# Генетичко програмирање

Семинарски рад у оквиру курса
Методологија стручног и научног рада
Математички факултет

Александра Стојановић, Ивана Ивановић,
Александар Стефановић, Оливера Поповић
mi16048@alas.matf.bg.ac.rs, mi16120@alas.matf.bg.ac.rs,
ai16222@alas.matf.bg.ac.rs, mi16064@alas.matf.bg.ac.rs

1. april 2020.

**Апстракт**

Аутоматско решавање проблема од стране рачунара је централни проблем вештачке интелигенције и машинског учења. Због тога је генетичко програмирање неизоставна тема онога што је Тјуринг називао „машинском интелигенцијом". У овом раду је укратко приказан појам генетичког програмирања, уз осврт на историјат, опис самог алгоритма, његове примене и неке додатне методе попут мета-генетичког програмирања. Напослетку, дат је пример имплементације генетичког програмирања у програмском језику Python.

## Садржај

# 1 Увод

Генетичко програмирање је метода која је настала као један од покушаја да се спроведе аутоматизација прављења рачунарских програма за решавање проблема задатих на веома апстрактном нивоу. Основни циљ је да ови проблеми буду тако дефинисани да је познато једино шта треба да буде урађено, без додатних информација о форми или структури решења. Да би машине овако радиле посао који иначе ради човек, потребно је да оне искажу неки вид интелигенције [3].

# 2 Историјат

Први запис предлога за развијање програма вероватно је запис Алана Тјуринга. Он је 1948. године написао шаховски програм за рачунар који још увек није постојао, док је 1952. године сам симулирао програм. Дошло је до размака од 25 година пре објављивања књиге „Прилагођавање природним и вештачким системима“ Џона Холанда, која је поставила теоријске и емпиријске темеље науке. Џон Холанд постао је познат по генетском алгоритму, али о програмима је већ говорио у свом семинарском раду из 1962. године. Дуго се сматрало да излагање рачунарских програма случајним силама мутације и рекомбинације неће донети одрживе програме. Сматрало се да је рачунарски код сувише слаб да би се могао побољшати случајним генерисањем у еволуцији. Појам генетичко програмирање описује истраживачко подручје у пољу еволуцијског рачунања које се бави еволуцијом рачунарског кода. Њени алгоритми имају за циљ да олакшају решења проблема машинског учења или да индукују прецизна решења у облику граматички исправних (језичких) структура за аутоматско програмирање рачунара.

Дуго је прошло док није утврђено да није немогуће еволуирати рачунарски код. Овај развој захтевао је неколико различитих (мањих) посредних корака. Пре свега оригиналне генетске алгоритме који користе низове битова фиксне дужине да представе бројеве у проблемима оптимизације. Прво схватање било је да се систем правила може подвргнути еволуцији. У другом семинарском раду, Холанд и Раитман представили су систем класификатора који је омогућио да се правила развију током еволуције. Кључни допринос система класификатора је био да су сва правила знак обележја програмских језика и извршавања сложенијих рачунарских кодова.

Убрзо након Холанда и Раитмана, Смит је увео репрезентацију промењиве дужине. Омогућио је да се уједине правила у програме засноване на њима, који могу да реше задатак дефинисан функцијом прилагођености, о којој ће бити више речи у наредном поглављу. Ричард Форсит је 1981. године демонстрирао успешну еволуцију малих програма, представљених као стабла, за извршење класификације доказа о месту злочина за матичну канцеларију у Великој Британији. Форсит је објавио логички систем правила с параметрима који нам омогућавају класификацију примера у различите класе. За обучавање класификатора се користи низ обука узорака. Програми би логички и нумерички процењивали узорке података да би добили закључке о класи примера.

У другој половини 1980-их, број раних генетички програмираних (ГП) система се повећавао. Крамер је 1985. представио два еволуциона програмска система заснована на различитим једноставним јези-

цима. Хиклин, Фујики и Дикинсон написали су системе који представљају претече за одређене апликације користећи стандардни програмски језик Lisp [10], а Коза је 1989. коначно документовао методу која је користила универзални језик и примењена је на много различитих проблема. Генетичко програмирање је наставило са напредовањем објављивањем књиге Џона Козе 1992. године. Коза је објавио серију од 4 књиге, почев од 1992. године са пратећим видео снимцима. Након тога, дошло је до огромног повећања броја публикација са „Библиографијом о генетичком програмирању“, премашивши 10 000 уноса.

Коза је 1996. започео годишњу конференцију о генетичком програмирању коју је 1998. пратила годишња конференција ЕуроГП, и прва књига у ГП серији коју је уредио Коза. 1998. године представљен је и први уџбеник ГП-а. Рик Риоло је наставио процват ГП-а, што је довело до првог специјалистичког часописа опште праксе, а три године касније, 2003. године основана је годишња радионица „Теорије и праксе генетичког програмирања“. Радови о генетичком програмирању и даље се објављују на различитим конференцијама и повезаним часописима. Данас постоји деветнаест ГП књига укључујући неколико књига за студенте [1].

