Explicatii algoritmi si pseudocod

După citirea din fișier al elementelor este returnată o listă care conține mai multe liste: prima listă sunt obiectele date în problemă, a doua listă este valoarea maximă a greutății, a treia listă este numărul elementelor din prima listă. Pentru lista numită „obj”:

```
# obj[0] - lista obiecte, obj[1] - valoare maxima, obj[2] - nr obiecte
```

Functii folosite pentru ambele cautari

```
def validateArray(array, objects, value):
    sum = 0
    pentru i pana la lungime(array):
        daca array[i] == 1:
            sum += objects[i][1]
        daca sum > value:
            returneaza Fals
    returneaza Adevarat
```

Funcția „validateArray” verifică dacă o listă compusă din 0 și 1 este validă, adică dacă suma greutăților elementelor din vector nu depășește greutatea dată.

```
def validSolution(obj):
    randomArray = np.random.choice([0, 1], size=(obj[2],))
    cat timp validateArray(randomArray, obj[0], obj[1]) nu este Fals:
        randomArray = np.random.choice([0, 1], size=(obj[2],))

    returneaza randomArray
```

Funcția „validSolution” returnează o listă random compusă din 0 și 1, care respectă condițiile funcției „validateArray” (adică greutatea elementelor să nu depășească greutatea dată). Am folosit această funcție pentru a mă asigura că programul lucrează doar cu liste (soluții ale problemei) care sunt valide.

```
def fitness(array, objects, value):
    gr = 0
    valoare = 0
    pentru i pana la lungime(array):
        daca array[i] == 1:
            gr = gr + objects[i][1]
            valoare = valoare + objects[i][0]
    daca gr > value:
```

```
        returneaza -1
    returneaza valoare
```

Funcția de fitness returnează valoarea obiectelor, cât timp acestea nu depășesc greutatea dată. Dacă nu se depășește greutatea dată, este returnată valoarea obiectelor. Dacă valoarea dată este depașită, se returnează -1.

Pentru Random Search

```
def kSolutions(k: int, obj: list, nr_executii: int, filename):
    # obj[0] - lista obiecte, obj[1] - valoare maxima, obj[2] - nr obiecte

    max: int = -1
    maxSol: list = []
    solutii: list = []

    pentru executie pana la nr_executii:
        start_time = time.time()
        pentru  i pana la k:
            randomArray = validSolution(obj)

            value = fitness(randomArray, obj[0], obj[1])
            daca value > max:
                maxSol = randomArray
                max = value

        solutii.adauga_in_lista([maxSol, max, start_time])

    scrie_in_fisier(filename, k, nr_executii, solutii);
```

Aceasta este funcția pentru random search. Primește ca parametri k (numărul de soluții dorite), numărul de execuții și numele fișierului în care vor fi scrise rezultatele. Pentru fiecare “i” până la „k”, se găsește o soluție validă a problemei, se calculează valoarea obiectelor cu funcția de fitness, iar dacă valoarea este mai mare decât maximul (începem cu maxim = -1), atunci se reține soluția respectivă. După ce au fost calculate toate soluțiile (k), se reține cea mai buna pentru a fi scrisă în fișier.

Pentru Random Hill Climbing(RHC)

```
def vecin(array, index):
    a = array
    daca a[index] == 0:
        a[index] = 1
    altfel
```

```
        a[index] = 0
    returneaza a
```

Funcția „vecin” găsește vecinul unei liste, pentru un index dat.

```
def bestSol(array, obj, sol, i):
    cat timp i < len(array) executa
        vec = vecin(array, i)
        fitVec = fitness(vec, obj[0], obj[1])

        daca fitVec > sol:
            returneaza vec, fitVec, i
        i = i + 1

    return [], -1, -1
```

