Branch & Bound: problema da separação de grupo minimizando conflitos

Ariel Evaldt Schmitt, Paulo Mateus Luza Alves

Departamento de Informática – Universidade Federal do Paraná (UFPR)

aes20@inf.ufpr.br (GRR20203949) pmla20@inf.ufpr.br (GRR20203945)

1. O PROBLEMA DA SEPARAÇÃO DE GRUPO MINIMIZANDO CONFLITOS

O problema da separação consiste em um conjunto de elementos, representando algum objeto, como por exemplo pessoas, os quais possuem afinidades ou conflitos entre si. A solução para o problema deve ser uma divisão desses elementos em dois grupos distintos, os quais devem respeitar todas as afinidades existentes, tendo como objetivo minimizar o número de conflitos existentes na divisão deste conjunto. A seguir apresentaremos a representação matemática do problema.

Dados os conjuntos $H, C\ e\ A$, onde H é o conjunto $[1..n]$, e $C\ e\ A$ conjuntos de pares $(x, y) \mid x, y \in H$. De forma que C representa os conflitos e A representa as afinidades entre os elementos de H.

Portanto, buscamos encontrar duas partições $H_1\ e\ H_2$ de H onde todas as afinidades são satisfeitas, ou seja S.P.G. escolhemos H_1:

$$\text{seja } (x, y) \in A, \text{ então } x \in H_1 \Leftrightarrow y \in H_1$$

2. A MODELAGEM

A solução do problema implementada faz o uso do conceito de *branch & bound* que consiste inicialmente em gerar uma árvore de soluções utilizando o *backtracking* o qual poda as ramificações em que não geram soluções válidas para depois podar as ramificações que não obterão soluções melhores do que uma já encontrada (*branch & bound*).

Sendo assim, o espaço de soluções são todas as possíveis bipartições do conjunto H enquanto que o espaço de soluções viáveis são todas as possíveis bipartições deste conjunto que respeitam as afinidades representadas pelo conjunto A.

Para realizarmos o corte das ramificações que não geram soluções viáveis, ou seja, para realizar o *backtracking* precisamos verificar se a escolha do grupo para o próximo nó a ser inserido na árvore irá preservar a viabilidade da solução, ou seja, se a inserção deste nó em um dos grupos $H_1\ ou\ H_2$ não irá desrespeitar as afinidades definidas em A.

Já para o corte de otimalidade, ou seja, para cortar os ramos que não obterão soluções melhores do que uma já encontrada, utilizamos funções de *bounding* as quais iremos apresentar abaixo.

2.2 FUNÇÃO FORNECIDA

Dados o conjunto de elementos com grupos já escolhidos (E), sabendo que $g(i)$ é o grupo do elemento i (já escolhido), definimos o conjunto C_E como sendo o conjunto dos conflitos que envolvem apenas elementos com grupos escolhidos.

Um triângulo em um conjunto $C' \subseteq C$ é uma tripla (x, y, z) tal que $(x, y), (x, z), (y, z) \in C'$.

Seja t_E o número de triângulos em $C \setminus C_E$ que não compartilham nenhum par de elementos.

Podemos então definir a função $B_1(E)$ por:

$$B_1(E) \;=\; |(x,\ y) \in C_E \,|\, g(x) \;=\; g(y)| \;+\; t_E$$

Ou seja, $B_1(E)$ é o número de conflitos onde os dois elementos envolvidos já foram colocados em um mesmo grupo mais o número de vezes que um triângulo exclusivo de conflitos aparece no conjunto de conflitos ainda não decididos.

2.3 FUNÇÃO DESENVOLVIDA

A função fornecida anteriormente realiza a soma da quantidade de conflitos já existentes na solução parcial com a quantidade de triângulos exclusivos entre os elementos que ainda não foram escolhidos. Com base nisso, percebemos que não era considerada a quantidade de conflitos que a inserção do nó atual causaria com os elementos cujo seus grupos já haviam sido definidos, ou seja com seus ancestrais na árvore.

Sendo assim, seja $[x_0,\ \dots\ , x_{l-1}]$ a solução parcial computada e x_l o nó da vez a ser inserido na árvore, e seja C_N o conjunto de conflitos que envolvem elementos com grupos já escolhido mais x_l, ou seja, elementos de $[x_0,\ \dots\ , x_{l-1},\ x_l]$, definimos c_E como:

$$c_E \;=\; |(x,\ y) \in C_N \;|\; x = x_l \;ou\; y = x_l\,|$$

Dessa forma, definimos B_2 como:

$$B_2(E) \;=\; |(x,\ y) \in C_E \,|\, g(x) \;=\; g(y)| \;+\; t_E \;+ c_E$$

que também podemos definir em função de $B_1(E)$:

$$B_2(E) \;=\; B_1(E) \;+ c_E$$

3. A IMPLEMENTAÇÃO

O algoritmo de implementação desta modelagem foi desenvolvido em linguagem C, e para o correto funcionamento, o código foi dividido em 3 etapas, sendo elas:

1. Leitura do *stdin*, coletando todas as informações necessárias e as armazenando na struct referenciada abaixo;
2. Computação da solução para a entrada fornecida;
3. Escrita no *stdout* no formato indicado na especificação.

