Universitatea Tehnică „Gheorghe Asachi” din Iași

Facultatea de Automatică și Calculatoare

Domeniul: Tehnologia Informației

Algoritmul NSGA-II

pentru optimizare multi-obiectiv

Proiect la disciplina

Inteligența Artificială

Studenți: Muraru Alexandru

Enachi Vasile

Ilioi Alexandru

Anul: 4

Grupa: 1409B

Contents

Capitolul 1. Descrierea problemei considerate

Problema clasică a rucsacului este o problemă de optimizare combinatorică.

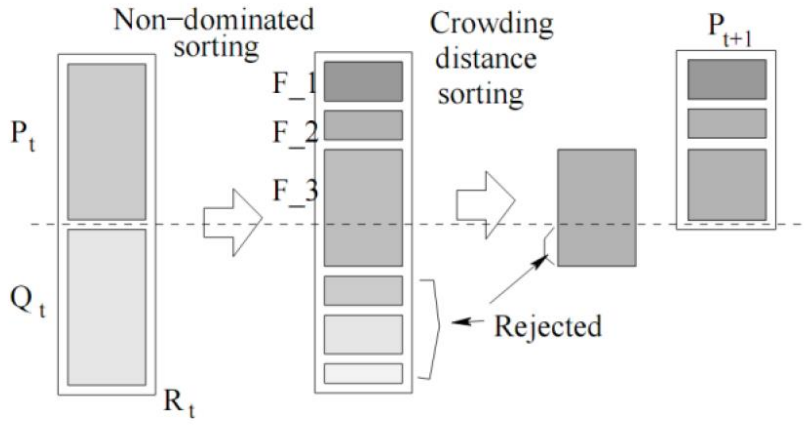
Dată fiind o mulțime de elemente, fiecare item are o greutate și o valoare. Scopul problemei acela de a determina combinația optimă de elemente selectate, dată fiind o greutate maximă, astfel încât valoarea itemelor să fie maximă.

Problema rezolvată în proiectul nostru reprezintă o extindere a problemei rucsacului în sensul că valoarea unui item este reprezentată de perechea (timp, cost). În cazul clasic optimizare se realizează pe o singură dimensiune, dar în cazul nostru avem de a face cu o problemă multi-dimensională, care necesită o abordare ușor diferită și de aceea am ales rezolvarea ei prin tehnici de inteligență artificială.

Capitolul 2. Aspecte teoretice privind algoritmul

Optimizarea multi-obiectiv (cunoscută și sub numele de programare multi-obiectivă, optimizarea vectorială, optimizarea multicriteriilor, optimizarea multi-atributului sau optimizarea Pareto) este o zonă de luare a deciziilor cu criterii multiple care se referă la probleme de optimizare matematică care implică mai multe funcții obiective care urmează să fie optimizate simultan.

NSGA-II reprezintă un algoritm ce face parte din categoria algoritmilor de optimizare multi-obiectiv.



Acesta este alcătuit din 4 pași de execuție:

- Producerea populației viitoare(Qt) prin operațiile genetice aplicate părinților(Pt) și reinserarea lor în vectorul populației(Rt):
  - Selecție (în cazul prezent prin Turnir)
  - Crossover
  - Mutație
  - Reinserție
- Sortarea Non-Dominantă, în urma căreia populația este împărțită în fronturi, urmând ca parte din acestea (numărul elementelor din Pt initial) să refacă vectorul parinte(Pt)
- Aplicarea Crowding Distance Sorting ultimului front ce trebuie împărțit înainte de reinserare
- Reinserarea datelor în populația părinte(Pt)

Capitolul 3. Modalitatea de rezolvare

Au fost create 4 module:

- **Algorithm –** implementarea funcționalității algritmului NSGA-II
- **CLI** – rularea algoritmului din command line folosind un mock data
- **GUI** – rularea algoritmului din interfața grafică, cu posibilitatea de adăugare iteme noi, vizualizare rezultate
- **Tests** – modul pentru testarea unor funcționalități a cromozomilor

Structurile de date construite și folosite sunt:

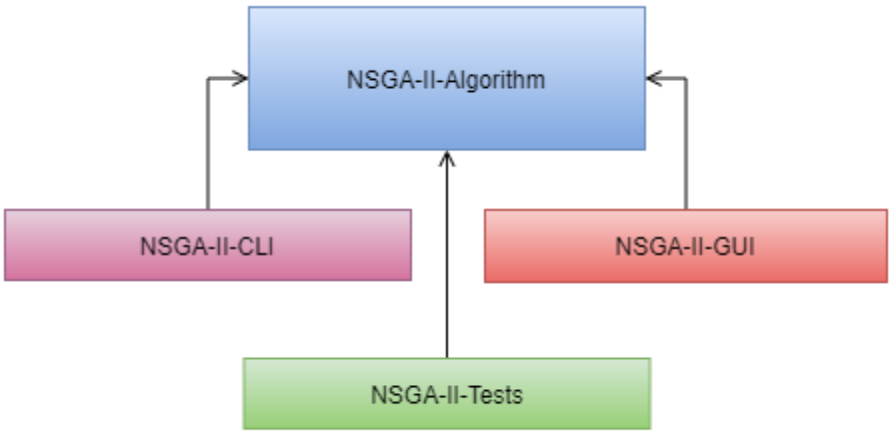
Item – conține informațiile despre denumirea acelui item, valoarea, greutatea și timpul necesar colectării acestuia.

