Fúlvio Taroni Monteforte

Thiago Lima Bahia Santos

Trabalho Prático I: O Problema da Mochila

CEFET-MG

Belo Horizonte

2019

1. INTRODUÇÃO

1.1 Contextualização

O Problema da Mochila (*Knapsack Problem*) consiste em um problema de otimização combinatória que tem por objetivo determinar a configuração de itens a serem incluídos em uma mochila, tal que:

1. O peso total dos itens seja menor ou igual à capacidade da mochila;
2. O valor total dos itens seja o maior possível.

Seu estudo é amplamente difundido há mais de um século, com trabalhos antigos que datam de 1897[1]. O nome "Problema da Mochila" remonta aos primeiros trabalhos do matemático Tobias Dantzig[2] (1884-1956) e refere-se às dificuldades do cotidiano de empacotar os itens mais valiosos ou úteis em uma bagagem, sem sobrecarregá-la.

Matematicamente, pode-se assim descrevê-lo:

- Maximizar $\{f(x)\}$ = Maximizar $\{\sum_{i=1 \text{ até } n} x(i)\, p(i)\}$, sujeito à
  - $\sum_{i=1 \text{ até } n} x(i)\, p(i) \leq P$
  - $x(i) \in \{0, 1\}, i = 1, 2, \ldots, n$

Os dois algoritmos propostos e implementados neste trabalho prático contemplam, respectivamente, o paradigma de força bruta e o paradigma guloso como como técnicas de solução do problema.

1.2 Soluções Propostas

1.2.1 Algoritmo de Força Bruta

O algoritmo de força bruta, ou algoritmo de tentativa e erro, é uma técnica de resolução de problemas muito geral e relativamente simples, que consiste em enumerar sistematicamente todos os possíveis candidatos à solução e verificar se cada candidato satisfaz a afirmação do problema.

A solução por força bruta foi inspirada na técnica apresentada pelo artigo científico "Diferentes Abordagens para resolver o Problema da Mochila 0/1"[3], de autoria de Maya Hristakeva e Dipti Shrestha, da Simpson College. Para essa implementação é preciso entender que, se houver 'n' itens para escolha, haverá '2^n' combinações de itens para a mochila. Como cada item pode ou não ser escolhido, existem '2' condições possíveis, que podem ser representadas pelos bits '0' e '1'. Assim, é possível construir a tabela verdade para o problema por meio de um vetor de bits de tamanho 'n', contendo apenas 0s e 1s, onde i-ésimo termo representa se o item em questão foi ou não selecionado e, então, é possível testar, para cada linha da tabela verdade, a combinação de itens que maximiza o valor em função da capacidade da mochila.

---

[1] Mathews, G. B. "On the partition of numbers". Proceedings of the London Mathematical Society.
[2] Dantzig, T. Numbers: The Language of Science.
[3] Hristakeva, M. Different Approaches to Solve the 0/1 Knapsack Problem.

Pseudocódigo para o algoritmo de força bruta:

```
//Entrada: int: Capacidade da mochila, Quantidade de itens;
         struct: lista de itens; // estrutura contendo o peso e o valor de cada item.
         struct: lista de itens a serem colocados na mochila; //variável referência para o
programa principal. inicializada com 0s, ao final da execução, a lista conterá 1s nas
posições de mesmo índice dos itens selecionados e 0s nas posições de mesmo índice dos
itens não selecionados.

Algoritmo:
Variáveis: struct: lista de itens temporária; //inicializada com 0s, a lista temporária será a
variável onde serão armazenadas e testadas as diversas combinações de itens.
          item: item temporário, melhor item; //ambos contendo peso e valor.

para: i ← 1 até 2^n
        j ← n
        peso temporário ← 0;
        valor temporário ← 0;
        enquanto: lista de itens temporária[j] != 0 e j > 0
                lista de itens temporária [j] ← 0;
                j ← j - 1;
        lista de itens temporária [j] ← 1;
        para: k ← 1 até n
                se (lista de itens temporária [k] == 1), então:
                        peso temporário = peso temporário + peso do item selecionado[k];
                        valor temporário = valor temporário + valor do item selecionado[k];
        se: (valor temporário > melhor valor) e
            (peso temporário <= capacidade da mochila), então:
                melhor peso ← peso temporário;
                melhor valor ← valor temporário;
        para: k ← 1 até n
                lista de itens a serem colocados na mochila[k] ← lista de itens temporária[k];
                //variável referência para o programa principal.
```

