Pontifícia Universidade Católica do Rio de Janeiro

Desenvolvimento de algoritmos para o problema da mochila e suas variações

INF2926 | PAA | Pós-graduação em Informática

Isabella Vieira Ferreira
Rebecca Porphírio da Costa de Azevedo

1 Introdução

O problema da mochila (em inglês, *Knapsack problem*) é um problema de otimização combinatória. O nome se dá devido ao problema de 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 [3].

O problema da mochila é um dos 21 problemas NP-completos de Richard Karp, exposto em 1972. A formulação do problema é extremamente simples, porém sua solução é mais complexa. [1]. Normalmente, este problema é resolvido com programação dinâmica, obtendo então a resolução exata do problema, mas também sendo possível usar algoritmo guloso e meta-heurísticas (ex. algoritmos genéticos) para soluções aproximadas.

O objetivo desse trabalho é resolver o problema da mochila e suas variações, sendo elas o problema da mochila fracionária, o problema da mochila binária e o problema da mochila com conflitos. A descrição de cada problema bem como sua implementação se encontra nas Seções 2, 3 e 4, respectivamente.

2 Mochila Fracionária

O objetivo do problema da mochila fracionária é selecionar um subconjunto de itens com o intuito de maximizar o valor total, de forma que este não exceda a capacidade da mochila. Neste caso, é possível utilizar 10% de um item, por exemplo. Sendo assim, uma mochila possui capacidade B e dada uma lista de n items i, cada um é caracterizado por pesos não negativos w_i e valores p_i.

2.1 Algoritmo e sua otimalidade

O pseudocódigo 1 apresenta a solução gulosa para o problema da mochila fracionária. Nele, temos que k é a capacidade total da mochila, P é um vetor de lucros e W é um vetor de pesos dos itens da mochila. Primeiramente, ordenamos os itens em ordem descrescente pela razão do seu lucro pelo seu peso. Tal ordenação possui complexidade *O(n log n)*, onde n é o número de itens. Com isso, enquanto o peso interno da mochila não exceder sua capacidade total, nós vamos inserindo items dentro da mochila. A verificação de todos os itens a serem inseridos na mochila possui, no pior caso, complexidade *O(n)*. Para verificar se o item cabe ou não na mochila, verificamos se o peso interno da mochila mais o peso do item é menor ou igual à capacidade total da mochila. Caso seja, o item é inserido na mochila e seu peso interno é atualizado. Caso contrário, adicionamos a fração do item que cabe na mochila e atualizamos o peso como a capacidade total da mochila, não sendo possível inserir mais itens na mesma. A complexidade de cada instrução está demonstrada em cada linha no pseudocódigo 1. Com isso, temos que a complexidade final do algoritmo da mochila fracionária é *O (n log n)*.

Algorithm 1 Fractional Knapsack

```
1:  procedure FractionalKnapsack(k, P, W)
2:      items ← sortItemsBy(p/w)                    ▷ O (n log n)
3:      weight ← 0                                  ▷ O (1)
4:      x ← [0, 0, ..., 0]                          ▷ O (1)
5:      while weight<k do                           ▷ O (n)
6:          i, w ← getNextItem()                    ▷ O (1)
7:          if weight + w <= k then                 ▷ O (1)
8:              x[i] ← 1                            ▷ O (1)
9:              weight ← weight + w                 ▷ O (1)
10:         else
11:             x[i] ← (k − weight)/w               ▷ O (1)
12:             weight ← k                          ▷ O (1)
13:     return x
```

Para provar que o método guloso para resolver o problema da mochila fracionária é ótimo, vamos considerar a solução G do algoritmo guloso e a solução O, que é uma solução ótima. Vamos supor que o índice j é o último índice que possui solução comum em G e O. Sendo assim, se trocarmos o item O_{j+1} pelo item G_{j+1}, a solução O será mais próxima de G. Ao fazer operações de *swap* para todos os itens, temos que a solução O pode ser convertida em G, sem piorar seu valor. Sendo assim, temos que a solução G é ótima.

2.2 Estrutura de dados e sua complexidade

Utilizamos dois vetores que armazenam lucro e peso de cada item. Os vetores de lucro e peso são de tamanho n, onde n é o número de itens. Sendo assim, temos que a complexidade dos vetores é $O(n) + O(n) = O(n)$.