Chromosome – conține un tablou de bool-uri care semnifică dacă un item dintr-o listă de iteme este selectat sau nu. Conține funcții de calculare a valorii fitness după valoarea și timp necesar colectării

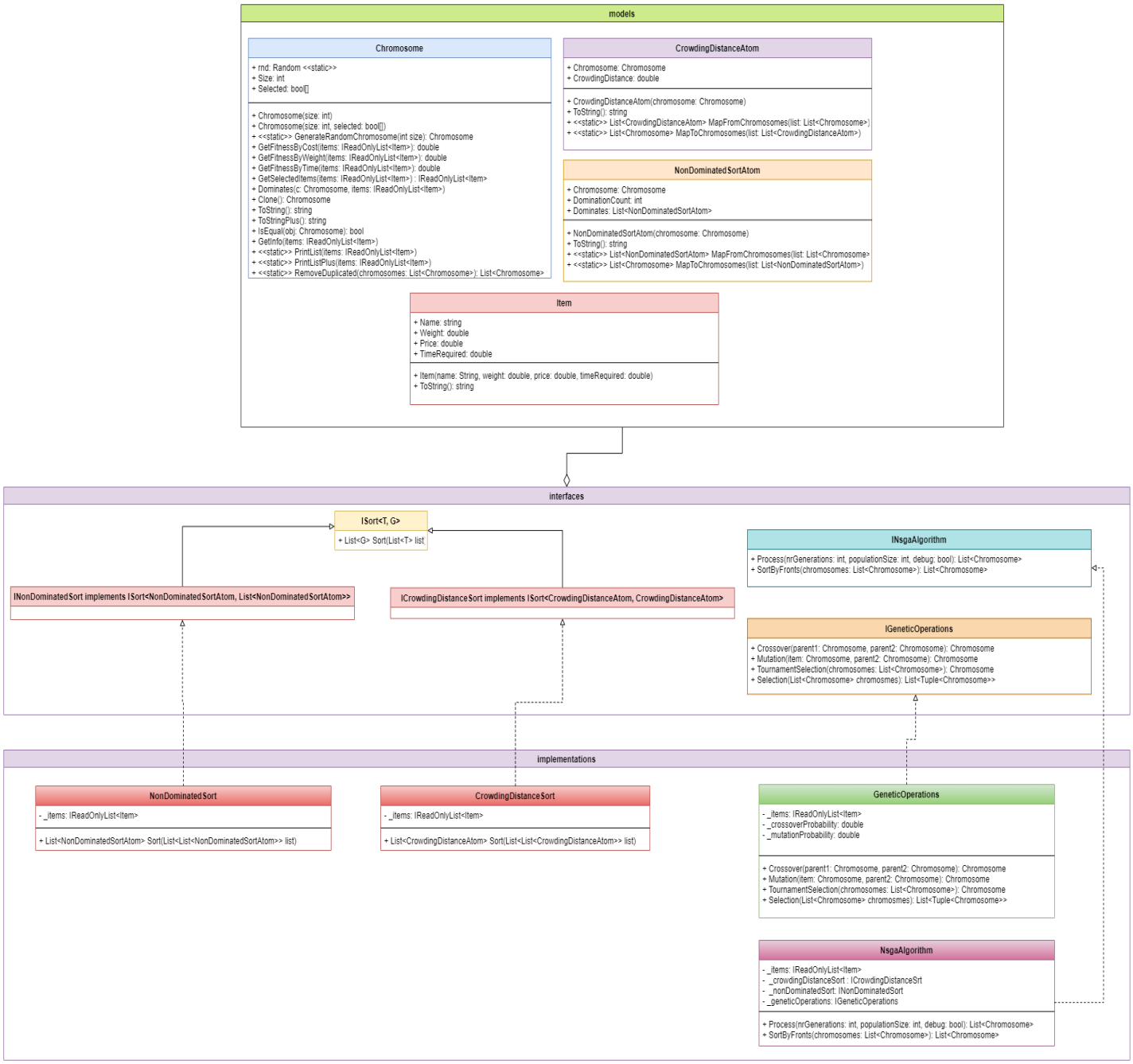
CrowdingDistanceAtom – conține un cromozom țintă și o distanță de aglomerare.

NonDominatedSort – conține un cromozom țintă, dominationCount care arată de câți atomi este dominat și dominates care reprezintă o listă de atomi dominați.

3.1 Interacțiunea modulelor



3.2 Diagrama de clase a modulului de algoritm



Capitolul 4. Analiza codului algoritmului

4.1 Sortarea pe fronturi

1. Primește o listă de cromozomi
2. Cromozomii sunt sortați pe fronturi folosind NonDominatedSort
3. Este returnată lista de fronturi

```
/// <summary>
/// Sort the list of chromosomes in fronts
/// </summary>
/// <param name="chromosomes">List of chromosomes</param>
/// <returns>List of fronts</returns>
0+1 usages  ALEX Alex *
public List<List<Chromosome>> SortByFronts(List<Chromosome> chromosomes)
{
    var fronts :List<List<...>> = _nonDominatedSort.Sort( list: NonDominatedSortAtom.MapFromChromosomes(chromosomes));
    return fronts.Select(NonDominatedSortAtom.MapToChromosomes).ToList();
}
```

