# Resolução do problema EP04

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

# O Problema:

A polícia está tentando desmantelar uma rede de criminosos, essa rede de criminosos está conectada em forma de grafos, de modo que é possível alcançar todos os pontos por pelo menos um caminho. O plano da polícia é identificar um elo fraco, um elo fraco é qualquer ligação que se removida impede pelo menos um criminoso de se comunicar com outro na rede.

# Entradas:

O input é composto por vários casos de teste, em que a primeira linha de cada um desses casos possuem dois números, N e M em que:

- N é o número de criminosos.
- M é o número de conexões diretas entre esse criminosos.

As linhas seguintes devem possuir dois criminosos, **a** e **b,** indicando que **a** se conecta diretamente com **b.**

A última linha da entrada deve conter os números ‘0 0’,
indicando o final do input.

# Saídas:

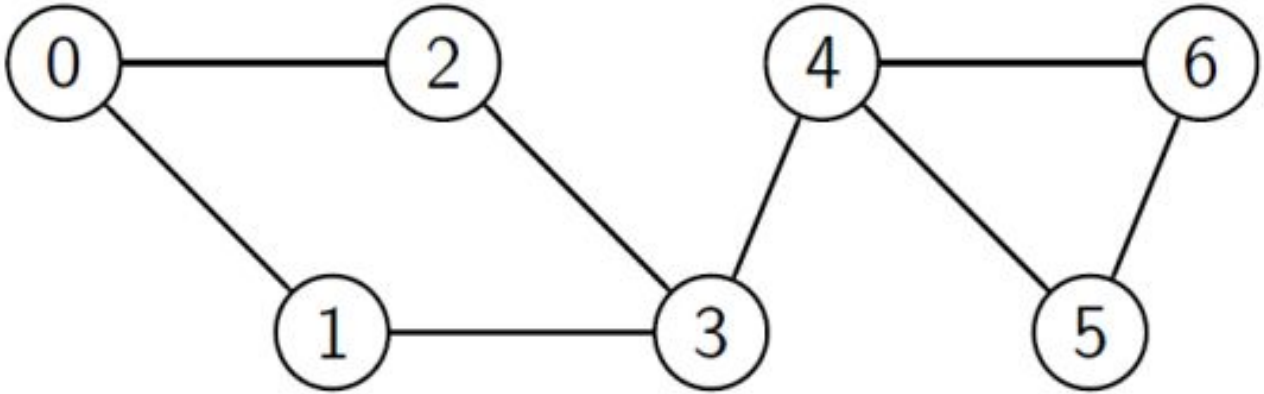
O programa deve fornecer um output para cada um dos casos teste do input, ou seja, o programa pode ter um ou vários outputs.

Se o programa conseguir encontrar elos fracos, primeiro ele mostra o número de elos fracos na rede, em seguida ele irá mostrar quais são esses elos fracos em ordem crescente.

Se o programa não encontrar elos fracos, ele irá apenas mostrar o número ‘0’ indicando que não há elos fracos.

# Exemplo:

| Entrada: | Saída: |
| --- | --- |
| 7 8 | 1 3 4 |
| 0 1 | |
| 0 2 | |
| 1 3 | |
| 2 3 | |
| 3 4 | |
| 5 4 | |
| 6 4 | |
| 5 6 | |
| 0 0 | |



# 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, o método readGraph() que realiza a leitura e criação do grafo e o método dfsBridges() que realiza uma busca por profundidade, armazenando as pontes do grafo.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, o valor do vértice (dado no input), seu id e se o vértice é seguro.
  - De métodos, contém Getters e Setters básicos para cada atributo, além de métodos como addToAdjacency(), que inserem vértices no array de adjacência.
- Sort.hpp: Não contém atributos
  - Possui quatro métodos estáticos que recebem um vetor de duplas de inteiros e os ordena em ordem crescente usando o método QuickSort.

```cpp
#include "Vertex.hpp"
#include <iostream>
#include <fstream>

class Graph
{
private:
  std::vector<Vertex*> vertices;
  int size;
public:
  //Construtores:
  Graph();
  Graph(int size);
  //Getters e Setters
  int getSize();
  void setSize(int size);
  //Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(int n, int m);
  //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();
  void printVertices();
  void dfsBridges(int u, int *dfs_numbercounter, std::vector<int>&
dfs_num,std::vector<int>& dfs_low, std::vector<int>& dfs_parent,std::vecto
articulacao,int *dfsRoot, int *rootChildren, std::vector<std::pair<int,int
*pontes);
};
```

```cpp
#include <vector>

class Vertex
{
private:
  int id;
  bool safe;
  bool marked;
  std::vector<Vertex*> adjacency;
public:
  //Construtores:
  Vertex();
  Vertex(int id);

  //Getters:
  int getId();
  bool isSafe();
  bool isMarked();
  std::vector<Vertex*> getAdjacency();

  //retorna o primeiro adjacente não colorido do vertice
  Vertex* getAdjacencyNotColored();
  //Setters:
  void setId(int id);
  void mark();
  void unmark();
  void setSafe(bool safe);
  //Adiciona o vertice v na lista de adjacencia
  void addToAdjacency(Vertex *v);
  //Printa as informacoes de id e valor do vertice atual
  void print();
  //Printa a lista de adjacencia do vertice atual
  void printAdjacency();
};
```

