SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 957

USPOREDBA UČINKOVITOSTI PRIKAZA STABLA U OPTIMIZACIJSKIM POSTUPCIMA

Dražen Popović

Zagreb, lipanj 2009.

Sadržaj

Ovaj rad posvećen je mojoj najdražoj „beautiful" tetki Željki, mamki Slavki, čaletu „Ćolimu" Zoranu te najboljem starijem burazu na svijetu Nenadiću.

Vi ste moj oslonac i moj temelj, bez Vas nema mene i nema ove pobjede...naše pobjede.

Uvod

Optimizacija, kao pojam u kontekstu matematike i računarske znanosti, jest postupak u kojemu je cilj izabrati najbolji element među dostupnim alternativama. Današnji broj optimizacijskih postupaka je velik te uključuje, uz ostale, i heurističke postupke. Heuristički postupci optimizacije na kojima se ovaj rad bazira jesu genetski algoritmi i genetsko programiranje i dio su evolucijskog računarstva.

Problematika koju ovaj rad obrađuje je optimizacijske prirode, tj. cilj je povećati učinkovitost optimizacijskih postupaka. Fokus rada je na stablenoj strukturi podataka, učinkovitoj programskoj implementaciji i interpretaciji iste u sklopu genetskih algoritama, preciznije genetskog programiranja. Stablo, kao struktura podataka, u obliku stablenog genotipa predstavlja osnovu genetskog programiranja, te učinkovita izvedba stablenog genotipa poboljšava cijeli postupak optimizacije.

Opisana su dva programska rješenja stablenog genotipa. Ona se razlikuju ne samo u memorijskom zapisu stablene strukture već i u programskoj tehnici na kojoj su izgrađena. Jedan genotip je ostvaren prefiksnim zapisom stabla u memoriji (polje čvorova u prefiksnom obliku). Drugi genotip je ostvaren infiksnim zapisom stabla u memoriji (čvor roditelj sadrži pokazivače na djecu čvorove), te se pomoću programske tvorevine jezika C++, znane kao pokazivači na funkcije, pokušalo zaobići neželjene nuspojave objektnih mehanizama programskog jezika C++. Oba genotipa su testirana na stvarnim problemima iz svijeta genetskog programiranja. Problemi nad kojima ćemo mjeriti performanse genotipa jesu simbolička regresija i centriranje vozila.

Programska rješenja stablenih genotipa i pripadajućih evolucijskih operatora (križanje, mutacija, evaluacija) napravljena su u sklopu *Evolutionary Computation Frameworka* (skraćeno ECF). ECF jest C++ programska biblioteka koja predstavlja okruženje za razvoj aplikacija iz područja evolucijskog računarstva.

U prvom poglavlju *„Struktura podataka stablo“* opisuje se osnovna struktura stabla kao podatkovne strukture, uvodi se pojam čvora, dubine i veličine stabla i čvora, različiti zapisi stabla u memoriji. Nadalje, u drugom poglavlju *„Genetsko programiranje“* uvodi se u domenu genetskog programiranja. Opisuje se generički model stablenog genotipa te

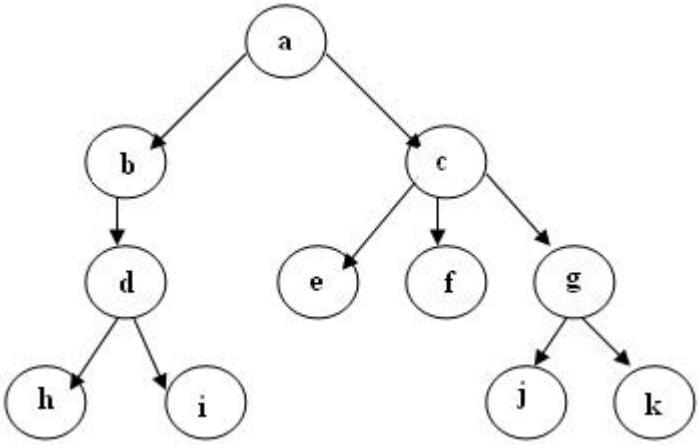
pripadajući genetski operatori (križanje, mutacija, evaluacija). Također se formuliraju navedeni problemi domene genetskog programiranja, simbolička regresija i centriranje vozila. Slijedi treće poglavlje *„Opis rada ECF-a“* gdje je ukratko opisano to programsko okruženje i njegov način rada. *„Genotip Tree“* je četvrto poglavlje i u njemu je detaljno opisan prvi genotip, odnosno njegovo programsko rješenje (uključujući genetske operatore). *„Genotip TreePtr“* jest peto poglavlje u kojemu je opisan drugi genotip. Slijedi šesto poglavlje u kojem se nalaze detaljno prodiskutirani rezultati mjerenja učinkovitosti genotipa i pripadajućih operatora nad zadanim problemima (simbolička regresija, centriranje vozila).

1. Struktura podataka stablo

U računarskoj znanosti stablo, kao struktura podataka, je često korištena struktura za prikaz podataka u hijerarhijskom odnosu. Kao takvo našlo je svoje mjesto u širokom spektru primjena u računarstvu. Različite programske implementacije stablene strukture očituju se u različitim performansama i zauzećem računalnih resursa. Opisana su dva načina zapisa stabla u memoriji i njhove prednosti i mane, s obzirom na količinu memorije koju zauzimaju i procesorsko vrijeme koje se troši pri interpretaciji tih stablenih struktura.

1.1. Opis stablene strukture podataka

Stablo je konačan skup čvorova sa svojstvima. Postoji poseban čvor koji se naziva *korijen* (eng. *root*) i predstavlja prvi čvor stabla. Ostali čvorovi su podijeljeni u *k* disjunktnih podskupova od kojih je svaki stablo i nazivaju se *podstabla*. *Stupanj* čvora jest broj podstabla tog čvora.



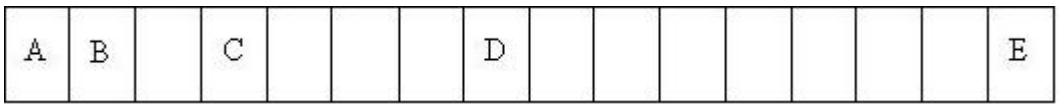
Sl. 1.1 Osnovni model stablene strukture

Referencirajući se na sliku (Sl. 1.1) stupanj čvora s oznakom *„d“* jednak je 2 jer ima dva postabla sa korijenskim čvorovima *„h“* i *„i“*. Čvor *„a“* predstavlja korijen stabla, a skup čvorova {*h,i,e,f,j,k*} skup krajnjih čvorova (*listovi)*. Korijeni podstabla nekog čvora (npr.

čvora „a“) su *djeca* tog čvora (čvorovi „b“ i „c“), a taj čvor nazivamo *roditeljem. Dubina čvora* određuje se iz definicije da je korijen razine 1, a da su razine djece nekog čvora razine *k* jednaka *k+1*. *Dubina stabla* jest maksimalna dubina čvora u tom stablu, tako u ovom slučaju dubina stabla jest 4 jer se na toj dubini nalaze listovi *„h“*, *„i“*, *„j“* i *„k“*. *Veličina čvora* jest broj čvorova u podstablima tog čvora, a *veličina stabla* jest broj čvorova tog stabla.

1.2. Prikaz stabla u memoriji

Zapisi stabla u memoriji se mogu ostvariti na različite način. Statičke strukture poput polja konstantnih veličina se mogu koristiti za prikaz binarnih[1] stabala ili općenito stabala sa konstantnim stupnjem čvorova. Takve strukture su se pokazale memorijski neučinkovite zbog količine prostora koja ostaje neiskorištena.



Sl. 1.2 Koso binarno stablo u statičkom polju

Na gornjoj slici (Sl. 1.2) je prikazano koso[2] binarno stablo unutar polja konstante veličine koje zapravo predstavlja, što se memorijskih resursa tiče, najgori slučaj binarnog stabla. Prednost ovakvog prikaza stabla jest u vremenskoj složenosti pristupa čvorovima O(1), ali bilo kakva modifikacija stabla zahtjeva skidanje i stavljanje elemenata stabla što je vremenski veoma zahtjevno.

Nadalje, zahtjevi na stablene strukture ne uključuju samo binarna stabla već stabla proizvoljne i promjenljive veličine, te različitih stupnjeva čvorova. Tu u priču ulaze dinamičke strukture poput dinamičkih polja i struktura čvorova sa pokazivačima na djecu.

---

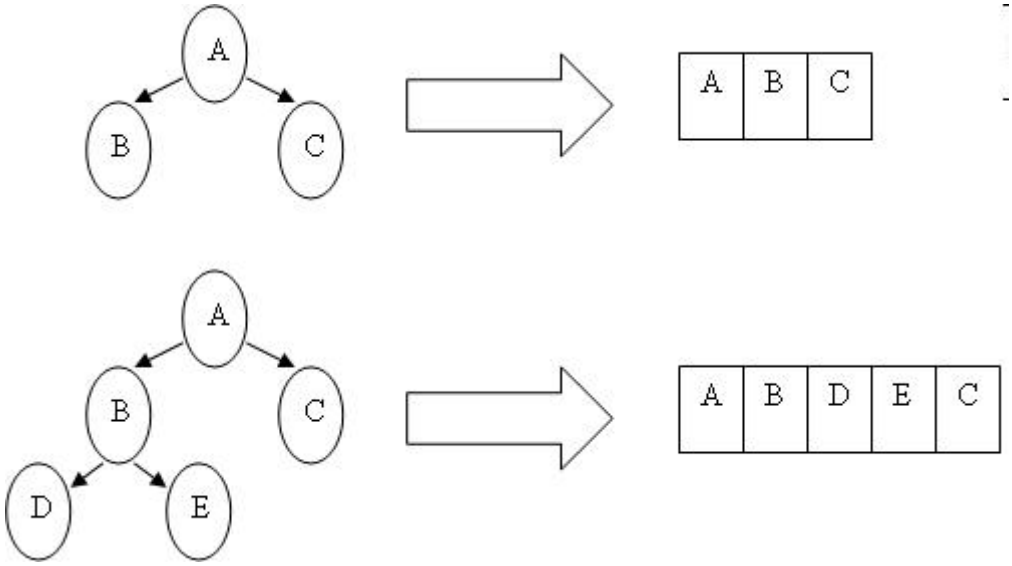
[1] Binarno stablo je stablo koje se sastoji od nijednog, jednog ili više čvorova drugog stupnja.

