**Univerzitet u Banjoj Luci**

**Elektrotehnički fakultet**

**Katedra za automatiku**

**Metodi vještačke inteligencije**

Izvještaj o urađenom projektnom zadatku

**Genetički algoritmi**

Student:

Halim Bačić, 1330/14

# Opis problema

Korištenjem genetičkog algoritma, određujemo minimum funkcije:

i to na intervalu gdje

U drugom dijelu projekta, potrebno je odrediti maksimum presjeka date funkcije i ravni nastele presjekom x, y, z ose za x=0.

# Rješenje

Do rješenja se dolazi korištenjem programa pisanim u Java programskom jeziku. Bitne klase koje su korištene u rješavanju problema su: Point3D, Point2D, Population i Simulation.

Point3D i Point2D su klase koje nasljeđuju klasu Point. Point3D se koristi u prvom djelu zadatka, a Point2D se koristi u rješavanju drugog dijela zadatka. Klasa Point sadrži atribut *fitness* koji se koristi u selekciji rješenja i selekciji jedinki populacije pri korištenju genetičkog algoritma. Metoda compareTo je implementirana tako da se pri poređenju objekata tipa Point2D i Point3D vrši poređenje po vrijednosti atributa *fitness.*

Klasa Population sadrži dva atributa, a to su trenutna populacija i redni broj populacije. Bitne metode koje se nalaze u klasi Population su generatePopulationPoint3D I generatePopulationPoint2D koje se koriste za inicijalizovanje početnih populacija. Početnu populaciju čine slučajno generisane koordinate tački. Vrijednosti atributa se nalaze u pomenutim intervalima za svaku od generisanih tačaka. Klasa Population sadrži metodu sortPopulation koja sortira jedinke u populaciji na osnovu fitness vrijednosti, te se na taj način biraju elementi koji će se prenositi u narednu generaciju.

Klasa Simulation je glavna klasa zadužena za implementaciju algoritma. Atributi klase su vrijednosti koje su u praksi ključne za implementaciju algoritma, a to su veličina populacije, broj iteracija (koji su cjelobrojne vrijednosti) te vjerovatnoće za ukrštanje jedinki i mutaciju novih jedinki. Klasa Simulation sadrži metode kojima se vrši sortiranje populacije, izračunavanje fitness vrijednosti, funkcije za ukrštanje te funkcije za mutaciju novih jedinki. Klasa Simulation implementira interfejs BinaryRepresentation koji sadrži metode za konverziju Double vrijednost u binarni oblik i obrnuto da bi se održao koncept genetičkog algoritma.

**1. Generisanje inicijalne populacije**

Nakon unosa vrijednosti za veličinu populacije (n), šanse za ukrštanje (c), šanse za mutaciju (m), broja iteracija (it) i tolerancije (t) spremni smo da pokrenemo simulaciju genetičkog algoritma. Generiše se na slučajan način n objekata tipa Point3D ili Point2D, zavisno od dijela zadatka koji se izvšava. Koriste se funkcije generatePopulationPoint3D I generatePopulationPoint2D. Fitnes vrijednost je pri inicijalnoj generaciji jednaka 0 za sve jedinke.

Inicijalna populacija je generisana sa dvije metode, u zavisnosti od problema koji se rješava. Ukoliko je potrebno rješavati prvi dio zadatka korištena je metoda generatePopulationPoint3D(Integer num), odnosno generatePopulationPoint2D(Integer num) za drugi dio zadatka. Parametar num koji primaju obje funkcije se odnosi na broj elemenata u populaciji.

private void generatePopulationPoint3D(Integer numOfPopulation){  
 while (numOfPopulation > 0) {  
 Random random = new Random();  
 Double x = (3 - random**.**nextInt(6)) \* random**.**nextDouble();  
 Double y = (4-random**.**nextInt(8)) \* random**.**nextDouble();  
 Point3D point = new Point3D(x, y);  
 population**.**add((T) point);  
 numOfPopulation--;  
 }  
}  
  
private void generatePopulationPoint2D(Integer numOfPopulation){  
 while (numOfPopulation > 0) {  
 Random random = new Random();  
 Double y = random**.**nextInt(4) \* random**.**nextDouble();  
 Point2D point = new Point2D(y);  
 population**.**add((T) point);  
 numOfPopulation--;}

