OPTIMIZARE CU ROI DE PARTICULE (PSO)

Proiect în cadrul cursului de Inteligenta Artificiala,
Facultatea de Automatică și Calculatoare, Universitatea Tehnica “Gheorghe Asachi”
Iaşi

Profesor Coordonator,
Prof. Hulea Mircea

Autori,
Ghiurca Andrei Cosmin 1406B
Ciorap Silviu George 1406B

CUPRINS

1. INTRODUCERE ȘI DESCRIEREA PROBLEMELOR CONSIDERATE

1.1. Contextul general al optimizării

Inteligența Artificială modernă nu se rezumă doar la învățare automată (Machine Learning), ci include și o ramură vastă dedicată algoritmilor de căutare și optimizare. În inginerie, multe probleme reale pot fi formulate ca probleme de optimizare, unde scopul este găsirea unei configurații de parametri care minimizează sau maximizează o funcție de cost (funcție obiectiv), respectând în același timp un set de constrângeri.

În acest proiect, ne-am propus să studiem și să implementăm algoritmul Particle Swarm Optimization (PSO), o tehnică de optimizare stocastică bazată pe populații. Pentru a evita abordările triviale (precum minimizarea unor funcții matematice abstracte de tip Rastrigin sau Sphere), am ales două probleme cu aplicabilitate directă în lumea reală: navigația roboților (Pathfinding) și telecomunicațiile (Wi-Fi Planning). Aceste probleme prezintă spații de căutare continue, neliniare și cu constrângeri geometrice complexe.

1.2. Problema 1: Planificarea traseelor (Pathfinding) în mediu cu obstacole

Prima problemă abordată este găsirea unui traseu optim între un punct de start și un punct de final, într-un spațiu populat cu obstacole statice. Deși algoritmi clasici precum A* sau Dijkstra sunt eficienți pe grafuri discrete, în robotică și simulări fizice este adesea necesară operarea într-un spațiu continuu.

Definirea problemei:

- Spațiul de căutare: un plan 2D sau un volum 3D de dimensiuni fixe (ex: 100x100 unități).
- Variabile: pozițiile unui set de puncte intermediare (waypoints): P1, P2, … PN.
- Obiectiv: minimizarea distanței euclidiene totale a drumului format prin unirea punctelor Start -> P1 -> … -> End.
- Constrângeri: traseul nu trebuie să intersecteze niciun obstacol (cercuri în 2D, sfere în 3D).

Dificultatea constă în faptul că funcția de fitness nu este convexă; există numeroase minime locale create de obstacole (de exemplu, un obstacol în formă de „U” poate bloca algoritmul într-o soluție suboptimală).

1.3. Problema 2: Optimizarea acoperirii rețelelor wireless (Wi-Fi)

A doua problemă vizează plasarea optimă a emițătoarelor (routere) într-o clădire pentru a asigura o acoperire maximă a semnalului. Aceasta este o problemă de tip „Set Cover” transpusă în continuu.

Definirea problemei:

- Spațiul: o incintă delimitată de pereți (obstacole care blochează sau atenuează semnalul).
- Variabile: coordonatele X, Y sau X, Y, Z ale celor K routere disponibile.
- Obiectiv: maximizarea ariei (sau volumului) în care intensitatea semnalului depășește un prag minim.
- Funcția de fitness: se calculează prin discretizarea spațiului într-o grilă de puncte de test și numărarea punctelor „neacoperite” (care se află la o distanță mai mare decât raza de emisie față de cel mai apropiat router).

Această problemă este deosebit de dificilă deoarece „peisajul” funcției de fitness conține multe zone plate (plateaus) – mutarea ușoară a unui router s-ar putea să nu schimbe numărul de puncte acoperite, ceea ce face dificilă ghidarea particulelor.