[2] Koso binarno stablo je binarno stablo u kojem čvorovi imaju samo po jedno dijete.

Prije nego definiramo stablenu strukturu izvedenu preko dinamičkog polja, definiramo strukturu čvora kao građevnu jedinicu stabla.

```
struct cvor{
  tip podatak;
  int brojDjece;
};
```

Pri opisu strukture koristimo programski jezik C/C++. Struktura čvora je veoma jednostavna i sadrži podatkovni član *podatak* koji predstavlja svojstvo čvora podatkovnog tipa *tip*, te sadrži cjelobrojni podatkovni član *brojDjece* koji predstavlja broj djece (podstabla) tog čvora. Upravo član *brojDjece* omogućuje ovoj izvedbi stablene strukture da, memorijski, optimalno iskoristi dinamičko polje jer elementi (čvorovi) se u polje slažu slijedno bez preskakanja polja i u prefiksnom obliku.



Sl. 1.3 Stablena struktura (binarnog stabla) u dinamičkom polju

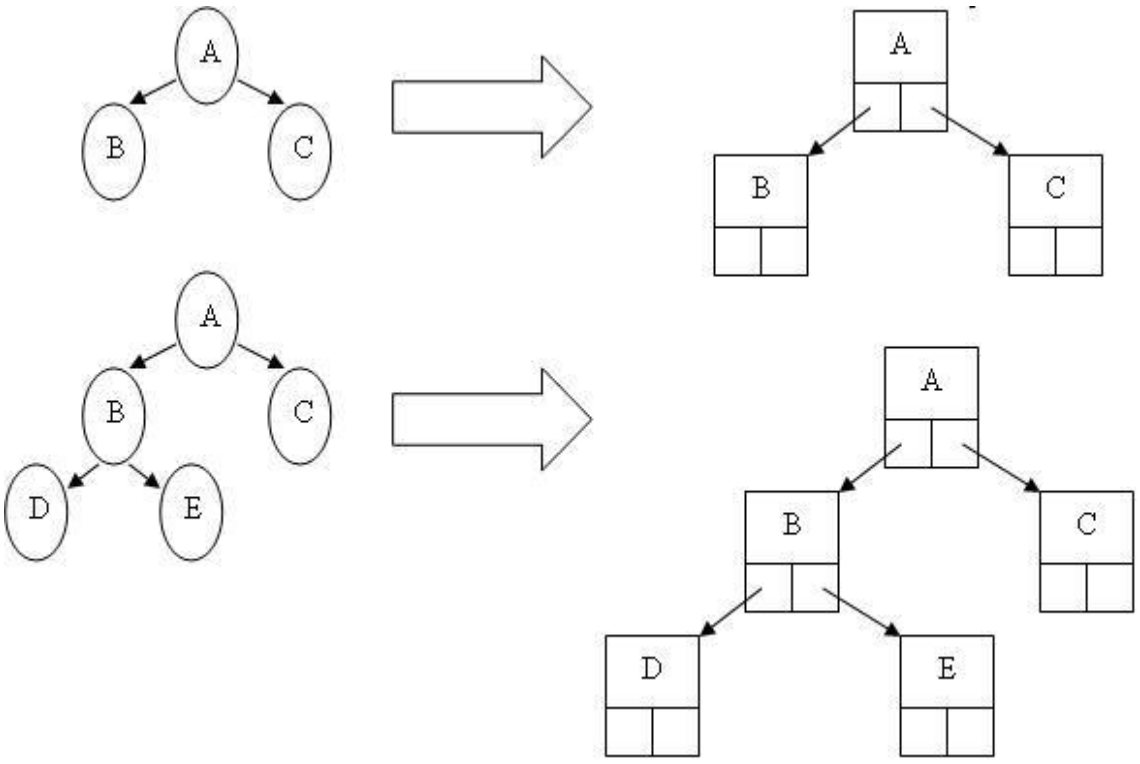
Jednako kao i kod statičkog polja, manu ove izvedbe predstavlja vremenska zahtjevnost modifikacije stabla. Na slici (Sl. 1.3) sa lijeve strane su dva osnovna modela stabla, a sa desne ta ista stabla pohranjena u dinamičko polje. Donje stablo je proširenje gornjeg stabla u čvoru s oznakom „*B*“ na način da mu je dodano dvoje djece (čvorovi s oznakom „*D*“ i „*E*“). Postupak koji bi modifikacijom gornjeg stabla napravio donje bi zahtijevao micanje čvora s oznakom „*C*“ iz polja, dodavanje novih čvorova „*D*“ i „*E*“ te naposljetku vračanje čvora „*C*“ u polje. Također vremenska složenost pristupa

određenom čvoru više nije O(1) već se slijedno pretražuje stablo. Može se primijetiti prefiksni oblik zapisa stablene strukture u polje. Genotip *Tree* koji je opisan u poglavlju 4 izveden je preko dinamičkih polja.

Napredak u vremenskoj zahtjevnosti modifikacije stabla se dobiva izvedbom stabla preko strukture čvora u kojoj su i pokazivači na djecu. Ta struktura čvora opisana u programskom jeziku C/C++ izgleda ovako:

```
struct cvor{
  tip podatak;
  int brojDjece;
  struct cvor pokazivaciNaDjecu[MAX_DJECE];
};
```

Glavna mana ove izvedbe jest u memorijskom trošku uzrokovanom podatkovnim članom strukture *pokazivaciNaDjecu.* Taj član predstavlja statičko polje u kojem se nalaze pokazivači na djecu čvorove i s obzirom na broj djece pojedinog čvora i na veličinu stabla, može predstavljati velik trošak na memorijske resurse.



Sl. 1.4 Stablena struktura (binarnog stabla) sa pokazivačima na djecu

Sa slike (Sl. 1.4) se može vidjeti jednostavnost u modificiranju stabla. Naime, od gornjeg stabla se dobiva donje na način da se čvoru „*B*“, koji proširujemo djecom „*D*“ i „*E*“, u polje pokazivača na djecu *pokazivaciNaDjecu* stave pokazivači na stvorene objekte (čvorove „*D*“ i „*E*“). Vremenska složenost pristupu čvoru je jednaka prethodno navedenoj izvedbi stablene strukture preko dinamičkih polja, jer se članu može pristupiti jedino slijedno. Grafički prikaz ove izvedbe (infiksni oblik) ukazuje na sličnost ove izvedbe i osnovnog modela stablene strukture, što programerima uvelike olakšava interpretaciju stablene strukture. Genotip *TreePtr* iz poglavlja 5 izveden je preko ovog modela uz jednu preinaku da je polje *pokazivačiNaDjecu* izvedeno kao dinamičko polje što povećava memorijsku iskoristivost.

1.3. Usporedba učinkovitosti prikaza stabla u memoriji

Kao što se pokazalo, različite izvedbe sa sobom povlače neke mane i neke prednosti. Na programeru je da procijeni zahtjeve svoje aplikacije (memorijske, procesorske, programerske) i odluči koju će od izvedbi stablene strukture iskoristiti.

Tablica 1.1 Usporedba učinkovitosti izvedbe stablene strukture

	Iskoristivost memorije	**Dodatni trošak memorije**	**Vremenska zahtjevnost modifikacije**	**Vremenska složenost traženja čvora**
Statičko polje	U prosjeku nepotpuna. Ako stablo nije potpuno postoji alocirana neiskorištena memorija (Sl. 1.2).	Ne postoji. Struktura čvora sadrži samo svojstvo čvora (*podatak*).	Velika. Zahtjeva skidanje elemenata (čvorova) iz polja i stavljanje.	O(1). Djeca *i*-tog čvora u binarnom stablu s *n*-čvorova se nalaze u polju na indeksima 2**i* i 2**i* + 1.

	Iskoristivost memorije	**Dodatni trošak memorije**	**Vremenska zahtjevnost modifikacije**	**Vremenska složenost traženja čvora**
Dinamičko polje	Potpuna. Čvorovi se u polje dodaju slijedno i polje se proširuje (Sl. 1.3).	Malen. Struktura čvora sadrži član *brojDjece*.	Velika. Zahtjeva skidanje elemenata (čvorova) iz polja i stavljanje.	O(n) (*n* je veličina stabla). Slijedno se pretražuju čvorovi.
Struktura čvora s pokazivačima na djecu	Potpuna. Svaki alocirani čvor predstavlja jedan čvor iz osnovnog modela stabla (Sl. 1.1)	Velik. Uz podatkovni član *brojDjece* sadrži i polje pokazivača na čvorove djecu (*pokazivaciNaDjecu*).	Mala. Pokazivači na djecu se stavljaju u polje *pokazivaciNaDjecu* čvora roditelja.	O(n) (*n* je veličina stabla). Slijedno se pretražuju čvorovi.

