SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Traženje optimalne funkcije paljenja potisnika *Moonlandera* genetskim programiranjem u Common Lispu

Veljko Dragšić

Voditelj: *Domagoj Jakobović*

Zagreb, Listopad, 2009.

Table of Contents

Uvod

Cilj seminara je uz pomoć genetskog programiranja pronaći matematičku funkciju koja bi optimalno uključivala potisnik *Moonlandera* odnosno letjelice. Slijetanje je uspješno ukoliko je brzina letjelice prilikom dodira sa površinom što manja. Kako bi to postigli moramo osim paljenja potisnika i trenutne brzine voditi računa i o visini, količini preostalog goriva i masi letjelice. Rješavanje ovog problema nije odveć kompleksno, pa je stoga dobar primjer za upoznavanje sa genetskim programiranjem. Za implementaciju je odabran Common Lisp, moderan dijalekt Lispa. To je ujedno i drugi cilj ovog seminara, upoznavanje sa programskim jezikom višeg razreda koji svojim velikim dijelom ulazi u funkcijsku paradigmu programiranja.

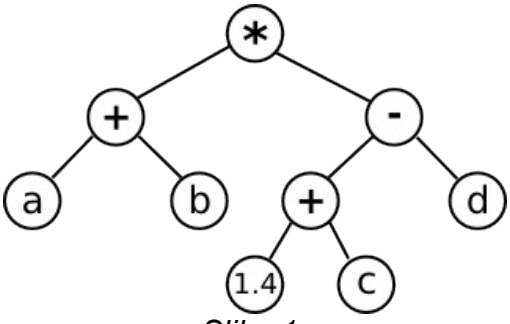
Uvod u genetsko programiranje

Genetsko programiranje ulazi u skupinu evolucijskih algoritama, inspirirano je promatranjem prirodne evolucije u pogledu dobivanja novih jedinki razmjenom (boljih) osobina odnosno genetskog materijala roditelja kroz operatore križanja i mutacije.

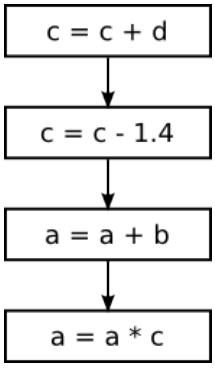
Za razliku od genetskih algoritama kojima pokušavamo dobiti točno određeno rješenje, najčešće broj koji zadovoljava kriterije pretrage, kod genetskog programiranja tražimo cijelu funkciju koja bi nam dala dovoljno dobro rješenje. Za primjer u ovom seminaru tražimo optimalnu funkciju paljenja potisnika kod letjelice sa ciljem što "ugodnijeg" dodira sa površinom.

Prikazi jedinke

Jedna jedinka (pritom misleći na funkciju, program kod GP-a) se sastoji od operatora i terminala. Operatori mogu biti logičke funkcije poput *AND*, *OR*, *XOR*, *NOT* ili jednostavnije matematičke funkcije, poput zbrajanja, oduzimanja, množenja, dijeljenja, sinusa, logaritma i tako dalje. Terminali su konstante ili varijable koje proizvoljno odabiremo, naravno u našem cilju je da direktno utječu na problem koji rješavamo. Na početku rada sa genetskim programiranjem odabiremo željene operatore i terminale.



Slika 1.



Slika 2.

Jedinku možemo prikazati kao stablo (Slika 1.), linearnu strukturu (Slika 2.) ili graf. Jedinka odnosno program zapisan kao stablo se izvodi obilazeći čvorove s lijeva na desno, prvo prema dubini. Programi linearno zapisani se izvode proceduralno, funkcija za funkcijom. Prikaz grafom bi se najlakše mogao usporediti sa automatom u kojem su stanja predstavljena operatorima i terminalima, te se program izvršava prelaskom iz stanja u stanje. Prikaz stablom je pogodniji za funkcijske jezike poput Lispa, jer su znatno prilagođeniji za rad sa listama, dok su za linearne više pogodni proceduralni jezici.

