Examen Grafos 01/09/22

Se dispone de un laberinto de NxNxN casillas del que se conocen las casillas de entrada y salida del mismo.
Si te encuentras en una casilla sólo puedes moverte en las siguientes cuatro 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:

```cpp
#include <iostream>
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include "grafoPMC.h"
#include "matriz.h"
#include <vector>

using namespace std;

//-------------------------|| Aqui comienza el examen ||-------------------------

typedef typename GrafoP<int>::tCoste tCoste;
typedef typename GrafoP<int>::vertice vertice;

struct casilla
{
    int x,y,z;    //fila será x ,columna será y, profundidad será z
};


/*
    CasillaToNodo -> Recibe una casilla y el tamaño del laberinto, y devuelve el nodo correspondiente a la casilla
*/
vertice CasillaToNodo(casilla c,size_t N)
{
    return c.x + c.y*N + c.z*(N*N);
}

/*
    NodoToCasilla -> Recibe una nodo y el tamaño del laberinto, y devuelve la casilla correspondiente a la casilla
*/
casilla NodoToCasilla(vertice v,size_t N)
{
    casilla caux;
    caux.x = v % N; // ejemplos: casilla 3 (x = 0), casilla 10 (x = 1)...
    caux.y = (v / N) % N; // ejemplos: casilla 5 (y = 1), casilla 25(x = 2)...
    caux.z = v / (N*N);
    return caux;
}

/*
    abyacentes -> Recibe dos casillas, y devuelve true si las dos casilla son adyacentes (se puede realizar un movimiento entre ellas) o false si no
*/
bool abyacentes (casilla c1, casilla c2)
{
    if(c1.z == c2.z) // si estan en el mismo tablero:
    {
        return (abs(c1.x- c2.x) == 1 && abs(c1.y - c2.y) == 1);
    }
    else
    {
        return (c1.x == c2.x && c1.y == c2.y && abs(c1.z - c2.z) == 1);
    }
}

```

```cpp
/*
    laberinto -> Recibe el tamaño del laberinto, un vector con las casillas que poseen piedras, la casilla de entrada y la casilla de salida,
    y devuelve el coste minimo de ir de la casilla entrada hasta la casilla salida
*/
tCoste laberinto(size_t& N,vector<casilla>& piedras, casilla& entrada, casilla& salida)
{
    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


    //Habilitamos los caminos entre las casillas que sean adyacentes

    for(vertice i = 0; i < G.numVert() ; i++) // podriamos usar int o unsigned tambien, pero uso vertice para hacer referencia a que me muevo de un nodo a otro :)
    {
        for(vertice j = 0; j < G.numVert() ; j++)
        {
            if(abyacentes(NodoToCasilla(i,N),NodoToCasilla(j,N))) // Si las casillas representadass por los vertices son adyacentes:
            {
                G[i][j] = 1; // Existe un camino de 1 unidad de longitud entre ellas (direccion i -> j)
                G[j][i] = 1; // Existe un camino de 1 unidad de longitud entre ellas (direccion j -> i)
            }
        }
    }

    //Bloqueamos aquellos caminos en los que la casilla sea una piedra
    for(vertice i = 0; i < G.numVert() ; i++) // podriamos usar int o unsigned tambien, pero uso vertice para hacer referencia a que me muevo de un nodo a otro :)
    {
        for(size_t j = 0; j < piedras.size() ; j++)  //recorremos el vector donde tenemos las casillas que contienen una piedra
        {
            G[i][CasillaToNodo(piedras[j],N)] = GrafoP<tCoste>::INFINITO; //NO podemos ir desde cualquier casilla hacia esa casilla
            G[CasillaToNodo(piedras[j],N)][i] = GrafoP<tCoste>::INFINITO; //no necesario ya que incomunicamos la casilla que tiene la piedra, udado para tener un Grafo mas realista
        }
    }

    //Por ultimo usamos Djistra para calcular el recorrido mas corto entre la entrada y salida
    vector<vertice> P(G.numVert());
    vector<tCoste> D = Dijkstra(G,CasillaToNodo(entrada,N),P); //En D tenemos los costes de ir desde entrada a cualquier nodo

    return D[CasillaToNodo(salida,N)];  //devolvemos el coste de ir desde entrada hacia casilla
}

/*
    DEFINICIONES:

    - GrafoP<tCoste> G(size_t n) -> Define un grafo Ponderado representado por una matriz de costes de n*n
    - G.numVert() -> Devuelve el numero de aristas que posee el grafo
    - 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
*/

//------------------------|| Aqui termina el examen ||------------------------


//Main con el objetivo de probar el programa y comprobar su funcionamiento
int main()
{
    size_t N = 3; //Tamaño del laberinto 3

    // DEFINICION DE LA PIEDRAS
    vector<casilla> piedras(3);
    piedras[0].x = 0; piedras[0].y= 0; piedras[0].z= 0; // casilla 0
    piedras[1].x = 2; piedras[1].y= 0; piedras[1].z= 1; // casilla 11
    piedras[2].x = 2; piedras[2].y= 1; piedras[2].z= 1; // casilla 14
    // FIN DEFINICION DE PIEDRAS

    casilla entrada,salida;
    entrada.x = 2; entrada.y = 0; entrada.z = 0;
    salida.x = 2; salida.y = 0; salida.z = 2;


    tCoste resultado = laberinto(N,piedras,entrada,salida);

    cout << "El coste de ir desde la entrada a la salida es de: " << resultado << endl;

    return 0;
}
```