É importante salientar que os índices remissivos das variáveis aqui denominadas ‘Listas de itens’ variam de ‘0’ à ‘n – 1’, sendo ‘n’ o número de itens avaliados.

1.2.2 Algoritmo Guloso

O algoritmo guloso, como o próprio nome sugere, é um algoritmo que segue uma heurística para solução de problemas de fazer a escolha que parece ser a melhor naquele momento, com a intenção de que essa escolha leve a uma solução ideal global.

Nessa abordagem cada item possui em sua estrutura peso, valor e uma razão calculada a partir da divisão do valor pelo peso. Partindo dessa referência é possível que a implementação seja capaz de tomar suas decisões localmente pegando, da maior para a menor razão os itens avaliados e, à cada iteração, verificando se a mochila está cheia. Vale ressaltar que o algoritmo de ordenação utilizado foi o *Selection sort*, o qual será melhor discutido posteriormente.

Pseudocódigo para o algoritmo guloso:

```
//Entrada: struct: lista de itens; //Lista de itens é uma estrutura contendo índice, peso, valor e razão de cada item. O índice é usado apenas na impressão do arquivo e a razão é usada apenas pelo algoritmo de ordenação.
        int: quantidade de itens, capacidade da mochila, peso máximo, valor máximo;

Algoritmo:
Variáveis:
    int: peso temporário ← 0;
    int: valor temporário ← 0;

//Considere que Lista de itens já é uma lista ordenada da maior para a menor razão.
para: i ← 1 até quantidade de itens
    se: (peso do item selecionado + peso temporário <= capacidade), então
        peso temporário ← peso temporário + peso do item selecionado[i];
        valor temporário ← valor temporário + valor do item selecionado[i];
    se não: encerra o loop
peso máximo ← peso temporário;
valor máximo ← valor temporário;
```

Assim como na implementação anterior, os índices remissivos de cada item variam de ‘0’ à ‘n – 1’, para os ‘n’ itens avaliados.

2 ANÁLISE DE COMPLEXIDADE

2.1 Algoritmo de Força Bruta

Um olhar mais atento ao pseudocódigo do algoritmo de força bruta revela que, para obter a seleção ótima de itens para a mochila, é necessário o uso de um laço maior, externo, que se repete ‘2^n’ vezes, para ‘n’ igual ao tamanho da entrada de dados. Internamente 3 outros laços são executados sequencialmente, com ‘n’ repetições.

Logo, tem-se que:

(1) $O(2^n) \times (O(n) + O(n) + O(n))$
(2) $O(2^n) \times (3\ O(n))$ // Como 3 é uma constante
(3) $O(2^n) \times O(n)$
(4) $\therefore O(n2^n)$

Portanto, a ordem de complexidade do algoritmo de força bruta é $O(n2^n)$. Como seu crescimento é exponencial, seu uso é indicado apenas para entradas pequenas, uma vez que para entradas muito grandes seu custo de tempo o inviabiliza.

2.2 Algoritmo Guloso

Para que o algoritmo guloso selecione uma combinação de itens adequada para a mochila, é necessário, antes, que seja feita uma ordenação de prioridade dos itens a serem adicionados. Conforme já discutido, essa prioridade é dada pela razão entre o valor e o

peso item. Para a ordenação o programa faz uso de um algoritmo do tipo *Selection sort* que possui complexidade de tempo $O(n^2)$.

Já a função para a seleção dos itens utiliza apenas um único laço de repetição que para o pior caso executa ‘n’ vezes, ou seja, sua complexidade é O(n).

