UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAŞI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DOMENIUL: Calculatoare şi tehnologia informaţiei
SPECIALIZAREA:Tehnologia informaţiei

Algoritmul NSGA-II pentru optimizare multiobiectiv

PROIECT LA DISCIPLINA
INTELIGENŢĂ ARTIFICIALĂ

Profesor îndrumător
Florin Leon

Studenti
Avarvarei Elena-Mădălina, 1410B
Pintilie Elena-Cristiana, 1410B

Cuprins

1. Descrierea problemei considerate

Aplicație a algoritmului NSGA-II pentru optimizare multi-obiectiv în vederea achiziționării unui calculator. Se va face o alegere optimă, luând în calcul mai mulți factori precum: preț, procesor, frecvență, memorie etc, dar poate face și monitorizarea vanzarilor laptopurilor de către furnizori.

2. Algoritmul NSGA-II – Noțiuni teoretice

Cele mai multe probleme de optimizare care apar în practică au un caracter multi-obiectiv deoarece în cadrul lor este necesar să fie optimizate simultan mai multe funcții obiectiv. Comparativ cu problemele de optimizare mono-obiectiv care pot admite și o singura soluție de optim sau un număr finit de soluții de optim, optimizările multiobiectiv ale unor obiective conflictuale conduc implicit la obținerea unui număr infinit de soluții optime, numite Pareto optimale, fiecare din ele indicând un posibil consens între obiectivele considerate. Reprezentarea acestor soluții în spațiul obiectiv reprezintă frontul optimal. [1]

În rezolvarea problemelor de optimizare multiobiectiv, principala dificultate rezultă din faptul că este posibilă doar o ordonare parțială a soluțiilor. Ordonarea soluțiilor în funcție de gradul lor de adaptare conduce implicit la subseturi de soluții asociate cu același rang, i.e. cu același grad de adaptare. Schemele de asociere a rangurilor pot include și preferințe suplimentare, extrase din mecanismul decizional. Aceste modificări trebuie însă gestionate astfel încât să evite orientarea prematură a explorării către o anumită preferință. Scopurile principale ale unei metode de optimizare multiobiectiv sunt obținerea unor aproximări cât mai bune pentru frontul optimal și distribuirea bună a soluțiilor de-a lungul acestuia.

Algoritmii evolutivi de optimizare multiobiectiv sunt recomandați în rezolvarea problemelor de optimizare multiobiectiv cu obiective conflictuale. Lucrând pe multiple de soluții la fiecare iterație, aceștia s-au remarcat prin robustețe, conducând la rezultate foarte bune în diverse aplicații.

O structură sau individ este o soluție codificată a unei probleme. În mod obișnuit un individ este reprezentat de un șir (sau șir de șiruri) ce corespunde la un genotip biologic. Acest genotip este compus din unul sau mai mulți cromozomi, unde fiecare cromozom este compus din gene care iau diverse valori (alele) dintr-un alfabet genetic. Un loc (poziție) identifică o poziție a genei în cadrul unui cromozom. În final o mulțime de cromozomi este numită populație. [1]

Ca și în natură operatorii evolutivi operează pe o populație încercând să genereze soluții cu o cât mai mare potrivire (fitness). Operatorii principali sunt operatorii de mutație, recombinare și selecție.

Componentele unui algoritm evolutiv sunt:

- populația - mulțime de indivizi (soluții);
- părinții - membrii generației curente;
- copiii - membrii generației următoare;
- generația –o populație creată în cadrul unei iterații.

Structura de date este compusă din:

- cromozomi - formă de a codifica soluția: vector (șir) ce constă din gene cu alele asignate

• potrivire (fitness) - număr asignat unei soluții (ceea ce se dorește). [1]

2.1 Descrierea principiului în problema considerată

În acest proiect se realizează o analiză a modului de rezolvare a problemelor de optimizare multi-obiectiv, cu accent pe rezolvarea bazată pe algoritmul evolutiv NSGA-II. Aplicația are ca prinicipala functionalitate monitorizarea vanzării laptopurilor dintr-un magazin în luna curentă.

Limbajul folosit pentru implementarea algoritmului este Python, folosindu-se librăriile corespunzătoare pentru afişarea graficului frontului Pareto.

