Implementacija algoritama optimizacije rojem čestica

Gazdek, Denis

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:249428

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Denis Gazdek

Implementacija algoritama optimizacije rojem čestica

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Denis Gazdek

JMBAG: 0016122863

Studij: Informacijsko i programsko inženjerstvo

Implementacija algoritama optimizacije rojem čestica

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Nikola Ivković

Varaždin, rujan 2021.

Denis Gazdek

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovog diplomskog rada su različite verzije algoritma optimizacije roja čestica te njihova implementacija i evaluacija. Optimizacija rojem čestica je relativno mlado polje istraživanja; prva verzija algoritma optimizacije rojem čestica definirana je tek 1995. godine. U prvom, teorijskom dijelu ovog rada sam prikazat originalni algoritam optimizacije rojem čestica, motivaciju znanstvenika koji su ga prvi definirali te kako su ga, kroz vrijeme, različiti znanstvenici pokušali poboljšati, bilo kroz podešavanje parametara, kroz kombiniranje algoritma s nekim drugim algoritmima optimizacije, ili kroz kompletnu eliminaciju ulaznih parametara. U drugom, praktičnom dijelu rada sam implementirao neke od opisanih algoritama u programskom jeziku Python, s različitim topologijama čestica te sam algoritme usporedio i analizirao na nekoliko problema.

Ključne riječi: optimizacija, algoritam, inteligencija rojeva, računalna inteligencija, Python

Sadržaj

1. Uvod

Cilj ovog rada je opisati originalni algoritam optimizacije rojem čestica. Nakon opisa originalnog algoritma i opisa motivacije autora algoritma ovaj rad opisuje promjene nad algoritmom kojim se pokušava poboljšati originalni algoritam optimizacije rojem čestica, najčešće manipuliranjem vektora brzine ili promjenom topologije čestica. Nakon toga, ovaj rad predstavlja neke od hibridnih algoritama, koji algoritam optimizacije rojem čestica kombinira s drugim algoritmima optimizacije, kao što su genetski algoritam, algoritam diferencijalne evolucije i algoritam simuliranog prekaljivanja. Zadnji algoritam koji je ovaj rad predstavio je adaptivni algoritam optimizacije rojem čestica TRIBES, koji eliminira inicijalizacijske parametre kako bi se algoritam sam prilagodio problemu koji se optimizira. U praktičnom dijelu, ovaj rad prikazuje implementacije algoritama navedenih u teorijskom dijelu u programskom jeziku Python. Nakon implementacije, testirao sam algoritme na nekoliko problema te sam usporedio performanse algoritama kroz nekoliko metrika: broj iteracija potrebnih da se dođe do prihvatljivog rješenja, vrijeme potrebno da se izvrši 1000 iteracija algoritma, rezultat nakon 1000 iteracija te postotak izvršavanja koji su došli do rješenja.

1.1. Optimizacija

Optimizacija je proces pronalaska optimalnog rješenja nekog problema. Algoritmi optimizacije pretražuju neki prostor rješenja kako bi došli do što boljeg rješenja. Optimum nekog problema može biti maksimum ili minimum. Problemi koji se optimiziraju mogu biti ograničeni ili neograničeni. Neograničeni problemi nemaju neko ograničenje na domenu problema; prihvatljiva rješenja mogu biti iz cijelog područja domene. Ograničeni problemi, kao što ime nalaže, imaju neko ograničenje na domeni. Ako neko rješenje dobiveno optimizacijom krši neko od ograničenja, to rješenje se smatra nevažećim [1].

1.2. Inteligentni algoritmi

Inteligentni algoritmi su algoritmi koji pokušavaju imitirati prirodnu inteligenciju i biološke procese kako bi riješili neki problem. Takvi algoritmi su dizajnirani tako da se mogu prilagoditi širokoj lepezi problema. Često koriste neku razinu nasumičnosti kako bi dobili bolje rješenje; takvi algoritmi se još zovu stohastički algoritmi. Inteligentni algoritmi često koriste neku populaciju jedinki, koje međusobno komuniciraju te iz te komunikacije proizlazi „uvjetovano ponašanje“ (eng. *emergent behavior*), tj. iz komunikacije pojedinaca proizlazi

inteligentno ponašanje cijele populacije. Inteligentni algoritmi reagiraju na promjene u okolini te se prilagođuju promjenama.

Dva ključna pojma kod inteligentnih algoritama, i kod optimizacije općenito, su istraživanje (eng. *exploration*) i iskorištenje ili eksploatacija (eng. *exploitation*). Istraživanje znači da algoritam široko pretražuje po prostoru pretrage. Nakon istrage, algoritam iskorištava poznata dobra rješenja, tj. oko tog rješenja detaljnije pretražuje prostor pretrage, u nadi da će naći optimum [1].

1.3. Evolucijski algoritmi

Evolucijski algoritmi su algoritmi koji evoluira rješenje kroz više iteracija. Svaka iteracija naziva se jednom generacijom. Svaku iteraciju, populacija jedinki poboljšava svoje rješenje, kako bi se na kraju algoritma došlo do dobrog rješenja. Evolucijski način na neki način oponašaju proces prirodne evolucije. Često, no ne uvijek (ovisno o algoritmu), u evolucijskom algoritmu lošije jedinke izumiru, a bolje jedinke stvaraju nove čestice [1].

1.4. Optimizacija rojem čestica

Algoritam optimizacije rojem čestica je stohastički evolucijski algoritam. Algoritam je evolucijski jer jedinke, koje se u algoritmu nazivaju čestice, kroz vrijeme poboljšavaju svoje rješenje. No za razliku od većine evolucijskih algoritama, nove čestice se ne stvaraju, niti se stare čestice brišu (osim kod adaptacijske verzije algoritma opisane u šestom poglavlju). Prvo je opisan 1995. godine i inspiriran je kretanjem jata ptica u prirodi [1].

2. Metode i tehnike rada

Praktični dio ovog rada rađen je u programskom jeziku Python te moduli NumPy i matplotlib. Sam kod je pisan u okruženju Visual Studio Code. Python je programski jezik primarno dizajniran za manipuliranje velikim setovima podataka te za učenje programiranja. Izvorni kod jezika je izdan pod otvorenom licencom te je za njega izrađen veliki broj specijaliziranih modula, koji programeru omogućavaju brzo definiranje i rješavanje specifičnih problema [2].

NumPy je modul koji olakšava matematičke operacije u jeziku Python, pogotovo s matricama i vektorima. Kao i Python, NumPy je izdan pod otvorenom licencom te je izvorni kod dostupan na internetu. Temeljni tip podataka kod NumPy je *ndarray*, koji predstavlja n-dimenzionalni niz objekata, koji omogućavaju efikasnije izvršavanje operacija nad matricama [3].

Matplotlib je Python modul koji omogućava generiranje grafova i ostalih vizualizacija u Pythonu. Svi grafovi funkcija u ovome radu izrađene su s bibliotekom matplotlib [4].

Visual Studio Code je editor koda dostupan na operativnim sustavima Windows, Linux i macOs. Razvio ga je Microsoft te je originalno bio zatvorenog koda. Danas je većina izvornog koda otvoren i dostupan na internetu [5]. Koristio sam Visual Studio Code umjesto editora IDLE uključenog s Pythonom, jer omogućava lakše upravljanje datotekama u Python programima s više datoteka te uključuje mnogo više mogućnosti prilagodbe te je moguće koristiti IntelliSense, koji automatski dopunjuje Python kod.

3. Originalni algoritam optimizacije rojem čestica

Inspirirani kretanjem roja ptica i grupa riba te potaknuti radovima u području simulacije na istu temu, Kennedy i Eberhart definiraju algoritam koji imitira njihovo prirodno kretanje kod pronalaska najboljeg izvora hrane. Taj algoritam nazivaju algoritmom optimizacije rojem čestica. Algoritam koji su definirali je ispao dosta uspješan nad funkcijama koje se koriste za benchmarking ostalih optimizacijskih algoritama.

3.1. Motivacija

Algoritam optimizacije rojem čestica je prvi puta opisan 1995. godine u radu „Particle swarm optimization“ Jamesa Kennedyja i Russela Eberharta. Svoj rad temelje na radovima Craiga Reynoldsa iz 1987. godine i Franka Heppnera i Ulfa Grenandera iz 1990. godine [6]. Reynoldsa je zanimalo simuliranje kretanja grupe životinja (u radu se je fokusirao na jato ptica) u svrhu pojednostavljenja procesa računalne animacije, kako animatori ne bi morali ručno modelirati svaku jedinku [7]. Heppner i Grenander su pokušali modelirati let ptica te odrediti set pravila pomoću kojih se kaotično jato počne kretati sinkronizirano i obrnuto, bez nekog očitog „vođe“ koji bi jato vodio. Heppner i Grenander u svoj model uvode točku „legla“ (eng. *roost*), oko koje jato oblijeće prije nego sleti te koje ih tijekom leta privlači. [8]. Oba rada definiraju model u kojem pojedine ptice definiraju svoje pokrete prema svojoj okolini, tj. prema ostalim pticama u jatu, gdje se pokušavaju držati što bliže jatu bez da se sudare te gdje se u pokretima pojedinih ptica uvodi „ludost“ (eng. *crazyness*), kako jato ne bi odredilo jedan smjer leta, bez ikakve promjene. Ta „ludost“ je u jato uvela varijaciju u letu te su izgledi jata bili vjerniji stvarnom životu.

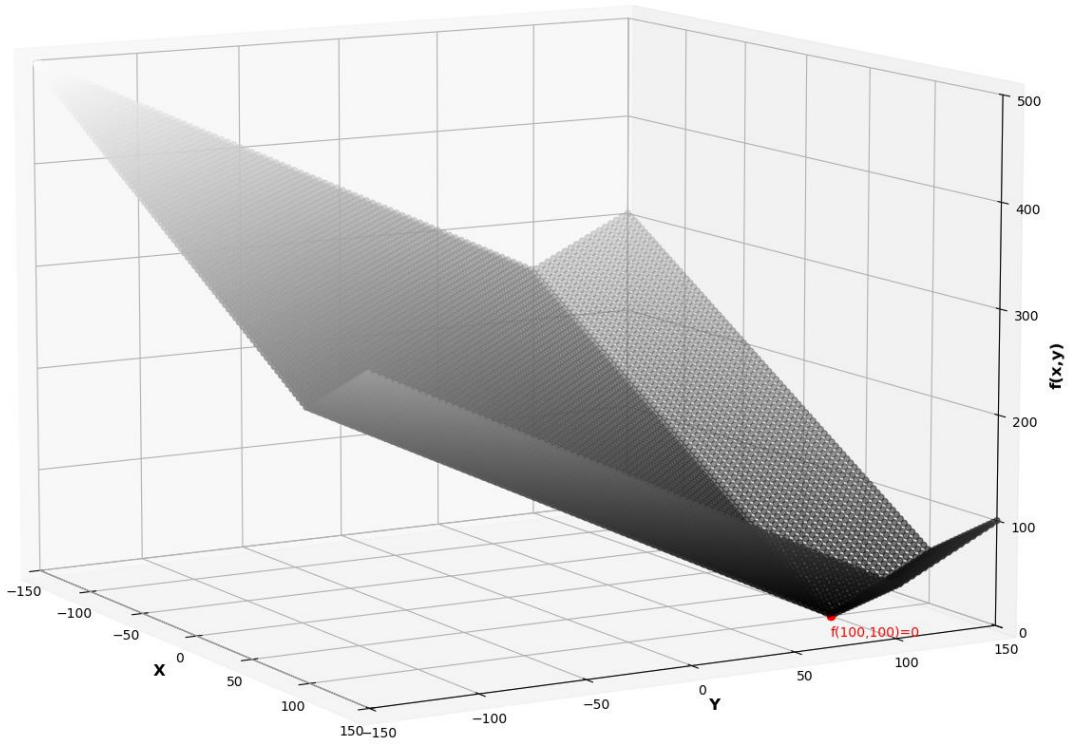
3.2. Algoritam

Kennedy i Eberhart su točku legla zamijenili s točkom izvora hrane te su zaključili da, kako jato ptica vrlo brzo nađe najbolje izvore hrane u nekom području o kojem nemaju prethodne informacije, na neki način jato ptica kolektivno dođe do najboljeg rješenja. Nakon toga pojam ptice mijenjaju s pojmom agenta. Svaki agent su definirali tako da, s obzirom na svoju poziciju, evaluira funkciju: [6].

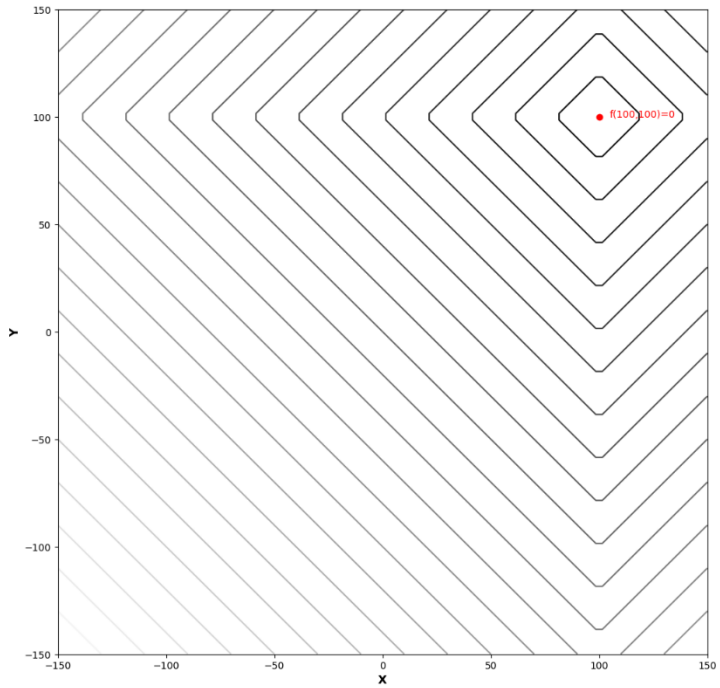
$$Eval = \sqrt{(presentx - 100)^2} + \sqrt{(presenty - 100)^2}$$

U kojoj je $presentx$ trenutna vrijednost x koordinate agenta, a $presenty$ trenutna vrijednost y koordinate agenta. Na slici 1 se nalazi 3D plot funkcije, dok se na slici 2 nalazi

konturni graf iste funkcije. Na tim slikama se može vidjeti da se minimum funkcije nalazi u točki $(100,100)$ te iznosi 0.



Slika 1: 3D plot funkcije "Eval". Na grafu je crveno označen optimum funkcije, točka s koordinatama (100, 100) i vrijednosti funkcije 0



Slika 2: Konturni graf funkcije "Eval". Na grafu je crveno označen optimum funkcije, točka s koordinatama (100, 100) i vrijednosti funkcije 0

Svaki agent pamti najbolju vrijednost funkcije koju je postigao te koordinate na kojima je postigao tu vrijednost. Osobne najbolje vrijednosti pamte u varijabli $pbest[]$, a koordinate u $pbestx[]$ i $pbesty[]$. Ovdje su te varijable redovi, u gdje i-ti element reda predstavlja vrijednosti i koordinate za i-ti agent.

