Practica-7-EDNL-Resuelta.pdf

QuesoViejo_

Estructuras de Datos no Lineales

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

Estructuras de Datos no Lineales

Práctica 7

Problemas de grafos II

TRABAJO PREVIO

Antes de asistir a la sesión de prácticas es obligatorio:

1. Imprimir copia de este enunciado.
2. Lectura profunda del mismo.
3. Reflexión sobre el contenido de la práctica y generación de la lista de dudas asociada a dicha práctica y a los problemas que la componen.
4. **Esbozo serio de solución** de los problemas en papel (al menos de los que se hayan entendido).

PASOS A SEGUIR

1. Para cada uno de los problemas escribir un módulo (de nombre por ejemplo `ejercicioN.cpp`) que contenga las funciones requeridas en el enunciado, para lo cual se hará uso de las clases y algoritmos de grafos proporcionados.
2. Escribir un programa de prueba de la solución propuesta para el problema, donde se realicen las llamadas a las funciones correspondientes definidas en el paso anterior, comprobando el resultado de salida para una batería suficientemente amplia de casos de prueba. Esto se puede hacer de dos maneras: Incluyendo la función `main()` en el fichero `ejercicioN.cpp` del paso anterior; o bien, creando un nuevo fichero `.cpp` para la función `main()`, que se compilará por separado y se enlazará con este `ejercicioN.cpp` anterior.

PROBLEMAS

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.

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.

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.

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.

Nota Importante:

A partir del problema 5 (el viajero alérgico), empiezan a aparecer en los enunciados el uso de diferentes medios de transporte a la hora de realizar un viaje. En nuestros problemas (tanto en prácticas como en exámenes) asumiremos que

a) Definición de trasbordo : En el contexto de los problemas de la asignatura, consideraremos trasbordo el cambio de medio de transporte**.**

b) Trasbordos libres y gratuitos por defecto: Si el enunciado del problema no indica lo contrario los trasbordos en nuestros problemas son libres y gratuitos**.**

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.

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.

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.

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.

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.

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.

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 !!!

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.

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

¿DÍA DE CLASES infinitas?

WRIGLEY'S Orbit

masca y fluye

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.

```
/*-------------------------------------------------------------*/
/* alg_grafoPMC.h                                              */
/*                                                             */
/* Algoritmos para grafos ponderados representados             */
/* mediante matriz de costes (clase GrafoP<T>).                */
/*                                                             */
/*-------------------------------------------------------------*/
/*
Funciones:

   template <typename tCoste> tCoste suma(tCoste x, tCoste y);
      Suma de costes. Devuelve GrafoP<tCoste>::INFINITO si alguno de los
      dos parámetros vale GrafoP<tCoste>::INFINITO.

   template <typename tCoste>
   vector<tCoste> Dijkstra(const GrafoP<tCoste>& G,
                           typename GrafoP<tCoste>::vertice origen,
                           vector<typename GrafoP<tCoste>::vertice>& P);
      Calcula los caminos de coste mínimo entre origen y todos los
      vértices del grafo G. Devuelve un vector de tamaño G.numVert()
      con estos costes mínimos y un vector P, también de tamaño
      G.numVert(), tal que P[i] es el último vértice del camino de
      coste mínimo de origen a i.

   template <typename tCoste> typename GrafoP<tCoste>::tCamino
   camino(typename GrafoP<tCoste>::vertice orig,
          typename GrafoP<tCoste>::vertice i,
          const vector<typename GrafoP<tCoste>::vertice>& P);
      Devuelve el camino de coste mínimo entre los vértices orig e i
      a partir de un vector P obtenido mediante la función Dijkstra().

   template <typename tCoste>
   matriz<tCoste> Floyd(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 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.

   template <typename tCoste> typename GrafoP<tCoste>::tCamino
   camino(typename GrafoP<tCoste>::vertice i,
          typename GrafoP<tCoste>::vertice j,
          const matriz<typename GrafoP<tCoste>::vertice>& P);
      Devuelve el camino de coste mínimo desde i hasta j a partir
      de una matriz P obtenida mediante la función Floyd().

   template <typename tCoste>
   GrafoP<tCoste> Prim(const GrafoP<tCoste>& G)
      Devuelve un árbol generador de coste mínimo
      de un grafo no dirigido ponderado y conexo G.

   template <typename tCoste>
   GrafoP<tCoste> Kruskall(const GrafoP<tCoste>& G)
      Devuelve un árbol generador de coste mínimo
      de un grafo no dirigido ponderado y conexo G.

*/
#ifndef ALG_GRAFO_P_H
#define ALG_GRAFO_P_H

#include <cassert>
#include "grafoPMC.h"       // grafo ponderado
#include <vector>           // para Dijkstra
#include "matriz.h"         // para Floyd
#include "apo.h"            // para Prim y Kruskall
#include "particion.h"      // para Kruskall

/*-------------------------------------------------------------------------*/
/* Caminos de coste mínimo                                                 */
/*-------------------------------------------------------------------------*/
// Suma de costes (Dijkstra y Floyd)
template <typename tCoste> tCoste suma(tCoste x, tCoste y)
{
```

```
   const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
   if (x == INFINITO || y == INFINITO)
      return INFINITO;
   else
      return x + y;
}

template <typename tCoste>
vector<tCoste> Dijkstra(const GrafoP<tCoste>& G,
                        typename GrafoP<tCoste>::vertice origen,
                        vector<typename GrafoP<tCoste>::vertice>& P)
// Calcula los caminos de coste mínimo entre origen y todos los
// vértices del grafo G. En el vector D de tamaño G.numVert()
// devuelve estos costes mínimos y P es un vector de tamaño
// G.numVert() tal que P[i] es el último vértice del camino
// de origen a i.
{
   typedef typename GrafoP<tCoste>::vertice vertice;
   vertice v, w;
   const size_t n = G.numVert();
   vector<bool> S(n, false);                  // Conjunto de vértices vacío.
   vector<tCoste> D;                          // Costes mínimos desde origen.

   // Iniciar D y P con caminos directos desde el vértice origen.
   D = G[origen];
   D[origen] = 0;                             // Coste origen-origen es 0.
   P = vector<vertice>(n, origen);

   // Calcular caminos de coste mínimo hasta cada vértice.
   S[origen] = true;                          // Incluir vértice origen en S.
   for (size_t i = 1; i <= n-2; i++) {
      // Seleccionar vértice w no incluido en S
      // con menor coste desde origen.
      tCoste costeMin = GrafoP<tCoste>::INFINITO;
      for (v = 0; v < n; v++)
         if (!S[v] && D[v] <= costeMin) {
            costeMin = D[v];
            w = v;
         }
      S[w] = true;                          // Incluir vértice w en S.
      // Recalcular coste hasta cada v no incluido en S a través de w.
      for (v = 0; v < n; v++)
         if (!S[v]) {
            tCoste Owv = suma(D[w], G[w][v]);
            if (Owv < D[v]) {
               D[v] = Owv;
               P[v] = w;
            }
         }
   }
   return D;
}

//Práctica 6, Ejercicio 1

template <typename tCoste>
vector<tCoste> DijkstraInv(const GrafoP<tCoste>& G, typename GrafoP<tCoste>::vertice
destino,
                                                    vector<typename
GrafoP<tCoste>::vertice>& P)
{

    typedef typename GrafoP<tCoste>::vertice vertice;

    size_t n = G.numVert();
    vector<tCoste> costes(n);
    vector<bool> visitado(n,false);

    P=vector<vertice>(n,destino);

    for(int i = 0 ; i < n ; i++){
        costes[i] = G[i][destino];
    }
    costes[destino]=0;
```

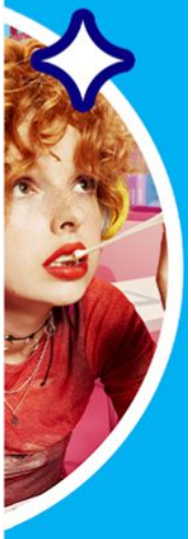
```
    visitado[destino] = true;    //De destino a destino = 0, por lo que ya lo tengo
    for(int k = 0 ; k < n-2 ; k++){ //n -1(por destino) -1(porque al hacerlo con los
otros, queda con el valor adecuado
        tCoste costemin = GrafoP<tCoste>::INFINITO;
        typename GrafoP<tCoste>::vertice v;
        for(int i = 0 ; i < n ; i++){
            if(costes[i] < costemin && !visitado[i]){
                costemin=costes[i];
                v = i;
            }
        }

        visitado[v]=true;
        for(int i = 0 ; i < n ; i++){
            tCoste ivd;
            ivd = suma(G[i][v],costes[v]);
            if(ivd<costes[i]){
                costes[i]=ivd;
                P[i]=v; //P[i]=siguiente valor al que ir hasta destino. Que vaya antes a v
            }

        }
    }
    return costes;


}

template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice orig,
       typename GrafoP<tCoste>::vertice v,
       const vector<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo entre los vértices orig e v
// a partir de un vector P obtenido mediante la función Dijkstra().
{
   typename GrafoP<tCoste>::tCamino C;

   C.insertar(v, C.primera());
   do {
      C.insertar(P[v], C.primera());
      v = P[v];
   } while (v != orig);
   return C;
}

template <typename tCoste>
matriz<tCoste> Floyd(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]) {
               A[i][j] = ikj;
```

