

DRŽAVNI UNIVERZITET U NOVOM PAZARU
Departman Tehničko-tehnološke nauke



Završni rad
PARALELIZACIJA ALGORITMA GWO POMOĆU PROGRAMSKOG OKVIRA
APACHE SPARK

Mentor: Dr Irfan Fetahović

Student: Lamija Koca

Novi Pazar, Jun 2023.

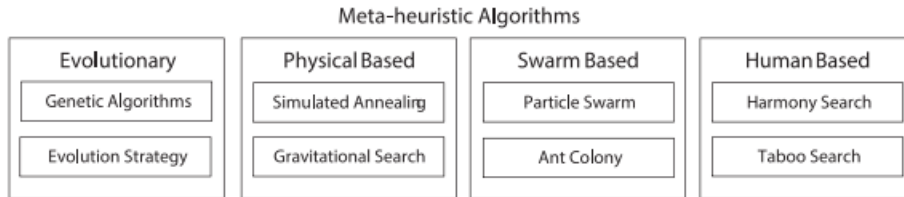
Sadržaj

1. Uvod	1
2. Optimizacioni algoritam sivi vuk	2
2.1. Definicija.....	3
2.2. Primer	5
2.3. Primena	7
2.3.1. Hiperparametarsko izvršavanje	8
2.3.2. Izbor funkcija.....	8
2.3.3. Robotika i planinarenje.....	8
2.3.4. Obrada slike	9
2.3.5. Grupisanje.....	9
2.3.6. Medicina i bioinformatika	9
3. Apache Spark Framework.....	9
3.1. Arhitektura sistema Apache Spark.....	10
4. Podešavanje eksperimentalnog okruženja.....	11
5. Optimizacioni algoritam sivi vuk baziran na Spark-u	13
6. Evaluacija	14
7. Zaključak	19
8. Prilozi	20
9. Reference.....	22

1. UVOD

U mnogim današnjim tehničkim primenama, sekunde mogu da naprave značajnu razliku u produktivnosti. Dostupnost, skalabilnost i tačnost su važni za svaki definisani problem koji može imati više rešenja, od kojih se jedno može smatrati “optimalnim”. Ove probleme nazivamo probleme optimizacije a prethodna istraživanja su ustanovila različite tipove algoritama za otkrivanje optimalnog rešenja za problem optimizacije.

Metaheuristički algoritmi inspirisani prirodom koji rešavaju probleme optimizacije postaju sve popularniji. Ovi algoritmi se mogu podeliti u algoritme evolucije, algoritmi bazirani na fizičkim procesima, algoritmi bazirani na ljudima i njihovom ponašanju i algoritme rojeva, kao sto je prikazano na Slici 1.



Sl.1. Metaheuristički algoritmi inspirisani prirodom [4]

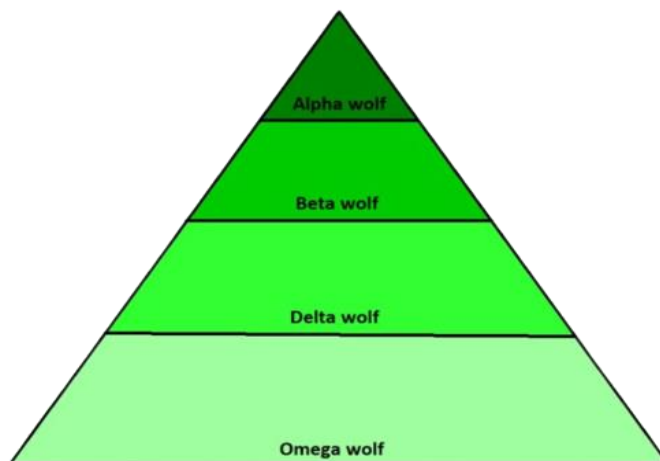
Evolucionirajući algoritmi su inspirisani konceptom Darvinove evolucije; najslabija rešenja se eliminišu dok se jače, održivije opcije zadržavaju i ponovo procenjuju u sledećoj evoluciji, sa ciljem da dođe do optimalnih akcija za postizanje željenih rezultata. Popularan algoritam evolucije je genetski algoritam. Algoritmi bazirani na fizičkim procesima se razvijaju prateći fizičke uslove u svetu. Npr. simulirani algoritam kaljenja, jedan od najpopularnijih metaheurističkih algoritama, zasnovan je na fizičkom kaljenju u stvarnom životu. Fizičko kaljenje je proces zagrevanja materijala dok se ne dostigne temperatura kaljenja, a zatim će se polako hladiti kako bi se materijal promenio u željenu strukturu. Algoritmi zasnovani na ljudima prate imitaciju ljudskog ponašanja.

Algoritmi rojeva (en. Swarm algorithms) se fokusiraju na ideju oponašanja životinja, a Kennedy J. i Eberhart R. su razvili najpopularniji algoritam za optimizaciju roja čestica koji oponaša jato ptica u pokušaju da optimizuju kontinuirane nelinearne funkcije.

2. OPTIMIZACIONI ALGORITAM SIVI VUK

GWO (Grey Wolf Optimizier) algoritam je metaheuristički algoritam inteligencije roja koji su prvobitno predstavili Mirjalili i Lewis. Od svog razvoja, nekoliko istraživača ga je primenilo za rešavanje različitih problema. Ovaj algoritam spada među nedavno predložene optimizacijske algoritme bazirane na socijalnoj inteligenciji. Optimizator sivog vuka je inspirisan sivim vukovima (*Canis lupus*).

Sivi vuk je veoma društvena životinja, oni žive u grupi od 5 do 12 jedinki, što rezultira složenom društvenom hijerarhijom. Ovaj hijerarhijski sistem, gde su vukovi rangirani prema snazi i moći, naziva se “hijerarhija vođstva/dominacije”. Četiri vrste sivih vukova, alfa, beta, delta i omega, koriste se za simulaciju hijerarhije vođstva.



Sl.2. Hijerarhija sivih vukova

Kao što je prikazano na Sl. 2. vođe grupe sivih vukova su alfa mužjak i ženka koji su na vrhu hijerarhije i predvode čopor. Njihova uloga nije samo dominacija i agresija već su tu i da pruže pomoć ugroženim članovima čopora koji ne mogu sami da love. Uopšteno, ostali vukovi u grupi moraju se pridržavati odluka koje donese alfa. Međutim, u nekim slučajevima se mogu videti i bezklasne akcije u društvenoj hijerarhiji sivih vukova. U takvim slučajevima, alfa može poslušati druge vukove u grupi. Važno je napomenuti da alfa ne mora nužno da bude najjači vuk u grupi. Glavna odgovornost alfe je nadgledanje grupe.

Nakon alfe, sledeća karika u društvenoj hijerarhiji je beta, a odgovornost bete je da pomaže alfi u procesu donošenja odluka. Bilo koji od mužjaka ili ženki vukova može biti beta, i beta može biti najpogodniji kandidat za zamenjivanje alfe u bilo kojem trenutku kada alfa nastrada. Beta služi kao savetnik alfi i zadužen je za kažnjavanje svakog prestupnika u grupi. Beta ističe naredbe alfe i prenosi odgovore članova grupe alfi.