2.2 Dominația și frontul Pareto

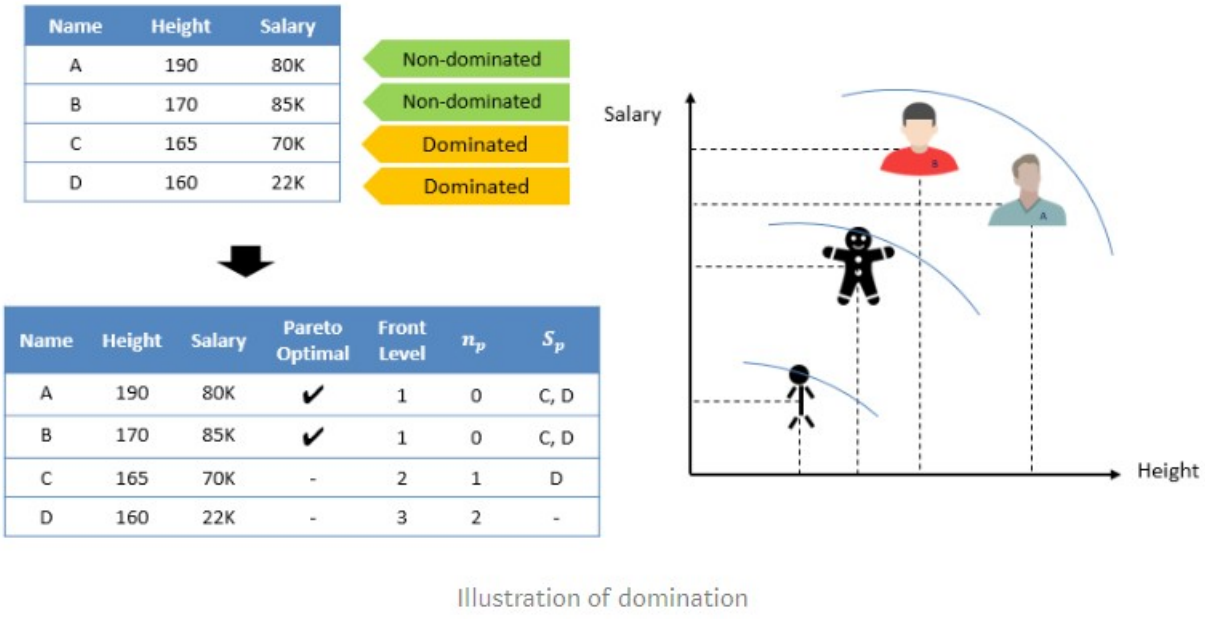


Figura 1: Exemplu de ilustrare a solutiilor dominante si non-dominante[2]

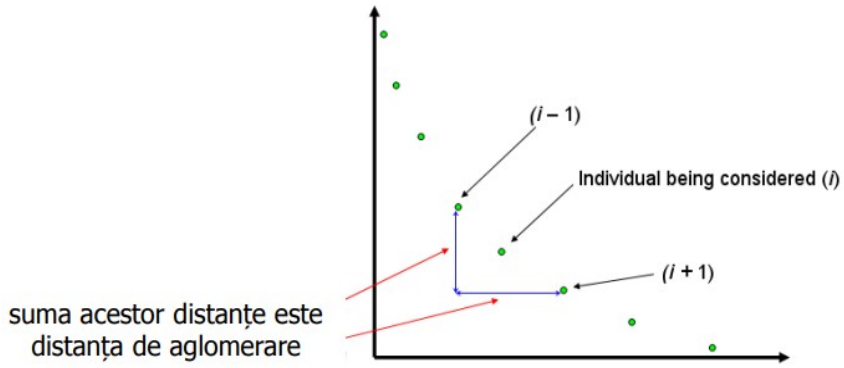
Într-o problemă de optimizare pe o singură funcție obiectiv, este uşor de găsit calitatea soluției, ca de exemplu un scor mai mare, este o soluție mai bună. Deşi se spune că este o problemă de optimizare multiobiectiv, soluția optimă nu este atât de uşor de identificat. Atunci când sunt conflicte între funcțiile obiectiv. De aceea, folosim dominația pentru a judeca dacă soluția este bună sau nu.

În exemplul de mai sus, A,B,C,D sunt angajați care sunt clasificați după înălțime şi salariu. A şi B sunt soluțiile non-dominante deoarece, clasificarea după cele 2 criterii nu coincide. A este cel mai înalt, însă nu are cel mai mare salariu. B câştigă cel mai mult, însă este pe locul al doilea după înălțime. C este pe locul 3, iar D pe locul 4 din toate punctele de vedere. De aceea C şi D se numesc soluții dominante, iar A şi B sunt aduşi pe frontul soluțiilor non-dominante.

