Todas-las-practicas-Grafos-con-i...

Warabaringo

Estructuras de Datos no Lineales

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

Grafos EDNL

Warabaringo

Índice

Práctica 6 **3**
Ejercicio 1 . . . 3
Ejercicio 2 . . . 6
Ejercicio 3 . . . 8
Ejercicio 4 . . . 10
Ejercicio 5 . . . 12

Práctica 7 **14**
Ejercicio 1 . . . 14
Ejercicio 2 . . . 16
Ejercicio 3 . . . 19
Ejercicio 4 . . . 22
Ejercicio 5 . . . 24
Ejercicio 6 . . . 26
Ejercicio 7 . . . 28
Ejercicio 8 . . . 30
Ejercicio 9 . . . 32
Ejercicio 10 . . . 35
Ejercicio 11 . . . 39
Ejercicio 12 . . . 41
Ejercicio 13 . . . 43

Práctica 8 **45**
Ejercicio 1 . . . 45
Ejercicio 2 . . . 47
Ejercicio 3 . . . 51
Ejercicio 4 . . . 52
Ejercicio 5 . . . 53
Ejercicio 6 . . . 54

Ejercicio 7 56

Práctica 6

Ejercicio 1

Añadir una función genérica, llamada DijkstraInv, en el fichero alg_grafoPMC.h para resolver el problema inverso al de Dijkstra, con los mismos tipos de parámetros y de resultado que la función ya incluida para éste. La nueva función, por tanto, debe hallar el camino de coste mínimo hasta un destino desde cada vértice del grafo y su correspondiente coste.

```
1  template <typename tCoste>
2  vector<tCoste> Dijkstra_inv(const GrafoP<tCoste>& G,
3                      typename GrafoP<tCoste>::vertice
                        ↪  destino,
4                      vector<typename
                        ↪  GrafoP<tCoste>::vertice>& P)
5  // Calcula los caminos de coste mínimo desde todos los
    ↪  vértices
6  // del grafo G hasta destino. En el vector D de tamaño
    ↪  G.numVert()
7  // devuelve estos costes mínimos y P es un vector de tamaño
8  // G.numVert() tal que P[i] es el vértice siguiente a i
    ↪  para
9  // llegar a destino en coste mínimo.
10 {
11 typedef typename GrafoP<tCoste>::vertice vertice;
12 vertice v, w;
13 const size_t n = G.numVert();
14 vector<bool> S(n, false);                  // Conjunto de
    ↪  vértices vacío.
15 vector<tCoste> D(n);                        // Costes
    ↪  mínimos desde destino.
16
17 // Iniciar D y P con caminos directos hasta el vértice
    ↪  destino.
18 for(size_t j = 0; j < n; j++)
19   D[j] = G[j][destino];
20 D[destino] = 0;                             // Coste
    ↪  destino-destino es 0.
21 P = vector<vertice>(n, destino);
22
```

```
23  // Calcular caminos de coste mínimo hasta cada vértice.
24  S[destino] = true;                          // Incluir
    ↪  vértice destino en S.
25  for (size_t i = 1; i <= n-2; i++)
26  {
27    // Seleccionar vértice w no incluido en S
28    // con menor coste desde destino.
29    tCoste costeMin = GrafoP<tCoste>::INFINITO;
30    for (v = 0; v < n; v++)
31       if (!S[v] && D[v] <= costeMin)
32       {
33          costeMin = D[v];
34          w = v;
35       }
36    S[w] = true;                             // Incluir vértice w
      ↪  en S.
37    // Recalcular coste hasta cada v no incluido en S a
      ↪  través de w.
38    for (v = 0; v < n; v++)
39       if (!S[v])
40       {
41          tCoste vwD = suma(G[v][w], D[w]); // Coste desde v a
            ↪  Destino pasando por w
42          if (vwD < D[v])
43          {
44             D[v] = vwD;
45             P[v] = w;
46          }
47       }
48  }
49  return D;
50  }
51
52  template <typename tCoste> typename GrafoP<tCoste>::tCamino
53  camino_inv(typename GrafoP<tCoste>::vertice destino,
54         typename GrafoP<tCoste>::vertice v,
55         const vector<typename GrafoP<tCoste>::vertice>& P)
56  // Devuelve el camino de coste mínimo entre los vértices v
    ↪  y destino
57  // a partir de un vector P obtenido mediante la función
    ↪  Dijkstra_inv().
```

```
58  {
59     typename GrafoP<tCoste>::tCamino C;
60
61     C.insertar(v, C.fin());
62     do
63     {
64        C.insertar(P[v], C.fin());
65        v = P[v];
66     } while (v != destino);
67     return C;
68  }
```

Gemini

Lleva tu estudio al siguiente nivel con Gemini, tu asistente de IA de Google ✨

¡Pruébalo ahora!

Ejercicio 2

Definiremos el pseudocentro de un grafo conexo como el nodo del mismo que minimiza la suma de las distancias mínimas a sus dos nodos más alejados. Definiremos el diámetro del grafo como la suma de las distancias mínimas a los dos nodos más alejados del pseudocentro del grafo.
Dado un grafo conexo representado mediante matriz de costes, implementa un subprograma que devuelva la longitud de su diámetro.

```cpp
#ifndef DIAMETRO_HPP
#define DIAMETRO_HPP

#include "../Grafos/grafoPMC.h"
#include "../Grafos/alg_grafoPMC.h"

// Problema 2

// Pre: El grafo debe ser conexo
template <typename tCoste>
tCoste calcular_diametro(const GrafoP<tCoste> &G)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    matriz<vertice> P;
    vertice a, b;
    matriz<tCoste> costes_minimos = Floyd(G, P);
    tCoste primer_mayor = std::numeric_limits<tCoste>::min(), diametro =
    ↪  GrafoP<tCoste>::INFINITO, segundo_mayor = std::numeric_limits<tCoste>::min();
    size_t n = costes_minimos.dimension();
    vertice pseudocentro; // £Sobra?

    for(size_t i = 0; i <= n - 1; i++)
    {
        primer_mayor = segundo_mayor = std::numeric_limits<tCoste>::min();
        for(size_t j = 0; j <= n - 1; j++)
        {
            // Obtenemos los dos mayores
            if(costes_minimos[i][j] > primer_mayor)
            {
                segundo_mayor = primer_mayor;
                b = a;
                primer_mayor = costes_minimos[i][j];
                a = j;
            }
            else
                if(costes_minimos[i][j] > segundo_mayor){
                    segundo_mayor = costes_minimos[i][j];
                    b = j;
                }

        }
        // std::cout « "Diametro de " « i « " :"  « suma(primer_mayor, segundo_mayor) «
        ↪  std::endl;
        // Vamos buscando la suma mas pequeña
        if(suma(primer_mayor, segundo_mayor) < diametro)
        {
            pseudocentro = i;
            diametro = suma(primer_mayor, segundo_mayor);
        }
```

```
48      }
49
50      std::cout « "Los dos nodos mas alejados del pseudocentro son: " « a « " y " « b «
        ↪  std::endl;
51      std::cout « "El pseudocentro es: " « pseudocentro « std::endl;
52      // std::cout « "Con 1 es " « suma(costes_minimos[1][a], costes_minimos[1][b]) «
        ↪  std::endl;
53      return diametro;
54  }
55
56  #endif // DIAMETRO_HPP
```

Ejercicio 3

Tu empresa de transportes PEROTRAVEZUNGRAFO S.A. acaba de recibir la lista de posibles subvenciones del Ministerio de Fomento en la que una de las más jugosas se concede a las empresas cuyo grafo asociado a su matriz de costes sea acíclico. ¿Puedes pedir esta subvención?
Implementa un subprograma que a partir de la matriz de costes nos indique si tu empresa tiene derecho a dicha subvención.

```
#ifndef PERO_OTRA_VEZ_HPP
#define PERO_OTRA_VEZ_HPP

#include "../Grafos/grafoPMC.h"
#include "../Grafos/matriz.h"

template <typename tCoste>
matriz<tCoste> Floyd_mod(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;                    // No modificamos la diagonal principal
      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>
bool es_aciclico(const GrafoP<tCoste> &G)
{
   matriz<typename GrafoP<tCoste>::vertice> P;

   matriz<tCoste> costes_minimos = Floyd_mod(G, P);

   std::cout « costes_minimos « std::endl;
```

```
50
51      bool aciclico = true;
52      for(size_t i = 0; (i <= costes_minimos.dimension() - 1) and aciclico; i++)
53         if(costes_minimos[i][i] != GrafoP<tCoste>::INFINITO)
54            aciclico = false;
55      return aciclico;
56   }
57
58   #endif // PERO_OTRA_VEZ_HPP
```

Ejercicio 4

Se necesita hacer un estudio de las distancias mínimas necesarias para viajar entre dos ciudades cualesquiera de un país llamado Zuelandia. El problema es sencillo pero hay que tener en cuenta unos pequeños detalles:

a) La orografía de Zuelandia es un poco especial, las carreteras son muy estrechas y por tanto solo permiten un sentido de la circulación.

b) Actualmente Zuelandia es un país en guerra. Y de hecho hay una serie de ciudades del país que han sido tomadas por los rebeldes, por lo que no pueden ser usadas para viajar.

c) Los rebeldes no sólo se han apoderado de ciertas ciudades del país, sino que también han cortado ciertas carreteras, (por lo que estas carreteras no pueden ser usadas).

d) Pero el gobierno no puede permanecer impasible ante la situación y ha exigido que absolutamente todos los viajes que se hagan por el país pasen por la capital del mismo, donde se harán los controles de seguridad pertinentes.

