
**UNIVERSIDADE FEDERAL DO PARANÁ**

Carlos Iago Bueno [GRR20190171]
Gabriel Lüders [GRR20190172]

# MODELAGEM DO PROBLEMA DO ELENCO REPRESENTATIVO

**CURITIBA**
**2022**

## RESUMO

Com o objetivo de dar voz a diferentes camadas da sociedade ou apenas de evitar polêmicas de representatividade de elenco, uma produtora quer garantir que os diferentes grupos sociais estejam representados no elenco de sua produção. Dentre os atores que a produtora dispõe, há inúmeros possíveis grupos que cobririam os grupos sociais em sua elencagem, mas estes grupos acabariam sendo muito custosos. Chamamos de problema do elenco representativo, o desafio de achar o grupo viável de menor custo dentre todas as possibilidades.

Neste trabalho, iremos construir uma função recursiva baseada no método *branch-and-bound* para escolher o subconjunto **ótimo** de atores, dentre as soluções candidatas. Ou seja, encontrar o elenco de menor custo que cubra todos os diferentes grupos sociais em sua elencagem.



**Palavras-chave:** Representatividade, Branch and Bound, Modelagem, Otimização.

## SUMÁRIO

# INTRODUÇÃO

Conforme as pautas identitárias têm ganhado força, diversas empresas e instituições começaram a incluir as palavras “inclusão” e “diversidade” em seus vocabulários. Em parte, isto é consequência das políticas públicas de ações afirmativas implantadas e ampliadas nas duas últimas décadas [Pereira, F. 2022].

É com base nas preocupações advindas das pautas identitárias e da pressão midiática, que uma produtora quer que todos os grupos sociais sejam representados na elencagem de seu novo filme. Nesse contexto, há inúmeros fatores que, somados, tornam esta tarefa nada trivial. Um ator pode fazer parte de mais de um destes grupos. Como consequência, o número de atores pode ser menor que o número de grupos. Além disso, todos os personagens têm de ter um ator associado. E ainda, o custo do elenco tem de ser mínimo.

Novamente, o problema da elencagem representativa é apenas um contexto para entendermos como alguns problemas de enumeração do mundo real podem ser resolvidos pela técnica *branch-and-bound*. Esta técnica é empregada para resolver problemas de otimização através de uma “função limitadora” [Arsalan. 2022] .

Este método firma-se na premissa de desenvolver uma enumeração esperta das soluções candidatas à solução ótima de um problema. Nesta técnica, apenas uma parcela das soluções factíveis é realmente examinada. Esmiuçando o termo *branch-and-bound, branch* implica que o método efetua partições no espaço das soluções e o *bound* ressalta que a prova da otimalidade da solução utiliza-se de limites calculados ao longo da enumeração [Dos Santos, M. O. 2011].

## DADOS DO PROBLEMA

Para que possamos formular nossa função recursiva, há de se identificar as variáveis que fazem parte deste problema. Ou seja, descrever conceitualmente o arcabouço deste cenário.

Cada personagem do filme estará associado a um ator, portanto, precisamos saber a quantidade de personagens existentes no filme e, consequentemente, o conjunto de atores disponíveis:

$$n\text{: } número\ de\ personagens\ no\ filme$$
$$m\text{: } número\ de\ atores$$
$$A\text{: } o\ conjunto\ de\ atores\ disponíveis$$

Há uma preocupação da empresa em representar grupos sociais diversos a fim de evitar polêmicas de representatividade, portanto, aparece a necessidade de uma variável para este conjunto de grupos sociais:

$$S\text{: } o\ conjunto\ de\ grupos\ sociais\ a\ serem\ representados$$
$$l\text{: } número\ de\ grupos$$

Nos é útil também, uma variável para saber em quais grupos sociais um ator $a$ está associado:

$$S_a\text{: } o\ conjunto\ de\ grupos\ sociais\ dos\ quais\ o\ ator\ a\ faz\ parte$$

É notório que os atores não recebem os mesmos salários, isso pode flutuar de acordo com a qualidade do ator, o sucesso, a popularidade, etc.