# Método para resolução do problema:

- Leitura do tamanho do grafo, das entradas e preenchimento do grafo.
  - Usamos para isso a função Graph* Graph::readGraph() da nossa classe de Grafo descrita em Graph.hpp e Graph.cpp, essa função é chamada para cada um dos casos teste.
- A função realiza 3 etapas:
  - Adiciona os vértices de 0 até n.
  - Realiza as conexões entre os vértices.
  - Retorna o Grafo com todos os vértices conectados.

```
// Faz toda a leitura de entrada e cria o grafo:
Graph *Graph::readGraph(int n, int m) {
  int entry1, entry2, k;
  Graph *g;
  Vertex *v1, *v2;
  g = new Graph();

  // cria vertices com id baseando-se em n
  for (int i = 0; i < n; i++) {
    v1 = new Vertex(i);
    g->addVertex(v1);
  }

  // conecta todos os vertices que deverão ser conectados
  for (int i = 0; i < m; i++) {
    std::cin >> entry1;
    std::cin >> entry2;

    v1 = g->getVertex(entry1);
    v2 = g->getVertex(entry2);
    v1->addToAdjacency(v2);
    v2->addToAdjacency(v1);
  }
  // retorna o grafo com todos os vertices
  return g;
}
```

```
// Adiciona novo vertice na lista de vertices:
void Graph::addVertex(Vertex *v) { vertices.push_back(v); }

// Dado um id retorna o vertice naquela posição:
Vertex *Graph::getVertex(int id) {
  for (Vertex *i : vertices)
    if (i->getId() == id)
      return i;
  return NULL;
}
// Adiciona o vertice v na lista de adjacencia:
void Vertex::addToAdjacency(Vertex *v) { adjacency.push_back(v); }
```

# Nossa Resolução:

Como fizemos para encontrar as pontes e colocá-las em ordem?

O programa realiza o dfs visto em sala de aula para buscar as articulações em um grafo, mas com algumas pequenas alterações para que a função receba um vetor de duplas de inteiros e armazene as pontes que ele encontrar nesse vetor.

Como são vários casos testes, ordenamos cada um dos vetores de pontes usando o Quicksort e depois armazenamos em um outro vetor, para que depois possamos mostrar os resultados de cada um dos casos testes na formatação exigida.

```
  std::cin >> n;
  std::cin >> m;
  while (n != 0 || m != 0) {
    std::cout << n << m << "\n";
    //realiza leitura do grafo
    Graph *g = Graph::readGraph(n, m);

    //prepara o grafo para o dfs
    for (int i = 0; i < n; i++) {
      dfs_num.push_back(-1);
      dfs_low.push_back(0);
      dfs_parent.push_back(-1);
      articulacao.push_back(0);
    }
    //realiza o dfs e armazena as pontes
    for (int u = 0; u < n; u++) {
      if (dfs_num.at(u) == -1) {
        dfsRoot = u;
        rootChildren = 0;
        g->dfsBridges(u, &dfs_numbercounter, dfs_num, dfs_low, dfs_parent,
                      articulacao, &dfsRoot, &rootChildren, &pontes);
        articulacao.at(u) = (rootChildren > 1);
      }
    }
    //ordena a lista de pontes e armazena em um vetor de lista de pontes
    Sort::quicksort(pontes, 0, pontes.size() - 1);
    respostas.push_back(pontes);

    //limpa os vetores para a leitura do próximo grafo
    pontes.clear();
    dfs_num.clear();
    dfs_low.clear();
    dfs_parent.clear();
    articulacao.clear();

    std::cin >> n;
    std::cin >> m;
  }
```

```
  //printa todos os dados do vetor de listas de pontes
  for (int i = 0; i < respostas.size(); i++) {
    std::cout << "\n" << respostas[i].size() << " ";
    for (int j = 0; j < respostas[i].size(); j++) {
      std::cout << respostas[i][j].first << " " << respostas[i][j].second
                << " ";
    }
  }
```

FIM

# Referências:

Repositório com os códigos fonte: https://replit.com/@Gezero/TG2-EP02

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