# INF05010 Otimização Combinatória
# Trabalho Final:
# Busca Tabu para o Problema da Mochila Conexa

Pedro Salgado Perrone

Novembro de 2018

## 1 Introdução

O objetivo deste trabalho é a implementação de uma meta-heurística para a resolução do Problema da Mochila Conexa (PMC). A meta-heurística implementada foi a Busca Tabu, cujos detalhes serão apresentados na seção 3.

O problema da mochila conexa é definido como: dado um grafo $G = (V, E)$ com pesos $w_i \in R_+$ e valores $v_i \in R_+$ para cada $i \in V$ e uma capacidade máxima $U \in R_+$ para a mochila, encontrar o subconjunto $H \subseteq G$ tal que $\sum_{i \in H} v_i$ seja otimizado, $\sum_{i \in H} w_i$ seja menor ou igual a $U$ e $H$ forme um grafo conexo.

O problema contém o problema da mochila binária, que é conhecidamente $NP$-Completo [3]. Portanto, também é $NP$-Completo.

## 2 Formulação do PMC como Programa Inteiro

Para verificar a conectividade da solução, a ideia é definir um vetor que selecione uma origem entre os ítens selecionados e enviar mensagens a partir deste item, como no problema do fluxo [1]. Sendo $n$ a quantidade de itens que compõem a solução, este item de origem deve enviar $n-1$ mensagens e cada um dos outros itens deve receber uma mensagem a mais do que recebe, fazendo uma expansão em largura. As restrições para que a solução caiba na mochila já foram exploradas [2] e também se encontram presentes.

**Constantes:**

$M$ representando um valor suficientemente grande,
$N$ representando a quantidade de vértices no grafo de entrada,
$G_{ij}, \quad \forall i \in [N], \quad \forall j \in [N]$, representando o grafo da instância, onde:

$$G_{ij} = \begin{cases} 1 & \text{Caso o vértice i incide em j no grafo de entrada} \\ 0 & \text{Caso o contrário} \end{cases},$$

$W_i \in R_+, \quad \forall i \in [N]$ representando o peso do vértice $i$,
$V_i \in R_+, \quad \forall i \in [N]$ representando o valor do vértice $i$ e
$w$ representando a capacidade máxima da mochila.

**Variáveis:**

$S_i, \quad \forall i \in [N]$ onde:

$$S_i = \begin{cases} 1 & \text{Caso o vértice } i \text{ seja parte da solução} \\ 0 & \text{Caso o contrário.} \end{cases},$$

$F_{ij} \in Z_+, \quad \forall i \in [N], \quad \forall j \in [N]$ representando a quantidade de mensagens enviadas pelo vértice $i$ para o vértice $j$ e
$O_i, \quad \forall i \in [N]$ onde:

$$O_i = \begin{cases} 1 & \text{Caso o vértice } i \text{ seja o vértice de origem da verificação de conectividade da solução} \\ 0 & \text{Caso o contrário.} \end{cases}$$

**Função Objetivo:**

$$\max \sum_{i \in [N]} V_i S_i$$

**Restrições:**

$$\sum_{i \in [N]} W_i S_i \leq w \tag{1}$$

$$F_{ij} \leq G_{ij} M, \quad \forall i \in [N], \quad \forall j \in [N] \tag{2}$$

$$F_{ij} \leq S_i M, \quad \forall i \in [N], \quad \forall j \in [N] \tag{3}$$

$$F_{ij} \leq S_j M, \quad \forall i \in [N], \quad \forall j \in [N] \tag{4}$$

$$\sum_{i \in [N]} O_i = 1 \tag{5}$$

$$\sum_{j \in [N]} F_{ij} \geq ((\sum_{n \in [N]} S_n) - 1) - (1 - O_i) M, \quad \forall i \in [N] \tag{6}$$

$$\sum_{j \in [N]} F_{ij} \leq (\sum_{n \in [N]} S_n) - 1, \quad , \forall i \in [N] \tag{7}$$