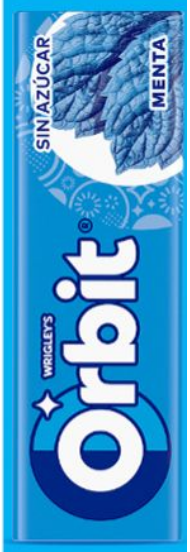
```
                P[i][j] = k;
             }
          }
       }
   return A;
}

template <typename tCoste> typename GrafoP<tCoste>::tCamino
caminoAux(typename GrafoP<tCoste>::vertice v,
          typename GrafoP<tCoste>::vertice w,
          const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo entre v y w, excluidos estos,
// a partir de una matriz P obtenida mediante la función Floyd().
{
   typename GrafoP<tCoste>::tCamino C1, C2;
   typename GrafoP<tCoste>::vertice u = P[v][w];

   if (u != v) {
      C1 = caminoAux<tCoste>(v, u, P);
      C1.insertar(u, C1.fin());
      C2 = caminoAux<tCoste>(u, w, P);
      C1 += C2; // Lista<vertice>::operator +=(), concatena C1 y C2
   }
   return C1;
}

template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice v,
       typename GrafoP<tCoste>::vertice w,
       const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo desde v hasta w a partir
// de una matriz P obtenida mediante la función Floyd().
{
   typename GrafoP<tCoste>::tCamino C = caminoAux<tCoste>(v, w, P);
   C.insertar(v, C.primera());
   C.insertar(w, C.fin());
   return C;
}

/*-------------------------------------------------------------------------*/
/* Árboles generadores de coste mínimo                                     */
/*-------------------------------------------------------------------------*/
template <typename tCoste>
GrafoP<tCoste> Prim(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
   assert(!G.esDirigido());

   typedef typename GrafoP<tCoste>::vertice vertice;
   typedef typename GrafoP<tCoste>::arista arista;
   const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
   arista a;
   const size_t n = G.numVert();
   GrafoP<tCoste> g(n);       // Árbol generador de coste mínimo.
   vector<bool> U(n, false); // Conjunto de vértices incluidos en g.
   Apo<arista> A(n*(n-1)/2-n+2); // Aristas adyacentes al árbol g
                                 // ordenadas por costes.
   U[0] = true;    // Incluir el primer vértice en U.
   // Introducir en el APO las aristas adyacentes al primer vértice
   for (vertice v = 1; v < n; v++)
      if (G[0][v] != INFINITO)
         A.insertar(arista(0, v, G[0][v]));
   for (size_t i = 1; i <= n-1; i++) {     // Seleccionar n-1 aristas.
      // Buscar una arista a de coste mínimo que no forme un ciclo.
      // Nota: Las aristas en A tienen sus orígenes en el árbol g.
      do {
         a = A.cima();
         A.suprimir();
      } while (U[a.dest]); // a forma un ciclo (a.orig y a.dest están en U y en g).
      // Incluir la arista a en el árbol g y el nuevo vértice u en U.
      g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste;
      vertice u = a.dest;
      U[u] = true;
      // Introducir en el APO las aristas adyacentes al vértice u
      // que no formen ciclos.
```

```
      for (vertice v = 0; v < n; v++)
         if (!U[v] && G[u][v] != INFINITO)
            A.insertar(arista(u, v, G[u][v]));
   }
   return g;
}

template <typename tCoste>
GrafoP<tCoste> Kruskall(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
   assert(!G.esDirigido());

   typedef typename GrafoP<tCoste>::vertice vertice;
   typedef typename GrafoP<tCoste>::arista arista;
   const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
   const size_t n = G.numVert();
   GrafoP<tCoste> g(n);    // Árbol generador de coste mínimo.
   Particion P(n);    // Partición inicial del conjunto de vértices de G.
   Apo<arista> A(n*n);     // Aristas de G ordenadas por costes.

   // Copiar aristas del grafo G en el APO A.
   for (vertice u = 0; u < n; u++)
      for (vertice v = u+1; v < n; v++)
         if (G[u][v] != INFINITO)
            A.insertar(arista(u, v, G[u][v]));

   size_t i = 1;
   while (i <= n-1) {    // Seleccionar n-1 aristas.
      arista a = A.cima(); // arista de menor coste
      A.suprimir();
      vertice u = P.encontrar(a.orig);
      vertice v = P.encontrar(a.dest);
      if (u != v) { // Los extremos de a pertenecen a componentes distintas
         P.unir(u, v);
         // Incluir la arista a en el árbol g
         g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste;
         i++;
      }
   }
   return g;
}

#endif   // ALG GRAFO P H
```

```
/*-------------------------------------------------------------*/
/* alg_grafo E-S.h                                             */
/*                                                             */
/* Operadores de inserción en flujos de salida para            */
/* visualizar los resultados de los algoritmos de grafos       */
/* implementados en alg_grafoMA.[h|cpp] y alg_grafoPMC.h       */
/*                                                             */
/*-------------------------------------------------------------*/
/*
Sobrecarga de operadores:

   template <typename T>
   ostream& operator <<(ostream& os, const vector<T>& v);
      Inserción de vector<T> en un ostream (para probar Dijkstra).

   template <typename T>
   ostream& operator << (ostream& os, const matriz<T>& m);
      Inserción de matriz<T> en un ostream (para probar Floyd).

   template <>
   ostream& operator << <bool>(ostream& os, const matriz<bool>& m);
      Especialización para matriz<bool> del operador << para la
      clase genérica matriz<T> (para probar Warshall).

   template <>
   ostream& operator << <size_t>(ostream& os, const Lista<size_t>& L);
      Inserción de una lista de vértices (size_t) de un grafo en un
      flujo de salida (para probar recorridos y caminos de Dijkstra
      y Floyd).

-------------------------------------------------------------*/

#ifndef ALG_GRAFO_E_S_H
#define ALG_GRAFO_E_S_H
#include "grafoPMC.h"  // grafo ponderado
#include <vector>
#include "matriz.h"    // matriz cuadrada
#include "listaenla.h" // para mostrar listas de vértices
#include <ostream>

using std::ostream;

// Inserción de vector<T> en un ostream (para probar Dijkstra)
template <typename T>
ostream& operator <<(ostream& os, const vector<T>& v)
{
   for (size_t i = 0; i < v.size(); i++) {
      os << std::setw(4);
      if (v[i] != GrafoP<T>::INFINITO)
         os << v[i];
      else
         os << "-";
   }
   return os;
}

// Inserta una matriz<T> en un flujo de salida (para probar Floyd)
template <typename T>
ostream& operator << (ostream& os, const matriz<T>& m)
{
   const size_t n = m.dimension();
   os << "    ";
   for (size_t j = 0; j < n; ++j)
      os << std::setw(4) << j;
   os << std::endl;
   for (size_t i = 0; i < n; ++i) {
      os << std::setw(4) << i;
      for (size_t j = 0; j < n; ++j) {
         os << std::setw(4);
         if (m[i][j] == GrafoP<T>::INFINITO)
            os << "-";
         else
            os << m[i][j];
      }
      os << std::endl;
```

```
    }
    return os;
}

// Inserción de matriz<bool> en un flujo de salida (para probar Warshall)
template <>
ostream& operator << <bool>(ostream& os, const matriz<bool>& m);

// Inserción de Lista<size_t> en un ostream (para probar recorridos
// y caminos de Dijkstra y Floyd))
template <typename T>
ostream& operator <<(ostream&, const Lista<T>&);
template <>
ostream& operator << <size_t>(ostream& os, const Lista<size_t>& c);

#endif   // ALG_GRAFO_E_S_H
```

