SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE U RAČUNARSTVA

DIPLOMSKI RAD br. 1042

STROGO TIPIZIRANO I SEMANTIČKO GENETSKO PROGRAMIRANJE

Branimir Gregov

Zagreb, lipanj 2015.

Sadržaj

1. Uvod

Čovjek svakodnevno rješava razne probleme s kojima se i najjača računala danas muče. To je zato što računala donose odluke drugačije nego ljudi. Računala su u svojoj naravi deterministička, za razliku od ljudi čiji je način razmišljanja probabilistički. Mi „odvagujemo“ odluke i, potpomognuti iskustvom, biramo onu za koju je najizglednije da će ispasti dobra.

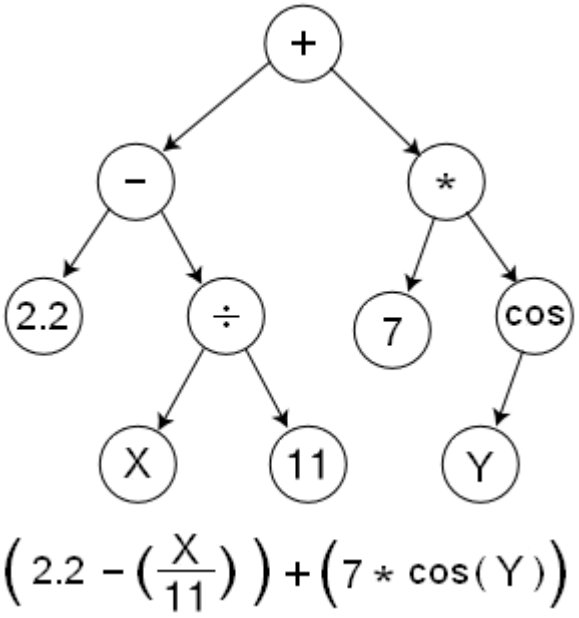
Tu u igru dolazi evolucijsko računarstvo. Ono računalima približava procese iz prirode – isprobavanje, učenje, evolucija. Računalo uči iz iskustva i postiže sve bolja i bolja rješenja koja nisu nužno optimalna, ali su zadovoljavajuća. Genetsko programiranje je posebna grana evolucijskog računarstva. Pomoću genetskog programiranja računalo uči i razvija određeni program ili matematičku funkciju. Svako moguće rješenje (jedinka) je predstavljeno u obliku stablaste strukture. Jedan mogući primjer prikazan je na slici 1.1.

Cilj bilo kojeg evolucijskog algoritma pa tako i genetskog programiranja je da pomoću „Darwinovih operatora“ (križanja, mutacije i selekcije) uzima „bolja“ rješenja i iz njih izgrađuje nova rješenja za koja se nadamo da će biti još bolja, a ona lošija odbacuje. Pritom je riječ *bolja* pod navodnicima jer je ponekad prilično teško definirati što uopće znači dobro rješenje, ali to je izvan opsega ovog rada.

Ono čime ćemo se baviti u ovom radu su strogo tipizirano i semantičko genetsko programiranje, a ono je specifično u odnosu na klasično genetsko programiranje zbog strožih pravila pri križanju i mutaciji (i, naravno, pri izgradnji početnih rješenja). Poanta je kako bi ti operatori trebali izgraditi rješenja koja slijede određena pravila. Strogo tipizirano genetsko programiranje nam omogućuje da jedinku, tj. stablo izgradimo pomoću čvorova koji primaju parametre različitih tipova i imaju drugačije povratne vrijednosti. Npr. možemo imati čvor koji kao ulazne parametre prima vrijednosti tipa *boolean, integer* i *double*, a vraća vrijednost *double*. Jedan takav mogući čvor je programska funkcija

IF (X > 0) RETURN Y^X ELSE RETURN Y^{-X},

gdje je X > 0 predstavlja ulaznu varijablu tipa *boolean*, Y je tipa *double*, X je tipa *integer*, a povratna vrijednost te funkcije je tipa *double*. To ne bi mogli prikazati sa klasičnim genetskim programiranjem jer bi se moglo dogoditi da umjesto ulazne vrijednosti koja mora biti *boolean* dobijemo neku koja je npr. cjelobrojnog tipa i tada ne bi mogli odlučiti hoće li se izvršiti jedna ili druga grana tog čvora. Strogo tipizirano genetsko programiranje pazi da su svi tipovi podataka ispravni prilikom križanja i mutacije.



Slika 1.1 Stablasta reprezentacija matematičke funkcije sa dvije varijable[1]

Semantičko genetsko programiranje postavlja neka druga ograničenja na moguća rješenja. Ono pokušava generirati rješenja koja su semantički ispravna u matematičkom smislu, tj. uvodi mjerne jedinice koje pokušava poštivati pri izvođenju matematičkih operacija (zbrajanja, oduzimanja, množenja, dijeljenja, kvadriranja, itd.). Npr. kod zbrajanja i oduzimanja je potrebno imati ulazne parametre istih

[1] Izvor: *http://en.wikipedia.org/wiki/Genetic_programming*

mjernih jedinica (kako bi inače zbrojili npr. metar i sekundu) i pritom će izlazni parametar imati istu mjernu jedinicu, dok je kod množenja i dijeljenja dozvoljeno imati ulazne parametre različitih mjernih jedinica koji će pritom dati izlazni parametar s nekom novom mjernom jedinicom.

Cilj ovog rada nije ispitivanje strogo tipiziranog i semantičkog genetskog programiranja jer su te teme već obrađene u drugim radovima. Cilj ovog rada je implementacija tih tehnika u razvojno okruženje za rad s evolucijskim računarstvom (*ECF*, engl. *Evolutionary Computational Framework*. ECF je napravljen kako bi olakšao, prvenstveno studentima, a onda i svima ostalima, isprobavanje raznih tehnika evolucijskog računarstva. Konstantno je proširivan sa novim i isprobanim tehnikama koje se onda kasnije mogu koristiti s nekim novima u svrhu testiranja i istraživanja.

