Orbit

¿DÍA DE CLASES infinitas?

masca y fluye

A continuación se propone la resolución de los problemas de prácticas de grafos + Ejercicios de exámenes. No tiene por qué ser la solución perfecta, puedes tomarlo como una posible guía/idea para la resolución.

Atentamente: CamavirgaGII

Ejercicios Práctica 7

Ejercicio 1:

1. Tu agencia de viajes "OTRAVEZUNGRAFO S.A." se enfrenta a un curioso cliente. Es un personaje sorprendente, no le importa el dinero y quiere hacer el viaje más caro posible entre las ciudades que ofertas. Su objetivo es gastarse la mayor cantidad de dinero posible (ojalá todos los clientes fueran así), no le importa el origen ni el destino del viaje.

Sabiendo que es imposible pasar dos veces por la misma ciudad, ya que casualmente el grafo de tu agencia de viajes resultó ser acíclico, devolver el coste, origen y destino de tan curioso viaje. Se parte de la matriz de costes directos entre las ciudades del grafo.

```
#include"grafoPMC.h"
#include"alg_grafo_E-S.h"
#include"alg_grafoPMC.h"
#include<iostream>
using namespace std;


struct costeorigendestino
{
   double coste;
   GrafoP<unsigned int>::vertice origen,destino;
};



costeorigendestino ejercicio1(const GrafoP<unsigned int>&g);


int main()
{

   costeorigendestino c;

   GrafoP<unsigned int> g("grafo.txt");

   c=ejercicio1(g);

   cout<<c.coste<<endl;
   cout<<c.origen<<endl;
   cout<<c.destino<<endl;
```

Orbit ¿DÍA DE CLASES infinitas?

```
    return 0;
}



costeorigendestino ejercicio1(const GrafoP<unsigned int>&g)
{

    matriz<unsigned int> A;
    matriz<GrafoP<unsigned int>::vertice> P;
    costeorigendestino co;


    A=FloydMaximizado(g,P);


    //procedemos a buscar el coste mayor a la vez que origen y destino

    unsigned int cmayor=0;
    GrafoP<unsigned int>::vertice ori,dest;

    for(size_t i=0;i<A.dimension();i++)
    {
        for(size_t j=0;j<A.dimension();j++)
        {
            if(A[i][j]> cmayor && A[i][j]!=GrafoP<unsigned
int>::INFINITO)
            {
                cmayor=A[i][j];
                ori=i;
                dest=j;
            }
        }
    }


    co.origen=ori;
    co.destino=dest;
    co.coste=cmayor;
```

masca y fluye

```
   return co;
}
```

Floyd Maximizado:

```
template <typename tCoste>
matriz<tCoste> FloydMaximizado(const GrafoP<tCoste>& G,
                    matriz<typename GrafoP<tCoste>::vertice>& P)
// Calcula los caminos de coste mínimo entre cada
// par de vértices del grafo G. Devuelve una matriz
// de costes mínimos A de tamaño n x n, con n = G.numVert()
// y una matriz de vértices P de tamaño n x n, tal que
// P[i][j] es el vértice por el que pasa el camino de coste
// mínimo de i a j, si este vértice es i el camino es directo.
{
  typedef typename GrafoP<tCoste>::vertice vertice;
  const size_t n = G.numVert();
  matriz<tCoste> A(n);    // matriz de costes mínimos

  // Iniciar A y P con caminos directos entre cada par de vértices.
  P = matriz<vertice>(n);
  for (vertice i = 0; i < n; i++) {
     A[i] = G[i];                    // copia costes del grafo
     A[i][i] = 0;                    // diagonal a 0
     P[i] = vector<vertice>(n, i);   // caminos directos
  }
  // Calcular costes mínimos y caminos correspondientes
  // entre cualquier par de vértices i, j
  for (vertice k = 0; k < n; k++)
     for (vertice i = 0; i < n; i++)
        for (vertice j = 0; j < n; j++) {
           tCoste ikj = suma(A[i][k], A[k][j]);
           if (ikj > A[i][j] && ikj!=GrafoP<tCoste>::INFINITO && (ikj>
A[i][j] || A[i][j] == GrafoP<tCoste>::INFINITO) ) {
              A[i][j] = ikj;
              P[i][j] = k;
           }
        }
  return A;
```

```
}
```

Ejercicio 2:

2. Se dispone de un laberinto de NxN 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). Por otra parte, entre algunas de las

---

casillas hay una pared que impide moverse entre las dos casillas que separa dicha pared (en caso contrario no sería un verdadero laberinto).

Implementa un subprograma que dados

- N (dimensión del laberinto),
- la lista de paredes del laberinto,
- la casilla de entrada, y
- la casilla de salida,

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

Voy a definir una pared como una estructura con dos casillas
Una casilla será una estructura con sus coordenadas i,j

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;


struct Casilla
{
   int i,j; //no lo pongo como size_t porque la funcion abs obviamente
no lo permite
};

struct Pared
{
```

Orbit ¿DÍA DE CLASES infinitas?

```
    Casilla c1,c2;
};

//funciones necesarias
GrafoP<unsigned int>::vertice Casilla_A_Nodo(Casilla&c,size_t n)
{
    return (c.i*n + c.j);

}


Casilla Nodo_A_Casilla(GrafoP<unsigned int>::vertice v,size_t n)
{
    Casilla c;
    c.i=v / n;
    c.j= v % n;

    return c;
}


bool CasillasAdyacentes(Casilla &c1, Casilla& c2)
{
    //dos casillas son adyacentes cuando  difieren en una fila/columna

    int resf=c1.i - c2.i;
    int resc=c1.j - c2.j;


    if((resf==0 && (resc==1 || resc==-1)) || ((resc==0) && (resf==1 ||
resf==-1)))
        return true;

    return false;
}



GrafoP<unsigned int>::tCamino laberinto2D(size_t
N,vector<Pared>&vP,GrafoP<unsigned int>::vertice entrada,
GrafoP<unsigned int>::vertice salida);
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S
masca y fluye

```cpp
// Función auxiliar para imprimir el camino como coordenadas
void imprimirCamino(const GrafoP<unsigned int>::tCamino& cam, size_t N)
{
    cout << "Camino encontrado:" << endl;
    for (auto it = cam.primera(); it != cam.fin(); it =
cam.siguiente(it)) {
        Casilla c = Nodo_A_Casilla(cam.elemento(it), N);
        cout << "(" << c.i << "," << c.j << ") ";
    }
    cout << endl;
}

int main() {
    size_t N = 3;
    vector<Pared> paredes = {
        {{0, 1}, {0, 2}}, // Bloquea la casilla (0,1) con (0,2)
        {{1, 1}, {2, 1}}  // Bloquea la casilla (1,1) con (2,1)
    };

    Casilla entrada_c = {0, 0};
    Casilla salida_c = {2, 2};
    auto entrada = Casilla_A_Nodo(entrada_c, N);
    auto salida = Casilla_A_Nodo(salida_c, N);

    auto camino = laberinto2D(N, paredes, entrada, salida);
    imprimirCamino(camino, N);

    return 0;
}



GrafoP<unsigned int>::tCamino laberinto2D(size_t
N,vector<Pared>&vP,GrafoP<unsigned int>::vertice entrada,
GrafoP<unsigned int>::vertice salida)
{
    GrafoP<unsigned int>::tCamino cam;
    GrafoP<unsigned int> g(N*N); //creamos el grafo
    //vectores necesarios para Dijkstra
    vector<unsigned int> A;
    vector<GrafoP<unsigned int>::vertice> P;
```

```
   //procedo a rellenar el grafo primero sin tener en cuenta las
paredes
   for(size_t i=0;i<g.numVert();i++)
   {
       for(size_t j=0;j<g.numVert();j++)
       {
           if(i!=j) //evito ponerme un 1 para no crear ciclos
           {
               Casilla c1=Nodo_A_Casilla(i,N);
               Casilla c2=Nodo_A_Casilla(j,N);

               if(CasillasAdyacentes(c1,c2))
               {
                   g[i][j]=1;
                   g[j][i]=1;
                   cout<<"Ha entrado"<<endl;


               }

               //en el else que no hace falta ponerlo, se pondria a
infinito
           }

           else
               g[i][j]=0;
       }
   }

   cout<<"Comprobación de que el grafo es correcto: "<<endl;
   cout<<g<<endl;
   //procedemos a poner las paredes

   for(size_t i=0;i<vP.size();i++)
   {
       Casilla c1=vP[i].c1;
       Casilla c2=vP[i].c2;

       GrafoP<unsigned int>::vertice v1=Casilla_A_Nodo(c1,N);
       GrafoP<unsigned int>::vertice v2=Casilla_A_Nodo(c2,N);

       g[v1][v2]=GrafoP<unsigned int>::INFINITO;
       g[v2][v1]=GrafoP<unsigned int>::INFINITO;
```

Orbit ¿DÍA DE CLASES infinitas?

```
    }

    cout<<"Comprobación de que el grafo es correcto con paredes:
"<<endl;
    cout<<g<<endl;

    //procedemos a aplicar Dijkstra
    A=Dijkstra(g,entrada,P);

    cam=camino<unsigned int>(entrada,salida,P);


    return cam;
}
```

masca y fluye

Ejercicio 3:

3. Eres el orgulloso dueño de una empresa de distribución. Tu misión radica en distribuir todo tu stock entre las diferentes ciudades en las que tu empresa dispone de almacén.

Tienes un grafo representado mediante la matriz de costes, en el que aparece el coste (por unidad de producto) de transportar los productos entre las diferentes ciudades del grafo.

Pero además resulta que los Ayuntamientos de las diferentes ciudades en las que tienes almacén están muy interesados en que almacenes tus productos en ellas, por lo que están dispuestos a subvencionarte con un porcentaje de los gastos mínimos de transporte hasta la ciudad. Para facilitar el problema, consideraremos despreciables los costes de volver el camión a su base (centro de producción).

He aquí tu problema. Dispones de

- el centro de producción, nodo origen en el que tienes tu producto (no tiene almacén),
- una cantidad de unidades de producto (*cantidad*),
- la matriz de costes del grafo de distribución con N ciudades,
- la capacidad de almacenamiento de cada una de ellas,
- el porcentaje de subvención (sobre los gastos mínimos) que te ofrece cada Ayuntamiento.

Las diferentes ciudades (almacenes) pueden tener distinta capacidad, y además la capacidad total puede ser superior a la *cantidad* disponible de producto, por lo que debes decidir cuántas unidades de producto almacenas en cada una de las ciudades.

Debes tener en cuenta además las subvenciones que recibirás de los diferentes Ayuntamientos, las cuales pueden ser distintas en cada uno y estarán entre el 0% y el 100% de los costes mínimos.

La solución del problema debe incluir las cantidades a almacenar en cada ciudad bajo estas condiciones y el coste mínimo total de la operación de distribución para tu empresa.

La estrategia a seguir es la siguiente: La ciudad que sea más rentable (por coste de tranposrte/subvención dada) se transportará la máxima cantidad de productos posibles hasta la misma, dejando lo mínimo para las demás ciudades.

Este ejercicio es satánico.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;

pair<vector<unsigned int>, double> Distribucion(GrafoP<unsigned
int>::vertice &origen, size_t cantidadtotal, const GrafoP<unsigned int>
&costesviaje, const vector<unsigned int> &almacenamiento, const
vector<double> &subvencion);
```

```
int main()
{ /*pruebas....*/
}

pair<vector<unsigned int>, double> Distribucion(GrafoP<unsigned
int>::vertice &origen, size_t cantidadtotal, const GrafoP<unsigned int>
&costesviaje, const vector<unsigned int> &almacenamiento, const
vector<double> &subvencion)
{
   size_t cantidadrestante = cantidadtotal;
   vector<unsigned int> cantalmacenar(costesviaje.numVert(), 0);
   double coste = 0.;
   vector<GrafoP<unsigned int>::vertice> P;
   vector<unsigned int> A;

   // voy a considerar que es necesario volver al centro de producción
cada vez que abasteces a una ciudad, aunque el problema te lo deja en
el aire pa variar

   A = Dijkstra(costesviaje, origen, P);

   // Procedo a buscar la ciudad que más me interesa según la relación
coste mínimo del viaje/subvención aportada

   size_t ciudad;
   double mejorc = __DBL_MAX_10_EXP__;

   for (size_t i = 0; i < A.size(); i++)
   {
       if (A[i] / subvencion[i] < mejorc)
       {
           mejorc = A[i] / subvencion[i];
           ciudad = i;
       }
   }

   // una vez encontrada la mejor ciudad en relación coste
viaje/subvención, procedo a guardar la máxima cantidad de productos en
dicha ciudad sin pasarme
   // ya que he de abastecer a las demás ciudades también.
```

Orbit ¿DÍA DE CLASES infinitas?