2. ASPECTE TEORETICE PRIVIND ALGORITMUL PSO

2.1. Principiul biologic și metafora roiului

Algoritmul Particle Swarm Optimization (PSO), introdus de Kennedy și Eberhart în 1995, este inspirat din etologie, știința care studiază comportamentul animalelor. Modelul observă dinamica stolurilor de păsări sau a bancurilor de pești care se deplasează sincronizat pentru a găsi hrană sau pentru a evita prădătorii.

În PSO, fiecare soluție posibilă a problemei este modelată ca o „particulă” care zboară prin hiperspațiul soluțiilor. Particula nu este o entitate inteligentă izolată, ci face parte dintr-un sistem social. Ea își ajustează traiectoria bazându-se pe două tipuri de informații:

- Memoria proprie (experiența cognitivă): cea mai bună poziție pe care a găsit-o individul până în prezent.
- Cunoașterea colectivă (experiența socială): cea mai bună poziție găsită de vecinii săi sau de întregul roi.

2.2. Modelul matematic al actualizării particulelor

Fiecare particulă i din roi, la momentul de timp t, este caracterizată de doi vectori în spațiul D-dimensional:

- Vectorul de poziție: $xi(t) = (xi1, xi2, ..., xiD)$
- Vectorul de viteză: $vi(t) = (vi1, vi2, ..., viD)$

Ecuațiile fundamentale de actualizare sunt:

$$vi(t+1) = w * vi(t) + c1 * r1 * (pbest,i - xi(t)) + c2 * r2 * (TargetSocial - xi(t))$$
$$xi(t+1) = xi(t) + vi(t+1)$$

Unde r1, r2 sunt vectori de numere aleatoare distribuite uniform în [0, 1].

2.3. Analiza componentelor ecuației de viteză

Ecuația vitezei este o sumă ponderată a trei tendințe comportamentale, esențiale pentru echilibrul explorare–exploatare:

1. Componenta inerțială (w * vi(t))
   Simulează momentul cinetic al particulei. Îi permite să își continue deplasarea pe direcția anterioară, facilitând explorarea unor zone noi. În implementarea noastră, am utilizat o strategie de descreștere liniară a inerției: parametrul w scade de la 0.9 la 0.4 pe parcursul iterațiilor.
   - La început (w ≈ 0.9): explorare agresivă, particule cu viteză mare.
   - La final (w ≈ 0.4): exploatare fină, particule care se stabilizează în jurul optimului.
2. Componenta cognitivă (c1 * r1 * (pbest – xi))
   Reprezintă „nostalgia” particulei sau încrederea în sine. Trage particula înapoi spre cea mai bună zonă pe care a descoperit-o personal.

3. Componenta socială (c2 * r2 * (TargetSocial – xi))
   Reprezintă colaborarea. Trage particula spre cea mai bună zonă descoperită de grup. Dacă c2 este prea mare comparativ cu c1, roiul converge rapid (risc de optim local). Dacă c1 este prea mare, particulele rătăcesc individual. În proiect am folosit valorile standard c1 = c2 = 1.49.

2.4. Topologii de vecinătate
Modul în care particulele comunică (definiția termenului TargetSocial) influențează drastic performanța. Am implementat trei topologii:

1. Globală (Star / gbest)
   Toate particulele sunt conectate între ele. Informația despre o nouă descoperire se propagă instantaneu în tot roiul.
   Avantaj: convergență foarte rapidă.
   Dezavantaj: sensibilitate mare la minime locale.
2. Socială (Ring / lbest)
   Fiecare particulă comunică doar cu k vecini imediați din lista de indici ai roiului (ex: particula 5 comunică cu 4 și 6).
   Avantaj: menține diversitatea populației mai mult timp; roiul poate explora mai multe optime locale simultan.
3. Geografică (Euclidiană)
   Vecinătatea este dinamică. La fiecare iterație, pentru fiecare particulă se calculează distanțele euclidiene față de restul roiului și se selectează cei mai apropiați k vecini „fizici”. Aceasta este cea mai realistă simulare a unui roi biologic.