Oba prikaza predstavljaju istu funkciju, ali na različite načine. Za ovaj seminar se koristi prikaz stablom, tako da se ostali prikazi neće detaljnije objašnjavati.

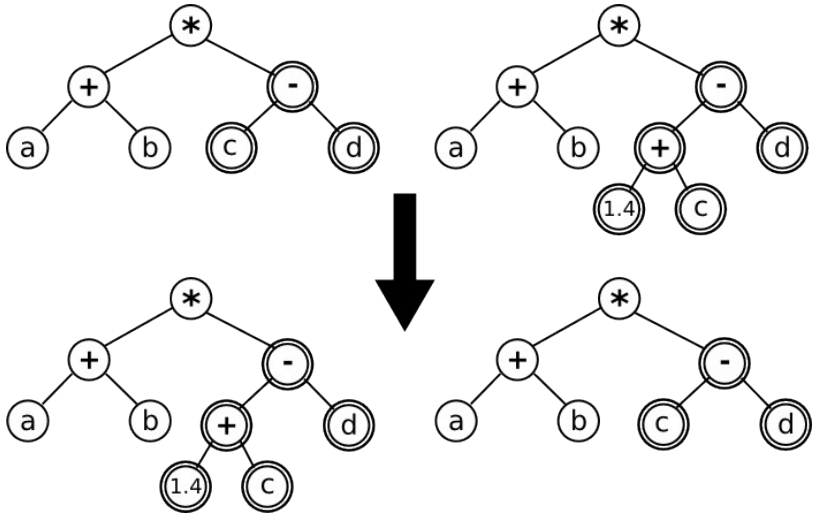
Genetski operatori

Za inicijalizaciju genetskog programiranja osim odabira skupa operatora i terminala, trebamo još odabrati veličinu populacije (količina jedinki), dubinu stabla, vjerovatnost mutacije, način i vjerovatnost križanja. Nakon toga nekom slučajnom metodom kreiramo

inicijalnu populaciju različitih jedinki. Također se moramo odlučiti za način određivanja kvalitete određene jedinke, tj. koliko nam dobiveni rezultat blizak željenom, te način odabira jedinki za primjenu genetskih operatora, odnosno križanja i mutacije. Kroz rad genetskog programiranja možemo mijenjati neke inicijalno zadane parametre, poput vjerovatnosti mutacije i načina križanja.

Križanje

Najvažniji i ujedno najsloženiji genetski operator je križanje (Slika 3.), njime se dijelovi programa (jednike) izmjenjuju između dvije jednike, najčešće stvarajući dvije nove. Biološki primjer je razmjena genetskog materijala između roditelja prilikom stvaranja potomstva. Na primjeru stabla to se izvodi odabirom podstabla kod oba "roditelja" i njihovom zamjenom.

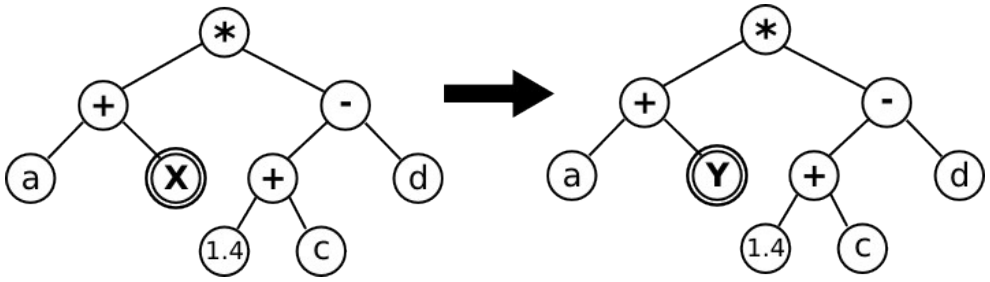


Slika 3.

Način odabira podstabla je izrazito bitan i može biti jednako tako složen. Najjednostavniji način je nasumični odabir, složeniji može u obzir uzimati mnogobrojne parametre koji utječu na kvalitetu križanja. Kod nasumičnog odabira je križanje često destruktivnije nego konstruktivnije, odnosno češće nove jednike imaju lošiju ocijenu kvalitete nego roditelji. Jedno od poboljšanja je da se na dva roditelja napravi više nasumičnih križanja i odaberu bolja. U biologiji se izmjenjuje sličan genetski materijal, isto tako i kod naših jedinki možemo uspoređivati sličnost podstabla koja planiramo križati, na temelju veličine, sličnosti operatora i terminala, njihovom međusobnoj udaljenosti i slično.

