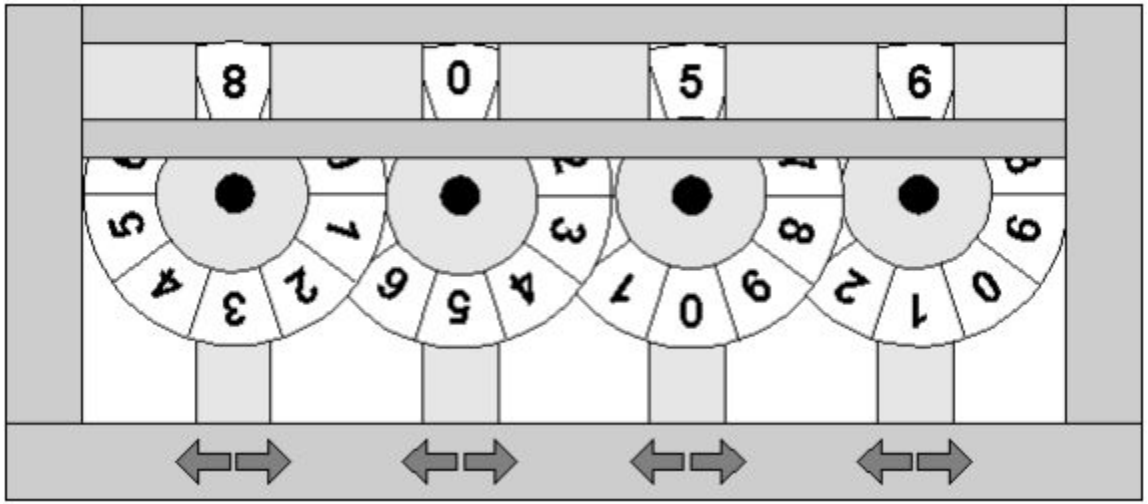
Resolução do problema EP01

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

O Problema:

Dado um jogo com 4 rodas numeradas de 0 a 9 em sentido horário na qual os botões abaixo da roda gira a roda em uma unidade para a direção selecionada. Devemos desenvolver um programa que dada um número de entrada, encontrar um número de saída sem passar por uma lista de números proibidos.



Entradas:

A entrada consiste em, nessa ordem:

- Quantidade **n** de casos de teste;
- A primeira linha de cada caso possui a sequência inicial nas rodas;
- A segunda linha possui a sequência final que devemos alcançar;
- A terceira linha possui um número **m** de sequências proibidas
- As próximas **m** linhas possuem uma sequencia proibida

Cada caso de teste é separado por uma linha vazia;

Saídas:

Cada linha representa um caso de entrada, na qual retorna o menor número de botões pressionados para se chegar na sequência final ou então retorna -1 caso não seja possível chegar no final.

Exemplo:

8 0 5 6 para 6 5 0 8

Sequências proibidas:

8 0 5 7

8 0 4 7

5 5 0 8

7 5 0 8

6 4 0 8

Menor caminho:

7 0 5 6

6 0 5 6

6 1 5 6

6 2 6 6

6 3 5 6

6 4 5 6

6 5 5 6

6 5 4 6

6 5 3 6

6 5 2 6

6 5 1 6

6 5 0 6

6 5 0 7

6 5 0 8

Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar a cada sequência como um vértice e cada vez que um botão é pressionado como uma aresta para outro vértice

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.
  - De métodos temos métodos para a geração do grafo, um método de solução e impressão do grafo.
- Vertex.hpp: contém, de atributos, uma lista de vértices adjacentes, a marcação do vértice e sua segurança.
  - De métodos temos a função print, que imprime o vértice dado e a função que adiciona um vértice na lista de adjacência.

```cpp
#include <vector>
#include <string>
#include <iostream>

class Vertex{
public:
  int id;
  bool mark;
  int dist;
  std::vector<Vertex*> adjacency;

  Vertex();
  Vertex(int id);

  //add do vertice na adjacencia
  std::vector<Vertex*> getAdjacency();
  //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();
};
```

```cpp
#include "Vertex.hpp"
#include <fstream>
#include <queue>
class Graph
{
public:
  std::vector<Vertex*> vertices;

//Construtores:
  Graph();
  Graph(int linha, int coluna);

//Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph();
  void insereAdj(int i);
//Imprime todo o grafo:
  void print();
  void bfs(int S);
  void resetMarks();
};
```

