**Documentatie Laborator 2**

Cosntantinescu Ana-Gabriela, 331

**Tabu Search – Problema Rucsacului**

Pentru anumite functii, am refolosit codul facut pentru Laboratorul 1:

* citirea din fisier
* generarea de solutii random
* calcularea fitness

**Generarea tuturor vecinilor non-tabu ai unei solutii**

* se parcurg toti vecinii unei solutii date
* daca vecinul este tabu, atunci acesta se ignora (nu este adaugat in lista de vecini ce va fi returnata)
* se returneaza lista de vecini

def get\_neighbors\_non\_tabu(*n*, *objects*, *max\_sum*, *tabu\_list*, *sol*)  
 *neighbors* = []  
 (1..*n*).each do |*i*|  
 if *tabu\_list*[*i*] == 0  
 *neighbor* = []  
 *sol*.each{|*e*| *neighbor* << *e*.dup}  
 *neighbor*[*i*] = 1 - *neighbor*[*i*]  
 *neighbors*.push(*neighbor*) if is\_solution(*n*, *neighbor*, *objects*, *max\_sum*)  
 end  
 end  
 *return\_solution* = []  
 *neighbors*.each{|*e*| *return\_solution* << *e*}  
 *return\_solution*end

**Generarea celei mai bune solutii non-tabu**

* se parcurge lista tuturor vecinilor non-tabu
* se returneaza cel mai bun vecin dintre ei (cu cel mai mare fitness)

def get\_best\_neighbor\_non\_tabu(*n*, *objects*, *max\_sum*, *tabu\_list*, *sol*)  
 *neighbors* = get\_neighbors\_non\_tabu(*n*, *objects*, *max\_sum*, *tabu\_list*, *sol*)  
 *best\_sol* = []  
 *best\_fit* = -1  
 *best\_poz* = 1  
 *poz* = 1  
 *neighbors*.each do |*curr\_sol*|  
 *curr\_fit* = eval(*n*, *curr\_sol*, *objects*)  
 if *curr\_fit* > *best\_fit  
 best\_sol* = []  
 *curr\_sol*.each { |*e*| *best\_sol* << *e*.dup}  
 *best\_fit* = *curr\_fit  
 best\_poz* = *poz* end  
 *poz* += 1  
 end  
 *return\_solution* = []  
 *best\_sol*.each{|*e*| *return\_solution* << *e*.dup}  
 [*return\_solution*, *best\_fit*, *best\_poz*]  
end

**Actualizarea memoriei**

* functia primeste ca parametru lista tabu actuala si o pozitie, care reprezinta pozitia bitului care s-a schimbat pentru a genera noul vecin
* se parcurge lista tabu
* daca pozitia din lista este egala cu pozitia din parametru, atunci tabu[pozitie] primeste valoarea corespunzatoare memoriei scurte
* din orice valoare din lista diferita de 0 se scade 1

def update\_memory(*tabu\_list*, *poz*, *n*, *memory*)  
 *ret\_list* = []  
 *tabu\_list*.each { |*e*| *ret\_list* << *e* }  
 (1..*n*).each do |*i*|  
 if *i* == *poz  
 ret\_list*[*i*] = *memory* elsif *ret\_list*[*i*] != 0  
 *ret\_list*[*i*] += -1  
 end  
 end  
 *ret\_list*end

**Tabu Search**

* generez solutia greedy (sau o solutie random) – care devine solutia curenta
* generez cel mai bun vecin non-tabu
* daca vecinul are fitnessul mai bun decat solutia curenta, solutia curenta primeste valoarea vecinului
* repet pasii 2 si 3 de k ori

def tabu\_search(*n*, *k*, *objects*, *max\_sum*, *memory*)  
 *best\_sol* = greedy(*n*, *objects*, *max\_sum*)[1]  
 *curr\_sol* = []  
 *best\_sol*.each{|*e*| *curr\_sol* << *e*}  
 *best\_fit* = eval(*n*, *best\_sol*, *objects*)  
 *tabu\_list* = []  
 (1..*n*).each do |*i*|  
 *tabu\_list*[*i*] = 0  
 end  
 *i* = 0  
 while *i* < *k  
 response* = get\_best\_neighbor\_non\_tabu(*n*, *objects*, *max\_sum*, *tabu\_list*, *curr\_sol*)  
 *tabu\_list* = update\_memory(*tabu\_list*, *response*[2], *n*, *memory*).dup  
 *curr\_sol* = *response*[0].dup  
 if *response*[1] > *best\_fit  
 best\_sol* = *curr\_sol*.dup  
 *best\_fit* = *response*[1]  
 end  
 *i* += 1  
 end  
 [*best\_fit*, *best\_sol*]  
end

