Resolução do problema EP01

Geraldo Rodrigues de Melo Neto
Gustavo Duarte Ventino
Maria Luisa Gabriel Domingues
Pedro de Araújo Ribeiro
Lucas Marques Pinho Tiago

O Problema:

Um grupo de amigos foi a uma viagem. Na volta, alguns deixaram de ser amigos, mas ocorreu que as despesas da viagem não foram divididas igualmente. Agora é necessário uma solução computacional para analisar de há como resolver o desbalanço via contato somente entre as pessoas que continuaram amigas.

Entradas:

A entrada consiste em, nessa ordem:

- Número de pessoas **(2 <= N <= 10000)** e amizades restantes **(0 <= M <= 50000)**.
- Valor a dever **(< 0)** ou a receber **(> 0)**, um por cada pessoa escrita em N .
- Pares de valores x,y **(0 <= x < y <= N-1)** que representam quais pessoas ainda possuem amizade entre si.

Saídas:

O programa deve responder **“POSSIBLE”** para casos onde é possível quitar as diferenças só entre os ainda amigos, e **“IMPOSSIBLE”** caso contrário.

Nossa Resolução:

Decidimos resolver esse problema com Grafos, tendo em vista que se torna mais fácil de fazer comunicar somente as pessoas amigas via busca, pois a conexão entre elas se dá estruturalmente no grafo.

Definimos dois TADs nos quais estão, respectivamente, classes presentes: uma para o grafo e outra para os vértices do grafo.

Nossa Resolução:

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, além do número de vértices e arestas.
  - De métodos, contém Getters e Setters básicos para cada atributo, além de métodos de busca (tanto em largura quanto profundidade) que auxiliaram na conclusão de que o grafo é ou não bipartido, e de reset de marcações (de cor e visita) para efetuar ambos métodos de busca.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, além do valor em dinheiro que carrega e sua identificação (dado no input)
  - De métodos, contém Getters e Setters básicos para cada atributo, além de métodos como addToAdjacency(), que inserem vértices no array de adjacência, gerando novas arestas.

```cpp
#include <vector>
class Vertex
{
private:
  int id;
  int value;
  std::vector<Vertex*> adjacency;
public:
  //Construtores:
  Vertex();
  Vertex(int id, int value);
  //Getters:
  int getId();
  int getValue();
  std::vector<Vertex*> getAdjacency();
  //Setters:
  void setId(int id);
  void setValue(int value);
  //Adiciona o vertice v na lista de adjacencia
  void addToAdjacency(Vertex *v);
  //Printa as informacoes de id e valor do vertice atual
  void print();
```

```cpp
#include "Vertex.hpp"
class Graph{
private:
  std::vector<Vertex*> vertices;
  int size;
  int edges;
public:
  //Construtores:
  Graph();
  Graph(int size, int edges);
  //Getters
  int getSize();
  //Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph();
  //Adiciona novo vertice na lista de vertices:
  void addVertex(Vertex* v);
  //Dado um id retorna o vertice naquela posição:
  Vertex* getVertex(int id);
  //Imprime todo o grafo:
  void print();
  //Retorna a soma entre dívidas e excedentes dos amigos
  int dfs(Vertex* v, bool* cor);
```

Método para resolução do problema:

Consiste em navegar cada parte conectada do grafo por busca em profundidade, marcando a visita em cada vértice através de um vetor de valores Booleanos, cada posição representando um vértice do grafo.

Para cada vértice visitado, se acumula o valor que aquele vértice guarda junto com o valor de seu adjacente, buscado via recursão. Caso o vértice analisado não tenha adjacente ou todos os seus adjacentes já tenham sido visitados, o método retorna somente o valor daquele vértice

```
21    //Retorna a soma entre dívidas e excedentes dos amigos
22    int dfs(Vertex* v, bool* cor);
```

