SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Diplomski seminar

Petlje u genetskom programiranju

Maja Kontrec

mentor: prof. dr. sc. Domagoj Jakobović

Zagreb, svibanj 2013.

Sadržaj

1. Uvod

Programske petlje važan su dio svakog programa, no u genetskom programiranju ne koriste se često zbog složenosti njihovog evoluiranja. Osnovne programske funkcije, poput iteriranja po n-dimenzionalnom polju, u genetskom programiranju teško su rješive upravo iz tog razloga. Učinkovit način definiranja i evoluiranja petlji genetsko programiranje mogao bi proširiti na puno veće područje primjene.

Problemi koji se javljaju prilikom implementacije počinju već pri samoj formulaciji petlje, pri čemu je najteže riješiti pitanje njene konačnosti, odnosno, završetka u nekom konačnom vremenu. Drugi problem koji se javlja je ugnježđivanje, odnosno kako definirati petlju unutar petlje koja se također koristi podacima o stanju nad-petlje.

Postoji nekoliko pokušaja rješavanja ovih problema, no ni jedan nije dovoljno općenit i primjenjiv na različite vrste problema. Također, pitanje konačnosti petlje nije razrješeno, te dosta rješenja taj problem rješava tako da petlju zaustavi kada primjeti da je broj prolazaka kroz petlju dosegao neki definirani broj.

U nastavku su opisana postojeća rješenja, te je predložen nov način za ostvarenje petlji u genetskom programiranju.

2. Eksplicitne for-petlje

Jedan od načina ostvarenja petlji u genetskom programiranju je eksplicitna for-petlja [1]. Postoje dvije vrste tih petlji, jedna u kojoj je definiran samo broj ponavljanja naredbi petlji (`FOR-LOOP1`; oznaka preuzeta iz [1]), te druga, koja više liči na klasičnu for-petlju, u kojoj je definiran početni i završni indeks (`FOR-LOOP2`; oznaka preuzeta iz [1]) pomoću kojeg se određuje broj ponavljanja naredbi petlje.

```
1. (FOR-LOOP1 NUMBER_OF_ITERATIONS BODY)

2. (FOR-LOOP2 START END BODY)
```

Slika 1: Definicije petlji

Na slici 1 vide se definicije tih petlji. `NUMBER_OF_ITERATIONS` označava broj ponavljanja petlje, `START` početni indeks, `END` završni indeks, a `BODY` tijelo petlje.

S obzirom na to kako se brojevi ponavljanja određuju razlikujemo *simple loop* i *unrestricted loop.* Razlika između ove dvije vrste je ta da se kod *simple loop* petlje vrijednosti varijabli odgovornih za broj ponavljanja određuju nasumično, dok se kod *unrestricted loop* petlji, te vrijednosti izračunavaju pomoću aritmetičkih ili nekih drugih funkcija.

U ovakvoj implementaciji petlje, ne postoji mogućnost za beskonačnom petljom.

2.1. FOR-LOOP1

Kod *simple loop* `FOR-LOOP1` petlje, broj ponavljanja određuje se nasumično, i to u intervalu [1, `MAX_ITERATIONS`], gdje je `MAX_ITERATIONS` je broj zadan od strane programera. Prilikom mutacije i križanja, vrijednost varijable `NUMBER_OF_ITERATIONS` uvijek ostaje u zadanom rasponu [1, `MAX_ITERATIONS`].

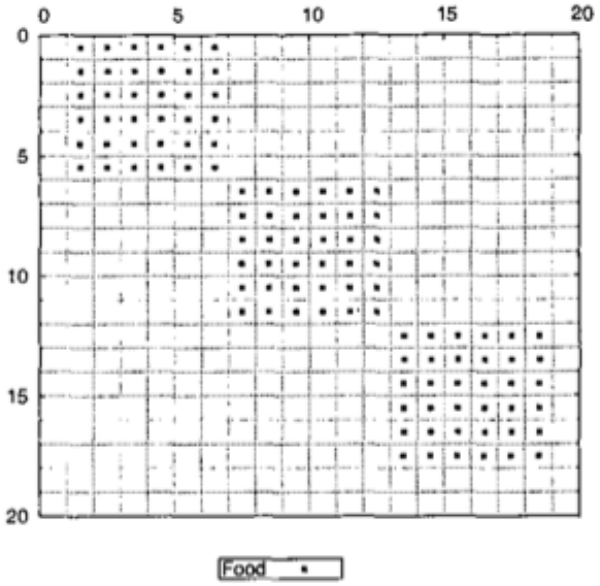
Kod *unrestricted loop* `FOR-LOOP1` petlje, broj ponavljanja izračunava kao rezultat neke funkcije.

