*. Zanimljivo opažanje jest da semantiku ove jedinke možemo zamisliti kao točku u n-dimenzionalnom prostor (gdje je n duljina vektora *semantics*). Pomoću toga možemo definirati i semantičku sličnost (odnosno različitost) dvije jedinke a i b kao $\mathit{semantics}(a) - \mathit{semantics}(b)$ što će nam biti korisno u daljnjim razmatranjima.

Uz ovakvu definiciju semantike moguće je i izračunati semantiku ne samo cijele jedinke već i pojedinih podstabala jedinke. Najbitnija implikacija ovakve definicije semantike jest da se dobrota jedinke može gledati kao udaljenost izlaznog vektora (odnosno semantike) od optimalnog rješenja. Zbog tog razloga semantički krajolik dobrote (eng. *semantic fitness landscape*) u slučaju korištenja samo semantičkih operatora izgleda kao čunj, slika 4.2. Kada se postigne takav izgled krajolika dobrote puno se olakšava posao pronalaženja optimalne jedinke.



Slika 4.2: Okoliš dobrote čunjastog oblika

4.2. Semantički operatori

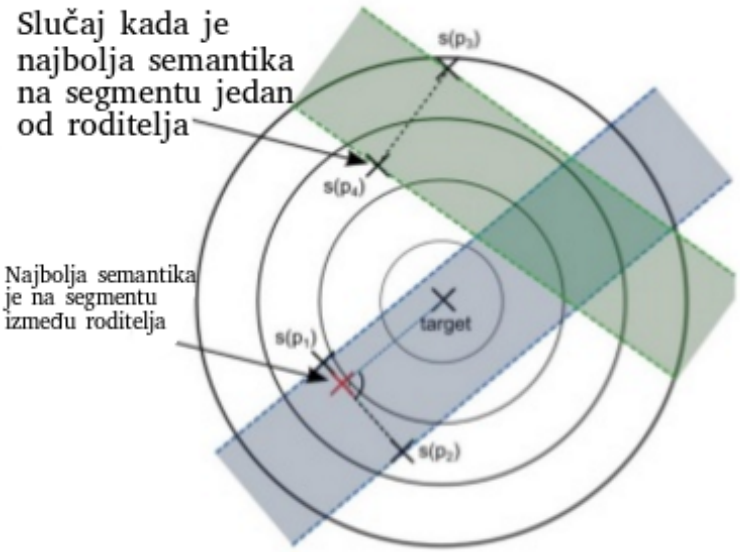
Kao što smo već uočili, razlika semantika dviju jedinki predstavlja njihovu tzv. *"semantičku udaljenost"* u semantičkom krajoliku dobrote. Pomoću tog opažanja možemo definirati semantičke operatore koji će moći kontrolirano usmjeriti tijek evolucije u povoljnom smjeru (tj. prema nastanku optimalne jedinke). Proučiti ćemo dvije vrste semantičkih operatora:

- Aproksimacijski geometrijski semantički operatori
- Geometrijski semantički operatori

Obje vrste se nazivaju *"geometrijskim"* iz razloga što rade na razini semantičkog krajolika dobrote te uzimaju u obzir semantičke odnose jedinki (npr. semantičku udaljenost djeteta od oba roditelja i slično).

4.2.1. Aproksimacijski geometrijski semantički operatori

Aproksimacijski geometrijski semantički operatori rade na principu pokušaja i promašaja (eng. *"trial and error"*). Rade na način da više puta izvode uobičajene operatore za križanje (eng. *crossover*) i mutaciju te evaluiraju dobivene jedinke. Na temelju evaluacije njihovih semantika i na semantičkoj udaljenosti djece i roditelja odlučuju je li dobivena jedinka prihvatljiva ili nije. Druga mogućnost je da od svih stvorenih jedinki uzimaju onu koja je najprihvatljivija po svojoj semantici.



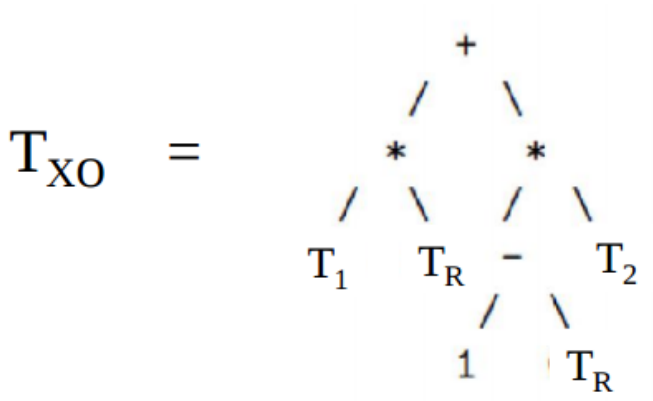
Slika 4.3: Prikaz roditelja i djeteta tijekom *crossovera* u okolišu dobrote [1]