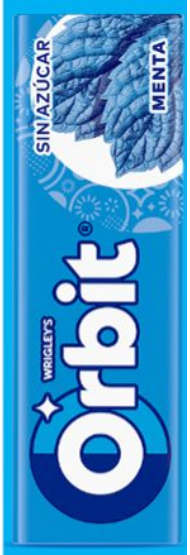




```cpp
/*------------------------------------------------------------*/
/* alg_grafo_E-S.cpp                                          */
/*                                                            */
/* Implementación de operadores de inserción en flujos de     */
/* salida para visualizar los resultados de los               */
/* algoritmos de grafos implementados en                      */
/* alg_grafoMA.[h|cpp] y alg_grafoPMC.h                       */
/*                                                            */
/*------------------------------------------------------------*/

#include "alg_grafo_E-S.h"
#include <iomanip>           // std::setw

// Inserción de matriz<bool> en un flujo de salida (para probar Warshall)
template <>
ostream& operator << <bool>(ostream& os, const matriz<bool>& m)
{
   const size_t n = m.dimension();
   os << "   ";
   for (size_t j = 0; j < n; ++j)
      os << std::setw(3) << j;
   os << std::endl;
   for (size_t i = 0; i < n; ++i) {
      os << std::setw(3) << i;
      for (size_t j = 0; j < n; ++j)
         os << std::setw(3) << m[i][j];
      os << std::endl;
   }
   return os;
}

// Inserción de Lista<size_t> en un ostream (para probar recorridos
// y caminos de Dijkstra y Floyd))
template <>
ostream& operator << <size_t>(ostream& os, const Lista<size_t>& L)
{
   typedef Lista<Grafo::vertice>::posicion posicion;
   for (posicion i = L.primera(); i != L.fin(); i = L.siguiente(i))
      os << L.elemento(i) << ' ';
   return os;
}
```

```
/*-------------------------------------------------------*/
/* grafoPMC.h                                            */
/*                                                       */
/* clase GrafoP<T>: Matriz de costes de un grafo.        */
/*                                                       */
/*-------------------------------------------------------*/
/*
Tipos públicos:

   GrafoP<T>::tCoste      // tipo de los pesos de las aristas
   GrafoP<T>::vertice // valores entre 0 y Grafo::numVert()-1
   GrafoP<T>::arista          // arista de un grafo ponderado
      Miembros públicos:
         vertice orig, dest;    // extremos de la arista
         tCoste coste;          // coste de la arista
         // Constructor
         explicit arista(vertice v = vertice(),
                         vertice w = vertice(),
                         tCoste c = tCoste());
         // Orden de aristas para Prim y Kruskall
         bool operator <(const arista& a) const;
   GrafoP<T>::tCamino // secuencia de vértices que forman un camino

Atributo público:

   static const tCoste GrafoP<T>::INFINITO = std::numeric_limits<T>::max();
      Máximo del rango de valores de tCoste. Peso de una arista inexistente.

Métodos públicos:

   explicit GrafoP(size_t n);
      Grafo ponderado de n vértices sin aristas.

   explicit GrafoP(const std::string& nf);
      Grafo ponderado extraído desde un fichero de texto de
      nombre nf, que contiene el número de vértices seguido de
      los pesos de las aristas en forma de matriz de costes.
      Ejemplo: tCoste = unsignded int, INFINITO = 4294967295
      5
      4294967295         60         10        100 4294967295
      4294967295 4294967295 4294967295 4294967295          5
      4294967295         40 4294967295         30 4294967295
      4294967295 4294967295 4294967295 4294967295         20
      4294967295 4294967295          5 4294967295 4294967295

   GrafoP(const Grafo& G);
      Constructor de conversión. Construye un GrafoP<T> a partir
      de uno no ponderado representado mediante matriz de adyacencia,
      es decir, cambiando la representación a matriz de costes con
      aristas de coste 1.

   size t numVert() const;
      Número de vértices

   const vector<tCoste>& operator [](vertice v) const;
   vector<tCoste>& operator [](vertice v);
      Pesos de las aristas adyacentes al vértice v.
      Fila v de la matriz de costes.

   bool esDirigido() const;
      Indica si un grafo ponderado es dirigido (true) o no (false).

Sobrecarga de operador externo:

   template <typename T>
   std::ostream& operator <<(std::ostream& os, const GrafoP<T>& G);
      Inserción de un grafo ponderado en un flujo de salida.

-------------------------------------------------------------*/

#ifndef GRAFO_PON_H
#define GRAFO_PON_H
#include <vector>
#include <limits>
#include <ostream>
```

```cpp
#include <fstream>
#include <iomanip>
#include <string>
#include "listaenla.h" // requerido por GrafoP<T>::tCamino
#include "grafoMA.h"   // requerido por GrafoP<T>::GrafoP(const Grafo&)

using std::vector;

template <typename T> class GrafoP {
public:
   typedef T tCoste;
   typedef size_t vertice;
   struct arista {
      vertice orig, dest;
      tCoste coste;
      explicit arista(vertice v = vertice(), vertice w = vertice(),
                      tCoste c = tCoste()): orig(v), dest(w), coste(c) {}
      bool operator <(const arista& a) const {return coste < a.coste;}
   };
   typedef Lista<vertice> tCamino;

   static const tCoste INFINITO;

   explicit GrafoP(size_t n): costes(n, vector<tCoste>(n, INFINITO)) {}
   explicit GrafoP(const std::string& nf);
   GrafoP(const Grafo& G);
   size_t numVert() const {return costes.size();}
   const vector<tCoste>& operator [](vertice v) const {return costes[v];}
   vector<tCoste>& operator [](vertice v) {return costes[v];}
   bool esDirigido() const;
private:
   vector< vector<tCoste> > costes;
};

// Definición de INFINITO
template <typename T>
const T GrafoP<T>::INFINITO = std::numeric_limits<T>::max();

// Constructor desde fichero
template <typename T>
GrafoP<T>::GrafoP(const std::string& nf)
{
   std::ifstream fg(nf);            // apertura del fichero
   unsigned n;                      // núm. vértices
   fg >> n;
   costes = vector<vector<T> >(n, vector<T>(n));
   for (vertice i = 0; i < n; i++)
      for (vertice j = 0; j < n; j++)
         fg >> costes[i][j];
   fg.close();                      // cierre del fichero
}

// Construye un GrafoP<T> a partir de uno no ponderado representado
// mediante matriz de adyacencia, es decir, cambiando la representación
// a matriz de costes con aristas de coste 1.
template <typename T>
GrafoP<T>::GrafoP(const Grafo& G):
   costes(G.numVert(), vector<T>(G.numVert()))
{
   const size_t n = G.numVert();
   for (vertice i = 0; i < n; i++)
      for (vertice j = 0; j < n; j++)
         costes[i][j] = G[i][j] ? 1 : GrafoP<T>::INFINITO;
}

// dirigido = true, no dirigido = false
template <typename T>
