

Resolução do problema EP06

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

O Problema:

Alice é uma treinadora de futebol que ocasionalmente leva sua equipe de futebol para explorar a Caveland (que pode ser modelada como um grafo não direcionado, não ponderado e conectado) para um evento especial.

A Caveland é bastante propensa a inundações, mas isso não impede que a Alice e sua equipe de futebol façam o que eles gostam. Você é o Bob, bom amigo da Alice. Você deseja garantir que a Alice e sua equipe de futebol estejam o mais seguros possível, informando a ela quais cruzamentos são mais seguros do que os demais. Você decide que um cruzamento é considerado seguro, quando independente do túnel atualmente inundado, Alice e sua equipe ainda possam sair desse cruzamento até a entrada da Caveland por um caminho não inundado.

Entradas:

A primeira linha contém um número inteiro positivo $2 \leq N \leq 10000$, que representa o número de vértices;

A segunda linha contém um número inteiro positivo $1 \leq M \leq \min(N(N-1)/2, 100000)$, que representa a quantidade de arestas;

As próximas linhas contêm as adjacências entre os vértices, ou seja os túneis da Caveland:

9	10
0	1
0	2
1	3
2	3
2	8

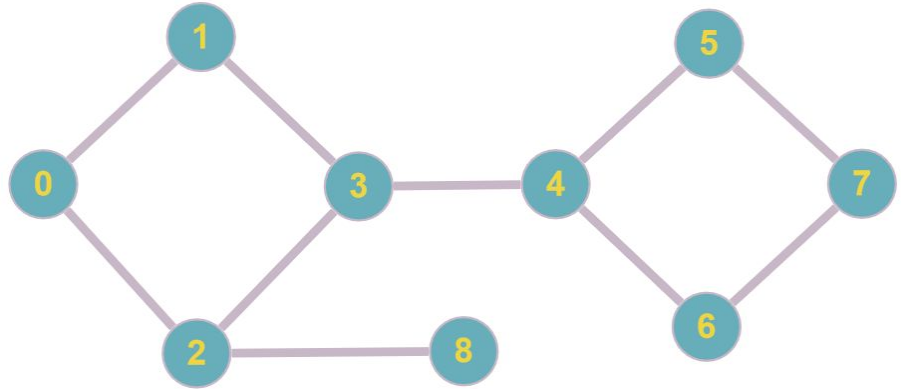
3	4
4	5
4	6
5	7
6	7

Visualizando as entradas:

Entradas:

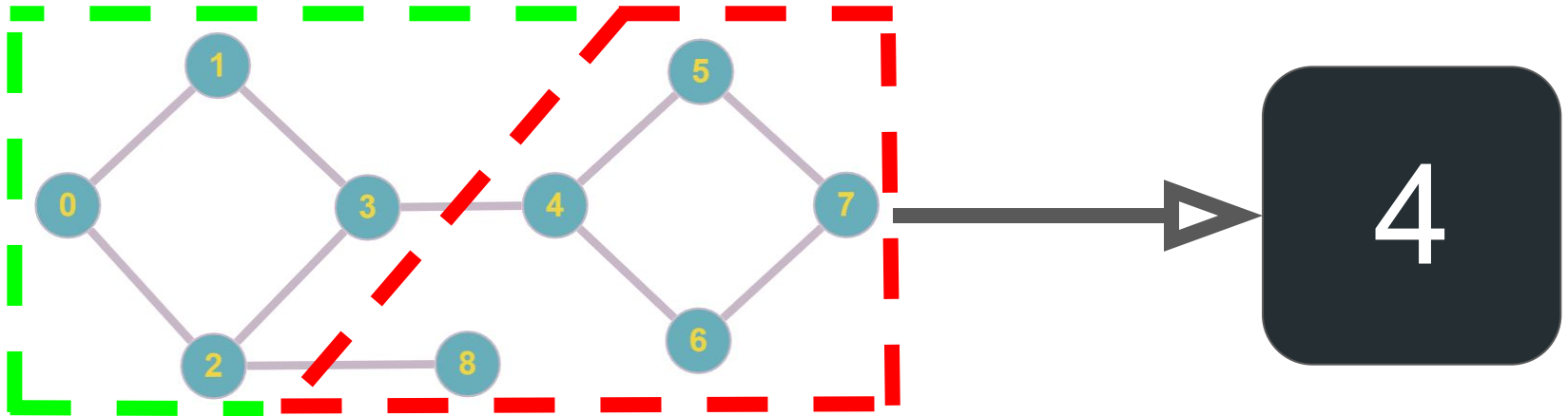
9	10	3	4
0	1	4	5
0	2	4	6
1	3	5	7
2	3	6	7
2	8		

Grafo:



Saídas:

A saída deve consistir em uma única linha contendo a quantidade de cruzamentos seguros.



