Universitatea Tehnică “Gheorghe Asachi”
din Iași Facultatea de Automatică și
Calculatoare Domeniul Calculatoare și Tehnologia Informației
Specializarea Tehnologia Informației

Inteligența Artificială
Optimizarea Managementului Comenzilor Fast-Food utilizând algoritmul NSGA II

Studenții:
Ciobanu Maria-Denisa
Lupu George
Tăbușcă Codrina-Florentina

1. Descrierea problemei considerate

În acest proiect se analizează problema optimizării managementului comenzilor într-un restaurant de tip fast-food, în cazul nostru McDonald's, în care apar doua obiective contradictorii:

- Minimizarea costurilor făcute pentru angajați
- Minimizarea timpului mediu de așteptare al clienților

Fiind un algoritm multi-obiectiv vom avea două funcții obiectiv (ambele de minimizare):

1. COST: dorim o alocare a personalului în care sa utilizăm cât mai puțini angajați, dar păstrând eficiența pregătirii comenzilor.
2. TIMP DE AȘTEPTARE: dorim o alocare a personalului care să ne asigure un timp cât mai minim de așteptare al cumpărătorilor.

Relație contradictorie:
→ Puțini angajați ar duce la un timp mare de așteptare al clienților, iar un număr mare de angajați ar duce la un cost foarte mare pentru salariul acestora.
Scopul aplicației este de a descoperi setul de soluții care reprezintă cel mai bun compromis între aceste două obiective, ambele fiind de minimizare.

2. Aspecte teoretice privind algoritmul

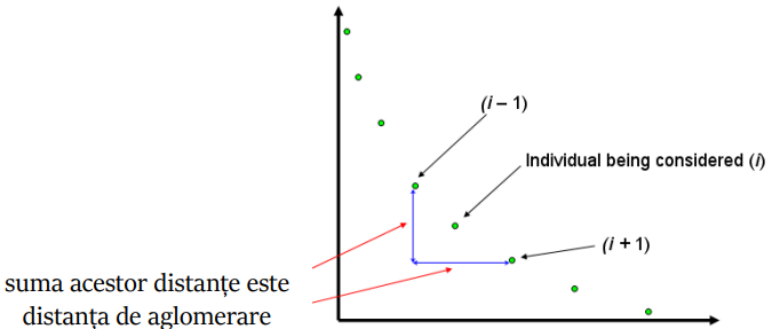
NSGA (Non-dominated Sorting Genetic Algorithm): reprezintă faptul ca acesta utilizează o tehnică de sortare bazată pe dominanță Pareto pentru clasificarea soluțiilor.

NSGA-II este un algoritm genetic de optimizare multi-obiectiv, utilizat pentru determinarea unui set de soluții Pareto în probleme cu obiective contradictorii. Scopul algoritmului este de a determina un set de soluții Pareto optime, acestea fiind niște compromisuri între funcțiile noastre obiectiv (în cazul nostru, Cost și Timp de Așteptare).

Concepte cheie:.

1. **Crowding Distance**
   Distanța de aglomerare este modul prin care ne asigurăm că soluțiile sunt bine distribuite.

   Calculul distanței de aglomerare:
   - Pentru fiecare obiectiv, soluțiile sunt sortate pe baza valorilor lor obiective.
   - Soluțiile de margine vor avea valoarea infinit (pentru a fi mereu păstrate)
   - Apoi, pentru fiecare solutie interna, se calculează distanța de înghesuire



2. **Elitismul**
   Elitismul este mecanismul prin care se asigură faptul că soluțiile bune găsite nu sunt pierdute în procesul de evoluție către generația următoare. În optimiziarea multi-obiectiv, acest concept garantează o convergență constantă către frontul Pareto optim. Spre deosebire de algoritmul de bază NSGA, elitismul este o îmbutățire adusă prin NSGAII.
   Funcționare:
   - **Combinarea populațiilor:** La fiecare iterație algoritmul unește populația actuală de părinți cu populația nouă de descendenți, rezultând o populație dublă de 2N.
   - **Competiție directă:** Prin această combinare, părinții cei mai buni, de elită, concurează cu proprii copii care pot fi mai buni sau mai slabi din cauza mutațiilor.
   - **Selecția supraviețuitorilor:** Sortarea non-dominată se aplică pe întreaga populație 2N indivizi. Sunt selectați pentru generația urmatoare doar primii N indivizi, începând cu Frontul 1.

