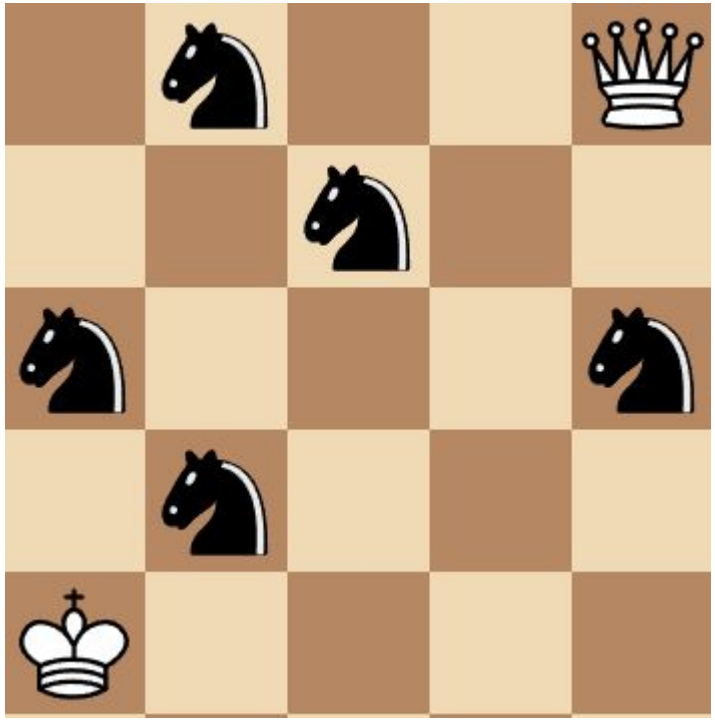
Resolução do problema EP02

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

Problema:

O Rei Peter vive no reino A e deseja visitar sua filha que vive no reino B no qual são separados por uma floresta cheia de assassinos em cavalos. A floresta é dividida como uma tabuleiro de xadrez de **M** linhas e **N** colunas e áreas onde os assassinos estão e podem se locomover para são áreas não seguras para andar. Por isso devemos verificar se a rota é segura para Peter ver sua filha.

Exemplo:



Entradas:

A entrada consiste em diversas verificações seguindo o modelo:

- Número de verificações **T**;
- Dois inteiros positivos **N** e **M**, onde **N** é o número de junções e **M** o número de ruas
- Próximas **M** linhas são compostas por **N** na qual são:
  - . - Posição vazia;
  - Z - Cavalo ocupa a posição;
  - A - Reino A;
  - B - Reino B.

Saídas:

A saída deve ser composta por **T** linhas na qual podem ser:

- Minimal possible length of a trip is L
- King Peter, you can’t go now!

Onde L é a distância mínima possível entre A e B;

Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar cada posição do tabuleiro como vértices de um grafo, sendo assim possível encontrar menores caminhos utilizando um BFS.

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 um método para a geração do grafo, um método para verificar segurança dos vértices, um método de solução e impressão da solução.
- 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 "Vertex.hpp"
#include <fstream>
#include <queue>
class Graph
{
public:
  std::vector<Vertex*> vertices;
  int linha;
  int coluna;
  int reinoA;
  int reinoB;

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

//Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph();

//Imprime todo o grafo:
  void print();

//Imprime a resposta
  void printResul();

//torna perigoso regiões com cavalos e potencialmente
//posições que cavalos podem acessar com 1 movimento
  void markUnsafePlaces();

//bfs para caminhos mínimos
  void bfs(int e);
};
```

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

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

  Vertex();
  Vertex(int id, char value);

  //UM setter
  void setUnsafe();
  void setMark();
  //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();
  //funcoes auxiliares:
  bool removeAdjacency(int id);
};
```

Método para resolução do problema

Para resolver o problema, primeiro fizemos a leitura do grafo a partir da função readGraph(), e depois verificamos quais vértices não eram seguros para atravessar, e por fim aplicamos um BFS para encontrar o menor caminho possível para o reino B.

