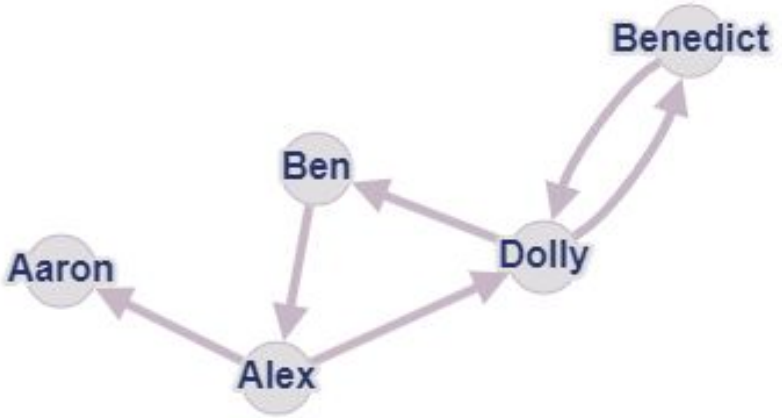
Resolução do problema EP02

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

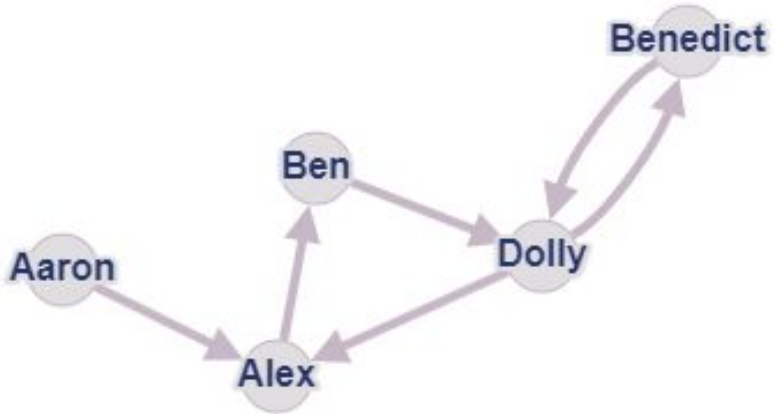
O Problema:

Para calcular o melhor círculo de chamadas para um cliente, escolhemos n pessoas e m conexões para vermos cada um faz chamadas para outra pessoa seja ela diretamente ou indiretamente.

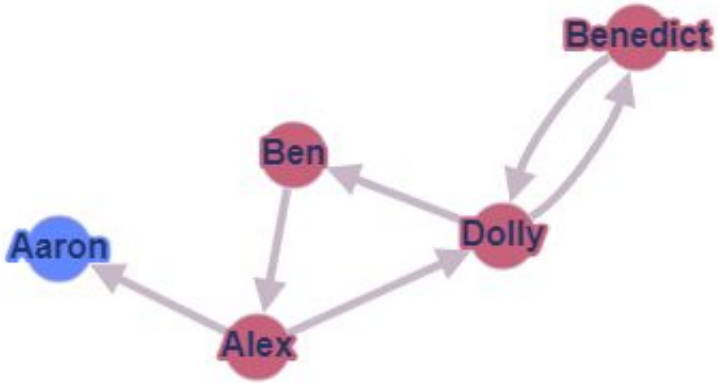
Exemplo:



Exemplo: (Grafo Transposto)



Exemplo: (Componentes Fortemente Conectados)



Entradas:

A entrada consiste em diversas verificações seguindo o modelo:

- Quantidade **n** de pessoas (1 < n < 25);
- Quantidade **m** de ligações;
- Próximas **m** linhas com dois nomes diferentes;

Para encerrar a entrada devemos ter que escrever **0 0** na verificação de **n** e **m**

Saídas:

O programa deve imprimir os círculos de chamada de cada teste da seguinte forma:

- “Calling circles for data set **x**” onde **x** é o número do teste;
- O conjunto de cada componente fortemente conectado do teste **x**

Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar a existência de uma ligação A para B como as adjacências de um grafo direcionado.