Mutacija

Drugi genetski operator je mutacija (Slika 4.). Mutacija uključuje samo jednu jedinku kojoj se najčešće nasumično odabrani dio programa promijeni. Promjena može biti ubacivanje nekog drugog operatora ili terminala na mjesto postojećeg. Čest način je i zamjena dva čvora u jednoj jedinci. Mutacija se ne mora nužno koristiti, a kada se koristi njena vjerovatnost izrazito mala zbog velike vjerovatnosti destruktivnosti. Njome se postiže uvođenje novog genetskog materijala, za primjer kako bi se izbjegli mogući lokalni minimumi.



Slika 4.

Ocjenjivanje kvalitete jedinke

Osim genetskih operatora moramo voditi i računa o načinu ocjenjivanja kvalitete pojedine jedinke i načina odabira jedinka za križanje. Kvaliteta nam je bitna jer najčešće na temelju nje odlučujemo koje ćemo jedinke križati i pomoću nje određujemo je li neka jedinka postigla rezultat koji želi postići. Kvalitetu reprezentiramo brojem, neki od mogućih načina su da promatramo promjenu kvalitete jedinke, uzimamo standardiziranu kvalitetu uzevši da je recimo nula najbolji rezulat, a ostali što su veći to su lošiji, još možemo kvalitete svih jedinki normalizirati u neki interval, npr. [0, 1].

Način odabira jedinki za reprodukciju možemo podijeliti na dvije osnovne podjele, uzimati one sa boljom ocjenom kvalitete i zatim primjenom genetskih operatora stvarati potomstvo ili odabrati veći broj jedinki i stvarati veće potomstvo na kojem onda provodimo selekciju ovisno o dobivenoj kvaliteti.

Neke od metoda su odabir jedinki s obzirom na njihovu kvalitetu, primjenjujući neku funkciju za selekciju s obzirom na njihov rang ako su poredane po kvaliteti, nasumični odabir nekoliko jediniki i zatim turnirsko određivanje koje će preživjeti i neke druge.

Primjer genetskog programiranja

U nekom opećenitom primjeru genetskog programiranja prvo odaberemo operatore i terminale, te osnovne parametre poput količine populacije. Zatim stvorimo inicijalnu populaciju jedniki i evaluiramo njihovu kvalitetu. Nakon toga odabiremo jedinke i na njima provodimo genetske operatore križanja i mutacije, ubacujemo novodobivene jedinke u populaciju, a najlošije izbacujemo. Na taj način stvaramo nove generacije dok neka jedinka ne postigne rezultat koji nas zadovoljava.

Lisp i Common Lisp kao jedan od njegovih nasljednika

Lisp dolazi kao skračenica od naziva List Processing, a osmislio ga je John McCarthy još davne 1958. godine na prestižnom Massachusetts Institute of Technology. Zamišljen je kao matematička notacija za računalne programe, nasuprot Fortranu koji je bio više inžinjerskog odnosno proceduralnog pristupa.

Osnovna obilježja Lispa

Glavne osobine su mu pretežito funkcijska paradigma i korištenje izraza (engl. *expression*) nasuprot deklaracijama (engl. *statement*), baziranje na lambda računu, mogućnost metaprogramiranja, veliko oslanjanje na makro naredbe, *garbage collector*, a računalni kod se piše kao simolički izrazi (engl. *S-expressions*) odnosno kao skupina ugnježđenih lista što ga čini vizualno prepoznatljivim ponajviše zbog količine zagrada u programskom kodu. Sve navedeno ga svrstava u skupinu visoko dinamičnih i fleksibilnih programskih jezika više klase. Za Lisp je predviđeno da se izvodi na Lisp mašinama, posebno dizajniranim računalima za izvršavanje Lisp programa. Kako Lisp nikada nije stekao popularnost poput Fortrana, C-a i drugih u to vrijeme, tako su i tvrtke koje su se komercijalno bavile njime propale. Bitno je spomenuti kako su se u njega polagale izrazito velike nade pogotovo na području umjetne inteligencije, no prevelika zanesenost umjetnom inteligencijom uopće je brzo splasnula.