Repare que $O(n^2)$ é assintoticamente dominante em relação à O(n). Portanto, o algoritmo de ordenação é o maior responsável pelo tempo de execução do programa.

Logo, tem-se que:

(1) $O(n^2) + O(n)$
(2) $O(max(n^2, n))$ // n^2 é assintoticamente dominante em relação a n
(3) $\therefore O(n^2)$

3 DISCUSSÃO DA OTIMALIDADE DAS SOLUÇÕES

As duas abordagens propostas para a solução do Problema da Mochila possuem características muito pessoais. A solução por tentativa e erro é naturalmente intuitiva e geralmente constitui o paradigma mais simples para solucionar um determinado problema, qualquer que seja, sempre retornando a solução ótima. No caso do Problema da Mochila, o seu custo de tempo exponencial $O(n2^n)$ a torna completamente inviável à medida que o número de combinações de itens a serem testados cresce, uma vez que esse paradigma necessita testar exaustivamente todas as combinações possíveis para garantir a otimalidade da solução.

Para esse tipo de problema, o emprego de heurísticas garante um equilíbrio entre o custo de tempo para a execução do programa e a garantia de que se tenha como retorno uma boa solução, mesmo que essa solução não seja a melhor. Nesse sentido, a proposta do algoritmo guloso aparece como uma alternativa à força bruta, com menor custo de tempo, na ordem de $O(n^2)$, mas que não necessariamente retornará a melhor solução. Resolvendo o problema localmente é possível obter ganhos consideráveis, sobretudo quando o volume de dados de entrada é alto.

Ainda sobre o paradigma guloso é importante perceber que, como o algoritmo de ordenação utilizado é o principal responsável pela ordem de complexidade determinada, alterá-lo para uma implementação mais eficiente em termos de custo de tempo, como o *Shell sort* por exemplo, afetaria diretamente os testes realizados.

4 DOCUMENTAÇÃO DO PROGRAMA

De forma geral, o programa funciona fazendo a leitura de um arquivo que contém os dados necessários para o funcionamento dos algoritmos que resolveram o problema da mochila e no final gerando um outro arquivo com o resultado, além disso o programa também informa no terminal, os tempos relacionados à execução do programa. Quanto a implementação do código, no algoritmo guloso foi decidido utilizar um algoritmo de ordenação do tipo *Selection Sort*, ele funciona passando sempre o menor valor do vetor para a primeira posição, depois o segundo menor para a segunda posição e sucessivamente até que ordene o penúltimo elemento restante, (pois o último é o elemento

que sobrar). Apesar de ser um algoritmo ineficiente para vetores grandes, foi decidido usar ele pois é o de fácil implementação, não necessitando se basear em algoritmos de terceiros. No algoritmo de força bruta, foi decido usar a função *pow()* da biblioteca *math.h*, isso torna o código mais simples de mais fácil entendimento, porém limita o teste máximo que pode ser realizado. A função *pow()* foi usada para calcular o número de iterações que o laço mais externo deveria fazer (2^n iterações, sendo n a quantidade itens), gravando o resultado dentro uma variável *unsigned long long int*, o número máximo de itens que se poderia testar é 63. Vale lembrar que para uma entrada com 63 itens não existe tempo útil para obter o resultado.

Os programas fazem uso de módulos que são interdependentes, em ambos os programas existem módulos separados para que seja feita a leitura do arquivo de entrada e impressão do arquivo com o resultado. No programa para força bruta, apenas um módulo é responsável por calcular qual a melhor mochila possível, enquanto no programa com o algoritmo guloso, o módulo para o cálculo da melhor mochila é dependente de um módulo para ordenação dos itens. Além disso, em ambos os programas, existe um módulo suprimido caso o usuário queira que seja criado um arquivo aleatório para teste.

As TADs utilizadas no programa são uma lista de estrutura do tipo item. Na versão para o algoritmo guloso, essa estrutura contém os dados valor, peso, razão e ID. Na versão para o algoritmo de força bruta, a estrutura contém somente os dados peso e valor.