Koordinate svakog agenta su se, nakon svake iteracije, mijenjale ovako: ako je x koordinata agenta u trenutnoj iteraciji bila veća od najbolje poznate točke $pbestx$, agentov vektor kretanja po X osi (njegova x-brzina, ili v_x) se smanjuje za umnožak nasumičnog broja između 0 i 1 i nekog parametra sustava ($p_{increment}$).

$$v_x[] = v_x[] - rand() \cdot p_{increment}$$

Ako je x koordinata agenta bila manja od najbolje poznate točke, onda se taj umnožak dodaje vrijednosti v_x umjesto da se oduzima.

$$v_x[] = v_x[] + rand() \cdot p_{increment}$$

Na slični način se određuje i vektor kretanja po Y osi (y-brzina, ili v_y). Ako je y koordinata agenta veća od idealnog rješenja $pbesty$, onda se od v_y smanjuje za umnožak broja između 0 i 1 i $p_{increment}$. Ako je manja, onda se za isti povećava.

$$v_y[] = v_y[] - rand() \cdot p_{increment}$$

$$v_y[] = v_y[] + rand() \cdot p_{increment}$$

Osim što pamte osobno najbolje rješenje, agenti pamte i najbolje rješenje koje je našao bilo koji od agenata te koordinate tog rješenja. Na isti način kako se podešavaju v_x i v_y brzine s obzirom na osobno najbolje rješenje, iste brzine se podešavaju s obzirom na globalno poznato najbolje rješenje: Ako je x koordinata agenta manja od najboljeg globalno znanog rješenja, vrijednosti v_x se dodaje umnožak nasumičnog broja između 0 i 1 i još jednog parametra sustava ($g_{increment}$). Ako je x koordinata veća od x koordinate najboljeg znanog rješenja, od vrijednosti v_x se taj umnožak oduzima. Ako je y koordinata agenta manja od najboljeg znanog rješenja, vrijednosti v_y se dodaje umnožak nasumičnog broja između 0 i 1 i parametra $g_{increment}$. Ako je y koordinata manja, taj umnožak se oduzima.

$$v_x[] = v_x[] - rand() \cdot g_{increment}$$

$$v_x[] = v_x[] + rand() \cdot g_{increment}$$

$$v_y[] = v_y[] - rand() \cdot g_{increment}$$

$$v_y[] = v_y[] + rand() \cdot g_{increment}$$

Nakon što su definirali način modifikacije brzine pojedinih agenata te su gore navedenom evaluacijskom funkcijom utvrdili da bi algoritam mogao davati dobre rezultate, Kennedy i Eberhart su počeli eliminirati parametre iz prethodnih radova koji ne utječu na rezultat. Dva parametra koja su uklonili iz algoritma su „ludost“ jata i ujednačavanje brzine između agenata. Nakon što su maknuli ta dva parametra, agenti se više ne kreću sinkronizirano kao jato, nego kao roj nezavisnih agenata. Sljedeće su zaključili da su točke p_{best} (osobno najbolje rješenje agenta) i g_{best} (globalno najbolje rješenje pronađeno) te pripadni parametri $p_{increment}$ i $g_{increment}$ ključni za rad algoritma. Kennedy i Eberhart također analiziraju omjer parametara $p_{increment}$ i $g_{increment}$: ako je parametar $p_{increment}$ znatno veći od parametra $g_{increment}$, roj agenata puno „luta“ po prostoru rješenja, bez da se odluče na neku točku. S druge strane, ako je parametar $g_{increment}$ znatno veći od parametra $p_{increment}$, roj se brzo skuplja u točkama lokalnih minimuma. Navode da je idealno odrediti parametre $p_{increment}$ i $g_{increment}$ tako da budu otprilike jednaki. Za svoju verziju algoritma uzimaju da su $p_{increment} = g_{increment} = 2$, kako bi srednja vrijednost $2 * rand()$ bio 1, tj. kako bi, u 50 posto slučajeva, čestica premašila cilj. Uz to, generaliziraju algoritam na više dimenzija te dolaze do završne formule:

$$v_{id}^{k+1} = v_{id}^{k} + 2 \cdot rand() \cdot \left(p_{id}^{k} - X_{id}^{k}\right) + 2 \cdot rand() \cdot \left(p_{gd}^{k} - X_{id}^{k}\right)$$

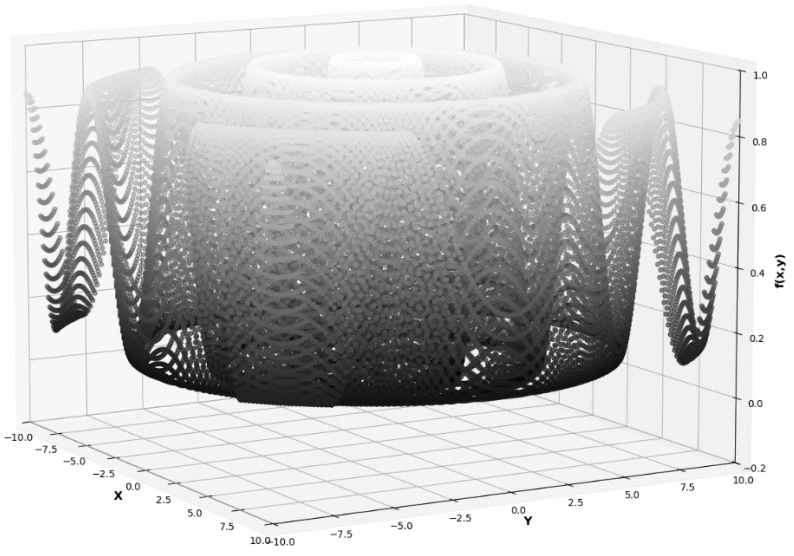
Gdje su v_{id}^{k} komponenta brzine i-te čestice u d-toj dimenziji u k-toj iteraciji algoritma, p_{id}^{k} pozicija osobnog najboljeg rješenja i-te čestice u d-toj dimenziji u k-toj iteraciji algoritma, X_{id}^{k}

pozicija i-te čestice u d-toj dimenziji u k-toj iteraciji algoritma, a p_{gd}^{k} pozicija globalno najbolje poznatog rješenja u d-toj dimenziji u k-toj iteraciji algoritma [6].

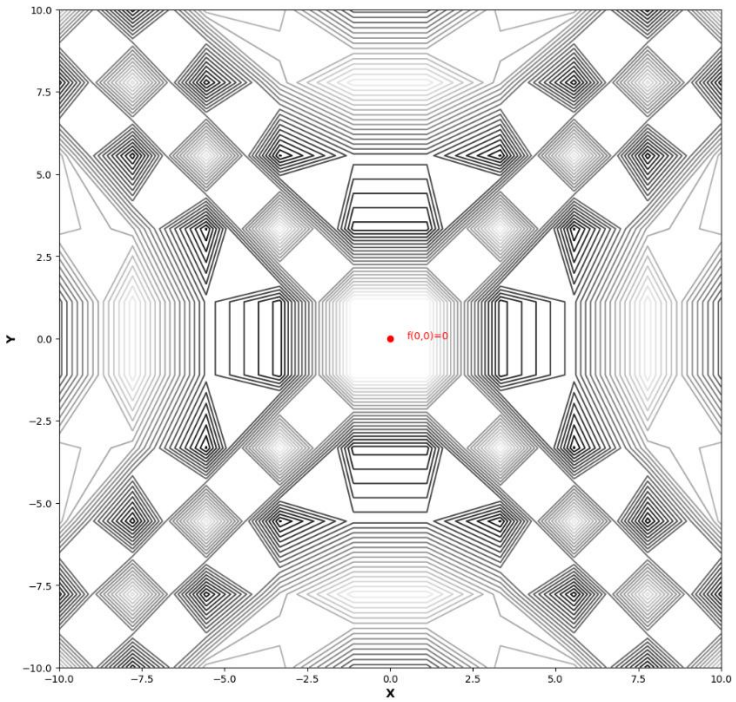
3.3. Rezultati algoritma

Kennedy i Eberhart su svoj algoritam testirali, ne samo na prije navedenoj evaluacijskoj funkciji, nego su ga iskoristili i za treniranje neuronske mreže, pošto je trening neuronske mreže zapravo problem optimizacije njenih parametara. Neuronska mreža koju su uzeli za inicijalno testiranje sastojala se je od dva ulazna neurona, jednog izlaznog neurona, i tri skrivena neurona u jednom sloju te opisuje neuronsku mrežu za rješavanje funkcije XOR. Ukupno, takva neuronska mreža sastoji se od 13 parametara (tri težinske vrijednosti neurona u skrivenom sloju, jedna težinska vrijednost neurona u izlaznom sloju te devet težinskih vrijednosti veza između neurona).

Neuronske mreže se najčešće optimiziraju metodom propagacije unatrag, pa su Kennedy i Eberhart htjeli usporediti algoritam optimizacije rojem čestica s metodom propagacije unatrag. Otkrili su da je algoritam optimizacije rojem čestica jednako dobar, a u nekim slučajevima i bolji od metode propagacije unatrag. Uz neuronske mreže, algoritam su testirali i na Schaferovoj F6 funkciji, vidljivu na slikama 3 i 4, koja se često koristi za testiranje genetskih algoritama. Zbog velike količine lokalnih minimuma (koncentrične kružnice oko ishodišta), algoritmi optimizacije znaju često zapeti. Algoritam optimizacije rojem čestica je svaki put uspio naći dobar rezultat [6].



Slika 3:3D plot Schaferove f6 funkcije



Slika 4:Konturni graf Schaferove f6 funkcije. Na grafu je crveno označen optimum funkcije, točka s koordinatama (0, 0) i vrijednosti funkcije 0

4. Poboljšanja algoritma

Originalni algoritam je bio dobar, no imao je neke probleme. Najveći problem je bio to što čestice uglavnom nisu konvergirale (skupljale) na jednu točku te s tim nisu poboljšavale nađeno dobro rješenje, nego bi je oblijetale. Uz to, algoritam je često znao zapeti u neki lokalni optimum. Kako bi riješili problem konvergencije, znanstvenici uvode ograničenje brzine i težinske vrijednosti koje usporavaju vektor kretanja te podešavaju konstante c_1 i c_2, a problem lokalnih minimuma pokušavaju rješiti tako da istražuju utjecaj topologije čestica na rezultate algoritma.

4.1. Ograničenje brzine

Kako se čestice ne bi razletjele po prostoru pretrage, potrebno je na neki način ograničiti prvu komponentu novog vektora kretanja, trenutni vektor kretanja. Brzina se može ograničiti na nekoliko načina: jednostavno ograničenje brzine, težinska vrijednost inercije, faktor konstrikcije, ili neka kombinacija navedenih.

4.1.1. Maksimalna brzina

Kennedy i Eberhart [9] uvode jednostavno ograničenje brzine, kod kojeg vektor u nekoj dimenziji ne može prijeći preko određene brzine. Russell Eberhart i Yuhui Shi predlažu da se konzistentno najbolji rezultati dobiju kada se najveća brzina u nekoj dimenziji V_{max} postavi kao ograničenje prostora pretrage X_{max} (i V_{min} kao $-X_{max}$, u suprotnom smjeru) [10], pa se vektor brzine u nekoj dimenziji računa na sljedeći način:

$$v_{id}^{k+1'} = v_{id}^{k} + c_1 \cdot rand() \cdot \left(p_{id}^{k} - X_{id}^{k}\right) + c_2 \cdot rand() \cdot \left(p_{gd}^{k} - X_{id}^{k}\right)$$

$$v_{id}^{k+1} = \min\left(\max\left(v_{id}^{k+1'}, V_{\min}\right), V_{max}\right)$$

4.1.2. Inercija i težinska vrijednost inercije

Eberhart i Shi u istom radu za ograničenje originalnog vektora kretanja uvode novi parametar u funkciju, koji nazivaju težinska vrijednost inercije (eng. *Inertia weight*).

$$v_{id}^{k+1} = \omega \cdot v_{id}^{k} + c_1 \cdot rand() \cdot \left(p_{id}^{k} - X_{id}^{k}\right) + c_2 \cdot rand() \cdot \left(p_{gd}^{k} - X_{id}^{k}\right)$$

Ova formula razlikuje se od originala tako da je parametar $p_{increment}$ preimenovan u c_1, parametar $g_{increment}$ je preimenovan u c_2, parametri c_1 i c_2 nisu fiksirani na vrijednost 2 te je prijašnji vektor kretanja (vektor inercije) pomnožen s novim parametrom težinska vrijednost

inercije (ω). Visoka vrijednost parametra ω potiče česticu na pretragu šireg područja rješenja, dok niska vrijednost parametra ω potiče detaljniju lokalnu pretragu. Autori navode da dobro odabrani parametar ω smanjuje broj iteracija potreban za pronalaženje rješenja. Dodavanjem parametra ω sve čestice se, kroz vrijeme, približavaju rješenju, umjesto da ga oblijetaju. Ova verzija algoritma se u literaturi često naziva standardnim algoritmom optimizacije čestica [10].

4.1.2.1. Promjena težinske vrijednosti inercije

Ako u algoritmu želimo, kroz izvršenje algoritma, mijenjati težinsku vrijednost inercije, postoji nekoliko načina kako ju možemo mijenjati može mijenjati.

Kod linearnog smanjenja kroz vrijeme (eng. *Linear Time Varying*, ili LTV), vrijednost parametra ω smanjuje se nakon svake iteracije, linearno. Formula za izračun je:

$$\omega(t) = \omega_{min} + (\omega_{max} - \omega_{min}) \cdot (1 - \frac{t}{T})$$

Gdje su ω_{max} početna i najviša vrijednost parametra ω, ω_{min} završna i najmanja vrijednost parametra ω, t je broj trenutne iteracije, a T ukupni broj iteracija algoritma. Parametar ω_{min} se uglavnom uzima kao 0,4, a ω_{max} kao 0,9. Takvo smanjenje prvi predlaže Ponnuthurai Suganthan [11].

Kod nelinearnog smanjenja kroz vrijeme (eng. *Nonlinear Time Varying*, ili NLTV), formula za izračun parametra ω je nelinearna, takva da parametar ω brže opada te da algoritam može provesti više vremena u lokalnoj pretrazi, umjesto globalne [12].

4.1.3. Faktor konstrikcije

Kennedy i Maurice Clerc definiraju faktor konstrikcije χ, kojim se množi cijeli novi vektor kretanja, ne samo originalni vektor kretanja:

$$v_{id}^{k+1} = \chi \cdot \left(v_{id}^{k} + c_1 \cdot rand() \cdot \left(p_{id}^{k} - X_{id}^{k}\right) + c_2 \cdot rand() \cdot \left(p_{gd}^{k} - X_{id}^{k}\right)\right)$$

Sam faktor konstrikcije se računa na sljedeći način:

$$\chi = \frac{2\kappa}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|}$$

Gdje je ϕ zbroj parametara c_1 i c_2, a κ neki broj između 0 i 1. Kada je vrijednost κ bliža 1, tada roj čestica radi temeljitu globalnu pretragu prije nego konvergiraju na neku točku. Kako bi roj čestica konvergirao na jednu točku, potrebno je odabrati parametre c_1 i c_2 takve da $\phi > 4$. Autori navode kako definiranje maksimalne brzine V_{max} nije eksplicitno potrebno, no da bi mogao blago poboljšati performanse algoritma [13].

4.2. Topologija čestica

Umjesto da sve čestice za izračun sljedećeg pomaka koriste globalno pronađeno najbolje rješenje, neki autori predlažu da se čestice povežu u susjedstva (eng. *neighborhoods*) te da svako susjedstvo čestica računa prema lokalno nađenom najboljem rješenju.

U kontekstu algoritama optimizacije rojem čestica, topologija predstavlja stupanj povezanosti između čestica u roju [12]. Kennedy i Eberhart predlažu dvije topologije. U prvoj topologiji sve čestice računaju svoje pokrete prema osobnom najboljem rezultatu (p_{best}) i prema globalno poznatom najboljem rezultatu (g_{best}). U drugoj topologiji, čestice prate osobni najbolji rezultat, no umjesto jednog globalno poznatog rješenja, čestice su grupirane u susjedstva te svako susjedstvo pamti najbolji rezultat susjedstva (l_{best}). Kod druge, lokalne topologije, sve čestice ne konvergiraju na jedno rješenje, nego se skupljaju na više rješenja, što daje veću vjerojatnost da je jedno od tih rješenja optimalno. Čestice se grupiraju tako da i-ta čestica uspoređuje svoje vrijednosti sa česticom $i-1$ i česticom $i+1$ (ako je veličina susjedstva 2) [9]. Kennedy u kasnijem radu topologiju s jednim, globalnim rješenjem naziva zvjezdastom topologijom (slika 5, gore lijevo, primjer s 12 čestica), jer su sve čestice međusobno povezane, dok topologiju s lokalnim rješenjima naziva kružnom topologijom (slika 5, gore desno, primjer s 12 čestica i veličinom susjedstva 2) [14].