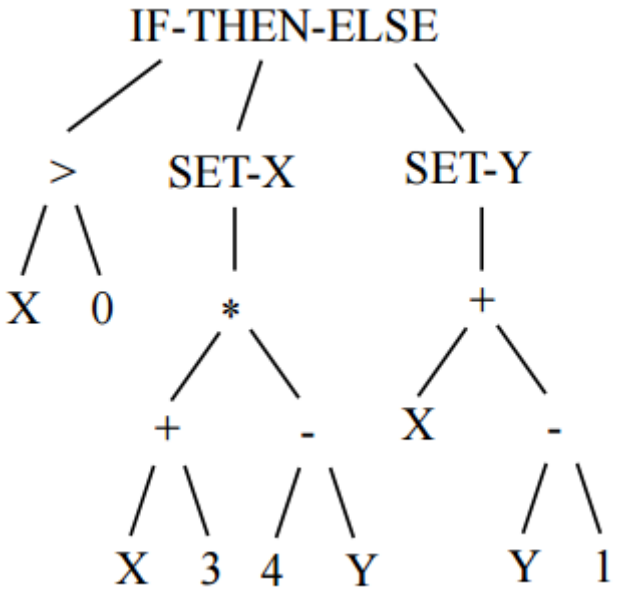
U spomenutom razvojnom okruženju već postoji implementiran genotip stabla, kao i razni operatori križanja i mutacije, ali i gotovi algoritmi koji su neovisni o korištenom genotipu. U ovom radu su napravljena dva nova genotipa:

1) Strogo tipizirano stablo (u kôdu nazvano *STTree*)
2) Semantičko stablo (u kôdu nazvano *SemTree*)

zajedno sa pripadnim operatorima križanja i mutacije koji su prilagođeni za rad sa tim genotipovima. Ti operatori u programskom kôdu imaju prefiks *ST* odnosno *Sem*, ovisno o genotipu za koji su namijenjeni.

2. Strogo tipizirano genetsko programiranje

Strogo tipizirano genetsko programiranje unosi raznolikost tipova podataka s kojima evolucijski algoritam može raditi. Konkretno za stablaste strukture koje se koriste u genetskom programiranju, dodaje mogućnost korištenja čvorova koji nisu svi istog tipa (slika 2.1.). U ovom radu je navedena funkcionalnost implementirana na način da su svi mogući primitivi, koje čvorovi koriste za izračun vrijednosti, razvrstani u skupine ovisno o tipu povratne vrijednosti (*integer, double, boolean, character* ili *string*). Jednom kada je taj dio odrađen, dalje se stvar svodi na jednostavno biranje nasumičnog čvora iz željene skupine.



Slika 2.1. Primjer stabla sa različitim tipovima čvorova[2]

[2] Izvor: Montana, D. J. *Strongly Typed Genetic Programming*, Cambridge 2002.

Za korisnika koji će koristiti *ECF* nema puno razlike u odnosu na klasično genetsko programiranje. Samo je potrebno nadgraditi metode koje definiraju tipove ulaznih parametara i povratne vrijednosti te u datoteci s parametrima pod genotip navesti *STTree* umjesto *Tree*.

2.1. Implementacija *STGP*-a u *ECF*-u

Prvi dio koji nam je potreban za implementaciju *STGP*-a već postoji u trenutnoj implementaciji *ECF*-a. Riječ je o definiranju tipova terminalnih vrijednosti u *XML* datoteci s parametrima.

Sljedeći i ključni dio cijele implementacije se nalazi u zaglavlju klase koja predstavlja skup primitiva korištenih u genotipu (slika 2.1.1.).

```
PrimitiveP getRandomTerminal();
PrimitiveP getRandomFunction();
PrimitiveP getRandomPrimitive();

PrimitiveP getTerminalByName(std::string name);
PrimitiveP getFunctionByName(std::string name);
PrimitiveP getPrimitiveByName(std::string name);
```

Slika 2.1.1. Isječak kôda iz datoteke PrimitiveSet.h

U prikazanom isječku kôda vidimo definicije funkcija koje se koriste u algoritmima izgradnje, mutacije i križanja stabala kada je potrebno dohvatiti čvorove određenog tipa. U trenutno implementiranom genotipu moguće je birati samo između terminala (*getRandomTerminal* / *getTerminalByName*), funkcija (*getRandomFunction* / *getFunctionByName*) ili primitiva (*getRandomPrimitive* / *getPrimitiveByName*) koje uključuju i terminale i funkcije.

Pri implementaciji klase *STPrimitiveSet* koju koristi naš novi genotip *STTree* gore navedene funkcije su definirane kako je prikazani na slici 2.1.2.

```
STPrimitiveP getRandomTerminalOfType(STPrimitives::terminal_type terminalType);
STPrimitiveP getRandomFunctionOfType(STPrimitives::terminal_type terminalType);
STPrimitiveP getRandomPrimitiveOfType(STPrimitives::terminal_type terminalType);

STPrimitiveP getTerminalByNameOfType(std::string name, STPrimitives::terminal_type terminalType);
STPrimitiveP getFunctionByNameOfType(std::string name, STPrimitives::terminal_type terminalType);
STPrimitiveP getPrimitiveByNameOfType(std::string name, STPrimitives::terminal_type terminalType);
```

Slika 2.1.2. Isječak kôda iz datoteke STPrimitiveSet.h

Na slici se jasno vidi razlika u odnosu na prethodnu – kao ulazni parametar svake od navedenih funkcija dodan je još *terminalType* koji se ne odnosi isključivo na terminale nego i na ostale primitive, ali budući da je za terminale već bio definiran taj tip, koristili smo ga i na drugim mjestima zbog unazadne kompatibilnosti[3]. Zbog lakšeg spremanja i dohvaćanja različitih tipova primitiva, odlučili smo za njih koristiti različite spremnike ovisno o njihovom tipu, tj. tipu povratnog parametra tog primitiva.

Sljedeći dio koji je trebalo implementirati je izgradnja novog stabla. Budući da smo već podijelili primitive po tipovima, izgradnja strogo tipiziranog stabla (slika 2.1.4.) se ne razlikuje previše od izgradnje klasičnog stabla (slika 2.1.3.).

