# Trabalho 2 de Otimização

Carlos Alberto Pedroso Junior

Fevereiro 2021

## 1 Introdução

Podemos definir o problema de otimização combinatória sendo aquele que visa combinar conjuntos finitos para se obter um resultado, seja ele de minimização ou de maximização. A combinação desses conjuntos ajudam na busca para encontra uma solução ótima em diversos problemas, combinando os elementos de cada conjunto para chegar-se a um resultado ótimo. Um exemplo clássico de otimização combinatória é o problema da mochila, onde busca-se preencher a mochila utilizando seu peso máximo, com um conjunto de itens onde cada 'item' tem um valor e um peso definido, assim deseja-se escolher os itens de maior valor e com menor peso. Existem diversas formas de resolver problema de combinatória, seja utilizando algoritmos de busca como gulosos, *Backtracking*, programação dinâmica ou *Branch and Bound*. Segundo [2], o método denominado de *Branch and Bound*, baseia-se na ideia de desenvolver uma enumeração inteligente dos pontos candidatos à solução ótima inteira de um problema.

O termo *branch* refere-se ao fato de que o método efetua partições no espaço das soluções. O termo *bound* ressalta que a prova da ótima da solução utiliza-se de limites calculados ao longo da enumeração dos ramos. De acordo com [1] o *Branch and Bound* pode ser definido como uma enumeração de todas as soluções viáveis de um problema de otimização combinatorial de minimização ou maximização, tal que propriedades ou atributos não compartilhados por qualquer solução ótima são detectados previamente. Um atributo, chamado também de ramo da árvore de enumeração, define um subconjunto do conjunto de todas as soluções viáveis do problema original onde cada elemento do subconjunto satisfaz este atributo.

Assim, o presente trabalho busca resolver por meio de *branch and bound* o problema **Elenco Representativo** de otimização combinatória que visa encontrar um conjunto de atores de forma a satisfazer a condição de minimização do custo e representatividade de grupos da sociedade. Para isso, definimos o problema, propomos uma modelagem e implementamos uma solução para validação.

## 2 Problema

Seguindo as informações do trabalho proposto, define-se o problema de escolha de elenco representativo. Sendo composto de um conjunto de grupos $S$ que devem ser representados por um conjunto de atores $A$ que devem preencher um conjunto requerido de personagem $P$. O problema está em escolher os atores mais baratos minimizando o custo, de forma que todos os grupos estejam representados pelos atores escolhidos. Uma forma de olhar para o problema e tentar resolve-ló está no uso de análise combinatória, pois necessitamos satisfazer duas condições em relação aos atores escolhidos, representatividade e custo. Os formatos de entrada para o problema declara três números representados $l = |S|$, $m = |A|$ e $n = |P|$, seguidos de $v$ correspondendo ao valor do ator e $s$ definindo o número de grupos da sociedade que ele representa.

## 3 Modelagem

A modelagem visa criar um modelo matemático para explicar e compreender as formas de resolver o problema proposto satisfazendo todas as condições e restrições atribuídas. Dessa forma, para melhor modelar o problema, foi criada uma variável binaria $x_a$, onde é 1 se a $\in$ X e 0 caso contrário. A função objetivo de minimização $min$ de custos e as restrições são definidas e explicadas abaixo:

$$min : \sum_{a \in A} v_a . x_a \tag{1}$$

$$sujeito : \sum_{a \in A} x_a = n$$

$$\sum_{a \epsilon A : g \in s_a} x_a \geq 1 \ \forall g \in S$$

$$x_a \in \{0, 1\}$$

A Equação 1 define a função objetivo para minimização do custo da escolha dos atores para conjunto de personagens. O somatório reapresenta o cálculo do conjunto de atores que satisfazem a condição da variável binária $x_a$ sendo representa apenas por 0 e 1 caso o atore seja escolhido. A primeira restrição está relacionadas com o tamanho do conjunto de personagens a ser escolhido, onde o somatório dos atores escolhidos deve ser igual ao número de personagens requeridos. A segunda restrição está relacionada a cobertura dos grupos da sociedade, onde a soma dos atores escolhidos deve cobrir todos os elementos do grupo da sociedade de forma que todos estejam representados. Outras restrições estão relacionas aos valores de entrada, sendo que $a$ deve representar pelo menos um elemento de $S$, ou seja, não pode ser 0 ou valores negativos. Todos os grupos da sociedade $S$ devem estar representados pelos elementos do conjunto $A$, ou seja, não pode faltar a cobertura de nenhum grupo pelos atores na entrada.