bool GrafoP<T>::esDirigido() const
{
   bool s = true;    // matriz simétrica ==> no dirigido
   const size_t n = numVert();
   vertice i = 0;
   while (s && i < n) {
      vertice j = i+1;
      while (s && j < n) {
```



```
            s = (costes[i][j] == costes[j][i]);
            ++j;
         }
         ++i;
      }
      return !s;    // no simétrica ==> dirigido
   }

   // Inserción de un grafo ponderado en un flujo de salida.
   template <typename T>
   std::ostream& operator <<(std::ostream& os, const GrafoP<T>& G)
   {
      typedef typename GrafoP<T>::vertice vertice;
      const size_t n = G.numVert();
      os << n << " vertices" << std::endl;
      os << "    ";
      for (vertice j = 0; j < n; j++)
         os << std::setw(4) << j;
      os << std::endl;
      for (vertice i = 0; i < n; i++) {
         os << std::setw(4) << i;
         for (vertice j = 0; j < n; j++) {
            os << std::setw(4);
            if (G[i][j] == GrafoP<T>::INFINITO)
               os << "-";
            else
               os << G[i][j];
         }
         os << std::endl;
      }
      return os;
   }

   #endif   // GRAFO_PON_H
```

```
/*-------------------------------------------------------*/
/* listaenla.h                                           */
/*                                                       */
/* clase Lista genérica mediante celdas enlazadas.       */
/*    Después de una inserción o eliminación en una      */
/*    posición p, las variables externas de tipo posición */
/*    posteriores a p continúan representado las         */
/*    posiciones de los mismos elementos.                */
/*-------------------------------------------------------*/

#ifndef LISTA_ENLA_H
#define LISTA_ENLA_H
#include <cassert>

template <typename T> class Lista {
   struct nodo;    // declaración adelantada privada
public:
   typedef nodo* posicion;    // posición de un elemento
   Lista();                   // constructor, requiere ctor. T()
   Lista(const Lista<T>& l); // ctor. de copia, requiere ctor. T()
   Lista<T>& operator =(const Lista<T>& l); // asignación de listas
   void insertar(const T& x, posicion p);
   void eliminar(posicion p);
   const T& elemento(posicion p) const; // acceso a elto, lectura
   T& elemento(posicion p);    // acceso a elto, lectura/escritura
   posicion buscar(const T& x) const; // T requiere operador ==
   posicion siguiente(posicion p) const;
   posicion anterior(posicion p) const;
   posicion primera() const;
   posicion fin() const;       // posición después del último
   ~Lista();                   // destructor
   // Usado para algoritmos de grafos
   Lista<T>& operator +=(const Lista<T>& l); // concatenación
private:
   struct nodo {
       T elto;
       nodo* sig;
       nodo(const T& e, nodo* p = 0): elto(e), sig(p) {}
   };

   nodo* L;    // lista enlazada de nodos

   void copiar(const Lista<T>& l);
};

/*-------------------------------------------------------*/
/* clase Lista genérica mediante celdas enlazadas con    */
/* cabecera.                                             */
/*    La posición de un elemento se representa mediante  */
/*    un puntero al nodo anterior.                       */
/*    La primera posición es un puntero al nodo cabecera. */
/*    La posición fin es un puntero al último nodo.      */
/*                                                       */
/* Implementación de operaciones                         */
/*-------------------------------------------------------*/

// Método privado
template <typename T>
void Lista<T>::copiar(const Lista<T> &l)
{
   L = new nodo(T());  // crear el nodo cabecera
   nodo* q = L;
   for (nodo* r = l.L->sig; r; r = r->sig) {
      q->sig = new nodo(r->elto);
      q = q->sig;
   }
}

template <typename T>
inline Lista<T>::Lista() : L(new nodo(T())) // crear cabecera
{}

template <typename T>
inline Lista<T>::Lista(const Lista<T>& l)
{
```

```
   copiar(l);
}

template <typename T>
Lista<T>& Lista<T>::operator =(const Lista<T>& l)
{
   if (this != &l) {  // evitar autoasignación
      this->~Lista(); // vaciar la lista actual
      copiar(l);
   }
   return *this;
}

template <typename T> inline
void Lista<T>::insertar(const T& x, Lista<T>::posicion p)
{
   p->sig = new nodo(x, p->sig);
   // el nuevo nodo con x queda en la posición p
}

template <typename T>
inline void Lista<T>::eliminar(Lista<T>::posicion p)
{
   assert(p->sig); // p no es fin
   nodo* q = p->sig;
   p->sig = q->sig;
   delete q;
   // el nodo siguiente queda en la posición p
}

template <typename T> inline
const T& Lista<T>::elemento(Lista<T>::posicion p) const
{
   assert(p->sig);   // p no es fin
   return p->sig->elto;
}

template <typename T>
inline T& Lista<T>::elemento(Lista<T>::posicion p)
{
   assert(p->sig);   // p no es fin
   return p->sig->elto;
}

template <typename T>
typename Lista<T>::posicion
   Lista<T>::buscar(const T& x) const
{
   nodo* q = L;
   bool encontrado = false;
   while (q->sig && !encontrado)
      if (q->sig->elto == x)
         encontrado = true;
      else q = q->sig;
   return q;
}

template <typename T> inline
typename Lista<T>::posicion
   Lista<T>::siguiente(Lista<T>::posicion p) const
{
   assert(p->sig);   // p no es fin
   return p->sig;
}

template <typename T>
typename Lista<T>::posicion
   Lista<T>::anterior(Lista<T>::posicion p) const
{
   nodo* q;

   assert(p != L);   // p no es la primera posición
   for (q = L; q->sig != p; q = q->sig);
   return q;
}
```

```cpp
template <typename T>
inline typename Lista<T>::posicion Lista<T>::primera() const
{
   return L;
}

template <typename T>
typename Lista<T>::posicion Lista<T>::fin() const
{
   nodo* p;
   for (p = L; p->sig; p = p->sig);
   return p;
}

// Destructor: destruye el nodo cabecera y vacía la lista
template <typename T> Lista<T>::~Lista()
{
   nodo* q;
   while (L) {
      q = L->sig;
      delete L;
      L = q;
   }
}

/*--------------------------------------------------------------------------*/
/* Para algoritmos de grafos                                                */
/* -------------------------------------------------------------------------*/
// Concatenación de listas (para recorridos)
template<typename T>
Lista<T>& Lista<T>::operator +=(const Lista<T>& l)
{
   for (Lista<T>::posicion i = l.primera(); i != l.fin(); i = l.siguiente(i))
      insertar(l.elemento(i), fin());
   return *this;
}

template<typename T>
Lista<T> operator +(const Lista<T>& l1, const Lista<T>& l2)
{
   return Lista<T>(l1) += l2;
}

#endif // LISTA ENLA H
```