Usporedbom te dvije funkcije vidimo da je jedina razlika u izboru primitiva koje moraju imati povratni parametar istog tipa kao i ulazni parametar roditeljskog čvora.

[3] engl. *backwards compatibility*

```
uint Tree::growBuild(PrimitiveSetP primitiveSet, int myDepth)
{
    Node* node = new Node();
    node->depth_ = myDepth;

    if(node->depth_ < this->initMinDepth_) {
        node->setPrimitive(primitiveSet->getRandomFunction());
        this->addNode(node);
    }
    else if(node->depth_ < this->initMaxDepth_) {
        node->setPrimitive(primitiveSet->getRandomPrimitive());
        this->addNode(node);
    }
    else {
        node->setPrimitive(primitiveSet->getRandomTerminal());
        this->addNode(node);
    }

    for(int i = 0; i < node->primitive_->getNumberOfArguments(); i++) {
        node->size_ += growBuild(primitiveSet, myDepth + 1);
    }

    return node->size_;
}
```

Slika 2.1.3. Rekurzivna funkcija izgradnje novog stabla

```
uint STTree::growBuild(PrimitiveSetP primitiveSet, int myDepth, STPrimitives::terminal_type wantedTT)
{
    STNode* node = new STNode();
    node->depth_ = myDepth;

    if (node->depth_ < this->initMinDepth_ && primitiveSet->getFunctionSetSizeOfType(wantedTT) > 0) {
        node->setPrimitive(primitiveSet->getRandomFunctionOfType(wantedTT));
        this->addNode(node);
    }
    else if (node->depth_ < this->initMaxDepth_ && primitiveSet->getPrimitivesSizeOfType(wantedTT) > 0) {
        node->setPrimitive(primitiveSet->getRandomPrimitiveOfType(wantedTT));
        this->addNode(node);
    }
    else {
        node->setPrimitive(primitiveSet->getRandomTerminalOfType(wantedTT));
        this->addNode(node);
    }

    for(int i = 0; i < node->primitive_->getNumberOfArguments(); i++) {
        node->size_ += growBuild(primitiveSet, myDepth + 1, node->primitive_->getArgumentType(i));
    }

    return node->size_;
}
```

Slika 2.1.4. Rekurzivna funkcija izgradnje novog strogo tipiziranog stabla

S time zaključujemo bazu genotipa strogo tipiziranog stabla. Još samo ostaje za implementirati operatore križanja i mutacije koje korisnik želi. U ovom radu je od operatora križanja implementirano samo jednostavno križanje koje najprije odabere nasumični čvor iz prvog roditelja, a zatim i nasumični čvor drugog roditelja koji je istog tipa kao prvo odabrani čvor te ih zamijeni zajedno sa odgovarajućim podstablima. Jedini dio ovog operatora koji se razlikuje od već implementiranog operatora križanja je kôd koji provjerava koje tipove čvorova sadrže oba stabla i zatim nasumično bira jednog od njih (slika 2.1.5.).

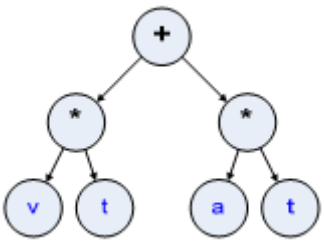
```
std::set<STPrimitives::terminal_type> mtt, ftt;
std::vector<STPrimitives::terminal_type> intersection;
for (uint i = 0; i < mRange; i++)
{
    mtt.insert(male->at(i)->primitive_->getReturnType());
}
for (uint i = 0; i < fRange; i++)
{
    ftt.insert(female->at(i)->primitive_->getReturnType());
}
std::set_intersection(mtt.begin(), mtt.end(), ftt.begin(),
    ftt.end(), std::back_inserter(intersection));

STPrimitives::terminal_type chosenTT = intersection[
    state_->getRandomizer()->getRandomInteger(intersection.size())];
```

Slika 2.1.5. Isječak kôda koji pronalazi tipove primitiva sadržanih u oba stabla

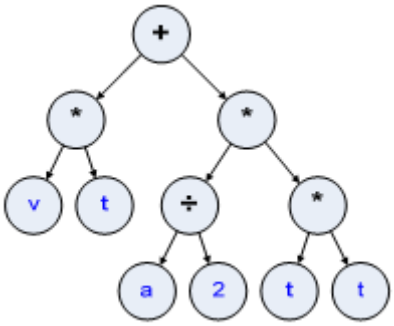
Na sličan način su implementirani i operatori mutacije. U ovom radu to su zamjena čvora komplementarnim čvorom, zamjena čvora nasumičnim čvorom te zamjena podstabla s novo generiranim podstablom.

3. Semantičko genetsko programiranje

Semantičko genetsko programiranje se u okvirima ovog rada koristi za evoluciju izraza koji moraju poštivati mjerne jedinice. To je veoma korisno jer stvari koje čovjek kvantizira ne predstavljaju apsolutno ništa bez mjernih jedinica. Kako bi se problemi lakše predstavili računalu, mjerne jedinice su često izostavljene iz opisa samog problema. Računalo će tada evoluirati izraze koje su vrlo precizni u procjeni rješenja, ali to rješenje gotovo sigurno neće imati ispravnu mjernu jedinicu. Npr. računalo će zbrajati masu i vrijeme ili nešto još više nespojivo. Iz tog razloga nam je potrebno semantičko genetsko programiranje.



Slika 3.1. Semantički neispravno stablo (izraz: $v * t + a * t$)[4]



Slika 3.2. Semantički ispravno stablo (izraz: $v * t + {}^{a}/_{2} * t^{2}$)[5]

[4] Izvor: Dragoljević O. *Semantičko genetsko programiranje*, FER 2008.
[5] Izvor: Dragoljević O. *Semantičko genetsko programiranje*, FER 2008.