4.2 Încrucișarea

1. Primește ca parametru 2 cromozomi
2. Este generat un index al genei în intervalul [0..gene_size]
3. Se creează un nou cromozom care conține până la indexul selectat genele de la primul părinte și după index acel index – genele de la al doilea părinte

```
/// <summary>
/// Select a slice index between 0..parents.genes.size and do the crossover between them
/// </summary>
/// <param name="parent1">Parent one chromosome</param>
/// <param name="parent2">Parent one chromosome</param>
/// <returns>New cross-over-ed chromosome</returns>
0+1 usages  enaki
public Chromosome Crossover(Chromosome parent1, Chromosome parent2)
{
    if (parent1.Size != parent2.Size)
        throw new Exception( message: "Size of chromosome should be the same");

    var child = new Chromosome(parent1.Size);

    var sliceIndex :int = rnd.Next(parent1.Size);
    for (var index = 0; index < sliceIndex; ++index)
        child.Selected[index] = parent1.Selected[index];

    for (var index :int = sliceIndex; index < parent1.Size; ++index)
        child.Selected[index] = parent1.Selected[index];

    return child;
}
```

4.3 Mutația

1. Primește ca parametru un cromozom
2. Iterează prin genele acelui cromozom și le modifică cu o probabilitate de mutație
3. Cromozomul generat este returnat

```
/// <summary>
/// Iterate through each gene of the chromosome and mutates it.
/// </summary>
/// <param name="item">Chromosome to be mutated</param>
/// <returns>A mutated chromosome with the specific probability for each gene</returns>
0+1 usages  enaki
public Chromosome Mutation(Chromosome item)
{
    var clone :Chromosome  = item.Clone();
    for (var i = 0; i < item.Size; ++i)
    {
        if (rnd.NextDouble() >= _mutationProbability)
        {
            clone.Selected[i] = !clone.Selected[i];
        }
    }
    return clone;
}
```

4.4 Selecția pe o listă de cromozomi

1. Primește o listă de cromozomi
2. Se creează o listă de n perechi tuple, unde n este numărul de cromozomi din lista parametru
3. Perechea este creată prin metoda turnirului

```
/// <summary>
/// This functions return a list of pair of parents of the same length as the input list using tournament method
/// </summary>
/// <param name="chromosomes">List of chromosomes</param>
/// <returns>A list of pairs of selected parents using the tournament method</returns>
0+1 usages  enaki
public List<Tuple<Chromosome, Chromosome>> Selection(List<Chromosome> chromosomes)
{
    var result = new List<Tuple<Chromosome, Chromosome>>();

    for (var i = 0; i < chromosomes.Count; ++i)
    {
        result.Add( item: Tuple.Create(TournamentSelection(chromosomes), TournamentSelection(chromosomes)));
    }
    return result;
}
```

4.5 Selecția prin turnir

1. Primește o listă de cromozomi
2. Sunt selectați noi indecși aleator din lista de cromozomi și este returnat cromozomul dominant
3. Dacă nici unul dintre cromozomi nu este dominant, este returnat cu o probabilitate de 50% unul dintre ei

```
/// <summary>
/// This function Select a chromosome using the tournament method
/// </summary>
/// <param name="chromosomes">List of chromosomes</param>
/// <returns>A chromosome selected using tournament method</returns>
2 usages  enaki
public Chromosome TournamentSelection(List<Chromosome> chromosomes)
{
    var idx1 :int  = rnd.Next(0, chromosomes.Count);
    var idx2 :int  = rnd.Next(0, chromosomes.Count);

    var dominate :int  = chromosomes[idx1].Dominates(chromosomes[idx2], _items);
    switch (dominate)
    {
    case 1:
        return chromosomes[idx1];
    case -1:
        return chromosomes[idx2];
    default:
        return rnd.NextDouble() < 0.5 ? chromosomes[idx1]: chromosomes[idx2];
    }
}
```

4.6 Crowding Distance Sort

1. Primește o listă de modele de tip CrowdingDistanceAtom
2. Atomilor care au cost maxim și celor care au timp minim le este setată distanța de aglomerare cu valoarea Infinit, asigurându-ne că vor fi transmiși în populația următoare
3. Este calculat crowding distance-ul pentru fiecare atom în parte și se returnează o listă descrescătoare după distanța de aglomerare

```
/// <summary>
/// This function updates their crowding distance values
/// </summary>
/// <param name="list">List of crowdingDistanceAtoms</param>
/// <returns>List of crowdingDistanceAtoms with the distance updated</returns>
2+3 usages  enaki
public List<CrowdingDistanceAtom> Sort(List<CrowdingDistanceAtom> list)
{
    var maxByCost :List<CrowdingDistanceAtom>  = list.MaxBy( selector: atom => atom.Chromosome.GetFitnessByCost(_items)).ToList();
    var minByCost :List<CrowdingDistanceAtom>  = list.MinBy( selector: atom => atom.Chromosome.GetFitnessByCost(_items)).ToList();
    // cei cu max ByCost, le setam distantele pe infinit, asigurandu-ne ca sunt transmisi mai departe
    foreach (var atom in maxByCost)
    {
        atom.CrowdingDistance = double.PositiveInfinity;
    }

    var maxByTime :IExtremaEnumerable<CrowdingDistanceAtom>  = list.MaxBy( selector: atom => atom.Chromosome.GetFitnessByTime(_items));
    var minByTime :IExtremaEnumerable<CrowdingDistanceAtom>  = list.MinBy( selector: atom => atom.Chromosome.GetFitnessByTime(_items));
    // cei cu min ByTime, le setam distantele pe infinit, asigurandu-ne ca sunt transmisi mai departe
    foreach (var atom in minByTime)
    {
        atom.CrowdingDistance = double.PositiveInfinity;
    }

    var timeDifference :double  = maxByTime.First().Chromosome.GetFitnessByTime(_items) -
                                  minByTime.First().Chromosome.GetFitnessByTime(_items);
    var costDifference :double  = maxByTime.First().Chromosome.GetFitnessByCost(_items) -
                                  minByTime.First().Chromosome.GetFitnessByCost(_items);

    //deoarece sunt pe acelasi front nu conteaza daca sortam dupa Cost sau dupa Time
    var sortedAtomsByFitness :List<CrowdingDistanceAtom>  = list.OrderBy( keySelector: atom => atom.Chromosome.GetFitnessByCost(_items)).ToList();

    for (var i = 1; i < sortedAtomsByFitness.Count - 1; ++i)
    {
        sortedAtomsByFitness[i].CrowdingDistance =
            (sortedAtomsByFitness[i + 1].Chromosome.GetFitnessByCost(_items) -
            sortedAtomsByFitness[i - 1].Chromosome.GetFitnessByCost(_items))/costDifference;
        sortedAtomsByFitness[i].CrowdingDistance +=
            (sortedAtomsByFitness[i + 1].Chromosome.GetFitnessByTime(_items) -
            sortedAtomsByFitness[i - 1].Chromosome.GetFitnessByTime(_items)) / timeDifference;
    }

    return sortedAtomsByFitness.OrderByDescending( keySelector: atom => atom.CrowdingDistance).ToList();
}
```