U operatorima križanja pod semantički "prihvatljivom" jedinkom smatramo jedinkom koja se nalazi u semantičkom prostoru između svoja dva roditelja, jedan primjer rezultata dobrog semantičkog križanja se vidi na slici 4.3. Kao što je ranije spomenuto i kao što se vidi na ovoj slici semantički krajolik dobrote čunjastog je oblika, u sredini se nalazi ciljna (optimalna) jedinka. Kada pogledamo *crossover* dolje lijevo na slici, između roditelja $p1$ i $p2$ za povoljno dijete se uzima ona jedinka koja je po semantici na pravcu između svojih dvaju roditelja i najbliže je ciljnoj jedinki (tj. ima najbolju dobrotu). Naravno, u aproksimacijskim operatorima neće uvijek biti moguće izabrati takvu jedinku, kao što je slučaj kod križanja u gornjem dijelu slike. Kod križanja roditelja $p3$ i $p4$ očito je da su sve jedinke koje se nalaze na segmentu između ta dva roditelja na većoj udaljenosti od ciljne jedinke u središtu od roditelja $p4$. U tom slučaju križanje će proizvesti dijete koje je jednako roditelju $p4$ pošto je on jedinka najveće dobrote na segmentu $p3$ - $p4$.

4.2.2. Geometrijski semantički operatori

Princip pokušaja i promašaja često nije vremenski efikasan i još k tome ne daje potomstvo koje se precizno uklapa u semantičke zahtjeve. Iz tog razloga cilj je pronaći operatore koji će direktno iz jedinki roditelja stvoriti semantički prihvatljivo dijete jedinku. Evoluiranje programa uz pomoć semantičke analize zahtjeva više podataka o domeni primjene od klasičnog genetskog programiranja, iz tog razloga nije moguće u potpunosti gledati na probleme kao crne kutije (eng. *black-box*). Dalje ćemo razmatrati operatore za križanje i mutaciju za pojedine slučajeve iz domena *booleovih* funkcija, algebarskih funkcija i formalnih programa.

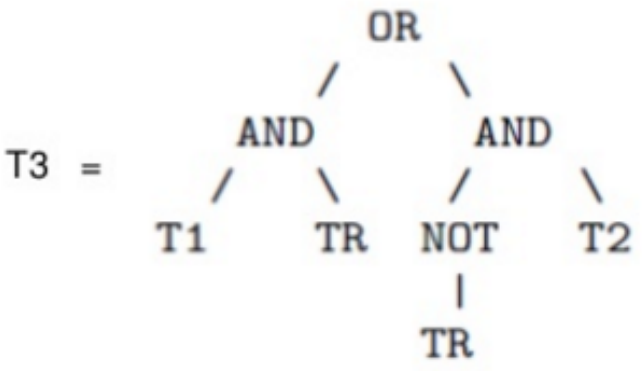
Operator križanja



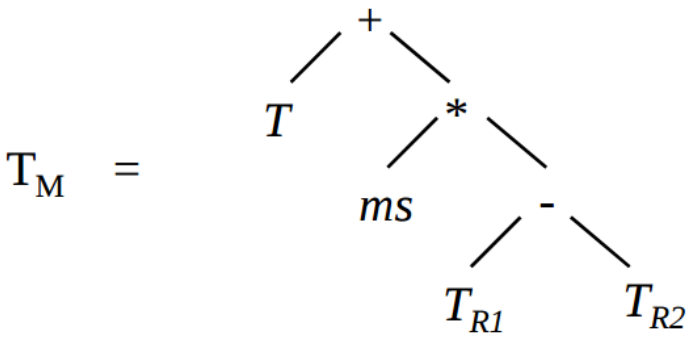
Slika 4.4: Operator križanja za algebarske funkcije[1]

Za algebarske funkcije jednostavan semantički operator križanja prikazan je na slici 4.4. Txo je dijete koje se dobije iz roditelja $T1$ i $T2$, dok je Tr nasumično generirano stablo čija je kodomena u rasponu $[0, 1]$. Kao što se vidi ovo križanje se zasniva na linearnoj kombinaciji roditelja gdje su koeficijenti s kojima se množe roditelji u rasponu $[0, 1]$. Iz tog razloga će se dijete u okolišu dobrote sigurno nalaziti negdje na segmentu između svojih dvaju roditelja, a točna pozicija na segmentu ovisi o vrijednosti nasumično generiranog stabla Tr.

Za boolean funkcije semantički operator križanja prikazan je na slici 4.5. Kao i kod algebarskog križanja $T1$ i $T2$ su roditelji dok je $T3$ njihovo dijete, a Tr je nasumično generirano stablo.



