**Documentatie Laborator 1**

Constantinescu Ana-Gabriela, 331

**Problema Rucsacului**

Se dau n obiecte, fiecare avand o valoare (value – v) si o greutate (weight – w). Puneti in rucsac valoarea maxima fara a depasi greutatea maxima admisa (W).

Fie x – o solutie gasita. Solutia este reprezentata ca un array de lungime n.

* xi = 0 daca obiectul **i** nu se afla in ghiozdan
* xi = 1 daca obiectul **i** se afla in ghiozdan

Formula pentru a calcula fitnessul unei solutii (valoarea totala a obiectelor care au incaput in ghiozdan):

Pentru a verifica ca un array **x** este solutie, atunci acesta trebuie sa respecte:

Pentru a rezolva aceasta problema, voi implementa 2 metode:

* Cautare aleatoare (Random Search)
* Next Ascent Hill Climbing

Limbajul ales pentru a implementa solutiile este Ruby, iar rezultatele rularilor vor fi salvate in fisiere csv.

**Citirea din fisier**

Pentru a rezolva aceasta problema se vor considera 2 seturi de date de test:

* 20 de obiecte si un rucsac cu W = 524
* 200 de obiecte si un rucsac cu W = 112648

Aceste date de test se afla in 2 fisiere separate: *20\_input\_file.txt* si *200\_input\_file.txt*.

Pentru a citi si retine in memorie aceste date, am creat o metoda **read\_config\_file** care:

* are ca parametru un string, numele fisierului in care se afla datele de test
* returneaza un array de marime 3, care are:
  + pe pozitia 0: un intreg care reprezinta valoarea lui n
  + pe pozitia 1: un intreg care reprezinta greutatea maxima admisa (W)
  + pe pozitia 2: un array de dimensiune n, care continue valoarea si greutatea fiecarui obiect. Este un array de dictionare, fiecare dictionar avand 2 chei, value si weight

**Generarea unei solutii aleatoare. Fitnessul unei solutii**

Pentru a genera solutii aleatoare am create 3 metode:

1. **generate\_random\_array**: genereaza un array random, de dimensiune n, cu valori de 0 sau 1

*# n - integer, total number of objects  
# @return - array*def generate\_random\_array(*n*)  
 *rand\_solution* = []  
 (1..*n*).each do |*i*|  
 *rand\_solution*[*i*] = rand(0..1)  
 end  
 *rand\_solution*end

1. **is\_solution**: verifica daca un array este solutie (daca suma valorilor obiectelor puse in ghiozdan este mai mica sau egala decat W)

*# n - integer, total number of objects  
# rand\_arr - array, a possible solution  
# objects - array of hashes, contains value and weight for each object  
# max\_sum - integer, maximum weight that fits in the backpack  
# @return - boolean*def is\_solution(*n*, *rand\_arr*, *objects*, *max\_sum*)  
 *sum* = 0  
 (1..*n*).each do |*i*|  
 *sum* += *objects*[*i*]['weight']\**rand\_arr*[*i*]  
 end  
 return true if *sum* <= *max\_sum* false  
end

1. **generate\_random\_solution**: genereaza o solutie random

*# n - integer, total number of objects  
# objects - array of hashes, contains value and weight for each object  
# max\_sum - integer, maximum weight that fits in the backpack  
# @return - array*def generate\_random\_solution(*n*, *objects*, *max\_sum*)  
 *rand\_arr* = generate\_random\_array(*n*)  
 while is\_solution(*n*, *rand\_arr*, *objects*, *max\_sum*) == false  
 *rand\_arr* = generate\_random\_array(*n*)  
 end  
 *rand\_arr*end

Pentru a verifica care este fitnessul unei solutii, am create o metoda **eval**:

*# n - integer, total number of objects  
# objects - array of hashes, contains value and weight for each object  
# solution - array, a solution  
# @return - integer*def eval(*n*, *solution*, *objects*)  
 *sum* = 0  
 (1..*n*).each do |*i*|  
 *sum* += *objects*[*i*]['value']\**solution*[*i*]  
 end  
 *sum*end

**Cautare aleatoare**

Am creat o metoda **generate\_best\_solution**, unde:

1. Se genereaza o solutie aleatoare **best\_sol**
2. Se genereaza o noua solutie aleatoare
3. Daca solutia generata la pasul 2 are un fitness mai bun decat **best\_sol**, in **best\_sol** se retine noua solutie
4. Se repeat pasii 2 si 3 de **k** ori, iar apoi se returneaza **best\_sol**

