Panaite Mircea - 332

Inteligență artificială

Tema – 3

Descrierea algoritmilor

Algoritm Evolutiv pentru Knapsack

```java
public class EvolutiveAlgorithmForKnapsack {
    private final List<Integer> values;
    private final List<Integer> weights;
    private final int capacity;
    private final int populationSize;
    private final int maxGenerations;
    private final double crossoverRate;
    private final double mutationRate;
    private final Random rand = new Random();

    public EvolutiveAlgorithmForKnapsack(List<Integer> values, List<Integer> weights, int capacity, int populationSize,
                                         int maxGenerations, double crossoverRate, double mutationRate) {
        this.values = values;
        this.weights = weights;
        this.capacity = capacity;
        this.populationSize = populationSize;
        this.maxGenerations = maxGenerations;
        this.crossoverRate = crossoverRate;
        this.mutationRate = mutationRate;
    }

    // Generate an initial population of valid solutions
    private List<List<Integer>> generateInitialPopulation() {
        List<List<Integer>> population = new ArrayList<>();
        for (int i = 0; i < populationSize; i++) {
            List<Integer> solution = generateRandomValidSolution();
            population.add(solution);
        }
        return population;
    }
 private List<List<Integer>> selectParents(List<List<Integer>> population) {
     List<List<Integer>> parents = new ArrayList<>();
     for (int i = 0; i < 2; i++) {
         int tournamentSize = 5;
         List<Integer> bestSolution = new ArrayList<>();
         int bestFitness = 0;
         for (int j = 0; j < tournamentSize; j++) {
             int index = rand.nextInt(population.size());
             List<Integer> solution = population.get(index);
             int fitness = evaluateFitness(solution);
             if (fitness > bestFitness) {
                 bestSolution = solution;
                 bestFitness = fitness;
             }
         }
         parents.add(bestSolution);
     }
     return parents;
 }

 // Perform single-point orderCrossover on the parents
 private List<List<Integer>> crossoverWithOneCutPoint(List<List<Integer>> parents) {
     List<List<Integer>> children = new ArrayList<>();
     children.add(new ArrayList<>());
     children.add(new ArrayList<>());
     if (rand.nextDouble() < crossoverRate) {
         int crossoverPoint = rand.nextInt( bound: values.size() - 1) + 1;
         for (int i = 0; i < crossoverPoint; i++) {
             children.get(0).add(parents.get(0).get(i));
```

```java
private List<List<Integer>> crossoverWithOneCutPoint(List<List<Integer>> parents) {
    List<List<Integer>> children = new ArrayList<>();
    children.add(new ArrayList<>());
    children.add(new ArrayList<>());
    if (rand.nextDouble() < crossoverRate) {
        int crossoverPoint = rand.nextInt( bound: values.size() - 1) + 1;
        for (int i = 0; i < crossoverPoint; i++) {
            children.get(0).add(parents.get(0).get(i));
            children.get(1).add(parents.get(1).get(i));
        }
        for (int i = crossoverPoint; i < values.size(); i++) {
            children.get(0).add(parents.get(1).get(i));
            children.get(1).add(parents.get(0).get(i));
        }
    } else {
        children.set(0, parents.get(0));
        children.set(1, parents.get(1));
    }
    return children;
}

// Perform random mutation on a child
private void mutate(List<Integer> child) {
    for (int i = 0; i < child.size(); i++) {
        if (rand.nextDouble() < mutationRate) {
            if (child.get(i) == 1) {
                child.set(i, 0);
            } else {
                child.set(i, 1);
```

```java
private List<List<Integer>> selectSurvivors(List<List<Integer>> population, List<List<Integer>> children) {
    List<List<Integer>> nextGeneration = new ArrayList<>(population);
    int worstIndex = 0;
    int worstFitness = Integer.MAX_VALUE;
    for (int i = 0; i < populationSize; i++) {
        int fitness = evaluateFitness(population.get(i));
        if (fitness < worstFitness) {
            worstIndex = i;
            worstFitness = fitness;
        }
    }
    for (int i = 0; i < 2; i++) {
        int index = rand.nextInt(populationSize);
        nextGeneration.set(index, children.get(i));
    }
    nextGeneration.set(worstIndex, children.get(0));
    return nextGeneration;
}

// Run the evolutionary algorithm
public void evolutiveAlgorithm(int option) {
    long startTime = System.nanoTime();
    List<List<Integer>> population = generateInitialPopulation();
    int bestFitness = 0;
    List<Integer> bestSolutions = new ArrayList<>();
    List<Double> averageFitness = new ArrayList<>();
    for (int i = 0; i < maxGenerations; i++) {
        List<List<Integer>> children = new ArrayList<>();
        for (int j = 0; j < 2; j++) {
            List<List<Integer>> parents = selectParents(population);
```

```
for (int i = 0; i < maxGenerations; i++) {
    List<List<Integer>> children = new ArrayList<>();
    for (int j = 0; j < 2; j++) {
        List<List<Integer>> parents = selectParents(population);
        List<List<Integer>> offspring = new ArrayList<>();
        if (option == 1) offspring = crossoverWithTwoCutPoints(parents);
        else if (option == 2) offspring = crossoverWithOneCutPoint(parents);
        for (List<Integer> child : offspring) {
            mutate(child);
            children.add(child);
        }
    }
    population = selectSurvivors(population, children);
    int totalFitness = 0;
    for (List<Integer> solution : population) {
        int fitness = evaluateFitness(solution);
        if (fitness > bestFitness && isValid(solution)) {
            bestFitness = fitness;
        }
        totalFitness += fitness;
    }
    double avgFitness = (double) totalFitness / population.size();
    bestSolutions.add(bestFitness);
    averageFitness.add(avgFitness);
}
System.out.println("Generation\tBest\tAverage");
for (int i = 0; i < maxGenerations; i++) {
    System.out.println((i + 1) + "\t\t" + bestSolutions.get(i) + "\t" + averageFitness.get(i));
}
long endTime = System.nanoTime();
double elapsedTimeInSeconds = (double) (endTime - startTime) / 1_000_000_000.0;
```

```
    System.out.println("Time is: " + elapsedTimeInSeconds + " seconds");
}

public boolean isValid(List<Integer> solution) {
    int weight = 0;
    for (int i = 0; i < weights.size(); i++) {
        weight += solution.get(i) * weights.get(i);
    }
    return (weight <= capacity);
}

public int evaluateFitness(List<Integer> solution) {
    return IntStream.range(0, solution.size())
            .map(i -> solution.get(i) * values.get(i))
            .sum();
}

private List<Integer> generateRandomValidSolution() {
    Random random = new Random();
    List<Integer> solution = new ArrayList<>();
    boolean isValidFlag = false;
    while (!isValidFlag) {
        solution.clear();
        for (int i = 0; i < values.size(); i++) {
            solution.add(random.nextInt( bound: 2));
        }
        isValidFlag = isValid(solution);
    }
    return solution;
}
```

```java
private List<List<Integer>> crossoverWithTwoCutPoints(List<List<Integer>> parents) {
    List<List<Integer>> children = new ArrayList<>();
    children.add(new ArrayList<>());
    children.add(new ArrayList<>());
    if (rand.nextDouble() < crossoverRate) {
        int cutPoint1 = rand.nextInt( bound: values.size() - 2) + 1;
        int cutPoint2 = rand.nextInt( bound: values.size() - cutPoint1) + cutPoint1;
        for (int i = 0; i < cutPoint1; i++) {
            children.get(0).add(parents.get(0).get(i));
            children.get(1).add(parents.get(1).get(i));
        }
        for (int i = cutPoint1; i < cutPoint2; i++) {
            children.get(0).add(parents.get(1).get(i));
            children.get(1).add(parents.get(0).get(i));
        }
        for (int i = cutPoint2; i < values.size(); i++) {
            children.get(0).add(parents.get(0).get(i));
            children.get(1).add(parents.get(1).get(i));
        }
    } else {
        children.set(0, parents.get(0));
        children.set(1, parents.get(1));
    }
    return children;
}
```