Semantičko genetsko programiranje je teže za implementirati unutar *ECF*-a nego što je to slučaj sa strogo tipiziranim genetskim programiranjem. Daleko je teže kreirati novo stablo koje ima unaprijed zadanu mjernu jedinicu korijenskog čvora jer tada moramo pažljivije birati čvorove koje ćemo dodavati stablu. Pritom ne moramo paziti da biramo čvorove koje imaju određeni tip povratne vrijednosti jer u trenutnoj implementaciji svi čvorovi primaju i vraćaju vrijednosti tipa *double*, ali moramo paziti da izaberemo čvorove koji će u svakoj razini stabla promijeniti mjernu jedinicu na takav način da u zadnjoj razini postoje samo terminali sa najjednostavnijom mogućom mjernom jedinicom (ili bez mjerne jedinice ili samo jedna mjerna jedinica sa eksponentom 1, npr. *s* (sekunda) ili *g* (gram), ali nikako složene jedinice poput s^2 ili $g * m^3$).

3.1. Implementacija semantičkog GP-a u *ECF*-u

Prva stvar za koju smo se morali odlučiti pri implementaciji semantičkog GP-a je prikaz mjernih jedinica pojedinog čvora. *N* mjernih jedinica (za primjer genetskog algoritma sa mjernim jedinicama gram, metar i sekunda *N*=3) predstavljeno je vektorom veličine *N* u kojem svaka komponenta predstavlja eksponent mjerne jedinice. Za navedeni primjer sa mjernim jedinicama gram, metar i sekunda, jedan mogući vektor mjernih jedinica bio bi (4, -2, 3) što bi predstavljalo $g^4m^{-2}s^3$.

Kao što smo rekli, implementacija semantičkog GP-a je daleko teža od strogo tipiziranog GP-a jer je daleko teže konstruirati novo semantički ispravno stablo nego je to slučaj sa strogo tipiziranim stablom. Izgradnja semantički ispravnog stabla odozdo prema gore je skoro pa nemoguća jer ne možemo čvorove spajati po volji, a izgradnja stabla odozgo prema dolje je prilično kompliciranа pa je u prvoj iteraciji izgradnja semantičkog stabla bila implementirana na način da se stablo gradi odozgo prema dolje te da stablo ne mora nužno biti semantički ispravno, ali će se ta neispravnost kažnjavati na sljedeći način – križanje i mutacija će se ponoviti

nekoliko puta umjesto samo jednom, a zatim će se kao rezultirajuća jedinka tih operacija uzeti ona koja je semantički „najispravnija“. Tu (ne)ispravnost je definirao Keijzer na sljedeći način[6]: ako *x* predstavlja vektor jedinica danog čvora, a *y* predstavlja željeni vektor jedinica koji bi očuvao semantiku stabla, tada je „ispravnost“ definirana izrazom $\sum_{i=1}^{N} |x_i - y_i|$.

Budući da opisani postupak ne daje stabla koja su potpuno semantički ispravna, funkcija za izgradnju stabla pisana je iznova. Stablo se i dalje gradi odozgo prema dolje, ali ovog puta je svjesno mjernih jedinica. Potrebno je zadati željenu mjernu jedinicu vršnog čvora, a algoritam izgradnje stabla će pametno birati čvorove na način koji će garantirati da terminali imaju jednostavne mjerne jedinice kao što je spomenuto u poglavlju 3. Najbitniji dio tog algoritma je onaj koji bira funkciju sljedećeg čvora (slika 3.1.1.). Skup funkcija s kojima ovaj algoritam raspolaže trenutno je ograničeno na zbrajanje, oduzimanje, množenje i dijeljenje. Iz kôda vidimo da algoritam provjerava na kojoj dubini stabla se trenutni čvor nalazi i pritom pokušava za sljedeći čvor odabrati funkciju koja će smanjiti najveću potenciju po apsolutnoj vrijednosti među mjernim jedinicama.

Jednom kada je odabrana funkcija, potrebno je odrediti mjerne jedinice čvorova djece. Budući da smo ograničili skup funkcija kojima algoritam raspolaže, sigurni smo da je broj odabrane djece jednak 2. Isječak kôda na slici 3.1.2. pokazuje kako se mjerna jedinica raspoređuje na to dvoje djece. Iz kôda vidimo da su moguća dva slučaja:

1) Funkcija roditeljskog čvora je ili zbrajanje ili oduzimanje
2) Funkcija roditeljskog čvora je ili množenje ili dijeljenje

Ukoliko se radi o prvom slučaju, mjerne jedinice djece moraju biti identične mjernoj jedinici roditeljskog čvora jer je za zbrajanje i oduzimanje potrebno imati argumente

[6] Izvor: Keijzer M., Babovic V. *Dimensionally Aware Genetic Programming*, Hørsholm 1999.

istih mjernih jedinica. Ako se pak radi o drugom slučaju, algoritam pokušava ravnomjerno rasporediti mjerne jedinice na djecu.

```
SemPrimitiveP SemPrimitiveSet::getRandomFunctionByUnits(SemNode *node, SemTree *tree)
{
    int numOfNonZero = 0;
    for (uint i = 0; i < node->currentUnit.size(); i++)
    {
        if (node->currentUnit[i] != 0)
            numOfNonZero++;
    }

    if (numOfNonZero == 1) {
        int upper = tree->getUpperBound(node);
        int lower = tree->getLowerBound(node);

        int exp = 0;
        for (uint i = 0; i < node->currentUnit.size(); i++)
        {
            if (node->currentUnit[i] != 0)
            {
                exp = (int)(node->currentUnit[i]);
                break;
            }
        }

        if (exp < lower) {
            return getFunctionByName("/");
        }
        else if (exp > upper) {
            return getFunctionByName("*");
        }
        else {
            return getRandomFunction();
        }
    }
    else {
        int positiveUnitCounter = 0;
        int negativeUnitCounter = 0;
        for (uint i = 0; i < node->currentUnit.size(); i++)
        {
            if (node->currentUnit[i] > 0) positiveUnitCounter++;
            if (node->currentUnit[i] < 0) negativeUnitCounter++;
        }

        if (positiveUnitCounter == numOfNonZero) {
            return getFunctionByName("*");
        }
        else {
            return getFunctionByName("/");
        }
    }
}
```