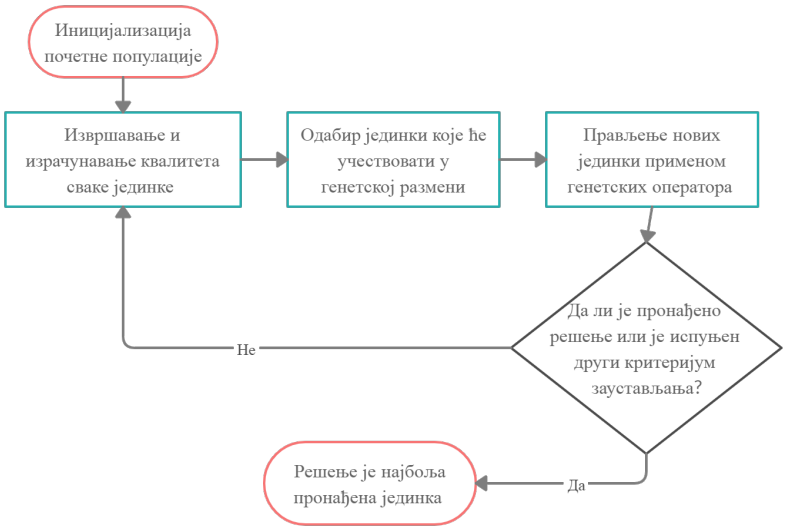
# 3 Опис алгоритма

Идеја на којој генетичко програмирање почива је процес еволуције који се јавља у природи. Самим тим то је техника која спада у групу алгоритама **еволутивног израчунавања** [6]. Еволуција подразумева промену особина укупне популације кроз више смењених генерација. Постојало је током времена више теорија еволуције, док је данас опште прихваћена теорија чији је творац британски биолог Чарлс Дарвин. По његовој теорији, сви живи организми се развијају процесом природне селекције [4]. Тим процесом јединке које су боље прилагођене имају већу вероватноћу да преживе и да оставе потомство, које је углавном једнако или чак и боље прилагођено од родитеља. Захваљујући томе што ћелије сваког живог бића садрже хромозоме, који су сачињени од гена, они могу да оставе потомство. Ген је основна јединица наслеђивања, која преноси поруке из генерације у генерацију. Дакле, репродукција подразумева комбинацију гена родитеља уз мале количине мутације. Карактеристике, односно генетски материјал прилагођенијих јединки углавном опстаје кроз генерације и у неком тренутку почиње да доминира, док остале карактеристике најчешће ишчезну. У овом одељку биће описано како се процес еволуције осликава на генетичко програмирање, које су његове основне компоненте као и то које одлуке треба донети при прилагођавању алгоритма задатом проблему.

## 3.1 Општи алгоритам

У генетичком програмирању популацију чине рачунарски програми. Еволуцијом се од почетних, најчешће на случајан начин генерисаних програма, добијају нови, у нади бољи и ефикаснији програми. Како је то случајан процес, резултат никада не може бити гарантован. Ипак, управо та случајност даје могућност превазилажења неких проблема које имају други детерминистички алгоритми [12]. На слици 1 приказан је дијаграм контроле тока опште верзије алгоритма. У наставку

биће детаљно описан сваки од наведених корака алгоритма.




Слика 1: Општи алгоритам генетичког програмирања

## Репрезентација јединки

У генетичком програмирању јединке су заправо рачунарски програми који се могу извршавати. Иако то на први поглед делује логично, оне неће бити представљене линијама кода. Разлог томе је што је неопходно да над изабраном репрезентацијом буду дефинисани генетски оператори, тако да извршавање буде ефикасно. Репрезентација која то испуњава је она у виду **синтаксних стабала** [2].

Код синтаксних стабала, основно је да се променљиве и константе налазе у листовима (најнижи чворови) и називају се терминалима, док се у осталим чворовима налазе функције. Функције могу бити различите природе, на пример аритметичке, тригонометријске или логичке, али могу бити и различите програмске конструкције попут условних израза и петљи. Свако овакво стабло мора имати скуп дефинисаних терминала и функција који заједно чине такозвани **примитивни скуп** генетичког програмирања.

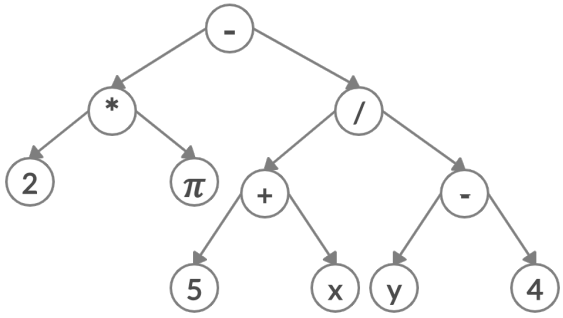
**Пример 3.1** *Нека је проблем задат аритметичком формулом:*

$$2 * \pi - \frac{5 + x}{y - 4} \tag{1}$$

*Одговарајући скуп терминала: {2, π, 5, x, y, 4}, скуп функција: {+, -, *, /}, док је одговарајуће синтаксно стабло дато на слици 2.*

Репрезентација стаблима има неколико импликација које треба имати на уму [6]:

- Величина (висина), облик (какве су гране чворова) и сложеност јединки нису фиксирани, већ се могу разликовати. Те особине




Слика 2: Синтаксно стабло које одговара формули (1)

се чак и код једне јединке могу мењати током времена, услед примене оператора за репродукцију.

