Optimizarea functiei lui Ackley

Studenti: *Hosu Razvan, Nistor Dorin, Pop Alex, Pop Cristian, Vint Alexandru*

Ianuarie 2025

Cuprins

1 Introducere

Funcția lui Ackley este una dintre cele mai cunoscute funcții de testare utilizate pentru diverse optimizări. Aceasta face parte din clasa funcțiilor multimodale, caracterizându-se prin prezența unui număr mare de minime locale, ceea ce o face deosebit de potrivită pentru testarea algoritmilor de optimizare globală. Forma sa matematică este definită astfel:

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

unde n reprezintă dimensiunea vectorului x, iar a, b și c sunt constante ale funcției, în proiect având valorile $a = 20$, $b = 0.2$ și $c = 2\pi$ [1]. Obiectivul principal în optimizarea acestei funcții este identificarea punctului de minim global, situat în jurul valorii 0, unde f(x) și toate componentele vectorului x sunt egale cu zero.

Am ales acest proiect într-un mod aleatoriu, punând întreaga listă de proiecte pe site-ul *random.org*. Astfel că primul rezultat a fost titlul acestui proiect. Căutând mai multe detalii despre ce reprezintă funcția lui Ackley, respectiv optimizarea acesteia, ne-am gândit să alegem să optimizăm această funcție, căutând minimul acesteia.

Algoritmul este scris în limbajul *Python*, deoarece este un limbaj ușor de folosit, cu o sintaxă apropiată de pseudocod, astfel că citirea codului este mai ușoară. Totodată, dispune de diverse module pentru a genera grafice sau a calcula diferite expresii matematice într-un mod foarte ușor.

2 Analiza și viziunea proiectului

2.1 Contextul Aplicației

Scopul proiectului este de a aproxima punctul de minim pentru funcția lui Ackley. Așadar, folosind algoritmi genetici, ne dorim să obținem cel mai bun rezultat posibil pentru funcția dată pe n dimensiuni. Având în vedere că problema noastră nu poate fi rezolvată cu un simplu algoritm de backtracking, am dezvoltat o aplicație ce aplică algoritmii. Pentru a ajunge cât mai aproape de ceea ce ne-am setat ca rezultat final, am realizat o hibridizare a acestui algoritm de optimizare, datorită căreia se pot apropia rezultatele cât mai mult de cel mai ideal rezultat, acesta fiind 0.

2.2 Viziunea finala a programului

Pentru a proiecta acest algoritm, trebuie să avem în vedere următoarele etape ale unui algoritm genetic. În primul rând, este esențial de definit funcția de fitness, care este însăși funcția lui Ackley, așa cum a fost definită în introducere, dar și mai jos:

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

După ce am definit funcția de fitness, pentru a putea aplica algoritmul genetic pentru optimizarea funcției lui Ackley avem nevoie să generăm o populație inițială, pentru care aplicăm operatorul de selecție a părinților. Părinții vor fi selectați prin metoda *turnir*. Părinților li se va aplica operatorul de combinare, rezultând doi copii. Pe copii se va aplica operatorul de mutație, iar după aceea se va aplica atât pe părinți, cât și pe copii operatorul de supraviețuire. În acest pas se va selecta cel mai bun individ, respectiv părinții vor fi eliminați, iar copiii vor deveni noii părinți.

3 Aspecte teoretice

În scopul înțelegerii complete a ideii proiectului, vom clarifica ce reprezintă anumiți termeni-cheie din această documentație, aceștia fiind algoritmii genetici.

Un algoritm genetic (GA) este un algoritm de optimizare inspirat din procesele evoluționiste din natură, incluzând selecția naturală, precum și termenul de „generație" între indivizi, mutația, care schimbă aleatoriu caracteristici ale unor indivizi din anumite generații, hibridizarea, care îmbunătățește în mod specific anumiți indivizi conform unui tipar prestabilit, și crossover-ul, prin care indivizii selectați prin procesul de selecție ajung să creeze o nouă generație, care va fi descendentă a celei curente. Se face distincția între mutația aleatorie și hibridizarea, care este determinată de anumiți factori[2].

Acest proces se repetă cu un număr prestabilit de generații, o rată anume a mutației și o metodă concretă de selecție, scopul fiind de a obține cel mai bun rezultat posibil, aproximând soluția problemei[3].

În aplicarea unui algoritm genetic, trebuie să ținem cont de anumiți factori care determină necesitatea utilizării acestor algoritmi, tipul de problemă pentru care dorim să efectuăm optimizarea folosind GA, iar scopul optimizării poate varia de la o funcție la alta, sau poate fi determinat de anumite cerințe și necesități prestabilite. Exemplul din cazul nostru, „Funcția lui Ackley", presupune că prin optimizarea acestei funcții folosind algoritmi genetici dorim să îmbunătățim rezultatul funcției (fitness-ul funcției), astfel încât să ajungem la un rezultat cât mai potrivit în aflarea punctului de minim global.[1]

La începutul generării unui scenariu, se presupune că se va crea o generație inițială, care va trece prin niște procese pentru a obține setul de date dorit. Prin populația inițială înțelegem faptul că aceasta reprezintă un set de soluții candidate generate la începutul unui proces evolutiv; prin urmare, ceea ce va urma va sta la baza acestei generații inițiale[4].

