Documentatie Lab2

Cerinta

1. Să se implementeze algoritmul *Tabu Search* pentru problema rucsacului.
    a. Generare soluţie iniţială şi vecin
    b. Algoritm şi parametrizare
    c. Experimente pe aceleaşi instanţe de la Tema 1
    d. Comparaţii cu rezultatele obţinute de căutarea locală

Citire si prelucrare date

Exact la fel ca in laboratorul 1

Pseudocod

Algorithm 1: Tabu Search

Data: S - the search space, $maxIter$ - the maximal number of iterations, f - the objective function, the definition of neighborhoods, and the aspiration criteria.
Result: the best solution found
Choose the initial candidate solution $s \in S$
$s^* \leftarrow s$ `// Initialize the best-so-far solution.`
$k \leftarrow 1$
while $k \leq MaxIter$ **do**
  `/* Sample the allowed neighbors of` s `*/`
  Generate a sample $V(s,k)$ of the allowed solutions in $N(s,k)$
    // $s' \in V(s,k) \iff (s' \notin T) \vee (a(k,s') = true)$
  Set s to the best solution in $V(s,k)$
  `/* Update the best-so-far if necessary */`
  if $f(s) < f(s^*)$ **then**
    $s^* \leftarrow s$
  end
  Update T and a
  `/* Start another iteration */`
  $k \leftarrow k + 1$
end

Implementare

- ### Parametri
  - objects = lista de obiecte
  - max_iter = numarul maxim de iteratii
  - bag_max_weight = capacitatea maxima a rucsacului
  - tabu_number = numarul de iteratii pentru care blocam flipuirea unui bit
  - fitness_function = functia de calculare a fitness-ului(defatul este valoarea obiectului)

1. Generarea random a unei solutii candidat initiale

```
# step 1
# generate a valid random solution
while True:
    solution_index, solution_binary = generate_solution(objects)
    rez = verify_solution(solution_binary, objects, bag_max_weight)
    if rez[0] is True:
        current_solution, current_fitness = solution_binary,
fitness_function(rez[1], rez[2])
        break
```

2. Initializarea celei mai bune solutii cu solutia initiala

```
best_solution = current_solution
best_fitness = current_fitness
```

3. Generarea vecinilor

```
# get neighbors
neighbors = get_neighbors_bit_index(current_solution)
```

4. Gasirea celui mai bun vecin non-tabu

```
# from neighborhood solutions, get the best candidate that is not on the Tabu
List(memory list)
if rez_neighbor[0] is True and fitness_neighbor > max_neighbour_non_tabu_fitness
and memory[index] == 0:
    max_neighbour_non_tabu_sol = point
    max_neighbour_non_tabu_fitness = fitness_neighbor
    memory_index_to_update = index
```

5. Actualizarea celei mai bune solutii daca este cazul

```
if max_neighbour_non_tabu_fitness > best_fitness:
    best_solution = max_neighbour_non_tabu_sol
    best_fitness = max_neighbour_non_tabu_fitness
```

6. Pentru iteratia urmatoare ne mutam in vecin chiar daca nu este mai bun decat solutia in care suntem

```
current_solution = max_neighbour_non_tabu_sol
current_fitness = max_neighbour_non_tabu_fitness
```

7. Actualizam memoria

```
# update the memory list with the best candidate solution in this iteration
memory[memory_index_to_update] = tabu_number
```