**2. Izvšavanje koraka ukrštanja, mutacije i sortiranja određen broj puta**

Naredni koraci se izvršavaju unešeni broj puta te se ovaj dio algoritma dijeli na podkorake:

**2a) Ukrštanje** – biraju se jedinke u trenutnoj populaciji te se provjerava da li će doći do ukršanja između njih. Poredi se slučajna vrijednost (s) u opsegu 0-1,0 sa unešenom vrijednost c. Ukoliko je s > c doći će do ukrštanja jedinki te se generiše novi objekat klase Point.

PRIMJER Ukrštanja

-3.302 → **-011.0100110101**

+5.406 → **+101.0110011111**

Vrijednost nastala ukrštanjem je skup crvenih bitova na svojim pozicijama**.**

3.280 → **+011.0100011111**

Na ovaj način, omogućili smo ukrštanje dvije jedinke tako da dobije osobine oba roditelja ravnomjerno.

Funkcije kojima se vrši ukrštanje su: protected void crossing(), protected boolean tryCrossing(Point one, Point two), protected Point makeNewPoint(Point one, Point two), protected String makeCross(String one, String two).

Izvršavanjem pomenutih funkcija u datom redoslijedu, simulira se naredni proces: 1. Simulacija procesa ukrštanja funkcijom protected void crossing() 2. Provjera da li će se dvije tačke u populaciji ukrstiti funkcijom protected boolean tryCrossing(Point one, Point two) 3. Generisanje nove tačke pomoću funkcije protected Point makeNewPoint(Point one, Point two) 4. Simulacija ukrštanja ukoliko su predhodna dva koraka uspješno izvršena pomoću funkcije protected String makeCross(String one, String two).

**2b) Mutacija –** nakon što je generisan novi objekat koji je nastao ukrštanjem roditeljskih objekata, slučajno se generiše vrijednost 0-1,0. Ukoliko je ta vrijednost > m vrijednosti, tada dolazi do mutacije novog objekta. Mutacije se vrši na način da se koordinate tačaka pomoćnim funkcijama konvertuju iz vrijednosti tipa Double u binarnu reprezentaciju. Tada se vrijednosti bita na slučajnim mjestim mjenjaju u obratnu binarnu vrijednost. Kada se proces mutacije završi, binarna vrijednost se pomoćnim funkcijama konvertuje u Double vrijednost, a novi objekat se kao takav dodaje u generaciju.

Prilikom izrade programa, za simuliranje procesa mutacije korištene su tri funkcije: protected Point mutation(Point point), protected Point3D mutation3D(Point3D point) i protected Point2D mutation2D(Point2D point). Point)

**2c) Sortiranje** -kada se u novu generaciju dodaju objekti nastali nakon ukrštanja i mutacije, sve jedinke (ili objekti) u generaciji se sortiraju na osnovu fitness vrijednosti, dobijene u odnosu na to koliko su vrijednost minimuma (ili maksimuma) funkcije bliske stvarnom rješenju. U tom momentu, generacija ima više jedinki od unešenog broja n zbog čega se vrši eliminacija nepotrebnih rješenja, a najboljih n jedinki se propagira u narednu iteraciju gdje se ponavlja navedeni proces.

Kvalitet svakog rješenja tj. Fitness funkcija implementirana je na sljedeći način:

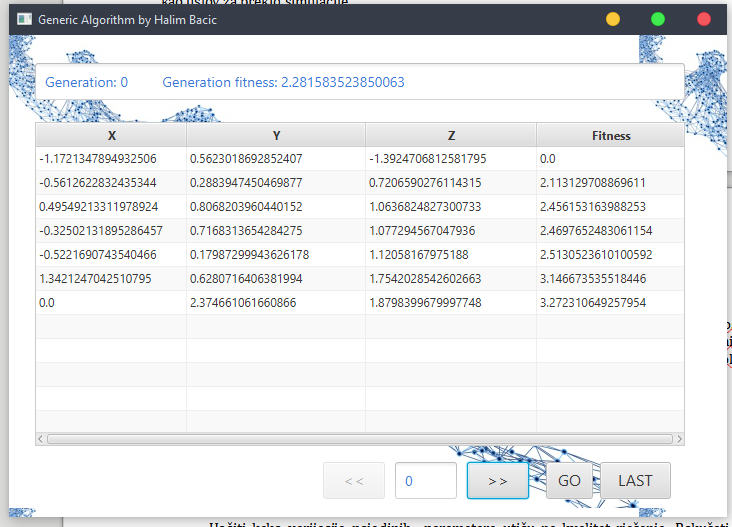
protected void fitness() {  
 for(T p : population**.**getPopulation())  
 {  
 Double ft = p**.**getZ()-population**.**getPopulation()**.**get(0)**.**getZ();  
 p**.**setFitness(ft);  
 }  
}

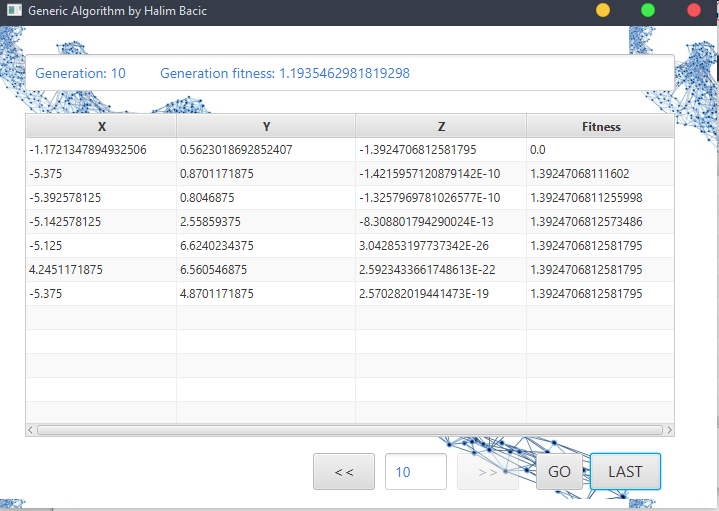
Korak 2 se ponavlja sve dok se ne izvrši dati broj puta ili dok se ne ispuni uslov o “toleranciji”. Naime, ukoliko se najbolji element ne mijenja ni nakon t iteracija, algoritam se prekida. Vrijednost tolerancije je cjelobrojna vrijednost koju unosi korisnik, te je poželjno da vrijednost tolerancije ne bude značajno manja od vrijednosti broja iteracija. Ukoliko je tolerancija značajno manja od vrijednost broja iteracija, potencijalno se dobija lošije rješenje.

**Tok eksperimenta:** Posmatrajmo svaku generaciju počevši od inicijalne mjenjajući 5 parametara koji uključuju veličinu populacije (n), broj iteracija (it), šanse za ukrštanje (c), mutaciju (m) i parametar tolerancije (t) koji označava potreban broj generacija u kojima je najbolji element jednak kao uslov za prekid simulacije.

*1. Eksperiment – n = 7 c=0.8 m=0.2 it=20 t=10*

Cilj ovog izbora parametara je da je početna populacija relativno malena, a uprkos velikoj šansi ukrštanja i velikom broju iteracija, nećemo postići odgovarajući rezultat minimalne vrijednosti. Naime, broj jedinki je malen te slučajno generisana rješenja nemaju dovoljan broj jedinki da bi se ukrštanjem, bez mutacija postigla značajno bolja rješenja.