Zahtjevi na stablene genotipe obrađene u ovom radu eliminirali su prvu izvedbu preko statičkog polja zbog promjenljivosti veličine stabla i stupnja pojedinih čvorova unutar genotipa. Razlika (Tablica 1.1) ostalih dviju izvedbi, dinamičkog polja i strukture čvora s pokazivačima na djecu, je motiv za izradu ta dva genotipa obrađenih u ovom radu i testiranja istih.

2. Genetsko programiranje

U umjetnoj inteligenciji, genetsko programiranje (GP) je metoda bazirana na evolucijskom algoritmu inspirirana biološkom evolucijom za pronalazak računalnih programa koji izvode definirani zadatak. GP je specijalizacija genetskih algoritama (GA) u kojoj je svaka jedinka računalni program. Shodno tome, GP predstavlja tehniku strojnog učenja korištenu za optimiranje populacije računalnih programa prema *prikladnost* (eng. *fitness*) jedinke koja je mjera sposobnosti jedinke da izvodi zadani zadatak.

Razlika GP-a u odnosu na GA-a je u tome što kromosom, odnosno genotip, predstavlja računalni *program* koji je rješenje zadanog problema. Tipična struktura podataka koja se kod genetskog programiranja koristi za prikaz genotipa jest stablo.

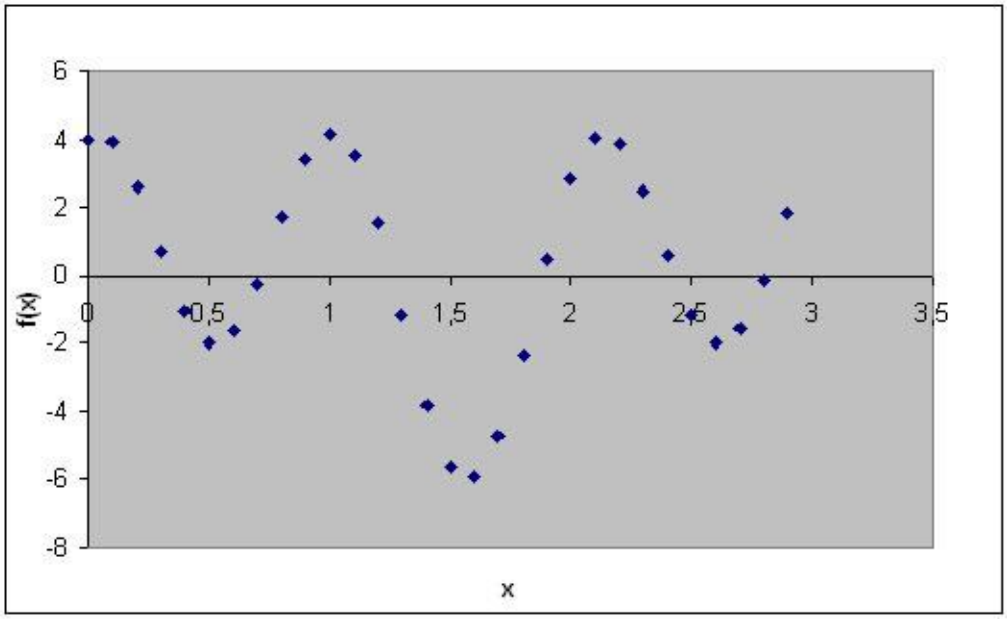
Pogledajmo to na jednostavnom primjeru. Mjerenjem izlaza nekog procesa snimljena je njegova ulazno-izlazna karakteristika.

Tablica 2.1 Ulazno-izlazna karakteristika promatranog sustava

x	**f(x)**	**X**	**f(x)**	**x**	**f(x)**
0	4	1	4,122921	2	2,816585
0,1	3,892383	1,1	3,485439	2,1	4,031366
0,2	2,578716	1,2	1,548364	2,2	3,846619
0,3	0,657845	1,3	-1,15971	2,3	2,480139
0,4	-1,0855	1,4	-3,82031	2,4	0,548066
0,5	-1,96498	1,5	-5,59958	2,5	-1,16275
0,6	-1,63934	1,6	-5,93108	2,6	-1,97962
0,7	-0,23462	1,7	-4,70869	2,7	-1,58571
0,8	1,700922	1,8	-2,32285	2,8	-0,13352

0,9	3,393531	1,9	0,472592	2,9	1,809713

Zanima nas algebarski izraz koji najbolje opisuje tabeliranu funkciju (grafički prikaz dan je na Sl. 2.1).

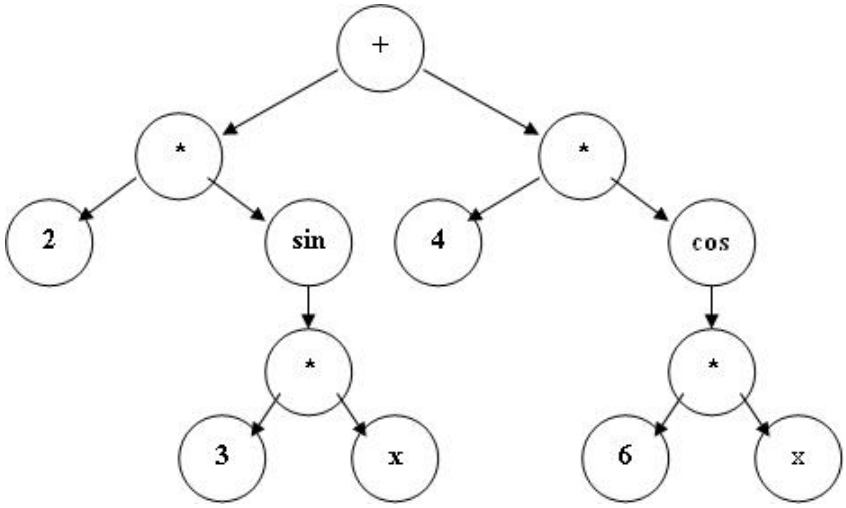


Sl. 2.1Ulazno-izlazna karakteristika sustava

Zadatak genetskog programiranja jest pronaći funkciju, odnosno eksplicitni izraz koji opisuje dani uzorak na danoj domeni. Sjetimo se da se svaka funkcija može prikazati operatorskim stablom. Primjerice, funkciju:

$$f(x) = 2 * \sin(3 * x) + 4 * \cos(6 * x)$$

je moguće prikazati sljedećim stablom:



Sl. 2.2 Prikaz genotipa kod genetskog programiranja

Genotipovi kod genetskog programiranja upravo su operatorska stabla, čiji čvorovi mogu biti, primjerice, aritmetičke operacije (+, -, *, /), ugrađene funkcije (sin, cos), konstante (bilo koji cijeli ili decimalni broj) te varijable. Naziv za skup operatora u stablu jest *primitivi*, s time da su funkcijski operatori *funkcijski primitivi*, a varijable i konstante su *terminalni primitivi* ili *terminali*. Što se više vrsta čvorova dozvoli, veća je šansa pronalaska prikladog stabla koje odgovara zadanoj funkciji, ali raste i vrijeme potrebno za njegov pronalazak.

2.1. Evaluacija *prikladnosti* jedinke

Evaluacija genotipa u našem slučaju može se izvesti kao ukupna suma razlike izlaza funkcije u promatranoj točki i izmjerenog podatka (ili možemo računati srednje kvadratno odstupanje). Navedeni problem je upravo jedan od problema nad kojima ćemo mjeriti performanse genotipa i zove se *simbolička regresija*.

Problem *centriranja vozila* je također problem domene genetskog programiranja. Problem se sastoji od vozila koje se može kretati lijevo i desno ograničenom brzinom i ubrzanjem s ciljem zaustavljanja u zadanoj poziciji. Vozilo je opisano fizikalnim veličinama *masom* i *maksimalnom silom* koju mogu proizvesti motori vozila (lijevi i desni). Ulazni parametri za ovaj problem jesu *početna brzina vozila, početna pozicija vozila* te *odredišna pozicija vozila* gdje se vozilo naposljetku treba zaustavit. Zadatak genetskog programiranja jest razviti funkciju brzine po prostoru koja bi upravljala motorima s ciljem što bržeg zaustavljanja vozila u odredišnoj poziciji. Vrijeme potrebno

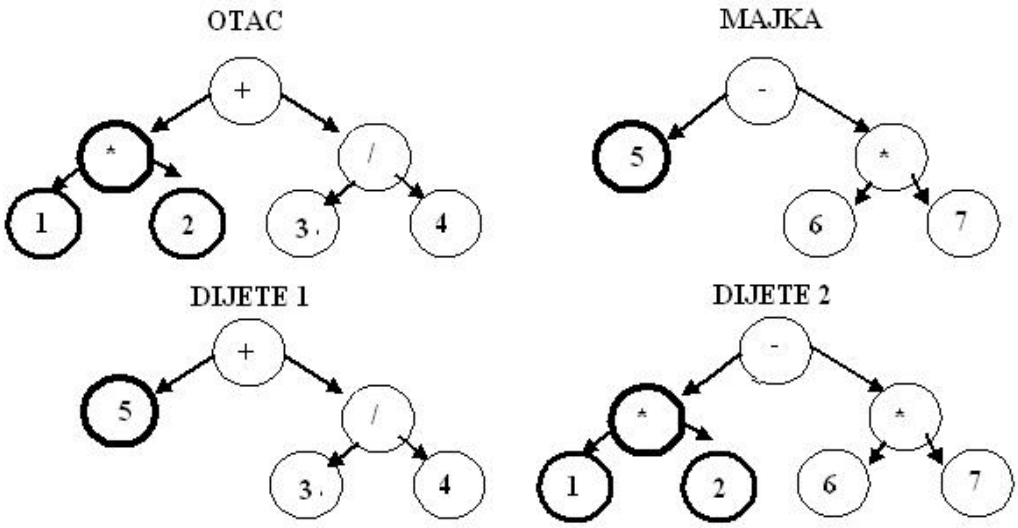
za zaustavljanje vozila u toj poziciji predstavlja prikladnost jedinke. Prikladnost se računa se kao:

$$Tzaustavljanja = \sum_{i=0}^{n-1} (X[i] - X[i-1]) / ((V[i] + V[i-1]) / 2)$$

gdje *n* predstavlja broj uzoraka dobivene funkcije (brzina/prostor) nad kojima se evaluira jedinka (funkcija).