3. MODALITATEA DE REZOLVARE ȘI ARHITECTURA APLICAȚIEI

3.1. Arhitectura software și tehnologii utilizate
Pentru a îndeplini cerințele proiectului, am dezvoltat o aplicație desktop completă utilizând limbajul Python datorită ecosistemului său bogat pentru calcul științific. Arhitectura este modulară, separând logica de calcul de interfața cu utilizatorul.

Componentele principale sunt:

- Back-end (logică): modulele core și problems. Aici sunt implementate clasele pentru PSO și definițiile problemelor. S-a utilizat biblioteca NumPy pentru vectorizarea calculelor.
- Front-end (UI): modulul ui. Implementat cu Tkinter, oferă controale pentru parametrii simulării (slider-e pentru numărul de particule, iterații, complexitate).
- Vizualizare: Matplotlib integrat în Tkinter (FigureCanvasTkAgg), pentru randarea graficelor 2D și 3D direct în fereastra aplicației.

3.2. Gestionarea concurenței (Multithreading)
O provocare tehnică majoră a fost menținerea responsivității interfeței grafice în timpul rulării algoritmului. Deoarece PSO este un algoritm CPU-bound, rularea sa pe firul principal (Main Thread) ar bloca interfața.

Soluția implementată a fost utilizarea modulului threading:

- la apăsarea butonului „START”, se lansează un daemon thread care execută metoda pso.optimize();
- thread-ul calculează pozițiile pentru toate iterațiile și le salvează în lista history;
- după finalizarea calculului, thread-ul secundar semnalizează interfața să înceapă animația folosind root.after(), astfel încât desenarea să se facă pe thread-ul principal (Tkinter nu este thread-safe).

3.3. Definirea funcțiilor de fitness și a penalizărilor
Elementul critic în PSO este funcția de fitness.

Pentru Pathfinding:
Fitness-ul este o sumă ponderată:
$F(x) = D\_total + P\_coliziune$
unde D_total este lungimea drumului, iar P_coliziune este o penalizare mare (soft constraint) dacă drumul intersectează obstacole.

Pentru detectarea coliziunilor în 3D, am implementat un algoritm geometric care calculează proiecția centrului sferei pe segmentul de drum. Dacă distanța de la centrul sferei la segment este mai mică decât raza, avem coliziune.

Pentru Wi-Fi:
Fitness-ul este bazat pe eșantionare spațială:

- se generează o grilă de puncte (ex: 50 x 50) în interiorul camerei;
- pentru fiecare punct se calculează distanța minimă până la cel mai apropiat router;
- fitness-ul este numărul de puncte pentru care distanța minimă este mai mare decât raza de semnal. Scopul este minimizarea acestui număr (adică minimizarea ariei neacoperite).

4. IMPLEMENTARE: STRUCTURA CODULUI SURSĂ

În această secțiune prezentăm fragmente esențiale din codul sursă care ilustrează conceptele teoretice discutate.

4.1. Nucleul algoritmic (PSO Core)
Fișier: src/core/pso_algorithm.py
Clasa PSO gestionează logica de actualizare. Metoda optimize conține bucla principală și implementarea scăderii liniare a inerției.

```
for i, particle in enumerate(self.swarm):

   target_social = self._get_social_target(i)

   r1 = np.random.random(self.dim)

   r2 = np.random.random(self.dim)



   cognitive = self.c1 * r1 * (particle.best_position - particle.position)
```

```
    social = self.c2 * r2 * (target_social - particle.position)


    particle.velocity = (self.w * particle.velocity) + cognitive + social

    particle.velocity = np.clip(particle.velocity, -self.v_max, self.v_max)

    particle.position += particle.velocity


    for d in range(self.dim):

        particle.position[d] = max(self.bounds[d][0],
min(particle.position[d], self.bounds[d][1]))
```