Najniže rangirani sivi vuk je omega. Omega ima ulogu žrtvenog jagnjeta. Oni uvek moraju da se potčine svim ostalim vukovima i samim tim su poslednji vukovi koji smeju da jedu. Iako omega izgleda kao najmanje važan vuk u grupi, bez omega bi bilo teško uočiti postojanje unutrašnjih sukoba i drugih problema. Preostali vukovi, osim alfe, bete i omega, nazivaju se sekundarni, delta. Delta vukovi su podređeni alfama i betama, a vladaju nad omegama. Oni funkcionišu kao špijuni, čuvari, starije jedinke, lovci i stražari u grupi. Špijuni su zaduženi za brigu o granicama i teritoriji. Takođe podižu upozoravajući alarm u slučaju bilo kakve opasnosti sa kojom se grupa suočava. Čuvari su odgovorni za čuvanje grupe. Starije jedinke su iskusni vukovi koji su kvalifikovani da budu alfe i bete. Lovci pomažu alfama i betama u lovu na plen, dok stražari brinu o slabim, bolesnim i povređenim vukovima. Vredno je napomenuti da dominacija opada od vrha hijerarhije naniže.

Pored društvene hijerarhije vukova, grupni lov je još jedno zanimljivo društveno ponašanje sivih vukova. Oni love u čoporima i rade u grupama kako bi odvojili plen od krda, a zatim će jedan ili dva vuka juriti i napasti plen. C. Muro je opisao strategiju lova na čopor vukova i ona uključuje sledeće faze:

- Praćenje, jurenje i približavanje plenu
- Gonjenje, opkoljavanje i hvatanje plena dok ne prestane da se kreće
- Napad na plen kad je iscrpljen

2.1. DEFINICIJA

Primenjujući prethodno opisanu metodu na naš problem optimizacije, u svakom koraku ćemo označiti tri najbolja rešenja sa alfa, beta i delta, a ostala rešenja sa omega. U osnovi, to znači da proces optimizacije teče u skladu sa tim najboljim rešenjima. Takođe, plen će biti optimalno rešenje optimizacije. Većina logike prate jednačine:

$$\vec{D} = |\vec{C} \cdot \vec{X}_p(t) - \vec{X}(t)| \quad (1)$$

$$\vec{X}_{(t+1)} = \vec{X}_{p(t)} - \vec{A} \cdot \vec{D}, \quad (2)$$

gde t označava trenutnu iteraciju, \vec{A} i \vec{C} su vektori koeficijenata, vektor \vec{X}_p je vektor položaja plena i vektor \vec{X} je pozicija vuka.

Vektori \vec{A} i \vec{C} su jednaki:

$$\vec{A} = 2\vec{a} \cdot \vec{r}_1 - \vec{a} \quad (3)$$

$$\vec{C} = 2\vec{r}_2, \quad (4)$$

gde se komponente \vec{a} linearno smanjuju od 2 do 0 kroz iteracije i vektori \vec{r}_1 , \vec{r}_2 su nasumični vektori čije su vrednosti $[0, 1]$, izračunati za svakog vuka u svakoj iteraciji. Vektor \vec{A} kontroliše balans između istraživanja i eksploatacije, dok vektor \vec{C} uvek dodaje određeni stepen slučajnosti. Ovo je neophodno jer naši agenti mogu da zaglave u lokalnom optimumu a većina metaheuristika ima način da to izbegne.

S obzirom da ne znamo stvarnu poziciju optimalnog rešenja, \vec{X}_p zavisi od tri najbolja rešenja, a formule za ažuriranje svakog od agenata tj, vukova su:

$$\vec{D}_\alpha = |\vec{C}_1 \cdot \vec{X}_\alpha - \vec{X}| \quad (5)$$

$$\vec{D}_\beta = |\vec{C}_2 \cdot \vec{X}_\beta - \vec{X}| \quad (6)$$

$$\vec{D}_\delta = |\vec{C}_3 \cdot \vec{X}_\delta - \vec{X}| \quad (7)$$

$$\vec{X}_1 = \vec{X}_\alpha - \vec{A}_1 \quad (8)$$

$$\vec{X}_2 = \vec{X}_\beta - \vec{A}_2 \quad (9)$$

$$\vec{X}_3 = \vec{X}_\delta - \vec{A}_3 \quad (10)$$

$$\vec{X}_{(t+1)} = \frac{\vec{x}_1 + \vec{x}_2 + \vec{x}_3}{3} \quad (11)$$

gde \vec{X} predstavlja trenutnu poziciju agenta, a $\vec{X}_{(t+1)}$ je ažurirana pozicija agenta. Formula iznad ukazuje da će pozicija vuka biti ažurirana u skladu sa tri najbolja vuka iz prethodne iteracije. Primetite da ona neće biti tačno jednaka proseku tri najbolja vuka, već će se zbog vektora \vec{C} dodati mali slučajni pomak. Ovo ima smisla jer s jedne strane želimo da naši agenti budu vođeni najboljim jedinkama, a sa druge strane ne želimo da zapnemo u lokalnom optimumu.

Pseudokod GWO algoritma:

Algorithm 1: Grey Wolf Optimizer

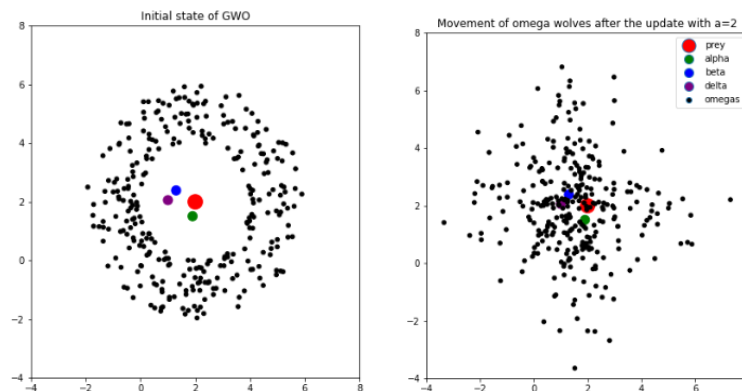
```
Initialize the grey wolf population  $X_i, i = \overline{1, n}$ 
Initialize a, A and C
Calculate the fitness of each search agent
 $X_\alpha$  = the best search agent
 $X_\beta$  = the second best search agent
 $X_\delta$  = the third best search agent
while  $t < \text{max number of iteration}$  do
    for each search agent do
        Randomly initialize  $r_1$  and  $r_2$ 
        Update the position of the current search agent by the
        equations (5,6,7)
    Update a, A and C
    Calculate the fitness of all search agents
    Update  $X_\alpha, X_\beta$  and  $X_\delta$ 
     $t = t + 1$ 
return  $X_\alpha$ 
```

Primećujemo da vremenska složenost algoritma zavisi od broja iteracija, broja agenata i veličine vektora, ali generalno vremensku složenost možemo aproksimirati kao $O(k \cdot n)$ gde je k broj iteracija, a n broj agenata.

