Ejercicios tipo Examen Estructura de Datos No Lineales

Índice general

1. Ejercicios Árboles

En este apartado vas a encontrar todos los ejercicios tipo examen sobre árboles binarios, generales, etc.

Árbol Binario Reflejado

Enunciado: *A partir de un árbol binario, creamos otro intercambiando los subárboles del mismo.*

```
#include <iostream>
#include "abin.h"

template <typename T>
Abin<T> AbinReflejado(const Abin<T> &A){
  //Creamos el nuevo árbol binario
  Abin<T> Reflejado;
  if(!A.arbolVacio()){
    Reflejado.insertarRaiz(A.elemento(A.raiz()));
    return AbinReflejado_rec(A.raiz(),Reflejado.raiz(),A,Reflejado);
  }
  return Reflejado; //devolvemos el árbol
}

template <typename T>
void AbinReflejado_rec(typename Abin<T>::nodo a, typename Abin<T>::nodo
↪  b, const Abin<T> &A, Abin<T> &B){
  if(a != Abin<T>::NODO_NULO){
    //Vamos a insertar en el subárbol izquierdo del nodo b de B el
    ↪  subárbol derecho del nodo a de A y viceversa.
    if(A.hijoDrcho(a)!=Abin<T>::NODO_NULO)
      B.insertarHijoIzqdo(b,A.elemento(A.hijoDrcho(a)));
    //Llamamos a la función recursiva con el subárbol derecho de A y
    ↪  izquierdo de B.
    AbinReflejado(A.hijoDrcho(a),A.hijoIzqdo(b),A,B);
    if(A.hijoIzqdo(a) != Abin<T>::NODO_NULO)
      B.insertarHijoDrcho(b,A.elemento(A.hijoIzqdo(a)));
    //Llamamos a la función con el subárbol izquierdo A y derecho de B.
    AbinReflejado(A.hijoIzqdo(a),B.hijoDrcho(b),A,B);
  }
  //Devolvemos por referencia el árbol B (Reflejado),
}
```

Para llevar a cabo este ejercicio hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `Abin<T> Abin();`
  - *Post*: Crea y devuelve un Abin vacío.
- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `void insertarRaiz(const T& e);`
  - *Pre*: Árbol vacío
  - *Post*: Inserta el elemento en la raíz del árbol.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `void insertarHijoIzqdo(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol y no tiene hijo izquierdo previo.
  - *Post*: Inserta el elemento en el hijo izquierdo de n.

- `void insertarHijoDrcho(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol y no tiene hijo derecho previo.
  - *Post*: Inserta el elemento en el hijo derecho de n.

Árbol General Reflejado

Enunciado: *A partir de un árbol general, creamos otro intercambiando los subárboles del mismo*

```
#include <iostream>
#include "agen.h"

template <typename T>
Agen<T> AgenReflejado(const Agen<T> &A){
  //Creamos el agen a devolver
  Agen<T> Reflejado;
  Reflejado.insertarRaiz(A.elemento(A.raiz()));
  //Si la raiz de A no tiene hijo izquierdo, es un árbol que solo contiene
  ↪  un nodo.
  if(A.hijoIzqdo(A.raiz()) != Agen<T>::NODO_NULO)
    return AgenReflejado_rec(A.raiz(),Reflejado.raiz(),A,Reflejado);
  return Reflejado; //devolvemos el árbol general Reflejado
}

template <typename T>
void AgenReflejado_rec(typename Agen<T>::nodo a, typename Agen<T>::nodo
↪  b, const Agen<T> &A, Agen<T> &B){
  if(a != Agen<T>::NODO_NULO){ //Comprobamos que a no sea un nodo nulo
    //Vamos a ir recorriendo los hijos de a (y sus nietos, ...) e
    ↪  insertando en el árbol B
    typename Agen<T>::nodo hijo = A.hijoIzqdo(a);
    while(hijo != Agen<T>::NODO_NULO){
      //Si insertamos un hijo izquierdo que ya existe, este se
      ↪  convertirá en hermano derecho automaticamente del nuevo hijo
      ↪  izquierdo que se inserta
      B.insertarHijoIzqdo(b,A.elemento(hijo));
      //Ahora acedemos al hijo de hijo
      AgenReflejado(hijo,b,A,B);
      hijo = A.hermDrcho(hijo); //ahora accedemos a los hermanos.
    }
  }
  //Devolvemos el árbol general Reflejado por referencia
}
```

Al igual que en el ejercicio con el árbol binario hemos hecho uso de la representación enlazada pero ahora del árbol general, ya que no tenemos un número de nodos predefinidos, siendo está la parte privada del TAD:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
}; //fin del TAD Agen
```

También hemos hecho uso de los siguientes métodos públicos del TAD Agen:

- `Agen<T> Agen();`
  - *Post*: Crea y devuelve un Agen vacío.
- `void insertarRaiz(const T& e);`
  - *Post*: Inserta el elemento en la raíz del árbol.
- `const T& elemento (nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.
- `void insertarHijoIzqdo(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Inserta el elemento en el hijo izquierdo de n, si existe, el anterior se convierte en su hermano derecho.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Árbol General Similar

Enunciado: *Árboles generales similares. Dos árboles generales son similares si tienen la misma estructura y el contenido de sus nodos hojas deben de ser el mismo.*

Para saber si son similares el árbol general tiene que cumplir dos cosas:

1. Tiene que tener la misma estructura de ramificación → el nodo a del árbol A y el nodo b del árbol B tienen el mismo número de hijos.
2. Los nodos hojas de ambos árboles tienen que tener el mismo contenido.

Vamos a craear un programa que nos compruebe ambas cosas.

```
#include <iostream>
#include "agen.h"

template <typename T>
bool Similares(const Agen<T> &A, const Agen<T> &B){
  //Si ambos tienen solamente un nodo (raiz), diremos que son similares
  if(A.raiz() == Agen<T>::NODO_NULO && B.raiz() == Agen<T>::NODO_NULO)
    return true;
  return Similares_rec(A.raiz(),B.raiz(),A,B);
}

//Función auxiliar que nos calcula el número de hijos que tiene un nodo
template <typename T>
size_t nHijos(typename Agen<T>::nodo n, const Agen<T> &A){
  size_t nhijos = 0;
  if (n == Agen<T>::NODO_NULO)
    return nhijos;
  else{
    //Vamos a recorrer toda la lista de hijos del nodo n
    typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
    while(hijo != Agen<T>::NODO_NULO){
      nhijos++;
      hijo = A.hermDrcho(hijo);
    }
    return nhijos;
  }
}
//Función que nos comprueba si el nodo es una hoja
template <typename T>
bool esHoja(typename Agen<T>::nodo n, const Agen<T> &A){
  return A.hijoIzqdo(n) == Agen<T>::NODO_NULO;
}

template <typename T>
```

```
bool Similares_rec(typename Agen<T>::nodo a, typename Agen<T>::nodo b,
↪  const Agen<T> &A; const Agen<T> &B){
  //Ahora tenemos que comprobar las diferentes condiciones
  if(a == Agen<T>::NODO_NULO && b == Agen<T>::NODO_NULO) return true;
  else if(a == Agen<T>::NODO_NULO || Agen<T>::NODO_NULO)
    return false; //si existe a pero no b, o viceversa no son similares.
  else if(esHoja(a,A) && esHoja(b,B))
    //Comprobamos el contenido de las hojas
    return A.elemento(a) == B.elemento(b);

  else{ //no son hojas, vamos a comprobar la ramificación
  //nos creamos una flag que nos devolerá si son similares o no
    bool flag = nHijos(a,A) == nHijos(b,B); //tiene los mismos hijos, true
    while(flag){ //mientra que sean similares, comprobamos los hijos de
     ↪  los hijos
      flag &= Similares_rec(A.hijoIzqdo(a),B.hijoIzqdo(b),A,B); //llamamos
       ↪  con sus hijos
      a = A.hermDrcho(a);
      b = B.hermDrcho(b);
    }
    return flag; //devolvemos si son similares o no.
  }
}
```

Como no tenemos un número determinados del árbol general, para la resolución de este ejercicio, hemos hecho uso de la representación enlazada del TAD agen, siendo su parte privada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //nodo raíz del árbol
};
```

También hemos hecho uso de los métodos públicos del TAD Agen:

- `const T& elemento (nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`

- *Pre*: El nodo n existe en el árbol.
- *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.

- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Tres Nietos

Enunciado: *Un nodo verde es aquel nodo que cumple una cierta codición, por ejemplo en este caso, nuestra condición es aquel que tiene estrictamente 3 nietos.*
Es decir, vamos a devolver el número de nodos que tienen 3 en el árbol.

Este ejercicio la vamos a hacer tanto para árboles binarios como árboles generales.

Tres Nietos Árboles Binarios

```
#include <iostream>
#include "abin.h"

template <typename T>
size_t ContarNodosVerdesAbin(const Abin<T> &A){
  if(A.arbolVacio()) return 0;
  return ContarNodosVerdesAbin_rec(A.raiz(),A);
}

//Para tener 3 nietos tiene que tener si o si dos hijos
template <typename T>
bool DosHijos(typename Abin<T>::nodo n, const Abin<T> &A){
  return (A.hijoIzqdo(n) != Abin<T>::NODO_NULO && A.hijoDrcho(n) !=
   ↪  Abin<T>::NODO_NULO);
}

template <typename T>
size_t TresNietos(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return 0;
  else{
    size_t nietos = 0;
    if(DosHijos(n,A)){
      //Vamos a recorrer los nietos de un nodo para ver si tiene 3
       ↪  (descendientes propios de él).
      if(A.hijoIzqdo(A.hijoIzqdo(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo izquierdo y nieto izquierdo
      if(A.hijoIzqdo(A.hijoDrcho(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo izquierdo y nieto derecho
      if(A.hijoDrcho(A.hijoIzqdo(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo derecho y nieto izquierdo
      if(A.hijoDrcho(A.hijoDrcho(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo derecho y nieto derecho
    }
    return nietos;
  }
}
```

```
template <typename T>
size_t ContarNodosVerdesAbin_rec(typename Abin<T>::nodo n, const Abin<T>
↪  &A){
  if(n == Abin<T>::NODO_NULO) return 0;
  else if(TresNietos(n,A) == 3) //condición a cumplir
      return 1 + ContarNodosVerdesAbin_rec(A.hijoIzqdo(n),A) +
       ↪  ContarNodosVerdesAbin_rec(A.hijoDrcho(n),A);
  else
    return 0 + ContarNodosVerdesAbin_rec(A.hijoIzqdo(n),A) +
     ↪  ContarNodosVerdesAbin_rec(A.hijoDrcho(n),A);
}
```

Hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.