## 3.1 Funções limitantes

As funções limitantes estão relacionadas a poda da árvore binaria gerada, elas tem como objetivo encontrar pontos ótimos de corte, definindo que deste ponto em diante não será possível encontrar melhor solução do que já obtida. Assim, diminuem o custo de processamento pelo algoritmo. Segundo [3] as funções *bounding* são métodos sofisticados de poda que auxiliam os algoritmos a serem mais precisos e dinâmicos. Para o algoritmo proposto foram definidas duas funções de *bound*, uma *default* e outra segundaria (-a) explicadas abaixo.

A Equação 2 define a função *bound default*, ela está relacionado com custo do *cast* escolhido até o momento, onde eu tenho o somatório dos valores dos atores já escolhidos $v_i$ até o momento considerando o tamanho de $n$. Desta forma, eu verifico o ramo e vejo se a solução viável que tenho em mãos é mais barata do que próxima, caso seja eu mantenho ela, caso contrária eu troco.

$$B_a = \sum_{i=1}^{n} v_i \tag{2}$$

A Equação 3 define a função de *bound* segundaria (-a), ela também utiliza como parâmetro o custo do *cast* atual escolhido. de além de olhar para frente, porém varrendo apenas o próximo elemento da árvore $nextv_ai$ e verificando seu custo $v_a$ em relação o que já possui. Desta forma, são verificadas as condições do ramo olhando apenas para próximo nó, evitando uma maior descida na árvore e consequentemente reduzindo o custo de execussão e criação de nós.

$$B_b = \sum_{i=1}^{n} v_a - nextv_ai \tag{3}$$

Utilizamos um valor de $OptV$ definido na Equação 4, para auxiliar nas funções de *bound* nesse caso optou-se por inicializar com o valor da soma de todos elementos do conjunto de atores. Essa escolha se deu pelo fato de escolher uma solução viável para fins de comparação.

$$OptV = \sum_{i=1}^{m} \tag{4}$$

Além das duas funções de *bound* elaboradas foi elaborado uma função de corte visando auxiliar a *branch*. Essa função descrita na Equação 5 utiliza a verificação do nível atual $n_a$ somando o valor dos atores escolhidos *cast* menos o nível do nó $n_nivel$. O objetivo de auxiliar na otimização da função de *branch*, e nos *bounds* também, evitando descidas desnecessárias na árvore, porém, olhando apenas para nível. Todas as funções definidas acima tem o intuído de melhorar a otimização do algoritmo proposto.

$$\sum cast + (n_a - n_nivel) \tag{5}$$

Como foi seguida uma modelagem binária, é possível estimar o tamanho total da árvore pelo número de atores do conjunto $A$. Assim, a construção da

árvore binária $2 * (2^n) - 1$ onde $n$ é o número de atores, o -1 é referente a não contabilização da raiz, e a multiplicação por 2 defino os ramos de 0 e 1.

# 4 Exemplos

Os exemplos descritos abaixo objetivam validar a modelagem proposta, através da atribuição da função objetivo e suas restrições. Desta maneira, foi aplicada a modelagem nos exemplos 1 e 2 descritos no trabalho. As entradas estão explicitas da seguinte maneira: na primeira linha têm-se os grupos $l$, depois número de atores $m$ e personagens $n$. Após temos, valor do ator $v_a$ e grupos que ele representa $s$. Segundo, temos a aplicação da função objetivo de minimização $min$ e as restrições $Sujeito$ nos exemplos apresentados. E em terceiro, obtemos os resultados, em que temos a primeira linha os atores escolhidos e baixo o custo desses atores.