2.2.PRIMER

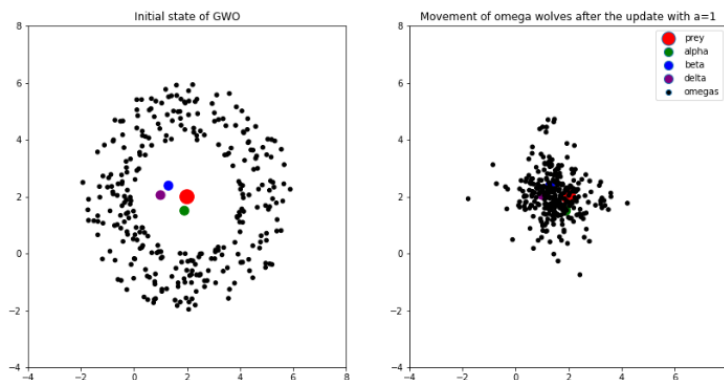
Radi boljeg razumevanja ovog algoritma, predstavimo jedan konkretan primer ponašanja čopora vukova. Na slici ispod možemo videti početno stanje agenata, gde plen ili optimalno rešenje ima crvenu boju. Vukovi koju su najbliži plenu, tačnije alfa, beta i delta, imaju zelenu, plavu i ljubičastu boju. Crne tačke predstavljaju druge vukove, omega.

Zatim, ako ažuriramo položaj omega vukova prema prethodno objašnjenim formulama, možemo posmatrati ponašanje agenata na slici 4.



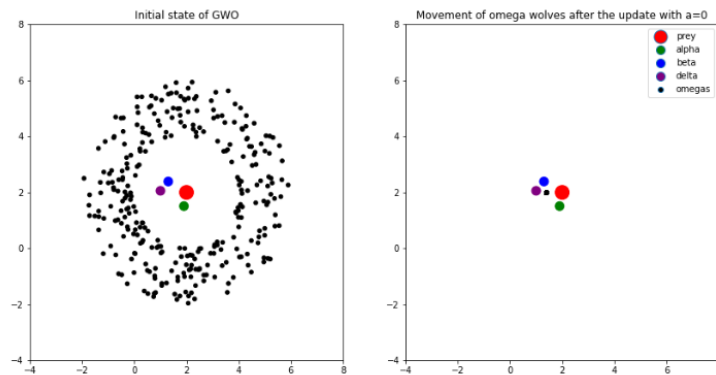
Sl. 4. Ponašanje agenata ako je $a=2$

Sledeća slika prikazuje ponašanje čopora vukova kada je promenljiva a postavljena na vrednost 1. To znači da smo usred procesa optimizacije i da se proces eksploatacije sve češće pojavljuje.



Sl. 5. Ponašanje agenata ako je $a=1$

I konačno, na slici ispod promenljiva a je jednaka 0. To znači da je proces optimizacije na kraju i nadamo se da se približavamo optimalnom rešenju prateći tri najbolja vuka. Možemo da primetimo kako se omega vukovi okupljaju između tri najbolja vuka, oko tačke koja geometrijski predstavlja centar i nalazi se između alfa, beta i delta agenata.



Sl. 6. Ponašanje agenata ako je $a=0$

U prethodnim koracima vidimo da nismo ažurirali položaj alfa, beta i delta agenata samo zbog poređenja sa omegama. U suprotnom ažuriramo poziciju svih agenata, a nakon toga biramo nove alfa, beta i delta.

2.3.PRIMENA

Mirjalili i Lewis su razvili algoritam optimizacije sivog vuka inspirisani lovnim ponašanjem roja vukova. Performanse algoritma su zatim evoluirale, a rezultati su potkrepljeni relativnom studijom sa nekim popularnim algoritmima. Istraživanja su pokazala da je GWO proizveo izvrsne rezultate u poređenju sa ostalim poznatim heurističkim algoritmima. Autori su takođe koristili algoritam da bi ponudili rešenje za neke probleme inženjerskog projektovanja te su raspravljali o primeni tehnike u dizajnu optičkih instrumenata.

Međutim, kao i ostali algoritmi optimizacije, GWO ima svoje nedostatke. Na primer, algoritam često zapinje u lokalnom optimum pri rešavanju visoko-dimenzionálnih nelinearnih funkcija. Osim toga, GWO se teže konvergira u nekim slučajevima. To je zato što GWO teško održava ravnotežu između procesa eksploatacije i istraživanja. Zbog tih nedostataka, istraživači rade na razvoju visokopeformantnih varijanti GWO kako bi preovladali ove slabosti. Pa su tako neki istraživači koristili grupni optimizator sivih vukova [9] za pronalaženje optimalnih parametara za svoje probleme.

Uspešno se koristi u rešavanju problema kao što su:

- Problem trgovačkog putnika (TSP)
- Problem minimalnog razapinjućeg stabla
- Problem protoka struje

Osim toga, ovaj algoritam ima široku primenu i u mašinskom učenju, u robotici, obrađivanju slika i ostalog, što ćemo površno razmotriti u nastavku.

2.3.1. Hiperparametarsko podešavanje

Jedna zanimljiva primena može biti podešavanje hiperparametara bilo kog algoritma za mašinsko učenje. Glavna ideja koja stoji iza podešavanja hiperparametara je pronalaženje optimalnog skupa vrednosti za parameter kako bi se maksimizirale performanse datog algoritma. Na primer, koristeći GWO mogli bismo da optimizujemo hiperparametre Gradient Boosting tehnike, tako što bismo konstruisali vektor pozicije na način da svaka od vektorskih komponenti ima određen hiperparametar. Parametri uopšte ne moraju da imaju numeričku vrednost, jer uvek možemo da definišemo sopstvene metrike za izračunavanje udaljenosti između agenata.

2.3.2. Izbor funkcija

Algoritam funkcioniše tako što se konstruiše skup mogućih karakteristika i iterativno modifikuje ove karakteristike na osnovu neke mere performansi dok se ne pronađe željeno rešenje. Ovaj pristup je mnogo jeftiniji u smislu vremenske složenosti pošto se složenost izbora najboljeg podskupa karakteristika eksponencijalno povećava sa svakom dodatnom promenljivom. To je zato što je ukupan broj podskupova sa n elemenata 2^n . Znači, ako imamo 30 različitih karakteristika kao input, output kao broj mogućih podskupova od 30 karakteristika će biti $2^{30} = 1.073.741.824$. Prema tome, bilo bi izvodljivije predstaviti 30 karakteristika kao binarni vektor veličine 30 gde svaki bit predstavlja jednu posebnu karakteristiku (1 označava da je karakteristika uključena, 0 da nije). Taj binarni vektor predstavlja poziciju agenta i može ga lako koristiti GWO ili neka druga metaheuristika.

2.3.3. Robotika i planinarenje

Zhang i njegovi saradnici [10] su koristili GWO za rešavanje problema planinarenja putanje dvodimenzionalnog bespilotnog borbenog aviona. Performanse GWO su se pokazale vrlo konkurentne u poređenju sa drugim tehnikama. Dok su Jain i njegovi saradnici [11] razvili hibridni sistem koji kombinuje PSO i GWO za rešavanje problema lokalizacije mirisa u robotskom domenu.

