

Práctica 3: la clase GRAFO.

Profesor responsable: Sergio Alonso.

Dificultad: media.

Tutorización: semana del 27 de marzo

Corrección: semana del 10 de abril

1. Objetivo

El objetivo de esta práctica es escribir una librería en c++ que defina una estructura de datos para almacenar grafos, una serie de métodos para su carga a partir de sus sucesores o predecesores, en el caso de los grafos dirigidos, y con las adyacencias, en el caso de los grafos no dirigidos, además de implementar algunos procedimientos que pongan a prueba la clase grafo creada. Es decir, crearemos la clase GRAFO que, desde esta primera práctica, estará dotada de lo esencial para poder codificar los grafos y trabajar con ellos en la resolución de distintos problemas de optimización combinatoria con la implementación de diversos algoritmos. En esta primera práctica, añadiremos distintas implementaciones sencillas de recorridos para poner a prueba la programación de la clase. Para la gestión de la clase GRAFO, usaremos un sencillo programa principal que, en forma de menú, irá incorporando las utilidades que añadamos a la clase GRAFO.

2. Plan de trabajo

Esta práctica se divide en dos fases. En la inicial, se centrará en escribir y comprobar el buen funcionamiento del constructor y del actualizador de la clase GRAFO, leyendo la información desde fichero de distintas instancias de grafos, tanto dirigidos como no dirigidos. El constructor y su actualizador, analizarán la información del fichero de texto de partida, y según sea dirigido o no dirigido, implementarán su carga en el objeto de la clase GRAFO a través de la estructura de sucesores o predecesores (para grafos dirigidos o digrafos) o usando la lista de adyacencia (grafos no dirigidos). Para ello, podrás descargar ficheros de c++ como plantillas, descargables desde el campus virtual, para ser estudiados como propuesta inicial de trabajo y que deberán ser finalizados por el alumnado.

Una vez comprobado el buen funcionamiento del constructor y su actualizador, en la segunda fase, deberás implementar recorridos sobre grafos, en profundidad o en amplitud, lo que permitirá analizar el grafo cargado pero también comprobar su correcto funcionamiento.

Esto es, durante la semana del 27 de marzo, justo antes de Semana Santa se explicarán durante la hora de laboratorio los detalles de la implementación de la clase GRAFO y su programa de test, así como las implementaciones de los recorridos, en amplitud y en profundidad. La semana del 10 de abril será de corrección y evaluación del trabajo en la práctica.

Semana	del 27 de marzo	del 3 de abril	del 10 de abril
	Tutorización de la práctica	Semana Santa	Corrección de la práctica 3

3. Primera fase

3.1. Implementación, codificación y primeras estructuras

La estructura del programa a implementar será la de un menú de opciones, con la característica básica de que tales opciones serán distintas según el grafo de trabajo sea dirigido o no dirigido.

Por tanto, el programa necesita cargar un grafo desde un fichero de texto para poder iniciar el menú de opciones, ya que debe analizarlo y cargarlo correctamente en la estructura de la clase grafo. El fichero de texto con la información del grafo problema ha de tener el siguiente formato para su correcta lectura:

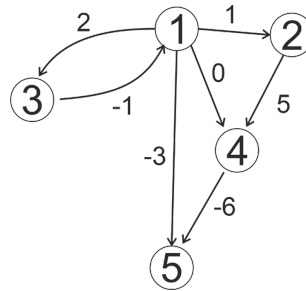
```
n m esdirigido?  
i1 j1 c1  
...  
im jm cm
```

donde n es el número de nodos, m el número de arcos o aristas y `esdirigido?` nos indica si el grafo es dirigido situando un 1, o si no lo es situando un 0. Las siguientes m líneas nos indican los arcos o aristas presentes en el grafo a través de los nodos o vértices y su coste, esto es, lo que cuesta usarlo en la solución. Este atributo, el coste, será importante para la siguiente práctica.

El grafo se lee del fichero, pero se almacena su lista de sucesores y su lista de predecesores, en el caso de un grafo dirigido, y su lista de adyacencia en el caso de un grafo no dirigido, por lo que es clave la información que aparece en la primera línea del fichero de texto.

El constructor del objeto grafo, debe implementar bajo c++, estas listas. Vamos a ilustrarlo con un ejemplo, trabajando inicialmente la codificación del mismo en un fichero de texto, y analizaremos las estructuras en c++ necesarias para almacenar las listas de forma eficiente.

Trabajamos con el grafo dirigido siguiente:



Si codificamos el grafo anterior en un fichero de texto, tal y como se ha expresado, tendríamos:

```

5 7 1
1 2 1
1 3 2
1 5 -3
1 4 0
2 4 5
3 1 -1
4 5 -6
  
```

Cuyo conjunto de sucesores, sin la información de los pesos, sería, por tanto:

$\Gamma_1^+ = \{2, 3, 5, 4\}$
 $\Gamma_2^+ = \{4\}$
 $\Gamma_3^+ = \{1\}$
 $\Gamma_4^+ = \{5\}$
 $\Gamma_5^+ = \emptyset$

La estructura mediante vectores de c++ que hemos de construir para implementar la codificación del grafo de partida a través de la lista de sucesores, tendría esta *idealización* gráfica:

	0	1	2	3
0	1 1	2 2	4 -3	3 0
1	3 5			
2	0 -1			
3	4 -6			
4				

Esto es, para almacenar la información del grafo en listas de sucesores, predecesores o adyacencia, usaremos un objeto perteneciente a la clase GRAFO que será definida como un vector de vectores de registros. El hecho de usar esta estructura nos provee de eficiencia en los recursos, puesto que sólo usamos la memoria que necesitamos para almacenar la información. Si usáramos una matriz, un vector bidimensional, estaríamos reservando memoria que no aseguremos que usemos.