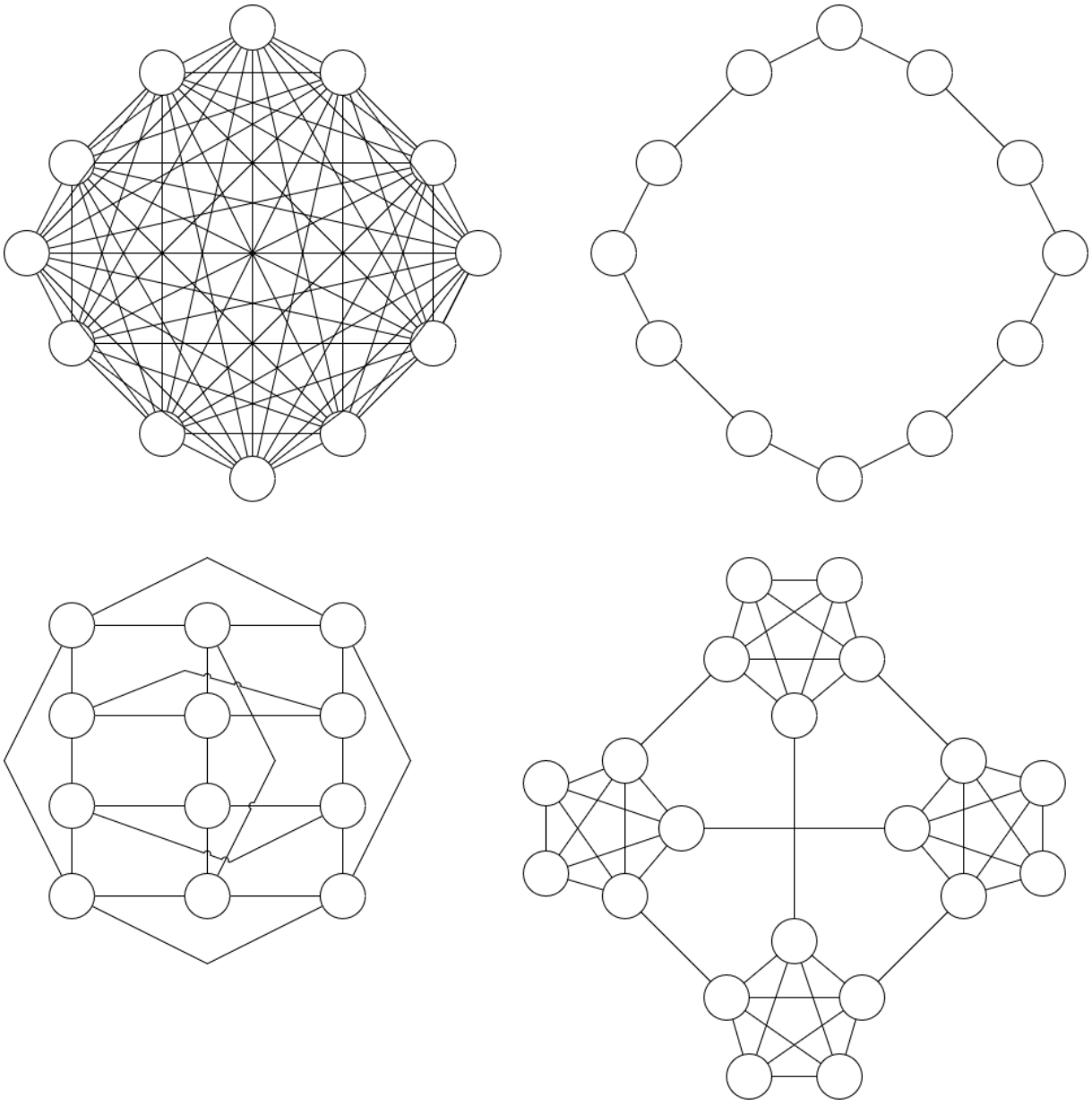
U istom radu iznosi još dvije topologije: kotač, gdje je jedna čestica povezana sa svim česticama, a ostale čestice samo s tom jednom (slika 5, u sredini lijevo, 12 čestica), i nasumični rubovi, gdje se za N čestica nasumično povlači N simetričnih veza (slika 5, u sredini desno, 12 čestica i 12 nasumičnih veza). U kasnijim radovima predstavlja von Neumann topologiju, gdje su čestice postavljene u 2D rešetku te je svaka čestica spojena sa susjedom iznad, ispod, lijevo i desno (slika 5, dolje lijevo, 12 čestica), piramidalna topologija, gdje su čestice složene u „trodimenzionalni žičani trokut“ (eng. *three-dimensional wire-frame pyramid*) [15], i četiri grupe, gdje su čestice podijeljene u četiri grupe, koje su međusobno povezane s par veza, dok su čestice unutar grupe potpuno povezane (slika 5, dolje desno, primjer s 20 čestica, gdje je u svakoj grupi 5 čestica) [16].

4.2.1. Usporedba topologija

Rui Mendes, James Kennedy i José Neves usporedili su performanse nekih od navedenih topologija, koje su prikazane na slici 5:

- Zvjezdastu topologiju (gbest, gore lijevo)
- Kružnu topologiju (lbest, gore desno)
- Piramidalnu topologiju

- von Neumann topologiju (dolje, lijevo)
- Četiri grupe (dolje, desno)



Slika 5: Vizualizacija topologija rojeva čestica

Za sve topologije, osim četiri grupe, dodatno razlikuju verzije gdje se čestica ubraja u susjedstvo (prefiks *s*, npr. *slbest*) od verzija gdje se ne ubrajaju, kod provjere lokalnog najboljeg rješenja. Testirali su tri vrste algoritma: kanonsku verziju (koju su skraćeno nazvali PSO), verzija kod koje na vektor brzine imaju utjecaj osobna najbolja rješenja svih čestica u susjedstvu podjednako (koji su nazvali PSON), i verziju PSON kod koje bolji rezultati čestica imaju veći utjecaj na vektor brzine (PSOWN).

Kod PSO algoritma, vektor brzine su računali na sljedeći način:

$$v_{t+1} = \chi\big(v_t + \phi \cdot (P_m - X_t)\big)$$

gdje je parametar χ postavljen na 0,729844, a P_m je točka konvergencije, koja se računa na sljedeći način:

$$P_m = \frac{\phi_1 P_i + \phi_2 P_b}{\phi}$$

Gdje su P_i osobno najbolje rješenje čestice, P_b najbolje rješenje susjedstva, a ϕ_1 i ϕ_2 parametri takvi da vrijedi:

$$\phi_1 = rand(0 \ldots \frac{\phi_{max}}{2})$$

$$\phi_2 = rand(0 \ldots \frac{\phi_{max}}{2})$$

$$\phi = \phi_1 + \phi_2$$

Parametar ϕ_{max} je postavljen na 4,1.

Kod verzije PSON vrijednosti ϕ se računaju drukčije. Definiramo $\mathcal{N}$ kao skup čestica povezanih sa česticom i. Tada se P_m računa kao

$$P_m = \frac{\sum_{k \in \mathcal{N}} \phi_k \cdot P_k}{\phi}$$

Gdje su P_k osobni najbolji rezultat čestice K iz skupa $\mathcal{N}$, a ϕ_k se računa kao

$$\phi_k = rand\left(0, \frac{\phi_{max}}{|\mathcal{N}|}\right), \forall k \in \mathcal{N}$$

$$\phi = \sum_{k \in \mathcal{N}} \phi_k$$

Kod verzije PSOWN, P_m se računa kao

$$P_m = \frac{\sum_{k \in \mathcal{N}} \frac{\phi_k \cdot P_k}{f_k}}{\sum_{k \in \mathcal{N}} \frac{\phi_k}{f_k}}$$

Gdje je $f_k$vrijednost funkcije koju optimiziramo, u točki P_k.

Mendes i sur. su testirali navedene algoritme na sljedećim funkcijama, koje se često koriste za testiranje algoritama optimizacije rojem čestica i genetske algoritme:

- 30-dimenzionalna sfera, s minimumom $f_1(0) = 0$. Domena pretrage je ograničena na ± 100, a prihvatljivo odstupanje je postavljeno na 0,01.

$$f_1(x) = \sum_{i=1}^{n} x_i^2$$

- Rastrigin funkcija u 30 dimenzija, s minimumom $f_2(0) = 0$ i puno lokalnih minimuma Domena pretrage je ograničena na $\pm 5{,}12$, a prihvatljivo odstupanje je postavljeno na 100.

$$f_2(x) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 10$$

- Griewank funkcija, u 10 i 30 dimenzija, izrazito teška funkcija s minimumom $f_3(0) = 0$. Domena pretrage je ograničena na ± 600, a prihvatljivo odstupanje je postavljeno na 0,05.

$$f_3(x) = \sum_{i=1}^{n} \frac{(x_i - 100)^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$$

- Rosenbrock funkcija, u 30 dimenzija. Domena pretrage je ograničena na ± 30, a prihvatljivo odstupanje je postavljeno na 100.

$$f_4 = \sum_{i=1}^{n-1} 100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2$$

- Schaferova f6 funkcija, dvodimenzionalna funkcija dizajnirana da prevari optimizacijske algoritme sa koncentričnim prstenima lokalnih minimuma. Domena pretrage je ograničena na ± 100, a prihvatljivo odstupanje je postavljeno na 0,00001.

$$\mathrm{f}_5(x, y) = 0.5 + \frac{\sin\left(\sqrt{x^2 + y^2}\right)^2 - 0.5}{\left(1 + 0.001(x^2 + y^2)\right)^2}$$

Njihovi rezultati vidljivi su u tablici 1. Kombinacije algoritama i topologija rangirali su prema uspješnosti (postotak izvršavanja koji su došli unutar prihvatljivog odstupanja), prema medijanu broja iteracija koje su bile potrebne da se dođe do prihvatljivog odstupanja te prosječnoj vrijednosti algoritma nakon 1000 iteracija.

Rezultati istraživanja su pokazali da topologije *lbest* i von Neumann topologije konzistentno daju najbolje rezultate te iako je kombinacija *lbest*-PSOWN daje najbolje rezultate, ta kombinacija je dosta spora (22. kombinacija po redu po performansama nakon 1000 iteracija). No, vidimo da kombinacija von Neumann-PSON daje jako dobre rezultate, uz najbolje performanse. Autori zaključuju da rezultati koje algoritam PSOWN daje često nisu bolji od algoritma PSON, pogotovo što se tiče performansi [16].

Tablica 1: Prikaz rezultata usporedbe različitih algoritama i topologija

Algoritam	Topologija	Uspješnost	Uspješnost - rank	Broj iteracija - rank	Performanse - rank
PSOWN	lbest	100,00%	**1**	19	22
PSON	vneumm	98,75%	2	4	**1**
PSON	slbest	98,75%	2	11	14
PSOWN	slbest	98,33%	4	5	6
PSON	lbest	96,67%	5	22	23
PSOWN	vneumm	96,25%	6	3	2
PSO	vneumm	92,50%	7	10	10
PSO	četiri grupe	92,08%	8	9	15
PSON	pyramid	91,67%	9	6	5
PSOWN	četiri grupe	91,67%	9	2	9
PSON	četiri grupe	91,25%	11	7	7
PSO	slbest	91,25%	11	13	20
PSO	lbest	90,89%	13	12	12
PSOWN	pyramid	90,00%	14	**1**	8
PSO	svneumm	87,50%	15	18	16
PSO	pyramid	87,50%	16	17	13
PSOWN	svneumm	86,67%	17	8	3
PSO	pyramid	85,42%	18	16	17
PSON	svneumm	81,25%	19	20	4
PSOWN	spyramid	78,33%	20	21	11
PSO	gbest	75,42%	21	15	19
PSO	sgbest	75,42%	21	14	21
PSON	spyramid	70,42%	23	23	18
PSOWN	gbest	22,08%	24	25	25
PSOWN	sgbest	18,33%	25	26	26
PSON	gbest	16,67%	26	24	27
PSON	sgbest	13,75%	27	27	24

(Izvor: [16])

4.3. Kognitivni i socijalni koeficijenti

Parametri originalnog algoritma optimizacije rojem čestica c_1 i c_2 se u literaturi redom često nazivaju kognitivni koeficijent (jer utječe na pomak prema točki koja je čestici osobno najbolja, p_{best}) i socijalni koeficijent (jer utječe na pomak prema globalno najboljoj točki g_{best}, ili prema najboljoj točki susjedstva l_{best}). Ako je kognitivni koeficijent postavljen na 0, čestica zanemaruje svoje znanje te se orijentira samo prema znanju ostalih čestica. Ako je socijalni koeficijent, čestica ignorira kolektivno znanje te prati samo svoje rješenje. Sengupta i sur. navode da su idealne vrijednosti ovih parametara dobivene empirički te da iznose 2,05 [12].

5. Hibridni algoritmi

Kako bi dalje poboljšali performanse algoritma optimizacije rojem čestica, neki autori su algoritam kombinirali drugim algoritmima optimizacije. Neki od tih algoritama su genetski algoritmi, diferencijalna evolucija i simulirano prekaljivanje … [12].

5.1. Genetski algoritam

Genetski algoritam je heuristički algoritam optimizacije koji koristi proces prirodne selekcije, kako bi došao do rješenja. Za dobivanje optimalnog rješenja koristi korake križanja (eng. *crossover*), mutacije i selekcije. Na početku algoritma se nasumično inicijaliziraju jedinke, a njihovi parametri se definiraju u tzv. kromosome, često u binarnom obliku. U svakoj iteraciji, populacija jedinki se evaluira prema funkciji koju želimo optimizirati. Nakon što evaluiramo jedinke, generira se novi set jedinki. Selektiraju se najbolje jedinke (mogu se odabrati najbolje jedinke direktno, metodom mini turnira, u kojoj se nasumično odaberu dvije jedinke pa se uzima bolja od tih dviju, ili se može koristiti tzv. metoda ruleta, u kojoj se bilo koja jedinka može odabrati, no vjerojatnost odabira jedinke je veća što je bolja evaluacijska vrijednost jedinke) te se iz njih, križanjem, generiraju nove jedinke. Svaki kromosom nekih dviju jedinke se podijeli na istoj poziciji u obje jedinke te se prvi dio kromosoma prve jedinke spaja s drugim dijelom kromosoma druge jedinke, i obrnuto. Selekcija i križanje se ponavljaju dok jedinki u novoj generaciji postane jednak broju jedinki u prošloj generaciji. Kako se najbolja rješenja prošle generacije ne bi izgubila, najboljih n jedinki iz prošle generacije se jednostavno kopira u novu generaciju (tzv. elitizam). Nakon što se generiraju jedinke, nad nasumičnim brojem jedinki se u nasumičnim kromosomima mijenja nasumični bit kromosoma, iz 1 u 0 ili iz 0 u 1, kako bi se u populaciju umiješao novi genetski materijal. Taj proces se naziva mutacijom. Navedeni koraci se ponavljaju odabrani broj generacija te se često dobije jako dobro rješenje problema [17]. Algoritam optimizacije rojem čestica se kombinira s genetskim algoritmima tako da se jedan algoritam provodi nakon što drugi završi, ili da se oba algoritma koriste paralelno.

Bo Yang i sur. predstavljaju „hibridni evolucijski algoritam“ (eng. *Hybrid evolutionary algorithm*, ili HEA), u kojoj se algoritmi provode u fazama. U prvoj fazi koristi se algoritam optimizacije rojem čestica, kako bi se dobilo dobro rješenje problema. Nakon toga se koristi genetski algoritam kako bi se rješenje dalje poboljšalo te kako bi se čestice iščupale iz potencijalnih lokalnih minimuma. U radu demonstriraju kako je njihov hibridni algoritam daje bolje rezultate od samog algoritma optimizacije rojem čestica i od samog genetskog algoritma [18].

5.2. Diferencijalna evolucija

Algoritam diferencijalne evolucije je metaheuristički algoritam. Na početku algoritma definiramo funkciju koju optimiziramo, dimenziju funkcije D, faktor mutacije $F \in [0,2]$, konstantu križanja $CR \in [0,1]$, broj generacija G i N nasumično odabranih D-dimenzionalnih vektora $x_1, x_2, \dots, x_N$, koji postaju početna populacija. Na početku g-te generacije provodimo mutaciju tako da za svaki vektor $x_{i,g}$ (target vektor) definiramo mutant vektor $v_{i,g+1}$ na sljedeći način

$$v_{i,g+1} = x_{r1,g} + F \cdot (x_{r2,g} - x_{r3,g})$$

Gdje su $r_1, r_2, r_3 \in \{1,2, \dots, N\}$ nasumično odabrani indeksi populacije vektora, takvi da su svi indeksi međusobno različiti te da su različiti od indeksa trenutnog vektora x_i.