2.3.4. Obrada slike

Vinothini i Bakkiyaraj [12] primenili su GWO za poboljšanje kontrasta, očuvanje svetlosti i povećanje kvaliteta slika niskog dinamičkog raspona. Performanse GWO su bile superiornije u odnosu na GA, kako je pokazano u njihovom eksperimentu.

2.3.5. Grupisanje (en. Clustering)

Jeet [13] je primenio MOGWGA algoritam za grupisanje pet uzoraka softvera. Za primenu tih algoritama uzeti su u obzir šest ciljeva klasterovanja. Rezultati su poređeni s metodama temeljenim na Two-Archivem MOGA I NSGA-II. Rezultati pokazuju da MOGWGA algoritam nadmašuje ostala dva pristupa koja su korišćena za problem opisan u radu.

2.3.6. Medicina i bioinformatika

GWO se koristi za klasifikaciju, pohranu i analizu biohemijskih i bioloških informacija. Li i njegovi saradnici [14] primenili su hibrid binarnog GWO-a i ELM (en. Extreme Learning Machine) za odabir karakteristika u ranoj dijagnozi Parkinsonove bolesti i raka dojke.

3. APACHE SPARK FRAMEWORK

Apache Spark je framework razvijen na Univerzitetu Kalifornije u Berkliju kao objedinjeni mehanizam za obradu velikih skupova podataka. Dizajniran je da isporuči računarsku brzinu, skalabilnost i programibilnost potrebne za velike podatke, posebno za grafičke podatke, mašinsko učenje i aplikacije veštačkog učenja.

Sparkov analitički mehanizam obrađuje podatke 10 do 100 puta brže od alternative. Skalira se tako što distribuira rad obrade na velike grupe računara, sa ugrađenim paralelizmom i tolerancijom na greške. Čak uključuje i API za različite programske jezike koji su popularni, tačnije za Scala, Java, Python i R. Spark se često poredi sa Apache Hadoop, a posebno sa MapReduce. Glavna razlika između Spark i MapReduce je u tome što Spark obrađuje i čuva podatke u memoriji za naredne korake, bez pisanja na disk ili čitanje sa diska, što rezultira daleko većom brzinom.

3.1. ARHITEKTURA SISTEMA APACHE SPARK

Apache Spark ima hijerarhijsku master-slave arhitekturu. Spark drajver je glavni čvor koji kontroliše menadžera klaster, koji upravlja slave čvorovima i isporučuje rezultate podataka klijentu aplikacije.

Spark stvara RDD-ove (Resilient Distributed Datasets) iz spoljnih podataka i primenjuje paralelne operacije. RDD su kolekcije elemenata otpornih na greške i predstavljaju osnovnu strukturu Apache Spark. U RDD-u se izvršavaju dva tipa operacija, uključujući transformacije koje definišu nove RDD-ove na osnovu prethodnih i akcije koje pokreću posao koji će biti izvršen na klasteru. RDD-ovi se mogu kreirati ili izvršavanjem postojeće kolekcije u programu upravljača paralelno ili upućivanjem na spoljni sistem za skladištenje, kao što je Hadoop Distributed File System (HDFS), deljeni fajl sistema, HBase ili bilo koji drugi izvor podataka koji nudi Hadoop. Kolekcije izvršene paralelno kreiraju se putem metode koja se naziva SparkContext. Elementi kolekcije se kopiraju u RDD, koji se zatim može paralelno obrađivati. Na klasteru, zadaci se izvršavaju na radnim čvorovima gde postoje particije. Stoga se svako izračunavanje koje se obavlja na bilo kom zadatku izvršava na particijama. Ovo dovodi do situacije u kojoj broj zadataka koji će biti izvršeni u jednoj fazi zavisi od broja particija koje postoje na svakom radnom čvoru. Osim toga, broj particija definiše nivo paralelizma na svakoj fazi izvršavanja. Podrazumevano ponašanje particionisanja u Sparku u potpunosti zavisi od izvora RDD-ova.

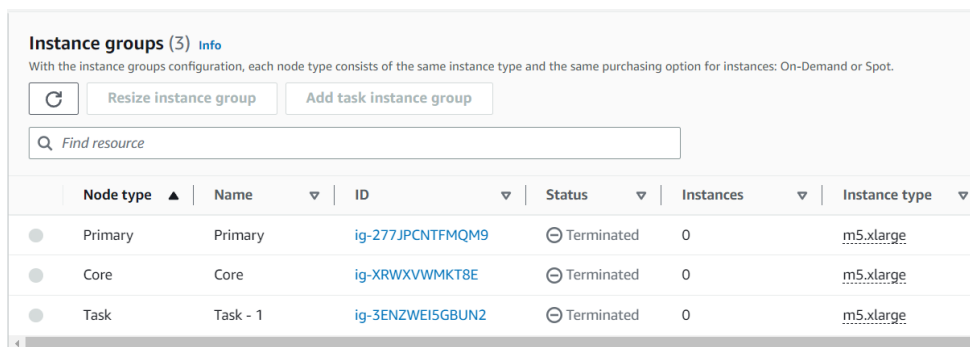
Izvršavanje Sparka u klasteru obavlja se preko glavnog čvora (driver node) i radnih čvorova (worker nodes). Glavni čvor gradi plan izvršavanja za zavisne i nezavisne zadatke, koji se prikazuje kao usmereni aciklički graf (DAG) koji pokazuje zavisnosti i paralelizam zadataka. S druge strane, radni čvorovi su fizički odvojeni računari u klasteru koji obavljaju određene zadatke koje upravlja glavni čvor. Funkcije dobijene od glavnog čvora se izvršavaju na radnim čvorovima, tačnije na particijama unutar radnih čvorova. Na primer, funkcija "sortiranje" se generiše od strane glavnog čvora i izvršava se na particijama unutar svakog radnog čvora, pri čemu se svaka particija sortira na osnovu određenih parametara. Spark koristi emitovane (broadcast) promenljive koje omogućavaju programeru da čuva samo za čitanje promenljive keširane na svakom radnom čvoru umesto da kopira kada se zadatak izvršava. Emitovane promenljive mogu se koristiti da bi se svakom čvoru dodelila kopija velikih podataka. Spark takođe distribuira emitovane promenljive koristeći efikasne algoritme emitovanja kako bi smanjio troškove komunikacije. Metode za akciju u Sparku se izvršavaju u određenim fazama koje su odvojene "shuffle" operacijama, a podaci koji se

emituju se keširaju u serijskom obliku i dekodiraju pre izvršavanja svakog zadatka. Dakle, kreiranje emitovanih promenljivih je korisno samo kada zadaci u više faza zahtevaju iste podatke.

