# Relatório do Segundo Trabalho - Otimização

Luan Machado Bernardt - GRR20190363

3 de dezembro de 2021

**Resumo**

Neste relatório, é explicada a abordagem, modelagem e a implementação de um algoritmo que resolve o problema da mochila ou *knapsack problem* com restrições, por meio do método de *branch and bound*, bem como uma análise acerca da função limitante e dos resultados obtidos.

## 1 Problema

**Mochila Química.** Um ladrão, que entrou em uma loja de produtos químicos, deseja preencher sua mochila, cujo limite de peso deve ser cumprido, com itens (que possuem peso e valores variados) de maneira que seu ganho seja máximo. No entanto, por se tratarem de produtos químicos, certos produtos não podem estar juntos na mochila, pois podem haver reações químicas. Dados os pesos e valores de cada produto, deve-se retornar uma lista com aqueles que fazem com o ladrão tenha o maior ganho, respeitando as restrições.

O problema apresentado uma versão do *Knapsack Problem* visto na seção 1.4 do livro [1], com a diferença de que nessa versão existem restrições entre produtos, o que diminui consideravelmente o conjunto de possíveis soluções.

## 2 Implementação

Para resolução do problema, foi implementado um algoritmo de *branch and bound*, que é uma versão de *backtracking* com funções limitantes, chamadas de *bounds*, que ajudam na escolha dos melhores caminhos e na eliminação daqueles que não servem para encontrar a melhor solução.

Criou-se um vetor de produtos, estrutura de dados que guarda o valor, o peso e se o produto possui alguma restrição. Além disso, há um vetor de restrições que armazena o índice dos produtos envolvidos. Utiliza-se também um vetor x com tamanho **n** que guarda se na solução parcial, o produto i foi ou não colocado na mochila (1 ou 0).

### 2.1 Entrada

A entrada do programa será recebida na forma:

```
n r c
p1, p2, p3, ... pn
v1, v2, v3, ... vn
r11, r12
r21, r22
...
rx1 rx2
```

Onde, $n$ e $r$ são o número de produtos, de restrições respectivamente, e $c$ é a capacidade da mochila. De *p1* até *pn* e *v1* até *vn* são recebidos os pesos e os valores dos produtos, que serão armazenados em seus respectivos índices no vetor de produtos, e as x linhas seguintes representam as restrições, que serão guardadas no vetor de restrições, onde *ri1* representa o produto 1 da restrição i e *ri2*, o segundo produto.

## 2.2 Saída

A saída do programa será dada na forma:

optP
p1 , p2 , ... , pk

Onde optP é a melhor solução encontrada e p1, p2, ..., pk, são os produtos que devem ser inseridos na mochila para chegar na solução, ou seja, as posições do vetor x que tiverem valor 1.

## 2.3 Modelagem

A modelagem da árvore utilizada foi a binária, ou seja, o conjunto de escolha de todos os nodos, os quais representam os produtos, é 0, 1, onde 1 denota o fato de que o produto foi inserido na mochila e o fluxo seguiu à esquerda, e 0 o contrário. Se analisada na forma ingênua, ou seja, sem funções limitantes, a modelagem se trata simplesmente de um *backtracking*, sendo notável que a altura da árvore será igual ao $n$ (número de produtos) e o número de nodos igual a $2^n$, o que se torna um problema para grandes quantidades de produtos, pois é necessário percorrer todos os nodos para encontrar a solução ótima.

## 2.4 Função Limitante

Para otimizar o algoritmo e transformá-lo efetivamente em *branch and bound*, deve-se definir uma função limitante, que deve atender à dois critérios:

- Ser fácil de calcular.
- Ser próxima da solução ótima.

A função escolhida foi a chamada de *Rational Knapsack*, apresentada na seção 4.6.1 do livro [1], que calcula o ganho máximo da mochila utilizando frações dos produtos para preencher o que resta do limite de peso. A função limitante B(x) é calculada da seguinte maneira:

$$B(x) = curV + rKnap([ph, ph+1, ph+2, ..., pn-1], c - curW)$$

Onde curV é o valor e curW é o peso da solução parcial, sendo:

$$curV = \sum_{i}^{h-1} vi * xi$$

$$curW = \sum_{i}^{h-1} pi * xi$$

Xi representa o valor conjunto escolhido para o nodo i. na solução parcial, ou seja 1 ou 0. rKnap(a, b) calcula o ganho máximo possível no espaço restante b na mochila com os produtos no vetor a que ainda não foram analisados. Para cada nodo percorrido B(x) será calculado, podendo ou não ter valor diferente.

