

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:

Pega vareta é um jogo onde uma coleção de varetas coloridas são despejadas em uma pilha emaranhada na mesa, os jogadores se revezam tentando pegar uma vareta de cada vez sem mover nenhum dos outros gravetos.

Como é muito difícil remover um graveto se tiver outro em cima, os jogadores devem tentar pegar gravetos em uma ordem em que eles não peguem um graveto que está embaixo de outro.

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 gravetos da pilha, os gravetos vão de 1 até N.
- M é o número de linhas de cada caso.

As linhas seguintes devem possuir dois gravetos, **a** e **b**, indicando que em algum momento o graveto **a** se encontra em cima do graveto **b**.

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



3	2
1	2
2	3
0	0

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 retirar todos os palitos, ele mostrará a ordem em que os palitos devem ser removidos para que as regras não sejam violadas, se ele não conseguir remover todos os palitos ele irá apenas imprimir a palavra “IMPOSSIBLE”.

Nossa Resolução:

- Graph.hpp : contém um array com os vértices do grafo.
 - De métodos, contém adicionar vértices e buscar vértices no vetor, o método de geração do grafo e leitura dos inputs, e funções para preencher as listas de adjacência dos vértices de acordo com o input fornecido.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, além do id que representa o número do vértice.
 - 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.
- As implementações desses métodos estão descritas nos arquivos .cpp .

```

1  #include <vector>
2
3  class Vertex
4  {
5  private:
6      int id;
7      std::vector<Vertex*> adjacency;
8      bool check;
9  public:
10     //Construtores:
11     Vertex();
12     Vertex(int id);
13     //Getters:
14     int getId();
15     bool isChecked();
16     std::vector<Vertex*> getAdjacency();
17     //Setters:
18     void setId(int id);
19     void checkVertex();
20     //Adiciona o vertice v na lista de adjacencia
21     void addToAdjacency(Vertex *v);
22     //Printa as informacoes de id e valor do vertice atual
23     void print();
24     //Printa a lista de adjacencia do vertice atual
25     void printAdjacency();
26     //funcoes auxiliares:
27     bool removeAdjacency(int id);
28 };
29

```

```

1  #include "Vertex.hpp"
2
3  class Graph
4  {
5  private:
6      std::vector<Vertex*> vertices;
7  public:
8      //Construtores:
9      Graph();
10     //Faz toda a leitura de entrada e cria o grafo:
11     Graph *readGraph(int m, int n);
12     //Adiciona novo vertice na lista de vertices:
13     void addVertex(Vertex* v);
14     //Dado um id retorna o vertice naquela posição:
15     Vertex* getVertex(int id);
16     //GetVertices:
17     std::vector<Vertex*> getVertices();
18     //Imprime todo o grafo:
19     void print();
20     //solucao:
21     void solve();
22 };

```

Nossa Resolução:

Nossa Resolução consiste em duas etapas:

- 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 que deverão ser conectados, caso não existam até esse momento.
 - Realiza as conexões entre os vértices.
 - Adiciona os vértices que não estão conectados a ninguém.

```

20 std::vector<Vertex *> Graph::getVertices() { return vertices; }
21
22 // Faz toda a leitura de entrada e cria o grafo:
23 v Graph *Graph::readGraph(int m, int n) {
24     int entry1, entry2;
25     Graph *g;
26     Vertex *v1, *v2;
27     g = new Graph();
28
29     //cria e conecta todos os vertices que deverão ser conectados
30 v for (int i = 0; i < m; i++) {
31         std::cin >> entry1;
32
33         if (entry1 == 0)
34             break;
35 v         if (g->getVertex(entry1) == NULL) {
36             v1 = new Vertex(entry1);
37             g->addVertex(v1);
38 v         } else {
39             v1 = g->getVertex(entry1);
40         }
41
42         std::cin >> entry2;
43 v         if (g->getVertex(entry2) == NULL) {
44             v2 = new Vertex(entry2);
45             g->addVertex(v2);
46 v         } else {
47             v2 = g->getVertex(entry2);
48         }
49         v2->addToAdjacency(v1);
50     }

```

```

52     //adiciona os vertices que não estão conectados a ninguém
53 v for (int i = 1; i < n; i++) {
54 v     if (g->getVertex(i) == NULL) {
55         v1 = new Vertex(i);
56         g->addVertex(v1);
57     }
58 }
59
60 //retorna o grafo com todos os vertices
61 return g;
62 }

```


Nossa Resolução:

Como fizemos para remover os palitos em ordem?

Fizemos a inserção de uma maneira para facilitar a remoção dos palitos em ordem.

As listas de adjacência são montadas junto com o preenchimento dos vértices no grafo, os únicos vértices que possuem elementos em suas listas de adjacência são os que possuem “palitos” acima deles, deixando os vértices do “topo” com uma lista de adjacência vazia. Guardamos na lista de adjacência de um vértice o palito imediatamente acima dele.