Conceptul de hibridizare dorește să apară în programe cu un scop bine stabilit, acesta fiind de a îmbunătăți performanța algoritmului. Prin acest concept, putem înțelege multiple metode de hibridizare, acestea fiind combinarea metodelor evolutive (ex. algoritmi genetici, strategii evolutive, programare genetică) și integrarea tehnicilor clasice (optimizarea locală sau metode deterministe) cu cele evolutive.[6]

Funcția de fitness definește criteriul pentru clasificarea și selecția ipotezelor potențiale, care vor fi incluse probabilistic în generația următoare a populației. În cazul sarcinilor de învățare a regulilor de clasificare, funcția de fitness include, de obicei, un component care evaluează acuratețea clasificării regulii pe baza unui set de exemple de generații furnizate anterior.[7]

Menținerea diversității în cadrul mediului soluțiilor este importantă deoarece aceasta poate să arunce traiectoria evoluției algoritmului în părți extreme în mod rapid. De unde reies tehnici care sunt utilizate pentru maximizarea diversității soluțiilor inițiale, astfel încât algoritmul să nu fie atras de minime locale. Această strategie este extrem de valoroasă pentru funcțiile cu multiple extreme, cum este Ackley, unde o ex-

plorare insuficientă poate duce la obținerea unor soluții suboptimale. Algoritmii genetici pot încorpora aceste principii prin ajustarea operatorilor evolutivi, cum ar fi ratele de mutație și selecție bazată pe fitness.[5]

Funcția lui Ackley este o funcție de testare bine cunoscută pentru optimizarea globală și este utilizată pentru a compara și testa algoritmi, precum algoritmi genetici, algoritmi evolutivi, roiurile de particule. datoriă multitudinii de minime locale. Este răspândită peste multe domenii cum ar fi cercetare, robotică sau învățare automată.[8]

Optimizarea funcției Ackley poate fi abordată prin adoptarea diverselor modele biologice inspirate de natură. Algoritmii care integrează comportamente precum cele de roi, selecția naturală și adaptarea ecologică contribuie la o explorare mai eficientă a spațiului soluțiilor. De exemplu, modelele bazate pe comportamentul de roi utilizează interacțiunea dintre indivizi pentru a optimiza simultan explorarea globală și exploatarea locală. În paralel, selecția naturală asigură transmiterea caracteristicilor favorabile către generațiile următoare, în timp ce mecanismele ecologice promovează diversitatea și adaptarea dinamică la peisaje multimodale, precum cel al funcției Ackley. Această combinație diversificată de strategii biologice oferă un cadru robust pentru depășirea minimele locale și găsirea unor soluții globale optime.[9]

Influența dimensionalității problemei asupra performanței algoritmilor evolutivi, evidențiază faptul că, pe măsură ce crește numărul de variabile, explorarea spațiului soluțiilor devine semnificativ mai dificilă. Pentru funcții complexe, precum Ackley, această provocare poate fi gestionată prin utilizarea unor operatori specializați, care ajustează mutațiile pentru a prioritiza direcțiile promițătoare, reducând timpul de convergență.P roiecțiile sau transformările spațiului de căutare,simplifică problema și a optimizează performanța algoritmului în privința aceasta. Aceste abordări permit algoritmilor genetici să se adapteze mai eficient la probleme complexe.[1]

Hill Climbing este utilizat pentru rafinarea algoritmilor genetici, mai exact soluțiile generate în mod evolutiv, accelerând convergența și reducând riscul de stagnare în soluții suboptimale. Această metodă hibridă îmbunătățește precizia și eficiența optimizării, oferind un cadru robust aplicabil în diverse probleme complexe, care de altfel presupune mutații succesive, pentru a îmbunătăți indivizii selectați în fiecare generație a algoritmului genetic.[10]

4 Implementarea proiectului

4.1 Clasa Specimen

Clasa *Specimen* stochează și prelucrează informații despre fiecare individ. Această clasă permite calcularea și manipularea fitness-ului indivizilor, folosind funcția lui Ackley.

La instanțierea unui specimen, următoarele date sunt stocate: dimensiunea problemei, cromozomii, respectiv fitnessul specimenului - care este calculat la instanțierea clasei. Metodele acestei clase sunt descrise în tabelul de mai jos:

Metodă	**Descriere**
`get_chromosome(self)`	Returnează o copie a cromozonului curent
`set_chromosome(self, chromosome: list)`	Actualizează cromozonul și recalculează fitness-ul acestuia.
`get_fitness(self)`	Returnează valoarea fitness-ului a specimenului.
`__calculate_fitness(self)`	Calculează fitness-ul cu funcția lui Ackley.
`__exp(self, expression)`	Metodă care calculează e^x.