4.7 Non Dominated Sort

1. Primește o listă de atomi de tip NonDominatedSortAtom
2. Se actualizează indicele de tip DominationCount si lista de Dominates din cadrul unui atom
3. Se separă pe fronturi lista de atomi folosindu-ne de parametrii specificați mai sus
4. Se returnează lista de fronturi

```
/// <summary>
/// Sort the atom list to fronts
/// </summary>
/// <param name="list">List nonDominatedSortAtom</param>
/// <returns>Sorted List of fronts</returns>
3+3 usages  enaki +1
public List<List<NonDominatedSortAtom>> Sort(List<NonDominatedSortAtom> list)
{
    var frontList = new List<List<NonDominatedSortAtom>>();

    // update the domination count and dominates list
    for (var i = 0; i < list.Count-1; ++i)
    {
        for (var j int  = i+1; j < list.Count; ++j)
        {
            var dominateResult int  = list[i].Chromosome.Dominates(list[j].Chromosome, _items);
            switch (dominateResult)
            {
                case 1:
                    list[j].DominationCount++;
                    list[i].Dominates.Add(list[j]);
                    break;
                case -1:
                    list[i].DominationCount++;
                    list[j].Dominates.Add(list[i]);
                    break;
                default:
                    break;
            }
        }
    }

    // creating the fronts according to domination count and dominates list
    var running = true;
    while (running)
    {
        var currentFront = new List<NonDominatedSortAtom>();
        running = false;
        var dominatedList = new List<NonDominatedSortAtom>();
        for (var index = 0; index < list.Count; ++index)
        {
            if (list[index].DominationCount == 0)
            {
                running = true;
                list[index].DominationCount = -1;
                currentFront.Add(list[index]);

                foreach (var dominated in list[index].Dominates)
                {
                    dominatedList.Add(dominated);
                }
            }
        }
        foreach (var dominated in dominatedList)
        {
            dominated.DominationCount--;
        }

        if (currentFront.Count > 0)
        {
            frontList.Add(currentFront);
        }
    }

    return frontList;
}
```