```
   if (almacenamiento[ciudad] > cantidadtotal) // debemos guardar en
dicho almacen cantidadtotal-numerociudades, para luego tener un
producto para llevar a cada ciudad
   {
       cantalmacenar[ciudad] = cantidadtotal - costesviaje.numVert();
       cantidadrestante = costesviaje.numVert();
       coste += A[ciudad] * (subvencion[ciudad] / 100);

       // posteriormente, volvemos al almacen (coste despreciable segun
el enunciado) y abastecemos a las ciudades restantes. Producto por
ciudad
       for (size_t i = 0; i < costesviaje.numVert(); i++)
       {
           if (i != ciudad) // no contamos la ciudad ya abastecida
           {
               cantalmacenar[i] = 1;
               cantidadrestante--;
               coste += A[i] * (subvencion[i] / 100);
           }
       }
   }

   else // la cantidad que poseemos es mayor a la cantidad total de la
ciudad que más nos compensa, guardamos todo lo que podamos pero
teniendo en cuenta las demas ciudades
   {
       if (cantidadtotal - almacenamiento[ciudad] >
costesviaje.numVert()) // hemos de buscar de nuevo, la segunda ciudad
que más nos compensa para almacenar
       {
           cantalmacenar[ciudad] = almacenamiento[ciudad];
           cantidadrestante -= almacenamiento[ciudad];
           coste += A[ciudad] * (subvencion[ciudad] / 100);

           A.erase(A.begin() + ciudad); // eliminamos la ciudad
previamente escogida para buscar la nueva que más nos interesa

           double mejorc = __DBL_MAX_10_EXP__;

           for (size_t i = 0; i < A.size(); i++)
           {
               if (A[i] / subvencion[i] < mejorc)
               {
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S
masca y fluye

```
                    mejorc = A[i] / subvencion[i];
                    ciudad = i;
                }
            }

            // ahora repetimos el proceso
            if (almacenamiento[ciudad] > cantidadrestante) // debemos
guardar en dicho almacen canitdadrestante-numerociudades, para luego
tener un producto para llevar a cada ciudad
            {
                cantalmacenar[ciudad] = cantidadrestante -
costesviaje.numVert();
                cantidadrestante = costesviaje.numVert();
                coste += A[ciudad] * (subvencion[ciudad] / 100);

                // posteriormente, volvemos al almacen (coste
despreciable segun el enunciado) y abastecemos a las ciudades
restantes. Producto por ciudad
                for (size_t i = 0; i < costesviaje.numVert(); i++)
                {
                    if (i != ciudad) // no contamos la ciudad ya
abastecida
                    {
                        cantalmacenar[i] = 1;
                        cantidadrestante--;
                        coste += A[i] * (subvencion[i] / 100);
                    }
                }
            }

            else
            {
                cantalmacenar[ciudad] = cantidadrestante -
(almacenamiento[ciudad] - costesviaje.numVert());
                cantidadrestante -= cantidadrestante -
(almacenamiento[ciudad] - costesviaje.numVert());
                coste += A[ciudad] * (subvencion[ciudad] / 100);

                // procedemos ahora a repartir entre todas las demas
ciudades partiendo de la base
                // posteriormente, volvemos al almacen (coste
despreciable segun el enunciado) y abastecemos a las ciudades
restantes. Producto por ciudad
```

```
                for (size_t i = 0; i < costesviaje.numVert(); i++)
                {
                    if (i != ciudad) // no contamos la ciudad ya
abastecida
                    {
                        cantalmacenar[i] = 1;
                        cantidadrestante--;
                        coste += A[i] * (subvencion[i] / 100);
                    }
                }
            }
        }

        else // hemos de guardar la diferencia entre cantidadtotal y
(almacenamiento de la ciudad - numero ciudades)
        {
            cantalmacenar[ciudad] = cantidadtotal -
(almacenamiento[ciudad] - costesviaje.numVert());
            cantidadrestante -= cantidadtotal - (almacenamiento[ciudad]
- costesviaje.numVert());
            coste += A[ciudad] * (subvencion[ciudad] / 100);

            // procedemos ahora a repartir entre todas las demas
ciudades partiendo de la base
            // posteriormente, volvemos al almacen (coste despreciable
segun el enunciado) y abastecemos a las ciudades restantes. Producto
por ciudad
            for (size_t i = 0; i < costesviaje.numVert(); i++)
            {
                if (i != ciudad) // no contamos la ciudad ya abastecida
                {
                    cantalmacenar[i] = 1;
                    cantidadrestante--;
                    coste += A[i] * (subvencion[i] / 100);
                }
            }
        }
    }

    return {cantalmacenar, coste};
}
```

Orbit ¿DÍA DE CLASES infinitas?

Ejercicio 4:

4. Eres el orgulloso dueño de la empresa "Cementos de Zuelandia S.A". Empresa dedicada a la fabricación y distribución de cemento, sita en la capital de Zuelandia. Para la distribución del cemento entre tus diferentes clientes (ciudades de Zuelandia) dispones de una flota de camiones y de una plantilla de conductores zuelandeses.

El problema a resolver tiene que ver con el carácter del zuelandés. El zuelandés es una persona que se toma demasiadas "libertades" en su trabajo, de hecho, tienes fundadas sospechas de que tus conductores utilizan los camiones de la empresa para usos particulares (es decir indebidos, y a tu costa) por lo que quieres controlar los kilómetros que recorren tus camiones.

Todos los días se genera el parte de trabajo, en el que se incluyen el número de cargas de cemento (1 carga = 1 camión lleno de cemento) que debes enviar a cada cliente (cliente = ciudad de Zuelandia). Es innecesario indicar que no todos los días hay que enviar cargas a todos los clientes, y además, puedes suponer razonablemente que tu flota de camiones es capaz de hacer el trabajo diario.

Para la resolución del problema quizá sea interesante recordar que Zuelandia es un país cuya especial orografía sólo permite que las carreteras tengan un sentido de circulación.

Implementa una función que dado el grafo con las distancias directas entre las diferentes ciudades zuelandesas, el parte de trabajo diario, y la capital de Zuelandia, devuelva la distancia total en kilómetros que deben recorrer tus camiones en el día, para que puedas descubrir si es cierto o no que usan tus camiones en actividades ajenas a la empresa.

Este problema básicamente consiste en hacer Dijkstra y Dijkstra inverso para hallar la distancia mínima entre origen y las ciudades y viceversa, y a partir del parte de trabajo calcular la distancia mínima total necesaria para transportar las cargas de cemento.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S masca y fluye

```
struct PartesTrabajo
{
   size_t numcargas;
   GrafoP<unsigned int>::vertice destino;
};



double distanciaMinimaNecesaria(const GrafoP<unsigned
int>&g,GrafoP<unsigned int>::vertice& capital,vector<PartesTrabajo>&
partes);



int main()
{
   GrafoP<unsigned int> g("grafo.txt");
   vector<PartesTrabajo> v;
   GrafoP<unsigned int>::vertice cap=0;
   PartesTrabajo pt1,pt2;

   pt1.destino=2;
   pt1.numcargas=2;

   pt2.destino=4;
   pt2.numcargas=3;

   v.insert(v.end(),pt1);
   v.insert(v.end(),pt2);

   double dist=distanciaMinimaNecesaria(g,cap,v);

   cout<<"La distancia minima necesaria es: "<<dist<<endl;

   return 0;
}
```

```
double distanciaMinimaNecesaria(const GrafoP<unsigned
int>&g,GrafoP<unsigned int>::vertice& capital,vector<PartesTrabajo>&
partes)
{
   double dist=0.;
   //matrices necesarias para Dijkstra y Dijkstra inverso
   vector<GrafoP<unsigned int>::vertice> P1,P2;
   vector<unsigned int> A1,A2;

   A1=Dijkstra(g,capital,P1);
   A2=Dijkstrainv(g,capital,P2);

   cout<<A1<<endl;
   cout<<A2<<endl;

   //procedo a recorrer el vector de Partes para saber la distancia
minima total

   for(size_t i=0;i<partes.size();i++)
   {
       PartesTrabajo pt=partes[i];

       dist+= A1[pt.destino] * pt.numcargas;
       dist+=A2[pt.destino]*pt.numcargas;
   }




   /*versión con Floyd
   matriz<GrafoP<unsigned int>::vertice> P;
   matriz<unsigned int> A;


   A=Floyd(g,P);


   //procedo a recorrer el vector de Partes para saber la distancia
minima total
   for(size_t i=0;i<partes.size();i++)
   {
       PartesTrabajo pt=partes[i];
```

Orbit ¿DÍA DE CLASES infinitas?

```
        dist+=A[capital][pt.destino] * pt.numcargas;
        dist+=A[pt.destino][capital] * pt.numcargas;
    }

    */


    return dist;
}
```

5. Se dispone de tres grafos que representan la matriz de costes para viajes en un determinado país pero por diferentes medios de transporte, por supuesto todos los grafos tendrán el mismo número de nodos. El primer grafo representa los costes de ir por carretera, el segundo en tren y el tercero en avión. Dado un viajero que dispone de una determinada cantidad de dinero, que es alérgico a uno de los tres medios de transporte, y que sale de una ciudad determinada, implementar un subprograma que determine las ciudades a las que podría llegar nuestro infatigable viajero.

Para comenzar descartamos una matriz ya que el problema dice que es alérgico a uno de los medios de transporte. Luego, lo que debemos de hacer es un Dijkstra desde el origen para determinar el coste mínimo de llegar a las demás ciudades, y a partir de ahí vemos a partir del dinero que posee el viajero, las ciudades a las que puede llegar.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;



vector<GrafoP<unsigned int>::vertice> infatigableViajero(const
GrafoP<unsigned int>&carretera, const GrafoP<unsigned int>&tren, const
GrafoP<unsigned int>&avion, size_t alergia,
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S masca y fluye

```
unsigned int dinerodisponible, const GrafoP<unsigned int>::vertice
&origen);



int main()
{
   GrafoP<unsigned int> g("grafo.txt");
   GrafoP<unsigned int> g2("grafo2.txt");
   GrafoP<unsigned int> g3("grafo3.txt");

   GrafoP<unsigned int>::vertice origen=0;

   size_t alergico=3;

   unsigned int dinero=40;

   vector<GrafoP<unsigned int>::vertice>
ciud=infatigableViajero(g,g2,g3,alergico,dinero,origen);

   cout<<"El vector de las ciudades es: "<<ciud<<endl;

   return 0;
}



vector<GrafoP<unsigned int>::vertice> infatigableViajero(const
GrafoP<unsigned int>&carretera, const GrafoP<unsigned int>&tren, const
GrafoP<unsigned int>&avion, size_t alergia,
unsigned int dinerodisponible, const GrafoP<unsigned int>::vertice
&origen)
{
   //este algoritmo devuelve las ciudades a las que puede viajar en
orden, es decir, primero devuelve las ciudades a las que puede viajar
en carretera y tren por ejemplo
   //es posible que se repitan ciudades, por eso se devuelve en orden.

   vector<GrafoP<unsigned int>::vertice> v;
   vector<unsigned int> A1,A2;
   vector<GrafoP<unsigned int>::vertice> P1,P2;
```

```
switch (alergia)
{
case 1: //alergico a carretera
    A1=Dijkstra(tren,origen,P1);
    A2=Dijkstra(avion,origen,P2);

    //procedemos a ver a que ciudades puede llegar
    for(size_t i=0;i<A1.size();i++)
    {
        if(i!=origen)
            if(A1[i]<dinerodisponible)
                v.insert(v.end(),i);

    }
    for(size_t i=0;i<A2.size();i++)
    {
        if(i!=origen)
            if(A2[i]<dinerodisponible)
                v.insert(v.end(),i);

    }

    break;

case 2: //alergico a tren
    A1=Dijkstra(carretera,origen,P1);
    A2=Dijkstra(avion,origen,P2);

    //procedemos a ver a que ciudades puede llegar
    for(size_t i=0;i<A1.size();i++)
    {
        if(i!=origen)
            if(A1[i]<dinerodisponible)
                v.insert(v.end(),i);

    }
    for(size_t i=0;i<A2.size();i++)
    {
        if(i!=origen)
            if(A2[i]<dinerodisponible)
                v.insert(v.end(),i);
```

Orbit ¿DÍA DE CLASES infinitas?

```
        }


        break;


    case 3: //alergico a avion

        A1=Dijkstra(carretera,origen,P1);
        A2=Dijkstra(tren,origen,P2);

        //procedemos a ver a que ciudades puede llegar
        for(size_t i=0;i<A1.size();i++)
        {
            if(i!=origen)
                if(A1[i]<=dinerodisponible)
                    v.insert(v.end(),i);

        }
        for(size_t i=0;i<A2.size();i++)
        {
            if(i!=origen)
                if(A2[i]<dinerodisponible)
                    v.insert(v.end(),i);

        }

    break;

    default:
        break;
    }

    return v;
}
```

masca y fluye

6. Al dueño de una agencia de transportes se le plantea la siguiente situación. La agencia de viajes ofrece distintas trayectorias combinadas entre N ciudades españolas utilizando tren y autobús. Se dispone de dos grafos que representan los costes (matriz de costes) de viajar entre diferentes ciudades, por un lado en tren, y por otro en autobús (por supuesto entre las ciudades que tengan línea directa entre ellas). Además coincide que los taxis de toda España se encuentran en estos momentos en huelga general, lo que implica que sólo se podrá cambiar de transporte en una ciudad determinada en la que, por casualidad, las estaciones de tren y autobús están unidas.

Implementa una función que calcule la tarifa mínima (matriz de costes mínimos) de viajar entre cualesquiera de las N ciudades disponiendo del grafo de costes en autobús, del grafo de costes en tren, y de la ciudad que tiene las estaciones unidas.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;

matriz<unsigned int> agenciatransportes(const GrafoP<unsigned int>
&tren, const GrafoP<unsigned int> &bus, GrafoP<unsigned int>::vertice
cambio);

int main()
{
   GrafoP<unsigned int> g1("grafo.txt");
   GrafoP<unsigned int> g2("grafo2.txt");
   GrafoP<unsigned int>::vertice cambio = 3;
   matriz<unsigned int> m = agenciatransportes(g1, g2, cambio);

   cout << "La matriz resultante es: " << endl;
   cout << m << endl;

   return 0;
}
```

```
matriz<unsigned int> agenciatransportes(const GrafoP<unsigned int>
&tren, const GrafoP<unsigned int> &bus, GrafoP<unsigned int>::vertice
cambio)
{
   matriz<unsigned int> gfinal(bus.numVert());
   matriz<unsigned int> A1, A2;
   matriz<GrafoP<unsigned int>::vertice> P1, P2;

   A1 = Floyd(tren, P1);
   A2 = Floyd(bus, P2);

   // procedo a hallar el mínimo para cada vértice

   unsigned int costemin = GrafoP<unsigned int>::INFINITO;


   cout<<tren<<endl;
   cout<<bus<<endl;
   cout<<A1<<endl;
   cout<<A2<<endl;


   for (size_t i = 0; i < A1.dimension(); i++)
   {
       for (size_t j = 0; j < A1.dimension(); j++)
       {
           if (i != j)
           {
               if (A1[i][j] < costemin) // coste directo en tren
                   costemin = A1[i][j];

               if (A2[i][j] < costemin) // coste directo en bus
                   costemin = A2[i][j];

               if (A1[i][cambio]!=GrafoP<unsigned int>::INFINITO &&
A2[cambio][j]!=GrafoP<unsigned int>::INFINITO && (A1[i][cambio] +
A2[cambio][j] < costemin)) // coste de partir en tren, ir a la ciudad
de cambio y terminar en bus
                   costemin = A1[i][cambio] + A2[cambio][j];

               if (A2[i][cambio]!=GrafoP<unsigned int>::INFINITO &&
A1[cambio][j]!=GrafoP<unsigned int>::INFINITO && (A2[i][cambio] +
```

Orbit ¿DÍA DE CLASES infinitas?