Često je uvriježeno mišljenje kako je Lisp spor, memorijski prezahtjevan i zastarjeli programski jezik, što nikako nije točno. Lisp se kroz desetljeća razvio u nekoliko pravaca odnosno dijalekata, najpoznatiji su Scheme i Common Lisp, kojim se i bavimo u ovom seminaru. Osim što se razvio u moderne jezike, također su i neki drugi danas izrazito raširenі programski jezici poput Jave, Pythona, Rubya i drugih počeli uvoditi neke mogućnosti koje Lisp posjeduje već desetljećima, poput *lambde*, *closurea*, *map*/*reducea* itd. Neki tek nadolazeći jezici poput Clojurea, Dylana i Haskella su velikim dijelom inspirirani Lispom.

REPL

Danas za izvođenje Common Lisp programa imamo na desetke implementacija prevodioca i razvojnih okruženja za sve operacijske sustave. U ovom seminaru je korištena Steel Bank Common Lisp implemetacija i Slime razvojno okruženje u Emacsu.

Kroz Slime pristupamo Lisp "konzoli" koju točnije zovemo REPL (engl. *Read-eval-print-loop*). Primjer (1.) ilustrira evaluiranje funkcije zbrajanja kojoj dajemo tri parametra.

```
CL-USER> (+ 1 2 3)
6                                                    (1.)
CL-USER>
```

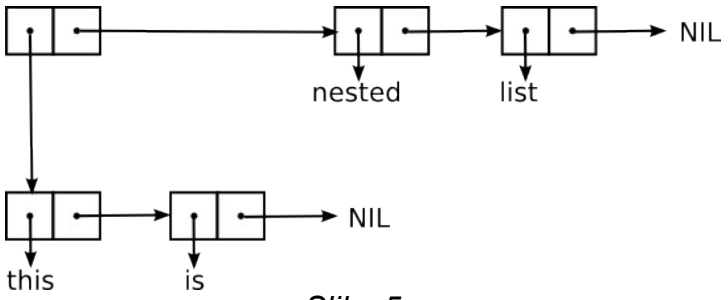
Kroz REPL može interaktivno komunicirati sa Lispom, definirati varijable, funkcije i makro naredbe, izvoditi izraze, pokretati debugger i ocenito vidjeti trenutno stanje sustava. Lisp je dinamičan jezik, tako da prvo pročita programski kod (engl. *read*), zatim ga izvede (engl. *evaluate*) i eventualno ispiše (engl. *print*). Naravno najveći dio se odvija prilikom izvođenja funkcija, poput ekspandiranja makroa i slično.

Common Lisp nudi mnogo ugrađenih funkcija, ali malo njih je zapisano u strojnom jeziku,

većina njih su makro naredbe koje koriste prije spomenute "ugrađene" funkcije i pomoću njih ekspandiraju tj. proširuju sintaksu. Makro naredbe kod Lispa ne treba miješati sa predprocesorskim makroima u na primjer C-u, daleko su moćniji i Lisp nakon što ih proširi ih i izvede, na taj način se proširuje sintaksa i uvodi pojam metaprogramiranja.

Liste i tipovi podataka

U Common Lispu je podržan rad sa cijelim, realnim i kompleksnim brojevima, razlomcima i drugim tipovima podataka. Neophodno je korištenje atoma i lista podataka koje se sastoje od *cons* čelija, zapravo se radi o povezanim listama (Slika 5.).



Slika 5.