¿DÍA DE CLASES infinitas?

WRIGLEY'S Orbit

masca y fluye

```
/*Ejercicio 1 Práctica 7*/
```

```cpp
/*Ejercicio 2 Práctica 7*/


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


struct Casilla{
    Casilla()=default;
    Casilla(int f, int c):f(f),c(c){}
    int f,c;
};

struct pared{
    pared(bool arriba = false, bool derecha = false):arriba(arriba), derecha(derecha){}
    bool arriba,derecha;

};



typename GrafoP<int>::vertice CasillatoNodo(Casilla c, size_t N);

Casilla nodotoCasilla(typename GrafoP<int>::vertice v, size_t N);

bool adyacente(Casilla c1, Casilla c2);
//En la matriz cada muro solo lo representa la casilal de abajo o la de la izqda



template <typename tCoste>
Lista<Casilla> ResolverLaberinto(size_t N, matriz<pared> muros, Casilla origen,
                                                    Casilla salida);




using namespace std;
//Para un int, INFINITO es: 2147483647


int main(){
    size_t N = 3;
    matriz<pared> mpared(N*N);
    Casilla origen(0,0), salida(2,2);

    Lista<Casilla> listaC = ResolverLaberinto<int>(N,mpared,origen,salida);

    typename Lista<Casilla>::posicion p = listaC.primera();

    cout<<"El camino es: \n";
    while (p != listaC.fin()){
        cout<<" "<<listaC.elemento(p).f<<" "<<listaC.elemento(p).c<<endl;;
        p = listaC.siguiente(p);
    }
}

int abs(int a){return (a>=0)?a:-a;}

typename GrafoP<int>::vertice CasillatoNodo(Casilla c, size_t N){return c.f*N+c.c;}

Casilla nodotoCasilla(typename GrafoP<int>::vertice v, size_t N){
//cout<<"NODOTOCASILLA: de "<<v<<" a "<<v/N<<"|"<<v%N<<endl;
    return Casilla(v/N,v%N);}

bool adyacente(Casilla c1, Casilla c2){return (abs(c1.f - c2.f) + abs(c1.c - c2.c)) == 1;}
//En la matriz cada muro solo lo representa la casilal de abajo o la de la izqda
```

```cpp
template <typename tCoste>
Lista<Casilla> ResolverLaberinto(size_t N, matriz<pared> muros, Casilla origen,
                                                    Casilla salida)
{
    size_t len = N*N;
    GrafoP<tCoste> G(len);
    typedef typename GrafoP<tCoste>::vertice vertice;


    cout<<"FUN"<<endl;

    //Generar el Laberinto:
    for(vertice i = 0 ; i < len ; i++){
        for(vertice j = 0; j < len ; j++){
            if(adyacente(nodotoCasilla(i,N),nodotoCasilla(j,N))){
                G[i][j] = 1;
                cout<<i<<" y "<<j<<" son adyacentes"<<endl;
            }
            else    G[i][j] = GrafoP<tCoste>::INFINITO;
        }
        G[i][i] = 0;    //El bucle de dentro lo pondrá a infinito, ahora se pone a 0

    }
    cout<<"El grafo ahora mismo es:\n"<<G;



    matriz<vertice> vertices(len);
    matriz<tCoste> Mfloyd = Floyd(G,vertices);
    cout<<"FLYOD\n: "<<Mfloyd<<endl;

    typename GrafoP<tCoste>::tCamino lista =
camino<tCoste>(CasillatoNodo(origen,N),CasillatoNodo(salida,N),vertices);
    cout<<"test1"<<endl;
    typename Lista<vertice>::posicion p = lista.primera();
    Lista<Casilla> listaC;

    cout<<"Camino interno: \n"<<endl;
    while(p != lista.fin()){
            cout<<" "<<lista.elemento(p);
        listaC.insertar(nodotoCasilla(lista.elemento(p),N),listaC.fin());
        p = lista.siguiente(p);
    }
    cout<<endl;
    return listaC;



}
```

```
/*Ejercicio 3 Práctica 7*/


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

using namespace std;

template <typename tCoste>
tCoste DistribuirProducto(GrafoP<tCoste> G, typename GrafoP<tCoste>::vertice origen,
size_t cant,
                          const vector<double>& subvenciones, const vector<size_t>&
capacidades, vector<size_t>& envio);


/*
Ej3.dat



2147483647  1           2           2147483647  2147483647
2147483647  2147483647  2147483647  3           4
2147483647  2147483647  2147483647  100         2147483647
2147483647  2147483647  100         2147483647  2147483647
2147483647  2147483647  2147483647  2147483647  2147483647


*/

int main(){

    GrafoP<int> G("Ej3.dat");   //Costes por unidad
    cout<<"EL grafo es: \n"<<G<<endl;

    size_t n = G.numVert(),cant;
    vector<double> subvenciones(n);
    vector<size_t> capacidades(n),envio(n,0);  //En envio se meterán las cantidades a
enviar, en cantidades la capacida

    typename GrafoP<int>::vertice origen;
    cout<<"Introduce la ciudad origen: ";
    cin >> origen;
    cout<<"Introduce la cantidad de producto: ";
    cin >> cant;

    for(int i = 0 ; i < G.numVert() ; i++){
        cout<<"Subvenciones para la ciudad "<<i<<": ";
        cin>>subvenciones[i];
    }

    for(int i = 0 ; i < G.numVert() ; i++){
        cout<<"Capacidad que puede almacenar la ciudad "<<i<<": ";
        cin >> capacidades[i];
    }
    capacidades[origen] = 0;

    int coste = DistribuirProducto(G,origen,cant,subvenciones,capacidades,envio);
    cout<<"El coste ha sido: "<<coste<<endl;

    for(int i = 0 ; i < n ; i++){
        cout<<"Ciudad "<<i<<": "<<envio[i]<<" unidades"<<endl;
    }
}

template <typename tCoste>
tCoste DistribuirProducto(GrafoP<tCoste> G, typename GrafoP<tCoste>::vertice origen,
size_t cant,
                          const vector<double>& subvenciones, const vector<size_t>&
capacidades, vector<size_t>& envio)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t n = G.numVert();
```

```
    //Primero hay que calcular los costes con la subvencion. Se hace en G (para eso se
recibe por copia)
    for(vertice i = 0 ; i < n ; i++){
        for(vertice j = 0 ; j < n ; j++){
            if(G[i][j] != GrafoP<tCoste>::INFINITO) G[i][j] = G[i][j] *
(1-subvenciones[j]);
        }
    }
    cout<<"Grafo tras subvenciones: "<<G<<endl;

    //Ahora determinaremos los costes mínimos desde origen hasta las otras ciudades =>
Dijkstra
    vector<vertice> verticesDij(n);
    vector<tCoste> vecDij = Dijkstra(G,origen,verticesDij);

    cout<<"VEC DIJ: "<<endl;
    for(vertice i = 0 ; i < n ; i++){
        cout<<" "<<vecDij[i];
    }
    cout<<endl;

    //A continuación, se determinan cuantas unidades enviar a cada ciudad:
    tCoste acum = 0;

    while(cant > 0){
        tCoste minimo = GrafoP<tCoste>::INFINITO;
        vertice v;
        for(vertice i = 0 ; i < n ; i++){
            if(i != origen && envio[i] == 0 && vecDij[i]<minimo){
                minimo = i;
                v = i;
            }
        }

        //Una vez seleccionada la ciudad, se le envían todos los que se puedan
        if(capacidades[v] > cant){
            envio[v]=cant;
            acum+=(cant*vecDij[v]);

            cant=0;
        }else{
            envio[v] = capacidades[v];
            acum+=(cant*capacidades[v]);
            cant-=capacidades[v];
        }
    }

    return acum;

}
```

```cpp
/*Ejercicio 4 Práctica 7*/


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


template <typename tCoste>
tCoste TrabajoDiario(const GrafoP<tCoste>& G, vector<size_t> parte, typename
GrafoP<int>::vertice capital);


/*
Ej4.dat

5
2147483647   60           10           100          2147483647
2            2147483647   22           2147483647   5
3            40           44           30           59
4            2147483647   2147483647   2147483647   20
5            23           5            33           2147483647



*/


int main(){

    GrafoP<int> G("Ej4.dat");
    cout<<"El grafo actual es: "<<G;

    vector<size_t> parte = {0,0,100,50,25};
    typename GrafoP<int>::vertice capital = 0;

    int distancia = TrabajoDiario(G,parte,capital);
    cout<<"La distancia es: "<<distancia;

}

template <typename tCoste>
tCoste TrabajoDiario(const GrafoP<tCoste>& G, vector<size_t> parte, typename
GrafoP<int>::vertice capital)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t n = G.numVert();
    tCoste distancia = 0;
    //Calculamos los costes de capital a la ciudad:

    vector<vertice> verDij(n),verDijInv(n);
    vector<tCoste> vecDij, vecDijInv;

    vecDij = Dijkstra(G,capital,verDij);
    vecDijInv = DijkstraInv(G,capital,verDijInv);

    cout<<"VEC DIJ: "<<vecDij<<endl;
    cout<<"VEC DIJINV: "<<vecDijInv<<endl;


    for(vertice i = 0 ; i < n ; i++){
        distancia+=(parte[i]*vecDij[i]);
        distancia+=(parte[i]*vecDijInv[i]);
    }
    return distancia;
}
```

```
/*Ejercicio 5 Práctica 7*/


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


template <typename tCoste>
vector<bool> CalcularDestinos(size_t alergico, int presupuesto, typename
GrafoP<tCoste>::vertice origen,
                              const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,const
GrafoP<tCoste>& G3);
/*


Ej5_1.dat

5
0           2147483647  10          2147483647  22
2147483647  0           2147483647  2147483647  2147483647
40          30          0           30          2147483647
2147483647  22          2147483647  0           22
2147483647  2147483647  56          2147483647  0


Ej5_2.dat

5
0           2147483647  2147483647  2147483647  2
2147483647  0           9           2147483647  5
10          10          0           30          40
2147483647  2147483647  2147483647  0           100
62          2147483647  2147483647  2147483647  0


Ej5_3.dat

5
0           20          10          100         20
21          0           80          2147483647  5
2147483647  2147483647  0           2147483647  40
2147483647  65          77          0           2147483647
21          2147483647  2147483647  22          0




*/
int main(){

    cout<<"HOLA"<<endl;
    GrafoP<int> G1("Ej5_1.dat"),G2("Ej5_2.dat"),G3("Ej5_3.dat");
    cout<<"Los grafos son: "<<endl;
    cout<<G1<<endl;
    cout<<G2<<endl;
    cout<<G3<<endl;

    size_t alergico = 3;
    int presupuesto = 10;   //Aumentar para tener otro resultado
    typename GrafoP<int>::vertice origen = 0;

    vector<bool> disponible = CalcularDestinos(alergico,presupuesto,origen,G1,G2,G3);

    cout<<"Puede viajar a: "<<endl;
    for(int i = 0 ; i < G1.numVert() ; i++){
        if(disponible[i]){
            cout<<" "<<i;
        }
    }
}
```

```cpp
template <typename tCoste>
vector<bool> CalcularDestinos(size_t alergico, int presupuesto, typename
GrafoP<tCoste>::vertice origen,
                              const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,const
GrafoP<tCoste>& G3)
{

    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t n = G1.numVert();

    GrafoP<tCoste> A(n);
    switch(alergico){
    case 1:{
        for(vertice i = 0 ; i < n ; i++){
            for(vertice j = 0 ; j < n ; j++){
                if(G2[i][j] < G3[i][j])   A[i][j] = G2[i][j];
                else A[i][j] = G3[i][j];
            }
        }
    };break;
    case 2:{
        for(vertice i = 0 ; i < n ; i++){
            for(vertice j = 0 ; j < n ; j++){
                if(G1[i][j] < G3[i][j])   A[i][j] = G1[i][j];
                else A[i][j] = G3[i][j];
            }
        }
    };break;
    case 3:{
        for(vertice i = 0 ; i < n ; i++){
            for(vertice j = 0 ; j < n ; j++){
                if(G1[i][j] < G2[i][j])   A[i][j] = G1[i][j];
                else A[i][j] = G2[i][j];
            }
        }
    }
    }

    cout<<"Grafo resul: "<<A;

    vector<vertice> vert(n);
    vector<tCoste>  vecDij = Dijkstra(A,origen,vert);
    vector<bool> viajable(n,false);

    for(vertice i = 0 ; i < n ; i++){
        if(vecDij[i] != GrafoP<tCoste>::INFINITO && vecDij[i]<presupuesto){
            viajable[i]=true;
        }
    }
    return viajable;
}
```

