Panaite Mircea - 332

# Inteligență artificială

## Tema – 2

## Descrierea algoritmilor

### Algoritmul Tabu Search pentru Knapsack problem

```java
public void tabuSearch() {
    int[] currentSolution = generateRandomValidSolution();
    int[] best = Arrays.copyOf(currentSolution, currentSolution.length);
    int[] memoryList = initMemory(numberOfObjects);
    for (int i = 0; i < maxIterations; i++) {
        int[] xSolution = getBestNeighbourNonTabu(currentSolution, memoryList);
        updateMemory(memoryList, xSolution, currentSolution, tabuSize);
        currentSolution = xSolution;
        if (computeFitnessValue(currentSolution) > computeFitnessValue(best) && isValid(currentSolution)) {
            best = Arrays.copyOf(currentSolution, currentSolution.length);
        }
    }
    System.out.println("The best value is: " + computeFitnessValue(best));
}

private int[] initMemory(int numberOfObjects) {
    return new int[numberOfObjects];
}

private void updateMemory(int[] memoryList, int[] xSolution, int[] currentSolution, int tabuSize) {
    for (int i = 0; i < memoryList.length; i++) {
        if (xSolution[i] != 0) {
            memoryList[i] = xSolution[i] - 1;
        } else {
            memoryList[i] = xSolution[i];
        }
    }
    int retrievedTerm = retrieveTabuTermChange(currentSolution, xSolution);
    memoryList[retrievedTerm] = tabuSize;
`
private List<int[]> getAllNeighborhoods(int[] solution) {
    List<int[]> neighborhood = new ArrayList<>();
    for (int i = 0; i < numberOfObjects; i++) {
        generateNeighborhoodBy1Flip(solution, neighborhood, i);
    }
    return neighborhood;
}

private void generateNeighborhoodBy1Flip(int[] solution, List<int[]> neighborhood, int i) {
    int[] neighbor = Arrays.copyOf(solution, solution.length);
    neighbor[i] = 1 - neighbor[i];
    neighborhood.add(neighbor);
}
```

Funcția generateNeighborhoodBy1Flip setează la un anumit indice i complementara valorii (1 pentru 0 / 0 pentru 1) și acutalizează vecinătatea.

Funcția initMemory(int numberOfObjects) generează o listă de 0 cu dimensiunea egală cu numărul de obiecte.

Funcția updateMemory(int[] memoryList, int[] xSolution, int[] currentSolution, int tabuSize) actualizează memoria tabu pentru prin decrementare la pozițiile unde apare tabuSize pentru xSolution si se actualizează la indicile unde se va pune obiect în rucsac.

Funcția tabuSearch() inițializează random o soluție (binară generată cu funcția random din Java) și memoria (memoryList). Parcurgem numărul maxim de iterații și alegem o soluție din vecinătatea nonTabu. Actualizăm memoria și soluția curentă. Comparăm fitness-ul curent cu best-ul actual și îl actualizăm dacă e mai mare.