 Slika 2.1 *Prikaz prve generisane populacije*

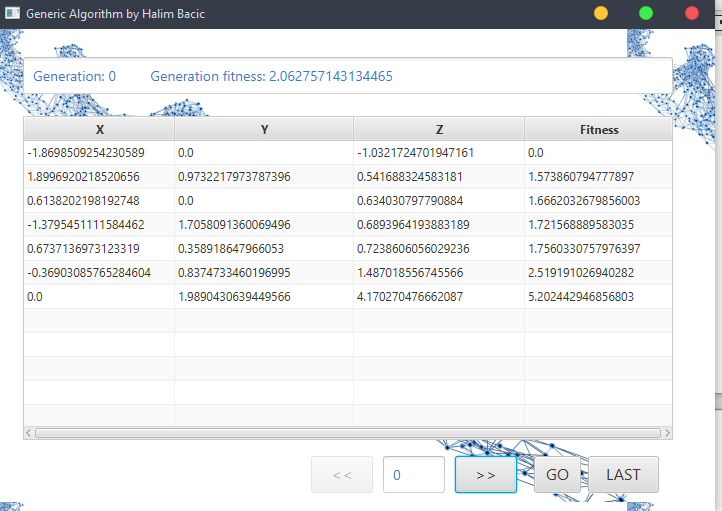
**

Slika 2.2 *Prikaz posljednje generisane populacije*

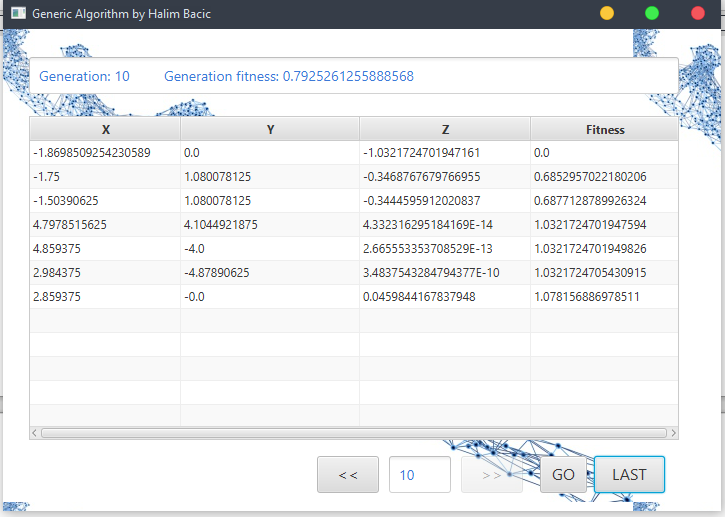
Primjetimo da je posljednja generacija bolja, ali bolje rješenje nije pronađeno. Prvi element je po funkciji ukrštanja trebao biti ukršten sa svim ostalim elementima ali bolje rješenje nije pronađeno.

*2. Eksperiment – n = 7 c=0.8 m=0.9 it=20 t=10*

Ukoliko povećamo mogućnost mutacije, povećava se ukupna ocjena populacije, ali se najbolje rješenje ne poboljšava, kao ni u 1. eksperimentu.

****

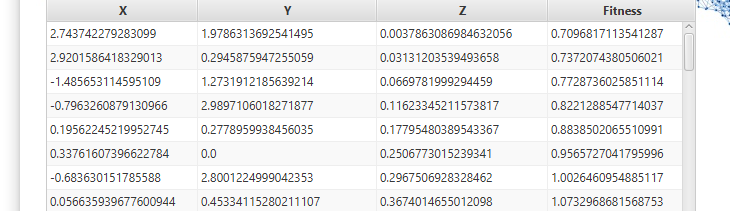
Slika 2.3 *Prva generacija ako ne m>0.5*

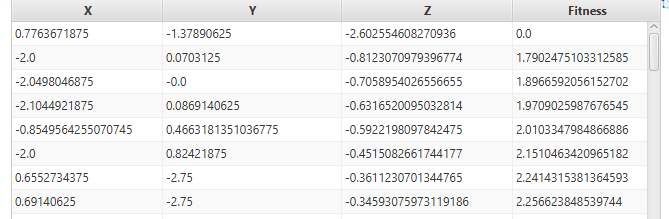
****

Slika 2.4 *Posljednja generacija ako je m>0.5*

*3. Eksperiment – n = 70 c=0.2 m=0.2 it=20 t=10*

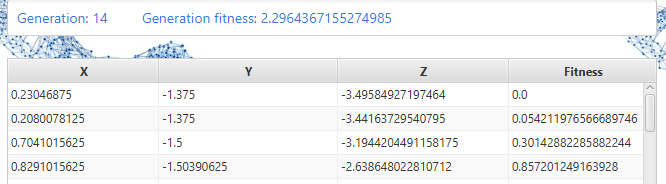
Ukoliko zahtjevamo veći broj jedinki u generaciji, rješenje je bolje nego u prvom slučaju te imamo raznovrsniju populaciju. U ovom eksperimentu je početna populacija imala samo 2 negativna rješenja funkcije, konkretno -0.592 i -0.305. Posljednja generacija je uprkos tome imala sva negativna rješenja od kojih je najbolja vrijednost bila -2.602.