- За сваки проблем посебно се мора дефинисати примитивни скуп. Као додатак, могу бити дефинисана и семантичка правила која обезбеђују конструкцију исправних стабала. На пример: *Делилац никада не сме да буде једнак 0.*

## Иницијализација популације

Генерисање почетне популације обично се врши на случајан начин. Притом се поштују сва дефинисана ограничења, уколико она постоје. Две најстарије и најједноставније методе су **потпуна метода** (eng. *full method*) и **метода раста** (eng. *grow method*), док се често користи и њихова комбинација [12].

Прва метода зове се тако јер креира потпуна стабла. Стабло је потпуно уколико су му сви листови на истој дубини. **Дубина чвора** представља број грана које треба прећи да би се од корена дошло до тог чвора. *Дубина чвора са вредношћу π приказаног на слици 2 износи 2.* По потпуној методи, вредности у чворовима се из скупа функција бирају на случајан начин све док се не дође до максималне дубине. Максимална дубина је један од унапред задатих параметара. Након тога се на случајан начин вредности бирају из скупа терминала. По другој методи, методи раста, чворови се бирају на случајан начин из целог примитивног скупа док се не дође до максималне дубине, а након тога се такође бирају само терминали. Основна разлика између ове две методе је што метода раста дозвољава креирање стабала са више различитих величина и облика.

Ипак, ниједна од ове две методе не пружа довољно велику разноврсност. Због тога се данас нашироко користи њихова комбинација коју је 1992. предложио Џон Коза [9]. Она подразумева да се прва половина генерације креира коришћењем потпуне методе, а друга методом раста.

## Функција прилагођености

Функција прилагођености (eng. *fitness function*) даје оцену **квалитета јединке**. Она треба да буде одабрана тако да може да се

израчуна за сваку јединку, а да израчунавање притом буде што ефикасније [8]. Ефикасност је неопходна јер ће се вредност ове функције израчунавати велики број пута, за сваку јединку у свакој генерацији. Зависи од проблема који се решава, али се при генетичком програмирању најчешће реализује тако што постоји скуп тест случајева којима се свака јединка евалуира. То значи да постоји одређени број улаза за које се знају очекивани излази, па се евалуира колико излаза је јединка, односно програм погодио. Осим као мера квалитета јединке, функција прилагођености се такође може користити за доделу пенала оним јединкама које нису прихватљиве или нису семантички тачне.

## Селекција

Селекција се користи како би се пронашле боље прилагођене јединке које ће учествовати у репродукцији. На основу вредности функције прилагођености, јединке се узимају са одређеном вероватноћом. Најчешће се користе **турнирска селекција** и селекција пропорционална вредности функције прилагођености, али се могу користити и друге селекционе методе еволутивног израчунавања [6]. Код турнирске селекције узима се одређени број јединки из популације, на случајан начин. Оне се упоређују и бира се најбоља од њих, на основу израчунате вредности функције прилагођености. Она ће учествовати у репродукцији. Постоје и друге варијанте турнирске селекције, али је њена предност, у сваком случају, то што и средње квалитетне јединке могу добити прилику да оставе потомство. Пример селекције пропроционалне вредности функције прилагођености је **рулетска селекција** (eng. *roulette wheel selection*). По њој, вероватноћа да ће јединка бити изабрана једнака је:

$$p_i = \frac{f(i)}{\sum_j^N f(i)} \tag{2}$$

где је f(i) вредност функције прилагођености јединке i, док је N укупан број јединки у популацији [8].

## Репродукција

Репродукција представља начин на који се од два, селекцијом изабрана родитеља добија потомство. Код генетичког програмирања најчешће се користи метод укрштања подстабала. На случајан начин одаберу се тачке укрштања код оба родитеља, након чега се једноставно замене подстабла родитеља са кореном у тачки укрштања. Постоје две опције, креирање једног или два детета разменом. Проблем код овог метода је то што се најчешће барата са стаблима код којих сваки чвор има најмање двоје деце, што говори да већи број чворова припада скупу терминала. Последица тога је што се размењују мање количине генетског материјала, јер се чешће на случајан начин погоди управо неки од терминала. Да би се избегао овај проблем, Коза је 1992. предложио нашироко коришћен приступ бирања функција 90% времена, а листова преосталих 10%. Постоји још много видова репродукције који се користе у генетичком програмирању [12].

## Мутација

Мутација се примењује како током времена јединке не би постале сувише сличне. Такође помаже у избегавању локалних екстремума, до ког би јединке највероватније једино дошле уколико не би било мутације. Мутација се најчешће имплементира тако да се на случајан начин изабере тачка јединке, па се подстабло са кореном у тој тачки замени случајно генерисаним стаблом. Још један чест начин имплементације је да се, такође на случајан начин, изабере тачка у стаблу јединке, али се сада вредност у том чвору замени случајно одабраном вредношћу исте арности из примитивног скупа. Ако нема других примитива исте арности чвор се не дира.