3. **Fast NonDominated Sort**
   Sortarea non-dominată este o tehnică utilizată pentru a grupa soluțiile în diferite fronturi Pareto. O soluție este considerată **non-dominată** dacă nicio altă soluție din populație nu este mai bună la toate funcțiile obiectiv. Primul front Pareto (F1) conține soluțiile non-dominate, acestea fiind considerate cele mai bune.
   Funcționare:
   - Se va face o listă goală pentru fiecare front Pareto.
   - Se va calcula pentru fiecare soluție numărul de indivizi ce o domină și lista indivizilor pe care îi domină
   - Primele soluții găsite nedominate sunt adăugate în Frontul 1
   - Pentru următoarele fronturi:
     - Pentru fiecare soluție ce era dominată de p se decrementeaza contorul de dominanță a acesteia
     - Soluțiile ce ajung cu numărul de soluții ce o domină = 0 vor fi adăugate în frontul curent
   - Procesul se repetă până când toate soluțiile sunt atribuite unui front.

4. **DominaceComparer**
   Această funcție stabilește relația dintre doi indivizi.
   Funcționare:
   O soluție I1 este considerată mai bună decât o soluție I2 dacă sunt îndeplinite condițiile:

- Soluția I1 este cel puțin egală la toate obiectivele față de I2
- Soluția I1 este strict mai bună decât I2 la cel puțin un obiectiv

3. Modalitatea de rezolvare.

Implementarea acestui proiect a pornit de la structura laboratorului 8 cu algoritmi evolutivi, pe care am adaptat-o și extins-o pentru a satisface cerințele noastre.

Organizarea proiectului:

- Domain: conține entitățile de bază ale problemei: Individual si OptimizationProblem (unde definim constrangerile pentru soluția dorită)
- NSGA-II: Include interfețele și clasele care implementează logica specifică nsga: sortare non-dominată, distanță de aglomerare, creare fronturi pareto etc.
- Infrastructure: se implementează funcțiile obiectiv specifice cerinței noastre de fast-food.
- Application: locul în care sunt coordonate toate componentele.

4. Listarea părților semnificative din codul sursă însoțite de explicații și comentarii.

4.1 Sortarea Non-Dominată

```
public List<List<Individual>> FastNonDominatedSort(List<Individual> population)
    {
        List<List<Individual>> fronts = new List<List<Individual>>();

        //frontul 1 va fi cel mai bun
        List<Individual> F1 = new List<Individual>();

        //daca p domina q --> adaugam q in DominatedSet (pe cine domina p)
        //daca q domina p --> DominatedCount++ (cati inedivizi il domina pe p)

        foreach(var p in population)
        {
            p.DominationCount = 0;
            p.DominatedSet.Clear();

            foreach ( var q in population)
            {
                if (p == q) continue; //individul nu e comparat cu el insusi

                int result = _comparer.Dominates(p, q);

                if(result==1) //p domina q
                {
                    p.DominatedSet.Add(q);
                }else if ( result == -1) //q domina p
                {
                    p.DominationCount++;
```

```
            }
        }

        //indivizii nedominati de nimeni vor fi pusi in Frontul 1
        if (p.DominationCount == 0)
        {
            p.Rank = 1;
            F1.Add(p);
        }
    }

    fronts.Add(F1);

    //gasim fronturile urmatoare
    int i = 0; //indexul fontului curent ( 0 pt F1)
    while( fronts[i].Any())
    {
        List<Individual> Q = new List<Individual>(); //Q va fi urmatorul front
        foreach( var p in fronts[i])
        {
            foreach( var q in p.DominatedSet)
            {
                //decrementam DominationCount al lui q
                q.DominationCount--;

                //daca q nu mai e dominat de nimeni => se adauga in frontul curent
                if ( q.DominationCount == 0)
                {
                    q.Rank = i + 2;
                    Q.Add(q);
                }
            }
        }

        i++;
        if (Q.Any())
            fronts.Add(Q);
        else
            break;
    }
    return fronts;


}
```