```
A1[cambio][j] < costemin)) // coste de partir  en bus, ir a la ciudad
de cambio y terminar en tren
                    costemin =A2[i][cambio] + A1[cambio][j];

                if (costemin == GrafoP<unsigned int>::INFINITO)
                    gfinal[i][j] = GrafoP<unsigned int>::INFINITO;

                else
                    gfinal[i][j] = costemin;
            }

            else // coste de ir de una ciudad a si misma 0
                gfinal[i][j] = 0;

            costemin = GrafoP<unsigned int>::INFINITO;
        }
    }

    return gfinal;
}
```

masca y fluye

7. Se dispone de dos grafos (matriz de costes) que representan los costes de viajar entre *N* ciudades españolas utilizando el tren (primer grafo) y el autobús (segundo grafo). Ambos grafos representan viajes entre las mismas *N* ciudades.

Nuestro objetivo es hallar el camino de coste mínimo para viajar entre dos ciudades concretas del grafo, *origen* y *destino*, en las siguientes condiciones:

- La ciudad *origen* sólo dispone de transporte por tren.
- La ciudad *destino* sólo dispone de transporte por autobús.
- El sector del taxi, bastante conflictivo en nuestros problemas, sigue en huelga, por lo que únicamente es posible cambiar de transporte en dos ciudades del grafo, *cambio1* y *cambio2*, donde las estaciones de tren y autobús están unidas.

Implementa un subprograma que calcule la ruta y el coste mínimo para viajar entre las ciudades *Origen* y *Destino* en estas condiciones.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;

pair<GrafoP<unsigned int>::tCamino, unsigned int> doscambios(const
GrafoP<unsigned int> &tren, const GrafoP<unsigned int> &bus,
GrafoP<unsigned int>::vertice origen,

GrafoP<unsigned int>::vertice destino, GrafoP<unsigned int>::vertice
cambio1, GrafoP<unsigned int>::vertice cambio2);

template <typename T>
Lista<T> invertirLista(const Lista<T> &original)
{
   Lista<T> invertida;
   for (typename Lista<T>::posicion p = original.primera(); p !=
original.fin(); p = original.siguiente(p))
   {
```

```
        invertida.insertar(original.elemento(p), invertida.primera());
// lo vamos insertando todo al principio
    }
    return invertida;
}

int main()
{
}

pair<GrafoP<unsigned int>::tCamino, unsigned int> doscambios(const
GrafoP<unsigned int> &tren, const GrafoP<unsigned int> &bus,
GrafoP<unsigned int>::vertice origen,

GrafoP<unsigned int>::vertice destino, GrafoP<unsigned int>::vertice
cambio1, GrafoP<unsigned int>::vertice cambio2)
{
    vector<GrafoP<unsigned int>::vertice> P1, P2;
    vector<unsigned int> A1, A2;
    GrafoP<unsigned int>::tCamino cam;
    GrafoP<unsigned int>::tCamino cam2;
    GrafoP<unsigned int>::tCamino camfinal;

    A1 = Dijkstra(tren, origen, P1);
    A2 = Dijkstrainv(bus, destino, P2);

    // procedo a mirar con cuales de los cambios es más rentable
    unsigned int costemin = GrafoP<unsigned int>::INFINITO;
    GrafoP<unsigned int>::vertice mejorCambio;

    if (A1[cambio1] + A2[cambio1] < costemin)
    {
        costemin = A1[cambio1] + A2[cambio1];
        mejorCambio = cambio1;
    }

    if (A1[cambio2] + A2[cambio2] < costemin)
    {
        costemin = A1[cambio2] + A2[cambio2];
        mejorCambio = cambio2;
    }

    cam = camino<unsigned int>(origen, mejorCambio, P1);
```

8. “UN SOLO TRANSBORDO, POR FAVOR”. Este es el título que reza en tu flamante compañía de viajes. Tu publicidad explica, por supuesto, que ofreces viajes combinados de TREN y/o AUTOBÚS (es decir, viajes en tren, en autobús, o usando ambos), entre *N* ciudades del país, que ofreces un servicio inmejorable, precios muy competitivos, y que garantizas ante notario algo que no ofrece ninguno de tus competidores: que en todos tus viajes COMO MÁXIMO se hará un solo transbordo (cambio de medio de transporte).

Bien, hoy es 1 de Julio y comienza la temporada de viajes.

¡Qué suerte! Acaba de aparecer un cliente en tu oficina. Te explica que quiere viajar entre dos ciudades, *Origen* y *Destino*, y quiere saber cuánto le costará.

Para responder a esa pregunta dispones de dos grafos de costes directos (matriz de costes) de viajar entre las *N* ciudades del país, un grafo con los costes de viajar en tren y otro en autobús.

Implementa un subprograma que calcule la tarifa mínima en estas condiciones.

Mucha suerte en el negocio, que la competencia es dura.

Basta con hacer Dijkstra y Dijkstra inverso en cada grafo e ir mirando cual compensa más.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;



unsigned int unSoloTransbordo(const GrafoP<unsigned int>&tren,const
GrafoP<unsigned int>&bus,GrafoP<unsigned int>::vertice& origen,
GrafoP<unsigned int>::vertice& destino);









int main()
{

   GrafoP<unsigned int> g1("grafo.txt");
   GrafoP<unsigned int> g2("grafo2.txt");
   GrafoP<unsigned int>::vertice origen,destino;
   origen=0;
   destino=4;

   unsigned int coste=unSoloTransbordo(g1,g2,origen,destino);
```

```
    cout<<"El coste minimo de ir de origen a destino es: "<<coste<<endl;
    return 0;
}




unsigned int unSoloTransbordo(const GrafoP<unsigned int>&tren,const
GrafoP<unsigned int>&bus,GrafoP<unsigned int>::vertice& origen,
GrafoP<unsigned int>::vertice& destino)
{
    unsigned int coste=GrafoP<unsigned int>::INFINITO;

    vector<GrafoP<unsigned int>::vertice> P1,P2,P3,P4;
    vector<unsigned int> A1,A2,A3,A4;

    A1=Dijkstra(tren,origen,P1);
    A2=Dijkstrainv(tren,destino,P2);

    A3=Dijkstra(bus,origen,P3);
    A4=Dijkstrainv(bus,destino,P4);


    if(A1[destino]<coste) //coste directo tren
        coste=A1[destino];


    if(A3[destino]<coste) //coste directo bus
        coste=A2[destino];



    for(size_t i=0;i<A1.size();i++)
    {
        if(i!=destino && A1[i]!=GrafoP<unsigned int>::INFINITO &&
A4[i]!= GrafoP<unsigned int>::INFINITO &&(A1[i]+A4[i]<coste)) //
transbordo en tren y llegada en bus
            coste=A1[i]+A2[i];
```

Orbit ¿DÍA DE CLASES infinitas?

```
        if(i!=destino && A3[i]!=GrafoP<unsigned int>::INFINITO &&
A2[i]!= GrafoP<unsigned int>::INFINITO && (A3[i] + A2[i] < coste))
//transbordo en bus y llegada en tren
            coste=A3[i] + A2[i];
    }



    return coste;
}
```

9. Se dispone de dos grafos que representan la matriz de costes para viajes en un determinado país, pero por diferentes medios de transporte (tren y autobús, por ejemplo). Por supuesto ambos grafos tendrán el mismo número de nodos, *N*. Dados ambos grafos, una ciudad de origen, una ciudad de destino y el coste del taxi para cambiar de una estación a otra dentro de cualquier ciudad (se supone constante e igual para todas las ciudades), implementa un subprograma que calcule el camino y el coste mínimo para ir de la ciudad origen a la ciudad destino.

Aquí ya entramos con los supergrafos, ya que la cantidad de cambios que podemos hacer entre las ciudades no está limitada, y el coste de dicho cambio entre medio de transporte ya no es 0.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
#include"imprimecamino.h"
using namespace std;



pair<GrafoP<unsigned int>::tCamino, unsigned int> transbordoTaxi(const
GrafoP<unsigned int>&tren,const GrafoP<unsigned int>&bus,unsigned int
costetaxi,
GrafoP<unsigned int>::vertice origen,GrafoP<unsigned int>::vertice
destino);
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S masca y fluye

```cpp
int main()
{
   GrafoP<unsigned int> g("grafo.txt");
   GrafoP<unsigned int> g1("grafo2.txt");
   GrafoP<unsigned int>::vertice origen=0;
   GrafoP<unsigned int>::vertice destino=4;
   auto [cam, coste] = transbordoTaxi(g, g1, 15, origen, destino);

   imprimeCamino(cam);

   cout<<"El coste mínimo del viaje es: "<<coste<<endl;


   return 0;
}




pair<GrafoP<unsigned int>::tCamino, unsigned int> transbordoTaxi(const
GrafoP<unsigned int>&tren,const GrafoP<unsigned int>&bus,unsigned int
costetaxi,
GrafoP<unsigned int>::vertice origen,GrafoP<unsigned int>::vertice
destino)
{
   GrafoP<unsigned int> g(tren.numVert()+bus.numVert());

   GrafoP<unsigned int>::tCamino cam;
   unsigned int costemintotal=0;


   //procedemos a rellenar el supergrafo

   for(size_t i=0;i<tren.numVert();i++)
   {
       for(size_t j=0;j<tren.numVert();j++)
       {
           g[i][j]=tren[i][j];
       }
   }
```

```
    auto k=0;
    auto l=0;

    for(size_t i=tren.numVert();i<tren.numVert() + bus.numVert();i++)
    {
        for(size_t j=tren.numVert();j<tren.numVert() +
bus.numVert();j++)
        {
            g[i][j]=bus[k][l];

            l++;
        }
        k++;
        l=0;
    }


    //ahora procedo a conectar las ciudades con el coste del taxi

    for(size_t i=0;i<g.numVert();i++)
    {
        for(size_t j=0;j<g.numVert();j++)
        {
            if(i + tren.numVert() == j)
            {
                g[i][j]=costetaxi;
                g[j][i]=costetaxi;
            }
        }
    }


    cout<<"Supergrafo construido: "<<endl;
    cout<<g<<endl;

    //procedo a aplicar Dijkstra
    vector<unsigned int> A,A1;
    vector<GrafoP<unsigned int>::vertice> P,P1;
    A=Dijkstra(g,origen,P);
    A1=Dijkstra(g,origen+bus.numVert(),P1);
```

Orbit ¿DÍA DE CLASES infinitas?

```
    if(A[destino] <= A1[destino])
    {
        cam=camino<unsigned int>(origen,destino,P);

        costemintotal=A[destino];
    }

    else
    {
        cam=camino<unsigned int>(origen + bus.numVert(),destino,P1);
        costemintotal=A1[destino];
    }

    return {cam,costemintotal};
}
```

10. Se dispone de tres grafos que representan la matriz de costes para viajes en un determinado país, pero por diferentes medios de transporte (tren, autobús y avión). Por supuesto los tres grafos tendrán el mismo número de nodos, *N*.

Dados los siguientes datos:

- los tres grafos,
- una ciudad de origen,
- una ciudad de destino,
- el coste del taxi para cambiar, dentro de una ciudad, de la estación de tren a la de autobús o viceversa (*taxi-tren-bus*) y
- el coste del taxi desde el aeropuerto a la estación de tren o la de autobús, o viceversa (*taxi-aeropuerto-tren/bus*)

y asumiendo que ambos costes de taxi (distintos entre sí, son dos costes diferentes) son constantes e iguales para todas las ciudades, implementa un subprograma que calcule el camino y el coste mínimo para ir de la ciudad origen a la ciudad destino.

Exactamente igual que el anterior pero añadiendo un grafo más y un coste más

```
pair<GrafoP<unsigned int>::tCamino, unsigned int>
transbordoTaxi3Grafos(const GrafoP<unsigned int> &tren, const
GrafoP<unsigned int> &bus, const GrafoP<unsigned int> &avion,

unsigned int costetaxiestacion, unsigned int costetaxiaeropuerto,
```

masca y fluye

```
GrafoP<unsigned int>::vertice origen, GrafoP<unsigned int>::vertice
destino)
{

   GrafoP<unsigned int>::tCamino cam;
   unsigned int costemintotal = GrafoP<unsigned int>::INFINITO;
   vector<unsigned int> A1, A2, A3;
   vector<GrafoP<unsigned int>::vertice> P1, P2, P3;
   GrafoP<unsigned int> g(tren.numVert() + bus.numVert() +
avion.numVert());

   // procedo a montar el supergrafo
   for (size_t i = 0; i < tren.numVert(); i++)
   {
       for (size_t j = 0; j < tren.numVert(); j++)
       {
           g[i][j] = tren[i][j];
       }
   }

   size_t k = 0, l = 0;

   for (size_t i = tren.numVert(); i < tren.numVert() + bus.numVert();
i++)
   {
       for (size_t j = tren.numVert(); j < tren.numVert() +
bus.numVert(); j++)
       {
           g[i][j] = bus[k][l];
           l++;
       }
       l = 0;
       k++;
   }

   k = 0, l = 0;

   for (size_t i = tren.numVert() + bus.numVert(); i < tren.numVert() +
bus.numVert() + avion.numVert(); i++)
   {
       for (size_t j = tren.numVert() + bus.numVert(); j <
tren.numVert() + bus.numVert() + avion.numVert(); j++)
       {
```

```
            g[i][j] = avion[k][l];
            l++;
        }
        l = 0;
        k++;
    }

    // procedo a conectar las ciudades correspondientes con su coste
correspondiente
    for (size_t i = 0; i < tren.numVert(); i++)
    {
        for (size_t j = 0; j < tren.numVert(); j++)
        {
            size_t cbus = i;
            size_t ctren = i + tren.numVert();
            size_t cavion = i + 2 * tren.numVert();

            g[cbus][ctren] = costetaxiestacion;
            g[ctren][cbus] = costetaxiestacion;

            g[cbus][cavion] = costetaxiaeropuerto;
            g[cavion][cbus] = costetaxiaeropuerto;

            g[ctren][cavion] = costetaxiaeropuerto;
            g[cavion][ctren] = costetaxiaeropuerto;
        }
    }

    cout << "Supergrafo construido: " << endl;
    cout << g << endl;