Esto es a donde queremos llegar, pero como dijimos, iremos despacio. En poco todo se aclara. **Paciencia.** Hemos de explicar con detalle, descubriendo paso a paso la propuesta, y para hacerlo debemos transitar desde el interior hacia el exterior. Primero definimos la estructura, struct, que llamaremos ElementoLista, con dos miembros, en uno de ellos, .j, se almacenaría el nodo sucesor, predecesor,

Optimizaci3n

adyacente, seg3n se trate, y en el otro, .c, almacenaremos otro atributo del arco o arista, en este caso, situaremos un coste o peso, de tipo int. Esta estructura la podemos hacer crecer, dependiendo del problema a resolver o de las necesidades de implementaci3n del algoritmo.

```
typedef struct
{
    unsigned j; // nodo
    int c; // atributo para expresar su peso, longitud, coste }
ElementoLista;
```

Un ElementoLista podr3amos idealizarlo gráficamente como un eslab3n con la siguiente forma:

3	0
---	---

Es decir, la codificaci3n del nodo sucesor, predecesor o adyacente - en este caso aparece el valor unsigned 3 -, y el coste, - en este caso aparece el valor int 0 -. A continuaci3n construimos el vector de estos registros ElementoLista, sabiendo que las listas de sucesores, predecesores o adyacencia son vectores de 3stos.

```
typedef vector<ElementoLista> LA_nodo;
```

Esto es, mediante un vector del tipo LA_nodo, podemos almacenar la informaci3n de los sucesores de un nodo, los predecesores de un nodo o los adyacentes de un nodo. En tal sentido, podemos idealizarlo gráficamente, como una cadena cuyos eslabones son ElementoLista y que tendr3an la idealizaci3n gráfica siguiente:

Esto es, LA_nodo es un vector, y mostramos gráficamente sus cuatro posiciones: 0, 1, 2 y 3. Como sabemos, los vectores en c++ se indexan por posiciones que comienzan en 0, no en 1. Es por lo que, cuando guardemos en la estructura de la lista de sucesores, predecesores o adyacencia, la informaci3n de los nodos, siempre restaremos una unidad. As3, la informaci3n que guardamos en este LA_nodo, 1, 2, 4 y 3, corresponde a los nodos, 2, 3, 5 y 4.

	0	1	2	3
0	1	1	2	2
	4	-3	3	0

Por tanto, se comprueba que el vector que mostramos gráficamente corresponde a los sucesores del nodo 1 en el grafo que estamos usando como ejemplo.

Finalmente, un vector de LA_nodo, ser3a suficiente para almacenar la informaci3n de toda la lista de sucesores, o toda la lista de predecesores, en el caso de un grafo dirigido, o de toda la lista de adyacencia, en el caso de un grafo no dirigido.

```
vector<LA_nodo> LS; // lista de sucesores o de adyacencia
```

Veamos sobre el grafo de trabajo todo a la vez:

$$\Gamma_1^+ = \{2, 3, 5, 4\}$$

$$\Gamma_2^+ = \{4\}$$

$$\Gamma_3^+ = \{1\}$$

$$\Gamma_4^+ = \{5\}$$

$$\Gamma_5^+ = \emptyset$$

	0	1	2	3
0	1 1	2 2	4 -3	3 0
1	3 5			
2	0 -1			
3	4 -6			
4				

Con el esquema superior de referencia ¿Cómo acceder a los sucesores del nodo 1?

Los sucesores del nodo 1 se sitúan en el vector $LS[1-1]$, esto es, $LS[0]$, que tiene 4 elementos ($LS[0].size()$ es 4); por tanto, accedemos a cada una de las esas cuatro posiciones: $LS[0][0]$, $LS[0][1]$, $LS[0][2]$, $LS[0][3]$, que son de tipo `ElementoLista`. Gráficamente, sería:

$LS[0][0]$	0	1	2	3
0	1 1	2 2	4 -3	3 0
$LS[0][1]$	0	1	2	3
0	1 1	2 2	4 -3	3 0
$LS[0][2]$	0	1	2	3
0	1 1	2 2	4 -3	3 0
$LS[0][3]$	0	1	2	3
0	1 1	2 2	4 -3	3 0

Y como cada uno de ellos es de tipo `ElementoLista`, tienen dos miembros: encontramos en $LS[0][0].j$, $LS[0][1].j$, $LS[0][2].j$, $LS[0][3].j$, los valores siguientes, 1, 2, 4 y 3, respectivamente, que corresponden a los nodos, 2, 3, 5 y 4, y por tanto, los arcos (1, 2), (1, 3), (1, 5) y (1, 4).