2.3 Prova da complexidade do algoritmo com instâncias

Para verificar se a complexidade teórica do algoritmo acontece, de fato, na prática, analisamos o tempo de execução do algoritmo pelo número de itens a serem inseridos na mochila. O Gráfico 1 apresenta uma comparação entre o tempo de execução do nosso algoritmo com sua função de complexidade.

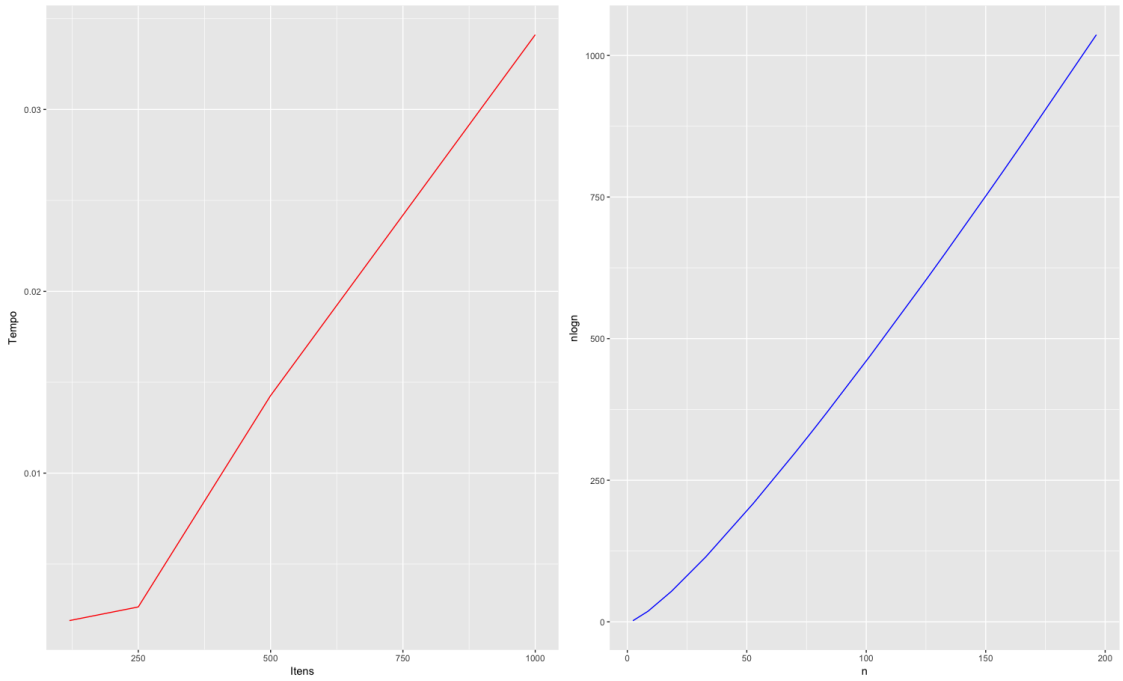

Figura 1: Gráfico de tempo (s) x número de itens em comparação com a função de complexidade $O\ (n\ log\ n)$

3 Mochila Binária

O problema da mochila binária é o mesmo que o da mochila fracionária, descrito na Seção 2, exceto pelo fato de que aqui não são permitidos itens fracionários. O algoritmo guloso desenvolvido para o problema da mochila fracionária nem sempre resulta em uma solução ótima, sendo assim, essa estratégia não funciona para a mochila binária. Como um contra-exemplo, temos uma mochila que tem capacidade máxima de 100kg e três itens: item 1 pesa 20kg e custa 120 reais; item 2 pesa 40kg e custa 200 reais; item 3 pesa 60kg e custa 240 reais. Dessa forma, o valor custo por peso do item 1 é 6, do item 2 é 5 e do item 3 é 4. Com isso, a estratégia gulosa colocaria primeiramente o item 1 na mochila, visto que seu custo por peso é maior que o dos itens 2 e 3. Entretanto, a solução ótima é colocar os itens 2 e 3 na mochila; e não carregar o item 1. As duas possíveis soluções que envolvem o item 1 são subótimas. Carregar o item 1 não funciona para o problema da mochila binária, visto que não é possível preencher a capacidade total da mochila. Com isso, no problema da mochila binária nós devemos comparar a solução do subproblema no qual o item foi excluído anteriormente para fazer a escolha, utilizando programação dinâmica.

