Proiect - Inteligență artificială

Algoritmul NSGA-II pentru optimizare multi-obiectiv

Proiect realizat de : Asofronie Rareș-Flavian

Vatamanu Alexandru-Ștefan

Grupa 1405B

Profesor de laborator: Hulea Mircea

Descrierea problemei considerate

Problema considerată în acest proiect reprezintă optimizarea proiectării unui vehicul din punct de vedere al consumului de combustibil și al vitezei maxime.

Cele 2 obiective sunt contradictorii, deoarece reducerea consumului de combustibil necesită o greutate redusă și coeficient aerodinamic scăzut, ceea ce poate limita viteza maximă.

De aceea, obiectivele analizate sunt:

- Minimizarea consumului de combustibil
- Maximizarea vitezei maxime a vehiculului

Această problemă ilustrează un conflict clasic în optimizarea multi-obiectivă, deoarece creșterea vitezei maxime presupune adesea un consum mai mare de combustibil și invers.

Aspecte teoretice privind algoritmul

NSGA-II (Non-dominated Sorting Genetic Algorithm II) este un algoritm evolutiv destinat optimizării problemelor multi-obiectiv.

Principalele componente ale NSGA-II sunt:

1. Sortare nedominată : Soluțiile sunt clasificate pe baza dominanței Pareto. Soluțiile din frontul 1 nu sunt dominate de alte soluții, iar soluțiile din frontul 2 sunt dominate doar de cele din frontul 1 etc.
2. Distanța de aglomerare : Măsoară diversitatea soluțiilor în cadrul unui front folosind distanța relativă dintre puncte în spațiul obiectivelor.
3. Selcție elitistă : Populația este selectată pentru generația următoare pe baza fronturilor și a distanței de aglomerare, asigurând diversitate și convergență.
4. Operatori genetici : Includ încrucișarea și mutația pentru a genera variație în populație.

Modalitate de rezolvare

Am implementat un model bazat pe NSGA-II pentru a soluționa această problemă.

Principalele etape sunt :

1. Inițializarea populației : Generarea aleatorie a unei populații inițiale de soluții candidate.
2. Evaluarea obiectivelor : Calcularea consumului și a vitezei maxime pentru fiecare soluție.
3. Sortare și selecție : Clasificarea soluțiilor folosind sortarea nedominată și selectarea celor mai bune soluții bazate pe diversitate.
4. Generarea urmașilor : Crearea de soluții noi prin încrucișare și mutație.
5. Repetare : Procesul continuă până la atingerea unui număr de generații sau a unei alte condiții de oprire.

Părți semnificative din codul sursă

```python
import numpy as np
import matplotlib.pyplot as plt

# Parametrii problemei
DIMENSIUNE_POPULATIE = 100
GENERATII = 50
RATA_INCRUCISARE = 0.9
RATA_MUTATIE = 0.1


# Intervalele variabilelor de decizie
LIMITA_PUTERE = [50, 300]  # Puterea motorului (kW)
LIMITA_AERODINAMICA = [0.2, 0.8]  # Coeficientul aerodinamic
LIMITA_GREUTATE = [800, 2000]  # Greutatea vehiculului (kg)

```

S-au definit parametrii care guvernează procesul de optimizare. Aceștia includ:

- **DIMENSIUNE_POPULATIE:** Specifică numărul de indivizi (vehicule proiectate) din populația inițială și din fiecare generație ulterioară.
- **GENERATII:** Numărul total de generații pe care algoritmul le va genera pentru a evolua soluțiile.
- ***RATA_INC*RUCISARE:** Probabilitatea ca doi părinți să se împreuneze pentru a genera urmași noi.
- **RATA_MUTATIE:** Probabilitatea ca un individ să sufere modificări aleatorii (mutații) pentru a explora soluții neconsiderate anterior.