4.8 Algoritmul NSGA-II

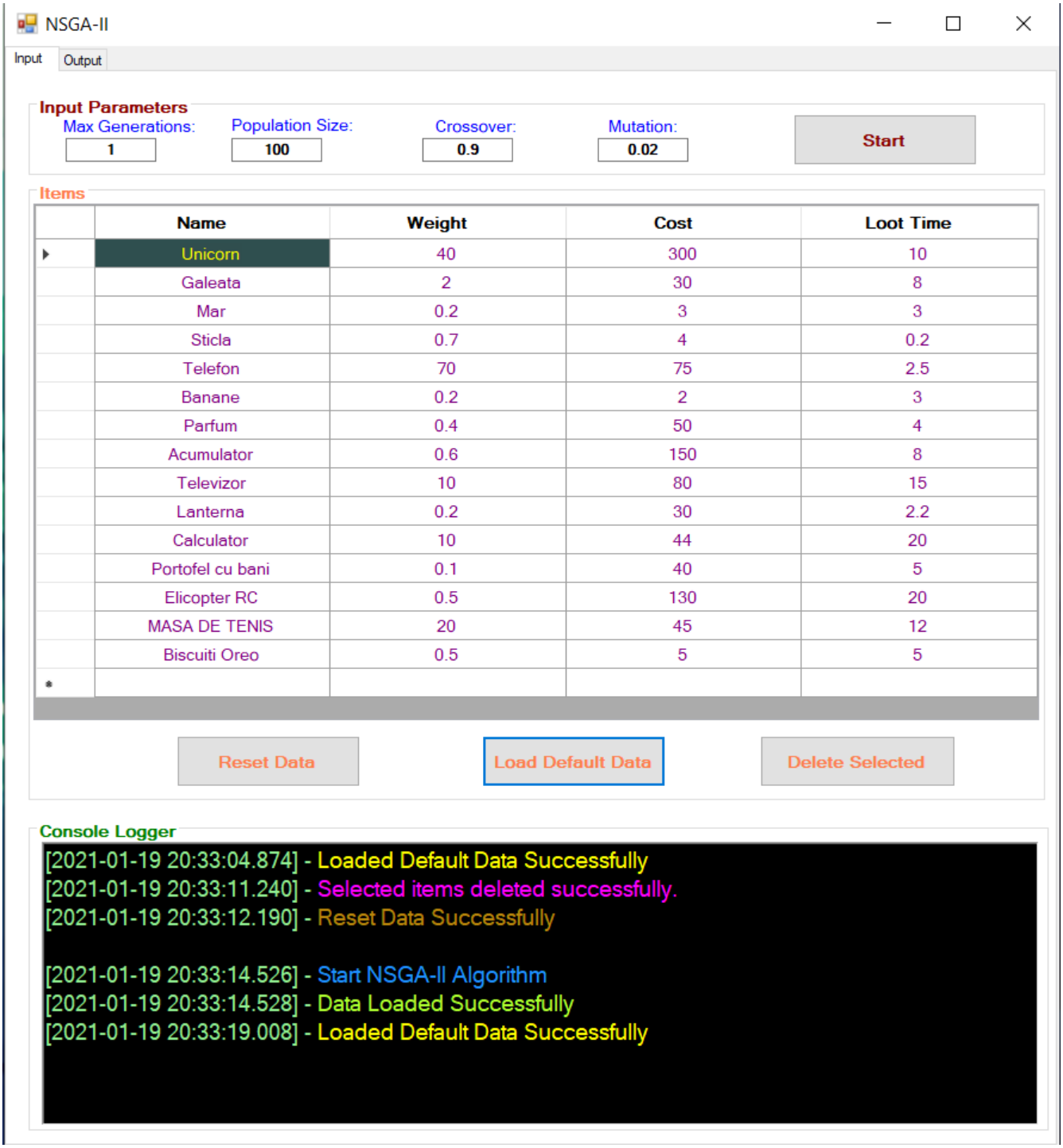
1. Primește numărul de generații și dimensiunea populației
2. Se generează populația inițială de cromozomi
3. La fiecare generație se repetă următoarele operații:
   - selectează perechile de părinți prin tournament
   - încrucișare și generare de copii
   - mutația asupra copiilor și inserarea lor în populație
   - sortarea non dominantă pe lista de cromozomi
   - sortarea de aglomerare pentru ultimul front de inserat în populație necesar stabilirii dimensiunii inițiale a populației
4. Returnează lista de cromozomi în urma evoluției

```
/// <summary>
/// Generate a list of chromosomes.
/// Repeat nrGenerations the follwing:
/// - Select parents with tournament method
/// - Crossover
/// - Mutation
/// - Append generate children to the current population
/// - Non dominated Sort
/// - Crowding Distance Sort for the last front to be inserted in order to complete the newPopulation to the last size
/// </summary>
/// <param name="nrGenerations">Number of generations for evolution</param>
/// <param name="populationSize">Population Size of Chromosome</param>
/// <param name="debug">Display Info Variable</param>
/// <returns>A list of chromosomes</returns>
1+1 usages  enaki
public List<Chromosome> Process(int nrGenerations, int populationSize, bool debug=false)
{
    var chromosomes = new List<Chromosome>();
    for (var i = 0; i < populationSize; i++)
    {
        chromosomes.Add( item: Chromosome.GenerateRandomChromosome(_items.Count));
    }

    if (populationSize > Math.Pow(2, _items.Count))
    {
        populationSize = (int) (Math.Pow(2, _items.Count) / 2);
    }

    //chromosomes = Chromosome.RemoveDuplicated(chromosomes).Take(Math.Min(populationSize, chromosomes.Count)).ToList();

    for (var currentGeneration = 0; currentGeneration < nrGenerations; ++currentGeneration)
    {
        for (int i = chromosomes.Count; i < populationSize; ++i)
        {
            chromosomes.Add( item: Chromosome.GenerateRandomChromosome(_items.Count));
        }
        if (debug)
            Console.WriteLine($"\n############### GENERATION {currentGeneration+1}###########");

        //TournamentSelection of Parents
        var selectedPairParents :List<Tuple<...>>  = _geneticOperations.Selection(chromosomes);
        if (debug)
            Console.WriteLine($"\nSelected Pair Parents size: {selectedPairParents.Count}");
```