Slika 4.5: *Crossover* operator za boolean funkcije[1]

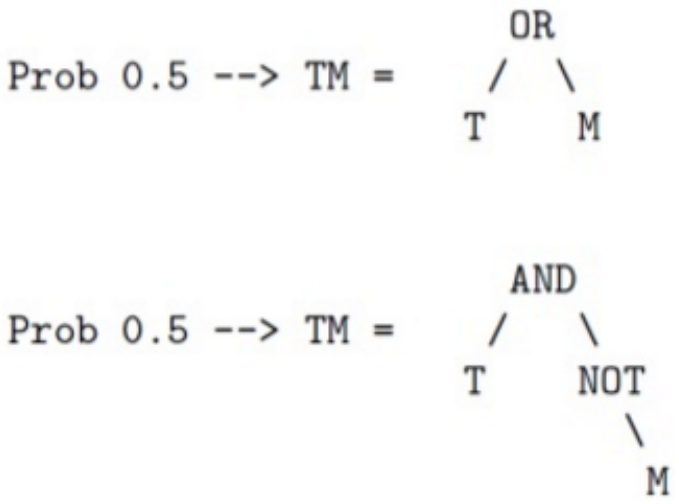


Slika 4.6: Operator mutacije za algebarske funkcije[1]

Operator mutacije

Operatori mutacije su vrlo slični već spomenutim operatorima za križanje. Na slici 4.6 je prikazana semantička mutacija za algebarske funkcije. T jest jedinka koju mutiramo, $Tr1$ i $Tr2$ su nasumično generirane funkcije i ms je konstanta koja se naziva korak mutacije (eng. *mutation step*) a služi za postavljanje finoće mutacije. Što veću konstantu postavimo za korak mutacije to će mutacija imati većeg utjecaja na jedinku. Ovakva mutacija je slaba perturbacija jedinke budući da je razlika $Tr1 - Tr2$ centrirana oko nule, što je u semantičkom smislu točno ono što želimo - mutacijom želimo napraviti manje semantičke promjene na jedinki.

Operator mutacije za boolean funkcije prikazan je na slici 4.7. T je, kao i prije, jedinka na kojoj vršimo mutaciju, a M je nasumično generiran minterm, odnosno boolean funkcija kojoj je točno jedan izlaz u tablici istinitosti *true*. Kao što se vidi sa slike postoje dva moguća slučaja od kojih uzimamo jednog s vjerojatnošću 0.5. Postoji mogućnost preoblikovanja ovog operatora ujedinjenjem ova dva slučaja tako da



Slika 4.7: Operator mutacije za boolean funkcije[1]

zamijenimo *OR* ili *AND* i *NOT* funkcije s *XOR* funkcijom. U tom slučaju bi mutirana jedinka bila oblika T *XOR* M.

4.3. Potencijalni problemi semantičkih operatora

Semantički operatori nam omogućuju finiju kontrolu nad smjerom evolucije ali nisu bez nedostataka. Već je spomenuto da je princip pokušaja i promašaja kod aproksimacijskih operatora opisanih u potpoglavlju 4.2.1. često vremenski neefikasan. Međutim, postoji problem i kod operatora iz prethodnog poglavlja. Kod operatora za križanje može se vidjeti eksponencijalan rast veličina jedinki u populaciji iz generacije u generaciju, što znači da postoji problem velike memorijske složenosti. Na primjer, veličina djeteta kod semantičkog križanja za boolean funkcije je: $size(T3) = 4 + 2 * size(Tr) + size(T1) + size(T2)$ iz čega možemo zaključiti da prosječna veličina jedinke iz generacije u generaciju raste eksponencijalno i to (otprilike) s faktorom 2. Kod operatora za mutaciju taj problem nije toliko značajan iz razloga što se kod njih manifestira linearan rast jedinki. Na primjer, za prikazanu mutaciju boolean funkcija veličina djeteta jest $size(TM) = 2 + size(M) + size(T)$, pošto želimo biti što efikasniji funkcija M će biti što manja pa možemo pretpostaviti linearan rast jedinki iz generacije u generaciju.

4.3.1. Moguća rješenja za eksponencijalan rast

Pojednostavljenje djece

Jedno od mogućih rješenja za eksponencijalan rast jedinki jest pojednostavljenje jedinki (bez gubitka informacija). Pod "pojednostavljenje" podrazumijevamo, na primjer, algebarsku simplifikaciju izraza u slučaju algebarskih funkcija. Jednostavan primjer algebarske simplifikacije je sljedeći – u slučaju da evoluira jedinka $x * x + x - x$ moguće ju je simplificirati bez gubitka informacija na oblik $x * x$.

Teškoće s ovim rješenjem su u različitosti domena i odsustvu generičkog postupka pojednostavljenja. Ovakva pojednostavljenja su, također, vremenski skupa. Dobra strana ovog pristupa je čitljivost krajnjih rješenja.