2.2. Križanje

Križanje dvaju roditelja tipično se radi tako da se neko podstablo jednog roditelja zamijeni nekim podstablom drugog roditelja. Tim postupkom se dobiva dvoje djece i ovisno o izvedbi križanja i samog algoritma može se uzeti jedno[3] dijete ili oboje.



Sl. 2.3 Jednostavno križanje

Na slici (Sl. 2.3) je prikazano jednostavno križanje. Podebljani čvorovi predstavljaju podstabla određena za kopiranje u novu jedinku. Algoritam selekcije je odabrao dvije jedinke/stabla/programa za proces križanja.

$$OTAC = (1 * 2) + (3 / 4)$$

$$MAJKA = 5 - (6 * 7)$$

[3] *Evolutionary Computation Framework* (ECF) koristi postupak križanja u kojem je ishod križanja jedno dijete.

Ishod križanja su dvije jedinke, dvoje djece.

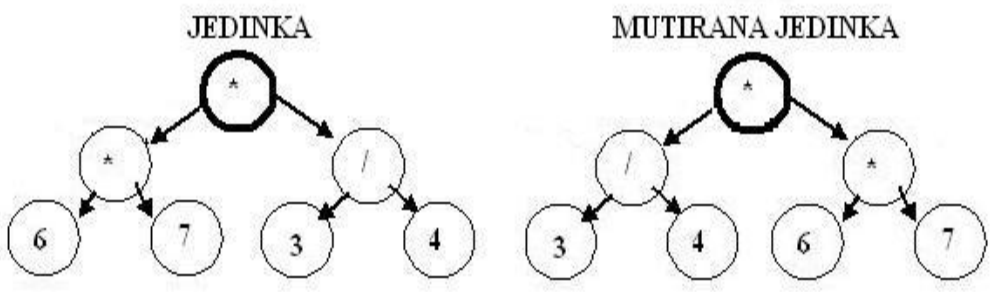
$$DIJETE1 = 5 + (3/4)$$

$$DIJETE2 = (1*2) - (6*7)$$

Kao što je napomenuto, ovisno o implementaciji križanja. postupak križanja može kao rezultat vratiti obje novostvorene jedinke (*DIJETE1*, *DIJETE2*) ili samo jednu.

2.3. Mutacija

Operator mutacije može se izvesti na više načina – primjerice, slučajnom zamjenom čvora, brisanjem podstabla i zamjenom s novim slučajno stvorenim podstablom, zamjenom redoslijeda djece (*permutacija*[4]) i sl.



Sl. 2.4 Permutacija čvora

Na slici (Sl. 2.4) prikazan je proces mutacije, zvan permutacija čvora. Slučajno se odabere čvor (u našem slučaju to je korijen) i zamjeni se redoslijed djece. Redoslijed se može izmijeniti na *n!* (faktorijela) načina, *n* predstavlja broj djece odabranog čvora. U našem slučaju *n* = 2, tako da postoji dva moguća redoslijeda djece. Jedinka odabrana za mutaciju je:

$$JEDINKA = (6*7)*(3/4)$$

Jedinka dobivena permutacijom čvora:

$$MUTIRANA\ JEDINKA = (3/4)*(6*7)$$

[4] Permutacija je jedan oblik mutacije korišten u stablenim genotipovima ECF-a.

Operatori križanja i mutacije te evaluacija genotipa su intenzivno korišteni u evolucijskom algoritmu (u svakoj generaciji), stoga je njihova učinkovita izvedba presudna za optimalan rad algoritma. Operatori križanja i mutacije modificiraju stablenu strukturu te su kao takvi vremenski zahtjevni. U poglavlju 1.3 uspoređene su izvedbe stablenih struktura korištene za izvedbu stablenih genotipova. U tablici Tablica 1.1 se vidi kako je vremenska zahtjevnost modifikacije stabla izvedbe stablene strukture preko strukture čvora s pokazivačima na djecu manja od vremenske zahtjevnosti kod izvedbe stabla preko dinamičkog polja.

3. Opis rada "*Evolutionary Computation Framework"-a*

U ovom je poglavlju ukratko objašnjen rad *Evolutionary Computation Framework-a*[5] (ECF). ECF je okruženje za evolucijsko računanje napisano u programskom jeziku C++ tako da ga je moguće pokrenuti u bilo kojoj razvojnoj okolini koja sadrži C++ prevodilac (npr. Microsoft Visual Studio, Eclipse, Dev-C++). Zadnja razvojna verzija se može besplatno skinuti sa *Subversion* repozitorija[6], a prije pokretanja potrebno je sa interneta skinuti i instalirati biblioteku[7] koja služi za rukovanje sa pametnim pokazivačima te je uključiti kao parametar kod prevođenja koda.

Za mnoge postupke evolucijskog računanja potrebno je razvijati posebna programska rješenja, što se pokazuje izuzetno neučinkovitim te oduzima programerima i ostalim zainteresiranim stranama previše vremena.

Izrada programskog okruženja za postupke evolucijskog računanja rješava taj nedostatak na način da predstavlja neku vrstu "univerzalnog" alata koji pruža izrazitu fleksibilnost i mogućnost konstantne implementacije novih postupaka te korisnicima omogućava puno manje kontakta s izvornim kodom i jednostavniji način definiranja problema kojeg će to programsko okruženje obraditi (npr. unos pretpostavljenih vrijednosti u konfiguracijsku datoteku).

3.1. Konfiguracijska datoteka

ECF u fazi inicijalizacije čita konfiguracijsku datoteku u kojoj su definirani korišteni algoritmi te genotipovi. Za zapis konfiguracije korišten je XML (eng. *Extensible Markup Language*).

---

[5] http://gp.zemris.fer.hr/ecf/

[6] svn://smaug.zemris.fer.hr/ecf/trunk URL na kojem se nalazi ECF biblioteka.

[7] Naziv biblioteke je Boost i može se skinuti sa http://www.boost.org/users/download/

Konfiguracijska datoteka sadrži sljedeće čvorove:

- čvor *Algorithm* definira parametre algoritma,
- u čvoru *Genotype* navode se korišteni genotipovi zajedno sa svojim parametrima,
- čvor *Registry* sadrži opće parametre koji nisu ovisni o pojedinom algoritmu ili genotipu, kao što su veličina populacije, vjerojatnost mutacije, broj generacija i sl..

Parametri se definiraju u obliku:

```
<Entry key="parametar">vrijednost</Entry>
```

Ukoliko neki od parametra nije definiran, uzimaju se podrazumijevane vrijednosti. Konfiguracijska datoteka može izgledati ovako:

```
<ECF>
 <Algorithm>
  <SteadyStateTournament>
   <Entry key="tsize">3</Entry>
  </SteadyStateTournament>
 </Algorithm>

 <Genotype>
  <BitString>
   <Entry key="size">20</Entry>
  </BitString>
  <Tree>
   <Entry key="functionset">sin cos + - / *</Entry>
   <Entry key="terminalset">X Y</Entry>
  </Tree>
 </Genotype>

 <Registry>
  <Entry key="population.size">30</Entry>
  <Entry key="term.maxgen">100</Entry>
 </Registry>
</ECF>
```

Konfiguracijska datoteka definira jedan algoritam selekcije *Steady State Tournament*[8] sa odgovarajućim parametrom *tsize* koji predstavlja broj jedinki koji ulazi u algoritam selekcije. Dalje se definiraju genotipovi. Vidi se da se koriste dva genotipa *BitString* i *Tree* (jedan od genotipa opisan u ovome radu). Parametar *BitString* genotipa jest *size* i

[8] Više o *Steady State Tournament* selekciji na http://en.wikipedia.org/wiki/Tournament_selection .

predstavlja veličinu genotipa (polja). *functionset* i *terminalset* su parametri *Tree* genotipa te predstavljaju skup funkcijskih primitiva i terminala (respektivno) nad kojima će se izgraditi inicijalna populacija.

3.2. Program

Jedino što korisnik ECF-a mora sam isprogramirati jest *evaluacijski* operator koji utjelovljuje *fitness* funkciju nad definiranim genotipovima. Jednom kad je operator evaluacije napisan, treba ga *referencirati* u *main* funkciji i pokrenuti ECF. Tipična *main* funkcija bi dakle izgledala ovako:

```
int main(int argc, char **argv)
{
    StateP state = static_cast<StateP> (new State);
    state->setEvalOp(static_cast<EvaluateOpP> (new
MyEvalOperator));

    state->initialize(argc, argv);
    state->run();
    return 0;
}
```

Kada je korisnik isprogramirao svoj evaluacijski operator i utjelovio ga u *EvaluateOp* klasu, treba ga instancirati i predati referencu evaluacijskog objekta *state* objektu preko metode *State::setEvalOp(EvaluateOp).* Naposljetku treba pokrenuti ECF preko metode *State::run(void)*, koja pokreće iščitavanje konfiguracijske datoteke te obavlja zadani algoritam sa definiranim genotipovima i genetskim operatorima. Nad S*tate::run()* metodom je vršeno mjerenje.