Dadas estas cuatro condiciones, se pide implementar un subprograma que dados

- El grafo (matriz de costes) de Zuelandia en situación normal,
- la relación de las ciudades tomadas por los rebeldes,
- la relación de las carreteras cortadas por los rebeldes
- y la capital de Zuelandia,

calcule la matriz de costes mínimos para viajar entre cualesquiera dos ciudades zuelandesas en esta situación.

```
1  #ifndef ZUELANDIA_HPP
2  #define ZUELANDIA_HPP
3
4  #include <vector>
5  #include <iostream>
6
7  #include "../Grafos/grafoPMC.h"
8  #include "../Grafos/alg_grafoPMC.h"
9
10 typedef double km;
11
12 typedef GrafoP<km>::vertice Ciudad;
```

```cpp

struct Carretera{
    Ciudad a;
    Ciudad b;
};

matriz<km> calcular_costes_zuelandia(const GrafoP<km> &situacion_normal, const std::vector<bool> &ciudades_tomadas,
                                     const std::vector<Carretera> &carreteras_cortadas, Ciudad capital)
{
    matriz<km> costes_minimos_zuelandia(situacion_normal.numVert(), GrafoP<km>::INFINITO);
    GrafoP<km> zuelandia_en_guerra(situacion_normal);

    for(size_t i = 0; i < ciudades_tomadas.size(); i++) // Aislamos las ciudades tomadas
        if(ciudades_tomadas[i])
            for(size_t j = 0; j < zuelandia_en_guerra.numVert(); j++)
                zuelandia_en_guerra[i][j] = zuelandia_en_guerra[j][i] = GrafoP<km>::INFINITO;

    for(size_t i = 0; i < carreteras_cortadas.size(); i++) // Cortamos las carreteras
    {
        zuelandia_en_guerra[carreteras_cortadas[i].a][carreteras_cortadas[i].b] =
        zuelandia_en_guerra[carreteras_cortadas[i].b][carreteras_cortadas[i].a] =
        GrafoP<km>::INFINITO;
    }
    std::vector<Ciudad> P; // requerido por Dijkstra
    std::vector<km> costes_hacia_capital = Dijkstra_inv(zuelandia_en_guerra, capital, P);
    std::vector<km> costes_desde_capital = Dijkstra(zuelandia_en_guerra, capital, P);

    for(Ciudad i = 0; i < zuelandia_en_guerra.numVert(); i++)
    {
        for(Ciudad j = 0; j < zuelandia_en_guerra.numVert(); j++)
        {
            if(i == j)
                costes_minimos_zuelandia[i][j] = 0;
            else
            {
                costes_minimos_zuelandia[i][j] = suma(costes_hacia_capital[i], costes_desde_capital[j]);
                costes_minimos_zuelandia[j][i] = suma(costes_hacia_capital[j], costes_desde_capital[i]);
                // std::cout << "Desde " << i << " hasta " << j << ": " << suma(costes_hacia_capital[i], costes_desde_capital[j]) << std::endl;
            }
        }
    }

    return costes_minimos_zuelandia;
}

#endif // ZUELANDIA_HPP
```

Ejercicio 5

Escribir una función genérica que implemente el algoritmo de Dijkstra usando un grafo ponderado representado mediante listas de adyacencia.

```
#ifndef DIJISKTRA_LISTA_HPP
#define DIJISKTRA_LISTA_HPP

#include "../Grafos/grafoPLA.h"

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_lista(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;
   typedef typename GrafoP<tCoste>::vertice_coste vertice_coste;
   vertice v, w;
   const size_t n = G.numVert();
   vector<bool> S(n, false);                  // Conjunto de vértices vacío.
   vector<tCoste> D(n, GrafoP<tCoste>::INFINITO);                          // Costes
   ↪  mínimos desde origen.

   // Iniciar D y P con caminos directos desde el vértice origen.
   for(typename Lista<vertice_coste>::posicion it = G.adyacentes(origen).primera(); it
   ↪  != G.adyacentes(origen).fin(); it = G.adyacentes(origen).siguiente(it))
    {
        vertice i = G.adyacentes(origen).elemento(it).v;
        D[i] = G.adyacentes(origen).elemento(it).c;
    }
   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.
```

```
54          for (v = 0; v < n; v++)
55             if (!S[v]) {
56                typename Lista<vertice_coste>::posicion it = G.adyacentes(w).primera();
57                while((it != G.adyacentes(w).fin()) and (G.adyacentes(w).elemento(it).v !=
                  ↪  v))
58                    it = G.adyacentes(w).siguiente(it);
59                tCoste sumando = ((it == G.adyacentes(w).fin()) ? GrafoP<tCoste>::INFINITO :
                  ↪  G.adyacentes(w).elemento(it).c);
60                tCoste Owv = suma(D[w], sumando);
61                if (Owv < D[v]) {
62                   D[v] = Owv;
63                   P[v] = w;
64                }
65             }
66       }
67       return D;
68    }
69
70    #endif // DIJISKTRA_LISTA_HPP
```

Práctica 7

Ejercicio 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

```cpp
#ifndef OTRA_VEZ_HPP
#define OTRA_VEZ_HPP

#include "../Grafos/grafoPMC.h"
#include "../Grafos/matriz.h"

/* PROBLEMA 1 PRACTICA 7 */

template <typename tCoste>
matriz<tCoste> Floyd_max(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]) and ikj != GrafoP<tCoste>::INFINITO) {
               A[i][j] = ikj;
               P[i][j] = k;
            }
         }
   return A;
```

```
42  }
43
44  template <typename tCoste>
45  using Ciudad = GrafoP<tCoste>::vertice;
46  template <typename tCoste>
47  tCoste otra_vez(const GrafoP<tCoste> &G, Ciudad<tCoste> &origen, Ciudad<tCoste>
    ↪  &destino)
48  {
49     matriz<typename GrafoP<tCoste>::vertice> P;
50     matriz<tCoste> costes_maximos = Floyd_max(G,P);
51
52     tCoste max = std::numeric_limits<tCoste>::min();
53
54     for(Ciudad<tCoste> v = 0; v <= G.numVert() - 1; v++)
55        for(Ciudad<tCoste> w = 0; w <= G.numVert() - 1; w++)
56           if((max < G[v][w]) and G[v][w] != GrafoP<tCoste>::INFINITO)
57           {
58              origen = v;
59              destino = w;
60              max = G[v][w];
61           }
62
63     return max;
64  }
65
66
67  #endif // OTRA_VEZ_HPP
```