Izbacivanje križanja

Kao rješenje eksponencijalnog rasta od operatora možemo koristiti samo mutaciju kako je operator križanja glavni faktor u problemu eksponencijalnog rasta jedinki. Često je linearan rast jedinki uzrokovan semantičkom mutacijom prihvatljiv. Makar izbacivanje križanja iz evolucije mijenja tijek evolucije, korištenje samo mutacije često proizvodi dovoljno dobre rezultate. Nedostatak ovog pristupa je pojava čovjeku nečitkih rješenja.

Kompaktifikacija

Najkompleksnije rješenje od svih navedenih ovdje je kompaktifikacija. Ideja kompaktifikacije je da se u memoriji ne stvaraju nove jedinke već da se djeci daju pointeri na svoje roditelje. U detaljnije objašnjenje implementacije ovog pristupa nećemo ulaziti. Kako se svi operatori zasnivaju na nekakvoj vrsti linearne kombinacije jedinki ovo je rješenje valjano. Prednosti ovog rješenja su linearan rast korištenja memorije čak i s korištenjem semantičkog križanja te mogućnost korištenja neovisno o domeni. Očiti nedostaci ovog pristupa su kompliciranа implementacija i nemogućnost čitanja rješenja (rješenja su crne kutije).

5. Implementacija algoritma SHC

Za programsku implementaciju ovog rada implementiran je semantički uspon na brdo (eng. SHC - *semantic hill climber*) u sklopu genetskog programiranja sa semantičkom analizom.

5.1. *Evolutionary computation framework*

Implementacija je pisana uz pomoć *Evolutionary Computation Framework-a* (ili kraće ECF-a). Ovo okruženje pisano u C++-u je načinjeno za razne primjene u evolucijskom računarstvu, između ostaloga i za genetsko programiranje. Međutim, nedostaje podrška za bilo kakvu semantičku analizu u genetskom programiranju te je tog razloga nastao je ovaj rad.

5.2. Algoritam SHC

Razlog odabira algoritma SHC je djelomično objašnjeno u prijašnjem poglavlju kada se pregledavaju problemi semantičkih operatora i rješenja tih problema, konkretno, algoritam ne koristi operator križanja te time sprječava eksponencijalan rast veličine jedinki. Još neki od razloga za odabir ovog algoritma su jednostavnost implementacije tj. algoritam se "lijepo uklapa" u već postojeće okruženje.

Kao što je već prije napomenuto ovaj algoritam koristi samo operator mutacije te se iz tog razloga naziva algoritmom "uspona na brdo". Mutacija omogućava jedinkama da se korak po korak pomiču po okolišu dobrote prema boljim rješenjima.

5.2.1. Pseudokod

Slijedi pseudokod algoritma SHC[2].

Ovo je jednostavan prikaz rada algoritma ali u njemu neke stvari ostaju nedorečene. Kao što se vidi algoritam radi dok god nije postignut uvjet za kraj evolucije. Taj uvjet

Algorithm 1 Semantic hill climber

```
initialize(population)
n = size(population)
repeat
  for (i := 0; i < n; i++) do
    nTries := 0
    repeat
      individual = population[i]
      mutated = semanticMutation(individual)
      nTries := nTries + 1
    until isBetterThan(mutated, individual) or nTries >= limit
    if isBetterThan(mutated, individual) then
      population[i] := mutated
    else
      reinitialize(population[i])
    end if
  end for
until terminationCondition == true
```

može biti:

- dostignut maksimalan broj generacija,
- dostignut broj generacije bez napretka,
- evoluirana jedinka zadovoljavajuće dobrote.

U svakoj iteraciji algoritma, algoritam svaku jedinku u populaciji pokušava mutirati (semantičkim operatorom za mutaciju) dok god mutacijom ne uspije poboljšati dobrotu jedinke ili dok ne dosegne maksimalan broj pokušaja (varijabla *limit*). U slučaju da je mutirana jedinka bolja od originalne, originalna jedinka u populaciji se zamjeni s mutiranom, a u slučaju neuspjeha mutacije da poboljša jedinku, jedinka se reinicijalizira.

5.2.2. Implementacija u ECF

Ovdje je napisan glavni isječak konkretne implementacije algoritma u ECF u jeziku C++. Isječak se odnosi na logiku koja se događa za jednu generaciju evolucije.

```
bool advanceGeneration(StateP state, DemeP deme){
  for(uint i = 0; i < deme->getSize(); i++){
    evaluate(deme->at(i));

     /* Keep trying to mutate individual for
      *  maximum of limit_ number of tries or if
      *  you get a good mutation */
    bool goodMutation = false;
    for(uint j = 0; j < limit_; j++){
      IndividualP mutated = static_cast<IndividualP>(deme->at(i)->copy());

       /* Try to mutate, mutation return
        * 'false' if trees depth crosses
        * the maxDepth limit set in parameters */
      if(!mutate(mutated))
        continue;

      evaluate(mutated);
      if(mutated->fitness->isBetterThan(deme->at(i)->fitness)){
        goodMutation = true;
        deme->replace(i, static_cast<IndividualP>(mutated));
        break;
      }
    }

     /* If you didn't get a
      * good mutation, reinitialize the individual */
    if(!goodMutation){
      deme->at(i)->initialize(state);
    }
  }

  return true;
}
```