    A1 = Dijkstra(g, origen, P1);                          // desde tren
    A2 = Dijkstra(g, origen + tren.numVert(), P2);      // desde bus
    A3 = Dijkstra(g, origen + 2 * tren.numVert(), P3); // desde avión

    unsigned int minCost = GrafoP<unsigned int>::INFINITO;
    vector<GrafoP<unsigned int>::vertice> bestP;
    GrafoP<unsigned int>::vertice bestStart = 0, bestEnd = 0;

    // Recorrido desde tren
    if (A1[destino] < minCost)
    {
        minCost = A1[destino];
```

```
    if (A3[destino] < minCost)
    {
        minCost = A3[destino];
        bestStart = origen + 2 * tren.numVert();
        bestEnd = destino;
        bestP = P3;
    }
    if (A3[destino + tren.numVert()] < minCost)
    {
        minCost = A3[destino + tren.numVert()];
        bestStart = origen + 2 * tren.numVert();
        bestEnd = destino + tren.numVert();
        bestP = P3;
    }
    if (A3[destino + 2 * tren.numVert()] < minCost)
    {
        minCost = A3[destino + 2 * tren.numVert()];
        bestStart = origen + 2 * tren.numVert();
        bestEnd = destino + 2 * tren.numVert();
        bestP = P3;
    }


    cam=camino<unsigned int>(bestStart,bestEnd,bestP);

    return {cam, minCost};
}
```

11. Disponemos de tres grafos (matriz de costes) que representan los costes directos de viajar entre las ciudades de tres de las islas del archipiélago de las Huríes (Zuelandia). Para poder viajar de una isla a otra se dispone de una serie de puentes que conectan ciudades de las diferentes islas a un precio francamente asequible (por decisión del Prefecto de las Huríes, el uso de los puentes es absolutamente gratuito).

Si el alumno desea simplificar el problema, puede numerar las N_1 ciudades de la isla 1, del 0 al N_1-1, las N_2 ciudades de la isla 2, del N_1 al N_1+N_2-1, y las N_3 de la última, del N_1+N_2 al $N_1+N_2+N_3-1$.

Disponiendo de las tres matrices de costes directos de viajar dentro de cada una de las islas, y la lista de puentes entre ciudades de las mismas, calculad los costes mínimos de viajar entre cualesquiera dos ciudades de estas tres islas.

¡¡¡ QUE DISFRUTÉIS EL VIAJE !!!

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
#include "imprimecamino.h"
using namespace std;


struct puente
{
   size_t i,j; //puente que va desde i a j y viceversa

};


matriz<unsigned int> Huries(const GrafoP<unsigned int>&isla1, const
GrafoP<unsigned int>&isla2, const GrafoP<unsigned int>&
isla3,vector<puente>& v);



int main()
{
   GrafoP<unsigned int> g("grafo.txt");
   GrafoP<unsigned int> g1("grafo2.txt");
   GrafoP<unsigned int> g2("grafo2.txt");

   puente p1,p2;
   p1.i=0;
   p1.j=6;
```

```
   for(size_t i=isla1.numVert(); i< isla1.numVert() + isla2.numVert();
i++)
   {
       for(size_t j=isla1.numVert(); j< isla1.numVert() +
isla2.numVert(); j++)
       {
           g[i][j]=isla2[k][l];
           l++;
       }

       l=0;
       k++;
   }

   k=0,l=0;

   for(size_t i=isla1.numVert() + isla2.numVert(); i< isla1.numVert() +
isla2.numVert() + isla3.numVert(); i++)
   {
       for(size_t j=isla1.numVert() + isla2.numVert(); j<
isla1.numVert() + isla2.numVert() + isla3.numVert(); j++)
       {
           g[i][j]=isla3[k][l];
           l++;
       }

       l=0;
       k++;
   }


   //procedo a conectar las islas segun los puentes dados
   for(size_t i=0;i<v.size();i++)
   {
       g[v[i].i][v[i].j]=0;
       g[v[i].j][v[i].i]=0;
   }

   m=Floyd(g,P);


   return m;
```

```
}
```

12. El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen N_1 y N_2 ciudades, respectivamente, de las cuales C_1 y C_2 ciudades son costeras (obviamente $C_1 \leq N_1$ y $C_2 \leq N_2$). Se desea construir un puente que una ambas islas. Nuestro problema es elegir el puente a construir entre todos los posibles, sabiendo que el coste de construcción del puente se considera irrelevante. Por tanto, escogeremos aquel puente que minimice el coste global de viajar entre todas las ciudades de las dos islas, teniendo en cuenta las siguientes premisas:

1. Se asume que el coste viajar entre las dos ciudades que una el puente es 0.
2. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipiélago. Por ejemplo, se considerará que el número de viajes entre la ciudad P de Fobos y la Q de Deimos será el mismo que entre las ciudades R y S de la misma isla. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

Dadas las matrices de costes directos de Fobos y Deimos y las listas de ciudades costeras de ambas islas, implementa un subprograma que calcule las dos ciudades que unirá el puente.

El problema se basa en a partir de las ciudades costeras de una isla y otra, ir probando puentes para quedarte con el menor coste total. Para ello, debemos montar el supergrafo correspondiente.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
#include "imprimecamino.h"
using namespace std;



pair<GrafoP<unsigned int>::vertice,GrafoP<unsigned int>::vertice>
Grecoland(const GrafoP<unsigned int>&Fobos, const GrafoP<unsigned
int>&Deimos,
vector<size_t>& costerasFobos, vector<size_t>&costerasDeimos);
```

```
        }
    }

    size_t k=0;
    size_t l=0;

    for(size_t i=Fobos.numVert(); i< Fobos.numVert() +
Deimos.numVert();i++)
    {
        for(size_t j=Fobos.numVert(); j< Fobos.numVert()+
Deimos.numVert();j++)
        {
            g[i][j]=Deimos[k][l];
            l++;
        }
        k++;
        l=0;
    }

    //a continuación, procedo a probar 8todos los puentes posibles, y
haciendo Floyd, hallo aquellas ciudades que minimicen el coste
    for(size_t i=0;i<costerasFobos.size();i++)
    {
        for(size_t j=0;j<costerasDeimos.size();j++)
        {
            //aplico puente a coste 0
            g[costerasFobos[i]][costerasDeimos[j]]=0;
            g[costerasFobos[j]][costerasDeimos[i]]=0;


            A=Floyd(g,P);

            //procedo a calcular el coste total
            for(size_t o=0;o<A.dimension();o++)
            {
                for(size_t p=0;p<A.dimension();p++)
                {
                    if(A[o][p]!=GrafoP<unsigned int>::INFINITO)
                    costeactual+=A[o][p];
                }
            }

            //he encontrado un puente que minimiza el coste total
```

```
            if(costemintotal>costeactual)
            {
                costemintotal=costeactual;
                ciudad1=costerasFobos[i];
                ciudad2=costerasDeimos[j];
            }

            //deshago el cambio
            g[costerasFobos[i]][costerasDeimos[j]]=GrafoP<unsigned
int>::INFINITO;
            g[costerasFobos[j]][costerasDeimos[i]]=GrafoP<unsigned
int>::INFINITO;
        }

        costeactual=0;
    }




    return {ciudad1,ciudad2};
}
```

Orbit ¿DÍA DE CLASES infinitas?

13. El archipiélago de las Huríes acaba de ser devastado por un maremoto de dimensiones desconocidas hasta la fecha. La primera consecuencia ha sido que todos y cada uno de los puentes que unían las diferentes ciudades de las tres islas han sido destruidos. En misión de urgencia las Naciones Unidas han decidido construir el mínimo número de puentes que permitan unir las tres islas. Asumiendo que el coste de construcción de los puentes implicados los pagará la ONU, por lo que se considera irrelevante, nuestro problema es decidir qué puentes deben construirse. Las tres islas de las Huríes tienen respectivamente N_1, N_2 y N_3 ciudades, de las cuales C_1, C_2 y C_3 son costeras (obviamente $C_1 \leq N_1$, $C_2 \leq N_2$ y $C_3 \leq N_3$) . Nuestro problema es elegir los puentes a construir entre todos los posibles. Por tanto, escogeremos aquellos puentes que minimicen el coste global de viajar entre todas las ciudades de las tres islas, teniendo en cuenta las siguientes premisas:

1. Se asume que el coste viajar entre las ciudades que unan los puentes es 0.
2. La ONU subvencionará únicamente el número mínimo de puentes necesario para comunicar las tres islas.
3. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipélago. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

Dadas las matrices de costes directos de las tres islas y las listas de ciudades costeras del archipiélago, implementad un subprograma que calcule los puentes a construir en las condiciones anteriormente descritas.

Este ejercicio es exactamente igual que el anterior pero con 3 islas. El problema es que tenemos que conectar 2 puentes, teniendo 3 posiblidades:

- De la isla 1 a la 2 y de la 2 a la 3
- De la isla 1 a la 2 y de la 1 a la 3
- De la isla 2 a la 3 y de la 1 a la 3

Estas 3 posibilidades hacen el problema un poco más complejo. El resto se hace exactamente igual que el anterior.

Como pide calcular los puentes, voy a devolver la matriz de costes que contiene el coste total mínimo de viajar entre las 3 islas, ya que no sé exactamente cómo quiere que le devuelva los puentes.

```
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
```

masca y fluye

```
#include "imprimecamino.h"
using namespace std;



matriz<unsigned int> Huries(const GrafoP<unsigned int>&isla1, const
GrafoP<unsigned int>&isla2, const GrafoP<unsigned int>&isla3,
vector<GrafoP<unsigned int>::vertice>&costeras1,vector<GrafoP<unsigned
int>::vertice>&costeras2, vector<GrafoP<unsigned
int>::vertice>&costeras3);




int main()
{}




matriz<unsigned int> Huries(const GrafoP<unsigned int>&isla1, const
GrafoP<unsigned int>&isla2, const GrafoP<unsigned int>&isla3,
vector<GrafoP<unsigned int>::vertice>&costeras1,vector<GrafoP<unsigned
int>::vertice>&costeras2, vector<GrafoP<unsigned
int>::vertice>&costeras3)
{
   matriz<unsigned int> m;
   matriz<unsigned int> A;
   unsigned int costemin=GrafoP<unsigned int>::INFINITO;
   unsigned int costeactual=0;
   matriz<GrafoP<unsigned int>::vertice> P;
   GrafoP<unsigned int> g(isla1.numVert()+ isla2.numVert() +
isla3.numVert());


   //procedo a montar el supergrafo

   for(size_t i=0;i<isla1.numVert();i++)
   {
       for(size_t j=0;j<isla1.numVert();j++)
       {
           g[i][j]=isla1[i][j];
```

```
        }
    }

    size_t k=0,l=0;

    for(size_t i=isla1.numVert();i< isla1.numVert() +
isla2.numVert();i++)
    {
        for(size_t j=isla1.numVert();j<isla1.numVert() +
isla2.numVert();j++)
        {
            g[i][j]=isla2[k][l];
            l++;
        }
        l=0;
        k++;
    }

    l=0,k=0;



    for(size_t i=isla1.numVert() + isla2.numVert();i< isla1.numVert() +
isla2.numVert() + isla3.numVert();i++)
    {
        for(size_t j=isla1.numVert() + isla2.numVert();j<isla1.numVert()
+ isla2.numVert()  + isla3.numVert();j++)
        {
            g[i][j]=isla2[k][l];
            l++;
        }
        l=0;
        k++;
    }


    //a continuación procedo a poner los puentes
    for(size_t i=0;i<costeras1.size();i++)
    {
        for(size_t j=0;j<costeras2.size();j++)
        {

            for(size_t k=0;k<costeras3.size();k++)
```

Orbit ¿DÍA DE CLASES infinitas?

```
            {
                //puente entre isla1-isla2 y isla 2-isla3

                g[costeras1[i]][costeras2[j]]=0;
                g[costeras2[j]][costeras1[i]]=0;

                g[costeras2[j]][costeras3[k]]=0;
                g[costeras3[k]][costeras2[j]]=0;

                A=Floyd(g,P);

                //procedo a calcular el coste
                for(size_t l=0; l < A.dimension();l++)
                {
                    for(size_t n=0;n<A.dimension();n++)
                    {
                        costeactual+=A[l][n];
                    }
                }

                if(costeactual<costemin)
                {
                    costemin=costeactual;
                    m=A;
                }

                //deshago el cambio
                g[costeras1[i]][costeras2[j]]=GrafoP<unsigned
int>::INFINITO;
                g[costeras2[j]][costeras1[i]]=GrafoP<unsigned
int>::INFINITO;;

                g[costeras2[j]][costeras3[k]]=GrafoP<unsigned
int>::INFINITO;;
                g[costeras3[k]][costeras2[j]]=GrafoP<unsigned
int>::INFINITO;;



                //puente entre isla1-isla2, isla1-isla3
                g[costeras1[i]][costeras2[j]]=0;
                g[costeras2[j]][costeras1[i]]=0;
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S
masca y fluye

```
                g[costeras1[i]][costeras3[k]]=0;
                g[costeras3[k]][costeras1[i]]=0;


                A=Floyd(g,P);

                //procedo a calcular el coste
                for(size_t l=0; l < A.dimension();l++)
                {
                    for(size_t n=0;n<A.dimension();n++)
                    {
                        costeactual+=A[l][n];
                    }
                }

                if(costeactual<costemin)
                {
                    costemin=costeactual;
                    m=A;
                }


                //deshago el cambio
                g[costeras1[i]][costeras2[j]]=GrafoP<unsigned
int>::INFINITO;
                g[costeras2[j]][costeras1[i]]=GrafoP<unsigned
int>::INFINITO;

                g[costeras1[i]][costeras3[k]]=GrafoP<unsigned
int>::INFINITO;
                g[costeras3[k]][costeras1[i]]=GrafoP<unsigned
int>::INFINITO;




                //puente entre 2-3 y 1-3
                g[costeras2[j]][costeras3[k]]=0;
                g[costeras3[k]][costeras2[j]]=0;

                g[costeras1[i]][costeras3[k]]=0;
                g[costeras3[k]][costeras1[i]]=0;
```

```
                A=Floyd(g,P);

                //procedo a calcular el coste
                for(size_t l=0; l < A.dimension();l++)
                {
                    for(size_t n=0;n<A.dimension();n++)
                    {
                        costeactual+=A[l][n];
                    }
                }

                if(costeactual<costemin)
                {
                    costemin=costeactual;
                    m=A;
                }


                //deshago el cambio
                g[costeras2[j]][costeras3[k]]=GrafoP<unsigned
int>::INFINITO;
                g[costeras3[k]][costeras2[j]]=GrafoP<unsigned
int>::INFINITO;

                g[costeras1[i]][costeras3[k]]=GrafoP<unsigned
int>::INFINITO;
                g[costeras3[k]][costeras1[i]]=GrafoP<unsigned
int>::INFINITO;
            }


        }



    }
```

Ejercicios Práctica 8

1. El archipiélago de Tombuctú, está formado por un número indeterminado de islas, cada una de las cuales tiene, a su vez, un número indeterminado de ciudades. En cambio, sí es conocido el número total de ciudades de Tombuctú (podemos llamarlo *N*, por ejemplo).