	0	1	2	3
0	1 1	2 2	4 -3	3 0

Y encontramos en $LS[0][0].c$, $LS[0][1].c$, $LS[0][2].c$, $LS[0][3].c$ los costes de esos arcos, esto es, 1, 2, -3 y 0, respectivamente.

	0	1	2	3
0	1 1	2 2	4 -3	3 0

En cambio, no hay sucesores al nodo 5, pues al ir a buscar en la posición $5-1=4$, vemos que $LS[4].size()$ nos devuelve el valor 0.

4

En resumen ¿De qué tipos son?

LS[2] es un vector de ElementoLista, que tiene tamaño 1; almacena los sucesores del nodo 3.

LS[2][0] es un registro del tipo ElementoLista; para acceder a su información, usamos LS[2][0].j que es de tipo unsigned, y LS[2][0].c, que almacenaría el coste del arco (3, LS[2][0].j+1), es decir, el arco (3, 1), que es -1.

3.2. La clase GRAFO

La clase GRAFO incluye la información necesaria para la gestión de los datos, su manipulación y se prepara para incluir como métodos, los procedimientos que implementan los algoritmos que resolverán algunos de los problemas que trataremos en la asignatura. Inicialmente, para esta primera parte del trabajo, es esta:

```
class GRAFO
{
    unsigned dirigido;
    unsigned n;
    unsigned m;
    vector<LA_nodo> LS;
    vector<LA_nodo> LP;
    vector<LA_nodo> A;
    void destroy();
    void build (char nombrefichero[85], int &errorapertura);
public:
    GRAFO(char nombrefichero[], int &errorapertura);
    void actualizar (char nombrefichero[], int &errorapertura);
    unsigned Es_dirigido();
    void Info_Grafo();
    void Mostrar_Listas(int l);
    void Mostrar_Matriz();
    ~GRAFO();
};
```

Para implementar el constructor del objeto GRAFO, usaremos un método privado:

```
void build (char nombrefichero[85], int &errorapertura);
```

Este método se encarga de leer la codificación del grafo desde el fichero de texto con nombre nombrefichero, desde un tipo ifstream y de asignar los atributos y las estructuras de la clase GRAFO, construyendo un objeto que denominaremos G, y devolviendo 0 en errorapertura si no ha habido incidencia alguna, y 1 si la ha habido. Con este valor devuelto, podremos saber si la carga del grafo ha sido correcta, y mostrar el menú de acciones.

¿C3mo abrir y leer de un fichero de texto?

Usaremos la librería `fstream` para poder gestionar ficheros de entrada de datos, `ifstream`. As3, las primeras l3neas de c3digo del constructor deber3n incluir:

```
ifstream textfile; //definimos el objeto textfile como ifstream
textfile.open(nombrefichero); // abrimos el fichero para lectura

if (textfile.is_open()) //verificamos que se ha accedido correctamente
/*leemos por conversi3n impl3cita el n3mero de nodos, arcos y el
atributo dirigido en la primera l3nea de fichero de texto con la
informaci3n del grafo problema */
    textfile >> (unsigned &) n >> (unsigned &) m >> (unsigned &) dirigido;
/* recuerda: los nodos internamente se numeran desde 0 a n-1 */
/* creamos la lista de sucesores... sigue leyendo*/
```

Tras la lectura de la primera l3nea, ya conocemos la dimensi3n del grafo, esto es, el n3mero de nodos, el n3mero de arcos o aristas, - lo cual nos indica el n3mero de l3neas que a3n hemos de leer del fichero -, y el tipo de grafo que es: dirigido o no dirigido. Si es dirigido, debemos construir tanto la lista de sucesores como la de predecesores, que se almacenar3n en `LS` y `LP`, respectivamente.

Conocido `n`, podemos dimensionar `LS` con el m3todo `resize`, esto es, `LS.resize(n)`, obteniendo el acceso a los vectores vac3os, `LS[0]`, `LS[1]`, ..., `LS[n-1]`. Se trata de guardar en `LS[i-1]` la informaci3n de los sucesores del nodo `i`.

Para ello, cada vez que leamos una l3nea del fichero de texto de la forma `i j c` usamos una variable auxiliar, que podemos llamar `dummy`, del tipo `ElementoLista` para asignar estos dos3ltimos valores. Luego, con el m3todo `push_back`, introducimos la informaci3n del sucesor en `LS[i-1]`, esto es, `LS[i-1].push_back(dummy)`, donde `dummy.j` es `j-1` y `dummy.c` es `c`.

Por tanto, insistimos, es importante saber que, si bien el usuario trabaja con un conjunto de nodos $\{1, 2, 3, 4, \dots, n\}$, en la estructura GRAFO, la informaci3n que se almacena trabaja, respectivamente, con el conjunto $\{0, 1, 2, 3, \dots, n-1\}$. Esto ha de tratarse con cuidado porque puede ser fuente de errores.

¿Qu3 ocurre con los grafos no dirigidos?