$$O_i \leq S_i, \quad \forall i \in [N] \tag{8}$$

$$\sum_{i \in [N]} F_{ij} \leq (1 - O_j) M, \quad \forall j \in [N] \tag{9}$$

$$(\sum_{j \in [N]} F_{ij}) + 1 \leq (\sum_{j \in [N]} Fji) + M(O_i + (1 - S_i)), \quad \forall i \in [N] \tag{10}$$

$$(\sum_{j \in [N]} F_{ij}) + 1 \geq (\sum_{j \in [N]} Fji) - M(O_i + (1 - S_i)), \quad \forall i \in [N] \tag{11}$$

$$F_{ii} = 0, \quad \forall i \in [N] \tag{12}$$

(1) Garante que a solução cabe na mochila;
(2) Impede que haja fluxo em arestas que não existem;

(3) e (4) Impede que arestas adjacentes a vértices que não fazem parte da solução tenham fluxo;

(5) Garante que somente um vértice é a origem na verificação da conectividade do grafo;

(6) Define que o vértice de origem emite $n-1$ mensagens, sendo $n$ a quantidade de itens pertencentes à solução;

(7) Define que nenhum vértice emite mais de $n-1$ mensagens, sendo $n$ a quantidade de itens pertencentes à solução. Em função das outras restrições, esta acaba atuando somente sobre o vértice de origem da verificação de conectividade;

(8) Garante que o vértice de origem na verificação da conectividade pertence à solução;

(9) Garante que o vértice de origem não recebe mensagens;

(10) e (11) Definem que todos os itens pertencentes a solução e que não são o vértice de origem na verificação da conectividade recebem uma mensagem a mais do que enviam;

(12) Garante que nenhum vértice envia uma mensagem para si mesmo.

# 3 Busca Tabu

## 3.1 Parâmetros

Para a implementação da meta-heurística foi preciso parametrizar três elementos:

- **tabu_iterations_rate:** ao ser multiplicado pela quantidade de vértices na entrada do problema, determina a quantidade de iterações nas quais um elemento será tabu;
- **iterations_rate:** ao ser multiplicado pela quantidade de vértices na entrada do problema, determina quantas iterações acontecerão na busca por uma solução;
- **random_seed:** serve de semente para a seleção de um item que formará a solução inicial.

## 3.2 Pseudocódigo

**Algorithm 1** Busca Tabu

```
1: melhor_solução ← [ ]
2: solução_atual ← [VérticeAleatório(random_seed)]
3: contador_de_iterações ← 0
4: while contador_de_iterações < quantidade_de_vértices · iterations_rate do
5:     vizinhos ← GeraVizinhos(solução_atual)
6:     OrdernaPor(vizinhos, vizinho → ValorDoSubconjunto(vizinho))
7:     melhor_vizinho ← PrimeiroOnde(vizinhos, vizinho → VizinhoVálido(vizinho))
8:     if not melhor_vizinho = NULL then
9:         solução_atual ← melhor_vizinho.novo_conjunto
10:        if ValorDoSubconjunto(melhor_vizinho) > ValorDoSubconjunto(melhor_solução) then
11:            melhor_solução ← melhor_vizinho.novo_conjunto
12:    AtualizaListaTabu(quantidade_de_vértices · tabu_iterations_rate), melhor_vizinho.item_modificado)
13:    contador_de_iterações ← contador_de_iterações + 1
14: return melhor_solução
```

## 3.3 Solução Inicial

A solução inicial não pode ser completamente aleatória porque existiria a chance de ser gerada uma solução inválida e com todos os vizinhos inválidos, o que impediria o algoritmo de encontrar uma solução. Portanto, o caminho seguido foi sortear um vértice e o colocar na solução inicial. Isso evita que a primeira iteração selecione sempre o item de maior valor.

### 3.4 Geração de vizinhos

A geração de um vizinho de uma solução inverte a seleção de um vértice. Em outras palavras, um vizinho é o resultado da remoção de um vértice que faz parte da solução atual ou da adição de um vértice que não faça. Para cada vértice da instância do problema é gerado um vizinho.

### 3.5 Validade de uma solução

Uma solução somente será válida se o somatório dos pesos de seus vértices não ultrapassar o valor suportado pela mochila e o subgrafo composto pelos vértices de tal solução for conexo. Para verificar a conectividade do subgrafo, foi implementada uma busca em profundidade.

