**Explicații algoritmi și pseudo cod problema 1**

**Problema rucsacului**

* n obiecte, fiecare obiect are o valoare (v) și o greutate (w)
* Obiectiv: puneți în rucsac valoarea maximă fără a depăși greutatea maximă admisă W
* 𝑥𝑖=1 înseamnă obiectul i este pus în rucsac
* 𝑥𝑖=0 înseamnă obiectul i nu este pus în rucsac

****

**Pentru Tabu Search**

1. Se generează o soluție aleatorie și validă **c**;
2. Se inițializează **best** cu soluția generată și memoria tabu drept un șir de zerouri de lungime egală cu lungimea unei soluții;
3. Se determină cel mai bun vecin x, care nu este tabu, al soluției generate precedent;
4. **c** ia valoarea celui mai bun vecin;
5. Se face update la memoria tabu după cum urmează:  
   Dacă index = indexul elementului pe care s-a făcut bit flip, **memorie(index) = p**;  
   Altfel dacă memorie(index) > 0, **memorie(index) = memorie(index) – 1**;
6. Dacă **fitness(c)** **> fitness(best**), best devine c;
7. Se execută pașii de la 2 până la 6 de **k** ori;
8. Se returnează **best.**

**Parametri problemei sunt k(numărul de iterații) și p(perioada pentru care un element din memorie este tabu).**

func tabu\_search(k, p, obiecte, capacitate):  
 c = generareSoluțieValidă(lungime(obiecte), obiecte, capacitate)  
 memorie = [0] \* len(objects) # memoria tabu  
 best = c[:]  
 câtTimp k > 0:  
 x, index = bestNonTabuNeighbor(c, memorie, obiecte, capacitate)  
 c = x[:]  
 memorie = actualizareMemorie(memorie, index, p)  
 Dacă fitness(obiecte, c, capacitate) > fitness(obiecte, best, capacite):  
 best = c[:]  
 k -= 1  
  
 returnează best, fitness(obiecte, best, capacitate), get\_weight(obiecte, best)

**Tabele de date**

**Pentru Tabu Search(TS)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Instanța problemei | K | P | Valoare medie | Greutatea medie | Valoarea cea mai bună | Număr execuții | Timpul mediu de execuție |
| Rucsac-20.txt | 50 | 11 | 696.0 | 512.7 | 706 | 10 | 0.020657157897949217 |
| 100 | 11 | 624.9 | 511.6 | 716 | 0.03289594650268555 |
| 200 | 11 | 702.5 | 515.5 | 726 | 0.056236982345581055 |
| 1000 | 11 | 708.3 | 516.7 | 726 | 0.25512166023254396 |
| 20 | 706.8 | 516.9 | 726 | 0.187325119972229 |
| Rucsac-200.txt | 50 | 101 | 132097.1 | 112617.1 | 132728 | 2.2130138158798216 |
| 100 | 101 | 132007.1 | 112637.1 | 132643 | 3.6374948978424073 |
| 200 | 101 | 132412.7 | 112632.7 | 133138 | 7.024287486076355 |
| 1000 | 101 | 132569.5 | 112629.5 | 133147 | 30.512607312202455 |
| 200 | 132951.9 | 112631.9 | 133844 | 27.690643095970152 |
| Obiecte = [(5, 60), (60, 5), (80, 10), (60, 10), (60, 20)] Capacitate maxima = 65 | 50 | 3 | 260.0 | 45.0 | 260 | 0.0015588045120239259 |
| 100 | 3 | 260.0 | 45.0 | 260 | 0.0015169620513916016 |
| 200 | 3 | 260.0 | 45.0 | 260 | 0.004621720314025879 |
| 1000 | 3 | 260.0 | 45.0 | 260 | 0.021823358535766602 |
| 5 | 260.0 | 45.0 | 260 | 0.024944639205932616 |

**Observații**

*Următoarele observații sunt date în funcție de datele din tabelele de mai sus.*

Pentru un număr foarte mic date, 5 obiecte spre exemplu, avem în vedere următoarele observații:

* Valoarea medie o sa fie in general valoarea optima;
* Valoarea cea mai bună poate fi ușor identificată drept valoarea optimă;
* Nu există diferențe majore între RS, RHC și TS

Pentru un număr mediu de date, 20 obiecte spre exemplu(rucsac-20.txt), avem în vedere următoarele observații:

* Pentru acest număr de elemente valorile medii, indiferent de parametri primiți, sunt cu mult mai bune în cazul RS și RHC. Spre exemplu, cele mai bune valori obținute după RHC și RS sunt 678 și 698;

Pentru un număr mare de date, 200 obiecte spre exemplu(rucsac-200.txt), avem în vedere următoarele observații:

* Cele mai bune valori obținute din TS sunt mai mari decât cele obținute din RS și RHC, dar nu pentru toate cazurile. Cea mai bună valoare obținută în urma TS este 133844, care este mult mai mare decât valoarea maximă obținută după RHC sau RS, acestea sunt 133342 și respectiv 133164 ;
* Valorile au devenit mai bune, cu cât k a devenit mai mare decât p;
* Valorile obținute sunt cu mult mai consistente în TS.