4.2 Distanța de aglomerare

```
public void CrowdingDistanceAssignment(List<Individual> front)
      {
         if(front.Count <= 2)
         {
            foreach( var individual in front)
            {
               individual.CrowdingDistance = double.PositiveInfinity;

            }
            return;
         }

         foreach( var individual in front)
         {
            individual.CrowdingDistance = 0;

         }

         //calcul pt prima functie obiectiv
         var sortedByCost = front.OrderBy(i => i.Costs).ToList();

         //capetele vor avea avea infinit pt crowdingDistance
         sortedByCost.First().CrowdingDistance = double.PositiveInfinity;
         sortedByCost.Last().CrowdingDistance = double.PositiveInfinity;

         double costMin = sortedByCost.First().Costs;
         double costMax = sortedByCost.Last().Costs;
         double costRange = costMax - costMin;

         if( costRange > 0)
         {
            for(int i= 1; i < sortedByCost.Count-1; i++)
            {
               sortedByCost[i].CrowdingDistance +=
                  (sortedByCost[i + 1].Costs - sortedByCost[i - 1].Costs) / costRange;
            }
         }

         //calcul pt a doua functie obiectiv
         var sortedByTime = front.OrderBy(i => i.WaitingTime).ToList();

         //setam capetele la infinit (doar daca nu sunt deja setate de mai sus
         if (sortedByTime.First().CrowdingDistance != double.PositiveInfinity)
            sortedByTime.First().CrowdingDistance = double.PositiveInfinity;
         if (sortedByTime.Last().CrowdingDistance != double.PositiveInfinity)
```

```
            sortedByTime.Last().CrowdingDistance = double.PositiveInfinity;

        double timeMin = sortedByTime.First().WaitingTime;
        double timeMax = sortedByTime.Last().WaitingTime;
        double timeRange = timeMax - timeMin;

        if (timeRange > 0)
        {
            for ( int i = 1; i< sortedByTime.Count-1; i++)
            {
                sortedByTime[i].CrowdingDistance +=
                    (sortedByTime[i + 1].WaitingTime - sortedByTime[i - 1].WaitingTime) /
timeRange;
            }
        }
    }
```

4.3 DominanceComparer

```
public int Dominates(Individual p, Individual q)
    {
        if (p == null && q == null) return 0;
        if (p == null) return -1;
        if (q == null) return 1;

        bool pNotWorse = (p.Costs <= q.Costs) && (p.WaitingTime <= q.WaitingTime);
        bool qNotWorse = (q.Costs <= p.Costs) && (q.WaitingTime <= p.WaitingTime);

        bool pStrictBetter = (p.Costs < q.Costs) || (p.WaitingTime < q.WaitingTime);
        bool qStrictBetter = (q.Costs < p.Costs) || (q.WaitingTime < p.WaitingTime);

        if (pNotWorse && pStrictBetter) return 1;
        if (qNotWorse && qStrictBetter) return -1;

        return 0;
    }
```

4.4 Mutație

```
public void Mutate(Individual individual)
    {
        for(int i = 0; i < individual.Genes.Length; i++)
        {
```

```
            if (_rnd.NextDouble() < _problem.MutationRate)
            {
                //folosim proprietatile din OptimizationProblem
                int min = (int)_problem.MinEmployeesPerHour;
                int max= (int)_problem.MaxEmployeesPerHour;
                //se genereaza un nr aleatoriu intre min si max si am pus +1 deoarece vrem sa
exludem valoarea maxima
                int newEmployeeCount = _rnd.Next(min, max + 1);
                //inlocuim gena veche cu cea noua
                individual.Genes[i] = newEmployeeCount;
            }
        }
    }
```