Es importante saber que, en el caso de los grafos no dirigidos, la adyacencia es sim3trica. Esto es, si leemos del fichero de texto la arista (i, j) , debemos situar `j` como adyacente de `i`, pero tambi3n a `i` como adyacente de `j`, excepto en el caso de que se trate de un bucle. Es decir, en el caso de (i, i) s3lo hay que realizar una inserci3n con el `push_back`. La informaci3n de la lista de adyacencia se almacena en `LS`, y `LP` no se usa.

¿Por tanto, cómo se implementaría el método build()?

Si unimos lo expresado hasta el momento, el método comenzaría de la forma siguiente:

```
void GRAFO :: build (char nombrefichero[85], int &errorapertura)
{
    ElementoLista dummy;
    ifstream textfile;
    textfile.open(nombrefichero);
    if (textfile.is_open())
    {
        unsigned i, j, k;
        // leemos por conversión implícita el número de nodos, arcos y el atributo dirigido
        textfile >> (unsigned &) n >> (unsigned &) m >> (unsigned &) dirigido;
        // los nodos internamente se numeran desde 0 a n-1
        // creamos las n listas de sucesores
        LS.resize(n);
        // leemos los m arcos
        for (k=0;k<m;k++)
        {
            textfile >> (unsigned &) i >> (unsigned &) j >> (int &) dummy.c;
            //damos los valores a dummy.j;
            //situamos en la posición de LS que corresponda a dummy mediante push_back()
            //¿Cómo construimos LS en el caso de un grafo no dirigido?
            //atención: ¿cómo ir construyendo LP?

            ...
        }
    }
}
```

Una vez terminado, el constructor sería, simplemente:

```
GRAFO::GRAFO(char nombrefichero[85], int &errorapertura)
{
    build (nombrefichero, errorapertura);
}
```


Por otro lado, para limpiar y liberar la memoria en donde se almacena el grafo, LS, y LP si el grafo es dirigido, usamos el sencillo m3todo `destroy()`.

```
void GRAFO :: destroy()
{
    for (unsigned i=0; i< n; i++)
    {
        LS[i].clear();
        if (dirigido == 1)
        {
            LP[i].clear();
        };
    }
    LS.clear();
    LP.clear();
}
```

Y por tanto, el destructor ser3:

```
GRAFO::~~GRAFO()
{
    destroy();
}
```

¿C3mo actualizar el grafo cargando un nuevo grafo desde fichero?

En esta ocasi3n usaremos un m3todo que combina dos: primero libera la memoria que ocupaba el anterior grafo con `destroy()`, y luego carga la informaci3n del nuevo grafo desde fichero con `build`.

```
void GRAFO:: actualizar (char nombrefichero[85], int &errorapertura)
{
    //Limpiamos la memoria ocupada en la carga previa, con el destructor
    destroy();
    //Leemos del fichero y actualizamos G con nuevas LS y, en su caso, LP
    build(nombrefichero, errorapertura);
}
```

3.3. Resumen de la primera fase de la pr3ctica: ficheros a incluir, m3todos a implementar

Para poder avanzar e implementar en esta primera parte la carga de los ficheros de grafos problema, la construcci3n del objeto grafo y su visualizaci3n y comprobaci3n, son necesarios tres ficheros:

El fichero **grafo.h** es el de cabecera de las biblioteca de la clase GRAFO. Contiene las instrucciones `include` de las bibliotecas que necesitaremos, as3 como constantes, adem3s de las definiciones de las estructuras que usaremos, entre ellas, la del objeto grafo y los m3todos que lo gestionan. Puede descargarse desde el campus virtual una versi3n inicial con los tipos de datos y la estructura de la clase para la primera parte de la pr3ctica.

El fichero **grafo.cpp** desarrolla la implementación de los métodos del objeto definido en el fichero anterior que es una plantilla a completar ustedes para esta primera parte de la práctica.

Según hemos visto, los métodos necesarios implementar de la biblioteca **grafo.h** en **grafo.cpp** para esta primera fase de la práctica son:

```
void build (char nombrefichero[85], int &errorapertura);
```

Es el método para cargar desde fichero los datos del grafo, y según sea un grafo dirigido o no dirigido, construir la lista de sucesores y la lista de predecesores, LS y LP, respectivamente en el primer caso, o sólo la lista de adyacencia en LS. Si ocurriera algún error en la apertura del fichero de texto, devolvería un 1 en **errorapertura**.

```
void destroy();
```

Es el método para liberar la memoria de las estructuras LS y, en su caso, LP. Este método se detalla en el presente guión y se encontrará disponible en la plantilla de **grafo.cpp**.

```
GRAFO(char nombrefichero[], int &errorapertura);
```

Es el método constructor, y sus parámetros son los mismos que el método **build** por razones obvias. Su detalle se incluye en el presente guión y se encontrará disponible en la plantilla **grafo.cpp**.

```
~GRAFO();
```

Es el destructor, y se dedica a liberar la memoria de las listas de adyacencia. No olvides ejecutarlo antes de la finalización del programa.

```
void actualizar (char nombrefichero[], int &errorapertura);
```