4. Genotip „*Tree*“

Stableni genotip *Tree* je ostvaren stablenom strukturom preko dinamičkog polja. Karakteristike takve izvedbe navedene su u Tablica 1.1. Slijedi da od ovog genotipa očekujemo potpunu iskoristivost memorije, malen dodatni trošak na memoriju, veliku vremensku zahtjevnost modifikacije stabla i O(n) vremensku složenost pristupa čvoru.

4.1. Čvorovi i primitivi

Logičan sljedeći korak je pokazati strukturu čvora kao građevnu jedinicu stablenog genotipa.

```
class Node
{
public:
 Node(void);
 Node( PrimitiveP primitive_);
 unsigned int size;
 unsigned int depth;
 void printPrimitive();
 void setPrimitive( PrimitiveP primitive_ );
 ~Node(void);
 PrimitiveP primitive;
};

typedef boost::shared_ptr<Node> NodeP;
```

Struktura čvora je enkapsulirana unutar klase *Node. Node::size* i *Node::depth* podatkovni članovi su cjelobrojni članovi i predstavljaju veličinu i dubinu čvora respektivno (kao što je definirano u 1.1). Ova dva člana predstavljaju određeni implementacijski memorijski trošak koji će uvelike smanjiti vremensku zahtjevnost modificiranja stabla. Podatak, odnosno svojstvo koje nosi čvor jest primitiv i nalazi se u podatkovnom članu *Node::primitive* koji je tipa *PrimitiveP*[9].

[9] *PrimitiveP* je *smart pointer* na objekt klase *Primitive* definiran kao *boost::shared_ptr<Primitive>*. Općenito *KlasaP* je definiran u ECF-u kao *boost::shared_ptr<Klasa>*.

Primitivi su enkapsulirani u klasu *Primitive*.

```
class Primitive
{
public:
 Primitive(void);
 virtual void execute(void* result, Tree& tree) = 0;
 virtual void setValue(void* value);
 void getNextArgument(void* result, Tree& tree);
 int getNumberOfArguments();
 void setDesignation(std::string designation);
 std::string getDesignation();
 virtual ~Primitive(void);
protected:
 std::string designation_;
 int nArguments_;
};
typedef boost::shared_ptr<Primitive> PrimitiveP;
```

Ova klasa je apstraktna klasa koju nasljeđuju konkretne klase koje predstavljaju funkcijske primitive poput klasa *Sin, Cos, Mul, Div* itd., i terminale poput klase *Terminal*.

Relevantne članove za ovaj rad predstavljaju zaštićeni podatkovni član *int Primitive::nArguments_* i virtualni funkcijski član *void Primitive::execute(void*, Tree&)*. *Primitive::nArguments_* član je nužni trošak memorije iz zahtjeva na čvorove stabla da imaju različite stupnjeve i predstavlja broj djece čvora koji zadrži taj primitiv. Navedeni funkcijski član *Primitive::execute()* predstavlja usko grlo u procesu evaluacije jedinke jer se izvršava svake generacije, za broj jedinki koje odredi algoritam (obično sve) te za svaki čvor jedinke. Ta metoda poziva tu istu metodu za svako od svoje djece te rezultate zbraja, oduzima, množi ili što već taj primitiv radi. Vidimo da rekurzivno *preorder*[10] obilazimo sve čvorove stabla što je vremenski veoma zahtjevno zbog same procedure[11] koje operacijski sustav izvodi pri pozivu funkcije. Jedan veliki vremenski trošak pri pozivu ove metode je uzrokovan mehanizmima objektno orijentirane paradigme jezika C++, a to su *virtualne*[12] funkcije. Naime, pri pozivu *execute()* metode dolazi do

[10] *Preorder* način rekurzivnog obilaska stabla – roditelj->dijete1->dijete2...->dijeteN.

[11] Procedura poziva funkcije podrazumijeva dohvaćanje argumenata iz memorije, stavljanje istih na stog skupa s povratnom adresom te naposljetku poziv potprograma.

[12] Virtualna funkcija definirana u baznom razredu se može nadjačati s istom funkcijom u izvedenom razredu. Svaki objekt nekog razreda implicitno sadrži pokazivač na pripadajuću *v-tablicu* u kojoj se nalaze

pretraživanja *v-tablice* s ciljem nalaženja odgovarajuće funkcije koju treba pozvati. Ta pojava se zove *dinamičko povezivanje* (engl. *dynamic-binding*) i ovisno o veličini *v-tablice*, odnosno broju virtualnih funkcija tog razreda, može predstavljati veliki vremenski trošak.

U našem slučaju sa dvije virtualne funkcije, vremenski trošak uzrokovan dinamičkim povezivanjem bi trebao biti minimalan.

4.2. Stablo

Nakon definiranja građevne jedinice, čvora, preostalo je definirati i samu stablenu strukturu. Ovaj stableni genotip je izveden preko dinamičkog polja, odnosno polja čvorova koji su slijedno poslagani u prefiksnom obliku.

```
class Tree : public std::vector<NodeP>, public Genotype
{
public:
 /*Javne metode*/
 Tree();
 Tree(int min, int max);
 void growBuild(PrimitiveSetP primitiveSet);
 void fullBuild(PrimitiveSetP primitiveSet);
 void update();
 void calculate(void*);
 void addNode(Node* node);
 void addNode(NodeP node);
 void setTerminalValue(std::string, void*);
 void printTreeFunction();
 ~Tree();
 /*ECF: nasljeđene metode klase Genotype*/
 Tree* copy();
 void write(XMLNode&);
 void read(XMLNode&);
 void registerParameters(StateP);
 bool initialize (StateP state);
 std::vector<MutationOpP> getMutationOp();
 std::vector<CrossoverOpP> getCrossoverOp();

 /*Javni podatkovni članovi*/
 PrimitiveSetP primitiveSet_;
 uint startDepth;
 uint maxDepth;
 uint minDepth;
 uint iNode;
private:
 /*Privatne metode*/
 uint growBuild_(PrimitiveSetP primitiveSet, int myDepth);
```

pokazivači na sve virtualne funkcije tog razreda/objekta. Vidi http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming .

```
  uint fullBuild_(PrimitiveSetP primitiveSet, int myDepth);
  uint getSize_();
  void setDepth_(int myDepth);
};
typedef boost::shared_ptr<Tree> TreeP;
```

Stablena struktura jest enkapsulirana unutar klase *Tree* koja nasljeđuje dinamičko polje čvorova *vector*[13]<*NodeP*> i razred *Genotype*[14].

Relevantni članovi za ovaj rad su funkcijski članovi *void Tree::fullBuild(PrimitiveSetP)* , *void Tree::update() i void Tree::calculate(void*)* te podatkovni članovi *Tree::primitiveSet_* tipa *PrimitiveSetP*, cjelobrojni članovi *Tree::startDepth*, *Tree::maxDepth* i *Tree::minDepth*. Član *Tree::primitiveSet* predstavlja skup primitiva nad kojim će se izgraditi jedinka, metodom *Tree::fullBuild()* ili *Tree::growBuild()*. *Tree::startDepth* predstavlja početnu dubinu stabla, odnosno dubinu korijenskog čvora. *Tree::minDepth* je minimalna dubina (relativna početnoj) stabla koje se može pojaviti u populaciji. Nasuprot tome, *Tree::maxDepth* jest maksimalna dubina stabla. Metoda *Tree::fullBuild()* nije važna za učinkovit rad algoritma jer se poziva pri inicijalizaciji populacije (znači jednom na početku algoritma), ali ova metoda je korištena za izgradnju *potpunih*[15] stabala nad kojima su vršena mjerenja pa je potrebno znati funkcionalnost te metode. *Tree::update()* metoda je korištena u operatoru križanja gdje osvježava dubinu i veličinu svih čvorova nastale jedinke. Križanje je konstantno prisutan proces unutar genetskog algoritma, stoga je važno napomenuti da ova metoda nosi određeni vremenski trošak (rekurzivno prolazi kroz sve čvorove jedinke).

4.3. Evaluacija

Tree::calculate() je važna metoda, jer se ona poziva pri evaluaciji jedinke. Ova metoda je zapravo jednostavna jer samo poziva *Primitive::execute()* metodu primitiva korijenskog čvora. Ova metoda predstavlja glavnu stavku ispitivanja i kao takva glavnu

[13] *vector<Class>* tip podatka iz *Standard Template Library* C++ biblioteke. Predstavlja dinamičko polje.

[14] Apstraktna klasa *Genotype* ECF biblioteke koja utjelovljuje genotip. Nasljeđuju je konkretni genotipovi (*Tree, TreePtr, Binary, BitString, PermutationVector...*).

[15] Potpuno stablo je stablo u kojemu su dubine svih listova jednake zadanoj maksimalnoj dubini stabla.

stavku ovog rada, uz metodu *TreePtr::execute()* (poglavlje 5.3), nad kojom će se vršiti mjerenja.