Ejercicio 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.

```
#ifndef LABERINTO_HPP
#define LABERINTO_HPP

#include "../Grafos/grafoPMC.h"
#include "../Grafos/matriz.h"
#include "../Grafos/alg_grafoPMC.h"

/* PROBLEMA 2 PRACTICA 7 */


typedef int paso;
typedef typename GrafoP<paso>::vertice nodo;

struct Casilla
{
    int fila,columna;
};

struct Pared
{
    Casilla a, b;
};

Casilla nodo_to_casilla(nodo n, int N)
{
    // Casilla c;
    // c.fila = n / N;
    // c.columna = n % N;
    // return c;
    return Casilla{int{n}/N, int{n}%N};
}

nodo casilla_to_nodo(Casilla c, int N)
{
```

```
35      return c.fila * N + c.columna;
36  }
37
38  bool son_adyacentes(Casilla c1, Casilla c2)
39  {
40      return ((std::abs((c1.fila - c2.fila)) + std::abs((c1.columna - c2.columna))) == 1);
41  }
42
43  void rellenar_adyacentes(GrafoP<paso> &G, int N)
44  {
45      for(nodo i = 0; i <= G.numVert() - 1; i++)
46          for(nodo j = 0; j <= G.numVert() - 1; j++)
47          {
48              if(i == j)
49                  G[i][j] = 0;
50              else
51                  if(son_adyacentes(nodo_to_casilla(i,N), nodo_to_casilla(j, N)))
52                      G[i][j] = 1;
53                  // else // De por si ya estan a infinito
54          }
55
56  }
57
58  void construir_paredes(GrafoP<paso> &laberinto, const Lista<Pared> &lista_paredes, int
    ↪  N)
59  {
60      for(Lista<Pared>::posicion it = lista_paredes.primera(); it != lista_paredes.fin();
        ↪  it = lista_paredes.siguiente(it))
61      {
62          nodo nodo_a = casilla_to_nodo(lista_paredes.elemento(it).a, N),
63               nodo_b = casilla_to_nodo(lista_paredes.elemento(it).b, N);
64          laberinto[nodo_a][nodo_b] = laberinto[nodo_b][nodo_a] = GrafoP<paso>::INFINITO;
65      }
66  }
67
68  typedef Lista<Casilla> Camino;
69
70  Camino camino_casillas(nodo orig, nodo v, const vector<nodo>& P, int N)
71  {
72     Camino C;
73
74     C.insertar(nodo_to_casilla(v, N), C.primera());
75     do {
76        C.insertar(nodo_to_casilla(P[v], N), C.primera());
77        v = P[v];
78     } while (v != orig);
79     return C;
80  }
81
82  std::pair<Camino, paso> resolver_laberinto(int N, const Lista<Pared> &lista_paredes,
    ↪  Casilla entrada, Casilla salida)
83  {
84      int dimension = N*N;
85      GrafoP<paso> laberinto(dimension);
86      nodo nodo_entrada = casilla_to_nodo(entrada, dimension),
87           nodo_salida = casilla_to_nodo(salida, dimension);
88
89      rellenar_adyacentes(laberinto, dimension);
90      construir_paredes(laberinto, lista_paredes, dimension);
91
92      std::vector<nodo> P;
```

```
93      std::vector<paso> costes_minimos = Dijkstra(laberinto, nodo_entrada, P);
94
95      // Puedo usar la funcion camino y luego transformar camino_nodo a Camino (de
        ↪  casillas)
96      // typename GrafoP<paso>::tCamino camino_nodos = camino<paso>(nodo_entrada,
        ↪  nodo_salida, P);
97      // Pero mejor lo hago directamente con camino_casillas
98
99      return std::make_pair(camino_casillas(nodo_entrada, nodo_salida, P, dimension),
        ↪  costes_minimos[nodo_salida]);
100 }
101
102
103
104
105 #endif // LABERINTO_HPP
```

Ejercicio 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.

```
1 #ifndef SUBVENCION_HPP
2 #define SUBVENCION_HPP
3
4 #include "../Grafos/alg_grafoPMC.h"
5
6 #include <algorithm>
7
```

```cpp
/* PROBLEMA 3 PRACTICA 7 */

typedef double precio;

typedef GrafoP<precio>::vertice Ciudad;

typedef std::vector<precio> vector_cantidades;

typedef std::vector<double> vector_porcentajes;

GrafoP<precio> matriz_to_grafo(const matriz<precio> &M)
{
    GrafoP<precio> G(M.dimension());
    for(size_t i = 0; i <= G.numVert() - 1; i++)
        for(size_t j = 0; j <= G.numVert() - 1; j++)
            G[i][j] = M[i][j];
    return G;
}

void subvencionar(vector_cantidades &v, const vector_porcentajes &subvenciones)
{
    for(size_t i = 0; i <= v.size() - 1; i++)
        v[i] -= v[i] * (subvenciones[i] / 100);
}

// Si todas las ciudades estan llenas devolvera n
size_t seleccionar_ciudad_mas_barata_sin_llenar(const vector_cantidades &costes_minimos,
↪  const vector_cantidades &stock_ciudades)
{
    precio min = std::numeric_limits<precio>::max();
    size_t mas_barata = costes_minimos.size();
    for(size_t i = 0; i < costes_minimos.size(); i++)
        if((costes_minimos[i] < min) and (stock_ciudades[i] > 0))
        {
            min = costes_minimos[i];
            mas_barata = i;
        }
    return mas_barata;
}

precio calcular_cantidades_ciudades_aux(const vector_cantidades &costes_minimos,
↪  vector_cantidades &stock_ciudades,
                                        unsigned cantidad, vector_cantidades
                                        ↪  &cantidad_de_cada_ciudad, size_t N)
{
    bool tengo_cantidad = cantidad > 0, ciudades_llenas = false;
    precio total = 0;
    while(tengo_cantidad and !ciudades_llenas)
    {
        size_t indice_mas_barata =
        ↪  seleccionar_ciudad_mas_barata_sin_llenar(costes_minimos, stock_ciudades);
        if(indice_mas_barata == N)
            ciudades_llenas = true;
        else
        {
            if(cantidad < stock_ciudades[indice_mas_barata])
            {
                cantidad_de_cada_ciudad[indice_mas_barata] = cantidad;
                // stock_ciudades[indice_mas_barata] -= cantidad; // no hace falta porque
                ↪  terminamos aqui
                cantidad = 0; // realmente esto tampoco pero aporta claridad
```

Gemini

Lleva tu estudio al siguiente nivel con Gemini, tu asistente de IA de Google ✨

¡Pruébalo ahora!

```
64                  tengo_cantidad = false;
65              }
66              else
67              {
68                  cantidad_de_cada_ciudad[indice_mas_barata] =
                    ↪  stock_ciudades[indice_mas_barata];
69                  stock_ciudades[indice_mas_barata] = 0;
70                  cantidad -= stock_ciudades[indice_mas_barata];
71              }
72              total += costes_minimos[indice_mas_barata];
73          }
74      }
75
76      return total;
77  }
78
79  std::pair<vector_cantidades, precio> calcular_cantidades_ciudades
80                                      (Ciudad centro_produccion, unsigned cantidad, const
                                        ↪  matriz<precio> &costes_directos,
81                                       vector_cantidades stock_ciudades, const
                                         ↪  vector_porcentajes &subvenciones)
82  {
83      std::vector<Ciudad> P; // requerido por Dijkstra
84      vector_cantidades costes_minimos = Dijkstra(matriz_to_grafo(costes_directos),
        ↪  centro_produccion, P);
85      bool tengo_cantidad = cantidad > 0, ciudades_llenas = false;
86      size_t N = costes_minimos.size();
87
88      // Lo que nos pide el enunciado:
89      vector_cantidades cantidad_de_cada_ciudad(N, 0);
90
91      subvencionar(costes_minimos, subvenciones);
92
93      costes_minimos[centro_produccion] = GrafoP<precio>::INFINITO;
94      precio total = calcular_cantidades_ciudades_aux(costes_minimos, stock_ciudades,
        ↪  cantidad, cantidad_de_cada_ciudad, N);
95
96      return std::make_pair(cantidad_de_cada_ciudad, total);
97  }
98
99  #endif // SUBVENCION_HPP
```

Ejercicio 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.

```
#ifndef CEMENTOS_DE_ZUELANDIA_HPP
#define CEMENTOS_DE_ZUELANDIA_HPP

#include "../Grafos/alg_grafoPMC.h"

/* PROBLEMA 4 PRACTICA 7 */


typedef double km;

typedef GrafoP<km>::vertice Ciudad;

typedef size_t carga;

typedef std::vector<carga> vector_carga; // En cada indice estara el numero de carga de esa ciudad, en caso de que no haya carga en esa ciudad, habra un 0 en ese indice
                                         // Ejemplo: en el indice 3 hay 4 cargas === a la ciudad 3 hay que enviarle 4 cargas

km calcular_km_totales(const GrafoP<km> &costes_directos,const vector_carga &parte_diario, Ciudad capital)
```

```
{
    std::vector<Ciudad> P;
    std::vector<km> desde_capital = Dijkstra(costes_directos, capital, P);
    std::vector<km> hacia_capital = Dijkstra_inv(costes_directos, capital, P);
    km total = 0;

    for(size_t i = 0; i <= parte_diario.size() - 1; i++)
        total += (desde_capital[i] + hacia_capital[i]) * parte_diario[i];

    return total;
}


#endif // CEMENTOS_DE_ZUELANDIA_HPP
```

Ejercicio 5

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.