Slika 3.1.1. Isječak kôda koji bira funkciju sljedećeg čvora

```
std::string funcName = function->getName();
if (funcName == "+" || funcName == "-") {
    newNeededUnits.push_back(parentUnitsCopy);
    newNeededUnits.push_back(parentUnitsCopy);
}
else {
    std::vector<double> first, second;
    for (uint i = 0; i < parentUnitsCopy.size(); i++) {
        first.push_back(0);
        second.push_back(0);
    }

    for (uint i = 0; i < parentUnitsCopy.size(); i++) {
        if (parentUnitsCopy[i] == 0) continue;

        int firstExp = (int)parentUnitsCopy[i] / 2;
        int secondExp = (int)parentUnitsCopy[i] - firstExp;

        if (abs((int)parentUnitsCopy[i]) > 1) {
            if (funcName == "/")
                secondExp *= -1;

            first[i] += firstExp;
            second[i] += secondExp;
        }
        else if (abs((int)parentUnitsCopy[i]) == 1) {
            if (parentUnitsCopy[i] == 1) {
                first[i] += secondExp;
                second[i] += firstExp;
            }
            else {
                if (funcName == "/")
                    secondExp *= -1;

                first[i] += firstExp;
                second[i] += secondExp;
            }
        }
    }

    newNeededUnits.push_back(first);
    newNeededUnits.push_back(second);
}
```

Slika 3.1.2. Isječak kôda koji definira mjerne jedinice djece zadanog čvora

4. Primjeri korištenja i rezultati

4.1. Strogo tipizirano genetsko programiranje

Za demonstraciju korištenja novoimplementirane funkcionalnosti strogo tipiziranog genetskog programiranja koristit ćemo dva primjera sa simboličkom regresijom koja se razlikuju samo po matematičkom izrazu kojeg evoluiraju. Za klasično stablo su napisani posebni primitivi koji rade samo sa cjelobrojnim vrijednostima. Na slici 4.1.1. vidimo primjer primitiva koji zbraja 2 argumenta.

```
class Add: public Tree::Primitives::Primitive
{
public:
    Add()
    {
        nArguments_ = 2;
        name_ = "ADD";
    }

    void execute(void* result, Tree::Tree& tree)
    {
        uint32_t& res = *(uint32_t*)result;
        uint32_t arg1, arg2;

        getNextArgument(&arg1, tree);
        getNextArgument(&arg2, tree);

        res = arg1 + arg2;
    }

    ~Add()
    {   }
};
```

Slika 4.1.1. Primitiv klasičnog stabla koji zbraja dva cjelobrojna argumenta

Ovaj pristup je dobar ukoliko su svi argumenti istog tipa, ali ako želimo npr. zbrojiti dva argumenta od kojih je jedan cjelobrojan, a drugi s pomičnim zarezom trebamo koristiti strogo tipizirano genetsko programiranje. Na slici 4.1.2. je implementiran primitiv kojemu je prvi argument cjelobrojnog tipa, a drugi argument i povratna vrijednost s pomičnim zarezom.

```
class Add_ID_D : public STTree::STPrimitives::STPrimitive
{
public:
    Add_ID_D()
    {
        nArguments_ = 2;
        name_ = "ADD_ID_D";
    }

    void execute(void* result, STTree::STTree& tree)
    {
        double& res = *(double*)result;
        int arg1;
        double arg2;

        getNextArgument(&arg1, tree);
        getNextArgument(&arg2, tree);

        res = arg1 + arg2;
    }

    ~Add_ID_D()
    {   }

    STTree::STPrimitives::terminal_type getReturnType()
    {
        return STTree::STPrimitives::Double;
    }

    STTree::STPrimitives::terminal_type getArgumentType(int argInd)
    {
        if (argInd == 0) return STTree::STPrimitives::Int;
        else if (argInd == 1) return STTree::STPrimitives::Double;
        else                  throw("ADD_ID_D::getReturnType wrong argInd!");
    }
};
```

Slika 4.1.2. Primitiv strogo tipiziranog stabla koji zbraja argumente različitih tipova

Ključan dio pisanja primitiva za strogo tipiziranog stabla su virtualne metode koje treba nadgraditi – *getReturnType* i *getArgumentType*. One u konkretnom primjeru osiguravaju da je prvi argument cjelobrojnog tipa, drugi argument s pomičnim zarezom, ali i da će roditeljski čvor ovog primitiva zatražiti vrijednost s pomičnim zarezom. Na sličan način je potrebno implementirati navedene metode svih ostalih primitiva.

Za usporedbu rezultata koristit ćemo primjer simboličke regresije pomoću klasičnog stabla te pomoću strogo tipiziranog stabla. Evaluacija stabla se vrši na

način da se najprije definiraju domena i kodomena. U našem primjeru domena se sastoji od svih parnih x-eva u intervalu $[2, 20]$, a preslikavanje domene u kodomenu je definirano kao funkcija $f(x) = x^2 + 3x - {}^{x}/_{4}$. Nakon što smo definirali domenu i kodomenu, uzmemo već izgrađeno stablo, zamijenimo vrijednosti terminala sa vrijednostima iz domene, izračunamo matematički izraz koji predstavlja dano stablo te vrijednost koju smo dobili usporedimo sa vrijednostima iz kodomene. Dobrota stabla je tim manja što se dobivene vrijednosti više razlikuju od onih iz kodomene.

Za izgradnju klasičnog stabla korištene su funkcije zbrajanja, množenja te rotacije i posmaka po bitovima ulijevo ili udesno. Sve od navedenih funkcija primaju i vraćaju cjelobrojne vrijednosti. Za izgradnju strogo tipiziranog stabla korištene su sve funkcije navedene za klasično stablo uz dodatak funkcija koje zbrajaju i množe dva argumenta s pomičnim zarezom i vraćaju vrijednost s pomičnim zarezom te funkcije koje zbrajaju ili dva cjelobrojna argumenta ili jedan cjelobrojni i jedan argument s pomičnim zarezom, a vraćaju vrijednost s pomičnim zarezom.