Tabela 1: Metodele clasei `Specimen`

4.2 Clasa Generation

Clasa *Generation* reprezintă o listă de specimene ce fac parte dintr-o generație. Clasa permite următoarele operații: adăugare specimen nou, ștergere specimen sau actualizare specimen.

La instanțierea unei noi generații, se creează fie o listă goală, fie se poate crea o copie a unei liste de generații deja existente. Metodele acestei clase sunt descrise în tabelul următor:

Metodă	**Descriere**
`__init__(generation)`	Inițializează un obiect Generation.
`append(specimen)`	Adaugă un obiect Specimen la o listă Generation cu obiecte Specimen.
`get_generation()`	Returnează o listă curentă de obiecte de tip Specimen.
`get_fitness(self)`	Returnează valoarea fitness-ului a specimenului.
`get_length()`	Returnează numărul total de obiecte Specimen.
`update_generation( specimen_list)`	Înlocuiește o listă curentă cu o nouă listă de obiecte de tip Specimen.
`remove(specimen)`	Șterge un obiect Specimen din lista Generation

Tabela 2: Metodele clasei `Generation`

4.3 Clasa Algoritmi Genetici

Clasa GA este cea care rulează algoritmul genetic. Aici se fac inițializările, se aplică operatorii genetici, se caută minimul global, respectiv se generează populația inițială. Metodele acestei clase se regăsesc în tabelul de mai jos:

Metodă	**Descriere**
`__init__(self, dimension, initial_population, mutation_rate)`	Inițializează obiectul GA cu dimensiunea prestabilită și rata de mutație dorită
`run(self)`	Rulează programul până în punctul în care se ajunge la un număr de sub 4 indivizi
`local_search(self, children: Generation)`	Optimizează cromozomii indivizilor printr-o căutare locală (hill climbing)
`selection(self, specimen_list: list)`	Selectează părinții folosind metoda turnirului
`crossover(self, parents: list['Specimen'])`	Creează copii prin combinarea cromozomilor părinților
`mutation(self, children: Generation)`	Modifică cromozomii copiilor prin adăugarea unei valori aleatoare
`survival(self, children_generation: Generation)`	Decide care indivizi supraviețuiesc și actualizează generația curentă
`get_best_specimen(self)`	Returnează cel mai bun specimen identificat până în acel moment
`get_best_fitnesses_list(self)`	Returnează lista cu cele mai bune fitness-uri pentru fiecare generație
`__generate_population(self)`	Generează cromozomi aleatori pentru fiecare specimen și creează un obiect Specimen pentru fiecare cromozom și îl adaugă generației
`__generate_chromosomes(self)`	Generează un cromozom aleator în domeniul funcției Ackley
`__get_worst_specimen( self, specimen_list: list['Specimen'])`	Sortează indivizii după fitness și returnează pe cel cu fitness-ul maxim

Tabela 3: Metodele clasei `Algoritmi Genetici`

4.4 Clasa Plot

Clasa Plot este o componentă care se ocupă cu reprezentarea grafică a unui set de valori sub forma unui grafic. Graficul reprezintă toate minimele găsite în fiecare generație în parte. Totodată, soluția problemei este evidențiată cu un punct roșu.

La instanțierea acestei clase, se creează o listă de valori sau se copiază una existentă deja. Totodată, se caută soluția minimă din listă, dacă nu este goală. Metodele acestei clase sunt descrise în tabelul următor:

Metodă	**Descriere**
`set_values(self, values: list)`	Setează o listă de valori pentru reprezentarea grafică
`plot(self)`	Generează graficul pe baza listei de valori.
`__calculate_min_value(self)`	Calculează cea mai mică valoare din listă, respectiv indexul unde se află.