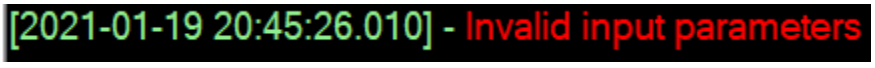
```
            //Crossover
            var children = new List<Chromosome>();

            foreach (var selectedParents :Tuple<Chromosome,Chromosome> in selectedPairParents)
            {
                children.Add( item: _geneticOperations.Crossover(selectedParents.Item1, selectedParents.Item1));
            }

            if (debug)
            {
                Console.WriteLine($"\nChildren size: {children.Count}");
                Chromosome.PrintList(children);
            }

            //Mutation
            children = Chromosome.RemoveDuplicated( chromosomes: children.Select(child :Chromosome => _geneticOperations.Mutation(child)).ToList());
            if (debug)
            {
                Console.WriteLine($"\nAfter Mutation Children size: {children.Count}");
                Chromosome.PrintList(children);
            }

            //New Population
            chromosomes.AddRange(children);
            //chromosomes = Chromosome.RemoveDuplicated(chromosomes).Take(Math.Min(populationSize, chromosomes.Count)).ToList();
            if (debug)
                Console.WriteLine($"\nNew Population size: {chromosomes.Count}");

            //Non Dominated Sort
            if (debug)
                Console.WriteLine($"\n############ NON DOMINATED SORT ###############");
            var fronts :List<List<...>> = _nonDominatedSort.Sort( list: NonDominatedSortAtom.MapFromChromosomes(chromosomes));

            if (debug)
                for (int i = 0; i < fronts.Count; ++i)
                {
                    Console.WriteLine($"\r\nFront {i+1} front size {fronts[i].Count}");
                    Chromosome.PrintListPlus(NonDominatedSortAtom.MapToChromosomes(fronts[i]), _items);
                }

            var newPopulation = new List<Chromosome>();

            var currentFront = 0;
            while (currentFront < fronts.Count && newPopulation.Count + fronts[currentFront].Count <= populationSize)
            {
                newPopulation.AddRange( collection: NonDominatedSortAtom.MapToChromosomes(fronts[currentFront]));
                currentFront++;
            }

            if (debug)
            {
                Console.WriteLine($"Front for crowding distance {currentFront}");
                Console.WriteLine($"Current new Population Size {newPopulation.Count}");

                //Crowding Distance Sort
                Console.WriteLine($"\n############ CROWDING DISTANCE SORT ###############");
            }

            var selectedAtoms :List<CrowdingDistanceAtom> = _crowdingDistanceSort.Sort(
                 list: CrowdingDistanceAtom.MapFromChromosomes(
                     list: NonDominatedSortAtom.MapToChromosomes(fronts[currentFront])));
            if (debug)
                Console.WriteLine($"Current Selected Atoms {selectedAtoms.Count}");

            newPopulation.AddRange( collection: CrowdingDistanceAtom.MapToChromosomes(selectedAtoms.Take(populationSize - newPopulation.Count).ToList()));
            if (debug)
                Chromosome.PrintListPlus(newPopulation, _items);

            chromosomes = newPopulation;
            chromosomes = Chromosome.RemoveDuplicated(chromosomes).Take(Math.Min(populationSize, chromosomes.Count)).ToList();

        }
        return chromosomes;
    }
}
```

Capitolul 5. GUI

Partea de interfață grafică a fost implementată folosind Windows Forms, iar pe partea de analiză a rezultatului s-a folosi pachetul **Live Charts**.



Au fost tratate și erorile de input, user-ul fiind atenționat de date invalide în console logger.



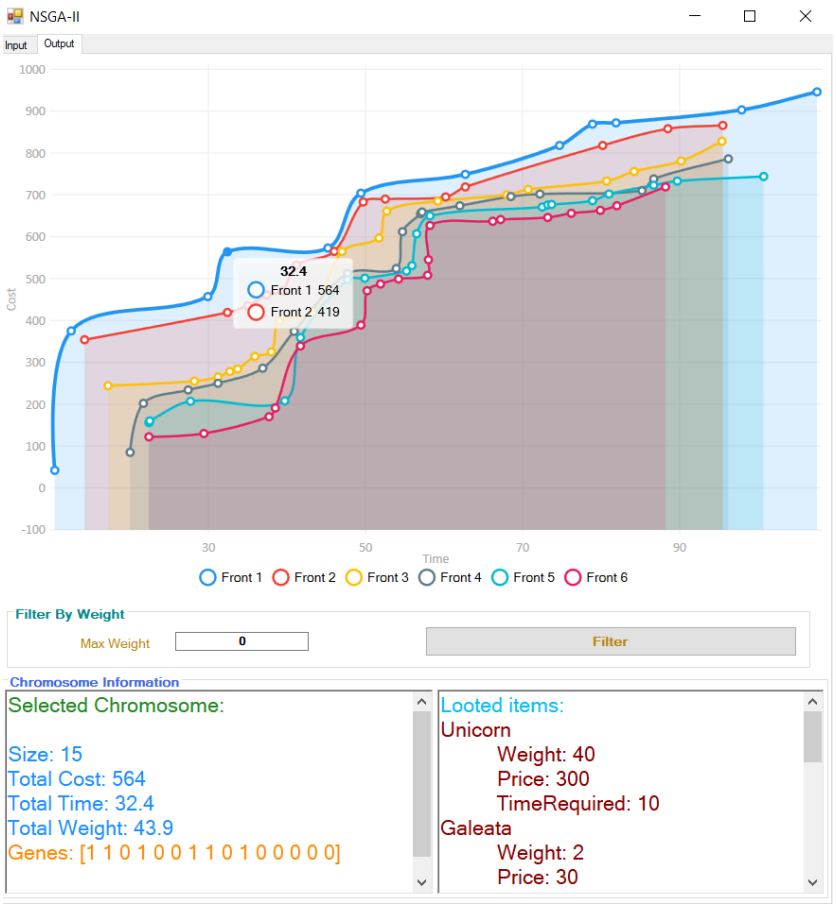
Funcționalități interfața grafică:

- **console logger -** pentru înregistrarea fiecărei operații efectuate
- **adăugare item în lista de itemi**
- **resetare lista de itemi**
- **încărcare lista de itemi impliciți**
- **șterge itemi selectați**
- **setare parametri algoritm**
- **vizualizare fronturi după executarea algoritmului**
- **filtrare cromozomi după greutate**
- **vizualizare detalii a unui cromozom din populația finală**