Tres Nietos Árboles General

```cpp
#include <iostream>
#include "agen.h"

template <typename T>
size_t ContarNodosVerdesAgen(const Agen<T> &A){
  if(A.raiz() == Agen<T>::NODO_NULO) return 0;
  return ContarNodosVerdesAgen_rec(A.raiz(),A);
}

//Ahora para que un nodo pueda tener nietos, como mínimo tiene que tener
↪  un hijo izquierdo.

//Función para calcular el número de hijos que tiene un nodo, tenemos que
↪  realizar una búsqueda en anchura
template <typename T>
size_t TresNietosAgen(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  else{
    size_t nietos = 0;
    typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
    while(hijo != Agen<T>::NODO_NULO){
      //Vamos a recorrer los nietos en anchura
      typename Agen<T>::nodo nieto = A.hijoIzqdo(hijo);
      while(nieto != Agen<T>::NODO_NULO){
        nietos++;
        nieto = A.hermDrcho(nieto);
      }
      hijo = A.hermDrcho(hijo);
    }
    return nietos;
  }
}

template <typename T>
size_t ContarNodosVerdesAgen_rec(typename Agen<T>::nodo n, const Agen<T>
↪  &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  else{
    if(TresNietosAgen(n,A) == 3)//cumple la codición, se suma.
      return 1+ContarNodosVerdesAgen_rec(A.hijoIzqdo(n),A);
    else
      return 0+ContarNodosVerdesAgen_rec(A.hijoIzqdo(n),A);
  }
}
```

Para la resolución de este ejercicio, hemos hecho uso de la representación enlazada del TAD agen, siendo su parte privada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //nodo raíz del árbol
};
```

También hemos hecho uso de los métodos públicos del TAD Agen:

- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Flotar Nodos

Enunciado: *Dado un nodo cualquiera flotarlo hasta que se cumpla una condición. Por ejemplo, flotar un nodo hasta que el elemento del mismo sea menor que el de su padre.*

Vamos a implementar un ABB mediante un Abin, para ello vamos a ir recorriendo el árbol binario comparando los elementos de los nodos y ordenándolos.

Como vamos a ir modificando el árbol, este será una referencia no constante y se devolerá por referencia.

```
#include <iostream
#include "abin.h"

template <typename T>
void FlotarNodos(const T& e, Abin<T> &A){
  if(!A.arbolVacio()){
    //Buscamos el nodo a flotar
    Abin<T>::nodo encontrado = Abin<T>::NODO_NULO;
    if(BuscaNodo(e,A.raiz(),encontrado,A))
      //llamamos al método flotar
      FlotarNodos_rec(encontrado,A);
  }
  //Si está vacío no se hace nada
}

//Nos creamos un método que nos busque el nodo
template <typename T>
bool BuscaNodo(const T& e, typename Abin<T>::nodo n, typename
↪  Abin<T>::nodo &encontrado, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return false;
  //Si no es nulo, buscamos el nodo.
  if(A.elemento(n) == e){
    encontrado = n;
    return true;
  }
    //Seguimos buscando el nodo
    return BuscaNodo(e,A.hijoIzqdo(n),encontrado,A) ||
    ↪  BuscaNodo(e,A.hijoDrcho(n),encontrado,A);
}

template <typename T>
void FlotarNodos_rec(typename Abin<T>::nodo n, Abin<T> &A){
  //Tenemos que comprobar que ni él ni el padre sean nulos.
  if(n != Abin<T>::NODO_NULO || A.padre(n) != Abin<T>::NODO_NULO){
    if(A.elemento(n) > A.elemento(A.padre(n))){
      //Si el elemento de n es mayor, flotamos.
      T elto_aux = A.elemento(n);
```

```
      A.elemento(n) = A.elemento(A.padre(n));
      A.padre(n) = elto_aux;
      //llamamos a la recursiva con el padre de n
      FlotarNodos_rec(A.padre(n),A);
    }
    //si el elemento del hijo < padre, termina
  }
  //si alguno es nulo, no hace nada
}
```

Para llevar a cabo este ejercicio hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo padre(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el padre del nodo n, si no existe devuelve `NODO_NULO`.

Hundir Nodos

Enunciado:*Dado un elemento, buscar el nodo con dicho elemento y hundirlo, cuando sea hoja se elimina.*

Este ejercicio es muy parecido al 'FlotarNodos', pero ahora buscamos el nodo con dicho elemento y lo hundimos (elemento de n ¡ elemento de sus descendientes) y cuando sea hoja lo eliminamos.

```
template <typename T>
void HundirNodos (const T& e, Abin<T> &A){
  if(!A.arbolVacio()){
    //Buscamos el nodo a hundir
    typename Abin<T>::nodo encontrado = Abin<T>::NODO_NULO;
    if(BuscaNodo(e,A.raiz(),encontrado,A))
      //Si hemo encontrado el nodo, lo hundimos
      HundirNodos_rec(encontrado,A);
  }
}

//Función que nos busca un nodo a partir de su elemento
template <typename T>
bool BuscaNodo(const T& e, typename Abin<T>::nodo n, typename
↪  Abin<T>::nodo &encontrado, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO)
    return false; //No se ha podido encontrar el nodo
  //Si no es nulo, buscamos el nodo
  if(A.elemento(n) == e){
    //hemos encontrado el nodo
    encontrado = n;
    return true;
  }
  //Seguimos buscando el nodo
  return BuscaNodo(e,A.hijoIzqdo(n),encontrado,A) ||
  ↪  BuscaNodo(e,A.hijoDrcho(n),encontrado,A);
}

//Función que nos comprueba si el nodo es una hoja
template <typename T>
bool EsHoja(typename Abin<T>::nodo n, const Abin<T> &A){
  return (A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
  ↪  Abin<T>::NODO_NULO);
}

template <typename T>
void HundirNodos_rec(typename Abin<T>::nodo n, Abin<T> &A){
  if(n != Abin<T>::NODO_NULO){
    //Creamos un nodo auxiliar
```

```
    typename Abin<T>::nodo mayorHijo = Abin<T>::NODO_NULO;
    //Vamos a obtener el elemento mayor de los hijos de n
    if(!EsHoja(n,A)){
      //n tiene ambos hijos.
      if(A.elemento(A.hijoIzqdo(n)) > A.elemento(A.hijoDrcho(n))){
        //hizq > hder.
        mayorHijo = A.hijoIzqdo(n);
      }
      else
      //hder > hizq.
        mayorHijo = A.hijoDrcho(n);
    }
    if(A.elemento(n) < A.elemento(mayorHijo)){
      //Hundimos el nodo
      T elto_aux = A.elemento(n);
      A.elemento(n) = A.elemento(mayorHijo);
      A.elemento(mayorHijo) = elto_aux;
      HundirNodos_rec(mayorHijo,A);
    }
    //Compruebo que sea hoja, para eliminarlo
    else if(EsHoja(n,A)){
      //lo eliminamos, llamando al padre
      if(A.elemento(A.hijoIzqdo(A.padre(n))) == A.elemento(n)){
        //n es hijo izquierdo del padre
        A.eliminarHijoIzqdo(A.padre(n));
      }
      else if(A.elemento(A.hijoDrcho(A.padre(n))) == A.elemento(n)){
        //n es hijo derecho del padre
        A.eliminarHijoDrcho(A.padre(n));
      }
    }
  }
  //Si es nulo, no se hace nada
}
```

Para poder realizar este ejercicio hemos hecho uso de la representación enlazada del TAD Abin, cuya parte privada es:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //raíz del Árbol
}; //fin del TAD Abin.
```

También hemos hecho uso de los siguientes métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true, si vacío, si no false.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si el árbol está vacío devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: Nodo 'n' se encuentra en el árbol.
  - *Post*: Devuelve el elemento del nodo 'n'.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del nodo n, si el árbol está vacío devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si el árbol está vacío devuelve `NODO_NULO`.
- `nodo padre(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el padre del nodo n, si el árbol está vacío devuelve `NODO_NULO`.
- `void eliminarHijoIzqdo(nodo n);`
  - *Pre*: El nodo 'n' existe en el árbol, existe hijoIzqdo de n y es una hoja.
  - *Post*: Elimina el hijo izquierdo del nodo n.
- `void eliminarHijoDrcho(nodo n);`
  - *Pre*: El nodo 'n' existe en el árbol, existe hijoDrcho de n y es una hoja.
  - *Post*: Elimina el hijo derecho de n.

Subárbol Izquierdo menor que Derecho

Enunciado: *Dado un árbol A, comprobar que todos los valores del subárbol izquierdo son estrictamente menores que los valores del nodo raíz de A y todos los valores del subárbol derecho, estrictamente mayores. Dicha condición se debe dar de igual modo en dichos subárboles.*

Parece que se puede hacer mediante un ABB, pero este ya nos lo daría hecho, por tanto, vamos a hacer uso de un Abin comprobando que los elementos del hijo izquierdo de n < el elemento de la raíz n < elemento del hijo derecho de n

```
template <typename T>
bool MenorSubarbol(const Abin<T> &A){
  if(!A.arbolVacio()){
    return MenorSubarbol_rec(A.raiz(),A);
  }
  return true;
}

template <typename T>
bool MenorSubarbol_rec(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return true;
  //Tenemos ambos hijos:
  if(A.hijoIzqdo(n) != Abin<T>::NODO_NULO && A.hijoDrcho(n) !=
  ↪  Abin<T>::NODO_NULO)
  {
    if(A.elemento(A.hijoIzqdo(n)) < A.elemento(n) && A.elemento(n) <
    ↪  A.elemento(A.hijoDrcho(n)))
      //Se cumple que el hijo izquierdo es menor que n
      return MenorSubarbol_rec(A.hijoIzqdo(n),A) &&
      ↪  MenorSubarbol_rec(A.hijoDrcho(n),A);
    else
      return false; //No se cumple que hizq < n < hder.
  }
  //Solo tenemos hijo izquierdo
  else if(A.hijoIzqdo(n) != Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
  ↪  Abin<T>::NODO_NULO){
    if(A.elemento(A.hijoIzqdo(n)) < A.elemento(n))
      return MenorSubarbol_rec(A.hijoIzqdo(n),A);
    else
      return false;
  }
  else if(A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) !=
  ↪  Abin<T>::NODO_NULO){
    //solamente tenemos hijo derecho
    if(A.elemento(n) < A.elemento(A.hijoDrcho(n))){
      return MenorSubarbol_rec(A.hijoDrcho(n),A);
    }
```

```
    return false; //no se cumple que n < hder
  }
  else{
    return true; //no tiene hijos
  }
}
```

Hemos hecho uso del TAD Abin con representación enlazada debido a que no tenemos un número determinado de nodos en el árbol, siendo la parte privada del TAD:

```
  private:
    struct Celda{
      T elto;
      nodo padre, hizq, hder;
      Celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p)
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
}; //fin del TAD Abin
```

También he hecho uso de los métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true, si el árbol está vacío, si no false.
- `nodo raiz()const;`
  - *Post*: Devuelve la raíz del árbol, si el árbol está vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo de 'n', si este no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo derecho del 'n', si este no existe devolerá `NODO_NULO`.
- `const T& elemento (nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el elemento contenido en el nodo 'n'.

Comprobar si es AVL o no

Enunciado: *Construye una función que, dado un Árbol Binario, devuelva true si es un AVL y false en el caso contrario.*

Sabemos que un AVL es un árbol binario de búsqueda pero con la condición de que está equilibrado, es decir, su factor de desequilibrio es ≤ 1.

