UNIVERSIDADE FEDERAL DO PARANÁ

Discente: Yuri Junqueira Tobias - GRR20211767
Discente: Pedro Amaral Chapelin - GRR20206145
Docente: André Guedes
Disciplina: Otimização - CI1238 **Data de entrega:** 28/06/2023

Modelagem e implementação, por Branch and Bound, o problema *Separação de grupo minimizando conflito*.

1. Introdução

Para este trabalho, precisamos implementar soluções para o problema de ***Separação de grupos minimizando conflitos***, pelo método *Branch&Bound*. Foi apresentado um exemplo desse problema utilizando super-heróis, basicamente é da seguinte forma: os super-heróis estão a caminho para uma missão importante e para isso precisam se subdividir em 2 grupos distintos. Entretanto, há **restrições** na parte da formação desses grupos, pois existem heróis que possuem mais afinidade com outros, e também existem heróis que possuem conflitos com outros heróis. Para os grupos, a primeira restrição é que os pares de heróis que possuem afinidade devem estar no mesmo grupo e; a segunda diz que nenhum dos grupos pode terminar vazio após a divisão, ou seja, ambos precisam ser populados.

> Considere que cada super-herói é indexado pelos números de 1 a n. Seja C o conjunto de pares (a_i, b_i), com $1 \leq i \leq k = |C|$, tais que a_i e b_i tem conflito, e seja A o conjunto de pares (a_i, b_i), com $1 \leq i \leq m = |A|$, tais que a_i e b_i tem grande afinidade.
>
> Dados n, C e A, queremos encontrar uma separação em dois grupos não vazios de tal forma que para todo par $(x, y) \in A$, x e y estão no mesmo grupo, e que minimiza o número de pares $(u, v) \in C$ tais que u e v estão no mesmo grupo[1].

Figura 1: Descrição do problema e das restrições

Todas as modelagens a seguir foram baseadas nas definições e conceitos de backtracking apresentados no livro **Combinatorial Algorithms - Generation, Enumeration and Search.** [Kreher, Stinson. CRC Press. 1999.] O mesmo foi

escolhido pois é didático e também foi altamente recomendado pelo professor da disciplina, André Guedes.

2. Modelagem e explicação

Para modelar o problema, optamos inicialmente por fazer o backtracking padrão, sem cortes por viabilidade e nem por otimalidade, de modo que o mesmo irá montar a árvore com todas as variações possíveis para os grupos, seguindo a seguinte lógica (em pseudo-código):

Algoritmo: Backtracking(l)
Global: QuantidadeConflitosÓtima, SoluçãoÓtima

// L é o "nível" da árvore/subárvore, se L == N, então a solução é completa, ou seja, todos os heróis pertencem a algum grupo.
Se (**já tem uma solução completa**)
{
  Se (**A solução completa segue as restrições de viabilidade**)
  {
    QuantidadeConflitosAtual ← (*Verifica quantidade de conflitos da solução atual*)

    Se (**QuantidadeConflitosAtual** < **QuantidadeConflitosÓtima**)
    {
      QuantidadeConflitosÓtima ← QuantidadeConflitosAtual
      SoluçãoÓtima ← SoluçãoAtual
    }
  }
}
Senão //**solução incompleta ainda**
{
  // x_l é o caminho a ser tomado na árvore de soluções, 1 é o caminho para adicionar o herói (l + 1) no grupo "A", 0 é o caminho para adicioná-lo no grupo "B"
  $x_l \leftarrow 1$ // Adiciona o herói no grupo "A"
  Backtracking(l + 1);
  $x_l \leftarrow 0$ // Adiciona o herói no grupo "B"
  Backtracking(l + 1);
}

Feita a modelagem acima, demos início aos estudos para encontrar formas de modelar o problema com cortes no caminho visando otimizar o tempo para encontrar a solução ótima, visto que a opção apresentada acima percorre todos os nós de possibilidades. Dito isso, optamos por iniciar com os cortes por **viabilidade**, ou seja, se uma solução (*incompleta*) já rompia com alguma das restrições de viabilidade, então nem precisávamos considerar dali para frente. É interessante ressaltar que a ideia utilizada foi a seguinte: antes de descer na árvore você pergunta se o herói para o qual será tomada a decisão de ir para o grupo A ou B possui afinidade com algum dos heróis que já foram inseridos em algum desses dois grupos, se sim, então obrigatoriamente este novo herói deverá ir para o mesmo grupo, do contrário, o mesmo pode ir para qualquer um dos grupos, da seguinte maneira(em pseudo-código):

