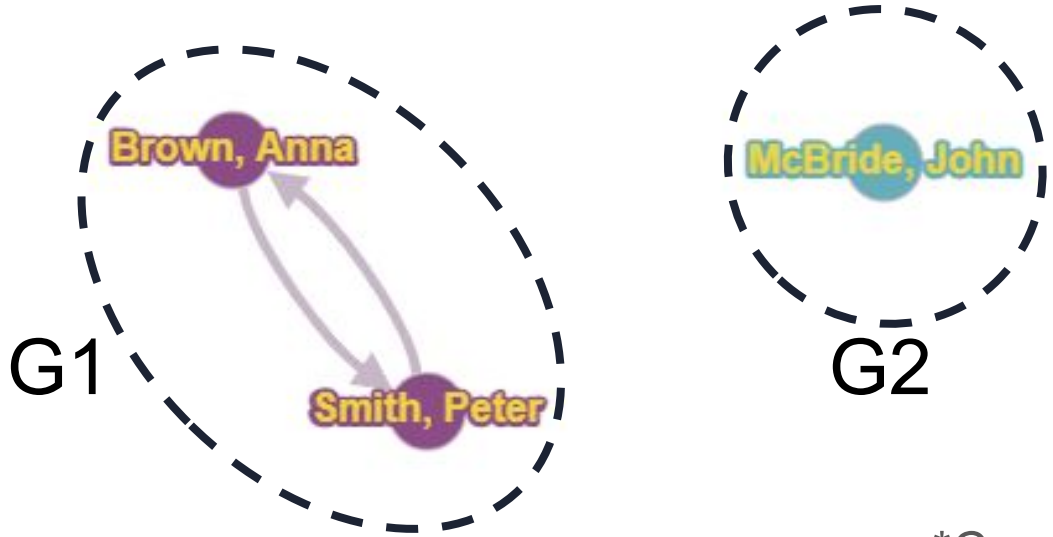
Resolução do problema EP03

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

O Problema:

Formar um número mínimo de grupos de trabalho estáveis, nos quais se envolvem somente pessoas que se confiam mutuamente.



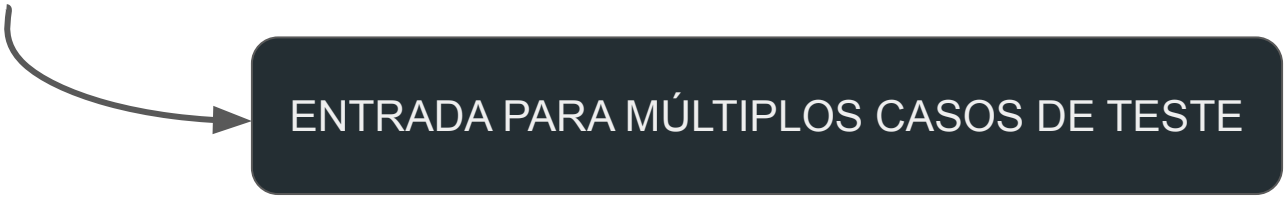
*Componentes fortemente conectados.

Entradas:

A entrada consiste em diversas verificações seguindo o modelo:

- Quantidade **P** de pessoas (1 < P < 1000);
- Quantidade **T** de conexões de confiança (0 ≤ m ≤ 999000);
- As próximas **P** linhas contêm o nome das pessoas envolvidas no problema;
- As próximas **2T** linhas contêm o nome das pessoas que formam uma relação;

Para encerrar a entrada devemos ter **P = T = 0** na leitura de estações e bombardeios.