Toata functia

```
def tabu_search(objects, max_iter, bag_max_weight, tabu_number,
fitness_function=fitness_sum):
    best_solution = ""
    best_fitness = 0
    memory = [0] * len(objects)

    # step 1
    # generate a valid random solution    while True:
        solution_index, solution_binary = generate_solution(objects)
        rez = verify_solution(solution_binary, objects, bag_max_weight)
        if rez[0] is True:
            current_solution, current_fitness = solution_binary,
fitness_function(rez[1], rez[2])
            break

    best_solution = current_solution
    best_fitness = current_fitness

    while max_iter != 0:
        # update memory
        update_memory(memory)

        # step 2
        # get neighbors
        neighbors = get_neighbors_bit_index(current_solution)

        memory_index_to_update = 0

        max_neighbour_non_tabu_sol = ""
        max_neighbour_non_tabu_fitness = 0
```

```
        for (point, index) in neighbors:
            rez_neighbor = verify_solution(point, objects, bag_max_weight)
            fitness_neighbor = fitness_function(rez_neighbor[1],
rez_neighbor[2])

            # from neighborhood solutions, get the best candidate that is not on
the Tabu List(memory list)
            if rez_neighbor[0] is True and fitness_neighbor >
max_neighbour_non_tabu_fitness and memory[index] == 0:
                max_neighbour_non_tabu_sol = point
                max_neighbour_non_tabu_fitness = fitness_neighbor
                memory_index_to_update = index

        if max_neighbour_non_tabu_fitness > best_fitness:
            best_solution = max_neighbour_non_tabu_sol
            best_fitness = max_neighbour_non_tabu_fitness

        current_solution = max_neighbour_non_tabu_sol
        current_fitness = max_neighbour_non_tabu_fitness

        # update the memory list with the best candidate solution in this
iteration
        memory[memory_index_to_update] = tabu_number

        max_iter -= 1

    return best_solution, best_fitness
```

- Functia care genereaza venici este aceeasi ca la laboratorul 1, doar ca returneaza si indexul bitului flip-uit

```
def get_neighbors_bit_index(binary_solution):
    """
    Compute the neighbors of a solution by flipping a bit at a time
    :param binary_solution: a valid solution
    :return: the list with the neighbors and the flipped bit index
    """
    neighbors = []
    for index, bit in enumerate(binary_solution):
        neighbor = binary_solution.copy()
        if bit == 0:
            neighbor[index] = 1
        else:
            neighbor[index] = 0
        neighbors.append((neighbor, index))
    return neighbors
```

Functia care face update la memorie este urmatoarea
Pentru fiecare element din memorie, daca este diferit de 0 se decrementeza cu 1

```
def update_memory(memory):
    for index, elem in enumerate(memory):
        if elem > 0:
            memory[index] = elem - 1
```

Am rulat de cate 10 ori pentru o configuratie si am salvat cea mai buna solutie, cea mai slaba si media solutiilor

```
def run_tabu_search():
    solutions = []
    objects_number, bag_max_weight, objects = read_data_from_file("rucsac-
200.txt")
    for i in range(10):
        sol, val = tabu_search(objects, 100, bag_max_weight, 80)
        print(sol, val)
        solutions.append([sol, val])

    print("-----------------------------------------------")
    best = max(solutions, key=lambda item: item[1])
    worst = min(solutions, key=lambda item: item[1])
    average = mean([sol[1] for sol in solutions])
    print("Best solution: ", best)
    print("Worst solution: ", worst)
    print("Avg solution: ", average)
```

Experimente

- Pentru fisierul cu 20 de obiecte

rucsac-20.txt
Tabu_number = 8

	Best	Average	Worst
10 iter	726	663	621
100 iter	726	722	710
1000 iter	726	716	696
10000 iter	726	711	671
100000 iter	726	712	695

Concluzii:

- cele mai bune rezultate se obtin pentru 100 de iteratii

- solutia cea mai buna este aceeasi indiferent de numarul de iteratii

- se observa ca un numar de iteratii mai mare de 100 nu garanteaza o solutie mai buna
  Comparatie cu rezultatele de la Steepest Ascent Hill Climbing

Hill climbing
Fitness = sum=value
Numar obiecte = 20
k=numar rulari(number_of_executions)

	k=10	k=100	k=1000
Best	726	726	726
Average	681.8	722.8	726

- se observa ca media solutiilor in acest caz creste cu cresterea numarului de iteratii, in timp ce la tabu search nu se intampla acest lucru

- In ambele cazuri se obtine aceeasi cea mai buna solutie indiferent de numarul de iteratii cu care s-au efectuat experimentele

- Pentru fisierul cu 200 de obiecte

rucsac-200.txt
Tabu_number = 80

	Best	Average	Worst
10 iter	132547	131526	130534
100 iter	133033	132541	132038
1000 iter	133235	132651	132104

Concluzii:

- cele mai bune rezultate s-au obtinut cu 1000 de iteratii
  Comparatie cu rezultatele de la Steepest Ascent Hill Climbing

Fitness = sum=value
Numar obiecte = 200

	k=10	k=100	k=1000
Best	132057	132815	133276
Average	122723.78166	122690.3588	122999.0628

- tabu search cu tabu-number=80 ofera rezultate ceva mai bune decat sahc, insa cele mai bune solutie in ambele cazuri(indiferent de numarul de iteratii) sunt foarte apropiate. Media solutiilor la tabu_search este mai buna.