A função responsável pela conexão e inserção dos vértices é a:

```
Graph *readGraph(int m, int n);
```

Nossa Resolução:

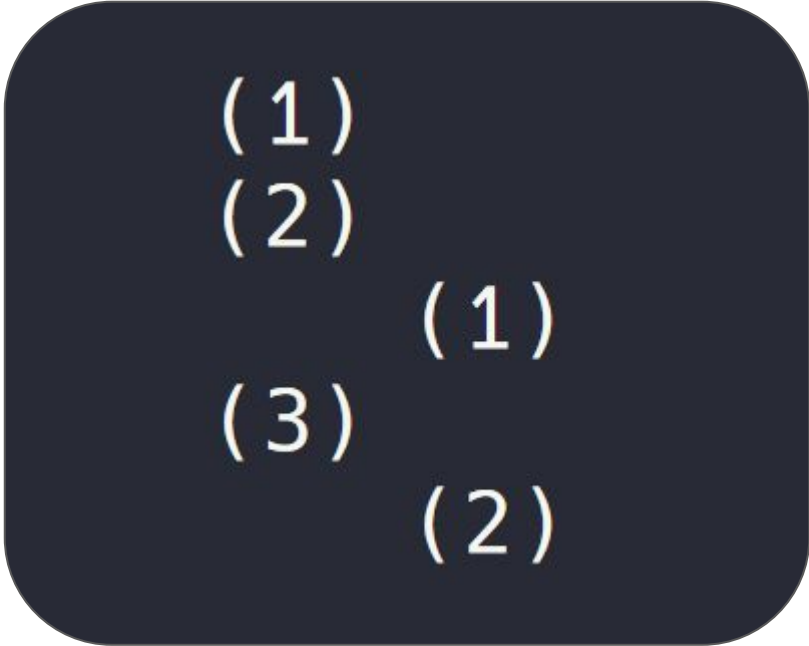
Vamos ver um exemplo:

Input:



3	2
1	2
2	3
0	0

Lista de adjacência:



(1)	
(2)	
	(1)
(3)	
	(2)

Nossa Resolução:

Continuação do exemplo.

Grafo:



Lista de adjacência:

```
(1)
(2)
(3) (1)
      (2)
```

Nossa Resolução:

Agora que temos as nossas listas de adjacência montadas, temos que buscar os vértices que possuem lista de adjacência vazia, já que esses são os “gravetos” que não possuem nenhum outro “graveto” acima deles. A partir disso podemos chegar em 3 casos:

- Todos os vértices tem lista de adjacência vazia, logo todos podem ser removidos sem dificuldade.
- Pelo menos um vértice possui lista de adjacência não vazia, o programa ignora esse tipo de vértice e remove os vértices com lista de adjacência vazia, até que todos os vértices estejam com a lista de adjacência vazia.
- Pelo menos dois vértices estão fortemente conectados, ou seja, o ‘graveto a’ está acima do ‘graveto b’ e o ‘graveto b’ está acima do ‘graveto a’. Esse é um caso impossível de resolver, logo o programa imprime a palavra **‘IMPOSSIBLE’**

A função responsável por isso é a `bool Graph::solve();`

```

71 void Graph::solve() {
72     std::queue<Vertex *> queue;
73     Vertex *v = NULL;
74     /*
75     Isso aqui procura por vertices v de lista de adjacencia vazia, se achar
76     ele remove esse v das listas de adjacencia de todos os vertices bota v numa
77     fila e remove v do grafo. Depois disso tira os elementos da fila.
78     */
79
80     // Percorre todos os vértices da lista de vértices
81     for (int j = 0; j < vertices.size(); j++) {
82         v = vertices.at(j);
83         // verifica se tem lista vazia:
84         if (v->getAdjacency().empty()) {
85             // removendo das listas de adj:
86             for (Vertex *w : vertices)
87                 w->removeAdjacency(v->getId());
88             // botando na fila:
89             queue.push(v);
90             // removendo do grafo:
91             vertices.erase(vertices.begin() + j);
92             j = -1;
93         }
94     }
95 }

```

```

96     //se pelo menos um dos vertices não pode ser removido,
97     //printa "IMPOSSIBLE" e limpa o grafo
98     if (vertices.size() > 0) {
99         std::cout << "IMPOSSIBLE"
100             << "\n";
101         vertices.clear();
102         return;
103     }
104     // desenfilera os valores:
105     while (queue.size() != 0) {
106         std::cout << queue.front()->getId() << std::endl;
107         queue.pop();
108     }
109 }

```

FIM

Referências:

Repositório com os códigos fonte: <https://replit.com/@Ventinos/TG-SO2EP03>

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