Acești parametri afectează viteza de convergență a algoritmului și sunt importanți pentru a controla evoluția populația.

```python

# Functiile obiectiv
def functii_obiectiv(individ):
    putere, coeficient, greutate = individ
    consum_combustibil = (greutate * coeficient) / putere
    viteza_maxima = putere / (coeficient * greutate)
    return consum_combustibil, viteza_maxima


```

Această funcție calculează valorile funcțiilor obiectiv pentru un individ dat : consum_combustibil (calculat ca raportul dintre produsul greutății și coeficientul aerodinamic și puterea motorului, scopul fiind minimizarea acestui consum) și viteza_maxima (calculat ca raportul dintre putere și produsul dintre coeficientul aerodinamic și greutate, scopul fiind maximizarea vitezei maxime).

```python
# Initializarea populatiei
def initializare_populatie(dimensiune):
    populatie = []
    for _ in range(dimensiune):
        putere = np.random.uniform(*LIMITA_PUTERE)
        coeficient = np.random.uniform(*LIMITA_AERODINAMICA)
        greutate = np.random.uniform(*LIMITA_GREUTATE)
        populatie.append([putere, coeficient, greutate])
    return np.array(populatie)

```

Se generează o populație inițială de soluții, iar fiecare din acestea este reprezentată printr-un vector ce conține 3 variabile : putere(puterea motorului generată aleatoriu în intervalul specificat de LIMITA_PUTERE), coeficient (coeficientul aerodinamic, generat aleator în intervalul LIMITA_AERODINAMICA) și greutate (greutatea vehiculului generat aleator în intervalul LIMITA_GREUTATE). Funcția returnează populația sub formă de matrice (array), fiecare linie reprezentând un individ.

```python
# Sortarea nedominata
def sortare_nedominata(populatie):
    fronturi = []
    numar_dominatii = np.zeros(len(populatie))
    solutii_dominante = [[] for _ in range(len(populatie))]

    for i in range(len(populatie)):
        for j in range(len(populatie)):
            f1_i, f2_i = functii_obiectiv(populatie[i])
            f1_j, f2_j = functii_obiectiv(populatie[j])

            if (f1_i < f1_j and f2_i > f2_j) or (f1_i <= f1_j and f2_i > f2_j) or (f1_i < f1_j and f2_i >= f2_j):
                solutii_dominante[i].append(j)
            elif (f1_j < f1_i and f2_j > f2_i) or (f1_j <= f1_i and f2_j > f2_i) or (f1_j < f1_i and f2_j >= f2_i):
                numar_dominatii[i] += 1

        if numar_dominatii[i] == 0:
            fronturi.append([i])

    return fronturi


```

Această funcție clasifică soluțiile în fronturi pe baza principiului dominanței Pareto:

- → Pasul 1: Pentru fiecare soluție din populație, se compară valorile funcțiilor obiectiv cu cele ale celorlalte soluții.
- → Pasul 2: O soluție domină o alta dacă este mai bună la un obiectiv și nu este mai rea la celelalte obiective.
- → Pasul 3: Soluțiile care nu sunt dominate de nicio altă soluție sunt plasate în Frontul 1.
- → Pasul 4: Soluțiile dominate doar de cele din Frontul 1 sunt plasate în Frontul 2, și așa mai departe.

Funcția returnează o listă de fronturi, fiecare front conținând indicii soluțiilor respective.

```python
# Calcularea distantei de aglomerare
def distanta_aglomerare(front, populatie):
    distante = np.zeros(len(front))
    obiective = np.array([functii_obiectiv(populatie[i]) for i in front])

    for m in range(obiective.shape[1]):
        indices_sortate = np.argsort(obiective[:, m])
        distante[indices_sortate[0]] = distante[indices_sortate[-1]] = float('inf')
        for i in range(1, len(front) - 1):
            distante[indices_sortate[i]] += (
                obiective[indices_sortate[i + 1], m] - obiective[indices_sortate[i - 1], m]
            )

    return distante
```

Funcția de mai sus calculează diversitatea soluțiilor dintr-un front folosind distanța de aglomerare. Pentru fiecare obiectiv (consum , viteză) soluțiile sunt sortate, marginea frontului primește o distanță infinită, soluțiile interioare primesc valori calculate pe baza diferenței dintre vecinii lor.