- generateInitialPopulation()
  Această metodă este utilizată pentru a genera o populație inițială de soluții valide. În această metodă se generează un număr de soluții aleatoare (dimensiunea populației) și se verifică dacă sunt valide, adică dacă greutatea totală a obiectelor din soluție este mai mică sau egală cu capacitatea rucsacului. Dacă nu este validă, se generează o altă soluție până când se obține o soluție validă.

- evaluateFitness()
  Această metodă este utilizată pentru a evalua valoarea de fitness a unei soluții. Valoarea de fitness este calculată ca suma valorilor obiectelor incluse în soluție.

- selectParents()
  Această metodă este utilizată pentru a selecta doi părinți pentru a fi utilizați în crossover. Se utilizează o selecție tip turneu, adică se aleg aleatoriu un număr de soluții (dimensiunea turneului) din populație și se selectează cea mai bună soluție din acest grup. Acest proces se repetă până când se obțin cei doi părinți.

- crossoverWithOneCutPoint()
  Această metodă este utilizată pentru a realiza crossover între doi părinți utilizând un punct de tăiere unic. Punctul de tăiere este ales aleatoriu, iar obiectele dinaintea punctului de tăiere sunt copiate din primul părinte, iar cele după punctul de tăiere sunt copiate din al doilea părinte. În final se obțin doi copii.

- mutate(List<Integer> child)
  Această funcție efectuează mutații aleatoare pe un copil dat, reprezentat sub forma unei liste de numere întregi. Fiecare element al listei (reprezentând selectarea sau nu a unui obiect) este parcurs și, cu o probabilitate dată de mutationRate, se inversează valoarea elementului (de exemplu, dacă elementul este 1, va fi setat la 0 și invers).

- Funcția generateInitialPopulation() generează o populație inițială de soluții aleatoare valide. Pentru fiecare soluție generată, se verifică dacă soluția este validă (adică dacă nu depășește capacitatea rucsacului), iar dacă soluția nu este validă, se generează o nouă soluție.

- Funcția selectParents() selectează doi părinți din populația actuală, folosind selecția turneu. Se aleg aleatoriu tournamentSize soluții din populația actuală și se alege cea mai bună soluție dintre acestea ca părinte. Această selecție se face de două ori, pentru a obține doi părinți.
- Funcția crossoverWithOneCutPoint() execută crossover-ul dintre cei doi părinți folosind un singur punct de tăiere. Se alege un punct aleatoriu în soluție și se interschimbă subsecțiile de soluție dintre cei doi părinți la stânga și la dreapta punctului de tăiere.
- Funcția mutate() execută mutația aleatoare asupra unei soluții copil. Se parcurge fiecare genă din soluția copil și, dacă o probabilitate aleatoare este mai mică decât rata de mutație, se schimbă valoarea genei. Dacă valoarea genei este 1, se schimbă în 0 și invers.
- Funcția selectSurvivors() selectează următoarea generație folosind selecția elitistă. Se alege cea mai proastă soluție din populația actuală și se înlocuiește cu cea mai bună soluție din copiii generați în această iterație. Apoi se alege alte două soluții copii aleatoare pentru a completa următoarea generație.
- Funcția evolutiveAlgorithm() rulează algoritmul evolutiv pentru a găsi cea mai bună soluție. Se generează o populație inițială și se parcurg iterații pentru a obține o nouă populație. În fiecare iterație, se selectează doi părinți, se execută crossover-ul și mutația asupra copiilor și se selectează următoarea generație. În timpul iterațiilor, se calculează fitness-ul soluțiilor și se păstrează cea mai bună soluție găsită până în acel moment. La final, se afișează fitness-ul celei mai bune soluții găsite și media fitness-ului populației, pentru fiecare iterație.