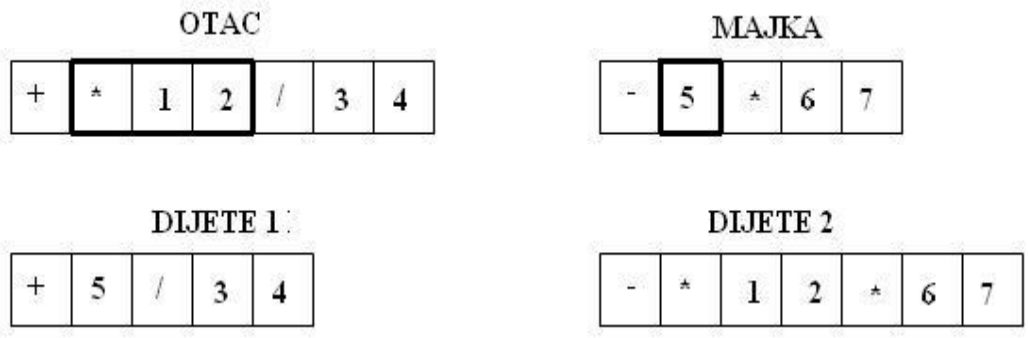
4.4. Križanje

Križanje[16] jest proces koji zahtjeva modifikaciju stablene strukture (novonastale jedinke), te uzimajući u obzir izvedbu ovog stablenog genotipa preko dinamičkog polja, taj proces je vremenski vrlo zahtjevan. Dodajući činjenicu da je križanje konstantno prisutan proces u genetskom algoritmu, izvedba križanja može predstavljati važnu stavku u optimizaciji cjelokupnog algoritma.

Algoritam križanja je identičan jednostavnom križanju opisanome u poglavlju 22.2 uz napomenu da je rezultat križanja *jedna* nova jedinka. Metoda[17] koja ostvaruje križanje izgleda:

```
/*križaj dva genotipa*/
bool mate(GenotypeP gen1, GenotypeP gen2, GenotypeP child);
```

Memorijski zapis stabala sa slike Sl. 2.3 preko dinamičkih polja prikazan je na slici (Sl. 4.1).



Sl. 4.1 Jednostavno križanje preko dinamičkih polja

[16] Operatori križanja u ECF-u su enkapsulirani u izvedene klase bazne klase *OperatorCrxOp.*

[17] Ova metoda je čista virtualna funkcija baznog razreda *CrossoverOp* i kao takva mora biti nadjačana u konkretnim izvedenim razredima koji predstavljaju operatore križanja za pojedini genotip.

Kao što je spomenuto prije, može se primijetiti prefiksni zapis stabla u polju. Podebljana polja su podstabla određena za zamjenu. Kako bi se dobila jedinka *DIJETE 1* postupak križanja bi podrazumijevao:

1. Odabir slučajnog čvora jedinke *OTAC,*
2. Odabir slučajnog čvora jedinke *MAJKA,*
3. Kopiranje čvorova sa početka polja jedinke *OTAC* u jedinku *DIJETE 1* dok se ne dosegne odabrani čvor,
4. Kopiranje podstabla jedinke *MAJKA* u jedinku *DIJETE 1,*
5. Zaobilaženje podstabla kojem je odabrani čvor korijen te kopiranje ostatka stabla jedinke *OTAC* u jedinku *DIJETE 1.*

Uzimajući u obzir da kopiranje podstabla podrazumijeva rekurzivno putovanje po stablu s ciljem nalaženja indeksa polja koji predstavljaju početak i kraj podstabla, vremenska zahtjevnost ovog postupka je velika. Kao što je napomenuto prije, za *DIJETE 1* se poziva metoda *Tree::update()* koja rekurzivno prolazi kroz sve čvorove stabla i osvježava dubine i veličine pojedinih čvorova. Shodno tome, cijeli postupak križanja vremenski je vrlo zahtjevan.

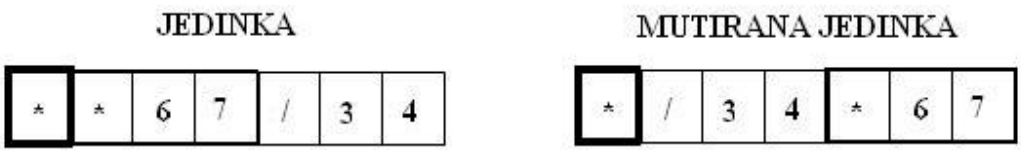
4.5. Mutacija

Postupak mutiranja jedinke također zahtjeva modificiranje postojeće stablene strukture jedinke odabrane za mutaciju. Metoda[18] koja obavlja postupak mutacije jest:

```
/*Mutiraj genotip*/
bool mutate(GenotypeP);
```

Permutacija, kao oblik mutacije ovog genotipa, zahtjeva zamjenu redoslijeda podstabla slučajno odabranog čvora kao što je prikazano na slici Sl. 2.4.

[18] Ova metoda je čista virtualna funkcija baznog razreda *MutationOp* i kao takva mora biti nadjačana u konkretnim izvedenim razredima koji predstavljaju operatore mutacije za pojedini genotip.



Sl. 4.2 Mutacija preko dinamičkih polja

Na slici (Sl. 4.2) su prikazane stablene strukture sa slike Sl. 2.4 u dinamičkom polju. Za permutaciju je odabran korijen (pravokutnik s najdebljim crtama) stabla. Skup čvorova unutar pravokutnika sa malo podebljanim crtama predstavljaju lijevo podstablo odabranog čvora, a skup čvorova u pravokutniku sa nepodebljanim crtama desno podstablo. Postupak koji bi od *JEDINKE* napravio *MUTIRANU JEDINKU* je sljedeći:

1. Odabrati slučajni čvor *JEDINKE* (u našem slučaju odabran je čvor),
2. Odrediti indekse početka i kraja lijevog i desnog podstabla odabranog čvora,
3. Zamijeniti podstabla.

Korak 2. u ovom postupku predstavlja najveći trošak vremena jer rekurzivno obilazi stablo ne bi li našao odgovarajuće indekse.

4.6. Pregled genotipa

Naposljetku za ovaj genotip se može pretpostaviti kakve će performanse imati. Od evaluacije koja svoj vremenski trošak iskazuje kroz rekurzivno obilaženje stabla i izvođenje funkcija koje zahtijevaju minimalnu pretragu *v-tablice* do postupaka križanja i mutiranja koji su vremenski zahtjevni jer modificiraju stablenu strukturu. Mora se napomenuti da se evaluacija izvodi puno više puta od križanja i mutacije. Kako izgleda karakteristika učinkovitosti genotipa ovisi o omjeru poziva tih postupaka unutar genetskog procesa. Shodno tome definitivni rezultati su dobiveni tek ispitivanjem nad stvarnim problemima (vidi poglavlje 6).

5. Genotip „*TreePtr*“

Ovaj genotip je također stableni genotip izveden preko strukture čvora s pokazivačima na djecu. Shodno tome uporabom ove izvedbe stablene strukture za genotip dobiva se potpuna iskoristivost memorije, velik trošak na memoriju uzrokovan pokazivačima na djecu, malu vremensku zahtjevnost modificiranja stabla te O(n) vremensku složenost pretraživanja stabla.

5.1. Čvor i primitivi

Uobičajeno se prvo definira struktura čvora kao građevne jedinice stabla. Čvor je enkapsuliran u klasu *NodePtr*.

```
/*Definicija pokazivača na funkciju*/
typedef double (*PrimitiveFuncPtr)(NodePtrP);
/*Klasa NodePtr*/
class NodePtr
 {
 public:
       NodePtr(void);
       /*Oznaka Primitive*/
       std::string primitiveDesignation;
       /*Broj Podstabla Čvora*/
       int primitiveNumArgs;
       /*Primitiv u obliku pokazivača na funkciju*/
       PrimitiveFuncPtr execPrimitive;
       /*Dinamičko polje pokazivala na djecu*/
       std::vector<NodePtrP> vChildren;
       /*Pokazivač Na Trenutnu Vrijednost Terminala*/
       double *ptrTermValue;
       /*Dubina Čvora*/
       int depth;
       /*Veličina Čvora */
       int size;
       ~NodePtr(void);
 };
typedef boost::shared_ptr<NodePtr> NodePtrP;
```

Uzrok dodatnog troška memorije (idenitificirani u poglavlju 1.3) predstavljaju podatkovni članovi *int NodePtr::primitiveNumArgs* (broj podstabla čvora) i dinamičko polje pokazivača na djecu *std::vector<NodePtrP> NodePtr::vChildren*.

Primitiv je, kao svojstvo čvora, sadržan u obliku pokazivača[19] na funkciju *NodePtr::execPrimitive.*

Pokazivač na funkcije je programska tvorevina jezika C/C++ i baš kao obični pokazivači na podatke u sebi sadrže memorijsku adresu. Razlika između običnih pokazivača i pokazivača na funkcije jest u tome što obični u sebi sadrže memorijsku adresu podatka, dok pokazivač na funkcije sadrži memorijsku adresu potprograma koji se treba pozvati. Jezik C++ je uveo dva nova tipa pokazivača na funkcije, *pokazivač na statičke funkcijske članove razreda* i *pokazivač na nestatičke funkcijske članove razreda* Kao posljedica promijenila se struktura pokazivača na funkcije na način da zauzima tri do četiri puta više mjesta od običnog pokazivača na podatke. Uzrok toj promjeni jest potreba za pohranu dodatnih informacija o funkciji na koju pokazuje taj pokazivač. Poziv statičkih funkcijskih članova i običnih funkcija se razlikuje od poziva nestatičkih funkcijskih članova, stoga podatkovni tip pokazivača na funkciju mora biti dovoljno velik da spremi informaciju o vrsti funkcije na koju pokazuje, broj argumenata te funkcije i njihove memorijske adrese.

PrimitiveFuncPtr jest tip podatka koji predstavlja pokazivač na običnu funkciju (ne funkcijski član razreda) koja ima povratni tip *double* i prima jedan argument tipa *NodePtrP* što predstavlja pokazivač na čvor koji sadrži taj primitiv. Pošto je primitiv izveden kao funkcijska apstrakcija, svi dodatni podaci primitiva (oznaka, broj argumenata) su spremljeni u čvor koji sadrži taj primitiv.

