Studija algortima vučjeg čopora i njegovo poređenje sa genetskim algoritmom

Sadržaj

1. Uvod

Tehnike optimizacije bazirane na meta-heuristikama su u poslednje 3 decenije postale prilično popularne. Toliko da su neke najpopularnije (GA [1], ACO [2], PSO [3]) poznate ne samo naučnicima iz računarske sfere, već i drugim naučnicima iz širokog spektra specijalizacija. U ovom polju postoji veoma veliki broj teorijskih istraživanja, a pored toga ima i ogroman broj aplikacija ovih tehnika na realnim problemima. Odatle proističe pitanje, šta ove tehnike čini toliko moćnima? Objašnjenje bi moglo da se da pripiše određenim atributima koji ih razdvajaju. Ti atributi su: jednostavnost, fleksibilnost, sposobnost izbegavanja lokalnih optimuma, činjenica da se ne oslanjaju na izvod funkcije kako bi je optimizovali.

Meta-heurisitke su suštinski jednostavne. One crpe inspiraciju iz prirode, odnosno ponašanja životinja, fizičkih fenomena ili evolutivnih koncepata. Jednostavnost omogućava relativno lako kombinovanje različitih pristupa, praveći hibridne meta-heuristike koje mogu da imaju izuzetan aplikativni potencijal. Još jedna prednost ove jednostavnosti je što ih je lako razumeti, tako da i neko ko ima relativno malo iskustva je u stanju da implementira gotovo svaku meta-heuristiku ako razume principe njenog rada.

Drugi atribut koji čini meta-heurisitke ovako snažnim je fleksibilnost. Ovaj atribut je možda i glavni razlog zašto se ove tehnike toliko često pojavljuju u praksi. Naime, meta-heuristike se razvijaju tako da mogu da reše proizvoljan optimizacioni problem. Naravno, postoje ograničenja (hiper-parametri) koji moraju da se ispoštuju, ali je generalno svaka meta-heuristika “crna kutija” kojoj se prosledi problem i ona vrati rešenje, ma šta taj problem bio.

Naredni atribut je sposobnost izbegavanja lokalnih optimuma. U tehnikama kao što je gradijentni spust ovo je nemoguće uraditi, već je jedini način oslanjati se na sreću. Ovo je naravano krajnje nepoželjno ponašanje. Meta-heuristike, iako nisu imune na lokalne optimume, sadrže najrazličitije tehnike njihovog izbegavanja. Same tehnike izbegavanja se razlikuju od algoritma do algoritma.

Ne oslanjaju se na izvod funkcije kako bi je optimizovali. Tehnike kao što je gradijentni spust i slične se oslanjaju na izvod funkcije kako bi našli optimum. Ovo može biti problem s obzirom da funkcije u nekim tačkama nemaju definisan izvod, sama ta činjenica čini te tehnike beskorisnim u tim slučajevima. Meta-heursitike ovo prevazilaze svojom stohastičkom prirodom. Njihova početna rešenja se inicijalizuju na slučajan način, a potom se nekom logikom modeluju nove generacije.

Ipak, meta-heuristike nisu svemoguće i iako rešavaju veliki broj problema, nekad je teško uočiti način da se primene. Čak iako se uoči potencijal za primenu meta-heuristike, ostaje pitanje: Koju primeniti? Ovo može da bude problem s obzirom da različite meta-heuristike mogu da se ponašaju na najrazličitije načine za isti problem. Pored toga, ako se i zaključi optimalna meta-heuristika za problem od interesa, može da bude težak i mukotrpan proces podesiti sve njene parametre kako bi se problem rešio na zadovoljavajući način. Ovde je vredno spomenuti takozvanu “teoremu o nepostojanju besplatnog ručka” (eng. *No Free Lunch,* u daljem tekstu *NFL)* [4], koja suštinski govori o ovome.

Postoji generalna podela meta-heuristika na dve klase. Tehnike koje dolaze do rešenja krećući od jednog potencijalnog rešenja i unapređujući ga kroz životni ciklus optimizacije. Ovakve tehnike se nazivaju tehnike jednog rešenja (eng. *single-solution-based).* Pored njih postoje i populacione tehnike (eng. *population-based)*. Ove tehnike utilizuju proizvoljan broj potencijalnih rešenja u svakoj generaciji, kombinuju ta postojeća rešenja i od njih proizvode nove kandidate.

Populacione tehnike imaju određene prednosti u odnosu na tehnike jednog rešenja. Te prednosti su:

- Podela informacija između sebe, omogućavajući brže skokove jedinki ka delu prostora koji može biti od posebnog interesa.
- Međusobno asistiranje kako bi uspešnije izbegli lokalne optimume.
- Generalno bolja pretraga prostora od interesa u odnosu na tehnike jednog rešenja.

Jedna podkategorija populacionih tehnika su tehnike koje se jednim imenom nazivaju tehnikama “inteligencije roja” (eng. *Swarm Intelligence,* dalje tekstu *SI),* prvi put predstavljena 1993 [26]. Inteligencija roja predstavlja grupu agenata, gde svaki agent može da sadrži informaciju koja može da utiče na kretanje ostalih agenata. Inspiracija za *SI* tehnike je potekla iz prirodnih gupacija (kolonija, čopora, jata, itd.). Neke od prednosti *SI* tehnika su sledeće:

- Čuvaju informacije o prostoru pretrage tokom iteracija, dok evolutivni algoritmi ne pamte te podatke.
- Najčešće čuvaju najbolje dotadašnje rešenje u svakom trenutku.
- Imaju manje parametara za podešavanje.
- Imaju manje operacija unutar sebe u odnosu na evolutivne pristupe (mutacija, reprodukcija, itd.)
- Jednostavne za implementaciju.

No, bez obzira na vrstu meta-heuristike sve sadrže fazu pretrage i fazu eksploatacije. Meta-heuristike pokušavaju da nađu balans između ove dve faze [5 - 7].

U fazi pretrage algoritam pokušava da obiđe što veći deo prostora pretrage kako bi maksimizovao šansu da naiđe baš na globalni optimum. Ako algoritam ne pretraži dovoljno prostora pretrage, postoje velike šanse da upadne u lokalni optimum (pogotovo ako funkcija sadrži puno lokalnih optimuma). Ovo se dešava iz razloga što nije imao informacije da se van njemu poznatog prostora nalazi možda bolje rešenje.

U fazi eksploatacije algoritam pokušava da na što bolji način istraži deo prostora pretrage koji je označen kao interesantan. Eksploatacija je lokalna pretraga nekog dela prostora koji je označen kao posebno interesantan.

Ovaj rad se dalje bavi sledećim: U narednoj sekciji (2) je detaljno objašnjen *GWO* algoritam i data njegova implementacija. U sekciji 3 je objašnjen genetski algoritam, kao algoritam koji će služiti za poređenje sa *GWO.* U sekciji 4 su predstavljene test funkcije. U sekciji 5 je prikazan rezultat *GWO* algoritma sa

različitim parametrima, primenjenim na funkcije iz sekcije 4. U Sekciji 6 je dato poređenje rezultata između genetskog i *GWO* algoritma. Na kraju, u sekciji 7 je dat zaključak i mogući pravci daljeg istraživanja.

2. Optimizacija algoritmom vučjeg čopora (*Grey wolf optimization*)

U ovoj sekciji je najpre diskutovana motivacija algoritma vučjeg čopora (eng. *grey wolf optimization*) [8], potom je objašnjen matematički model. Na kraju je data implementacija samog algoritma.

2.1. Motivacija

Sivi vuk je jedna od najsposobnijih životinja na svetu. Nalaze se na samom vrhu lanca ishrane i u stanju su da savladaju plen koji je veći od njih samih. Ova vrsta živi u čoporima koji su prosečne veličine od 5 do 12 vukova. Čopori su hijerarhijski organizovani i razlikujemo alfa, beta, delta i omega vukove. Senioritet vukova je u redu navedenom u prethodnoj rečenici. Odnosno alfa vukovi su nadređeni beta vukovima, beta vukovi su nadređeni delta vukovima, itd. Svaki rang ima pravo da traži poslušnosti svih vukova ispod njegovog ranga (ne mora biti direktno nadređen). Alfa i beta vukovi su odgovorni za lov i generalno vođenje čopora. Delta vukovi čuvaju teritoriju, brinu o bolesnima i povređenima i ponekad učestvuju u lovu. Omega vukovi služe kako bi ostali vukovi iskalili bes nad njima. Iako naizgled nepotrebni, bez njih čopor teži nestabilnosti.

Vukovi ne dobijaju rang samo po snazi, već i po sposobnosti da donose odluke.

Pored socijalne hijerarhije sivi vukovi imaju i izuzetno zanimljiv model lova, koji im omogućava da savladaju i višestruko veće životinje od njih samih. Sam lov se može podeliti u 3 faze [9]:

- Praćenje, vijanje i prilaženje žrtvi
- Gonjenje, opkoljavanje i iscrpljivanje žrtve dok ne prestane da se kreće
- Napad na žrtvu

Ovaj algoritam kombinuje socijalnu hijerarhiju čopora i strategiju lova i na osnovu toga pravi matematički model koji je u stanju da vrši optimizaciju.

2.2. Matematički model

U ovaj podsekciji je objašnjen matematički model *GWO* algoritma i način na koji on dolazi do rešenja.

2.2.1. Društvena hijerarhija

Kako bi modelovali društvenu hijerarhiju čopora vukova, u ovom algoritmu izdvajamo 3 najbolja rešenja i na osnovu njih se krećemo. Najbolje rešenje je nazvano alpha (α), drugo najbolje rešenje beta (β) i treće najbolje rešenje delta (δ). Omega vukovi se po prostoru kreću na osnovu ove 3 najbolje vrednosti i na taj način se radi optimizacija.

2.2.2. Opkoljavanje žrtve

Kao što je već spomenuto, prilikom lova čopor teži da opkoli žrtvu. Ovo ponašanje je matematički modelovano sedećim formulama:

$$\vec{D} = \left|\vec{C} \cdot \vec{X}p(t) - \vec{X}(t)\right| \quad (2.1.)$$

$$\vec{X}(t+1) = \vec{X}p(t) - \vec{A} \cdot \vec{D} \quad (2.2.)$$

Gde *t* predstavlja trenutnu iteraciju algoritma, $\vec{A}$ i $\vec{C}$ su vektori koeficijenata, $\vec{X}p$ je vektor pozicije žrtve, a $\vec{X}$ označava poziciju vuka. Dimenzionalnost optimizacionog problema određuje dimenzije vektora. Množenje se vrši na nivou dimenzije (*pairwise*). Za $\vec{X}p$ uzimamo neko od najboljih rešenja za koje se smatra da imaju najbolju ideju gde je žrtva (ne znamo gde je optimum, ali pretpostavljamo na osnovu trenutno najboljih rešenja), pogledati podsekciju *Lov*.