Tabela 4: Metodele clasei `Plot`

4.5 Diagrama claselor

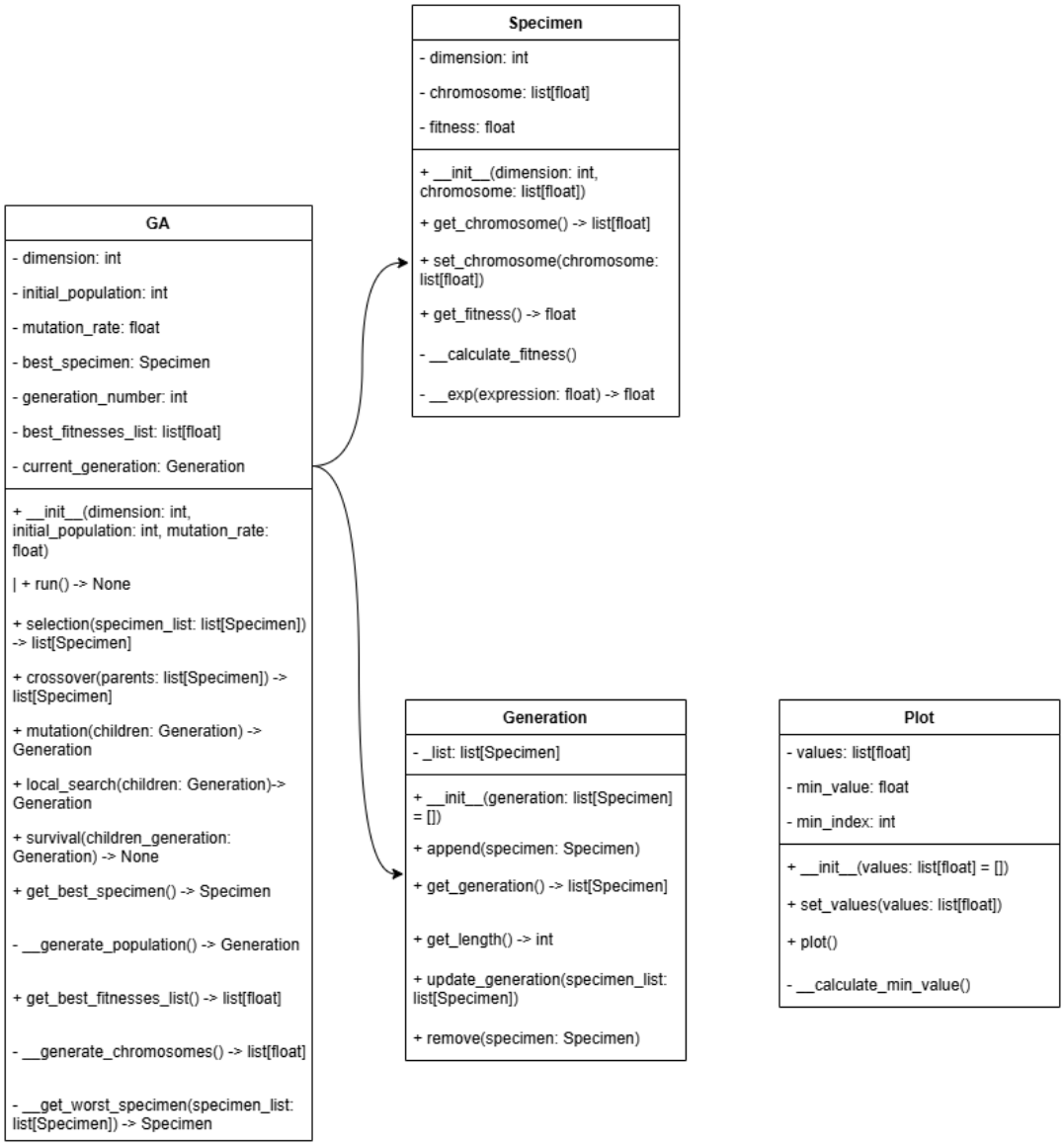

Figura 1: Diagrama de clasa completa

4.6 Descrierea algoritmului

Pentru implementarea algoritmului genetic al proiectului nostru, vor fi definite câteva clase, care au rolul de a stoca și prelucra datele. Următoarele clase vor fi prezente în proiect: *Specimen*, *Generation*, *Algoritm Genetic (GA)* si *Plot*.

Diagrama algoritmului genetic oferă o ilustrare clară a procesului de desfășurare a evenimentelor pentru găsirea soluției optime. Aceasta evidențiază mai multe etape esențiale pentru optimizarea soluțiilor într-un spațiu de căutare. Fiecare algoritm genetic include etape distincte, fiecare cu funcționalități și scopuri bine definite.

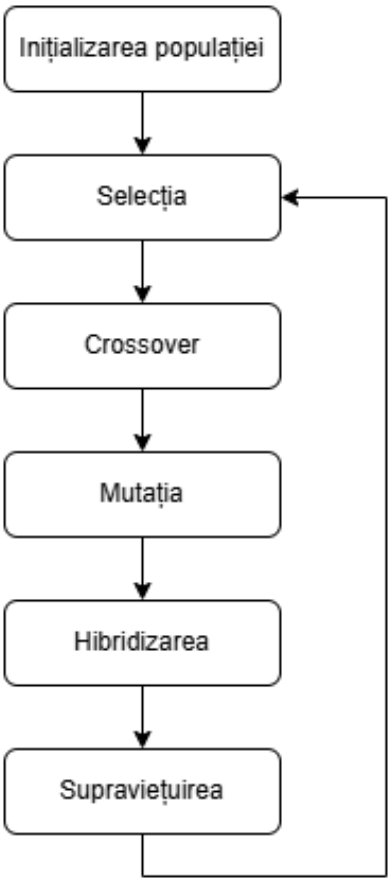

Figura 2: Schema algoritmului genetic

După cum se poate vedea și în figura de mai sus, prima etapă a algoritmului genetic este de a genera o populație inițială. Dimensiunea populației inițiale este dată de atributul *initial_population* din constructorul clasei *GA*, iar specimenele sunt generate aleatoriu. În funcție de dimensiunea funcției lui Ackley, de pildă *N*, fiecare specimen va avea cromozomi, având *N* gene.