Cerinta

2. Să se implementeze algoritmul *Simulated Annealing / Tabu Search* (la alegere) pentru problema TSP (instanţa este indicată la laborator).
    a. Generare soluţie iniţială şi vecin
    b. Algoritm şi parametrizare
    c. Experimente pe o instanţă TSP din lista de mai jos

Am ales sa implementez **Simulated Annealing**

Citire si prelucrare date

Fisierul pe care am lucrat este lin105.tsp

```
def read_city_data(filename):
    """
    Read cities data from file
    :param filename: the name of the file
    :return: a list of cities and dimension(the number of cities)
    """
    file = open(filename, 'r')
    # Read instance header
    name = file.readline().strip().split()[1]  # NAME
    type_ = file.readline().strip().split()[1]  # TYPE
    comment = file.readline().strip().split()[1]  # COMMENT
    dimension = int(file.readline().strip().split()[1])  # DIMENSION
    edge_weight_type = file.readline().strip().split()[1]  # EDGE_WEIGHT_TYPE
    file.readline()

    # Read node list
    cities = []
    N = int(dimension)
    for i in range(0, N):
        j, x, y = file.readline().strip().split()
        cities.append([int(j), int(x), int(y)])

    # Close input file
    file.close()

    return cities, dimension
```

Functia de mai sus parcurge fisierul, citeste datele din header apoi fiecare linie cu orase si adauga intr-o lista triplete de forma index_oras, prima coordonata, a doua coordonata.
Pe urma fieacare triplet din lista este transformat intr-un obiect de tip oras astfel:

```
def set_data(filename):
    """
    Transform the list of cities read from file into a list of city objects
    :param filename: the name of the file
    :return: a list of city objects and dimension
    """
    cities, dimension = read_city_data(filename)
    cities_object_list = []
    for elem in cities:
        cities_object_list.append(City(elem[0], elem[1], elem[2]))
    return cities_object_list, dimension
```

Unde clasa *CIty* este urmatoarea:

```
class City:
    def __init__(self, index, x, y):
        """
        :param index: the city index from the file
        :param x: x coordinate
        :param y: y coordinate
        """
        self.index = index
        self.x = x
        self.y = y

    def get_distance_from_city(self, other_city):
        """
        Compute the euclidian distance between two cities
        :param other_city: the city to find distance between
        :return: the distance as floating number
        """
        x_dif = self.x - other_city.x
        y_dif = self.y - other_city.y
        dist_euc_2d = math.sqrt(x_dif * x_dif + y_dif * y_dif)
        return dist_euc_2d
```

Metrica folosita pentru calcularea distantei dintre orase este distanta euclidiana in 2D
Lista cu biectele de tip oras si numarul lor le salvez intr-un obiect *Data* de unde vor fi transmise celorlalte clase

```
class Data:
    """
```

```
    A class for storing the initial list of city objects and dimension
    """
    def __init__(self, filename):
        self.cities, self.dimension = set_data(filename)

    def get_city(self, index):
        return self.cities[index]
```