Vektori $\vec{A}$ i $\vec{C}$ se dobijaju na sledeći način:

$$\vec{A} = 2\vec{a} \cdot \vec{r1} - \vec{a} \quad (2.3.)$$

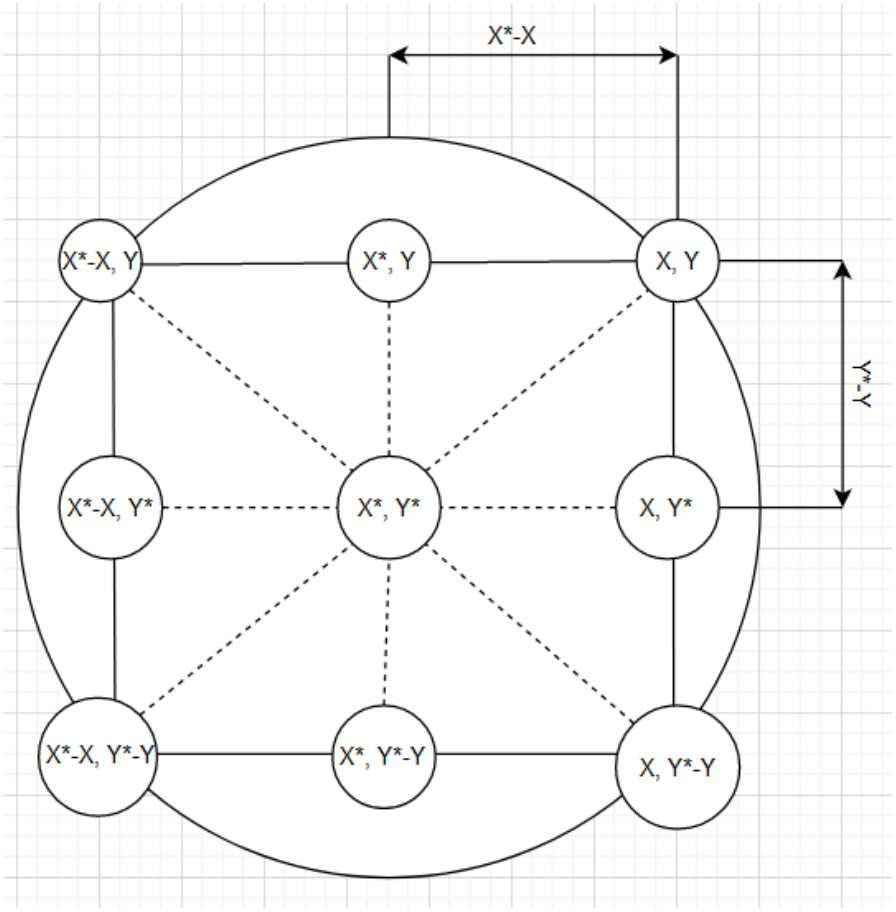
$$\vec{C} = 2 \cdot \vec{r2} \quad (2.4.)$$

Gde komponente vektora $\vec{a}$ bivaju linerano smanjivane u rasponu od 2 do 0, na osnovu broja trenutne iteracije. Vektori $\vec{r1}$ i $\vec{r2}$ su vektori čiji članovi su slučajne vrednosti u rasponu [0, 1].

Položaj vuka i neke od njegovih daljih mogućih pozicija su prikazane na slici 2.1. Kao što je prikazano na slici, vuk na položaju (X, Y) može da promeni svoj položaj na osnovu položaja žrtve (X*, Y*). Ovo se postiže tako što se na trenutnu poziciju uvrste različite vrednosti $\vec{C}$ i $\vec{A}$.

Treba obratiti pažnju da vuk može da se nađe i na bilo kojoj poziciji između prikazanih pozicija slučajnim menjanjem vektora $\vec{r1}$ i $\vec{r2}$. Što znači da vuk može da promeni svoj položaj u bilo kom pravcu unutar perimetra položaja žrtve.

Analogno se može posmatrati kretanje vuka u proizvoljnom broju dimenzija, samo što bi se u tom slučaju kretali u hiper-sferama oko najboljeg trenutnog rešenja.



Slika 2.1. Vuk (pozicija (X,Y)) i neki od njegovih mogućih budućih položaja u 2D prostoru.

2.2.3. Lov

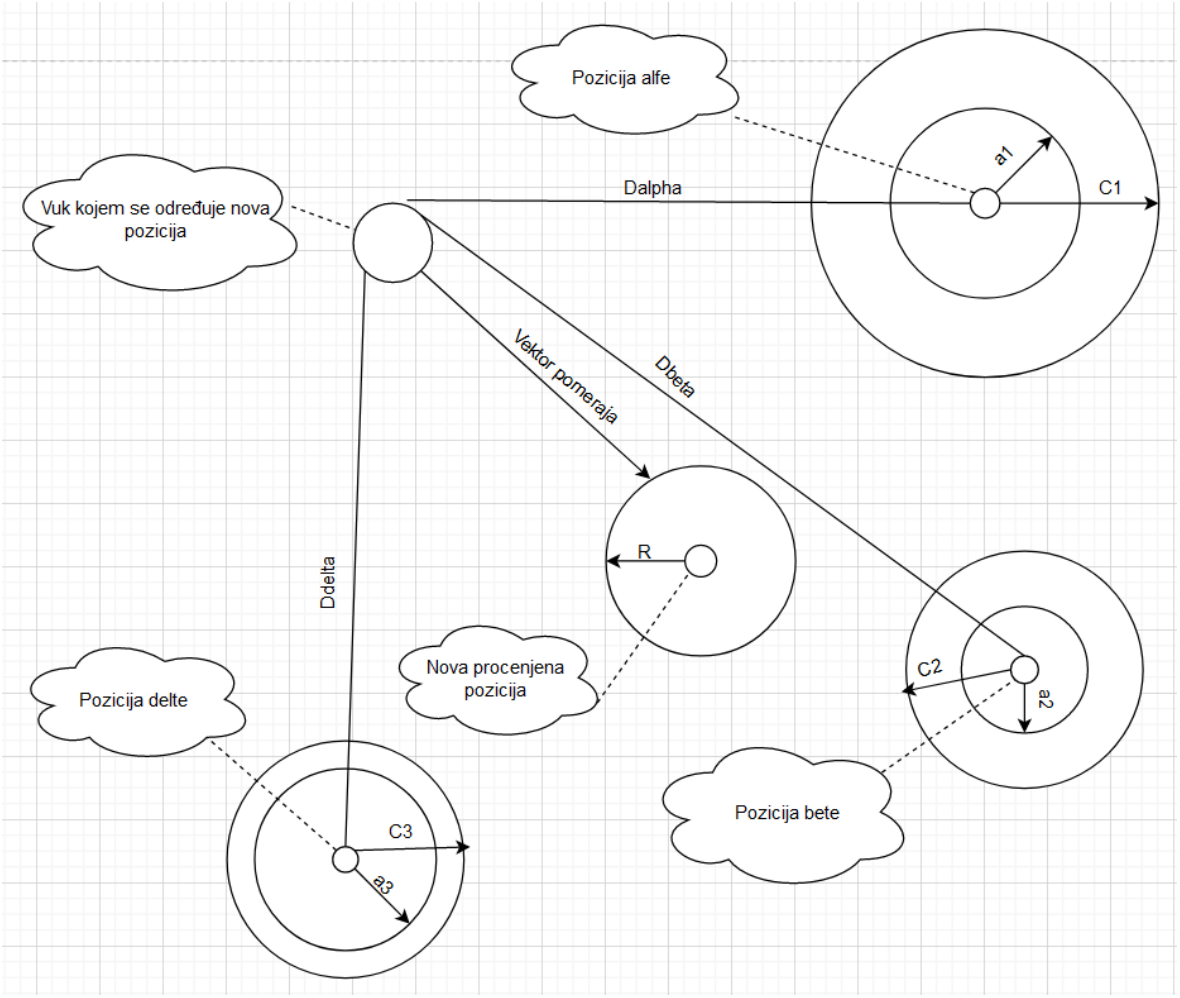
Vukovi imaju sposobnost da lociraju žrtvu i opkole je. Lov je najčešće rađen od strane alfa vuka, mada beta i delta vukovi mogu da učestvuju. Ipak, u apstraktnom prostoru pretrage mi ne znamo gde se žrtva nalazi. Kako bi ovo prevazišli, pretpostavaljamo da alfa (trenutno najbolje rešenje), beta (drugo najbolje rešenje) i delta (treće najbolje rešenje) vukovi imaju bolje informacije o lokaciji žrtve u odnosu na ostale vukove, te šaljemo signal ostalim vukovima da se kreću u pravcu alfe, bete i delte. Simuliranje ovog ponašanja je modelovano sledećim formulama:

$$\vec{D}\alpha = \left|\vec{C}1 \cdot \vec{X}a - \vec{X}\right|, \vec{D}\beta = \left|\vec{C}2 \cdot \vec{X}\beta - \vec{X}\right|, \vec{D}\delta = \left|\vec{C}3 \cdot \vec{X}\delta - \vec{X}\right| \quad (3.5.)$$

$$\vec{X}1 = \vec{X}a - \vec{A}1 \cdot \vec{D}a, \vec{X}2 = \vec{X}\beta - \vec{A}2 \cdot \vec{D}\beta, \vec{X}3 = \vec{X}\delta - \vec{A}3 \cdot \vec{D}\delta \quad (3.6.)$$

$$\vec{X}(t+1) = \frac{\vec{X}1 + \vec{X}2 + \vec{X}3}{3} \tag{3.7.}$$

Slika 2.2. prikazuje kako agent uzima novu poziciju na osnovu položaja alfe, bete i delte u dvodimenzionalnom prostoru. Može se primetiti da će nova pozicija vuka biti slučajna pozicija u perimetru (granicama) određenim na osnovu 3 najbolja rešenja u tom trenutku. Odnosno alfa, beta i delta odrede prostor u kojem se žrtva verovatno nalazi, a potom se ostali vukovi pozicioniraju unutar tog prostora na slučajne pozicije kako bi opkolili žrtvu.



Slika 2.2. Ažuriranje pozicija u *GWO* algoritmu.

2.2.4. Napadanje žrtve (eksploatacija prostora od interesa)

Kao što je već spomenuto, vukovi napadaju žrtvu onda kad ona prestane da se kreće. Ovo se modeluje smanjenjem parametra $\vec{a}$ srazmerno broju iteracija. Uočiti da se i površina $\vec{A}$ smanjuje smanjem $\vec{a}$. Odnosno $\vec{A}$ je slučajna vrednost u

opsegu [- $\vec{a}$, $\vec{a}$] gde je $\vec{a}$ u opsegu 2 i 0 smanjivano srazmerno trenutnoj iteraciji. Kada je vrednost $\vec{A}$ između -1 i 1 sledeća pozicija agenta može da bude bilo koja pozicija između njegove trenutne poziciji i žrtve.

Tehnike dosad objašnjene pokazuju da vukovi menjaju svoj položaj na osnovu alfa, beta i delta vukova, odnosno 3 trenutno najbolja rešenja. Takođe objašnjavaju na koji način vukovi napadaju žrtvu. Ipak, prisutan je problem stagnacije u lokalnim optimumima. Iako tehnika opkoljavanja žrtve podržava pretragu prostora u samoj okolini žrtve, to nije dovoljno zato što može da se desi da se veliki deo prostora pretrage izignoriše. Rešenje ovog problema je dato u narednoj podsekciji.

2.2.5. Potraga za žrtvom (pretraga prostora)

Potraga vukova za žtvom je gotovo u potpunosti diktirana na osnovu položaja alfe, bete i delte. Oni divergiraju jedan od drugog kako bi vršili pretragu prostora, a potom konvergiraju kako bi napali žrtvu. Matematički modelujemo divergenciju tako što koristimo slučajnu vrednost $\vec{A}$ kada je veca od 1, odnosno manja od -1 kako bi naterali agenta da divergira od žrtve. Ovo omogućava algoritmu da vrši globalnu pretragu pogotovo u ranim iteracijama.