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


template <typename tCoste>
matriz<tCoste> CalculaCostes(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, typename
GrafoP<tCoste>::vertice cambio);


/*

Ej6 1.dat

5
0            2147483647   2147483647   2147483647   200
2147483647   0            9            2147483647   5
10           10           0            30           40
2147483647   2147483647   2147483647   0            100
62           2147483647   2147483647   2147483647   0


Ej6_2.dat

5
0            2147483647   10           2147483647   220
2147483647   0            2147483647   2147483647   2147483647
40           30           0            30           2147483647
2147483647   22           2147483647   0            22
2147483647   2147483647   56           2147483647   0


*/

int main(){
    GrafoP<int> G1("Ej6_1.dat"),G2("Ej6_2.dat");
    typename GrafoP<int>::vertice cambio = 3;

    cout<<G1<<G2<<endl;
    matriz<int> costes = CalculaCostes(G1,G2,cambio);
    cout<<"La matriz es: "<<endl;
    cout<<costes;
}

template <typename tCoste>
matriz<tCoste> CalculaCostes(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, typename
GrafoP<tCoste>::vertice cambio)
{
    size_t n = G1.numVert();
    typedef typename GrafoP<tCoste>::vertice vertice;

    matriz<tCoste> costesfinales(n), costes(n);
    matriz<vertice> vertfloyd(n);

    //Inicializo la matriz con los costes de ir todo con G1 y luego la modifico si es
necesario
    costesfinales= Floyd(G1,vertfloyd);

    //Ahora veamos si viajando todo en G2, mejora en algo
    costes = Floyd(G2,vertfloyd);

    for(vertice i = 0 ; i < n ; i++){
        for(vertice j = 0 ; j < n ; j++){
            if(costes[i][j] < costesfinales[i][j]) costesfinales[i][j] = costes[i][j];
        }
    }

    vector<tCoste> Dij(n),DijInv(n);
    vector<vertice> vertDij;

    //Veamos el caso de ir en G1 hasta cambio y luego en G2
```

```
    DijInv = DijkstraInv(G1,cambio,vertDij);
    Dij = Dijkstra(G2,cambio, vertDij);

    for(vertice i = 0 ; i < n ; i++){
        for(vertice j = 0 ; j < n ; j++){
            if(costesfinales[i][j] > suma(DijInv[i],Dij[j]))    costesfinales[i][j] =
suma(DijInv[i],Dij[j]);
        }
    }

    //Primero en G1 hasta cambio y luego en G2
    DijInv = DijkstraInv(G2,cambio,vertDij);
    Dij = Dijkstra(G1,cambio, vertDij);

    for(vertice i = 0 ; i < n ; i++){
        for(vertice j = 0 ; j < n ; j++){
            if(costesfinales[i][j] > suma(DijInv[i],Dij[j]))    costesfinales[i][j] =
suma(DijInv[i],Dij[j]);
        }
    }

    return costesfinales;

}
```

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



template <typename tCoste>
tCoste CalcularViaje(const GrafoP<tCoste>&G1, const GrafoP<tCoste>&G2, typename
GrafoP<int>::vertice origen,
                     typename GrafoP<int>::vertice destino, typename GrafoP<int>::vertice
cambio1,
                     typename GrafoP<int>::vertice cambio2, typename
GrafoP<tCoste>::tCamino& ruta);



/*

Ej6_1.dat

5
0            2147483647   2147483647   2147483647   200
2147483647   0            9            2147483647   5
10           10           0            30           40
2147483647   2147483647   2147483647   0            100
62           2147483647   2147483647   2147483647   0


Ej6_2.dat

5
0            2147483647   10           2147483647   220
2147483647   0            2147483647   2147483647   2147483647
40           30           0            30           2147483647
2147483647   22           2147483647   0            22
2147483647   2147483647   56           2147483647   0


*/

int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej6_1.dat"),G2("Ej6_2.dat");
    cout<<G1<<G2<<endl;

    vertice origen=0,destino=4,cambio1=1,cambio2=2;

    //Ruta será el vector

    typename GrafoP<int>::tCamino ruta;
    int coste = CalcularViaje(G1,G2,origen,destino,cambio1,cambio2,ruta);

    cout<<"El coste es: "<<coste<<endl;
    typename GrafoP<int>::tCamino::posicion p = ruta.primera();
    cout<<"Camino: ";
    while(p != ruta.fin()){
        cout<<" "<<ruta.elemento(p);
        p = ruta.siguiente(p);
    }

}

template <typename tCoste>
tCoste CalcularViaje(const GrafoP<tCoste>&G1, const GrafoP<tCoste>&G2, typename
GrafoP<int>::vertice origen,
                     typename GrafoP<int>::vertice destino, typename GrafoP<int>::vertice
cambio1,
                     typename GrafoP<int>::vertice cambio2, typename
GrafoP<tCoste>::tCamino& ruta)
{
```

```
    size_t n = G1.numVert();
    typedef typename GrafoP<tCoste>::vertice vertice;

    matriz<vertice> mvert1(n),mvert2(n);
    matriz<tCoste> mfloyd1(n),mfloyd2(n);

    mfloyd1 = Floyd(G1,mvert1);
    mfloyd2 = Floyd(G2,mvert2);

    tCoste costemin;

    if(suma(mfloyd1[origen][cambio1], mfloyd2[cambio1][destino]) <
suma(mfloyd1[origen][cambio2],mfloyd2[cambio2][destino])){
        costemin = suma(mfloyd1[origen][cambio1], mfloyd2[cambio1][destino]);
        ruta =
camino<tCoste>(origen,cambio1,mvert1)+=camino<tCoste>(cambio1,destino,mvert2);
    }else{
        costemin = suma(mfloyd1[origen][cambio2], mfloyd2[cambio2][destino]);
        ruta = camino<tCoste>(origen,cambio2,mvert1) +=
camino<tCoste>(cambio2,destino,mvert2);
    }
}
```

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


template <typename tCoste>
tCoste CalcularCoste(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, typename
GrafoP<tCoste>::vertice origen,
                     typename GrafoP<tCoste>::vertice destino);


/*

Ej6 1.dat

5
0           2147483647  2147483647  2147483647  200
2147483647  0           9           2147483647  5
10          10          0           30          40
2147483647  2147483647  2147483647  0           100
62          2147483647  2147483647  2147483647  0


Ej6_2.dat

5
0           2147483647  10          2147483647  220
2147483647  0           2147483647  2147483647  2147483647
40          30          0           30          2147483647
2147483647  22          2147483647  0           22
2147483647  2147483647  56          2147483647  0


*/

int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej6_1.dat"),G2("Ej6_2.dat");
    cout<<G1<<G2<<endl;

    vertice origen=0,destino=4;

    int coste = CalcularCoste(G1,G2,origen,destino);

    cout<<"El coste minimo es: "<<coste<<endl;

}

template <typename tCoste>
tCoste CalcularCoste(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, typename
GrafoP<tCoste>::vertice origen,
                     typename GrafoP<tCoste>::vertice destino)
{

    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t n = G1.numVert();

    vector<vertice> verDij1,verDijInv1,verDij2,verDijInv2;
    vector<tCoste> vecDij1,vecDijInv1,vecDij2,vecDijInv2;

    //Calculamos los costes de origen a todas las ciudades (Dijkstra) tanto en medio de
transporte 1 como en 2
    vecDij1 = Dijkstra(G1,origen,verDij1);
    vecDij2 = Dijkstra(G2,origen,verDij2);

    //Calculamos los costes de ir desde cualquier ciudad a destino (Dijkstra Inverso) en
ambos casos
    vecDijInv1 = DijkstraInv(G1,destino,verDijInv1);
    vecDijInv2 = DijkstraInv(G2,destino,verDijInv2);
```

```
    //Ahora compruebo todos los posibles transbordos y me quedo con el menor
    tCoste costemin = min(vecDij1[destino],vecDij2[destino]);   //Inicializo costemin al
coste mínimo directo

    //Ahora compruebo si algún transbordo lo mejora
    for(vertice i = 0 ; i < n ; i++){
        if(i != origen && i != destino){
            //Haciendo trasbordo de 1 a 2
            if(suma(vecDij1[i],vecDijInv2[i]) < costemin) costemin =
suma(vecDij1[i],vecDijInv2[i]);