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, um método para a geração do grafo e um método para a verificação 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 e a marcação de vertices.
  - 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 "Vertex.hpp"
#include <fstream>
#include <queue>
class Graph
{
private:
  std::vector<Vertex*> vertices;
  int size;
  int edges;
public:
//Construtores, Getters e Setters:
  Graph();
  Graph(int size);
  Graph(int size, int edges);
  int getSize();
  int getEdge();
//Dado um id retorna o vertice naquela posição:
  Vertex* getVertex(int id);
  void setEdge(int edge);
  void setSize(int size);
//Faz toda a leitura de entrada e cria o grafo:
  static std::pair<Graph*,Graph*> readGraph(int n);
//Adiciona novo vertice na lista de vertices:
  void addVertex(Vertex* v);
//Imprime todo o grafo:
  void print();
//Verifica se a string (cidade) pertence à algum vertice do grafo
  bool verifyName(std::string name, int *vertex);
//descobre quais componentes são fortemente conectados
  static std::queue<std::string> verificaCirculo(Graph* g1, Graph* g2);
//realiza dfs e retorna o vetor de visitas
  void dfs(int n, bool vis[]);
};
```

```cpp
#include <vector>
#include <string>
class Vertex
{
private:
  int id;
  std::vector<Vertex*> adjacency;
  std::string name;
  int circulo;

public:
  //Construtores:
  Vertex();
  Vertex(int id);
  Vertex(int id,std::string city);
  //Getters:
  int getId();
  int getCirculo();
  std::string getName();
  std::vector<Vertex*> getAdjacency();
  //Setters:
  void setId(int id);
  void setCirculo(int value);
  void setName(std::string name);
  //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

Para resolver o problema, primeiro fizemos a leitura do grafo a partir da função readGraph, depois foi feita a verificação dos componentes fortemente conectados do grafo.

Vale ressaltar que a função readGraph já cria seu grafo transposto para ser usado na verificação de componentes fortemente conectados

Leitura de Dados:

```
122 bool Graph::verifyName(std::string city, int *vertex) {
123   for (auto i : vertices) {
124     if (i->getName().compare(city) == 0) {
125       *vertex = i->getId();
126       return true;
127     }
128   }
129   return false;
130 }
```

```
35 std::pair<Graph*,Graph*> Graph::readGraph(int n){
36   int vertex1 = 0, vertex2 = 0;
37   Vertex *v, *v2;
38   Graph *g1, *g2;
39   std::string name1, name2;
40   g1 = new Graph();
41   g2 = new Graph();
42   int size = 0;
43   for(int i = 0; i < n; i++){
44     std::cin >> name1;
45     std::cin >> name2;
46     if(!g1->verifyName(name1, &vertex1)){
47       v = new Vertex(size, name1);
48       v2 = new Vertex(size, name1);
49       g1->addVertex(v);
50       g2->addVertex(v2);
51       vertex1 = size;
52       size++;
53     }
54     if(!g1->verifyName(name2, &vertex2)){
55       v = new Vertex(size, name2);
56       v2 = new Vertex(size, name2);
57       g1->addVertex(v);
58       g2->addVertex(v2);
59       vertex2 = size;
60       size++;
61     }
62     g1->getVertex(vertex1)->addToAdjacency(g1->getVertex(vertex2));
63     g2->getVertex(vertex2)->addToAdjacency(g2->getVertex(vertex1));
64   }
65   g1->setSize(size);
66   g2->setSize(size);
67   return std::make_pair(g1, g2);
68 }
```

Verificação de componentes fortemente conectados:

```
66  std::queue<std::string> Graph::verificaCirculo(Graph *g1, Graph *g2) {
67    int cnt = 0, size = g1->getSize();
68    std::string s;
69    std::queue<std::string> v;
70    bool vis[size], vis2[size];
71    for (int i = 0; i < size; i++) {
72      vis[i] = false;
73      vis2[i] = false;
74    }
75    for (int i = 0; i < size; i++) {
76      if (g1->getVertex(i)->getCirculo() == 0) {
77        cnt++;
78        v.push(s);
79        s = "";
80        g1->dfs(i, vis);
81        g2->dfs(i, vis2);
82        for (int j = 0; j < size; j++) {
83          if (vis[j] && vis2[j])
84            if (g1->getVertex(j)->getCirculo() == 0) {
85              if (s == "")
86                s = s + g1->getVertex(j)->getName();
87              else
88                s = s + ", " + g1->getVertex(j)->getName();
89              g1->getVertex(j)->setCirculo(cnt);
90            }
91        }
92      }
```

```
93      for (int k = 0; k < size; k++) {
94        vis[k] = false;
95        vis2[k] = false;
96      }
97    }
98    v.push(s);
99    return v;
100 }
```

```
95  void Graph::dfs(int n, bool vis[]){
96    vis[n]=true;
97    for(auto i : getVertex(n)->getAdjacency()){
98      if(!vis[i->getId()])
99        dfs(i->getId(),vis);
100   }
101 }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Gezero/SO4EP02

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