Exemplo 1:

2 3 2
10 2
1
2
20 1
2
5 2
1
2

$$S = \{1, 2\}\,;\ A = \{a_1, a_2, a_3\}\,;$$

$$Sa_1 = \{1, 2\}\,;\ Sa_2 = \{2\}\,;\ Sa_3 = \{1, 2\}\,;$$

$$va_1 = 10;\ va_2 = 20;\ va_3 = 5;$$

$$min = 10.x_a1 + 20.x_a2 + 5.x_a3$$

$$sujeito : x_a1 + x_a2 + x_a3 = 2$$

$$1 \in S \rightarrow x_a1 + x_a3 \geq 1$$

$$2 \in S \rightarrow x_a1 + x_a2 + x_a3 \geq 1$$

$$x_a1 = 1, x_a2 = 0, x_a3 = 1$$

$$x = [1, 0, 1]$$

$$z = 10 * 1 + 20 * 0 + 5 * 1 = 15$$

$$1 + 0 + 1 = 2$$

$$1 + 1 \geq 1$$

$$1 + 0 + 1 \geq 1$$

Resultados exemplo 1:
bound default começando pelo 0:
1 3
15
Duração do Programa: 0.000158s
Numero de nós gerados: 12
Arvore geradora: 14

bound default começando pelo 1:
1 3
15
Duração do Programa: 0.000416s
Numero de nós gerados: 8
Arvore geradora: 14

bound (-a) começando pelo 0:
1 3
15
Duração do Programa: 0.000158s
Numero de nós gerados: 8
Arvore geradora: 14

bound (-a) começando pelo 1:
1 3
15
Duração do Programa: 0.000121s
Numero de nós gerados: 6
Arvore geradora: 14

Exemplo 2:
3 3 2
10 2
1
2
20 1
2
3
5 2
1
2

$$S = \{1, 2, 3\}\,;\ A = \{a_1, a_2, a_3\}\,;$$

$$Sa_1 = \{1, 2\}\,;\ Sa_2 = \{2\}\,;\ Sa_3 = \{3\}\,;$$

$$va_1 = 10;\ va_2 = 20;\ va_3 = 5;$$

$$min = 10.x_a1 + 20.x_a2 + 5.x_a3$$

$$sujeito : x_a1 + x_a2 + x_a3 = 2$$

$$1 \in S \rightarrow x_a1 + x_a3 \geq 1$$

$$2 \in S \rightarrow x_a1 + x_a2 + x_a3 \geq 1$$

$$3 \in S \rightarrow x_a1 + x_a2 + x_a3 \geq 1$$

$$x_a1 = 1, x_a2 = 0, x_a3 = 1$$

$$x = [1, 0, 1]$$

$$z = 10 * 1 + 20 * 0 + 5 * 1 = 15$$

$$1 + 0 + 1 = 2$$

$$1 + 1 \geq 1$$

$$1 + 0 + 1 \geq 1$$

Resultados exemplo 2:
bound default começando pelo 0:
1 3
15
Duração do Programa: 0.000416s
Numero de nós gerados: 10
Arvore geradora: 14

bound default começando pelo 1:
1 3
15
Duração do Programa: 0.000416s
Numero de nós gerados: 8
Arvore geradora: 14

bound (-a) começando pelo 0:
1 3
15
Duração do Programa: 0.000289s
Numero de nós gerados: 12
Arvore geradora: 14

bound (-a) começando pelo 1:
1 3
15
Duração do Programa: 0.000160s
Numero de nós gerados: 6
Arvore geradora: 14

# 5 Implementação

O modelo de algoritmo foi implementado na linguagem Python versão 3.6. Toda a implementação e testes foram conduzidos no sistema Linux Ubuntu Mate 20.04 LTS, com 20 GB de memória e processador core i7 sétima geração. Para criação do algoritmo não foi utilizada nenhuma biblioteca auxiliar. Os aquivos para excussão estão todos no diretório raiz, os arquivos de teste utilizados estão contidos no (tar.gz) no subdiretório exemplos. Já o código para execução chamado *elenco* e o arquivo *makefile* encontram-se no diretório raiz.