Funkcije i makro naredbe su također definirane kao liste kod kojih je prvi element naziv funkcije, tj. Lisp prvi element liste pokušava evaluirati u funkciju. Funkcijama možemo proslijeđivati varijabilan broj parametara, neki od načina su preko *&key*, *&optional* i *&rest* simbola. Od podataka su tu još i *bit-array*, *hash* tablice, vektori i neki drugi. Common Lisp posjeduje mnoštvo funkcija za rad sa listama, poput *car* koji vraća prvi elemnt *cons* čelije i *cdr* koji vraća rep odnosno ostatak na koji cons čelija dalje pokazuje. Naravno tu je još mnoštvo funkcija za dohvaćanje drugih elemenata liste, njihovo mijenjanje, prikupljanje podataka iz listi (*map*/*reduce*), nekoliko izrazito moćnih petlji (*do, dotimes, dolist, loop* makro naredbe), funkcija uvjetnog granjanja (*cond*, *if*), pridruživanja varijabli (*let*, *setf*, *setvar*), poziva i evaluiranja drugih funkcija i tako dalje. A kako su i funkcije u Lispu zapravo liste, onda možemo pretpostaviti kako uz prije navedene funkcije i makro naredbe možemo efikasno proširivati sintaksu, generirati odnosno proširivati (*macro-expand*) naš programski kod.

Primjer programskog koda

Za primjer ćemo prikazati funkciju sortiranja jedinki populacije po ocjeni kvalitete (2.) Sa *defun* definiramo funkciju, naziv funkcije je *sort-population* i ne prima nikakve parametre, u suprotnom bi ih naveli između praznih zagrada iza naziva funkcije. Slijedi dokumentacijski tekst za funkciju koji se ne mora nužno navesti, ali je praktičniji nego ubacivanje običnih komentara. Funkcija *setf* postavlja dobivenu listu od *sort* funkcije u globalnu varijablu **programs**, u Lispu je uobičajeno da globalne varijable imaju prefiks i sufiks u obliku zvijezdice. Sama funkcija (zapravo makro naredba) *sort* za prvi parametar prima listu, u ovom slučaju listu jedinki u populaciji, drugi argument je poziv funkcije sa kojom želimo obaviti sortiranje, oznaka *#'* služi za pozivanje druge funkcije, u ovom slučaju "<" tj. manje. Konačno trećim argumentom dajemo funkciji *key* parametar koji poziva funkciju *second* koja za svaku jedinku u listi **programs** uzima drugi element jedinke odnosno njezinu ocjenu kvalitete po kojoj i želimo obaviti sortiranje.

```
(defun sort-population ()
   "Sorts programs in population by fitness in ascedencing
order."                                                    (2.)

  (setf *programs* (sort *programs* #'< :key #'second)))
```

Navedeni primjer ilustrira zanimljive mogućnosti lispa, tj. kako je u malo programskog koda moguće mnogo toga postići, ponajviše zbog njegove fleksibilnosti.

U Common Lisp je uveden i objektni pristup programiranju CLOS (Common Lisp Object System). Po preformansa se Common Lisp zbog svoje fleksibilnosti teško može mjeriti sa C-om ili nekim drugim jezikom niže klase. S druge strane se ni Java, .Net ili neki drugi programski jezik koji se prevodi u međukod (engl. *bytecode*) ne može mjeriti sa C-om. Možemo zaključiti da je Lisp u vrijeme u kojem se pojavio bio procesorski i memorijski prezahtjevan za ondašnja računala u usporedbi sa C-om i Fortranom, iako je u mnogo aspekata bio daleko ispred ostalih. Danas je Lisp prerastao u moderan programski jezik koji posjeduje mnoštvo dodatnih bibliotek i čija popularnost polagano raste, a mnogi drugi jezici uvode njegove osobine ili su inspirirani njime.

Ostvarenje

Struktura ostvarenja se može podijeliti na tri cjeline, prva je definiranje nekih početnih varijabli, funkcija i konstanta, druga je realizacija simulacije slijetanja letjelice, što nam je potrebno za ocjenu kvalitete pojedine jedinke, a treća je samo genetsko programiranje odnosno generiranje populacije, primjena genetskih operatora i evaluiranje generacije.

U tekstu će se spomenuti samo bitnije programske funkcije, svaka funkcija i parametar imaju dokumentacijski string u izvornom kodu tako da je jednostavno odrediti njihovu svrhu i značenje.