Nossa Resolução (TAD):

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, e o número vértices.
 - De métodos, contém Getters e Setters básicos para cada atributo, o método de geração do grafo e leitura dos inputs, e funções para buscar as pontes e identificar os vértices seguros.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, não possui valor, apenas id e valores status booleanos marcado e seguro.
 - De métodos, contém Getters e Setters básicos para cada atributo.
- As implementações desses métodos estão descritas nos arquivos .cpp correspondentes.

```

1  #include <vector>
2
3  class Vertex{
4  private:
5      int id;
6      bool safe;
7      bool marked;
8      std::vector<Vertex*> adjacency;
9  public:
10     //Construtores:
11     Vertex();
12     Vertex(int id);
13     //Getters:
14     int getId();
15     bool isSafe();
16     bool isMarked();
17     std::vector<Vertex*> getAdjacency();
18     //retorna o primeiro adjacente não colorido do vertice
19     Vertex* getAdjacencyNotColored();
20     //Setters:
21     void setId(int id);
22     void mark();
23     void unmark();
24     void setSafe(bool safe);
25     //Adiciona o vertice v na lista de adjacencia
26     void addToAdjacency(Vertex *v);
27     //Printa as informacoes de id e valor do vertice atual
28     void print();
29     //Printa a lista de adjacencia do vertice atual
30     void printAdjacency();
31 };

```

```

1  #include "Vertex.hpp"
2  #include <iostream>
3  #include <fstream>
4
5  class Graph
6  {
7  private:
8      std::vector<Vertex*> vertices;
9      int size;
10 public:
11     //Construtores:
12     Graph();
13     Graph(int size);
14     //Getters e Setters
15     int getSize();
16     void setSize(int size);
17     //Faz toda a leitura de entrada e cria o grafo:
18     static Graph* readGraph(int *n);
19     //Adiciona novo vertice na lista de vertices:
20     void addVertex(Vertex* v);
21     //Dado um id retorna o vertice naquela posição:
22     Vertex* getVertex(int id);
23     //Imprime todo o grafo:
24     void print();
25     void printVertices();
26
27     void dfsBridges(int u, int *dfs_numbercounter, std::vector<int>&
dfs_num,std::vector<int>& dfs_low, std::vector<int>&
dfs_parent,std::vector<int>& articulacao,int *dfsRoot, int
*rootChildren, std::vector<std::pair<int,int>> *pontes);
28
29     int dfs(int e, std::vector<std::pair<int,int>> pontes);
30 };

```

Nossa Resolução:

Nossa Resolução consiste em duas etapas:

- Leitura do tamanho do grafo (Vértices e Arestas) e das entradas, além preenchimento do grafo e montagem das listas de adjacência.
 - Usamos para isso a função `Graph* Graph::readGraph()` da nossa classe de Grafo descrita em `Graph.hpp` e `Graph.cpp`.


```
Graph *Graph::readGraph() {
    int entry1, entry2, k;
    Graph *g;
    Vertex *v1, *v2;
    g = new Graph();
    // adiciona os vertices que não estão conectados a ninguém
    int m,n;
    std::cin >> n;
    g->size = n;
    std::cin >> m;
    // 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;
}
```

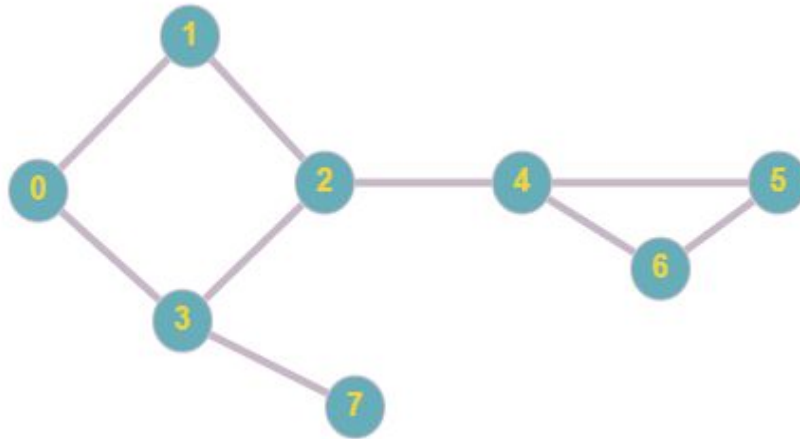
Nossa Resolução:

Vejamos um exemplo:

Input:

```
8 9
0 1
0 3
1 2
3 2
3 7
2 4
4 5
5 6
6 4
```

Grafo:



```
(0,0)
  (1,0)
  (3,0)
(1,0)
  (0,0)
  (2,0)
(2,0)
  (1,0)
  (3,0)
  (4,0)
(3,0)
  (0,0)
  (2,0)
  (7,0)
(4,0)
  (2,0)
  (5,0)
  (6,0)
(5,0)
  (4,0)
  (6,0)
(6,0)
  (5,0)
  (4,0)
(7,0)
  (3,0)
```

Nossa Resolução:

Agora que temos as nossas listas de adjacência montadas, podemos começar a resolver o problema procurando por pontes no grafo.

Nossa Resolução:

- Usamos a busca em profundidade do Hopcroft & Tarjan para determinação de pontes no grafo, e para descobrir quais pontos continuam alcançáveis apartir do vértice 0 quando essas pontes são bloqueadas.
 - Usamos para isso duas funções:
 - `void Graph::dfsBridges(...)` (**DFS HOPCROFT TARJAN QUE MARCA AS PONTES**).
 - `int Graph::dfs(int id, std::vector<std::pair<int, int>> pontes).`



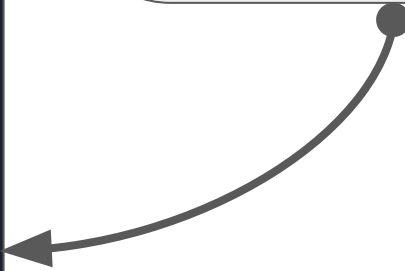
(DFS NORMAL QUE VERIFICA, CONSIDERANDO AS PONTES, QUAIS VÉRTICES PODEMOS ALCANÇAR NO GRAFO A PARTIR DA ENTRADA).

```

61 void Graph::dfsBridges(int u, int *dfs_numbercounter,
62   std::vector<int> &dfs_num,
63   std::vector<int> &dfs_low,
64   std::vector<int> &dfs_parent,
65   int *dfsRoot,
66   int *rootChildren,
67   std::vector<std::pair<int, int>>
68   *pontes) {
69   *dfs_numbercounter = *dfs_numbercounter + 1;
70   dfs_num.at(u) = *dfs_numbercounter;
71   dfs_low.at(u) = dfs_num.at(u);
72   for (auto v : getVertex(u)->getAdjacency()) {
73     if (dfs_num.at(v->getId()) == -1 /*unvisited*/) {
74       dfs_parent.at(v->getId()) = u;
75
76       if (u == *dfsRoot)
77         *rootChildren = *rootChildren + 1;
78
79       dfsBridges(v->getId(), dfs_numbercounter, dfs_num,
80         dfs_low, dfs_parent,
81         dfsRoot, rootChildren, pontes);
82
83       if (dfs_low.at(v->getId()) > dfs_num.at(u)) {
84         pontes->push_back(std::make_pair(v->getId(), u));
85       }
86
87       dfs_low.at(u) = std::min(dfs_low.at(u), dfs_low.at(v-
88         >getId()));
89     } else if (v->getId() != dfs_parent.at(u)) {
90       dfs_low.at(u) = std::min(dfs_low.at(u), dfs_low.at(v-
91         >getId()));

```

**AQUI USAMOS O DFS
HOPCROFT TARJAN
PARA IDENTIFICAR AS
PONTES.**



```

int Graph::dfs(int e, std::vector<std::pair<int, int>> pontes) {
    int u, ret = 0;
    std::stack<int> stack;
    stack.push(e);
    bool flag = false;

    while (!stack.empty()) {
        u = stack.top();
        stack.pop();
        Vertex *v = getVertex(u);

        if (!v->isMarked()) {
            v->mark();
            v->setSafe(true);
            ret++;
            for (Vertex *i : v->getAdjacency()) {
                if (!i->isMarked()) {
                    for (auto p : pontes) {
                        flag = (v->getId() == p.first && i->getId() == p.second) ||
                               (v->getId() == p.second && i->getId() == p.first);
                        if (flag)
                            break;
                    }
                }
            }
        }
    }
    return ret;
}

```

```

        if (!flag)
            stack.push(i->getId());
    }
}
}
}
return ret;
}

```

**AQUI NÓS VERIFICAMOS
QUAIS VÉRTICES SÃO
ALCANÇÁVEIS A PARTIR
DA ENTRADA DA
CAVERNA (VÉRTICE 0).**

FIM

Referências:

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

Repositório com os códigos fonte: <https://replit.com/@Ventinos/TG-EP03?v=1>