Leitura de Dados:

```
// Faz toda a leitura de entrada e cria o grafo:
Graph * Graph::readGraph( ) {
  int n = 0, m = 0, counter = 0;
  std::string s;
  char c;
  int v1 = 0, v2 = 0, size = 0;
  Vertex *v;
  Graph *g;
  std::cin >> m;
  std::cin >> n;
  g = new Graph(m,n);
  for(int i = 0; i < m; i++){
    std::cin >> s;
    for(int j = 0; j < n; j++){
      c = s.at(j);
      v = new Vertex(counter, c);
      if(c == 'A') g->reinoA = counter;
      if(c == 'B') g->reinoB = counter;
      counter++;
      g->vertices.push_back(v);
    }
  }
```

Conexões entre vértices:

```
  for(int i = 0; i < m*n; i++){
    //insert linha do elemento
    if((i+1)%n!=0) g->vertices.at(i)->addToAdjacency(g->vertices.at(i+1));
    if(i>0 && (i-1)/n == i/n) g->vertices.at(i)->addToAdjacency(g->vertices.at(i-1));
    //insert linha acima do elemento
    if(i-n>=0)
    {
      g->vertices.at(i)->addToAdjacency(g->vertices.at(i-n));
      if((i+1-n)%n!=0) g->vertices.at(i)->addToAdjacency(g->vertices.at(i+1-n));
    if((i-n)%n!=0 && ((i-1-n)/n == (i-n)/n)) g->vertices.at(i)->addToAdjacency(g-
>vertices.at(i-1-n));
    }
    //insert linha abaixo do elemento
    if(i+n<n*m)
    {
      g->vertices.at(i)->addToAdjacency(g->vertices.at(i+n));
      if((i+1+n)%n!=0) g->vertices.at(i)->addToAdjacency(g->vertices.at(i+1+n));
      if((i+n)%n!=0 && ((i-1+n)/n == (i+n)/n)) g->vertices.at(i)->addToAdjacency(g-
>vertices.at(i-1+n));
    }
  }
  return g;
}
```

Verificação de vértices perigosos:

```
for (auto j : Graph::vertices) {
  if(j->value == 'Z'){
    j->setUnsafe();
    int i = j->id;
    if((i+2) < (col*lin) && ((i+2)/col == i/col)) {//direita, cima e baixo
      if(i+2+col<col*lin && ((i+2+col)/col== i/col + 1)){
        Graph::vertices.at(i+2+col)->setUnsafe();
      }
      if(i+2-col>=0 && i+2-col<col*lin && ((i+2-col)/col == i/col - 1)){
        Graph::vertices.at(i+2-col)->setUnsafe();
      }
    }
    if((i-2) >= 0 && (i-2) < (col*lin) && ((i-2)/col == i/col)){//esquerda, cima e baixo
      if(i-2+col>=0 && i-2+col<col*lin && ((i-2+col)/col == i/col) + 1){
        Graph::vertices.at(i-2+col)->setUnsafe();
      }
      if(i-2-col>=0 && i-2-col<col*lin && ((i-2-col)/col== i/col - 1 )){
        Graph::vertices.at(i-2-col)->setUnsafe();
      }
    }
```

Verificação de vértices perigosos:

```
}
if((i+2*col) < (col*lin)){//abaixo, direita e esquerda
  if((i+1+2*col)<col*lin && (i+1+2*col)/col == (i+2*col)/col){
    Graph::vertices.at(i+1+2*col)->setUnsafe();
  }
  if(i-1+2*col<col*lin && (i-1+2*col)/col == (i+2*col)/col){
    Graph::vertices.at(i-1+2*col)->setUnsafe();
  }
}
if((i-2*col)>=0 && (i-2*col) < (col*lin)){//cima, direita e esquerda
  if((i+1-2*col)>=0 && (i+1-2*col)<col*lin && (i+1-2*col)/col == (i-2*col)/col){
    Graph::vertices.at(i+1-2*col)->setUnsafe();
  }
  if(i-1-2*col>=0 && i-1-2*col<col*lin && (i-1-2*col)/col == (i-2*col)/col){
    Graph::vertices.at(i-1-2*col)->setUnsafe();
  }
```

BFS:

```
112  void Graph::bfs(int e)
113  {
114    int a,d;
115    bool flag = true;
116    std::queue<int> queue;
117    queue.push(e);
118    Graph::vertices.at(e)->setMark();
119    while(!queue.empty())
120      {
121        a = queue.front();
122        queue.pop();
123        Vertex* v = Graph::vertices.at(a);
124        d = v->dist;
125        bool vmark = v->mark;
126          for(auto i : v->getAdjacency())
127            {
128              if(!i->mark && i->safe)//só analisa/insere elementos que são seguros e ainda não
     visitados
129              {
130                i->setMark();
131                i->dist = d+1;
132                queue.push(i->id);
133              }
134            }
135      }
136  }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@MariaLuisaGabri/S07EP02

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