Semantički operator mutacije

Osim navedene implementacije algoritma potrebno je implementirati i operator za semantičku mutaciju. Glavna logika tog operatora (za algebarske funkcije) dana je u nastavku u obliku pseudokoda. Pretpostavka je da su jedinke (stabla) prikazana kao vektori s prefiksnom notacijom operanada i operatora.

Algorithm 2 Semantic mutation operator[1]

$tree_1 := generateRandomTree()$
$tree_2 := generateRandomTree()$
$newTree.add(operator('+'))$
$newTree.add(oldTree)$
$newTree.add(operator('-'))$
$newTree.add(tree_1)$
$newTree.add(tree_2)$
$returnnewTree$

Semantički operator križanja

Za još jedan algoritam koji će kasnije biti ispitivan i koji kombinira semantičke i klasične operatore križanja i mutacije još moramo navesti implementaciju operatora semantičkog križanja. Kao i za mutaciju, pretpostavka je da su jedinke stabla prikazana kao vektori s prefiksnom notacijom.

Algorithm 3 Semantic crossover operator[1]

$rand_tree := generateRandomTree()$
$newTree.add(operator('+'))$
$newTree.add(operator('*'))$
$newTree.add(parentTree1)$
$newTree.add(rand_tree)$
$newTree.add(operator('*'))$
$newTree.add(operator('-'))$
$newTree.add(constant(1))$
$newTree.add(rand_tree)$
$newTree.add(parentTree2)$
$returnnewTree$

6. Rezultati i moguće nadogradnje

6.1. Rezultati

Ispitana je opisana implementacija algoritma SHC (oznaka *GPwS*), klasično genetsko programiranje (oznaka *GP*) i algoritam koji za operatore mutacije i križanja koristi i semantičke i klasične verzije (oznaka *GPcomb*). Za dobrotu koristi se pogreška jedinke u odnosu na ciljnu funkciju, što znači da se u ovom slučaju dobrota nastoji minimizirati. U eksperimentima svaka evolucija traje dok se ne pronađe optimalno rješenje (*fitness* $= 0$) ili dok ne prođe 100000 evaluacija dobrota jedinki. Svaki eksperiment je ponovljen 10 puta te je kao rezultat ispitivanja u tablicu upisan prosjek dobrote najboljeg rješenja u tih 10 ponavljanja. Broj jedinki u populaciji postavljen je na 100, a maksimalna dubina stabla svake jedinke postavljena je na 5 što se pokazalo vremenski prihvatljivom opcijom čak i za brz rast jedinki koji je specifičan za semantičke operatore.

Klasično genetsko programiranje u eksperimentima koristi turnirski algoritam (eng. *steady state tournament*) i uz njega koristi nasumično odabrane operatore za mutaciju i križanje.

Treći kandidat u usporedbama je, također, turnirski algoritam, ali osim klasičnih operatora mutacije i križanja koristi i semantičke operatore za mutaciju i križanje. Konkretne vjerojatnosti za odabir operatora križanja su:

- križanje s jednom točkom prijeloma (eng. *one point*) - 90
- semantičko križanje - 10

Vjerojatnosti za odabir operatora mutacije su:

- smanjujuća mutacija (eng. *shrinking*) - 40
- mutacija podrezivanje (eng. *hoist*) - 40
- semantička mutacija - 20

6.1.1. Algebarske funkcije

Korišten je skup funkcija:

- zbrajanje (+),
- oduzimanje (-),
- dijeljenje (/),
- množenje (*),
- sinus (sin),
- kosinus (cos),
- korijenovanje (sqrt).

Ispitivanje se vršilo na realne funkcije jedne i dvije varijable. Za svaki eksperiment zadana je ciljna funkcija i raspon vrijednosti varijabli funkcija, varijable su uzorkovane korakom 1.