Implementare

a. Generare solutie initiala si vecin

- O solutie valida reprezinta o permutare a oraselor (=un traseu care sa viziteze toate orasele o singura data)
- Calitatea solutiei reprezinta distanta totala a traseului la care se adauga distanta de la ultimul oras vizitat pana la orasul de inceput
- Distanta totala = suma distantelor dintre cate doua orase in ordinea de parcurgere a acestora
- Pentru reprezentarea unei solutii am implementat o clasa *Route* care retine o permutare a oraselor, numarul de orase(=dimensiunea) si calitatea solutiei(=distanta)

```
class Route:
    """
    A class for representing a possible solution of tsp, meaning a valid
configuration   of all cities
    """
    def __init__(self, dimension, cities):
        self.dimension = dimension
        self.cities = cities
        self.distance = self.get_route_distance()

    def get_random_route(self):
        """
        Generate a random route/configuration of cities
        :return: a Route object with random cities order
        """
        cities_copy = self.cities.copy()
        random.shuffle(cities_copy)
        return Route(self.dimension, cities_copy)

    def swap_2_cities(self):
        """
        Swap two random cities from the list
        :return: a Route object with two random cities swapped
        """
        index1 = random.randint(1, 105)  #primul index random pentru swap
```

```
        index2 = random.randint(1, 105)  # al doilea index random pentru swap
        city1_route_index = [self.cities.index(city) for city in self.cities if
city.index == index1][0]
        city2_route_index = [self.cities.index(city) for city in self.cities if
city.index == index2][0]
        new_cities_order = self.cities.copy()
        city1 = new_cities_order[city1_route_index]
        city2 = new_cities_order[city2_route_index]
        new_cities_order[city1_route_index] = city2
        new_cities_order[city2_route_index] = city1

        return Route(self.dimension, new_cities_order)

    def get_route_distance(self):
        """
        Computes the overall distance of a route, starting from the first city
in the list and ending into it
        :return: the total distance of the route
        """
        distance = 0
        for i in range(self.dimension-1):
            distance += self.cities[i].get_distance_from_city(self.cities[i+1])
        distance += self.cities[-1].get_distance_from_city(self.cities[0])  #
add the distance between last and first city to close the loop

        return distance

    def get_cities_order(self):
        """
        Get the cities indexes from the route
        :return: a list with cities indexes in the crossing order
        """
        return [city.index for city in self.cities]
```

- Generarea solutiei initiale se face prin generarea aleatoare a unei permutari a oraselor (metoda *get_random_route* amesteca(sfuffle) orasele si returneaza o permutare)
- Un vecin = o permutare rezultata de la permutarea initiala prin interschimbarea a doua orase(2-swap) si este realizata de metoda *swap_2_cities*

a. Algoritm si parametrizare

Implementarea algoritmului se bazeaza pe pseudocodul din seminarul 2:

Exemplu aplicare SA pentru TSP:

```
T = 10000; alpha = 0.9999; minT = 0.00001;
c = createRandomSolution();
while (T > minT)
        repeat
                x = GetVecin(c);  // swap 2 cities / 2-opt / etc
                delta = eval(x) – eval(c);
                if (delta < 0) then c = x
                else if random.NextDouble() < Math.Exp(-delta/T) then c=x
        until (max-iterations)
        T = alpha*T;
 end while
return c
```

Pentru implementare am creat clasa *SimulatedAnnealing*

```
class SimulatedAnnealing:
    """
    Class for simulated annealing algorithm
    """
    def __init__(self, temperature, min_temperature, alpha, max_iterations,
cities, dimension):
        self.temperature = temperature
        self.min_temperature = min_temperature
        self.alpha = alpha
        self.max_iterations = max_iterations
        self.cities = cities
        self.dimension = dimension

    def simulated_annealing_alg(self):
        current_route = Route(self.dimension, self.cities).get_random_route()  #
starts with a random route
        while self.temperature > self.min_temperature:
            iterations = self.max_iterations
            while iterations > 0:  # while there are iterations left
                neighbor_route = current_route.swap_2_cities()  # generate a
neighbor by swapping two cities
                delta = neighbor_route.get_route_distance() -
current_route.get_route_distance()
                if delta < 0:  # if the neighbor route is shorter move into it
                    current_route = neighbor_route
                elif random.uniform(0, 1) < math.exp(-delta/self.temperature):
# if the neighbour route is worst it can be choosen by a probability conditioned
```

```
by distance and temperature
                        current_route = neighbor_route
                iterations -= 1

            self.temperature = self.alpha * self.temperature  # decrease the
temperature

        return current_route


    def simulated_annealing_alg2(self):
        current_route = Route(self.dimension, self.cities).get_random_route()  #
starts with a random route
        iterations = self.max_iterations
        while iterations > 0: # while there are iterations left
            while self.temperature > self.min_temperature:
                neighbor_route = current_route.swap_2_cities()  # generate a
neighbor by swapping two cities
                delta = neighbor_route.get_route_distance() -
current_route.get_route_distance()
                if delta < 0:  # if the neighbor route is shorter move into it
                    current_route = neighbor_route
                elif random.uniform(0, 1) < math.exp(-delta/self.temperature):
# if the neighbour route is worst it can be choosen by a probability conditioned
by distance and temperature
                current_route = neighbor_route

                self.temperature = self.alpha * self.temperature  # decrease the
temperature
            iterations -= 1

        return current_route
```

Parametri

- temperature = temperatura maxima, initiala
- min_temperature = temperatura minima, de oprire (halting criteria)
- max_iterations = numarul maxim de iteratii (termination crieteria)
- alpha = un coeficient utilizat pentru a scadea temperatura
- cities = lista de orase
- dimension = numarul de orase

Observatie!

