Problema do Elenco Representativo: Solucionando Por Algoritmo Branch and Bound

Alejandro Gemin Rosales

Resumo

O algoritmo Branch and Bound é uma forma de implementação para que se possa encontrar, com soluções ótimas, o resultado para diversos problemas de otimização algorítmica. O método proposto por A. H. Land e A. G. Doing em 1960, consiste em gerar um grafo ou uma árvore de possíveis possibilidades onde o Branch é a sub-árvore (Ramo) e o Bound serias as funções limitantes, eliminando os Branches onde não será encontrado a solução ótima. Podemos utilizar o Branch and Bound em diversos problemas chamados de NP-completos, como por exemplo: Caixeiro viajante e o problema da mochila. O artigo a seguir possui como objetivo mostrar a solução do problema de otimização de escolha de atores, minimizando o custo, contudo também atendendo as condições necessárias, utilizando o método de Branch and Bound. O trabalho foi escrito em linguagem C++.

O Problema

Um diretor cineasta, quando busca atores para representar papéis em algum filme, deve pensar em como cada um se saiu em suas determinadas audições, assim posteriormente podendo tomar uma decisão de qual elenco montar. No nosso problema o diretor deve escolher, devido a representatividade, atores que façam parte de certos grupos. O desafio consiste em descobrir qual o menor custo possível de um elenco sendo que o mesmo se enquadre em todos os requisitos.

No problema do Elenco Representativo, devemos encontrar uma forma de criar um elenco que tenha um ator para cada papel e todos os grupos tenham um representante, de uma forma que o custo do elenco seja mínimo. É necessário um número *P* de Personagens, dentre *A* Atores que são representados por um Grupo G. Devemos encontrar um subconjunto $X \subseteq A$ tal que: $|X| = |P|$; $U_{a \in X} As = S$; $\sum_{a \in X} v_a$ seja mínimo.

Podemos então pensar como solução do problema uma estrutura de árvore.

Entradas e Saídas

Os formatos de entradas e saídas, são descritos a seguir e devem ser usados a entrada e saída padrão (stdin e stdout).

A entrada será formada por um conjunto de números inteiros. Os números podem estar separados por 1 ou mais espaços, tabs ou fim de linha.

Definição da entrada: É iniciado com os valores de l = |S|, m = |A| e n = |P| na primeira linha, são separados por espaço. Devemos considerar que S = {1, ..., l} e A = {1, ..., m}. Temos então m blocos, os quais são divididos um para cada ator. Um bloco começa com dois valores, v e s, que indicam o valor cobrado pelo ator e o número de grupos da sociedade que ele faz parte. Em seguida temos s números, que são os índices dos grupos.

Definição da saída: Em uma mesma linha, os números do conjunto X, separados por espaço e se m espaço no começo nem no fim da linha, são ordenados de forma crescente. Na linha seguinte, o valor total da contratação do elenco deve ser impressa. Caso a instância não possua uma solução viável deve ser informado com uma mensagem na saída, como por exemplo: “Inviável”.

Modelagem do Problema

A ideia que tive de início foi gerar uma árvore e fazer com que cada nível da árvore represente um Ator *A* e se ele possuir um filho, o ramo vai se estendendo a esquerda da árvore.

O primeiro passou foi achar um possível valor, ou uma possível solução, de alguma maneira para o problema. Por exemplo, precisamos contratar um número X de atores. Irei contratar até preencher todas as vagas, ou seja as X vagas. Após tal feito, somo o custo de cada um deles, para descobrir o custo total da contratação. Contudo esse não é o melhor custo, pois foi gerado de uma maneira aleatória. Por isso é necessário achar o menor custo de alguma forma.