Algoritmo: BacktrackingViabilidade(I)
Global: QuantidadeConflitosÓtima, SoluçãoÓtima, C_l

// L é o "nível" da árvore/subárvore, se L == N, então a solução é completa, ou seja, todos os heróis pertencem a algum grupo
// C_l é o conjunto de caminhos possíveis a partir de um nó.

```
Se (já tem uma solução completa)
{
  Se (A solução completa não apresenta grupos vazios)
  {
    QuantidadeConflitosAtual ← (Verifica quantidade de conflitos da solução atual)

    Se (QuantidadeConflitosAtual < QuantidadeConflitosÓtima)
    {
      QuantidadeConflitosÓtima ← QuantidadeConflitosAtual
      SoluçãoÓtima ← SoluçãoAtual
    }
  }
}
```

// Define o C_l a ser seguido, *corte de viabilidade*
Senão (**Solução incompleta ainda**)
{
  Se (**O herói de agora, possui afinidade com outro**)
  {
    $C_l \leftarrow$ {**Caminho para o grupo do outro**}
    restricao ← 1
  }
}
Se (**restricao == 1**) // Precisa ir para o caminho definido para não romper viabilidade
{
  $x_l \leftarrow$ Valor do C_l
  BacktrackingViabilidade(I + 1);
}
Senão (**restricao != 1**) // Pode pegar qualquer caminho
{
  $x_l \leftarrow 1$ // Adiciona o herói no grupo "A"
  BacktrackingViabilidade(I + 1);
  $x_l \leftarrow 0$ // Adiciona o herói no grupo "B"
  BacktrackingViabilidade(I + 1);
}

Concluída mais uma modelagem, agora está na hora de modelarmos o problema levando em consideração também os cortes por **Otimalidade,** que melhoram mais ainda a árvore de soluções por cortarem fora ramos em que há uma estimativa de existirem mais conflitos do que o ótimo até então.. Para realizar esse corte, é necessário uma **Função Limitante**, que será responsável por verificar um limitante inferior(estimativa) à solução atual. Os cortes por otimalidade devem ser realizados, possivelmente, de duas formas: a primeira somente com a função limitante que nos foi fornecida pelo docente e; a segunda, por uma função limitante desenvolvida pelos próprios discentes que realize mais cortes que a primeira e seja executada em menos tempo (segundos). A figura a seguir apresenta a função limitante fornecida no enunciado do trabalho:

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

Podemos então definir a função $B_{dada}(E)$ por:

$$B_{dada}(E) = |(x, y) \in C_E \mid g(x) = g(y)| + t_E.$$

Figura 2: Descrição da função limitante fornecida pelo docente (Parte 1).

Ou seja, $B_{dada}(E)$ é o número de conflitos onde os dois super-heróis envolvidos já foram colocados em um mesmo grupo mais o número de vezes que um triângulo de conflitos aparece no conjunto de conflitos ainda não decididos.

Figura 3: Descrição da função limitante fornecida pelo docente (Parte 2).

Daremos início apresentando a modelagem do corte por Otimalidade que realiza os cortes chamando a Função Limitante **calculaFigura**:

Algoritmo: BacktrackingOtimalidade(I)
External: calculaFigura
Global: QuantidadeConflitosÓtima, SoluçãoÓtima, C_l

// L é o "nível" da árvore/subárvore, se L == N, então a solução é completa, ou seja, todos os heróis pertencem a algum grupo
// C_l é o conjunto de caminhos possíveis a partir de um nó.
Se (**já tem uma solução completa**)
{
  Se (**A solução completa não apresenta grupos vazios**)
  {
    Se (**Fere a viabilidade de todos com afinidade estarem no mesmo grupo**)
      return
    QuantidadeConflitosAtual ← (*Verifica quantidade de conflitos da solução atual*)

