Centro Federal de Educação Tecnológica
de Minas Gerais

Algoritmos e Estruturas de Dados

Problema da mochila

Gabriel Dutra Dias

orientado por
Prof.ª Natália Cosse Batista

Belo Horizonte - MG
2017

Conteúdo

1 Apresentação

O problema da mochila (em inglês, *Knapsack problem*) é um problema de optimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo. [1]

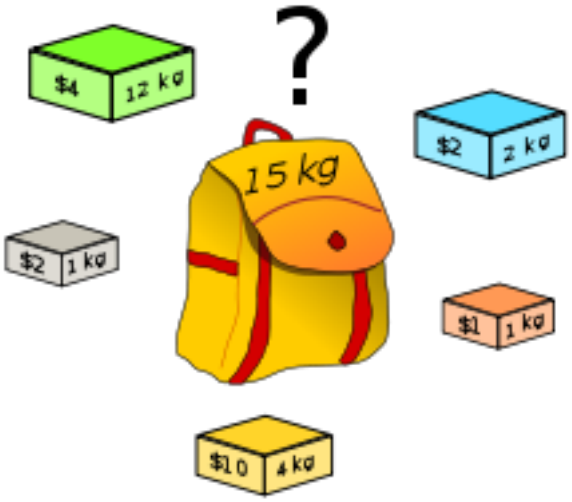

Figura 1: Ilustração - Problema da Mochila (Fonte: Wikipédia)

Neste trabalho apresentarei dois algoritmos para solução do problema da mochila. O primeiro utilizando *Backtracking* e o segundo usando uma estratégia gulosa. Ambos os métodos serão analisados, implementados e testados.

2 Solução usando tentativa e erro (*Backtracking*)

A solução de tentativa e erro representa uma força bruta, pois percorre todas as soluções possíveis e a partir delas escolhe uma ótima.

2.1 Exemplo 1

Para explicar a solução do problema da mochila usando tentativa e erro, tomarei um exemplo onde a mochila tem peso máximo 16 e a loja possui 4 itens.