Rezultatul este o măsurare a spațiului relativ dintre soluțiile din front, contribuind la selecția ulterioară.

```python
# Operatori pentru selectie, incrucisare si mutatie
def incrucisare(parinte1, parinte2):
    if np.random.rand() < RATA_INCRUCISARE:
        return (np.array(parinte1) + np.array(parinte2)) / 2
    return parinte1
```

Această funcție implementează operatorul de încrucișare. Dacă probabilitatea aleatorie este mai mică decât rata de încrucișare, soluțiile părinților sunt combinate prin calcularea mediei componentelor lor. Dacă nu, unul dintre părinți este returnat ca atare.

```python
def mutatie(individ):
    if np.random.rand() < RATA_MUTATIE:
        individ[0] += np.random.uniform(-10, high: 10)
        individ[1] += np.random.uniform(-0.05, high: 0.05)
        individ[2] += np.random.uniform(-50, high: 50)

        individ[0] = np.clip(individ[0], *LIMITA_PUTERE)
        individ[1] = np.clip(individ[1], *LIMITA_AERODINAMICA)
        individ[2] = np.clip(individ[2], *LIMITA_GREUTATE)

    return individ

```

Această funcție implementează operatorul de mutație. Dacă probabilitatea aleatorie este mai mică decât rata de mutație, componentele individului sunt modificate aleator (puterea este ajustată într-un interval specificat, coeficientul aerodinamic este ajustat într-un interval mai mic, greutatea este modificată în limitele stabilite).

Valorile sunt apoi limitate pentru a rămâne în limitele parametrilor problemei.

```python
populatie = initializare_populatie(DIMENSIUNE_POPULATIE)

for generatie in range(GENERATII):
    urmasi = []

    if len(populatie) < DIMENSIUNE_POPULATIE:
        populatie_suplimentara = initializare_populatie(DIMENSIUNE_POPULATIE - len(populatie))
        populatie = np.concatenate((populatie, populatie_suplimentara))

    for _ in range(DIMENSIUNE_POPULATIE // 2):
        parinte1, parinte2 = np.random.choice(range(len(populatie)), size: 2, replace=False)
        copil1 = incrucisare(populatie[parinte1], populatie[parinte2])
        copil2 = incrucisare(populatie[parinte2], populatie[parinte1])
        urmasi.append(mutatie(copil1))
        urmasi.append(mutatie(copil2))

    populatie = np.concatenate((populatie, urmasi))

    fronturi = sortare_nedominata(populatie)
    populatie_noua = []

    for front in fronturi:
        if len(populatie_noua) + len(front) > DIMENSIUNE_POPULATIE:
            distante = distanta_aglomerare(front, populatie)
            indici_sortati = np.argsort(-distante)
            populatie_noua.extend(
                [populatie[i] for i in indici_sortati[: DIMENSIUNE_POPULATIE - len(populatie_noua)]]
            )
```

```
            break
        populatie_noua.extend([populatie[i] for i in front])

    if len(populatie_noua) < DIMENSIUNE_POPULATIE:
        populatie_suplimentara = initializare_populatie(DIMENSIUNE_POPULATIE - len(populatie_noua))
        populatie_noua.extend(populatie_suplimentara)

    populatie = np.array(populatie_noua)

# Plotarea solutiilor
valori_f1, valori_f2 = zip(*[functii_obiectiv(individ) for individ in populatie])
plt.scatter(valori_f1, valori_f2, color="blue", label="Solutii Pareto")
plt.xlabel("Consumul de combustibil (f1)")
plt.ylabel("Viteza maxima (f2)")
plt.title("Frontul Pareto pentru optimizarea vehiculului")
plt.legend()
plt.show()
```