3.1 Algoritmo e sua otimalidade

O algoritmo 2 apresenta um pseudocódigo para o problema da mochila binária. Ao utilizar programação dinâmica, para encontrar o valor *OPT(n)*, nós não precisamos somente do valor *OPT(n-1)*, mas também precisamos saber a solução ótima utilizando os *n-1* itens e peso total k - w_n. Sendo assim, nós utilizaremos subproblemas: um para cada conjunto inicial de itens $\{1, ..., i\}$ e cada possível valor disponível para o peso w. Assumindo k como um inteiro (capacidade da mochila) e todos precisam de $\{1, ..., n\}$ com pesos w_i. Nós teremos subproblemas para cada item $\{1, ..., n\}$ e um inteiro $0 \leq w \leq k$. Vamos utilizar *OPT(i,w)* para denotar o valor ótimo de cada solução utilizando um subconjunto $\{1, ..., i\}$ com o máximo valor de peso w permitido [2]. O valor que estamos procurando no final é *OPT(n,k)*. Da mesma forma, utilizando os mesmos argumentos para os subconjunto de itens $\{1, ..., i\}$ e o máximo peso permitido w, temos a seguinte equação de recorrência:

- se w $< w_i$, então OPT(i,w) = OPT(i-1, w)
- do contrário, OPT(i,w) = max (OPT(i-1,w), w_i + OPT(i-1, w-w_i))

Com isso, temos que a complexidade final do algoritmo da mochila binária é *O(nk)*.

Algorithm 2 Binary Knapsack

```
1: procedure BinaryKnapsack(k, P, W)
2:     m[n][k] ← [0, 0, ..., 0]
3:     for i ← 0...n do                                   ▷ O (n)
4:         for j ← 1...k do                               ▷ O (k)
5:             if w[i] > j then                           ▷ O (1)
6:                 m[i, j] ← m[i − 1, j]                  ▷ O (1)
7:             else                                       ▷ O (1)
8:                 m[i, j] ← max(m[i − 1, j], P[i − 1] + m(i − 1, j − W[i − 1]])
9:     return m
```

3.2 Prova da complexidade do algoritmo com instâncias

Para verificar se a complexidade teórica do algoritmo acontece, de fato, na prática, analisamos o tempo de execução de acordo com a capacidade da mochila e o número de itens, como mostrado no Gráfico 2. Podemos perceber que, como mostrado pela complexidade de tempo do algoritmo de *O (nk)*, quanto maior o número de itens e a capacidade da mochila, maior é o tempo de execução do algoritmo.