Observații generale:

* Timpul de execuție pentru TS e în general mult mai lung decât pentru algoritmi utilizați precedent pentru rezolvarea problemei;
* Valoarea medie nu se îmbunătățește pentru un număr de iterații tabu mai mare;
* Valorile primite drept output devin mai consistente pentru un număr mai mare de iterații.

**Concluzie**

În concluzie, se poate observa că TS dă rezultate cu mult mai bune indiferent de parametri(probabil cu câteva excepții) sau de cazul pe care este utilizat, dar va avea un timp de execuție mai mare.

**Explicații algoritmi și pseudo cod problema 2**

**TSP**

* n orașe reprezentate prin coordonatele lor;
* Există un drum între fiecare două orașe;
* Distanța dintre orașe este distanța euclidiană;
* Se poate trece o singură dată printr-un oraș;
* Trebuie să se ajungă înapoi la punctul de start;
* Soluțiile sunt reprezentate drept permutări de n elemente;
* Valoarea unei soluții este dată de către distanța parcursă.
* Aceasta este o problemă de minimizare

**Pentru Simulated Annealing**

1. Se generează o soluție aleatorie și validă c;
2. Se inițializează T, tMin și alpha(între 0 și 1);
3. Se generează un vecin x pentru c
4. delta = fitness(x) - fitness(c);
5. Dacă delta < 0, c = x;  
   Altfel dacă random[0, 1) < , c = x;
6. Se execută pașii de la 3 la 5 de k ori;
7. T = T \* alpha;
8. Se execută pașii de la 3 la 7 cât timp T > tMin;
9. Se returnează c.

**Parametri problemei sunt k(numărul de iterații), T(temperatura inițială), tMin(temperatura minimă) și alpha(valoarea de răcire).**

func simulated\_annealing(T, alpha, tMin, max, dimensiune):  
 c = soluție(dimensiune)  
   
 câtTimp t > tMin:  
 k = 0  
 câtTimp k < max:  
 x = vecin(c)  
 delta = fitness(x, dm) - fitness(c, dm)  
 dacă delta < 0:  
 c = x[:]  
   
 altfel dacă random[0, 1) < e^((-delta)/T)):  
 c = x[:]  
   
 k += 1  
 T = alpha \* T  
 returnează c

**Tabele de date**

**Pentru Simulated Annealing**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instanța problemei | T | alpha | tMin | k | Distanța medie | Distața cea mai bună | Număr execuții | Timpul mediu de execuție |
| kroC100.tsp | 100 | 0.9999 | 0.000001 | 10 | 21463.0 | 21246 | 5 | 339.24531383514403 |
| 700 | 0.99 | 0.001 | 5 | 27726.6 | 26488 | 1.281034564971924 |
| 1000 | 0.6 | 0.000001 | 10 | 86973.2 | 82528 | 0.09256091117858886 |
| 1000 | 0.9999 | 0.000001 | 10 | 21206.2 | 20749 | 407.41435146331787 |
| 10000 | 0.9999 | 0.00001 | 3 | 21474.0 | 21037 | 120.17254495620728 |

**Observații**

*Observațiile următoare sunt făcute în comparație cu valoarea optimă a problemei, care are o distanță de 20749.*

* Dându-se destule încercări pe unele valori se poate obține chiar valoarea optimă(dar asta e în principal din pur noroc). Cazul despre care este vorba e:  
  Temperatura: 1000

Valoarea de răcire: 0.9999

Valoarea minima pentru temperatura: 0.0000001

Număr iterații: 10

Număr execuții: 5

* Valorile devin mai bune cu cât alpha, k, T sunt mai mari și tMin este mai mic. Numărul de iterații ne dă șansa să alegem unele opțiuni incredibile pentru un T destul de mare ;
* Un T mai mare dă posibilitatea de a da o șansă mai mare unei soluții mai slabe(adică cu fitness mai mare)
* Valoarea lui T ajunge să nu mai conteze pentru valori suficient de mari pentru k și alpha, și o valoare suficient de mică pentru tMin. Același lucru este valabil și pentru valori în sensul opus pentru parametri;
* Din reprezentarea grafică pentru soluțiile obținute se poate observa minimizarea soluției obținute.

***T = 100***

***alpha = 0.9999***

***tMin = 0.0000001***

***k = 10***

T = 100
alpha = 0.9999
tMin = 0.0000001
k = 10
5 execuții

**Alte cazuri**

Chart

Description automatically generatedChart

Description automatically generated

**Concluzie**

Simulate Annealing este oferă unele rezultate incredibile depinzând de parametri folosiți în cadrul problemei. Cu toate acestea algoritmul este foarte costisitor din punct de vedere al timpului de execuție, mai ales pentru unele valori pentru parametri.