2.2. FOR-LOOP2

Kod `FOR-LOOP2` petlje, tijelo se izvršava za svaku vrijednost brojača između `START` i `END`. Ukoliko je vrijednost varijable `START` veća od vrijednosti `END`, petlja se neće izvršiti. Analogno situaciji kod `FOR-LOOP1` petlje, u *simple loop* inačici ove petlje, `START` i `END` određuju se nasumično, a u *unrestricted loop* inačici kao rezultat nekih operacija.

2.3. Primjena na modicirani Santa Fe problem

Kako bi se pokazala učinkovitost ove implementacije petlji u genetskom programiranju, autori su evoluirali programe s petljom koji rješavaju modificirani Santa Fe problem, i to takav gdje je raspored hrane pravilan [1] (polje veličine 20x20 s 108 komada hrane posloženih u 3 kvadrate veličine 6x6). Ovakav raspored hrane, kao što se vidi na slici 2, potiče evoluciju petlji unutar programa zbog toga što, ukoliko se petlja ne koristi, rezultirajuće stablo bilo bi jako duboko. Dobrota se u ovom slučaju računa kao broj nepojedene hrane nakon 600 koraka. Optimalno rješenje ovog problema, u slučaju da program kreće od pozicije (0, 0), imalo bi oko 160 koraka, a rješenje obilaska grubom silom, koji prolazi kroz svako polje, oko 480 koraka.



Slika 2: polje modificiranog Santa fe problema

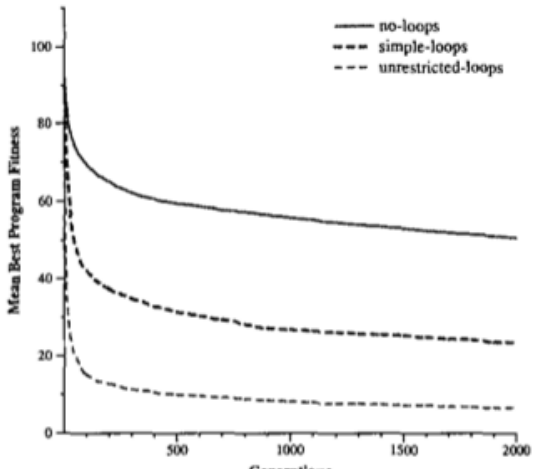
Funkcije i završni znakovi korišteni za rješavanje ovog problema prikazani su u talbici 1.

Tablica 1: korišteni završni znakovi i funkcije

naziv čvora	vrsta čvora	opis
`move`	`završni znak`	micanje za jedno mjesto ispred sebe - košta jedan korak
`turnLeft`	`završni znak`	okretanje lijevo u odnosu na smjer u kojem trenutno gleda - košta jedan korak
`turnRight`	`završni znak`	okretanje desno u odnosu na smjer u kojem trenutno gleda - košta jedan korak
`randTimes`	`funkcija`	generira nasumičan broj u

		intervalima [0,6], [0,20] ili [0,60]
`ifFoodAhead`	`funkcija`	prima dva argumenta - ukoliko postoji hrana ispred, izvršava prvi argument, inače drugi
`prog2`	`funkcija`	prima 2 argumenta i slijedno ih izvršava
`prog3`	`funkcija`	prima 3 argumenta i slijedno ih izvršava
`FOR-LOOP1`	`funkcija`	prima 2 argumenta - prvi argument određuje koliko puta će se drugi argument izvršiti vraća koliko je komada hrane preostalo nakon izvršavanja drugog argumenta