4.2. Implementarea problemelor specifice
Fișier: src/problems/problem_pathfinding.py
Se observă transformarea vectorului unidimensional al particulei într-o serie de puncte 2D (waypoints) și calculul penalizărilor.

```
def fitness_function(self, particle_position):

    waypoints = particle_position.reshape((self.num_waypoints, 2))

    full_path = [self.start] + list(waypoints) + [self.end]

    total_distance = 0

    penalty = 0

    for i in range(len(full_path) - 1):

        p1 = np.array(full_path[i])

        p2 = np.array(full_path[i + 1])


        dist = np.linalg.norm(p2 - p1)

        total_distance += dist


        samples = 5

        for t in np.linspace(0, 1, samples):

            sample_point = p1 + t * (p2 - p1)

            if self._is_point_in_obstacle(sample_point):

                penalty += 200
```

```
    return total_distance + penalty
```

5. REZULTATE EXPERIMENTALE ȘI SCENARII DE RULARE

Aplicația a fost testată intensiv într-o serie de scenarii pentru a valida corectitudinea algoritmului și stabilitatea convergenței.

5.1. Scenariul A: Pathfinding 2D și 3D

Configurare:

- Particule: 40
- Iterații: 100
- Waypoints: 5
- Obstacole: 4 cercuri/sfere dispuse strategic pentru a bloca linia directă.

Rezultate 2D:
În primele iterații, traseele generate de particule sunt haotice, intersectând frecvent obstacolele. Pe măsură ce algoritmul avansează, penalizările din funcția de fitness forțează particulele să deplaseze traseul în afara zonelor interzise. Soluția finală converge către un traseu care ocolește obstacolele și minimizează lungimea totală.

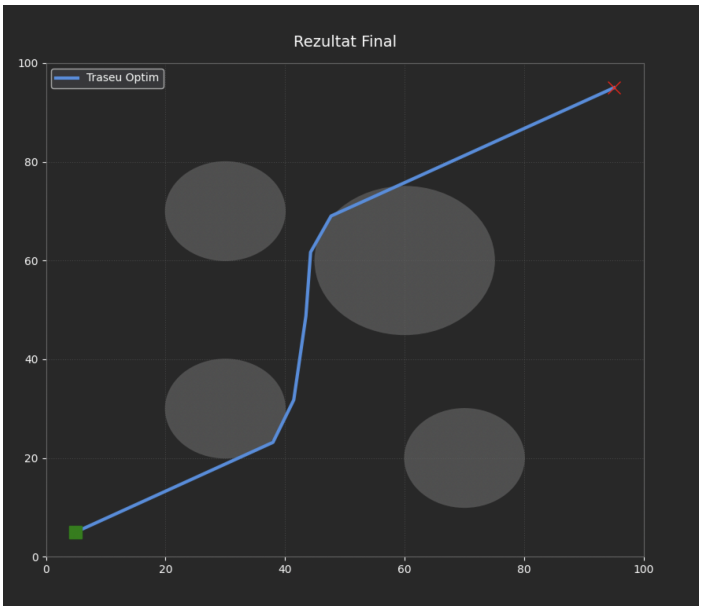

Fig. 1. Traseul optim generat în mediul 2D.

Rezultate 3D:
În varianta 3D, spațiul de căutare este mai vast. Algoritmul poate găsi soluții care ocolesc obstacolele nu doar lateral, ci și pe deasupra / pe dedesubt (folosind axa Z), uneori obținând o distanță totală mai mică decât o ocolire strict plană.