4. PODEŠAVANJE EKSPERIMENTALNOG OKRUŽENJA

Za potrebe eksperimenta, neophodno je podesiti eksperimentalno okruženje. Prvo, potrebno je kreirati nalog na Amazon Web Services (AWS) platformi kako bi dobili pristup Elastic MapReduce (EMR) usluzi. EMR je AWS-ova usluga koja omogućava lako korišćenje Apache Spark na velikim skupovima podataka. Nakon što je nalog kreiran i nakon prijavljivanja na AWS konzolu, potrebno je pratiti sledeće korake za konfiguraciju klastera na EMR:

- Kreiranje klastera, gde je potrebno izabrati verziju EMR, u ovom slučaju je to bila 6.10.0.
- Konfiguracija klastera, gde je potrebno izabrati tip instanci za klaster. Za ovaj eksperiment su izabrane instance m5.xlarge sa 4 vCPU jezgra (Sl. 7.).
- Odabir softvera, gde je potrebno izabrati željene softverke package za instalaciju na klasteru. Ovo uključuje Apache Spark čija je verzija 3.3.1.
- Konfigurisanje sigurnosti, gde je potrebno podesiti pristupne dozvole i sigurnost za klaster kako bismo osigurali da samo ovlašćeni korisnici mogu pristupiti podacima i resursima klastera.
- Završetak konfiguracije. Nakon što se završi sa konfiguracijom, potrebno je potvrditi sve postavke i sačekati 15 minuta nakon čega će biti moguće izvršavanje eksperimenta



Node type	Name	ID	Status	Instances	Instance type
Primary	Primary	ig-277JPCNTFMQM9	Terminated	0	m5.xlarge
Core	Core	ig-XRWXVWMKT8E	Terminated	0	m5.xlarge
Task	Task - 1	ig-3ENZWEI5GBUN2	Terminated	0	m5.xlarge

Sl. 7. Konfiguracija klastera, korišćene instance

Dodatno, potrebno je kreirati S3 (Amazon Simple Storage Service) bucket u kome je potrebno preneti kod koji je razvijen uz pomoć tehnologije PySpark i sve relevantne datoteke. To je omogućilo pristup kodu i podacima sa klastera koji je pokrenut na EMR.

Pomoću ove kombinacije EMR klastera i S3 bucketa, stvoreno je optimalno okruženje za izvršavanje eksperimenta.

Konačno, pseudokod GWO algoritma koji je modifikovan uz pomoć tehnologije PySpark radi paralelnog izvršavanja je:

Initialization:

- Distribute the objective function to all worker nodes
- Initialize the population
- Start SparkContext

For each iteration until the maximum number of iterations:

- Broadcast the population to all worker nodes
- Calculate fitness values for each solution in parallel
- Collect fitness values from all worker nodes
- Sort the population based on fitness values
- Update the values of α , β , δ
- For each solution in the population:
 - Calculate new positions using the formulas
- Update the population with the new positions

Calculate fitness values for each solution in the population after optimization

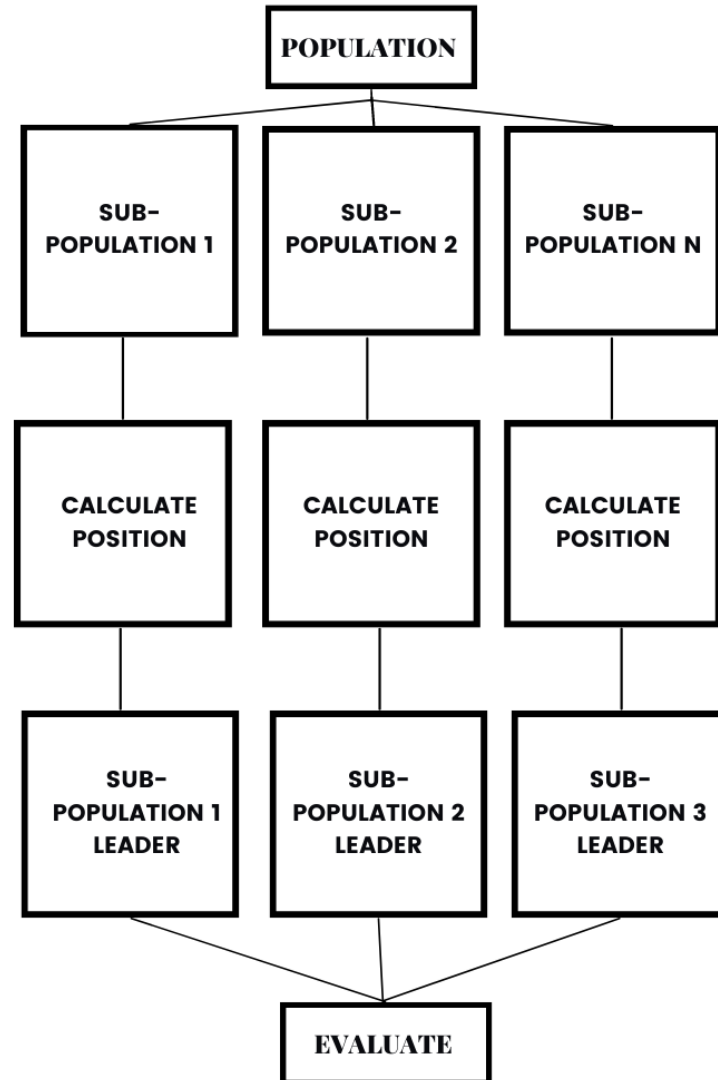
Find the best solution based on fitness values

Return the best solution and its fitness value

Posmatrajući ovaj pseudokod vidimo da je implementacija algoritma podeljena na dva koraka, gde prvi korak jeste inicijalizacija. U ovom koraku importujemo potrebne biblioteke, postavljamo parametre za optimizaciju, kao što su funkcija, dimenzionalnost, veličina populacije i maksimalan broj iteracija. Drugi i najbitniji korak je optimizacija. Ovo je korak u kom primenjujemo GWO algoritam, stvaramo SparkContext za distribuirano izračunavanje i izvršavamo petlju za svaku iteraciju. Za svaku iteraciju distribuiramo populaciju na sve radne čvorove, izračunavamo prilagođenost za svaku jedinku u populaciji paralelno, sortiramo populaciju prema prilagodljivosti, ažuriramo poziciju alfa, beta i delta, i za svaku jedinku u populaciji ažuriramo poziciju. Nakon petlje, izračunavamo prilagodljivost za svaku jedinku nakon optimizacije i pronalazimo najbolje rešenje.

Populacija je distribuirana na radne čvorove Spark-a uz pomoć metode “`sc.parallelize`”. Svaki radni čvor obrađuje deo populacije nezavisno, i to se može

smatrati implicitnom subpopulacijom. Rezultati se zatim sakupljaju pomoću “collect” i kombinuju nakon izvršenih paralelnih operacija. Vizuelni prikaz se može dočarati *Slikom 8*.



Sl. 8. Paralelno izvršavanje GWO algoritma na Sparku

5. OPTIMIZACIONI ALGORITAM SIVI VUK BAZIRAN NA SPARKU

Cilj GWO algoritma zasnovanog na Spark-u je da poboljša performanse rezultata izračunavanja za pronalaženje optimalnog rešenja. Štaviše, rezultat proizvodi veću skalabilnost algoritma upravljanjem složenijim problemima

optimizacije. Ideja o paralelizaciji GWO algoritma pokazala je efikasno poboljšanje ukupnih računarskih performansi.