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.

```
#ifndef VIAJERO_ALERGICO_HPP
#define VIAJERO_ALERGICO_HPP

#include "../Grafos/alg_grafoPMC.h"

#include <string>

/* PROBLEMA 5 PRACTICA 7 */

/* Pregunta: Por que no puedo hacer dos dijikstra y voy escogiendo el menor */
/* Respuesta: Si hago eso estoy condenando al viaje a solo tomar o un transporte u otro
↪  */

/* Pregunta: Como seria si tengo que calcular las ciudades que puedo alcanzar, con la
↪  ruta que pueda pasar por mas ciudades */
/* Respuesta: No es contenido de EDNL */

typedef double dinero;


typedef GrafoP<dinero>::vertice Ciudad;

void rellenar_fusion(const GrafoP<dinero> &transporte1, const GrafoP<dinero>
↪  &transporte2, GrafoP<dinero> &fusion)
{
    for(Ciudad i = 0; i <= transporte1.numVert() - 1; i++)
        for(Ciudad j = 0; j <= transporte1.numVert() - 1; j++)
```

```
25              fusion[i][j] = std::min(transporte1[i][j], transporte2[i][j]);
26      }
27
28      void rellenar_ciudades_alcanzables(const std::vector<dinero> &costes_minimos,
        ↪  std::vector<bool> &ciudades_alcanzables, dinero presupuesto)
29      {
30          for(Ciudad i = 0; i <= costes_minimos.size() - 1; i++)
31              if(costes_minimos[i] <= presupuesto)
32                  ciudades_alcanzables[i] = true;
33      }
34
35      std::vector<bool> calcular_ciudades_alcanzables(const GrafoP<dinero> &carretera, const
        ↪  GrafoP<dinero> &tren, const GrafoP<dinero> &avion,
36                                                      const std::string &alergia, Ciudad
                                                        ↪  origen, dinero presupuesto)
37      {
38          GrafoP<dinero> fusion_minima(carretera.numVert()); // Podría haber sido otro Grafo
39          std::vector<bool> ciudades_alcanzables(carretera.numVert(), false);
40          std::vector<dinero> costes_minimos(carretera.numVert());
41          std::vector<Ciudad> P(carretera.numVert());
42
43          if(alergia == "avion")
44              rellenar_fusion(carretera, tren, fusion_minima);
45          else
46              if(alergia == "tren")
47                  rellenar_fusion(carretera, avion, fusion_minima);
48              else
49                  if(alergia == "carretera")
50                      rellenar_fusion(tren, avion, fusion_minima);
51
52          costes_minimos = Dijkstra(fusion_minima, origen, P);
53
54          rellenar_ciudades_alcanzables(costes_minimos, ciudades_alcanzables, presupuesto);
55
56          ciudades_alcanzables[origen] = true;
57
58          return ciudades_alcanzables;
59      }
60
61
62      #endif // VIAJERO_ALERGICO_HPP
```

Ejercicio 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.

```
#ifndef UNA_CIUDAD_CON_TRANSBORDO_HPP
#define UNA_CIUDAD_CON_TRANSBORDO_HPP

#include "../Grafos/alg_grafoPMC.h"

/* PROBLEMA 6 PRACTICA 7 */

typedef double dinero;

typedef typename GrafoP<dinero>::vertice Ciudad;

matriz<dinero> calcular_tarifa_minima_con_una_ciudad(const GrafoP<dinero> &autobus,
↪  const GrafoP<dinero> &tren, Ciudad transbordo)
{
    matriz<Ciudad> P(autobus.numVert());
    matriz<dinero> minimos_autobus = Floyd(autobus, P);
    matriz<dinero> minimos_tren = Floyd(tren, P);
    matriz<dinero> tarifa_minima(autobus.numVert());

    for(size_t i = 0; i <= tarifa_minima.dimension() - 1; i++)
        for(size_t j = 0; j <= tarifa_minima.dimension() - 1; j++)
        {
            dinero transbordo_autobus_tren = minimos_autobus[i][transbordo] +
            ↪  minimos_tren[transbordo][j];
            dinero transbordo_tren_autobus = minimos_tren[i][transbordo] +
            ↪  minimos_autobus[transbordo][j];
            // Minimo entre
            tarifa_minima[i][j] = std::min(std::min(minimos_autobus[i][j],
            ↪  minimos_tren[i][j]), /* ir en autobus, en tren */
                                           std::min(transbordo_autobus_tren,
                                           ↪  transbordo_tren_autobus)) /* transbordar
                                           ↪  de autobus a tren o viceversa */;
        }

    return tarifa_minima;
}

```

```
32 #endif // UNA_CIUDAD_CON_TRANSBORDO_HPP
```

Ejercicio 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.

```
1  #ifndef DOS_CAMBIOS_HPP
2  #define DOS_CAMBIOS_HPP
3
4  /* PROBLEMA 7 PRACTICA 7 */
5
6  #include "../Grafos/alg_grafoPMC.h"
7
8  typedef double coste;
9
10 typedef GrafoP<coste>::vertice Ciudad;
11
12 typedef std::vector<Ciudad> vector_ciudades;
13
14 typedef Lista<Ciudad> Camino;
15
16
17 // Para concatenar
18 Camino operator+(const Camino &C1, const Camino &C2)
19 {
20     Camino res = C1;
21     res += C2;
22     return res;
23 }
24
25 std::pair<coste, Camino> calcular_ruta_coste_minimo_dos_cambios(Ciudad origen, Ciudad
   ↪  destino, Ciudad cambio1, Ciudad cambio2,
26                                                                 const GrafoP<coste>
                                                                   ↪  &tren, const
                                                                   ↪  GrafoP<coste>
                                                                   ↪  &autobus)
27 {
28     vector_ciudades parte1, parte2;
```

```
29      Camino ruta;
30      std::vector<coste> desde_origen = Dijkstra(tren, origen, parte1),
31                         hacia_destino = Dijkstra_inv(autobus, destino, parte2);
32      coste coste_cambio1 = suma(desde_origen[cambio1], hacia_destino[cambio1]),
33            coste_cambio2 = suma(desde_origen[cambio2], hacia_destino[cambio2]),
34            total = 0;
35
36      if(coste_cambio1 <= coste_cambio2)
37      {
38          ruta = camino<coste>(origen, cambio1, parte1) + camino_inv<coste>(destino,
            ↪  cambio1, parte2);
39          total = coste_cambio1;
40      }
41      else
42      {
43          ruta = camino<coste>(origen, cambio2, parte1) + camino_inv<coste>(destino,
            ↪  cambio2, parte2);
44          total = coste_cambio2;
45      }
46
47      return std::make_pair(total, ruta);
48  }
49
50
51  #endif // DOS_CAMBIOS_HPP
```

Ejercicio 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.

```cpp
#ifndef UN_SOLO_TRANSBORDO_HPP
#define UN_SOLO_TRANSBORDO_HPP

/* PROBLEMA 8 PRACTICA 7 */

#include "../Grafos/alg_grafoPMC.h"

typedef double coste;

typedef GrafoP<coste>::vertice Ciudad;

coste calcular_tarifa_minima(const GrafoP<coste> &tren, const GrafoP<coste> &autobus,
↪  Ciudad origen, Ciudad destino)
{
    size_t N = tren.numVert();

    std::vector<Ciudad> P;

    std::vector<coste> desde_origen_tren = Dijkstra<coste>(tren, origen, P),
                       desde_origen_autobus = Dijkstra<coste>(autobus, origen, P),
                       hacia_destino_tren = Dijkstra_inv<coste>(tren, destino, P),
                       hacia_destino_autobus = Dijkstra_inv<coste>(autobus, destino, P);
                       ↪

    coste coste_min = std::min(desde_origen_autobus[destino],
    ↪  desde_origen_tren[destino]);

    for(Ciudad i = 0; i <= N - 1; i++)
    {
        coste transbordo_tren = suma(desde_origen_tren[i], hacia_destino_autobus[i]),
              transbordo_autobus = suma(desde_origen_autobus[i], hacia_destino_tren[i]);
        coste_min = std::min(coste_min, std::min(transbordo_tren, transbordo_autobus));
    }
```

```
31
32      return coste_min;
33  }
34
35  #endif // UN_SOLO_TRANSBORDO_HPP
```