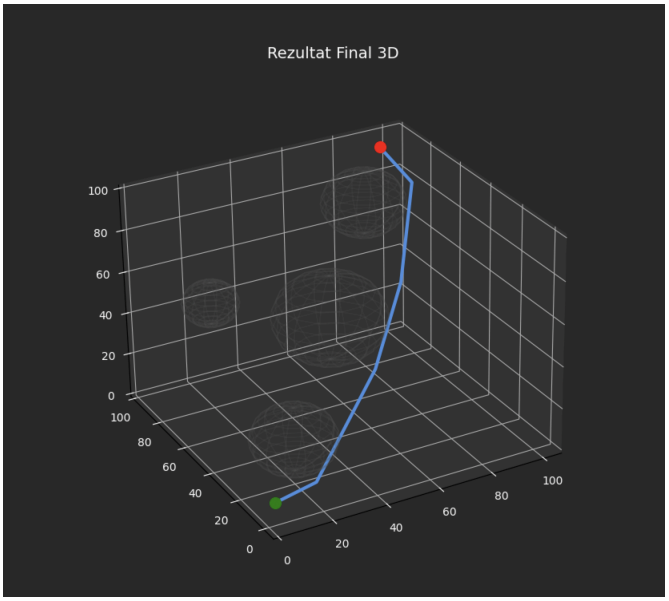

Fig. 2. Vizualizarea 3D a traseului și a obstacolelor sferice.

5.2. Scenariul B: Optimizare Wi-Fi și heatmaps
Configurare:

- Particule: 40
- Iterații: 100
- Routere: 5
- Raza semnal: 35 unități.

Aplicația generează la final un heatmap. Culorile calde reprezintă zonele cu acoperire bună, iar zonele reci zonele neacoperite. Algoritmul distanțează routerele și le poziționează astfel încât să maximizeze acoperirea. S-a observat o tendință de auto-organizare într-o configurație echilibrată.

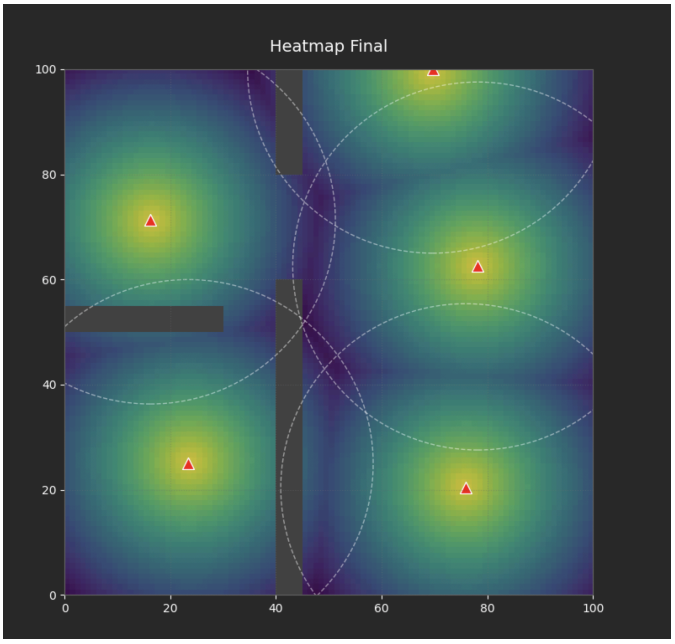

Fig. 3. Poziționarea optimă a routerelor și harta de acoperire.

5.3. Studiu comparativ al topologiilor

S-a rulat comparativ topologia globală, socială și geografică pentru problema Pathfinding. Graficul de convergență (Cost vs. Iterații) evidențiază:

- Topologia globală: scade abrupt în primele 10–20 iterații; converge rapid, dar poate rămâne într-un optim local.
- Topologia socială: scade mai lent; menține diversitatea și explorează mai multe rute candidate.
- Topologia geografică: compromis bun între explorare și exploatare, comportament stabil.