Algoritm Evolutiv pentru TSP

```java
public class EvolutiveAlgorithmForTSP {
    private static final String SPACE = " ";
    private final int populationSize;
    private final int maxGenerations;
    private int numberOfCities;
    private double[][] distances;
    private final List<int[]> population;

    public EvolutiveAlgorithmForTSP(int populationSize, int maxGenerations) throws IOException {
        this.populationSize = populationSize;
        this.maxGenerations = maxGenerations;
        population = new ArrayList<>();
        readDataFromFile();
    }

    private void readDataFromFile() throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new FileReader( fileName: "files/lin105.tsp"));
        ArrayList<double[]> citiesList = new ArrayList<>();
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            String[] tokens = line.split(SPACE);
            double[] city = new double[2];
            city[0] = Double.parseDouble(tokens[1]);  // X-coordinate
            city[1] = Double.parseDouble(tokens[2]);  // Y-coordinate
            citiesList.add(city);
        }
        bufferedReader.close();
        numberOfCities = citiesList.size();
        distances = new double[numberOfCities][numberOfCities];
        for (int i = 0; i < numberOfCities; i++) {
```

```
    numberOfCities = citiesList.size();
    distances = new double[numberOfCities][numberOfCities];
    for (int i = 0; i < numberOfCities; i++) {
        for (int j = i + 1; j < numberOfCities; j++) {
            // Euclidean distance
            double distance = Math.sqrt(Math.pow(citiesList.get(i)[0] - citiesList.get(j)[0], 2) +
                    Math.pow(citiesList.get(i)[1] - citiesList.get(j)[1], 2));
            distances[i][j] = distance;
            distances[j][i] = distance;
        }
    }
}

// Generate an initial population randomly
public void generateInitialPopulation() {
    for (int i = 0; i < populationSize; i++) {
        int[] individual = new int[numberOfCities];
        for (int j = 0; j < numberOfCities; j++) {
            individual[j] = j;
        }
        Collections.shuffle(List.of(individual));
        population.add(individual);
    }
}

// Select parents using tournament selection
public int[] selectParents() {
    int parent1Index = (int) (Math.random() * populationSize);
    int parent2Index = (int) (Math.random() * populationSize);
    while (parent2Index == parent1Index) {
```

```
public int[] selectParents() {
    int parent1Index = (int) (Math.random() * populationSize);
    int parent2Index = (int) (Math.random() * populationSize);
    while (parent2Index == parent1Index) {
        parent2Index = (int) (Math.random() * populationSize);
    }
    int[] parent1 = population.get(parent1Index);
    int[] parent2 = population.get(parent2Index);
    if (calculateFitness(parent1) < calculateFitness(parent2)) {
        return parent1;
    } else {
        return parent2;
    }
}

// Mutate an individual by swapping two cities
public void mutate(int[] individual) {
    int cityIndex1 = (int) (Math.random() * numberOfCities);
    int cityIndex2 = (int) (Math.random() * numberOfCities);
    while (cityIndex2 == cityIndex1) {
        cityIndex2 = (int) (Math.random() * numberOfCities);
    }
    int temp = individual[cityIndex1];
    individual[cityIndex1] = individual[cityIndex2];
    individual[cityIndex2] = temp;
}
```

```java
public int[] orderCrossover(int[] parent1, int[] parent2) {
    // different numbers
    int crossoverPoint1 = (int) (Math.random() * numberOfCities);
    int crossoverPoint2 = (int) (Math.random() * numberOfCities);
    while (crossoverPoint2 == crossoverPoint1) {
        crossoverPoint2 = (int) (Math.random() * numberOfCities);
    }
    if (crossoverPoint1 > crossoverPoint2) {
        int temp = crossoverPoint1;
        crossoverPoint1 = crossoverPoint2;
        crossoverPoint2 = temp;
    }
    int[] child = new int[numberOfCities];
    Arrays.fill(child, val: -1);
    // fill the child from crossoverPoint1 to crossoverPoint2 with the parent1 values
    if (crossoverPoint2 + 1 - crossoverPoint1 >= 0)
        System.arraycopy(parent1, crossoverPoint1, child, crossoverPoint1, length: crossoverPoint2 + 1 - crossoverPoint1);
    int j = 0;
    for (int i = 0; i < numberOfCities; i++) {
        if (j == crossoverPoint1) {
            j = crossoverPoint2 + 1;
        }
        if (!containsCity(child, parent2[i])) {
            child[j] = parent2[i];
            j++;
        }
    }
    return child;
}
```

```java
public int[] partiallyMappedCrossover(int[] parent1, int[] parent2) {
    int length = parent1.length;
    int[] child = new int[length];
    Arrays.fill(child, val: -1);

    int startPoint = (int) (Math.random() * length);
    int endPoint = (int) (Math.random() * (length - startPoint)) + startPoint;

    // copy the selected segment from parent1 to the child
    for (int i = startPoint; i <= endPoint; i++) {
        child[i] = parent1[i];
    }

    // fill the remaining positions of the child from parent2
    for (int i = 0; i < length; i++) {
        if (i < startPoint || i > endPoint) {
            int gene = parent2[i];

            // check if the gene is already present in the child
            while (containsCity(child, gene)) {
                gene = parent2[getIndex(parent1, gene)];
            }
            child[i] = gene;
        }
    }

    return child;
}
```

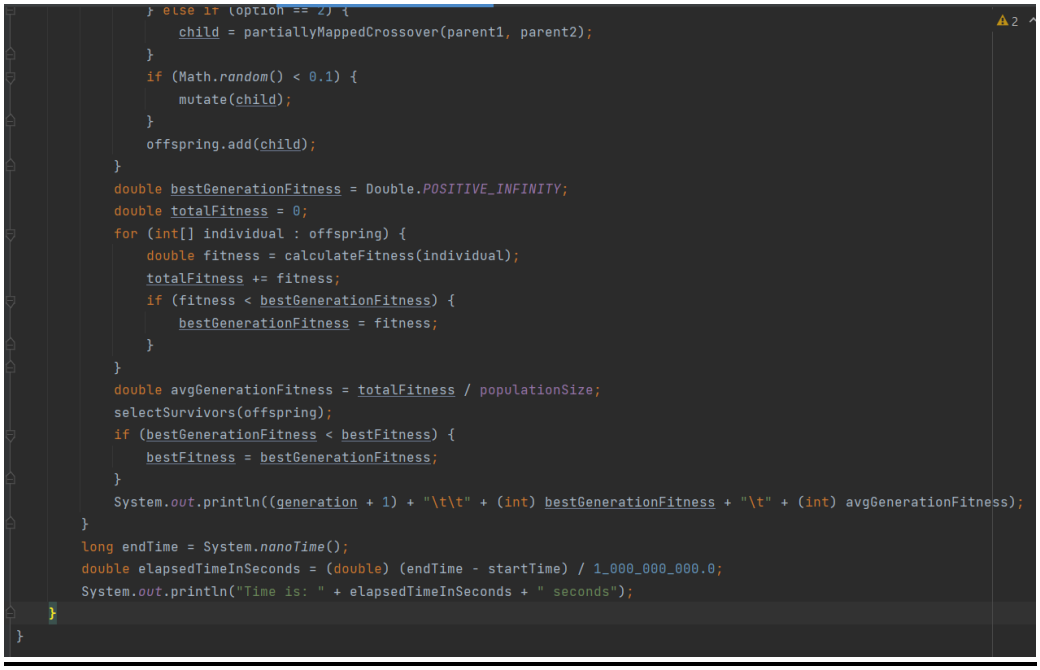
```java
// helper method to get the index of a given city in an array
private int getIndex(int[] parent, int city) {
    for (int i = 0; i < parent.length; i++) {
        if (parent[i] == city) {
            return i;
        }
    }
    return -1;
}

// Select survivors using elitism
public void selectSurvivors(List<int[]> offspring) {
    population.sort(Comparator.comparingDouble(this::calculateFitness));
    for (int i = 0; i < offspring.size(); i++) {
        population.set(populationSize - 1 - i, offspring.get(i));
    }
}

// Calculate the fitness of an individual
public double calculateFitness(int[] tour) {
    double distance = 0;
    for (int i = 0; i < numberOfCities - 1; i++) {
        distance += distances[tour[i]][tour[i + 1]];
    }
    distance += distances[tour[numberOfCities - 1]][tour[0]];  // Return to the first ci
    return distance;
}
```

```java
// Check if an individual contains a city
public boolean containsCity(int[] individual, int cityIndex) {
    for (int i = 0; i < numberOfCities; i++) {
        if (individual[i] == cityIndex) {
            return true;
        }
    }
    return false;
}

public void evolutiveAlgorithm(int option) {
    long startTime = System.nanoTime();
    System.out.println("Generation\tBest\tAverage");
    generateInitialPopulation();
    double bestFitness = Double.POSITIVE_INFINITY;
    for (int generation = 0; generation < maxGenerations; generation++) {
        List<int[]> offspring = new ArrayList<>();
        for (int i = 0; i < populationSize; i++) {
            int[] parent1 = selectParents();
            int[] parent2 = selectParents();
            int[] child = new int[0];
            if (option == 1) {
                child = orderCrossover(parent1, parent2);
            } else if (option == 2) {
                child = partiallyMappedCrossover(parent1, parent2);
            }
            if (Math.random() < 0.1) {
                mutate(child);
            }
            offspring.add(child);
        }
```



```
            } else if (option == 2) {
                child = partiallyMappedCrossover(parent1, parent2);
            }
            if (Math.random() < 0.1) {
                mutate(child);
            }
            offspring.add(child);
        }
        double bestGenerationFitness = Double.POSITIVE_INFINITY;
        double totalFitness = 0;
        for (int[] individual : offspring) {
            double fitness = calculateFitness(individual);
            totalFitness += fitness;
            if (fitness < bestGenerationFitness) {
                bestGenerationFitness = fitness;
            }
        }
        double avgGenerationFitness = totalFitness / populationSize;
        selectSurvivors(offspring);
        if (bestGenerationFitness < bestFitness) {
            bestFitness = bestGenerationFitness;
        }
        System.out.println((generation + 1) + "\t\t" + (int) bestGenerationFitness + "\t" + (int) avgGenerationFitness);
    }
    long endTime = System.nanoTime();
    double elapsedTimeInSeconds = (double) (endTime - startTime) / 1_000_000_000.0;
    System.out.println("Time is: " + elapsedTimeInSeconds + " seconds");
}
}
```