    Se (**QuantidadeConflitosAtual** < **QuantidadeConflitosÓtima**)
    {
      QuantidadeConflitosÓtima ← QuantidadeConflitosAtual
      SoluçãoÓtima ← SoluçãoAtual

```
      }
    }
}

Senão (Solução incompleta ainda)
{
   // Conta os conflitos até agora na solução construída
   (Do I até L)
      ConfAtual ← (Varre a solução e verifica no Conjunto de conflitos quantos existem)

   // Calcula os possíveis conflitos dos herói (L + 1) até o N
   (Do L até N)
      ConfAtual ← calculaFigura

   // Corte de otimalidade
   Se (ConfAtual > QuantidadeConflitosÓtima)
      return

   x_l ← 1  // Adiciona o herói no grupo "A"
   BacktrackingOtimalidade(I + 1);
   x_l ← 0 // Adiciona o herói no grupo "B"
   BacktrackingOtimalidade(I + 1);
}
```

Agora temos uma função que encontra uma estimativa para o menor número de conflitos entre os heróis e, dependendo do resultado dessa função, o algoritmo recursivo, que faz as escolhas de grupo para os heróis restantes, corta os ramos que levam para um número maior do que o ótimo já encontrado. A função **calculaFigura** funciona da seguinte forma (em pseudo-código):

Algoritmo: calcula Figura(round, round Max, primeiro, da Vez)
Global: QuantidadeConflitosÓtima, C_l, roundMax

// Round Max pode ser **3** ou **5**, dependendo da função a ser executada: **Triângulo** ou **Pentágono**
// Primeiro é o primeiro vértice da figura, usado no fim da execução para ver se fecha um triângulo ou pentágono

```
// daVez é o vértice que está sendo analisado
```

```
Se (round == roundMax)
{
  De (0 até numeroConflitos)
  {
    Se (Algum elemento do par dos conflitos é o daVez e seu par é o primeiro)
    {
      conflictCount ← conflictCount + 1
      return
    }
  }
}
Senão // round < roundMax, ainda percorrendo os vértices da figura
{
  De (0 até númeroConflitos)
  {
    Se (Um dos elementos do par dos conflitos é o daVez, e o outro do par é maior do
que ele)
    {
      Se (roundMax == 5 / se for uma execução de pentágono)
      {
        calculaFigura(round + 1, 5, primeiro, elemento do par de conflitos)
      }
      Senão se (roundMax == 3 / se for uma execução de triângulo)
      {
        calculaFigura(round + 1, 3, primeiro, elemento do par de conflitos)
      }
    }
  }
}
```

Como é possível perceber, a função acima pode ser utilizada mais de uma vez, e é exatamente essa a ideia. Inicialmente, quando batemos o olho pela primeira vez no problema, a ideia seria percorrer sempre o vetor de conflitos verificando se havia conflito entre três heróis de modo que estes fechavam um triângulo, conforme o enunciado. Porém, ao percebermos que o triângulo se trata de um grafo cíclico, a ideia foi buscar por triângulos de forma recursiva, ou seja, a partir de um herói (primeiro) você sai percorrendo os caminhos possíveis

(arestas de conflitos) até encontrar novamente o mesmo herói inicial sem passar por uma mesma aresta mais de uma vez. Com isso, a ideia é que se o algoritmo encontrou um triângulo em alguma chamada, então ele incrementa o número de conflitos, do contrário, mantém o número atual. Ao sair da função o número da estimativa de conflitos percorrendo determinado caminho terá sido armazenada e, com isso, será possível perceber se é válido descer naquele ramo ou não.

Levando em consideração as técnicas apresentadas no parágrafo anterior, como também nos foi solicitado o desenvolvimento de uma nova Função Limitante, que seja melhor do que a fornecida no enunciado, então a ideia foi juntar a quantidade mínima de conflitos possíveis de um triângulo com a quantidade mínima de conflitos de um pentágono, pois isso acaba fazendo com que o algoritmo percorra menos caminhos ainda, uma vez que ele identifica, possivelmente, mais conflitos de antemão. A ideia para encontrar o pentágono é literalmente a mesma utilizada para encontrar o triângulo, e a única diferença é que quando o usuário estiver utilizando a nossa função, nosso algoritmo irá chamar as duas sequencialmente, ou seja, a que calcula os possíveis conflitos de triângulos e a que calcula os possíveis conflitos de pentágonos.

Por fim, tendo realizado todos esses passos, podemos finalizar nossa modelagem de *Branch&Bound*, implementando em uma mesma função, todos os algoritmos apresentados previamente, ou seja: um Backtracking que possua cortes tanto de viabilidade quanto de otimalidade, de modo a reduzir fortemente a quantidade de nós percorridos na árvore total de soluções.