Основна разлика ове две имплементације је у примени. Примена прве подразумева мутацију тачно једног подстабла, док се код друге обично она примени на сваки чвор са одређеном вероватноћом, што омогућава да више чворова буде измењено.

## 3.2 Прилагођавање алгоритма

Генетичко програмирање у свом основном облику не користи специфичности ниједног проблема, те се може применити на широк спектар различитих проблема. Међутим, да би се могао користити за решавање неког конкретног проблема потребно га је прилагодити. Пример имплементације једноставног проблема дат је у A. Одлуке које притом треба донети [12]:

**1. Како изгледа примитивни скуп?**
Примитивни скуп треба да буде дефинисан тако да је могуће изразити решење коришћењем садржаних примитива. Нажалост, врло често није могуће предвидети примитивни скуп тако да задовољава ово својство. Ипак, захваљујући природи генетског програмирања, врло често иако решење не може да се представи, алгоритам може развити програме који представљају његову апроксимацију. То је најчешће и сасвим довољно. Примери функција дати су у табели 1.

Табела 1: Пример функција генетичког програмирања.

| природа функције | примери |
|---|---|
| аритметичка | $+$, -, $\times$, $\div$ |
| математичка | sin, cos, exp |
| логичка | $\wedge$, $\vee$, / |
| условни изрази | if-else |
| петље | for, while |

**2. Како изгледа функција прилагођености?**
Квалитет, односно прилагођеност јединке може се рачунати на разне начине, а то који ће се користити зависи од природе проблема. На пример може се рачунати као грешка између излаза и жељеног излаза, као тачност у класификацији објеката или као растојање од жељеног резултата.

**3. Који ће параметри бити коришћени?**
Пре покретања алгоритма потребно је подесити неколико параме-

тара, међу којима су величина популације, вероватноће примене оператора, максимална величина програма и слични детаљи. Иако вредности ових параметара зависе од проблема, генетско програмирање је у пракси углавном веома моћно те ће више параметара дати задовољавајуће резултате. Због тога постоје неке препоруке од којих вредности кренути. Правило је да величина популације буде највећа са којом систем који користимо може да ради, а да је то најмање 500 [12]. Укрштање се обично примењује са највећом вероватноћом, која је 90% или више. Насупрот томе, мутација се обично примењује са веома малом вероватноћом, обично око 1%. Вероватноћа мутације је обично толико мала јер би у супротном алгоритам брзо прерастао у случајну претрагу услед превише измена.

**4. Који ће бити критеријум заустављања?**
Као критеријум заустављања најчешће се користи унапред дефинисан максималан број генерација. Овај број не треба да буде велики. Поред тога може бити дефинисан још и неки предикат успеха који је специфичан за проблем који се решава. Такође, извршавање се обично прекида уколико се пронађе решење које задовољава проблем. Као резултат на крају извршавања најчешће се узима најбоља пронађена јединка до тог тренутка.

# 4 Примери примене

Примена генетичког програмирања у индустрији је разнолика и драстично се повећава годинама. Биће речи о неким од важнијих или занимљивих радова написаних о изабраним дисциплинама, што не значи да не постоји доста радова, односно симулација зарад решавања других проблема у разним другим дисциплинама.

## 4.1 Роботика

Једна од симулација јесте позната игра јурке коју играју два робота јурећи се својим возилима. Оба играча поседују информацију о томе ко јури и где му се апроксимативно налази противник. Циљ игре је што краће бити онај који треба да јури противника. Игра се у четири рунде од којих се свака састоји од 25 симулација. Улоге се, у току једне рунде, мењају кад онај који јури буде од свог противника на растојању од једног возила. Резултат играча за ту рунду рачуна се као број колико пута није био онај који јури подељен са 25 [13].

Табела 2: Карактеристике путања

| Назив путање | број зелених поља | број црвених поља | број поља путање |
|---|---|---|---|
| „Santa Fe" путања | 89 | 55 | 144 |
| „Los Altos Hills" путања | 157 | 65 | 222 |

