**SVEUČILIŠTE U SPLITU**

**FAKULTET ELEKTROTRHNIKE STROJARSTVA I BRODOGRADNJE**

**SEMINARSKI RAD IZ KOLEGIJA RAČUNSKA INTELIGENCIJA**

**PRMJER PROGRAMA BAZIRANOG NA GENETSKIM ALGORITMIMA**

**STUDENT:**

**IVAN SIČAJA**

SPLIT, lipanj, 2020

# UVOD

U ovom seminarskom radu objašnjavamo princip rada genetičkih algoritama na jednostavnom primjeru. Kao primer će nam poslužiti jedna riječ koju mi unesemo, a naš program treba pomoću korištenja genetskih algoritama sam generirati upravo onu riječ koju unesemo. Kada se ulazna riječ odnosno sekvenca poklopi sa sekvencom koju je generirao naš program korištenjem genetskih algoritama s minimalno 90% podudaranja tada možemo konstatirati da je naš program ispunio svoju funkciju. Ideja je koristit što je moguće jednostavniji primjer programa pomoću kojeg ćemo moći najjednostavnije objasniti **principe rada genetičkih algoritama** iz razloga što principi ostaju gotovo identični i za mnogo složenije primjere programa genetskih algoritama.

Za kodiranje programa koristimo programski jezik „Python“ te nestandardni modul naziva „fuzzywuzzy“ koji je potrebno dodatno instalirati uz „Python“.

# OBAJŠNJENJE GENETSKIH ALHORITAMA POMOĆU PROGRAMSKOG KODA

Prvi korak je učitati potrebne module s kojima želimo raditi. Moduli koje ćemo koristit su:

-fuzzywuzzy (jedna od funkcija mu je proračun poklapanja sekvence s traženom sekvencom)

-random (funkcija mu je generiranje „radnom“ slova)

-string (omogućava razne manipulacije stringovima)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Uključivanje potrebnih modula

from fuzzywuzzy import fuzz

import random

import string

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Zatim definiramo određene parametre koda, a to su:

-ulazna riječ ili sekvenca (riječ koju želimo da izgeneriraju genetički algoritmi)

-dužina ulazne riječi (bitna kako bi program generirao sekvence s upravo onoliko znakova koliko   
 ima ulazna riječ)

-broj primjera početne populacije (kreiranje određenog borja „random“ sekvenci koji imaju isti   
 broj slova kao i ulazna sekvenca)

-broj generacija (određuje broj koliko će se zapravo puta vršiti procesi selekcije, križanja, mutiranja   
 sekvenci populacije)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje željenog stringa

in\_str = None

#Dužina željenog stringa

in\_str\_len = None

#Određivanje broja populacije

population = 20

#Određivanje maksimalnog broja generacija kroz koje će se tražiti riješenje

generations = 10000

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Sljedeći korak je generiranje objekata odnosno klase od populacije. Klasa se naziva „Agent“. Objekte koristimo zbog mogućnosti vršenja definiranih operacija na relativno jednostavan način.

Unutar naše klase nalazi se konstruktor čija je funkcija pridruživanje slučajnog broja slova u jedan string, te definiranje postotka podudarnosti (tj. „fitness“) s ulaznim stringom.

Osim konstruktora definirali smo i jednu dodatnu metodu naziva („\_\_str\_\_“) nad objektom „agent“ čija je funkcija ispisivanje generirane sekvence uz postotak podudarnosti s ulaznom sekvencom.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje klase "Agent"

class Agent:

#Definiranje konstruktora

    def \_\_init\_\_(self, length):

#Nasumičan odabir određenog broja slova i pridriživanje i jedan string

        self.string = ''.join(random.choice(string.ascii\_letters) for \_ in xrange(length))

        #Predefinirana vrijednost poklapanja s ulaznim stringom

        self.fitness = -1

#Metoda za ispisivanje "random" stringa i postotka poklapanja sa željenim stringom

    def \_\_str\_\_(self):

        return 'String: ' + str(self.string) + ' Fitness: ' + str(self.fitness)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kada smo definirali klasu „agent“, napravili smo glavnu funkciju naziva

„main\_genetic\_algorythm\_function()“ unutar koje ćemo pozivati sve potrebne funkcionalnosti, a to su:

-kreiranje liste agenata

-ispis kreirane sekvence s postotkom poklapanja sa željenom sekvencom

-pozivanje funkcije za proračun poklapanja

-pozivanje funkcije za odabir 20% najboljih agenata

-pozivanje funkcije za križanje agenata

-pozivanje funkcije za mutiranje agenata

Određivanje zadovoljavajućeg praga podudarnosti kreirane sekvence s traženom sekvencom (za naš slučaj za zadovoljavajući prag podudarnosti sekvenci ćemo staviti 90% iz razloga što je podudarnost vrlo velika te na osnovu nje možemo provjeriti rad algoritma, dok je drugi razlog taj ako stavimo prag na 100% i ulazna sekvenca ima npr. 40 slova vrlo je mala vjerojatnost da će se rješenje pronaći u 10 000 generacija)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje funkcije Genetskih Algoritama, main\_genetic\_algorythm\_function()

def main\_genetic\_algorythm\_function():