### 3.6 Validade de um vizinho

Um vizinho válido é aquele que representa uma solução válida e cujo item que sofreu a inversão de seleção não se encontra em tabu.

### 3.7 Seleção de um vizinho

O vizinho selecionado será o vizinho válido cujos itens apresentam o maior somatório de valores.

### 3.8 Atualização da Lista Tabu

Quando um vizinho é selecionado, o vértice que sofreu a inversão de seleção é posto em tabu. Conforme descrito em 3.6, enquanto ele estiver nesta situação, todos os vizinhos gerados por novas inversões da sua seleção serão considerados inválidos. A quantidade de iterações nas quais o vértice permanecerá em tabu é o produto entre a quantidade de vértices na instância e o parâmetro *tabu_iterarions_rate*. Ao fim de cada iteração também é necessário decrementar a quantidade de iterações remanescentes em tabu dos elementos presentes na lista.

### 3.9 Melhor solução

A busca tabu é uma meta-heurística que aceita pioras. Assim sendo, é necessário diferenciar a solução atual da melhor solução já encontrada. Os novos vizinhos serão gerados a partir da solução inicial, mesmo que ela tenha um valor pior do que a melhor solução já encontrada.

## 4 Implementação

### 4.1 Plataforma de implementação

O trabalho foi implementado e testado no sistema operacional MacOS High Sierra (versão 10.13.6) 64 bit, com um processador Intel(R) Core(TM) i7-7700HQ com quatro núcleos físicos e oito virtuais de 2.8GHz, com cache L3 de 6MB e 16GB de memória. A linguagem utilizada foi Kotlin com o compilador *kotlinc* na versão 1.3.10. A versão utilizada do Java foi a 1.8.0_162.

### 4.2 Estruturas de dados utilizadas

O grafo da instância do problema é representado na implementação por uma matriz de adjacência. Ela foi implementada como um array de arrays do tipo booleano. Os pesos e os valores dos itens são arrays de floats e as variáveis que representam subconjuntos, como um vizinho, são arrays de booleanos. Neste última representação temos *true* na posição $i$ se o elemento de índice $i$ faz parte do subconjunto e *false* caso o contrário.

### 4.3 Paralelização

Uma versão inicial do algoritmo fazia a verificação de validade completa dos vizinhos em oito threads separadas. Porém, verificou-se que a operação mais custosa do laço principal era verificar a conectividade dos grafos vizinhos. Tendo isso em vista, foi preferido primeiro remover todos os vizinhos em tabu ou que extrapolavam o peso suportado pela mochila e verificar a conectividade sequencialmente do primeiro vizinho até o último. No cenário ideal, o grafo do primeiro vizinho é conexo e ele pode ser selecionado, evitando a computação da verificação de conectividade de todos os outros subgrafos. Com esta ideia, se resolveu remover o paralelismo da implementação.

## 5 Teste de parâmetros

### 5.1 Teste do parâmetro *iterations_rate*

Os testes com este parâmetro foram realizados com os valores inteiros de 1 a 5 em 10 instâncias diferentes. Fixou-se os parâmetros *tabu_iterations_rate* e *random_seed* em 0.1 e 2345, respectivamente. O gráfico na imagem 1 expõe a média dos resultados obtidos para cada instância.

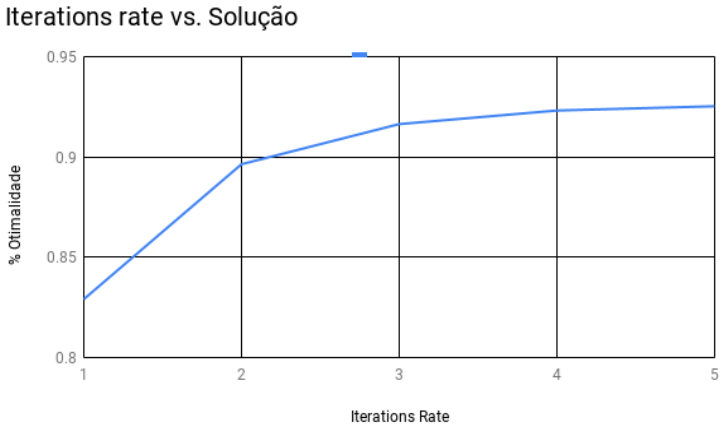


Figura 1: Variação do parâmetro *iterations_rate*

Pode-se observar que a diferença no percentual da otimalidade fica bastante pequena em incrementos no valor deste parâmetro a partir de 4. Como este parâmetro simplesmente define a quantidade de iterações, a variação do tempo de execução em função dele é trivial.