Dentro de cada una de las islas existen carreteras que permiten viajar entre todas las ciudades de la isla. Se dispone de las coordenadas cartesianas *(x, y)* de todas y cada una de las ciudades del archipiélago. Se dispone de un grafo (matriz de adyacencia) en el que se indica si existe carretera directa entre cualesquiera dos ciudades del archipiélago. El objetivo de nuestro problema es encontrar qué ciudades de Tombuctú pertenecen a cada una de las islas del mismo y cuál es el coste mínimo de viajar entre cualesquiera dos ciudades de una misma isla de Tombuctú.

Así pues, dados los siguientes datos:

- Lista de ciudades de Tombuctú representada cada una de ellas por sus coordenadas cartesianas.

- Matriz de adyacencia de Tombuctú, que indica las carreteras existentes en dicho archipiélago.

Implementen un subprograma que calcule y devuelva la distribución en islas de las ciudades de Tombuctú, así como el coste mínimo de viajar entre cualesquiera dos ciudades de una misma isla del archipiélago.

Lo que haría en este problema es aplicar Warshall, para sacar directamente en la matriz de adyacencia las ciudades que pertenecen a una misma isla. Posteriormente, una vez hallado dicha matriz, construyo el grafo a partir de la distancia euclídea entre las ciudades.
Como no se me dice nada, voy a suponer que el grafo es no dirigido.

```
#include"alg_grafoMA.h"
#include"listaenla.h"
#include"pilaenla.h"
#include"particion.h"
#include"matriz.h"
#include"apo.h"
#include"alg_grafo_E-S.h"
```

```cpp
#include<iostream>
#include<math.h>
#include"alg_grafoPMC.h"
using namespace std;

#define N 5

struct ciudad
{
   size_t i,j;
};


tuple<GrafoP<double>,matriz<double>,Particion> Tombuctu(const
vector<ciudad>&ciudadesTombuctu,const Grafo&carreteras);


int main()
{
   Grafo g("islas.txt");

   cout<<"Comprobación de si el grafo es correcto: "<<endl;

   cout<<g<<endl;

    vector<ciudad> V = {
       {0, 0},     // 0
       {1, 2},     // 1
       {2, 1},     // 2
       {10, 10},   // 3
       {11, 12},   // 4
       {4, 3}      // 5
   };

   auto [grf,mat,part]=Tombuctu(V,g);

   cout<<"Grafo: "<<endl;
   cout<<grf<<endl;

   cout<<"Matriz: "<<endl;
   cout<<mat<<endl;
```

Orbit ¿DÍA DE CLASES infinitas?

```
    return 0;
}



tuple<GrafoP<double>,matriz<double>,Particion> Tombuctu(const
vector<ciudad>&ciudadesTombuctu,const Grafo&carreteras)
{
    //creamos el grafo
    GrafoP<double> g(carreteras.numVert());

    Particion p(carreteras.numVert());


    //agrupamos todas las ciudades que están en una misma isla
    for(size_t i=0;i<carreteras.numVert();i++)
    {
        for(size_t j=0;j<carreteras.numVert();j++)
        {
            if(carreteras[i][j] && p.encontrar(i)!= p.encontrar(j))
                p.unir(p.encontrar(i),p.encontrar(j));
        }
    }

    for(size_t i=0;i<carreteras.numVert();i++)
    {
        for(size_t j=0;j<carreteras.numVert();j++)
        {
            if(carreteras[i][j])
            {
                double dx= ciudadesTombuctu[i].i -
ciudadesTombuctu[j].i;
                double dy= ciudadesTombuctu[i].j -
ciudadesTombuctu[j].j;

                g[i][j]= sqrt(dx*dx + dy*dy);
            }
        }
    }

    //ya poseemos las ciudades que pertenecen a cada isla y sus costes
directos mediante su distancia euclídea. Procedemos a aplicar Floyd
```

masca y fluye

```
    matriz<double> A;
    matriz<GrafoP<unsigned int>::vertice> P;

    A=Floyd(g,P);

    return{g,A,p};
}
```

2. El archipiélago de Tombuctú2 está formado por un número desconocido de islas, cada una de las cuales tiene, a su vez, un número desconocido de ciudades, las cuales tienen en común que todas y cada una de ellas dispone de un aeropuerto. Sí que se conoce el número total de ciudades del archipiélago (podemos llamarlo N, por ejemplo).

Dentro de cada una de las islas existen carreteras que permiten viajar entre todas las ciudades de la isla. No existen puentes que unan las islas y se ha decidido que la opción de comunicación más económica de implantar será el avión.

Se dispone de las coordenadas cartesianas *(x, y)* de todas y cada una de las ciudades del archipiélago. Se dispone de un grafo (matriz de adyacencia) en el que se indica si existe carretera directa entre cualesquiera dos ciudades del archipiélago. El objetivo de nuestro problema es encontrar qué líneas aéreas debemos implantar para poder viajar entre todas las ciudades del archipiélago, siguiendo los siguientes criterios:

1) Se implantará una y sólo una línea aérea entre cada par de islas.
2) La línea aérea escogida entre cada par de islas será la más corta entre todas las posibles.

Así pues, dados los siguientes datos:

- Lista de ciudades de Tombuctú2 representada cada una de ellas por sus coordenadas cartesianas.
- Matriz de adyacencia de Tombuctú que indica las carreteras existentes en dicho archipiélago,

Implementen un subprograma que calcule y devuelva las líneas aéreas necesarias para comunicar adecuadamente el archipiélago siguiendo los criterios anteriormente expuestos.

- Todas las ciudades tienen aeropuerto.
- Conocemos número total de ciudades
- Islas conexas
- Disponemos de:
    - Coordenadas de todas las ciudades
    - Matriz de adyacencia
- Encontrar qué lineas aéreas para viajar entre ciudades sabiendo:
    - Una línea aérea entre cada par de islas
    - Dicha línea será la más corta

A ver, este problema para mí es el más difícil de todas las prácticas de grafos. Debemos montar un supergrafo de nuevo, donde debemos conectar una línea aérea entre cada par de islas, es decir, si tenemos 3 islas, deberíamos tener líneas aéreas desde la isla 1 a la 2, desde la isla 1 a la 3, desde la isla 2 a la 1, desde la 2 a la 3, desde la 3 a la 1 y desde la 3 a la 2, es decir, un total de 6 líneas aéreas.

Como las posibilidades de conectar islas es inmensa, ya que cada ciudad de cada isla posee un aeropuerto y eso permite conectar todas las ciudades con todas las ciudades, usaré un **APO**, donde guardaré el tipo arista, que estará formado por origen destino y coste (el coste se calcularía por la distancia euclídea entre las ciudades)
Luego, sabiendo que en la cima del APO está la arista más barata miro si he conectado ya esas 2 islas haciendo uso del **TAD Partición.** Si ambas islas ya están conectadas, las

Orbit ¿DÍA DE CLASES infinitas?

conecto y las meto en el mismo conjunto. Este ejercicio es horroroso y largo, por dios que no caiga.

masca y fluye

4. La empresa EMASAJER S.A. tiene que unir mediante canales todas las ciudades del valle del Jerte (Cáceres). Calcula qué canales y de qué longitud deben construirse partiendo del grafo con las distancias entre las ciudades y asumiendo las siguientes premisas:

- el coste de abrir cada nuevo canal es casi prohibitivo, luego la solución final debe tener un número mínimo de canales.
- el Ministerio de Fomento nos subvenciona por Kms de canal, luego los canales deben ser de la longitud máxima posible.

```
#include"alg_grafoMA.h"
#include"listaenla.h"
#include"pilaenla.h"
#include"particion.h"
#include"matriz.h"
#include"apo.h"
#include"alg_grafo_E-S.h"
#include<iostream>
#include<math.h>
#include"alg_grafoPMC.h"
using namespace std;


template<typename T>
GrafoP<T> EMASAJERSA(GrafoP<T>&);









template<typename T>
GrafoP<T> EMASAJERSA(GrafoP<T>& G)
{
   GrafoP<T> arbolmin(G.numVert());

   //primero ponemos el grafo pasado por parámetro a coste negativo
   for(size_t i=0;i<G.numVert();i++)
```

```
    {
        for(size_t j=0;j<G.numVert();j++)
        {
            G[i][j]=-G[i][j];
        }
    }

    //aplicamos el algoritmo de Kruskall que nos devolverá el árbol de
expansión máximo
    arbolmin=Kruskall(G);


    //deshacemos el camibo en el grafo original
     for(size_t i=0;i<G.numVert();i++)
    {
        for(size_t j=0;j<G.numVert();j++)
        {
            G[i][j]=-G[i][j];
        }
    }

    return arbolmin;
}
```

6. La empresa EMASAJER S.A. tiene que unir mediante canales todas las ciudades del valle del Jerte (Cáceres), teniendo en cuenta las siguientes premisas:

- El coste de abrir cada nuevo canal es casi prohibitivo, luego la solución final debe tener un número mínimo de canales.
- El Ministerio de Fomento nos subvenciona por m^3/sg de caudal, luego el conjunto de los canales debe admitir el mayor caudal posible, pero por otra parte, el coste de abrir cada canal es proporcional a su longitud, por lo que el conjunto de los canales también debería medir lo menos posible. Así pues, la solución óptima debería combinar adecuadamente ambos factores.

Dada la matriz de distancias entre las diferentes ciudades del valle del Jerte, otra matriz con los diferentes caudales máximos admisibles entre estas ciudades teniendo en cuenta su orografía, la subvención que nos da Fomento por m^3/sg. de caudal y el coste por km. de canal, implementen un subprograma que calcule qué canales y de qué longitud y caudal deben construirse para minimizar el coste total de la red de canales.

La matriz de distancias la hemos de multiplicar por el coste por km del canal para hallar el coste real de la distancia desde i a j.
La matriz de caudales la hemos de multiplicar por la subvención para hallar el “descuento” de construir dicho canal.
Luego debemos restar ambas matrices y hacemos Kruskall/Prim y listo

```
#include"alg_grafoMA.h"
#include"listaenla.h"
#include"pilaenla.h"
#include"particion.h"
#include"matriz.h"
#include"apo.h"
#include"alg_grafo_E-S.h"
#include<iostream>
#include<math.h>
#include"alg_grafoPMC.h"
using namespace std;


template<typename T>
GrafoP<T> EMASAJERSA(const
GrafoP<T>distancias,matriz<T>caudalesmaximos,double subvencion,double
costePorKm);
```

```cpp
template<typename T>
GrafoP<T> EMASAJERSA(const
GrafoP<T>distancias,matriz<T>caudalesmaximos,double subvencion,double
costePorKm)
{
   GrafoP<T> g;

   //primero, multiplicamos la matriz de distancias por el coste por
km, teniendo así el coste total de cada canal
   for(size_t i=0;i<distancias.numVert();i++)
   {
       for(size_t j=0;j<distancias.numVert();j++)
       {
           distancias[i][j]*=costePorKm;
       }
   }

   //ahora multiplicamos la matriz de caudales por su subvencion, para
ver cuanto nos quitamos
   for(size_t i=0;i<caudalesmaximos.dimension();i++)
   {
       for(size_t j=0;j<caudalesmaximos.dimension();j++)
       {
           caudalesmaximos[i][j]*=subvencion;
       }
   }

   //posteriormente hacemos una resta de matrices, es decir, a los
costes les restamos la subvencion
   for(size_t i=0;i<distancias.numVert();i++)
   {
       for(size_t j=0;j<distancias.numVert();j++)
       {
           distancias[i][j]-=subvencion[i][j];
       }
   }

   g=Kruskall(distancias);

   return g;
}
```

Orbit ¿DÍA DE CLASES infinitas?

7. El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen N_1 y N_2 ciudades, respectivamente, de las cuales C_1 y C_2 ciudades son costeras (obviamente $C_1 \leq N_1$ y $C_2 \leq N_2$). Se dispone de las coordenadas cartesianas *(x, y)* de todas y cada una de las ciudades del archipiélago. El huracán Isadore acaba de devastar el archipiélago, con lo que todas las carreteras y puentes construidos en su día han desaparecido. En esta terrible situación se pide ayuda a la ONU, que acepta reconstruir el archipiélago (es decir volver a comunicar todas las ciudades del archipiélago) siempre que se haga al mínimo coste.

De cara a poder comparar costes de posibles reconstrucciones se asume lo siguiente:

1. El coste de construir cualquier carretera o cualquier puente es proporcional a su longitud (distancia euclídea entre las poblaciones de inicio y fin de la carretera o del puente).
2. Cualquier puente que se construya siempre será más caro que cualquier carretera que se construya.

De cara a poder calcular los costes de VIAJAR entre cualquier ciudad del archipiélago se considerará lo siguiente:

1. El coste directo de viajar, es decir de utilización de una carretera o de un puente, coincidirá con su longitud (distancia euclídea entre las poblaciones origen y destino de la carretera o del puente).