Por tanto, para ver si un árbol binario es un AVL, tenemos que comprobar estas dos condiciones:

1. Tiene que ser un ABB, elemento hijo izquierdo ¡ padre ¡ hijo derecho.
2. Tiene que estar equilibrado.

```
template <typename T>
bool esAVL(const Abin<T> &A){
  if(A.arbolVacio())  return true;
  bool flag = EsAbb(A.raiz(),A); //Comprobar que es ABB
  return flag &= EsAVL_rec(A.raiz(),A); //Comprobar que está equilibrado
}

//Método que comprueba si un nodo es una hoja o no
template <typename T>
bool EsHoja(typename Abin<T>::nodo n, const Abin<T> &A){
  return (A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
   ↪  Abin<T>::NODO_NULO);
}

//Función que comprueba si es una ABB
template <typename T>
bool EsAbb(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return true;

  //Tiene ambos hijos
  if(!EsHoja(n,A)){
    return (A.elemento(A.hijoIzqdo(n)) < A.elemento(n) && A.elemento(n) <
     ↪  A.elemento(A.hijoDrcho(n)) && EsAbb(A.hijoIzqdo(n),A) &&
     ↪  EsAbb(A.hijoDrcho(n),A));
  }
  //Solo tiene hijo izquierdo
  if(A.hijoIzqdo(n) != Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
   ↪  Abin<T>::NODO_NULO){
    return(A.elemento(A.hijoIzqdo(n)) < A.elemento(n) &&
     ↪  EsAbb(A.hijoIzqdo(n),A));
  }
  //Solo tiene hijo derecho
  if(A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) !=
   ↪  Abin<T>::NODO_NULO){
```

```
    return (A.elemento(n) < A.elemento(A.hijoDrcho(n)) &&
    ↪  EsAbb(A.hijoDrcho(n),A));
  }
  //Es una hoja
  else return true;

}

//Vamos a comprobar que el árbol está equilibrado, para ello vamos a
↪  crearnos una función auxiliar "altura" que como su nombre indica
↪  devolverá la altura del subárbol

template <typename T>
size_t altura (typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return 0; //la altura de un nodo nulo es 0.
  return 1 + std::max(altura(A.hijoIzqdo(n),A),altura(A.hijoDrcho(n),A));
}

//Comprobamos si está equilibrado o no
template <typename T>
bool esAVL_rec(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return true;

  size_t factor_desequilibrio = abs(altura(A.hijoIzqdo(n),A) -
  ↪  altura(A.hijoDrcho(n),A));

  return factor_desequilibrio <= 1 && esAVL_rec(A.hijoIzqdo(n),A) &&
  ↪  esAVL_rec(A.hijoDrcho(n),A);
}
```

Hemos hecho uso del TAD Abin con represetanción enlazada, debido a que no nos indican un número determinado de nodos del mismo, siendo la parte privada del TAD Abin con dicha represetanción:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};
```

También hemos hecho uso de los siguientes métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.

- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raíz del árbol, si está vacío, devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del nodo 'n', si este no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo 'n', si este no existe, devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el elemento contenido en el nodo 'n'.

Infimo y Supremo

Enunciado: *Dado un conjunto ordenado no vacío A, se define el ínfimo de x como el mayor elemento de A menor o igual que x, si existe.*
Análogamente, el supremo de x en A, si existe, es el menor elemento de A mayor o igual que x.

Implementa dos funciones de $O(\log n)$ en promedio que dados un valor x cualquiera y un Abb A no vacío devuelvan, respectivamente, el ínfimo y el supremo de x en A. Si no existe el ínfimo de x en A, la función correspondiente devolverá el mínimo de A. Así mismo, la otra función devolverá el máximo de A, en el caso de que no exista el supremo.
Nota: Es absolutamente necesario definir todos los tipos de datos implicados en la resolución del ejercicio, así como los prototipos de las operaciones utilizadas de los TADs conocidos.

```
//Función que devuelve el valor máximo del Abb
template <typename T>
T maximo(const Abb<T> &A){
  //Al estar ordenado, el máximo será el elemento del subárbol Derecho, si
  ↪  este no existe, será el elemento de la raiz.
  if(A.drcho().vacio())
    return A.elemento();

  return maximo(A.drcho());
}

//Función que devuelve el valor mínimo del Abb
template <typename T>
T minimo (const Abb<T> &A){
  //Al estar ordenado, el valor mínimo seá el del subárbol Izquierdo, si
  ↪  este no existe, será el elemento de la raiz.
  if(A.izqdo().vacio())
    return A.elemento();

  return minimo(A.izqdo());
}

//Función que nos devuelve el valor supremo, el menor elemento >= x
template <typename T>
T Supremo(const Abb<T> &A, const T& x){
  if(A.vacio())
    return maximo(A);

  //Vamos a buscar el elemento menor >= x
  const T &raiz = A.elemento();
  if(x < raiz){
    //subárbol izquierdo
```

```
    T supremo = Supremo(A.izqdo(),x);
    return(supremo >= x) ? supremo : raiz; //si cumple la condición que >=
     ↪  x, devuelvo el elemento supremo.
  }
  else if(x > raiz){
    //subárbol derecho
    return Supremo(A.drcho(),x);
  }
  else{
    // x == raiz
    return raiz;
  }
}

//Función que nos devuelve el valor ínfimo, el mayor elemento <= x
template <typename T>
T Infimo(const Abb<T> &A, const T& x){
  if(A.vacio())
    return minimo(A);
  const T& raiz = A.elemento();
  else if(x < raiz){
    //subárbol izquierdo
    return Infimo(A.izqdo(),x);
  }
  else if(x > raiz){
    T infimo = infimo(A.drcho(),x);
    return(infimo <= x) ? infimo : raiz; //si cumple la condición que <=
     ↪  x, devuelvo el elemento infimo.
  }
  else{
    //x == raiz
    return raiz;
  }
}
```

He hecho uso del TAD Abb con representación enlazada, siendo esta su parte privada:

```
  private:
    struct arbol{
      T elto;
      Abb izq, der;
      arbol(const T& e):elto(e),izq{},der{} {}
    };
  arbol *r; //raiz del árbol
};//fin del TAD Abb
```

He hecho uso de los siguientes métodos públicos del TAD:

- `const T& elemento()const;`

- *Post*: Devuelve el elemento que se encuentra en la raiz del subárbol.

- `bool vacio()const;`
  - *Post*: Devuelve true, si el subárbol está vacío, si no false.
- `const Abb &izqdo()const;`
  - *Pre*: Subárbol no vacío.
  - *Post*: Devuelve el subárbol izquierdo.
- `const Abb &drcho()const;`
  - *Pre*: Subárbol no vacío.
  - *Post*: Devuelve el subárbol derecho.

Agenda ABB

Enunciado: *Haz una agenda usando un Árbol binario de búsqueda. Utiliza como estructura una con los campos teléfono y nombre por ejemplo, y acuerdate de sobrecargar los operandos de comparación de la estructura.*

Para poder realizar este ejercicio, tenemos que crear la clase agenda, la cual tendrá asociadas a personas con su número de teléfono y usando un ABB podamos realizar diferentes operaciones en la misma.

```
struct Persona{
  std::string nombre_, telefono_;
  Persona(const string nombre, string telefono):
  ↪  nombre_(nombre),telefono_(telefono){}
  bool operator < (const Persona &p){return nombre_ < p.nombre_};
};

class Agenda{
  Abb<Persona>agenda_;
  const std::string& getNombre_rec(const std::string& tlf, const
  ↪  Abb<Persona>& A) const;
  public:
    void insertarPersona(const Persona &p);
    void eliminarPersona(const Persona &p);
    const std::string &getTLF(const std::string nom)const;
    const std::string &getNombre(const std::string tlf)const;
};

void Agenda::insertarPersona(const Persona &p){
  agenda_.insertar(p);
}

void Agenda::eliminarPersona(const Persona &p){
  agenda_.eliminar(p);
}

const std::string &Agenda::getTFL(const std::string nombre)const{
  return agenda_.buscar(Persona(nombre,"").elemento().telefono_);
}

const std::string &Agenda::getNombre(const std::string tlf)const{
  return getNombre_rec(tlf, A.izqdo()) && getNombre_rec(tlf,A.drcho());
}

const std::string &Agenda::getNombre_rec(const std::string tlf, const
↪  Abb<Persona> &A)const {
  if(!A.vacio()){
```

```
    if(A.elemento().telefono_ == tlf){
      return A.elemento().nombre_;
    }
    else{
      getNombre_rec(tlf,A.izqdo()) && getNombre_rec(tlf,A.drcho());
    }
  }
  return ""; //si no existe el tlf, no existe persona.
}
```

Para poder realizar este ejecicio he hecho uso del TAD Abb con la representación enlazada debido a que no nos dicen cuantas personas tiene que haber como máximo en la agenda. Esta es su parte privada:

```
  private:
    struct arbol{
      T elto;
      arbol izq, der;
      arbol(const T& e):elto(e),izq{}, der{} {}
    };
  arbol r; //raíz del subárbol.
}; //fin del TAD Abb
```

También he hecho uso de los siguientes métodos públicos del TAD Abb:

- `void insertar(const T& e);`
  - *Post*: Inserta el elemento en el Abb.
- `void eliminar(const T& e);`
  - *Pre*: Árbol no vaciío.
  - *Post*: Elimina el elemento del subárbol.
- `const Abb<T> &buscar(const T& e)const;`
  - *Pre*: Árbol no vacío.
  - *Post*: Devuelve el subárbol donde 'e' es la raíz.
- `bool vacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `const Abb<T> &izqdo()const;`
  - *Pre*: Árbol no vacío.
  - *Post*: Devuelve el subárbol izquierdo.
- `const Abb<T> &drcho()const;`
  - *Pre*: Árbol no vacío.
  - *Post*: Devuelve el subárbol derecho.

- `const T &elemento()const;`
  - *Pre*: Árbol no vacío.
  - *Post*: Devuelve el elemento de la raíz del subárbol.

Múltiplos de 3 Árbol General

Enunciado: *Implementa un subprograma que devuelva el porcentaje de descendientes propios de un árbol general que sean múltiplos de 3.*

Los descendientes propios son los hijos de ese nodo, al estar en un árbol general será toda la lista de hijos del nodo n.

```
template <typename T>
double Porcentaje(const Agen<T> &A){
  return Porcentaje_rec(A.raiz(),A);
}

//Función que nos devuelve el número de hijos de un nodo
template <typename T>
size_t NumHijos(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
  size_t nhijos = 0;
  while(hijo != Agen<T>::NODO_NULO){
    nhijos++;
    hijo = A.hermDrcho(hijo);
  }
  return nhijos;
}

//Función que devuelve el número de nodos que el número de hijos es
↪  múltiplo de 3
template <typename T>
size_t DescendientesMultiplo3(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  typename Agen<T>::nodo hijo =  A.hijoIzqdo(n);
  size_t nhijos = 0;
  while(hijo != Agen<T>::NODO_NULO){
    if(NumHijos(hijo,A) % 3 == 0){
      nhijos++;
    }
    nhijos += NumHijosM3(hijo,A);
    hijo = A.hermDrcho(hijo);
  }
  return nhijos;
}

template <typename T>
size_t DescendientesTotales(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
```

```
  size_t descendientes = 0;
  while(hijo != A.hijoIzqdo(n)){
    descendientes++;
    descendientes += DescendientesTotales(hijo,A);
    hijo = A.hermDrcho(hijo);
  }
  return descendientes;
}

template <typename T>
double Porcentaje_rec(typename Agen<T>::nodo n, const Agen<T> &A){
  if(A.raiz() == Agen<T>::NODO_NULO) return .0;
  //obtenemos el número de descendientes
  size_t descendientesTotales = DescendientesTotales(n,A);
  size_t descendientesMultiplo3 = DescendientesMultiplo3(n,A);

  if(descendientesTotales == 0) return 0.0; //evitamos la divisón por 0.

  return (descendientesMultiplo3/descendientesTotales) * 100.0;
}
```

He hecho uso del TAD agen con representación enlazada, siendo esta su parte privada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol.
};
```

También he hecho uso de los siguientes métodos públicos del TAD Agen:

- `nodo raiz()const;`
  - *Pre*: Árbol no vacío.
  - *Post*: Devuelve el nodo raíz del árbol.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del nodo 'n'.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo 'n'.