- generateInitialPopulation(): Această metodă generează o populație inițială de indivizi aleatoriu, cu o dimensiune dată. Fiecare individ este reprezentat sub formă de un șir de numere întregi care conține indicele orașelor într-o ordine aleatorie. Aceasta utilizează metoda shuffle() din clasa Collections pentru a amesteca indicii.

- selectParents(): Această metodă selectează doi părinți din populație folosind selecția prin turneu. Se aleg aleatoriu doi indivizi și li se compară fitness-ul (calculat folosind metoda calculateFitness()) pentru a determina câștigătorul. Câștigătorul este returnat ca părinte selectat.

- mutate(int[] individual): Această metodă mută un individ prin schimbarea aleatorie a pozițiilor a două orașe din reprezentarea sa (adică șirul de indici).

- orderCrossover(int[] parent1, int[] parent2): Această metodă efectuează încrucișarea ordonată între doi părinți pentru a genera un copil. Se selectează aleatoriu două puncte de încrucișare și se completează segmentul dintre ele cu valorile primului părinte. Se completează apoi pozițiile rămase ale copilului cu valorile celui de-al doilea părinte în ordinea în care apar, trecând peste orice valori care au fost deja adăugate la copil.

- partiallyMappedCrossover(int[] parent1, int[] parent2): Această metodă efectuează încrucișarea parțial mapată între doi părinți pentru a genera un copil. Se selectează aleatoriu un segment al primului părinte și se copiază la pozițiile corespunzătoare în copil. Se completează apoi pozițiile rămase ale copilului prin maparea valorilor din cel de-al doilea părinte la pozițiile corespunzătoare din segmentul copiat de la primul părinte. Maparea este realizată folosind o tabelă de căutare generată aleatoriu pentru fiecare operație de încrucișare.

- calculateFitness(int[] individual): Această metodă calculează fitness-ul unui individ, care este distanța totală a turului reprezentat de permutarea sa de indici de orașe. Folosește șirul distances pentru a căuta distanțele dintre fiecare pereche de orașe adiacente din tur.

- evolutiveAlgorithm(int option): Această metodă rulează algoritmul evolutiv pentru maxGenerations generații. Începe prin generarea unei populații inițiale folosind generateInitialPopulation(). Apoi selectează repetat părinți, efectuează încrucișări și mutații și evaluează fitness-ul copiilor rezultați pentru a crea o nouă populație pentru următoarea generație. Cel mai bun individ din fiecare generație este afișat în consolă. În cele din urmă

Tabele de analiză a soluțiilor

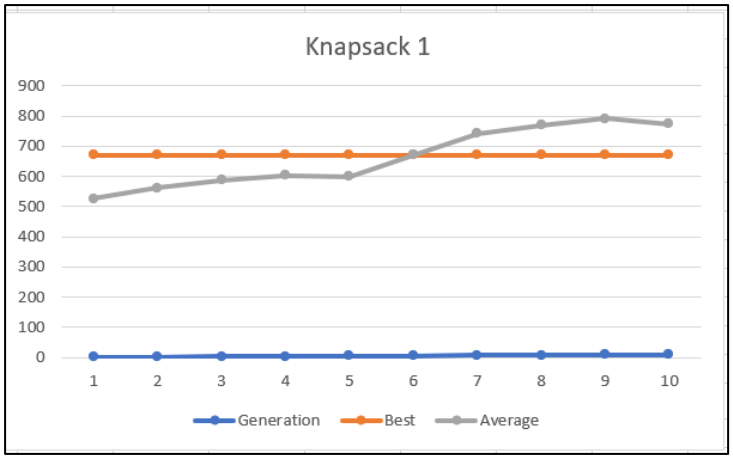
i) Instanța 1 – problema rucsacului pentru fișierul ``rucsac_20.txt``

Knapsack problem – EA - Varianta 1 - 1 punct de tăietură

Parametri: populationSize = 10, maxGenerations = 10

Nr rulare	Calitate (fitness value)
1	585
2	603
3	609
4	631
5	671
6	618
7	562
8	564
9	641
10	574