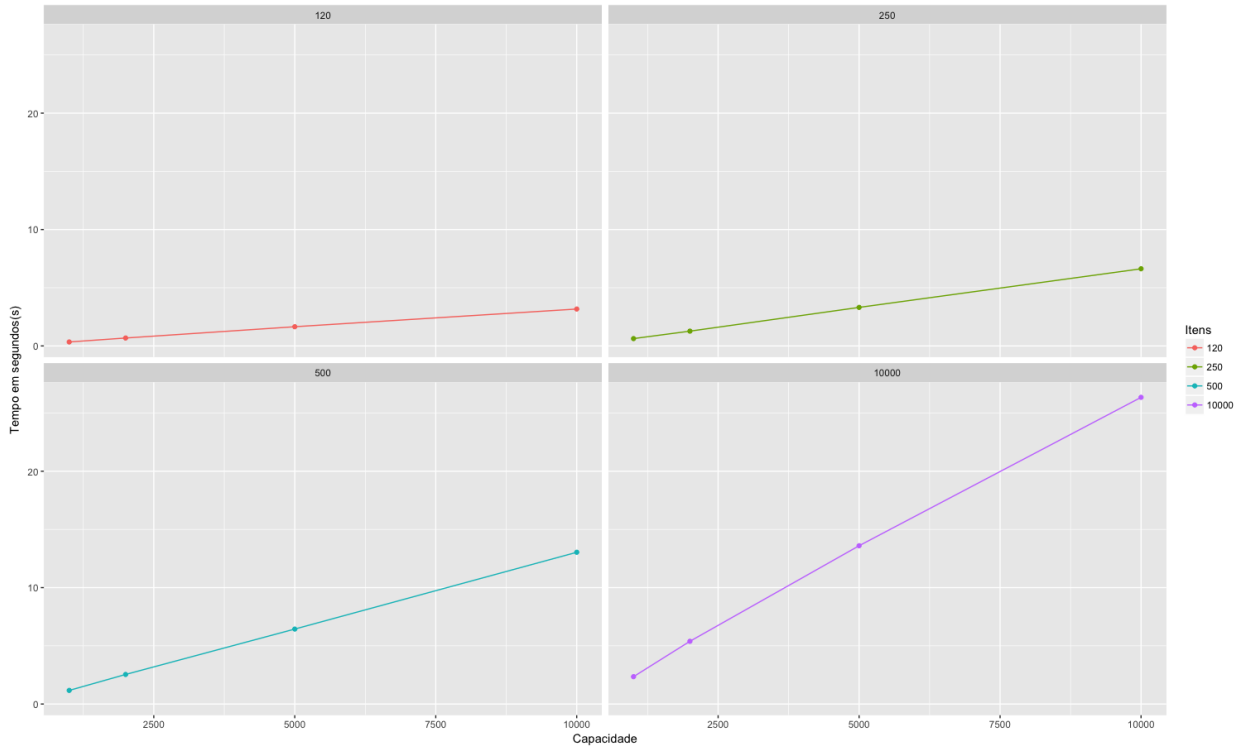

Figura 2: Gráfico de capacidade da mochila x número de itens x tempo (s)

4 Mochila com Conflitos

Este problema é similar ao da mochila binária descrita na Seção 3. Entretanto, agora o objetivo é maximizar o valor total de itens na mochila sem violar as restrições de conflitos entre itens.

4.1 Heurística proposta e sua complexidade

A heurística desenvolvida possui como base a solução proposta para a mochila binária (descrita na Seção 3). A diferença é que além de verificar se um item cabe na mochila, verificamos também se há algum item na mochila em conflito com o item que se deseja inserir. Caso não haja nenhum conflito e o item caiba na mochila (peso atual da mochila mais o peso do item é menor que a capacidade total da mochila), o item é inserido na mochila e o peso atual da mesma é atualizado. Ressaltamos que essa heurística não necessariamente retorna a solução ótima.

O algoritmo 3 apresenta a heurística para o problema da mochila com conflitos. Através da análise de complexidade descrita ao lado de cada instrução, concluimos que a complexidade final do algoritmo é $O(n \log n)$.

4.2 Estrutura de dados e sua complexidade

Utilizamos 2 vetores para armazenar lucro e peso; e uma lista de conjuntos para armazenar os conflitos dos itens. Os vetores de lucro e peso são de tamanho n, onde n é o número de itens. Para armazenar os conflitos, preferimos conjuntos (set()) que são estruturas de dados rápidas em python. A lista de conjuntos possui n conjuntos (número de itens) e, no pior caso, n itens dentro de cada conjunto (quando, por exemplo, um item tem conflito com todos os outros). Sendo assim, temos que a complexidade de espaço do algoritmo é de $O(n) + O(n) + O(n^2) = O(n^2)$.

Algorithm 3 Knapsack with Conflicts

```
1:  procedure KNAPSACK_CONFLICTS(k, P, W)
2:      items ← sortItemsBy(p/w)                          ▷ O (n log n)
3:      x ← [0, 0, ..., 0]                                ▷ O (1)
4:      weight ← 0                                         ▷ O (1)
5:      conflicted_items ← []                              ▷ O (1)
6:      while weight<k do                                  ▷ O (n)
7:          i, w, c ← getNextItem()                        ▷ O (1)
8:          if weight + w <= k ⋀ i ∉ conflicted_items then  ▷ O (1)
9:              x[i] ← 1                                   ▷ O (1)
10:             weight ← weight + w                        ▷ O (1)
11:             conflicted_items ← c
12:     return x
```

4.3 Prova da complexidade do algoritmo com instâncias

Para verificar se a complexidade teórica do algoritmo acontece, de fato, na prática, analisamos o tempo de execução do algoritmo pelo número de itens a serem inseridos na mochila. A Tabela 1 apresenta os resultados obtidos para cada instância, bem como o tempo de execução de cada uma. O Gráfico 3 apresenta uma comparação entre o tempo de execução do nosso algoritmo com sua função de complexidade.

