Problema rucsacului - Metode clasice si hillclimbing

Prepared for: Camelia Chira, PhD
Prepared by: Razvan-Stefan Grecu
28 February 2016
Group: TI 30643

REFERAT

Descrierea problemei

Problema rucsacului este o problema de optimizare. Avem un rucsac si o colectie de obiecte ce dorim sa le punem in rucsac. Obiectele au urmatoarele proprietati: o greutate, pe care o vom nota cu w_i si o valoare, notata cu v_i. Rucsacul are o capacitate maxima W care nu trebuie depasita. Problema cere gasirea unei combinatii de obiecte care pot fi puse in rucsac, avand valoarea maxima intre combinatiile candidate.

Pentru rezolvarea acestei probleme, vom presupune ca atat valoarea, dar mai ales greutatea sunt numere intregi si ca nu vom putea pune parti din obiecte ci doar in intregime.

Prezentarea algoritmului

Avand specificatia problemei, cerinta a fost implementarea unor algoritmi clasici pentru rezolvarea acestei probleme dar si utilizarea metodei hillclimbing (in diferite variatii). In continuare, vor fi prezentati toti algoritmii utilizati pentru rezolvarea problemei. Va fi prezentat pseudocodul si descrieri ale unor metode folosite. Data fiind natura functionala a unei parti vaste a codului, pseudocodul poate sa nu fie foarte diferit de implementarea propriu-zisa.

Cautare exhaustiva

Primul algoritm prezentat este cel de cautare exhaustiva in spatiul problemei. Dupa nume, putem deduce principiul din spatele sau: cautam in mod enumerativ fiecare stare posibila - in functie de reprezentare! - si alegem solutia cu valoarea maximala ce respecta constrangerile problemei. Reprezentarea unei solutii se face sub forma unui sir binar; sirul reprezinta lista de obiecte ce vor fi puse in rucsac, 0 insemnand ca nu luam obiectul si 1 insemnand ca il luam. Pseudocod:

```
generateBinaryStrings(itemCount) .collect { Solution solution -> createModelFrom(solution) }
                                 .findAll { Model model -> isValid(model) }
                                 .sort {m1, m2 -> m2.value - m1.value }.head()
```

Imperativ, pseudocodul are forma urmatoare:

```
var solutionSpace = generateBinaryStrings(itemCount)
var validModels = []
for (var candidateSolution in solutionSpace) do
        var model = createModelFrom(candidateSolution)
        if (isValid(model)) do validModels.add(model)
return maxByValue(validModels)
```

Metoda ***generateBinaryStrings()*** genereaza toate starile posibile ale modelului in spatiul problemei. Implementarea este rudimentara, iterand pana la $2^{itemCount}$, generand fiecare sir binar posibil. Aceasta metoda nu se recomanda a se folosi pe instante de date avand mai mult de 25-30 de obiecte din cauza timpului mare de rulare.

Cautare greedy

In practica vom avea input-uri mai mari decat 25-30 de obiecte si avem nevoie de o metoda care sa se descurce intr-un timp, cel putin rezonabil, cu acele instante. Una din acele metode este greedy. Folosindu-ne de anumite cunostinte despre obiecte se pot construi metrici de utilizat pentru rezolvarea problemei. Metrica propusa este cea clasica, in care sortam obiectele descrescator dupa raportul dintre valoare si greutate. In urma sortarii, punem primele k obiecte in rucsac, pana cand elementul k+1 va invalida constrangerea problemei. In urma acestui proces, se introduce inca un pas, si anume, daca, in lista de obiecte, exista unul cu valoare mai mare decat suma valorilor celorlalte elemente, si acesta este valid, atunci golim ghiozdanul si adaugam acel element. Astfel ajungem la timp polinomial, de la timp exponential - cea mai complexa operatie aici este sortarea. Insa, problema este ca rezultatul poate sa nu fie cel optim - aceasta metoda are o rata de functionalitate de cel putin 50% - in 50% din cazuri, gaseste optimul. Pseudocod:

```
var knapsack = [], items.sort { greedyRatioMetric }
for ( item in items ) do
    if ( item.weight + knapsack.currentWeight <= knapsack.maxWeight ) do
        knapsack.addItem(item)
var maxValueItem = getMaxByValue(items)
if ( knapsack.value < maxValueItem.value && maxValueItem.weight <= knapsack.maxWeight ) do
    clearKnapsack(knapsack)
    knapsack.addItem(maxValueItem)
return knapsack
```

Cautare random

O alta abordare ceruta a fost cea de cautare aleatoare. Aceasta este foarte simpla, in materie de complexitate, dat fiind avand dat un numar de iteratii, generam in mod aleator o solutie candidat, iar la final o alegem pe cea mai buna. Partea interesanta aici, este hotararea numarului de iteratii. In abordarea proprie, am decis sa folosesc o formula ce imi genereaza numarul de iteratii pe baza unui factor de certitudine epsilon dat.