3. Detalhes da implementação

Nesta seção, comentamos sobre alguns pontos importantes de como implementamos a solução para o problema apresentado, escolhemos a linguagem C para tal.

Ao compilar o programa com **make** digitar **./separa** no terminal, o programa entende que por default executará o *Branch&Bound* com cortes de viabilidade e otimalidade ativos e com a nossa Função Limitante desenvolvida. Tratamos a entrada da linha de comando com **getopt**, habilitando e desabilitando flags caso o usuário digite **-a**, **-f**, ou **-o**.

```
while ((option = getopt(argc, argv, "afo")) != -1) {
       switch (option) {
           case 'a': // -a digitado
               funcaoLimitanteFlag = 1;
```

```
                break;
            case 'f': // -f digitado
                viabilidadeFlag = 0;
                break;
            case 'o': // -o digitado
                otimalidadeFlag = 0;
                break;
            default:
                fprintf(stderr, "Atributos possiveis: %s -a -f -o\n",
argv[0]);
                exit(1);
        }
    }
```

Para a estrutura dos heróis, fizemos as seguintes structs:

```
typedef struct sets_t {
    int *i; // Herói 'i' da associação
    int *j; // Herói 'j' da associação
}sets_t;

typedef struct input_t {
    int hNum; // Número de heróis
    int cNum; // Número de conflitos
    int aNum; // Número de afinidades
    sets_t *cPairs; // Conjunto dos pares de conflitos
    sets_t *aPairs; // Conjunto dos pares de afinidades
}input_t;
```

Lemos todas da maneira como especificado no enunciado e as guardamos nesses conjuntos, que estão todos dentro da variável:

```
input_t *input = malloc(sizeof(input_t));
```

Como dito antes, para realizar o *Branch&Bound*, juntamos todas as funções desenvolvidas até então (descritas anteriormente), e adaptamos para cada combinação de parâmetros da linha de comando, a chamada ficou assim:

```
void separaGrupos(int l, int *x, int numH, int group[], int groupOpt[],
input_t *inp, int *qNodo, int a, int o, int f);
```

Sendo:

```
Opções de entrada:
• l: herói da vez;
```

```
• x: quantidade "ótima" de conflitos;
• numH: número total de heróis;
• group[]: vetor com os heróis do grupo com o 1 herói;
• groupOpt[]: vetor com os heróis do grupo com o 1 herói do caso
"ótimo"
• inp: input que contém os vetores de afinidades e conflitos
• qNodo: Quantidade de nodos percorridos;
• a == 0 -> default (usa a nossa), o == 0 -> sem corte de
otimalidade e f == 0 -> sem corte de viabilidade;
```

Dentro da função separaGrupos, colocamos várias verificações para sabermos se habilitamos os cortes selecionados e a Função Limitante selecionada seriam ligados, esses são pedaços dos códigos citados anteriormente.

Aqui um exemplo de quando o usuário entra com (**./separa -o**), dentro do separaGrupos, uma parte específica que é rodada:

```
if(f && !o) {
            for (int j = 0; j < inp->aNum; j++) {
                if((inp->aPairs->i[j] == l+1) && (inp->aPairs->j[j] <
l+1)) {
                    caminho = group[inp->aPairs->j[j] - 1];
                    restricao = 1;
                } else if((inp->aPairs->j[j] == l+1) &&
(inp->aPairs->i[j] < l+1)) {
                    caminho = group[inp->aPairs->i[j] - 1];
                    restricao = 1;
                }
            }
        }
```

4. Exemplos

Agora, pegamos alguns exemplos de execuções:

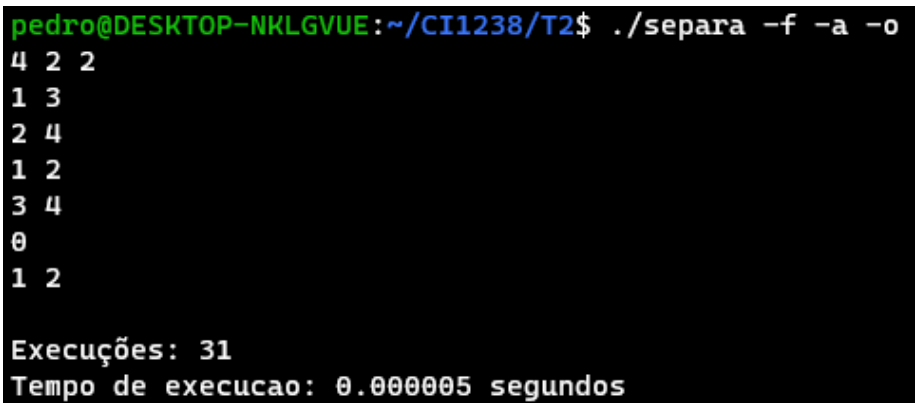

Figura 4: Exemplo 1 - Todos os cortes desabilitados.