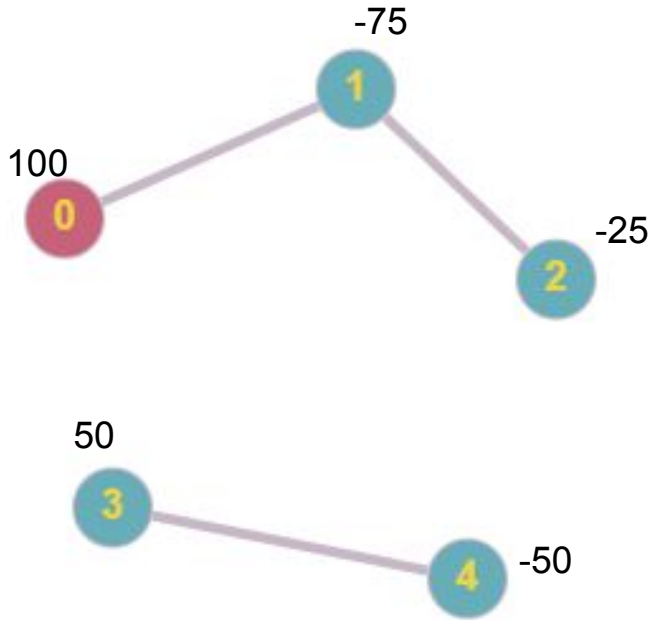
Método de resolução do problema:

O método usado verifica a soma entre dívidas e excedentes entre as pessoas ainda amigas, mas não verifica todos os grupos isolados.

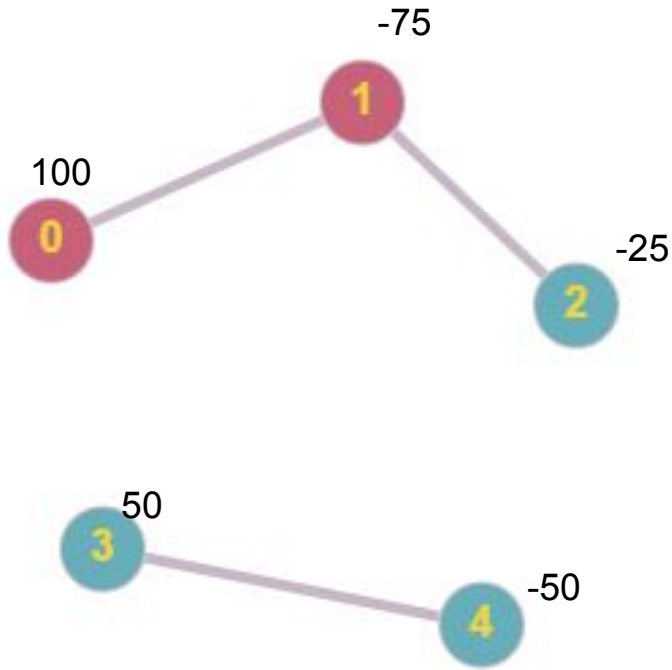
Por isso, foi feito um loop na main que faz com que a busca em profundidade passe por cada grupo isolado, e assim que achar um onde o resultado do seu retorno não é zero, já gera “IMPOSSIBLE” (caso contrário retorna “POSSIBLE”) de output.

```
17  for (int i = 0; i < gSize; i++){
18    if (Vpintado[i] == false){
19      resultado = g->dfs(g->getVertex(i), Vpintado);
20      if (resultado != 0){
21        std::cout << "IMPOSSIBLE";
22        return 0;
23      }
24    }
25  }
```

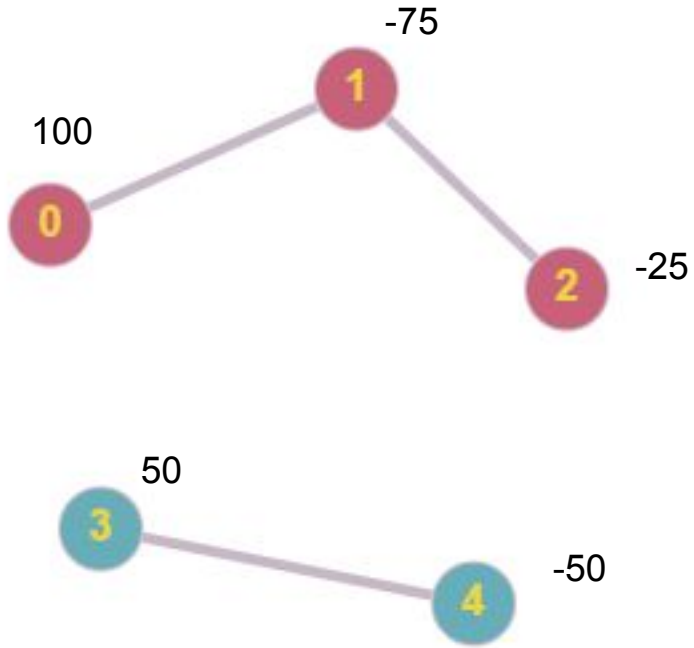
```
67  int Graph::dfs(Vertex* v, bool* cor)
68  {
69      cor[v->getId()] = true;
70      std::vector<Vertex*> adj = v->getAdjacency();
```



```
71      for (auto i : adj)
72      {
73          if (!cor[i->getId()])
74          {
75              return (Graph::dfs(i, cor) + v->getValue());
76          }
77      }
78
79      return v->getValue();
80  }
```