**Algoritm greedy**

def greedy(*n*, *objects*, *max\_sum*)  
 *solution* = []  
 (1..*n*).each do |*i*|  
 *solution*[*i*] = 1  
 end  
 *obj* = []  
 *objects*.each{|*e*| *obj* << *e*.dup}  
 *obj\_cpy* = []  
 *obj*[0] = {'value' => 10000, 'weight' => 10000}  
 *obj*.each { |*e*| *obj\_cpy* << *e*.dup}  
 *obj\_cpy*.sort\_by! { |*e*| *e*['value']}  
 *i* = 0  
 while is\_solution(*n*, *solution*, *objects*, *max\_sum*) == false  
 *a* = *obj*.index { |*e*| *e*['value'] == *obj\_cpy*[*i*]['value'] }  
 *solution*[*a*] = 0  
 *i* += 1  
 end  
 [eval(*n*, *solution*, *objects*), *solution*.dup]  
end

**Observatii - Solutie initiala random/greedy:**

Initial, am pornit cu o solutie initiala random, nu cu solutia greedy. Am luat decizia de a porni cu greedy deoarece solutiile aveau o valoarea mica, chiar si pentru rucsacul de 20. De exemplu:

*rucsac 20, k=1000, memorie=4*

|  |  |
| --- | --- |
| Worst | 547 |
| Best | 711 |
| Average | 629 |
| Runtime | 0.802452 |

*rucsac 20, k=10000, memorie=7*

|  |  |
| --- | --- |
| Worst | 497 |
| Best | 711 |
| Average | 615.6 |
| Runtime | 7.678879 |

Pornind de la o solutie greedy, algoritmul a devenit determinist. Astfel, pentru aceeasi parametri de rulare, nu am mai rulat de 10 ori, ci doar o singura data.

La rucsacul de 200, am rulat atat cu solutie greedy, cat si cu solutie random. Obtinand rezultate mai bune cu solutie random.

**Rezultate:**

*Rucsac 20*

Pornind de la solutia greedy, am obtinut rezultate bune. Am pastrat mereu memoria 3 (mai mult nu a fost necesar), astfel ca singurul parametru pe care l-am schimbat a fost k.

|  |  |
| --- | --- |
| k | Fitness |
| 1 | 671 |
| 2 | 699 |
| 3 | 726 |
| 5 | 726 |
| 10 | 726 |