Entradas:

```
3 2
McBride, John ➡ P1
Smith, Peter ➡ P2
Brown, Anna ➡ P3
Brown, Anna  } R1
Smith, Peter }
Smith, Peter } R2
Brown, Anna  }
```

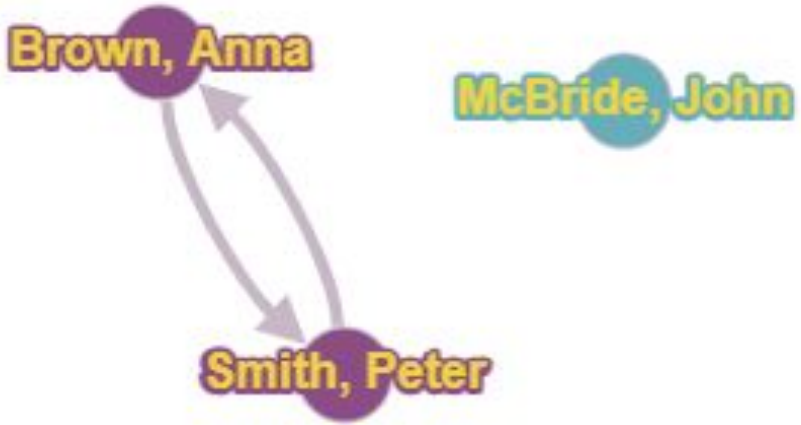
```
3 2
McBride, John ➡ P1
Smith, Peter ➡ P2
Brown, Anna ➡ P3
Brown, Anna  } R1
Smith, Peter }
McBride, John } R2
Smith, Peter  }
0 0
```

Saídas:

Deve-se retornar um número que representa o número mínimo de grupos estáveis, de acordo com as relações de confiança dadas na entrada.

OUTPUT:

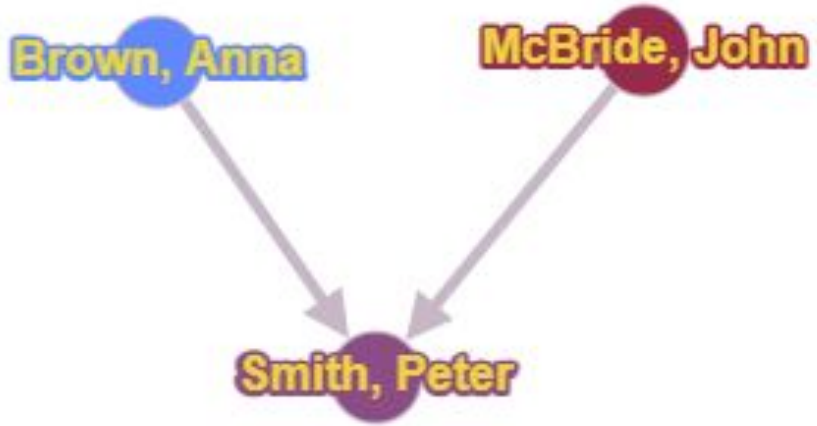
2



Exemplo:

OUTPUT:

3



Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar as relações de confiança como conexões direcionadas, e grupos estáveis como componentes fortemente conectados, onde todos os vértices envolvidos em cada um dos componentes é capaz de alcançar, em confiança, algum outro envolvido do grupo.

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

Nossa Resolução:

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, além do número de vértices.
  - De métodos, contém Getters e Setters básicos para cada atributo, um método para a geração do grafo e um método para a verificação da quantidade de componentes fortemente conectados do grafo.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, o valor do vértice (dado no input), seu id, a marcação de vértices e um contador de .
  - 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>
#include <string>
#include <iostream>

class Vertex{
private:
  int id;
  std::vector<Vertex*> adjacency;
  bool check;
  int circulo;
  std::string value;
public:
  Vertex();
  Vertex(int id, std::string value);
  //Getters:
  int getId();
  int getCirculo();
  std::string getValue();
  bool isChecked();
  std::vector<Vertex*> getAdjacency();
  //Setters:
  void setId(int id);
  void setCirculo(int circulo);
  void setValue(std::string value);
  void checkVertex();
  void uncheckVertex();
  //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();
  //funcoes auxiliares:
  bool removeAdjacency(int id);
```