## 5.1 Experimentação

Com objetivo de avaliar às duas funções limitantes de *bounds*, alguns experimentos foram conduzidos, visando entender os ganhos e perdas de empregar os *bounds*. Além disso, foram desenvolvidos 13 exemplos para realização dos experimentos com às duas funções de *bound*, além da função de corte. Foram utilizadas as métricas, *numero de nós visitados*, e *tempo de execução* do algoritmo, além de avaliar *o tamanho da árvore binária* e *numero de nós não percorridos*. As Tabelas criadas contam com exemplos (EX), entrada de grupos, atores e personagens (Entrada), arvore binária (Arvore), tempo de excussão (Tempo), nós percorridos (Nós), nós não percorridos (Não percorridos), atores escolhidos (Escolhidos) e custo dos atores escolhidos (Custo). Inicialmente testou-se o *bound default* com os exemplos do 1 ao 13 onde iniciamos a construção da árvore a partir do 0 e depois com a construção a partir do 1. Os resultados obtidos podem ser observados na Tabela 1 e Tabela 2, destaca-se que os exemplos **4** e **7** são inviáveis e por isso não apresentam nenhum valor. Além disso, destaca-se que por padrão optou-se por utilizar no código apenas inicialização da árvore binária pelo 1, pois demostrou melhores resultados.

A Tabela 1 apresenta os resultados considerando olhar para o 0 primeiro, nota-se que cobertura da árvore está diretamente relacionada a posição dos melhores atores, aqueles que representam menor custo e cobrem o maior número de grupos. Com exemplos menores o tempo de excussão torna-se mais rápido, porém percorremos quase toda a árvore para obter o resultado ótimo. Nos exemplos com maiores quantidades de atores caminhamos menos na árvore para achar a solução ótima, isso acontece devido a encontrar soluções viáveis mais facilmente, visto que temos mais combinações possíveis. A Tabela 2 apresenta os resultados considerando olhar primeiro para 1, observa-se que tivemos melhor relutados em relação ao número de nós percorridos, porém o tempo de execução tem um aumento considerável em relação os resultados da Tabela 1. Além disso, como apresentado na coluna escolhidos vemos que existe uma diferença entre os atores **Escolhidos**, entretanto isso não afeta encontrar a solução ótima de menor custo.

Os testes com a segunda função de bound (-a) também considerou os 13 exemplos a inicialização da árvore primeiro com 0 e depois com 1. Os resultados obtidos para ambos os experimentos estão sumarizados nas Tabela 3 e 4. A Tabela 3 considera os resultados olhando para o 0 primeiro, observa-se que

Table 1: Bound default iniciando com 0

| EX | Entrada | Árvore | Tempo | Nós | Não percorridos | Escolhidos | Custo |
|---|---|---|---|---|---|---|---|
| 1 | 2 3 2 | 14 | 0.000054s | 10 | 4 | 1 e 3 | 15 |
| 2 | 3 3 2 | 14 | 0.000160s | 10 | 4 | 1 e 3 | 15 |
| 3 | 3 3 2 | 14 | 0.000127s | 10 | 4 | 1 e 3 | 30 |
| 4 | 2 3 2 | | | | 0 | | |
| 5 | 5 5 2 | 62 | 0.000245s | 28 | 34 | 1 e 2 | 15 |
| 6 | 2 3 2 | 14 | 0.000096s | 10 | 4 | 2 e 3 | 25 |
| 7 | 2 2 3 | | | | 0 | | |
| 8 | 2 3 2 | 14 | 0.000263s | 10 | 4 | 1 e 3 | 15 |
| 9 | 5 5 4 | 62 | 0.000136s | 28 | 34 | 1, 2, 4 e 5 | 65 |
| 10 | 4 5 4 | 62 | 0.000095s | 24 | 38 | 1, 2, 4 e 5 | 35 |
| 11 | 5 7 3 | 254 | 0.003371s | 110 | 144 | 2, 5 e 6 | 30 |
| 12 | 5 10 4 | 2046 | 0.001585s | 374 | 1672 | 10, 2, 8 e 9 | 35 |
| 13 | 5 10 4 | 2046 | 0.001260s | 402 | 1644 | 10, 3, 7 e 9 | 35 |