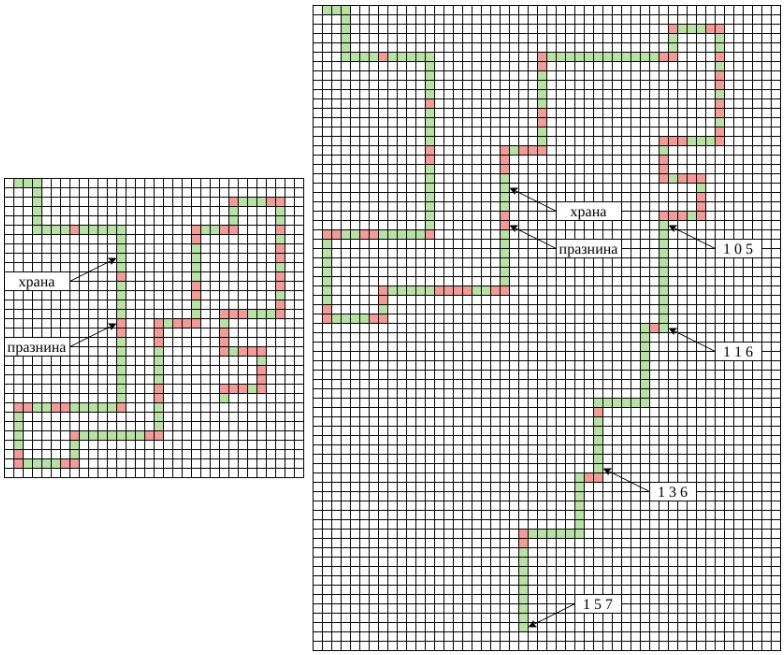
Такође, постоји и проблем навигације робота како би нашао храну која је постављена дуж неправилне линије где робот зна да извршава примитивне операције: померање напред, окретање лево/десно као

и „мирисање” хране. Путање које су коришћене у симулацији овог проблема, као и неке од њихових карактеристика дате су у табели 2. Прва од њих састоји се од линије хране између којих се налазе максимално 3 празнине које могу бити и на ивицама путање. Обе путање су представљене на слици 3, са које се јасно види да је почетни део друге путање управо прва путања. Он се завршава на 105. комаду хране, а након њега су додате још две ирегуларности у путањи:

– Налажење хране два места лево или десно од претходног комада хране. Прво појављивање после 116. комада хране.

– Померање једно место унапред, затим проналажење хране два места лево или десно. Прва појава после 136. комада хране.

За прву путању успешно је решен проблем у 21. генерацији, док је за другу путању било потребно стићи до 19. генерације [9].




Слика 3: Изглед „Santa Fe” и „Los Altos Hills” путања

## 4.2 Медицина

Иако генетичко програмирање не може бити једино решење када се анализирају биолошки подаци, оно је важно као додатан аналитички алат који има потенцијал да пружи изненађујуће и непредвидиве резултате. Способност да пронађе корисне комбинације својстава и да их прикаже у облику разумљивом за људе га чине погодним за истраживање болести попут рака [16].

Коришћење унапређеног генетичког програмирања је показало боље резултате у проблему класификације података од класичних алгоритама машинског учења попут линеарне регресије, неуронских мрежа и класификације базиране на суседима [15].

### 4.3 Економија

Можда један од најбитнијих проблема са којима се сусреће ова дисциплина је предвиђање банкрота компанија. Коришћењем генетичког програмирања дошло се до занимљивих карактеристика података ових предузећа [14].

Такође, у економији је од велике важности предвидети податке на тржиштима хартија од вредности (берзама), како због смањења губитака од куповине и продаје акција, тако и зарад максимизовања профита [7].

## 5 Мета-генетичко програмирање

Мета-генетичко програмирање подразумева упоредно еволуирање програма који врши генетичко програмирање, чиме се замењује ручно подешавање параметара и одлика алгоритма.

Постоје различити начини да се примени мета-генетичко програмирање. Према подели датој у [5], они се могу грубо сврстати у четири категорије, према њиховим приступима, који су:

- Додавање додатног генетичког материјала у геном,
- Прављење колекција модула и имплицитно мењање *језика* изражавања гена,
- Мењање фиксног скупа параметара, попут параметра учесталости мутације,
- Еволуирање оператора генетичког програмирања, тако да се не користе фиксни, унапред-познати оператори (даљи текст ће анализирати овај начин).

Треба узети у обзир да мета-генетички алгоритми изискују знатно више рачунарских ресурса, јер сада свака јединка на првом метанивоу захтева извршавање алгоритма генетичког програмирања, ради израчунавања њене прилагођености. Сходно томе, треба проценити да ли је додатно улагање ресурса вредно могућих добитака.

Обично су оператори генетичког програмирања унапред задати и фиксни, и најчешће се ради о комбинацији укрштања и мутација. Истраживани су и неки алтернативни оператори, и испоставља се да у неким случајевима, коришћење тих алтернативних оператора, заједно са „уобичајеним" операторима даје боље резултате. Дакле, паметним одабиром и укључивањем одређених оператора мутације, могуће је постићи бољи учинак програма који врши генетичко програмирање.

Међутим, одабир оператора није тривијалан, јер је могуће да неки јако добар оператор не буде ни лак за тумачење, ни интуитиван, односно, мало је вероватно да би нека особа сама осмислила такав оператор. То је нешто што сасвим одговара мотивацији генетичких алгоритама уопште, где је простор претраге јако велик, и посао проналажења решења је препуштен рачунару, а не особи.

Као и у типичном генетичком програмирању, јединке се приказују помоћу синтаксних стабала, с тим што оваква стабла садрже кораке које заједно сачињавају оператор мутације. У овом случају, прилагођеност је сразмерна прилагођености јединки које јединка мета-генетичког програмирања може да генерише својим извршавањем.