#Kreiranje liste agenata

    agents = init\_agents(population, in\_str\_len)

#Iplementiranje funkcija fitness,selection, crossover, mutation kroz

predefinirani broj generacija

    for generation in xrange(generations):

        #Ispis vrijednosti trenutne generacije te postotka poklapanja

        print ("Generation:"+ str(generation))

        #Proračun poklapanja određenog agenta (random primjera) s željenom

sekvencom

        agents = fitness(agents)

        #Odabir 20% najboljih aganata

        agents = selection(agents)

        #Križanje određenih odabranih agenata s dobrim svojstvima

        agents = crossover(agents)

        #Mutrianje određenih agenata

        agents = mutation(agents)

#Postavljanje zadovoljavajućeg praga poklapanja određenog primjera s traženom

sekvencom

        if any(agent.fitness >= 90 for agent in agents):

            print ('\nPrag poklapanja je zadovoljen! \n')

            exit(0)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Potrebno je kreirati listu agenata, to radimo pomoću funkcije „init\_agents()“ koja se nalazi u nastavku.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje funkcije za inicijalizaciju agenata

def init\_agents(population, length):

#Inicijaliziranje određenog broja aganata

    return [Agent(length) for \_ in xrange(population)]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kada smo kreirali agente na osnovu populacije potrebno je izračunati postotak poklapanja pojedinog agenta s traženom sekvencom. Za realiziranje ove funkcije koristimo modul „fuzzywuzzy“.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje funkcije za proračun poklapanja vrijednosti agenta s traženom

sekvencom

def fitness(agents):

#Pretraživanje liste agenata

    for agent in agents:

        #Proračun postotka poklapanja vrijednosti određenog agenta s traženom

vrijednošću

        agent.fitness = fuzz.ratio(agent.string, in\_str)

    return agents

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nakon proračuna „dobrote“ agenata odabrat ćemo 20% najboljih agenata koji ulaze u **bazen za reprodukciju**, odnosno agente koji će se križati s ciljem da njihovim preklapanjem dobijemo traženu sekvencu. Križanje vršimo pomoću funkcije „crossover(agents)“. U ovoj funkciji odabiremo dva agenta koje ćemo nazvati roditelj1 i roditelj2. Potom ćemo definirati dužinu agenta koji će nastati reprodukcijom agenta „roditelja“. Agente nastale reprodukcijom nazivamo „child1“ i „child2“. Za naš slučaj reprodukcija se vrši na način da odaberemo „random“ indeks prema kojem ćemo podjeliti string roditelja1 i roditelja2 te ih konkatenirati u nove stringove naziva „child1“ i „child2“. Radnom indeks naravno mora biti u rasponu od 0 do broja koji označava dužinu ulaznog agenta. Reproducirane agente dodajemo u niz agenta za novu generaciju.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje funkcije za križanje odabranih primjera

def crossover(agents):

    #Kreiranje niza za dodavanje primjere nastalih križanjem

    offspring = []

    for \_ in xrange((population - len(agents)) // 2):

        #Slučajni odabir nekog od agenta za "roditelja 1"

        parent1 = random.choice(agents)

        #Slučajni odabir nekog od agenta za "roditelja 2"

        parent2 = random.choice(agents)

        #Kreiranje novog agenta "child1"

        child1 = Agent(in\_str\_len)

        #Kreiranje novnog agenta "child2"

        child2 = Agent(in\_str\_len)

        #Odabir "random indexa" u rasponu od vrijednosti 0 do vrijednosti dužine ulazne sekvence odnosno dužine sekvence agenta

        split = random.randint(0, in\_str\_len)

        #Križanje dva "roditalja" te pridavanje vrijednosti novokreiranim agentima child1 i child2

        child1.string = parent1.string[0:split] + parent2.string[split:in\_str\_len]

        child2.string = parent2.string[0:split] + parent1.string[split:in\_str\_len]

        #Dodavanje novokreiranih agenata child1 i child2 u niz "offspring"

        offspring.append(child1)

        offspring.append(child2)

    #Proširivanje niza agenata nizom "offspring"

    agents.extend(offspring)

    #Vraćanje cjelokupnog izmjenjenog niza agenta

    return agents

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Primjenjivanje samo križanja u korištenju genetičkih algoritama nije preporučivo iz razloga što sam program može zapeti u neki problem koji u određenoj mjeri možemo usporediti s problemom lokalnog maksimuma kod algoritma uspona na vrh, te je potrebno napraviti nasumični skok kako bismo provjerili da li se nalazimo na globalnom maksimumu. Za naš slučaj rješavanja ovog problema koristimo funkciju **mutacije** odnosno „mutation(agents)“. Ova funkcija uzima sekvencu određenog agenta, procjenjuje koji je dio sekvence dobar, a koji ne. Dobri dio ostaje u sekvenci dok dio sekvence koji nam ne odgovara mutira odsnono odabire se nasumičan „char“ ili string te se dio sekvence koji je dobar i mutirani dio sekvence, konkateniraju u novu sekvencu odnosno novog agenta.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Definiranje funkcije za križanje odabranih primjera

def crossover(agents):