Tablica 6.1: Ispitne algebarske funkcije

Oznaka	Funkcija	Raspon varijabli
F1	$tan(x^2 - x) + x/5$	$x = [-20, 20]$
F2	$x^5 + 2 * x^3 + 4.5 * x^2 - 3.14 * x$	$x = [-20, 20]$
F3	$sin(sqrt(\|x\|))$	$x = [-20, 20]$
F4	$sin(x^2 + x) - cos(3 * x)$	$x = [-20, 20]$
F5	$log(x + 1) + log(x^2 + 1)$	$x = [0, 40]$
F6	$sin(x) + sin(y^2)$	$x, y = [-10, 10]$
F7	$x * y + sin((x + 1) * (y - 1))$	$x, y = [-10, 10]$
F8	$8/(2 + x^2 + y^2)$	$x, y = [-10, 10]$
F9	$x^3/5 + y^3/2 - x - y$	$x, y = [-10, 10]$

U tablici 6.1 su upisane sve ispitne funkcije i njihove pripadne oznake, a u tablici 6.2 upisani su rezultati pojedinog eksperimenta (funkcije) i medijani 10 najboljih rješenja pronađenih od strane, redom, klasičnim genetskim programiranjem (GP), algoritmom SHC (GPwS) i kombinacijom klasičnih i semantičkih operatora križanja i mutacije (GPcomb). Za svaki eksperiment (redak u tablici) najbolji algoritam je masno otisnuti. Iz rezultata se vidi da nema jasnog pobjednika u ovom eksperimentu. Osim kod druge, polinomijalne, funkcije algoritm SHC performansama parira klasičnom GP-u te je čak u nekim ispitivanjima za nijansu bolji. Kada se pogledaju funkcije

Tablica 6.2: Rezultati ispitivanja - Algebarske funkcije

Funkcija	GP medijan	GPwS medijan	GPcomb medijan
F1	81.67	**65.10**	95.15
F2	2829.66	1.25461+e6	**1971.06**
F3	13.38	**11.61**	14.72
F4	**21.76**	21.80	23.04
F5	**16.59**	18.95	44.84
F6	354.42	349.54	**348.36**
F7	**12107**	12114.05	12111.50
F8	**106.83**	106.95	106.96
F9	72763.8	73102.55	**72584.9**

u kojima je bolji, vidi se da su to funkcije relativno malih vrijednosti i skokova gdje je potrebno finije ugađanje što je upravo i princip rada algoritma SHC.

Iz grafova na slici 6.1 vidi se kako SHC konzistentnije konvergira prema lokalnom optimumu od klasičnog GP-a. Što se tiče GPcomb-a, također se vidi konzistentnija konvergencija prema optimumu ali zbog klasičnih operatora mutacije i križanja vide se i nasumični skokovi koji ponekad rezultiraju povoljnim rješenjima, poput anomalija (*outliera*) na donjem lijevom grafu na slici 6.1.

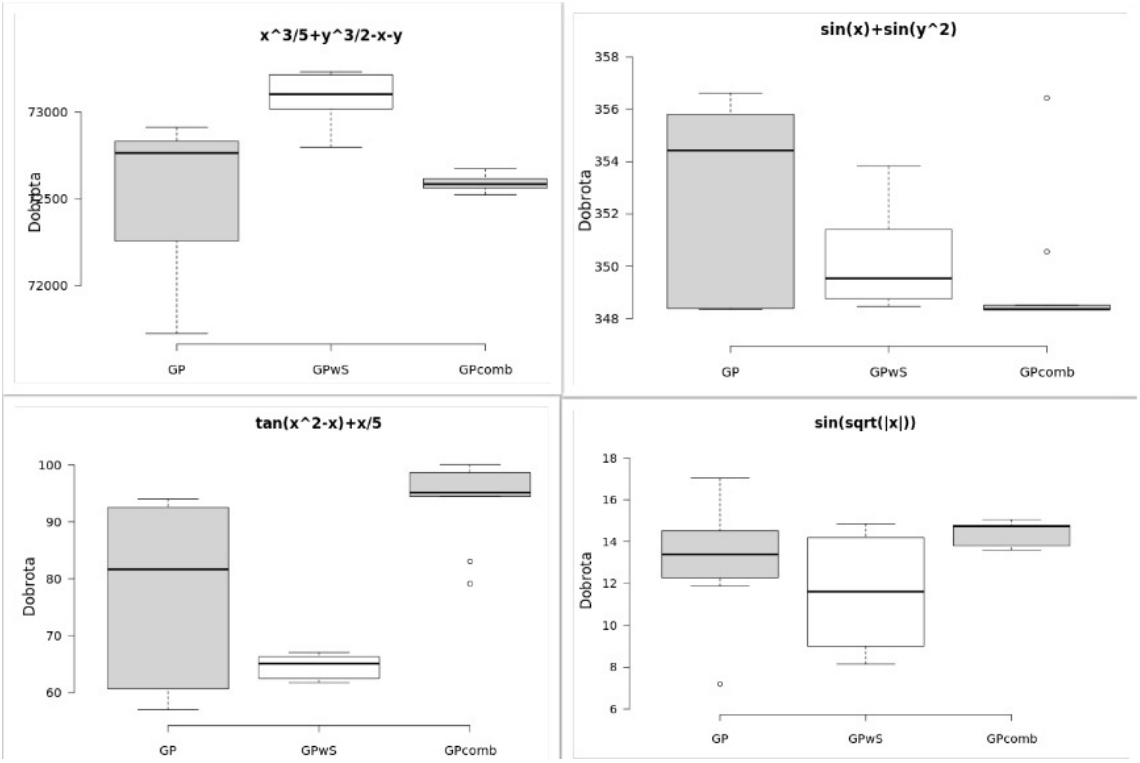
6.1.2. Boolean funkcije

Korišten je skupa booleovih primitiva:

- logički 'I' (AND),
- logički 'ILI' (OR),
- logički 'XILI' (XOR),
- logički 'NE' (NOT).