Nakon što smo generirali mutant vektore, iz svakog para target vektora i mutant vektora definiramo trial vektor $u_{i,g+1}$ na sljedeći način:

$$u_{i,g+1} = (u_{1i,g+1}, u_{2i,g+1}, \dots, u_{Di,g+1})$$

Gdje su:

$$u_{ji,g+1} = \begin{cases} v_{ji,g+1}, ako\ (randb(j) \leq CR\ ili\ j = rnbr(i)) \\ x_{ji,g+1}, ako (randb(j) > CR)\ i\ j \neq rnbr(i) \end{cases}$$

U ovoj funkciji, $randb(j)$ je j-ti nasumični broj između 0 i 1, a $rnbr(i)$ je nasumično odabrani indeks, kako bi barem jedan element iz mutant vektora bio prenesen u trial vektor. Ukratko, uzimamo elemente iz target i mutant vektora, nasumično prema konstanti križanja CR te kako bi barem jedan element iz mutant vektora bio odabran, nasumično odabiremo jedan indeks koji će se prenijeti iz mutant vektora.

Nakon mutacije se provodi selekcija. Evaluiraju i uspoređuju se target vektor $x_{i,g}$ i trial vektor $u_{i+1,g}$. Ako je trial vektor bolji od target vektora, trial vektor prelazi u sljedeću generaciju i postaje $x_{i,g+1}$. Ako nije, onda originalni target vektor prelazi u sljedeću generaciju i postaje $x_{i,g+1}$. Koraci mutacije i selekcije se ponavljaju G generacija. Ovdje opisani algoritam je prvo opisan od strane Rainera Storna i Kennetha Pricea te je posebnom notacijom označen kao DE/rand/1/bin, gdje DE označuje da se radi o algoritmu diferencijalne evolucije, *rand* označuje da se target vektor birao nasumično (autori još navode *best* metodu, gdje je target vektor najbolji vektor populacije), 1 označuje da se radi o jednom vektoru diferencije $(x_{r2,g} - x_{r3,g})$, a *bin* metoda križanja, ovdje križanje prema „nezavisnim binomialnim eksperimentima“ (eng. *independent binomial experiments*) [19].

Tim Hendtlass u svom radu kombinira algoritam optimizacije rojem čestica i algoritam diferencijalne evolucije, koji naziva SDEA (*swarm differencial evolution algorithm*). SDEA

uglavnom koristi algoritam optimizacije rojem čestica, no definira parametar T (idealno između 0,25 i 0,5), koji predstavlja vjerojatnost da će se u nekoj iteraciji, umjesto algoritma optimizacije rojem čestica, iskoristiti algoritam diferencijalne evolucije. U njihovim testovima, algoritam SDEA često daje bolje rješenje, no u prosjeku mu je trebalo više iteracija da dođe do prihvatljivog rješenja [20].

5.3. Simulirano prekaljivanje

Scott Kirpatrick i sur. definiraju algoritam simuliranog prekaljivanja (eng. *simulated annealing*), koji simulira prekaljivanje metala, čiji atomi se kreću jako kaotično i nasumično dok je temperatura visoka te se kroz vrijeme i postepeni pad temperature preslažu u stabilni i jaki raspored. U algoritmu se definira funkcija koju želimo optimizirati, parametar početne temperature T, broj iteracija I. Na početku, nasumično generiramo neko rješenje te ga evaluiramo na funkciji. Nakon toga se iz početnog rješenja generira novo rješenje tako da mu se komponente rješenja nasumično malo promjene. To novo rješenje se evaluira na funkciji. Računamo razliku između vrijednosti novog rješenja (x_{new}) i starog rješenja (x), ΔE:

$$\Delta E = f(x_{new}) - f(x)$$

Ako je ΔE negativni broj, znači da je novo rješenje bolje od starog te novo rješenje postaje trenutno rješenje problema. Ako je ΔE pozitivni broj, znači da je novo rješenje gore od staroga, no ne odbacujemo ga odmah, nego računamo vjerojatnost da ćemo novo rješenje ipak zadržati, prema formuli:

$$P(\Delta E) = e^{\frac{-\Delta E}{t}}$$

Gdje je t trenutna temperatura, a i broj trenutne iteracije:

$$t = T \cdot \left(1 - \frac{i}{I}\right)$$

U ovom slučaju, temperatura se linearno spušta do 0. Evaluiramo:

$$P(\Delta E) > rand(0,1)$$

Ako gornja formula vrijedi, onda zadržavamo novo rješenje, iako je gore od starog. Možemo vidjeti da, što je temperatura viša, to će eksponent biti veći, a vjerojatnost da ćemo prihvatiti rješenje veća. Također, što je razlika između rješenja veća, to je eksponent manji, i vjerojatnost da ćemo prihvatiti rješenje manja. Algoritam ponavljamo I iteracija [21].

Guangoyu Yang i sur. kombiniraju algoritam simuliranog prekaljivanja i algoritam optimizacije rojem čestica, kako bi pomogli algoritmu optimizacije rojem čestica da izađe iz lokalnih rješenja. Taj algoritam zovu PSOSA. Na početku se postave potrebni parametri, broj

čestica m, kognitivni i socijalni koeficijenti c_1 i c_2, maksimalni broj iteracija $maxIter$, koliko puta provodimo simulirano prekaljivanje po generaciji $maxN_1$ i $maxN_2$. Nasumično se generira m čestica te se one evaluiraju. Određuju se najbolja čestica početnog roja g_{best}, no određuje se i vrijednost najgore čestice početnog roja $f(g_{worst})$. Početna temperatura T za simulirano prekaljivanje se izračunava s formulom

$$T = \frac{-\left(f(g_{worst}) - f(g_{best})\right)}{\ln P_r}$$

Gdje je P_r mala konstanta. Trenutna temperatura t se postavlja na početnu temperaturu T. Provode se iteracije algoritma optimizacije rojem čestica, uz težinsku vrijednost inercije ω koja se smanjuje linearno te se koristi ograničenje brzine V_{max}. Ako nakon neke iteracije ne dođe do poboljšanja globalnog rješenja. Nad rojem se $maxN_1$ puta provodi sljedeće:

Nad svakom česticom roja do maxN_2 puta nad osobnim najboljim rješenjem j-te čestice generiramo novo rješenje uz pomoć formule:

$$new\ p_{best,d} = p_{best,d} \cdot (1 + \eta \cdot \sigma)$$

Gdje su d oznaka dimenzije za koju računamo novu vrijednost, σ nasumični broj generiran Gaussovom raspodjelom uz srednju vrijednost 0 i standardnu devijaciju 1, a η broj koji se dobiva kao:

$$\eta = \eta \cdot (1 - interdec)$$

Gdje je $interdec$ neka mala konstanta. Dok dobijemo novo rješenje p_{best}, evaluiramo ga. Ako je ono bolje od osobnog najboljeg rješenja čestice, novo rješenje postaje osobno najbolje rješenje čestice te prestajemo iterirati nad česticom. Ako nije, evaluiramo:

$$\min\left\{1, e^{\frac{-\left(f(p_{best,j}) - f(g_{best})\right)}{t}}\right\} > rand(0,1)$$

Ako izraz vrijedi, novo rješenje postaje trenutno rješenje čestice j te prestajemo iterirati nad česticom. Nakon što smo iterirali nad svim česticama, smanjimo temperaturu (u radu temperaturu modificiraju vrijednošću $TempRatio$, no nigdje u radu ne navode koliko taj $TempRatio$ iznosi). Nakon što smo gornji postupak proveli $maxN_1$, nastavljamo s algoritmom optimizacije rojem čestica te postupak ponavljamo svaki puta dok ne dođe do poboljšanja rješenja. Cijeli algoritam staje dok dođemo do dovoljno dobrog rješenja, a najviše $maxIter$ iteracija.

Autori algoritma navode da, iako pojedine iteracije algoritma u prosjeku dulje traju, u usporedbi sa standardnim algoritmom optimizacije rojem čestica, algoritam PSOSA nalazi jednako dobro rješenje u puno manje iteracija te puno bolje rješenje nakon jednakog broja

iteracija. Algoritam su usporedili i sa standardnim genetskim algoritmom te su ustanovili da algoritam PSOSA daje bolja rješenja i od genetskog algoritma [22].

6. Algoritam rojeva čestica bez parametara, TRIBES

Kako bi se eliminirala potreba za postavljanjem parametara (čije „najbolje vrijednosti“ često nisu matematični dokazane, nego empirijski „pogođene“ te čije „najbolje vrijednosti“ nisu iste za sve probleme), Maurice Clerc predstavlja algoritam TRIBES, adaptivni algoritam roja čestica koji počinje sa jednom česticom te ih generira kada je potrebno i briše one koji nisu potrebne. Algoritam počinje s jednom, nasumično generiranom česticom. Tijekom izvođenja algoritma, dok se dodaju nove čestice, one se stvaraju u grupama, koje se ovdje nazivaju „plemenima“ (fr. *tribes*), otkud i dolazi naziv algoritma. Autor ih naziva tako jer povezuje grupu čestica koje se kreću u prostoru rješenja s grupom pojedinaca koji se, u nepoznatom okruženju, kreću skupa prema najboljem području. Unutar plemena, čestice su povezane potpuno, dok su plemena povezana s malo veza, (često jednom), dovoljno da plemena razmjenjuju informacije.

Čestice pamte dvije iteracije unatrag jesu li poboljšale rješenje tu iteraciju. Ako je čestica u prošloj iteraciji poboljšala svoje rješenje, onda se čestica u novoj iteraciji smatra „dobrom“, a ako nije, smatra se „neutralnom“. Ako je čestica unatrag dvije iteracije poboljšala svoje rješenje, onda se ona smatra „odličnom“. Unutar plemena se određuju čestica koja ima najbolje rješenje, i ona koja ima najgore rješenje.

Svakom plemenu se prebrojava broj dobrih čestica (B) od ukupnih čestica (T). Generira se nasumični broj između 0 i T (p). Ako je $B > p$, pleme se smatra dobrim, ako je $B \leq p$, pleme se smatra lošim.

Brišu se čestice, kako bi se broj evaluacija funkcije optimizacije smanjio te kako bi se algoritam brže izvršavao. Čestice se brišu samo iz dobrih plemena, jer se pretpostavlja da brisanje najgore čestice dobrog plemena neće utjecati na rezultat, jer druge čestice plemena već imaju bolji rezultat. Ako je ta čestica bila povezana s drugim plemenom, ta veza se prenosi na drugu česticu plemena, uglavnom na najbolju česticu plemena. Ako dobro pleme ima samo jednu česticu, čestica se briše samo ako ima gore rješenje od neke čestice drugih plemena s kojim je povezana (povezana čestica naziva se informantom čestice). Nestajanjem čestice nestaje i pleme.

Sva loša plemena generiraju po dvije nove čestice. Svaka čestica je ili slobodna ili ograničena. Slobodne čestice se generiraju nasumično prema jednom od sljedećih pravila:

- Nasumično u cijelom prostoru pretrage
- Nasumično, na stranici prostora pretrage
- Nasumično, na bridu prostora pretrage

Ograničene čestice se generiraju prema lokaciji najbolje čestice (x) plemena koji ju stvara, i prema lokaciji najbolje povezane čestice (g) čestice x. Čestica se stvara nasumično u D-sferi sa središtem u točki g radijusa $\|g - x\|$. Sve novogenerirane čestice se grupiraju u novo pleme, a svaka čestica novog plemena je povezana s plemenom ih kojeg je generirana, uglavnom s najboljom česticom tog plemena.

Brisanje i generiranje čestica (adaptacija roja) ne događa se nakon svake iteracije. Neka je L ukupni broj poveznica između čestica, koji se ažurira nakon svake adaptacije roja. Sljedeća adaptacija roja se događa nakon $\frac{L}{2}$ iteracija. Na početku, broj poveznica je 1 (čestica „komunicira“ sama sa sobom), pa se nakon prve iteracije provodi generiranje novog roja. U drugoj iteraciji postoje tri čestice, sve međusobno povezane i povezane same sobom, pa je L=6, što bi značilo da se nove čestice generiraju i brišu, ako treba, za tri iteracije.

Kao što je prije navedeno, čestice pamte unazad dvije iteracije svoje promjene, to jest pamte jesu li u tim iteracijama poboljšale svoje rješenje (+), pogoršale svoje rješenje (-) ili nisu promijenile svoje rješenje (=). Prema tim podacima, određuju strategiju kretanja. Clerc navodi tri strategije kretanja: pivot, smeteni pivot i oko najbolje lokalne čestice, Gaussovom distribucijom. Kada se koja strategija primjenjuje, vidljivo je u tablici 2.

Tablica 2: Prikaz strategija i memorije čestice

Memorija kretanja čestice	Strategija kretanja
(-,-), (=,-), (+,-), (-,=), (=,=)	Pivot
(+,=), (-,+)	Pivot uz buku
(=,+), (+,+)	Lokalno uz Gaussovu distribuciju

(Izvor:[23])

Kod strategije pivot uzima se udaljenost r između točke najboljeg rješenja čestice p i točke najboljeg rješenja plemena, l. Oko tih točaka se definiraju hipersfere radijusa r, H_l i H_p. Nove koordinate točke se računaju na sljedeći način:

$$x = c_1 \cdot rand(H_l) + c_2 \cdot rand(H_p)$$

Ovdje, $rand(H)$ znači da se iz tih hipersfera odabere neka nasumična točka. Kako je ovo algoritam bez početnih parametara, c_1 i c_2 se moraju izračunati:

$$c_1 = \frac{f(p)}{f(p) + f(l)}, c_2 = \frac{f(l)}{f(p) + f(l)}$$

Kod pivota uz buku se svakoj koordinati novog rješenja dodaje šum, nasumični broj u Gaussovoj distribuciji, uz prosjek 0 i buku koja se dobije iz trenutnih rješenja točki p i l:

$$x = x \cdot \left(1 + randGauss\left(0, \frac{f(p) - f(l)}{f(p) + f(l)}\right)\right)$$

Kod strategije lokalno uz Gaussovu distribuciju računamo točku po Gaussovoj distribuciji, koordinate čestice postavljamo na:

$$x = l + randGauss(l - x, \|l - x\|)$$

Traženje najboljeg informanta možemo raditi direktnom usporedbom: čestica z je bolji informant čestici x od čestice y ako je evaluacija funkcije nad najboljim rezultatom čestice z ($\hat{z}$) bolja od evaluacije funkcije nad česticom y ($\hat{y}$):

$$z \succ_x y \Leftrightarrow f(\hat{z}) < f(\hat{y})$$

Ako je prostor pretrage metrički prostor, čestice možemo rangirati prema „pseudo-gradijentu“, kako bi prednost imale čestice koje su bliže čestici x:

$$z \succ_x y \Leftrightarrow \frac{f(\hat{x}) - f(\hat{z})}{\|\hat{x} - \hat{z}\|} > \frac{f(\hat{x}) - f(\hat{y})}{\|\hat{x} - \hat{y}\|}$$

Metrički prostor se definira ovako: Neka je M skup, a d realna funkcija za koju vrijedi

$$d: M \times M \to \mathbb{R}$$

Metrički prostor je uređeni par (M, d) ako za svaki $x, y, z \in M$ vrijedi:

$$1)\ d(x, y) = 0 \Leftrightarrow x = y$$

$$2)\ d(x, y) = d(y, x)$$

$$3)\ d(x, z) \leq d(x, y) + d(y, z)$$

Clerc navodi da je TRIBES algoritam izrazito dobar, pogotovo ako je moguće koristiti „pseudo-gradijente“. Kroz knjigu Clerc predstavlja i testira više verzija algoritma PSO te dolazi do zaključka da je TRIBES algoritam bolji od svih drugih navedenih te da je bolji i drugih algoritama koji se smatraju dobrim, kao što je genetički algoritam ili diferencijalna evolucija [23].