A entrada para o programa é um arquivo contendo na primeira linha a capacidade da mochila, na segunda linha o número itens, e nas linhas seguintes os pesos e valores dos itens lado a lado. A saída do programa é um arquivo contendo os itens que foram selecionados para a mochila. Os itens estão dispostos um em cada linha com seus respectivos IDs (0 para o primeiro item), peso e valor. Nas duas últimas linhas estão gravados a soma dos pesos e valor total contido na mochila respectivamente.

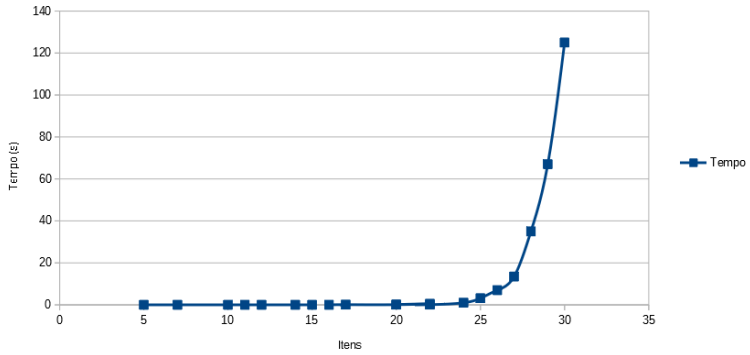
Para compilar o programa do algoritmo de força bruta, no terminal é preciso digitar o comando *-lm* para que seja reconhecida a função *pow()*, no algoritmo guloso isso não é necessário. Os programas não possuem interação como usuário em tempo de execução, logo é esperado que o nome do arquivo de entrada possua o mesmo nome que o programa está buscando (*arquivo_leitura.txt [força bruta]* ou *arquivo_entrada.txt [guloso]*) e que o arquivo esteja salvo na mesma pasta.

No algoritmo de força bruta foram realizados testes para entradas de 5, 10, 15, 20, 25 e 30 itens. No algoritmo guloso foram realizados testes para entradas de 50, 100, 500, 1000, 5000, 10000, 50000 e 100000 itens.

5 EXPERIMENTOS E ANÁLISES DOS RESULTADOS

5.1 Algoritmo de Força Bruta

Figura 1 – Tempo de Usuário para o Algoritmo de Força Bruta



Fonte: Produzido por Fúlvio Taroni e Thiago Bahia

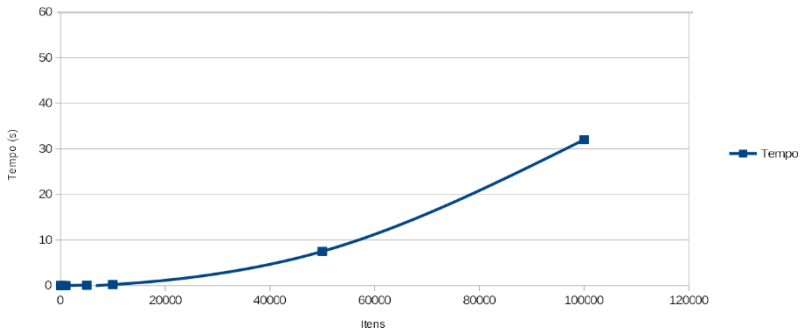
Conforme demonstrado na seção sobre análise de complexidade, o algoritmo de força bruta apresenta um tempo exponencial de execução. Os testes realizados demonstram que para até 24 entradas o tempo de execução sofre pequena variação, sendo praticamente instantâneo ao usuário. A partir de 25 entradas os valores começam a ficar mais significativos e o supercrescimento no tempo de execução do algoritmo de força bruta torna-se evidente.

Enquanto para 25 itens o programa leva pouco mais de 3 segundos de tempo de usuário, para 30 itens ele leva 1 minuto. Com 35 entradas, o tempo de execução sofre um salto expressivo para mais de 1 hora. Nesse sentido, se hipoteticamente fossem adicionados 300 itens à entrada do algoritmo de força bruta seria esperado que o programa leve mais do que $1{,}9 \times 10^{112}$ milênios para rodar.