Densidad de un Árbol General

Enunciado: *Calcular densidad de un árbol. La densidad se define como el Grado máximo de un árbol partido del número de nodos hojas.*

El grado de un árbol es el máximo de los grados de todos los nodos que lo componen, es decir, el grado de un árbol equivale al nodo con más hijos del mismo.

```
//Función que nos devuelve el número de nodos hojas del Árbol
template <typename T>
bool esHoja(typename Agen<T>::nodo n, const Agen<T> &A){
  return A.hijoIzqdo(n) == Agen<T>::NODO_NULO;
}

template <typename T>
size_t NodosHojas(typename Agen<T>::nodo n; const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
  size_t hojas = 0;
  while(hijo != Agen<T>::NODO_NULO){
    if(esHoja(hijo,A))
      hojas++;
    hojas += NodosHojas(hijo,A);
    hijo = A.hermDrcho(hijo);
  }
  return hojas;
}

//Función que nos devuelve el grafo máximo del Agen
template <typename T>
size_t nHijos(typeneme Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
  size_t hijos = 0;
  while(hijo != Agen<T>::NODO_NULO){
    hijo++;
    A.hermDrcho(hijo);
  }
  return hijos;
}

template <typename T>
size_t GradoMax(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  size_t maximo = nHijos(n,A);
  typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
  while(hijo != Agen<T>::NODO_NULO){
```

```
    if(nHijos(hijo,A) > maximo)
      maximo = std::max(maximo,GradoMax(hijo,A));
    hijo = A.hermDrcho(hijo);
  }
  return maximo;
}

template <typename T>
double DensidadAgen(const Agen<T> &A){
  size_t nodosHojas = NodosHojas(A.raiz(),A);
  size_t gradomax = GradoMax(A.raiz(),A);
  if(nodosHojas == 0) return 0.0;
  return (gradomax / nodosHojas);
}
```

He hecho uso del TAD Agen en su representación enlazada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //raíz del Árbol.
};
```

También he hecho uso de los siguientes métodos públicos del TAD Agen:

- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del nodo 'n', si este no existe devuelve `NODO_NULO`.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo 'n' existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo 'n', si este no existe devuelve `NODO_NULO`.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raíz del árbol, si este no existe devuelve `NODO_NULO`.

2. Ejercicios Grafos

En este apartado vas a encontrar todos los ejercicios tipo examen sobre Grafos, donde haremos uso de los diversos algoritmos vistos en las clases de teoría y seminarios.

Ciudades Rebeldes

Enunciado: *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.

Partimos de la base de que tenemos un grafo ponderado y dirigido (ya que solo se puede ir en una dirección), contamos con el conjunto de ciudades y carreteras (unión de dos ciudades) que están tomadas por los rebeldes, a las cuales no podremos acceder. Además tenemos la capital de Zuelandia, que nos servirá para hacer que todos los viajes se realicen por la misma.

Como queremos obtener las distancias mínimas de viajar entre dos ciudades cualesquiera de Zuelandia, haremos uso del algoritmo de Floyd, ya que este calcula los costes de viajar entre cada par de vértices de todo el grafo.

```
typedef std::pair<size_t, size_t> Carretera;
typedef matriz<size_t> CostesViajes;

CostesViajes ZuelandiaRebeldes(const GrafoP<size_t>& G, const
↪  vector<size_t>& CiudadesRebeldes, const vector<Carretera>&
↪  CarreterasRebeldes, size_t Capital) {
  // Copia del grafo original para no modificarlo
  GrafoP<size_t> Gmod = G;

  // Marcar ciudades rebeldes como inaccesibles
  for (size_t ciudad : CiudadesRebeldes)
    for (size_t i = 0; i < G.numVert(); i++)
      Gmod[i][ciudad] = GrafoP<size_t>::INFINITO;

  // Marcar carreteras rebeldes como cortadas
  for (const Carretera& carretera : CarreterasRebeldes)
    Gmod[carretera.first][carretera.second] = GrafoP<size_t>::INFINITO;

  // Obtener caminos mínimos desde y hacia la capital
  vector<size_t> VerticesD(G.numVert()), VerticesDI(G.numVert());
  vector<size_t> CostesMinimosD = Dijkstra(Gmod, Capital, VerticesD);
  vector<size_t> CostesMinimosDI = DijkstraInv(Gmod, Capital, VerticesDI);

  // Construir la matriz de costes mínimos obligando a pasar por la capital
  CostesViajes CostesMinimos(G.numVert(), GrafoP<size_t>::INFINITO);

  for (size_t i = 0; i < G.numVert(); i++) {
    for (size_t j = 0; j < G.numVert(); j++) {
      if (i != j && CostesMinimosD[i] != GrafoP<size_t>::INFINITO &&
      ↪  CostesMinimosDI[j] != GrafoP<size_t>::INFINITO) {
        CostesMinimos[i][j] = CostesMinimosD[i] + CostesMinimosDI[j];
      }
    }
  }

  return CostesMinimos;
}
```

Hemos hecho uso de las operaciones de los grafos y prototipos de las funciones:

- `size_t numVert() const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `matriz<size_t>(G.numVert());`
  - *Post*: Crea y devuelve una matriz de enteros sin signos vacía de tamaño G.numVert().
- `vector<tCoste>& operator[](vertice v)const;`

- *Post*: Devuelve un vector con los costes de las aristas adyacentes a v.

- `const T& GrafoP<tCoste>::INFINITO = std::numeric_limits<T>::max();`
  - *Post*: Devuelve el valor máximo permitido.
- `matriz<tCoste> Floyd(const GrafoP<size_t> &G, vector<typename GrafoP<tCoste>::vertice> &P);`
  - *Pre*: Recibe un Grafo ponderado y una matriz de vértices.
  - *Post*: Devuelve una matriz con los costes mínimos y una matriz con los vértices por los que pasan los caminos.

Puentes Grecoland

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

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

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

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

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

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

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

« Este ejercicio pertenece a la Práctica 8 Ejercicio 7 de grafos. »

El enunciado nos dice que Grecoland está formado por dos Islas (Fobos y Deimos) las cuales tienen un número arbitrario de Ciudades, de las cuales algunas de ellas son costeras. Estas ciudades están representadas por sus coordenadas cartesianas (dos enteros sin signos). Pero estas islas y sus ciudades no están comunicadas debido a un huracán, por lo que tendremos que reconstruir Grecoland con el el menor coste posibles.

Para poder reconstruir Grecoland, las ciudades que pertenecen a la misma isla se unen por carretera y las islas se unen mediante las ciudades costeras con puentes, el coste de estas carreteras y puentes equivalen a la distancia euclídea de ambas ciudades que

unen, sabiendo esto, el lógico que un puente sea más costoso de construir que una carretera, por lo que nos vamos a quedar con las ciudades de las islas unidas con el menor número de carreteras y puentes cuyo coste es el menor posibles, es decir, haremos uso de Kruskall.

Finalmente, cuando tengamos a Grecoland reconstruido, vamos a calcular el coste mínimo de ir desde la Ciudad Origen hacia la Destino, mediante el uso del algoritmo de Dijkstra (ya que nos devuelve los vectores de coste mínimo y el camino desde el vértice origen a los demás vértices del grafo).

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> Ciudad; //ciudad con las coordenadas x,y.

//Función que nos devuelve la distancia entre dos ciudades
size_t DistanciaEuclidea(Ciudad A, Ciudad B){
  return sqrt(pow(A.first-B.first,2) + pow(A.second-B.second,2));
}

size_t PuentesGrecoland(vector<Ciudad> &CiudadesFobos, vector<Ciudad>
↪  &CiudadesDeimos, vector<size_t> &CosterasFobos, vector<size_t>
↪  &CosterasDeimos, size_t Origen, size_t Destino){
//Vamos a crear las islas de Fobos y Deimos
GrafoP<size_t>GFobos(CiudadesFobos.size()),GDeimos(CiudadesDeimos.size());

//Los puentes son más caros que las carreteras por tanto estos tendrán un
↪  factor de coste extra
size_t factorPuente = 0;

//Ahora vamos a rellenar el grafo de Fobos con sus ciudades
for(size_t i = 0; i < CiudadesFobos.size(); i++)
  for(size_t j = i+1; j < CiudadesFobos.size(); j++)
    GFobos[i][j] = GFobos[j][i] =
    ↪  DistanciaEuclidea(CiudadesFobos[i],CiudadesFobos[j]);
    factorPuente = max(factorPuente, GFobos[i][j]);
//Ahora rellenamos el grafo de Deimos con sus ciudades.
for(size_t i = 0; i < CiudadesDeimos.size(); i++)
  for(size_t j = i+1; j < CiudadesDeimos.size(); j++)
    GDeimos[i][j] = GDeimos[j][i] =
    ↪  DistanciaEuclidea(CiudadesDeimos[i],CiudadesDeimos[j]);
    factorPuente = max(factorPuente, GDeimos[i][j])
//Ya rellenos ambas Islas (Grafos de Fobos y Deimos), vamos a quedarnos
↪  con las carreteras de menor coste, haciendo que no haya ciclos en las
↪  mismas
GFobos = Kruskall(GFobos);
GDeimos = Kruskall(GDeimos);

//Ahora vamos a crear el archipiélago Grecoland con las islas Fobos y
↪  Deimos
```

```
GrafoP<size_t> Grecoland(CiudadesFobos.size()+CiudadesDeimos.size());
for(size_t i = 0; i < CiudadesFobos.size(); i++)
  for(size_t j = 0; j < CiudadesFobos.size(); j++)
    Grecoland[i][j] = GFobos[i][j];
for(size_t i = 0; i < CiudadesDeimos.size(); i++)
  for(size_t j = 0; j < CiudadesDeimos.size(); j++)
    Grecoland[i+CiudadesFobos.size()][j+CiudadesFobos.size()] =
    ↪ GDeimos[i][j];

//Relleno el supregrafo con las islas, vamos a proceder a crear los
↪ puentes.
//Al estar contenidas las ciudades costeras en el vector de ciudades de
↪ cada isla, es un vector de enteros (índice que ocuparía en dicho
↪ vector)
for(size_t i : CosterasFobos){
  for(size_t j : CosterasDeimos){
    size_t CostePuente = DistanciaEuclidea(CiudadesFobos[i],
    ↪ CiudadesDeimos[j]) + factorPuente; //Los puentes siempre van a
    ↪ ser más caros que la carretera más cara por eso se coge el máximo.
    Grecoland[i][j+CiudadesFobos.size()] =
    ↪ Grecoland[j+CiudadesFobos.size()][i] = CostePuente;
  }
}
//Ahora que tenemos todos los puentes, nos vamos a quedar con los menos
↪ costosos sin que haya ciclo.
Grecoland = Kruskall(Grecoland);

//El coste de viajar es el mismo tanto para carretera como para puentes
for(size_t i = 0; i < Grecoland.numVert(); i++){
  for(size_t j = 0; j < Grecoland.numVert(); j++){
    if(Grecoland[i][j] > factorPuente){
      Grecoland[i][j] -= factorPuente;
    }
  }
}

//Finalmente, vamos a obtener el coste mínimo de ir desde el origen hacia
↪ nuestro destino.

vector<size_t>Vertices(Grecoland.numVert()),CostesMinimos(Grecoland.numVert());
CostesMinimos = Dijkstra(Grecoland,Origen,Vertices);
return CostesMinimos[Destino];
}
```

Operaciones de los grafos y prototipos de las funciones usadas:

- `GrafoP<tCoste> GrafoP(size_t n);`
  - *Post*: Crea un grafo ponderado de tamaño n.