Odabir početnih parametara

Za rješavanje dotičnog problema je prvo trebalo odrediti neko početno stanje, točnije početnu visinu, brzinu i količinu goriva letjelice, gravitaciju, snagu potisnika, potrošnju goriva i slično (Tablica 1.). Neke vrijednosti su uzete iz dostupnih informacija o slijetanju Lunarnog modula prilikom misije Apollo.

Tablica 1.

Opis	Varijabla	Vrijednost
Gravitacija Mjeseca	*gravity*	1.622 m/s
Početna visina	*start-altitude*	150 m
Početna brzina	*start-velocity*	21 m/s
Masa letjelice	*start-mass*	7000 kg
Početna količina goriva	*start-fuel*	2000 kg
Maksimalni potisak	*max-thrust*	45,00 kn
Potrošnja goriva	*thruster-consumption*	25 kg/s
Vremenski period iteracije	*deltatime*	0.1 sec
Uvijet zaustavljanja (brzina)	*stop-fitness*	1 (m/s)

Osim parametara za simulaciju je bilo potrebno odrediti i početne parametre za rad genetskog programiranja, lista operatora i terminala (Tablica 2.), veličina populacije, dubina stabla, vjerovatnost križanja i mutacije.

Tablica 2.

Lista operatora (prima 2 argumenta)	+, -, *, safe-/
Lista operatora (prima 1 argument)	sin, safe-log, safe-sqrt, square
Lista terminala	fuel, velocity, altitude

Prilikom korištenja operatora moramo voditi računa o mogučnosti dijeljenja sa nulom, logaritmiranja negativnih vrijednosti i slično. Navedene mogučnosti su izbjegnute definiranjem vlastitih funkcija. Funkcija *safe-/* u slučaju dijeljanja sa nulom vraća vrijednost

jedan, *safe-log* parametar manji od 0.1 zaokružuje na 0.1, a *safe-sqrt* koristi apsolutnu vrijednost parametra kako bi se izbjeglo korijenovanje negativnog broja. Zanimljivo je kako Common Lisp zna baratati sa kompleksnim brojevima, tako da je druga mogućnost bila dobivanje kompleksnog rješenja i zatim uzimanje njegove aposlutne vrijednosti.

Ostvarenje simulacije

Za svaku dobivenu jedinku (program, funkciju) se izvodi simulacija spuštanja letjelice kako bi mogli na temelju rezultata odrediti kvalitetu jedinke. Ulazni parametri u simulaciju su stanje koje se sastoji od trenutne visine, brzine i količine goriva. Formula (3.) prikazuje trenutnu silu na letjelicu, odnosno razliku gravitacijske sile i sile potisnika. Sljedeće stanje evaluiramo na način da evaluiramo funkciju jedinke i iz nje dobijemo postotak paljenja potisnika [0, 1], zatim izračunamo trenutnu silu potisnika (4.) , na temelju toga odredimo potrošnju goriva (5.) , promjenu brzine (6.) uz pomoć formule (3.) i konačno promjenu visine letjelice (7.) . Jednu iteraciju simulacije provodimo u zadanom periodu na početku programa, konkretno 0.1 sekunda. Simulacija završava ukoliko visina postane manja ili jednaka nuli, ili ako letjelica ostane bez goriva.

$$G - F = m * a = m * \frac{v}{t} \qquad (3.)$$

$$force = ignition * maxThrust \qquad (4.)$$

$$nextFuel = fuel - (ignition * thrusterConsumption * deltaTime) \qquad (5.)$$

$$nextVelocity = velocity + \frac{(mass * gravity) - force}{mass} * deltaTime \qquad (6.)$$

$$nextAltitude = altitude - (velocity * deltaTime) \qquad (7.)$$

Evaluacija funkcije paljenja je ostvarena korištenjem ugrađene funkciju *eval* u Common Lispu, navedena funkcija izvodi odnosno evaluira listu koju joj proslijedimo kao parametar.