Formula utilizata este urmatoarea:

$$((x^2-x+41)^{1/3} - \log^2_{10}(x * epsilon)) / (1 - epsilon)$$

Pseudocod:

```
var runtimeParams = adjustRuntimeParams(epsilon)
var bestResults = [], avgResults = []
for each i in runtimeParams.restarts do
    var samples = createSamples (runtimeParams.iterations)
    bestResults += getMaxAndValidByValue(samples)
    avgResults += getAverageResult(samples)
var results = bestResults + avgResults
return results.findAll { isValid(result) }.sort(byValue).take(5)
```

Hill climbing

Este o tehnica de optimizare prin cautare locala. Este un algoritm iterativ, ce porneste de la o solutie initiala ce este comparata iterativ (sau nu) cu vecinii sai; in caz ca exista un vecin mai bun, algoritmul continua cu vecinul, pana cand, un numar maxim de iteratii dat este ajuns sau pana cand nu mai exista imbunatatiri. Definitia de vecin este dependenta de metoda de reprezentare aleasa. In cazul sirurilor binare, s-a ales vecinatatea k-flip, in care se inverseaza valoarea a oricaror k biti pentru a obtine un vecin. Toate metodele abordate au fost augmentate cu conceptul de multiple restarts, pentru a evita blocarea intr-un optim local.

Stochastic hill climbing

Aceasta metoda genereaza o solutie candidat, iar pentru un numar de iteratii dat, genereaza un vecin aleator al solutiei curente. Daca acel vecin este mai bun decat solutia in urma evaluarii functiei de fitness, vecinul devine solutia si algoritmul continua. La final, vom avea sirul cu valoarea cea mai mare a functiei de fitness. In cazul restartarilor multiple, vom avea o lista, din care extragem cea mai buna solutie. Pseudocod:

```
var results = []
for each i in params.restarts do
    var currentHilltop = createHilltop (items.count)
    var currentKnapsack = createKnapsack(currentHilltop)
    for each iteration in params.iterations do
        var neighborHilltop = createRandomNeighborHilltop(currentHilltop)
        var neighborKnapsack = createKnapsack (neighborHilltop)
        if isNeighborBetter(currentKnapsack, neighborKnapsack) do
```

```
                currentHilltop = neighborHilltop
                currentKnapsack = neighborKnapsack
        if isValid(currentKnapsack) do
                results.add(currentKnapsack)
return results.sort(byValue).take(5)
```

Metodele de generare a unui hilltop si a unui vecin random sunt metode simple, ce folosesc un generator de numere random si un StringBuilder. Cea de generare a unui vecin random transforma stringul intr-o lista de booleene si inverseaza valoarea unui element la un index aleator.

Steepest ascent hill climbing

Aceasta metoda difera de cea aleatoare prin faptul ca nu generam un vecin random, ci ii generam pe toti, si il luam pe cel mai bun din lista cu toti vecinii solutiei, algoritmul continuand de acolo pana nu mai exista o imbunatatire sau se ajunge la numarul maxim de iteratii. Pseudocod:

```
var results = []
for each i in params.restarts do
        var currentHilltop = createHilltop(items.count)
        var currentKnapsack = createKnapsack(currentHilltop)
        for each iteration in params.iterations do
                var neighbors = getNeighbors(currentHilltop)
                var bestNeighbor = getSteepestNeighbor(neighbor)
                if isNeighborBetter(currentKnapsack, bestNeighbor.knapsack) do
                        currentHilltop = bestNeighbor.hilltop
                        currentKnapsack = bestNeighbor.knapsack
        if isValid(currentKnapsack) do
                results.add(currentKnapsack)
return results.sort(byValue).take(5)
```

Metoda de returnare a vecinilor va fi folosita si in urmatoarea metoda, aici oferind toti vecinii de grad 1. Metoda de returnare a celui mai bun vecin face un maxim intre toti vecinii si returneaza sirul binar si rucsacul cu cel mai bun vecin.