Razlog korištenja pokazivača na funkcije je izbjegavanje neželjenih nuspojava mehanizama objektne paradigme u C++-u. Uzimajući u obzir genotip *„Tree“* (poglavlje 4), problem virtualnih funkcija i pretraživanja *v-tablice* se zaobišao uporabom funkcijskih pokazivača.

Dakle podatkovni član *NodePtr::execPrimitive* se koristi za poziv funkcije odgovarajućeg primitiva. Nadalje funkcija primitiva poziva, preko *NodePtr::execPrimitive* člana djeteta, funkcije primitiva svoje djece i rezultate obrađuje u skladu sa svojom funkcionalnošću. Ti pozivi rekurzivno *inorder*[20] obilaze stablo i kao takvi su vremenski vrlo zahtjevni.

---

[19] U programskim jezicima C# i Visual Basic .NET pokazivači na funkcije su izvedeni kao *delegati.*

[20] Inorder način obilaska stabla *lijevo dijete1 -> lijevo dijeteN ->roditej->desno dijete1->desno dijeteN.*

5.2. Stablo

Stablo ovog genotipa, za razliku od genotipa *„Tree“* (poglavlje 4) koji sadrži sve čvorove, sadrži jedino pokazivač na korijenski čvor.

```
 class TreePtr : public Genotype
 {
 public:
       TreePtr(void);
       TreePtr(int depthStart, int depthMin, int depthMax);
       /*Grow i Full Metode Izgradnje Stabla*/
       void buildGrow();
       void buildFull();
       /*Rukovanje State Objektom*/
       void setState(StateP state);
       StateP getState();
       /*Rukovanje Skupom Primitiva*/
       void setPrimitives(PrimitiveSetPtrP primitives);
       PrimitiveSetPtrP getPrimitives();
       /*Postavljanje Vrijednosti Terminala*/
       void setTerminalValue(std::string designation ,double
value);
       /*Metoda Za Evaluaciju Stabla*/
       double execute();
       /*Iscrtavanje Stabla*/
       void print();
       /*Metoda Za Dobavljanje Čvora Indeksa index*/
       NodePtrP getNode(int index);
       /*Rukovanje Korijenom*/
       void setRoot(NodePtrP root);
       NodePtrP getRoot();
       /*Metoda Za Osvježavanje Dubina i Veličina Čvorova U
Stablu*/
       void update();
       /*Dubine*/
       int depthStart;
       int depthMin;
       int depthMax;
       /*Veličina Stabla*/
       int size;
       ~TreePtr(void);
       /*ECF: Nasljeđene Metode Klase Genotype*/
       bool initialize(StateP state);
       TreePtr* copy();
       std::vector<CrossoverOpP> getCrossoverOp();
       std::vector<MutationOpP> getMutationOp();
 private:
       /*State*/
       StateP state_;
       /*Primitive Set*/
       PrimitiveSetPtrP primitives_;
       int buildFull_(NodePtrP node, int depthCurrent);
       int buildGrow_(NodePtrP node, int depthCurrent);
       std::string print_(NodePtrP node);
       /*Root Node*/
       NodePtrP nodeRoot_;
 };
```

```
typedef boost::shared_ptr<TreePtr> TreePtrP;
```

Kao i u *„Tree“* genotipu (4) i ovdje član koji predstavlja skup primitiva (*TreePtr::primitives_*) nad kojim se metodama izgradnje (*TreePtr::buildFull i TreePtr::buildGrow*) gradi stablena jedinka. Cjelobrojni članovi *TreePtr::depthStart, TreePtr::depthMin i TreePtr::depthMax* predstavljaju dubinu korijena, minimalnu i maksimalnu dubinu čvora koja će se pojaviti unutar stabla (relativno dubini korijena), tim redom.

5.3. Evaluacija

Izračunavanje trenutne vrijednosti stabla, odnosno evaluacija stabla uz postavljenu vrijednost terminalnih primitiva obavlja se pozivom metode *double TreePtr::execute()* koja vraća vrijednost funkcije koje to stablo predstavlja. Ta metoda je jednostavna, ona poziva funkciju primitiva korijenskog čvora preko *NodePtr::execPrimitive* (poglavlje 5.1) pokazivača na funkciju. Ova metoda je intenzivno korištena u genetskom procesu jer se poziva pri evaluaciji jedinke, te je stoga mjerena (poglavlje 6) vremenska zahtjevnost iste i uspoređena sa sličnom metodom *Tree::calculate()* genotipa *„Tree“* opisanog u poglavlju 4.3.

5.4. Križanje

Postupak križanja za ovakvu izvedbu stablenog genotipa je vremenski malo zahtjevan. Razlog tome je sličnost osnovnog modela stabla s ovom izvedbom stablene strukture. Postupak se sastoji od sljedećih koraka (slika Sl. 2.3):

1. Odaberi slučajni čvor jedinke *OTAC*,
2. Odaberi slučajni čvor jedinke *MAJKA*,
3. U jedinku *DIJETE 1* kopiramo sve čvorove jedinke *OTAC* koji ne pripadaju odabranome podstablu,
4. Na mjesto odabranog čvora (*OTAC*) kopiramo odabrano podstablo jedinke *MAJKE*.

Odabir slučajnih čvorova (korak 1. i 2.) se izvodi preko *TreePtr::getNode(int)* metode koja kao argument prima slučajni indeks čvora koji treba dobaviti. Dobavlja se

rekurzivnim obilaskom stablo do trenutka nailaska na čvor sa odgovarajućim indeksom. Kopiranje je također rekurzivan proces. Treba napomenuti da se nakon križanja poziva postupak *TreePtr::update()* jedinke *DIJETE 1* koji rekurzivno prolazi sve čvorove osvježavajući njihovu dubinu i veličinu (vremenski najzahtjevnije u ovom procesu).

5.5. Mutacija

Postupak mutacije na ovu izvedbu stablene strukture nosi zanemariv vremenski trošak s obzirom na postupke evaluacije i križanja. Kako je oblik mutacije ovog genotipa također permutacija, sve što postupak mutacije treba učiniti jest zamijeniti redoslijed pokazivača na djecu unutar dinamičkog polja *TreePtr::vChildren* slučajno odabranog čvora.

5.6. Pregled genotipa

„TreePtr“ genotip (teoretski razmatrano) nosi manju vremensku zahtjevnost od *„Tree“* genotipa (poglavlje 4). Jedino što preostaje je pitanje memorijskih resursa računala na kojem će se koristiti ovaj genotip jer nosi veći memorijski trošak s obzirom na *„Tree“* genotip.

6. Mjerenje performansi genotipa „*Tree*“ i „*TreePtr*“

U ovom poglavlju mjeri se vrijeme izvođenja metoda *Tree::calculate()* i *TreePtr::execute()*. Te metode se pozivaju pri evaluaciji svake jedinke i kao takve su ključne u cjelokupnoj učinkovitosti genetskog algoritma. Također ćemo mjeriti performanse genotipa nad stvarnim problemima (simbolička regresija, centriranje vozila), iz čega se može zaključiti koji je genotip učinkovitiji jer uzima u obzir postupke evaluacije, križanja i mutacije tijekom cijelog procesa evolucije. Rezultati mjerenja su prodiskutirani u ovom poglavlju.

6.1. Mjerenje vremena izvođenja *execute()* i *calculate()*

Mjerenje[21] se vrši nad populacijom od 200 stabala dubine 10. Rezultat mjerenja predstavlja vrijeme potrebno da se cijela populacija evaluira. Stabla se grade potpunom metodom izgradnje (*Tree::fullBuild()* i *TreePtr::buildFull()*) nad skupom funkcijskih primitiva zbrajanje i oduzimanje i jednim terminalnim primitivom. Slijedi da će svako stablo u populaciji biti potpuno binarno stablo sa:

$$2^{\max Depth - startDepth + 1} - 1 = 2^{15-0+1} - 1 = 65535\, čvorova$$

Za svaki genotip zasebno izvršeno je 10 mjerenja. Dobivena prosječna vremena za evaluaciju populacije jesu 69,58 ms za metodu *TreePtr::execute()* te 18,31 ms za metodu *Tree::calculate()*. 3,8 puta manje vremena u prosjeku potrebno je za izvođenje *calculate()* metode od *execute()* metode. Daljnje istraživanje ovog rezultata dalo je odgovor u memorijskom trošku. Naime, pokušaj da se izbjegne pojava *dinamičkog povezivanja* kod „*Tree*“ genotipa uporabom pokazivača na funkcije u „*TreePtr*“

[21] Radi veće preciznosti korištene su *QueryPerformanceFrequency()* i *QueryPerformanceCounter()* funkcije Windows API-ja. Iste su enkapsulirane u klasu *CpreciseTimer* preuzete sa http://www.codeproject.com/KB/recipes/precisetimer.aspx .

genotipu jest urodilo zanemarivim ubrzanjem. Praćenje stanja radne memorije[22] računala tokom testiranja pokazalo je ogromnu razliku između memorijskog troška dvaju genotipa. Naime, 1.4 puta veći memorijski trošak *„TreePtr“* genotipa uzrokovalo je većim brojem promašaja u *priručnoj memoriji* (engl. *cache memory*) pri dohvatu podataka. Promašaji u *priručnoj memoriji* imaju za posljedicu povećanu latenciju uslijed dohvačanja podataka iz radne memorije. Nadalje, pri ispitivanju, povećavanjem veličine populacije i maksimalne dubine stabla, povećani memorijski trošak je uzrokovao intenzivnije straničenje radne memorije na tvrdi disk, što je uzrokovalo velik broj promašaja dohvata podataka iz radne memorije i veliko vrijeme izvođenja.