Particularitatea algoritmilor evolutivi de optimizare multiobiectiv este de a lucra cu o populație de indivizi permite găsirea unui set de soluții distribuite în apropierea setului Pareto, ilustrat tot în figura 1, dintr-o singură rulare. Considerând cele anterior amintite, algoritmii genetici multiobiectiv trebuie proiectați astfel încât să îndeplinească simultan cerințele principale: să aproximeze frontul Pareto utilizând un mecanism evolutiv de căutare şi să asigure diversitatea soluțiilor mai bine adaptate. A doua cerință este cel mai adesea rezolvată utilizând tehnici specifice precum partajarea sau gruparea.

2.3 Selecția

Tehnicile Pareto folosesc **selecția bazată pe ranguri** pentru a alege părinții pentru bazinul de reproducere și supraviețuitorii generației următoare. O diversitate sporită în cadrul populației crește relevanța sortării și permite obținerea unui front de soluții nedominate mai relevante pentru descrierea frontului Pareto-optimal (dacă aceste soluții nedominate sunt în vecinătatea frontului Pareto-optimal). [5]




```
#Selectia dupa rang
def selectie(valori1, valori2):
  S=[]
  n=[]
  rang=[]
  for i in range(0, len(valori1)):
    S.append([])
    n.append(0)
    rang.append(0)
  front = [[]]
  for x in range(0, len(valori1)):
    S[x] = []
    n[x] = 0
  for y in range(0, len(valori1)):
            # facem verificarile ca sa vedem care solutii sunt dominante/nedominante
            # Daca x domina y
            if (valori1[x] > valori1[y] and valori2[x] > valori2[y]) or (
                    valori1[x] >= valori1[y] and valori2[x] > valori2[y]) or (
                    valori1[x] > valori1[y] and valori2[x] >= valori2[y]):
                # anexam solutia gasita la lista noastra
                if y not in S[x]:
                    S[x].append(y)
            # Daca y domina x
            elif (valori1[y] > valori1[x] and valori2[y] > valori2[x]) or (
                    valori1[y] >= valori1[x] and valori2[y] > valori2[x]) or (
                    valori1[y] > valori1[x] and valori2[y] >= valori2[x]):
                # Se mareste numarul de fronturi de care este dominat
                n[x] = n[x] + 1
```

```
    # Daca nu mai este dominat de niciun front, ci el le domina pe toate:
          if n[x] == 0:
              # rangul apartine primului front
              rang[x] = 0
              if x not in front[0]:
                  front[0].append(x)

      # Initializam numarul de fronturi
      i = 0
      while (front[i] != []):
          # Q este folosit pentru a stoca lista de membri din frontul urmator
          Q = []
          for x in front[i]:
              for y in S[x]:
                  n[y] = n[y] - 1
          # Daca nu mai este dominat de niciun front
                if (n[y] == 0):
                    # Rangul apartine frontului urmator, adaugand in lista eleme
ntele necesare
                    rang[y] = i + 1
                    if y not in Q:
                        Q.append(y)
                    i = i + 1
        front.append(Q)
    del front[len(front) - 1]
    return front
```

Membrii de rang 1 constituie mulţimea nedominată, adică aproximarea curentă a frontului Pareto. **Sortarea populaţiei pe ranguri** este nedominată, astfel încât membrii de rang n domină toţi membrii de rang > n .

Indivizii care aparţin aceluiaşi front sunt sortaţi pe baza **distanţei de aglomerare (crowding distance):**

- Un individ mai bun are o distanţă de aglomerare mai mare
- Efectul este selecţia indivizilor aflaţi în regiuni mai puţin aglomerate
- Previne omogenizarea soluţiilor (convergenţa prematură)