Još jedna komponenta GWO algoritma koja pospešuje pretragu je $\vec{C}$. Kao što se može videti u jednačini 2.4., $\vec{C}$ vektor sadrži slučajne vrednosti u ospegu [0, 2]. Ova komponenta služi kao slučajna težina kako bi povećala ($\vec{C}$ > 1), odnosno smanjila ($\vec{C}$ < 1) uticaj žrtve prilikom definicije distance u jednačini 2.1. Ovo pospešuje GWO algoritam na slučajno ponašanje tokom optimizacije, ohrabrujući pretragu i izbegavanje lokalnih optimuma. $\vec{C}$ se ne smanjuje linearno sa brojem iteracija, kao što se $\vec{A}$ smanjuje. Namerno je ostavljeno da $\vec{C}$ pruža slučajne vrednosti tokom celog trajanja optimizacije kako bi se ohrabrila pretraga ne samo tokom inicijalnih iteracija već i kasnije. Drugim rečima ovo sprečava stagnaciju algoritma na lokalnom optimumu, pogotovo u kasnim iteracijama, odnosno kada $\vec{a}$ teži nuli.

Komponentu $\vec{C}$ generalno možemo smatrati kao prepreke u prirodi (reke, planine, slabo prohodne terene, itd.). Ove prepreke su česte i pojavljuju se prilikom lova, sprečavajući vukove da se kreću zamišljenom putanjom. Ovo je praktično u potpunosti preslikano na komponentu $\vec{C}$, gde onda može dati žrtvi slučajnu težinu i tako je napraviti nepristupačnom za vuka. Sa druge strane može se desiti da žrtva dobije takvu težinu da je vuku prilično lako da je dosegne.

2.2.6. Sažetak

Proces pretrage počinje sa slučajnim vrednostima svakog vuka, koje se tretiraju kao moguća rešenja. Tokom iteracija alfa, beta i delta vukovi (najbolja rešenja) procenjuju lokaciju žrtve. Ostali vukovi se pomeraju na novu lokaciju na

osnovu te estimacije. Parametar $\vec{a}$ se linearno smanjuje od 2 do 0 kako lov odmiče pospešujući pretragu, odnosno eksploataciju. Vukovi divergiraju od žrtve kada je $|\vec{A}$ > 1|, odnosno konvergiraju kad je $|\vec{A}$ < 1|. Uslov za završetak algoritma je zadovoljavanje uslova (npr. zadovoljen broj iteracija).

Sledeće teze prikazuju kako GWO teorijski rešava problem optimizacije:

- Diskutovana socijalna hijerarhija čopora čuva najbolje pronađeno rešenje do tog trenutka.
- Mehanizam kojim vukovi opkoljavaju žrtvu definiše krug (sferu, hiper-sferu u višim dimenzijama).
- Slučajni parametri $\vec{A}$ i $\vec{C}$ omogućavaju agentima da imaju sferu različitog (slučajnog) obima oko rešenja.
- Metod lova omogućava agentima da pretpostave moguć položaj žrtve.
- Pretraga i eksploatacija je garantovana na osnovu adaptivnih vrednosti $\vec{a}$ i $\vec{A}$.
- Adaptacija $\vec{a}$ i $\vec{A}$ vrednosti kako odmiče algoritam favorizuje eksploataciju u odnosu na pretragu.
- Kako se $\vec{A}$ smanjuje, polovina iteracija se fokusira na eksploataciju a polovina na pretargu.
- Algoritam manipulše sa samo 2 parametra ($\vec{a}$ i $\vec{C}$)

Pseudokod algoritma se nalazi na slici 2.3.

Initialize the grey wolf population X_i *(i = 1, 2, ..., n)*
Initialize a, A, and C
Calculate the fitness of each search agent
X_α*=the best search agent*
X_β*=the second best search agent*
X_δ*=the third best search agent*
while *(t < Max number of iterations)*
    for *each search agent*
        Update the position of the current search agent by equation (3.7)
    end for
    Update a, A, and C
    Calculate the fitness of all search agents
    Update X_α, X_β, *and* X_δ
    t=t+1
end while
return X_α

Slika 2.3. Pseudo kod *GWO* algoritma [8].

2.3. Implementacija

U ovoj podsekciji je data implementacija algoritma u programskom jeziku python.

2.3.1. Korišćene biblioteke

Korišćeni su standardni python moduli random, math i copy. Pored standardnih modula, korišćene su i biblioteke numpy i seaborn. Na slici 2.4. je prikazan kod koji importuje spomenute module i biblioteke.

```
1  import random
2  import numpy as np
3  import math
4  import copy
5  import seaborn as sns
```

Slika 2.4. Import navedenih biblioteka

2.3.1.1. Modul random

Modul *random* je standardni deo python programskog jezika korišćen za generisanje pseudo slučajnih vrednosti. Kompletan modul i njegove mogućnosti su dostupne u *python* dokumentaciji [10].

2.3.1.2. Modul math

Modul *math* je standardni deo python programskog jezika korišćen za razne matematičke operacije, poput sinusa, kosinusa, pi vrednosti, itd. Kompletan modul i njegove mogućnosti su dostupne u *python* dokumentaciji [11].

2.3.1.3. Modul copy

Modul *copy* je standardni deo python programskog jezika korišćen za kopiranje objekata. Podržava plitke i duboke kopije. Kompletan modul i njegove mogućnosti su dostupne u *python* dokumentaciji [12].

2.3.1.4. Biblioteka numpy

Numpy biblioteka je vrlo moćan open source alat koji je našao široku primenu kod naučnika. Naime, numpy podržava veoma širok spektar kompleksnih matematičkih operacija kao i operacija manipulacije mnogobrojnih podataka. Posebno je interesantna iz razloga što se kod izvršava efikasnije i brže u odnosu na ugrađene *python* sekvence. Sve mogućnosti *numpy* biblioteke je moguće pronaći u dokumentaciji [13].

2.3.1.5. Biblioteka seaborn

Seaborn je open source biblioteka napravljena nad python bibliotekom *matplotlib* *[16]*. Koristi se za vizualizaciju i istraživanje podataka. Dobro radi sa *dataframe*-ovima i *pandas* *[15]* biliotekom. Posebno je izdvaja što je moguća laka kustomizacija grafika. Sve mogućnosti *seaborn* biblioteke je moguće pronaći u njenon dokumentaciji [14].

2.3.2. Klasa ImportantValue

Klasa *ImportantValue* služi da predstavi vrednost jednog agenta u toku optimizacije. Ona sadrži dva atributa: poziciju i vrednost. Kod ove klase je prikazan na slici 2.5.

Atribut pozicija služi da predstavi poziciju datog agenta u prostoru. Ovaj atribut je niz čija svaka vrednost predstavlja poziciju datog agenta u jednoj dimenziji. Odnosno, prva vrednost niza je pozicija agenta u prvoj dimenziji prostora pretrage, druga vrednost je pozicija u drugoj dimenziji. Analogno važi za proizvoljan broj dimenzija.

Atribut vrednost predstavlja vrednost fitnes funkcije datog agenta na njegovoj trenutnoj poziciji. Ova vrednost se računa tako što se funkciji koja se optimizuje šalju koordinate (pozicija) agenta i ona vraća izračunatu vrednost.

```
1  class ImportantValue:
2      def __init__(self, position, value):
3          self.position = position
4          self.value = value
```

Slika 2.5. Klasa *ImportantValue*

2.3.3. Funkcija UpdateBestValues

Funkcija *UpdateBestValue* služi da ažurira alfa, beta i delta vrednosti kada se izvrši pomeraj vukova. Potpis funkcije se nalazi na slici 2.6. Funkcija prima 5 parametara: *position_of_wolves*, *fitness_function*, *alpha*, *beta*, *delta*.

Parametar *position_of_wolves* predstavlja matricu dimenzija *pack_size* x *number_of_dimensions* čije vrednosti označavaju pozicije svakog pojedinačnog vuka.

Parametar *fitness_function* je callable objekat, odnosno funkcija koja se optimizuje. Služi za računanje fitnesa svakog pojedinačnog vuka.

Parametri *alfa, beta i delta* su instance *ImportantValue* klase. Oni će po potrebi biti ažurirani.

Funkcija iterira kroz sve vukove i računa vrednost fitnes funkcije za svakog. Ukoliko je dobijena vrednost bolja od alfa vrednosti, alfa poprima tu vrednost a prethodna alfa vrednost se šiftuje na beta, koja se šiftuje na delta. Analgno važi ako je novonastala vrednost lošija od alfe, a bolja od bete, odnosno lošija od alfe i bete, a bolja od delte.

```
def UpdateBestValues(positions_of_wolves, fitness_function, alpha, beta, delta):
    for wolf in positions_of_wolves:

            wolfs_fitness = fitness_function(wolf)

            if(wolfs_fitness < alpha.value):
                delta.value = beta.value
                delta.position = beta.position.copy()

                beta.value = alpha.value
                beta.position = alpha.position.copy()

                alpha.value = wolfs_fitness
                alpha.position = wolf.copy()

            if(wolfs_fitness > alpha.value and wolfs_fitness < beta.value):
                delta.value = beta.value
                delta.position = beta.position.copy()

                beta.value = wolfs_fitness
                beta.position = wolf.copy()

            if(wolfs_fitness > alpha.value and wolfs_fitness > beta.value and
               wolfs_fitness < delta.value):
                delta.value = wolfs_fitness
                delta.position = wolf.copy()

```

Slika 2.6. Funkcija *UpdateBestValues*

2.3.4. Funkcija CalculateBestPosition

Funkcija *CalculateBestPosition* služi da izračuna formulu 2.5, takođe je zadužena i za generisanje slučajnih vrednosti *r1* i *r2*, kao i vrednosti *C* i *A*. Ova funkcija prima 4 parametra: *wolf, dim, important_position* i *a*. Implementacija funkcije je data na slici 2.7.

Parametar *wolf* predstavlja vuka u odnosu na kog se računa distanca.

Parametar *dim* označava u kojoj dimenziji se trenutno radi.

Parametar *important_value* predstavlja trenutnu dobru estimaciju položaja žrtve.

Parametar *a* služi za računanje *A* i *C*, u glavnoj petlji mu je dodeljena vrednost na osnovu trenutne iteracije.

```
1  def CalculateBestPosition(wolf, dim, important_position, a):
2      r1 = random.random()
3      r2 = random.random()
4
5      C = 2 * r2
6      A = (2 * a * r1) - a
7
8      Dx = (C * important_position.position[dim]) - wolf[dim]
9      X = important_position.position[dim] - (A * Dx)
10     #print X
11     return X
```

Slika 2.7. Implementacija funkcije *CalculateBestPosition*

2.3.5. Funkcija GWO

Funkcija *GWO* predstavlja funkciju algoritma (ona se poziva za izvršenje algoritma). Potpis funkcije *GWO* se nalazi na slici 2.8. Funkcija *GWO* prima 6 parametara: *pack_size, number_of_dimensions, number_of_iterations, fitness_function, lower_bound, upper_bound*.