$$v_a\text{: } o\ valor\ de\ custo\ associado\ a\ um\ ator\ a$$

Com as nossas variáveis devidamente descritas, podemos expressar matematicamente o que queremos encontrar. Neste trabalho, nosso objetivo é chegar em um subconjunto $X \subseteq A$ tal que tenhamos um número de atores igual a quantia de personagens:

$$|X| = n$$

Além disso, precisamos que a união dos grupos que cada ator representa seja igual ao conjunto de grupos que desejamos representar:

$$U_{a \in X} S_a = S$$

Por fim, também queremos que a soma dos preços cobrados pelos atores escolhidos seja mínima:

$$\sum_{a \in X} v_a \; seja \; mínimo$$

## RECURSÃO

Para resolver o problema, o grupo optou por utilizar um vetor característico, como visto em sala para o exemplo da mochila, para representar cada possível combinação de atores escolhidos/removidos. Para isso, as seguintes variáveis são utilizadas:

$$groups\text{: } vetor\ de\ inteiros\ para\ representar\ os\ grupos\ desejados$$
$$actors\text{: } vetor\ que\ armazena\ os\ atores\ da\ entrada$$
$$selection\text{: } vetor\ característico\ que\ representa\ cada\ instância$$
$$optSelection\text{: } vetor\ característico\ que\ representa\ a\ instância\ ótima$$
$$optSum\text{: } soma\ dos\ custos\ dos\ atores\ da\ instância\ ótima$$

A árvore de recursão é binária e cada nível $i$ representa uma decisão a ser tomada para o ator de índice $i$. De forma a tornar as funções limitantes mais diretas, o vetor de atores é ordenado do menor custo para o maior. Dessa forma, cada índice $i$ não necessariamente representa o ator $i$ e, portanto, os atores guardam na sua estrutura qual é o $id$ original atribuído na leitura dos dados.

Também vale ressaltar que, devido à ordenação, primeiro checamos os ramos em que a decisão de escolha é manter o ator na árvore, visto que acreditamos ser uma boa heurística manter os atores de custo menor primeiro.

Todas as variáveis acima são globais devido à natureza intrínseca do algoritmo e a função $BT$ recebe dois argumentos:

$$i\text{: índice para o qual a decisão deve ser tomada}$$
$$count\text{: } quantidade\ de\ atores\ selecionados$$

Com o intuito de manter a monotonicidade da recursão, nossa implementação inicia no nodo de índice 1 com um $count = m$. Ou seja, consideramos que todos os atores foram escolhidos para compor o elenco inicialmente. Dessa forma, se é possível cobrir todos os grupos, ao selecionarmos todos os atores garantimos que essa restrição foi coberta.

Os próximos passos então consistem em decidir, para cada ator, se esse ator deve permanecer no elenco ou não até termos um número "$n$" de atores selecionados.

Ao implementar dessa maneira, garantimos que a árvore é monotônica: a partir do momento que a remoção de um ator faz com que o elenco selecionado se torne inviável devido à restrição de grupo, é impossível, nas subárvores abaixo, tornar o elenco viável

novamente. O mesmo vale para o número de atores: a partir do momento que uma remoção deixe o elenco com um $número\ de\ atores\ <\ "n"$, nunca mais voltamos a ter "$n$" atores nas subárvores abaixo e, portanto, sempre teremos nós inviáveis por falta de personagens.

#### IDEIA GERAL DE IMPLEMENTAÇÃO

O grupo se baseou no pseudocódigo passando em sala para branch and bound:

```
BB(l)
  se(folha)
        valor = f(característico)
        Se valor < opt
              opt = valor
              optCaracterístico = característico
  senão
        calcule cl
        B = bound(característico)

        para todo el E cl
              se b >= opt
                    continue
              xl = el
              BB(l + 1)
```