7. Implementacija algoritama

U ovom dijelu rada implementirao sam nekoliko vrsta algoritama koje sam kroz rad opisao te sam ih usporedio na nekoliko problema. Algoritme sam implementirao u programskom jeziku Python. Algoritmi koje sam implementirao i usporedio su:

- Originalni algoritam optimizacije rojem čestica (PSO)
- Algoritam optimizacije rojem čestica s težinskom vrijednosti inercije, tzv. standardni PSO (SPSO)
- Algoritam PSO kod kojeg koje na vektor brzine imaju utjecaj osobna najbolja rješenja svih čestica u susjedstvu (PSON)
- Algoritam PSO kombiniran s genetskim algoritmom (HEA)
- Algoritam PSO kombiniran s diferencijalnom evolucijom (SDEA)
- Algoritam PSO kombiniran sa simuliranim prekaljivanjem (PSOSA)
- Algoritam PSO bez parametara (TRIBES)

7.1. PSO

Početak klase OriginalPSO i njenog konstruktora izgledaju ovako:

```
from copy import deepcopy
import numpy as np
import time
class OriginalPSO:
    def __init__(self, function, particleCount, dimensions,
minIterationCount, maxIterationCount, c1=2, c2=2, topology="gbest"):
        self.function=function
        self.particleCount=particleCount
        self.dimensions=dimensions
        self.xMax=self.function.getXMax()
        self.c1=c1
        self.c2=c2
        self.iterationCount=0
        self.minIterations=minIterationCount
        self.maxIterations=maxIterationCount

        self.timeAtAcceptableResult=None
        self.iterationAtAcceptableResult=None
        self.functionExecutionAtAcceptableResult=None
```

Originalnom algoritmu optimizacije rojem čestica se u konstruktoru (funkciji `__init__`) predaje objekt funkcije, u kojem se nalazi funkcija koju treba optimizirati te neke pomoćne

informacije, kao što je prihvatljiva greška funkcije, lokacija optimuma funkcije te širina pretrage u pojedinoj dimenziji. Osim funkcije, algoritmu se predaje broj dimenzija funkcije, broj čestica, minimalni i maksimalni broj iteracija, vrijednosti parametara c_1 i c_2 te topologija čestica. Uz navedene parametre, definiraju se i varijable u koje se zapisuje nakon koliko sekundi je algoritam došao do dobrog rješenja, nakon koliko iteracija te nakon koliko izvršenja funkcije koju optimiziramo.

Konstruktor nastavlja na sljedeći način:

```
            particleCoords=np.random.default_rng().uniform(low=-
    self.xMax,high=self.xMax, size=(self.particleCount, self.dimensions))
```

Definira se `self.particleCount` koordinata, svaka u `self.dimensions` dimenzija. Svaka dimenzija koordinate se postavlja između -`self.xMax` i `self.xMax`. Te koordinate se koriste kod generiranja čestica. Ako koristimo topologiju `gbest`, čestice se generiraju ovako:

```
        self.particles=[]
        if topology=="gbest":
            indexesOfP=list(range(0,particleCount))
            for p in particleCoords:
                self.particles.append(Particle(p,self.function,indexesOfP))
```

Konstruktoru svake čestice (koji sam objasniti kasnije u radu) se u parametru `indexesOfP` predaju indeksi svih čestica, što znači da su čestici sve ostale čestice susjedi. Ako koristimo topologiju `lbest`, kao parametar `topology` predajemo broj:

```
        elif isinstance(topology, int):
            allIndexes=list(range(0,particleCount))
            for i,p in enumerate(particleCoords):
                indexesOfP=[allIndexes[i]]
                for j in range(topology//2):
                    indexesOfP.append(allIndexes[(i+j+1)%self.particleCount
])
                    indexesOfP.append(allIndexes[i-j-1])
                self.particles.append(Particle(p,self.function,indexesOfP))
```

Taj broj označava broj susjednih čestica uključujući samu sebe, pa se česticama u konstruktor predaje `indexesOfP` koji sadrži indekse čestica koji su pomaknuti od pojedine čestice za do `topology//2`. Čestice u von Neumann topologiji generiramo ovako:

```
        elif topology=="vonNeumann":
            f1 = round(np.sqrt(self.particleCount))
            f2 = round(self.particleCount/f1)
            allIndexes=np.reshape(list(range(0,self.particleCount)),(f1,f2)
)
            for i,p in enumerate(particleCoords):
                (x,y)=np.where(allIndexes==i)
```

```
                x=x[0]
                y=y[0]
                indexesOfP=[allIndexes[x][y],allIndexes[x-
1][y],allIndexes[(x+1)%f1][y],allIndexes[x][y-1],allIndexes[x][(y+1)%f2]]
                self.particles.append(Particle(p,self.function,indexesOfP))
```

Broj čestica se rastavi na dva najveća faktora, pa se indeksi čestica rasporede u matricu dimenzija `f1 x f2`. U takvoj matrici je lakše povezati česticu sa svim susjednim česticama. Na kraju, nakon što su čestice s početnim koordinatama generirane, čestice provjeravaju svoje susjede i ažuriraju najbolji rezultat susjedstva.

```
        self.checkNeighborhood()
```

Ta funkcija je definirana tako da se iterira nad česticama te svaka čestica provjerava susjedstvo:

```
def checkNeighborhood(self):
        for p in self.particles:
            p.checkNeighborhood(self.particles)
```

Svaki algoritam definira svoju klasu čestice. Klasa čestice za PSO i njen konstruktor izgleda ovako: Čestici se predaju početne koordinate, objekt funkcije i indeksi susjeda. Uz koordinate se izračunavaju vrijednosti koordinata. Kako se radi o inicijalizaciji čestice, početne koordinate i vrijednosti se zapisuju i u varijablu za osobni najbolji rezultat i najbolji rezultat susjedstva. Početni vektor inercije se postavlja na nul-vektor.

```
def __init__(self, coords, function,informantIndexes):
        self.function=function
        eval=self.function.evaluate(coords)
        self.current=[coords,eval]
        self.personalBest=[coords,eval]
        self.bestKnown=[coords,eval]
        self.inertiaVector=np.zeros(coords.shape)
        self.informantIndexes=informantIndexes
```

Nakon inicijalizacije, sve čestice provjeravaju susjedstvo i ažuriraju najbolje rezultate susjedstva: Ako je rezultat nekog susjeda bolji od rezultata čestice, taj bolji rezultat se zapisuje u česticu.

```
    def checkNeighborhood(self, allParticles):
        betterIndex=None
        for i in self.informantIndexes:
            if(allParticles[i].personalBest[1]<self.bestKnown[1]):
                betterIndex=i
        if(betterIndex!=None):
            self.bestKnown=deepcopy(allParticles[betterIndex].personalBest)
```

Algoritam pokrećemo funkcijom iterate, koja `maxIterations` puta poziva funkciju `runIteration` te zaustavlja izvršavanje algoritma ako se dođe do prihvatljivog rješenja:

```
    def iterate(self):
        bestResults=np.array([])
        for i in range(self.maxIterations):
            self.iterationCount+=1
            self.runIteration()
            bestResults=np.append(bestResults,self.getBestResult())
            if(bestResults[-
1]<self.function.getAcceptableError() and self.iterationCount>=self.minIter
ations):
                return np.reshape(bestResults,(-1,2))
        return np.reshape(bestResults,(-1,2))
```

Funkcija `getBestResult` provjerava rezultate svih čestica i vraća najbolji rezultat:

```
    def getBestResult(self):
        currentResult=deepcopy(self.particles[0])
        for p in self.particles:
            if p.bestKnown[1]<currentResult.bestKnown[1]:
                currentResult=deepcopy(p)
        return currentResult.bestKnown
```

Funkcija `runIteration` izvršava `move` funkciju svake čestice. Nakon što se pomaknu sve čestice, čestice provjeravaju svoje susjede te se zapisuje najbolji rezultat iteracije. Ako je nakon neke iteracije postignut prihvatljiv rezultat, zapisuje se broj te iteracije, broj izvršenja funkcije i vrijeme kada je taj rezultat dostignut. Također se bilježi rezultat algoritma nakon `minIterations` iteracija:

```
    def runIteration(self):
        for p in self.particles:
            p.move(self.c1, self.c2,self.dimensions,self.xMax)
        self.checkNeighborhood()
        result=self.getBestResult()
        if(np.abs(result[1]-
self.function.getOptimalResult())<=self.function.getAcceptableError() and s
elf.iterationAtAcceptableResult==None):
            self.iterationAtAcceptableResult=self.iterationCount
            self.functionExecutionAtAcceptableResult=self.function.numberOf
Evaluations
            self.timeAtAcceptableResult=time.time()
        if(self.iterationCount==self.minIterations):
            self.resultAtMinIter=result
            self.evaluationsAtMinIter=self.function.numberOfEvaluations
            self.timeAtMinIter=time.time()
```

Funkcija čestice `move` radi sljedeće: sprema vektor kretanja prošle iteracije kao inercijsku komponentu novog vektora. Nakon toga, za svaku dimenziju generira nasumični broj između 0 i 1. Uz te nasumične brojeve izračunava osobnu komponentu novog vektora kretanja. Generira se još nasumičnih brojeva te se izračunava i komponenta najboljeg poznatog rješenja.

```
    def move(self, c1, c2, dimensions, velocityClamp):
        inertiaVectorComponent = self.inertiaVector

        personalBestVectorFactors = np.random.rand(dimensions,)
        personalBestVectorComponent = (self.personalBest[0]-
self.current[0])*personalBestVectorFactors*c1

        bestKnownVectorFactors = np.random.rand(dimensions,)
        bestKnownVectorComponent = (self.bestKnown[0]-
self.current[0])*bestKnownVectorFactors*c2
```

Kako bi dobili novi vektor kretanja, zbrojimo komponente koje smo izračunali: inercijsku komponentu, osobnu komponentu i komponentu najboljeg poznatog rješenja. Trenutnoj poziciji čestice pribrajamo vektor kretanja. Kako čestica ne bi izašla iz prostora pretrage, ograničimo koordinate nove čestice i elemente vektora kretanja. Vektor kretanja zapisujemo za sljedeću iteraciju. Izračunavamo vrijednost novog rješenja.

```
        movementVector = np.minimum(np.maximum(inertiaVectorComponent+perso
nalBestVectorComponent+bestKnownVectorComponent,-
velocityClamp),velocityClamp)
        newLocation=np.minimum(np.maximum(self.current[0]+movementVector,-
velocityClamp),velocityClamp)
        newLocationEval=self.function.evaluate(newLocation)
        self.inertiaVector=movementVector
        self.current=[newLocation,newLocationEval]
```

Ako je novo rješenje bolje od osobnog najboljeg rješenja čestice, ono postaje novo osobno najbolje rješenje. Isto vrijedi i za najbolje poznato rješenje.

```
        if(newLocationEval<self.personalBest[1]):
            self.personalBest=[newLocation,newLocationEval]
        if(newLocationEval<self.bestKnown[1]):
            self.bestKnown=[newLocation,newLocationEval]
```

7.2. SPSO

Standardni algoritam PSO razlikuje se od originalnog u tome da koristi težinsku vrijednost inercije, kako bi se smanjio utjecaj vektora inercije te kako on ne bi rastao u nedogled. U ovom radu koristim težinsku vrijednost koja se linearno smanjuje kroz vrijeme. U konstruktoru klase

SPSO se dodaju parametri *inertiaWeightMax* i *inertiaWeightMin*. Bilježimo i broj iteracije kako bi mogli izračunati težinsku vrijednost.

```
class StandardPSO:
      def __init__(
self, function, particleCount, dimensions,minIterationCount,maxIterationCou
nt, c1=2, c2=2, inertiaWeightMax=0.9, inertiaWeightMin=0.4, topology="gbest
"
      ):
                                          …
        self.inertiaWeightMax=inertiaWeightMax
        self.inertiaWeightMin=inertiaWeightMin
        self.iterationCount=0
```

Funkcija runIteration izgleda ovako: svakoj čestici se, kod pozivanja funkcije move, izračunava i predaje težinska vrijednost inercije te se ažurira broj iteracije.

```
    def runIteration(self):
        for p in self.particles:
            p.move(self.c1, self.c2,self.dimensions,self.xMax,self.calculat
eLinearInertiaWeight())
        self.checkNeighborhood()
        self.iterationCount+=1
                                          …
```

Funkcija `calculateLinearInertiaWeight` linearno smanjuje težinsku vrijednost inercije kroz iteracije:

```
    def calculateLinearInertiaWeight(self):
        weight = self.inertiaWeightMin+(self.inertiaWeightMax-
self.inertiaWeightMin)*(1-(self.iterationCount/self.maxIterations))
        return weight
```

Jedina razlika u klasi čestice je korištenje težinske vrijednosti inercije, kod izračunavanja inercijske komponente novog vektora.

```
def move(self, c1, c2, dimensions, velocityClamp, inertiaWeight):
        inertiaVectorComponent = self.inertiaVector * inertiaWeight
                                          …
```

7.3. PSON

Kod algoritma PSON se ne koristi niti bilježi najbolje rješenje susjedstva, nego za kretanje čestice koriste sva rješenja čestica u susjedstvu. U konstruktoru čestice se predaju parametri $\chi = 0{,}729844$ i $\phi_{max} = 4{,}1$. Funkcija kretanja čestice move izgleda ovako: Predaje joj se popis

svih čestica. Izračunava se dimenzija funkcije. Određuje se najveća moguća vrijednost parametra ϕ za pojedinu česticu, $\phi = \frac{\phi_{max}}{|\mathcal{N}|}$, gdje je $\mathcal{N}$ skup čestica povezanih s česticom.

```
def move(self, allParticles):
        dimensions=dimensions=self.current[0].shape[0]

        upperSum=0
        phi=np.zeros(dimensions)

        phiPerElement=self.phiMax/len(self.informantIndexes)
```

Za svaki element koordinate čestice generira se nasumična vrijednost između 0 i ϕ. Te vrijednosti se množe s pripadnim elementima koordinate najboljeg osobnog rješenja susjednih čestica. Uz pomoć svake čestice u susjedstvu generira se nova točka prema kojoj se čestica miče (točka privlačenja).

```
        phiPerElement=self.phiMax/len(self.informantIndexes)
        for p in self.informantIndexes:
            pb=allParticles[p]
            phiI=np.random.uniform(0,phiPerElement,dimensions)
            phi+=phiI
            upperSum+=np.multiply(phiI,pb.personalBest[0])
        attractorPoint=np.divide(upperSum,phi)
```

Iz točke privlačenja i trenutne lokacije čestice računa se vektor kretanja.

```
        a1=attractorPoint-self.current[0]
        a2=np.multiply(phi,a1)
        a3=self.inertiaVector+a2
        movementVector = self.chi*a3
```

Računa se nova lokacija te se provjerava je li novo rješenje bolje ili ne.

```
        newLocation=self.current[0]+movementVector
        newLocationEval=self.function.evaluate(newLocation)
        self.inertiaVector=movementVector
        self.current=[newLocation,newLocationEval]

        if(newLocationEval<self.personalBest[1]):
            self.personalBest=[newLocation,newLocationEval]
```

7.4. HEA

U algoritmu koji se kombinira s genetskim algoritmom, u konstruktoru predajemo parametre $reproductionRate = 0{,}7$, $crossoverRate = 0{,}25$ i $mutationRate = 0{,}05$.

```
class HEA:
    def __init__(self, function, particleCount, dimensions,minIterationCoun
t,maxIterationCount, c1=2, c2=2, inertiaWeightMax=0.9, inertiaWeightMin=0.4
, reproductionRate=0.7, crossoverRate=0.25, mutationRate=0.05, topology="gb
est"):
                                        …
        self.reproductionRate=reproductionRate
        self.crossoverRate=crossoverRate
        self.mutationRate=mutationRate
```

Funkcija algoritma runIteration izgleda ovako: prvo se normalno provodi kretanje čestica standardnim algoritmom PSO. Nakon kretanja se bilježe vrijednosti trenutnih lokacija čestica. Iz tih vrijednosti se računa vjerojatnost da se neka čestica odabire u sljedećim koracima.

```
    def runIteration(self):
        for p in self.particles:#do PSO
            p.move(self.c1, self.c2,self.dimensions,self.xMax,self.calculat
eLinearInertiaWeight())

        self.fitnessSet=[]
        probabilities=[]
        for p in self.particles:
            self.fitnessSet.append(p.current[1])

        fitnessSum=sum(self.fitnessSet)
        for f in self.fitnessSet:
            probabilities.append(f/fitnessSum
        probabilities.insert(0,0)
```

Generira se nova populacija čestica. Uzimaju se dvije po dvije čestice. U svakom odabiru te dvije čestice moraju biti različite. Čestice se odabiru rulet selekcijom: što je bolje rješenje čestice, veća je vjerojatnost da će čestica biti odabrana.

```
        newPopulation=[]
        for k in range(0,self.particleCount,2):#generating new population
            #pick two parents
            indexa=0
            indexb=0
            indexa=self.rouletteSelectIndex(probabilities,np.random.rand())
            indexb=self.rouletteSelectIndex(probabilities,np.random.rand())
            while (indexa==indexb):
                randb=np.random.rand()
                indexb=self.rouletteSelectIndex(probabilities,randb)
            xa=self.particles[indexa]
            xb=self.particles[indexb]
```