Como se ha indicado, es un método análogo al constructor, pero que te permitirá, tras liberar memoria de las listas, cargar la información de un nuevo grafo desde fichero. Su detalle se incluye en el presente guión y se encontrará disponible en la plantilla **grafo.cpp**.

```
unsigned Es_dirigido();
```

Devuelve 0 si el grafo es no dirigido, y 1 en otro caso. Esta función servirá para distinguir que tipo de grafo se ha cargado y, desde el programa principal, mostrar el menú de opciones para grafos dirigidos o no dirigidos.

```
void Info_Grafo();
```

Muestra la información básica de un grafo: número de nodos, su orden, número de arcos o aristas y su tipo.

```
void Mostrar_Listas(int l);
```

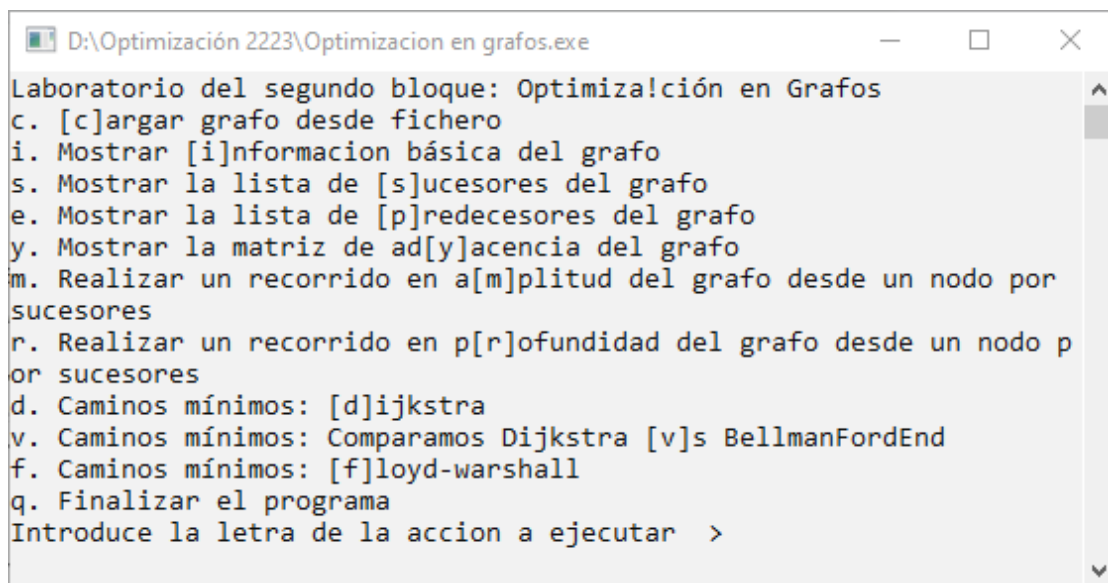
Según el parámetro **l**, mostrará la lista de sucesores y predecesores, o sólo la lista de adyacentes, según el tipo de grafo que sea. Por ejemplo, si el grafo es no dirigido, el valor de **l=0** mostrará la lista de adyacencia del grafo; si el grafo es dirigido, el valor **l=+1** mostrará la lista de sucesores y **l=-1** la lista de predecesores.

Optimización

La forma de mostrar la lista, con los datos de los nodos, debe ser compacta y fácil de leer en la pantalla. Para cada nodo, sus sucesores, predecesores o adyacentes, y sus costes o pesos, deberán caber en una línea, y toda la información, en una pantalla. Para ello usará el método `Mostrar_Lista(vector<LA_nodo> L)`, que simplemente recorrerá la lista `L` según necesitemos que sea `LS` - para la lista de sucesores o la lista de adyacentes según sea el grafo dirigido o no dirigido, respectivamente -, o `LP` - para la lista de predecesores -.

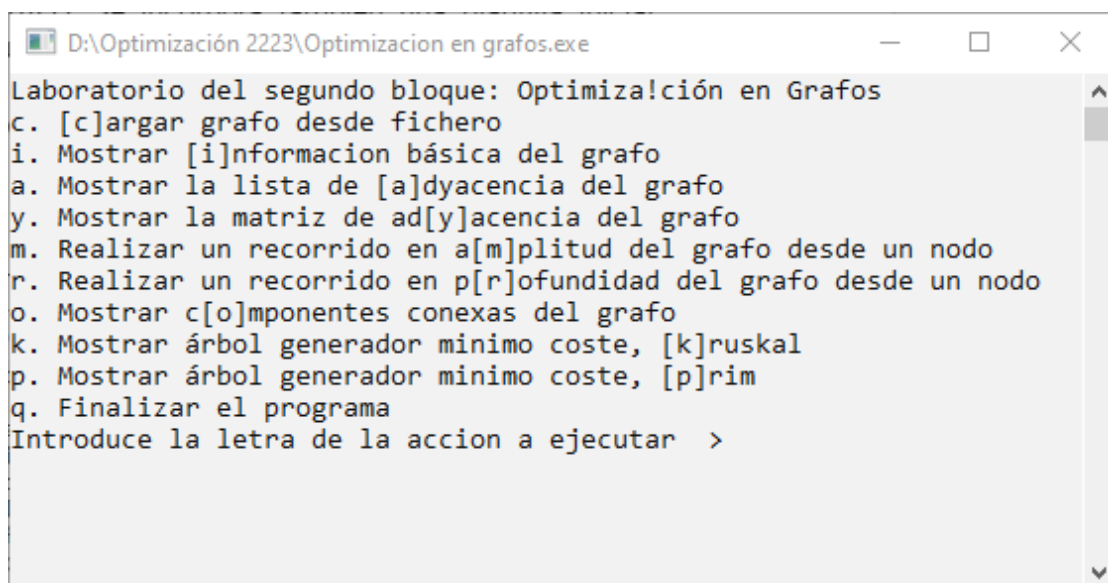
El fichero `main.cpp` implementa un sencillo programa menú que usa los métodos de la clase para testear la clase `GRAFO`. Se incorpora también una plantilla inicial incompleta de este fichero a los alumnos y alumnas.