S obzirom da Apache Spark radi na RDD-ovima, tako što kešira podatke u memoriji, rezultati bi trebalo da pokažu značajna poboljšanja u brzini računarske propusnosti i skalabilnosti algoritma.

Paralelizam GWO algoritma počinje inicijalizacijom populacije vukova. Aplikaciju zatim postavljamo na klaster radi poboljšanja računarske propusnosti. Apache Spark inicijalizuje populaciju keširanjem podataka u memoriji, kao i nakon podele populacije na subpopulacije koje traže optimalno rešenje. Svaka podpopulacija formira RDD u Spark-u, pri čemu svaki RDD sadrži različite particije koje predstavljaju agente koji imaju rešenje. Na kraju, primenjujemo operaciju sortiranja na poslednji RDD i generišemo optimalni globalni rezultat.

6. EVALUACIJA

U cilju procene performansi paralelizovanog GWO algoritma pomoću programskog okvira Apache Spark, algoritam je implementiran i testiran na osam poznatih funkcija kao što je prikazano u *Tabeli 2*. Nakon toga su upoređeni rezultati sa rezultatima serijskog izvršavanja GWO algoritma, koji su prikazani u *Tabeli 3*. A, konačni rezultati paralelizovanog GWO algoritma korišćenjem programskog okvira Apache Spark su predstavljeni u *Tabeli 4*. Vidljivo je da su rezultati u odnosu na serijsko izvršavanje značajno bolji. Ovo ukazuje na efikasnost paralelizovanog pristupa i sposobnost okvira Apache Spark da distribuira izračunavanje na više čvorova u klasteru.

Važno je napomenuti da je veličina populacije postavljena na 30 i da su funkcije testirane kroz 100 iteracija. Međutim, eksperimenti sa većom veličinom populacije i većim brojem iteracija mogu pružiti dodatne informacije o performansama algoritma i njegovoj skalabilnosti. Opis okruženja na kojem se izvelo testiranje je jasno prikazano u *Tabeli 1*.

Okruženje	Amazon EMR
vCPU	4 cores
Tip instance	m5.xlarge
Apache Spark framework	Apache Spark 3.3.1
Izdanje softvera	Emr-6.10.0

Tabela 1. Podešavanje Sistema

Ime	Funkcija
Sum Squares	$f(x) = \sum_{i=1}^D ix_i^2$
Step 2	$f(x) = \sum_{i=1}^D (x_i + 0.5)^2$
Rastrigin	$f(x) = 10d + \sum [x_i^2 - 10 \cos(2\pi x_i)]$
Sphere	$f(x) = \sum_{i=1}^D x_i^2$
Schwefel 2.20	$f(x) = \sum_{i=1}^D x_i ^2$
Csendes	$f(x) = \sum_{i=1}^D x_i^6 \left(2 + \sin\left(\frac{1}{x_i}\right)\right)$
Quartic	$f(x) = \sum_{i=1}^D ix_i^4 + \text{random}[0,1]$
Schwefel	$f(x) = \sum_{i=1}^D -x_i \sin(\sqrt{ x_i })$

Tabela 2. Testirane funkcije

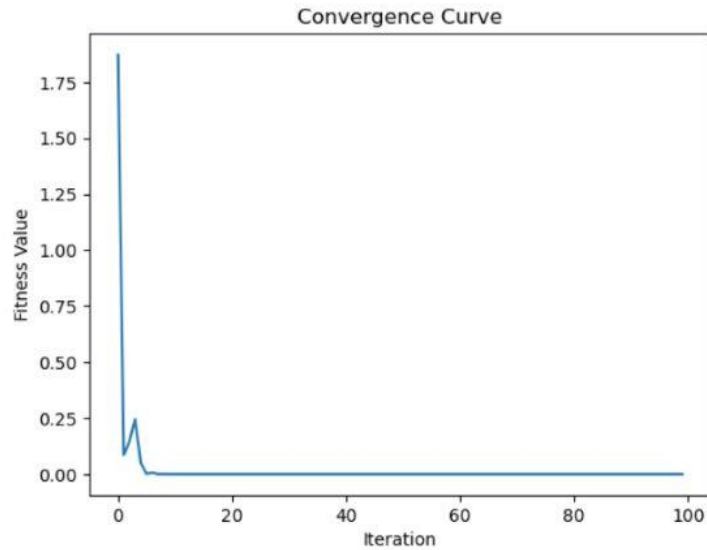
Tabela 3. i Tabela 4. imaju za cilj pregledno prikazivanja rezultata primene GWO algoritma na svaku funkciju posebno. U tabelama, kolona *Function* prikazuje naziv funkcije na kojoj se primenjuje GWO algoritam. Za svaku od ovih funkcija, koje su prethodno definisane u Tabeli 2., je cilj minimizovati vrednost funkcije. Kolona *Best* predstavlja najbolje postignuti rezultat nakon 100 iteracija GWO algoritma na određenu funkciju. Ovo je minimalna vrednost funkcije koju je algoritam uspeo postići. Što je vrednost niža, to je rezultat bolji. Kolona *Worst* predstavlja najgori postignuti rezultat nakon 100 iteracija GWO algoritma za određenu funkciju. *Mean* predstavlja srednju vrednost rezultata, i konačno *BF (Best Fitness)* kolona prikazuje rešenje (vrednost varijable) koje je dalo najbolji rezultat, odnosno minimalnu vrednost funkcije nakon 100 iteracija GWO algoritma. Dakle, *BF* nam omogućava da vidimo koja kombinacija varijabli je dala najbolje rezultate.

Function	Best	Worst	Mean	BF
Sum Squares	-1.537e-07	4.746e-03	-3.491e-07	2.100e-15
Step 2	-0.024258	-0.516009	-0.494969	0.2267502
Rastrigin	-2.372e-07	-3.265e-07	2.724e-07	1.506e-10
Sphere	1.739e-08	2.713e-08	2.221e-08	5.183e-15
Schwefel 2.20	-6.303e-10	1.415e-09	1.165e-09	1.069e-08
Csendes	-2.435e-07	3.219e-07	2.551e-07	6.666e-07
Quartic	-7.014e-07	-2.108e-05	4.841e-07	3.122e-24
Schwefel	-0.781	-26.1729	-26.1723	-25.92956

