Documentation Lab1

Problema Rucsacului

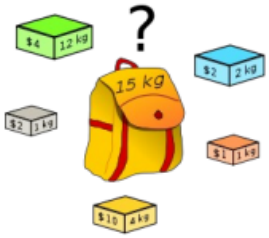
→ datele se citesc din fisiere, unul cu 20 de obiecte si unul cu 200
→ rezultatele experimentelor se gasesc in folderul results, cele actuale au cuvantul "new" in numele fisierului

4 PROBLEMA RUCSACULUI

- n obiecte, fiecare obiect are o valoare (v) şi o greutate (w)
- *Obiectiv: puneţi în rucsac valoarea maximă fără a depăşi greutatea maximă admisă W*
- $x_i=1$ înseamnă obiectul i este pus în rucsac
- $x_i=0$ înseamnă obiectul i nu este pus în rucsac

maximize $\sum_{i=1}^{n} v_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0,1\}$.



Notiuni utilizate in continuare:

- Spatiu de cautare = multimea tuturor solutiilor posibile, aici multimea tuturor vectorilor cu n termeni 0 sau 1
- Solutie valida = solutie din spatiul de cautare pentru care suma greutatilor obiectelor alese este mai mica decat capacitatea rucsacului

Functii de fitness folosite:

```python
def fitness_ratio(value, weight):
    return value / weight


def fitness_sum(value, weight=0):
    return value
```

Random search

1. Să se implementeze o metodă de căutare aleatoare (random search) pentru problema rucsacului.
   - Să se genereze o soluţie aleatoare şi să se verifice dacă este validă.
   - Să se determine calitatea soluţiei generate.
   - Pentru k soluţii generate aleator, să se determine cea mai bună soluţie.

Algoritm

Pasi

- Genearea unei solutii aleatoare din spatiul de cautare

```
def generate_solution(objects):
    """
    Generate a random binary solution
    :param objects: list of the form [(value, weight)]
        :return: a possible solution in real representation(solution_index) and
a binary
        representation
    """
    solution_index = []
    binary_representation = []
    for index, (o_value, o_weight) in enumerate(objects):
        choose_obj = np.random.choice([0, 1])
        if choose_obj == 1:
            solution_index.append(index)
        binary_representation.append(choose_obj)

    return solution_index, binary_representation
```

Pseudocod:

```
lista_solutie=[ ]
for index in obiecte do:
         genereaza 0 sau 1 si adauga in lista_solutie
returneaza lista_solutie
```

→functia generate_solution(objects) de mai sus primeste ca parametri lista cu obiecte si returneaza atat reprezentarea binara a solutiei cat si reprezentarea reala(lista cu indecsi obiectelor)

- Validarea solutiei

```
def verify_solution(binary_solution, object_list, max_weight):
    """
    Verifies is a solution is valid. A Solution is valid if the total objects
weight is
    less that the bag capacity
    :param binary_solution: the binary representation of the solution(list of
bits)    :
    param object_list: the list with object value and weight
    :param max_weight: the bag max weight
    :return: True is the solution is valid, False otherwise and the total weight
```

```
and
    value put in the bag
    """
    solution_weight = [bit * obj[1] for bit, obj in zip(binary_solution,
object_list)]
    solution_value = [bit * obj[0] for bit, obj in zip(binary_solution,
object_list)]

    rez_w = sum(solution_weight)
    rez_v = sum(solution_value)

    if rez_w <= max_weight:
        return True, rez_v, rez_w
    return False, rez_v, rez_w
```

→ functia de mai sus "inmulteste" reprezentarea binara a solutiei cu lista de obiecte, prin urmare vor fi luate in considerare greutatile si valorile obiectelor carora le corespunde 1 in reprezentare
→ se insumeaza acecste greutati si valori
→ se returneaza 3 valori, prima(True sau False) indica daca solutia e valida, apoi urmeaza valoarea si greutatea totala