Tabela 1: Itens da loja no exemplo 1

Item	Valor	Peso
1	$40	2
2	$30	5
3	$50	10
4	$10	5

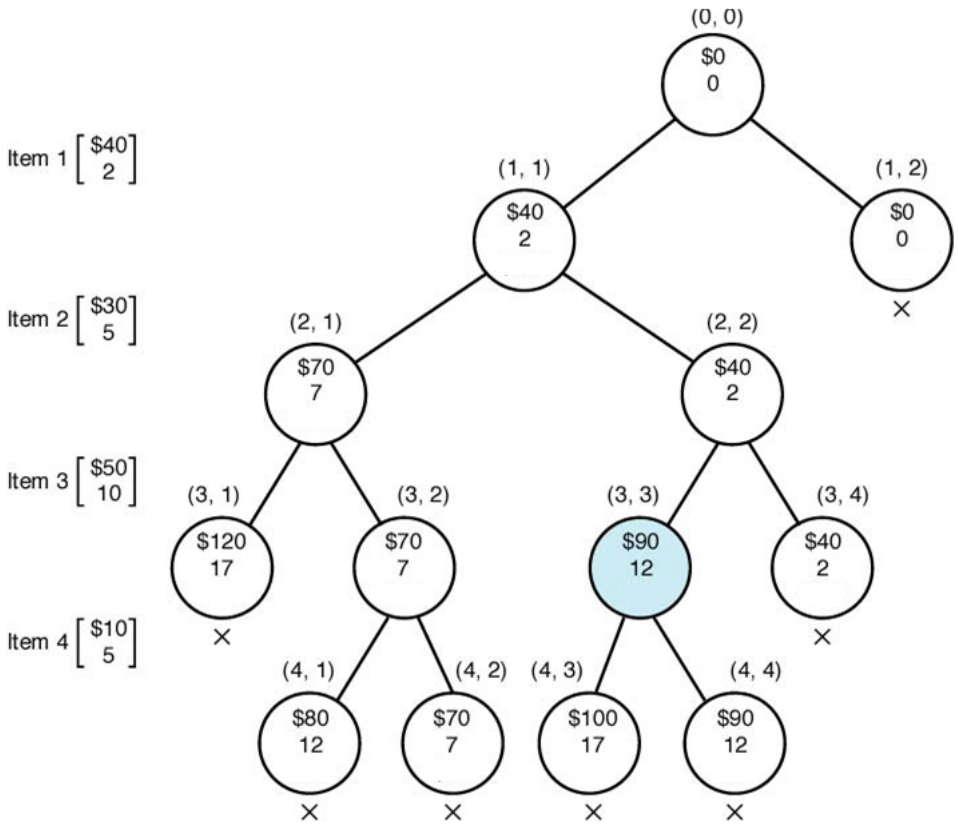

Figura 2: Árvore de possibilidades - Exemplo 1 (Fonte: zakarum.tistory.com)

Para cada item há duas possibilidades de escolhas: levar ou não levar o item. Dessa forma, obtemos a árvore de estados acima. Cada nó representa um estado para o problema da mochila. Esse estado pode ser uma solução possível ou não (na representação acima, as não possíveis estão marcadas com um X). A solução ótima é o estado representado pelo nó destacado. Dessa forma, podemos usar o algoritmo de Backtracking para encontrar a solução ótima no problema da mochila.

2.2 Pseudocódigo

Algoritmo 1: Problema da mochila - Solução usando Backtracking

Entrada: w, C (Capacidade da mochila e conjunto de itens na loja)

Saída: Itens para se levar na mochila

```
1  início
2      solucaoOtima ← {}
3      valorOtimo ← 0
4      pesoOtimo ← 0
5      Função Backtracking(item : inteiro, acumuladorValor : inteiro, acumuladorPeso :
         inteiro, S : Item_n) : vazio
6      inicio
7          valorAtual ← acumuladorValor + C(item)_valor
8          pesoAtual ← acumuladorPeso + C(item)_peso
9          se pesoAtual ≤ w então
10             se valorAtual > valorOtimo então
11                 valorOtimo ← valorAtual
12                 pesoOtimo ← pesoAtual
13                 solucaoOtima ← S + C(i)
14             fim
15         fim
16         se i + 1 ≤ Tamanho(C) e acumuladorPeso < w então
17             Backtracking(C(i + 1), valorAtual, pesoAtual, S + C(i))
18             Backtracking(C(i + 1), acumuladorValor, acumuladorPeso, S)
19         fim
20     fim
21     Backtracking(C(0), 0, 0, ∅)
22 fim
23 retorna solucaoOtima
```

2.3 Análise do algoritmo proposto

A solução por *Backtracking* percorre todas as possibilidades de solução para o problema da mochila e as compara para obter uma solução ótima. Desse modo, temos um algoritmo que garantidamente nos dá uma solução ótima.

Para fazer o cálculo da função de complexidade $f(n)$ vamos considerar o número de comparações realizadas. Consideraremos também o pior caso, em que todos os itens podem ser colocados na mochila, dessa forma, a função sempre chama ela mesma duas vezes. Para cada chamada de função há 4 comparações. Desse modo temos a seguinte equação de recorrencia $T(n)$ em que n é o número de itens:

$$\begin{cases} T(n) = 2T(n-1) + 4 \\ T(0) = 0 \end{cases}$$

Ao fazer a expansão da equação de recorrência temos:

$$\begin{gathered} T(1) = 4 \\ T(2) = 2 \times 4 + 4 = 12 \\ T(3) = 2 \times 12 + 4 = 28 \\ T(4) = 2 \times 26 + 4 = 60 \\ \vdots \\ T(n) = 4(2^n - 1) = O(2^n) \end{gathered}$$

3 Solução usando estratégia gulosa

A solução gulosa baseia-se em escolhermos ótimos locais para no final selecionar um ótimo global. No caso do problema da mochila, o *i-ésimo* item escolhido será aquele com o melhor custo-benefício na loja (otimizando valor e peso).

Desse modo teremos uma solução aproximada para o problema, já que a solução não necessariamente será a ótima global.