El resultado se ilustra en los siguientes dos pantallazos. El primero, donde el grafo cargado es dirigido y se ofertan todas las opciones posibles para este tipo de grafo, más de las que se piden en esta primera práctica.



```
D:\Optimización 2223\Optimizacion en grafos.exe
Laboratorio del segundo bloque: Optimiza!ción en Grafos
c. [c]argar grafo desde fichero
i. Mostrar [i]nformacion básica del grafo
s. Mostrar la lista de [s]ucesores del grafo
e. Mostrar la lista de [p]redecesores del grafo
y. Mostrar la matriz de ad[y]acencia del grafo
m. Realizar un recorrido en a[m]plitud del grafo desde un nodo por
sucesores
r. Realizar un recorrido en p[r]ofundidad del grafo desde un nodo p
or sucesores
d. Caminos mínimos: [d]ijkstra
v. Caminos mínimos: Comparamos Dijkstra [v]s BellmanFordEnd
f. Caminos mínimos: [f]loyd-warshall
q. Finalizar el programa
Introduce la letra de la accion a ejecutar >
```

La segunda, muestra el menú de opciones para un grafo no dirigido, también con más opciones que las que explicamos en esta primera práctica.



```
D:\Optimización 2223\Optimizacion en grafos.exe
Laboratorio del segundo bloque: Optimiza!ción en Grafos
c. [c]argar grafo desde fichero
i. Mostrar [i]nformacion básica del grafo
a. Mostrar la lista de [a]dyacencia del grafo
y. Mostrar la matriz de ad[y]acencia del grafo
m. Realizar un recorrido en a[m]plitud del grafo desde un nodo
r. Realizar un recorrido en p[r]ofundidad del grafo desde un nodo
o. Mostrar c[o]mponentes conexas del grafo
k. Mostrar árbol generador minimo coste, [k]ruskal
p. Mostrar árbol generador minimo coste, [p]rim
q. Finalizar el programa
Introduce la letra de la accion a ejecutar >
```

Como se comprueba, hay opciones comunes en ambos men3, como son la opci3n de cargar un nuevo grafo, mostrar la informaci3n b3sica, o finalizar el programa.

4. Segunda fase

4.1. Implementando los recorridos en un grafo

Seg3n hemos visto en clase, un recorrido sobre un grafo es una estrategia que permite visitar sus nodos de una forma sistem3tica y ordenada. Se pretende que, dada cierta filosof3a de orden o prioridad que sea de inter3s, se visiten todos los nodos de manera eficiente: todos visitados y sin repetir visita.

La analog3a m3s cercana a la filosof3a de un recorrido es la b3squeda de un tesoro escondido en un *laberinto*: debemos asegurarnos de no dejar zona por estudiar, pero sin repetir zonas porque nos har3an entrar en ciclos en nuestra b3squeda. Por ello, precisamente, es 3til llevar una memoria de los sitios visitados y de aquellos que dejamos pendientes por investigar. Siguiendo con la analog3a, se trata de que, llegando a un punto (nodo) desde el que se abren a varios pasillos, (aristas o arcos), elijamos cualquiera de los pasillos para continuar el examen, pero apuntemos que debemos volver al mismo punto para optar por los pasillos pendientes. Esto lo haremos con dos sencillas herramientas:

- Un atributo para cada nodo que indique si ha sido o no visitado.
- Una lista de nodos pendientes por visitar.

Un esquema general de recorrido desde el nodo *i* ser3a el siguiente:

```
{inicializaci3n}
Para todo nodo v, visitado[v] = falso;
{preparamos el inicio del recorrido desde el nodo i}
Visitado[i] = verdadero; ToDo = {i};
{bucle principal}
Mientras ToDo no vac3o hacer
    Sea k en ToDo
    ToDo = ToDo - {k}
    Para todo adyacente j de k hacer
        Si visitado[j] = falso entonces
            Visitado[j]=verdadero;
            ToDo = ToDo + {j}
```

En este pseudoc3digo se pretende simplificar la estrategia de un recorrido, y no se define, por tanto, la gesti3n de los nodos que han de ser revisados y que est3n en la estructura general ToDo. ToDo puede ser, desde una bolsa opaca donde vamos introduciendo los nodos a ser estudiados sin priorizaci3n de salida alguna, o, en cambio, puede ser gestionado como una pila o como una cola.