*# n - integer, total number of objects  
# k - integer, the number of randomly generated solutions  
# objects - array of hashes, contains value and weight for each object  
# max\_sum - integer, maximum weight that fits in the backpack  
# @return - array*def generate\_best\_solution(*n*, *objects*, *k*, *max\_sum*)  
 *i* = 1  
 *best\_fitness* = 0  
 *best\_sol* = []  
 while *i* <= *k  
 rand\_sol* = generate\_random\_solution(*n*, *objects*, *max\_sum*)  
 *fitness* = eval(*n*, *rand\_sol*, *objects*)  
 if *fitness* > *best\_fitness  
 best\_fitness* = *fitness  
 best\_sol* = *rand\_sol* end  
 *i* += 1  
 end  
 [*best\_fitness*, *best\_sol*]  
end

**Analiza solutiilor obtinute**

Pentru a analiza solutia implementata, am creat o alta metoda ajutatoare **write\_data**:

*# n - integer, total number of objects  
# k - integer, the number of randomly generated solutions  
# objects - array of hashes, contains value and weight for each object  
# max\_sum - integer, maximum weight that fits in the backpack  
# repeat - integer, number of final solutions*def write\_data(*n*, *objects*, *k*, *max\_sum*, *repeat*: 10)  
 *i* = 1  
 *all\_sol* = []  
 *all\_fit* = []  
 *t0* = Time.now  
 while *i* <= *repeat  
 sol* = generate\_best\_solution(*n*, *objects*, *k*, *max\_sum*)  
 *all\_fit*.push(*sol*[0])  
 *all\_sol*.push(*sol*)  
 *i* += 1  
 end  
 *t1* = Time.now - *t0  
 worst* = *all\_fit*.min  
 *best* = *all\_fit*.max  
 *avg* = *all\_fit*.sum(0.0)/*all\_fit*.size  
 *all\_sol*.push(['Worst', *worst*])  
 *all\_sol*.push(['Best', *best*])  
 *all\_sol*.push(['Average', *avg*])  
 *all\_sol*.push(['Runtime', *t1*])  
 File.write("rs\_#{*n*}\_solutions\_k\_#{*k*}.csv", *all\_sol*.map(&:to\_csv).join)  
end

Aceasta metoda ruleaza de un numar de ori (numar stabilit prin parametrul **repeat** – care are implicit valoarea 10) metoda **generate\_best\_solution**. Apoi, salveaza intr-un fisier csv:

* fitnessul fiecarei solutii gasite
* solutia (reprezentata de array)
* cea mai buna solutie
* cea mai proasta solutie
* media solutiilor
* durata (in secunde) de generare a celor 10 solutii

Am rulat programul, atat pentru rucsacul de 20, cat si cel de 200, pentru 5 valori diferite ale lui k: 100, 1000, 10000, 100000 si 1000000. Acestea sunt rezultatele:

**Cautare aleatoare – n = 20:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** | **Runtime (s)** |
| **100** | 670 | 560 | 626.8 | 0.0117173 |
| **1000** | 689 | 630 | 669.3 | 0.1184913 |
| **10000** | 716 | 677 | 690.2 | 1.2331192 |
| **100000** | 726 | 702 | 713.7 | 11.4502773 |
| **1000000** | 726 | 718 | 724.4 | 116.194905 |

**Cautare aleatoare – n = 200:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** | **Runtime (s)** |
| **100** | 132439 | 131325 | 131831.8 | 0.111051 |
| **1000** | 133311 | 132399 | 132703.8 | 1.2243612 |
| **10000** | 133555 | 132735 | 133150.1 | 10.8784278 |
| **100000** | 133906 | 133290 | 133530.8 | 109.1494106 |
| **1000000** | 134179 | 133590 | 133782.9 | 1123.5718215 |

**Observatii:**

* pentru rucsacul de 20, se poate spune (aproape) cu certitudine ca cea mai buna solutie are fitnessul 726. Pentru k = 1000000, solutia cu fitness 726 s-a obtinut de 8 ori (din 10 iteratii)
* pentru rucsacul de 200, nu se pot face astfel de afirmatii
* cresterea lui k este proportionala cu cresterea fitnessului solutiilor obtinute (best, worst si average)
* metoda este foarte rapida. Pentru n = 20 si k = 1000000, solutiile au fost generate in sub 2 minute. Pentru n = 200 si k = 1000000, solutiile au fost generate in aproximativ 19 minute

**Next Ascent Hill Climbing**

Am implementat o metoda **generate\_best\_solution**, unde:

1. Se selecteaza un punct aleator **c** in spatiul de cautare.
2. Se considera pe rand vecinii **x** ai punctului **c**. Daca fitnessul lui **x** este mai bun decat fitnessul lui **c**, atunci **c = x** si nu se mai evalueaza restul vecinilor lui **c**. Se continua pasul 2 cu noul **c** si se considera vecinii lui **c** mai departe (pornind din acelasi punct din vecinatate unde s-a ramas cu vechiul **c**).
3. Daca niciun vecin **x** al punctului **c** nu duce la o evaluare mai buna, se salveaza **c** si se continua procesul de la pasul 1.
4. Dupa un numar **k** de evaluari, se returneaza cel mai bun **c**.

