Laborator 1 AI

Algoritmi folositi pentru ambele probleme

```
def generareSolAleatoare(obiecte):
    n = len(obiecte)
    sol = np.random.randint(2, size=n)
    print("prima solutie", sol)
    return sol
```

Functia “genereazaSolAleatoare(obiecte)”, genereaza o solutie aleatoare binara formata din 0 si 1

```
def evaluare(obiecte, sol):
    greutate = 0
    valoare = 0
    l = len(obiecte)
    for i in range(l - 1):
        greutate = greutate + sol[i] * obiecte[i][1]
        valoare = valoare + sol[i] * obiecte[i][0]
    print(greutate, valoare)
    return greutate, valoare

def fctFitness(obiecte, sol, greutateTotala):
    greutate, valoare = evaluare(obiecte, sol)
    print("greutate", greutate)
    return greutate <= greutateTotala, valoare
```

Functiile “evaluare” si “fctFitness” valideaza solutia aleatoare, returneaza greutatea si costul daca greutatea este mai mica ca si greutatea totala care poate fi pusa in ghiozdan

```
def run10times(k, obiecte, greutateTotala):
    i = 0
    sum = 0
    average = 0
    best = 0
    cost = 0
    indice = 0
    costuri = []
    while i<10:
        cost = generareSolutieAleatoare(obiecte, greutateTotala, k)
        i=i+1
        sum = sum + cost
        costuri.append(cost)

    for p in range(10):
        if costuri[p] > best:
            best = costuri[p]
            indice = p
    print("final sum", sum)
```

```
    average = sum / 10
    print("lista costuri finale", costuri)
    print("cel mai bun cost este", costuri[indice])
    print("media este", average)
    with open('solutiiTime.txt', 'a') as f:
        f.write(str(k))
        f.write(" ")
        f.write(str(costuri[indice]))
        f.write(" ")
        f.write(str(average))
        f.write(" ")
```

Functia run10times ruleaza de 10 ori acelasi algoritm pentru un anumit numar k, calculand cel mai bun cost si costul mediu al solutiilor generate in cele 10 rulari

```
end = time.time()
with open('solutiiTimeSahc1.txt', 'a') as f:
    f.write(str({end - start}))
    f.write("\n")
```

Aceste linii de cod calculeaza timpul de executie si il pune intr-un fisier

Problema Rucsacului Random Search

```
def generareSolutieAleatoare(obiecte, greutateTotala, k):
    i = 0
    solutii = []
    costuri = []
    sirGreutate = []
    indice = -1
    print(i, k)
    sum = 0
    c = 0
    while i<k:
        print("a ajuns in while")
        sol = generareSolAleatoare(obiecte)
        print(sol)
        greutate, valoare = fctFitness(obiecte, sol, greutateTotala)
        print("greutate din while", greutate)
        if greutate:
            print("greutate din if", greutate)
            solutii.append(sol)
            costuri.append(valoare)
            sirGreutate.append(greutate)
            sum = sum + valoare
            c = c + 1
            i = i+1
    p = 0
    best = 0
    for p in range(k):
        if costuri[p] > best:
            best = costuri[p]
            indice = p
    average = sum/c
```

```
    print("average",average)
    with open('solutii.txt', 'a') as f:
        f.write(str(k))
        f.write(" ")
        f.write(str(costuri[indice]))
        f.write(" ")
        f.write(str(solutii[indice]))
        f.write("\n")
    print("rezultat final")
    print("solutii",solutii)
    print("costuri", costuri)
    print("sol fin", solutii[indice])
    print("cost fin", costuri[indice])
    print("greutate fin", sirGreutate[indice])
    return costuri[indice]
```

Functia generareSolutieAleatoare genereaza cea mai buna solutie cu cel mai bun cost dupa care se ruleaza run10times unde se afla cea mai buna solutie din 10 rulari si se calculeaza media costului si se salveaza valorile intr-un fisier