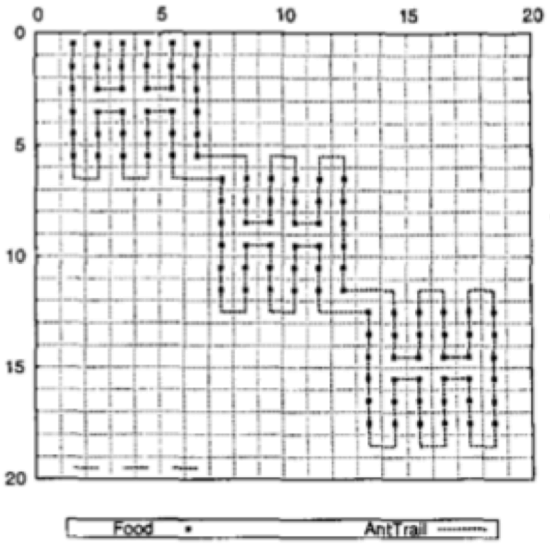
vaj pristup rješavanja problema rezultirao je dobrim rješenjima. Na slici 3, vidimo usporedbu dobrota između rješenja koje ne koriste petlje, rješenja koje koriste *simple loop* i rješenja koja koriste *unrestricted loop*.



Slika 3 usporedba rješenja s obzirom na korištenje ili nekorištenje petlji

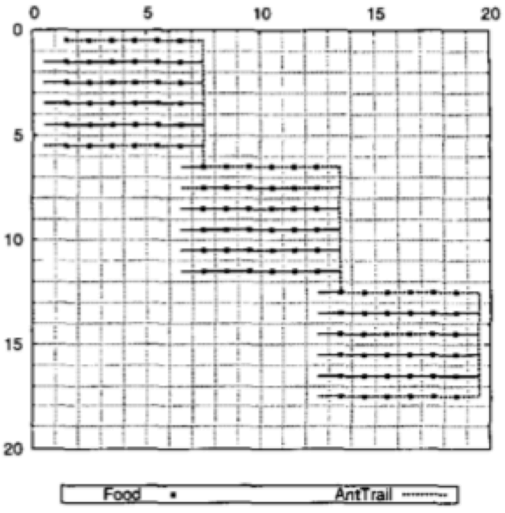
Na slikama 4 i 5 prikazana su dobivena rješenja, sami programi i rezultat njihovog izvođenja na zadanom polju.

```
(FOR-LOOP1 times5
   (FOR-LOOP1 times5
         (FOR-LOOP1 times5(ifFoodAhead
               ((prog2 move turnRight) turnRight)
               ( prog2 turnLeft (ifFoodAhead
                                    (prog2 move move) move)))))))
```



Slika 4 prikaz jednog od rješenja

```
(FOR-LOOP1 times5 (FOR-LOOP1 times4
(prog3 (FOR-LOOP1 times4 move)
(prog3 (iffoodAhead (prog3 move move move) move)
(prog3 (prog2 turnRight move) turnRight
(iffoodAhead (FOR-LOOP1 times4 move)
(prog2 turnRight turnleft)))
(iffoodAhead (prog3 move move move) move))
(prog3 (ifFoodAhead (ifFoodAhead
(prog3 tumkft turnRight move) move) turnLeft) turnLeft (Prog2 move move)))))
```



Slika 5: prikaz jednog od rješenja

2.4. Primjena na sortiranje niza

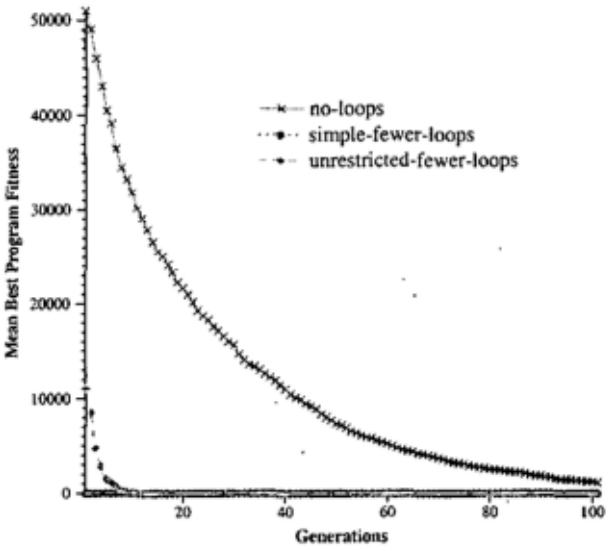
Sortiranje niza brojeva jedan je od problema koji prirodno zahtjevaju upotrebu programskih petlji. U [1], provedeni su eksperimenti sortiranja pomoću `FOR-LOOP2` petlje korištene su funkcije i završni znakovi prikazani u tablici 2. Eksperimenti su, iz razloga da ne bi predugo trajali, provedeni na polju duljine 7. Dobrota je jednaka količini brojeva koji nisu na svojem mjestu i to za svih 7! permutacija početnog niza.