Realizacija genetskog programiranja počinje od generiranja inicijalne populacije, nasumično se odabere operator i zatim se rekurzivno poziva funkcija za njegovo lijevo i desno podstablo, pritom se vodi računa da neki operatori primaju samo jedan argument (npr. sinus), konačno se zaustavllja kada dubina stabla dostigne zadano.

Ostvarenje Genetskog programiranja

Jedinka se sastoji od liste u kojoj je prvi element funkcija, a drugi ocjena kvalitete, a populacija je lista jedinki (8.) .

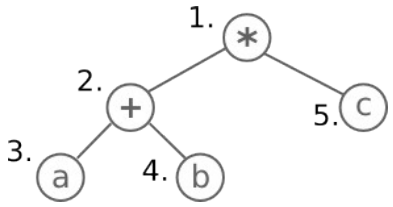
```
(((+ a (- b c)) 10)
 ((* (+ a b) (- c d)) 20)
 ...
)
```
(8.)

Ocjenu kvalitete određujemo s obzirom na stanje parametara letjelice, najviše nas zanima da letjelica što "ugodnije" dotakne površinu, tj. da je u tom trenutku brzina što manja, zato

uzimo da je idealan slučaj kada je brzina jednaka nuli. To je ujedno i najbolja ocjena, najgora ocjena je 100 koju možemo dobiti ukoliko je letjelica ostala bez goriva ili nije sletjela. Dakle ocjene skaliramo na interval [0, 100] gdje je 0 najbolja ocjena. U sustav ocjenjivanja smo mogli uključiti i druge parametre poput količine goriva, na primjer da ga se što više potroši ili opet što manje.

Populaciju prilikom svakog evaluiranja kvalitete jedinki sortiramo uzlazno, tako da su nam one jednike sa boljom ocjenom kvalitete uvijek na početku liste, to kasnije olakšava odabir najboljih i izbacivanje najlošijih jedinki.

Za primjenu genetskih operatora su definirane neke pomoćne funkcije poput prebrojavanja čvorova u stablu (*tree-size*) i dohvaćanja podstabla po indeksu (*subtree-nth*). Čvorove u stablu brojimo na način da prvo uzimamo lijevo podstablo i idemo prema dubini, dakle s lijeva na desno, prvo po dubini (Slika 6.).



Slika 6.

Genetski operator križanja je ostvaren kroz funkciju *gp-crossover*, za parametre prima dva indeksa programa u listi jedinki odnosno populaciji. Slučajnim odabirom se odabire podstablo u jednom i drugom programu. Navedeni programi se dupliciraju zbog destruktivne prirode križanja, tj. kako bi se roditelji očuvali. Podstabla se zamijene i tako dobivamo dva nova programa, tj. potomstvo. Ubacujemo ih na početak liste i evaluiramo njihove ocjene kvalitete.

Funkcija *gp-mutation* ostvaruje genetski operator mutacije, za parametar također prima indeks programa u listi jedinki, slučajnim odabirom mijenja jedan čvor u programu, pritom provjeravajući radi li se o operatoru ili terminalu. Na kraju se ponovo evaluira ocjena kvalitete programa.

Samo genetsko programiranje je ostvareno sa funkcijom *test-gp* koja opcionalno može primiti parametar veličine populacije, inače uzima pretpostavljenu vrijednost od 30 jediniki. Stvara se inicijalna populacija i evaluiraju se ocjene kvalitete jedinki. Zatim se pokreće beskonačna petlja kojoj je uvijet zaustavljanja da najbolja jednika postigne ocjenu kvalitete manju od 1, tj. da je brzina letjelice prilikom dodira sa površinom manje od 1 m/s. Prvo se primjenjuje operator križanja na prve dvije jedinke, ujedno i dvije najbolje jednike. Izbacuju se posljednje dvije jednike, ujedno najlošije jednike. Na taj način održavamo količinu populacije konstantnom. Još prvodimo operator mutacije na svakoj jedinki sa vjerovatnošću od 1%. Konačno sortiramo jedinke u populaciji s obzirom na ocjenu kvalitete. Navedeni postupak ponavljamo sve dok ne zadovoljimo uvijet zaustavljanja.