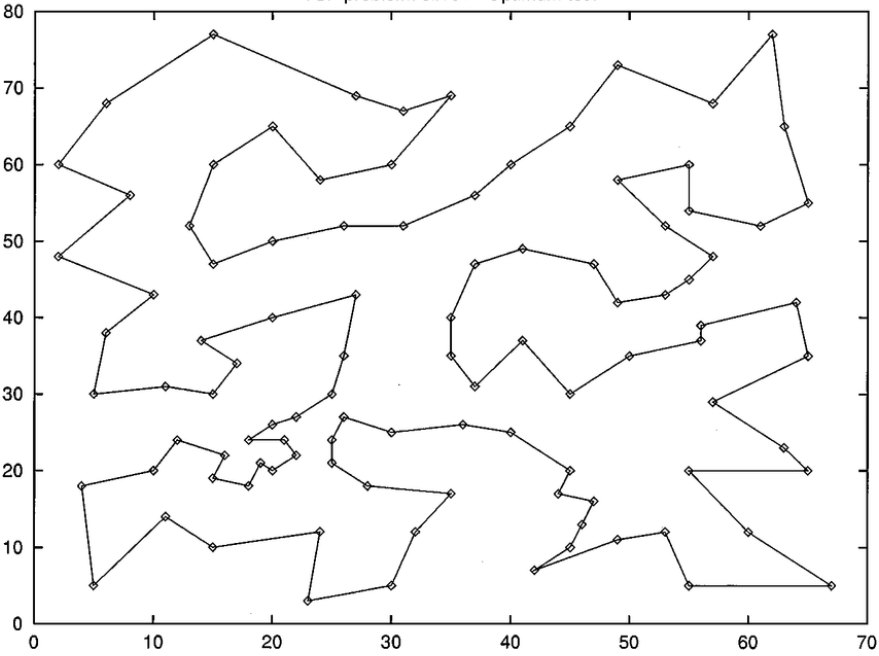
*Rucsac 200*

|  |  |  |
| --- | --- | --- |
| k | Memorie | Fitness |
| 1 | 3 | 129493 |
| 2 | 3 | 129632 |
| 3 | 3 | 129632 |
| 8 | 7 | 129632 |
| 10 | 3 | 129632 |
| 50 | 3 | 129632 |
| 100 | 3 | 129632 |
| 1000 | 5 | 129632 |
| 1000 | 7 | 129632 |
| 5000 | 12 | 129632 |
| 10000 | 7 | 129632 |
| 10000 | 10 | 129632 |
| 10000 | 20 | 129632 |
| 10000 | 50 | 129632 |
| 50000 | 3 | 129632 |
| 50000 | 8 | 129632 |
| 100000 | 30 | 129632 |

Din moment ce solutia greedy mi-a dat aceste rezultate cu fitness slab, am decis sa modific codul si sa pornesc de la o solutie generata random. Rezultatele avute au fost mai bune:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| k | Memorie | Best | Worst | Average | Runtime(s) |
| 1000 | 5 | 132502 | 131448 | 131952.7 | 161 |
| 1000 | 7 | 132212 | 131147 | 131672.1 | 69 |
| 5000 | 5 | 132622 | 130824 | 131590.5 | 352 |
| 10000 | 5 | 132411 | 130621 | 131640.6 | 704 |
| 10000 | 7 | 132434 | 130735 | 131500.5 | 1761 |

**Simulated Annealing – TSP**



**Citirea din fisier**

def read\_config\_file(*file\_path*)  
 *lines* = []  
 File.open(*file\_path*, 'r') do |*file*|  
 *lines* = *file*.readlines  
 end  
 *objects* = []  
 *n* = *lines*[0].strip.to\_i  
 (1..*n*).each do |*i*|  
 *line* = *lines*[*i*].strip  
 *parts* = *line*.split(' ')  
 *objects*[*i*-1] = {}  
 *objects*[*i*-1]['poz'] = *parts*[0].to\_i  
 *objects*[*i*-1]['x'] = *parts*[1].to\_i  
 *objects*[*i*-1]['y'] = *parts*[2].to\_i  
 end  
 [*n*, *objects*]  
end

**Calcularea distantei dintre 2 orase**

def distance(*a*, *b*)  
 *dif1* = (*a*['x'] - *b*['x']) \* (*a*['x'] - *b*['x'])  
 *dif2* = (*a*['y'] - *b*['y']) \* (*a*['y'] - *b*['y'])  
 Math.sqrt(*dif1* + *dif2*).round  
end

**Calcularea fitnessului unei solutii**

def fitness(*n*, *objects*, *solution*)  
 *fit* = 0  
 (0..*n*-2).each do |*i*|  
 *fit* += distance(*objects*[*solution*[*i*] - 1], *objects*[*solution*[*i*+1] - 1])  
 end  
 *fit* + distance(*objects*[*solution*[*n*-1] - 1], *objects*[*solution*[0] - 1])  
end

**Gasirea celui mai apropiat oras dintr-o lista data de orase**

def get\_closest\_city(*poz*, *cities*, *objects*)  
 *closest\_city\_distance* = 999999  
 *closest\_city\_poz* = -1  
 *home* = *objects*[*poz*]  
 *cities*.each do |*city*|  
 *dist* = distance(*home*, *city*)  
 if *dist* < *closest\_city\_distance  
 closest\_city\_distance* = *dist  
 closest\_city\_poz* = *city*['poz'] - 1  
 end  
 end  
 *closest\_city\_poz*end

**Generarea unei solutii greedy**

* functia aceasta este nedeterminista

def greedy(*n*, *objects*)  
 *start\_poz* = rand(*n*-1)  
 *obj* = []  
 *objects*.each{|*e*| *obj* << *e*.dup}  
 *obj*.delete\_at(*start\_poz*)  
 *solution* = []  
 *curr\_object\_poz* = *objects*[*start\_poz*]['poz'] - 1  
 *solution* << *objects*[*start\_poz*]['poz']  
 until *obj*.empty?  
 *closest\_city\_poz* = get\_closest\_city(*curr\_object\_poz*, *obj*, *objects*)  
 *solution* << *closest\_city\_poz* + 1  
 *obj*.each do |*city*|  
 if *city*['poz'] == *closest\_city\_poz* + 1  
 *obj*.delete(*city*)  
 break  
 end  
 end  
 *curr\_object\_poz* = *closest\_city\_poz* end  
 *solution*end