É importante ressaltar que todos os dados utilizados para a projeção do gráfico foram obtidos por meio da função *getrusage()* e que as medições do tempo de sistema começam a aparecer somente a partir de 24 entradas. O resultado entregue por essa implementação sempre será o resultado ótimo.

5.2 Algoritmo Guloso

Figura 2 – Tempo de Usuário para o Algoritmo Guloso



Fonte: Produzido por Fúlvio Taroni e Thiago Bahia

Diferentemente do caso anterior, aqui a projeção do gráfico deixa evidente uma função com outro tipo de comportamento de crescimento: o polinomial. Nesse caso, o algoritmo consegue lidar com um número relativamente maior de entradas, uma vez que, aumentando a entrada, o tempo de execução não cresce tão rapidamente quanto na solução por força bruta.

Assim para uma lista com 10.000 itens ou menos o programa executa de forma muito célere, já para entradas maiores a curva de crescimento começa a ficar aparente. Com 50.000 itens a implementação levou, em média, 7 segundos de tempo de execução. Já com 100.000, o tempo necessário foi de aproximadamente 32 segundos.

Os dados utilizados para a projeção do gráfico para o algoritmo guloso também foram obtidos por meio da função *getrusage()*. Nesse caso, as medições do tempo de sistema aparecem por volta de 20.000 entradas.

Apesar das vantagens relacionadas ao tempo de execução do programa, esse algoritmo raramente entrega a solução ótima para uma determinada lista de itens.

5.3 Configurações da máquina utilizada

Sistema:

- Windows 10 Home
- Processador: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
- Memória instalada (RAM): 8,00 GB (utilizável 7,86 GB)

Máquina Virtual:

- Ubuntu 18.04 Desktop
- Processador(es) utilizados: 2 de 8 threads
- Memória base: 4,00 GB

A implementação e os testes dos algoritmos foram realizadas no ambiente de uma máquina virtual emulada a partir do software *Oracle VM VirtualBox*.

6 CONCLUSÕES

Analisando os testes realizados com os dois algoritmos, foi possível concluir que o algoritmo de força bruta é adequado para resolver o problema da mochila apenas quando há uma quantidade pequena de itens (30 ou menor nos testes realizados), ainda mais adequado do que o algoritmo guloso nessa situação devido a garantia de se ter a solução ótima. Já para quantidades grandes de itens, o algoritmo de força bruta é extremamente lento em entregar o resultado, dessa forma o algoritmo guloso surge como uma opção para resolver o problema. Apesar do algoritmo guloso raramente ter apresentado a solução ótima nos testes realizados, ele consegue entregar uma resposta satisfatória de forma rápida, inclusive podendo ainda ser melhorado utilizando métodos de ordenação mais adequados para listas com tamanhos grandes como o *merge sort*, *quicksort* e *shell sort*.

7 REFERÊNCIAS BIBLIOGRAFICAS

Antonio Alfredo Ferreira Loureiro. **Projeto e Análise de Algoritmos: Análise de Complexidade.** Notas de aula, 2010.

Dantzig, Tobias. **Numbers: The Language of Science.** 1930.

G. B. Mathews. **On the Partition of Numbers.** Proceedings of the London Mathematical Society, Volume s1-28, Issue 1, November 1896, Pages 486–490. Disponível em: <https://doi.org/10.1112/plms/s1-28.1.486>

Maya Hristakeva, Dipti Shrestha. **Different Approaches to Solve the 0/1 Knapsack Problem.** Simpson College. Disponível em: <https://pdfs.semanticscholar.org/7647/b5adfd7a326bdc8795a3ab2491700b321ce1.pdf>

Nivio Ziviani. **Projeto de algoritmos: com implementações em Java e C++.** 3 ed. Editora Cengage Learning, 2007.

Thomas Cormen et al. **Introduction to Algorithms.** Editora Prentice Hall, 2006.