Já com análise de tempo e execuções, podemos ver que ao executar o algoritmo sem nenhum corte, ele percorrerá a árvore toda de possibilidades, sendo n = 4, a árvore tem tamanho $2^{n+1} - 1$.

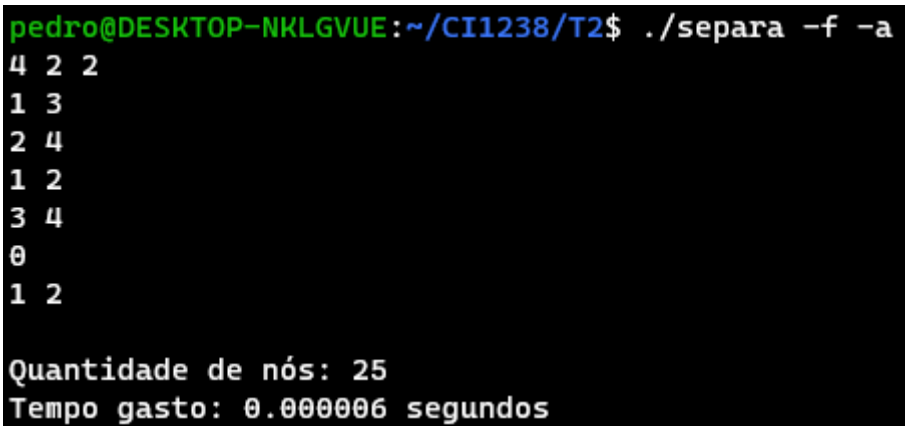

Figura 5: Exemplo 1 - Cortes por Otimalidade habilitados, com a função do enunciado.

Aqui podemos perceber que quando ligamos os cortes por otimalidade as execuções são reduzidas e também o tempo total de execução, produzindo o mesmo resultado mas de maneira mais eficiente.

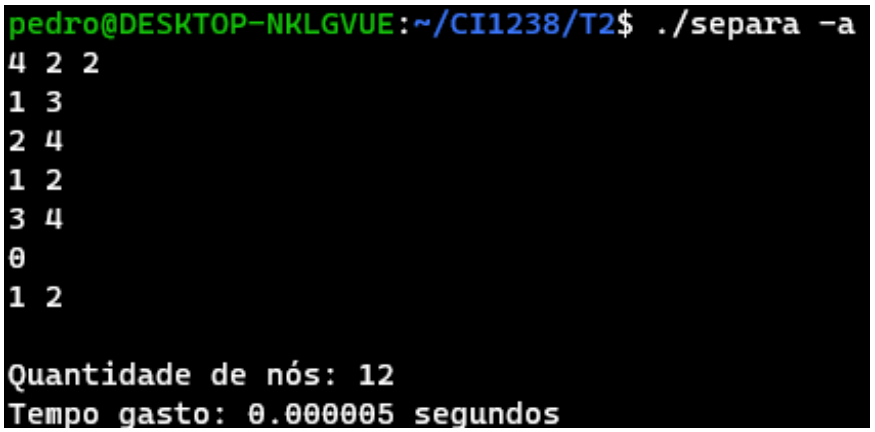

Figura 6: Exemplo 1 - Cortes por viabilidade e otimalidade ativos, com a função do enunciado.

Ainda no mesmo exemplo, quando ligamos o *Branch&Bound* de fato, com as duas verificações e cortes, podemos ver que as execuções e o tempo também foi reduzido consequentemente.