După ce avem populația inițială, vom trece la următoarea etapă, și anume *Selecția*. În această etapă se selectează părinții care vor lua parte la procesul de combinare. Selecția se realizează sub formă de turnir, adică vor fi luați câte 2 specimeni, iar cel mai bun specimen, adică cel care are fitness-ul mai mic, va fi considerat câștigător și va fi adăugat

într-o listă de părinți. La final, vom avea o listă de părinți, care va fi aproximativ jumătate din populația care a intrat în procesul de selecție.

Pentru crossover, va fi luată în considerare lista de părinți obtinută din etapa anterioară. În acest pas, se iau câte 2 părinți și se aplică operația genetică crossover. Aici se generează "*copiii*" (2 la număr) în felul următor: se consideră un punct de tăiere aleatoriu pe intervalul $[1; N]$, daca dimensiunea $N > 1$, respectiv $[0.01, 0.99]$ în caz contrar, iar cromozomii părinților vor fi tăiați în punctul de tăiere obținut. Primul copil va avea ca cromozomi prima parte din primul părinte, respectiv a doua parte din al doilea părinte. Al doilea copil va primi a doua parte din cromozomii primului părinte, respectiv prima parte din cromozomii celui de-al doilea părinte. Copiii rezultați vor fi adăugați într-o nouă generație.

În etapa de mutație, copiilor obținuți anterior li se vor aplica diverse mutații. Mai pe scurt, mutația se aplică pe genele cromozomului. Pentru fiecare cromozom în parte se generează o valoare aleatorie pe intervalul $(0, 100)$. Dacă valoarea generată este mai mică decât rata noastră de mutație, atunci pe respectiva genă se va adăuga sau scădea o valoare aleatorie între $[-0.5, 0.5]$.

Următoarea etapă a algoritmului reprezintă procesul de hibridizare, care are scopul de a îmbunătăți soluțiile locale, în cazul în care nu s-a găsit un nou minim global după cel puțin 2 generații. În algoritmul nostru, folosim căutarea locală *Hill Climbing*, căutând vecinii optimi ai fiecărui specimen.

Ultima etapă a algoritmului este supraviețuirea. În acest stadiu vom considera faptul că vechea generație (părinții) va fi scoasă din algoritm, iar noua generație (copiii) vor rămâne în populație. Totodată, aici se verifică care este cel mai bun specimen din generația nouă, iar apoi se compară cu soluția globală a problemei. Dacă cel mai bun specimen din generația curentă este mai bun decât soluția globală, atunci acesta va fi considerat noua soluție a problemei noastre.

Acest proces se repetă până când dimensiunea generației rămâne 2. Totodată, toți specimenii care au avut cel mai bun fitness din fiecare generație sunt puși într-o listă pentru a genera un Plot cu cele mai bune soluții din fiecare generație.

5 Testare si validare

5.1 Testări manuale

Pentru testarea manuală, dorim să verificăm modul în care valorile fitness-ului evoluează pe generații, funcționalitatea algoritmului genetic și totodată să testăm dacă evoluția specimenilor se îndreaptă către soluția optimă căutată, astfel aceasta trebuie să fie cât mai apropiată de valoarea 0.

Printre testele efectuate se numără:

- Algoritmul genetic evoluează corect populația prin selecție, crossover, mutație și supraviețuire.
- Valorile fitness-ului sunt calculate corect pentru fiecare specimen în parte, folosind funcția lui Ackley.
- Mutația aplicată asupra cromozomilor respectă probabilitatea mutației folosite.
- Cel mai bun fitness al fiecărei generații este corect identificată.
- Datele de performanta, respectiv numărul generației și fitness-ul cel mai bun al generației respective sunt corect salvate în fișierul JSON.
- Graficul generat afișează cu exactitate evoluția fitness-ului pe timpul generațiilor
- Algoritmul rulează în condiții normale chiar și pentru populații mari.

5.2 Unit Testing

5.2.1 Testarea Modulului Specimen

În cadrul modulului Specimen există teste care trebuie îndeplinite pentru a confirma funcționalitatea corectă a clasei cu același nume. Aceste Unit Test uri sunt esențiale pentru a verifica dacă modulul furnizează date corecte și pentru a identifica eventualele omisiuni în proiectarea acestuia.