Analiza calităţii	
Best value	671
Average value	603
Worst value	562



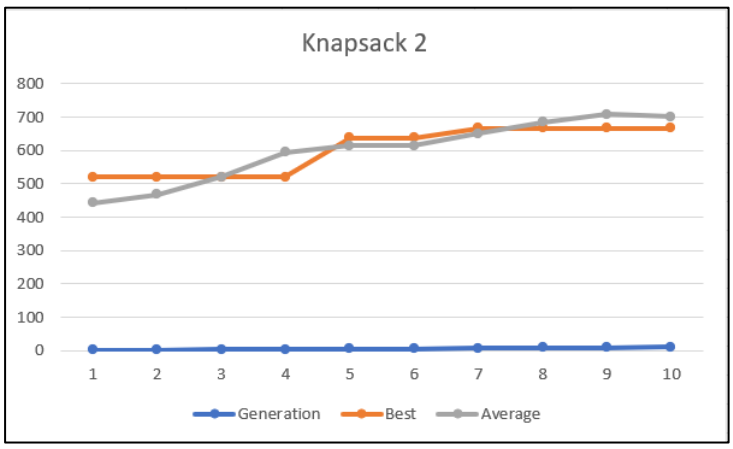
Time: 0.0013917 seconds

Knapsack problem – EA - Varianta 2 - 2 puncte de tăietură

Parametri: populationSize = 10, maxGenerations = 10

Nr rulare	Calitate (fitness value)
1	562
2	612
3	585
4	666
5	661
6	612
7	609
8	583
9	601
10	611

Analiza calităţii	
Best value	666
Average value	606
Worst value	562



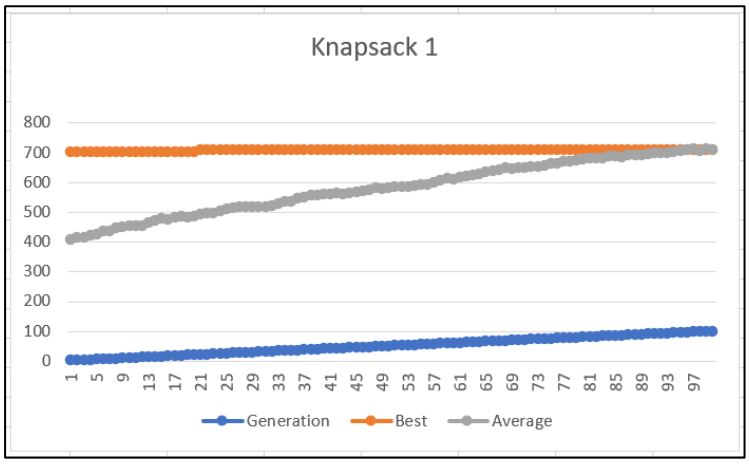
Time: 0.0012318 seconds

Knapsack problem – EA - Varianta 1 - 1 punct de tăietură

Parametri: populationSize = 100, maxGenerations = 100

Nr rulare	Calitate (fitness value)
1	624
2	675
3	710
4	628
5	599
6	652
7	679
8	690
9	640
10	636

Analiza calităţii	
Best value	710
Average value	653
Worst value	599



Time: 0.0025607 seconds

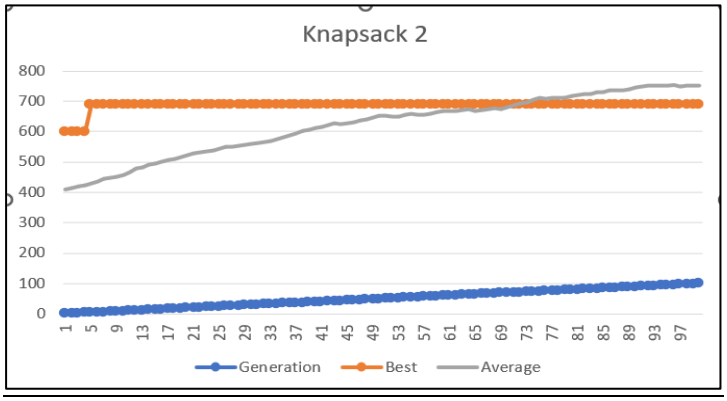
Knapsack problem – EA - Varianta 2 - 2 punct de tăietură

Parametri: populationSize = 100, maxGenerations = 100

Nr rulare	Calitate (fitness value)
1	685
2	669
3	667
4	637
5	658
6	633
7	689

8	662
9	630
10	662

Analiza calităţii	
Best value	689
Average value	660
Worst value	630



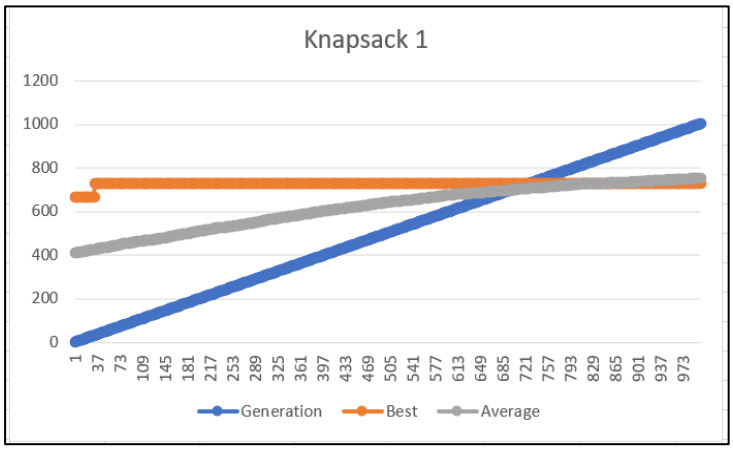
Time: 0.9124567 seconds

Knapsack problem – EA - Varianta 1 - 1 puncte de tăietură

Parametri: populationSize = 1000, maxGenerations = 1000

Nr rulare	**Calitate (fitness value)**
1	726
2	675
3	681
4	698
5	718
6	684
7	686
8	677
9	677
10	689

Analiza calităţii	
Best value	726
Average value	691
Worst value	675



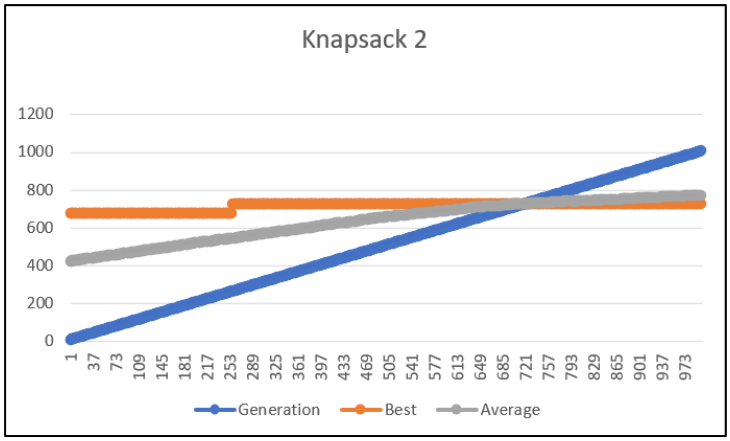
Time: 0.9039123 seconds

Knapsack problem – EA - Varianta 2 - 2 punct de tăietură

Parametri: populationSize = 1000, maxGenerations = 1000

Nr rulare	Calitate (fitness value)
1	702
2	683
3	664
4	688
5	679
6	718
7	676
8	698
9	690
10	677