    #Kreiranje niza za dodavanje primjere nastalih križanjem

    offspring = []

    for \_ in xrange((population - len(agents)) // 2):

        #Slučajni odabir nekog od agenta za "roditelja 1"

        parent1 = random.choice(agents)

        #Slučajni odabir nekog od agenta za "roditelja 2"

        parent2 = random.choice(agents)

        #Kreiranje novnog agenta "child1"

        child1 = Agent(in\_str\_len)

        #Kreiranje novnog agenta "child2"

        child2 = Agent(in\_str\_len)

        #Odabir "random indexa" u rasponu od vrijednosti 0 do vrijednosti dužine ulazne sekvence odnosno dužine sekvence agenta

        split = random.randint(0, in\_str\_len)

        #Križanje dva "roditalja" te pridavanje vrijednosti novokreiranim agentima child1 i child2

        child1.string = parent1.string[0:split] + parent2.string[split:in\_str\_len]

        child2.string = parent2.string[0:split] + parent1.string[split:in\_str\_len]

        #Dodavanje novokreiranih agenata child1 i child2 u niz "offspring"

        offspring.append(child1)

        offspring.append(child2)

    #Proširivanje niza agenata nizom "offspring"

    agents.extend(offspring)

    #Vraćanje cjelokupnog izmjenjenog niza agenta

    return agents

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Budući da smo sve potrebne funkcionalnosti isprogramirali odnosno naš genetski algoritam je gotov napravit ćemo sažetak koda koji nije obavezan ali nam daje puno bolju preglednost. Sažetak koda radi povećane preglednosti se nalazi u nastavku.

U sažetku koda unosimo sekvencu koji želimo generirati pomoću genetskih algoritama te pozivamo funkciju sa svim isprogramiranim funkcionalnostima odnosno funkciju

„main\_genetic\_algorythm\_function()“.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

if \_\_name\_\_ == '\_\_main\_\_':

    #Definiranje sekvence koju želimo dobiti pomoću genetskih algoritama

    in\_str = 'Split'

    #Određivanje dužine ulazne sekvence

    in\_str\_len = len(in\_str)

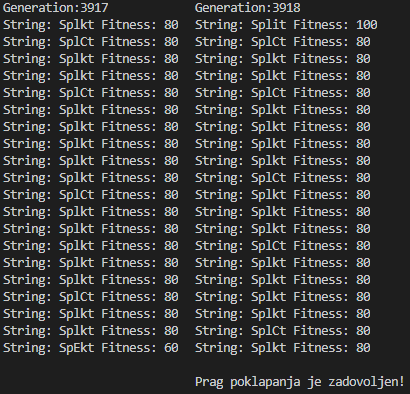
    #Primjena genetičkih funkcija za dati primjer

    main\_genetic\_algorythm\_function()

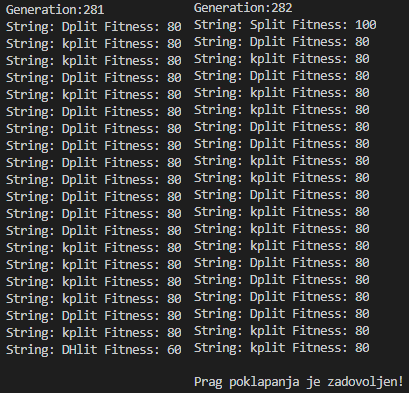
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# TEST

Za ulaznu sekvencu odabrat ćemo string „Split“.



Slika 3.1 Prikaz rješenja u prvoj iteraciji programa



Slika 3.2 Prikaz rješenja u drugoj iteraciji programa

Prilikom pokretanja programa tražena sekvenca „Split“ je pomoću genetskih algoritama u prvoj iteraciji generirana u 3918. generaciji, dok je u drugoj iteraciji programa generirana u 282. generaciji što čini ogromnu razliku u utrošenom vremenu. Razlog ovakve razlike je upravo nasumičan, „random“ način generiranja slova sekvence koji odmah u početku može generirati vrlo dobru polaznu sekvencu, a može generirati i vrlo lošu polaznu sekvencu, što će zahtijevati manje odnosno više vremena da se dođe do ulazne sekvence. Nebitno radi li se o nasumično dobro ili loše generiranoj početnoj sekvenci za jednostavne sekvence (sekvence koje se ne sastoje od jako velikog broja slova) rješenje će biti u konačnici pronađeno.

# ZAKLJUČAK

Genetski algoritmi predstavljaju način pronalaska određenog rješenja koristeći prirodne zakone reprodukcije i mutacije koji se provode nad određenom populacijom nastojeći ju približiti konačnom rješenju kroz neku od sljedećih generacija. Samo vrijeme pronalaska konačnog rješenja za istu ulaznu sekvencu može biti vrlo različito, za određene vrlo složene probleme vremenski period pronalaska rješenja može bit vrlo dug, ali konačno dug, što ovaj način rješavanja problema čini vrlo moćnima alatom za rješavanja problema u različitim područjima.