## Algoritmul Simulated Annealing pentru Traveling Salesman Problem

```java
// Method to initialize a random tour
private int[] generateRandomSolution() {
    List<Integer> tourList = new ArrayList<>();
    for (int i = 0; i < numberOfCities; i++) {
        tourList.add(i);
    }
    // Randomize the solution
    Collections.shuffle(tourList, new Random());
    int[] tour = new int[numberOfCities];
    for (int i = 0; i < numberOfCities; i++) {
        tour[i] = tourList.get(i);
    }
    return tour;
}

// Method to calculate the total distance of a tour
private double calculateTourDistance(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 city
    return distance;
}
```

```java
// Method to simulate annealing to find the shortest TSP tour
public void simulatedAnnealing() {
    // Initialize tour with a random permutation of cities
    // Current best tour
    int[] tour = generateRandomSolution();
    // Current best distance
    double bestDistance = calculateTourDistance(tour);
    // Simulated annealing loop
    double temperature = INITIAL_TEMPERATURE;
    for (int iteration = 0; iteration < maxIterations && temperature > FINAL_TEMPERATURE; iteration++) {
        // Generate a random neighbor
        int[] neighbor = generateNeighborWith2Swap(tour);
        double neighborDistance = calculateTourDistance(neighbor);
        // Decide whether to accept the neighbor
        double deltaDistance = neighborDistance - bestDistance;
        // Check that either delta is lower than 0 or e^(-delta/temperature) > random number
        if (deltaDistance < 0 || Math.exp(-deltaDistance / temperature) > Math.random()) {
            tour = neighbor;
            bestDistance = neighborDistance;
        }
        // Update the temperature
        temperature *= COOLING_FACTOR;
    }
    System.out.println("The best cost is: " + (int) bestDistance);
}
```

```java
// Method to generate a random neighbor
private int[] generateNeighborWith2Swap(int[] tour) {
    int i = (int) (Math.random() * (numberOfCities - 1));
    int j = i + 1 + (int) (Math.random() * (numberOfCities - i - 1));
    int[] neighbor = Arrays.copyOf(tour, numberOfCities);
    // Perform 2-swap switch
    while (i < j) {
        int temp = neighbor[i];
        neighbor[i] = neighbor[j];
        neighbor[j] = temp;
        i++;
        j--;
    }
    return neighbor;
}
```

Funcția de generateRandomSolution() parcurge numărul total de orașe și pe baza lui generează un tur (o permutare) random al orașelor folosind funcția de shuffle din Java, returnându-l ca soluție inițială.

Funcția calculateTourDistance(int[] tour) are ca argument lista indicilor orașelor și calculează distanța iterativ, adunând distanța euclidiană dintre două orașe consecutive progresiv. Distanța de întoarcere la primul oraș e calculată, dar orașul nu se afișează în traseu.

Funcția de generateNeighborWith2Swap(int[] tour) preia ca argument un o lista de indici de orașe și generează doi indici i și j random pentru interschimbare.

Funcția simulatedAnnealing calculează traseul optim. Pornim de la o serie de constante INITIAL_TEMPERATURE = 10000, FINAL_TEMPERATURE = 0.00001, COOLING_FACTOR = 0.99. Pornim de la o soluție random și de la o temperatura inițială. Parcurg numărul de iterații și am grijă ca temperatura să nu treaă de cea finală. Inițializez un vecin random folosind 2-swap și îi calculez costul. Calculez deltaDistance ca diferența de cost dintre vecin și best-ul curent. Dacă deltaDistance e mai mic decât 0 sau $e^{(-deltaDistance/temperature)}$ > număr rațional aleator, actualizăm noua soluție. Actualizăm și temperatura prin înmulțirea cu factorul de răcire.

# Tabele de analiză a soluțiilor

**i) Instanţa 1 – problema rucsacului pentru fişierul ``rucsac_20.txt``**

## Tabu search - Knapsack problem

**Parametri: maxIterations = 10, tabuSize = 4**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 501 |
| **2** | 413 |
| **3** | 465 |
| **4** | 594 |
| **5** | 595 |
| **6** | 527 |
| **7** | 463 |
| **8** | 495 |
| **9** | 458 |
| **10** | 511 |

| Analiza calităţii | |
|---|---|
| **Best value** | 595 |
| **Average value** | 547 |
| **Worst value** | 413 |

## Knapsack – cu Random Hill Climbing

**Parametru: iterations = 10**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 654 |
| **2** | 612 |
| **3** | 652 |
| **4** | 607 |
| **5** | 622 |
| **6** | 536 |
| **7** | 618 |
| **8** | 568 |
| **9** | 626 |
| **10** | 578 |

| Analiza calităţii | |
|---|---|
| **Best value** | 654 |
| **Average value** | 607 |
| **Worst value** | 536 |

# Tabu search - Knapsack problem

**Parametri: maxIterations = 100, tabuSize = 3**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 538 |
| **2** | 587 |
| **3** | 505 |
| **4** | 598 |
| **5** | 478 |
| **6** | 433 |
| **7** | 557 |
| **8** | 474 |
| **9** | 529 |
| **10** | 646 |

| Analiza calităţii | |
|---|---|
| **Best value** | 646 |
| **Average value** | 527 |
| **Worst value** | 433 |