Agora outro problema:

```
pedro@DESKTOP-NKLGVUE:~/CI1238/T2$ ./separa -f -a -o
3 3 0
1 2
1 3
2 3
1
1 2

Execuções: 15
Tempo de execucao: 0.000003 segundos
```

Figura 7: Exemplo 2 - Todos os cortes desabilitados

Podemos ver que a lógica se repete neste outro exemplo.

```
pedro@DESKTOP-NKLGVUE:~/CI1238/T2$ ./separa -a -o
3 3 0
1 2
1 3
2 3
1
1 2

Execuções: 15
Tempo de execucao: 0.000003 segundos
```

Figura 8: Exemplo 2 - Corte por viabilidade habilitado

Agora um caso curioso, mesmo ligando o corte por viabilidade o resultado será o mesmo? **Sim**, pois neste exemplo podemos ver que o número de afinidades é **zero**, logo não cai em nenhuma das verificações se rompeu restrição dos com afinidades estarem no mesmo grupo.

```
pedro@DESKTOP-NKLGVUE:~/CI1238/T2$ ./separa -a
3 3 0
1 2
1 3
2 3
1
1 2

Quantidade de nós: 13
Tempo gasto: 0.000006 segundos
```

Figura 9: Exemplo 2 - Cortes por viabilidade e otimalidade ativos, com a função do enunciado.

Já quando ligamos o *Branch&Bound*, podemos perceber uma grande queda no tempo de execução.

Agora um último exemplo:

Figura 10: Exemplo 3 - Todos os cortes desabilitados

Esse já é um pouco maior, priorizamos em testar o máximo de conflitos possível, nesse caso ainda realizando mais de 8 mil execuções para achar o ótimo.

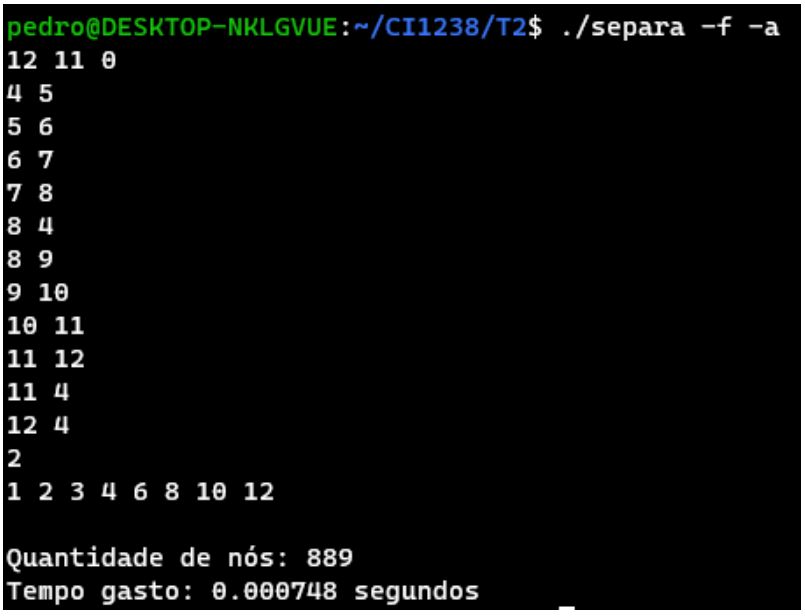

Figura 11: Exemplo 3 - Corte por otimalidade ativo, com a função do enunciado

Aqui podemos ver em como o corte por otimalidade é impactante em instâncias com um alto número de conflitos, ligando com a Função Limitante do enunciado já temos uma redução de 10 vezes no número de execuções.