Următoarele teste au fost concepute pentru a asigura conformitatea cu cerințele proiectului, următoarele teste fiind:

- Test pentru calculul fitness-ului, obținerea cromozomului;
- Test pentru lungimea invalidă a cromozomului, dimensiune invalidă;
- Test pentru recalcularea fitness-ului la setarea cromozomului;
- Testare pentru dimensiuni mai mari față de cele standard;
- Testare pentru valori extrem de mari la cronozomi;
- Testare pentru precizia fitness-ului și performanță generarii ;

5.2.2 Testarea Modulului Generation

Rolul acestui test este de a verifica funcționalitatea modulului de Generation, precum și metodele din cadrul acestuia, astfel încât să fie testate următoarele:

- Crearea instanțelor de Specimen, pentru crearea ulterioară a generației;
- Crearea unei instanțe de Generation;
- Adăugarea de specimene într-o generație folosind metoda *append()*;
- Obținerea numărului de specimene din generație *metoda get_length()*;
- Accesarea listei de specimene *metoda get_generation()*;
- Verificarea actualizării corecte a stării generației după modificări;

5.2.3 Testarea Modulului Algoritmi Genetici

Rolul acestui test este de a verifica funcționalitatea modulului de Algoritmi Genetici, precum și metodele din cadrul acestuia, astfel încât să fie testate următoarele:

- Selecția părinților prim metoda Tournament selection;
- Rezultatul Crossover ului pentru selectia pe o dimensiune și N dimensiuni;
- Validarea valorii returnate de metoda de mutație;
- Supraviețuirea generației cu un număr par și impar de specimene;
- Păstrarea celui mai bun specimen în metoda *survival()*

6 Rezultate

Pentru prima testare vom folosi următoarele setări pentru algoritmul genetic:

- DIMENSION = 2 - cromozomii vor avea doua elemente (ex. [x1, x2])
- INITIAL_POPULATION = 5000 - la faza incipientă, vom avea o populatie de 5000 de specimeni
- MUTATION_RATE = 0.25 - ceea ce inseamnă că fiecare valoare din cromozom are o probabilitate de 25% ca aceasta să se schimbe în timpul mutației.

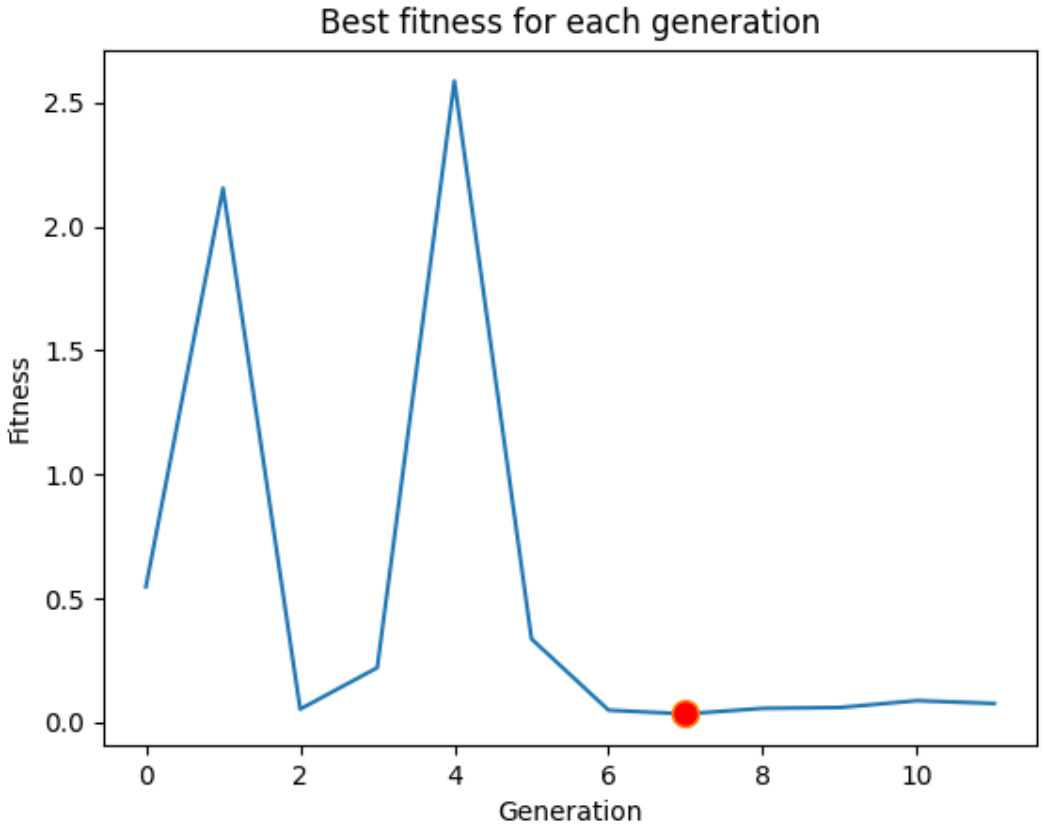

Figura 3: Rezultat grafic

Graficul conține numeroase componente precum:

- o axa X pentru reprezentarea generațiilor
- o axă Y pentru reprezentarea fitness-ului cel mai bun din fiecare generație
- o bulină roșie care indică fitness-ul cel mai bun pe parcursur tuturor generațiilor

```
[Generation 0] New best specimen: [0.06268261135845421, -0.0803415959176945 2] with fitness: 0.546417682821247
[Generation 2] New best specimen: [-0.015989119030642332, -0.002754661867590 1025] with fitness: 0.05288508251181101
[Generation 6] New best specimen: [-0.005383556981424703, -0.014098575654640066] with fitness: 0.04873942352376348
[Generation 7] New best specimen: [0.005937030503871718, -0.009049562231661284] with fitness: 0.03372961515347983
Best specimen: [0.005937030503871718, -0.009049562231661284] with fitness 0.03372961515347983
Elapsed time: 0.1042142000515014 seconds
```

Figura 4: Output consolă

Conform output-ului din consolă, se poate observa faptul că fitness-ul generației 0 a fost cel mai mare, însă treptat acesta a scăzut până la generația 7, unde s-a descoperit cel mai bun fitness dintre toate generațiile.

De asemenea timpul de rulare este de aproximativ 0.1 secunde, semn că algoritmul este eficient.

Pentru a doua testare vom folosi următoarele setări pentru algoritmul genetic:

- DIMENSION = 4 - cromozomii vor avea patru elemente (ex. [x1, x2, x3, x4])
- INITIAL_POPULATION = 25000 - la faza incipientă, vom avea o populatie de 25000 de specimeni
- MUTATION_RATE = 0.25 - ceea ce inseamnă că fiecare valoare din cromozom are o probabilitate de 25% ca aceasta să se schimbe în timpul mutației.

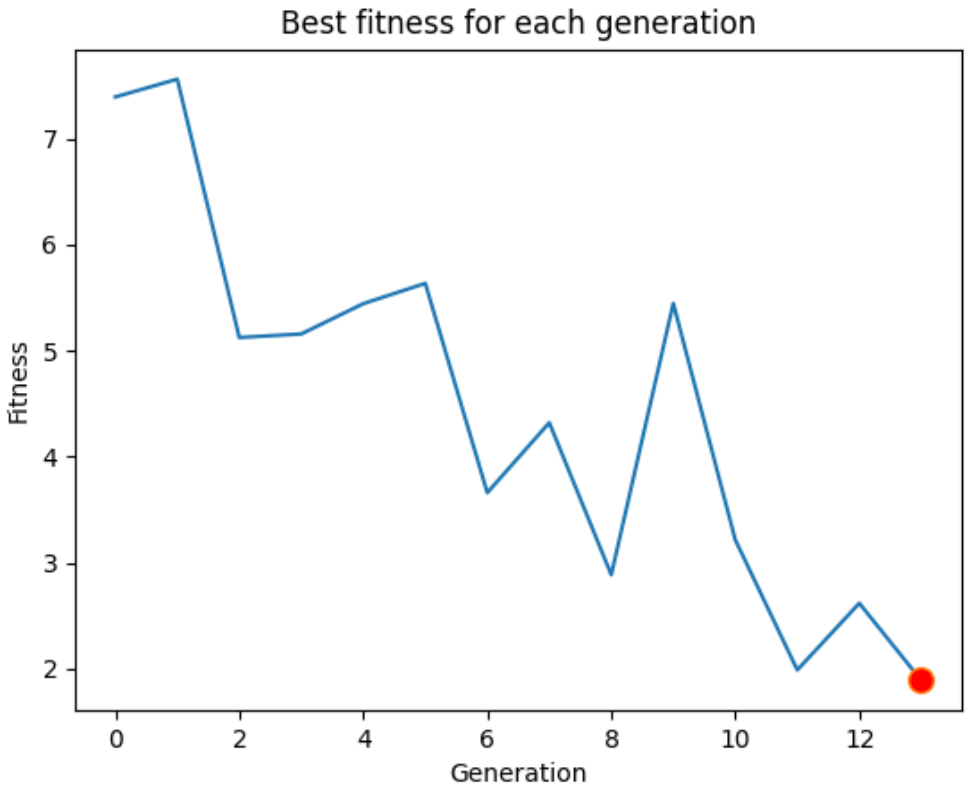

Figura 5: Rezultat grafic

Printr-o comparație simplă între cele doua grafice, se poate observa faptul că, diferența dintre fitness-ul generației 0 și fitness-ul generației urmatoare din output se mărește datorită seturilor mari de date introduse.