Ispitivanje se obavljalo na način da se algoritmu zada cijela istinitosna tablica, što je zapravo evaluacija ciljne logičke funkcije za sve moguće kombinacije ulaznih varijabli. To znači da je istinitosna tablica koja se daje algoritmu kao ulaz zadana tako da svaki njen redak predstavlja jednu kombinaciju ulaznih varijabli te pripadajući izlazni bit. Ukupan broj redaka u istinitosnoj tablici je 2^n gdje je n broj ulaznih varijabli funkcije.

Kao funkciju dobrote algoritam, kao i u prethodnom problemu s algebarskim funkcijama, koristi ukupnu pogrešku u odnosu na ciljnu funkciju. Evaluira se jedinka te



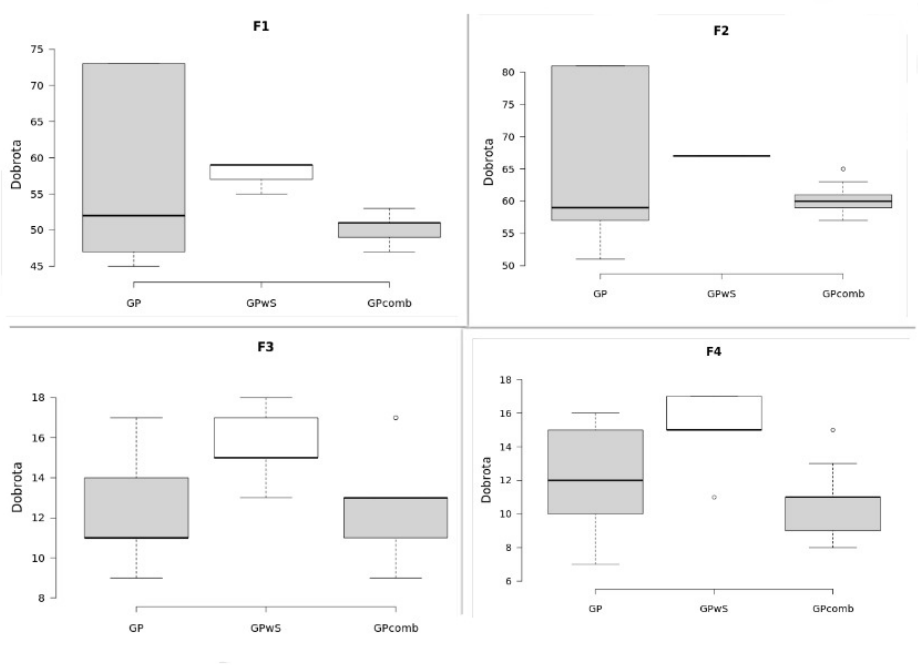
Slika 6.1: Dijagrami s uzorcima za 4 ispitne funkcije

se dobije izlazni vektor bitova veličine 2^n, taj vektor se uspoređuje sa izlaznim vektorom ciljne funkcije te se za svaki različiti bit dobrote jedinke poveća za 1. Kako je i u ovom slučaju dobrota zapravo ukupna greška jedinke ponovno nam je cilj minimizirati pogrešku, odnosno dobrotu.

Tablica 6.3: Rezultati testiranja

Oznaka	GP medijan	GPwS medijan	GPcomb medijan
F1	52	59	**51**
F2	**59**	67	60
F3	**11**	15	13
F4	12	15	**11**

U tablici 6.3 nalaze se rezultati ispitivanja, medijani za 10 najboljih jedinki koje su proizveli svaki od 3 algoritma. Kao što se vidi po rezultatima algoritam SHC (GPwS) lošiji je od klasičnog GP i kombiniranog GP-a, razlog tome vjerojatno leži u činjenici da SHC radi premale korake da bi bio u mogućnosti izvući se iz lokalnog optimuma, što mu kod booleovih funkcija predstavlja veći problem nego kod algebarskih funkcija iz prošlog poglavlja. Klasični i kombinirani GP su prilično izjednačeni po pitanju



Slika 6.2: Dijagrami s uzorcima za 4 ispitne booleove funkcije

performansi.