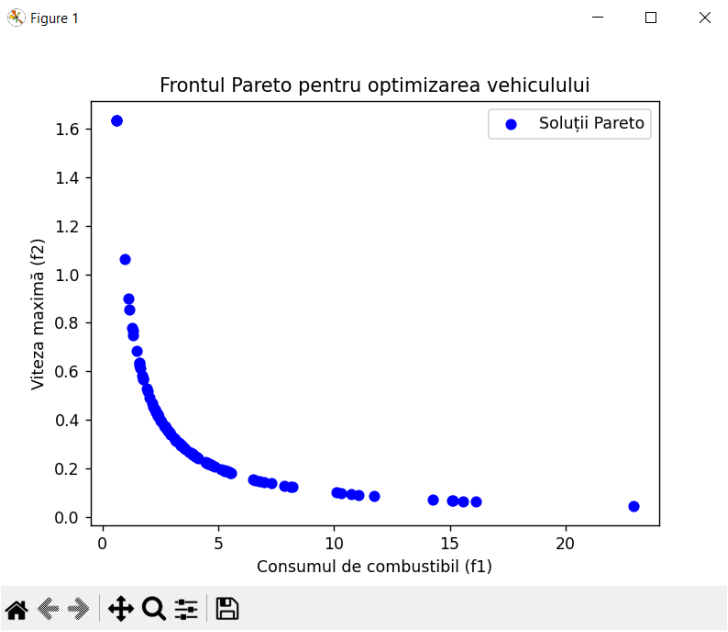
Se creează o populație inițială de soluții, unde fiecare individ este generat aleator conform limitelor parametrilor problemei. Fiecare individ reprezintă o configurație a vehiculului definită de putere, coeficient aerodinamic și greutate. Se parcurge un număr predefinit de generații (GENERATII), simulând procesul evolutiv al populației. Dacă populația este incompletă (de exemplu, în urma selecției), se generează soluții suplimentare pentru a o completa. Generarea urmașilor :

- Selecția părinților: Doi părinți sunt selectați aleatoriu fără înlocuire.
- Crearea urmașilor: Urmașii sunt generați prin încrucișarea celor doi părinți, unde valorile variabilelor sunt combinate.
- Mutația: Fiecare urmaș este supus unui proces de mutație aleatorie pentru a explora soluții noi.
- Urmașii sunt adăugați într-o listă (urmasi).

Plotarea soluțiilor:

- Soluțiile finale din populație sunt evaluate și reprezentate grafic.
- Pe axa X: Consumul de combustibil care trebuie minimizat.
- Pe axa Y: Viteza maximă care trebuie maximizată.
- Graficul afișează Frontul Pareto, care conține cele mai bune compromisuri între cele două obiective contradictorii.

Rezultatele obținute prin rularea programului



Graficul prezentat ilustrează frontul Pareto pentru optimizarea proiectării unui vehicul, având cele două obiective contradictorii:

- Pe axa X: Consumul de combustibil (f_1), care trebuie minimizat.
- Pe axa Y: Viteza maximă (f_2)care trebuie maximizată.

Graficul reflectă o relație inversă între cele două obiective. Creșterea vitezei maxime (f_2) determină o creștere a consumului de combustibil (f_1), ceea ce evidențiază conflictul dintre aceste obiective.

Soluțiile nedominate sunt distribuite pe o curbă ce reprezintă cel mai bun compromis posibil între consum și viteză.

Concluzii

Codul implementează un proces complet de optimizare multi-obiectivă utilizând NSGA-II. Aceste soluții oferă utilizatorului o gamă variată de opțiuni, în funcție de prioritățile stabilite: fie minimizarea consumului de combustibil, fie maximizarea vitezei maxime. Prin generarea, evaluarea și selecția soluțiilor în funcție de dominanța Pareto, algoritmul converge la soluțiile nedominate care oferă un echilibru optim între consum și viteză. Acest flux este vizualizat prin graficul final, oferind informații valoroase pentru decizii de proiectare.

Bibliografii

1. Pagina disciplinei : https://edu.tuiasi.ro/course/view.php?id=640
2. O introducere mai detaliată a acestor algoritmi : https://staff.fmi.uvt.ro/~daniela.zaharie/cne2014/curs/cne2014_slides7.pdf
3. Exemplu concret : https://webspace.ulbsibiu.ro/adrian.florea/html/Planificari/EvolutionaryComputing/Course_4/pdf/IA04_Optimizare.pdf (pag. 66 ->)

Împărțirea task-urilor

Asofronie Rareș-Flavian : documentarea și analiza teoretică, vizualizarea rezultatelor și redactarea concluziilor

Vatamanu Alexandru-Ștefan : implementarea codului , vizualizarea rezultatelor și redactarea concluziilor