У [5], коришћени су ови листови:

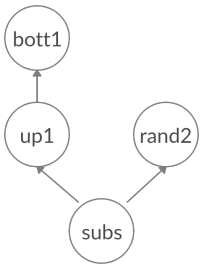
- `rand1` — Вратити прво стабло, са насумично изабраним чвором

- `rand2` — Вратити друго стабло, са насумично изабраним чвором
- `bott1` — Вратити прво стабло, са изабраним кореним чвором
- `bott2` — Вратити друго стабло, са изабраним кореним чвором

И следећи унутрашњи чворови:

- `top` — Одсећи део стабла тако да изабрани чвор сада постане нови корен
- `up1` — Вратити исто стабло, али променити изабрани чвор са тренутног на његово прво дете (ако такво не постоји, вратити неизмењено стабло)
- `up2` — Вратити исто стабло, али променити изабрани чвор са тренутног на његово друго дете (ако такво не постоји, вратити неизмењено стабло)
- `down` — Вратити исто стабло, али променити изабрани чвор са тренутног на његовог родитеља (ако родитељ не постоји, вратити неизмењено стабло)
- `down1` — Исто као `down` (како оператори не би имали склоност према `up` операторима)
- `subs` — Вратити стабло које је први аргумент, тако да је његов изабрани чвор замењен стаблом које је други аргумент.

На пример, оператор који има структуру задату сликом 4 се може тумачити као „прво дете корена првог стабла замени са другим стаблом и вратити тако добијено стабло”.




Слика 4: Пример једног оператора

# 6 Закључак

Од првог коришћења генетичког програмирања 1990. године, рачунска снага се повећала за фактор од 40 000 по Муровом закону сваких 18 месеци. Како Коза истиче, иако су у почетку били доступни само лаки проблеми у оквиру генетичког програмирања, касније повећање рачунске снаге и методолошког напретка генетичког програмирања омогућили су нова решења патентираних изума, као и потпуно нове проналаске који су потом патентирани.

# Литература


[1] Ajith Abraham, Nadia Nedjah, and Luiza Mourelle. *Evolutionary Computation: from Genetic Algorithms to Genetic Programming*, volume 13, pages 1–20. 05 2006.

[2] A Aho, M.S. Lam, R. Sethi, and J. Ullman. *Compilers: principles, techniques and tools*. Addison-Wesley Longman Publishing Co., 2006.

[3] B.J. Copeland. *The Essential Turing*, chapter 10. Oxford University Press, USA, 2004.

[4] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859.

[5] B. Edmonds. Meta-genetic programming: Co-evolving the operators of variation. *Elektrik*, 9(1):13–30, 2001.

[6] A.P. Engelbrecht. *Computational Intelligence: An Introduction*. John Wiley & Sons, Ltd, 2007.

[7] Sasaki T. Iba H. Using genetic programming to predict financial data. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*, pages 244–251, july 1999.

[8] Predrag Janičić and Mladen. Nikolić. *Vestačka inteligencija*. Matematički fakultet u Beogradu, 2019.

[9] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT Press, London, 1992.

[10] John McCarthy. Lisp, 1958. on-line at: https://lisp-lang.org/.

[11] S. Mišković. Genetičko programiranje, 1. april 2020. on-line at: http://poincare.matf.bg.ac.rs/~stefan/ri/7.html.

[12] R Poli, W.B. Langdon, and N.F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[13] Craig W. Reynolds. Competition, Coevolution and the Game of Tag. In *Proceedings of Artificial Life IV*, pages 59–69, jan 1994.

[14] Terje Lensberg Thomas E. McKee. Genetic Programming and rough sets: A hybrid approach to bankruptcy classification. *European Journal of Operational Research*, 1(138):436–451, 2002.

[15] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Using enhanced genetic programming techniques for evolving classifiers in the context of medical diagnosis. *Genetic Programming and Evolvable Machines*, 10:111–140, 06 2009.

[16] William Worzel, Jianjun Yu, Arpit Almal, and Arul Chinnaiyan. Applications of genetic programming in cancer research. *The International Journal of Biochemistry & Cell Biology*, 41:405–413, 02 2009.


# A Додатак

## Пример имплементације алгоритма

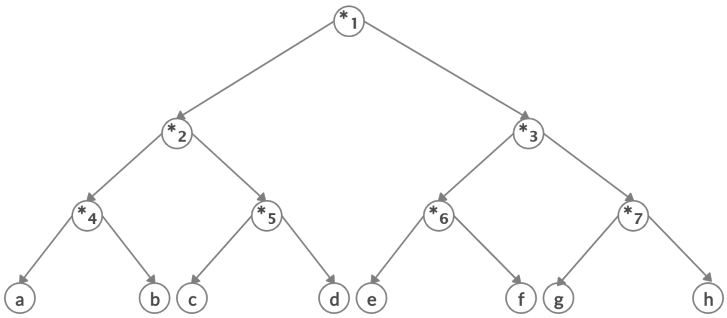
У овом одељку биће описан једноставан пример имплементације генетичког програмирања, који би требало да илуструје цео процес

прилагођавања од почетка до краја. Сав иложени код написан је у програмском језику Python. Пример је инспирисан кодом датом на [11].