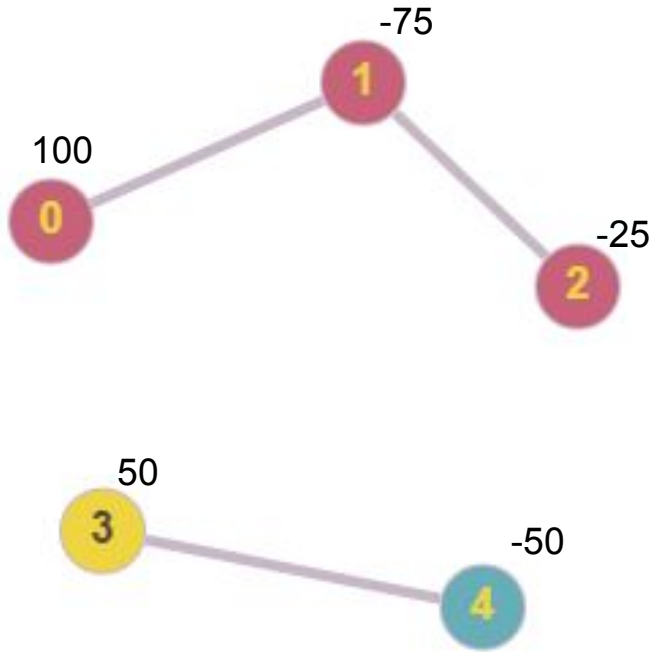


```
71      for (auto i : adj)
72      {
73          if (!cor[i->getId()])
74          {
75              return (Graph::dfs(i, cor) + v->getValue());
76          }
77      }
78
79      return v->getValue();
80  }
```



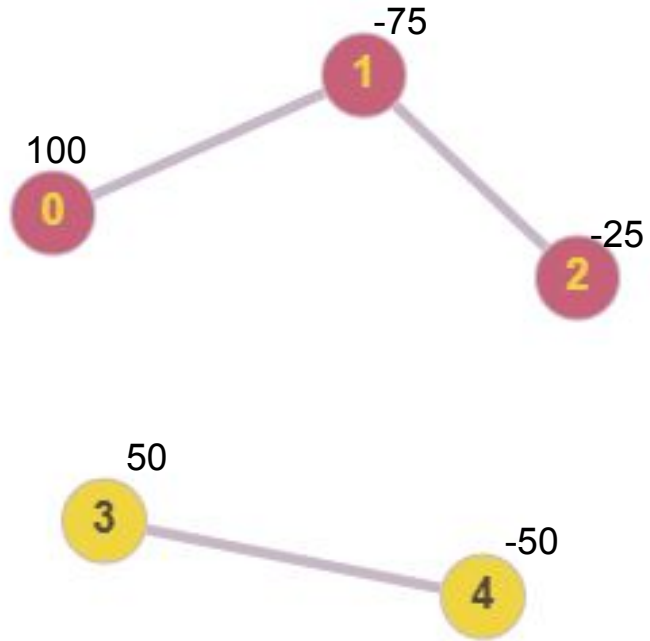
Resultado do grupo 1: 100 + (-75) + (-25) = 0 OK

```
71      for (auto i : adj)
72      {
73          if (!cor[i->getId()])
74          {
75              return (Graph::dfs(i, cor) + v->getValue());
76          }
77      }
78
79      return v->getValue();
80  }
```



Resultado do grupo 1: 100 + (-75) + (-25) = 0 OK

```
71      for (auto i : adj)
72      {
73          if (!cor[i->getId()])
74          {
75              return (Graph::dfs(i, cor) + v->getValue());
76          }
77      }
78
79      return v->getValue();
80  }
```



Resultado do grupo 1: 100 + (-75) + (-25) = 0 OK
Resultado do grupo 2: 50 + (-50) = 0 OK

OUTPUT: “POSSIBLE”

Referências:

Imagens de grafos tiradas de: https://graphonline.ru/en/#

Repositório com os códigos fonte: https://replit.com/@Ventinos/TG-EP01?v=1