Parametar *pack_size* predstavlja broj vukova koji će učestvovati u lovu. Ovaj broj generalno nije ograničen, ali kako algoritam radi sa 3 najbolje vrednosti poželjno da veličina čopora bude veća od 3. Praksa (kako u prirodi, tako i ovde) pokazuje da se algoritam dobro ponaša ako se prosledi vrednost između 5 i 12. Pack_size mora biti ceo broj.

Parametar *number_of_dimensions* predstavlja broj dimenzija u kojem je očekivano da algoritam radi. Predstavlja ceo broj veći od 0.

Parametar *number_of_iterations* predstavlja očekivan broj iteracija koje algoritam treba da ispuni. Predstavlja ceo broj veći od 0.

Parametar *fitness_function* predstavlja callable objekat, odnosno funkciju koju algoritam treba da optimizuje. Potrebno je da prosleđena funkcija prima niz, čije će elemente koristiti za izračunavanje fitnesa trenutnog rešenja, i da vrati rezultat.

Parametar *lower_bound* predstavlja donju granicu prostora pretrage po dimenzijama. Moguće je proslediti broj ili niz brojeva. Ukoliko se prosledi samo broj algoritam će svakoj dimenziji dodeliti istu (prosleđenu) donju granicu, a ukoliko se prosledi niz brojeva, svaka dimenzija će biti ograničena prosleđenim brojem na datoj lokaciji.

Parametar *upper_bound* predstavlja gornju granicu prostora pretrage po dimenzijama. Moguće je proslediti broj ili niz brojeva. Ukoliko se prosledi samo broj algoritam će svakoj dimenziji dodeliti istu (prosleđenu) gornju granicu, a ukoliko se prosledi niz brojeva, svaka dimenzija će biti ograničena prosleđenim brojem na datoj lokaciji.

```
1  def GWO(pack_size, number_of_dimensions, number_of_iterations, fitness_function, lower_bound,
2          upper_bound):
```

Slika 2.8. Potpis *GWO* funkcije.

2.3.5.1. Inicijalizacija vrednosti

U ovoj podsekciji je prikazana inicijalizacija početnog čopora, alfa, beta i delta vrednost kao i donjih i gornjih ograničenja.

Na slici 2.9. je prikazana inicijalizacija alfa, beta i delta vrednosti. Svaka od ovih vrednosti je instanca klase *ImportantValue*. Inicijalna pozicija ovih vrednosti je

postavljena na koordinatni početak prostora pretrage, a inicijalna vrednost na plus beskonačno.

```
11      alpha_position = np.zeros(number_of_dimensions)
12      alpha_value = float('inf')
13      alpha = ImportantValue(alpha_position, alpha_value)
14
15      beta_position = np.zeros(number_of_dimensions)
16      beta_value = float('inf')
17      beta = ImportantValue(beta_position, beta_value)
18
19      delta_position = np.zeros(number_of_dimensions)
20      delta_value = float('inf')
21      delta = ImportantValue(delta_position, delta_value)
```

Slika 2.9. Inicijalizacija alfa, beta i delta vrednosti

Na slici 2.10. je prikazana obrada donje i gornje granice. Odnosno, nakon provere ako nije prosleđen niz vrednosti, kreira se novi i svi članovi mu se inicijalizuju na prosleđen broj. Ako jeste prosleđen niz, za granice se uzimaju respektivne vrednosti za svaku dimenziju.

```
6       if not isinstance(lower_bound, list):
7           lower_bound = [lower_bound] * number_of_dimensions
8       if not isinstance(upper_bound, list):
9           upper_bound = [upper_bound] * number_of_dimensions
10
```

Slika 2.10. Obrada donje i gornje granice

Na slici 2.11. je prikazana inicijalizacija početnih pozicija vukova. Najpre se kreira matrica odgovarajućih dimenzija (*pack_size x number_of_dimensions*) i popunjava nulama. Potom se inicijalizuje matrica popunjavam slučajnim vrednostima po kolonama, gde se uzima ograničenje gornje i donje granice.

```
23      positions_of_wolves = np.zeros((pack_size, number_of_dimensions))
24
25      for i in range(number_of_dimensions):
26          positions_of_wolves[:, i] = np.random.uniform(lower_bound[i], upper_bound[i],
27                                                        pack_size)
28
```

Slika 2.11. Inicijalizacija i popunjavanje matrice položaja vukova

Na slici 2.12. je prikazan poziv funkcije *UpdateBestValues* u kojoj su postavljene vrednosti alfa, beta i delta na osnovu početne pozicije vukova.

```
29      UpdateBestValues(positions_of_wolves, fitness_function, alpha, beta, delta)
```

Slika 2.12. Poziv funkcije *UpdateBestValues*

2.3.5.2. Glavna petlja algoritma

Na slici 2.13. je prikazana glavna petlja algoritma koja se ponavlja prosleđeni broj iteracija. U ovoj petlji se najpre računa *a* na osnovu trenutne iteracije. Potom se za svakog vuka i za svaku njegovu dimenziju računa nova pozicija. Izračunata pozicija se dodaje vuku, s tim što se u liniji 42 vodi računa da gornja i donja granica ostanu zadovoljene. Odnosno ako nova koordinata bude veća od gornje granice, za tu koordinatu se dodeljuje gornja granica. Analogno važi i za donju granicu.

Kada se iterira kroz ceo čopor i kada se dodele nove vrednosti svakom vuku, poziva se funkcija *UpdateBestValues* kako bi se, ako je potrebno, ažurirale alfa, beta i delta vrednosti.

Alfa vrednost na kraju svake iteracije se dodaje u niz konvergencije, koji posle služi za vizuelni prikaz koraka do rešenja.

Funkcija vraća *ImportantValue* objekat i niz konvergencije.

```
31      for iter in range(0, number_of_iterations):
32          a = 2 - (2/number_of_iterations) * iter
33
34          for wolf in positions_of_wolves:
35              for dim in range(number_of_dimensions):
36                  X1 = CalculateBestPosition(wolf, dim, alpha, a)
37                  X2 = CalculateBestPosition(wolf, dim, beta, a)
38                  X3 = CalculateBestPosition(wolf, dim, delta, a)
39
40                  wolf[dim] = (X1 + X2 + X3) / 3
41
42                  wolf[dim] = np.clip(wolf[dim], lower_bound[dim], upper_bound[dim])
43
44
45          UpdateBestValues(positions_of_wolves, fitness_function, alpha, beta, delta)
46          convergency.append(alpha.value)
```

Slika 2.13. Glavna petlja *GWO* funkcije

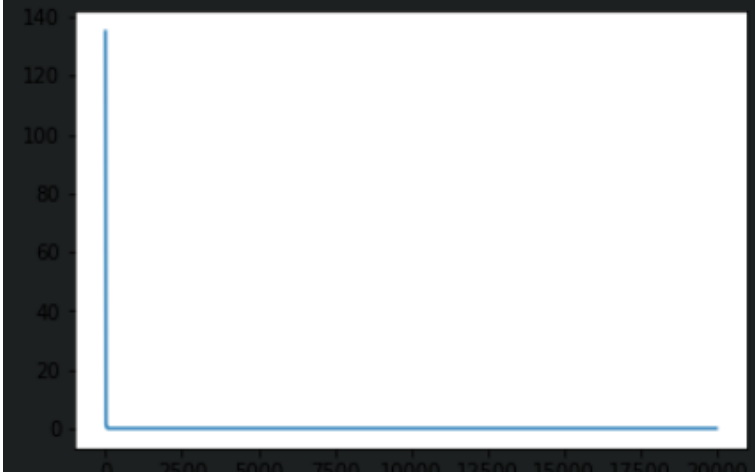
2.3.6. Poziv funkcije algoritma

Na slici 2.14. se vidi poziv funkcije *GWO* sa sledećim parametrima:
pack_size = 10,
number_of_dimensions = 10,
number_of_iterations = 20000,
fitness_function = *griewank*,
lower_bound = -600,
upper_boud = 600.

U ispisu nakon završetka algoritma je prikazana pozicija nađenog optimuma i vrednosti fitnes funkcije u toj tački. Takođe je prikazan grafik konvergencije čija X-osa predstavlja broj iteracija, a Y-osa vrednost fitnes funkcije najboljeg rešenja određene iteracije.

```
best_alpha = ImportantValue([0, 0], float('inf'))

alpha, convergency = GWO(10, 10, 20000, Greiwank, -600, 600)

print(alpha.value)
print(alpha.position)

sns.lineplot(data=convergency)
```

```
0.0
[-9.48655259e-09  1.03622446e-09 -1.30099977e-08 -6.72736635e-09
  6.17756421e-09  2.30118714e-08 -6.91906453e-09  5.51855938e-09
  1.57401082e-08  1.41637742e-08]

<matplotlib.axes._subplots.AxesSubplot at 0x1fd5699ec50>
```



Slika 2.14. Poziv *GWO* funkcije, njen rezultat i grafik konvergencije

3. Genetski algoritam (Genetic algorithm - GA)

U ovoj sekciji je najpre diskutovana motivacija i ideja genetskog algoritma (eng. genetic algorithm), a potom su diskutovane startegije različitih koraka algoritma.

3.1. Motivacija

Genetski algoritam je baziran na Darvinovoj teoriji evolucije. Darvinova teorija evolucije kaže da su na zemlji milionima godina unazad živela izuzetno prosta stvorenja, najvećim delom jednoćelijski organizmi. Ukraštanjem i adaptacijom u periodima od više miliona godina ta stvorenja su se razvijala i dobijala nove atribute. Neke vrste su dominirale (one koje su bile bolje prilaogđene), neke su odumirale (one koje su bile preosetljive/ previše slabe da se staraju o sebi). Takođe određeni atributi su se pokazali korisnim i počeli da se pojavljuju i u drugim vrstama i opstali čak do danas, dok su neki atributi bili beskorisni i ne bi se zadržavali dugo (mada je ovde reč o milionima godina, tako da je dugo i kratko relativno). Pored toga, pojavljivale bi se slučajne deformacije koje su najčešće bile katastrofalne po jedinku, mada moguće je da postoje slučajevi gde je takva deformacija uticala pozitivno (možda se čak i zadržala u narednim generacijama). Najčešće su se parile samo najjače i najsposobnije jedinke vrste, te time povećavali šansu da se dobra genetika prosledi narednim generacijama. Genetski algoritam gotovo u potpunosti koristi ove ideje kako bi našao optimum optimizacionog problema.

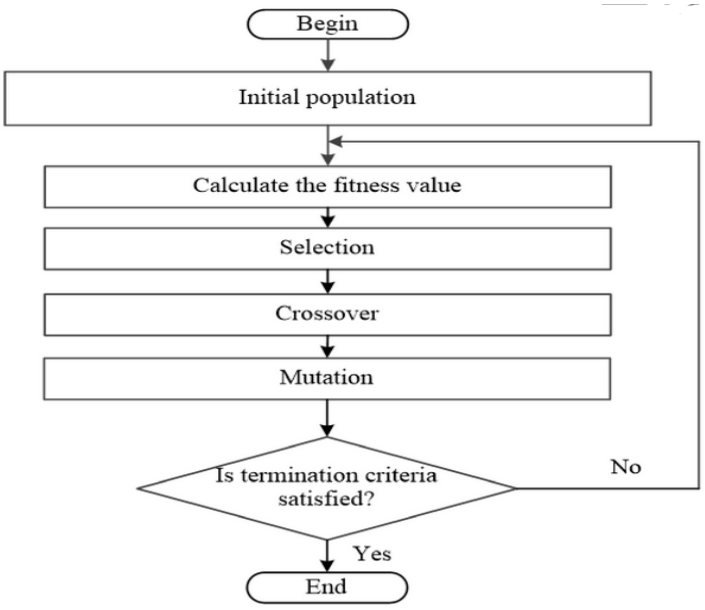
Genetski algoritam spada u klasu evolutivnih algoritama, što znači da menja rešenja kroz iteracije. Ovaj algoritam radi tako što nad početnom populacijom jedinki (rešenja) primenjuje ukrštanje i povremenu mutaciju u nadi da će posle dovoljnog broj generacija jedinke evoluirati u “savršeno biće” (optimalno rešenje).