Steepest Ascent Hill-Climbing

```
def vecini2(sol, index):
    vecini = sol
    pozSchimbare = 0
    if vecini[index] == 0:
        vecini[index] = 1
        pozSchimbare = index
    else:
        vecini[index] = 0
        pozSchimbare = index
        print("pozSchimbare", pozSchimbare)
    return vecini, pozSchimbare

def generareTotiVec(sol):
    n = len(sol)
    index = 0
    listaVecini = []
    pozSchimbare2 = 0
    for f in range(n):
        if sol[f] == 0:
            index = f
            break

    vecini, pozSchimbare = vecini2(sol,index)
    i = pozSchimbare + 1
    vecini[pozSchimbare] = 0
    print("pozSchimbare", pozSchimbare)
    for i in range(n):
        if vecini[i] == 0:
            vecini[i] = 1
            pozSchimbare2 = i
```

```
            vec = copy.copy(vecini)
            vec2 = copy.copy(vec)
            listaVecini.append(vec2)
        else:
            vecini[i] = 0
            pozSchimbare2 = i
            vec = copy.copy(vecini)
            vec2 = copy.copy(vec)
            listaVecini.append(vec2)

        vecini[pozSchimbare2] = 0
        print("lista vecini din fct de vecini", listaVecini)

    return listaVecini
```

Functiile de mai sus calculeaza toti vecinii unei solutii valide

```
def solInitValida(obiecte, greutateTotala):
    solInit = generareSolAleatoare(obiecte)
    greutate, valoare = fctFitness(obiecte, solInit, greutateTotala)
    if greutate:
        print("greutate", greutate, "valoare", valoare)
        return solInit, valoare
    else:
        solInitValida(obiecte, greutateTotala)
```

Functia solInitValida este o functie recursiva care genereaza o solutie valida

```
def sahcR(obiecte, greutateTotala, solInit, best, k):
    print("solInit", solInit)
    print("Valoare", best)

    listaValori = []
    bestinit = best
    i = 0
    listaVecini = generareTotiVec(solInit)
    print("lista vecini genrare", listaVecini)
    listaVeciniValizi = []

    l = len(listaVecini)
    while i < k:
        for j in range(l):
            print("lista vecini", listaVecini[j])
            greutate, valoare1 = fctFitness(obiecte, listaVecini[j], greutateTotala)
            if greutate:
                listaValori.append(valoare1)
                listaVeciniValizi.append(listaVecini[j])
                i= i+1
        p = len(listaValori)
        for p in range(p):
            if listaValori[p] > best:
                best = listaValori[p]
```

```
            solInit = listaVeciniValizi[p]

    if best == bestinit:
        print("best oprit", best)
        return best
    else:
        sahcR(obiecte, greutateTotala, solInit, best, k)
return best
```

Functia de mai sus este o functie recursiva care urmareste principiul steepest ascent hill climbing, se genereaza o solutie aleatoare valida, se gasesc toti vecinii solutiei respective, se calculeza fitnessul vecinilor si se alege vecinul cu cel mai bun fitness se continua asa pana ce se afla cea mai buna solutie

Tabele de date

Random Search

Instanța problemei	k	Valoarea medie	Cea mai buna valoare	Nr executii	Timp mediu de executie
rucsac20.txt	10	539.4	651	10	4.880072593688965
	100	608.1	652	10	6.275818347930908
	300	693	637.6	10	9.950208425552185
	500	636.2	669	10	12.529924869537354
rucsac200.txt	10	129745.3	131670	10	7.219818353652954
	100	131613.7	132182	10	12.957956314086914
	300	132219.1	132900	10	31.739037752 15149

	500	132388.8	133016	10	49.071276187 89673

Steepest ascent hill climbing

Instanța problemei	k	Valoarea medie	Cea mai buna valoare	Nr executi	Timp mediu de executie
Rucsac20.txt	10	136.5	465	10	6.7832400798 79761
	100	130.9	409	10	14.098872423 171997
	300	148.9	589	10	28.269467830 65796
	500	152.4	624	10	41.783003568 64929
Rucsac200.tx t	10	15777.8	120734	10	1571.2594685 554504
	100	15893.5	121891	10	1494.7514536 380768
	300	16302.4	125980	10	1856.9951226 711273
	500	16744.5	130401	10	2701.3981130 12314

Primul lucru pe care il observam cand comparam cei doi algoritmi este ca pentru steepest ascent hill climbing timpul de rulare este unul mult mai mare ca si pentru Random search. Valorile medii sunt cu mult mai mari pentru Random search datorita faptului ca in Steepest Ascent hill climbing se iau toti vecinii solutiei, inclusiv cei cu putine obiecte in “rucsac” ceea ce duce la o scarede a valorii medi