En estas condiciones, implementa un subprograma que calcule el coste mínimo de viajar entre dos ciudades de Grecoland, origen y destino, después de haberse reconstruido el archipiélago, dados los siguientes datos:

1. Lista de ciudades de Fobos representadas mediante sus coordenadas cartesianas.
2. Lista de ciudades de Deimos representadas mediante sus coordenadas cartesianas.
3. Lista de ciudades costeras de Fobos.
4. Lista de ciudades costeras de Deimos.
5. Ciudad origen del viaje.
6. Ciudad destino del viaje.

```
#include "alg_grafoMA.h"
#include "listaenla.h"
#include "pilaenla.h"
#include "particion.h"
#include "matriz.h"
#include "apo.h"
#include "alg_grafo_E-S.h"
#include <iostream>
#include <math.h>
#include "alg_grafoPMC.h"
using namespace std;


struct Ciudad
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S
masca y fluye

```
{
    size_t coordx;
    size_t coordy;
};


unsigned int
Grecoland(vector<Ciudad>&cFobos,vector<Ciudad>&cDeimos,vector<Ciudad>&c
osterasFobos,vector<Ciudad>&costerasDeimos,GrafoP<unsigned
int>::vertice origen,
GrafoP<unsigned int>::vertice destino);



int main()
{}




unsigned int
Grecoland(vector<Ciudad>&cFobos,vector<Ciudad>&cDeimos,vector<Ciudad>&c
osterasFobos,vector<Ciudad>&costerasDeimos,GrafoP<unsigned
int>::vertice origen,
GrafoP<unsigned int>::vertice destino)
{
    //montamos el supergrafo
    GrafoP<unsigned int> g(cFobos.size() + cDeimos.size());
    //coste minimo a devolver
    unsigned int costemin=GrafoP<unsigned int>::INFINITO;
    unsigned int costemaxcarretera=0;

    //variables para Dijkstra
    vector<GrafoP<unsigned int>::vertice> P;
    vector<unsigned int>A;

    //procedo a rellenar el supergrafo. Conecto las carreteras de las
primera isla y me quedo con la mas grande
    for(size_t i=0;i<cFobos.size();i++)
    {
        for(size_t j=0;j<cFobos.size();j++)
        {
```

```
            if(i!=j)
            {
                double dx=cFobos[i].coordx - cFobos[j].coordx;
                double dy=cFobos[i].coordy - cFobos[j].coordy;

                g[i][j]=sqrt(dx*dx + dy*dy);

                if(g[i][j]> costemaxcarretera)
                    costemaxcarretera=g[i][j];
            }

            else
                g[i][j]=0;
        }
    }

    size_t k=0;
    size_t l=0;
    //procedo con la segunda isla
    for(size_t i=cFobos.size();i<cFobos.size() + cDeimos.size();i++)
    {
        for(size_t j=cFobos.size();j<cFobos.size() + cDeimos.size();
j++)
        {
            if(i!=j)
            {
                double dx= cDeimos[k].coordx - cDeimos[l].coordx;
                double dy= cDeimos[k].coordy - cDeimos[l].coordy;


                g[i][j]=sqrt(dx*dx + dy*dy);

                if(g[i][j] > costemaxcarretera)
                    costemaxcarretera=g[i][j];

                l++;
            }
            k++;
            l=0;
        }
    }
```

Orbit ¿DÍA DE CLASES infinitas?

```
   //a continuación, procedo a conectar todas las ciudades costeras de
una isla con todas las ciudades costeras de la otra teniendo en cuenta
que cada puente lleva el coste
   //de la carretera más cara

   for(size_t i=0; i<costerasFobos.size();i++)
   {
       for(size_t j=0; j<costerasDeimos.size();j++)
       {
           double dx=costerasFobos[i].coordx -
costerasDeimos[j].coordx;
           double dy= costerasFobos[i].coordx -
costerasDeimos[j].coordy;

           g[i][j+cFobos.size()]=(dx*dx + dy*dy) + costemaxcarretera;
           g[j + cFobos.size()][i]=(dx*dx + dy*dy) + costemaxcarretera;
       }
   }

   GrafoP<unsigned int> arbolmin(cFobos.size() + cDeimos.size());

   arbolmin=Kruskall(g);

   //ahora cambios el coste de viajar entre islas. Coincide con su
distancia euclídea
     // el coste de viajar coincide con la distancia euclídea
   for (size_t i = 0; i < arbolmin.numVert(); ++i)
   {
       for (size_t j = i + 1; j < arbolmin.numVert(); ++j)
       {
           if (arbolmin[i][j] != GrafoP<unsigned int>::INFINITO)
           {
               bool esFobos_i = (i < cFobos.size());
               bool esFobos_j = (j < cFobos.size());
               if (esFobos_i != esFobos_j) // Uno está en Fobos y otro
en Deimos ⇒ puente
               {
                   arbolmin[i][j] -= costemaxcarretera;
                   arbolmin[j][i] -= costemaxcarretera;
               }
           }
       }
   }
```

masca y fluye

```
    //a continuación basta con aplicar Dijkstra
    A=Dijkstra(arbolmin,origen,P);

    //por último hallo el coste mínimo

    costemin=A[destino];

    return costemin;
}
```

Ejercicio Examen Grafos EDNL 2024

El archipiélago de Grecoland (Zuelandia) está formado únicamente por **tres** islas, Fobos, Deimos y **Europa**, que tienen N1, N2 y N3 ciudades, respectivamente, de las cuales C1, C2 y C3 ciudades son costeras (obviamente $C1 \leq N1$, $C2 \leq N2$ y $C3 \leq N3$). Se dispone de las coordenadas cartesianas (x, y) de todas y cada una de las ciudades del archipiélago.

El huracán Isadore acaba de devastar el archipiélago, con lo que todas las carreteras y puentes construidos en su día han desaparecido. En esta terrible situación se pide ayuda a la ONU, que acepta reconstruir el archipiélago (es decir volver a comunicar todas las ciudades del archipiélago) siempre que se haga al mínimo coste. De cara a poder comparar costes de posibles reconstrucciones se asume lo siguiente:

1. El coste de construir cualquier carretera o cualquier puente es proporcional a su longitud (distancia euclídea entre las poblaciones de inicio y fin de la carretera o del puente).
2. Cualquier puente que se construya siempre será **más barato** que cualquier carretera que se construya.

De cara a poder calcular los costes de VIAJAR entre cualquier ciudad del archipiélago se considerará lo siguiente:

1. El coste directo de viajar, es decir de utilización de una carretera o de un puente, coincidirá con su longitud (distancia euclídea entre las poblaciones origen y destino de la carretera o del puente).

En estas condiciones, implementa un subprograma que calcule el coste mínimo de viajar entre dos ciudades de Grecoland, origen y destino, después de haberse reconstruido el archipiélago, dados los siguientes datos:

1. Lista de ciudades de Fobos representadas mediante sus coordenadas cartesianas.
2. Lista de ciudades de Deimos representadas mediante sus coordenadas cartesianas.
3. Lista de ciudades de Europa representadas mediante sus coordenadas cartesianas.
4. Lista de ciudades costeras de Fobos.
5. Lista de ciudades costeras de Deimos.
6. Lista de ciudades costeras de Europa.
7. Ciudad origen y destino del viaje.

```
//Cabecera:double Grecoland(const vector<Ciudad>&ciudadesFobos,const
vector<Ciudad>&ciudadesDeimos,const vector<Ciudad>&ciudadesEuropa,const
vector<Ciudad>&costerasFobos,
//const vector<Ciudad>&costerasDeimos, const
vector<Ciudad>&costerasEuropa,GrafoP<unsigned int>::arista origen,
GrafoP<unsigned int>::arista destino);
//Precondicion: Vectores cuyo contenido sean las coordenadas
cartesianas de las ciudades/ciudades costeras, origen y destino
existentes en el dominio de la isla
//Postcondición: Devuelve el coste mínimo de ir de origen a destino
dadas las condiciones del problema
double Grecoland(const vector<Ciudad>&ciudadesFobos,const
vector<Ciudad>&ciudadesDeimos,const vector<Ciudad>&ciudadesEuropa,const
vector<Ciudad>&costerasFobos,
const vector<Ciudad>&costerasDeimos, const
vector<Ciudad>&costerasEuropa,GrafoP<double>::vertice origen,
GrafoP<double>::vertice destino)
{
   double costemin=0;
   unsigned int puentemascaro=0; //para obtener el puente más caro y
sumarselo a las carreteras
   GrafoP<double> g(ciudadesFobos.size() + ciudadesDeimos.size() +
ciudadesEuropa.size()); //supergrafo que recogerá todas las conexiones
posibles
   size_t k=0,l=0;
   GrafoP<double> arbolmin(ciudadesFobos.size() + ciudadesDeimos.size()
+ ciudadesEuropa.size());
   vector<GrafoP<double>::vertice> P;
   vector<double> A;


   //como cualquier puente debe ser más barato que cualquier carretera
que se construya, comienzo creando los puentes y me quedo con el más
caro, y ese se lo sumo
   //a las carreteras
   //conecto las costeras de Fobos con las de Deimos
   for(size_t i=0;i<costerasFobos.size();i++)
   {
       for(size_t j=0;j<costerasDeimos.size();j++)
       {
```

```
            double dx=costerasFobos[i].coordx -
costerasDeimos[j].coordx;
            double dy=costerasFobos[i].coordy -
costerasDeimos[j].coordy;


            g[i][j + costerasFobos.size()]=sqrt(dx*dx + dy*dy);
            g[j + costerasFobos.size()][i]=sqrt(dx*dx + dy*dy);

            if(g[i][j+costerasFobos.size()] > puentemascaro)
                puentemascaro=g[i][j+costerasFobos.size()];
        }
    }

    //conecto las costeras de Fobos con las de Europa
    for(size_t i=0;i<costerasFobos.size();i++)
    {
        for(size_t j=0;j<costerasEuropa.size();j++)
        {
            double dx=costerasFobos[i].coordx -
costerasEuropa[j].coordx;
            double dy=costerasFobos[i].coordy -
costerasEuropa[j].coordy;

            g[i][j+costerasFobos.size() +
costerasDeimos.size()]=sqrt(dx*dx + dy*dy);
            g[j+ costerasFobos.size() +
costerasDeimos.size()][i]=sqrt(dx*dx + dy*dy);

            if(g[i][j+costerasFobos.size() + costerasDeimos.size()] >
puentemascaro)
                puentemascaro= g[i][j+costerasFobos.size() +
costerasDeimos.size()];
        }
    }


    //conecto las costeras de Deimos con las de Europa
    for(size_t i=0;i<costerasDeimos.size();i++)
    {
        for(size_t j=0;j<costerasEuropa.size();i++)
        {
```

Orbit ¿DÍA DE CLASES infinitas?

```
            double dx=costerasDeimos[i].coordx -
costerasEuropa[j].coordx;
            double dy=costerasDeimos[i].coordy -
costerasEuropa[j].coordy;

            g[i+costerasFobos.size()][j+costerasFobos.size() +
costerasDeimos.size()]=sqrt(dx*dx + dy*dy);
            g[j+costerasFobos.size() +
costerasDeimos.size()][i+costerasFobos.size()]=sqrt(dx*dx + dy*dy);

            if(g[i+costerasFobos.size()][j+costerasFobos.size() +
costerasDeimos.size()] > puentemascaro)

puentemascaro=g[i+costerasFobos.size()][j+costerasFobos.size() +
costerasDeimos.size()];
        }
    }



    //a continuación monto carreteras. Le voy a sumar a cada carretera
el coste del puente más caro.
    //empiezo por Fobos
    for(size_t i=0;i<ciudadesFobos.size();i++)
    {
        for(size_t j=0;j<ciudadesFobos.size();j++)
        {
            if(i!=j)
            {
                double dx=ciudadesFobos[i].coordx -
ciudadesFobos[j].coordx;
                double dy=ciudadesFobos[i].coordy -
ciudadesFobos[j].coordy;

                g[i][j]=sqrt(dx*dx + dy*dy) + puentemascaro;
            }

            else
                g[i][j]=0;
        }
    }
```

masca y fluye

```
   //continuamos por Deimos, me apoyo en las variables auxiliares k y l
   for(size_t i=ciudadesFobos.size();i<ciudadesFobos.size() +
ciudadesDeimos.size();i++)
   {
       for(size_t j=ciudadesFobos.size();j<ciudadesFobos.size() +
ciudadesDeimos.size();j++)
       {
           if(i!=j)
           {
                double dx=ciudadesDeimos[k].coordx -
ciudadesDeimos[l].coordx;
               double dy=ciudadesDeimos[k].coordy -
ciudadesDeimos[l].coordy;
               l++;

               g[i][j]=sqrt(dx*dx + dy*dy) + puentemascaro;
           }

           else
               g[i][j]=0;
       }
       k++;
       l=0;
   }

   k=0,l=0;

   //continúo por Europa con el mismo funcionamiento que en Deimos

   for(size_t i=ciudadesFobos.size() +
ciudadesDeimos.size();i<ciudadesFobos.size() + ciudadesDeimos.size() +
ciudadesEuropa.size();i++)
   {
       for(size_t j=ciudadesFobos.size() + ciudadesDeimos.size();
j<ciudadesFobos.size() + ciudadesDeimos.size() +
ciudadesEuropa.size();j++)
       {
           if(i!=j)
           {
               double dx=ciudadesDeimos[k].coordx -
ciudadesDeimos[l].coordx;
               double dy=ciudadesDeimos[k].coordy -
ciudadesDeimos[l].coordy;
```

```
                l++;

                g[i][j]=sqrt(dx*dx + dy*dy) + puentemascaro;
            }

            else
                g[i][j]=0;

        }
            l=0;
            k++;
    }

    //a continuación aplico Kruskall y al grafo resultante, a cada
carretera le quito el valor del puente más caro

    arbolmin=Kruskall(g);

    //a continuación, a cada ciudad de cada isla le resto el coste del
puente más caro

    for(size_t i=0;i<arbolmin.numVert();i++)
    {
        for(size_t j=0;j<arbolmin.numVert();j++)
        {
            if(i!=j) //para no tener ciudades con coste negativo
                arbolmin[i][j]-=puentemascaro;