Nedostatak ovih petlji je svakako nemogućnost korištenja indeksa petlji u ugnježđenim petljama (kao npr. u jednostavnom *bubble sortu*), što se i vidi u evoluiranim rješenjima.

Tablica 2: korištene funkcije i završni znakovi

naziv čvora	vrsta čvora	opis
`POS`	`završni znak`	nasumičan broj u intervalu [0,6]
`IfLessThanSwap (ILETs)`	`funkcija`	prima 2 argumenta - ako je prvi argument manji od drugoga, oni zamjenjuju pozicije te se vraća vrijednost većeg argumenta
`Prog2`	`funkcija`	prima dva argumenta i izvršava ih slijedno
`FOR-LOOP2`	`funkcija`	prima 3 argumenta - `START`, `END` i `BODY` (opisano u 2.2)
`+, -, ×, /`	`operator`	uobičajeno značenje

Na slici 6 prikazan je graf usporedbe evoluiranih rješenja sa i bez petlji. Na slici 7 prikazan je primjerak jednog evoluiranog programa koji je uspio sortirati niz uz 22 usporedbe i 10 zamjena.



Slika 6: usporedba evoluiranih rješenja bez i sa petlji

```
(Prog2 (Prog2 (Prog 2
(FOR-LOOP2 POS3 POS4
      (ILETs i (i+l)))
(FOR-LOOP2 POS2 POS6
      (ILETs i (,+I))))
(Prog2
      (FOR-LOOP2 POS3 POS4 (ILETs
      i (i+1)))
(FOR-LOOP2 POS4 POS5
      (ILETs i (i+l)))))
(Prog2
      (FOR-LOOP2 POSl POS6 (ILETs
      i (,+I)))
(Prog2
      (FOR-LOOP2 POSl POS3 (ILETs
      i (,+I)))
(ForLoop2 POSO POS6 (ILETs I
      (i+l))))))
```

Slika 7: primjer evoluiranog programa

3. Indeksirane for-petlje

Indeksirane for-petlje omogućuju upotrebu indeksa, odnosno vrijednosti iteratora petlje za izračune unutar samog tijela petlje. Oblikovanje ovakve petlje opisano je u [2]. Iako je rješenje prilično uspješno primjenjeno na sortiranje niza, unutar petlje moguće je koristiti samo vlastiti indeks, no ne i onaj od nadpetlje ukoliko ona postoji.

3.1. Primjena na sortiranje niza

Kako bi se ostvario program za sortiranje niza koji se može evoluirati, a sadržava indeksirane for-petlje definirani su sljedeći završni znakovi i funkcije (tablica 3):

Tablica 3: korišteni završni znakovi i funkcije za realizaciju indeksirane for-petlje

naziv čvora	vrsta čvora	opis
`*len*`	`završni znak`	veličina dvodimenzionalnog polja
`index`	`završni znak`	iterator polja
`e-`	`funkcija`	prima dva argumenta i vraća njihovu razliku `(e- n m) = n - m` ukoliko argumenti nisu brojevi, vraća 0
`e1+`	`funkcija`	prima jedan argument i vraća isti argument uvećan za jedan `(e1+ n) = n + 1` ukoliko argumenti nisu brojevi ili je rezultat manji od 0, vraća 0
`e1-`	`funkcija`	prima jedan argument i vraća isti argument umanjen za jedan `(e1- n) = n - 1` ukoliko argumenti nisu brojevi ili je rezultat manji od 0, vraća 0