3.1 Pseudocódigo

Algoritmo 2: Problema da mochila - Solução gulosa

Entrada: w, C (Capacidade da mochila e conjunto de itens na loja)

Saída: Itens para se levar na mochila

```
1  início
2      S ← {}
3      pesoTotal ← 0
4      repita
5          maisValioso ← ∅
6          custoBeneficioMV ← 0
7          para cada item ∈ C faça
8              custoBeneficio ← item_valor / item_peso
9              se custoBeneficio > custoBeneficioMV e pesoTotal + item_peso < w então
10                 maisValioso ← item
11                 custoBeneficioMV ← custoBeneficio
12             fim
13         fim
14         S ← S + maisValioso
15         C ← C − maisValioso
16         pesoTotal ← pesoTotal + maisValioso_peso
17     até maisValioso = ∅;
18 fim
19 retorna S
```

3.2 Análise do algoritmo proposto

Para fazer a análise do algoritmo tomarei como base o número de comparações realizadas. A única comparação feita está na linha 9, ela ocorre $Tamanho(C)$ vezes para cada seleção local e o tamanho de C diminui em 1 para cada uma dessas escolhas. Iremos considerar n como o número inicial de itens e todas as informações do algoritmo acima. O pior caso será quando todos os itens da loja puderem ser colocados na mochila, ou seja, $\sum_{i=1}^{n} C(i)_{peso} < w$. A análise de custo do algoritmo no pior caso pode ser representada pela seguinte função de custo:

$$f(n) = \sum_{i=1}^{n} \sum_{j=1}^{n-i} 1 = \frac{n^2 - n}{2} = O(n^2) \tag{1}$$

4 Resultados

Ambos os algoritmos propostos foram implementados em C (`https://github.com/gabrieldutra/ProblemaDaMochila`) e um Dataset foi gerado com script desenvolvido em Shell Script para esse fim.

Resultados foram recolhidos variando N com 10, 20, 30, 100, 300 e 500 e a razão $\frac{Capacidade}{Peso_{total}}$ variando com 50, 25 e 12,5%. O valor dos itens foi gerado aleatoriamente entre \$10 e \$200. O tempo e a diferença apontada no valor total entre os dois algoritmos foram anotados.

Tabela 2: Tabela de Resultados

N	%	Tempo Guloso	Tempo Backtracking	Diferença no resultado
10	50	<1 ms	<1 ms	\$28
	25	<1 ms	<1 ms	\$45
	12,5	<1 ms	<1 ms	\$14
20	50	<1 ms	365 ms	\$35
	25	<1 ms	23 ms	\$39
	12,5	<1 ms	2 ms	\$11
30	50	<1 ms	**	**
	25	<1 ms	**	**
	12,5	<1 ms	35 ms	\$69
100	50	<1 ms	**	**
	25	<1 ms	**	**
	12,5	<1 ms	**	**
300	50	<1 ms	**	**
	25	<1 ms	**	**
	12,5	<1 ms	**	**
500	50	1.5 ms	**	**
	25	1.5 ms	**	**
	12,5	1.5 ms	**	**

4.1 Análise

A maioria dos resultados para valores de N grande não puderam ser computados já que excediam o limite da máquina utilizada nos testes.

É possível notar o crescimento rápido do tempo de execução do algoritmo de Backtracking em comparação com o Guloso, que se mantém quase que constante durante os testes.

A maior diferença em valor computada foi \$69, e no banco de testes utilizado o algoritmo Guloso não obteve a solução ótima nenhuma vez.

4.2 Tempo de entrada e saída

Os tempos de entrada e saída também foram anotados, o maior tempo de entrada foi 15 ms para N=500.

5 Conclusão

É possível concluir que o algoritmo guloso mesmo não dando uma solução ótima, nos dá uma solução com uma diferença em valor não muito grande, o que pode ser suficiente dependendo do problema que estamos lidando.

Desse modo, antes de aplicar uma solução a um problema é necessário fazer uma análise de custo em cima do contexto do problema. Muitos dos casos de teste não tiveram resultado com o algoritmo de Backtracking, pois ele possui complexidade exponencial de tempo.

Referências

[1] Mitsuo Ge Xinjie Yu. *Introduction to Evolutionary Algorithms, p.270-271.* 2003.