Experimental hill climbing

Aceasta abordare proprie este diferita de cele prezentate anterior si nu facea parte din lista abordarilor obligatorii pentru aceasta tema. Este un algoritm ce se bazeaza pe steepest ascent, dar are cateva imbunatatiri precum restarturi multiple, pentru evitarea unui optim local, regiuni multiple si vecini de grad variabil. Se genereaza in mod adaptiv parametrii de rulare obisnuiti (numarul de restartari si iteratii) dar si numarul maxim de regiuni si gradul maxim de vecinatate. De aici, pentru un numar dat de restartari, generam un numar de regiuni specificat in parametri (o regiune este o solutie candidat - avand regiuni multiple, inseamna ca incepem cu mai multe solutii candidat). Pentru fiecare regiune, aplicam in mod paralel/concurent algoritmul de steepest neighbor, unde gradul vecinilor solutiei este dat de fie numarul iteratiei curente, fie primul grad, in caz ca numarul iteratiilor depaseste gradul maxim. Rezultatele au fost satisfacatoare, insa timpul de rulare in cazul numerelor mai mari de 100 a fost destul de mare. Pentru asta mai trebuie lucrat la optimizare, existand diferite proceduri ce ar putea fi mai rapide/consume mai putine resurse. Pseudocod:

```
var finalResult = [], regionIterationsHilltop = [], regionRestartsHilltop = []
for each restart in params.restarts do
    var candidateHilltops = createHilltopRegions(params.regions)
    for each hilltop in candidateHilltops do parallel
        var knapsack = createKnapsack(hilltop)
        for each iteration in params.iterations do
            var neighbors = getNeighbors(hilltop, getDegree(params.degree, iteration))
            apply steepest neighbor
        if isValid(knapsack) do
            regionIterationsHilltops.add(knapsack)
    if regionIterationsHilltop not empty do
        regionRestartsHilltop.add(getMaxByValue(regionIterationsHilltop))
if regionRestartsHilltop not empty do
    finalResult.addAll(regionRestartsHilltops)
if finalResult is empty do return [ ] else return finalResult.sort(byValue).take(5)
```

Rezultate experimentale

Pe langa datele de test oferite, am generate instante avand 10, 15, 20, 50, 100, 200, 1000, 5000, 10000, 50000 si 100000 de obiecte. Toate instantele au aceleasi constrangeri la valoarea maxima unui obiect si greutate. In schimb, greutatea ghiozdanului a crescut proportional la fiecare crestere de numar de obiecte.

EXHAUSTIVE SEARCH RESULTS			
ITEM COUNT	CONFIDENCE FACTOR	TOTAL VALUE	WEIGHT/MAX WEIGHT
10	-	101	95/100
15	-	154	147/150
20 (generated)	-	157	199/200
20	-	853	597/600

GREEDY SEARCH RESULTS			
ITEM COUNT	CONFIDENCE FACTOR	TOTAL VALUE	WEIGHT/MAX WEIGHT
10	-	80	96/100
15	-	146	139/150
20 (generated)	-	104	191/200
20	-	752	597/600
50	-	428	499/500
100	-	824	997/1000
200 (generated)	-	1829	1999/2000
200	-	170855	199755/200000
1000	-	12706	19999/20000
2000	-	26512	39999/40000
5000	-	56297	80000/80000
10000	-	149717	2480085/250000
50000	-	569112	800000/800000
100000	-	1101450	1500000/1500000

RANDOM SEARCH RESULTS

ITEM COUNT	CONFIDENCE FACTOR	ITERATIONS & RESTARTS	BEST VALUE & WEIGHT	AVG VALUE & WEIGHT #1	AVG VALUE & WEIGHT #2	AVG VALUE & WEIGHT #3	MAX WEIGHT
10	.95	97/24	101/95	101 /95	89 /99	89 /99	100
15	.94	100/25	153/150	140 /115	135 /142	132 /134	150
20	.93	102/26	128/198	128 /172	124 /187	117 /191	200
21	.80	37/10	814/580	764 /580	761 /559	750 /591	600
50	.92	161/40	387/499	369 /464	365 /475	361 /500	500
100	.91	228/50	808/989	730 /1000	720 /971	707 /981	1000
200	.89	100/10	1496/ 1995	1440 /1892	1439 /1972	1435 /2000	2000
201	.80	100/10	152996/ 179096	151240 / 175240	142958 / 166158	138208 / 160808	200000
1000	.80	100/10	8550/ 14594	8423 / 14242	8367 / 13533	8353 / 13660	20000
2000	.75	500/10	17112/ 27257	16973 / 27643	16908 / 26737	16883 / 27844	40000
5000	.70	80/8	40230 / 67814	40140 / 66811	40039 / 66067	39958 / 65592	80000
10000	.65	50/5	78668 / 130011	78634 / 127712	78310 / 126922	78297 / 126899	250000
50000	.60	50/5	380169 / 631699	374684 / 627644	N/A	N/A	800000