# Knapsack – cu Random Hill Climbing

**Parametru: iterations = 100**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 617 |
| **2** | 596 |
| **3** | 591 |
| **4** | 663 |
| **5** | 649 |
| **6** | 655 |
| **7** | 657 |
| **8** | 608 |
| **9** | 586 |
| **10** | 607 |

| Analiza calităţii | |
|---|---|
| **Best value** | 663 |
| **Average value** | 622 |
| **Worst value** | 586 |

# Tabu search - Knapsack problem

**Parametri: maxIterations = 1000, tabuSize = 2**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 528 |
| **2** | 580 |
| **3** | 669 |
| **4** | 686 |
| **5** | 500 |
| **6** | 641 |
| **7** | 632 |
| **8** | 611 |
| **9** | 493 |
| **10** | 574 |

| Analiza calităţii | |
|---|---|
| **Best value** | 686 |
| **Average value** | 596 |
| **Worst value** | 493 |

# Knapsack – cu Random Hill Climbing

**Parametru: iterations = 1000**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 661 |
| **2** | 698 |
| **3** | 652 |
| **4** | 602 |
| **5** | 681 |
| **6** | 535 |
| **7** | 618 |
| **8** | 569 |
| **9** | 654 |
| **10** | 589 |

| Analiza calităţii | |
|---|---|
| **Best value** | 652 |
| **Average value** | 610 |
| **Worst value** | 535 |

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

## Tabu search - Knapsack problem

**Parametri: maxIterations = 10, tabuSize = 4**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 129478 |
| **2** | 127670 |
| **3** | 130051 |
| **4** | 130090 |
| **5** | 130241 |
| **6** | 131319 |
| **7** | 129513 |
| **8** | 129550 |
| **9** | 130968 |
| **10** | 129517 |

| Analiza calității | |
|---|---|
| **Best value** | 131319 |
| **Average value** | 129864 |
| **Worst value** | 127670 |

## Knapsack – cu Random Hill Climbing

**Parametru: iterations = 10**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 133826 |
| **2** | 134022 |
| **3** | 133914 |
| **4** | 133838 |
| **5** | 134040 |
| **6** | 133838 |
| **7** | 133746 |
| **8** | 133994 |
| **9** | 133747 |
| **10** | 134047 |

| Analiza calității | |
|---|---|
| **Best value** | 134047 |
| **Average value** | 133902 |
| **Worst value** | 133746 |

# Tabu search - Knapsack problem

**Parametri: maxIterations = 100, tabuSize = 3**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 129394 |
| **2** | 130721 |
| **3** | 130317 |
| **4** | 131002 |
| **5** | 129434 |
| **6** | 130154 |
| **7** | 130915 |
| **8** | 132087 |
| **9** | 129727 |
| **10** | 130732 |

| Analiza calităţii | |
|---|---|
| **Best value** | 132087 |
| **Average value** | 130017 |
| **Worst value** | 129394 |

# Knapsack – cu Random Hill Climbing

**Parametru: iterations = 100**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 133053 |
| **2** | 133128 |
| **3** | 133022 |
| **4** | 133066 |
| **5** | 133411 |
| **6** | 133489 |
| **7** | 133206 |
| **8** | 132748 |
| **9** | 133538 |
| **10** | 133337 |

| Analiza calităţii | |
|---|---|
| **Best value** | 133538 |
| **Average value** | 133235 |
| **Worst value** | 132748 |

# Tabu search - Knapsack problem

**Parametri: maxIterations = 1000, tabuSize = 2**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 129825 |
| **2** | 131680 |
| **3** | 129509 |
| **4** | 129962 |
| **5** | 132017 |
| **6** | 131582 |
| **7** | 131178 |
| **8** | 130352 |
| **9** | 132002 |
| **10** | 130771 |

| Analiza calităţii | |
|---|---|
| **Best value** | 132638 |
| **Average value** | 130925 |
| **Worst value** | 127682 |