 Slika 2.5 *Početni elementi prve generacije*

Slika 2.6 *Posljednja generacija*

*4. Eksperiment – n = 70 c=0.6 m=0.8 it=90 t=10*

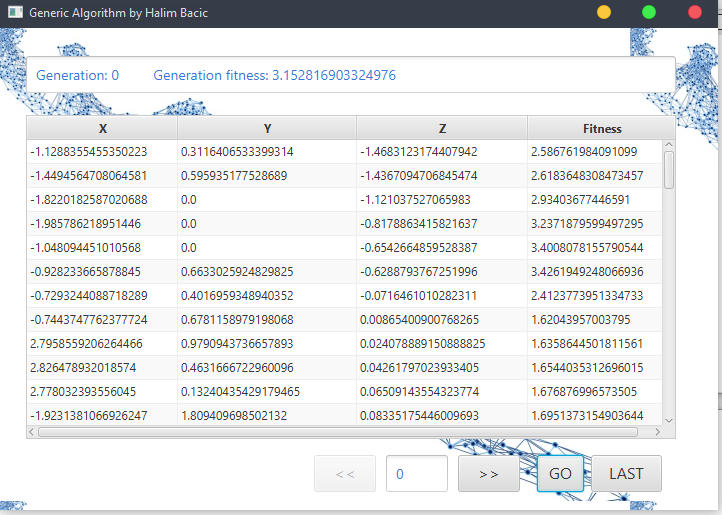
Povećanjem broja iteracija i povećanjem šanse za mutaciju I ukrštanje, najbolje rješenje je -3.49, međutim, uzastopnim ponavljanjem eksperimenta sa podjednakim uslovima, dobijala su se i bolja rješenja. Zanimljivo je primjetiti da je razlika između najboljih rješenja znatno manja nego u predhodnom eksperimentu što se može vidjeti na slici 2.7.



Slika 2.7 *Posljednja generacija eksperimenta 4*

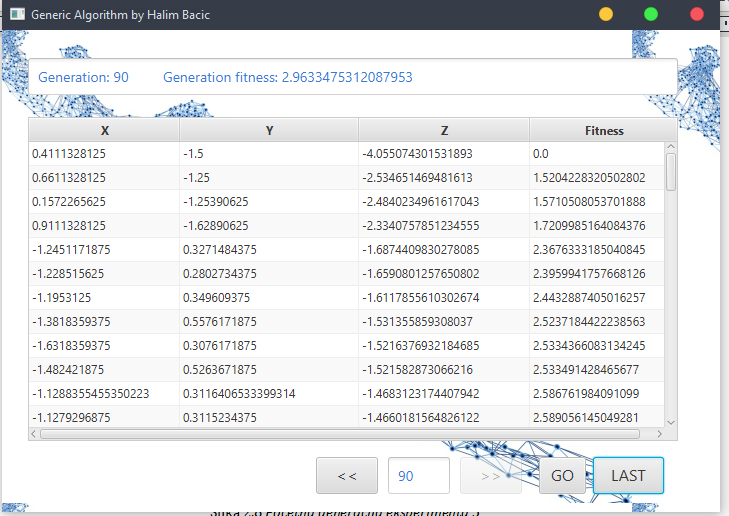
*5. Eksperiment – n = 70 c=0.6 m=0.8 it=90 t=80*

U eksperimentu 4 posljednja generacija je bila generacija 14 jer je od generacije 4 do generacije 14 najbolje rješenje bilo jednako te je eksperiment završen uprkos velikom broju iteracija. U ovom slučaju, najbolja rješenja su približna 4 uz očiglednu razliku između inicijalno najboljeg rješenja i rješenja iz posljednje generacije.

