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:

Sam encontrou um grande conjunto de mapas do antigo Mestre Aemon, que à primeira vista devem indicar, cada um, a localização de um baú cheio de obsidiana. No entanto, após uma análise mais cuidadosa, alguns mapas tinham erros óbvios, enquanto outros só poderiam ser verificados enviando uma equipe de exploradores.

Dado que há muitos mapas, os irmãos da Patrulha da Noite são poucos e o inverno se aproxima, nossa tarefa é escrever um programa para verificar se um mapa leva ou não a um baú com obsidiana.

O Problema:

Os mapas possuem as seguintes características:

- O ponto de partida é sempre no canto superior esquerdo.
- Os mapas são retangulares e cada ponto do mapa possui um destes símbolos:
  - Um espaço de terreno atravessável.
  - Uma seta, representando uma possível mudança de direção.
  - Um baú.

Dado que os lugares descritos por esses mapas são muito perigosos, é vital que o caminho descrito no mapa seja rigorosamente seguido.

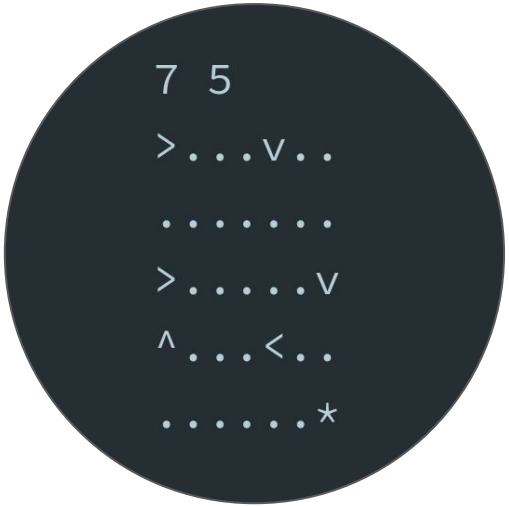
Entradas:

A primeira linha contém um número inteiro positivo **x < 100**, representando a largura do mapa.

A segunda linha contém um número inteiro positivo **y < 100**, representando a altura do mapa.

As próximas linhas contêm vários caracteres dentro das dimensões do mapa. Os caracteres válidos são:

- Uma seta para a direita: **>**
- Uma seta para a esquerda: **<**
- Uma seta apontando para baixo: **v**
- Uma seta apontando para cima: **^**
- Um espaço representando terreno atravessável: **.**
- Um baú: *****



Saídas:

A saída deve consistir em uma única linha contendo um único caractere **!** ou *.

! significa que o mapa é inválido.

* significa que o mapa é válido.

Nossa Resolução:

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, além do número de linhas e colunas.
  - De métodos, contém Getters e Setters básicos para cada atributo, o método de geração do grafo e leitura dos inputs, e funções para preencher as listas de adjacência dos vértices de acordo com seu valor.
- Vertex.hpp: contém, de atributos, um array de vértices adjacentes, além do valor que pode ser um dos símbolos apresentados anteriormente.
  - 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 .

```cpp
#include "Vertex.hpp"

class Graph
{
private:
  std::vector<Vertex*> vertices;
  int x;
  int y;
public:
//Construtores:
  Graph();
  Graph(int x, int y);
//Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(bool *start, bool *chest);
//Adiciona novo vertice na lista de vertices:
  void addVertex(Vertex* v);
//Dado um id retorna o vertice naquela posição:
  Vertex* getVertex(int id);
//GetVertices:
  std::vector<Vertex*> getVertices();
//Imprime todo o grafo:
  void print();
//Faz as conexoes no mapa:
//Conecta na proxima seta da direcao indicada, se houver:
  void connectVertices(Vertex* i, int x, int y, char value);
//Realiza a conexao da funcao
//connectVertices(...):
  bool connect(Vertex* i, int id, int j);
//Percorre o caminho descrito no mapa:
  bool isReachable();
};

```

```cpp
#include <vector>
class Vertex
{
private:
  int id;
  char value;
  std::vector<Vertex*> adjacency;
  bool check;
public:
  //Construtores:
  Vertex();
  Vertex(int id, char value);
  //Getters:
  int getId();
  char getValue();
  bool isChecked();
  std::vector<Vertex*> getAdjacency();
  //Setters:
  void setId(int id);
  void setValue(char value);
  void checkVertex();
  //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();
};

```

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(bool *start, bool *chest) da nossa classe de Grafo descrita em Graph.hpp e Graph.cpp.
- Avaliação se 3 requisitos estão sendo atendidos:
  - Se há um ponto de partida válido, isto é nosso vértice de id = 0 tem valor de > ou v.
  - Se há ao menos um baú no nosso mapa.
  - Se o caminho descrito tem fim em um baú.

```cpp
// Faz toda a leitura de entrada e cria o grafo:
Graph *Graph::readGraph(bool *start, bool *chest) {
  int x = 0, y = 0, size = 0;
  char entry, value;
  Graph *g;
  Vertex *v;
  *start = true;
  *chest = false;

  std::cin >> x;
  std::cin >> y;
  size = x * y;
  g = new Graph(x, y);

  for (int i = 0; i < size; i++) {
    std::cin >> entry;
    v = new Vertex(i, entry);
    g->addVertex(v);
  }
  // Fazendo as conexoes entre os vertices:
  for (Vertex *i : g->getVertices()) {
    value = i->getValue();
    if (value == '.' || value == '*') {
      if (value == '*') {
        *chest = true;
      }
    } else {
      if ((value == '^' || value == '.') && i->getId() == 0) {
        *start = false;
      }
      g->connectVertices(i, x, y, value);
    }
  }
  return g;
}
```

Nossa Resolução:

Como fizemos para percorrer o caminho descrito?