- Determinarea celei mai bune solutii si teste pentru valori diferite ale lui k
- functia de mai jos este rulata de cate 10 ori pentru iteration_number = 10, 100 si 1000

```
def run_random_search(iteration_number):
    objects_number, bag_max_weight, objects = read_data_from_file("rucsac-
200.txt")
    best_solution_quality_sum = 0
    best_solution_quality_ratio = 0
    avg_solution_quality_sum = 0
    avg_solution_quality_ratio = 0
    best_sol_reprez_sum = []
    best_sol_reprez_ratio = []
    valid_iter = 0

    for i in range(iteration_number):
        # -------run algorithm--------
        solution_index, binary_representation = generate_solution(objects)
        result = verify_solution(binary_representation, objects, bag_max_weight)

        # --------stats about solutions' quality-------
        quality1 = 0
        quality2 = 0

        if result[0] is True:
```

```
            valid_iter += 1
            quality1 = result[1]  # the quality for the solution with
fitness=sum
            quality2 = fitness_ratio(result[1], result[2])  # the quality for
the solution with fitness=ratio
            if best_solution_quality_sum < quality1:
                best_solution_quality_sum = quality1
                best_sol_reprez_sum = binary_representation
            if best_solution_quality_ratio < quality2:
                best_solution_quality_ratio = quality2
                best_sol_reprez_ratio = binary_representation
            avg_solution_quality_sum += quality1
            avg_solution_quality_ratio += quality2

        print('Iteration:{} Valid:{} Solution:{} Weight:{}  Value:{}
Fitness_sum:{}  Fitness_ratio:{} '.format(i, result[0],

binary_representation,

result[2],

result[1],

quality1,

quality2))

    print("Valid solutions:{}".format(valid_iter))
    avg_solution_quality_sum = avg_solution_quality_sum / valid_iter
    avg_solution_quality_ratio = avg_solution_quality_ratio / valid_iter
    print("Best solution sum(v):{}, solution:
{}".format(best_solution_quality_sum, best_sol_reprez_sum))
    print("Best solution ratio(v/w):{}, solution:
{}".format(best_solution_quality_ratio, best_sol_reprez_ratio))
    print("Avg solution sum(v):{}".format(avg_solution_quality_sum))
    print("Avg solution ratio(v/w):{}".format(avg_solution_quality_ratio))
```

Parametri: numarul de iteratii = k
→functia genereaza k solutii din spatiul de cautare, le verifica si calculeaza cea mai buna solutie dintre cele gasite, precum si solutia medie

Sinteza Rezultate

→rezultate mai detaliate se pot vedea in fisierele txt cu rezultate din proiectul cu codul sursa

→pentru functia de fitness luata ca valoarea obiectului rezultatele sunt

Random search
Fitness = sum=value
Numar obiecte = 20

	k=10	k=100	k=1000
Best	684	685	699
Average	405.3647	409.4213	411.5360

Fitness = sum=value
Numar obiecte = 200

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

Concluzii

```
- cu cat creste numarul de iteratii cea mai buna solutie este mai buna (sansele de a optine o solutie obtima cat mai buna sunt mai mari)
- un numar mai mare de iteratii ofera si o solutie medie mai buna
-pentru k=1000 se obtine cea mai buna solutie si cea mai buna medie a solutiilor, insa diferenta fata de k=100 nu este mare din punct de vedere al calitatii
```

→pentru functia de fitness luata ca raport value/weight rezultatele sunt

Fitness = value/weight
Numar obiecte = 20

	k=10	k=100	k=1000
Best fitness	1.4615384615384615	1.4461538461538461	1.794238683127572
Average fitness	1.0381216667497215	0.9991018651925152	0.9850701935773968

Fitness = value/weight
Numar obiecte = 200

	k=10	k=100	k=1000
Best fitness	1.172366059068866	1.1853375317957884	1.1889175127544578
Average fitness	1.1673027453948397	1.1676660164059147	1.170317805046888

→ cu cat raportul este mai mare cu atat valoarea / unitate de greuate adaugata in rucsac este mai mare

Steepest hill climbing

Algoritm

STEEPEST ASCENT HILL-CLIMBING (SAHC)

1. Se selectează un punct aleator **c** (*current hilltop*) în spaţiul de căutare.
2. Se determină toate punctele *x* din vecinătatea lui c: $x \in N(c)$
3. Dacă oricare $x \in N(c)$ are un fitness mai bun decât c atunci c=x , unde x are cea mai bună valoare eval(x).

4. Dacă nici un punct $x \in N(c)$ nu are un fitness mai bun decât **c**, se salvează **c şi** se trece la ***pasul 1.*** Altfel, se trece la ***pasul 2*** cu noul **c**.
5. După un numar maxim de evaluări, se returnează cel mai bun **c** (hilltop).

Pseudocod:

Algorithm 2: Stochastic Hill Climbing Algorithm

```
1  current position = initial solution;
2  repeat
3      for All neighbours of current position do
4          Obtain a random neighbour;
5          if cost of neighbour ≤ cost of current position then
6              current position = neighbour position;
7              break;
8          end
9      end
10 until cost of current position ≤ cost of all its neighbours;
```