- En el caso de una *pila*, el 3ltimo nodo que entre es el primero en salir a ser estudiado, se tratar3a de un recorrido en profundidad, en ingl3s, *deep first search*, o su acr3nimo *dfs*.
- En el caso de una *cola*, donde el nodo m3s antiguo es el primero en salir, hablamos de un recorrido en amplitud, en ingl3s, *breadth search first*, o su acr3nimo *bfs*.

En el material dispuesto en el campus virtual sobre los recorridos, en las transparencias donde se detallan las trazas, tanto del recorrido en amplitud como del recorrido en profundidad, aparecen las propuestas de código de ambos. Para el recorrido en profundidad, tanto de la versión recursiva, la que usaremos en esta actividad, como de su versión no recursiva.

4.2. Trabajando el recorrido en profundidad, DFS

Para desarrollar los recorridos partiremos en ambos casos de la expresión en pseudocódigo de las transparencias de la traza de este recorrido, para implementarlo en c++. La expresión del recorrido en profundidad que usaremos es la recursiva por cuestión de diseño eficiente: se hace coincidir la pila que gestiona los nodos pendientes de visitar, `ToDo`, con la pila de llamadas recursivas. Debemos añadir el siguiente método en la parte privada del objeto GRAFO en `grafo.h`, y su desarrollo en `grafo.cpp`:

```

void GRAFO::dfs_num(      unsigned i, //nodo desde el que realizamos el recorrido en profundidad
                          vector<LA_nodo> L, //lista que recorremos, LS o LP; por defecto LS
                          vector<bool> &visitado, //vector que informa de si un nodo ha sido visitado
                          vector<unsigned> &prenum, //almacenamos en la posición i el preorden del nodo i+1
                          unsigned &prenum_ind, //contador del preorden
                          vector<unsigned> &postnum, //almacenamos en la posición i el postorden del nodo i+1
                          unsigned &postnum_ind) //contador del postorden
                          //Recorrido en profundidad recursivo con recorridos enum y postnum
{
    visitado[i] = true; //visitamos el nodo i+1
    prenum[prenum_ind++]=i; //asignamos el orden de visita prenum que corresponde el nodo i+1
    for (unsigned j=0; j<L[i].size(); j++) //recorremos la adyacencia del nodo visitado, esto es, i+1
        if (!visitado[L[i][j].j])
        {
            dfs_num(L[i][j].j, L, visitado, prenum, prenum_ind, postnum, postnum_ind);
        };
    postnum[postnum_ind++]=i; //asignamos el postorden de visita que corresponde al nodo i+1 al regreso
}

```

Con este método puede ya construirse el procedimiento del recorrido en profundidad, que inicialice los vectores, pida al usuario el nodo del que partirá el recorrido por la lista LS, y que, finalmente, muestre el preorden y el postorden resultante.

```

void GRAFO::RecorridoProfundidad()
{
    //creación e inicialización de variables y vectores
    //solicitud al usuario del nodo inicial del recorrido en profundidad
    //mostrar en pantalla el preorden
    //mostrar en pantalla el postorden
}

```

Es muy importante entender que el preorden y el postorden depende del lugar en el que aparecen los arcos o aristas en LS o, en su caso, LP. Y ese orden, depende de cómo aparezcan en el fichero de texto que usamos para introducir el programa. Téngase esto en cuenta a la hora de corregir el resultado del programa con una prueba hecha por uno mismo. Para el grafo siguiente,

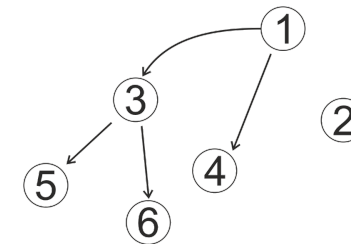
$$\Gamma_1^+ = \{3, 4\}; \Gamma_2^+ = \emptyset; \Gamma_3^+ = \{5, 6\}; \Gamma_4^+ = \emptyset; \Gamma_5^+ = \emptyset; \Gamma_6^+ = \emptyset;$$

El preorden y postorden que construye son los siguientes:

```

"X:\Mi unidad\Optimizacion\2021\Laboratorio\bin\Release\Optimizacion en grafos.exe"
Vamos a construir un recorrido en profundidad
Elija nodo de partida? [1-6]: 1
Orden de visita de los nodos en preorden
[1] -> [3] -> [5] -> [6] -> [4]
Orden de visita de los nodos en postorden
[5] -> [6] -> [3] -> [4] -> [1]
Presione una tecla para continuar . . .