Generira se nasumični broj između 0 i 1. Ako je broj između 0 i $reproductionRate$, provodi se reprodukcija, što znači da se odabrane čestice ne mijenjaju te se samo prenose u novu populaciju.

```
r=np.random.rand()
if(r<=self.reproductionRate):
    newPopulation.append(xa)
    newPopulation.append(xb)
```

Ako je broj između $reproductionRate$ i $reproductionRate + crossoverRate$, provodi se križanje čestica. Generira se niz nasumičnih brojeva između 0 i 1, $rArr$. Nove koordinate se generiraju tako da se koordinate prvog roditelja pomnožene s $rArr$ zbrajaju s koordinatama drugog roditelja pomnoženih s $1 - rArr$. Drugi set novih koordinati se generira na isti način, samo što roditelji mijenjaju mjesta. Evaluiraju se nove koordinate i zapisuju u čestice te se te čestice prenose u sljedeću generaciju.

```
elif (r<=self.reproductionRate+self.crossoverRate):
    rArr=np.random.uniform(0,1,self.dimensions)
    childACoords=np.multiply(rArr,xa.current[0]) +np.multiply(n
p.ones(self.dimensions)-rArr,xb.current[0])
    childBCoords=np.multiply(rArr,xb.current[0]) +np.multiply(n
p.ones(self.dimensions)-rArr,xa.current[0])
    evalA=self.function.evaluate(childACoords)
    evalB=self.function.evaluate(childBCoords)
    xa.current=[childACoords,evalA]
    xb.current=[childBCoords,evalB]
    if(xa.current[1]<xa.personalBest[1]):
        xa.personalBest=[childACoords,evalA]
    if(xa.current[1]<xa.bestKnown[1]):
        xa.bestKnown=[childACoords,evalA]
    if(xb.current[1]<xb.personalBest[1]):
        xb.personalBest=[childBCoords,evalB]
    if(xb.current[1]<xb.bestKnown[1]):
        xb.bestKnown=[childBCoords,evalB]
    newPopulation.append(xa)
    newPopulation.append(xb)
```

Ako je broj između $reproductionRate + crossoverRate$ i 1, radi se Gaussova mutacija čestica. Koordinate čestica se mutiraju tako da se koordinate pomaknu u svakoj dimenziji po normalnoj distribuciji, uz srednju vrijednost 0 i devijaciju $0{,}1 \cdot 2 \cdot XMax$, gdje je $XMax$ granica prostora pretrage u jednoj dimenziji. Nakon generiranja svih čestica, čestice provjeravaju susjedstvo i bilježe bolja rješenja.

```
else:
```

```
                childACoords=xa.current[0]*(1+np.random.normal(0,np.sqrt(0.
1*2*self.function.getXMax())))
                childBCoords=xb.current[0]*(1+np.random.normal(0,np.sqrt(0.
1*2*self.function.getXMax())))
                evalA=self.function.evaluate(childACoords)
                evalB=self.function.evaluate(childBCoords)
                xa.current=[childACoords,evalA]
                xb.current=[childBCoords,evalB]
                if(xa.current[1]<xa.personalBest[1]):
                    xa.personalBest=[childACoords,evalA]
                if(xa.current[1]<xa.bestKnown[1]):
                    xa.bestKnown=[childACoords,evalA]
                if(xb.current[1]<xb.personalBest[1]):
                    xb.personalBest=[childBCoords,evalB]
                if(xb.current[1]<xb.bestKnown[1]):
                    xb.bestKnown=[childBCoords,evalB]
                newPopulation.append(xa)
                newPopulation.append(xb)

        self.particles=newPopulation
        self.checkNeighborhood()
```

7.5. SDEA

Kod algoritma koji se kombinira s algoritmom diferencijalne evolucije, u konstruktoru se predaju parametri $differentialEvolutionChance = 0{,}25$, $crossoverProbability = 0{,}5$, $mutationRate = 0{,}8$:

```
class SDEA:
    def __init__(self, function, particleCount, dimensions,minIterationCoun
t,maxIterationCount, c1=2, c2=2, inertiaWeightMax=0.9, inertiaWeightMin=0.4
, differentialEvolutionChance=0.25, crossoverProbability=0.5, mutationRate=
0.8, topology="gbest"):
                                        …
        self.differentialEvolutionChance=differentialEvolutionChance
        self.crossoverProbability=crossoverProbability
        self.mutationRate=mutationRate
```

Svaku iteraciju, generira se nasumični broj između 0 i 1 i provjerava se je li taj broj manji od $differentialEvolutionChance$. Ako nije, tu iteraciju se normalno provodi standardni PSO. Ako jest, umjesto PSO se provodi diferencijalna evolucija.

```
    def runIteration(self):

        if np.random.rand()<self.differentialEvolutionChance:#Differential
evolution
            for i,p in enumerate(self.particles):
```

```
                p.differentialEvolution(self.crossoverProbability,self.muta
tionRate, self.particles,i)
        else:
            for p in self.particles:#do PSO
                p.move(self.c1, self.c2,self.dimensions,self.xMax,self.calc
ulateLinearInertiaWeight())
```

Kod diferencijalne evolucije, uz česticu nad kojom se provodi, odabiru se još tri čestice, koje su različite međusobno i različite od trenutne čestice.

```
    def differentialEvolution(self,mutationRate, particles, ownIndex):
        particleCount=len(particles)

        particle1Index=np.random.randint(particleCount)
        while ownIndex==particle1Index:
            particle1Index=np.random.randint(particleCount)

        particle2Index=np.random.randint(particleCount)
        while ownIndex==particle2Index or particle1Index==particle2Index:
            particle2Index=np.random.randint(particleCount)

        particle3Index=np.random.randint(particleCount)
        while ownIndex==particle3Index or particle1Index==particle3Index or
 particle2Index==particle3Index:
            particle3Index=np.random.randint(particleCount)

        particle1=particles[particle1Index]
        particle2=particles[particle2Index]
        particle3=particles[particle3Index]
```

Određuju se target vektor, koji je trenutna lokacija čestice, mutant vektor i trial vektor, koji predstavlja novu lokaciju koja se uspoređuje s rješenjem čestice. Odabiremo neki indeks lokacije čestice, kako bi barem jedan element lokacije kasnije bio zamijenjen s novogeneriranim vektorom mutacije. Generira se niz nasumičnih vrijednosti između 0 i 1, jednake veličine kao koordinate čestica.

```
        targetVector=self.current[0]
        mutantVector=particle1.current[0]+mutationRate*(particle2.current[0
]-particle3.current[0])
        trialVector=np.array([])

        randomIndex=np.random.randint(len(targetVector))

        rand1 = np.random.uniform(0,1,len(targetVector))
```

Uz pomoć funkcije `generateCrossoverArray`, koja svaki element `rand1` vektora postavlja na 0 ili 1 ovisno je li vrijednost veća ili manja od $crossoverProbability$, odlučuje se koji će

elementi mutant vektora zamijeniti elemente trenutne lokacije. Prije odabrani nasumični indeks se postavlja na 1, kako bi barem jedan element mutirao. Generira se trial vektor i uspoređuje se s postojećim rješenjima.

```
        rand1=self.generateCrossoverArray(rand1)
        rand1[randomIndex]=1
        trialVector=np.multiply(mutantVector,rand1)+np.multiply(targetVecto
r,np.ones(len(targetVector))-rand1)

        trialVectorValue=self.function.evaluate(trialVector)
        if(trialVectorValue<self.current[1]):
            self.current=[trialVector,trialVectorValue]
        if(self.current[1]<self.personalBest[1]):
            self.personalBest=[trialVector,trialVectorValue]
```

7.6. PSOSA

Kod hibridnog algoritma sa simuliranim prekaljivanjem koriste se parametri $maxN1$, $maxN2$, η, $interdec$, P_r i $tempRatio$. Autori originalnog algoritma nisu naveli vrijednosti tih parametara, pa sam ih ja postavio ovako: $maxN1 = 5$, $maxN2 = 5$, $\eta = 1$, $interdec = \frac{particleCount}{1000}$, $P_r = 0.01$, $tempRatio = \frac{1}{maxN1 \cdot maxIterations \cdot particleCount}$. Kod postavljanja algoritma se dohvaćaju najbolje i najgore rješenje početnih čestica te se iz njih izračunava početna temperatura.

```
class PSOSA:
    def __init__(self, function, particleCount, dimensions,minIterationCoun
t,maxIterationCount, c1=2, c2=2, inertiaWeightMax=0.9, inertiaWeightMin=0.4
, maxN1=5, maxN2=10, eta=1, topology="gbest"):
…
        self.maxN1=maxN1
        self.maxN2=maxN2
        self.eta=eta
        self.interdec=particleCount/1000

…
        fBest=self.getBestStarting()
        fWorst=self.getWorstStarting()
        Pr=0.01
        startingTemperature=-(fWorst[1]-fBest[1])/np.log(Pr)
        self.temperature=startingTemperature
        self.tempRatio=1/(self.maxN1*self.maxIterations*self.particleCount)
```

U funkciji runIteration se prvo provodi standardni PSO. Bilježi se najbolji rezultat nakon kretanja čestica. Ako nije došlo do poboljšanja rješenja, provodi se simulirano prekaljivanje nad svakom česticom. Nakon prekaljivanja se ažurira parametar η te se provjeravaju rezultati.

```
    def runIteration(self):
        for p in self.particles:#do PSO
            p.move(self.c1, self.c2,self.dimensions,self.xMax,self.calculat
eLinearInertiaWeight())

        self.bestResults=np.append(self.bestResults,self.getBestResult())
        self.bestResults=np.reshape(self.bestResults,(-1,2))

        if (self.bestResults[-2][1]-self.bestResults[-1][1])==0:
            for p in self.particles:
                self.temperature=p.simulatedAnnealing(self.maxN1, self.maxN
2,self.temperature, self.eta,self.tempRatio)

        self.eta=self.eta*(1-self.interdec)
                                                    …
```

Samo prekaljivanje se izvodi na sljedeći način: $maxN1$ puta se nad svakom česticom provodi još jedna petlja. Ta petlja se provodi do $maxN2$ puta. U toj petlji se generira niz nasumičnih brojeva dobivenih normalnom distribucijom sa srednjom vrijednošću 0 i standardnom devijacijom 1 te se taj niz množi sa parametrom η. Vrijednost pomaka se dobije množenjem rezultata iz prošle iteracije petlje `newPersonalBest` (u prvoj iteraciji, najbolji osobni rezultat čestice) sa nizom nasumičnih brojeva. Pomak se pribraja `newPersonalBest` te se ta točka evaluira. Ako je novo rješenje bolje od najboljeg osobnog rješenja čestice, ono se sprema, a $maxN2$ petlja prestaje te se prelazi na novu iteraciju petlje $maxN1$. Ako novo rješenje nije bolje, onda se, uz pomoć temperature, odlučuje hoće li se novo rješenje svejedno uzeti, ili će se poslati u novu iteraciju petlje $maxN2$. Na kraju iteracije petlje $maxN1$ ažurira se temperatura.

```
    def simulatedAnnealing(self,maxN1,maxN2, temperature, eta, tempRatio):
        t=temperature
        for i in range(maxN1):
            newPersonalBest=deepcopy(self.personalBest[0])
            for j in range(maxN2):
                rands=np.random.default_rng().normal(0,1, size=(self.curren
t[0].size))*eta
                delta=newPersonalBest*rands

                newPersonalBest=np.minimum(np.maximum(newPersonalBest+delta
,-self.function.getXMax()),self.function.getXMax())
                newPersonalBestValue=self.function.evaluate(newPersonalBest
)
```

```
                if(newPersonalBestValue<self.personalBest[1]):
                    self.personalBest=[newPersonalBest,newPersonalBestValue
]
                    break
                elif min(np.exp(-(self.personalBest[1]-
self.bestKnown[1])/t),1)>np.random.rand():
                    self.current=[newPersonalBest,newPersonalBestValue]
                    break
            t=t*(1-tempRatio)
        return t
```

7.7. TRIBES

Algoritam TRIBES se predaje samo funkcija, minimalni i maksimalni broj iteracija, dimenziju funkcije te hoće li algoritam koristiti pseudogradijent. U ovom radu sam, kod kretanja čestica, postavio da se koriste pseudogradijenti. Na početku se generira jedna, početna čestica te se dodjeljuje prvom plemenu. Postavlja se broj veza između čestica $L = 1$.

```
class TRIBES:
    def __init__(self, function,minIterationCount,maxIterationCount, dimens
ions, pseudogradient):
                                        …
        self.particles=[]
        particleCoords=np.random.default_rng().uniform(low=-
self.xMax,high=self.xMax, size=(self.dimensions))
        self.particles.append(Particle(deepcopy(self.particleIndexes),parti
cleCoords,self.function,deepcopy(self.tribeIndexes),[],[],self.pseudogradie
nt))
        self.particleIndexes+=1
        self.tribeIndexes+=1
        self.L=1
        self.tribes=[0]
```

Svaku iteraciju, algoritam radi sljedeće: Smanjuje se parametar L za 1. Ako je L manji od 0, provodi se brisanje postojećih i generiranje novih čestica i plemena (adaptacija roja). Kod adaptacije se određuju dobra i loša plemena. U svakom plemenu se prebrojava broj dobrih čestica (čestice koje su u prethodnoj iteraciji poboljšale svoje rješenje). Za svako pleme odabire se nasumični cijeli broj između 0 i veličine plemena. Ako je broj dobrih čestica veći od nasumičnog broja, pleme se smatra dobrim. Ako je broj dobrih čestica manji, pleme se smatra lošim.

```
    def runIteration(self):
        goodTribes=[]
        badTribes=[]
```

```
        self.L-=1

        if self.L<0:#swarm adaptation
            for i in self.tribes:
                tribeMembers=list(filter(lambda x:(x.tribe==i),self.particl
es))
                goodMembers=list(filter(lambda x:(x.previousMemory==1),trib
eMembers))
                if(len(tribeMembers)==0):
                    self.tribes.remove(i)
                    continue
                p=np.random.randint(len(tribeMembers))
                if(len(goodMembers)>p):
                    goodTribes.append(i)
                else:
                    badTribes.append(i)
```

Prolazi se kroz dobra plemena. Iz svakog dobrog plemena briše se jedna čestica. Ako pleme ima više od jedne čestice, briše se ona čestica koja ima najgore rješenje te se vanjske veze čestica prespajaju na ostale članove plemena, kako pleme ne bi izgubilo vezu s ostalim plemenima.

```
                for i in goodTribes
                    tribeMembers=list(filter(lambda x:(x.tribe==i),self.par
ticles))
                    if(len(tribeMembers)>1):
                        tribeMembers.sort(reverse=True,key=lambda x:(x.best
Known[1]))
                        worstParticle=tribeMembers[0]
                        bestParticle=tribeMembers[-1]
                        bestParticle.externalInformants=list(set(bestPartic
le.externalInformants+worstParticle.externalInformants))

                        if (len(worstParticle.externalInformants)==0):
                            pass
                        else:
                            for link in worstParticle.externalInformants:
                                externalParticle=list(filter(lambda x:(x.in
dex==link),self.particles))[0]

                                externalParticle.externalInformants.append(
deepcopy(bestParticle.index))
                                bestParticle.externalInformants.append(deep
copy(externalParticle.index))

                        for p in self.particles:
                            try:
```

```
                                p.internalInformants.remove(worstParticle.i
ndex)
                            except ValueError:
                                pass
                            try:
                                p.externalInformants.remove(worstParticle.i
ndex)
                            except ValueError:
                                pass
                            p.externalInformants=list(set(p.externalInforma
nts))
                        self.particles = [i for i in self.particles if i.in
dex!=worstParticle.index]
```