Analiza calității	
Best value	718
Average value	689
Worst value	664



Time: 0.9012167 seconds

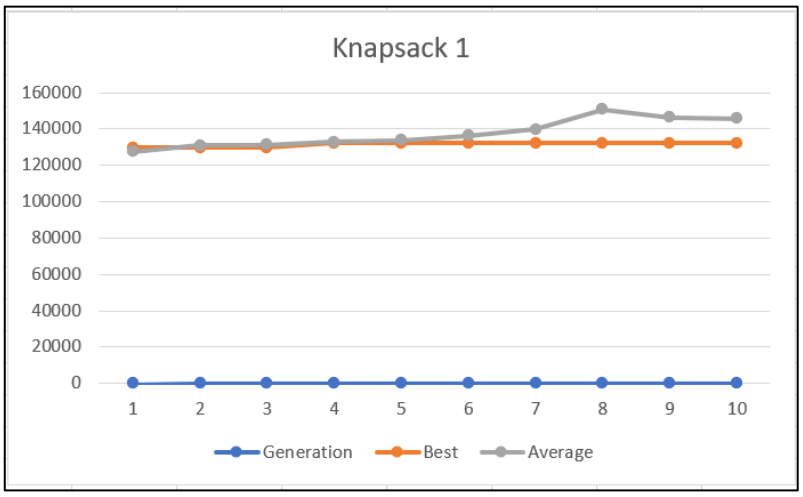
ii) Instanța 2 – problema rucsacului pentru fișierul ``rucsac_200.txt``

Knapsack problem – EA - Varianta 1 - 1 puncte de tăietură

Parametri: populationSize = 10, maxGenerations = 10

Nr rulare	Calitate (fitness value)
1	131300
2	131959
3	131564
4	130656
5	131624
6	131319
7	130865
8	130163
9	130968
10	132282

Analiza calităţii	
Best value	132282
Average value	131013
Worst value	130163



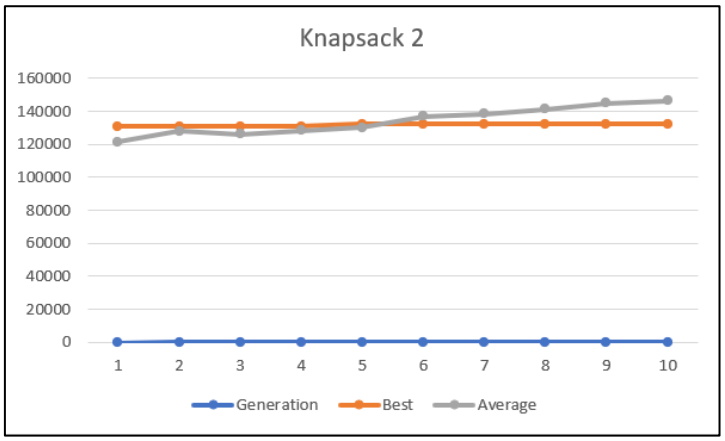
Time: 0.0022347 seconds

Knapsack problem – EA - Varianta 2 - 2 puncte de tăietură

Parametri: populationSize = 10, maxGenerations = 10

Nr rulare	Calitate (fitness value)
1	131087
2	131509
3	132248
4	130603
5	130955
6	130637
7	131092
8	132220
9	132279
10	131102

Analiza calităţii	
Best value	132279
Average value	131553
Worst value	130603



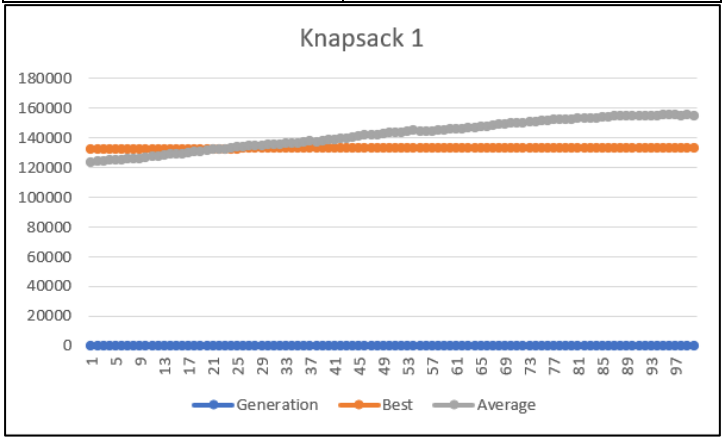
Time: 0.0021009 seconds

Knapsack problem – EA - Varianta 1 - 1 puncte de tăietură

Parametri: populationSize = 100, maxGenerations = 100

Nr rulare	Calitate (fitness value)
1	132746
2	132839
3	133436
4	133027
5	132344
6	132982
7	132673
8	132885
9	132517
10	133027

Analiza calității	
Best value	132746
Average value	132142
Worst value	131486



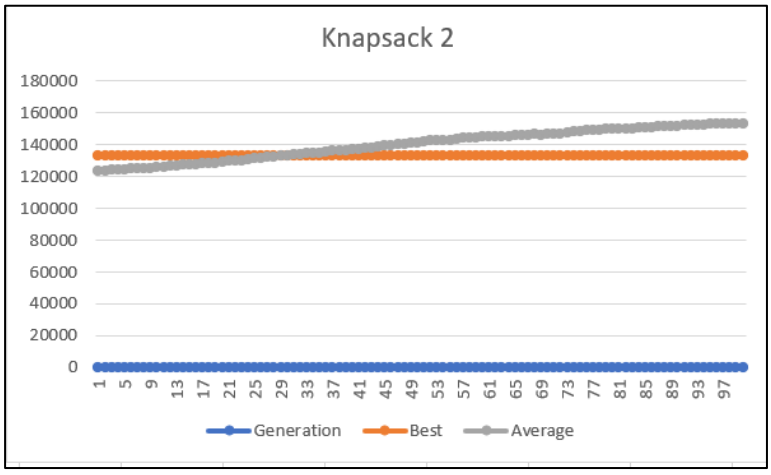
Time: 0.0051898 seconds

Knapsack problem – EA - Varianta 2 - 2 puncte de tăietură

Parametri: populationSize = 100, maxGenerations = 100

Nr rulare	Calitate (fitness value)
1	132898
2	132048
3	131708
4	132254
5	131915
6	132052
7	132032
8	132748
9	131893
10	132251