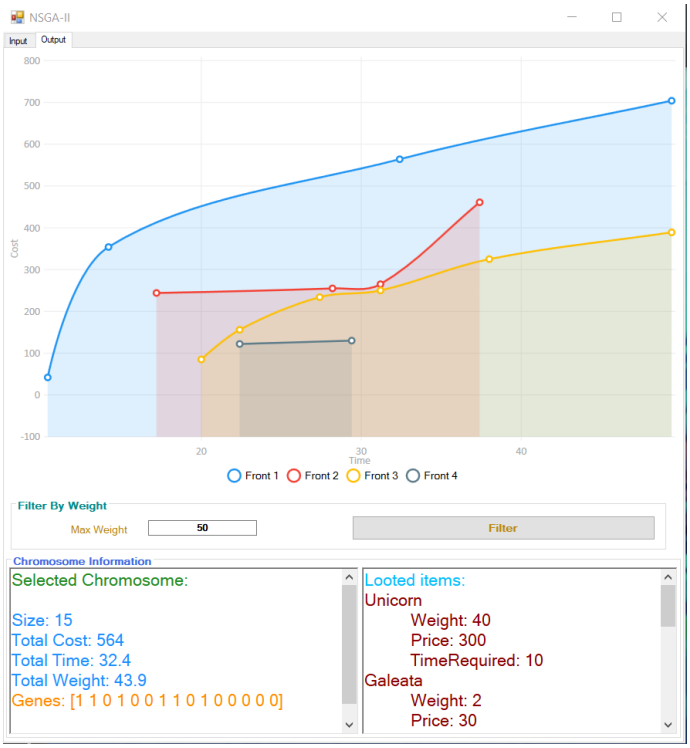
Capitolul 6. Rezultatele obținute prin rularea programului în diverse situații

Datele implicite pot fi vizualizate în figura de la capitolul GUI.

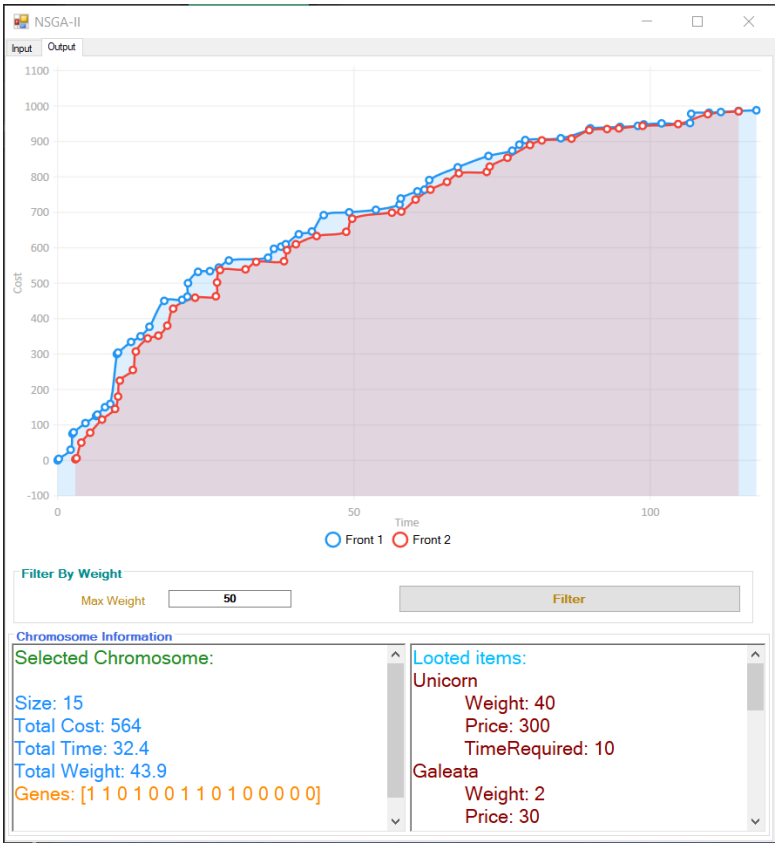
1. **Rulare algoritm cu date inițiale implicite, o singură generație, fără filtrare după greutate**



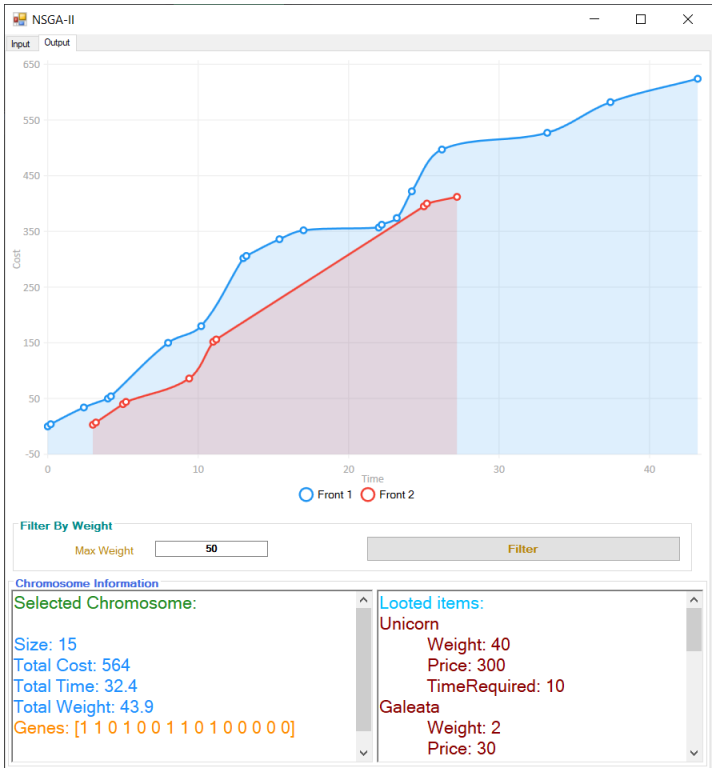
2. **Rulare algoritm cu date inițiale implicite, o singură generație, cu filtrare greutate < 50**