```



4.3. Trabajando el recorrido en amplitud, BFS

Este es el único recorrido que, de forma obligatoria, hay que implementar en la práctica: el recorrido en amplitud. Para implementar este recorrido en c++, también desde la expresión en pseudocódigo de las transparencias, debemos contar con la librería que gestiona las colas para la estructura ToDo. De la librería que carguemos, queue, cuya línea incluye ya consta en la plantilla de grafo.h, necesitamos:

- el tipo cola, queue, que se inicializa como cola vacía
- el método push, que introduce un elemento en la cola
- el método empty, que nos indica si la cola está vacía
- el método front, que nos informa del primer elemento de la cola
- el método pop, que hace salir el primer elemento de la cola

Debemos añadir el siguiente método en la parte privada del objeto GRAFO en grafo.h y su desarrollo en grafo.cpp:

```
void GRAFO::bfs_num(      unsigned i, //nodo desde el que realizamos el recorrido en amplitud
                          vector<LA_nodo> L, //lista que recorremos, LS o LP; por defecto LS
                          vector<unsigned> &pred, //vector de predecesores en el recorrido
                          vector<unsigned> &d) //vector de distancias a nodo i+1
//Recorrido en amplitud con la construcción de pred y d: usamos la cola
{
    vector<bool> visitado; //creamos e iniciamos el vector visitado
    visitado.resize(n, false);
    visitado[i] = true;

    pred.resize(n, 0); //creamos e inicializamos pred y d
    d.resize(n, 0);
    pred[i] = i;
    d[i] = 0;

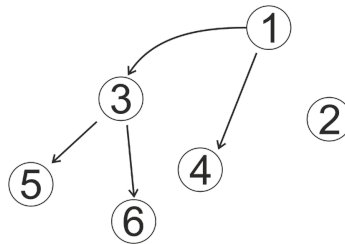
    queue<unsigned> cola; //creamos e inicializamos la cola
    cola.push(i); //iniciamos el recorrido desde el nodo i+1

    while (!cola.empty()) //al menos entra una vez al visitar el nodo i+1 y continúa hasta que la cola se vacíe
    {
        unsigned k = cola.front(); //cogemos el nodo k+1 de la cola
        cola.pop(); //lo sacamos de la cola
        //Hacemos el recorrido sobre L desde el nodo k+1
        for (unsigned j=0; j<L[k].size(); j++)
            //Recorremos todos los nodos u adyacentes al nodo k+1
            //Si el nodo u no está visitado
            {
                //Lo visitamos
                //Lo metemos en la cola
                //le asignamos el predecesor
                //le calculamos su etiqueta distancia
            };
        //Hemos terminado pues la cola está vacía
    };
}
```


Con este método puede ya construirse el procedimiento del recorrido en amplitud, que inicialice los vectores, pida al usuario el nodo del que partirá el recorrido por la lista LS, y que, finalmente, muestre los valores de los predecesores y la etiqueta distancia.

```
void GRAFO::RecorridoAmplitud()
{
    //creación e inicialización de variables y vectores
    //solicitud al usuario del nodo inicial del recorrido en amplitud
    //mostrar en pantalla la etiqueta distancia
    //mostrar en pantalla los predecesores
}
```

En la presente propuesta, se mostraría, para cada nodo, su etiqueta distancia; y para cada nodo, su predecesor. Sin embargo, en el ejecutable que se comparte, para el recorrido en amplitud, tanto la etiqueta distancia, como los predecesores, se muestran de forma distinta. Para el caso de la etiqueta distancia, se muestran los nodos ordenados según su valor de etiqueta distancia: primero los de etiqueta distancia cero, - que es sólo el nodo inicial -, luego aquellos nodos a distancia 1 arco o arista, luego aquellos nodos a distancia 2 arcos o aristas, y así sucesivamente. En el caso de los predecesores, se muestran las cadenas o caminos que conectan el nodo inicial a cada uno del resto de nodos accesibles. La pantalla siguiente ilustra un ejemplo de salida sobre este grafo:



```
D:\Optimización 2223\Optimizacion en grafos.exe
Vamos a construir un recorrido en amplitud
Elije nodo de partida? [1-6]: 1
Nodo inicial: 1

Nodos según distancia al nodo inicial en número de aristas
Distancia 0 aristas : 1
Distancia 1 aristas : 4 : 3
Distancia 2 aristas : 5 : 6

Ramas de conexión en el recorrido
1 - 3
1 - 4
1 - 3 - 5
1 - 3 - 6
Presione una tecla para continuar . . .
```

Evaluación

Para superar esta práctica, los procedimientos de carga de las listas de sucesores, predecesores y adyacencia, para ambos tipos de grafos, deben estar correctos, debiendo compilar y ejecutar sin problema el programa menú que testea las opciones. Además debe ser capaz de cargar varios ficheros de grafos sin salir del ejecutable y sin constatar errores al mostrar lo cargado. Todas las opciones de la primera fase son obligatorias.

También es obligatorio la implementación del recorrido en amplitud mostrando tanto la etiqueta distancia d de cada nodo, así como los predecesores, $pred$, en la arborescencia que crea el recorrido.

Para el apto +, se les propondrá incluir o modificar el código para que presente alternativas y se valorará el resultado.

Asimismo, al finalizar la evaluación de la práctica, si es apto -, apto o apto +, el alumno responderá a un corto cuestionario para evaluar sus conocimientos sobre la práctica y los contenidos teóricos que la sustentan.