****

Slika 2.8 *Početna generacija eksperimenta 5*

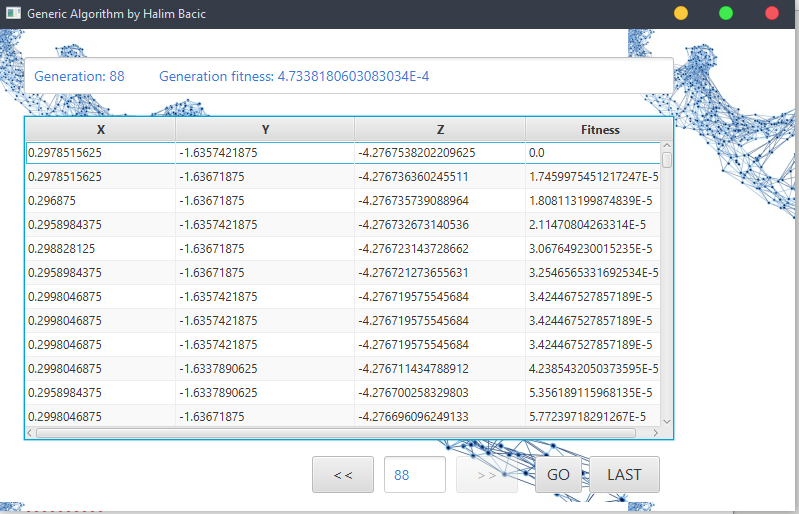
Slika 2.8 *Prva generacija eksperimenta 5*

****

Slika 2.9 *Posljednja generacija eksperimenta 5*

*6. Eksperiment – n = 200 c=0.7 m=0.8 it=90 t=80*

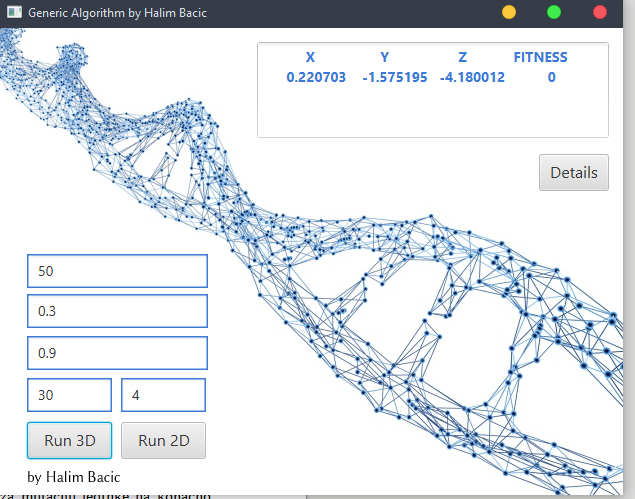
Ukoliko smo birali navedene vrijednosti parametara, vrijednost minimuma funkcije je prilikom svakog narednog izvođenja eksperimenta bila je približna -4.27. Preciznije, iako je početna generacija za vrijednosti x i y imala širok interval za vrijednost z od -3.57 do 5.57, u posljednjoj generaciji je vrijednost z za svih 200 članova generacije bila -4.27, a razlika se pojavljuje na 5 decimali, što se može vidjeti na prikazu sa slike.

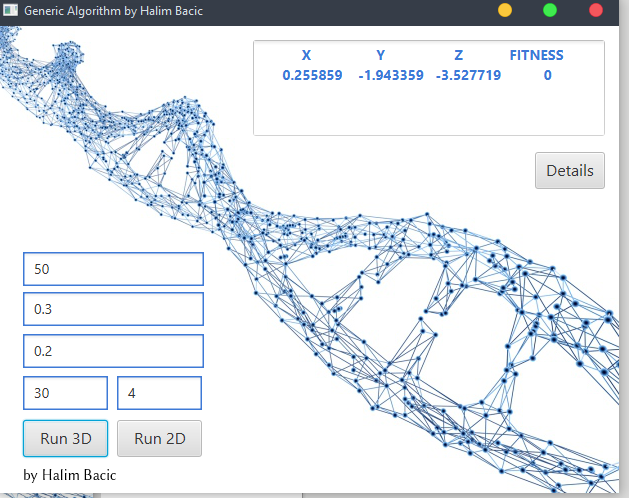
Slika 2.10 *Najbolje jedinke posljednje generacije*