Table 2: Bound default iniciando com 1

| EX | Entrada | Árvore | Tempo | Nós | Não percorridos | Escolhidos | Custo |
|---|---|---|---|---|---|---|---|
| 1 | 2 3 2 | 14 | 0.000416s | 8 | 6 | 1 e 3 | 15 |
| 2 | 3 3 2 | 14 | 0.000416s | 8 | 6 | 1 e 3 | 15 |
| 3 | 3 3 2 | 14 | 0.000255s | 10 | 4 | 1 e 2 | 30 |
| 4 | 2 3 2 | | | | 0 | | |
| 5 | 5 5 2 | 62 | 0.000255s | 22 | 40 | 1 e 2 | 15 |
| 6 | 2 3 2 | 14 | 0.000255s | 10 | 4 | 2 e 3 | 25 |
| 7 | 2 2 3 | | | | 0 | | |
| 8 | 2 3 2 | 14 | 0.000255s | 8 | 6 | 1 e 2 | 15 |
| 9 | 5 5 4 | 62 | 0.000593s | 28 | 34 | 1, 2, 4 e 5 | 65 |
| 10 | 4 5 4 | 62 | 0.000286s | 24 | 38 | 1, 2, 4 e 5 | 35 |
| 11 | 5 7 3 | 254 | 0.001908s | 96 | 158 | 1, 2 e 3 | 30 |
| 12 | 5 10 4 | 2046 | 0.003939s | 336 | 1710 | 1, 2, 6 e 8 | 35 |
| 13 | 5 10 4 | 2046 | 0.002547s | 360 | 1686 | 1, 2, 3 e 7 | 35 |

percorremos quase toda árvore com exemplos menores, quando olhamos para exemplos com maiores possibilidades tem um bom desempenho percorrendo menos nós. Isso ocorre por que esse *bound* olha apenas para o próximo nó no ramo e assim torna-se necessário investigar outras possibilidades de solução viável e consequentemente percorre-se mais nós. A Tabela 4 apresenta os resultados considerando olhar primeiro para o 1, assim, como pode ser observado obtivemos um ganho muito superior em resultados de nós percorridos. Esse resultado além de percorrer menos nós também apresentam um menor custo quando os mel-

Table 3: Bound (-a) iniciando com 0

| Ex | Entrada | Árvore | Tempo | Nós | Não percorridos | Escolhidos | Custo |
|---|---|---|---|---|---|---|---|
| 1 | 2 3 2 | 14 | 0.000302s | 10 | 4 | 1 e 3 | 15 |
| 2 | 3 3 2 | 14 | 0.000226s | 10 | 4 | 1 e 3 | 15 |
| 3 | 3 3 2 | 14 | 0.000044s | 10 | 4 | 1 e 2 | 30 |
| 4 | 2 3 2 | | | | 0 | | |
| 5 | 5 5 2 | 62 | 0.000894s | 28 | 34 | 1 e 2 | 15 |
| 6 | 2 3 2 | 14 | 0.000654s | 10 | 4 | 2 e 3 | 25 |
| 7 | 2 2 3 | | | | 0 | | |
| 8 | 2 3 2 | 14 | 0.000254s | 10 | 4 | 1 e 3 | 15 |
| 9 | 5 5 4 | 62 | 0.000564s | 28 | 34 | 1, 2, 4 e 5 | 65 |
| 10 | 4 5 4 | 62 | 0.000591s | 28 | 34 | 1, 2, 4 e 5 | 35 |
| 11 | 5 7 3 | 254 | 0.001698s | 110 | 144 | 2,5 e 6 | 30 |
| 12 | 5 10 4 | 2046 | 0.003696s | 658 | 1388 | 10, 2, 8 e 9 | 35 |
| 13 | 5 10 4 | 2046 | 0.010460s | 658 | 1388 | 10, 3, 7 e 9 | 35 |