Operamos sobre algumas regras simples, a primeira se refere às listas de adjacência dos nossos vértices.

As listas de adjacência são montadas após o preenchimento dos vértices. Os únicos vértices que possuem elementos em suas listas de adjacência são as setas e elas guardam a primeira seta que aparece na direção que ela aponta e a deixa marcada, itens marcados quando revisitados param a busca, entende-se que estamos em um loop.

A função responsável pela conexão dos vértices é a

```
void connectVertices(Vertex* i, int x, int y, char value);
```

Nossa Resolução:

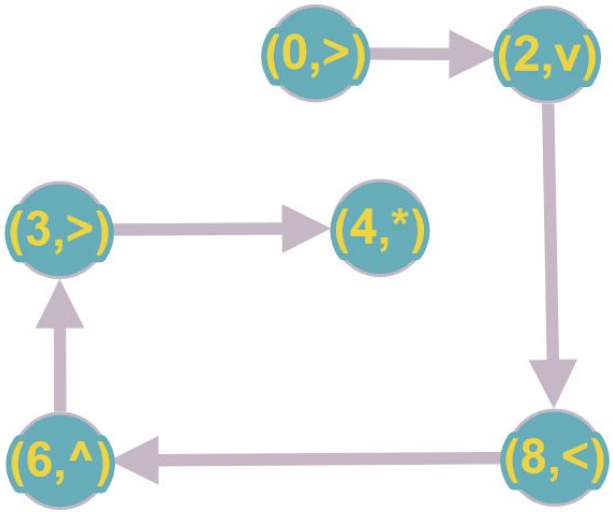
Vamos ver um exemplo:

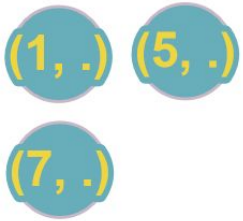
Input:

Listas de adjacência:

Nossa Resolução:

Visualizando o grafo do exemplo dado:





Nossa Resolução:

Como o código faz isso?

- Dependendo do valor do vértice andamos de uma maneira diferente pelo grafo fazendo as inserções nas listas de adjacência.

```
case '>':
  if (id % x != x - 1) {
    for (int j = 1; id + j % x != 0; j++) {
      if (!Graph::connect(i, id, j)) {
        break;
      }
    }
  }
  break;
```

```
case '<':
  if (id % x != 0) {
    for (int j = -1; (id + j) % x != x - 1; j--) {
      if (!Graph::connect(i, id, j)) {
        break;
      }
    }
  }
  break;
```

Nossa Resolução:

Justificando os condicionais:

- O nosso problema fica bem fácil de enxergar se dispormos os valores numa matriz da forma (sem perder a generalidade):

0	1	2
3	4	5
6	7	8

Tendo em vista que nossos valores serão dispostos dessa forma, cumprindo com os dois primeiros valores lidos **x**(quantidade de colunas) e **y**(quantidade de linhas), podemos utilizar de algumas expressões matemáticas para acessar todas as posições em uma determinada direção da matriz independente do ponto de partida.

>	.	v
>	*	.
^	.	<

Nossa Resolução:

Como o código faz isso?

- Dependendo do valor do vértice andamos de uma maneira diferente pelo grafo fazendo as inserções nas listas de adjacência.

```
case 'v':
  if (!(id >= x * (y - 1) && id <= (x * y) - 1)) {
    for (int j = x; id+j <= (x*y)-1; j += x) {
      if (!Graph::connect(i, id, j)) {
        break;
      }
    }
  }
  break;
```

```
case '^':
  if (!(id >= 0 && id <= x - 1)) {
    for (int j = -x; id + j >= 0; j -= x) {
      if (!Graph::connect(i, id, j)) {
        break;
      }
    }
  }
  break;
```

Nossa Resolução:

Definimos diferentes condições de parada pois as nossas inserções nas listas de adjacência foram feitas usando uma regra geral aplicada pela função:

bool connect(Vertex* i, int id, int j)

Vamos ver sua implementação:

```
void Vertex::addToAdjacency(Vertex *v) {
  adjacency.push_back(v);
}
```

```
// auxilia a funcao connectVertices(...):
bool Graph::connect(Vertex *i, int id, int j) {
  Vertex *v = getVertex(id + j);
  char vertexValue = v->getValue();
  //se tirarmos a funcao de adicionar desse if
guardamos os pontos nas listas:
  if (vertexValue != '.') {
    i->addToAdjacency(v);
    return false;
  }
  return true;
}
```

Nossa Resolução:

Agora que temos as nossas listas de adjacência montadas, temos que basicamente partir do primeiro elemento, e ir seguindo de lista em lista das setas até chegarmos em 3 conclusões possíveis:

- Ou chegamos em um baú então temos uma solução.
- Ou chegamos num caminho sem fim, caracterizado por uma seta de lista vazia, indicando que não temos solução.
- Ou voltamos em uma seta que já foi marcada, o que nos permite inferir que chegamos num loop, indicando que não temos solução para nosso mapa.

A função responsável por isso é a isReachable();

```
bool Graph::isReachable() {
  std::vector<Vertex *> adj = getVertex(0)->getAdjacency();
  char value;
  Vertex *w;
  int size = adj.size();
  getVertex(0)->checkVertex();
  int j = 0;

  for (int i = 0; i < x * y; i++) {

    if (adj.size() > j)
      w = adj.at(j);
    else
      return false;
    value = w->getValue();
    if (getVertex(w->getId())->isChecked()) {
      return false;
    }
    j++;

    if (value != '.' && value != '*') {
      getVertex(w->getId())->checkVertex();
      adj = getVertex(w->getId())->getAdjacency();
      j = 0;
    } else if (value == '*') {
      return true;
    }
  }
  return false;
}
```

FIM

Referências:

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

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