Rad genetskog algoritma se sastoji iz 3 glavna koraka:

- Selekcija jedinki
- Ukrštanje jedinki
- Mutacija jedinki

Izlaz iz algoritma se postiže kad se ispuni uslov za izlaz. Neki od primera uslova su ispunjen određen broj iteracija, u poslednjih N (N je prirodan broj) generacija se ostvario napredak manji od zadatog praga, itd.

Na slici 3.1. je prikazan dijagram toka izvršavanja genetskog algoritma.



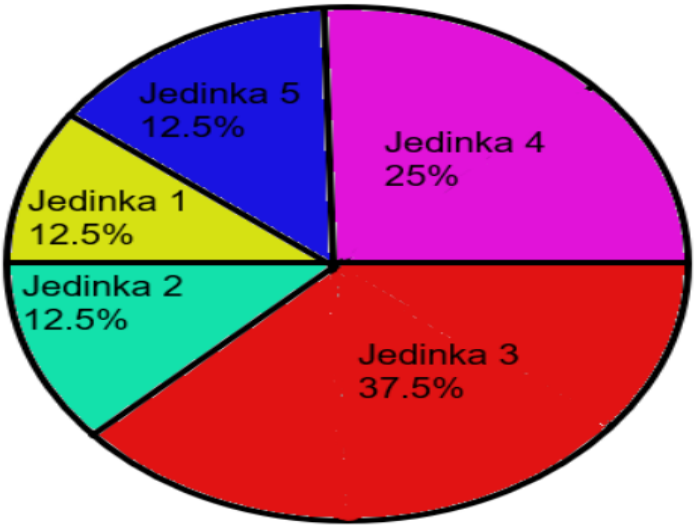
Slika 3.1. Dijagram toka izvršavanje genetskog algoritma [17]

3.2. Selekcija jedinki za ukrštanje

Genetski algoritam na početku biva inicijalizovan slučajnom populacijom. Veličina populacije može biti proizvoljan ceo broj, najčešće se postavlja u zavisnosti od problema. Da bi prešao na naredni korak (ukrštanje) algoritam mora na neki način da zaključi koje jedinke da ukrsti. Ovaj problem se rešava računanjem fitnes funkcije za svaku jedinku u populaciji. Na osnovu rezultata se potom svakoj jedinki dodeljuje težina. Ideja je da one jedinke koje imaju veliku težinu imaju i veliku šansu da budu odabrane za ukrštanje. Ipak, ovo ne znači da jedinke koje imaju malu težinu bivaju odbačene, već da i dalje mogu biti izabrane za ukrštanje samo je šansa da se to desi prilično mala.

Primer: Pretpostvimo da ima 5 jedinki u populaciji i da je zbir njihovi težina 100 (100%). Pretpostavimo da je težina najjbolje jedinke 37.5, a najgore 12.5. Ova situacija je ilustrovana na slici 3.2. Kao što se vidi sa slike, moguće je da se točak zaustavi i odabere jedinku 2, ali je to mnogo manje verovatno nego da se zaustavi na jedinki 3. Razlika površina koje ove 2 jedinke pokrivaju je očigledna.

Ovim se postiže da sledeća generacija ima puno “gena” najboljih jedinki prethodne generacija.



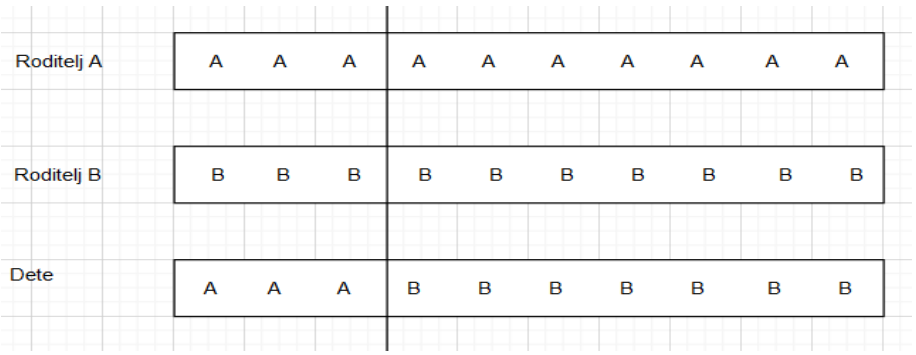
Slika 3.2. Ilustracija selekcije jedinki za ukrštanje.

3.3. Ukrštanje jedinki

Nakon selekcije jedinki, kako bi dobili novu generaciju, potrebno ih je na neki način ukrstiti. Ovo se svodi na mešanje “gena” selektovanih jedinki na određen način. U genetskom algoritmu ne postoji način ukrštanja za koji bi se moglo reći da je najbolji, već ima nekoliko načina koji daju dobre rezultate i koji se najčešće koriste. Preporuka je da se način ukrštanja izabere na osnovu problema.

3.3.1. Ukrštanje u jednoj tački (Single point crossover)

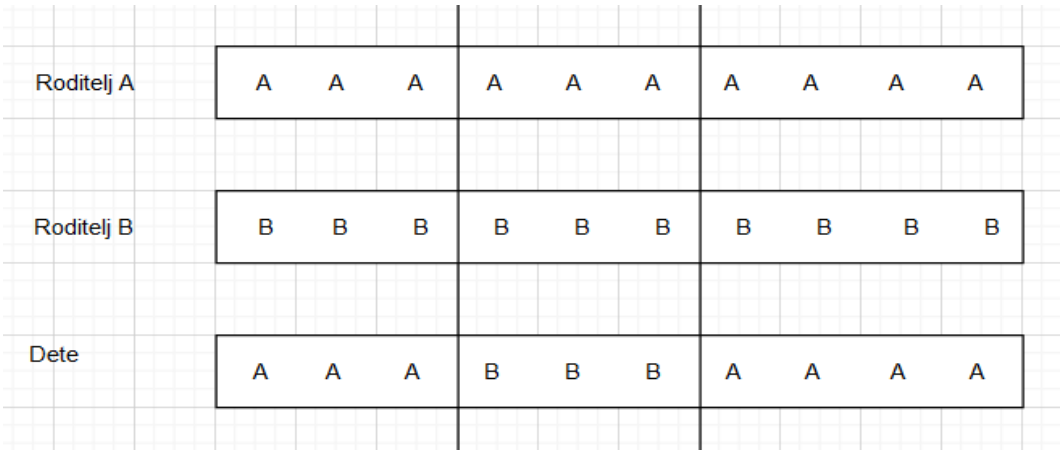
Ukrštanje u jednoj tački je najjednostavniji način ukrštanja. On podrazumeva da se odabere tačka u genetskom zapisu roditelja. Potom formira genetski zapis deteta preslikavajući deo genetskog kod roditelja levo od odabrane tačke na genetski kod deteta takođe levo od odabrane taće, odnosno genetski kod roditelja desno od uzete tačke na genetski kod deteta takođe desno od uzete tačke. Ovaj način ukrštanja je ilustrovan na slici 3.3.



Slika 3.3. Ukrštanje u jednoj tački.

3.3.2. Ukrštanje u dve tačke

Slično kao kod ukrštanja u jednoj tački, osim što se ovde biraju 2 tačke, te se genetski kod roditelja naizmenično preslikava na genetski kod deteta. Ovaj način ukrštanja je ilustrovan na slici 3.4.

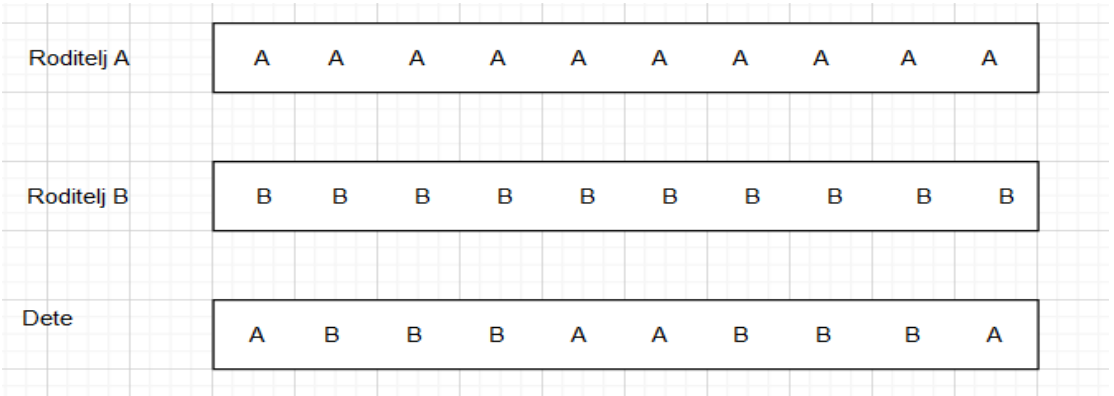


Slika 3.4. Ukrštanje u dve tačke.

Treba napomenuti da je analogno ovome moguće ukrštanje u N tačaka (gde je N prirodan broj).

3.3.3. Uniformno ukrštanje

Ovaj vid ukrštanja podrazumeva da se genetski kod deteta kreira naizmeničnim uzimanjem intervala genetskog koda roditelja slučajne dužine sve dok se genetski kod deteta ne formira. Ovaj vid ukrštanja je ilustrovan na slici 3.5.



Slika 3.5. Uniformno ukrštanje.

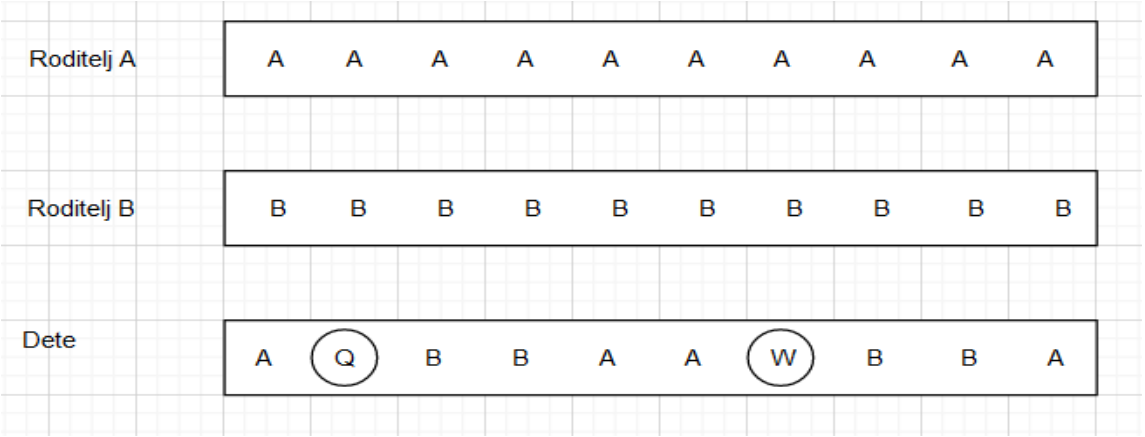
3.4. Mutacija

Tehnike obrađene do sada bi imale konvergenciju, ali se uopšte ne baziraju na pretragu prostora, rešenje bi zavisilo isključivo od početne populacije. Ovo bi najčešće imalo za posledicu konvergenciju u lokalni optimum. Kako bi se izborio sa ovim, genetski algoritam uvodi mutaciju.