Método para resolução do problema:

Para resolver o problema, primeiro fizemos a leitura do grafo a partir da função readGraph(), e depois marcamos os vértices proibidos, e por fim aplicamos um BFS para encontrar o menor caminho possível para se chegar a sequência desejada.

Criação do Grafo:

```
10  // Faz toda a leitura de entrada e cria o grafo:
11  Graph *Graph::readGraph() {
12    Graph *g;
13    g = new Graph();
14
15    //cria 10k de vertices e bota no grafo:
16    for (int i = 0; i < 10000; i++) {
17      g->vertices.push_back(new Vertex(i));
18    }
19
20    //faz as insercoes devidas nas listas de adj de cada vertice:
21    for (int i = 0; i < 10000; i++)
22      g->insereAdj(i);
23
24    //retorna o grafo pronto:
25    return g;
26  }
```

Inserção das adjacências:

```
28  void Graph::insereAdj(int i) {
29    short int d[4], dUp[4], dDown[4], temp1, temp2;
30    int valor = i, mult = 1;
31    int numUp = 0, numDown = 0;
32
33    // separando digitos:
34    for (int j = 0; j < 4; j++) {
35      // separando os digitos
36      d[j] = valor % 10;
37      valor /= 10;
38    }
39
40    // copiando de d para dUp e dDown
41    for (int k = 0; k < 4; k++) {
42      dUp[k] = d[k];
43      dDown[k] = d[k];
44    }
45
46    for (int j = 0; j < 4; j++) {
47      // caso pra cima
48      temp1 = dUp[j];
49      dUp[j]++;
50      dUp[j] %= 10;
52      // caso pra baixo
53      temp2 = dDown[j];
54      dDown[j] += 9;
55      dDown[j] %= 10;
56
57      // concatenando os digitos de up e down:
58      for (int j = 0; j < 4; j++) {
59        numUp += dUp[j] * mult;
60        numDown += dDown[j] * mult;
61        mult *= 10;
62      }
63
64      // adicionando na lista de adj de i o numUp e numDown:
65      this->vertices.at(i)->addToAdjacency(this->vertices.at(numUp));
66      this->vertices.at(i)->addToAdjacency(this->vertices.at(numDown));
67
68      // resetando variaveis:
69      dUp[j] = temp1;
70      dDown[j] = temp2;
71      numUp = 0;
72      numDown = 0;
73      mult = 1;
74    }
75  }
```

BFS:

```
84  //bfs usado para medir os cliques:
85  void Graph::bfs(int S) {
86    int a, d;
87    std::queue<int> queue;
88    //bota na fila
89    queue.push(S);
90    //marca
91    Graph::vertices.at(S)->mark = true;
92    Graph::vertices.at(S)->dist = 0;
93    //enquanto a fila n estiver vazia:
94    while (!queue.empty()) {
95      //pega o primeiro elemento da fila:
96      a = queue.front();
97      queue.pop();
98      //pega o vertice desse id:
99      Vertex *v = Graph::vertices.at(a);
100     //pega a distancia ate ele:
101     d = v->dist;
102     //para cada vertice adjacente:
103     for (auto i : v->getAdjacency()) {
104       //se n estiver marcado:
105       if (!i->mark) {
106         //marca:
107         i->mark = true;
108         //faz a distancia ate ele d+1:
109         i->dist = d + 1;
110         //bota na fila:
111         queue.push(i->id);
112       }
113     }
114   }
115 }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Ventinos/S08EP01