- `size_t numVert()const;`
  - *Post*: Devuelve el número de aristas del grafo.
- `vector<tCoste>& operator[](vertice v)cosnt;`
  - *Post*: Devuelve un vector con los costes de las aristas adyacentes al vértice v.
- `GrafoP<tCoste> Kruskall(GrafoP<tCoste> &G);`
  - *Post*: Devuelve el árbol generado de coste mínimo de un grafo ponderado y no dirigido G.
- ```
  vector<tCoste> Dijkstra(const GrafoP<tCoste> &G,
      typename GrafoP<tCoste>::vertice Origen,
      vector<typename GrafoP<tCoste>::vertice> &P);
  ```
  - *Post*: Calcula los caminos de coste mínimo entre el origen y todos los vértices del Grafo ponderado y devuelve el conjunto de vértices del camino origen - P[i] (último vértice del camino).
  ```

Toxicidad Zuelandia (Junio 2017)

Enunciado: *La capital de Zuelandia esta alcanzando niveles de toxicidad muy elevados, por ello se ha decretado el cierre a la ciudad como paso de transito hacia otras ciudades (Para ir de una ciudad a otra no se podrá pasar por C.Zuelandia pero si se podrá ir si es residente de la misma empleandola como ciudad destino u origen).*

Implemente un subprograma que dada la capital y un grafo ponderado con los kilómetros de las carreteras existentes entre las ciudades del país, nos devuelva los caminos resultantes de nuestra nueva política "Sin pasar por la capital, por favor".

Nota importante: Se ha de definir todos los tipos de dato, prototipo de las operaciones empleadas en los TADs y también los prototipos de los grafos vistos en clase que se empleen.

Tenemos que la capital de Zuelandia está cortada, por lo que todos los caminos que se puedan realizar deberán de evitar la capital, menos los que son residentes que lo pueden usar como origen o destino de sus viajes.

Partimos de que tenemos un grafo ponderado (matriz de costes) y la capital de Zuelandia. Para poder calcular todos los caminos de coste mínimo que se pueden realizar, primero vamos a quitar todos los caminos que pasan por la capital y luego haremos uso del algoritmo de Floyd para poder obtener dichos caminos.

```
//Definimos los tipos de datos a usar
typedef matriz<size_t>Caminos;

Caminos ZuelandiaToxica(GrafoP<size_T> &G, size_t Capital){
  size_t CZuelandia = Capital;
  //Creamos una copia del grado
  GrafoP<size_t> Copia(G);
  //Recorremos el grafo quitando los caminos que cruzan la capital
  for(size_t i = 0; i < Copia.numVert(); i++)
    //Como es un grafo no dirigido:
    Copia[i][CZuelandia] = Copia[CZuelandia][i] =
    ↪  GrafoP<size_t>::INFINITO;

  //Ahora podemos hacer uso de Floyd para obtener todos los caminos de
  ↪  coste mínimo que no pasan por la capital

  ↪  matriz<size_t>VerticesF(Copia.numVert()),CostesMinimosF(Copia.numVert());
  CostesMinimosF = Floyd(Copia,VerticesF);

  //Ahora vamos a calcular los costes de los caminos de los residentes de
  ↪  la capital, siendo esta origen de los caminos.
  vector<size_t>VerticesD(G.numvert()),CostesMinimosD(G.numVert());
  CostesMinimosD = Dijkstra(G,CZuelandia,VerticesD);

  //Ahora hacemos lo mismo pero siendo esta el destino
```

```
  vector<size_t>VerticesInv(G.numVert()),CostesMinimosInv(G.numVert());
  CostesMinimosInv = DijkstraInv(G,CZuelandia,VerticesInv);

  //Ahora copiamos los caminos resultantes de realizar Dijkstra y
  ↪  DijkstraInv en la matriz de caminos
  for(size_t i = 0; i < G.numVert(); i++){
    VerticesF[Capital][i] = VerticesD[i];
    VerticesF[i][Capital] = VerticesInv[i];
  }
  //Ahora podemos devolver los caminos de coste mínimo
  return VerticesF;
}
```

Hemos hecho uso de las siguientes operaciones de los grafos y prototipos de funciones:

- `matriz();`
  - *Post*: Creamos y devolvemos una matriz vacía.
- `size_t numVert()const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `const vector<size_t>& operator[](vertice v);`
  - *Post*: Devuelve un vector no modificable con los costes de las aristas adyacentes al vértice v.
- `size_t GrafoP<tCoste>::INFINITO = std::numeric_limits:max();`
  - *Post*: Devuelve el máximo valor permitido.
- `matriz<tCoste> Floyd(const GrafoP<tCoste> &G, matriz<typename GrafoP <tCoste>::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado y una matriz de vértices
  - *Post*: Devuelve dos matrices, una con los costes mínimos de ir entre cualquier par de vértices del grafo y otra con los vértices del camino.
- `vector<tCoste> Dijkstra(const GrafoP<tCoste> &G, typename GrafoP <tCoste>::vertice Origen, vector<typename GrafoP<tCoste> ::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado, un vértice que es el origen del camino y un vector de vértice
  - *Post*: Devuelve dos vectores, uno con el coste mínimo de ir desde el origen a cualquier otro vértice del grafo y el camino (vértices) por los que pasa.
- `vector<tCoste> DijkstraInv(const GrafoP<tCoste> &G, typename GrafoP <tCoste>::vertice Destino, vector<typename GrafoP<tCoste> ::vertice> &P);`
- *Post*: Devuelve dos vectores, uno con el coste mínimo de ir desde cualquier vértice del grafo al destino y el camino (vértices) por los que pasa.

Lineas Aéreas Tumbuctú (Junio 2019)

Enunciado:*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.

« Este ejercicio pertenece a la Práctica 8 Ejercicios 1 y 2 de grafos. »

Según el enunciado, el archipiélago Tumbuctú está formado por un número arbitrario de Islas, estas Islas tienen una serie de ciudades (formadas por las coordenadas cartesianas x e y, es decir dos enteros) que la forman. Las Islas del archipiélago se unen mediante líneas aéreas, todas las ciudades del archipiélago tienen un aeropuerto por lo que tendremos $n(n-1)/2$ líneas aéreas posibles. Esto es una gran cantidad de líneas aéreas pero el enunciado nos dice que solamente puede haber una por cada par de islas, para conseguir esto haremos uso del TAD APO, ya que podemos acceder al mínimo en orden constante, es decir en $O(1)$, esto nos reduce mucho el número de las líneas aéreas del archipiélago.

Partimos de un grafo matriz de adyacencia y de la lista de todas las ciudades del archipiélago, por tanto, siguiendo los siguientes pasos vamos a poder obtener las líneas aéreas de nuestro archipiélago:

1. Construir las Islas del archipiélago haciendo uso de la matriz de adyacencia y de la lista de ciudades.

2. Construir el archipiélago insertando en el mismo las Islas creadas.

3. Obtener todas las líneas aéreas posibles de nuestro archipiélago.

4. Insertar en el conjunto de las líneas aéreas las de menor coste y solamente 1 por cada par de Islas.

5. Devolver dicho conjunto.

```
//Primero de todo voy a definir los tipos de datos a usar así como las
↪  funciones auxiliares

typedef std::pair<size_t, size_t> Ciudad; // Ciudad con coordenadas x, y

struct Isla {
  std::vector<Ciudad> CiudadesIsla; // Conjunto de ciudades de la isla
  matriz<size_t> CostesIslas; // Costes mínimos de viajar entre cualquier
  ↪  ciudad de la isla
};

typedef std::vector<Isla> Archipielago; // El archipiélago Tumbuctú es un
↪  conjunto de Islas

struct LineaAerea {
  size_t Origen, Destino;
  size_t CosteLA;
};

typedef std::vector<LineaAerea> LineasAereas; // Conjunto de líneas
↪  aéreas a devolver

// Función que calcula la distancia entre dos ciudades de distinta isla
size_t DistanciaEuclidea(Ciudad A, Ciudad B) {
  return sqrt(pow(A.first - B.first, 2) + pow(A.second - B.second, 2));
}

// Función auxiliar que me devuelve una matriz de coste mínimo de una Isla
matriz<size_t> CalculaCostesIsla(const std::vector<Ciudad>& ciudades) {
  // Creamos el grafo que será la isla
  GrafoP<size_t> GIsla(ciudades.size());
  // Rellenamos el grafo con las ciudades de la isla
  for (size_t i = 0; i < ciudades.size(); i++) {
    for (size_t j = i + 1; j < ciudades.size(); j++) {
      GIsla[i][j] = GIsla[j][i] = DistanciaEuclidea(ciudades[i],
      ↪  ciudades[j]);
    }
  }
```

```
  // Ahora que tenemos el grafo relleno, haré uso de Floyd para quedarme
  ↪ con los costes mínimos de viajar por la isla
  matriz<size_t> VerticesF(GIsla.numVert()),
  ↪ CostesMinimos(GIsla.numVert());
  return CostesMinimos = Floyd(GIsla, VerticesF);
}

// Función que me calcula las líneas aéreas del archipiélago
LineasAereas Tumbuctu2(const Grafo& MA, const std::vector<Ciudad>&
↪ Ciudades) {
  // Primero de todo, voy a crear las Islas, como dije antes, cada
  ↪ representante será la "Capital de la Isla" y todas aquellas ciudades
  ↪ que estén ligadas a ese representante, irá a esa Isla.
  Particion P(MA.numVert());
  for (size_t i = 0; i < MA.numVert(); i++) {
    for (size_t j = i + 1; j < MA.numVert(); j++) {
      if (MA[i][j] == true) { // Hay camino directo entre el par de
      ↪ vértices
        if (P.encontrar(i) != P.encontrar(j)) {
          P.unir(P.encontrar(i), P.encontrar(j));
        }
      }
    }
  }