Testovi i rezultati

Testiranje je provedeno za različite dubine početkog stabla i različitu inicijalnu populaciju. Inicijalne dubine stabla su redom 3, 4 i 5 čvorova, za svaku dubinu je napravljen test sa populacijom od 10, 25 i 50 jedinki. Svako testiranje je provedeno 20 puta. Neki testovi nisu dali odgovarajući rezultat u razumnom vremenu (~20 sec), stoga nisu uvršteni u izračun srednje vrijednosti svakog testa, ali je naveden broj uspješnih testova.

Dubina stabla	Veličina populacije	Broj uspješnih testova	Prosječan broj generacija	Prosječno trajanje testa (sec)
3	10	17	204,8	10,9
3	25	15	11,3	0,9
3	50	19	15,8	1,5
4	10	14	31,2	1,1
4	25	19	18,9	3,6
4	50	18	6,3	10,4
5	10	12	27,3	1,5
5	25	17	22,4	1,4
5	50	20	11,3	1,2

Iz rezultata se može zaključiti kako su se prilikom veće inicijalne dubine stabla dobili bolji rezultati, tj. GP je prije došao do zadovoljavajućeg rezultata. Također je sa većom populacijom prije GP dolazio do zadovoljavajućeg rezultata, što je za očekivati jer veća inicijalna populacija daje veću šansu da se odmah pogodi zadovoljavajuće rješenje. Kako je problem relativno jednostavan i kako genetski operator križanja uvijek uzima prve, ujedno i najbolje, dvije jedinke sama veličina populacije ne utječe mnogo na daljnji ishod GP-a, već mu samo u startu daje veće šanse.

Prosječno trajanje pojedinog testa nije dalo previše iskoristive podatke, glavni razlog je što duljina trajanja ne ovisi samo o broju iteracija, već i o veličini podstabla koje zamijenjujemo prilikom operatora križanja, a kako je odabir slučajan tako je i prosječno trajanje testa nekonzistentno.

Zaključak

Kako je seminar imao dvije svrhe, upoznavanje sa Common Lispom kroz primjer Genetskog programiranja, tako je i zaključak dvosmislen.

Common Lisp je svakako jezik koji je i više nego dobro upoznati i savladati, odmah možemo reći kako ga je lakše upoznati nego savladati. Riječ je modernom dijalektu Lispa koji ima svoju budućnost, uveo je mnoge zanimljive principe u programiranje koje neki današnji znatno raširenij jezici tek počinju uvoditi. Iz tog razloga je poznavanje Lispa izuzetno dobro zbog boljeg razumijevanja i drugih jezika i koncepata programiranja uopće.

Svakako bi napomenuo kako je na Internetu relativno teško pronaći primjere (Common) Lisp programskog koda, pogotovo na području genetskog programiranja, točnije prilikom pisanja seminara sam naišao samo na implementaciju genetskog programiranja od J.R. Koze koji se intezivno bavio tim područjem.

Drugi cilj seminara je bio dobivanje optimalne funkcije paljenja potisnika kod *Moonlandera* genetskim programiranjem. Problem sam po sebi nije odviše kompleksan tako da je upotreba genetskog programiranja upitna, no svakako je dobar "školski" primjer za rad sa GP-om. Problem bi se mogao dodatno zakomplicirati uvođenjem dodatnih parametara u ocjenu kvalitete, poput količine preostalog goriva i slično. Također bi bilo poželjno ostvariti kompleksniju tehniku odabira jedinki za križanje, trenutna tehnika samo uzima najbolje jedinke što u praksi ne mora davati najbolji rezultat. No i bez dodatnog kompliciranja ostvareni algoritam je u razumnom vremenu dao željene rezultate.

Literatura

Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D. (1998), *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its*

Peter Seibel (2005), *Practical Common Lisp*

David S. Touretzky (1989), *Common Lisp: A Gentle Introduction to Symbolic Computation*

(Listopad 2009), http://www.lispworks.com/documentation/HyperSpec/Front/index.htm

(Listopad 2009), http://en.wikipedia.org/wiki/Common_lisp

(Listopad 2009), http://en.wikipedia.org/wiki/Lisp