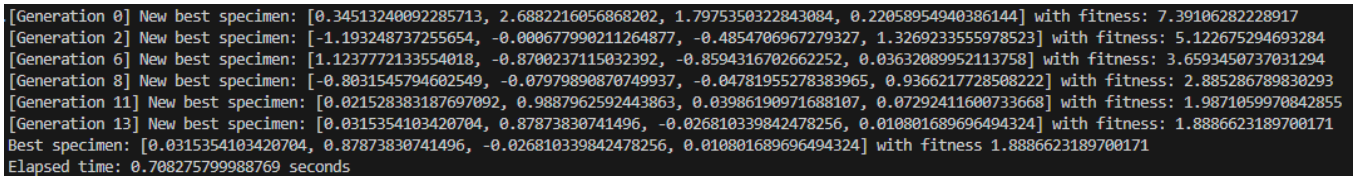

Figura 6: Output consolă

O dimensiune mai mare a cromozomilor crește complexitatea funcției, ceea ce înseamnă că durata timpului de execuție va crește considerabil. Ca o comparație între cele doua rezultate, la primul rezultat s-au folosit cromozomi cu 2 componente, unde a rezultat un timp de execuție foarte bun de aproximativ 0.1 secunde, în timp ce al doilea rezultat, unde s-au folosit cromozomi cu 4 componenete a obtinut aproximativ 0.7 secunde la timpul de execuție.

O altă remarcă este la nivelul rezultatului fitness-ului cel mai bun. Cu cât dimensiunea cromozomilor crește, cu atâta și valoarea celui mai bun fitness din toate generațiile, se va îndepărta considerabil de 0, drept dovadă că în prima execuție unde am avut dimensiunea cromozomului egală cu 2, am valoarea de aproximativ 0.03 la cel mai bun fitness din toate generațiile, în timp ce la a doua execuție unde am avut dimensiunea cromozomilor egală cu 4, am obținut valoarea de aproximativ 1.88 la cel mai bun fitness din toate generațiile.

7 Concluzii

Proiectul privind Optimizarea funcției lui Ackley a fost realizat ca un demers practic și aplicat pentru aprofundarea conceptelor de optimizare globală, utilizând algoritmi genetici și algoritmi evolutivi. Acest proiect ne-a oferit, ca echipă, oportunitatea de a ne dezvolta competențele tehnice într-un mod diferit, implicând atât proiectarea unei aplicații funcționale, cât și elaborarea unei documentații detaliate. Astfel, am reușit să abordăm și să implementăm soluții complexe în domeniul algoritmilor inspirați biologic, demonstrând o înțelegere solidă și o aplicare practică a conceptelor studiate.

Deși aplicația îndeplinește cu succes obiectivul principal, există câteva direcții prin care ar putea fi îmbunătățită:

- Extinderea reprezentărilor grafice pentru o analiză mai detaliată a performanțelor algoritmului.
- Implementarea unor metode alternative de hibridizare pentru o optimizare mai rapidă.

Procesul de dezvoltare nu a fost lipsit de dificultăți, gestionarea complexității algoritmilor genetici, implementarea unei funcționalități robuste și validarea rezultatelor au fost provocări ce ne-au testat abilitățile. Totuși, aceste obstacole au contribuit semnificativ la consolidarea cunoștințelor noastre. Mulțumim pentru această oportunitate de învățare și explorare academică.

8 Referințe

[1] Thomas Bäck *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, 1996

[2] Tom V. Mathew, *Genetic Algorithm*, Indian Institute of Technology Bombay, 2012.

[3] Stephanie Forrest *Genetic Algorithms*, University of New Mexico, Albuquerque, 1996.

[4] Silja Meyer-Nieberg and Hans-Georg Beyer *Self-Adaptation in Evolutionary Algorithms*, 2007

[5] Xiang Zhao ,Yuan Yao, Liping Yan, *Learning algorithm for multimodal optimization*, ISSN 0898-1221, 2009

[6] Tarek A. El-Mihoub, Adrian A. Hopgood, Lars Nolle, Alan Battersby, *Hybrid Genetic Algorithms: A Review*, School of Computing and Informatics, Nottingham Trent University ,2006

[7] L. Haldurai, T. Madhubala, R. Rajalakshmi, *A Study on Genetic Algorithm and its Applications*, ISSN: 2347-2693, 2016.

[8] Sudhir G. Akojwar, Pravin R. Kshirsagar, *Performance Evolution of Optimization Techniques for Mathematical Benchmark Functions*, International Journal of Computers, ISSN: 2367-8895, 2016

[9] Jhen-Jai Hu, Yu-Te Su, Tzuu-Hseng S. Li, *A novel ecological-biological-behavior praticle swarm optimization for Ackley's function* ISSN 2324-8017, 2010.

[10] Sathiyaraj Chinnasamy, M. Ramachandran, M. Amudha, Kurinjimalar Ramu, *A Review on Hill Climbing Optimization Methodology*, ISBN 978-81-936097-6-7, 2022