### 5.2 Teste do parâmetro *tabu_iterations_rate*

Os testes com esse parâmetro foram realizados com valores entre 0.05 e 0.15 com variações de 0.025 entre eles. Fixou-se os parâmetros *iterations_rate* e *random_seed* em 4 e 2345, respectivamente. Os resultados se encontram nas imagens 2 e 3.

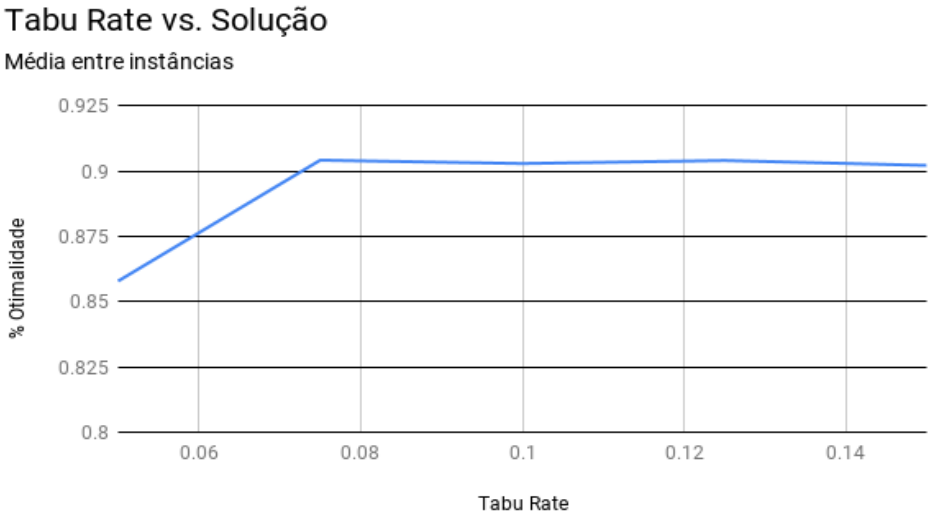


Figura 2: Resultado médio da variação do parâmetro *tabu_iterations_rate*

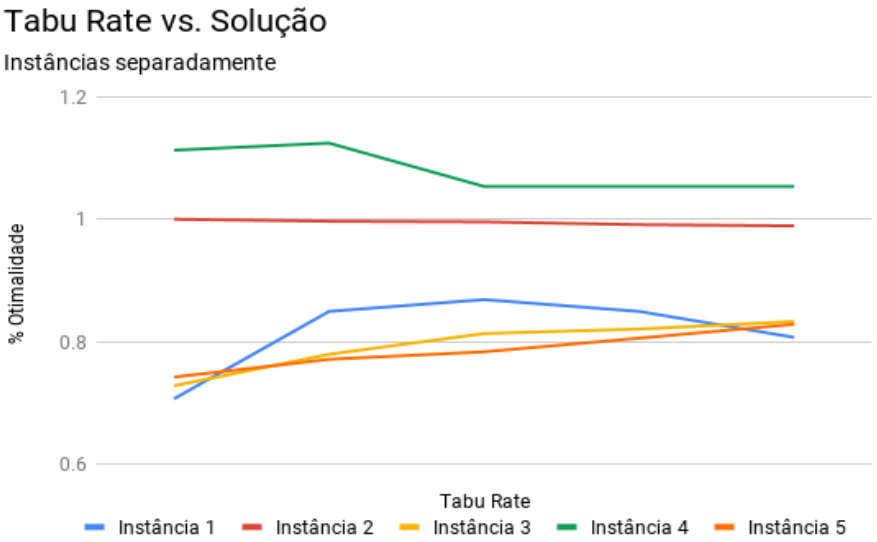


Figura 3: Resultados individuais da variação do parâmetro *tabu_iterations_rate*

No caso deste parâmetro foi interessante inserir não somente um gráfico das médias, mas também das instâncias separadamente. Com isso, se pode mostrar que existe um comportamento médio de melhorar o resultado em relação a otimalidade enquanto o valor de *tabu_iterations_rate* cresce até que atinja 0.75. Depois disso, há uma estabilização no resultado com apenas leves oscilações. Apesar disso, não há um padrão no comportamento de uma instância em específico. As cinco que foram apresentadas demonstram diversos tipos de comportamento. A *Instância 1*, por exemplo, apresenta um formato similar a uma parábola invertida, com o ponto de maior valor em 0.1. Enquanto isso, a *Instância 2* apresenta um comportamento quase constante e a *Instância 5* um crescimento quase linear. Não foi encontrado nenhuma característica que se relacionasse com o comportamento da variação do resultado.

Outro aspecto que vale ser analizado nos testes de variação deste parâmetro é o tempo de execução. No gráfico da imagem 4 está a média normalizada dos tempos de execução das instâncias descritas acima.

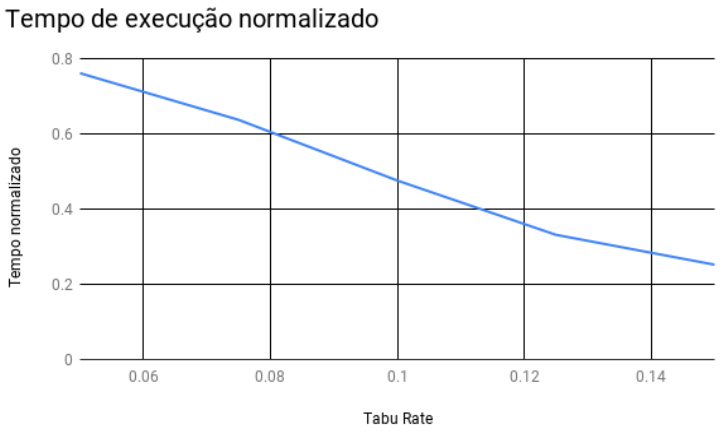


Figura 4: Variação do tempo de execução de acordo com o parâmetro *tabu_iterations_rate*

