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

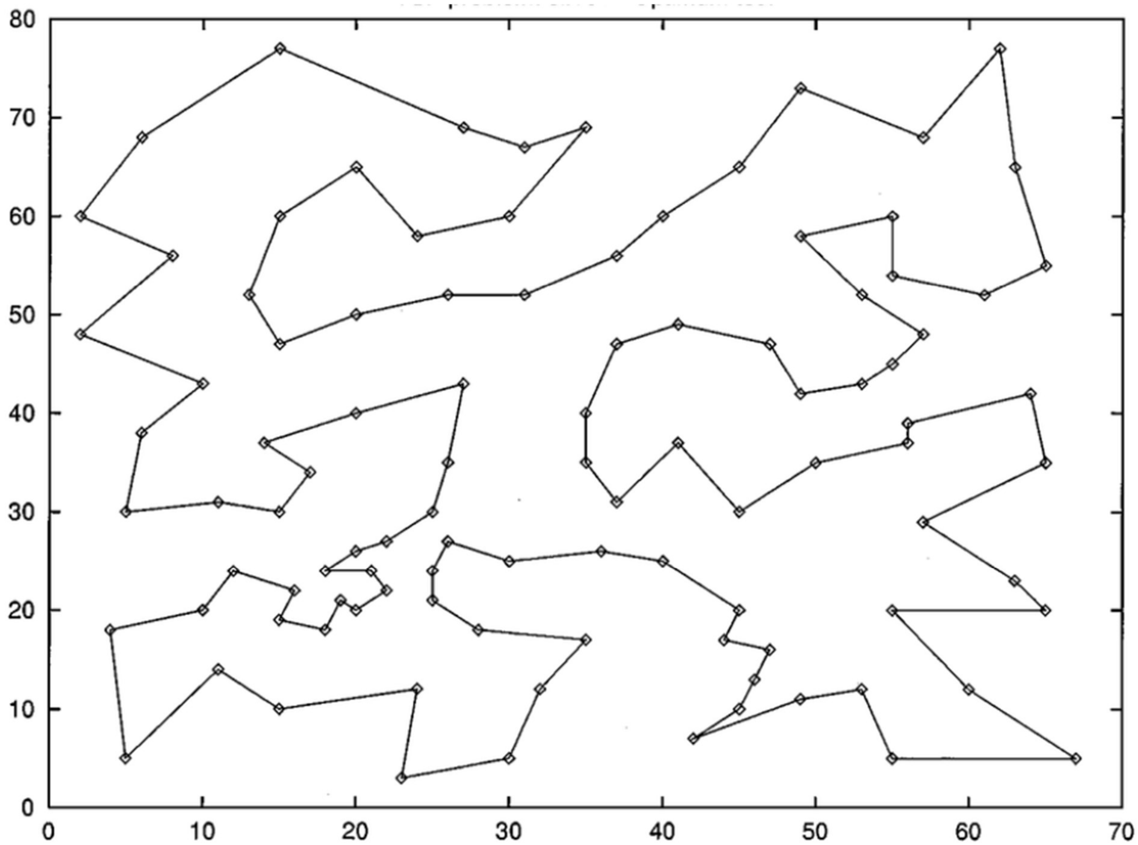
Rucsac 200

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

```ruby
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

```ruby
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

```ruby
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
```

```ruby
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.