Tablica 4.1.1. Skup funkcija korištenih za izgradnju klasičnog stabla

Ime funkcije	Tip prvog argumenta	Tip drugog argumenta	Tip povratne vrijednosti
Zbrajanje (ADD)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Množenje (MUL)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Rotacija ulijevo (RL)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Rotacija udesno (RR)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Posmak ulijevo (SL)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Posmak udesno (SR)	Cjelobrojni	Cjelobrojni	Cjelobrojni

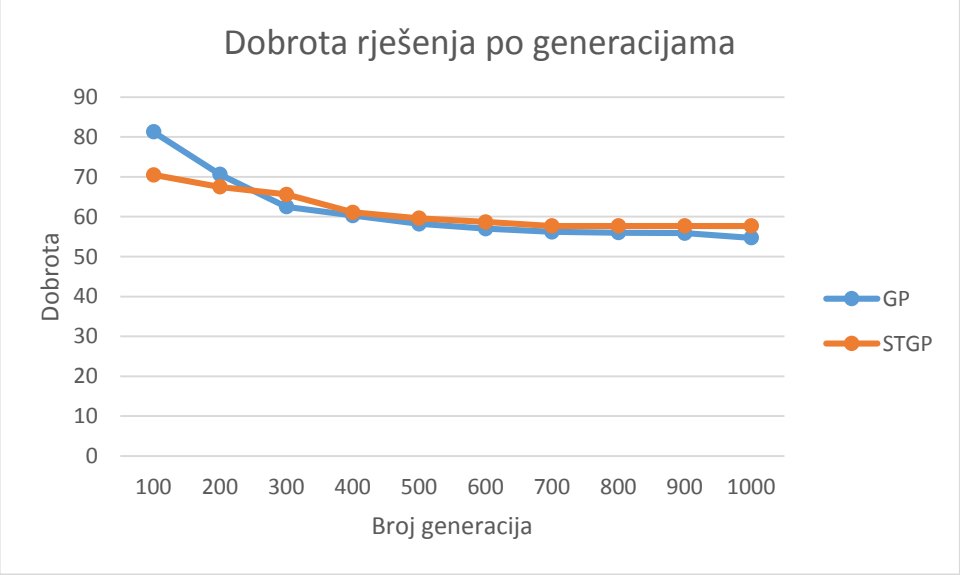
Tablica 4.1.2. Skup funkcija korištenih za izgradnju strogo tipiziranog stabla

Funkcija (ime)	Tip prvog argumenta	Tip drugog argumenta	Tip povratne vrijednosti
Cjelobrojno zbrajanje (+I)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Cjelobrojno množenje (*I)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Rotacija ulijevo (RL)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Rotacija udesno (RR)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Posmak ulijevo (SL)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Posmak udesno (SR)	Cjelobrojni	Cjelobrojni	Cjelobrojni
Zbrajanje s pomičnim zarezom (+D)	S pomičnim zarezom	S pomičnim zarezom	S pomičnim zarezom
Množenje s pomičnim zarezom (*D)	S pomičnim zarezom	S pomičnim zarezom	S pomičnim zarezom
Posebno zbrajanje 1 (ADD_II_D)	Cjelobrojni	Cjelobrojni	S pomičnim zarezom
Posebno zbrajanje 2 (ADD_ID_D)	Cjelobrojni	S pomičnim zarezom	S pomičnim zarezom
Posebno množenje (MUL_ID_D)	Cjelobrojni	S pomičnim zarezom	S pomičnim zarezom

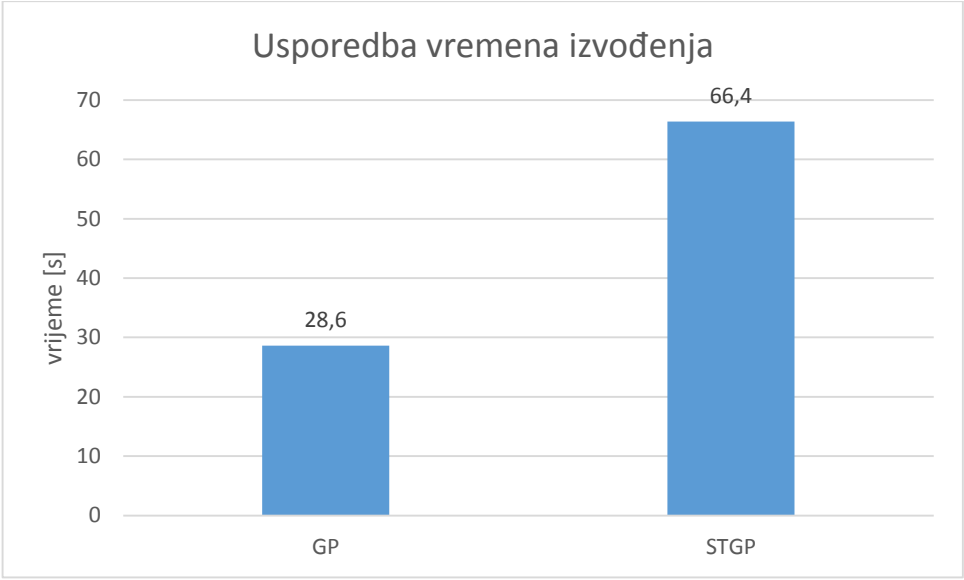
Za evoluciju su u oba slučaja korišteni isti operator križanja i mutacije. Riječ je o jednostavnom križanju gdje se nasumično biraju čvorovi u oba stabla za križanje (uz modifikaciju kod strogo tipiziranog stabla koja pazi da odabere čvorove sa istim povratnim tipom) te mutaciji koja nasumično odabrano podstablo zamjenjuje novim.

Na slici 4.1.3. je grafički prikaz usporedbe prosječnog najboljeg rješenja po generacijama između klasičnog genetskog programiranja i strogo tipiziranog genetskog programiranja dobivenih iz deset uzastopnih mjerenja. Iz grafa se može uočiti da dobrota poprima približne vrijednosti što je bilo i za očekivati jer nismo bitnije mijenjali skup korištenih primitiva.

Slika 4.1.4. predstavlja usporedbu prosječnog vremena izvođenja dobivenu iz deset uzastopnih mjerenja. Za uvjet zaustavljanja postavljen je maksimalan broj generacija – 1000. Nažalost, vrijeme izvođenja kod strogo tipiziranog GP-a je za ovaj primjer skoro 2,5 puta veće, ali za primjere gdje evaluacija jedinke traje duže, razlika bi trebala biti manja. Iako se za konkretan primjer ne isplati koristiti strogo tipizirani GP zbog višestruko većeg vremena izvođenja, za primjer koji je spomenut u uvodnom poglavlju, a i mnoge druge, strogo tipizirani GP itekako ima smisla.