`order`	`funkcija`	prima 2 argumenta i zamjenjuje njihov redoslijed u polju ukoliko je element na poziciji prvog argumenta veći od onog na poziciji drugog uporaba: `(order x y)`
`swap`	`funkcija`	slično kao funkcija `order`, prima 2 argumenta te zamjenjuje elemente na njihovim pozicijama unutar polja uporaba: `(swap x y)`
`wismaller, wibigger`	`funkcija`	primaju dva argumenta i vraćaju onaj na čijoj je poziciji manji, odnosno veći element uporaba: `(wismaller x y)`
`if-lt`	`funkcija`	prima 3 argumenta: `x`, `y` i `work` - ukoliko je element polja na poziciji `x` manji od onog na poziciji `y` obavlja neki rad - `work` uporaba: `(if-lt x y work)`
`dobl`	`funkcija`	for-petlja; prima 3 argumenta: `start`, `end` i `work` postavlja `index` na vrijednost `start` i povećava njegovu vrijednost za jedan sve dok nije jednak varijablama `end` ili `*len*`

Ovako definiranim funkcijama i završnim znakovima, evoluirano je nekoliko uspješnih sortirajućih algoritama prikazanih u nastavku. Zanimljivo je primjetiti kako su sva rješenja na kraju evoluirala u oblik algoritma *bubble sort.*

3.1.1. Order sort

Ovaj evoluirani algoritam koristi funkciju order za sortiranje. Najbolji dobiveni algoritam, prikazan na slici 8, je ustvari *bubble sort.* Izraz `e- index index` je prekrižen iz razloga što uvijek vraća 0.

```
(dobl index
      *len*
      (dobl (e- index index)
            *len*
            (order (e1- index) index)))
```

Slika 8: evoluirani algoritam koji koristi funkciju order

3.1.2. Swap wismaller wibigger sort

Na slici 9 prikazan je sort koji sadrži funkcije `swap, wismaller` i `wibigger.` Radi jednak posao kao i malo prije spomenuti *Order sort*, ali uz uporabu drugih funkcije.

```
(dobl index
      *len*
      (dobl (swap index *len*)
            *len*
            (swap
              (wibigger (e1- index)
                        index)
              index)))
```

Slika 9: evoluirani algoritam koji koristi funkcije swap, wismaller i wibigger

3.1.3. If-lt swap sort

Na slici 10 prikazan je evoluirani sort s funkcijama `if-lt` i `swap`. Prekriženi izraz je prekrižen iz razloga što njegov rezultat ne donosi nikakvu promjenu u nizu. Ovaj algoritam je također malo drugačija inačica *bubble sorta.*

```
(dobl index
      *len*
      (dobl (swap *len* *len*)
            *len*
            (if-lt index
                   (e1- index)
                   (swap
                     (e1- index)  index))))
```

Slika 10: evoluirani algoritam koji koristi funkcije if-lt i swap

3.1.4. If less swap sort

Na slici 11 prikazan je sort koji koristi funkcije `if, less` i `swap`. Ponovno, prekriženi izraz je prekrižen iz razloga što njegov rezultat ne donosi nikakvu promjenu u nizu. Također, ovaj algoritam je evoluirao u *bubble sort.*

```
(dobl
  index
  *len*
  (dobl (swap *len* index)
        *len*
        (if (less (el+ index) index)
            (swap index
                  el+ index)))))
```