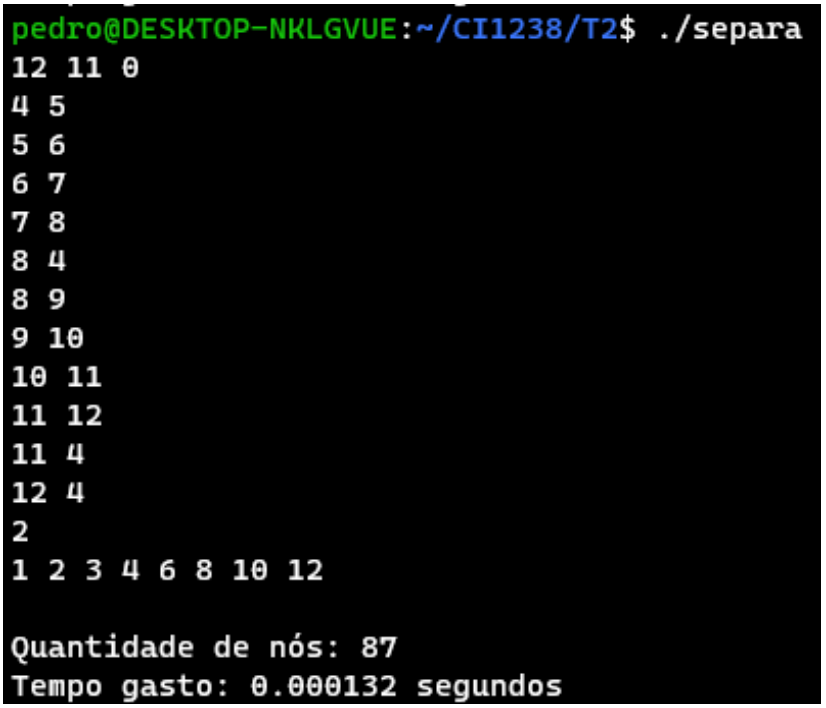

Figura 12: Exemplo 3 - Cortes por viabilidade e otimalidade ativos, com a nossa função limitante proposta

5. Análise do uso das funções limitantes

Como podemos perceber dos exemplos da seção anterior, o uso das funções limitantes influencia fortemente na redução de custos. Para o *Branch&Bound* elas são essenciais, pois não existe um bom corte de otimalidade sem uma boa Função Limitante.

A seguir, para exemplificar e fornecer uma aproximação do quão melhor a nova solução proposta é em comparação com a solução proposta pelo docente, nós executamos o programa com a mesma instância de problema para os dois casos. Os resultados obtidos foram:

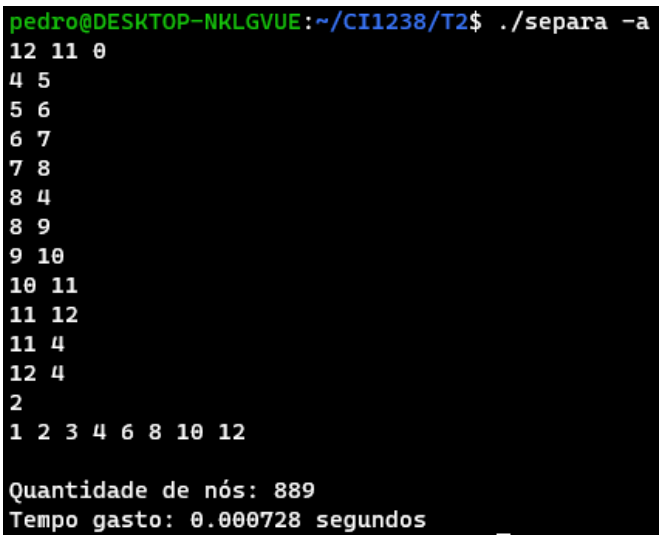

Figura 13: Chamada com a função limitante fornecida no enunciado.

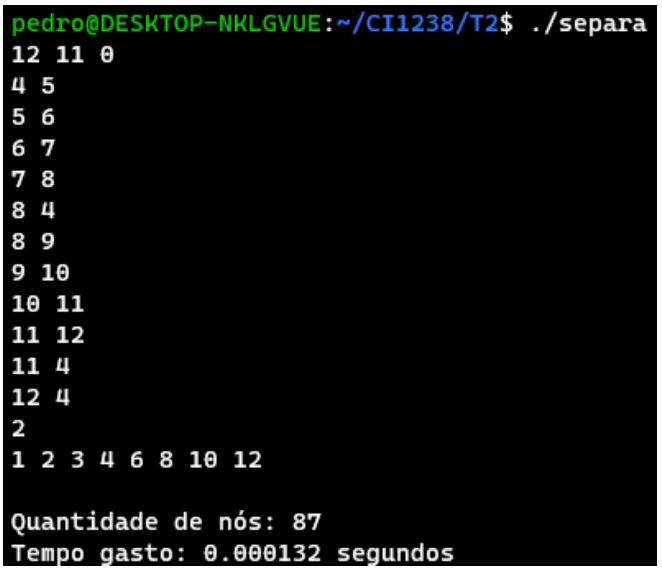

Figura 14: Chamada com a nova solução proposta.

Então, como é possível perceber, a nossa função acabou percorrendo um número significativamente menor de nodos ao buscar a solução ótima para dada instância do problema, sendo que isso também acabou diminuindo consideravelmente o tempo gasto para encontrar a mesma (solução ótima). Isso acontece porque nesse caso o número de triângulos e pentágonos é razoavelmente balanceado, de modo que encontrar os pentágonos e os triângulos corta mais do que encontrar apenas triângulos.