Analiza calităţii	
Best value	132898
Average value	132107
Worst value	131708



Time: 0.0047235 seconds

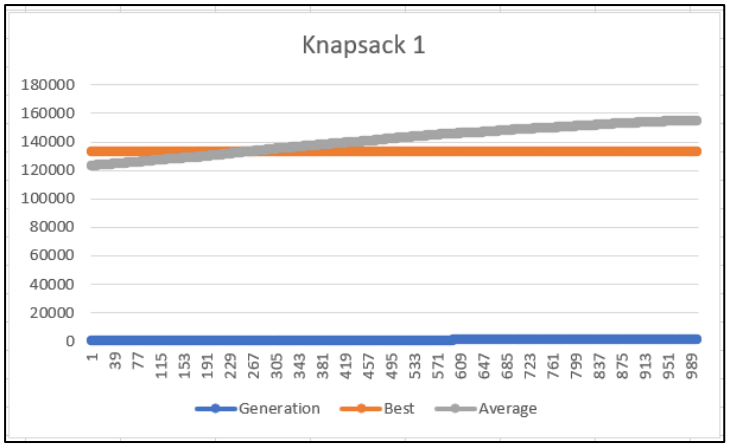
Knapsack problem – EA - Varianta 1 - 1 punct de tăietură

Parametri: populationSize = 1000, maxGenerations = 1000

Nr rulare	Calitate (fitness value)
1	132636
2	133084
3	133006
4	132822
5	132399

6	132425
7	133013
8	133134
9	132638
10	133013

Analiza calității	
Best value	133134
Average value	132816
Worst value	132399



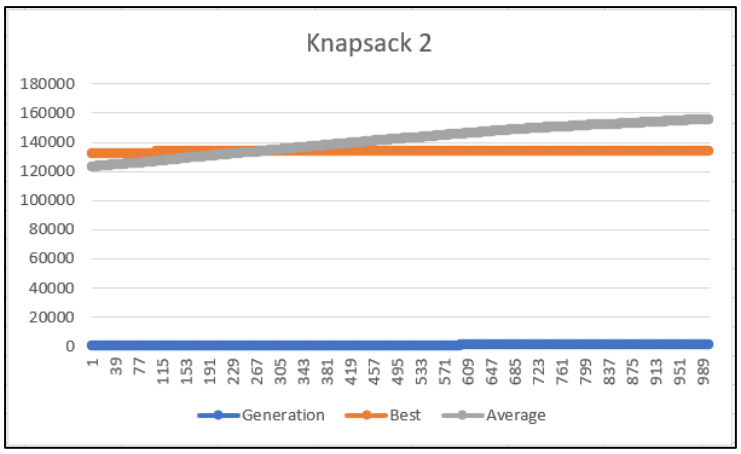
Time is: 1.874713 seconds

Knapsack problem – EA - Varianta 2 - 2 puncte de tăietură

Parametri: populationSize = 1000, maxGenerations = 1000

Nr rulare	**Calitate (fitness value)**
1	132839
2	132772
3	133436
4	133027
5	132793
6	132982
7	132885
8	133323
9	132391
10	133135

Analiza calității	
Best value	133436
Average value	133178
Worst value	132391



Time: 1.72214 seconds

Instanța – problema comis voiajor pentru fișierul ``lin105.tsp``

Traveling Salesman Problem – Evolutive algorithm

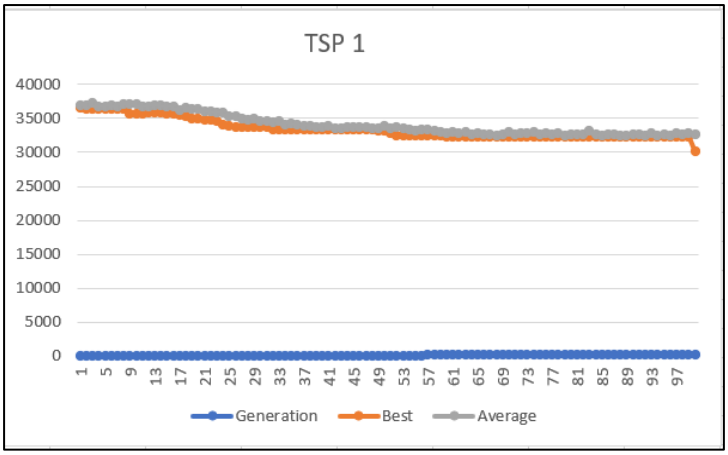
Varianta 1 - Order Crossover

Parametri: populationSize = 100 maxGenerations = 100

Nr rulare	Cost (distanță) – fitness value
1	31152
2	32567
3	30674
4	29945
5	30379
6	31067
7	31035
8	32944
9	31254
10	32166

Analiza calității	
Best value	29945
Average value	31347
Worst value	32944

Panaite Mircea - 332



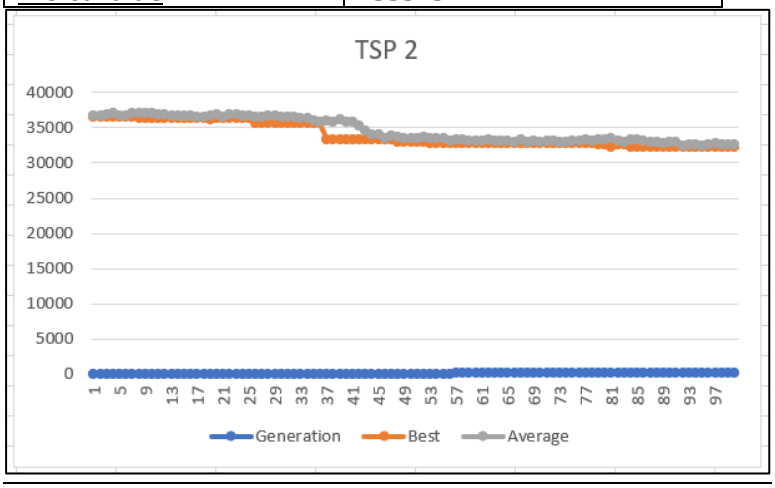
Time : 0.0828776 seconds

Varianta 2 - Partially Mapped Crossover

Parametri: populationSize = 100 maxGenerations = 100

Nr rulare	Cost (distanță) – fitness value
1	32877
2	33586
3	34662
4	33494
5	34501
6	33764
7	33270
8	32254
9	33675
10	33975