A explicação encontrada é que como o filtro que descarta vizinhos que estão em tabu é o primeiro dos filtros a ser executado e, no geral, os elementos ficarem mais iterações em tabu é sinônimo de ter menos vizinhos válidos por iteração, há menos computação a ser executada nos outros filtros, que são mais custosos computacionalmente. A tabela 1 apresenta a quantidade de vizinhos que não foram eliminados pelo filtro do tabu com o menor e o maior valor analizados para o parâmetro em questão.

Tabela 1: Vizinhos não invalidados pela lista tabu

| Instância | Vizinhos não descartados com tabu_iterations_rate igual a 0.05 | Vizinhos não descartados com tabu_iterations_rate igual a 0.15 |
|---|---|---|
| 1 | 63818 | 47180 |
| 2 | 552528 | 477891 |
| 3 | 576094 | 374265 |
| 4 | 385664 | 293442 |
| 5 | 3470977 | 3399072 |

# 6 Testes das instâncias

Para a confecção deste relatório foram feitos e documentados testes sobre as 10 instâncias disponíveis em `http://www.inf.ufrgs.br/~mrpritt/oc/moc.zip`. A tabela 2 mostra informações sobre elas.

Tabela 2: Valores referentes a cada instância

| Instância | Quantidade de Vértices | Quantidade de Arestas | Capacidade da Mochila | BKV |
|---|---|---|---|---|
| moc01 | 500 | 12000 | 5058.57 | 259 |
| moc02 | 500 | 2000 | 75 | 67314.1 |
| moc03 | 1000 | 10000 | 150.15 | 2035 |
| moc04 | 1000 | 2000 | 5031.19 | 19625.2 |
| moc05 | 2000 | 8000 | 15429.9 | 4575 |
| moc06 | 2000 | 20000 | 9994.93 | 102206 |
| moc07 | 5000 | 100000 | 1250.61 | 2216 |
| moc08 | 5000 | 20000 | 25048.1 | 170769 |
| moc09 | 10000 | 40000 | 5011.71 | 5186 |
| moc10 | 10000 | 200000 | 150.36 | 144262 |

### 6.1 Execução da meta-heurística

Os resultados obtidos pela execução da meta-heurística para cada uma das instâncias da tabela 2 se encontram na tabela 3. Para as execuções os parâmetros escolhidos foram: 0.1 para *tabu_iterations_rate*, 5 para *iterations_rate* e 2345 para *random_seed*.