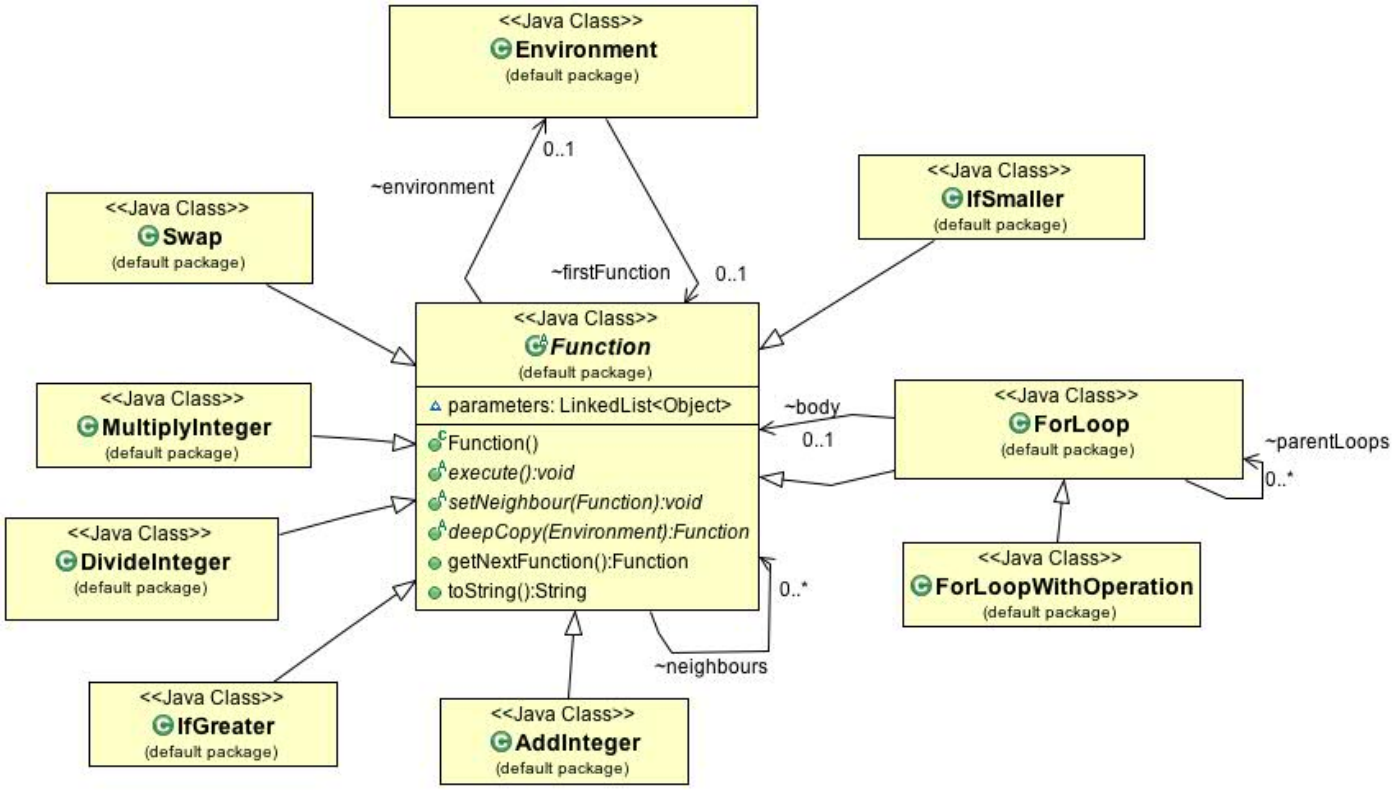
Slika 11: evoluirani algoritam koji koristi funkcije if less i swap

4. Vlastita implementacija

U ovdje implementiranim for-petljama moguće je koristiti, osim vlastitog, i indekse nadpetlji ukoliko je petlja ugniježđena. U nastavku je opisana implementacija primijenjena na sortiranje niza cijelih brojeva.

4.1. Jedinka

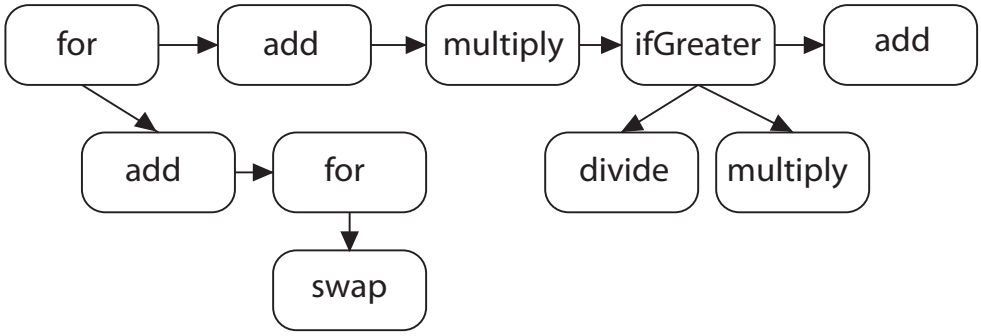
Jedinka je predstavljena okolinom koja sadrži varijable i program koji te varijable koristi ili mijenja. Program sa sastoji od niza funkcija koje se redom izvršavaju. Implementirano je osam funkcija: *for-loop*, *for-loopWithOperation*, *swap*, *ifSmaller*, *ifGreater*, *add*, *multiply* i *divide*. Ovakvom podjelom na funkcije koje su spojene u listu omogućeno je jednostavno križanje između programa. Na slici 12 prikazan je dijagram razreda implementirane jedinke.