Ejercicio 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.

```
#ifndef DOS_GRAFOS_TAXI_HPP
#define DOS_GRAFOS_TAXI_HPP

/* PROBLEMA 9 PRACTICA 7 */

/*  bus_bus     bus_tren
    -------------------------
    |           |           |
    |           |           |
    |           |           |
    -------------------------
    |           |           |
    |           |           |
    |           |           |
    -------------------------
    tren_bus    tren_tren
*/

#include "../Grafos/alg_grafoPMC.h"

#include <algorithm>
// #include <map>

typedef double coste;

typedef GrafoP<coste>::vertice Ciudad;

typedef Lista<Ciudad> Camino;

enum Cuadrante {primer_cuadrante, segundo_cuadrante, tercer_cuadrante,
↪  cuarto_cuadrante};

// N es N, es decir numVert() / 2
Cuadrante que_cuadrante_soy(Ciudad i, Ciudad j, size_t N)
{
    // Primer o segundo cuadrante
    if(i <= N - 1)
    {
        if(j <= N - 1)
            return primer_cuadrante;
        else
            return segundo_cuadrante;
    }
    else // Tercer o cuarto cuadrante
    {
```

```
45          if(j >= N)
46              return tercer_cuadrante;
47          else
48              return cuarto_cuadrante;
49      }
50  }
51
52  void rellenar_fusion(GrafoP<coste> &fusion, const GrafoP<coste> &bus, const
    ↪  GrafoP<coste> &tren, coste coste_taxi, size_t N)
53  {
54      for(Ciudad i = 0; i <= 2*N - 1; i++)
55          for(Ciudad j = 0; j <= 2*N - 1;j++)
56          {
57              Cuadrante actual = que_cuadrante_soy(i,j,N);
58              switch(actual)
59              {
60                  case primer_cuadrante:
61                      fusion[i][j] = bus[i][j];
62                      break;
63                  case segundo_cuadrante: // Hacen lo mismo
64                  case tercer_cuadrante:
65                      if((i % N) == (j % N))
66                          fusion[i][j] = coste_taxi;
67                      break;
68                  case cuarto_cuadrante:
69                      fusion[i][j] = tren[i][j];
70                      break;
71              }
72          }
73  }
74
75  std::pair<coste, Camino> calcular_camino_y_coste_con_taxi(const GrafoP<coste> &bus,
    ↪  const GrafoP<coste> &tren,
76                                                            Ciudad origen_bus, Ciudad
                                                              ↪  destino_bus, coste
                                                              ↪  coste_taxi)
77  {
78      // El origen y destino del bus es el mismo, el del tren hay que cambiarlo
79      size_t N = bus.numVert();
80      Ciudad origen_tren = origen_bus + N, destino_tren = destino_bus + N;
81      GrafoP<coste> fusion(2*N);
82
83      rellenar_fusion(fusion, bus, tren, coste_taxi, N);
84
85      std::vector<Ciudad> camino_desde_bus, camino_desde_tren;
86      std::vector<coste> desde_bus = Dijkstra(fusion, origen_bus, camino_desde_bus),
87                         desde_tren = Dijkstra(fusion, origen_tren, camino_desde_tren);
88
89      // coste coste_minimo = std::min(std::min(desde_bus[destino_bus],
        ↪  desde_bus[destino_tren]),
90      //                               std::min(desde_tren[destino_tren],
        ↪  desde_tren[destino_bus]));
91      coste coste_minimo = std::min({desde_bus[destino_bus], desde_bus[destino_tren],
92                                     desde_tren[destino_tren], desde_tren[destino_bus]});
93
94      Camino camino_minimo;
95
96
97      if(coste_minimo == desde_bus[destino_bus])
98          camino_minimo = camino<coste>(origen_bus, destino_bus, camino_desde_bus);
99      else
```

```
100         if(coste_minimo == desde_bus[destino_tren])
101             camino_minimo = camino<coste>(origen_bus, destino_tren, camino_desde_bus);
102         else
103             if(coste_minimo == desde_tren[destino_tren])
104                 camino_minimo = camino<coste>(origen_tren, destino_tren,
                    ↪  camino_desde_tren);
105             else
106                 camino_minimo = camino<coste>(origen_tren, destino_bus,
                    ↪  camino_desde_tren);
107
108     return std::make_pair(coste_minimo, camino_minimo);
109 }
110
111
112 #endif // DOS_GRAFOS_TAXI_HPP
```

Ejercicio 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.

```
#ifndef TRES_GRAFOS_TAXI_HPP
#define TRES_GRAFOS_TAXI_HPP

#include "../Grafos/alg_grafoPMC.h"

#include <algorithm>

/* PROBLEMA 10 PRACTICA 7 */

enum Cuadrante {primer_cuadrante, segundo_cuadrante, tercer_cuadrante,
                cuarto_cuadrante, quinto_cuadrante, sexto_cuadrante,
                septimo_cuadrante, octavo_cuadrante, noveno_cuadrante};

// enum Columna {izquierda, centro, derecha};

typedef double coste;

typedef GrafoP<coste>::vertice Ciudad;

typedef Lista<Ciudad> Camino;

/*      bus         tren         avion
        ------------------------------------
        |           |           |          |
bus     |    1      |    2      |    3     |
        |           |           |          |
        ------------------------------------
        |           |           |          |
tren    |    4      |    5      |    6     |
```

```
30          |           |           |           |
31          ---------------------------------------
32          |           |           |           |
33  avion   |     7     |     8     |     9     |
34          |           |           |           |
35          ---------------------------------------
36
37  */
38
39  Cuadrante que_cuadrante_soy(Ciudad i, Ciudad j, size_t N)
40  {
41      if(i <= N-1)
42      {
43          if(j <= N-1)
44              return primer_cuadrante;
45          else
46              if(N <= j  and j <= 2*N-1)
47                  return segundo_cuadrante;
48              else
49                  return tercer_cuadrante;
50      }
51      else
52      {
53
54          if(N <= i and i <= 2*N-1)
55          {
56              if(j <= N-1)
57                  return cuarto_cuadrante;
58              else
59                  if(N <= j and j <= 2*N-1)
60                      return quinto_cuadrante;
61                  else
62                      return sexto_cuadrante;
63          }
64          else
65          {
66
67              if(j <= N-1)
68                  return septimo_cuadrante;
69              else
70                  if(N <= j and j <= 2*N-1)
71                      return octavo_cuadrante;
72                  else
73                      return noveno_cuadrante;
74          }
75      }
76  }
77
78  bool es_diagonal(Ciudad i, Ciudad j, size_t N)
79  {
80      return (i % N) == (j % N);
81  }
82
83  void rellenar_fusion(const GrafoP<coste> &bus, const GrafoP<coste> &tren, const
    ↪  GrafoP<coste> &avion, GrafoP<coste> &fusion, size_t N,
84                       coste taxi_tren_bus, coste taxi_avion)
85  {
86      for(Ciudad i = 0; i <= 3*N - 1; i++)
87          for(Ciudad j = 0; j <= 3*N - 1; j++)
88          {
89              switch(que_cuadrante_soy(i, j, N))
```

```
90              {
91                  case primer_cuadrante:
92                      fusion[i][j] = bus[i][j];
93                      break;
94                  case segundo_cuadrante:
95                  case cuarto_cuadrante:
96                      if(es_diagonal(i,j,N))
97                          fusion[i][j] = taxi_tren_bus;
98                      break;
99                  case tercer_cuadrante: // Todos los aviones con tren-bus
100                 case sexto_cuadrante:
101                 case septimo_cuadrante:
102                 case octavo_cuadrante:
103                     if(es_diagonal(i,j,N))
104                         fusion[i][j] = taxi_avion;
105                     break;
106                 case quinto_cuadrante:
107                     fusion[i][j] = tren[i][j];
108             }
109         }
110 }
111
112
113 std::pair<coste, Camino> calcular_camino_y_coste_con_taxi(const GrafoP<coste> &bus,
    ↪  const GrafoP<coste> &tren, const GrafoP<coste> &avion,
114                                                         Ciudad origen, Ciudad destino,
                                                            ↪  coste taxi_tren_bus, coste
                                                            ↪  taxi_avion, size_t N)
115 {
116     GrafoP<coste> fusion(3*N);
117
118     // Ciudades origen y destino bus, tren y avion
119     Ciudad origen_bus = origen, origen_tren = origen + N, origen_avion = origen + 2*N,
120            destino_bus = destino, destino_tren = destino + N, destino_avion = destino +
               ↪  2*N;
121
122     rellenar_fusion(bus, tren, avion, fusion, N, taxi_tren_bus, taxi_avion);
123
124     // Calculamos desde cada transporte
125     std::vector<Ciudad> camino_desde_bus, camino_desde_tren, camino_desde_avion;
126     std::vector<coste> desde_bus = Dijkstra(fusion, origen_bus, camino_desde_bus),
127                        desde_tren = Dijkstra(fusion, origen_tren, camino_desde_tren),
128                        desde_avion = Dijkstra(fusion, origen_avion, camino_desde_avion);
129
130     // Asociamos costes partiendo de cada ciudad origen a ciudad destino
131     coste bus_bus = desde_bus[destino_bus], bus_tren = desde_bus[destino_tren],
    ↪  bus_avion = desde_bus[destino_avion],
132           tren_bus = desde_tren[destino_bus], tren_tren = desde_tren[destino_tren],
          ↪  tren_avion = desde_tren[destino_avion],
133           avion_bus = desde_avion[destino_bus], avion_tren = desde_avion[destino_tren],
          ↪  avion_avion = desde_avion[destino_avion];
134
135     coste minimo = std::min({bus_bus, bus_tren, bus_avion, tren_bus, tren_tren,
    ↪  tren_avion, avion_bus, avion_tren, avion_avion});
136
137     Camino camino_minimo;
138
139     // Calculamos camino minimo
140     if(minimo == bus_bus)
141         camino_minimo = camino<coste>(origen_bus, destino_bus, camino_desde_bus);
142     else if(minimo == bus_tren)
```

```
143             camino_minimo = camino<coste>(origen_bus, destino_tren, camino_desde_bus);
144         else if(minimo == bus_avion)
145             camino_minimo = camino<coste>(origen_bus, destino_avion, camino_desde_bus);
146         else if(minimo == tren_bus)
147             camino_minimo = camino<coste>(origen_tren, destino_bus, camino_desde_tren);
148         else if(minimo == tren_tren)
149             camino_minimo = camino<coste>(origen_tren, destino_tren, camino_desde_tren);
150         else if(minimo == tren_avion)
151             camino_minimo = camino<coste>(origen_tren, destino_avion, camino_desde_tren);
152         else if(minimo == avion_bus)
153             camino_minimo = camino<coste>(origen_avion, destino_bus, camino_desde_avion);
154         else if(minimo == avion_tren)
155             camino_minimo = camino<coste>(origen_avion, destino_tren, camino_desde_avion);
156         else if(minimo == avion_avion)
157             camino_minimo = camino<coste>(origen_avion, destino_avion, camino_desde_avion);
158 
159         return std::make_pair(minimo, camino_minimo);
160     }
161 
162     #endif // TRES_GRAFOS_TAXI_HPP
```