Analiza calității	
Best value	32254
Average value	34662
Worst value	33573



Time is: 0.0754848

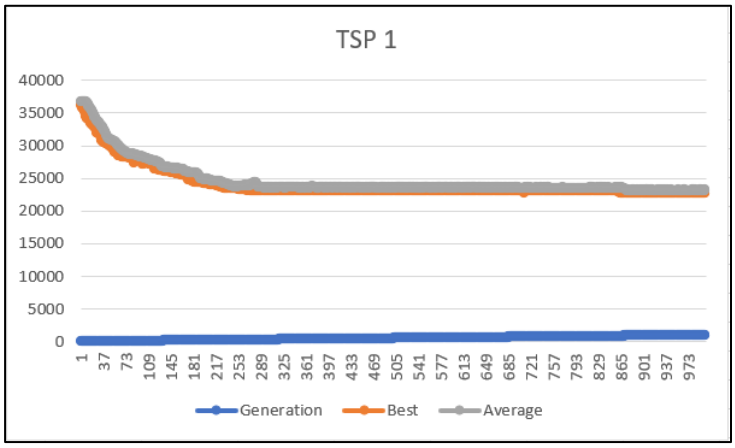
Panaite Mircea - 332

Varianta 1 - Order Crossover

Parametri: populationSize = 1000 maxGenerations = 1000

Nr rulare	Cost (distanță) – fitness value
1	23423
2	24465
3	22728
4	23747
5	23069
6	23788
7	23100
8	23163
9	23969
10	22967

Analiza calității	
Best value	22728
Average value	23335
Worst value	24465



Time: 8.6156771s

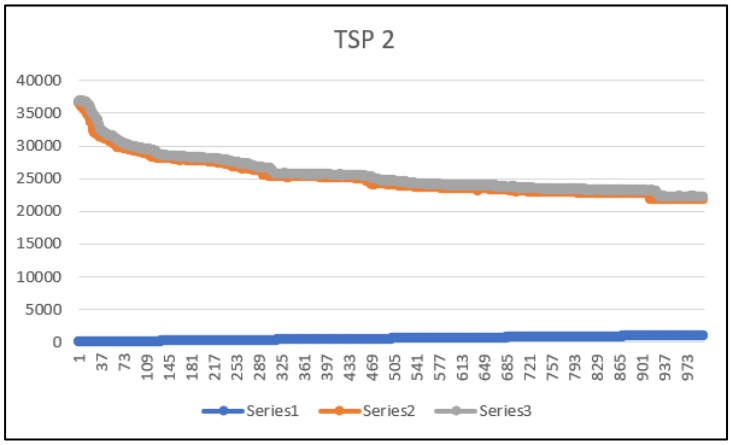
Varianta 2 - Partially Mapped Crossover

Parametri: populationSize = 1000 maxGenerations = 1000

Nr rulare	Cost (distanță) – fitness value
1	23274
2	21775
3	23106
4	22539
5	23845
6	24622
7	22244
8	23301
9	23864

10	23384

Analiza calităţii	
Best value	21775
Average value	23175
Worst value	24622



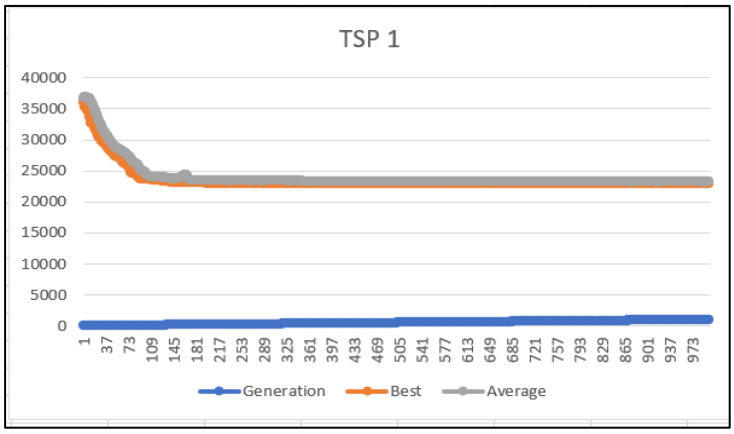
Time: 8.12111s

Varianta 1 - Order Crossover

Parametri: populationSize = 10000 maxGenerations = 1000

Nr rulare	Cost (distanță) – fitness value
1	23251
2	22960
3	23046
4	23803
5	23794
6	22831
7	22823
8	22887
9	22909
10	22819

Analiza calităţii	
Best value	22819
Average value	23022
Worst value	23803



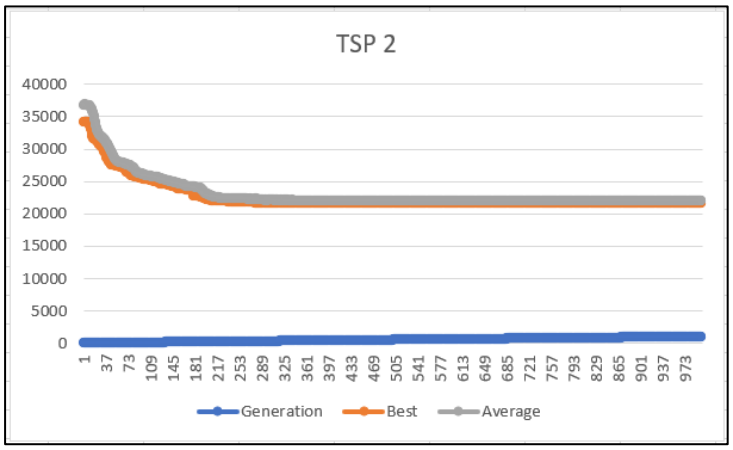
Time : 42.003 s

Varianta 2 - Partially Mapped Crossover

Parametri: populationSize = 10000 maxGenerations = 1000

Nr rulare	Cost (distanță) – fitness value
1	23064
2	22272
3	22687
4	21534
5	22022
6	21800
7	22233
8	23061
9	22639
10	22278

Analiza calității	
Best value	21534
Average value	23064
Worst value	22466



Time : 41.99133 s

Concluzii

i) Knapsack

Pe măsura ce creștem populationSize, maxGenerations obținem rezultate din ce în ce mai bune apropiate de optim și păstrăm mutationRate si crossover constante iar timpul nu crește considerabil. Am folosit două tipuri diferite de mutații – cu un punct și cu două puncte de tăietură. Pentru cel de-al doilea tip am înregistrat rezultate de best si average mai bune și timpi cu foarte puțin sesizabil mai buni. Însă nici dacă măresc parametrii maim ult nu obțin rezultate mai bune pentru best, ducând la o convergență prematură și la rezultate egale pentru prea multe generații. Rezultatele sunt mult mai bune decât pentru Tabu Search și căutările locale și aleatoare.

ii) TSP

Cu cât creștem populationSize și maxGenerations obținem rezultate din ce în ce mai bune apropiate de optim și păstrăm mutationRate si crossover constante iar timpul nu crește considerabil. Am folosit două tipuri diferite de mutații – ordered și partially mapped. Pentru cel de-al doilea tip am înregistrat rezultate de best si average mai bune și timpi cu foarte puțin sesizabil mai buni. Însă cu cât creștem numărul de generații creștem și timpul de execuție substantial. Pentru mai mult de 1000 populationSize ne apopiem de convergență prematură. Rezultatele sunt mult mai bune decât pentru simulated annealing.