Pentru a sorta soluţii de aceeaşi frontieră, se foloseşte măsura distanţei de aglomerare. Pentru fiecare funcţie obiectivă, soluţiile din aceeaşi frontieră sunt sortate în ordine crescătoare. Pe baza distanţei dintre 2 soluţii consecutive, măsura distantei de aglomerare se calculează după cum urmează în funcţia prezentată mai jos. De menţionat este că datorită imporanţei soluţiilor limită ale unei frontiere în detectarea formei sale, prima şi ultima distanţă de aglomerare sunt stabilite la infinit. În cele din urmă, pentru a selecta cele mai bune persoane pentru generaţia următoare, soluţiile sunt sortate în ordine crescătoare, în funcţie de rangul lor. Printre soluţiile cu acelaşi rang se preferă soluţiile cu o distanţă mai mare de aglomerare.

```
def distanta_aglomerare(valori1, valori2, front):
  distanta=[]
  for i in range(0, len(front)):
    distanta.append(0)
    # Sortarea in functie de fiecare functie obiectiv
     sortarea1 = sortare(front, valori1[:])
     sortarea2 = sortare(front, valori2[:])

     # Punctele extreme/limita vor fi mereu selectate
     distanta[0] = infinit
     distanta[len(front) - 1] = infinit

     # luam fiecare cromozom in parte
     for k in range(1, len(front) - 1):
         # calculam distanta de aglomerare ca fiind suma dintre distanta calcu
lata pana in acel moment si
         # numitorul reprezinta diferenta dintre cel mai mare element si urmat
orul cel mai mic
         distanta[k] = distanta[k] + (valori1[sortarea1[k + 1]] - valori2[sort
area1[k - 1]]) / (
                 max(valori1) - min(valori1))
     for k in range(1, len(front) - 1):
         distanta[k] = distanta[k] + (valori1[sortarea2[k + 1]] - valori2[sort
area2[k - 1]]) / (
                 max(valori2) - min(valori2))
     return distanta
```

2.4 Încrucișarea și mutația

Într-un proces natural, crossover-ul se produce între perechile de cromozomi prin schimbul de părți din acestea. Într-un algoritm genetic, două șiruri sunt prelevate aleatoriu din plaja de împerechere și unele porțiuni sunt schimbate pentru a crea două șiruri noi. Dacă se folosește o probabilitate de încrucișare numită pc, din punct de vedere al implementării, se generează un număr aleatoriu între 0 și 1. Operatorul de mutație urmează operatorul de încrucișare. Mutația asigură modificarea valorilor unor gene pentru a evita situațiile în care o anumită alelă (valoare posibilă a unei gene) nu apare în populație deoarece nu a fost generată de la început. În felul acesta poate fi crescută diversitatea populației și se poate evita convergența într-un optim local. [3][4]

```
def crossover_mutatie(mama, tata):
  # aplicam incrucisarea reala, aritmetica + calculam probabilitatea de incrucisa
re
  r = random.random()
  # calculam probabilitatea de a returna mama / tatal
  if r > 0.5:
    probabilitate = random.random()
    # mutatia genei se produce doar daca probabilitatea este satisfacuta
    if probabilitate < 1:
      # resetam gena respectiva cu o noua valoare intre valorile minima si maxima
       mama = min_x + (max_x - min_x) * probabilitate

      return mama
    else:
      probabilitate = random.random()
      # mutatia genei se produce doar daca probabilitatea este satisfacuta
      if probabilitate < 1:
        # resetam gena respectiva cu o noua valoare intre valorile minima si max
  ima
        tata = min_x + (max_x - min_x) * probabilitate
      return tata
```

2.5 Algoritmul NSGA-II – implementare

Pentru implementarea propriu-zisă a algoritmului s-au apelat în mare parte funcţiile implementate, unele dintre ele s-au prezentat în subcapitolele anterioare. S-a urmat exemplul pseudocodului prezentat la cursul de Inteligență Artificială.

```
# Initialiarea populatiei
  solutie = initializare_populatie(dim_populatie, min_x, max_x)

  while (numarul_generatiei < nr_max_indivizi):
    # Evaluarea functiilor obiectiv
    valori_pentru_pret=eval_fct_pret(dim_populatie,solutie)
    valori_pentru_rata=eval_fct_rata(dim_populatie,solutie)

    # Selectia dupa rang
    rez_selectie = selectie(valori_pentru_pret[:], valori_pentru_ram[:])
    afisare_front(solutie,rez_selectie,numarul_generatiei)
    valori_pentru_distanta_aglomerare=generare_valori_distanta_aglomerare(valori_
pentru_pret,valori_pentru_rata,rez_selectie)

    # Generarea copiilor
    solutie_noua = solutie[:]
    while (2 * dim_populatie != len(solutie_noua)):
      # Solutia noua cuprinde copilul generat de parinti prin crossover si mutati
a dupa ce au fost selectati parintii(vectorul solutie, dupa distanta de aglomera
re)
cross_mut=crossover_mutatie(solutie[random.randint(0, dim_populatie - 1)], solut
ie[random.randint(0, dim_populatie - 1)])
        solutie_noua.append(cross_mut)
       # Evaluarea functiilor obiectiv pentru solutia copil
         valori_pentru_pret_2 = eval_fct_pret(2 * dim_populatie, solutie_noua)
         valori_pentru_rata_2 = eval_fct_rata(2 * dim_populatie, solutie_noua)

         # Selectia dupa rang
         rez_selectie_2 = selectie(valori_pentru_pret_2[:], valori_pentru_rata_2[
:])
```