Ejercicio 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 N1 ciudades de la isla 1, del 0 al N1-1, las N2 ciudades de la isla 2, del N1 al N1+N2-1, y las N3 de la última, del N1+ N2 al N1+N2+ N3-1.
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!!!

```cpp
#ifndef TRES_ISLAS_PUENTES_HPP
#define TRES_ISLAS_PUENTES_HPP

/* PROBLEMA 11 PRACTICA 7 */

#include "../Grafos/alg_grafoPMC.h"
#include "../Grafos/matriz.h"

typedef double coste;

typedef GrafoP<coste>::vertice Ciudad;

// typedef std::pair<Ciudad, Ciudad> Puente;
struct Puente{
    Ciudad a, b;
};

enum Isla {Primera, Segunda, Tercera};

Isla determinar_isla(Ciudad i, Ciudad j, size_t N1, size_t N2, size_t N3)
{
    if(i <= N1-1 and j <= N1-1)
        return Primera;
    else
        if((N1 <= i and i <= N1+N2-1) and (N1 <= j and j <= N1+N2-1))
            return Segunda;
        else
            if((N1+N2 <= i and i <= N3-1) and (N1+N2 <= j and j <= N3-1))
                return Tercera;
}

GrafoP<coste> crear_fusion(const GrafoP<coste> &Isla1, const GrafoP<coste> &Isla2, const
↪  GrafoP<coste> &Isla3, size_t N1, size_t N2, size_t N3)
{
    size_t dimension = N1+N2+N3;
    GrafoP<coste> fusion(dimension);
```

```
36
37      for(Ciudad i = 0; i <= dimension-1; i++)
38          for(Ciudad j = 0; j <= dimension-1; j++)
39              {
40                  switch(determinar_isla(i,j,N1,N2,N3))
41                  {
42                      case Primera:
43                          fusion[i][j] = Isla1[i][j];
44                          break;
45                      case Segunda:
46                          fusion[i][j] = Isla2[i - N1][j - N1];
47                          break;
48                      case Tercera:
49                          fusion[i][j] = Isla3[i - N1 - N2][j - N1 - N2];
50                          break;
51                  }
52              }
53      return fusion;
54  }
55
56  void construir_puentes(GrafoP<coste> &G, const std::vector<Puente> &lista_puentes)
57  {
58      for(size_t i = 0; i <= lista_puentes.size() - 1; i++)
59      {
60          Puente puente = lista_puentes[i];
61          G[puente.a][puente.b] = G[puente.b][puente.a] = 0;
62      }
63  }
64
65  matriz<coste> calcular_coste_minimo_con_puentes(const GrafoP<coste> &Isla1, const
    ↪  GrafoP<coste> &Isla2, const GrafoP<coste> &Isla3,
66                                                  const std::vector<Puente>
                                                    ↪  &lista_puentes)
67  {
68      size_t N1 = Isla1.numVert(), N2 = Isla2.numVert(), N3 = Isla3.numVert();
69      GrafoP<coste> fusion = crear_fusion(Isla1, Isla2, Isla3, N1, N2, N3);
70      construir_puentes(fusion, lista_puentes);
71      matriz<Ciudad> P;
72      return Floyd(fusion, P);
73  }
74
75  #endif // TRES_ISLAS_PUENTES_HPP
```

Ejercicio 12

El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen N1 y N2 ciudades, respectivamente, de las cuales C1 y C2 ciudades son costeras (obviamente $C1 \leqslant N1$ y $C2 \leqslant N2$). 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.

```cpp
#ifndef EJ12_P7_HPP
#define EJ12_P7_HPP

#include "../Grafos/alg_grafoPMC.h"

/* PROBLEMA 12 PRACTICA 7 */

typedef double coste;

typedef GrafoP<coste>::vertice Ciudad;

coste calcular_coste_total(const std::vector<coste> &costes)
{
    coste total = 0;
    for(size_t i = 0; i <= costes.size() - 1; i++)
        total = suma(total, costes[i]);
    return total;
}

Ciudad calcular_ciudad_costera_mas_barata(const GrafoP<coste> &Isla, const
↪  std::vector<Ciudad> &costa)
{
    coste min = GrafoP<coste>::INFINITO;
    std::vector<Ciudad> P;
```

```
24      Ciudad ciudad_mas_barata;
25      for(size_t i = 0; i <= costa.size() - 1; i++)
26      {
27          coste total = calcular_coste_total(Dijkstra(Isla, i, P)); // Calculo el total de
            ↪  ir de la ciudad costera i a cada una de las demas ciudades
28          if(min <= total)
29          {
30              min = total;
31              ciudad_mas_barata = i;
32          }
33      }
34      return ciudad_mas_barata;
35  }
36
37  std::pair<Ciudad, Ciudad> calcular_ciudades_con_puentes(const GrafoP<coste> &Fobos,
    ↪  const GrafoP<coste> &Deimos,
38                                                          const std::vector<Ciudad>
                                                            ↪  &costa_fobos, const
                                                            ↪  std::vector<Ciudad>
                                                            ↪  &costa_deimos)
39  {
40      Ciudad ciudad_fobos = calcular_ciudad_costera_mas_barata(Fobos, costa_fobos),
41             ciudad_deimos = calcular_ciudad_costera_mas_barata(Deimos, costa_deimos);
42
43      return std::make_pair(ciudad_fobos, ciudad_deimos);
44  }
45
46  #endif // EJ12_P7_HPP
```

Ejercicio 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 N1, N2 y N3 ciudades, de las cuales C1, C2 y C3 son costeras (obviamente $C1 \leqslant N1$, $C2 \leqslant N2$ y $C3 \leqslant N3$) . 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.

```
1  #ifndef EJ13_P7_HPP
2  #define EJ13_P7_HPP
3
4  #include "../Grafos/alg_grafoPMC.h"
5  #include "Ej12P7.hpp" // Vamos a reutilizar la funcion para calcular la ciudad mas
   ↪  barata
6
7  typedef double coste;
8
9  typedef GrafoP<coste>::vertice Ciudad;
10
11 struct Puente
12 {
13     Ciudad a, b;
14 };
15
```

```cpp
std::pair<Puente, Puente> calcular_puentes_tres_islas(const GrafoP<coste> &Isla1, const GrafoP<coste> &Isla2, const GrafoP<coste> &Isla3,
                                                      const std::vector<Ciudad> &costa1, const std::vector<Ciudad> &costa2, const std::vector<Ciudad> &costa3)
{
    Ciudad ciudad_isla1 = calcular_ciudad_costera_mas_barata(Isla1, costa1),
           ciudad_isla2 = calcular_ciudad_costera_mas_barata(Isla2, costa2),
           ciudad_isla3 = calcular_ciudad_costera_mas_barata(Isla3, costa3);

    Puente puente12 = Puente{ciudad_isla1, ciudad_isla2},
           puente13 = Puente{ciudad_isla1, ciudad_isla3};

    return std::make_pair(puente12, puente13);
}

#endif // EJ13_P7_HPP
```