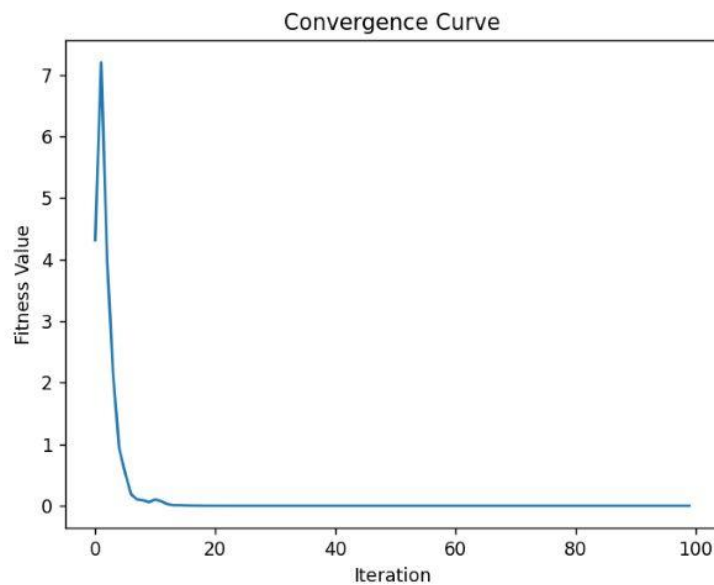
Tabela 3. GWO optimalna rešenja bez paralelizacije

Na Slici 9. i Slici 10. su ilustrovani rezultati GWO optimalnog rešenja *Quartic* i *Sum Squares* funkcije koristeći biblioteku matplotlib, koja pruža moćne mogućnosti za vizualizaciju podataka, radi poređenja sa vizuelnim prikazom istih funkcija (Sl. 11., Sl. 12.) izvršene paralelnim pristupom GWO algoritma na Spark-u.

Na slikama je prikazana *Fitness Value* u odnosu na *Iteration*, gde su one međusobno nezavisne. Na x-osi je prikazana vrednost *Iteration*, dok je na y-osi prikazana vrednost *Fitness Value*.



Sl.9. GWO optimalno rešenje za *Quartic* funkciju

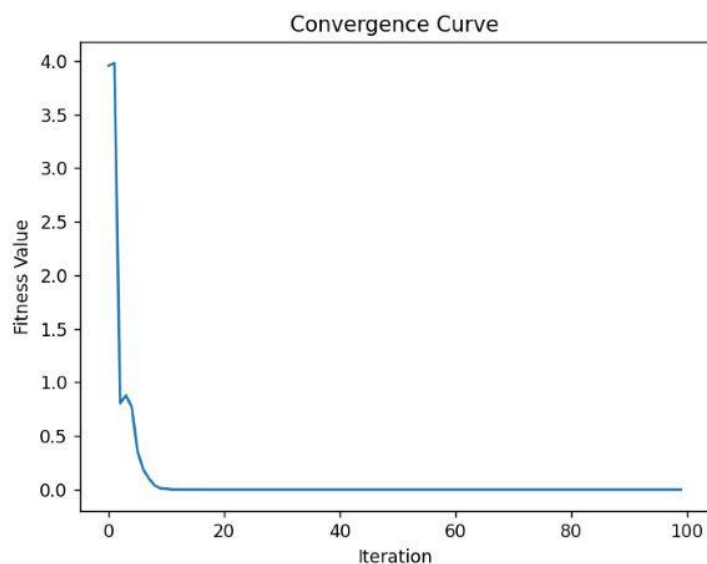


Sl.10. GWO optimalno rešenje za *Sum Squares* funkciju

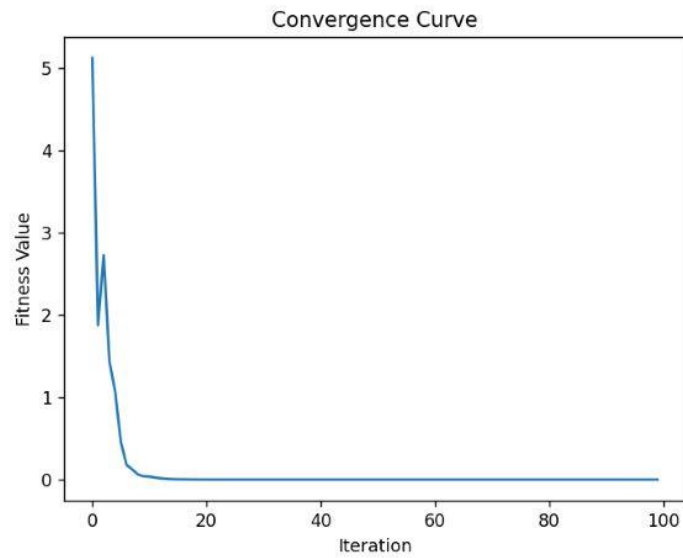
Function	Best	Worst	Mean	BF
Sum Squares	-9.131e-08	1.835e-01	1.835e-02	5.049e-13
Step 2	0.00121	-0.5021	-0.4981	0.5004
Rastrigin	-4.955e-06	5.314e-06	1.228e-07	3.907e-08
Sphere	-3.956e-07	2.860e-07	-4.949e-08	7.223e-13
Schwefel 2.20	-7.197e-08	6.885e-08	-1.013e-10	6.359e-07
Csendes	-7.196e-08	6.885e-08	0.000000	1.959e-29
Quaric	-2.823e-07	1.8535	1.6798e-07	7.809e-22
Schwefel	1.971e-04	5.2936	3.4898	-27.666

Tabela 4. Rešenja paralelizovanog GWO algoritma uz pomoću programskog okvira Apache Spark

Analizirajući podatke iz *Tabele 4.* primećujemo da rezultati nekih funkcija, a u ovom slučaju su to funkcije Rastrigin, Sphere i Schwefel 2.20, nisu tako dobri kao kod algoritma koji nije paralelizovan. Postoji nekoliko faktora koje treba uzeti u obzir prilikom objašnjavanja tog fenomena. Prvi faktor je slabija skalabilnost. Paralelizacija može puno doprineti u smislu bržeg izvršavanja algoritma, ali ne uvek i boljih rezultata. Postoje situacije u kojima algoritam gubi na skalabilnosti prilikom paralelizacije, što može uticati na kvalitet dobijenih rezultata. To se najčešće dešava zbog složenosti problema, komunikacije između paralelnih procesa ili drugih faktora koji ograničavaju učinkovitost paralelnog izvršavanja. Konfiguracija i parametri takođe mogu biti razlog, jer u ovom slučaju potrebno je pažljivo prilagoditi fakore kao što su broj radnih čvorova, veličina populacije, broj iteracija i ostalih parametara kako bi se postigao kvalitetet rešenja.



Sl. 11. Paralelno izvršren GWO algoritam za Quartic funkciju



Sl.12. Paralelno izvršen GWO algoritam za Sum Squares funkciju

7. ZAKLJUČAK

GWO algoritam je nedavno predloženi algoritam za optimizaciju rojeva inspirisan prirodom, koji polako biva zapažen i korišćen u različitim disciplinama zbog svojih performansi u rešavanju problema optimizacije.

Vremenska efikasnost GWO algoritma se može drastično povećati kada se primeni za rešavanje složenih problema iz stvarnog sveta.

Paralelna implementacija GWO algoritma koristeći programski okvir Apache Spark pruža značajne prednosti u pogledu vremenske efikasnosti i skalabilnosti, posebno kada se koristi za rešavanje složenih problema iz stvarnog sveta.