Slika 12: Dijagram jedinke

Sve funkcije nasljeđuju klasu *Function* čime se osigurava da svaka funkcija sadrži listu parametara i okolinu, te metode za izvođenje i postavljanje pokazivača na susjeda. Susjed je ustvari sljedeća naredba programa. Program se jednostavno izvršava pokretanjem prve funkcije. Prva funkcija, nakon obavljanja svoje zadaće poziva sebi susjednu funkciju da obavi

svoju. Jednako tako, svaka sljedeća funkcija nakon obavljanja svoje zadaće poziva izvršavanje susjedne funkcije (slika 13).



Slika 13: izgled programa

Ovakva atomarna struktura jedinke omogućuje jednostavno vršenje genetskih operacija poput križanja i mutacije.

4.2. Funkcije

Funkcije su realizirane kao objekti koji prilikom stvaranja primaju referencu na okolinu u kojoj se nalaze. Osim toga, primaju i podatke koji govore o tome koje podatke iz okoline trebaju koristiti, te kako ih trebaju koristiti.

Funkcija parametre, odnosno varijable kojima se koristi, ne prima direktno već ih dohvaća putem indeksa. Indeksi govore gdje potrebne varijable nalaze unutar proslijeđene okoline. Do stvarnih vrijednosti parametara, dakle, funkcija dolazi dohvaćanjem elementa iz određenih lista varijabli okoline.

4.2.1. Swap

Funkcija *Swap* obavlja zamjenu elemenata unutar niza. Kao argumente prilikom stvaranja prima indekse elemenata - koji mogu biti indeksi for petlje (ukoliko se funkcija nalazi unutar iste) ili obične varijable i polje unutar kojega treba obaviti zamjenu. Ukoliko funkcija želi dohvatiti elemente na indeksima izvan niza, njeni indeksi pomiču se na granice niza.

4.2.2. MultiplyInteger

Ova funkcija, kao što joj i ime kaže, dohvaća zadanu varijablu iz okoline i množi je s nekim cjelobrojnim operandom.

4.2.3. DivideInteger

Jednako kao i *MultiplyInteger*, ova funkcija dohvaća zadanu varijablu iz okoline i dijeli je s nekim cjelobrojnim operandom. U slučaju da je operand jednak nuli, kako bi se izbjeglo dijeljenje s nulom, on se mijenja u 1.

4.2.4. AddInteger

Jednako kao i prethodne dvije funkcije, ova operacija uzima određenu varijabli iz okoline i dodaje joj vrijednost operanda.

4.2.5. IfGreater

Funkcija *IfGreater* uspoređuje dvije vrijednosti (bilo varijabli ili indeksa for-petlji) te na osnovi rezultata izvodi neku radnju. Rezultat je istinit ukoliko je prvi argument veći od drugoga. Ima tri susjedne funkcije: prvu, koja se izvršava ako je rezultat istinit, drugu, koja se izvršava ako nije, te treću koja se izvršava nakon prvog ili drugog susjeda.

4.2.6. IfSmaller

Analogno funkciji *IfGreater*, ova funkcija također uspoređuje dvije vrijednosti, no istinitost rezultata ovisi o tome da li je prvi argument manji od drugoga (istina) ili ne (neistina). Također ima tri susjedne funkcije s jednakim značenjem kao u funkciji *IfGreater.*

4.2.7. ForLoop

ForLoop petlja pri stvaranju prima pet parametara: dubinu, početni indeks, završni indeks, tijelo i okolinu. Dubina petlje označava koliko petlja ima nadpetlji (petlja dubine 0 nema nadređenu petlju, dok ona s dubinom 2 ima dvije nadređene petlje). Jednako kao i "klasična" for-petlja, ona svoje tijelo izvršava *end-start* puta.

Prilikom stvaranja, petlja u svoju okolinu postavlja vrijednost svojeg indeksa, te ga tijekom svog izvođenja ažurira, te time omogućuje korištenje njenih indeksa od strane svog tijela i ugniježđenih petlji.