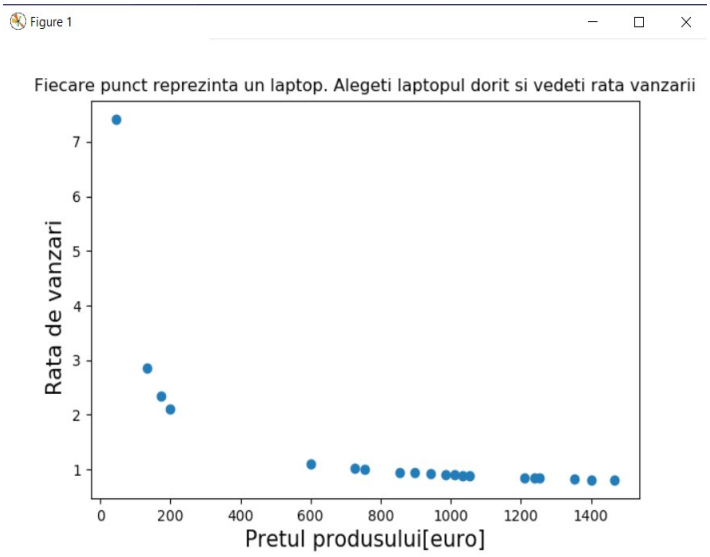
```
# Calculam distanta de aglomerare dintre ceea ce s-a selectat in solutia noua
    valori_pentru_dist_aglomerare_2=generare_valori_distanta_aglomerare(valori_
pentru_pret_2,valori_pentru_rata_2,rez_selectie_2)

    # Calcul solutie finala
    solutie_finala = []

    # Generarea frontului Pareto in urma sortarii solutiilor non-
dominante, parcurgand matricea de elemente selectate
    non_dominated_sorted_solution=[]
    for i in range(0, len(rez_selectie_2)):
      for j in range(0, len(rez_selectie_2[i])):
        non_dominated_sorted_solution.append(rez_selectie_2[i].index(rez_selecti
e_2[i][j]))
      #Aici se sorteaza propriu zis, solutiile non-
dominante in functie de lista de indecsi si crowding distance
      front22 = sortare(non_dominated_sorted_solution[:], valori_pentru_dist_ag
lomerare_2[i][:])
      for j in range(0, len(rez_selectie_2[i])):
        front = [rez_selectie_2[i][front22[j]]]
        front.reverse()
        for value in front:
          solutie_finala.append(value)
          if (len(solutie_finala) == dim_populatie):
            break
      if (len(solutie_finala) == dim_populatie):
        break
    solutie = [solutie_noua[i] for i in solutie_finala]
    numarul_generatiei = numarul_generatiei + 1
```