Ako pleme ima samo jednu česticu, ona se briše samo ako je barem jedan od njezinih povezanih čestica bolji od nje. Ako takva povezana čestica postoji, prespajaju se poveznice između plemena te se čestica i pleme brišu.

```
                    else:
                        worstParticle=tribeMembers[0]
                        connectedExternalParticles=list(filter(lambda x:(wo
rstParticle.index in x.externalInformants),self.particles))
                        betterInformantExists=0
                        for p in connectedExternalParticles:
                            if(p.bestKnown[1]<tribeMembers[0].bestKnown[1])
:
                                betterInformantExists=1
                                break
                        if betterInformantExists:
                            for p in connectedExternalParticles:
                                otherConnections=[x for x in connectedExter
nalParticles if x.index!=p.index]

                                othersOfTribe = list(filter(lambda x:(x.tri
be==p.tribe),self.particles))
                                for p2 in otherConnections:
                                    othersOfAnotherTribe = list(filter(lamb
da x:(x.tribe==p2.tribe),self.particles))
                                    anotherConnectionExists=0
                                    for p3 in othersOfTribe:
                                        for p4 in othersOfAnotherTribe:
                                            if(p3.index in p4.externalInfor
mants):
                                                anotherConnectionExists=1
                                                break
                                            if(anotherConnectionExists):
                                                break
                                        if(anotherConnectionExists):
```

```
                                        break
                                if (not anotherConnectionExists):
                                    p.externalInformants.append(deepcop
y(p2.index))
                                    p2.externalInformants.append(deepco
py(p.index))
                                try:
                                    p2.externalInformants.remove(worstP
article.index)
                                except ValueError:
                                    pass
                            try:
                                p.externalInformants.remove(worstPartic
le.index)
                            except ValueError:
                                pass

                        self.particles = [j for j in self.particles if
j.index!=worstParticle.index]
                        self.tribes=[j for j in self.tribes if j!=i]
```

Nakon što se iz dobrih plemena obrišu čestice, iz loših plemena se generiraju nove čestice. Iz svakog lošeg plemena se generiraju dvije čestice: jedna, nevezana čestica, koja se generira nasumično bilo gdje u prostoru pretrage (ili na strani ili rubu prostora), i druga, vezana čestica, koja se generira između najbolje čestice plemena i najbolje povezane čestice te čestice.

```
            newParticles=[]
            newIndexes=[]
            for i in badTribes:
                tribeMembers=list(filter(lambda x:(x.tribe==i),self.particl
es))
                tribeMembers.sort(reverse=True,key=lambda x:(x.bestKnown[1]
))
                bestMember=tribeMembers[-1]
                newCoords=np.random.default_rng().uniform(low=-
self.xMax,high=self.xMax, size=(self.dimensions))
                generationRule=np.random.randint(3)
                if(generationRule==0):
                    pass
                elif(generationRule==1):
                    dimensionToChange=np.random.randint(self.dimensions)
                    prefix=-1 if np.random.randint(2) else 1
                    newCoords[dimensionToChange]=prefix*self.function.getXM
ax()
                elif(generationRule==2):
                    dimensionToChange1=np.random.randint(self.dimensions)
                    dimensionToChange2=np.random.randint(self.dimensions)
                    while(dimensionToChange2==dimensionToChange1):
```

```
                        dimensionToChange2=np.random.randint(self.dimension
s)
                    prefix1=-1 if np.random.randint(2) else 1
                    prefix2=-1 if np.random.randint(2) else 1
                    newCoords[dimensionToChange1]=prefix1*self.function.get
XMax()
                    newCoords[dimensionToChange2]=prefix2*self.function.get
XMax()
                newParticles.append(Particle(deepcopy(self.particleIndexes)
,newCoords,self.function,deepcopy(self.tribeIndexes),[],[deepcopy(bestMembe
r.index)],self.pseudogradient))
                newIndexes.append(deepcopy(self.particleIndexes))
                bestMember.externalInformants.append(deepcopy(self.particle
Indexes))
                self.particleIndexes+=1
                informantOfBestParticle=list(filter(lambda x:(bestMember.in
dex in x.internalInformants or bestMember.index in x.externalInformants),se
lf.particles))
                informantOfBestParticle.sort(reverse=True,key=lambda x:(x.b
estKnown[1]))
                if(len(informantOfBestParticle)==0):
                    pass
                else:
                    bestInformant=deepcopy(informantOfBestParticle[-1])
                    distance=np.sqrt(np.sum(np.square(bestInformant.bestKno
wn[0]-bestMember.bestKnown[0])))
                    r=np.random.default_rng().uniform(low=-
distance,high=distance, size=(self.dimensions))
                    newCoords2=deepcopy(bestInformant.bestKnown[0])+r
                    newParticles.append(Particle(deepcopy(self.particleInde
xes),newCoords2,self.function,deepcopy(self.tribeIndexes),[],[],self.pseudo
gradient))
                    newIndexes.append(deepcopy(self.particleIndexes))
                    self.particleIndexes+=1
```

Sve novogenerirane čestice se stavljaju u novo pleme. Nakon generiranja čestica, parametar L se postavlja na pola broja veza u cijelom roju čestica te se provodi kretanje čestica. Ako nije došlo do adaptacije roja, samo se miču čestice.

```
            self.particles=np.append(self.particles,newParticles)
            self.L=self.getConnections()//2

        for p in self.particles:
            p.move(self.iterationCount)
```

Čestice se kreću na sljedeći način: definira se strategija kretanja, ovisno o memoriji čestice. Strategije su pivot, pivot s bukom te lokalno uz Gaussovu distribuciju. Koja strategija

se kada koristi je objašnjeno u šestom poglavlju. Nakon što se strategija odredi, ona se i iskoristi za kretanje čestice te se izračunava vrijednost novog rješenja. Bilježi se memorija čestice te se novo rješenje uspoređuje s postojećim rješenjima čestice.

```
    def move(self,iter):
        self.iteration=iter


        movementStrategy=self.defineStrategy()
        newLocation=movementStrategy()
        newLocationEvaluation=self.function.evaluate(newLocation)

        self.pastPreviousMemory=self.previousMemory
        if(newLocationEvaluation<self.current[1]):
            self.previousMemory=1
        elif(newLocationEvaluation>self.current[1]):
            self.previousMemory=-1
        else:
            self.previousMemory=0
        self.current=[newLocation,newLocationEvaluation]

        if(newLocationEvaluation<self.personalBest[1]):
            self.personalBest=[newLocation,newLocationEvaluation]
        if(newLocationEvaluation<self.bestKnown[1]):
            self.bestKnown=[newLocation,newLocationEvaluation]
```

Strategija pivot se koristi ovako: izračuna se udaljenost između najboljeg osobnog rješenja čestice i najboljeg rješenja plemena. Određuje se dimenzija prostora te se udaljenost dijeli s dimenzijom prostora, kako u ranim iteracijama nove čestice ne bi izletjele iz prostora pretrage. Nasumično, oko točaka osobnog najboljeg rješenja i najboljeg rješenja plemena, odabiru se dvije nove točke. Računaju se parametri c_1 i c_2 iz vrijednosti osobnog najboljeg rješenja i najboljeg rješenja plemena te se generira novo rješenje.

```
    def pivot(self):
        dist=np.sqrt(np.sum(np.square(self.personalBest[0]-
self.bestKnown[0])))
        dimensions=self.personalBest[0].size

        distance=dist/dimensions

        r1=np.random.default_rng().uniform(low=-
distance,high=distance, size=(dimensions))
        r2=np.random.default_rng().uniform(low=-
distance,high=distance, size=(dimensions))
        point1=np.minimum(np.maximum(self.personalBest[0]+r1,-
self.function.getXMax()),self.function.getXMax())
```

```
        point2=np.minimum(np.maximum(self.bestKnown[0]+r2,-
self.function.getXMax()),self.function.getXMax())

        c1=self.personalBest[1]/(self.personalBest[1]+self.bestKnown[1])
        c2=self.bestKnown[1]/(self.personalBest[1]+self.bestKnown[1])

        x=np.minimum(np.maximum(c1*point1+c2*point2,-
self.function.getXMax()),self.function.getXMax())

        return x
```

Strategija pivota s bukom je skoro identična, jedino se na kraju svaka koordinata nove točke pomiče za nasumičnu vrijednost dobivenu normalnom distribucijom. Prosjek distribucije je 0, a devijacija se računa iz vrijednosti osobnog najboljeg rješenja i najboljeg rješenja plemena.

```
    def noisyPivot(self):
        dist=np.sqrt(np.sum(np.square(self.personalBest[0]-
self.bestKnown[0])))
…
        x=np.minimum(np.maximum(c1*point1+c2*point2,-
self.function.getXMax()),self.function.getXMax())

        deviation=(self.personalBest[1]-
self.bestKnown[1])/(self.personalBest[1]+self.bestKnown[1])
        noise=np.random.default_rng().normal(0,abs(deviation), size=(dimens
ions))
        x=x+noise
        return x
```

Kod strategije kretanja lokalno uz Gaussovu distribuciju, izračunava se točka između najboljeg osobnog rješenja i najboljeg rješenja plemena. Računa se udaljenost između tih točki te se iz nove točke i udaljenosti generira niz nasumičnih brojeva po normalnoj distribuciji, koji se pribraja trenutnom rješenju čestice.

```
    def localGauss(self):
        point=self.bestKnown[0]-self.current[0]
        dimensions=self.personalBest[0].size
        dist=np.sqrt(np.sum(np.square(self.bestKnown[0]-self.current[0])))
        distance=dist/dimensions
        gauss=np.random.normal(point,distance,dimensions)
        x=np.minimum(np.maximum(self.bestKnown[0]+gauss,-
self.function.getXMax()),self.function.getXMax())
        return x
```

8. Funkcije za optimizaciju

Kako bih ocijenio prije navedene algoritme, odabrao sam nekoliko funkcija različitih vrsta [24] .Funkcije koje sam odabrao za testiranje navedenih algoritama su:

- 30-dimenzionalna sfera, funkcija u obliku zdjele, s minimumom $f_1(0) = 0$. Domena pretrage je ograničena na $\pm 5{,}12$, a prihvatljivo rješenje je postavljena na 0,01.

$$f_1(x) = \sum_{i=1}^{n} x_i^2$$

- Rastrigin funkcija u 30 dimenzija, s minimumom $f_2(0) = 0$ i puno lokalnih minimuma Domena pretrage je ograničena na $\pm 5{,}12$, a prihvatljivo rješenje je postavljeno na 100.

$$f_2(x) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 10$$

- Griewank funkcija, u 30 dimenzija, izrazito teška funkcija s minimumom $f_3(0) = 0$. Domena pretrage je ograničena na ± 600, a prihvatljivo rješenje je postavljeno na 0,05.

$$f_3(x) = \sum_{i=1}^{n} \frac{(x_i - 100)^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$$

- Rosenbrock funkcija, u 30 dimenzija. Domena pretrage je ograničena na ± 30, a prihvatljivo rješenje je postavljeno na 100.

$$f_4 = \sum_{i=1}^{n-1} 100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2$$

- Zakharov funkcija u obliku tanjura, u 30 dimenzija. Domena pretrage je ograničena na ± 10, a prihvatljivo rješenje je postavljeno na 10.

$$f_5 = \sum_{i=1}^{n-1} 100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2$$

9. Rezultati testiranja

Svaki od algoritama sam testirao s topologijama lbest sa 3 i 5 susjeda i topologijom von Neumann, osim algoritma TRIBES, koji ne koristi topologije. Gdje se koriste, parametre algoritma sam postavio ovako

$$\omega_{max} = 0{,}9$$

$$\omega_{min} = 0{,}4$$

$$c_1 = c_2 = 2{,}05$$

$$N = 20$$

$$numberOfRounds = 20$$

$$minIter = 1000$$

$$maxIter = 10000$$

Za svaki algoritam i svaku topologiju provodi se 20 rundi optimizacije, što znači da se je svaki algoritam, osim algoritma TRIBES, izvršio 60 puta. Težinska vrijednost inercije se smanjivala linearno. Parametri X_{max} i V_{max} su bili drukčiji, ovisno o funkciji koje su algoritmi optimizirali. Svaki od algoritama je definiran u posebnoj klasi, a te klase se izvršavaju u glavnoj Python datoteci. Svaki algoritam je izvršen minimalno 1000 iteracija. Ako je u tih 1000 iteracija algoritam došao do prihvatljivog rješenja, runda izvršavanja je stala. Prihvatljiva rješenja funkcija definirana su u poglavlju 8. Ako nije došlo, algoritam je nastavio dok nije došao do prihvatljivog rješenja, ili do 10000 iteracija, što je došlo prije. Izvorni kod algoritama dostupan je na repozitoriju https://github.com/dgaz97/diplomski-rad. Uspješnost algoritama sam ocijenio na tri načina: uspješnost algoritma (u koliko posto slučajeva algoritam nađe prihvatljivo rješenje), kvaliteta rješenja nakon 1000 iteracija, vrijeme potrebno za 1000 iteracija, i medijan broja iteracija potrebnih da se dobije prihvatljivo rješenje. Te mjere za ocjenjivanje stohastičkih optimizacijskih algoritama preporučuju Ivković i sur. [25].

Uspješnost algoritama prikazana je u tablici 3. Originalni algoritam optimizacije rojem čestica je daleko najgori od navedenih algoritama: U 300 izvršavanja funkcije, do prihvatljivog rješenja, algoritam je došao samo jednom. Često, algoritam PSON nije došao do prihvatljivog rješenja, pogotovo s funkcijom Zakharov i Griewank. Algoritam SDEA se je mučio s funkcijama Rosenbrock i Zakharov, a SPSO s funkcijom Rosenbrock. Algoritmi HEA, PSOSA i TRIBES su uvijek našli rješenje unutar 10000 iteracija. U tablici 3 u stupcu topologija, 3 znači lbest topologija s 3 susjeda, 5 znači lbest s 5 susjeda

Tablica 3: Kombinacije algoritama, topologija i funkcija koje nisu bile 100 % uspješne

Algoritam	Topologija	Funkcija	Uspješno %
PSON	3	Rosenbrock	95%
PSON	von Neumann	Rosenbrock	95%
SPSO	5	Griewank	95%
SPSO	von Neumann	Zakharov	95%
PSON	3	Griewank	90%
PSON	5	Rosenbrock	85%
PSON	5	Griewank	75%
SPSO	5	Rosenbrock	55%
PSON	von Neumann	Griewank	45%
SPSO	von Neumann	Rosenbrock	45%
SDEA	3	Rosenbrock	40%
SPSO	3	Rosenbrock	40%
SDEA	5	Rosenbrock	35%
PSON	5	Zakharov	30%
PSON	von Neumann	Zakharov	25%
SDEA	von Neumann	Rosenbrock	20%
SDEA	3	Zakharov	0%
SDEA	5	Zakharov	0%
SDEA	von Neumann	Zakharov	0%

U tablici 4 su prikazane srednja vrijednost i medijan vrijednost algoritama nakon 1000 iteracija, za kombinacije algoritama i topologija. Ako pogledamo prosječne rezultate algoritama nakon 1000 iteracija vidimo da, nad testnim funkcijama, algoritmi HEA i PSOSA daju daleko najbolje rezultate. Sljedeći najbolji algoritam je PSON, dok prihvatljive rezultate još daje i TRIBES algoritam. Algoritmu PSO prosjek izrazito kvari funkcija Rosenbrock, dok se SDEA algoritam ne snalazi sa funkcijama Rosenbrock i Zakharov. Topologija na rezultate ne utječe puno, no vidimo da je najčešće najbolja topologija von Neumann, a nakon nje lbest s tri susjeda. Ako pogledamo medijane algoritama, tu se definitivno ističe algoritam PSOSA, koji daje oko devet redova veličine bolji medijan rješenja od sljedećeg najboljeg algoritma, HEA. Uz njih, još TRIBES i PSON algoritmi daju dobra rješenja.