Table 4: Bound (-a) iniciando com 1

| EX | Entrada | Árvore | Tempo | Nós | Não percorridos | Escolhidos | Custo |
|---|---|---|---|---|---|---|---|
| 1 | 23 2 | 14 | 0.000378s | 6 | 8 | 1 e 3 | 15 |
| 2 | 3 3 2 | 14 | 0.000127s | 6 | 8 | 1 e 3 | 15 |
| 3 | 3 3 2 | 14 | 0.000113s | 10 | 4 | 1 e 2 | 30 |
| 4 | 2 3 2 | | | | 0 | | |
| 5 | 5 5 2 | 62 | 0.000295s | 28 | 34 | 1 e 2 | 15 |
| 6 | 2 3 2 | 14 | 0.000067s | 10 | 4 | 2 e 3 | 25 |
| 7 | 2 2 3 | | | | 0 | | |
| 8 | 2 3 2 | 14 | 0.000221s | 6 | 8 | 1 e 3 | 15 |
| 9 | 5 5 4 | 62 | 0.000434s | 28 | 34 | 1, 2, 4 e 5 | 65 |
| 10 | 4 5 4 | 62 | 0.000322s | 28 | 34 | 1, 2, 4 e 5 | 35 |
| 11 | 5 7 3 | 254 | 0.001879s | 110 | 144 | 1,2 e 3 | 30 |
| 12 | 5 10 | 2046 | 0.010245s | 658 | 1388 | 1, 2, 6 e 8 | 35 |
| 13 | 5 10 4 | 2046 | 0.006572s | 658 | 1388 | 1, 2, 3 e 7 | 35 |

hores nós estão em primeiro lugar, entretanto a depende a posição dos melhores nós percorre-se mais nós na árvore e tem maior aumento de processamento.

Em comparação entre às duas funções existe uma relação de *trade off* muito interessante, em casos onde os melhores atores (aqueles que tem menor custo e cobrem um grande número de grupos) estejam posicionados no incio o *bound (-a)* com 1 tem melhores resultados para exemplos com menos atores no conjunto $A$. O *bound default* com 1 apresenta melhor resultado com exemplos maiores.

Desta-se também, que ambas as funções quando inicializadas com 0 percorrem uma maior número de nós, porém apresentam melhores resultados em tempo de excussão. Outro ponto que fica evidente com as funções limitantes é o auxilio da função de corte pelo nível, quando utilizada ela evita percorrermos muito ramos que são piores que a solução viável que já temos.

# 6 Conclusão

Este trabalho abordou o problema combinatório de Elenco Representativo para minimizar os custo de escolher um número de atores que representem todos os grupos da sociedade por meio do uso de *branch and bound*. Para isso foi elaborada uma modelagem, e implementada em Python para validação do modelo proposto. Além disso, foram desenvolvidas duas funções limitantes para o algoritmo e comparadas para verificar sua eficácia. Os resultados obtidos demonstraram que ambas as funções limitantes mesmo apresentam bons resultados para os casos avaliados e também uma função desenvolvida olhando apenas para o nível atual eliminando a necessidade de percorrer boa parte da árvore para encontrar uma solução ótima.

# References


[1] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. Scheduling on one processor. In *Scheduling Computer and Manufacturing Processes*, pages 73–137. Springer, 1996.

[2] Marco Goldbarg. *Otimização combinatória e programação linear-2a ediç*, volume 2. Elsevier Brasil, 2005.

[3] Donald L Kreher and Douglas R Stinson. Combinatorial algorithms: generation, enumeration, and search. *ACM SIGACT News*, 30(1):33–35, 1999.