  // Ya tenemos creados los subconjuntos de las ciudades con su
  ↪ representante, ahora vamos a insertarlos en las Islas
  ↪ correspondientes
  Isla islaux;
  Archipielago archipielagoAux;
  size_t representante_actual = P.encontrar(0);
  for (size_t i = 0; i < Ciudades.size(); i++) {
    if (P.encontrar(representante_actual) == P.encontrar(i)) {
      islaux.CiudadesIsla.push_back(Ciudades[i]);
    } else { // Cambiamos de representante y de isla
      // Primero guardamos los cambios en la isla anterior
      islaux.CostesIslas = CalculaCostesIsla(islaux.CiudadesIsla);
      // Guardamos la isla en el archipiélago
      archipielagoAux.push_back(islaux);
      // Asignamos la nueva isla limpiando las ciudades de islaux anterior
      ↪ ya previamente guardada
      islaux.CiudadesIsla.clear();
      islaux.CiudadesIsla.push_back(Ciudades[i]); // insertamos la nuevas
      ↪ islas
      representante_actual = P.encontrar(i);
    }
  }
```

```
  // Guardamos la última isla
  islaux.CostesIslas = CalculaCostesIsla(islaux.CiudadesIsla);
  archipielagoAux.push_back(islaux);

  // Ya tenemos creadas las islas, ahora vamos a crear el APO para obtener
  ↪ las líneas aéreas de menor coste
  size_t n = archipielagoAux.size();
  Apo<LineaAerea> A(n * (n - 1) / 2);
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      // Solo una línea aérea por cada par de islas, por tanto, tenemos
      ↪ que comprobar que los representantes sean diferentes
      if (P.encontrar(i) != P.encontrar(j)) // Distintas islas, se inserta
      ↪ en el APO
        A.insertar({Ciudades[i], Ciudades[j],
        ↪ DistanciaEuclidea(Ciudades[i], Ciudades[j])});
    }
  }

  // Ahora, construido el APO, vamos sacando desde la raíz (el de menor
  ↪ coste) por cada par de islas.
  matriz<bool> IslasConectadas(n, false); // Creamos una matriz para
  ↪ comprobar si están o no conectadas ya las islas
  LineasAereas vector_LA(n * (n - 1) / 2); // tiene que tener el tamaño
  ↪ del APO
  size_t indice = 0;
  while (!A.vacio() && indice < (n * (n - 1) / 2)) {
    LineaAerea MenosCostosa = A.cima();
    // Ahora comprobamos que el origen y destino de la línea aérea sean de
    ↪ diferente isla
    if (P.encontrar(MenosCostosa.Origen) !=
    ↪ P.encontrar(MenosCostosa.Destino) &&
    ↪ !IslasConectadas[MenosCostosa.Origen][MenosCostosa.Destino]) {
      IslasConectadas[MenosCostosa.Origen][MenosCostosa.Destino] = true;
      IslasConectadas[MenosCostosa.Destino][MenosCostosa.Origen] = true;
      vector_LA[indice] = MenosCostosa;
      indice++;
    }
  }
  return vector_LA;
}
```

Hemos hecho uso de las operaciones de los grafos y prototipos de funciones:

- `matriz<size_t>(size_t n)`
  - *Post*: Creamos y devolvemos una matriz de tamaño n vacía.
- `GrafoP<size_t> GrafoP(size_t n)`

- *Post*: Crea y devuelve un grafo ponderado de tamaño n

- `vector<size_t>& operator[](vertice v)`
  - *Post*: Devuelve un vector con las aristas adyacentes al vértice v
- `size_t numVert()const;`
  - *Post*: Devuelve el número de aristas del grafo.
- `Particion(size_t n);`
  - *Post*: Crea y devuelve una Particion de tamaño n.
- `size_t encontrar(size_t i)const;`
  - *Post*: Devuelve el represante del subconjunto al que pertenece i
- `void unir(size_t i, size_t j);`
  - *Post*: Une en el mismo subconjunto ambos elementos, si estos tienen el mismo represante no hace nada
- `matriz<size_t> Floyd(const GrafoP<size_t> &G,`
  `matriz<typename GrafoP<size_t>::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado y una matriz de vértices.
  - *Post*: Devuelve dos matrices una que cotiene los costes mínimos de viajar entre cada par de vértices de todo el grafo y otra los vértices del camino.
- `Apo(size_t n);`
  - *Post*: Crea y devuelve un APO de tamaño n.
- `size_t Cima()const;`
  - *Pre*:Apo no vacío.
  - *Post*: Devuelve el elemento que se encuentra en la cima (menor).
- `void insertar(const T& e)`
  - *Post*: Inserta el elemento 'e' en el APO.

Repartidor de Bebidas (Junio 2022)

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

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

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

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

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

Nos dan un grafo ponderado (matriz de costes), la capacidad máxima de nuestro camión y el vértice donde se situa nuestro almacén, y queremos calcular el coste total de abastecer a todos los clientes.

Si tenemos dicho grafo, para poder obtener la ruta y costes minimos podemos hacer uso de Dijkstra o Floyd. Como queremos obtener todos los caminos de coste mínimo haremos uso en este caso de Floyd.

```
int Pedido(); //nos devuelve el número de cajas que necesita cada cliente

size_t DistanciaCamion(GrafoP<size_t> &G, size_t CapacidadMaxima, size_t
↪  Almacen){
  //Vamos a guardar los datos recibidos en variables, ya que las vamos a
  ↪  modificar.
  size_t cajascamion = CapacidadMaxima;
  size_t almacen = Almacen;

  //Primero de todo, obtenemos la ruta mínima que puede hacer el camión.
  matriz<size_t> Vertices(G.numVert()), CostesMinimos(G.numVert());
  CostesMinimos = Floyd(G,Vertices);

  //Ahora vamos a obtener la distancia total del camión
  size_t DistanciaTotal = 0;
  vector<bool>ClientesAbastecidos(G.numVert(), false); //clientes
  ↪  abastecidos.
  size_t actual = almacen; //vertice actual.
  size_t siguiente; // vertice al que nos dirigimos.

  //Mientras que al menos haya un cliente sin ser abastecido:
  while(ClientesPendientes(ClientesAbastecidos)){
    //Nos quedamos con el siguiente cliente
    siguiente =
    ↪  VerticeMasCercano(CostesMinimos,actual,ClientesAbastecidos);
    // Añadir la distancia al cliente más cercano
    DistanciaTotal += CostesMinimos[actual][siguiente];
    size_t cajascliente = Pedido(siguiente); //Guardamos el número de
    ↪  cajas que necesita el cliente

    if(cajascamion <= cajascliente){
     //No abastecemos al cliente y tenemos que volver al almacen a reponer
      cajascliente -= cajascamion;
      cajascamion = 0; //para que no haya negativos.
      DistanciaTotal += CostesMinimos[siguiente][almacen];
      cajascamion = CapacidadMaxima; //reponemos el camión
      actual = almacen; //estamos en el almacen
    }
    else{//Hemos abastecido al cliente
      cajascamion -= cajascliente; //Restamos las cajas al camión.
      //Indicamos que hemos abastecido al cliente
      ClientesAbastecidos[siguiente] = true;
      actual = siguiente; //nos quedamos en el vértice del cliente para
      ↪  avanza
    }
  }
```

```
  //Regresamos al almacen después de abastecer a todos los clientes.
  DistanciaTotal += CostesMinimos[actual][almacen];
  return DistanciaTotal;
}

//Función que nos devuelve si hay clientes pediente (no han sido
↪  abastecidos)
bool ClientesPendientes(const vector<bool> &clientesabastecidos){
  for(size_t i = 0; i < clientesabastecidos.size(); i++){
    if(!clientesabastecidos[i]){ //no se ha abastecido a dicho cliente.
      return true;
    }
  }
  return false;
}

//Función que nos devuelve el vértice más cercano a otro, no abastecido.
size_t VerticeMasCercano(matriz<size_t> &M, size_t vertice,const
↪  vector<bool> &clientesabastecidos){
  size_t minimo = GrafoP<size_t>::INFINITO;
  size_t verticemascercano = 0;
  for(size_t i = 0; i < M.dimension(); i++){
    if(!clientesabastecidos[i] && M[vertice][i] < minimo){
      minimo = M[vertice][i];
      verticemascercano = i;
    }
  }
  return verticemascercano;
}
```

Hemos hecho uso de las siguiente operaciones de los grafos y funciones:

- `size_t numVert() const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `vector<tCoste>& operator[](vertice v);`
  - *Post*: Devuelve el vector con los costes de las aristas adyacentes al vértice v.
- `matriz<tCoste> Floyd(const GrafoP<tCoste> &G, matriz<typename GrafoP <tCoste>::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado y una matriz de vértices.
  - *Post*: Devuelve dos matrices, una con los costes mínimos de entre cada par de vértices del grafo y otra de vértices por el que pasa el camino de coste mínimo.
- `matriz(size_t n);`

- *Post*: Crea y devuelve una matriz de tamaño n.

- `size_t dimension()const;`
  - *Post*: Devuelve la dimension de la matriz.

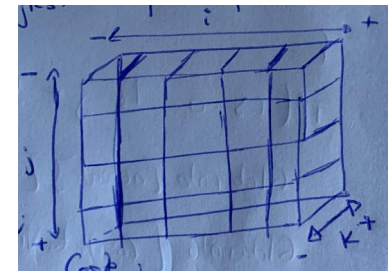
Laberinto3D (Septiembre 2022)

Enunciado: *Se dispone de un laberinto de NxNxN casillas del que se conocen las casillas de entrada y salida del mismo. Si te encuentras en una casilla sólo puedes moverte en las siguientes seis direcciones (arriba, abajo, derecha, izquierda, adelante, atrás). Por otra parte, entre algunas de las casillas hay una pared que impide moverse entre las dos casillas que separa dicha pared (en caso contrario no sería un verdadero laberinto).*

Implementa un subprograma que dados:

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

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



Tenemos un laberinto de $NxNxN$ casillas, donde nos podemos mover en 6 direcciones diferentes, también encontramos las paredes del mismo y como queremos obtener el camino y coste mínimo de ir desde la casilla de entrada a la casilla de salida, primero tendremos que construir el grafo (convirtiendo así una casilla en un vértice del mismo), luego rellenaremos con los movimientos del jugador y las paredes y por último heremos uso del algoritmo de Dijkstra para poder calcular tanto el camino como el coste mínimo.