Funcția „bestSol” găsește primul cel mai bun vecin pentru o listă de soluții și un index dat, pentru a găsi vecinul pornind de la acel index. Dacă nu există un vecin mai bun, se returnează -1.

```
def hillClimbing(k, obj: list, nr_executii, filename):

    solutii = []
    bestSolutii = []
    cpyk = k
    nrexec = nr_executii
    timp = []

    # bst - obiectul ce este returnat din functia “bestSol” - bst[0] vecinul,
bst[1] valoarea totala a vecinului, bst[2] indexul

    # obj[0] - lista obiecte, obj[1] - valoare maxima, obj[2] - nr obiecte

    cat timp nr_executii > 0 executa
        k = cpyk
        start_time = time.time()
        cat timp k > 0 executa
            c = validSolution(obj)
            sol = fitness(c, obj[0], obj[1])
            bst = bestSol(c, obj, sol, 0)

            cat timp True executa
                daca bst[1] == -1:
                    iesi_din_structura_repetitiva
                c = bst[0]
                sol = bst[1]
                bst = bestSol(c, obj, sol, bst[2] + 1)

            k = k - 1
            solutii.adauga_in_lista([c, sol])
```

```
        #print(solutii)

        bestSolutii.adauga_in_lista(returnBestSolution(solutii))
        timp.adauga_in_lista(start_time)
        nr_executii = nr_executii - 1

    scrie_in_fisier(filename, cpyk, nrexec, bestSolutii, timp)
```

Aceasta este funcția pentru Next Ascent Hill Climbing. Primește ca parametrii k – numărul soluțiilor dorite (sau numărul de evaluări), numărul de execuții și numele fișierului în care vor fi scrise rezultatele. Cât timp numărul de execuții este mai mare decât 0, vom găsi o soluție validă cu funcția „validSolution”, vom calcula valoarea soluției cu funcția de fitness, apoi vom găsi primul cel mai bun vecin al soluției cu funcția „bestSol”, având ca parametru 0 pentru index. Apoi cât timp funcția „bestSol” nu returnează -1, vom repeta apelarea acesteia, pentru a găsi primul cel mai bun vecin al soluției. Când funcția va returna -1, înseamnă că nu a găsit alt vecin mai bun, iar căutarea se reia de la primul pas, generându-se o altă soluție validă pentru care să se găsească vecini. Cea mai bună soluție de la fiecare execuție din numărul de execuții va fi scrisă în fișier.

Tabele de date

Random Search

Instanța problemei	k	Valoare medie	Valoarea cea mai buna	Număr execuții	Timpul mediu de execuție
rucsac-20.txt	100	653.65	658	20	32314847235.979713
	200	673.45	681		32314847321.87208
	300	688.0	688		32314847443.55495
rucsac-200.txt	100	132855.65	133199		32314849545.168007
	200	132725.55	132927		32314849738.040455
	300	132697.5	133115		32314849848.068707
Obiecte = [(91, 29), (60, 65), (61, 71), (9, 60), (79, 45)]	100	231.0	231		32314845122.1224
	200	231.0	231		32314845418.496666

Capacitate maximă = 175	300	231.0	231		32314845510.304394

Next Ascent Hill Climbing(NAHC)

Instanța problemei	K	Valoare medie	Valoarea cea mai buna	Numar execuții	Timpul mediu de execuție
rucsac-20.txt	100	691.85	703	20	1615742836.2319245
	200	701.5	705		1615742844.4788675
	300	692.15	716		1615742850.965983
rucsac-200.txt	100	133687.4	133727		1615742953.9820402
	200	133426.55	133528		1615743002.5614035
	300	133538.05	133597		1615743115.1554067
Obiecte = [(91, 29), (60, 65), (61, 71), (9, 60), (79, 45)] Capacitate maximă = 175	100	231.0	231		1615742763.095374
	200	231.0	231		1615742767.4605277
	300	231.0	231		1615742771.9093974

Observații

Pentru un număr foarte mic de date, cum ar fi 5 obiecte, ambii algoritmi dau aceleași valori la fiecare execuție. Pentru 20 și 200 de obiecte, algoritmii dau valori diferite, deși sunt totuși multe valori care se repetă. Next Ascent Hill Climbing a dat valori medii și maxime mai bune decât random search pentru fiecare instanță cu 20 de obiecte, la fel și pentru 200 de obiecte.