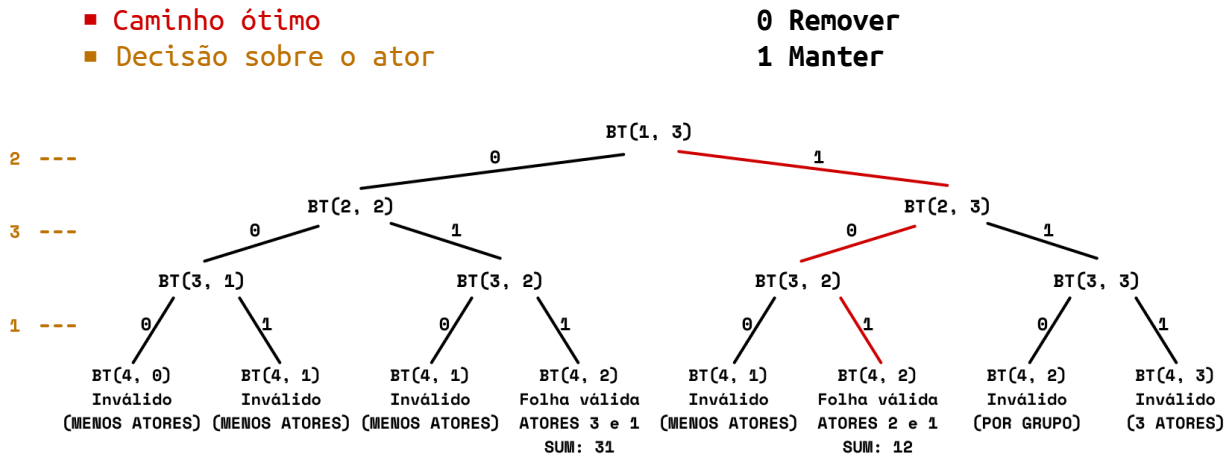
**RECURSÃO COMPLETA**

Para demonstrar a abertura completa da recursão, vamos utilizar o seguinte input como exemplo:

```
input 1.txt
3 3 2
10 2
1
2
2 2
2
3
5 1
3
```

```
$ ./elenco -fov < input1.txt
Número de nós acessados: 15
Cortes por viabilidade: 0
Cortes por otimalidade: 0
Tempo de execução: 0.00292 ms
1 2
12
```

Para o exemplo acima, como demonstra o relatório do nosso programa, temos 15 nós na recursão, demonstrados no esquema abaixo:




Perceba que para o `input1.txt`, após ordenação, temos o seguinte vetor de atores:
$actors: (2, 3, 1)$
Por consequência, o nível 1 da recursão, responsável por decidir o ator de índice 1, decide sobre o ator 2 e assim por diante.

**CORTES POR VIABILIDADE**

Seguimos algumas regras de viabilidade para decidir se podemos descer em uma subárvore:

Se a seleção de um nó é 0, o seguinte é analisado:

1) $count - 1 >= n \rightarrow$ *se temos atores o suficiente selecionados*
2) *se não violamos a viabilidade por grupo ao retirarmos esse ator*
3) $not\ (i + 1 > m\ and\ count - 1 > n) \rightarrow$ *se não estamos entrando em uma folha com um número maior de atores que o necessário*

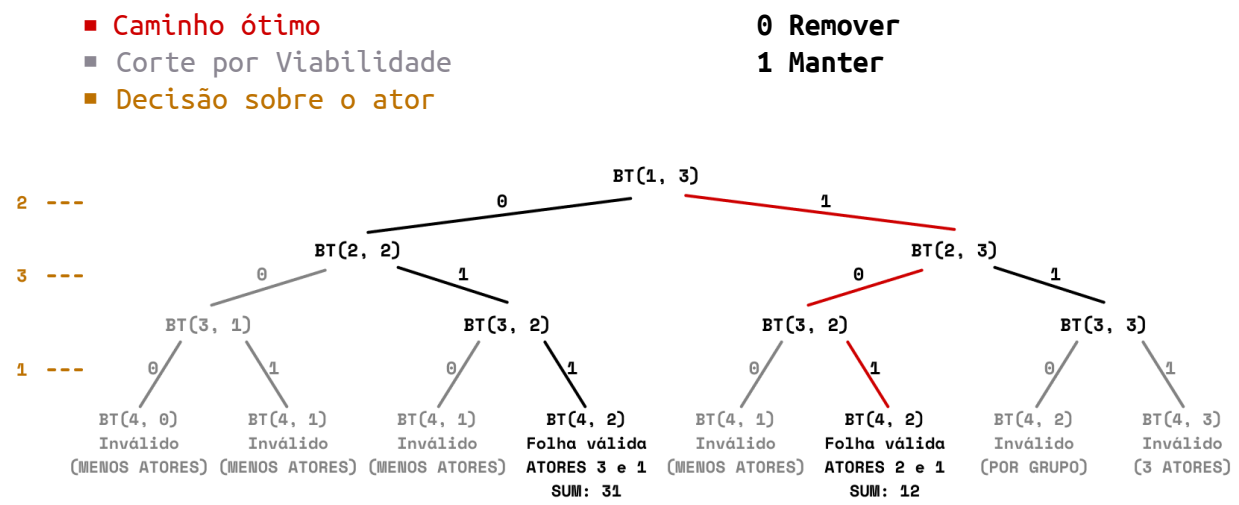
Se a seleção de um nó é 1, então analisamos:

1) $not\ (i + 1 > m\ and\ count > n) \rightarrow$ *se não estamos entrando em uma folha com um número maior de atores que o necessário*
2) $count - n <= m - i \rightarrow$ *se ainda temos níveis o suficiente para selecionar apenas o número de atores necessários*

Ao adicionarmos cortes por viabilidade ao input1.txt

```
$ ./elenco -ov < input1.txt
Número de nós acessados: 8
Cortes por viabilidade: 5
Cortes por otimalidade: 0
Tempo de execução: 0.00223 ms
1 2
12
```

temos a seguinte árvore:



Pelo relatório percebemos que realizamos cinco cortes: dois cortes no nodo (3, 3), um porque ao retirarmos o ator 1 não é possível cobrir os grupos e outro porque temos mais atores que o necessário; um no nodo (3, 2) devido a uma $quantidade\ de\ atores\ <\ n$; uma no nodo (2, 2) devido também a $atores\ <\ n$ - vale apontar também que o resto da subárvore da esquerda do nó (2, 2) nem sequer é cogitada devido a monotonicidade da recursão. Finalmente, temos um corte no nodo (3, 2) também devido a uma $quantidade\ de\ atores\ <\ n$.

**CORTES POR OTIMALIDADE**

Os cortes por otimalidade são mais simples:

```
Se boundValue >= optSum
  continue;
```

Ou seja, para cada seleção para o ator atual (1 ou 0), antes de descer na recursão, checamos o valor de $bound$. Se esse valor é maior ou igual ao ótimo já encontrado, não descemos, visto que é impossível atingir menor valor nesse ramo.

Dados atores já escolhidos (E) e um conjunto de atores ainda por decidir (F) a função que o grupo escolheu para confrontar a fornecida é a seguinte:

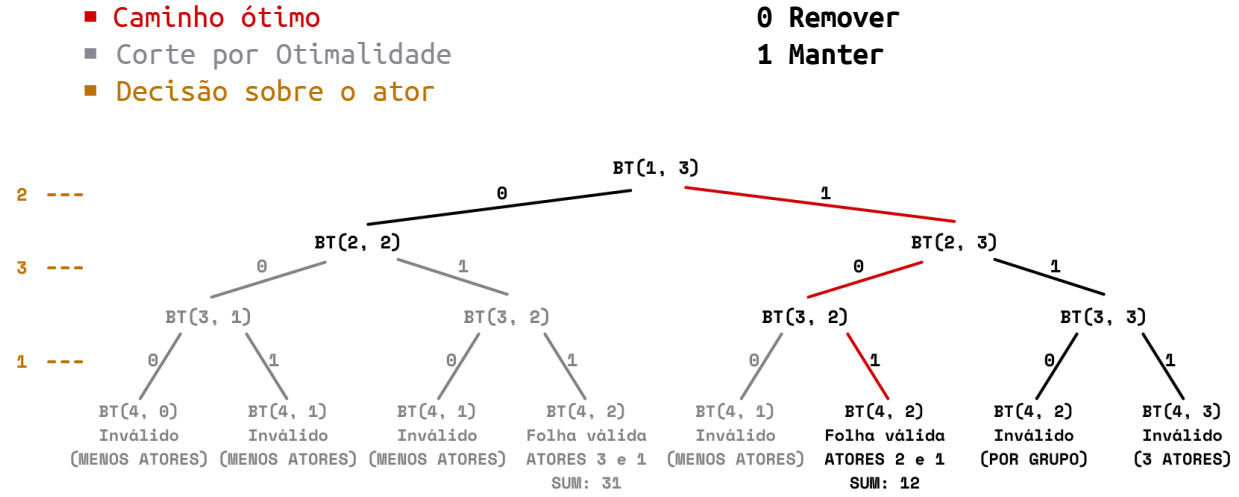
$B(E,\ F)\ =\ \sum_{a \in E} v_a\ +\ \sum_{i=1,\ a \in F}^{k} v_a$ , sendo k = número de atores que faltam para fechar n