U genetskom algoritmu mutacija podrazumeva nestandardnu, odnosno slučajnu i neočekivanu promenu nekih jedinki. Ovim se ohrabruje pretraga prostora, odnosno bez mutacije ne bi možda nikad imali jedinku koja bi istražila sektor potencijalnog globalnog optimuma.

Nema nekog ustaljenog pravila kako i koliko često treba primenjivati mutaciju na jedike. Odgovor na pitanje koliko često se može dobiti testiranjem algoritma na problem od interesa, te ako se uoči da se algoritam često zaglavi na lokalnom optimumu, treba razmisliti o povećanju frekvencije mutacija. Način vršenja mutacija nije bitan u suštini pošto, kao što i ime sugeriše, to predstavlja neočekivanu promenu genetskog koda. Stoga je samo potrebno vršiti takvu promenu i samim tim smanjiti sansu upada u lokalni optimum.

Na slici 3.6. je dat jedan primer mutacije.



Slika 3.6. Primer mutacije.

3.5. Elitizam

Možda ne ključna, ali svakako tehnika koja može da bude od koristi pri korišćenju genetskog algoritma je elitizam. Elitizam podrazumeva povremeno čuvanje izuzetno dobrih jedinki iz prethodne generacije, odnosno njihovo održavanje u životu i tokom sledeće generacije.

Nema univerzalno prihvaćenog načina kako ovo raditi, već je potrebno modifikovati u zavisnosti od problema.

Ova tehnika još više pospešuje širenje izuzetno dobrih gena u narednim generacijama.

3.6. Sumarizacija genetskog algoritma

Genetski algoritam je evolutivni optimizacioni algoritam. Optimizuje simulirajući Darvinovu teoriju evolucije. Služi se populacijom jedniki koje ukršta kako bi dobio novu generaciju. Povremeno zadržava jako dobre jedinke. Kako bi smanjio šansu upada u lokalni minimum povremeno vrši mutaciju nad jedinkama.

4. Predstavljanje funkcija korišćenih za testiranje GWO algoritma

U ovoj sekciji su prikazane funkcije korišćene za testiranje *GWO* algoritma, data su njihova ograničenja, optiumumi i prikazani grafici. Iako ima beskonačno mnogo funkcija koje je teško optimizovati, u ovom radu je za potrebe testiranja izabrano 5 koje imaju izuzetno veliki broj lokalnih optimuma.

Funkcije nad kojima je testiran *GWO* algoritam:

- *Bukin function N. 6*
- *Cross-In-Tray function*
- *Drop-Wave function*
- *Eggholder function*
- *Griewank function*

4.1. Bukin function N. 6

Bakinova šesta funkcija je definisana u dvodimenzionalnom prostoru. Ima puno lokalnih minimuma, gde svaki leži na takozvanoj litici [18].

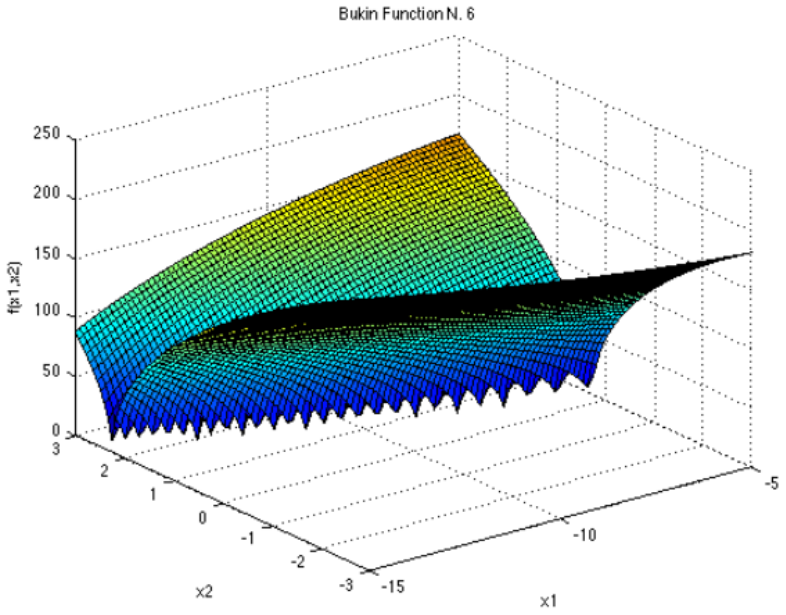
Ograničenja ove funkcije su sledeća: $x_1 \in [-15, -5]$, $x_2 \in [-3, 3]$.

Globalni optimum se nalazi u tački x* = (-10, 1), i ima vrednost f(x*) = 0.

Grafik i sama funkcija se nalaze na slici 4.2. a implementacija funkcije u *python* kodu se nalazi na slici 4.1.

```
1  def BukinN6(array):
2      under_sqrt = abs(array[1] - 0.01 * math.pow(array[0], 2))
3      return 100*math.sqrt(under_sqrt) + 0.01 * abs(array[0] + 10)
```

Slika 4.1. Implementacija bukinove šeste funkcije



$$f(\mathbf{x}) = 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

Slika 4.2. Bukinova šesta funkcija [18]

4.2. Cross-In-Tray function

Cross-In-Tray funkcija je definisana u dvodimenzinalnom prostoru, ima puno lokalnih minimuma. Prepoznatljiva je (kao što joj i ime kaže) po grafiku koji je sadržan od krstova [19]. Na slici 4.3. je prikazan grafik ove funkcije, s tim što je na desnom delu slike funkcija prikazana u manjem domenu kako bi se krst jasnije video. Na slici 4.4. je prikazana implementacija funkcije u *python*-u.

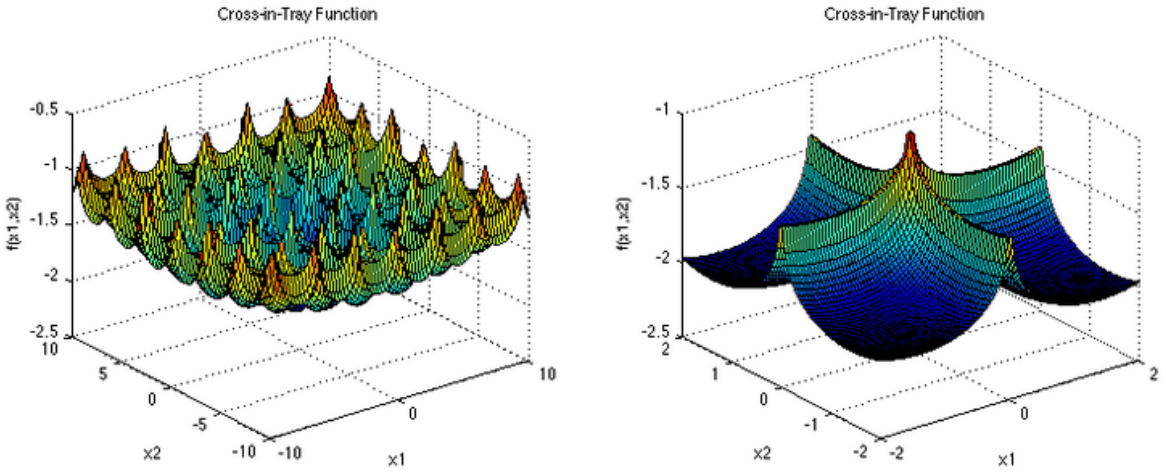
Ograničenja ova funkcije: $x_i \in [-10, 10]$, za svako i = 1, 2.

Globalni optimum:

$$f(\mathbf{x}^*) = -2.06261, \text{at } \mathbf{x}^* = (1.3491, -1.3491), (1.3491, 1.3491), (-1.3491, 1.3491) \text{ and } (-1.3491, -1.3491)$$

```
1  def CrossInTray(array):
2      exp_part =  math.exp(abs(100 - math.sqrt(pow(array[0],2) + pow(array[1], 2))/ math.pi))
3      return -0.0001 * math.pow(abs(math.sin(array[0]) * math.sin(array[1]) * exp_part) + 1, 0.1)
```

Slika 4.4. Implementacija funkcije



$$f(\mathbf{x}) = -0.0001 \left(\left| \sin(x_1)\sin(x_2)\exp\left(\left| 100 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi} \right| \right) \right| + 1 \right)^{0.1}$$

Slika 4.3. *Cross-In--Tray* funkcija [19]

4.3. Drop-Wave function

Drop-Wave funkcija je definisana u dvodimenzionalnom prostoru i izuzetno je kompleksna [20]. Ima grafik prepoznatljivog oblika.

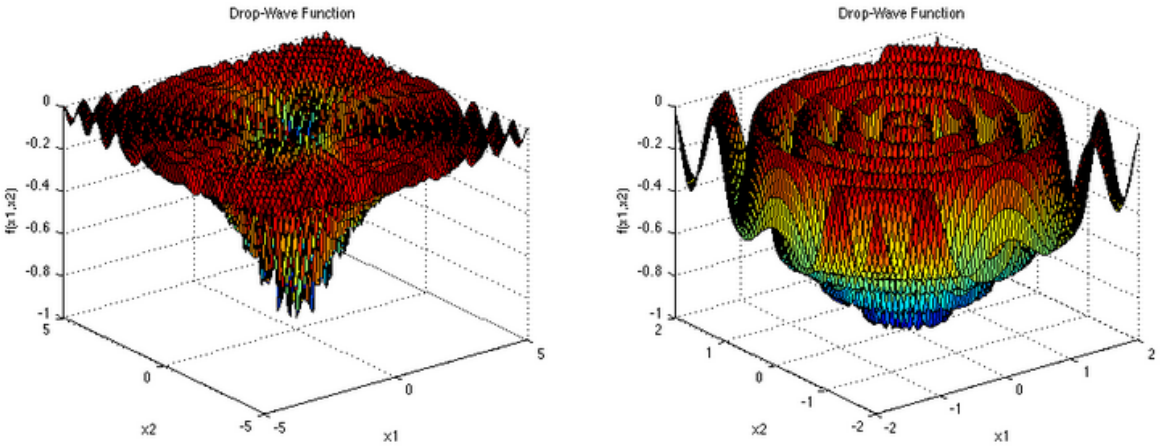
Ograničenja ove funkcije: $x_i \in [-5.12, 5.12]$, za svako i = 1, 2.

Optimum: x* = (0, 0), f(x*) = -1.