            //Haciendo transbordo de 2 a 1
            if(suma(vecDij2[i],vecDijInv1[i]) < costemin) costemin =
suma(vecDij2[i],vecDijInv1[i]);
        }

    }
    return costemin;

}
```

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



template <typename tCoste>
tCoste CalcularViajeEj9(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, tCoste taxi,
typename GrafoP<tCoste>::vertice origen,
                        typename GrafoP<tCoste>::vertice destino, typename
GrafoP<tCoste>::tCamino& ruta);



/*

Ej6_1.dat

5
0            2147483647   2147483647   2147483647   200
2147483647   0            9            2147483647   5
10           10           0            30           40
2147483647   2147483647   2147483647   0            100
62           2147483647   2147483647   2147483647   0


Ej6_2.dat

5
0            2147483647   10           2147483647   220
2147483647   0            2147483647   2147483647   2147483647
40           30           0            30           2147483647
2147483647   22           2147483647   0            22
2147483647   2147483647   56           2147483647   0


*/

int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej6 1.dat"),G2("Ej6 2.dat");
    cout<<G1<<G2<<endl;

    int taxi = 15;
    vertice origen=0,destino=4;
    typename GrafoP<int>::tCamino ruta;
    cout<<"MAIN"<<endl;
    int coste = CalcularViajeEj9(G1,G2,taxi,origen,destino,ruta);

    cout<<"El coste minimo es: "<<coste<<endl;

    typename GrafoP<int>::tCamino::posicion p = ruta.primera();
    while(p != ruta.fin()){
        cout<<" "<<ruta.elemento(p);
        p = ruta.siguiente(p);
    }


}

template <typename tCoste>
tCoste CalcularViajeEj9(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, tCoste taxi,
typename GrafoP<tCoste>::vertice origen,
                        typename GrafoP<tCoste>::vertice destino, typename
GrafoP<tCoste>::tCamino& ruta)
{
    size_t n = G1.numVert();
    size_t len = n+n;
    typedef typename GrafoP<tCoste>::vertice vertice;
```

```
    //Creamos el nuevo grafo
    GrafoP<tCoste> A(len);  //Por defecto todo infinito
    for(int i = 0 ; i< n ; i++){
            A[n+i][i] = A[i][n+i] = taxi;
        for(int j = 0 ; j < n ; j++){
            A[i][j] = G1[i][j];
            A[n+i][n+j] = G2[i][j];
        }
    }
    //Ahora tengo que calcular el mínimo de origen1 a destino1, de origen1 a destino2,
    // de origen2 a destino1 y de origen2 a destino2 (los 1 son los normales, los 2
sumándole n)

    vector<vertice> verDij1(len), verDij2(len);
    vector<tCoste> vecDij1(len), vecDij2(len);

    //Primero, de origen1 a todos(entre ellos destino1 y destino2)
    vecDij1 = Dijkstra(A,origen,verDij1);

    //Ahora de origen2 a todos(entre ellos destino1 y destino2)
    vecDij2 = Dijkstra(A,origen+n,verDij2);

    //Busco el mínimo de todos
    tCoste costemin =
min(min(vecDij1[destino],vecDij1[destino+n]),min(vecDij2[destino],vecDij2[destino+2]));

    //Ahora compruebo a ver de dónde he sacado ese costemin
    if(vecDij1[destino] == costemin){
        ruta = camino<tCoste>(origen,destino,verDij1);
    }else if(vecDij1[destino+n] == costemin){
        ruta = camino<tCoste>(origen,destino+n,verDij1);
    }else if(vecDij2[destino] == costemin){
        cout<<3<<endl;
        ruta = camino<tCoste>(origen+n,destino,verDij2);
    }else{  //vecDij2[destino+n] == costemin
        cout<<4<<endl;
        ruta = camino<tCoste>(origen+n,destino+n,verDij2);
    }
    typename GrafoP<tCoste>::tCamino::posicion p = ruta.primera();

    //NOTA: SE VA A DEJAR QUE APAREZCA EN LA LISTA 2 VECES SEGUIDAS UN MISMO NODO PARA
INDICAR QUE SE
    //HA HECHO UN TRANSBORDO EN ESA CIUDAD. SI NO SE QUISIERA ASÍ, AL IGUAL QUE SE RECORRE
LA LISTA
    //HACIENDO %n, SE PODRÍA ELIMINAR UN NODO CUANDO APAREZCA 2 VECES SEGUIDO EL MISMO
    while(p != ruta.fin()){
        ruta.elemento(p) = ruta.elemento(p) % n;    //Para dejar los nodos en su valor
original
        p = ruta.siguiente(p);
    }

    return costemin;
}
```

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



template <typename tCoste>
tCoste CalcularViajeEj10(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, const
GrafoP<tCoste>& G3,
                         typename GrafoP<tCoste>::vertice origen, typename
GrafoP<tCoste>::vertice destino,
                         tCoste taxi1_2, tCoste taxi3, typename GrafoP<tCoste>::tCamino&
ruta);

template <typename tCoste>
typename GrafoP<tCoste>::tCamino calcularRuta(tCoste costemin, typename
GrafoP<tCoste>::vertice origen,
                                              typename GrafoP<tCoste>::vertice destino,
size_t n, vector<tCoste> vecDij,
                                              vector<typename GrafoP<tCoste>::vertice>
verDij);


/*


Ej5_1.dat

5
0           2147483647  10          2147483647  22
2147483647  0           2147483647  2147483647  2147483647
40          30          0           30          2147483647
2147483647  22          2147483647  0           22
2147483647  2147483647  56          2147483647  0


Ej5_2.dat

5
0           2147483647  2147483647  2147483647  2
2147483647  0           9           2147483647  5
10          10          0           30          40
2147483647  2147483647  2147483647  0           100
62          2147483647  2147483647  2147483647  0


Ej5_3.dat

5
0           20          10          100         20
21          0           80          2147483647  5
2147483647  2147483647  0           2147483647  40
2147483647  65          77          0           2147483647
21          2147483647  2147483647  22          0


*/


int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej5_1.dat"),G2("Ej5_2.dat"),G3("Ej5_3.dat");
    cout<<G1<<G2<<G3;
    int taxi1_2 = 1, taxi3 = 1;        //Taxi3 es el taxi entre 1-3 y entre 2-3
    vertice origen = 0, destino = 3;

    typename GrafoP<int>::tCamino ruta;
    int coste = CalcularViajeEj10(G1,G2,G3,origen,destino,taxi1_2,taxi3,ruta);
```

```
    cout<<"Coste: "<<coste<<endl;
    cout<<"Camino: ";
    typename GrafoP<int>::tCamino::posicion p = ruta.primera();
    while(p != ruta.fin()){
        cout<<" "<<ruta.elemento(p);
        p = ruta.siguiente(p);
    }
    cout<<endl;
}

template <typename tCoste>
tCoste CalcularViajeEj10(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2, const
GrafoP<tCoste>& G3,
                         typename GrafoP<tCoste>::vertice origen, typename
GrafoP<tCoste>::vertice destino,
                         tCoste taxi1_2, tCoste taxi3, typename GrafoP<tCoste>::tCamino&
ruta)
{

    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t n = G1.numVert();
    size_t len = 3*n;

    // Desde G1 se podrá ir a cualquier ciudad por G1, a la estación del medio2 en la
ciudad en la que
    //se esté(coste taxi1 2) y a la estación de medio3 de la ciudad en la que se esté(coste
taxi3)

    //Construimos un Grafo gordo.
    GrafoP<tCoste> A(len);      //Inicializa a infinito todo

    for(int i = 0 ; i < n ; i++){
    //Para los costes de transbordo:
        A[n+i][i] = A[i][n+i] = taxi1_2;    //Transbordo entre medios 1 y 2
        A[2*n+i][i] = A[i][2*n+1] = A[2*n+i][n+i] = A[n+i][2*n+i] = taxi3;  //Costes
3-1,1-3,3-2,2-3

        //Para los costes de cada medio de transporte
        for(int j = 0 ; j < n ; j++){
            A[i][j] = G1[i][j];
            A[n+i][n+j] = G2[i][j];
            A[2*n+i][2*n+j]= G3[i][j];
        }
    }

    //Ahora que tengo el grafo inicializado, calculo los costes mínimos de ir desde
origen1,origen2, origen3 a cualquiera
    vector<vertice> verDij1(len),verDij2(len),verDij3(len);
    vector<tCoste> vecDij1(len),vecDij2(len),vecDij3(len);

    vecDij1 = Dijkstra(A,origen,verDij1);
    vecDij2 = Dijkstra(A,origen+n,verDij2);
    vecDij3 = Dijkstra(A,2*n+origen,verDij3);