Ou seja, a soma dos valores cobrados pelos atores já decididos a permanecer no elenco mais o valor dos próximos atores de F até completar o elenco. Como o vetor está ordenado do custo menor para o maior, o segundo somatório soma a taxa dos atores mais baratos sem decisão até fechar o elenco, sendo portanto uma estratégia gulosa similar à vista em sala para o problema da mochila.

Ao adicionarmos cortes por otimalidade, utilizando a função gulosa do grupo, ao input1.txt:

```
$ ./elenco -fv < input1.txt
Número de nós acessados: 8
Cortes por viabilidade: 0
Cortes por otimalidade: 3
Tempo de execução: 0.00171 ms
1 2
12
```

temos a seguinte árvore:




Os três cortes por otimalidade aqui são os seguintes: um no nó (3, 2) visto que já encontramos 12 e o bound para esses nós e suas subárvores é 12; dois no nó (2, 2) já que o bound para esse nó é 5 (manter o ator 3) + 10 (manter o ator 1) = 15 e nosso ótimo encontrado é 12.

## RESULTADOS

Um dos objetivos do trabalho era encontrar uma função limitante melhor que a fornecida. Nessa seção, vamos apontar exemplos de execução que demonstram quão melhor a estratégia escolhida pelo grupo foi e alguns exemplos que comprovam o funcionamento correto do algoritmo.

### ENTRADAS FORNECIDAS E COMPROVAÇÃO DE FUNCIONAMENTO

Arquivo input2.txt (viável com atores = (1, 3) e sum = 15):

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

```
$ ./elenco < input2.txt
Número de nós acessados: 4
Cortes por viabilidade: 1
Cortes por otimalidade: 2
Tempo de execução: 0.00136 ms
1 3
15
```

Arquivo input3.txt (inviável):

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

```
$ ./elenco < input3.txt
Inviável
```

Arquivo input4.txt (viável com atores = (1, 3) e sum = 15):

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

```
$ ./elenco < input4.txt
Número de nós acessados: 4
Cortes por viabilidade: 1
Cortes por otimalidade: 2
Tempo de execução: 0.00147 ms
1 3
15
```

**COMPARAÇÕES DE DESEMPENHO**

Para comparar o desempenho das funções de otimalidade e das escolhas de cortes por viabilidade vamos utilizar um input chamado “bigInput.txt” presente na pasta do trabalho. Não o adicionaremos ao relatório visto que é muito grande.

Em uma execução sem nenhum corte por viabilidade ou otimalidade, temos o seguinte:

```
$ ./elenco -fo < bigInput.txt
Número de nós acessados: 536870911
Cortes por viabilidade: 0
Cortes por otimalidade: 0
Tempo de execução: 315139 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594
```

O número total de nós acessados faz sentido visto que temos 28 atores e a árvore de decisão é binária, o que resulta em $2^{29} - 1$ nós, reforçando a confiabilidade da implementação. Como nenhum corte está ativo, a escolha de uma função de bound é irrelevante e portanto mostramos apenas a saída sem a opção “`-a`”.