Tabela 3: Resultados da execução da meta-heurística

| Instâncias | Valor Referência | Resultado Obtido | Desvio para Referência (%) | Desvio para Solução Inicial | Tempo de Execução (milissegundos) |
|---|---|---|---|---|---|
| moc01 | 259 | 225 | 13.12 | -7400 | 1570 |
| moc02 | 67314.1 | 67030.5 | 0.42 | -13035.5 | 6079 |
| moc03 | 2035 | 1655 | 18.67 | -12630.76 | 12809 |
| moc04 | 19625.2 | 20936.373 | -6.68 | -2244.16 | 15913 |
| moc05 | 4575 | 3593 | 21.46 | -51228.57 | 477700 |
| moc06 | 102206 | 84602.29 | 17.22 | -9372.58 | 74164 |
| moc07 | 2216 | 2169 | 2.12 | -6472.72 | 798208 |
| moc08 | 170769 | 158919.5 | 6.93 | -48263.33 | 3919670 |
| moc09 | 5186 | 5356 | -3.27 | -6353.01 | 9858373 |
| moc10 | 144262 | 137455.9 | 4.71 | -19917.72 | 10637117 |

### 6.2 Execução com o solver GLPK

A versão da linguagem Julia utilizada foi 1.0.2 e do solver 4.52. Foi estabelecido um tempo limite de 1 hora para o processamento de cada instância. Os resultados se encontram na tabela 4. Instâncias com o resultado obtido igual a 0 não convergiram para um valor dentro do limite de tempo estabelecido.

Tabela 4: Resultados da execução do uso do solver GLPK

| Instâncias | Resultado Obtido | Tempo de Execução |
|---|---|---|
| moc01 | 259 | 4 minutos e 12 segundos |
| moc02 | 0 | 1 hora |
| moc03 | 0 | 1 hora |
| moc04 | 0 | 1 hora |
| moc05 | 0 | 1 hora |
| moc06 | 0 | 1 hora |
| moc07 | 0 | 1 hora |
| moc08 | 0 | 1 hora |
| moc09 | 0 | 1 hora |
| moc10 | 0 | 1 hora |

Os experimentos sustentam que devido ao rápido crescimento da quantidade de restrições em função do tamanho da entrada para o programa inteiro modelado em 2, o uso do solver se torna inviável para a maioria das instâncias fornecidas para teste.

## 7 Conclusão

Considera-se que o algoritmo teve uma performance razoável, com um fator médio de otimalidade de 91%. Em algumas instâncias, inclusive, foi encontrado um conjunto de valor superior ao melhor valor conhecido. Para entradas pequenas, a execução é rápida. O problema é que como tanto iterações quanto geração de vizinhos têm a sua cardinalidade vinculada ao tamanho da entrada, a geração total de vizinhos se dá de forma quadrática. Com isso, entradas maiores podem levar horas executando.

Depois de testar os parâmetros individualmente, recomenda-se a seguinte configuração para o uso: $iteration\_rate = 5$ e $tabu\_iterations\_rate = 0.1$. Contudo, vale ressaltar que o teste com diferentes valores para *tabu_iterations_rate* e diferentes sementes é incentivado.

A codificação do algoritmo não apresentou maiores dificuldades. Várias otimizações puderam ser feitas no decorrer do desenvolvimento.

Tendo em vista os pontos citados acima e que o uso do solver vira impraticável com aumentos da entrada, considera-se que o projeto, como um todo, foi bem sucedido.

# Referências


[1] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. 1972.

[2] Fred Glover. Heuristics for integer programming using surrogate constraints. 1977.

[3] Silvano Martello and Paolo Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. 1981.