

Examen Grafos 01/09/22

Se dispone de un laberinto de  $N \times N \times N$  casillas del que se conocen las casillas de entrada y salida del mismo.

Si te encuentras en una casilla sólo puedes moverte (en el mejor caso) en las siguientes seis direcciones (arriba, abajo, derecha, izquierda, adentro, afuera).

Por otra parte, entre algunas de las casillas hay una piedra que impide moverse hacia ella. Implementa un subprograma que dados:

- $N$  (dimensión del laberinto),
- la lista de casillas que poseen una piedra,
- la casilla de entrada, y
- la casilla de salida,

calcule el coste del camino más corto para ir de la entrada a la salida y su longitud.

\*Nota: definir tipos de datos, prototipos de las operaciones de los TADs y de los algoritmos de grafos.

## Solución:

```

1  #include <iostream>
2  #include "alg_grafo_E-S.h"
3  #include "alg_grafoPMC.h"
4  #include "grafoPMC.h"
5  #include "matriz.h"
6  #include <vector>
7
8  using namespace std;
9
10 //-----|| Aquí comienza el examen ||-----
11
12 typedef typename GrafoP<int>::tCoste tCoste;
13 typedef typename GrafoP<int>::vertice vertice;
14
15 struct casilla
16 {
17     int x,y,z;    //fila será x ,columna será y, profundidad será z
18 };
19
20
21 /*
22  CasillaToNodo -> Recibe una casilla y el tamaño del laberinto, y devuelve el nodo correspondiente a la casilla
23 */
24 vertice CasillaToNodo(casilla c,size_t N)
25 {
26     return c.x + c.y*N + c.z*(N*N);
27 }
28
29 /*
30  NodoToCasilla -> Recibe una nodo y el tamaño del laberinto, y devuelve la casilla correspondiente a la casilla
31 */
32 casilla NodoToCasilla(vertice v,size_t N)
33 {
34     casilla caux;
35     caux.x = v % N; // ejemplos: casilla 3 (x = 0), casilla 10 (x = 1)...
36     caux.y = (v / N) % N; // ejemplos: casilla 5 (y = 1), casilla 25(x = 2)...
37     caux.z = v / (N*N);
38     return caux;
39 }
40
41 /*
42  abyacentes -> Recibe dos casillas, y devuelve true si las dos casilla son adyacentes (se puede realizar un movimiento entre ellas) o false si no
43 */
44 bool abyacentes (casilla c1, casilla c2)
45 {
46     if(c1.z == c2.z) // si estan en el mismo tablero:
47     {
48         return (abs(c1.x- c2.x) + abs(c1.y - c2.y) == 1);
49     }
50     else
51     {
52         return (c1.x == c2.x && c1.y == c2.y && abs(c1.z - c2.z) == 1);
53     }
54 }
55

```

```

56  /*
57  laberinto -> Recibe el tamaño del laberinto, un vector con las casillas que poseen piedras, la casilla de entrada y la casilla de salida,
58  y devuelve el coste mínimo de ir de la casilla entrada hasta la casilla salida
59  */
60  tCoste laberinto(size_t& N,vector<casilla>& piedras, casilla& entrada, casilla& salida)
61  {
62      GrafoP<tCoste> G(N*N*N); //creamos un grafo ponderado representado por una matriz de costes de N*N*N (numero de vertices) *Nota: por defecto inicializada a infinito
63
64
65      //Habilitamos los caminos entre las casillas que sean adyacentes
66
67      for(vertice i = 0; i < G.numVert(); i++) // podríamos usar int o unsigned tambien, pero uso vertice para hacer referencia a que me muevo de un nodo a otro :)
68      {
69          for(vertice j = 0; j < G.numVert(); j++)
70          {
71              if(abyacentes(NodoToCasilla(i,N),NodoToCasilla(j,N))) // Si las casillas representadas por los vertices son adyacentes:
72              {
73                  G[i][j] = 1; // Existe un camino de 1 unidad de longitud entre ellas (direccion i -> j)
74                  G[j][i] = 1; // Existe un camino de 1 unidad de longitud entre ellas (direccion j -> i)
75              }
76          }
77      }
78
79      //Bloqueamos aquellos caminos en los que la casilla sea una piedra
80      for(vertice i = 0; i < G.numVert(); i++) // podríamos usar int o unsigned tambien, pero uso vertice para hacer referencia a que me muevo de un nodo a otro :)
81      {
82          for(size_t j = 0; j < piedras.size(); j++) //recorremos el vector donde tenemos las casillas que contienen una piedra
83          {
84              G[i][CasillaToNodo(piedras[j],N)] = GrafoP<tCoste>::INFINITO; //NO podemos ir desde cualquier casilla hacia esa casilla
85              G[CasillaToNodo(piedras[j],N)][i] = GrafoP<tCoste>::INFINITO; //no necesario ya que comunicamos la casilla que tiene la piedra, udado para tener un Grafo mas realista
86          }
87      }
88
89      //Por ultimo usamos Djistra para calcular el recorrido mas corto entre la entrada y salida
90      vector<vertice> P(G.numVert());
91      vector<tCoste> D = Dijkstra(G,CasillaToNodo(entrada,N),P); //En D tenemos los costes de ir desde entrada a cualquier nodo
92
93      return D[CasillaToNodo(salida,N)]; //devolvemos el coste de ir desde entrada hacia casilla
94  }
95
96  /*
97  DEFINICIONES:
98
99  - GrafoP<tCoste> G(size_t n) -> Define un grafo Ponderado representado por una matriz de costes de n*n
100  - G.numVert() -> Devuelve el numero de aristas que posee el grafo
101  - Dijkstra(GrafoP<tCoste> G, vertice o, vector<vertice> P) -> Devuelve un vector de tCoste, con los costes de ir desde el vertice o a cualquier vertice
102  */
103
104  //-----|| Aquí termina el examen ||-----
105
106
107  //Main con el objetivo de probar el programa y comprobar su funcionamiento
108  int main()
109  {
110      size_t N = 3; //Tamaño del laberinto 3
111
112      // DEFINICION DE LA PIEDRAS
113      vector<casilla> piedras(3);
114      piedras[0].x = 0; piedras[0].y = 0; piedras[0].z = 0; // casilla 0
115      piedras[1].x = 2; piedras[1].y = 0; piedras[1].z = 1; // casilla 11
116      piedras[2].x = 2; piedras[2].y = 1; piedras[2].z = 1; // casilla 14
117      // FIN DEFINICION DE PIEDRAS
118
119      casilla entrada,salida;
120      entrada.x = 2; entrada.y = 0; entrada.z = 0;
121      salida.x = 2; salida.y = 0; salida.z = 2;
122
123
124      tCoste resultado = laberinto(N,piedras,entrada,salida);
125
126      cout << "El coste de ir desde la entrada a la salida es de: " << resultado << endl;
127
128      return 0;
129  }

```