    //Busco primero cual es el coste mínimo partiendo de cada origen:
    tCoste costemin1 = min(vecDij1[destino],min(vecDij1[destino+n],vecDij1[destino+2*n]));
    tCoste costemin2 = min(vecDij2[destino],min(vecDij2[destino+n],vecDij2[destino+2*n]));
    tCoste costemin3 = min(vecDij3[destino],min(vecDij3[destino+n],vecDij3[destino+2*n]));

    //Calculamos el mínimo
    tCoste costemin = min(costemin1,min(costemin2,costemin3));

    //Calculamos la ruta con una función auxiliar
    if(costemin==costemin1) ruta = calcularRuta(costemin, origen, destino, n, vecDij1,
verDij1);
    else if(costemin==costemin2)    ruta = calcularRuta(costemin, origen+n, destino, n,
vecDij2, verDij2);
    else    ruta = calcularRuta(costemin,origen+2*n, destino, n, vecDij3, verDij3);

    //Ahora dejamos la ciudad en su valor original. En el tCamino aparecerá 2 veces una
misma ciudad si es que se ha hecho transbordo
    typename GrafoP<tCoste>::tCamino::posicion p = ruta.primera();
    while(p != ruta.fin()){
        ruta.elemento(p) = ruta.elemento(p) % n;
        p = ruta.siguiente(p);
```

```
    }

    return costemin;
}

template <typename tCoste>
typename GrafoP<tCoste>::tCamino calcularRuta(tCoste costemin, typename
GrafoP<tCoste>::vertice origen,
                                              typename GrafoP<tCoste>::vertice destino,
size_t n, vector<tCoste> vecDij,
                                              vector<typename GrafoP<tCoste>::vertice>
verDij)
{
    typename GrafoP<tCoste>::tCamino ruta;

    if(vecDij[destino] < vecDij[destino+n]){
        if(vecDij[destino] < vecDij[2*n+destino]){
            ruta = camino<tCoste>(origen, destino, verDij);
        }else{
            ruta = camino<tCoste>(origen, destino+2*n, verDij);
        }
    }else{  //vecDij[destino+n] < vecDij[destino]
        if(vecDij[destino+n] < vecDij[2*n+destino]){
            ruta = camino<tCoste>(origen, destino+n, verDij);
        }else{
            ruta = camino<tCoste>(origen, destino+2*n, verDij);
        }
    }
    return ruta;
}
```

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

//Guarda las coordenadas del puente( conecta c1 con c2 y c2 con c1)
struct puente{
    puente() = default;
    puente(size_t c1, size_t c2):c1(c1),c2(c2){}
    size_t c1,c2;
};


template <typename tCoste>
matriz<tCoste> CalcularCostesEj11(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,
const GrafoP<tCoste>& G3,
                                  vector<puente> puentes);



/*

Ej11_1.dat

5
0           2147483647  10          2147483647  22
2147483647  0           22          2147483647  2147483647
40          30          0           30          2147483647
2147483647  22          2147483647  0           22
2147483647  2147483647  56          2147483647  0


Ej11_2.dat

5
0           2147483647  2147483647  2147483647  2
2147483647  0        9           2147483647  5
10          10          0           30          40
2147483647  2147483647  2147483647  0           100
62          2147483647  2147483647  2147483647  0


Ej11_3.dat

6
0           20          10          100         20          44
21          0           80          2147483647  5           23
2147483647  2147483647  0           2147483647  40          2147483647
2147483647  65          77          0           2147483647  2147483647
21          2147483647  2147483647  22          0           23
2147483647  2147483647  67          2147483647  55          0


*/

int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej11_1.dat"),G2("Ej11_2.dat"),G3("Ej11_3.dat");
    cout<<G1<<G2<<G3;
    vector<puente> puentes = { {0,5} , {5,10} , {10,0}};

    matriz<int> costes = CalcularCostesEj11(G1,G2,G3,puentes);

    cout<<"\nLos costes son: "<<endl;
    cout<<costes;
}

template <typename tCoste>
matriz<tCoste> CalcularCostesEj11(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,
const GrafoP<tCoste>& G3,
                                  vector<puente> puentes)
```

```
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t N1 = G1.numVert(), N2 = G2.numVert(), N3 = G3.numVert();
    size_t len = N1+N2+N3;

    //Creamos un grafo que englobe todo
    GrafoP<tCoste> A(len);  //Al principio todo a INFINITO

    for(int i = 0 ; i < N1 ; i++)
        for(int j = 0 ; j < N1 ; j++) A[i][j] = G1[i][j];

    for(int i = 0 ; i < N2 ; i++)
        for(int j = 0 ; j < N2 ; j++) A[i+N1][j+N1] = G2[i][j];

    for(int i = 0 ; i < N3 ; i++)
        for(int j = 0 ; j < N3 ; j++) A[i+N1+N2][j+N1+N2] = G3[i][j];

    for(int i = 0 ; i < puentes.size() ; i++) A[puentes[i].c1][puentes[i].c2] =
A[puentes[i].c2][puentes[i].c1] = 0;

    matriz<vertice> verFloyd;
    matriz<tCoste> Mfloyd = Floyd(A,verFloyd);

    return Mfloyd;


}
```

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


//Guarda las coordenadas del puente( conecta c1 con c2 y c2 con c1)
struct puente{
    puente() = default;
    puente(size_t c1, size_t c2):c1(c1),c2(c2){}
    size_t c1,c2;
};



template <typename tCoste>
puente CalcularPuenteTercerIntento(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,
vector<typename GrafoP<tCoste>::vertice>& costerasG1,
                    vector<typename GrafoP<tCoste>::vertice>& costerasG2);


/*

Ej11_1.dat

5
0           2147483647  10          2147483647  22
2147483647  0           22          2147483647  2147483647
40          30          0           30          2147483647
2147483647  22          2147483647  0           22
2147483647  2147483647  56          2147483647  0


Ej11_2.dat

5
0           2147483647  2147483647  2147483647  2
2147483647  0        9           2147483647  5
10          10          0           30          40
2147483647  2147483647  2147483647  0           100
62          2147483647  2147483647  2147483647  0


Ej11_3.dat

6
0           20          10          100         20          44
21          0           80          2147483647  5           23
2147483647  2147483647  0           2147483647  40          2147483647
2147483647  65          77          0           2147483647  2147483647
21          2147483647  2147483647  22          0           23
2147483647  2147483647  67          2147483647  55          0


*/


int main(){

    typedef typename GrafoP<int>::vertice vertice;

    GrafoP<int> G1("Ej11_1.dat"),G2("Ej11_3.dat");
    cout<<G1<<G2<<endl;

    vector<vertice> costerasG1 = {0,1}, costerasG2 = {1,2};
    //cout<<"VECTORES: "<<costerasG1<<" Y "<<costerasG2<<endl;

    puente puent = CalcularPuenteTercerIntento(G1,G2,costerasG1,costerasG2);

    //cout<<"El puente va de "<<puent.c1<<" del archipielago1 a "<<puent.c2<<" del
archipielago2"<<endl;
```

```
}


template <typename tCoste>
puente CalcularPuenteTercerIntento(const GrafoP<tCoste>& G1, const GrafoP<tCoste>& G2,
vector<typename GrafoP<tCoste>::vertice>& costerasG1,
                    vector<typename GrafoP<tCoste>::vertice>& costerasG2)
{

    typedef typename GrafoP<tCoste>::vertice vertice;
    size_t N1 = G1.numVert(), N2 = G2.numVert();

    matriz<vertice> m1,m2;
    matriz<tCoste> Mfloyd1,Mfloyd2;

    Mfloyd1 = Floyd(G1,m1);
    Mfloyd2 = Floyd(G2,m2);

    vertice c,c1,c2;
    tCoste costemin = GrafoP<tCoste>::INFINITO,costeiter;

    for(int i = 0 ; i < costerasG1.size() ; i++){
        c = costerasG1[i];
        costeiter=0;
        for(int j = 0 ; j < N1 ; j++)costeiter=suma(costeiter,Mfloyd1[c][j]);

        if(costeiter<costemin){
            costemin=costeiter;
            c1=c;
        }
    }

    costemin = GrafoP<tCoste>::INFINITO;
    for(int i = 0 ; i < costerasG2.size() ; i++){
        c = costerasG2[i];
        costeiter=0;
        for(int j = 0 ; j < N2 ; j++)costeiter=suma(costeiter,Mfloyd2[c][j]);

        if(costeiter<costemin){
            costemin=costeiter;
            c2=c;
        }
    }

    cout<<"Va de "<<c1<<" en arch1 a "<<c2<<" en arch2"<<endl;
    return puente();
}
```