Na slici 4.5. se nalazi implementacija funkcije u *python*-u, a na slici 4.6. se nalazi grafik funkcije.

```
1  def DropWave(array):
2      first_part = 1 + math.cos(12*math.sqrt(math.pow(array[0], 2) + math.pow(array[1],2)))
3      second_part = 0.5 * (math.pow(array[0], 2) + math.pow(array[1], 2)) + 2
4      return - (first_part / second_part)
```

Slika 4.5. Implementacija funkcije



$$f(\mathbf{x}) = -\frac{1 + \cos\left(12\sqrt{x_1^2 + x_2^2}\right)}{0.5(x_1^2 + x_2^2) + 2}$$

Slika 4.6. *Drop-Wave* funkcija [20]

4.4. Eggholder function

Eggholder funkcija je definisana u dvodimenzionalnom prostoru i oblik njenog grafika distinktivno podseća na karton za jaja, odatle i ime [21].

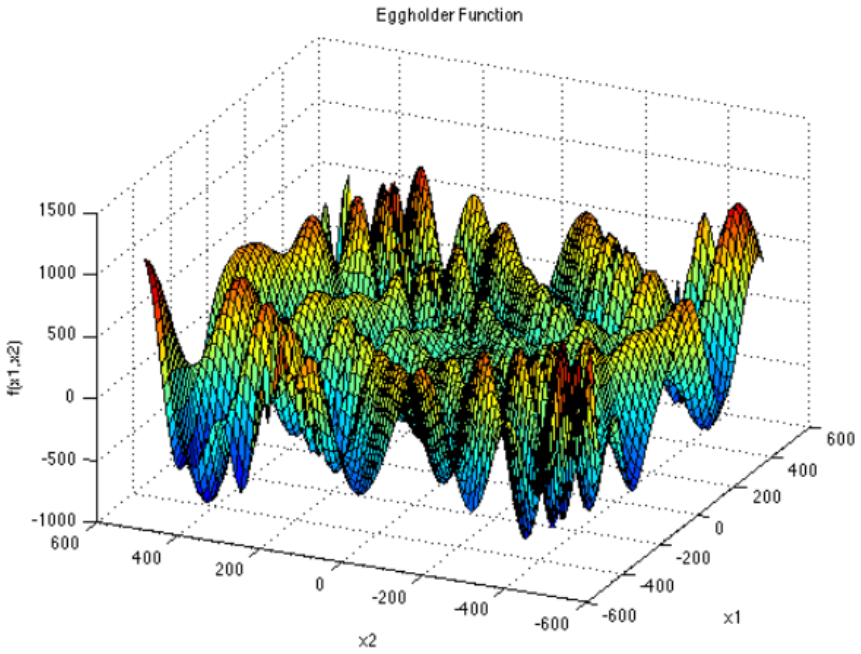
Ograničenja: $x_i \in [-512, 512]$, za svako i = 1, 2.

Optimum: x* = (512, 404.2319), f(x*) = -959.6407.

Na slici 4.8. je prikazan grafik ove funkcije, a na slici 4.7. njena implementacija u *python* kodu.

```
1  def Eggholder(array):
2      first_part = -(array[1] + 47) * math.sin(math.sqrt(abs(array[1] + array[0]/2 + 47)))
3      second_part = array[0] * math.sin(math.sqrt(abs(array[0] - (array[1]+47))))
4      return first_part - second_part
```

Slika 4.7. Implementacija funkcije



$$f(\mathbf{x}) = -(x_2 + 47)\sin\left(\sqrt{\left|x_2 + \frac{x_1}{2} + 47\right|}\right) - x_1\sin\left(\sqrt{|x_1 - (x_2 + 47)|}\right)$$

Slika 4.8. *Eggholder* funkcija [21]

4.5. Griewank function

Griewank funkcija je definisana u N-dimenzionom prostoru. Ima puno lokalnih minimuma koji su podjednako raspoređeni [22].

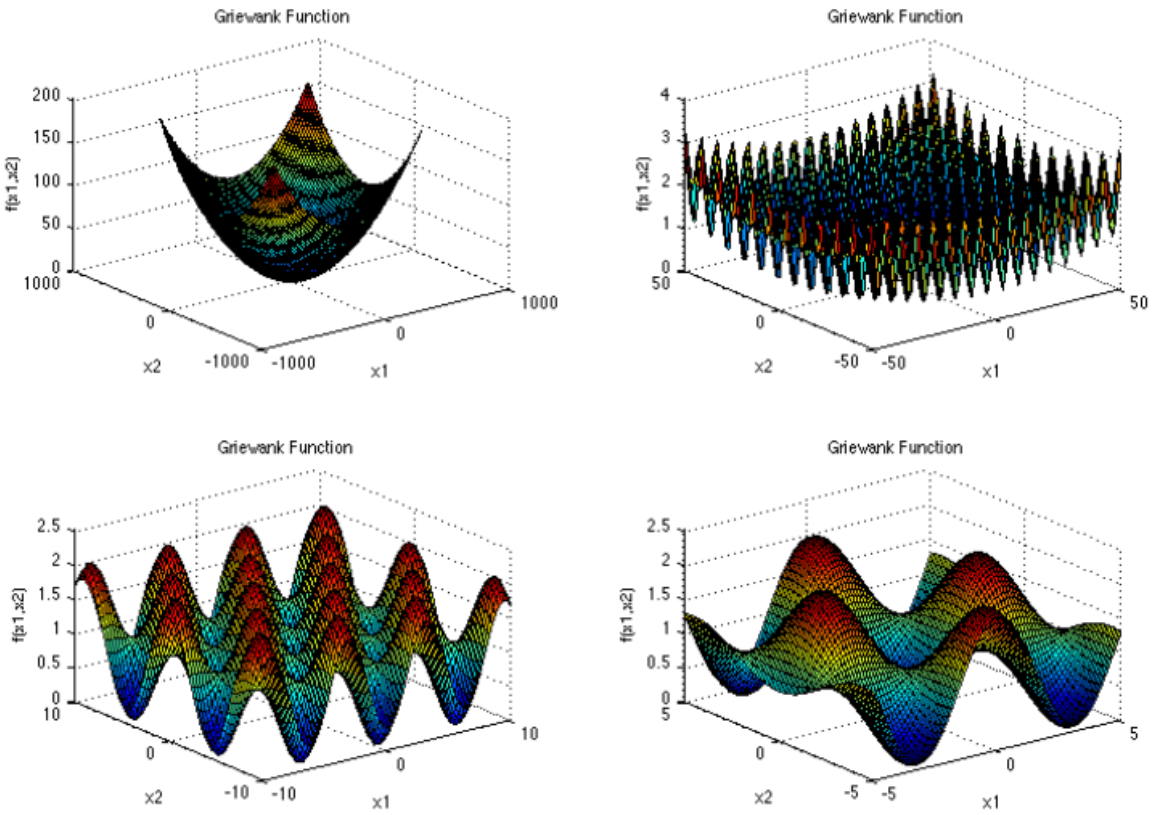
Ograničenja: $x_i \in [-600, 600]$, za svako i = 1, …, d.

Optimum: x* = (0, …, 0), f(x*) = 0.

Na slici 4.9. je prikazana *python* implementacija *griewank* funkcije, a na slici 4.10. je prikazan njen grafik sa različitim domenima.

```
1  def Greiwank(array):
2      sum_part = sum(map(lambda x : math.pow(x, 2) / 4000, array))
3      mul_part = 1
4      for i in range(1, len(array)+1):
5          #mul_part *= array[i]/math.sqrt(i)
6          mul_part = mul_part * math.cos(array[i-1]/math.sqrt(i))
7      return sum_part - mul_part + 1
```

Slika 4.9. Implementacija funkcije



$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Slika 4.10. *Griewank* funkcija [22]

5. Testiranje GWO algoritma

U ovoj sekciji su prikazani rezultati testova *GWO* algoritma na 5 različitih funkcija. *GWO* je nad svakom funkcijom pokrenut više puta, testirajući kako na rešenje utiče promena veličine čopora i broj ukupnih iteracija. Testovi su izvedeni tako da je *GWO* pokretan sa veličinom čopora u rasponu [7, 12], odnosno maksimalnim brojem iteracija [1000, 15000] sa korakom 2000.

5.1. Rezultati testa Bukinove šeste funkcije

U tabeli 5.1. su prikazani rezultati algoritma u slučaju optimizacije Bukinove šeste funkcije.

iterNum/ packSize	7	8	9	10	11	12
1000	x*=1.097	x*=0.091	x*=0.127	x*=0.284	x*=0.303	x*=0.512
3000	x*=0.184	x*=0.154	x*=0.114	x*=0.136	x*=0.055	x*=0.095
5000	x*=0.12	x*=0.088	x*=0.142	x*=0.089	x*=0.13	x*=0.212
7000	x*=0.103	x*=0.175	x*=0.147	x*=0.068	x*=0.29	x*=0.085
9000	x*=0.16	x*=0.172	x*=0.111	x*=0.159	x*=0.329	x*=0.113
11000	x*=0.09	x*=0.125	x*=0.124	x*=0.1	x*=0.047	x*=0.051
13000	x*=0.1	x*=0.077	x*=0.114	x*=0.064	x*=0.052	x*=0.073
15000	x*=0.146	x*=0.17	x*=0.154	x*=0.153	x*=0.056	x*=0.082

Tabela 5.1. Rezultati testova nad Bukinovom šestom funkcijom

Iz tabele se može primetiti da nije dostignut globalni optimum (x* = 0) upotrebom tesiranih parametara, ali da ima rešenja koja su zadovoljavajuće blizu optimumu. Takođe uočava se da u ovom konkretnom problemu veličina čopora nije primetno uticala na rešenje. Dok se povećavajući broj iteracija optimum uglavnom poboljšavao.

5.2. Rezultati testa CrossInTray funkcije

U tabeli 5.2. su prikazani rezultati algoritma u slučaju optimizacije *CrossInTray* funkcije.

iterNum/ packSize	7	8	9	10	11	12
1000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
3000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
5000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
7000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
9000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
11000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
13000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062
15000	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062	x*=-2.062

Tabela 5.2. Rezultati testova nad *CrossInTray* funkcijom

Testiranjem ove funkcije se primećuje da *GWO* algoritam pronalazi optimum u svim testiranim slučajevima. No, bitno je napomenuti da su optimumi u tabeli zaokruženi na 3 decimale zbog preglednosti, ali da je algoritam vraćao različite vrednosti optimuma posle šeste decimale. Kod ove funkcije je takođe bitno napomenuti da ima više globalnih optimuma (više tačaka koje daju optimum iz tabele). Ove tačke su kombinacije vrednosti (±1.34928811, ±1.34937011). Sve kombinacije ovih tačaka vode do globalnog optimuma.