6.2. Mjerenje izvođenja algoritma

U ovom poglavlju su predstavljeni i prodiskutirani rezultati testiranja genotipa *„Tree“* i *„TreePtr“* nad stvarnim problemima iz domene genetskog programiranja (simbolička regresija i centriranje vozila). Kao što je već bilo naglašeno, primjeri su napravljeni unutar ECF programske biblioteke.

Parametri genotipova i algoritma su napisani i spremljeni u konfiguracijsku datoteku (vidi Dodatak A). Preko konfiguracijske datoteke odabran je *3-turnirski* algoritam, genotip *„Tree“* ili *„TreePtr“*. Veličina populacije jest postavljena na 50 jedinki, a uvjet prestanka rada algoritma jest postavljen na broj generacija i iznosi 500 generacija.

Podatkovni uzorak koji predstavlja ulaz za simboličku regresiju identičan je onome u Tablica 2.1.

Evaluacijske klase za problem simboličke regresije jesu *SymbRegEvalOp* i *SymbRegPtrEvalOp*, za genotipe *„Tree“* i *„TreePtr“* respektivno. Za problem centriranja vozila to su klase *VehCentEvalOp* i *VehCentPtrEvalOp*.

Za svaki genotip je izvršeno 10 mjerenja metode *State::run()* (ECF) koja izvodi genetski algoritam za probleme simboličke regresije i centriranje vozila. Prosječna vremena izvođenja za genotip *„Tree“* i *„TreePtr“* jesu 3,97 i 4,59 za problem simboličke regresije te 3,10 i 3,45 za problem centriranja vozila. U oba slučaja genotip *„Tree“* se pokazao vremenski učinkovitijim. Uz pretpostavku da je, u prosjeku, omjer broja izvođenja

[22] Prije testiranja oba genotipa stanje memorije je bilo približno jednako.

postupaka evaluacije, križanja i mutacije približno jednak za oba genotipa, možemo zaključiti da je izvedba stablenog genotipa *„Tree“* i pripadajućih evolucijskih operatora učinkovitija od *„TreePtr“* genotipa. Veća vremenska složenost postupaka križanja i mutacije kod *„Tree“* genotipa naspram istih postupaka kod *„TreePtr“* genotipa se pokazalo da nosi manju težinu u sveukupnoj vremenskoj učinkovitosti, naspram vremenski učinkovitije evaluacijske metode *„Tree“* genotipa. Shodno tome, *„Tree“* stableni genotip jeste memorijski i vremenski učinkovitiji od *„TreePtr“* stablenog genotipa.

Zaključak

Problematika ovog rada je optimizacijske prirode, gdje je cilj optimirati genetski algoritam. Fokus rada je izvedba stablene strukture koja predstavlja osnovu genetskog programiranja u obliku stablenog genotipa. Počelo se sa teoretskim razmatranjima memorijske i vremenske učinkovitosti pojedinih izvedbi stablenih struktura. Među njima su se našle izvedbe preko statičkih polja, dinamičkih polja i strukture čvora s pokazivačima na djecu. Uspoređene su te tri izvedbe (Tablica 1.1) s ciljem odabira stablene strukture koja će se koristiti za izvedbu stablenih genotipa. Tako je pri analizi zahtjeva za stablene genotipe eliminirana prva izvedba stablenih struktura preko statičkih polja. Stableni genotipovi „*Tree*“ (poglavlje 4) i „*TreePtr*“ (poglavlje 5) su detaljno opisani uključujući njihove operatore evaluacije, križanja i mutacije. Opisan je način rada genotipa i pripadajućih genetskih operatora te su teoretski razmatrane memorijske i vremenske zahtjevnosti istih. Na temelju tih razmatranja moglo se pretpostaviti kako će „*TreePtr*“ genotip biti učinkovitiji od „*Tree*“ genotipa. Ta pretpostavka se pokazala krivom kod rezultata ispitivanja genotipa nad stvarnim problemima iz domene genetskog programiranja (simbolička regresija i centriranje vozila). Veliki memorijski trošak „*TreePtr*“ genotipa se pokazao glavnim uzrokom loše performanse tog genotipa zbog velikog broja promašaja pri dohvatu podataka iz memorije (priručne i radne). Kod „*Tree*“ genotipa mali memorijski trošak i zanemarivo veći vremenski trošak naspram „*TreePtr*“ genotipa rezultirao je dobrom sveukupnom performansom tog stablenog genotipa.

potpis:_____

Literatura

[1] Poli, Riccardo; B. Langdon, William; F. McPhee, Nicholas: *„A field guide to genetic programming“*, ožujak 2008.

[2] Koza, J. R.: *“Genetic programming – on the programming of computers by means of natural selection”*, MIT Press, 1992.

[3] Golub, Marin: *“Genetski algoritam, prvi dio”*, http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf , 10.06.2009.

[4] Wikipedia: *“Genetic algorithm”*, http://en.wikipedia.org/wiki/Genetic_algorithm , 10.06.2009.

[5] Wikipedia: *“Evolutionary algorithms”*, http://en.wikipedia.org/wiki/Evolutionary_algorithms , 10.06.2009.

[6] Wikipedia: *“Genetic programming”*, http://en.wikipedia.org/wiki/Genetic_programming , 10.06.2009

[7] Čupić, Marko: *“Prirodom inspirirani optimizacijski algoritmi”*, 03.06.2009.

[8] Kalpić, Damir *et al*: *“Algoritmi i strukture podataka: Složenost algoritama”*, 2009.

[9] Kalpić, Damir *et al*: *“Algoritmi i strukture podataka: Stabla”*, 2009.

[10] Haendel, Lars: *“The function pointer tutorials”*, 2005.

Sažetak

Naslov: *Usporedba učinkovitosti prikaza stabla u optimizacijskim postupcima*

U ovom radu opisani su načini programskog ostvarenja strukture stabla u optimizacijskim algoritmima, preciznije genetskim algoritmima. Metodologije rada sa stablima i interpetacija stablene strukture su opisane i razmotrene s obzirom na njihovu memorijsku i vremensku zahtjevnost. U sklopu *„Evolutionary Computation Framework“* programskog okruženja ostvarena su dva stablena genotipa koji predstavljaju osnovnu podršku za genetsko programiranje unutar *ECF*-a. Izvedbe tih genotipa se razlikuju u memorijskom zapisu strukture stabla i programskih tehnika jezika C++ korištenih za izvedbu. Memorijska i vremenska učinkovitost genotipova ispitana je nad stvarnim problemima iz domene genetskog programiranja, simbolička regresija i centriranje vozila.

Ključne riječi: genetsko programiranje, genetski algoritmi, evolucijski algoritmi, optimizacija, stablo, stableni genotip

Summary

Title: *Efficiency comparison of tree representations in optimization methods*

This paper describes various program implementations of tree structures within optimization methods, more accurately genetic algorithms. Methodologies of tree operations and interpretations of tree structures are described and considered including their time and memory consumption. Two tree genotypes have been developed within *„Evolutionary Computation Framework“* program library as basic support for genetic programming. The implementation of these two genotypes vary in memory representation as well as in program tehnic of C++ programming language used in their implementation. Memory and time efficiency of genotypes has been tested on real problems in genetic programming domain, symbolic regression and vehicle centering.

Keywords: genetic programming, genetic algorithms, evolutionary algorithms, optimization, tree, tree genotype

Dodatak A: Izgled konfiguracijske datoteke za probleme simboličke regresije i centriranja vozila

Konfiguracijska datoteka za problem simboličke regresije:

```
<ECF>
 <Algorithm>
       <SteadyStateTournament>
             <Entry key="tsize">3</Entry> <!-- velicina turnira --
>
       </SteadyStateTournament>
 </Algorithm>

 <Genotype>
       <TreePtr> <!–ili Tree -->
             <Entry key="maxdepth">6</Entry>
             <Entry key="mindepth">0</Entry>
             <Entry key="functionset">sin cos + - / *</Entry>
             <Entry key="terminalset">X</Entry>
       </TreePtr> <!–ili Tree -->
 </Genotype>

 <Registry>
       <Entry key="domain">0   0,1   0,11  0,12  0,13  0,14  0,15
 0,16  0,17  0,18  0,19  0,2   0,21  0,22  0,23  0,24  0,25  0,26
 0,27  0,28  0,29  0,3</Entry>
       <Entry key="codomain">4 3,892382873 3,808054983 3,711771383
 3,604030982 3,485372209 3,356370941 3,217638295 3,069818299
 2,91358544  2,749642113 2,578715965 2,401557149 2,218935511
 2,03163769  1,840464179 1,646226327 1,449743307 1,251839061
 1,053339231 0,855068082 0,65784544</Entry>
       <Entry key="randomizer.seed">0</Entry>
       <Entry key="population.size">50</Entry>
       <Entry key="mutation.indprob">1</Entry>
       <Entry key="term.maxgen">500</Entry>
 </Registry>
</ECF>
```

Konfiguracijska datoteka za problem centriranja vozila:

```
<ECF>
 <Algorithm>
       <SteadyStateTournament>
             <Entry key="tsize">3</Entry> <!-- velicina turnira --
>
       </SteadyStateTournament>
 </Algorithm>

 <Genotype>
       <TreePtr> <!–ili Tree -->
             <Entry key="maxdepth">6</Entry>
             <Entry key="mindepth">0</Entry>
             <Entry key="functionset">sin cos + - / *</Entry>
```

```
            <Entry key="terminalset">X</Entry>
      </TreePtr> <!—ili Tree -->
 </Genotype>

 <Registry>
      <Entry key="randomizer.seed">0</Entry>
      <Entry key="population.size">50</Entry>
      <Entry key="mutation.indprob">1</Entry>
      <Entry key="term.maxgen">500</Entry>
 </Registry>
</ECF>
```