RANDOM HILL CLIMBING SEARCH RESULTS

ITEM COUNT	CONFIDENCE FACTOR	ITERATIONS & RESTARTS	BEST VALUE & WEIGHT	AVG VALUE & WEIGHT #1	AVG VALUE & WEIGHT #2	MAX WEIGHT
10	.95	97/24	101/95	83/99	83/99	100
15	.94	100/25	128/145	117/146	115/max	150
20	.93	102/26	122/198	121/183	115/198	200

RANDOM HILL CLIMBING SEARCH RESULTS

ITEM COUNT	CONFIDENCE FACTOR	ITERATIONS & RESTARTS	BEST VALUE & WEIGHT	AVG VALUE & WEIGHT #1	AVG VALUE & WEIGHT #2	MAX WEIGHT
21	.80	37/10	774/595	760/598	689/592	600
50	.91	161/40	375/max	368/max	367/max	500
100	.85	228/50	734/997	660/998	639/999	1000
200	.8	100/10	1271/max	N/A	N/A	2000
201	.8	100/10	171439/ 199839	171381/ 199981	171135/ 199935	200000
1000	.75	100/10	8494/ 14006	8436/ 14902	8375/ 14498	20000
2000	.7	500/10	19021/ 31414	18932/ 30681	18816/ 30731	40000

STEEPEST ASCENT HILL CLIMBING SEARCH RESULTS

ITEM COUNT	CONFIDENCE FACTOR	ITERATIONS & RESTARTS	BEST VALUE & WEIGHT	AVG VALUE & WEIGHT #1	AVG VALUE & WEIGHT #2	MAX WEIGHT
10	.95	97/24	101/95	101/95	89/99	100
15	.94	100/25	154/141	153/145	131/138	150
20	.93	102/26	154/198	147/193	112/200	200
21	.80	37/10	812/589	801/576	787/589	600
50	.91	161/40	396/max	359/max	319/max	500
100	.85	228/50	787/998	711/max	561/max	1000
200	.8	100/10	N/A	N/A	N/A	2000
201	.8	100/10	173511/ 199911	173456/ 199956	173293/ 199993	200000
1000	.75	100/10	10426/ 14815	10235/ 15228	10222/ 15263	20000
2000	.7	500/10	27221/ 38636	27025/ 38233	26861/ 37997	40000

EXPERIMENTAL HILL CLIMBING SEARCH RESULTS							
ITEM COUNT	CONFIDENCE FACTOR	ITERATIONS & RESTARTS	REGIONS & MAX DEGREE	BEST VALUE & WEIGHT	AVG VALUE & WEIGHT #1	AVG VALUE & WEIGHT #2	MAX WEIGHT
10	.95	97/24	3/1	89/99	89/99	89/99	100
15	.94	100/25	4/2	154/147	154/147	154/147	150
20	.93	102/26	5/2	157/199	157/199	157/199	200
21	.80	37/10	4/1	853/597	850/582	850/582	600
50	.91	161/40	4/3	459/499	459/499	459/499	500
100	.85	228/50	6/2	732/max	732/max	701/999	1000
200	.8	100/10	7/2	1668/max	1668/max	1668/max	2000

Comparatii

COMPARISON OF SEARCH RESULTS						
ITEM COUNT	EXHAUSTIVE	GREEDY	RANDOM	STEEPEST ASCENT	RANDOM HC	EXPERIMENT HC
10	**101/95**	80/96	**101/95**	**101/95**	**101/95**	89/99
15	**154/147**	146/139	153/150	**154/141**	128/145	**154/147**
20 (generated)	**157/199**	104/191	128/198	154/198	122/198	**157/199**
20	**853/597**	752/597	814/580	812/589	774/595	**853/597**
50	N/A	428/499	387/499	396/500	375/500	**459/499**
100	N/A	**824/997**	*808/989*	787/998	734/997	732/1000
200 (generated)	N/A	**1829/1999**	1496/1995	N/A	1271/2000	*1668/1000*
200	N/A	170855/ 199755	152996/ 179096	**173511/ 199911**	*171439/ 199839*	N/A
1000	N/A	**12706/ 19999**	8550/ 14594	*10426/ 14815*	8494/ 14006	N/A
2000	N/A	*26512/ 39999*	17112/ 27257	**27221/ 38636**	19021/ 31414	N/A

COMPARISON OF SEARCH RESULTS						
ITEM COUNT	EXHAUSTIVE	GREEDY	RANDOM	STEEPEST ASCENT	RANDOM HC	EXPERIMENT HC
5000	N/A	**56297/ 80000**	40230/ 67814	N/A	N/A	N/A
10000	N/A	**149717/ 2480085**	78668/ 130011	N/A	N/A	N/A
50000	N/A	**569112/ 800000**	380169/ 631699	N/A	N/A	N/A
100000	N/A	**1101450/ 1500000**	N/A	N/A	N/A	N/A