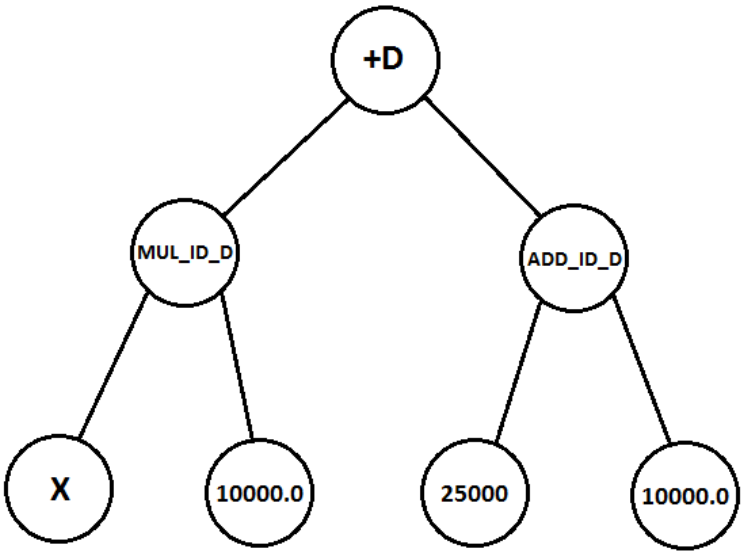


Slika 4.1.3. Grafička usporedba prosječne dobrote rješenja po generacijama



Slika 4.1.4. Grafička usporedba vremena izvođenja

Drugi primjer je po svemu identičan prvome osim matematičkog izraza koji evoluira i skupa definiranih terminala. Izraz koji se evoluira definiran je kao $f(x) = 10000x + 35000$, a skup terminala sadrži varijablu x te konstante 25000 i 10000.0. Pritom valja primijetiti da je prva konstanta cjelobrojna, a druga s pomičnim zarezom. Cilj ovog primjera je da iskonstruira jednostavno stablo u kojem će se jasno vidjeti snaga strogo tipiziranog genetskog programiranja. Dobiveno stablo koje predstavlja evoluirani matematički izraz identičan zadanom prikazano je na slici 4.1.5.



Slika 4.1.5. Evoluirano stablo koje predstavlja izraz $10000x + 35000$

4.2. Semantičko genetsko programiranje

Budući da implementirani algoritam izgradnje semantički ispravnog stabla ne radi s terminalima koji imaju složene mjerne jedinice (npr. akceleracija koja ima mjernu jedinicu $m * s^2$) za primjer pomoću kojeg ćemo usporediti klasično i semantičko genetsko programiranje smo odabrali problem raspoređivanja koji ima 9 različitih terminala s dvije moguće mjerne jedinice – ili je riječ o terminalu koji predstavlja vrijeme pa mu je mjerna jedinica sekunda ili je riječ o terminalu koji predstavlja određeni težinski faktor pa je bez mjerne jedinice.

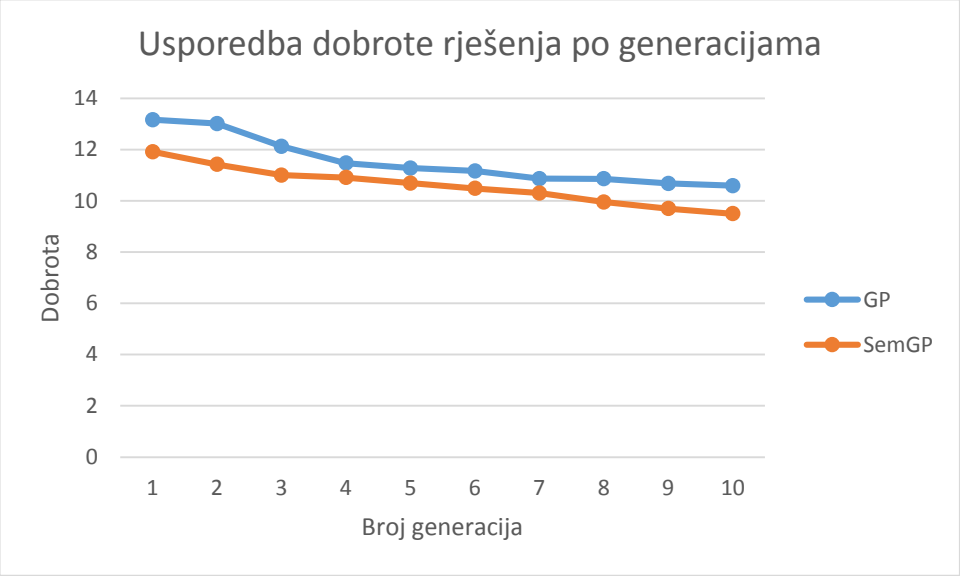
U datoteci s parametrima je uz skup terminala potrebno definirati i veličinu vektora jedinica (*unitvecdim* – u ovom slučaju je to 1 jer imamo samo sekundu za mjernu jedinicu; slučaj bez mjerne jedinice odgovara mjernoj jedinici s^0) te skup vektora u kojem je za svaki terminal definiran po jedan vektor jedinica (*unitvectors*). Za konkretan primjer, skup vektora bi sadržavao devet jednočlanih vektora. Također je potrebno definirati željenu mjernu jedinicu korijenskog čvora (*wantedunitvec*).