Нека је проблем наћи максимум функције дате формулом:

$$((a *_4 b) *_2 (c *_5 d)) *_1 ((e *_6 f) *_3 (g *_7 h)) \tag{3}$$

при чему је $*_x$ нека вредност из скупа функција {+, -, *}, док променљиве a, b, c, d, e, f, g и h имају вредност између 1 и 10. Синтаксно стабло које одговара овој формули приказано је на слици 5.




Слика 5: Синтаксно стабло које одговара формули (3)

Пратећи редослед дат у 3.1, први корак је одлука која ће репрезентација бити коришћена. Како су све функције из скупа исте арности (арности 2), могуће је синтаксно стабло представити на ефикаснији начин, низом елемената. Низ се попуњава редом, обиласком стабла слева на десно. Захваљујући информацији да су све функције исте арности, могуће је јединствено одредити на ком се месту у низу се налази родитељ, а на ком његова деца из одговарајућег синтаксног стабла. Ако се родитељ налази на месту x у низу, његова деца налазиће се на местима 2*x+1 и 2*x+2 (нумерисање чворова почиње од 0). Низ који одговара стаблу са слике 5 приказан је на слици 6.

| *1 | *2 | *3 | *4 | *5 | *6 | *7 | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Слика 6: Низ који одговара репрезентацији формуле (3)

Наредни корак је иницијализација популације. Свака јединка иницијализована је на случајан начин. За првих 7 елемената у низу, вредност се бира из скупа функција. Вредности преосталих елемената бирају се на случајан начин између 1 и 10. Овај поступак понавља се онолико пута колика је величина популације, која се задаје као параметар алгоритма.

У датом примеру, квалитет јединке једноставно се рачуна као вредност израза који је представљен низом. Код који приказује класу којом је описана јединка, а за њом и функција за иницијализацију, дати су у наставку.

```
import random

class Jedinka:
    def __init__(self):
        self.stablo = [self.slucajna_funkcija() for i in range(7)]
        self.stablo += [random.randrange(1, 11) for i in range(8)]
        self.prilagodjenost = self.funkcija_prilagodjenosti()

    def slucajna_funkcija(self):
        skup_funkcija = ['+', '-', '*']
        return random.choice(skup_funkcija)

    # izraz se pravi rekurzivno, polazeci od korena
    def napravi_izraz(self, t):
        if t == 15:
            return ''

        if t > 6:
            return str(self.stablo[t])

        if 2*t+1 > 6:
            return self.napravi_izraz(t*2+1) + \
                   self.stablo[t] + \
                   self.napravi_izraz(t*2+2)

        return '(' + self.napravi_izraz(t*2+1) + ')' + \
               self.stablo[t] + \
               '(' + self.napravi_izraz(t*2+2) + ')'

    def funkcija_prilagodjenosti(self):
        '''
        izraz predstavljen niskom prosledjuje se funkciji eval,
        koja izracunava njegovu vrednost
        '''
        return eval(self.napravi_izraz(0))

    def __gt__(self, druga_jedinka):
        '''
        neophodno je definisati operator > da bi se kasnije mogla
        sortirati populacija kako bi se pronasle najbolje jedinke
        '''
        return self.prilagodjenost > druga_jedinka.prilagodjenost


def inicijalizacija(velicina_populacije):
    return [Jedinka() for i in range(velicina_populacije)]
```

Након иницијализације почетне популације, потребно је дефинисати како се из популације бирају јединке које ће учествовати у репродукцији. У примеру коришћена је турнирска селекција, са величином турнира једнаком 5. Турнир се спроводи два пута, са различитим јединкама, како би била одабрана два родитеља. Одговарајући код приказан је у наставку.

```
def turnirska_selekcija(populacija):
    maksimum = float('-inf')
    najbolja_jedinka = -1

    # trazi se maksimum
    for i in range(5):
        jedinka = random.randrange(len(populacija))
```

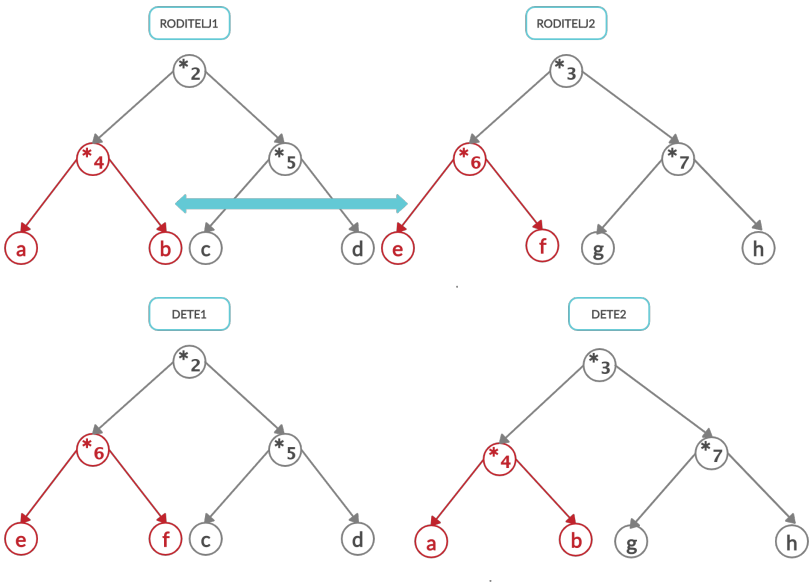
```
        kvalitet_jedinke = populacija[jedinka].prilagodjenost

        if kvalitet_jedinke > maksimum:
            maksimum = kvalitet_jedinke
            najbolja_jedinka = jedinka

    return populacija[najbolja_jedinka]
```