4.5 Încrucișare

```
public List<Individual> Crossover(Individual parent1, Individual parent2)
    {
        var children = new List<Individual>();

        if (_random.NextDouble() >= _problem.CrossoverRate)
        {
            children.Add(Individual.Clone(parent1));
            children.Add(Individual.Clone(parent2));

            return children;
        }

        double alpha = _random.NextDouble();
        int GenesLength = parent1.Genes.Length;

        var child1 = new Individual(GenesLength, parent1.MinValues, parent1.MaxValues);
        var child2 = new Individual(GenesLength, parent2.MinValues, parent2.MaxValues);

        for (int i = 0; i < GenesLength; i++)
        {
            double value1 = alpha * parent1.Genes[i] + (1 - alpha) * parent2.Genes[i];
            double value2 = alpha * parent2.Genes[i] + (1 - alpha) * parent1.Genes[i];

            int min = (int)_problem.MinEmployeesPerHour;
            int max = (int)_problem.MaxEmployeesPerHour;

            //child1.Genes[i] = Math.Clamp((int)Math.Round(value1), min, max);
            //child2.Genes[i] = Math.Clamp((int)Math.Round(value2), min, max);
```

```
            //ne asiguram child1 este intre min si max
            int result1 = (int)Math.Round(value1);
            result1 = Math.Min(result1, max);
            result1 = Math.Max(result1, min);
            child1.Genes[i] = result1;

            //ne asiguram child2 este intre min si max
            int result2 = (int)Math.Round(value2);
            result2 = Math.Min(result2, max);
            result2 = Math.Max(result2, min);
            child2.Genes[i] = result2;
        }

        children.Add(child1);
        children.Add(child2);

        return children;
    }
```

4.6. Selecție

```
public Individual RunTournament(List<Individual> populatie, int marimeTurneu)
    {
        if (populatie == null || populatie.Count == 0)
            throw new ArgumentException("Populatia nu poate fi nula sau goala.");

        if (marimeTurneu <= 0) marimeTurneu = 1;

        Individual celMaiTop = null;

        for (int i = 0; i < marimeTurneu; ++i)
        {
            var candidat = populatie[_random.Next(0, populatie.Count)];

            if (candidat == null) continue;

            if (celMaiTop == null || EsteMaiBun(candidat, celMaiTop))
            {
                celMaiTop = candidat;
            }
        }

        return celMaiTop ?? populatie[0];
```

```
    }
```