**Generarea vecinilor unei solutii date (2-swap si 2-opt)**

* generez 2 pozitii random
* 2-swap: inversez localitatile de pe pozitiile generate (fara a inversa localitatile dintre ele)
* 2-opt: inversez drumul de la localitatile corespunzatoare celor 2 pozitii generate

def get\_neighbor\_2\_swap(*n*, *solution*)  
 *poz1* = rand(*n*-1)  
 *poz2* = rand(*n*-1)  
 *cpy* = *solution*.dup  
 *aux* = *cpy*[*poz1*]  
 *cpy*[*poz1*] = *cpy*[*poz2*]  
 *cpy*[*poz2*] = *aux  
 cpy*end  
  
def get\_neighbor\_2\_opt(*n*, *solution*)  
 *poz1* = rand(*n*-1)  
 *poz2* = rand(*n*-1)  
 if *poz2* < *poz1  
 aux* = *poz1  
 poz1* = *poz2  
 poz2* = *aux* end  
 *cpy* = *solution*.dup  
 while *poz1* < *poz2  
 aux* = *cpy*[*poz1*]  
 *cpy*[*poz1*] = *cpy*[*poz2*]  
 *cpy*[*poz2*] = *aux  
 poz1* += 1  
 *poz2* -= 1  
 end  
 *cpy*end

**Simulated Annealing**

* parametrii: T, min\_t, alpha

c = random\_solution

best = c

while (T > min\_t)

repeat k times

x = neighbor

update\_best

delta = eval(x) – eval(c)

if (delta<0)

c = x

else

if (rand < Math.ecp(-delta/t)

c = x

T = alpha\*T

return best

def simulated\_annealing(*n*, *objects*, *t*, *min\_t*, *k*, *alpha*)  
 *curr\_sol* = greedy(*n*, *objects*)  
 *best\_sol* = *curr\_sol  
 best\_fit* = fitness(*n*, *objects*, *best\_sol*)  
 while *t* > *min\_t* p *t* p *best\_fit  
 i* = 0  
 while *i* < *k  
 neighbor* = get\_neighbor\_2\_opt(*n*, *curr\_sol*)  
 *neighbor\_fit* = fitness(*n*, *objects*, *neighbor*)  
 if *neighbor\_fit* < *best\_fit  
 best\_sol* = *neighbor*.dup  
 *best\_fit* = *neighbor\_fit* end  
 *delta* = *neighbor\_fit* - fitness(*n*, *objects*, *curr\_sol*)  
 if *delta* < 0  
 *curr\_sol* = *neighbor*.dup  
 elsif rand < Math.exp(-*delta*/*t*)  
 *curr\_sol* = *neighbor*.dup  
 end  
 *i* += 1  
 end  
 *t* = *t* \* *alpha* end  
 [*best\_fit*, *best\_sol*.dup]  
end

**Rezultate:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | K | T | T\_min | Alpha | Best | Worst | Average | Runtime(s) |
| swap | 500 | 100 | 0.001 | 0.99 | 748 | 782 | 761.5 | 592 |
| opt | 500 | 10 | 0.001 | 0.999 | 632 | 650 | 641.8 | 4144 |
| opt | 500 | 10 | 0.1 | 0.999 | 632 | 655 | 640.3 | 2224 |
| opt | 500 | 100 | 0.001 | 0.99 | 649 | 671 | 660.4 | 462 |
| opt | 500 | 100 | 0.00001 | 0.999 | 633 | 650 | 643.6 | 7647 |
| opt | 500 | 1000 | 0.00001 | 0.99 | 653 | 665 | 659.7 | 1096 |
| opt | 500 | 10000 | 0.00001 | 0.99 | 653 | 671 | 662.8 | 1236 |
| opt | 1000 | 10 | 0.1 | 0.999 | 631 | 643 | 634.6 | 4575 |

**Observatii:**

* am decis sa folosesc 2-opt, nu 2-swap
* cele mai bune schimbari le-am observant cand T era intre 3 si 0.5. De aceea, la ultima rulare pe care am facut-o, am crescut k, dar am scazut T la 10 si min\_t la 0.1. Astfel, am obtinut cele mai bune rezultate.