```cpp
#include "Vertex.hpp"
#include <stack>

class Graph
{
private:
  std::vector<Vertex*> vertices;
  int size;
public:
//Construtores:
  Graph();
//Faz toda a leitura de entrada e cria o grafo:
  static Graph *readGraph(int n, int m, Graph *inv);
//Adiciona novo vertice na lista de vertices:
  void addVertex(Vertex* v);
  void setSize(int size);
//Dado um id retorna o vertice naquela posição:
  Vertex *getVertex(std::string value);
  Vertex *getById(int id);
//GetVertices:
  int getSize();
  std::vector<Vertex*> getVertices();
//Imprime todo o grafo:
  void print();
//Resolução do problema:
  static int solve(Graph* g1, Graph* g2);
  void dfs(int id);
//reset
  void resetMark();
};
```

Método para resolução do problema

Para resolver o problema, usamos duas funções na seguinte ordem:

1- Graph* Graph::readGraph(int n, int m, Graph *inv);

2- int Graph::solve(Graph* g1, Graph* g2);

A primeira é responsável por ler as entradas e montar um grafo e seu transposto.

A segunda é responsável pela resolução em si do problema.

- Faz a contagem das componentes fortemente conectadas por meio do algoritmo de Kosaraju adaptado ao nosso TAD.

Leitura de dados:

```
// Faz toda a leitura de entrada e cria o grafo:
// n = nroVertices,  m = nroArestas, inv = grafo transposto
Graph *Graph::readGraph(int n, int m, Graph *inv) {
  Graph *g = new Graph();
  g->setSize(n);
  inv->setSize(n);

  Vertex *v = NULL;
  Vertex *u = NULL;

  std::string name1, name2;

  // verifica entradas:
  if (n == 0 && m == 0)
    return NULL;

  // leitura dos vértices do grafo
  for (int i = 0; i < n; i++) {
    std::getline(std::cin, name1);
    v = new Vertex(i, name1);
    u = new Vertex(i, name1);
    // inserindo vertices no normal:
    g->addVertex(v);
    // inserindo vertices no transposto:
    inv->addVertex(u);
  }

  // conexão entre vértices
  for (int i = 0; i < m; i++) {
    std::getline(std::cin, name1);
    std::getline(std::cin, name2);

    // montando o grafo normal:
    v = g->getVertex(name1);
    u = g->getVertex(name2);
    v->addToAdjacency(u);

    // montando o grafo transposto:
    v = inv->getVertex(name1);
    u = inv->getVertex(name2);
    u->addToAdjacency(v);
  }

  return g;
}
```

Verificações de quantidade de componentes:

```
int Graph::solve(Graph *g1, Graph *g2) {
  int cnt = 0;

  g1->resetMark();
  g2->resetMark();

  for (int i = 0; i < g1->getSize(); i++) {
    if (g1->getById(i)->getCirculo() == 0) {
      cnt++;
      g1->dfs(i);
      g2->dfs(i);
      for (int j = 0; j < g1->getSize(); j++) {
        if (g1->getById(j)->isChecked() && g2->getById(j)->isChecked())
          if (g1->getById(j)->getCirculo() == 0) {
            g1->getById(j)->setCirculo(cnt);
            g2->getById(j)->setCirculo(cnt);
          }
      }
    }
    g1->resetMark();
    g2->resetMark();
  }
  return cnt;
}
```

```
void Graph::dfs(int id) {
  Graph::getById(id)->checkVertex();
  for (Vertex *i : getById(id)->getAdjacency()) {
    if (!i->isChecked())
      dfs(i->getId());
  }
}
```

MAIN:

```cpp
int main() {
  int n, m;
  std::string lixo;

  Graph *g = new Graph();
  Graph *inv;

  std::queue<int> queue;

  while (true) {
    std::cin >> n;
    std::cin >> m;
    std::getline(std::cin, lixo);

    inv = new Graph();

    g = Graph::readGraph(n, m, inv);

    if (g == NULL)
      break;

    queue.push(Graph::solve(g, inv));
  }

  while (!queue.empty()) {
    std::cout << queue.front() << std::endl;
    queue.pop();
  }

  return 0;
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Ventinos/TG-S04EP3#Graph.cpp

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