Lab 2 AI documentatie

Problema rucsacului Tabu Search

Functiile de generare solutie de fitness si de evaluare sunt aceleasi functii folosite si in laboratorul 1 pentru cautarea aleatoare si steepest ascent hill climbing.

```
def memorieInitiala(obiecte):
    n = len(obiecte)
    memorie = []
    for i in range(n):
        memorie.append(0)
    print("memorie initiala", memorie)
    return memorie
```

Functia memorieInitiala initializeaza un vector de lungimea obiectelor din solutie “memoria” cu 0

```
def actualizeazaMemorie(memorie):
    for i in range(len(memorie)):
        if memorie[i] != 0:
            memorie[i] = memorie[i]-1
    return memorie
```

Aceasta functie actualizeaza memoria, daca aceasta este un nr tabu se scade tretat pana ce scade la 0 si este din nou disponibla

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

def bestVecin(obiecte, sol, greutateTotala, memorie):
    best = -1
    vecinBest = []
    pos = -1
    greutateVec = 0
    greutate, valoare = fctFitness(obiecte, sol, greutateTotala)
    if greutate:
        for i in range(len(sol)):
            if memorie[i] == 0:
                vecin = vecini(sol, i)
                greutateVec, valoareVec = fctFitness(obiecte, sol, greutateTotala)
                if greutateVec and valoareVec > best:
                    best = valoareVec
                    vecinBest = vecin
                    pos = i
    print("best", best, "vecin best", vecinBest, "pos", pos, "greutate", greutateVec)
```

```
    return best, vecinBest, pos
```

Cele doua functii genereaza cel mai bun vecin al solutiei initiale, functia se bazaza pe principiul tabu, daca memoria nu este egala cu 0 pe o anumita pozitie atunci se trece la urmatoarea

```
def tabuSearch(k, nrTabu, obiecte, greutateTotala, memorie):
    i = 0
    bestSol = []

    valoriBest = []
    rez = []
    best = 0
    worst = 10000000
    sol = genereazaSol(obiecte)
    greutateVec, valoareVec = fctFitness(obiecte, sol, greutateTotala)
    if greutateVec:
        while i < k:

            rez = bestVecin(obiecte, sol, greutateTotala, memorie)
            print("rez", rez)
            print("rez[0]", rez[0])
            actualizeazaMemorie(memorie)
            if rez[2] != -1:
                memorie[rez[2]] = nrTabu
            if rez[0] > best:
                best = rez[0]
                bestSol = rez[1]

            i = i +1
    with open('solutii.txt', 'a') as f:
        f.write(str(k))
        f.write(" ")
        f.write(str(best))
        f.write(" ")
        f.write(str(bestSol))
        f.write(" ")
        f.write(str(worst))
        f.write(" ")

    return best
```

Functia tabuSearch ia o solutie valida aleatoare, pe care o duce prin principiul algoritmului tabu. Memoria obliga algoritmul sa nu se intoarca la maximul local, astfel ajunge cat mai aproape de maximul global, acest algoritm este trecut prn functia run10 times care ruleaza de 10 ori algoritmul respectiv

Instanța problemei	k	Valoarea medie	Cea mai buna valoare	Tabu nr	Nr executii	Timp mediu de executie
rucsac20.txt	100	141.3	503	5	10	7.879764318466186 5
	300	194.6	581	5	10	9.098579168319702
	1000	270	588	5	10	9.922972917556763
	100	285	570	10	10	5.581806421279907
	300	345	657	10	10	9.147357702255249
	1000	167.8	615	10	10	{6.833710432052612
rucsac200.txt	100	39496.7	131735	5	10	6.392694711685181
	300	65681.9	132835	5	10	6.909295558929443
	1000	39340.9	131824	5	10	6.846782207489014
	100	52761.6	132219	10	10	8.499313592910767
	300	78620.9	132003	10	10	9.671289920806885
	1000	26338.2	132393	10	10	6.113893032073975

Rezultatele au o tendinta de crestere , desi cu cat creste numarul de iteratii poate sa creasca sau nu valoarea best, cand valoarea tabu creste avem parte de rezultate mai bune fata de rularile cu o valoare tabu mai mica.

Problma Comis voiajorului