        }
    }

    //a continuación, aplico Dijkstra y listo
    A=Dijkstra(arbolmin,origen,P);

    costemin=A[destino];
    return costemin;
}
```

Orbit
¿DÍA DE CLASES infinitas?


masca y fluye

Ejercicios Exámenes EDNL

Repartida de bebidas (Junio 2022)

Enunciado: *Un repartidor de una empresa de distribución de bebidas tiene que visitar a todos sus clientes cada dias. Pero, al comentar su jornada de trabajo, no conoce que cantidad de bebidas tiene que servir cada cliente, por lo que no puede planificar una ruta optima para visitarlos a todos. Por tanto, nuestro repartidor decide llevar a cabo la siguiente estrategia:*

- *El camión parte del almacen con la máxima carga permitida rumbo a su cliente más próximo.*
- *El repartidor descarga las cajas de bebidas que le pide el cliente, si no tiene suficientes cajas en el camión, le entrega todas las que tiene. Este cliente terminara de ser servido en algun otro momento a lo largo del dia cuando la estratefia de reparto vuelva a llevar al repartidor hasta él.*
- *Después de servir a un cliente:*
  - *Si quedan bebidas en el camión, el repartidor consulta su sistema de navegacion basado en el GPS para conocer la ruta que le lleva hasta su cliente más proximo pendiente de ser servido.*
  - *Si no quedan bebidas en el camión, vuelve al almacén por el camino más corto y otra vez carga el camión completamente.*
- *Despues de cargar el camión, el repartidor consulta su sistema de navegacion y se va por el camino mas corto a visitar al cliente pendiente de ser servido más proximo.*

Implementa un subprograma que calcule y devuelva, la distancia total recorrida en un dia por nuestro repartidor a partir de lo siguiente:

- *Grafo representado mediante la matriz de costes con las distancias de los caminos directos entre los clientes y entre ellos y la central.*
- *Capacidad máxima del camión (cantidad de cajas de bebidas).*
- *Asumiremos que existe una funcion int Pedido() que devuelve el numero de cajas que quedan por servir al cliente en el que se encuentra el repartidor.*

NOTA: Es absolutamente necesario definir todos los tipos de datos implicados en la resolucion del problema, así como los prototipos de las operaciones utilizadas de los TADS conocidos y tambien los prototipos de los algoritmos de grafo utilizados de los estudiados en las asignaturas.

→ Nos piden la distancia recorrida en un día por el repartidor. Datos:

- Grafo (matriz de costes)
- Capacidad máxima del camión.
- Función pedidos() que devuelve el nº de cajas.

→ Lo que haría es aplicar Floyd por 2 motivos:

1.- Si estoy en un vértice cualquiera del grafo, debo visitar el siguiente más cercano al mismo (que no haya sido visitado).

2.- Obtengo los costes mínimos de ir de cualquier nodo a cualquier nodo (incluida desde y hacia la base).

Posteriormente, crearía un vector de booleanos en el que indico si un nodo ya ha sido totalmente abastecido.

Orbit ¿DÍA DE CLASES infinitas?

```cpp
//Cabecera:double distanciaRepartidor(const GrafoP<unsigned int>&g,size_t capacidadcamion)
//Precondicion: grafo dado con matriz de costes y capacidad de camion
//Postcondición: Devuelve la distancia total mínima posible recorrida por el repartidor
double distanciaRepartidor(const GrafoP<unsigned int>&g,size_t capacidadcamion, GrafoP<unsigned int>::vertice origen)
{

    double dist=0.;
    vector<bool> visitados(g.numVert(),false); //vector que indicará si un nodo ya está totalmente abastecido
    matriz<GrafoP<unsigned int>::vertice> P1; //matriz de vértices para Floyd
    matriz<unsigned int> A1; //matriz de costes mínimos para Floyd
    GrafoP<unsigned int>::vertice copiaorigen=origen; //para tener guardado siempre el origen inicial

    A1=Floyd(g,P1);

    //ya tenemos en A1 los costes mínimos, ahora procedemos a ir visitando las ciudades más cercanas que no han sido abastecidas
    bool flag=false; //condición de parada del bucle

    while(!flag)
    {
        unsigned int min=__INT32_MAX__;
        size_t verticemin=0;
        //procedemos a visitar a las ciudades. Buscamos la ciudad más cercana a partir del origen
        for(size_t i=0;i<g.numVert();i++)
        {
            if(i!=origen && !visitados[i]) //si hemos encontrado una ciudad no visitada cuyo coste sea el más bajo...
            {
                if(A1[origen][i]<min)
                {
                    min=A1[origen][i];
                    verticemin=i;
                }
            }
        }

        //ya tenemos en vertice la ciudad más cercana al origen NO visitada, sumamos el coste
        dist+=A1[origen][verticemin];
        //comprobamos cuantas cajas necesita el cliente

        if(Pedido(verticemin)<capacidadcamion)
        {
            capacidadcamion-=Pedido(verticemin);
            //marcamos verticemin como visitado
            visitados[verticemin]=true;

            //ahora el origen es verticemin
            origen=verticemin;
        }


        else
        {
            if(Pedido(verticemin)==capacidadcamion) //podemos abastecerle, pero hemos de regresar a la base
            {
                capacidadcamion-=Pedido(verticemin);
                //marcamos verticemin como visitado
                visitados[verticemin]=true;

                //el origen seguirá siendo el vértice inicial, ya que hemos de volver a la base, sumamos el coste de volver a la misma
                dist+=A1[verticemin][copiaorigen];

                origen=copiaorigen;
            }

            else //la ciudad necesita más productos de los que posee el camión
            {
                capacidadcamion-=Pedido(verticemin);

                //no podemos marcar verticemin como visitado

                dist+=A1[verticemin][copiaorigen];
                origen=copiaorigen;
            }
        }



        //procedo a recorrer el vector para comprobar si están todos las ciudades visitadas
        if(std::count(visitados.begin(), visitados.end(), true) == visitados.size())
            flag=true;
    }
```


masca y fluye

```
        }



        //procedo a recorrer el vector para comprobar si están todos las ciudades visitadas
        if(std::count(visitados.begin(), visitados.end(), true) == visitados.size())
            flag=true;
    }




    return dist;
}
```

Laberinto 3D (Septiembre 2022)

Enunciado: *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 seis direcciones (arriba, abajo, derecha, izquierda, adelante, atrás). Por otra parte, entre algunas de las casillas hay una pared que impide moverse entre las dos casillas que separa dicha pared (en caso contrario no sería un verdadero laberinto).*

Implementa un subprograma que dados:

- *N (dimensión del laberinto),*
- *la lista de paredes del laberinto,*
- *la casilla de entrada, y*
- *la casilla de salida,*

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

→ Hemos de hallar el camino más corto para ir desde la entrada a la salida

Los problemas de laberinto se resuelven en preparar el grafo (matriz de costes) luego es hacer un Dijkstra y fuera.

Funciones a implementar:

```
size_t Casilla_a_Nodo (Casilla &c)   // convertir una Casilla a un nodo
Casilla Nodo_Casilla (size_t nodo)   // Convertir un Nodo a una Casilla
Bool Casillas_Adyacentes(Casilla &c1, Casilla &c2);  // Saber si 2 Casillas son adyacentes.
```

Lista de paredes ⇒ vector< Pared > v

struct Pared {

Casilla c1, c2; }.

Orbit ¿DÍA DE CLASES infinitas?

```cpp
struct Casilla
{
    size_t x,y,z; //coordenada de una casilla
};

struct Pared
{
    Casilla c1,c2;
};


GrafoP<unsigned int>::vertice Casilla_a_Nodo(const Casilla& c)
{
    return ((c.x * N*N) + (c.y*N) + c.z);
}

Casilla Nodo_a_Casilla(const GrafoP<unsigned int>::vertice v)
{
    Casilla c;
    c.x=v/(N*N);
    c.y=((v%(N*N))/N);
    c.z=v%N;

    return c;
}
```

```cpp
bool Adyacencia(const Casilla &c1, const Casilla &c2)
{
    int dx = abs(int(c1.x) - int(c2.x));
    int dy = abs(int(c1.y) - int(c2.y));
    int dz = abs(int(c1.z) - int(c2.z));

    // Son adyacentes si solo difieren en una coordenada y esa diferencia es 1
    return (dx + dy + dz == 1);
}


GrafoP<unsigned int>::tCamino Laberinto3D(size_t dimension, vector<Pared> paredes, const Casilla& entrada, const Casilla&salida);
```

```cpp
//Cabecera:GrafoP<unsigned int>::tCamino Laberinto3D(size_t dimension, vector<Casilla> paredes, const Casilla& entrada, const Casilla&salida);
//Precondicion: Dimension >=1 entrada y salidas casillas del laberinto
//Postcondicion: Devuelve el camino más corto para ir desde la entrada a la salida y su longitud
GrafoP<unsigned int>::tCamino Laberinto3D(size_t dimension, vector<Pared> paredes, const Casilla& entrada, const Casilla&salida)
{
    GrafoP<unsigned int>::tCamino cam;
    GrafoP<unsigned int> g(dimension*dimension*dimension);
    vector<GrafoP<unsigned int>::vertice> Pl;
    vector<unsigned int> Al;


    cout<<"Grafo antes de conectar: "<<g<<endl;


    for(size_t i=0;i<dimension * dimension *dimension;i++)
    {
        for(size_t j=0;j<dimension *dimension *dimension;j++)
        {
            Casilla c1,c2;
            c1=Nodo_a_Casilla(i);
            c2=Nodo_a_Casilla(j);

            if(Adyacencia(c1,c2))
                g[i][j]=1;
        }
    }
```

masca y fluye

```
//ahora procedo a poner las paredes en el laberinto
for(size_t i=0;i<paredes.size();i++)
{
        Casilla p1,p2;
        p1= paredes[i].c1;
        p2=paredes[i].c2;

        GrafoP<unsigned int>::vertice v1,v2;
        v1=Casilla_a_Nodo(p1);
        v2=Casilla_a_Nodo(p2);

        g[v1][v2]=GrafoP<unsigned int>::INFINITO;
        g[v2][v1]=GrafoP<unsigned int>::INFINITO;
}



cout<<"Grafo después de inicializar: "<<g<<endl;

GrafoP<unsigned int>::vertice  origen = Casilla_a_Nodo(entrada);
GrafoP<unsigned int>::vertice  destino = Casilla_a_Nodo(salida);


A1=Dijkstra(g,origen,P1);
cam=camino<unsigned int>(origen,destino,P1);

return cam;
```

Fauno (Junio 2023)

Enunciado: *Modelizaremos Narnia como una matriz de NxM casillas. Los movimientos del fauno se modernizarán con los movimientos de un caballo del ajedrez. Dicho de otra forma, cada movimiento del fauno debe ser un movimiento de caballo de ajedrez.*

A Narnia se llega a través de su entrada, casilla (0,0), y se marcha uno a través de una única salida, en la casilla (N-1, M-1). Sería un problema bastante fácil, pero Narnia es un país lleno de peligros, en particular si eres un fauno.

Para empezar, los lugareños han puesto dentro de Narnia una serie de trampas en determinadas casillas, de forma que si pasas por ellas mueres.

Pero no contentos con eso, los habitantes de Narnia han contratado Minos caballeros, que se colocan también en casillas estratégicas. En este caso, el fauno no muerte si cae en una de ellas, pero su muere en caso de caer en alguna de las casillas que rodean al caballero, entre 3 y 8, dependiendo de su posición, ya que su espada tiene longitud 1.

El problema nos pide dos cosas, la primera saber si el fauno puede hacer de forma segura el camino entre la entrada y la salida de Narnia, y, en caso afirmativo, cual sería el número mínimo de saltos necesarios para conseguirlo.

Dado los siguientes parámetros:

- *Dimensiones de Narnia N y M.*
- *Relación de casillas trampa.*
- *Relación de casillas ocupadas por caballeros.*

Implementa una función que calcula y devuelva si el fauno podrá llegar o no a la salida y, en caso afirmativo, el número mínimo de saltos necesarios para conseguirlo.

Este problema es similar a un laberinto con algunas variaciones:

función Casilla_Adyacentes(Casilla fC1, Casilla fC2). En este caso las adyacentes a una casilla son aquellas en las que te desplazas en forma de L (caballo).

Las funciones Casilla_A_Nodo y Nodo_A_Casilla son iguales que el que se explicó en clase.

Orbit ¿DÍA DE CLASES infinitas?