Práctica 8

Ejercicio 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.

```
1  #ifndef EJ1_P8_HPP
2  #define EJ1_P8_HPP
3
4  /* TOMBUCTU */
5  /* Problema 1 Practica 8*/
6
7  #include "../Grafos/grafoMA.h"
8  #include "../Grafos/particion.h"
9  #include "../Grafos/matriz.h"
10 #include "../Grafos/alg_grafoPMC.h"
11
12 #include <cmath>
13 #include <limits>
14
15 struct coordenadas
16 {
17     double x, y;
18 };
19
20 Particion calcular_distribucion(const Grafo &Adyacencia)
21 {
```

```
22      Particion distribucion(Adyacencia.numVert());
23
24      for(size_t i = 0; i <= Adyacencia.numVert() - 1; i++)
25          for(size_t j = 0; j <= Adyacencia.numVert() - 1; j++)
26              if(Adyacencia[i][j])
27              {
28                  int a = distribucion.encontrar(i), b = distribucion.encontrar(j);
29                  if(a != b)
30                      distribucion.unir(a,b);
31              }
32
33      return distribucion;
34  }
35
36  double calcular_distancia(coordenadas ciudad1, coordenadas ciudad2)
37  {
38      return sqrt(pow(ciudad1.x - ciudad2.x, 2) + pow(ciudad1.y - ciudad2.y, 2));
39  }
40
41  GrafoP<double> calcular_distancias(const Particion &distribucion, const
    ↪  std::vector<coordenadas> &ciudades)
42  {
43      GrafoP<double> costes(ciudades.size());
44
45      for(size_t i = 0; i <= costes.numVert() - 1; i++)
46          for(size_t j = 0; j <= costes.numVert() - 1; j++)
47              if(distribucion.encontrar(i) == distribucion.encontrar(j))
48                  costes[i][j] = calcular_distancia(ciudades[i], ciudades[j]);
49
50      return costes;
51  }
52
53  std::pair<Particion, matriz<double>> calcular_distribucion_y_costes_minimos(const Grafo
    ↪  &Adyacencia, const std::vector<coordenadas> &ciudades)
54  {
55      Particion distribucion = calcular_distribucion(Adyacencia);
56      GrafoP<double> costes = calcular_distancias(distribucion, ciudades);
57      matriz<GrafoP<double>::vertice> P;
58      return std::make_pair(distribucion, Floyd(costes, P));
59  }
60
61  #endif // EJ1_P8_HPP
```

Ejercicio 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.

```
1  #ifndef EJ2_P8_HPP
2  #define EJ2_P8_HPP
3
4  /* TOMBUCTU2 */
5  /* Problema 2 Practica 8 */
6
7  #include "../Grafos/alg_grafoPMC.h"
8  #include "../Grafos/particion.h"
9  #include "../Grafos/alg_grafoMA.h"
10 #include "../Grafos/particion.h"
11 #include "../Grafos/apo.h"
12
```

```cpp
#include <cmath>

typedef GrafoP<double>::vertice Ciudad;


struct coordenadas
{
    double x, y;
};

struct Ciudad_distancia // Estructura para dos ciudades y su distancia
{
    Ciudad a,b;
    double distancia;
};

struct Linea
{
    Ciudad a, b;
};

size_t calcular_islas_y_distribucion(const Grafo &Adyacencia, Particion &distribucion)
{
    size_t n_islas = Adyacencia.numVert();

    for(size_t i = 0; i <= Adyacencia.numVert() - 1; i++)
        for(size_t j = 0; j <= Adyacencia.numVert() - 1; j++)
            if(Adyacencia[i][j])
            {
                int a = distribucion.encontrar(i), b = distribucion.encontrar(j);
                if(a != b)
                {
                    distribucion.unir(a,b);
                    n_islas--;
                }
            }

    return n_islas;
}

double calcular_distancia(coordenadas ciudad1, coordenadas ciudad2)
{
    return sqrt(pow(ciudad1.x - ciudad2.x, 2) + pow(ciudad1.y - ciudad2.y, 2));
}

matriz<double> calcular_costes_minimos(const std::vector<coordenadas> &ciudades)
{
    GrafoP<double> costes(ciudades.size());
    matriz<GrafoP<double>::vertice> P;

    for(size_t i = 0; i <= costes.numVert() - 1; i++)
        for(size_t j = 0; j <= costes.numVert() - 1; j++)
            costes[i][j] = calcular_distancia(ciudades[i], ciudades[j]);

    return Floyd(costes, P);
}

Apo<Ciudad_distancia> crear_apo_distancias(const matriz<double> &costes_minimos)
{
    size_t n_ciudades = costes_minimos.dimension();
    Apo<Ciudad_distancia> A((n_ciudades * (n_ciudades - 1)) / 2);
```

```
74
75      for(size_t i = 0; i <= n_ciudades - 1; i++)
76          for(size_t j = i + 1; j <= n_ciudades - 1; j++)
77          {
78              A.insertar({i, j, costes_minimos[i][j]});
79          }
80
81      return A;
82  }
83
84  bool existe_en_vector(const std::vector<int> v, int x)
85  {
86      size_t i = 0;
87      bool encontrado = false;
88      while(i < v.size() and !encontrado)
89          if(v[i] == x)
90              encontrado = true;
91      return encontrado;
92  }
93
94  std::vector<int> crear_vector_representantes_islas(const Particion &distribucion, size_t
    ↪ n_ciudades)
95  {
96      std::vector<int> representantes_islas;
97      for(size_t i = 0; i <= n_ciudades - 1; i++)
98      {
99          int rep = distribucion.encontrar(i);
100         if(!existe_en_vector(representantes_islas, rep))
101             representantes_islas.push_back(rep);
102     }
103     return representantes_islas;
104 }
105
106 size_t buscar(const std::vector<int> &v, int x)
107 {
108     bool encontrado = false;
109     size_t i = 0;
110     while((i <= v.size() - 1) and !encontrado)
111     {
112         if(v[i] == x)
113             encontrado = true;
114         else
115             i++;
116     }
117     return i;
118 }
119
120 std::vector<Linea> calcular_lineas_aeropuerto(const std::vector<coordenadas> ciudades,
    ↪ const Grafo &Adyacencia)
121 {
122     std::vector<Linea> aeropuertos;
123     matriz<double> costes_minimos = calcular_costes_minimos(ciudades);
124     Apo<Ciudad_distancia> A = crear_apo_distancias(costes_minimos);
125
126     Particion distribucion(ciudades.size());
127
128     size_t n_islas = calcular_islas_y_distribucion(Adyacencia, distribucion);
129     std::vector<int> representantes_islas =
    ↪ crear_vector_representantes_islas(distribucion, ciudades.size()); // Creamos un
    ↪ vector de n_islas, cada indice va a ser cada isla y su contenido el representante
    ↪ en la particion
```

```
130     matriz<bool> islas_conectadas(n_islas, false); // Matriz para las islas conectadas si
        ↪  es true, las islas estan conectadas
131     while(!A.vacio())
132     {
133         Ciudad_distancia ciudad_distancia_candidata = A.cima();
134         A.suprimir();
135         int rep_a = distribucion.encontrar(ciudad_distancia_candidata.a), rep_b =
            ↪  distribucion.encontrar(ciudad_distancia_candidata.b);
136         size_t isla_a = buscar(representantes_islas, rep_a), isla_b =
            ↪  buscar(representantes_islas, rep_b); // Buscar da el indice, es decir la
            ↪  isla
137         if(rep_a != rep_b and !islas_conectadas[isla_a][isla_b]) // Si estan en
            ↪  diferentes islas y ademas no estan ya unidas...
138         {
139             aeropuertos.push_back({ciudad_distancia_candidata.a,
                ↪  ciudad_distancia_candidata.b}); // Creo una linea aeropuerto, es la menor
                ↪  porque lo he sacado de un APO
140             islas_conectadas[isla_a][isla_b] = islas_conectadas[isla_b][isla_a] = true;
                ↪  // Y uno las islas
141         }
142     }
143     return aeropuertos;
144 }
145
146
147
148 #endif // EJ2_P8_HPP
```