Kroz ovaj rad izvršeno je ispitivanje performansi paralelne implementacije GWO algoritma na osam različitih funkcija. Rezultati su pokazali da je paralelna implementacija značajno ubrzala proces optimizacije u poređenju sa serijskom implementacijom istog algoritma. Ova poboljšanja omogućavaju brže pronalaženje optimalnog rešenja za različite probleme optimizacije.

Kao što rekoh, pored ubrzanja, paralelna implementacija GWO algoritma korišćenjem Apache Spark-a pruža skalabilnost. To znači da se algoritam lako može prilagoditi za rad sa većim skupovima podataka ili za rešavanje problema veće složenosti. Ova fleksibilnost je od suštinskog značaja za probleme iz stvarnog sveta, gde se često suočavamo sa kompleksnim ciljnim funkcijama.

Sve prethodno navedene vrline ovog istraživanja mogu omogućiti istraživačima i praktičarima da efikasno rešavaju svoje probleme sa značajnim vremenskim uštedama, a u tom slučaju meni će drago biti što sam mali deo toga.

8. PRILOZI

```
In [1]: from pyspark import SparkContext, SparkConf
import random
import numpy as np
import time
import matplotlib.pyplot as plt
from objective_function import objective_function

class GrayWolfOptimizer:
    def __init__(
        self,
        objective_function,
        num_dimensions,
        population_size,
        max_iterations,
        alpha=0.5,
        beta=0.5,
        delta=2.0,
    ):
        self.objective_function = objective_function
        self.num_dimensions = num_dimensions
        self.population_size = population_size
        self.max_iterations = max_iterations
        self.alpha = alpha
        self.beta = beta
        self.delta = delta
        self.fitness_history = []

    def optimize(self):
        conf = SparkConf().setAppName("GrayWolfOptimizer")
        sc = SparkContext(conf=conf)

        # Distribute the objective function to all worker nodes
        sc.addPyFile(__file__)
        objective_function = sc.broadcast(self.objective_function)

        # Initialize the population
        population = np.random.uniform(
            low=-1, high=1, size=(self.population_size, self.num_dimensions)
        )

        for iteration in range(self.max_iterations):
            # Broadcast the population to all worker nodes
            population_bc = sc.broadcast(population)

            # Calculate fitness values for each solution in parallel
            fitness_values = (
                sc.parallelize(population_bc.value)
                .map(lambda x: objective_function.value(x))
                .collect()
            )
            fitness_values = np.array(fitness_values)
            self.fitness_history.append(min(fitness_values))

            # Sort the population based on fitness values
            sorted_indices = np.argsort(fitness_values)
            population = population[sorted_indices]

            # Update the alpha, beta, and delta values
            alpha_position = population[0]
            beta_position = population[1]
            delta_position = population[2]

            for i in range(self.population_size):
                a = 2 * (
                    1 - (iteration / self.max_iterations)
                ) # Linearly decreased from 2 to 0

                # Update each wolf's position
                for j in range(self.num_dimensions):
                    r1, r2 = random.random(), random.random()
                    A1 = 2 * a * r1 - a
                    C1 = 2 * r2

                    D_alpha = abs(C1 * alpha_position[j] - population[i, j])
                    X1 = alpha_position[j] - A1 * D_alpha

                    r1, r2 = random.random(), random.random()
                    A2 = 2 * a * r1 - a
                    C2 = 2 * r2

                    D_beta = abs(C2 * beta_position[j] - population[i, j])
                    X2 = beta_position[j] - A2 * D_beta
```

```

        r1, r2 = random.random(), random.random()
        A3 = 2 * a * r1 - a
        C3 = 2 * r2

        D_delta = abs(C3 * delta_position[j] - population[i, j])
        X3 = delta_position[j] - A3 * D_delta

        population[i, j] = (X1 + X2 + X3) / 3 # Update position

    self.fitness_history.append(fitness_values[sorted_indices[0]])

    # Calculate fitness values for each solution after optimization
    fitness_values = np.array([self.objective_function(x) for x in population])

    # Find the best solution
    best_solution_index = np.argmin(fitness_values)
    best_solution = population[best_solution_index]
    best_fitness = fitness_values[best_solution_index]

    return best_solution, best_fitness

optimizer = GrayWolfOptimizer(
    objective_function=objective_function,
    num_dimensions=10,
    population_size=30,
    max_iterations=100,
)

# Run the optimization
start_time = time.time()
best_solution, best_fitness = optimizer.optimize()
end_time = time.time()
fitness_history = optimizer.fitness_history

# Print the results
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)

execution_time = end_time - start_time
print("Execution time:", execution_time, "seconds.")

# Plot the convergence curve
plt.plot(range(len(fitness_history)), fitness_history)
plt.xlabel("Iteration")
plt.ylabel("Fitness Value")
plt.title("Convergence Curve")
plt.show()

```

9. REFERENCE

1. Seyedali Mirjalili, Andrew Lewis, Seyed Mohammad Mirjalili - Grey Wolf Optimizer
2. Mohammad Dehghani, Eva Trojovska, Tomas Zuscak – A new human-inspired metaheuristic algorithm for solving optimization problems based on mimicking sewing training
3. Fister I Jr, Yang X-S, Fister IFD. A brief review of nature-inspired algorithms for optimization. CoRR. 2013;80:abs/1307.4186
4. Peifeng Niu, Songpeng Niu, Nan Iiu - The defect of the Grey Wolf optimization algorithm and its verification method
5. Zhang Z, Wang W, Gao N, Zhao Y. Spark-based distributed quantum-behaved particle swarm optimization algorithm. Cooperative Design, Visualization, and Engineering. Springer International Publishing; 2018:295-298.
6. Juwei S, Yunjie Q, Farooq MU, et al. Clash of the titans: MapReduce vs. spark for large scale data analytics. Proc VLDB Endow. 2015;8(13):2110-2121.
7. YasserK,MohammadA, ImtiazA. Distributed whale optimization algorithm based onMapReduce.Concurr Comput Pract Exper. 2019;31(1):e4872.
8. Mohammad Alshayeji, Bader Behbehani, Imtiaz Ahmad - Spark-based parallel processing whale optimization algorithm
9. <https://spark.apache.org/>
10. Zhang, S., Zhou, Y., Li, Z., Pan, W., “Grey wolf optimizer for unmanned combat aerial vehicle path planning”,
11. Jain, U., Tiwari, R., Godfrey, W. W., “Odor source localization by concatenating particle swarm optimization and Grey Wolf optimizer”
12. Vinothini, J., Bakkiyaraj, R. A., “Grey Wolf Optimization Algorithm for Colour Image Enhancement Considering Brightness Preservation Constraint”
13. Jeet, K., “Grey wolf algorithm for software organization”
14. Li, Q., Chen, H., Huang, H., Zhao, X, Cai, Z. N., Tong, C., Liu, W., Tian, X., "An enhanced grey wolf optimization based feature selection wrapped kernel extreme learning machine for medical diagnosis"
15. <https://www.ibm.com/topics/apache-spark>
16. https://www.youtube.com/watch?v=MjHZ_60vyic