*# n - integer, total number of objects  
# k - integer, the number of randomly generated solutions  
# objects - array of hashes, contains value and weight for each object  
# max\_sum - integer, maximum weight that fits in the backpack  
# @return - array*def generate\_best\_solution(*n*, *objects*, *k*, *max\_sum*)  
 *i* = 1  
 *best\_solution* = []  
 *best\_solution\_fitness* = 0  
 while *i* <= *k  
 c* = generate\_random\_solution(*n*, *objects*, *max\_sum*)  
 *c\_fitness* = eval(*n*, *c*, *objects*)  
 *new\_try* = *c  
 j* = 1  
 while *j* <= *n* if *new\_try*[*j*] == 0  
 *new\_try*[*j*] = 1  
 *new\_try\_fitness* = eval(*n*, *new\_try*, *objects*)  
 if is\_solution(*n*, *new\_try*, *objects*, *max\_sum*) && *c\_fitness* < *new\_try\_fitness  
 c* = *new\_try  
 c\_fitness* = *new\_try\_fitness* else  
 *new\_try*[*j*] = 0  
 end  
 end  
 *j* += 1  
 end  
 if *c\_fitness* > *best\_solution\_fitness  
 best\_solution* = *c  
 best\_solution\_fitness* = *c\_fitness* end  
 *i* += 1  
 end  
 [*best\_solution\_fitness*, *best\_solution*]  
end

**Mentiune:** nu am lucrat cu toti vecinii lui **c**, doar cu cei cu potential de a avea un fitness mai bun. Mai precis, nu am transformat niciodata din 1 in 0, doar din 0 in 1.

**Analiza solutiilor obtinute**

Am folosit o metoda extrem de asemanatoare cu cea mentionata anterior (**write\_data**) unde am modificat doar metoda apelata (sa nu fie cea pentru RS, ci cea pentru NAHC) si numele fisierelor generate. Am rulat cu aceleasi date pentru **n** si **k**.

**Next Ascent Hill Climbing – n = 20:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** | **Runtime (s)** |
| **100** | 710 | 661 | 682.4 | 0.0542107 |
| **1000** | 718 | 688 | 703.2 | 0.4888292 |
| **10000** | 726 | 710 | 719.4 | 4.9542706 |
| **100000** | 726 | 718 | 725.2 | 49.0265785 |
| **1000000** | 726 | 726 | 726 | 552.8758813 |

**Next Ascent Hill Climbing – n = 200:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Worst** | **Average** | **Runtime (s)** |
| **100** | 133845 | 133077 | 133415.4 | 3.1862294 |
| **1000** | 134566 | 133668 | 133932.7 | 30.2527796 |
| **10000** | 134595 | 134040 | 134275.3 | 969.8917159 |
| **100000** | 134619 | 134314 | 134456.5 | 3358.7441986 |
| **1000000** | 135022 | 134611 | 134748.6 | 67789.8296231 |

**Mentiune:** pentru n = 200 si k = 1000000, am aproximat ca rularea va dura cam 12 ore, asa ca am lasat programul sa ruleze peste noapte, dar laptopul mi s-a stins. Am aproximat ca a stat stins cam 6 ore si jumatate. Astfel ca timpul real de rulare ar fi de aproximativ 12 ore si jumatate, nu 19 ore (asa cum apare mai sus si in fisierul cu rezultatele).

**Observatii:**

* pentru n = 20, mai hotarat decat in cazul RS, se poate spune ca cea mai buna solutie are fitness 726 intrucat, pentru k = 1000000, fiecare solutie obtinuta a avut fitness 726
* pentru n = 200, inca nu se poate afirma daca cea mai buna solutie a fost gasita sau nu
* cresterea lui k este proportionala cu cresterea fitnessului solutiilor obtinute (best, worst si average)
* aceasta metoda este destul de lenta. Pentru k = 1000000:
  + n = 20, timpul de rulare a fost putin peste 9 minute
  + n = 200, timpul de rulare a fost de aproximativ 12 ore si jumatate

**Comparatie intre cele 2 metode**

* RS este mult mai rapida decat NAHC (mai ales pentru valori foarte mari ale lui n si k)
* pentru n = 20, ambele metode au gasit solutia cu fitness 726 ca fiind cea mai buna
* pentru n = 200, NAHC a obtinut rezultate mult mai bune decat RS (atat pentru valori mici ale lui k, cat si pentru valori mari). Cea mai buna solutie gasita de RS a fost 134179, iar gasita de NAHC a fost 135022 (ambele gasite pentru k = 1000000).
* tinand cont ca cea mai buna solutie gasita de RS a fost 134179, se poate observa cum NAHC a gasit solutii mai bune decat aceasta incepand cu k = 1000 (cea mai buna solutie gasita pentru k = 1000 fiind 134566)