*7. Eksperiment – n = 50 c=0.3 m=0.7 it=30 t=10*

*8. Eksperiment – n = 50 c=0.3 m=0.2 it=30 t=10*

Da bi smo dobili uvid u to kako utiče promjena šansi za mutaciju jedinke na konačno rješenje, dovoljno je da m postavimo na 0.7 u 7 eksperimentu, a 0.2 u 8 ponavljanju eksperimenta. Primjećujemo da je tražena vrijednost bila znatno manja ukoliko je manja vrijednost mutacije čime je potvrđeno razmišljanje da mutacija može bitno da utiče na tačnost rješenja.

****

****

Slika 2.11 *Prikaz uticaja mutacije na rješenje*

*9. Eksperiment – n = 70 c=0.6 m=0.8 it=90 t= 50*

Ako posmatramo 4. ponavljanje simulacije, možemo doći do zaključka da se povećanjem parametra tolerancije, algoritam izvršava duži vremenski period, ali dobijamo bolje rješenje problema. Za razliku od 4. ponavljanja, kada je t parametar bio značajno manji, kao i vrijeme izvršavanja, za veću vrijednost parametra t, dobili smo odgovarajuće rješenje.

*10. Eksperiment – Optimizacija*

Da bi smo dobili optimalne vrijednosti parametara, potrebno je da postignemo što kraće vrijeme izvršavanja uz što veću preciznost. Ukoliko posmatramo predhodna izračunavanja, zaključujemo da je traženo rješenje približno -4.27. Kompleksnost algoritma će biti najbolja ukoliko imamo što manje vrijednosti parametara. Uzastopnim ponavljanjem eksperimenta te korigovanja datih parametara, nameće se zaključak da se najbolja rješenja dobijaju ukoliko je n = 30. c = 0.3, m=0.3, it=30, t=18. Ukoliko su parametri bili približno jednakih vrijednosti datima, rješenje je konstantno bilo oko -4.20. Pri tome, ovi parametri su dali odličnu kolektivnu vrijednost krajnje populacije koja je bila konstantno ispod 0,1. Zanimljivo je da je u većini slučajeva, poprilično precizna vrijednost sa sobom donosila i manji broj generacija od maksimalnih 30 čime je izvršavanje simulacije bilo ubrzano.

Ukoliko su se navedeni eksperimenti ponavljali i sa drugim dijelom zadatka, dobijani su slični rezultati koji su grafički predstavljeni u nastavku.

# 4. Zaključak

Nakon urađenih eksperimenata, zaključak je da genetički algoritam daje nepogrešivo tačne rezultate za veći broj iteracija i veći broj jedinki. Ukolliko bi smo problem rješavali nekim od klasičnih algoritama, gdje bi smo provjeravali sve kombinacije, trebalo bi znatno više vremena nego korištenjem genetičkog algoritma. Pravilnom implementacijom funkcije za ukrštanje i mutiranje, broj iteracija i broj jedinki bi mogao biti znatno manji, ali uz veće šanse za ukrštanje i mutiranje novih jedinki. Obzirom da vrijednosti početne populacije zavise od slučajno generisanih tački iz intervala, bolji rezultati bi se dobijali ukoliko ponavljamo eksperiment sa istim vrijednostima parametara. Bolja ukupna ocjena populacije se dobijala u slučaju da je broj iteracija veći od broja jedinki u populaciji. Za optimalne vrijednosti, pored veoma brzog izvršavanja algoritma, ukupna ocjena populacije je odlična. Sa druge strane, pogrešnim izborom parametara, što se vidi u prvih nekoliko izvođenja eksperimenata, rezultati su znatno lošiji.

Ukoliko dodatno želimo da poboljšamo brzinu izvršavanja, moguće je da postavimo ograničenje u vidu tolerancije. Taj parametar prekida algoritam ukoliko je najbolje rješenje identično u posljednjih n iteracija.