Vamos começar comparando as funções de otimalidade:

```
$ ./elenco -f < bigInput.txt
Número de nós acessados: 86
Cortes por viabilidade: 0
Cortes por otimalidade: 23
Tempo de execução: 0.01935 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594

$ ./elenco -fa < bigInput.txt
Número de nós acessados: 158562
Cortes por viabilidade: 0
Cortes por otimalidade: 158499
Tempo de execução: 16.7571 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594
```

Nota-se que a função gulosa escolhida é muito melhor que a função fornecida nesse caso. Com apenas 23 cortes por otimalidade, a nossa função reduziu a árvore a apenas 86 nós (0.00000428408% da original), enquanto que a função concorrente realizou 158.499 cortes e reduziu a árvore a 158.562 nós (0.02953447407 % da original) . Vale

ressaltar que o tempo de execução no nosso caso também foi significativemente menor, o que demonstra que a função não realiza nada exaustivo ou muito custoso para tomar decisões de limitação.

Olhando apenas para os cortes de viabilidade, temos:

```
$ ./elenco -o < bigInput.txt
Número de nós acessados: 593774
Cortes por viabilidade: 397215
Cortes por otimalidade: 0
Tempo de execução: 193.791 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594
```

O programa realizou 397.215 cortes, reduzindo a árvore a 593.774 nós (0.11059902629% da original), demonstrando que os critérios escolhidos para a viabilidade estão corretos. Como os cortes por otimalidade estão desativados, a escolha de uma função de bound é irrelevante e portanto mostramos apenas a saída sem a opção “`-a`”.

Com ambos os cortes ativos, temos:

```
$ ./elenco < bigInput.txt
Número de nós acessados: 29
Cortes por viabilidade: 5
Cortes por otimalidade: 23
Tempo de execução: 0.03434 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594

$ ./elenco -a < bigInput.txt
Número de nós acessados: 33390
Cortes por viabilidade: 12370
Cortes por otimalidade: 21019
Tempo de execução: 12.0105 ms
1 3 4 5 6 7 8 9 10 11 14 15 16 17 18 21 22 23 24 25 26 27 28
594
```

Nota-se aqui que a combinação dos cortes torna a nossa função ainda melhor para o exemplo em questão, uma vez que acessa apenas 29 nós contra 33.390 nós da função concorrente.

## CONCLUSÃO

Dadas as análises feitas acima, conclui-se que a nossa solução é satisfatória e que os objetivos do trabalho foram atingidos. Conseguimos implementar uma função de branch and bound para resolver o problema do Elenco Representativo e encontramos uma função limitante significativamente melhor para o exemplo analisado, além de tomar decisões para cortes por viabilidade corretas e funcionais.

Também podemos apontar a importância de se escolher a função correta ao se deparar com um problema de *branch-and-bound*. Se a estrutura da árvore é conhecida, é necessário que o programador analise as possibilidades, encontre as viabilidades e formule uma equação limitante relevante, visto que decisões corretas podem diminuir significativamente a complexidade de um problema para determinadas instâncias.

## REFERÊNCIAS BIBLIOGRÁFICAS

Guedes, A. P. 2022. Segundo Trabalho. Disponível em: <https://www.inf.ufpr.br/andre/Disciplinas/CI1238-2022-1/trabalho2.pdf>. Acesso em 09 de Setembro de 2022.

Dos Santos, M. O. 2011. Algoritmo de *Branch-and-Bound* (ou enumeração implícita). Disponível em: <https://sites.icmc.usp.br/mari/segundo2011/Aula0711.pdf>. Acesso em 09 de Setembro de 2022.

Arsalan, M. 2022. *Difference between Backtracking and Branch-N-Bound technique.* Disponível em: <https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/>. Acesso em 07 de Setembro de 2022.

Pereira, F. 2022. VIVO promove ação afirmativa que potencializa representatividade negra. Disponível em: <https://vejario.abril.com.br/coluna/fabiane-pereira/vivo-promove-acao-afirmativa-que-potencializa-representatividade-negra/>. Acesso em 06 de Setembro de 2022.