Slika 4.2.1. predstavlja zapise u datoteci s parametrima koji definiraju genotip klasičnog stabla (*maxdepth, mindepth, functionset, terminalset*) zajedno sa parametrima semantičkog stabla opisanim u prethodnom paragrafu.

```
<Genotype>
    <SemTree>
        <Entry key="maxdepth">4</Entry>
        <Entry key="mindepth">1</Entry>
        <Entry key="functionset">+ - / *</Entry>
        <Entry key="terminalset">pt dd w SL pmin pavg PAT MR age</Entry>
        <Entry key="unitvecdim">1</Entry>
        <Entry key="unitvectors">1 1 0 0 1 1 1 1 1</Entry>
        <Entry key="wantedunitvec">1</Entry>
    </SemTree>
</Genotype>
```

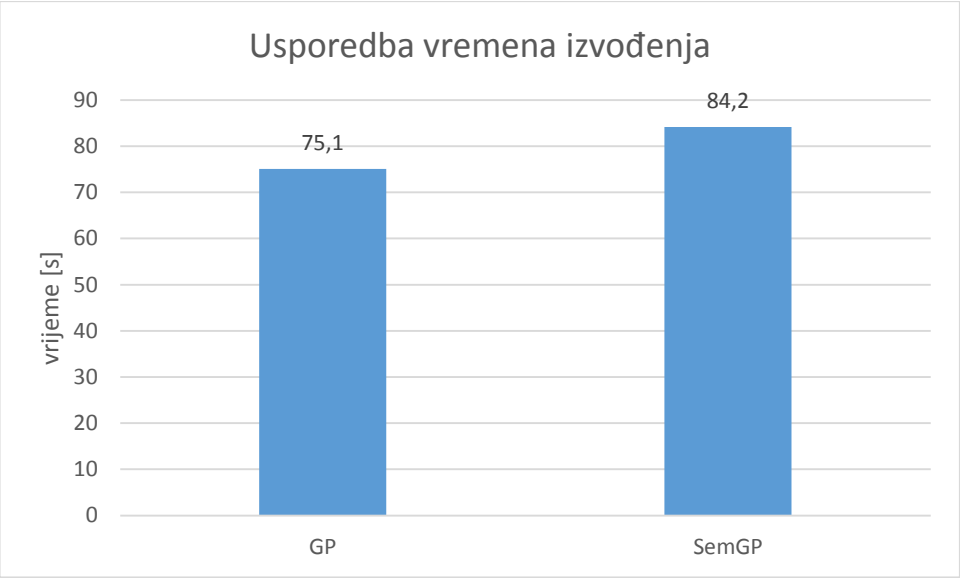
Slika 4.2.1. Isječak zapisa iz datoteke s parametrima

Na slici 4.2.2. vidimo grafički prikaz usporedbe prosječnog najboljeg rješenja po generacijama između klasičnog genetskog programiranja i semantičkog genetskog programiranja. Iz grafa se može jasno očitati da semantičko programiranje daje bolje rezultate od klasičnog genetskog programiranja što bi se mogli pripisati „usmjeravanju“ algoritma pretrage ka boljem rješenju jer uzima u obzir mjerne jedinice koje su u samoj srži zadanog problema.



Slika 4.2.2. Grafička usporedba prosječne dobrote rješenja po generacijama

Na slici 4.2.3. vidimo da razlika u vremenima izvođenja nije toliko drastična koliko je bila kod usporedbe klasičnog GP-a i strogo tipiziranog GP-a jer je u primjeru za semantički GP evaluacija jedinke dva do tri redova veličine puta dulja od evaluacije jedinke iz primjera za strogo tipizirani GP.



Slika 4.2.3. Grafička usporedba vremena izvođenja

5. Zaključak

Genetsko programiranje je vrlo koristan alat koji u svom osnovnom obliku može biti koristan za rješavanje velikog skupa različitih problema, ali kod određenih problema klasično genetsko programiranje nije dovoljno. Stoga se u ovom radu koriste dvije varijante genetskog programiranja koje proširuju funkcionalnost genetskog programiranja – strogo tipizirano te semantičko genetsko programiranje.

Strogo tipizirano genetsko programiranje omogućava genetskom programiranju da raspolaže s različitim tipovima vrijednosti koje se koriste pri izgradnji genetskog stabla. To korisniku povećava slobodu pri definiranju novih primitiva, a samim time i ekspresivnost genetskog stabla te veću mogućnost pronalaska boljeg rješenja ukoliko korisnik to zna iskoristiti.

Računala stvari predstavljaju sa brojevima, ali bez mjernih jedinica sami brojevi ne znače puno. Kvantificiranim vrijednostima su potrebne mjerne jedinice kako bi imali smisla. Ukoliko želimo evoluirati matematičke i fizikalne izraze koji su semantički ispravni pomoću genetskog programiranja potrebno nam je semantičko genetsko programiranje.

Strogo tipizirano genetsko programiranje je implementirano u potpunosti, ali kod semantičkog genetskog programiranja ima jako puno mjesta za poboljšanja. Prvenstveno, trebalo bi izmijeniti algoritam izgradnje novog stabla, ali u nedostatku ideje i vremena to je izostalo iz ovog rada. Ukoliko bi se implementirao kvalitetniji algoritam, dobilo bi se na ekspresivnosti stabla i time bi se povećao prostor pretraživanja rješenja. Trenutno su moguća samo rješenja koja sadrže terminale isključivo jednostavnih mjernih jedinica.

6. Literatura

- Keijzer M., Babovic V. *Dimensionally Aware Genetic Programming*, Hørsholm 1999.
- Montana, D. J. *Strongly Typed Genetic Programming*, Cambridge 2002.
- Dragoljević O. *Semantičko genetsko programiranje*, FER 2008.

Sažetak

U ovom radu je implementirano strogo tipizirano i semantičko genetsko programiranje u sklopu razvojnog okruženja za evolucijske algoritme *Evolutionary Computational Framework*. Uz opis i teorijsku pozadinu strogo tipiziranog i semantičkog genetskog programiranje dane su usporedbe rezultata kao i primjeri korištenja za buduće korisnike spomenutog razvojnog okruženja.

Ključne riječi: *Evolutionary Computational Framework*, genetsko programiranje, strogo tipizirano genetsko programiranje, semantičko genetsko programiranje

Abstract

This paper is about implementing strongly typed and semantic genetic programming within *Evolutionary Computational Framework*. It contains brief descriptions of strongly typed and semantic genetic programming as well as examples of how to use them as a reference for future users of said framework.

Keywords: *Evolutionary Computational Framework*, genetic programming, strongly typed genetic programming, semantic genetic programming