# Knapsack – RHC

**Parametru: iterations = 1000**

| Nr rulare | Calitate (fitness value) |
|---|---|
| **1** | 129895 |
| **2** | 131106 |
| **3** | 131109 |
| **4** | 130276 |
| **5** | 131338 |
| **6** | 131795 |
| **7** | 130819 |
| **8** | 132211 |
| **9** | 130046 |
| **10** | 132466 |

| Analiza calităţii | |
|---|---|
| **Best value** | 132466 |
| **Average value** | 131049 |
| **Worst value** | 129895 |

Panaite Mircea - 332

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

# Traveling Salesman Problem – Simulated Annealing

INITIAL_TEMPERATURE = 10000; FINAL_TEMPERATURE = 0.00001; COOLING_FACTOR = 0.99;

## 1)

**Parametri: maxIterations = 100**

| Nr rulare | Cost (distanță) – fitness value |
|---|---|
| **1** | 108743 |
| **2** | 100997 |
| **3** | 102271 |
| **4** | 112630 |
| **5** | 108415 |
| **6** | 106755 |
| **7** | 113813 |
| **8** | 106373 |
| **9** | 113087 |
| **10** | 105397 |

| Analiza calității | |
|---|---|
| **Best value** | 100997 |
| **Average value** | 107321 |
| **Worst value** | 113813 |

## 2)

**Parametri: maxIterations = 1000**

| Nr rulare | Cost (distanță) – fitness value |
|---|---|
| **1** | 50447 |
| **2** | 43415 |
| **3** | 51979 |
| **4** | 46055 |
| **5** | 47910 |
| **6** | 51346 |
| **7** | 51606 |
| **8** | 43354 |
| **9** | 51499 |
| **10** | 48525 |

| Analiza calității | |
|---|---|
| **Best value** | 43354 |
| **Average value** | 48126 |
| **Worst value** | 51979 |

# 3)

**Parametri: maxIterations = 10000**

| Nr rulare | Cost (distanță) – fitness value |
|---|---|
| **1** | 35087 |
| **2** | 35374 |
| **3** | 32725 |
| **4** | 29240 |
| **5** | 34397 |
| **6** | 32210 |
| **7** | 33401 |
| **8** | 34382 |
| **9** | 31502 |
| **10** | 33506 |

| Analiza calității | |
|---|---|
| **Best value** | 29240 |
| **Average value** | 33997 |
| **Worst value** | 35374 |

# Concluzii

i) Tabu Search

În urma analizei experimentale a celor doi algoritmi, în funcție de parametrii aleși am înregistrat o serie de constatări. Pentru Tabu Search, pe măsură ce creștem numărul de iterații maxim și păstrăm tabuSize mic calitatea soluțiilor crește. Cu toate că Tabu Search ne scoate din optime locale, diferența dintre soluții este mai mare decât în cazul soluțiilor generate cu căutarea locală – cu algoritmul de RHC. Analizând comparativ cei doi algoritmi, putem observa că există o anumită diferența privind toate cele trei valori – best, average, worst – însă aceasta nu este flagrantă. Timpul de execuție este similar cu cel din cazul RHC. Algoritmul merită rulat și pentru un număr de iterații mai mare.

ii) Simulated Annealing

Pentru problema comis-voiajor, în urma aplicării algoritmului de Simulated Annealing am constatat că eficientizarea costului este dată de creșterea numărului maxim de iterații (maxIterations). Pentru un număr ca 100 costurile minime sunt foarte mari, însă o creștere pana la 1000 de iterații duce la o eficientizare flagrantă a acestora și la o obținerea unor minimuri optime. Se poate observa că pe măsura creșterii numărului de iterații scade și diferența dintre soluțiile generate în urma celor 10 rulări. Cu toate acestea, pentru creșterea numărului de iterații și generarea tuturor vecinătăților timpul de execuție crește substanțial. Totodată, calitatea best-ului depinde și de calitatea soluției random generate inițial. Algoritmul merită rulat și pentru un număr de iterații mai mare.