Ejercicio 3

Implementa un subprograma para encontrar un árbol de extensión máximo. ¿Es más difícil que encontrar un árbol de extensión mínimo?

```cpp
#ifndef EJ3_P8_HPP
#define EJ3_P8_HPP

#include <cassert>

#include "../Grafos/grafoPMC.h"
#include "../Grafos/apo_inverso.h"
#include "../Grafos/particion.h"

/* Problema 3 Practica 8 */

// Con Prim seria igual

template <typename tCoste>
GrafoP<tCoste> Kruskall_inverso(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste maximo
// 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_inverso<arista> A(n*n);    // Aristas de G ordenadas por costes. Cambio Apo por
   ↪  Apo_inverso

   // 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 mayor 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 // EJ3_P8_HPP
```

Ejercicio 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.

```
1  #ifndef EJ4_P8_HPP
2  #define EJ4_P8_HPP
3
4  #include "Ej3P8.hpp" // Kruskall inverso
5
6  typedef double km;
7
8  typedef GrafoP<km>::vertice Ciudad;
9
10 GrafoP<km> calcular_canales_maximos(const GrafoP<km> &G)
11 {
12     return Kruskall_inverso(G);
13 }
14
15 #endif // EJ4_P8_HPP
```

Ejercicio 5

La nueva compañía de telefonía RETEUNI3 tiene que conectar entre sí, con fibra óptica, todas y cada una de las ciudades del país. Partiendo del grafo que representa las distancias entre todas las ciudades del mismo, implementad un subprograma que calcule la longitud mínima de fibra óptica necesaria para realizar dicha conexión.

```cpp
#ifndef EJ5_P8_HPP
#define EJ5_P8_HPP

#include "../Grafos/alg_grafoPMC.h"

typedef double km;

typedef GrafoP<km>::vertice Ciudad;

km calcular_longitud_minima_fibra(const GrafoP<km> &G)
{
    GrafoP<km> Grafo_min = Kruskall(G);
    km total = 0;

    // Recorremos la parte superior de la diagonal principal, si no tendriamos que
    ↪  dividir el resultado entre 2 y hacer el doble de interacciones
    for(size_t i = 0; i <= G.numVert() - 1; i++)
        for(size_t j = i + 1; j <= G.numVert() - 1; j++)
        total += G[i][j];

    return total;
}

#endif // EJ5_P8_HPP
```

Ejercicio 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

```cpp
#ifndef EJ6_P8_HPP
#define EJ6_P8_HPP

#include "../Grafos/alg_grafoPMC.h"
#include "../Grafos/matriz.h"

struct Canal
{
    double longitud;
    double caudal;
};

// Si en la matriz hay algun infinito en alguna casilla significa que ese canal no se contruye
matriz<Canal> calcular_canales_emasajer(const matriz<double> &distancias, const matriz<double> &caudales,
                                        double subvencion_caudal, double coste_longitud)
{
    double infinito = GrafoP<double>::INFINITO;
    size_t N = distancias.dimension();
    GrafoP<double> Precios(N);

    for(size_t i = 0; i <= N - 1; i++)
        for(size_t j = 0; j <= N - 1; j++)
            Precios[i][j] = distancias[i][j] * coste_longitud - distancias[i][j] * coste_longitud; // Lo que nos cuesta menos lo que nos dan

    GrafoP<double> Precios_minimos = Kruskall(Precios);
    matriz<Canal> Canales(N, {infinito, infinito});
```

```
27
28      for(size_t i = 0; i <= N - 1; i++)
29          for(size_t j = 0; j <= N - 1; j++)
30              if(Precios_minimos[i][j] != infinito) // Si fuese infinito se queda igual, a
                ↪  infinito longitud y canal
31                  Canales[i][j] = {distancias[i][j], caudales[i][j]};
32
33      return Canales;
34  }
35
36
37  #endif // EJ6_P8_HPP
```

Ejercicio 7

El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen N1 y N2 ciudades, respectivamente, de las cuales C1 y C2 ciudades son costeras (obviamente C1 N1 y C2 N2). 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.

```cpp
#ifndef EJ7_P8_HPP
#define EJ7_P8_HPP

#include "../Grafos/alg_grafoPMC.h"

#include <cmath>

typedef double km;

typedef GrafoP<km>::vertice Ciudad;

struct coordenadas
{
    double x, y;
};

enum Isla {FOBOS, DEIMOS};

km calcular_distancia_euclidea(coordenadas c1, coordenadas c2)
{
    return sqrt(pow((c1.x - c2.x), 2) + pow((c1.y - c2.y), 2));
}

GrafoP<km> vector_to_grafo_coordenadas(const std::vector<coordenadas> &v)
{
    size_t N = v.size();
    GrafoP<km> G(N);

    for(size_t i = 0; i <= N - 1; i++)
        for(size_t j = 0; j <= N - 1; j++)
            G[i][j] = calcular_distancia_euclidea(v[i], v[j]);

    return G;
}

GrafoP<km> crear_Grecolandia(const GrafoP<km> &Fobos_expansion, const GrafoP<km>
↪  &Deimos_expansion)
{
    size_t N1 = Fobos_expansion.numVert(), N2 = Deimos_expansion.numVert();
    GrafoP<km> Grecolandia(N1+N2);
    for(size_t i = 0; i <= N1 - 1; i++)
        for(size_t j = 0; j <= N1 - 1; j++)
            Grecolandia[i][j] = Fobos_expansion[i][j];

    for(size_t i = 0; i <= N2 - 1; i++)
        for(size_t j = 0; j <= N2 - 1; j++)
            Grecolandia[i + N1][j + N1] = Deimos_expansion[i][j];

   return Grecolandia;
}

km calcular_coste_segun_isla(const matriz<km> &M, Ciudad i, Ciudad j, Isla isla_origen,
↪  Isla isla_destino, size_t N1)
{
    double coste_origen_destino;
    if(isla_origen == FOBOS)
        if(isla_destino == DEIMOS)
            coste_origen_destino = M[i][j + N1];
        else
            coste_origen_destino = M[i][j];
```

```
59      else
60          if(isla_destino == DEIMOS)
61              coste_origen_destino = M[i + N1][j + N1];
62          else
63              coste_origen_destino = M[i + N1][j];
64      return coste_origen_destino;
65  }
66
67  km calcular_coste_minimo_Grecoland(const std::vector<coordenadas> &ciudades_fobos, const
    ↪  std::vector<coordenadas> &ciudades_deimos,
68                                     const std::vector<bool> &costeras_fobos, const
                                       ↪  std::vector<bool> &costeras_deimos, // Si es true
                                       ↪  costeras[i], entonces ciudades[i] es costera
69                                     Ciudad origen, Ciudad destino, Isla isla_origen, Isla
                                       ↪  isla_destino)
70  {
71      size_t N1 = ciudades_fobos.size(), N2 = ciudades_deimos.size();
72      GrafoP<km> Fobos = vector_to_grafo_coordenadas(ciudades_fobos), Deimos =
        ↪  vector_to_grafo_coordenadas(ciudades_deimos);
73      // Hace falta Kruskall?
74      GrafoP<km> Fobos_expansion = Kruskall(Fobos), Deimos_expansion = Kruskall(Deimos);
75      GrafoP<km> Grecolandia = crear_Grecolandia(Fobos_expansion, Deimos_expansion);
76
77      // Ahora simulamos construir cada puente y nos quedamos con el coste menor
78      km min = GrafoP<km>::INFINITO;
79      for(Ciudad i = 0; i <= N1 - 1; i++)
80          if(costeras_fobos[i])
81              for(Ciudad j = 0; j < N2 - 1; j++)
82              {
83              if(costeras_deimos[j])
84              {
85                  Grecolandia[i][j + N1] = Grecolandia[i + N1][j] =
                    ↪  calcular_distancia_euclidea(ciudades_fobos[i], ciudades_deimos[j]);
86                  matriz<Ciudad> P(N1+N2);
87                  matriz<km> costes_minimos = Floyd(Grecolandia, P);
88                  min = std::min(min, calcular_coste_segun_isla(costes_minimos, i, j,
                    ↪  isla_origen, isla_destino, N1));
89              }
90              Grecolandia[i][j + N1] = Grecolandia[i + N1][j] = GrafoP<km>::INFINITO; //
                ↪  Destruimos el puente
91              }
92
93      return min;
94  }
95
96  #endif // EJ7_P8_HPP
```