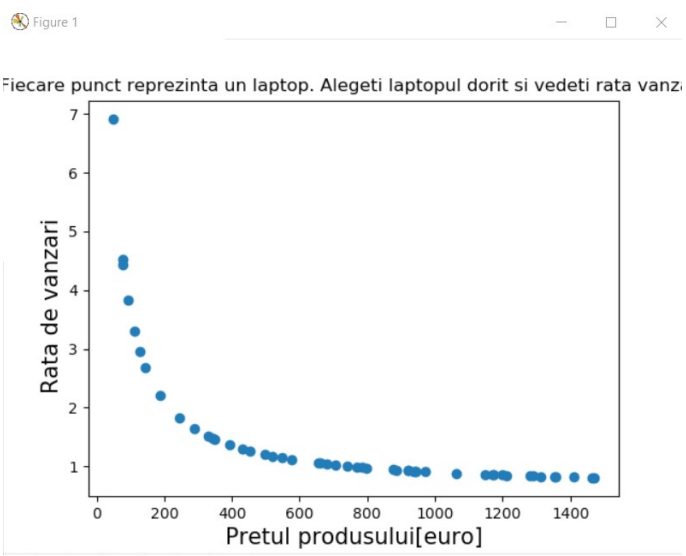
3. Rezultatele implementării

Valorile celor 20 de cromozomi cele mai optime pentru fiecare generatie

```
Cel mai bun front pentru generatia  96  este
0.368 0.045 0.429 0.701 0.775 0.577 0.389 0.065 0.136 0.138 0.435 0.335 0.367 0.011 0.185 0.599 0.835 0.907 0.207 0.809

Cel mai bun front pentru generatia  97  este
0.045 0.429 0.701 0.775 0.577 0.389 0.065 0.136 0.138 0.435 0.335 0.367 0.011 0.185 0.599 0.835 0.207 0.907 0.809 0.795

Cel mai bun front pentru generatia  98  este
0.429 0.701 0.775 0.577 0.389 0.065 0.136 0.138 0.435 0.335 0.367 0.011 0.185 0.599 0.835 0.207 0.809 0.907 0.795 0.919

Cel mai bun front pentru generatia  99  este
0.701 0.775 0.577 0.389 0.065 0.136 0.138 0.435 0.335 0.367 0.011 0.185 0.599 0.835 0.207 0.809 0.795 0.907 0.919 0.256
```

În imaginea din stânga este rulat algoritmul pentru o populație de 20, iar în cea din dreapta pentru dimensiunea populației de 50. S-a testat pentru același număr de generații: 100. În urma populației de 50, stocul de laptopuri este mult mai mare pentru aceleași specificații, și considerăm ca numărul optim pentru complexitatea problemei noastre și scopul ei (vizualizarea facilă a vanzarii unui laptop de către furnizor) este undeva la o dimensiune de 20-30.





4. Membrii echipei

Echipa a fost formată din 2 membri, iar sarcinile au fost împărțite astfel pentru fiecare:

Avarvarei Elena- Mădălina:

- Afișarea graficului cu frontul Pareto-optimal
- Funcțiile care contribuie la selecție, sortare și crowding-distance
- Contribuție în întocmirea documentației

Pintilie Elena-Cristiana:

- Structurarea funcției în care se desfășoară algoritmul NSGA-II (main)
- Funcția de crossover și mutație
- Contribuție în întocmirea documentației

5. Concluzii

Majoritatea aplicațiilor inginerești sau industriale presupun soluționarea unor probleme de optimizare multiobiectiv ce urmăresc optimizarea simultană a mai multor funcții obiectiv. Scopurile principale ale unei metode de optimizare multiobiectiv sunt obținerea unor aproximări cât mai bune pentru frontul optimal și distribuirea bună a soluțiilor de-a lungul acestuia. Algoritmii de acest tip sunt eficienți în sensul că produc soluții suboptimale într-un interval rezonabil de timp. Algoritmii evolutivi avantajează rezolvarea unei probleme privind un număr enorm de date și de aceea implementarea unui algoritm specializat nu ar fi fost soluția ideală.

Desigur, există și dezanvantaje, cum ar fi faptul că găsirea unei soluții bune într-un interval de timp mare înseamnă lipsa unei soluții bune într-un interval de timp acceptabil și că uneori sunt foarte lenți, mai lenți decât algoritmii specializați. [5]

6. Bibliografie

[1] Iulia Radulescu - REZOLVAREA UNOR PROBLEME DE OPTIMIZARE MULTI-OBIECTIV BAZATĂ PE ALGORITMI EVOLUTIVI : https://rria.ici.ro/wp-content/uploads/2015/06/06-art.-4-Radulescu-Iulia.pdf

[2] Figura 1: https://www.edupristine.com/blog/hadoop-mapreduce-framework, Accesat: 2020

[3] Florin Leon - Laborator Inteligență Artificială : http://florinleon.byethost24.com/lab_ia.htm

[4] Ashish Ghosh, Mrinal Kanti Das - Non-dominated Rank based Sorting Genetic Algorithms : https://www.isical.ac.in/~ash/Mrinal-fi-08.pdf

[5] METODE GENETICE DE OPTIMIZARE MULTIOBIECTIV, Corina Agrigoroaie http://doctorat.tuiasi.ro/doc/SUSTINERI_TEZE/AC/Agrigoroaie/2018_Teza_rezumat.pdf