4.7. Evaluate

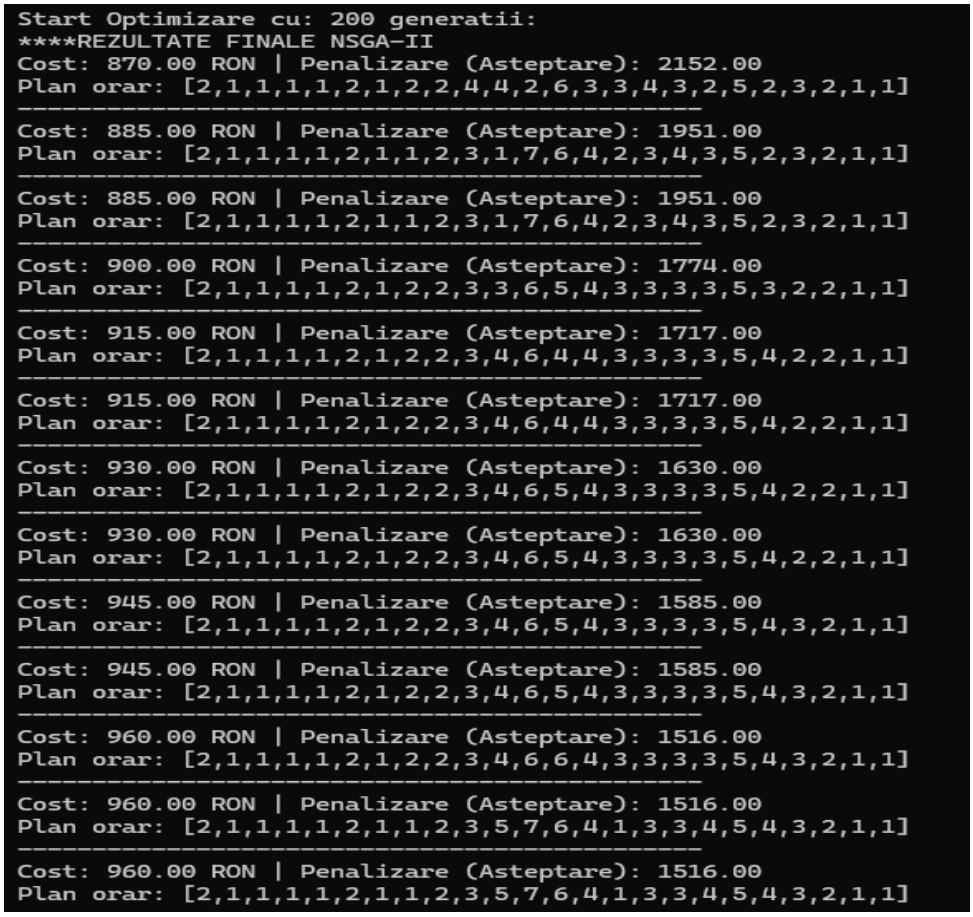
```
public void Evaluate(Individual individual)
    {
        // 1. Calcul Costuri: Suma Angajaților * Cost Orar
        individual.Costs = individual.Genes.Sum() * _problem.HourlyCost;

        // 2. Calcul Timp Așteptare (Penalizare)
        double totalPenalty = 0;
        // Presupunem că 1 angajat poate servi 3 clienți pe oră
        double serviceRate = 3.0;

        for (int i = 0; i < individual.Genes.Length; i++)
        {
            double demand = _customerDemand[i];
            double allocatedEmployees = individual.Genes[i];
            double capacity = allocatedEmployees * serviceRate;

            // Dacă capacitatea e mai mică decât cererea, se acumulează penalizare
            if (capacity < demand)
            {
                double deficit = demand - capacity;
                // Penalizare pătratică pentru a descuraja deficitele mari
                totalPenalty += deficit * deficit;
            }
        }
        individual.WaitingTime = totalPenalty;
    }
```

5. Rezultatele obținute prin rularea programului în diverse situații, capturi de ecran și comentarii asupra rezultatelor obținute.



Aceste puncte descriu începutul listei de rezultate, unde algoritmul prioritizează banii:

- **Minimizarea Obiectivului 1 (Costuri):** Soluțiile afișate reprezintă extremul inferior al costurilor (~870 - 960 RON), obținut prin utilizarea numărului minim de angajați permis de constrângeri (majoritar valori de 1 și 2 în planul orar).
- **Compromisul Calității (Trade-off):** Reducerea drastică a costurilor generează o penalizare uriașă pentru Timpul de Așteptare (>1500), indicând faptul că cererea clienților depășește constant capacitatea de servire.
- **Structura Cromozomului:** Vectorul de gene este dominat de valori minime, demonstrând că algoritmul a explorat marginea inferioară a spațiului de căutare pentru a satisface primul obiectiv.

```
-------------------------------------------
Cost: 1785.00 RON | Penalizare (Asteptare): 8.00
Plan orar: [1,1,2,2,2,2,2,4,5,7,8,10,9,8,7,6,7,8,8,7,5,4,1,3]
-------------------------------------------
Cost: 1800.00 RON | Penalizare (Asteptare): 5.00
Plan orar: [1,1,2,2,2,2,2,4,5,7,8,10,9,8,7,6,7,7,8,7,5,4,3,3]
-------------------------------------------
Cost: 1800.00 RON | Penalizare (Asteptare): 5.00
Plan orar: [1,1,2,2,2,2,2,4,5,7,8,10,9,8,7,6,7,7,8,7,5,4,3,3]
-------------------------------------------
Cost: 1845.00 RON | Penalizare (Asteptare): 2.00
Plan orar: [1,1,2,2,2,3,3,4,5,7,9,10,10,9,7,6,7,7,8,7,5,4,2,2]
-------------------------------------------
Cost: 1845.00 RON | Penalizare (Asteptare): 2.00
Plan orar: [1,1,2,2,2,3,3,4,5,7,9,10,10,9,7,6,7,7,8,7,5,4,2,2]
-------------------------------------------
Cost: 1860.00 RON | Penalizare (Asteptare): 1.00
Plan orar: [1,1,2,2,2,2,2,4,5,7,9,10,10,9,7,6,7,8,8,7,5,4,3,3]
-------------------------------------------
Cost: 1860.00 RON | Penalizare (Asteptare): 1.00
Plan orar: [1,1,2,2,2,2,2,4,5,7,9,10,10,9,7,6,7,8,8,7,5,4,3,3]
-------------------------------------------
Cost: 1875.00 RON | Penalizare (Asteptare): 0.00
Plan orar: [1,1,2,2,1,3,2,4,5,7,9,10,10,9,7,6,7,8,9,7,5,4,3,3]
-------------------------------------------
Cost: 1875.00 RON | Penalizare (Asteptare): 0.00
Plan orar: [1,1,2,2,1,3,2,4,5,7,9,10,10,9,7,6,7,8,9,7,5,4,3,3]
-------------------------------------------
```