Када су јединке селектоване, следи репродукција. Она ће бити реализована као једноставно укрштање родитеља (описано у секцији 3.1). Слика која илуструје тај процес дата је на слици 7. На слици су приказани само делови стабала ради лакше илустрације. Одговарајући код следи у наставку.

```
def ukrstanje(roditelj1, roditelj2, dete1, dete2):
    velicina_stabla = len(roditelj1.stablo)

    # bira se tacka ukrstanja
    koren = random.randrange(velicina_stabla)
    podstablo = []

    nadji_podstablo(koren, velicina_stabla, podstablo)

    for i in range(velicina_stabla):
        dete1.stablo[i] = roditelj2.stablo[i] if i in podstablo \
        else roditelj1.stablo[i]
        dete2.stablo[i] = roditelj1.stablo[i] if i in podstablo \
        else roditelj2.stablo[i]

def nadji_podstablo(koren, velicina_stabla, podstablo):
    '''
    da bi se dobilo podstablo iz niza, potrebno je proci
    rekurzivno kroz indekse i naci one koji pripadaju
    podstablu u datom korenu
    '''

    if koren > velicina_stabla:
        return

    podstablo.append(koren)

    # prvo dete nalazi se na indeksu 2*koren+1
    nadji_podstablo(2*koren+1, velicina_stabla, podstablo)

    # drugo dete nalazi se na indeksu 2*koren+2
    nadji_podstablo(2*koren+2, velicina_stabla, podstablo)
```

Преостало је још дефинисање мутације. Она је имплементирана тако да се примењује на сваки чвор, али са вероватноћом од 10%. Уколико је чвор из скупа функција, бира се опет, на случајан начин, функција из истог скупа. У супротном поново бира се вредност између 1 и 10. Одговарајући код дат је у наставку.

```
def mutacija(jedinka):
    for i in range(len(jedinka.stablo)):
        if random.random() > 0.1:
            continue

        jedinka.stablo[i] = jedinka.slucajna_funkcija() if i < 7 \
        else random.randrange(1, 11)
```




Слика 7: Пример укрштања

Пре извршавања комплетног алгоритма, потребно је дефинисати параметре за његову контролу. Величина популације постављена је на вредност 500, док је за број генерација узета вредност од 50 (поштујући препоруке дате у 3.2). При замени популација у смени генерација спроведен је и **елитизам**. Наиме, када се прави нова популација може се одабрати одређени број прилагођенијих јединки који ће бити директно пребачен у следећу генерацију. Мотивација иза тога је да се сачувају боље карактеристике, које би у процесу репродукције можда бити изгубљене. У примеру је одабрано да тај број буде једнак 100.

Као резултат извршавања добија се вредност од 100000000. Услед једноставности примера овај резултат је и био очекиван, јер се највећа вредност добија када се за све $*_x$ узме операција множења, а за све променљиве вредност 10. Као следећи корак могао би бити додат некакав предикат успеха. На пример, уколико се вредност не промени одређени број пута вероватно је да је алгоритам дошао до траженог максимума, те претрага може бити прекинута раније. Код извршавања комплетног алгоритма дат је у наставку.

```
1000 velicina_populacije = 500
     populacija = inicijalizacija(velicina_populacije)
1002
     broj_generacija = 50
1004 nova_populacija = populacija

1006 for iteracija in range(broj_generacija):
         # potrebno je svaki put populaciju sortirati
1008     # kako bi se sproveo elitizam
         populacija.sort()
1010
```

```
    # elitizam
    for i in range(100):
        nova_populacija[i] = populacija[i]

    for i in range(100, velicina_populacije, 2):
        roditelj1 = turnirska_selekcija(populacija)
        roditelj2 = turnirska_selekcija(populacija)

        ukrstanje(roditelj1, roditelj2, nova_populacija[i], \
        nova_populacija[i+1])

        mutacija(nova_populacija[i])
        mutacija(nova_populacija[i+1])

        nova_populacija[i].prilagodjenost = \
        nova_populacija[i].funkcija_prilagodjenosti()
        nova_populacija[i+1].prilagodjenost = \
        nova_populacija[i+1].funkcija_prilagodjenosti()

    populacija = nova_populacija

# potrebno je jos jednom sortirati populaciju,
# kako bi se izvukla najbolja vrednost
populacija.sort()
print(populacija[i].prilagodjenost)
```