Branch & Bound para elenco representativo

Richard Fernando Heise Ferreira GRR20191053
Anderson Aparecido do Carmo Frasão GRR20204069

[1]Universidade Federal do Paraná
Curitiba – PR – Brasil

1. Introdução ao problema

Elenco Representativo

Uma produtora de filmes quer fazer um filme e está a procura do elenco ideal. Tentando evitar polêmicas de representatividade do elenco em relação a grupos da sociedade, a produtora quer que todos os grupos (previamente elencados) da sociedade sejam representados. Um ator pode fazer parte de mais de um destes grupos, portanto, podemos ter menos atores que grupos. Obviamente, é preciso que cada personagem tenha um ator associado e cada ator cobra um valor para fazer o filme.

Dados um conjunto S de grupos, um conjunto A de atores, um conjunto P de personagens, e, para cada ator $a \in A$, um conjunto, $S_a \subseteq S$ indicando os grupos dos quais *a* faz parte, devemos encontrar um elenco que tenha um ator para cada personagem (todos os atores podem fazer todas as personagens) e todos os grupos tenham um representante. Além disso, também temos um valor, v_a, associado com cada ator $a \in A$, e queremos que o custo do elenco seja mínimo.

Ou seja, devemos encontrar um subconjunto $X \subseteq A$ tal que:

- $|X| = |P|$
- $\bigcup_{a \in X} S_a = S$; e
- $\sum_{a \in X} v_a$ seja mínimo.

2. Modelagem

O problema foi modelado para uma solução que utiliza da técnica de *Branch&Bound* (Ramificação e Poda). Essa solução prevê a criação de uma árvore binária de decisões em que a decisão, nesse caso, consiste em pegar ou não um ator em cada nível de profundidade da árvore. A seguir vamos ilustrar essa solução e descrever as variáveis que usamos:

Variáveis:

- otimo = { 'custo': float("inf"), 'melhores_atores' : [] }: variável com o custo ótimo e os atores que culminam nesse custo
- l: número de grupos a serem representados
- n: número de papeis a serem cobertos
- m: número de atores possíveis
- nodos: nodos visitados na árvore de decisão
- valores: vetor com os valores de cada ator, respectivo à ordem da entrada
- grupos: vetor de grupos de cada ator, respectivo à ordem da entrada

Ainda há duas variáveis chamadas *cortes_otimalidade* e *cortes_viabilidade*, que servem para guardar a quantidade de cortes por otimalidade e viabilidade, respectivamente; essas são, porém, variáveis de controle que vamos comentar mais sobre na sessão de implementação propriamente dita.

3. Análise de Funções Limitantes

A função limitante fornecida segue o seguinte algoritmo:

Algorithm 1 B_dada(*atores*)

result = custo(*atores*)
index_min = valores.index(min(valores_ainda_não_escolhidos))
result = result + (n - $|atores|$)*valores[index_min]
return result

Basicamente a função descobre o índice do valor mais barato entre os atores ainda não escolhidos e o multiplica pelo número de atores que faltam escolher.

A função limitante criada por nós segue o seguinte algoritmo:

Algorithm 2 B_nossa(*atores*)

result = custo(*atores*)
candidatos = atores_não_escolhidos
candidatos.ordenar()
remove_mais_caros(candidatos)
return result + soma(candidatos)

Basicamente a função ordena o vetor de candidatos ainda não escolhidos e preenche os papeis ainda sem atores com os mais baratos.

A função auxiliar *custo()* simplesmente soma o custo dos atores já escolhidos no vetor *atores*. Para entender melhor o funcionamento das funções limitantes vamos demonstrar como o algoritmo decide quais nodos olhar com os cortes de otimalidade e viabilidade ligados no exemplo fornecido pelos professores.

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

Para esse exemplo temos a seguinte árvore de decisão:

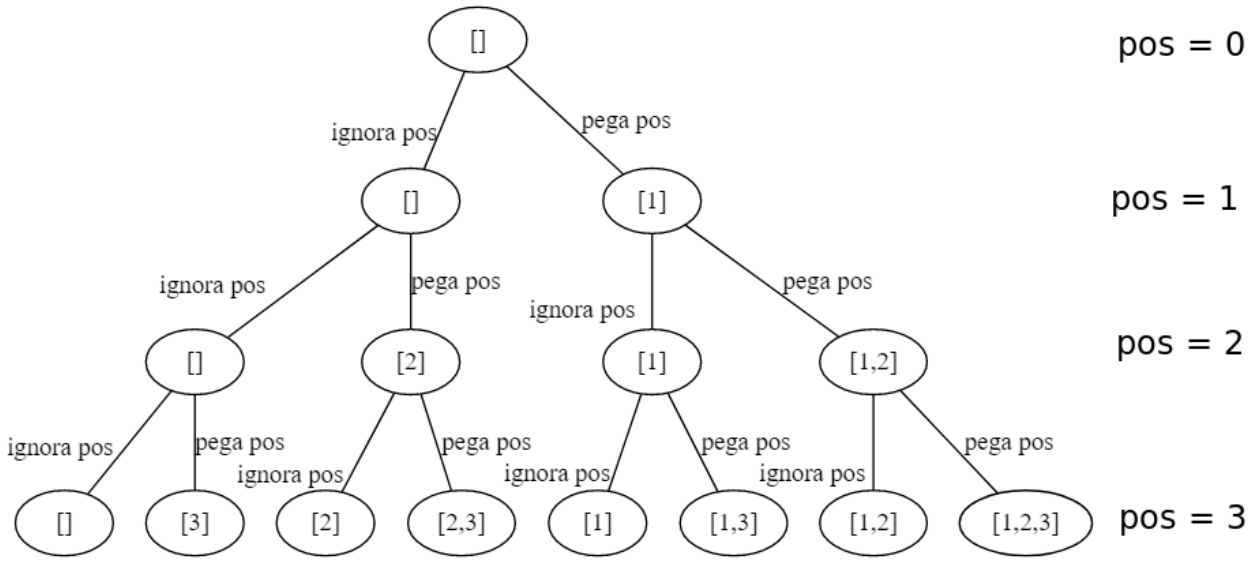

Figure 1. Arvore de decisão

Resultante do algoritmo de *Branch&Bound*. A implementação desse algoritmo em específico será definida na sessão a seguir que cobre a implementação, por hora gostaríamos de mostrar o caminho percorrido, em vermelho ,a figura 2, pela função limitante fornecida e o caminho percorrido, em azul na figura 3, pela nossa função limitante.

É possível observar que na figura 2 o caminho com o *bound* fornecido percorre 7 nodos da árvore, como demonstra o relatório do programa na saída de erro padrão. Em amarelo está o nodo que é pego temporariamente até se encontrar a solução ótima em vermelho. Já utilizando o nosso *bound* percorremos apenas 4 nodos da árvore, encontrando diretamente o vetor ótimo de atores [1,3] em azul escuro. Isso ocorre porque a nossa função limitante utiliza uma solução praticamente ótima, similar a de um algoritmo guloso, o que faz com que encontremos a melhor solução percorrendo consideravelmente menos nós do que a função limitante fornecida.

Agora, a fins de comparação de desempenho, utilizamos um exemplo fornecido na subpasta *exemplos/*, o nome do arquivo é *grande.txt* e consiste de uma entrada gerada por nós que possui 28 atores, 7 grupos e 12 papéis. Para essa entrada a função limitante fornecida resolveu o problema em aproximadamente 2 segundos, enquanto utilizando-se da nossa função limitante foram aproximadamente 00.000649 segundos (ou cerca de 600 microssegundos). Essa diferença também está nos nodos percorridos: a função fornecida fez com que o algoritmo percorresse 94351 nodos, enquanto a nossa percorreu apenas 29 nodos – praticamente o ideal pela solução gulosa. Ainda é possível averiguar que a nossa solução faz 28 cortes por otimalidade e 0 por viabilidade, enquanto a fornecida