```
def distantaOrase(x1, x2, y1, y2):
    dist = sqrt((x2 - x1)*(x2-x1)+(y2-y1)*(y2-y1))
    return dist
```

Acasta functie calculeaza distanta dintre 2 oarase

```
def fitness(orase, permutare):
    sum = 0
    listaDistante = []
   # permutare = list(np.random.permutation(len(orase)))
    print(permutare)
    for i in range(len(orase)-1):
        coord1 = orase[int(permutare[i])][1]
        coord2 = orase[int(permutare[i])][2]
        coord3 = orase[int(permutare[i+1])][1]
        coord4 = orase[int(permutare[i+1])][2]
```

```
        dist = distantaOrase(coord1, coord2, coord3, coord4)
        sum = sum + dist
        print("distanta dinte orasul", permutare[i],"si orasul",permutare[i+1], "este ",
dist )
        print("permutare2", permutare[i+1])
        print("coord3", coord3)
        print("coord4", coord4)
        print("permutare",permutare[i])
        print("coord1",coord1)
        print("coord2",coord2)
    coord1 = orase[int(permutare[0])][1]
    coord2 = orase[int(permutare[0])][2]
    dist = distantaOrase(coord1, coord2, coord3, coord4)
    sum = sum + dist
    print("distanta dinte orasul", permutare[0], "si orasul", permutare[len(orase)-1],
"este ", dist)

    return sum
```

functia calculeaza distanta totala dintre orase, adica drumul parcurs

```
def vecini(orase, permutare):
    n = len(orase)
    a = np.random.randint(0, n-1)
    b = np.random.randint(0, n-1)
    while a == b:
        a = np.random.randint(0, n - 1)
    for i in range(n-1):
        aux = permutare[a]
        permutare[a] = permutare[b]
        permutare[b] = aux
    return permutare
```

functia returneaza un vecin al permutarii date

```
def simulatedAnnealing(orase, iteratii):
    T = 100000
    alpha = 0.9999
    minT = 0.00001
    listaDistante = []
    p = 0
    c = list(np.random.permutation(len(orase)))
    sum = fitness(orase, c)

    while T > minT:

        while iteratii > 0:
            x = vecini(orase, c)
            delta = fitness(orase, x) - fitness(orase, c)
            if fitness(orase, c) < fitness(orase, x):
                c = x
            else:
                if np.random.randint(0,1) < math.exp(-delta/T):
```

```
                c = x
            iteratii = iteratii - 1
            listaDistante.append(fitness(orase, c))
            p = p+1
        T = alpha*T

    print("lista Distante",listaDistante, "cea mai buna ruta",fitness(orase, c))
    print("solutia",c)

    return c
```

Functia simulatedAnnealing determina cea mai buna solutie a algoritmului, se calculeaza avand in vedere temperaturile, maxima si minima si alpha. Se trece prin 2 structuri repetitive cat timp t este mai mare ca si t min si inca un while cu numarul de iteratii, unde se iau vecinii si se compara solutia cea mai buna cu solutia curenta, solutia cea mai buna luand rolul de solutie best pe viitor

Instanta problemei	Nr k	t	T min	alfa	Val medie	Val best	Nr executii	Timp executie
KroC100	100	0.00001	100000	0.9999	194867.66189047275	193561.52827596	10	{136.564089298 2483}
	300	0.00001	100000	0.9999	194008.14667173388	192951.49276824328	10	382.643 8672542 572
	500	0.00001	100000	0.9999	194260.26352420152	191926.1666265798	10	623.895 3146934 509}
	700	0.00001	100000	0.9999	194342.31077250073	191498.46251482685	10	991.393 6157226 562

Cu cat creste numarul de iteratii fitnessul e mai bun, timpul de rulare creste si el, cel mai bun rezultat il avem la 700 d iteratii.