```
struct Casilla
{
    int x, y;
};

typedef Casilla trampa;

GrafoP<unsigned int>::vertice Casilla_a_Nodo(const Casilla &c)
{
    return c.x * N + c.y;
}

Casilla Nodo_a_Casilla(const GrafoP<unsigned int>::vertice &nodo)
{
    Casilla c;
    c.x = nodo / N;
    c.y = nodo % N;

    return c;
}
```

```
// casillas adyacentes teniendo en cuenta que únicamente se puede mover como el caballo de ajedrez
bool CasillasAdyacentes(const Casilla &c1, const Casilla &c2)
{

    // dos casillas son adyacentes en este caso cuando difieren en +- 1 columna y +- 2 filas o al contrario
    bool dev = false;

    int resfilas = abs(c1.x - c2.x);
    int rescol = abs(c1.y - c2.y);

    if ((resfilas == 2 && rescol == 1) || (resfilas==1 && rescol==2))
        dev = true;

    return dev;
}

pair<GrafoP<unsigned int>::tCamino, bool> Fauno(size_t dimesionN, size_t dimensionM, const vector<trampa> &casillastrampa, const vector<Casilla> &caballeros);
```

int rescol

Generar resumen de Copilot

Ahora, el código de la función:

SIN AZÚCAR MENTA Orbit WRIGLEY'S

masca y fluye

```
// Cabecera:pair<GrafoP<unsigned int>::tCamino,bool> Fauno(size_t
dimesionN,size_t dimensionM,const vector<trampa>& casillastrampa, const
vector<Casilla>& caballeros);
// Precondición: N y M >=4 para que se pueda mover en L el caballo
// Postcondicion: Devuelve el camino, si existe y un booleano indicando
la existencia o no de camino
pair<GrafoP<unsigned int>::tCamino, bool> Fauno(size_t dimesionN,
size_t dimensionM, const vector<trampa> &casillastrampa, const
vector<Casilla> &caballeros)
{
   GrafoP<unsigned int> g(dimesionN * dimensionM);
   GrafoP<unsigned int>::vertice origen = 0; // 0*N + 0= 0
   vector<unsigned int> A1;
   vector<GrafoP<unsigned int>::vertice> P1;
   GrafoP<unsigned int>::tCamino cam;

   Casilla des;
   des.x=dimensionM -1;
   des.y=dimensionM-1;

   GrafoP<unsigned int>::vertice destino=Casilla_a_Nodo(des);


   // procedo a conectar todo el grafo (segun las casillas adyacentes)
y luego ya voy quitando aristas segun el enunciado del problema
   for (size_t i = 0; i < g.numVert(); i++)
   {
       for (size_t j = 0; j < g.numVert(); j++)
       {
           Casilla c1, c2;
           c1 = Nodo_a_Casilla(i);
           c2 = Nodo_a_Casilla(j);

           if (CasillasAdyacentes(c1, c2))
               g[i][j] = 1;
       }
   }

       cout<<"Grafo antes de las comprobaciones: "<<endl;
       cout<<g<<endl;
```

```
   // a continuación, procedo a poner las casillas trampas como
infinito

   for (size_t i = 0; i < casillastrampa.size(); i++)
   {
       GrafoP<unsigned int>::vertice v =
Casilla_a_Nodo(casillastrampa[i]);

       // pongo la fila y la columna a infinito
       for (size_t j = 0; j < dimensionM*dimensionM; j++)
       {
           g[v][j] = GrafoP<unsigned int>::INFINITO;
           g[j][v] = GrafoP<unsigned int>::INFINITO;
       }
   }

   // ahora procedo a poner todas las casillas alrededor del caballero
como infinito
   for (size_t i = 0; i < caballeros.size(); i++)
   {
       Casilla cab = caballeros[i];

       if (cab.x >= 1) // fila del caballero mayor o igual a 1, arriba
no se puede pisar
       {
           if (cab.y == 0 || cab.y <= dimensionM -1) // solo tapa una
casilla arriba
           {
               Casilla cabarriba;
               cabarriba.x = cab.x - 1;
               cabarriba.y = cab.y;
               Casilla cabarribadizq;

               if(cab.y==0)
               {
                   cabarribadizq.x=cab.x -1;
                   cabarribadizq.y=cab.y +1;
               }

               else
               {
                   cabarribadizq.x=cab.x -1;
                   cabarribadizq.y=cab.y -1;
```

Orbit ¿DÍA DE CLASES infinitas?

```
                }



                GrafoP<unsigned int>::vertice ver =
Casilla_a_Nodo(cabarriba);
                GrafoP<unsigned int>::vertice ver2 =
Casilla_a_Nodo(cabarribadizq);



                // pongo la fila y la columna a infinito
                for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[ver][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][ver] = GrafoP<unsigned int>::INFINITO;
                }
                  for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[ver2][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][ver2] = GrafoP<unsigned int>::INFINITO;
                }
            }

            else // tapa 3 casillas arriba
            {
                Casilla c1, c2, c3;
                c1.x = cab.x - 1;
                c1.y = cab.y;

                c2.x = cab.x - 1;
                c2.y = cab.y - 1;

                c3.x = cab.x - 1;
                c3.y = cab.y + 1;

                // ahora he de poner los nodos correspondientes a estas
3 casillas, su fila y columna a infinito
                GrafoP<unsigned int>::vertice v1, v2, v3;
                v1 = Casilla_a_Nodo(c1);
                v2 = Casilla_a_Nodo(c2);
                v3 = Casilla_a_Nodo(c3);
```

SIN AZÚCAR MENTA Orbit WRIGLEY'S
masca y fluye

```
                for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[v1][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][v1] = GrafoP<unsigned int>::INFINITO;
                }
                for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[v2][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][v2] = GrafoP<unsigned int>::INFINITO;
                }

                for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[v3][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][v3] = GrafoP<unsigned int>::INFINITO;
                }
            }
        }

        if (cab.x <= dimesionN - 1) // fila menor o igual a la dimensión
menos 1, abajo no se puede pisar
        {

            if (cab.y == 0 || cab.y <= dimensionM -1) // solo tapa 1
abajo
            {

                Casilla cababajo;
                cababajo.x = cab.x + 1;
                cababajo.y = cab.y;

                GrafoP<unsigned int>::vertice ver =
Casilla_a_Nodo(cababajo);

                for (size_t j = 0; j < dimensionM*dimensionM; j++)
                {
                    g[ver][j] = GrafoP<unsigned int>::INFINITO;
                    g[j][ver] = GrafoP<unsigned int>::INFINITO;
                }
            }
        }

        else // tapa 3 abajo
```

```
        {

            Casilla c1, c2, c3;
            c1.x = cab.x + 1;
            c1.y = cab.y;

            c2.x = cab.x + 1;
            c2.y = cab.y - 1;

            c3.x = cab.x + 1;
            c3.y = cab.y + 1;

            // ahora he de poner los nodos correspondientes a estas 3
casillas, su fila y columna a infinito
            GrafoP<unsigned int>::vertice v1, v2, v3;
            v1 = Casilla_a_Nodo(c1);
            v2 = Casilla_a_Nodo(c2);
            v3 = Casilla_a_Nodo(c3);

            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[v1][j] = GrafoP<unsigned int>::INFINITO;
                g[j][v1] = GrafoP<unsigned int>::INFINITO;
            }
            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[v2][j] = GrafoP<unsigned int>::INFINITO;
                g[j][v2] = GrafoP<unsigned int>::INFINITO;
            }

            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[v3][j] = GrafoP<unsigned int>::INFINITO;
                g[j][v3] = GrafoP<unsigned int>::INFINITO;
            }
        }

        if (cab.y >= 1 && cab.y <= dimensionM - 1) // tapa a izquierda y
derecha
        {
            Casilla c1, c2;
            c1.x = cab.x;
            c1.y = cab.y - 1;
```

Orbit ¿DÍA DE CLASES infinitas?

```
            c2.x = cab.x;
            c1.y = cab.y + 1;

            GrafoP<unsigned int>::vertice v1, v2;
            v1 = Casilla_a_Nodo(c1);
            v2 = Casilla_a_Nodo(c2);

            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[v1][j] = GrafoP<unsigned int>::INFINITO;
                g[j][v1] = GrafoP<unsigned int>::INFINITO;
            }
            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[v2][j] = GrafoP<unsigned int>::INFINITO;
                g[j][v2] = GrafoP<unsigned int>::INFINITO;
            }
        }

        else if (cab.y == dimensionM) // solo tapa a izquierda
        {
            Casilla izquierda;
            izquierda.x = cab.x;
            izquierda.y = cab.y - 1;

            GrafoP<unsigned int>::vertice ver =
Casilla_a_Nodo(izquierda);

            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[ver][j] = GrafoP<unsigned int>::INFINITO;
                g[j][ver] = GrafoP<unsigned int>::INFINITO;
            }
        }

        else // solo tapa a derecha
        {
            Casilla derecha;
            derecha.x = cab.x;
            derecha.y = cab.y + 1;

            GrafoP<unsigned int>::vertice ver = Casilla_a_Nodo(derecha);
```

masca y fluye

```
            for (size_t j = 0; j < dimensionM*dimensionM; j++)
            {
                g[ver][j] = GrafoP<unsigned int>::INFINITO;
                g[j][ver] = GrafoP<unsigned int>::INFINITO;
            }
        }
    }

    cout<<"Grafo después de las comprobaciones: "<<endl;
    cout<<g<<endl;

    A1=Dijkstra(g,origen,P1);

    if(A1[destino]==GrafoP<unsigned int>::INFINITO)
        return{cam,false};

    cam=camino<unsigned int>(origen,destino,P1);
    return{cam,true};
}
```

13. El archipiélago de las Huríes acaba de ser devastado por un maremoto de dimensiones desconocidas hasta la fecha. La primera consecuencia ha sido que todos y cada uno de los puentes que unían las diferentes ciudades de las tres islas han sido destruidos. En misión de urgencia las Naciones Unidas han decidido construir el mínimo número de puentes que permitan unir las tres islas. Asumiendo que el coste de construcción de los puentes implicados los pagará la ONU, por lo que se considera irrelevante, nuestro problema es decidir qué puentes deben construirse. Las tres islas de las Huríes tienen respectivamente N_1, N_2 y N_3 ciudades, de las cuales C_1, C_2 y C_3 son costeras (obviamente $C_1 \leq N_1$, $C_2 \leq N_2$ y $C_3 \leq N_3$) . Nuestro problema es elegir los puentes a construir entre todos los posibles. Por tanto, escogeremos aquellos puentes que minimicen el coste global de viajar entre todas las ciudades de las tres islas, teniendo en cuenta las siguientes premisas:

1. Se asume que el coste viajar entre las ciudades que unan los puentes es 0.
2. La ONU subvencionará únicamente el número mínimo de puentes necesario para comunicar las tres islas.
3. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipélago. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

Dadas las matrices de costes directos de las tres islas y las listas de ciudades costeras del archipiélago, implementad un subprograma que calcule los puentes a construir en las condiciones anteriormente descritas.

⇒ Este ejercicio es de la practica 7 (problema 13). Primero debemos montar el supergrafo y posteriormente por cada 2 puentes, hacer Floyd y verificar si se minimiza el coste.

Orbit ¿DÍA DE CLASES infinitas?

masca y fluye

```cpp
#include "grafoPMC.h"
#include "alg_grafo_E-S.h"
#include "alg_grafoPMC.h"
#include <iostream>
#include "listaenla.h"
using namespace std;

matriz<unsigned int> Huries(const GrafoP<unsigned int> &isla1,
GrafoP<unsigned int> &isla2, GrafoP<unsigned int> &isla3, const
vector<size_t> &costeras1, const vector<size_t> &costeras2, const
vector<size_t> &costeras3);

int main()
{
   GrafoP<unsigned int> g1("grafo.txt");
   GrafoP<unsigned int> g2("grafo2.txt");
   GrafoP<unsigned int> g3("grafo3.txt");

   matriz<unsigned int> m;

   vector<size_t> costeras1={1,3};
   vector<size_t> costeras2={0,2};
   vector<size_t> costeras3={2,4};


   m=Huries(g1,g2,g3,costeras1,costeras2,costeras3);


   cout<<"La matriz resultante es: "<<m<<endl;

   return 0;
}

// Cabecera:matriz<unsigned int> Huries(const GrafoP<unsigned
int>&isla1,GrafoP<unsigned int>&isla2,GrafoP<unsigned int>&isla3,const
vector<ciudadesCosteras>&costeras);
// Precondicion: El vector no puede estar vacío
// Postcondición: Devuelve los costes mínimos de viajar entre
cualesquiera de las 3 ciudades dada la relación de las ciudades
costeras. Se colocan los puentes que mejor convenga
matriz<unsigned int> Huries(const GrafoP<unsigned int> &isla1,
GrafoP<unsigned int> &isla2, GrafoP<unsigned int> &isla3, const
```

```
vector<size_t> &costeras1, const vector<size_t> &costeras2, const
vector<size_t> &costeras3)
{
   GrafoP<unsigned int> supergrafo(isla1.numVert() + isla2.numVert() +
isla3.numVert());
   unsigned int costetotal = -10;
   matriz<GrafoP<unsigned int>::vertice> P1, P2;

   // procedemos primero, a montar el supergrafo
   // monto primera isla
   for (size_t i = 0; i < isla1.numVert(); i++)
   {
       for (size_t j = 0; j < isla1.numVert(); j++)
       {
           supergrafo[i][j] = isla1[i][j];
       }
   }

   // monto segunda isla
   size_t k = 0;
   size_t l = 0;
   for (size_t i = isla1.numVert(); i < isla1.numVert() +
isla2.numVert(); i++)
   {
       for (size_t j = isla1.numVert(); j < isla1.numVert() +
isla2.numVert(); j++)
       {
           supergrafo[i][j] = isla2[k][l];
           l++;
       }
       k++;
       l = 0;
   }

   // monto última isla
   k = 0;
   l = 0;
   for (size_t i = isla1.numVert() + isla2.numVert(); i <
isla1.numVert() + isla2.numVert() + isla3.numVert(); i++)
   {

       for (size_t j = isla1.numVert() + isla2.numVert(); j <
isla1.numVert() + isla2.numVert() + isla3.numVert(); j++)
```

```
        {
            supergrafo[i][j] = isla3[k][l];
            l++;
        }
        l = 0;
        k++;
    }

    // vamos a probar los puentes entre las ciudades costeras

    // voy a probar cada costera de la isla1 con las costeras de las
otras islas
    unsigned int mejorCosteTotal = __INT16_MAX__;
    matriz<unsigned int> mejorDistancias;
    size_t offset1 = 0;
    size_t offset2 = isla1.numVert();
    size_t offset3 = isla1.numVert() + isla2.numVert();


    //preguntar a Juanfran/Jose antonio
    for (size_t i1 : costeras1)
    {
        for (size_t i2 : costeras2)
        {
            for (size_t i3 : costeras3)
            {
                // Casos:
                // 1. puente entre isla1-isla2 y isla1-isla3
                {
                    GrafoP<unsigned int> g = supergrafo;
                    g[offset1 + i1][offset2 + i2] = 0;
                    g[offset2 + i2][offset1 + i1] = 0;
                    g[offset1 + i1][offset3 + i3] = 0;
                    g[offset3 + i3][offset1 + i1] = 0;

                    auto dist = Floyd(g, P1);
                    unsigned int coste = 0;
                    for (size_t i = 0; i < dist.dimension(); ++i)
                        for (size_t j = 0; j < dist.dimension(); ++j)
                            if (i != j)
                                coste += dist[i][j];

                    if (coste < mejorCosteTotal)
```

```
                    if (coste < mejorCosteTotal)
                    {
                        mejorCosteTotal = coste;
                        mejorDistancias = dist;
                    }
                }
            }
        }
    }

    return mejorDistancias;
}
```