Tabela 1: Número de itens utilizados, peso total, lucro e tempo gasto para cada instância

Questão 3				
Instância	**Número itens utilizados**	**Peso Total**	**Lucro**	**Tempo Gasto (s)**
Data-120-Q1	6	473	435	0.0004830360413
Data-120-Q2	6	473	435	0.0004589557648
Data-120-Q3	6	473	435	0.0004699230194
Data-120-Q4	6	473	435	0.000416040420532
Data-250-Q1	8	824	1378	0.0009970664978
Data-250-Q2	8	824	1374	0.0008869171143
Data-250-Q3	8	824	1374	0.000883102417
Data-250-Q4	8	824	1374	0.001260042191
Data-500-Q1	10	895	3928	0.002166986465
Data-500-Q2	11	1125	3965	0.001775979996
Data-500-Q3	11	1125	3965	0.001807928085
Data-500-Q4	11	1125	3965	0.001794099808
Data-1000-Q1	10	919	8138	0.008740186691
Data-1000-Q2	10	919	8138	0.004012107849
Data-1000-Q3	10	919	8138	0.004637002945
Data-1000-Q4	10	919	8138	0.007377147675

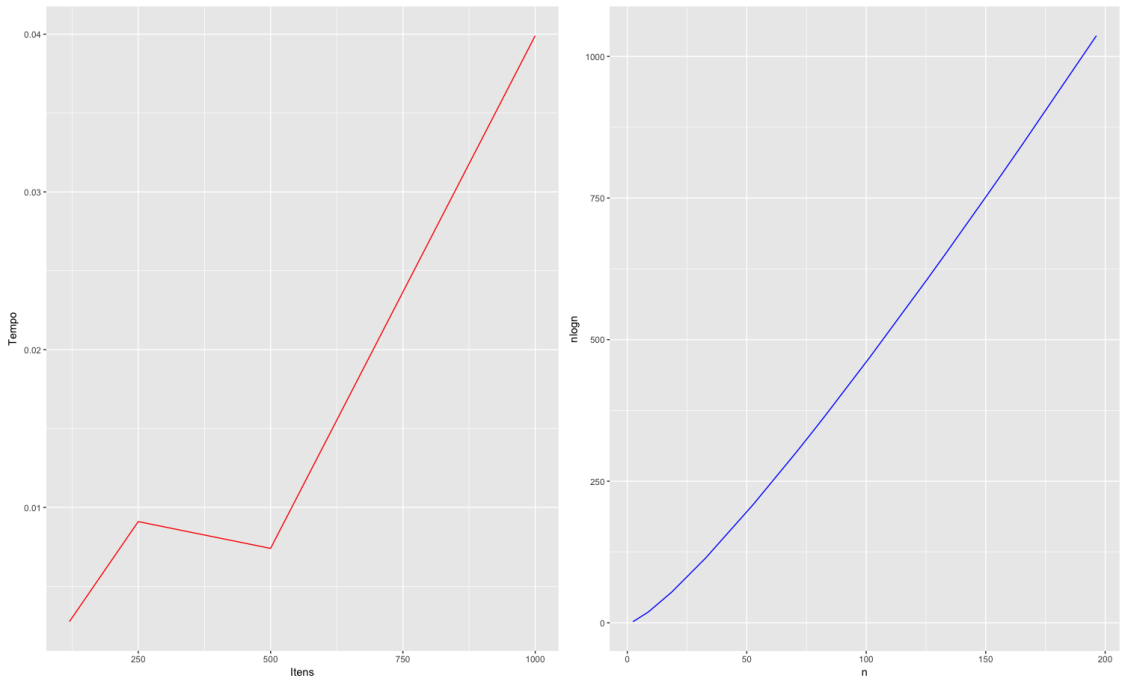

Figura 3: Gráfico de tempo (s) x número de itens na mochila

5 Instruções de como executar o algoritmo

Para executar o algoritmo, basta entrar no contexto da pasta e digitar:

./trabalho <caminho da instância> <questão>

onde:

- <caminho da instância> é o diretório onde estão os arquivos de entrada.
- <questão> é a questão a ser resolvida, podendo ser 1 para o problema da mochila fracionária, 2 para o problema da mochila binária ou 3 para o problema da mochila com conflitos.

Referências

[1] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of computer computations, Springer, 1972, pp. 85–103.

[2] J. Kleinberg and E. Tardos, *Algorithm design*, Pearson Education India, 2006.

[3] X. Yu and M. Gen, *Introduction to evolutionary algorithms*, Springer Science & Business Media, 2010.