Tablica 4: Rezultati algoritama, topologija i funkcija nakon 1000 iteracija

Algoritam	Topologija	Srednja vrijednost nakon 1000 iteracija	Medijan nakon 1000 iteracija
PSOSA	5	6,308224	8,09E-13
PSOSA	Neumann	6,205835	1,49E-12
PSOSA	3	6,033899	2,2E-12
HEA	Neumann	5,996494	0,006654
HEA	3	5,982082	0,006945
HEA	5	6,070364	0,009671
TRIBES		63,70818	18,50517
PSON	Neumann	23,82669	19,17358
PSON	3	36,75284	25,38268
PSON	5	32,73042	26,13769
SPSO	Neumann	36267,2	284,2427
SPSO	5	39289,53	288,0018
SDEA	Neumann	56071,44	294,9914
SPSO	3	41218,81	301,5757
SDEA	5	58969	302,9623
SDEA	3	58875,46	309,5211
OPSO	5	229292,8	323,595
OPSO	Neumann	244988	327,9194
OPSO	3	262171	345,3725

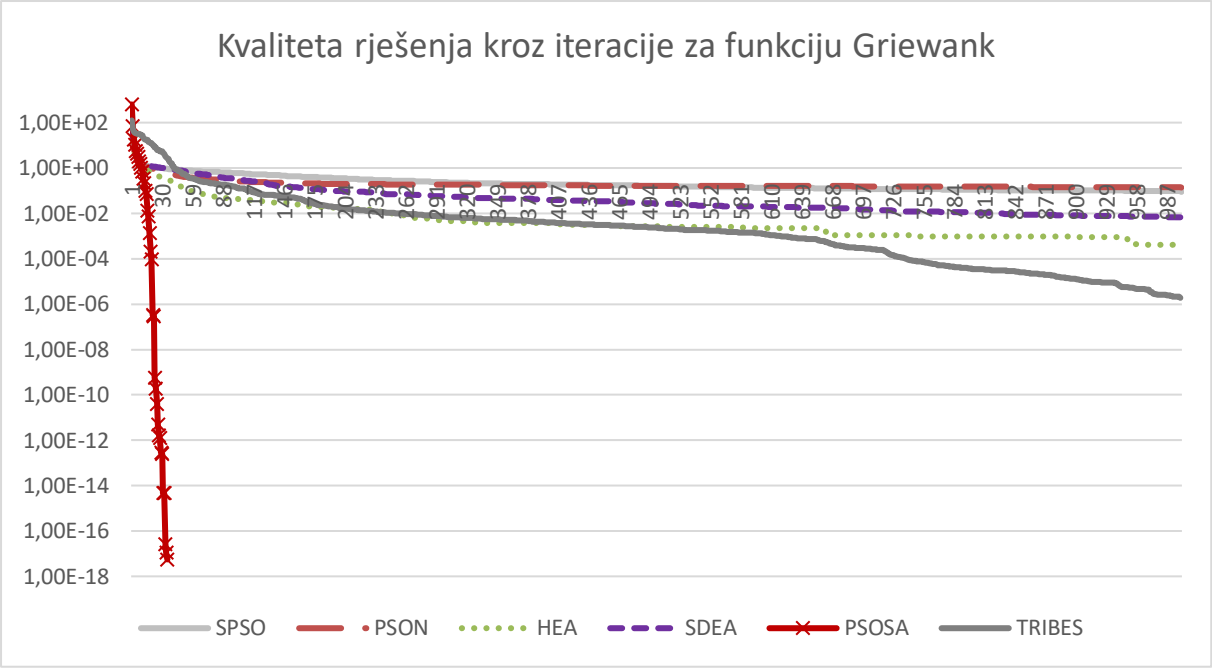
Što se tiče brzine izvršavanja algoritama, koja je vidljiva u tablici 5, SPSO algoritmu treba u prosjeku najmanje vremena da dođe do 1000 iteracija, no kao što smo vidjeli u tablici 4, u tom vremenu često ne dolazi do dobrog rješenja, dok mu je medijan broja iteracija iznad 5700. PSON je sljedeći po brzini izvršavanja te za razliku od SPSO daje dobre rezultate: medijan iteracija do prihvatljivog rješenja mu je između 200 i 500, ovisno o topologiji. SDEA je isto brz algoritam, no ne daje dobre rezultate te ima visoki medijan broja iteracija, oko 600. Algoritam HEA je dosta brz algoritam te mu je medijan iteracija do prihvatljivog rješenja jako dobar, između 60 i 90. PSOSA je malo sporiji algoritam od HEA, no daje prihvatljive rezultate u najmanji broj iteracija od svih algoritama: u 50 posto slučajeva, prihvatljivo rješenje nađe u manje od 20 iteracija. Algoritam TRIBES je daleko najsporiji od testiranih algoritama, no kako se on prilagođava funkciji koju pretražuje, vrlo je vjerojatno da će TRIBES algoritam uvijek naći rješenje, koliko god sporo, a medijan iteracija mu je prihvatljiv, malo iznad 1000. Originalni PSO algoritam nije prikazan jer nije barem u 50 posto slučajeva došao do prihvatljivog rješenja,

nakon 10000 iteracija. Algoritmi su brži ako koriste topologiju lbest s 3 susjeda, što ima smisla jer moraju provjeravati najmanji broj susjeda.

Tablica 5: Vrijeme izvršavanja 1000 iteracija i medijan iteracija do prihvatljivog rješenja

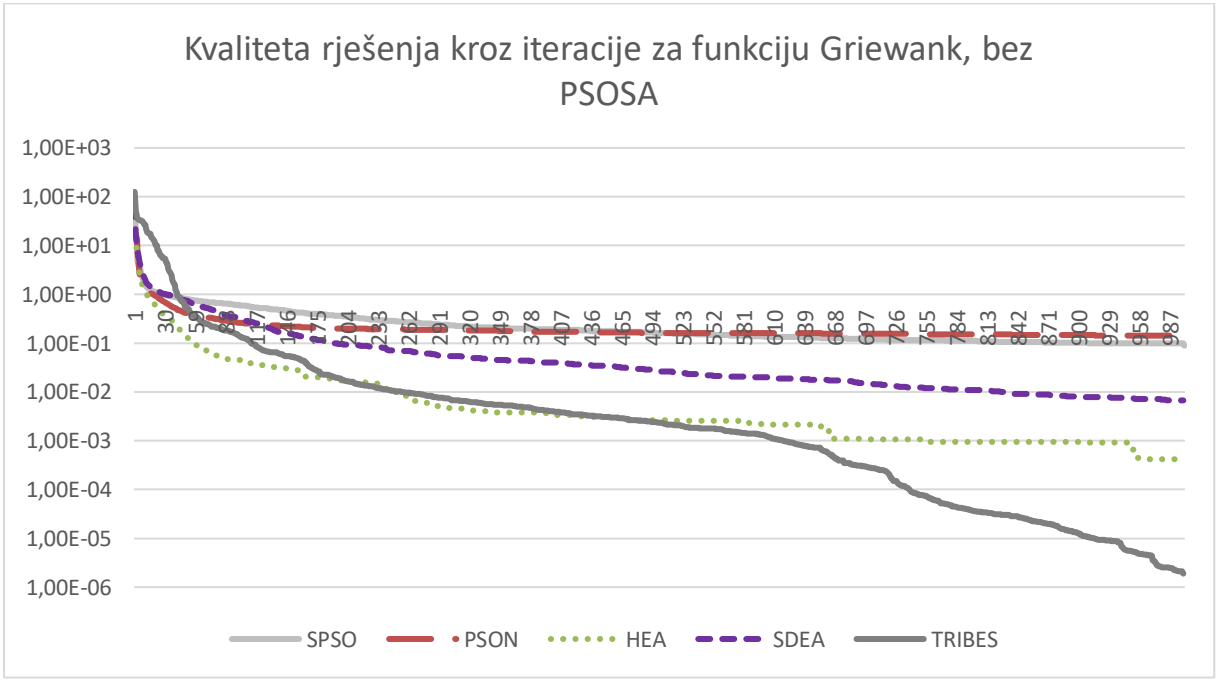
Algoritam	Topologija	Medijan vrijeme nakon 1000 iteracija, u sekundama	Medijan iteracija do prihvatljivog rješenja
PSOSA	5	5,90129	16,5
PSOSA	3	5,651298	17,5
PSOSA	Neumann	5,805246	18,5
HEA	3	4,364501	62
HEA	5	4,417035	87
HEA	Neumann	4,404753	88
PSON	Neumann	2,048978	205
PSON	5	2,05647	311
PSON	3	1,787064	467
TRIBES		51,70308	1047,5
SPSO	Neumann	1,64459	5761
SPSO	5	1,62205	5769,5
SDEA	5	2,894517	5916
SDEA	Neumann	2,894274	5965,5
SPSO	3	1,615537	5987
SDEA	3	2,885499	6293,5

Na slici 6 je vidljiv prosječni napredak algoritama kroz 1000 iteracija nad funkcijom Griewank. U graf nije uključen PSO algoritam. Skala grafa je logaritamska. Na grafu je jasno vidljivo kako algoritam PSOSA započne s najgorim rješenjem, no jako brzo dođe do jako dobrog rješenja. Vrlo brzo, Python je rješenje zaokružio na 0, što je na grafu vidljivo kao prestanak krivulje PSOSA, jer je na logaritamskoj skali nemoguće prikazati vrijednost 0.



Slika 6: Kvaliteta rješenja kroz iteracije za funkciju Griewank

Kako bi ostali algoritmi bili bolje vidljivi, sa grafa sam maknuo PSOSA algoritam. Novi graf vidljiv je na slici 7. SPSO i PSON algoritmi se dosta asimptotički približavaju vrijednosti 0,1. SDEA cijelo vrijeme napreduje prema sve boljem rješenju. HEA napreduje u koracima, gdje zna vrlo dugo biti na jednom rješenju te naglo, u nekoliko koraka, poboljšati rješenje. TRIBES stalno poboljšava rješenje, no ima i veća, iznenadna poboljšanja.



Slika 7: Kvaliteta rješenja kroz iteracije za funkciju Griewank. Algoritam PSOSA je maknut radi čitljivosti ostatka grafa

Na kraju sam u tablici 6 prikazao najbolji algoritam za svaku funkciju koju sam optimizirao u ovom radu. Za svaku funkciju sam izdvojio dva najbolja algoritma. Za većinu funkcija vidimo da je algoritam PSOSA algoritam ili HEA algoritam ili najbolji ili drugi najbolji algoritam. Iznimke je Griewank funkcija, kod koje je TRIBES drugi najbolji algoritam, i Sphere funkcija, kod koje je drugi najbolji algoritam PSON.

Tablica 6: Najbolji algoritmi za pojedine funkcije

Funkcija	Algoritam	Prosječno rješenje
Griewank	PSOSA	0
Griewank	TRIBES	0,000898
Rastrigin	PSOSA	3,29E-12
Rastrigin	HEA	0,265922
Rosenbrock	PSOSA	27,88015
Rosenbrock	HEA	29,71445
Sphere	PSOSA	1,06E-13
Sphere	PSON	2,66E-05
Zakharov	HEA	0,093701
Zakharov	PSOSA	2,955044

10. Zaključak

Rezultati testiranja pokazuju da je originalni algoritam optimizacije rojem čestica lošiji od ostalih varijanti algoritama te je očito zašto je brzo zamijenjen standardnim PSO algoritmom. Ako gledamo samo rezultate algoritama na kraju izvršavanja, najbolji izbor su algoritmi PSOSA, HEA i PSON, dok prihvatljiva rješenja daje i algoritam TRIBES. Ako pogledamo brzinu izvršavanja ovih algoritama, algoritmi PSON i HEA su jako brzi, PSOSA je malo sporiji, no još uvijek dosta brz, dok je TRIBES izrazito spor. Ako još razmotrimo prosječni broj iteracija do dobrog rješenja, ističu se algoritmi PSOSA, čiji je medijan broja iteracija do prihvatljivog rješenja manji od 20, i HEA, čiji je isti medijan manji od 100.

Algoritam optimizacije rojem čestica je relativno novi algoritam. Primijenjen je uspješno u treniranju neuronskih mreža, optimizaciji kontrole napona u visokonaponskim energetskim sustavima, aproksimaciji razine napona baterija, optimizaciji sastava hranjivih tvari za kulture mikroorganizama u laboratoriju [26], planiranje voznog reda vlakova te planiranje puta robota u zatrpanom prostoru [27]. Standardni algoritam je, kroz vrijeme, poboljšan i kombiniran s drugim algoritmima, kako bi se riješio problem prerane konvergencije na lokalni minimum.

Popis literature

[1] D. Simon, *Evolutionary optimization algorithms: biologically-Inspired and population-based approaches to computer intelligence*. Hoboken, New Jersey: John Wiley & Sons Inc., 2013. Accessed: Jul. 17, 2021. [Online]. Available: http://site.ebrary.com/id/10722521

[2] "Welcome to Python.org," *Python.org*. https://www.python.org/about/ (accessed Aug. 01, 2021).

[3] "What is NumPy? — NumPy v1.21 Manual." https://numpy.org/doc/stable/user/whatisnumpy.html (accessed Aug. 01, 2021).

[4] "Matplotlib: Python plotting — Matplotlib 3.4.2 documentation." https://matplotlib.org/ (accessed Aug. 01, 2021).

[5] "Visual Studio Code Frequently Asked Questions." https://code.visualstudio.com/docs/supporting/faq (accessed Aug. 01, 2021).

[6] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Nov. 1995, vol. 4, pp. 1942–1948 vol.4. doi: 10.1109/ICNN.1995.488968.

[7] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model." https://scholar.google.com/citations?view_op=view_citation&hl=en&user=PJm3IXAAAAAJ&citation_for_view=PJm3IXAAAAAJ:u5HHmVD_uO8C (accessed Jul. 18, 2021).

[8] F. Heppner and U. Grenander, "A Stochastic Nonlinear Model for Coordinate Bird Flocks," 1990.

[9] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Oct. 1995, pp. 39–43. doi: 10.1109/MHS.1995.494215.

[10] Y. Shi and R. C. Eberhart, "Parameter selection in particle swarm optimization," in *Evolutionary Programming VII*, Berlin, Heidelberg, 1998, pp. 591–600. doi: 10.1007/BFb0040810.

[11] P. N. Suganthan, "Particle swarm optimiser with neighbourhood operator," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Jul. 1999, vol. 3, pp. 1958-1962 Vol. 3. doi: 10.1109/CEC.1999.785514.

[12] S. Sengupta, S. Basak, and R. A. Peters, "Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives," *Machine Learning and Knowledge Extraction*, vol. 1, no. 1, Art. no. 1, Mar. 2019, doi: 10.3390/make1010010.

[13] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, Feb. 2002, doi: 10.1109/4235.985692.

[14] J. Kennedy, "Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Jul. 1999, vol. 3, pp. 1931-1938 Vol. 3. doi: 10.1109/CEC.1999.785509.

[15] J. Kennedy and R. Mendes, "Population structure and particle swarm performance," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, May 2002, vol. 2, pp. 1671–1676 vol.2. doi: 10.1109/CEC.2002.1004493.

[16] R. Mendes, J. Kennedy, and J. Neves, "Watch thy neighbor or how the swarm can learn from its environment," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*, Apr. 2003, pp. 88–94. doi: 10.1109/SIS.2003.1202252.

[17] O. Kramer, *Genetic Algorithm Essentials*. Springer International Publishing, 2017. doi: 10.1007/978-3-319-52156-5.
[18] B. Yang, Y. Chen, and Z. Zhao, "A Hybrid Evolutionary Algorithm by Combination of PSO and GA for Unconstrained and Constrained Optimization Problems," in *2007 IEEE International Conference on Control and Automation*, May 2007, pp. 166–170. doi: 10.1109/ICCA.2007.4376340.
[19] R. Storn and K. Price, "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec. 1997, doi: 10.1023/A:1008202821328.
[20] T. Hendtlass, "A Combined Swarm Differential Evolution Algorithm for Optimization Problems," in *Engineering of Intelligent Systems*, Berlin, Heidelberg, 2001, pp. 11–18. doi: 10.1007/3-540-45517-5_2.
[21] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science (New York, N.Y.)*, vol. 220, pp. 671–80, Jun. 1983, doi: 10.1126/science.220.4598.671.
[22] G. Yang, D. Chen, and G. Zhou, "A New Hybrid Algorithm of Particle Swarm Optimization," in *Computational Intelligence and Bioinformatics*, Berlin, Heidelberg, 2006, pp. 50–60. doi: 10.1007/11816102_6.
[23] M. Clerc, *Particle Swarm Optimization*. John Wiley & Sons, 2010.
[24] "Optimization Test Functions and Datasets." https://www.sfu.ca/~ssurjano/optimization.html (accessed Aug. 25, 2021).
[25] N. Ivkovic, D. Jakobovic, M. Golub, "Measuring Performance of Optimization Algorithms in Evolutionary Computation," IJMLC, vol. 6, no. 3, pp. 167–171, Jun. 2016, doi: 10.18178/ijmlc.2016.6.3.593.
[26] Eberhart and Y. Shi, "Particle swarm optimization: developments, applications and resources," in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, May 2001, vol. 1, pp. 81–86 vol. 1. doi: 10.1109/CEC.2001.934374.
[27] N. K. Jain, U. Nangia, and J. Jain, "A Review of Particle Swarm Optimization," *J. Inst. Eng. India Ser. B*, vol. 99, no. 4, pp. 407–411, Aug. 2018, doi: 10.1007/s40031-018-0323-y.

Popis slika

Popis tablica