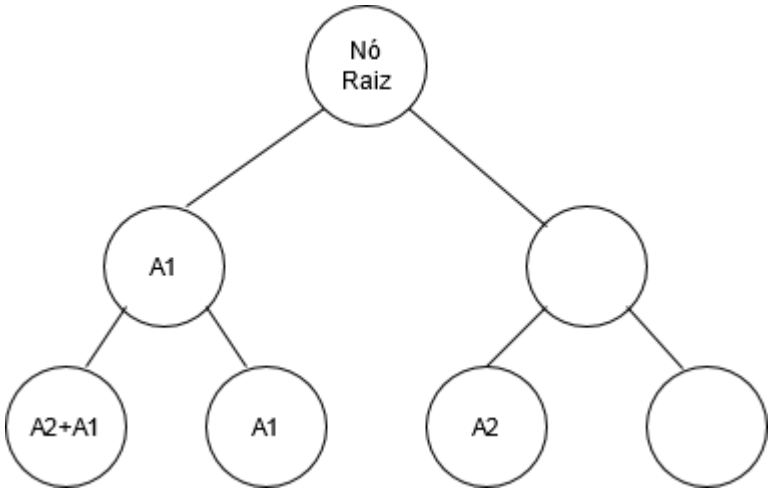

Figura: Exemplo de uma árvore ramificada do Branch and Bound.

Para os Bounds, foi utilizado a lógica que retorna, de todos os atores, os custos até que se chegue em um nó atual e um Bound se baseado em custo e se existir um novo grupo no elenco.

Um array foi utilizado para percorrer a Árvore, verificando se o nó irá gerar filhos, caso sim, os filhos serão adicionados ao início, possuindo prioridade na execução e removendo o nó que gerou os filhos da fila. Se o nó não gerar filhos, ele é retirado sem a adição de outro.

O nó da arvore representaria a contratação de cada ator. Caso a contratação do custo passe do resultado que foi achado anteriormente, concluímos que a solução que estamos chegando estará errada, pois, mesmo escolhendo os atores mais baratos para completar os papéis restantes, o custo ainda será alto. Seguindo essa linha é otimizado o algoritmo para não aceitar a contratação dos atores visto que não iria ser a melhor escolha.

Podemos chamar a implementação acima de implementação de Bound Preguiçoso, a qual tem como objetivo conseguir calcular a soma total do elenco atual mais o ator que será adicionado.

Outra ideia de implementação é a qual pode se verificar se o ator adicionado corresponde a um grupo que não foi representado até o momento, se for alguém novo, então será somado o custo igual como no Bound Preguiçoso, caso ocorra o contrário, não será percorrido o nó. A diferença dessa implementação que podemos chamar de Bound de Agrupamento, é que ela pode possuir um modo somente dela para gerar os nós. Quando o número de grupos for totalmente preenchido, o algoritmo pode então adicionar atores não utilizados, que possuem o menor custo, ao elenco. Após tal operação, pode se verificar se o custo total será menor que o atual custo. Caso sim, o algoritmo possui a solução melhor.

Implementação do Problema

A implementação desse trabalho foi feita na linguagem C++ e separada em funções a seguir:

Uma classe *class Ator*: onde são passados um identificador do ator, o custo para contrata-lo e a lista de grupos aos quais o ator pertence

Um construtor *Ator*

Um método *void adicionarGrupo:* o qual adiciona um grupo a uma lista de grupos do ator

Uma classe *class No*: a qual representa o nó da árvore. Uma variável que representa o custo de todos os atores contratados. Lista dos atores contratados e lista dos filhos do no. Um array que nos diz quais grupos foram representados.

Um construtor da classe *No*

Um método *void gerarFilhos*: o qual gera todos os possíveis filhos do nó.

O main do programa: Onde declaro as variáveis globais. Leio os dados de entrada do sistema. Construo a árvore. Imprimo a saída do problema, com as soluções encontradas.

Exemplos Testados

Entrada:

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

Saída:

1 3
15

Entrada:

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

Saída:

Inviável

Conclusão e Considerações Finais

Foi possível concluir que nos exemplos demonstrados acima, houve sucesso na execução do algoritmo. Apesar de não ter sido implementado ao código a opção do Bound de Agrupamento, podemos interpretar que o resultado comparado com o Bound Preguiçoso, poderá não ser tão eficiente, visto que o custo no é maior a ser percorrido do que o do seu concorrente. Todavia é possível que a situação seja revertida caso o número de grupos seja menor que o número de personagens e atores.