```
//Definimos los tipos de datos a usar
typedef std::pair<vector<size_t>,size_t> Camino_Coste;
typedef struct Casilla{
  size_t x, y, z;
};

Camino_Coste Laberinto3D(size_t N, vector<Casilla> &paredes, Casilla
↪  Entrada, Casilla Salida){
  //Primero de todo creamos el grafo no dirigido.
  GrafoP<size_t> GLaberinto(N*N*N);
  //Rellenamos el laberinto con los movimientos
  for(size_t i = 0; i < N; i++){
    for(size_t j = 0; j < N; j++){
      for(size_t k = 0; k < N; k++){
        //Nos quedamos con la casilla actual
        size_t casilla_actual = CasillaToVertice(i,j,k,N);
        if(i+1 < N){ //Derecha
          GLaberinto[casilla_actual][CasillaToVertice(i+1,j,k,N)] =
          ↪  GLaberinto[CasillaToVertice(i+1,j,k,N)][casilla_actual] =
          ↪  1;
        }
        if(i-1 >= 0){ //Izquierda
```

```
        GLaberinto[casilla_actual][CasillaToVertice(i-1,j,k,N)] =
        ↪  GLaberinto[CasillaToVertice(i-1,j,k,N)][casilla_actual] =
        ↪  1;
      }
      if(j+1 < N){ //Abajo
        GLaberinto[casilla_actual][CasillaToVertice(i,j+1,k,N)] =
        ↪  GLaberinto[CasillaToVertice(i,j+1,k,N)][casilla_actual] =
        ↪  1;
      }
      if(j-1 >= 0){//Arriba
        GLaberinto[casilla_actual][CasillaToVertice(i,j-1,k,N)] =
        ↪  GLaberinto[CasillaToVertice(i,j-1,k,N)][casilla_actual] =
        ↪  1;
      }
      if(k+1 < N){ //Atrás
        GLaberinto[casilla_actual][CasillaToVertice(i,j,k+1,N)] =
        ↪  GLaberinto[CasillaToVertice(i,j,k+1,N)][casilla_actual] =
        ↪  1;
      }
      if(k-1 >= 0){ //Delante
        GLaberinto[casilla_actual][CasillaToVertice(i,j,k-1,N)] =
        ↪  GLaberinto[CasillaToVertice(i,j,k-1,N)][casilla_actual] =
        ↪  1;
      }
    }
  }
}
//Ahora rellenamos el grafo con las piedras
for(Casilla &pared : paredes){
  for(size_t i = 0; i < GLaberinto.numVert(); i++){
    GLaberinto[i][CasillaToVertice(pared.x,pared.y,pared.z,N)] =
    ↪  GLaberinto[CasillaToVertice(pared.x,pared.y,pared.z,N)][i] =
    ↪  GrafoP<size_t>::INFINITO;
  }
}

//Ya tenemos el grafo con las casillas accesibles y no accesibles, por
↪  lo que vamos a proceder a obtener el camino y coste mínimo de ir
↪  desde la entrada hacia la salida.
size_t CEntrada = CasillaToVertice(Entrada.x,Entrada.y,Entrada.z, N);
size_t CSalida = CasillaToVertice(Salida.x, Salida.y, Salida.z, N);

vector<size_t>Vertices(GLaberinto.numVert()),
↪  CostesMinimos(GLaberinto.numVert());

CostesMinimos = Dijkstra(GLaberinto,CEntrada,Vertices);
size_t Coste = CostesMinimos[CSalida]; //Coste del camino
```

```
  //Ahora vamos a obtener el camino
  vector<size_t>Camino; //Camino Salida-Entrada
  size_t actual = CSalida;
  while(actual != CEntrada){
    Camino.push_back(actual);
    actual = Vertices[actual];
  }
  Camino.push_back(CEntrada);
  std::reverse(Camino.begin(), Camino.end()); //Camino Entrada-Salida.

  return{Camino,Coste}; //Devolvemos el camino y su coste mínimo.
}

//Función que dada una casilla nos devuelve un vértice del grafo (size_t)
size_t CasillaToVertice(size_t i, size_t j, size_t k, size_t N){
  return (i*(N*N) + j*N + k);
}
```

He hecho uso de las siguientes funciones de los grafos:

- `GrafoP<tCoste> GrafoP(size_t n);`
  - *Post*: Crea y devuelve un grafo ponderado de tamaño n.
- `size_t numVert()const;`
  - *Post*: Devuelve el número de vértices de un grafo.
- `vector<tCoste>& operator[](vertice v);`
  - *Post*: Devuelve un vector con los costes mínimos de las aristas adyacentes la vértice v.
- `size_t GrafoP<tCoste>::INFINITO = std::numeric_limits<T>::max();`
  - *Post*: Devuelve el valor máximo permitido.
- ```
  vector<tCoste> Dijkstra(const GrafoP<tCoste> &G,
      typename GrafoP<tCoste>::vertice Origen,
      vector<typename GrafoP<tCoste>::vertice> &P);
  ```
  - *Pre*: Recibe un grafo ponderado y un vector de vértices.
  - *Post*: Devuelve dos vectores, uno con el coste del camino desde el origen a cualquier vértice del grafo y el otro que contiene los vértices por los que pasa el camino.
  ```

Matriz Fauno (Junio 2023)

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

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

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

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

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

Dado los siguientes parámetros:

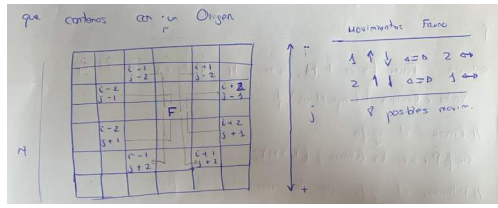
- *Dimensiones de Narnia N y M.*
- *Relación de casillas trampa.*
- *Relación de casillas ocupadas por caballeros.*

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

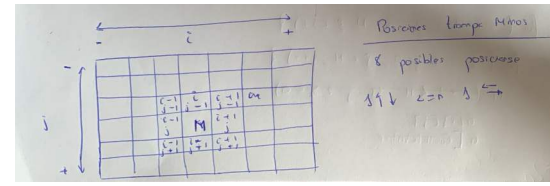
Contamos con una matriz de NxM casillas, donde el fauno solo se puede mover como el caballo del ajedrez (L), también sabemos que tenemos dos casilla, la de entrada (0,0) y la de salida (N-1,M-1). En la matriz también encontramos una serie de trampas a las cuales si el fauno accede morirá, lo mismo pasa con el caballero que si accedemos a las casillas adyacentes a donde se encuentra, el fauno morirá también.

Como nos pide devolver si ha llegado o no el fauno a la salida y el coste del recorrido haremos uso del algoritmo de Dijkstra ya que este devuelve el camino y coste mínimo de ir desde un origen a cualquiern vértice del grafo, pero para poder hacer esto tenemos que trabajar con un Grafo ponderado, no con una matriz, por lo que tendremos que buscar una manera de convertir una casilla en un vértice/nodo del grafo.

Ahora vamos a ver dentro de la matriz los movimientos que puede hacer el fauno y como se colocarían las trampas de los caballeros (minos):



Movimientos posibles del fauno.



Posiciones de las trampas de los caballeros (minos).

```
//Definición de los tipos de datos a usar
typedef std::pair<size_t,size_t>Casilla;
typedef pair<bool,size_t> Solucion; //lo que nos pide a devolver

//Método que nos convierte una Casilla en un vértice del grafo
size_t CasillatoVertice(size_t i, size_t j, size_t M){
  return i*M+j; //como es de dos dimensiones, si multiplicamos la filas
  ↪  por el número de columnas + las columnas obtendremos esa posición
  ↪  como un vértice del grafo.
}

Solucion Fauno(size_t N, size_t M, vector<Casilla> &trampas,
↪  vector<Casilla> &trampasMinos){
  GrafoP<size_t>GFauno(N*M); //Grafo de N*M vértices
  //Rellanamos el grafo con los movimientos del fauno
  for(size_t i = 0; i < M; i++)
    for(size_t j = 0; j < N; j++){
      //guardamos la casilla en la que nos encontramos
      size_t actual = CasillatoVertice(i,j,M);
      if(i+1 < M && j-2 >= 0)
        GFauno[actual][CasillatoVertice(i+1,j-2,M)]=
          GFauno[CasillatoVertice(i+1,j-2,M)][actual] = 1;
      if(i+2 < M && j-1 >= 0)
        GFauno[actual][CasillatoVertice(i+2,j-1,M)]=
          GFauno[CasillatoVertice(i+2,j-1,M)][actual] = 1;
      if(i+2 < M && j+1 < N)
        GFauno[actual][CasillatoVertice(i+2,j+1,M)]=
          GFauno[CasillatoVertice(i+2,j+1,M)][actual] = 1;
      if(i+1 < M && j+2 < N)
        GFauno[actual][CasillatoVertice(i+1,j+2,M)]=
          GFauno[CasillatoVertice(i+1,j+2,M)][actual] = 1;
      if(i-1 >= 0 && j+2 < N)
        GFauno[actual][CasillatoVertice(i-1,j+2,M)]=
          GFauno[CasillatoVertice(i-1,j+2,M)][actual] = 1;
      if(i-2 >= 0 && j+1 < N)
```

```
      GFauno[actual][CasillatoVertice(i-2,j+1,M)]=
        GFauno[CasillatoVertice(i-2,j+1,M)][actual] = 1;
    if(i-2 >= 0 && j-1 >= 0)
      GFauno[actual][CasillatoVertice(i-2,j-1,M)]=
        GFauno[CasillatoVertice(i-2,j-1,M)][actual] = 1;
    if(i-1 >= 0 && j-2 >= 0)
      GFauno[actual][CasillatoVertice(i-1,j-2,M)]=
        GFauno[CasillatoVertice(i-1,j-2,M)][actual] = 1;
  }
//Ya tenemos el grafo relleno con los movimientos del caballo, ahora
↪  vamos a colocar las trampas
for(auto & t : trampas){
  size_t pos_trampa = CasillatoVertice(t.first,t.second,M);
  for(size_t i = 0; i < GFauno.numVert(); i++){
    GFauno[i][pos_trampa] = G[pos_trampa][i] = GrafoP<size_t>::INFINITO;
  }
}
//Ahora vamos a colocar las trampas de los caballeros
for(auto &mt : trampasMinos){
  for(size_t i = 0; i < GFauno.numVert(); i++){
    if(mt.first >=0 && mt.second-1 >= 0)
      GFauno[i][CasillatoVertice(tm.first,tm.second-1,M)] =
        GFauno[CasillatoVertice(tm.first,tm.second-1,M)][i] =
        ↪  GrafoP<size_t>::INFINITO;
    if(mt.first+1 < M && mt.second-1 >= 0)
      GFauno[i][CasillatoVertice(tm.first+1,tm.second-1,M)] =
        GFauno[CasillatoVertice(tm.first+1,tm.second-1,M)][i] =
        ↪  GrafoP<size_t>::INFINITO;
    if(mt.first+1 < M && mt.second >= 0)
    GFauno[i][CasillatoVertice(tm.first+1,tm.second,M)] =
      GFauno[CasillatoVertice(tm.first+1,tm.second,M)][i] =
      ↪  GrafoP<size_t>::INFINITO;
    if(mt.first+1 < M && mt.second+1 < N)
      GFauno[i][CasillatoVertice(tm.first+1,tm.second+1,M)] =
        GFauno[CasillatoVertice(tm.first+1,tm.second+1,M)][i] =
        ↪  GrafoP<size_t>::INFINITO;
    if(mt.first >= 0 && mt.second+1 < N)
      GFauno[i][CasillatoVertice(tm.first,tm.second1,M)] =
        GFauno[CasillatoVertice(tm.first,tm.second+1,M)][i] =
        ↪  GrafoP<size_t>::INFINITO;
    if(mt.first-1 >= 0 && mt.second+1 < N)
      GFauno[i][CasillatoVertice(tm.first-1,tm.second+1,M)] =
        GFauno[CasillatoVertice(tm.first-1,tm.second+1,M)][i] =
        ↪  GrafoP<size_t>::INFINITO;
    if(mt.first-1 >= 0 && mt.second >= 0)
      GFauno[i][CasillatoVertice(tm.first-1,tm.second,M)] =
```

```
            GFauno[CasillatoVertice(tm.first-1,tm.second,M)][i] =
             ↪  GrafoP<size_t>::INFINITO;
        if(mt.first-1 >= 0 && mt.second-1 >= 0)
          GFauno[i][CasillatoVertice(tm.first-1,tm.second-1,M)] =
            GFauno[CasillatoVertice(tm.first-1,tm.second-1,M)][i] =
             ↪  GrafoP<size_t>::INFINITO;
      }
    }
    //Ya tenemos el grafo relleno con todo, asi que hora podemos calcular el
     ↪  camino de coste mínimo desde el Origen (0,0), hasta el
     ↪  Destino(N-1,M-1);
    vector<size_t>Vertices(GFauno.numVert()),
                  CostesMinimos(GFauno.numVert());
    CostesMinimos = Dijkstra(GFauno,0,Vertices);
    size_t Destino = CasillatoVertice(N-1,M-1,M);
    Solucion sol; //Solución del ejercicio.
    if(CostesMinimos[Destino] != GrafoP<size_t>::INFINITO){
      sol.first = true;
      sol.second = CostesMinimos[Destino];
    }
    else{
      sol.first = false;
      sol.second = 0; //devolvemos 0 si no se ha conseguido llegar al final.
    }
    return sol;
}
```