Aceste puncte descriu finalul listei, unde algoritmul prioritizează calitatea serviciului:

- **Convergența către Optimul Global (Penalty 0):** Algoritmul a reușit să identifice configurația ideală în care penalizarea ajunge la **0.00**, ceea ce înseamnă că cererea de clienți este satisfăcută integral în fiecare interval orar.
- **Adaptare Cererii:** Analiza vectorului de gene (ex: valori de 10, 9, 8 la mijloc) arată că algoritmul a "învățat" tiparul de trafic definit în cod (`_customerDemand`), alocând resurse exact acolo unde este nevoie.
- **Costul Performanței:** Pentru a elimina complet timpii de așteptare, costul salarial se dublează față de minim (~1875 RON), ilustrând clar prețul calității în cadrul analizei multi-obiectiv.

- **Stabilitatea Soluției:** Repetiția acelorași valori spre finalul listei indică faptul că populația a convergit și s-a stabilizat pe cele mai bune soluții posibile pentru acest set de parametri.

6. Concluzii.

În urma implementării acestui proiect folosind algoritmul NSGA-II pentru rezolvarea problemei noastre cu fast-food, s-au concluzionat următoarele idei:

- Algoritmul nu a oferit o singură soluție cea mai buna, ci un set de soluții optime ce reprezintă un compromis între cost și timp de așteptare.
- S-a demonstrat faptul că nu putem avea o organizare perfectă care să poată minimiza ambele scopuri, funcțiile noastre fiind în contradictoriu: minimizarea costurilor ar duce la un timp mare de așteptare al clienților.
- Utilizarea selecției prin turneu, bazată pe rang și distanța de îngrămădire a asigurat o diversitate a soluțiilor.

Prin acest proiect am reușit să facem o legătura între realitatea dintr-un restaurant fast-food și codul din spatele algoritmului. Algoritmul NSGA-II face trecerea de la resursele umane la eficiența operațională obținută prin diferite funcții.

7. Bibliografie.

https://florinleon.byethost24.com/Curs_IA/IA05_Optimizare2.pdf?i=1
https://www.geeksforgeeks.org/deep-learning/non-dominated-sorting-genetic-algorithm-2-nsga-ii/
https://www.sciencedirect.com/topics/engineering/non-dominated-sorting-genetic-algorithm-ii

8. O listă cu ce a lucrat fiecare membru al echipei.

Ciobanu Maria-Denisa:

- A creat entitățile de bază în folderul Domain, definind structura clasei Individual și parametrii problemei in OptimizationProblem
- Implementare operatori genetici: UniformMutation si TournamentSelection
- A stabilit ierarhia de foldere și interfețe, asigurând comunicarea între module

Tăbușcă Codrina-Florentina

- A implementat functiile specifice NSGA: FastNonDominated-Sort, CrowdingDistance si DominanceComparer
- A implementat mecanismul de încrucișare în AritmeticCrossover

Lupu George:

- A implementat folderul Infrastructure cu clasa FastFoodFitnessEvaluator
- A definit functionarea logicii specifice algorimtului in NSGARunner
- S-a ocupat de rularea simularilor și de implementarea datelor finale