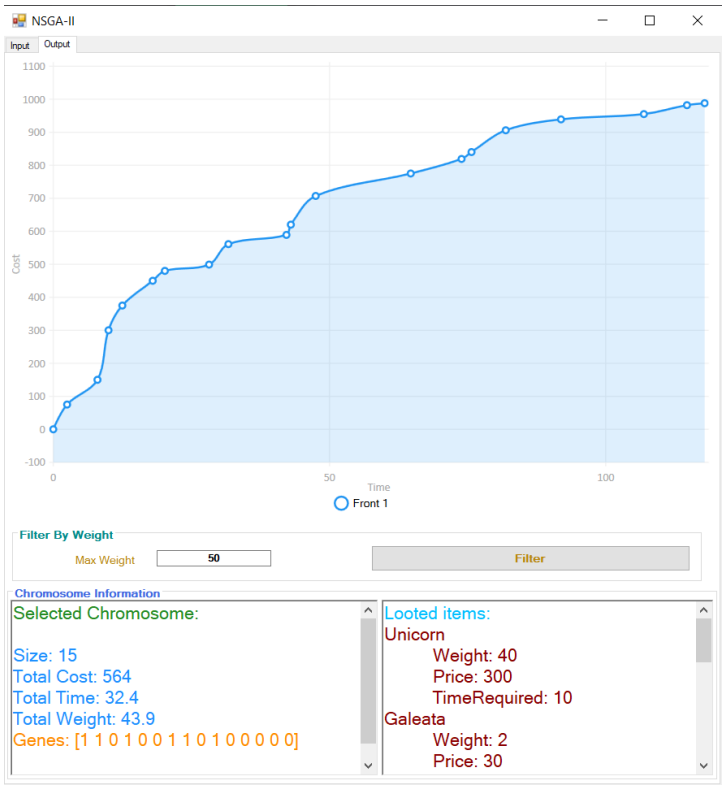
3. **Rulare algoritm cu date inițiale implicite, 100 de generații, fără filtrare după greutate**



4. **Rulare algoritm cu date inițiale implicite, dimensiunea populației de 20 indivizi, 100 de generații, filtrare după greutate <= 50**



5. **Rulare algoritm cu date inițiale implicite, dimensiunea populației de 20 indivizi, 100 de generații, fără filtrare după greutate**



Capitolul 7. Concluzii

Algoritmul NSGA-II reprezintă o metodă eficientă de optimizare multi-obiectiv care dă rezultate foarte bune în combinație cu algoritmii genetici.

Observația 1: Se observă faptul că creșterea numărului de generații, implică scăderea numărului de fronturi.

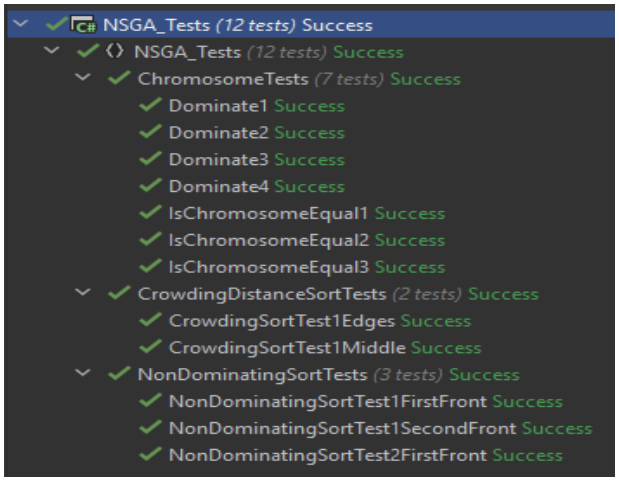
Observația 2: Se observă faptul că micșorarea numărului indivizilor, implică scăderea numărului de fronturi.

Aceste rezultate sunt apropiate de cele reale, iar posibilitatea reprezentării grafice a soluțiilor ajută la ușurarea luării unei decizii în funcție de parametrii de interes.

Capitolul 8. Testare

Au fost testate:

- funcțiile de Dominates și IsEqual din clasa Chromosome
- sortarea „non dominated sort”
- sortarea „crowding distance sort”



Capitolul 9. Bibliografie

- https://www.sciencedirect.com/science/article/pii/S1877705811022466
- http://florinleon.byethost24.com/lab_ia.html
- https://arxiv.org/pdf/1901.00577.pdf
- https://ocw.mit.edu/courses/institute-for-data-systems-and-society/ids-338j-multidisciplinary-system-design-optimization-spring-2010/lecture-notes/MITESD_77S10_lec14.pdf
- https://web.stanford.edu/group/sisl/k12/optimization/MO-unit5-pdfs/5.8Pareto.pdf

Capitolul 10. Roluri

Student	Rol
Enachi Vasile	Elaborarea modului de algoritm si CLI
Muraru Alexandru	Elaborarea GUI și integrarea cu algoritmul, modulul de test
Ilioi Alexandru	Documentație și proiectarea modulelor