3.1 AS STRUCTS

```
typedef struct optimize_state_t
{
   int *cur_solution;
   int *opt_solution;
   int opt_value;
   int nodes;
   double time;
} optimize_state_t;
```

Esta estrutura é responsável por armazenar as informações das soluções obtidas durante a execução do algoritmo.

```
typedef struct hero_pair_t
{
   int hero1, hero2;
} hero_pair_t;
```

```
typedef struct heroes_t
{
   int quantity, conflicts_qty, friendships_qty;
   hero_pair_t *conflicts;
   hero_pair_t *friendships;
   short int **aux_matrix;
} heroes_t;
```

Estas estruturas são responsáveis por armazenar todas as informações existentes no arquivo de entrada, que são necessárias para a resolução do problema.

3.2 PRINCIPAIS FUNÇÕES

```
int optimize_heroes(heroes_t *heroes, params_t *params, optimize_state_t *optimize);
```

A função *optimize_heroes* é *wrapper* da *branch & bound,* ou seja, nela apenas realizamos o cálculo do tempo utilizado para solucionar o input e chamamos a função recursiva da solução.

```
void *optimize_heroes_recursive(
    heroes_t *heroes,
    params_t *params,
    optimize_state_t *optimize,
    int depth);
```

A função *optimize_heroes_recursive* temos o *core* da solução, ou seja, a chamada recursiva de descida na árvore, onde todos os processos de poda de ramos, ou seja, o *backtracking* e o *branch & bound* são executados, e ao fim ela devolve a resposta ótima.

```
int profit(int *solution, heroes_t *heroes, int depth);
```

A função *profit* é responsável por calcular o valor da quantidade de conflitos existentes na solução enviada até o valor *depth* que é a profundidade atual da árvore.

```
int check_friendships(heroes_t *heroes, optimize_state_t *optimize, int cur_hero);
```

A função *check_friendships* se responsabiliza por verificar se a inserção do elemento atual mantém as afinidades, utilizada para a poda por meio do *backtracking*.

```
int partial_bound(
    heroes_t *heroes,
    params_t *params,
    int *partial_solution,
    int depth,
    int cur_group);
```

A função *partial_bound* é a nossa função *wrapper* para o *bounding*, ou seja, caso o parâmetro *-a* for enviado, utilizaremos a função fornecida pelo professor, caso contrário, a função desenvolvida será usada.

```
int check_feasibility(heroes_t *heroes, optimize_state_t *optimize);
```

A função *check_feasibility* é responsável por verificar se a resposta alcançada é válida. Esta função só é utilizada se o *backtracking* for desativado, pois caso contrário, ele mesmo já realiza esta verificação.

```
int check_conflicts(heroes_t *heroes, int *partial_solution, int cur_hero, int cur_group);
```

A função *check_conflicts* é utilizada apenas na função de *bounding* desenvolvida pela equipe, sendo responsável por calcular o valor do c_E indicado na modelagem.

4. RESULTADOS

A seguir apresentaremos os resultados obtidos com cada uma das funções que foram apresentadas. A saída possui o seguinte formato:

```
0
1 3 5 6 7 8 9 10 13 14 15
Tempo: 0.001768s
Nodos: 65535
```

⇒

```
valor ótimo
grupo do primeiro elemento
Tempo: tempo de execução
Nodos: número de nós da árvore
```

A solução do problema, ou seja, o valor ótimo e o grupo escolhido é impressa em *stdout* enquanto que as estatísticas do algoritmo são impressas em *stderr.*

4.1 EXEMPLO COM 15 ELEMENTOS

Um dos exemplos utilizados para testar a implementação foi

```
15 5 7
10 11
9 12
1 2
3 4
4 5
11 12
3 1
14 15
13 14
9 8
3 7
7 8
```

Força bruta

```
0
1 3 5 6 7 8 9 10 13 14 15
Tempo: 0.001768s
Nodos: 65535
```

Branch & bound desenvolvido

```
0
1 3 5 6 7 8 9 10 13 14 15
Tempo: 0.000015s
Nodos: 58
```

Backtracking

```
0
1 3 5 6 7 8 9 10 13 14 15
Tempo: 0.000040s
Nodos: 1251
```

Branch & bound fornecido

```
0
1 3 5 6 7 8 9 10 13 14 15
Tempo: 0.000014s
Nodos: 62
```

4.2 EXEMPLO COM 20 ELEMENTOS

Um dos exemplos utilizados para testar a implementação foi

```
20 10 7
1 10
17 13
13 12
8 5
3 19
20 11
4 3
18 8
14 6
9 7
1 2
3 5
20 12
8 14
18 3
9 17
7 6
```

Força bruta

```
0
1 2 3 5 6 7 11 13 15 16 18
Tempo: 0.090034s
Nodos: 2097151
```

Branch & bound desenvolvido

```
0
1 2 3 5 6 7 11 13 15 16 18
Tempo: 0.000114s
Nodos: 464
```

Backtracking

```
0
1 2 3 5 6 7 11 13 15 16 18
Tempo: 0.001319s
Nodos: 33817
```

Branch & bound fornecido

```
0
1 2 3 5 6 7 11 13 15 16 18
Tempo: 0.000117s
Nodos: 621
```