Na slici 6.2 nalaze se dijagrami koji prikazuju razdiobu uzoraka pojedinih algoritama u svakom eksperimentu. Podaci su u skladu s očekivanjima, klasični GP prikazuje relativno široku razdiobu u odnosu na konzistentno kombiniranog GP-a i algoritma SHC. Kao i kod eksperimenata s algebarskim funkcijama i ovdje se vide anomalije kod kombiniranog GP-a (točke daleko od medijana u funkcijama F2,F3 i F4).

6.2. Moguće nadogradnje

6.2.1. Algebarske funkcije

Za dosadašnju implementaciju semantike u kontekstu algebarskih funkcija moguće su sljedeće potencijalno korisne nadogradnje:

- funkciju za evaluaciju napisati tako da preferira kraća rješenja,
- dodavanje koeficijenta za korak mutacije u semantički operator za mutaciju,
- dodavanje pojednostavljenja algebarskih izraza,

– uz semantičku mutaciju otkriti kako iskoristiti i semantičko križanje.

Dodavanjem koeficijenta za korak mutacije omogućilo bi se finije podešavanje razine važnosti mutacije, uz to, ublažio bi se (ili potpuno uklonio) problem viđen kod polinoma jer bi se semantici omogućili "veći skokovi". Međutim, uz njegovo dodavanje rješenja će postati nečitljivija i veća zbog dodatne operacije množenja u svakoj iteraciji mutacije.

Pojednostavljenjem izraza riješili bi se najveći problemi semantičke analize – glomazna rješenja. Problem implementacije nekog od takvih algoritama za pojednostavljenje je njihova glad za hardverskim resursima, a naša je potreba iskoristiti ih u svakoj generaciji na svaku jedinku. Ako bi takve algoritme htjeli iskoristiti nakon same evolucije, pitanje je koliko bi u tom slučaju bili od koristi s obzirom na potencijalna velika i kompleksna rješenja.

6.2.2. Boolean funkcije

Osim pojednostavljenja izraza kao i kod algebarskih funkcija moguća nadogradnja za boolean funkcije jest promjena operatora za semantičku mutaciju. Operator trenutno radi na taj način da radi XOR operaciju starog stabla sa rezultatom AND operacije nad dvaju nasumično generiranih stabla. Makar vrijednost tog AND izraza teži prema tome da su sve vrijednosti njegovog izlaznog vektora bitova nula, tako da mutacija ne radi prevelike "skokove", očita nadogradnja bi bila da se radi XOR operacija starog stabla s nekim *minterm* izrazom. Na taj način semantička mutacija forsira promjenu samo jednog bita sa svakom operacijom mutacije.

7. Zaključak

Zbog svojih implementacijskih problema, potencijalno velike memorijske složenosti i problema s čitljivošću rješenja, postoji mogućnost da se semantički postupci ne koriste samostalno već uz neke klasične metode genetskog programiranja. Kao što se i vidjelo u ispitivanju semantika se može koristiti za fino ugađanje jedinki koje se već nalaze vrlo blizu potencijalnog optimalnog rješenja.

Uz sve do sad rečeno možemo zaključiti da se semantička analiza za genetsko programiranje još uvijek može jako unaprijediti. Dosadašnji rezultati (i ovdje prikazani i oni iz ostalih izvora [2]) su prilično obećavajući te je ovo područje genetskog programiranja koje nije za odbacivanje.

LITERATURA

1. Alberto Moraglio, Krzysztof Krawiec. Semantic genetic programming.
   http://www.slideshare.net/AlbertoMoraglio/semantic-genetic-programming-tutorial

2. Leonardo Vanneschi. Semantics and fitness landscapes in genetic programming.
   https://ciencias.ulisboa.pt/sites/default/files/fcul/dep/di/Vanneschi.pdf

3. Riccardo Poli, William B. Langdon, Nicholas F. McPhee. A field guide to genetic programming.
   http://www.gp-field-guide.org.uk/

Genetsko programiranje sa semantičkom analizom

Sažetak

Postojeći principi i tehnike u genetskom programiranju ne uzimaju u obzir semantiku programa (ili funkcija) koje evoluiraju. Iz tog razloga uvodimo semantičku analizu u genetsko programiranje sa ciljem dostizanja efikasnije evolucije. U radu se opisuje teorija semantičke analize, moguće implementacije te konkretna implementacija algoritma semantičko uspinjanje na brdo (eng. *semantic hill climber*).

Ključne riječi: ECF, evolucija, računarstvo, algebra

Genetic programming with semantic analysis

Abstract

Existing principles and techniques regarding genetic programming do not take into account semantics of the programs (or functions) which are being evolved. Due to that reason semantic analysis has been researched with the goal of making genetic programming more efficient. Theory of semantic analysis, possible implementations and concrete implementation of semantic hill climber algorithm in are all explained in this thesis.

Keywords: ECF, evolution, computer science, algebra