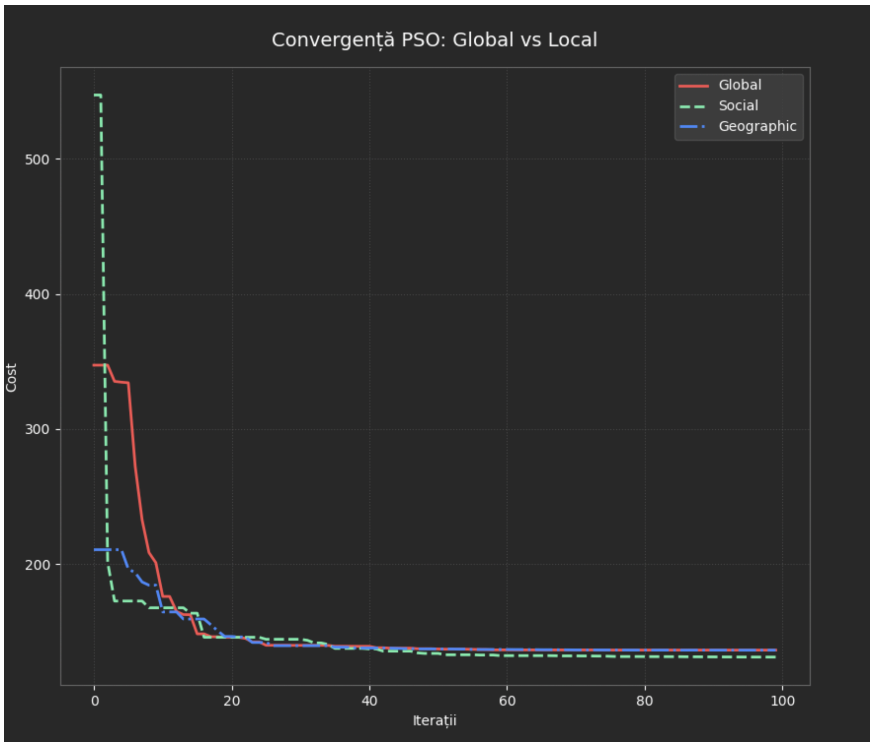

Fig. 4. Analiza comparativă a ratei de convergență.

6. CONCLUZII

Proiectul a demonstrat aplicabilitatea algoritmilor bio-inspirați în rezolvarea problemelor inginerești complexe. Prin implementarea Particle Swarm Optimization (PSO), am obținut soluții optime sau cvasi-optime pentru planificarea rutelor și pentru acoperirea semnalului.

Concluzii principale:

- Versatilitatea PSO: același motor algoritmic, cu modificări minime, a rezolvat probleme cu naturi diferite (minimizare distanță vs. minimizare arie neacoperită).
- Impactul parametrilor: alegerea factorului de inerție w și a strategiei de scădere a acestuia este critică.
- Rolul topologiei: nu există o topologie „supremă”; gbest este rapid, lbest este mai robust în peisaje dificile.
- Vizualizarea: interfața grafică a fost utilă atât pentru prezentare, cât și pentru înțelegerea dinamicii roiului.

Direcții viitoare: obstacole dinamice, modele mai realiste de propagare Wi-Fi, variante hibride PSO (combinare cu alte metaeuristici).

7. BIBLIOGRAFIE

NumPy Documentation: https://numpy.org/doc/
Matplotlib Documentation: https://matplotlib.org/
Python Tkinter Documentation: https://docs.python.org/3/library/tkinter.html

PSO Overview: https://www.geeksforgeeks.org/machine-learning

8. LISTA DE CONTRIBUȚII ALE MEMBRILOR ECHIPEI

Proiectul a fost realizat în regim de colaborare, sarcinile fiind distribuite între membrii echipei.

Ghiurca Andrei Cosmin:

- a proiectat și implementat nucleul algoritmului PSO (src/core/pso_algorithm.py), incluzând logica de actualizare a vitezei, poziției și gestionarea limitelor spațiului de căutare.
- a dezvoltat modulul pentru problema de Pathfinding (2D și 3D), implementând algoritmii de detecție a coliziunilor și funcțiile de fitness cu penalizări.
- a realizat structura de bază a interfeței grafice (Tkinter main window) și integrarea sistemului de multithreading.

Ciorap Silviu George:

- a dezvoltat modulul pentru problema de Optimizare Wi-Fi, implementând logica grilei de eșantionare și calculul acoperirii.
- a implementat vizualizările: generarea heatmap-urilor, animațiile 3D și modulul de studiu comparativ cu grafice de convergență.
- a adus îmbunătățiri interfeței grafice.
- a contribuit la redactarea documentației și la testarea finală a aplicației.