Hemos hecho uso de las operaciones de los grafos y de los prototipos de las funciones:

- `GrafoP<size_t> GrafoP(size_t n)`
  - *Post*: Crea y devuelve un grafo ponderado de tamaño n
- `vector<size_t>& operator[](vertice v)`
  - *Post*: Devuelve un vector con las aristas adyacentes al vértice v
- `size_t numVert()const;`
- `size_t GrafoP<size_t>::INFINITO = std::numeric_limits:max();`
  - *Post*: Devuelve el máximo valor permitido.
- `vector<tCoste> Dijkstra(const GrafoP<tCoste> &G,`
  `typename GrafoP<tCoste>::vertice Origen,`
  `vector<typename GrafoP<tCoste>::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado, un vértice origen del camino y un vector de vértices.
  - *Post*: Devuelve dos vectores, uno con el coste mínimo del camino y el otro, por referencia, que es el camino.

Viajes Alergias

Enunciado: *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.

« Este ejercicio pertenece a la Práctica 7, Ejercicio 5. »

Partimos de que tenemos 3 grafos (Tren, Bus y Avión), que representan los 3 medios de transporte que el viajero puede usar. Este viajero es alergico a uno de los 3 medios de transporte por lo que las combinaciones de medios de transporte disminuyen. También sabemos que tiene una cantidad de dinero determinada (para poder realizar los viajes) y una ciudad de origen del mismo.

Sabiendo esto, podemos intuir que algoritmo podemos usar, como tenemos una ciudad origen podemos hacer uso de Dijkstra.

```
//Definimos los tipos de datos a usar
typedef vector<bool> CiudadesViajero;

CiudadesViajero ViajesAlergias(const GrafoP<size_t> &Tren, const
↪  GrafoP<size_t> &Bus, const GrafoP<size_t> &Avion, size_t DineroTotal,
↪  size_t Origen, const string alergia){
  //Como contamos con 3 grafos, podemos crear un supergrafo pero solamente
  ↪  con aquello a los que no es alérgico nuestro viajero
  GrafoP<size_t>GrafoViajero(Tren.numVert());
  if(alergia == "Tren")
    RellenarGrafo(Bus,Avion,GrafoViajero);
  else if(alergia == "Bus")
    RellenarGrafo(Tren,Avion,GrafoViajero);
  else if(alergia == "Avion")
    RellenarGrafo(Tren,Bus,GrafoViajero);

  //Ahora que tenemos relleno el supergrafo dependiendo de la alergia,
  ↪  podemos calcular los costes mínimos del Grafo
  vector<size_t>Vertices(GrafoViajero.numVert()),
                CostesMinimos(GrafoViajero.numVert());

  CostesMinimos = Dijkstra(GrafoViajero,Origen,Vertices);

  //Ahora dependiendo del dinero del viajero, vamos a obtener las ciudades
  ↪  a las que puede ir.
```

```
  size_t dinero = DineroTotal;
  vector<bool>DestinosAlcanzables(GrafoViajero.numVert(), false);
  for(size_t i = 0; i < CostesMinimos.size(); i++){
    if(CostesMinimos[i] <= dinero)
      DestinosAlcanzables[i] = true;
  }
  DestinosAlcanzables[Origen] = true;

  //devolvemos las ciudades DestinosAlcanzables
  return DestinosAlcanzables;
}

//función que nos rellena el supergrafo, con el menor coste de los 2.
void RellenarGrafo(const GrafoP<size_t> &Transporte1, const
↪  GrafoP<size_t> &Transporte2, GrafoP<size_t> &Supergrafo){
  //Vamos a rellear el supergrafo
  for(size_t i = 0; i < Supergrafo.numVert(); i++)
    for(size_t j = 0; j < Supergrafo.numVert(); j++){
      if(Transporte1[i][j] != GrafoP<size_t>::INFINITO ||
      ↪  Transporte2[i][j] != GrafoP<size_t>::INFINITO)
        Supergrafo[i][j] = Supergrafo[j][i] =
        ↪  std::min(Transporte1[i][j],Transporte2[i][j]);
    }
}
```

Hemos hecho uso de las siguiente operaciones y prototipos de los grafos:

- `GrafoP<tCoste> GrafoP(size_t n);`
  - *Post*: Crea y devuelve un grafo ponderado de tamañom n.
- `size_t numVert()const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `vector<size_t> Dijkstra(const GrafoP<tCoste> &G, typename GrafoP<tCoste>::vertice, vector<typename GrafoP <tCoste>::vertice> &P);`
  - *Pre*: Recibe un grafo ponderado y un vector de vértices.
  - *Post*: Devuelve dos vectores, uno con los costes mínimos desde el origen hacia cualquier otro vértice del grafo y otro con los vértices por los que pasa el camino de coste mínimo.

Viajes Tren, Bus y Avión.

Enunciado: *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.

« Este ejercicio pertenece a la Práctica 7, Ejercicio 10. »

Contamos con 3 grafo (los 3 medios de transporte), las ciuades origen y destino del viaje y los costes del taxi (tanto para cambiar de tren-bus como para cambiar aeropuerto-tren, aeropuerto-bus).

Sabiendo esto y que queremos obtener el camino y coste mínimo de ir desde la ciudad Origen a la Destino, haremos uso de Dijkstra.

Como no nos obligan a cambiar de medio de transporte, el camino se puede realizasr de varias maneras:

- Viaje 1: Todo tren.
- Viaje 2: Todo bus.
- Viaje 3: Todo avión.
- Viaje 4: Tren - taxi - Bus.
- Viaje 5: Tren - taxi - Avión.
- Viaje 6: Bus - taxi - Tren.
- Viaje 7: Bus - taxi - Avión.
- Viaje 8: Avion - taxi - Tren.
- Viaje 9: Avion - taxi - Bus.

Para poder hacer Dijkstra de estos viajes, tendremos que crear un Supergrafo del tamaño 3*N. Siendo nuestro supergrafo:

	TREN	**BUS**	**AVION**
TREN	GRAFO TREN	TAXI TREN-BUS	TAXI TREN-AVION
BUS	TAXI BUS-TREN	GRAFO BUS	TAXI BUS-AVION
AVION	TAXI AVION-TREN	TAXI AVION-BUS	GRAFO AVION

```
//Definimos los tipos de datos a usar
typedef std::pair<vector<size_t>,size_t> Camino_Coste_Viaje;
↪  //Camino-Costes

Camino_Costes_Viaje Viaje(const GrafoP<size_t>& Tren, const
↪  GrafoP<size_t> &Bus, const GrafoP<size_t> &Avion, size_t taxiTB,
↪  size_t taxiAeropuerto, size_t Origen, size_t Destino){
  //Primero vamos a unir todos los grafos dados, en nuestro supergrafo
  size_t N = tren.numVert();
  GrafoP<size_t> GrafoViaje(3*N); //Como todos tienen el mísmo número de
  ↪  vertices podemos inicializarlo con cualquiera.

  //Ahora vamos a rellenar el Supergrafo_
  for(size_t i = 0; i < N; i++){
    for(size_t j = 0; j < N; j++){
      GrafoViaje[i][j] = Tren[i][j]; //Cuadrante solo Tren
      GrafoViaje[i+N][j+N] = Bus[i][j]; //Cuadrante solo Bus
      GrafoViaje[i+2*N][j+2*N] = Avion[i][j]; //Cuadrante solo Tren.
    }
  }
  //Ahora vamos a rellenar con los costes de los taxis.
  for(size_t i = 0; i < N; i++){
    GrafoViaje[i+N][j] = GrafoViaje[i][i+N] = taxiTB; //Tren-Bus y
    ↪  Bus-Tren.
    GrafoViaje[i+2*N][i] = GrafoViaje[i][i+2*N] = taxiAeropuerto;
    ↪  //Tren-Avion y Avion-Tren.
    GrafoViaje[i+2*N][i+N] = GrafoViaje[i+N][i+2*N] = taxiAeropuerto;
    ↪  //Bus-Avión y Avión-Bus.
  }

  //Calculamos ahora los costes mínimos con Dijkstra
  vector<size_t> Vertices(GrafoViaje.numVert()),
  ↪  CostesMinimos(GrafoViaje.numVert());
  CostesMinimos = Dijkstra(GrafoViaje,Origen,Vertices);

  size_t CosteDestino = CostesMinimos[Destino];

  //Ahora vamos a recuperar el camino realizado por el viajero.
  vector<size_t> Camino;
  size_t actual = Destino;
  while(actual != Origen){
    Camino.push_back(actual);
    actual = Vertices[actual];
  }
  Camino.push_backK(Origen);
  reverse(Camino.begin(),Camino.end()); //invertimos el vector para que
  ↪  tengamos el camino de manera Origen-Destino.
```

```
  //Devolvemos el camino junto con su coste
  return {Camino,CosteDestino};
}
```

He hecho uso de las siguiente funciones y prototipos:

- `size_t numVert()const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `GrafoP<tCoste> GrafoP(size_t n);`
  - *Post*: Crea y devuelve un grafo ponderado de tamaño n.
- `vector<tCoste>& operator[](vertice v);`
  - *Post*: Devuelve el conjunto de costes de las aristas adyacentes del vértice v.
- ```
  vector<tCoste> Dijkstra(const GrafoP<tCoste> &G,
      typename GrafoP<tCoste>::vertice Origen,
      vector<typename GrafoP<tCoste>::vertice &P);
  ```
  - *Pre*: Recibe un grafo ponderado, un vértice origen y un vector de vértices.
  - *Post*: Devuelve dos vectores, uno con el coste mínimo de ir desde el vértice origen a cualquier vértice del grafo y otro con los vértices de dicho camino.
  ```