5.3. Rezultati testa Eggholder funkcije

U tabeli 5.3. su prikazani rezultati algoritma u slučaju optimizacije *Eggholder* funkcije.

iterNum/ packSize	7	8	9	10	11	12
1000	x*=-894.578	x*=-786.524	x*=-557.368	x*=-557.368	x*=-894.578	x*=-718.167
3000	x*=-894.578	x*=-959.64	x*=-959.64	x*=-718.167	x*=-959.64	x*=-718.167
5000	x*=-894.578	x*=-786.525	x*=-894.578	x*=-959.64	x*=-959.64	x*=-894.578
7000	x*=-959.64	x*=-786.525	x*=-718.167	x*=-557.368	x*=-786.525	x*=-894.578
9000	x*=-959.64	x*=-894.578	x*=-894.578	x*=-718.167	x*=-894.578	x*=-959.64
11000	x*=-959.64	x*=-959.64	x*=-959.64	x*=-718.167	x*=-894.578	x*=-959.64
13000	x*=-786.525	x*=-894.578	x*=-557.368	x*=-959.64	x*=-959.640	x*=-959.64
15000	x*=-959.64	x*=-786.525	x*=-959.64	x*=-894.578	x*=-959.64	x*=-959.64

Tabela 5.3. rezultati testova *Eggholder funkcije*

Iako se iz tabele vidi da je algoritam u stanju da pronađe optimum *Eggholder* funkcije, takođe se iz tabele može uočiti da je dosta puta završio u lokalnom optimumu. Štaviše, uočavaju se nekoliko lokalnih optimuma koji se ponavljaju. Takođe se može uočiti da je algoritam najviše puta našao globalni optimum kada je veličina čopora bila 7, 11 i 12, što navodi da veličina čopora nije primetno pomogla. U ovom testu nije očigledno ni da broj iteracija pomaže, s obzirom da se globalni

optimum pojavljuje bar jednom kod svakog tesitanog broja iteracija. Dakle zaključujemo da za ovaj problem parametri nisu ključni, već da bi trebalo algoritam pokrenuti nekoliko puta, dok se ne dođe to globalnog optimuma.

5.4. Rezultati testa DropWave funkcije

U tabeli 5.4. su prikazani rezultati algoritma u slučaju optimizacije *DropWave* funkcije.

iterNum/ packSize	7	8	9	10	11	12
1000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
3000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
5000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
7000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
9000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
11000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
13000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1
15000	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1	x*=-1

Tabela 5.4. Rezultati testiranja nad funkcijom *DropWave*

Iz tabele se primećuje da je algoritam našao globalni optimum sa svim testiranim parametrima. Treba napomenuti da ima neznatna nestabilnost u koordinatama tačaka koje se dobiju kao optimum. Naime, sve koordinate treba da budu 0, ali se pojavljuje određeni šum posle devete decimale. Ovo se može pripisati zaokruživanju i manipulisanju izuzetno malim brojevima u računarstvu koje je samo po sebi nestabilno.

5.5. Rezultati testa Griewank funkcije

U tabeli 5.5. su prikazani rezultati algoritma u slučaju optimizacije *Griewank* funkcije.

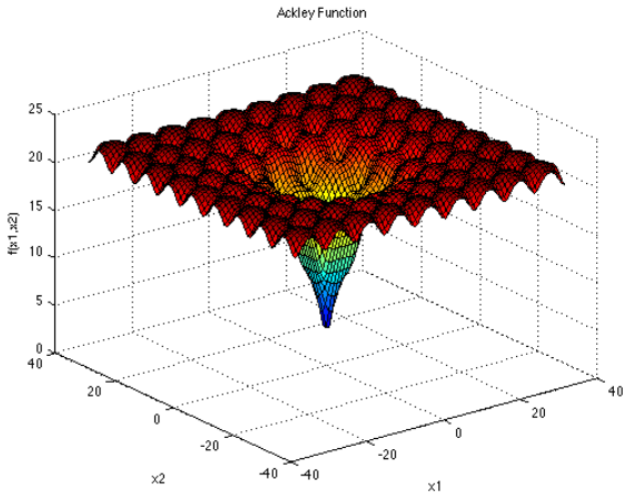
iterNum/ packSize	7	8	9	10	11	12
1000	x*=0.034	x*=0.038	x*=0	x*=0.057	x*=0.095	x*=0.014
3000	x*=0.072	x*=0.055	x*=0	x*=0.027	x*=0.015	x*=0
5000	x*=0	x*=0	x*=0.024	x*=0	x*=0	x*=0
7000	x*=0	x*=0.01	x*=0	x*=0	x*=0.015	x*=0.019
9000	x*=0	x*=0	x*=0	x*=0	x*=0	x*=0.017
11000	x*=0.01	x*=0	x*=0	x*=0	x*=0.007	x*=0
13000	x*=0.051	x*=0	x*=0	x*=0.032	x*=0	x*=0
15000	x*=0.022	x*=0	x*=0.024	x*=0	x*=0	x*=0

Slika 5.5. Rezultati testa *Griewank* funkcije

Iz tabele se može primetiti da iako nekad dolazi do lokalniih optimuma, oni su po vrednosti vrlo blizu globalnom optimu. No, može se reći da je *GWO* dobar algoritam za rešavanje ovog problema, posebno što je očigledno da globalni optimum počinje da dominira povećanjem broja iteracija.

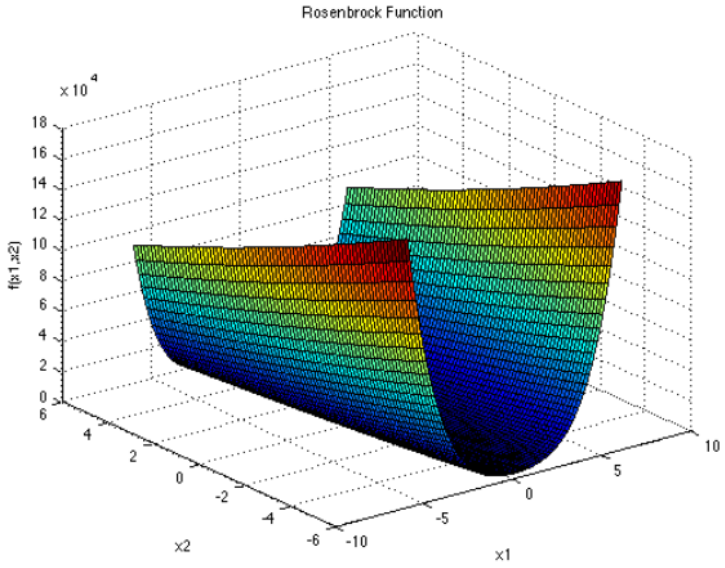
6. Poređenje rezultata GWO algoritma i genetskog algoritma

U ovom segmentu je dato poređenje rezultata rada *GWO* algoritma i genetskog algoritma. U poređenju su korišćene 3 funkcije: *Eggholder*, *Rosenbrock* [24] i *Ackley funkcije* [23]. Korisiti se implementacija genetskog algoritma iz biblioteke *geneAI [25],* odnosno implementacija *GWO* algoritma priložena u ovom radu. Genetski algoritam je pušten sa *default*-nim parametrima. U tabeli 6.3. su prikazani rezultati rada oba algoritma. Na slici 6.1. je prikazan grafik funkcije *Ackley*, a na slici 6.2. grafik funkcije *Rosenbrock.*



$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Slika 6.1. Funkcija *Ackley* [23]



$$f(\mathbf{x}) = \sum_{i=1}^{d-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Slika 6.2. Funkcija *Rosenbrock* [24]

	GWO	*GA*	Prave vrednosti funkcije
Funkcija	*Eggholder*		
Optimum	-959.640	-939.761	-959-6407
Pozicija optimuma	(512, 404.231)	/	(512, 404.231)
Funkcija	*Rosenbrock*		
Optimum	3.396e-10	1.287e-7	0
Pozicija optimuma	(0.999, 0.999)	/	(1, 1)
Funkcija	*Ackley*		
Optimum	4.44e-16	7.972e-8	0
Pozicija optimuma	(-9.092e-17, 1.671e-16)	/	(0, 0)

Tabela 6.3. Poređenje performansi *GWO* algoritma sa genetskim algoritmom

7. Zaključak

Meta-heuristike su sposobne da reše širok spekat problema. *GWO* kao jedna od njih se pokazao prilično dobro na testovima. Gotovo na svakom testu je uspeo da nađe globalni optimum, iako je povremeno završavao u lokalnim optimumima. U testovima koje je radio paralelno sa genetskim algoritmom se pokazao nešto bolji (mada razlika nije bila drastična).

GWO algoritam je jednostavan algoritam sa malim brojem parametara koji su mu potrebni za rad, uprkos tome postiže izuzetne rezultate. Odlično utilizuje strategiju lova čopora vukova, tj. njegov model prilično dobro uspeva da izbalansira faze pretrage i eksploatacije, gde ne odustaje od pretrage čak ni u kasnim iteracijama. Ne oslanja se na komplikovane računarske operacije, tako da se njegova brzina izvršavanja bazira na fitnes funkciji, broju dimenzija i broju iteracija. Odnosno, kompleksnost zavisi od problema. Ovaj algoritam je još jedan pokazatelj da posmatrajući prirodu i procese u njoj, može da se izvede model koji je u stanju da rešava probleme najviše kompleksnosti, te može samo da se pretpostavlja šta je još ostalo neotkriveno i kakav bi potencijalan uticaj imalo.

Dalji nastavak ovog rada bi mogao da podrazumeva još detaljnije testiranje *GWO* algoritma na jos kompleksnijim funkcijama, kao i njegovo poređenje sa ostalim algoritmima u klasi.

8. Reference

[1] Bonabeau E, Dorigo M, Theraulaz G. Swarm intelligence: from natural to artificial systems: OUP USA; 1999.
[2] Dorigo M, Birattari M, Stutzle T. Ant colony optimization. Comput Intell Magaz, IEEE 2006;1:28–39.
[3] Kennedy J, Eberhart R. Particle swarm optimization, in Neural Networks, 1995. In: Proceedings, IEEE international conference on; 1995. p. 1942–1948.
[4] Wolpert DH, Macready WG. No free lunch theorems for optimization. Evolut Comput, IEEE Trans 1997;1:67–82.
[5] Olorunda O, Engelbrecht AP. Measuring exploration/exploitation in particle swarms using swarm diversity. In: Evolutionary computation, 2008. CEC 2008 (IEEE World Congress on Computational Intelligence). IEEE Congress on; 2008. p. 1128–34.
[6] Alba E, Dorronsoro B. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. Evolut Comput, IEEE Trans 2005;9:126–42.
[7] Lin L, Gen M. Auto-tuning strategy for evolutionary algorithms: balancing between exploration and exploitation. Soft Comput 2009;13:157–68.
[8] Grey Wolf Optimizer, Seyedali Mirjalilia,⇑, Seyed Mohammad Mirjalilib, Andrew Lewisa
[9] Muro C, Escobedo R, Spector L, Coppinger R. Wolf-pack (Canis lupus) hunting strategies emerge from simple rules in computational simulations. Behav Process 2011;88:192–7.
[10] https://docs.python.org/3/library/random.html
[11] https://docs.python.org/3/library/math.html
[12] https://docs.python.org/3/library/copy.html
[13] https://numpy.org/doc/
[14] https://seaborn.pydata.org/
[15] https://pandas.pydata.org/
[16] https://matplotlib.org/
[17] Genetic Algorithm Based on Natural Selection Theory for Optimization Problems by Musatafa Abbas Albadr , Sabrina Tiun , Masri Ayob and Fahad AL-Dhief
[18] https://www.sfu.ca/~ssurjano/bukin6.html
[19] https://www.sfu.ca/~ssurjano/crossit.html
[20] https://www.sfu.ca/~ssurjano/drop.html
[21] https://www.sfu.ca/~ssurjano/egg.html
[22] https://www.sfu.ca/~ssurjano/griewank.html
[23] https://www.sfu.ca/~ssurjano/ackley.html
[24] https://www.sfu.ca/~ssurjano/rosen.html
[25] https://github.com/diogomatoschaves/geneal
[26] Beni G, Wang J. Swarm intelligence in cellular robotic systems. In: Robots and biological systems: towards a new bionics?, ed. Springer; 1993. p. 703–12.