Implementarea

→comentariile explica in detaliu ce se intampla la fiecare pas

```
def shc_alg(objects, max_iter, bag_max_weight, fitness_function):
    """
    Steepest hill climbing algorithm
    :param objects: list of objects with values and weight
    :param max_iter: number of restarting times when the alg is stuck in a local
maximum
    :param bag_max_weight: bag capacity
    :param fitness_function: the fitness function by which the neighbors are
evaluated
    :return: a list with all local maximum solutions for a run
    """
    results = []

    while True:
        # step 1
        # generate a valid random starting solution
        print("iteration: ", max_iter)
        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

        # step 2
        # get neighbors and do hill climbing
        while True:
            neighbors = get_neighbors(current_solution)
            found = False

            # verifies each neighbor and if its fitness is grater takes that
point as the current solution
            # continues until it finds the neighbor with the greatest fitness
            for point in neighbors:
                rez_neighbor = verify_solution(point, objects, bag_max_weight)
                fitness_neighbor = fitness_function(rez_neighbor[1],
rez_neighbor[2])

                if rez_neighbor[0] is True and fitness_neighbor >
current_fitness:
                    current_solution = point
                    current_fitness = fitness_neighbor
                    found = True
                    print("True ", current_solution, "fitness: ",
```

```
current_fitness, "value: ", rez_neighbor[1])

            # if all neighbors have smaller fitness save the result and go to
step 1 if the number of iteration is not 0
            if found is False:
                max_iter -= 1
                results.append([current_solution, current_fitness])
                print("False ", current_solution, current_fitness)
                break

        if max_iter == 0:
            break

    return results
```

Parametri:

- objects = lista de obiecte
- max_iter = numarul de reluari(salt la pasul1) cand algoritmul nu gaseste vecini mai buni = se blocheaza intr0un optim local (am rulat cu 10)
- bag_max_weight = capacitatea rucsacului
- fitness_function = functia de fitness folosita(una dintre cele doua de mai sus)
  → se returneaza o lista cu toate solutiile(optimele locale) salvate
  →din acesta lista ulterior extrac maximul

Am rulat cu 'number_of executions' = 10 si numarul de iteratii = 10, 100 si 1000 pentru fiecare fisier

```
def run_shc(max_iter, number_of_executions):
    objects_number, bag_max_weight, objects = read_data_from_file("rucsac-
200.txt")
    rez = iterate_sch_alg(objects, max_iter, bag_max_weight,
number_of_executions, fitness_function=fitness_sum)
    best_sol = []
    best_sol_val = 0
    avg_sol_val = 0
    print("-----------------------------------")

    for i in rez:
        print(i)
        if i[1] > best_sol_val:
            best_sol_val = i[1]
            best_sol = i[0]
        avg_sol_val += i[1]

    avg_sol_val = avg_sol_val / number_of_executions
    print("Number of executions: {}".format(number_of_executions))
    print("Number of restarting alg when stuck in local maximum:
```

```
{}".format(max_iter))
    print("Best solution: {}, value: {} ".format(best_sol, best_sol_val))
    print("Avg solution value: ", avg_sol_val)
```

Concluzii:

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

Hill climbing
Fitness = sum=value
Numar obiecte = 200
k=numar rulari(number_of_executions)

	k=10	k=100	k=1000
Best	132822	133644	133940
Average	132398.1	133003.1	133588.1

→pentru 20 de obiecte cea mai buna solutie este aceeasi(726) indiferent de numarul de iteratii
→cresterea numarului de iteratii ofera o medie a solutiei mai buna. Pentru 100 de iteratii se obtine o medie foarte aproape de cea mai buna solutie. Pentru 1000 de iteratii solutia medie este egala cu cea mai buna solutie

→pentru 200 de obiecte nu se obtine aceeasi cea mai buna solutie, insa cu cat creste numarul de iteratii solutia este din ce in ce mai buna, pentru k=1000 se obtine cea mai buna solutie dintre configuratiile pentru iteratii
→de asemenea, calitatea solutiei medii creste cu cresterea numarului de iteratii, insa cresterea nu este foarte mare

→pentru functia de fitness luata ca raport, algoritmul nu performeaza bine, maximizand raportul valoare/greutate va genera mereu(indiferent de numarul de rulari) solutia in care este ales un singur obiect, si anume acel obiect cu raportul mai mare

Pentru 20 obiecte
Best solution: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], value: 14.0
Avg solution value: 14.0

Pentru 200 de obiecte
Best solution: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], ratio:
3.5641025641025643
Avg solution ratio: 3.5641025641025643

Comparatii

Atat pentru 20 de obiecte cat si pentru 200, indiferent de numarul de rulari, hill climbing ofera rezultate mai bune, lucru vizibil mai ales cu 20 de obiecte unde ajunge la aceeasi cea mai buna solutie.

Pentru 200 de obiecte cele mai bune solutii cu random search se apropie de cele cu sahc, mai ales pentru un numar de rulari cat mai mare(k=1000). Media solutiilor este mai buna cu sahc decat cu random search.

Resurse
Curs 1 + Seminar 1
https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/