- Metoda *simulated_annealing_alg* (metoda 1) implementeaza exact speudocodul din seminar, dar a doua metoda *simulated_annealing_alg2* (metoda 2) interschimba cele doua conditii de oprire.

- In primul caz temperatura este conditie exterioara, iar numarul de iteratii conditie interioara.
- In al doilea caz temperatura este conditie interioara, iar numarul de iteratii conditie exterioara.

Modificarea aceasta influenteaza considerabil timpul de rulare. Primul caz este mai costisitor din acest punct de vedere.

c. Experimente pe o instanta TSP din lista

Experimentele au fost efectuate pe fisierul lin105.tsp, iar din documentatia corespunzatoare, cel mai bun rezultat obtinut este 14379.

Metoda 1

Rezultatele experimentelor

Metoda 1 temperatura in exterior

T=1 t_min=0.0001 Alpha = 0.999

	Best	Average	Worst
10 iter	25814.206	29685.307	35779.202
100 iter	24945.616	27414.698	32246.828
1000 iter	23284.562	29177.623	33457.722

T=10 t_min=0.001 Alpha=0.99

	Best	Average	Worst
10 iter	30876.056	35363.336	39561.993
100 iter	24413.670	27378.791	33311.428
1000 iter	21915.833	25916.316	28308.154
10000 iter	22113.633	25612.025	34256.523

T=1000 t_min=0.1 Alpha=0.9

	Best	Average	Worst
10 iter	52276.355	55784.855	60926.680
100 iter	27570.164	308529.201	37522.477
1000 iter	21633.260	23672.017	2589.135

T=10000 t_min=0.001 Alpha=0.99

	Best	Average	Worst
10 iter	28041.795	31258.816	34833.373
100 iter	21055.641	23198.165	25487.260
1000 iter	18107.581	19149.523	20616.221

- Cele mai bune rezultate(mai apropiate de 14379) se obtin cu ultima configuratie si 1000 de iteratii (avand un timp de rulare de aoproximativ 4 ore)
- Cu cat alpha este mai apropiat de 1 si diferenta dintre temperatura initiala si temperatura minima este mai mare, cu atat se obtin solutii mai bune, insa timpul de executie creste considerabil
- Valorile empirice rezultate din a doua configuratie nu valideaza ipoteza intuitiva conform careia calitatea solutiei creste mereu cu cresterea numarului de iteratii. Pentru 10000 de iteratii rezultatele experimentale sunt mai slabe calitativ decat

pentru 1000 de iteratii, doar in media colutiilor se observa o foarte mica imbunatatire care poate fi pur intamplatoare.

Metoda 2

Rezultatele experimentelor

Metoda 2 temperatura in interior

T=10000 t_min=0.00001 Alpha=0.9999

	Best	Average	Worst
10 iter	20612.039	23350.448	25345.700
100 iter	19590.789	22741.864	24687.640
1000 iter	24737.013	29194.503	33917.256
10000 iter	19748.678	23675.926	27535.024

T=100000 t_min=0.00001 Alpha=0.9999

	Best	Average	Worst
10 iter	22355.972	23840.150	25964.179
100 iter	22362.572	23710.788	26955.502
1000 iter	21640.513	23893.423	25272.797
10000 iter	20400.899	23311.330	26096.23
100000 iter	22214.075	23796.507	26683.074

T=1000000 t_min=0.00001 Alpha=0.9999

	Best	Average	Worst
10 iter	19879.688	22476.159	23657.041
100 iter	19470.220	22754.836	26145.042
1000 iter	20101.215	23014.039	25660.507
10000 iter	21733.366	23681.777	24944.078

- Cele mai bune rezultate au fost obtinute cu 100 de iteratii in ultima configuratie
- Nu putem stabili o legatura generala intre cresterea numarului de iteratii si cresterea calitatii solutiei.
- Cu cat temperatura este mai mare, din experimente pare ca solutia are o calitate mai buna(exemplu cele mai bune solutii din ultimul tabel)