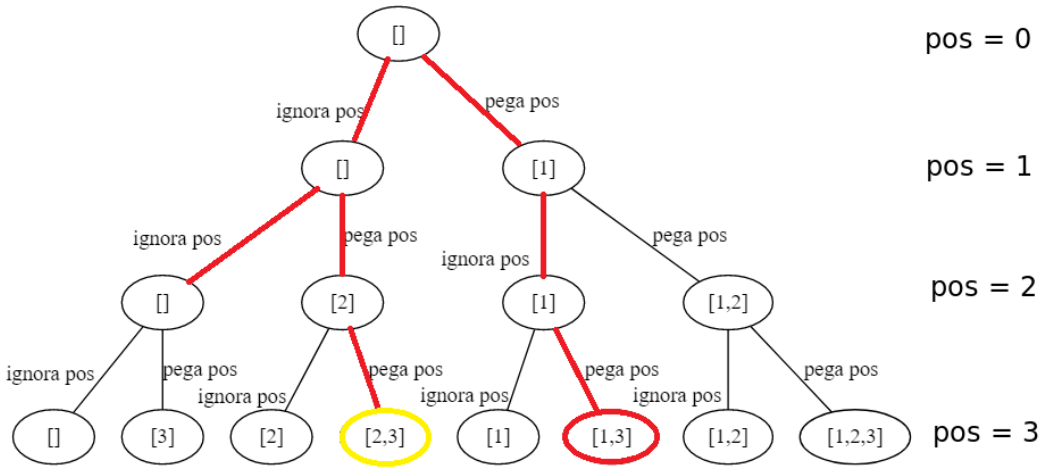

Figure 2. Caminho com *bound* fornecido

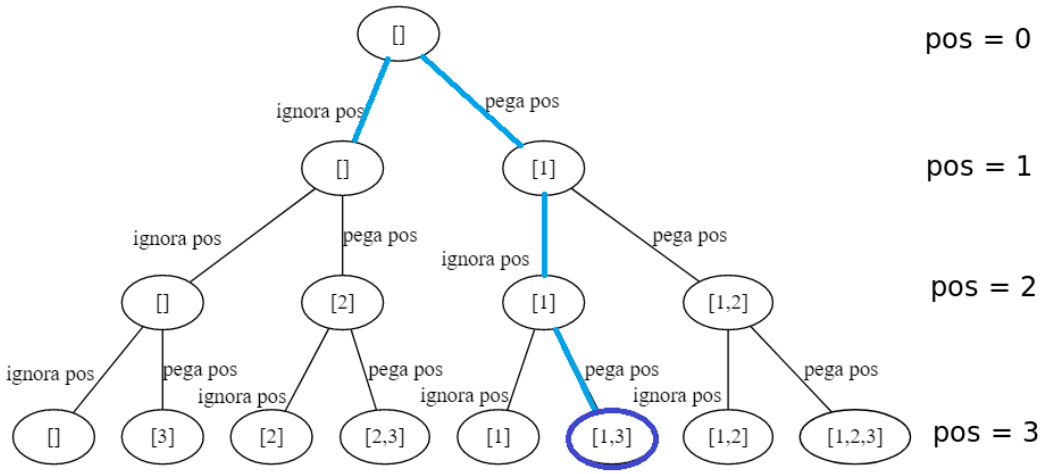

Figure 3. Caminho do nosso *bound*

realiza 73495 cortes por otimalidade e 19933 cortes por viabilidade (essas informações estão disponíveis rodando o algoritmo com a opção *verbose* indicada pelo argumento -v na linha de comando, mais detalhes sobre a implementação virão a seguir).

4. Detalhes de Implementação

Feito em python. São 6 funções ao todo:

- main: função principal que possui leitura de entrada e chamada da função que resolve o problema além de escrita na saída padrão e saída de erro.
- viavel(pos, atores, primeiro): verifica se a solução é viável para o ator indicado por *pos* e o vetor de atores escolhidos *atores*. Tem suas funções desligadas caso a flag *-f* esteja ativada.
- elenca(pos, atores, primeiro): função padrão de *Branch&Bound* definida a seguir.
- B_dada(pos, atores): definida na sessão anterior.
- B_nossa(pos, atores): definida na sessão anterior.
- custo(atores): função que calcula custo dos atores escolhidos por *atores*.

Temos 4 possíveis argumentos da linha de comando: *-a*, *-f*, *-o* e *-v*.

- *-a*: utiliza a função fornecida como limitante. A padrão é a nossa.
- *-f*: desliga cortes por viabilidade.
- *-o*: desliga cortes por otimalidade.
- *-v*: opção *verbose*. Escreve um relatório mais detalhado na tela com número de nós cortados por viabilidade e por otimalidade. Além do ”nome” de cada nó descido na árvore.

Nosso algoritmo *elenca()* possui dois casos bases: se o nó atual é viável e se chegamos em uma folha. Se é inviável retornamos, se chegamos em uma folha verificamos se temos um caminho ótimo. Se não estamos em um caso base, vamos verificar se devemos realizar cortes de otimalidade e/ou viabilidade. Dependendo do valor de cada flag o fluxo é desviado de maneira correspondente, para fins de entendimento geral, porém, vamos assumir que vamos realizar os cortes: nesse caso, calculamos o *bound* de ignorar o ator indicado por *pos* e o *bound* de pegar o ator indicado por *pos*. Realizamos a comparação de otimalidade e viabilidade para os dois casos e, finalmente, decidimos se vamos chamar a função recursivamente para cada caso com base na decisão ótima e/ou viável. Caso não queiramos realizar podas, simplesmente chamamos a função recursivamente para ambos casos: pegar ou ignorar o ator indicado por *pos*.

Para casos inviáveis a checagem é feita na primeira rodada do algoritmo caso a flag *-f* esteja ativa. Para nossos testes isso se mostrou suficiente.

Para rodar o algoritmo basta que o comando Make seja invocado, gerando um executável em python chamado ”elenca”. O executável, naturalmente, pode ser rodado com ./elenca e receberá argumentos de linha de comando além de uma entrada no formato correto na entrada padrão. a subpasta *exemplos/* tem algumas entradas e saídas esperadas.

Algorithm 3 elenca(*pos, atores, primeiro*)

```
nodo += 1
if not viavel(pos, atores, primeiro) then
    return
end if
if pos == m then
    custo_local = custo(atores)
    if custo_local < otimo[custo] then
        otimo[custo] = custo_local
        otimo[melhores_atores] = atores
        return
    end if
end if
if f == False then
    if o == False then
        bound_ignora = bound(pos+1, atores)
        bound_pega = bound(pos+1, atores+[pos])
        if bound_pega < bound_ignora then
            if viavel(pos+1, atores+[pos], 0 then
                elenca(pos+1, atores+[pos], 0)
            else
                corte_viabilidade += 1
            end if
        end if
        if bound_ignora < otimo[custo] then
            if viavel(pos+1, atores, 0) then
                elenca(pos+1, atores, 0)
            else
                cortes_viabilidade += 1
            end if
        else
            cortes_otimalidade += 1
        end if
        Testamos somente viabilidade
    end if
else
    if o == False then
        Testamos somente otimalidade
    else
        elenca(pos+1, atores, 0)
        elenca(pos+1, atores+[pos], 0)
    end if
end if
```