Universidade Federal do Paraná

Bacharelado em Ciência da Computação

Disciplina: Otimização

Relatório referente ao
Trabalho 2

Luan Marko Kujavski - GRR20221236
Marcelo Eduardo Marques Ribas - GRR20221258

Julho - 2024

1. Introdução

O problema da Comissão Representativa envolve a formação de uma comissão governamental que deve representar todos os grupos da sociedade. Cada grupo é previamente definido, e cada candidato pode pertencer a um ou mais desses grupos.

O objetivo é selecionar o menor número possível de candidatos de forma que todos os grupos sejam representados na comissão. Formalmente, dado um conjunto de grupos S, um conjunto de candidatos C, e um subconjunto $Sc \subseteq S$ para cada candidato c, que indica os grupos aos quais c pertence, deve-se encontrar um subconjunto $X \subseteq C$ tal que a união dos grupos representados pelos candidatos em X cubra todos os grupos em S e $|X|$ seja minimizado.

2. Modelagem

Para resolver o problema da Comissão Representativa, utilizamos o algoritmo Branch & Bound devido à sua eficácia em resolver problemas de otimização combinatória. O problema consiste em selecionar o menor número possível de candidatos de modo que todos os grupos da sociedade sejam representados. Este tipo de problema é intrinsecamente combinatório e pode ser modelado como um problema de cobertura mínima de conjuntos, que é um problema *NP-difícil*.

3. Detalhes da implementação

O programa que resolve o problema da Comissão Representativa foi feito utilizando a linguagem de programação Python. O nome do arquivo é comissao.py.

Para executar o programa, utilize a seguinte linha de código:

```
python3 comissao.py
```

Para facilitar a análise, entregamos os arquivos teste[1..4].txt, que podem ser usados para testar o programa. Para utilizá-los, utilize:

```
python3 comissao.py < teste.txt
```

Para facilitar a resolução do problema, criamos duas classes: Problema e Candidato. Cada Candidato tem um id único e seus grupos. O Problema armazena todas as informações passadas pela linha de comando. Um detalhe importante é que, ao receber todos os candidatos, ordenamos eles em forma decrescente no número de grupos a quais pertence cada um. O algoritmo branch and bound, então, tende a encontrar uma solução viável mais rapidamente em casos normais.

Para otimizarmos a comparação de dois objetos da classe Candidato, mudamos a forma como eles são representados em uma tabela hash, que é como dicionários são implementados.

A única forma de uma resposta ser inviável é se ela tem dois candidatos iguais. Dessa forma, o nosso corte por inviabilidade testa se o candidato adicionado já pertence ao grupo selecionado até a recursão do momento.

O corte por otimalidade implementado verifica se o candidato adicionado na recursão não contribui com a adição de algum grupo à lista de grupos dos candidatos da recursão.

A função limitante implementada verifica se, ao adicionarmos algum dos candidatos que restam na lista de candidatos da recursão, é possível que uma solução viável seja encontrada. Caso não, precisaríamos de no mínimo dois candidatos a mais para preenchermos todos os grupos. A diferença da função limitante dada pelo professor é que esta somente verifica se a solução até o momento é viável e, se não, precisaria de no mínimo mais um candidato para preencher todos os grupos.

Para desativar os cortes por otimalidade, digite -o na linha de comando. Para desativar os cortes por viabilidade, digite -f na linha de comando. Para usar a função limitante do professor, digite -a na linha de comando. Dessa forma, substitua *flag* pelo caractere desejado:

python comissao.py -*flag*

4. Análise das funções limitantes

O arquivo teste8.txt conta com 200 grupos e 100 candidatos. Utilizando esse caso de teste, constatamos que:

1. Utilizando a função limitante do professor e tirando os cortes por viabilidade e otimalidade, obtivemos os seguintes resultados:
    Tempo: 439 segundos
    Nós explorados: 7222309

2. Utilizando a nossa função limitante e tirando os cortes por viabilidade e otimalidade, obtivemos os seguintes resultados:
    Tempo: 47 segundos
    Nós explorados: 3223921

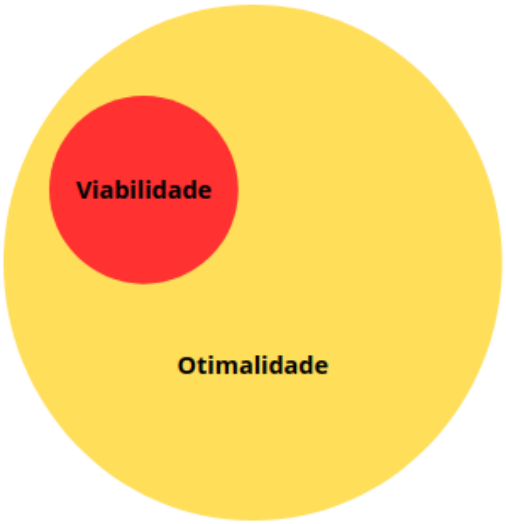
Os testes 1 e 2 serviram para testarmos a eficácia da nossa função limitante. A ideia é tirar qualquer corte por otimalidade e viabilidade para que possamos analisar apenas elas. Concluímos que nossa função limitante é várias vezes mais eficiente. A análise será em cima dela agora.

3. Utilizando a nossa função limitante e usando apenas cortes por viabilidade, obtivemos os seguintes resultados:
    Tempo: 3.97 segundos
    Nós explorados: 11313

4. Utilizando a nossa função limitante e usando apenas cortes por otimalidade, obtivemos os seguintes resultados:
    Tempo: 3.82 segundos
    Nós explorados: 10909

5. Utilizando a nossa função limitante e ativando os cortes por otimalidade e por viabilidade, obtivemos os seguintes resultados:
    Tempo: 3.84 segundos
    Nós explorados: 10909

Verificamos que, ao ativarmos os cortes por otimalidade e viabilidade ao mesmo tempo, o resultado é igual se ativarmos somente os cortes por otimalidade, vide testes 4 e 5. Isso ocorre porque o corte por viabilidade tem sua função contida no corte por otimalidade. Ou seja:



Isso se dá porque a função de viabilidade verifica se um candidato já foi adicionado no grupo de candidatos da recursão. A função de otimalidade verifica se a adição de um candidato faz diferença no conjunto de grupos da recursão. Além de tirar um candidato que já foi adicionado, a função de otimalidade retira casos em que um candidato está contido dentro de outro.