## 2.5 Resolução do Problema

O algoritmo utilizado é uma adaptação do algoritmo 4.9 do livro [1], cujos passos seguidos pelo algoritmo para encontrar a solução ótima, dada uma entrada de n produtos é:

1. Chama a função para i = 0 e peso = 0;
2. Pega o elemento i do vetor de produtos;
3. Se for um nodo folha, ou seja, i = n calcula o valor total obtido pela fórmula curV para h = n;

   Se o valor total for maior do que o encontrado até o momento, optP (que começa em 0), substitui-se optP por curV e altera-se o vetor X para a nova combinação e produtos inseridos ou não;
4. Se não for folha, calcula-se a solução parcial obtida, por meio de curV, com h = i, e soma-se isso a rKnap([i+1, n-1], c - curW), que é a solução racional da mochila com os produtos a partir de i+1 para o peso restante, obtendo-se então B(x);
5. Compara-se B(x) com optP;

   Se B(x) $\leq$ optP, a função retorna, pois não é possível melhorar a solução ótima por esse caminho, ou seja, de todas as combinações possíveis abaixo desse nodo, nenhuma encontraria uma solução mais proveitosa;
6. Se o produto atual puder ser inserido, isto é, cabe na mochila e não possui conflitos com os itens já presentes na mochila, são atribuídos os valores [1, 0] ao conjunto de escolha. Se não puder, seja por qualquer um dos motivos, o conjunto de escolha recebe [0];
7. Chama-se recursivamente o algoritmo para i+1 e para o novo peso, dependendo do caminho tomado conforme o valor do conjunto escolhido, se for 1, segue para o nodo na esquerda e acrescenta-se o peso do produto ao curW, se for 0, segue para a direita e curW permanece igual. E volta ao passo 2;

Ao fim do algoritmo, a variável optP conterá o maior lucro possível, respeitando as restrições impostas, e o vetor x conterá quais valores devem ser inseridos na mochila para a obtenção dessa solução.

## 2.6 Exemplo

Para a entrada a seguir:

```
5  2  26
12  7  11  8  9
24  13  23  15  16
2  4
1  5
```

Se não houvessem restrições, a melhor solução seria 2-3-4 com valor de 51, pois a soma 13 + 23 + 15 = 51 possui dois valores com os menores pesos da entrada, somado a terceiro com o melhor valor dentre os que cabem na mochila, mas está no limite de peso, logo a única maneira de tentar encontrar uma solução melhor seria com um conjunto de dois elementos, preferencialmente os com maiores valores, que nesse caso juntos somam 47. No entanto, existem restrições as quais são tratadas pelo algoritmo, que devolve:

```
47
1  3
```

O retorno do programa, nos mostra que a melhor solução é de fato 47, formada pelos itens 1 e 3, pois 2 4 que faziam parte a possível solução ótima são conflitantes.

## 2.7 Outro Exemplo

Para a seguinte entrada:

```
8  5  10
2  1  3  5  6  2  9  4
1  1  2  6  10  3  100  90
1  2
2  4
5  8
6  3
2  7
```

O programa retorna:

```
100
7
```

É fácil perceber que o sétimo produto é um forte candidato a estar na mochila, pois ele possui o maior peso de todos, se não houvessem restrições a solução ótima também incluiria o item 2, pois ele caberia na mochila, no entanto, como há conflito entre eles, a solução é apenas o 7. Apesar dos itens 5 e 8 caberem na mochila e terem o mesmo valor que o item 7, eles não são a solução retornada, pois a com o item 7 é encontrada antes, e como a segunda alternativa não melhora a solução, ela não é considerada. Se tanto o produto 5 como o 8 estivessem a frente do produto 7, eles estariam na solução retornada.

## 2.8 Análise da Função Limitante

Testando o algoritmo ingênuo e o *branch and bound* com a função limitante explicada acima, para entradas de tamanhos diferentes, obteve-se a seguinte tabela:

| | Ingênuo (nodos) | Ingênuo (tempo em ms) | B&B (nodos) | B&B (tempo em ms) | Cortes |
|---|---|---|---|---|---|
| 5 | 31 | 0 | 21 | 0 | 4 |
| 8 | 255 | 0 | 62 | 0 | 15 |
| 10 | 1023 | 0 | 40 | 0 | 8 |
| 15 | 32767 | 2 | 3948 | 0 | 1528 |
| 24 | 16777215 | 1753 | 4669 | 2 | 1815 |

Figura 1: Testes realizados para n de tamanhos diferentes.

Percebe-se que a diferença de nodos analisados fica cada vez maior, quando o número de produtos aumenta, isso se dá pelo fato da função limitante realizar mais cortes. Vale ressaltar, que o número de cortes não é o número de nodos ignorados pela *bounding function*, mas sim o número de subárvores que foram cortadas da análise, pois $B(x) \leq optP$ no nodo em que as mesma iniciariam. Existem também, cortes realizados por conflitos e pelo limite de peso, no entanto, estes não foram contabilizados, pois não fazem parte da função limitante em si.

Realizando testes, nota-se que muitas vezes a existência de restrições entre produtos, faz com que mais nodos sejam percorridos, pois um caminho classificado como promissor pela limitante, pode ter conflitos em nodos mais baixos da subárvore, logo ela não chega na possível solução apontada pela *bounding function*.

Uma mudança que poderia diminuir a contagem de nodos, seria a ordenação do vetor dos produtos em ordem decrescente por valor/peso, pois assim seriam avaliados primeiro aqueles que possuem um melhor custo benefício e que, consequentemente, poderiam levar a solução de maneira mais rápida, fazendo menos análises e percorrendo menos caminhos.

# Referências

[1] Douglas Stinson Donald Kreher. *Combinatorial Algorithms - Generation, Enumeration and Search*. CRC Press, 1999.

[2] *KNAPSACK_01. Data for the 01 Knapsack Problem*. URL: `https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html`.