4.2.8. ForLoopWithOperation

Ova funkcija nasljeđuje funkciju *ForLoop*, a razlikuje se u tome što se indeks petlje u svakoj iteraciji osvježava kao rezultat neke od aritmetičkih binarnih operacija (-, * ili /). Zbog toga, pri stvaranju mora primiti još sva dodatna argumenta koja određuju o kojoj se operaciji radi i koji operand se pri toj operaciji koristi.

4.3. Križanje

Zbog same strukture jedinke, križanje je vrlo jednostavno. Ostvareno križanje opisano je u nastavku.

4.3.1. Križanje varijabli okoline

Budući da je broj varijabli okoline jednak za sve jedinke u populaciji, križanje tih vrijednosti se vrši uniformno. Vjerojatnost da se u nekom trenutku uzme varijabla jednog od roditelja je 0.5.

4.3.2. Križanje funkcija

Križanje funkcija također se vrši uniformno. Prolazeći kroz stabla prvog i drugog roditelja, dijete se sastavlja uzimajući funkciju po funkciju od prvog ili drugog roditelja. Jednako kao kod križanja varijabli okoline, vjerojatnost da se uzme funkcija jednog od roditelja jednaka je 0.5.

4.4. Mutacija

Ostvarene su dvije vrste mutacija; mutacija brisanjem i mutacija promjenom parametra.

4.4.1. Mutacija brisanjem

Ova mutacija briše nasumično odabranu funkciju iz jednike. Na početku, pokazivač na odabranu funkciju pokazuje na početnu funkciju jedinke. S vjerojatnošću od 0.5, pokazivač se pomiče na sljedeću funkciju. Nakon što stane, briše se funkcija na koju pokazuje.

4.4.2. Mutacija promjenom parametara

Za ostvarenje ove mutacije, svaka funkcija ima specifično implementiranu metodu za mutaciju promjenom parametara. Tako npr. kod funkcije *MultiplyInteger*, mutacija mijenja parametar operanda, dok kod funkcije *Swap* mutacija mijenja indekse polja na kojim se nalaze elementi koje ona treba zamijeniti. Sve mutacije poštuju prethodno opisana ograničenja vlastite funkcije.

4.5. Dobrota

Pojedinačna dobrota se izračunava kao zbroj elemenata polja koji su na jednakom mjestu kao i oni u sortiranom nizu. Kako ne bi došlo do toga da rješenje evoluira za specifičan raspored elemenata a nizu, dobrota je jednaka zbroju pojedinačnih dobrota za određen (podesiv) broj nasumičnih permutacija zadanog niza.

4.6. Rezultati

Prikazano rješenje nije dalo nikakve značajne rezultate, te postoji više mogućih razloga zbog čega je tako. U nastavku su izneseni i opisani ti razlozi.

Križanje dviju jedinki može biti prilično destruktivno - vjerojatnost da se križanjem dviju jedniki dobije donekle dobra jedinka možda nije dovoljno velika. Budući da se križanje vrši uzimanjem funkcije po funkcije od svakog roditelja, petlje, nužne za općenit i ispravan algoritam sortiranja, mogu biti potpuno izbačene iz novodobivene jedinke.

Sličan problem javlja se i kod mutacije brisanjem. Naime, može se dogoditi da mutacija obriše čitav niz ugniježđenih petlji brisanjem prve petlje u nizu.

Također, dobrota bi možda trebala uzimati u obzir i duljinu samog programa te postojanje petlji (koje su, kao što je već prije spomenuto, nužne za općenit i ispravan algoritam sortiranja).

5. Zaključak

Iako petlje u genetskom programiranju nisu jednostavne za implementirati, prikazana rješenja donekle su uspjela u svome naumu. Unatoč tome, ovo područje i dalje ostaje otvoreno za pronalazak nekog boljeg i općenitijeg rješenja. Osim for-petlji, bilo bi dobro razmotriti i mogućnost ugradnje while petlje. Rješavanje ovog problema veći je izazov jer je pitanje beskonačnosti izraženiji nego kod for-petlje.

5. Literatura

[1] Vic Ciesielski, Xiang Li: *Experiments with explicit for-loops in genetic programming*

[2] Kenneth E. Kinnear, Jr.: *Generality and Difficulty in Genetic Programming: Evolving a sort*