



## Título de la actividad: Almacenamiento en la clase GRAFO.

Profesor Responsable: Sergio Alonso | Actividad I | Dificultad: media | Inicio: 24/03/2014

### Objetivo

El objetivo de esta actividad es escribir un programa que gestione la carga de los datos de un grafo a partir de las estructuras de sucesores o predecesores, en el caso de los grafos dirigidos, y con las adyacencias, en el caso de los grafos no dirigidos. Usaremos la clase GRAFO que, desde esta primera actividad, estará dotada de lo esencial para poder codificar los grafos y trabajar con ellos bajo distintos algoritmos. El programa de prueba tendrá forma de menú, que interactúa con el usuario para ejecutar las distintas opciones posibles.

### Temporalización

Esta actividad se divide en tres prácticas. En las dos primeras horas de tutorización, se repasará la estructura del grafo como representación de problemas de optimización combinatoria, además de las codificaciones usuales a través de sucesores y predecesores, y de la adyacencia. Con ello, podrá definirse la estructura en c++ que albergará la información del grafo problema, así como su carga desde un fichero de texto.

En la tercera y última hora, se presentará por parte del estudiante el ejecutable, que será corregido y evaluado.

### Implementación

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.

El programa **necesita** cargar un grafo desde un fichero de texto para poder iniciar el menú de opciones. El fichero de texto ha de tener el siguiente formato para su correcta lectura:

```
n m esdirigido?  
i1 j1
```

```
im jm
```

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.

El grafo se lee del fichero, pero se almacena su lista de sucesores y 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.

Para almacenar la información del grafo en listas de sucesores, predecesores o adyacencia, usaremos un objeto perteneciente a la clase GRAFO que es definida como un vector de vectores de registros. Para su implementación, primero definimos el registro, que denomina `ElementoLista`, y donde se almacenan el nodo sucesor, predecesor, adyacente, según se

trate, así como cualquier otro atributo del arco o arista. En este caso, aunque no sea de utilidad en la presente actividad, se introduce un coste.

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

A continuación construimos el vector de registros, sabiendo que las listas de sucesores, predecesores o adyacencia son vectores de éstos.

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

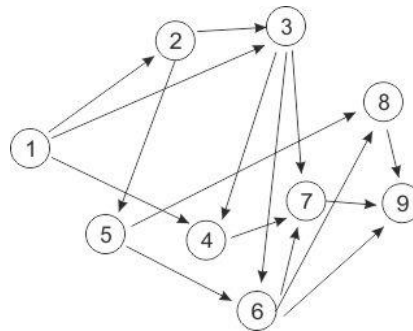
La clase GRAFO incluye la información necesaria para la gestión de los datos, su manipulación y se prepara para incluir cómo métodos, los algoritmos que resolverán los problemas que trataremos en la asignatura.

```
class GRAFO
{
    unsigned n; // número de nodos o vértices
    unsigned m; // número de arcos o aristas
    vector<LA_nodo> LS; // lista de sucesores o de adyacencia
    vector<LA_nodo> LP; // lista de predecesores
public:
    unsigned dirigido; //0 si el grafo es no dirigido,1 si es dirigido
    GRAFO(char nombrefichero[], int &errorapertura);
    void actualizar(char nombrefichero[], int &errorapertura);
    void Info_Grafo();
    void Mostrar_Listas(int l);
    void ListaPredecesores();
    ~GRAFO();
};
```

Las definiciones de las estructuras de datos necesarias, las de la clase GRAFO y otras comunes, se almacenarán en el fichero de cabeceras **grafo.h**

Usaremos el grafo siguiente para ilustrar la carga de datos. Se trata de un grafo dirigido de 9 nodos y 16 aristas, grafo1.gr, que encontrarás en la documentación del campus virtual de la asignatura.

```
9 16 1
1 2
1 3
1 4
2 3
2 5
3 4
3 6
3 7
4 7
5 6
5 8
6 7
6 8
6 9
7 9
8 9
```



El constructor `GRAFO(char nombrefichero[], int &errorapertura)` se encarga de leer el problema del fichero de texto `nombrefichero` de tipo `ifstream` y de asignar los atributos de la clase `GRAFO`, devolviendo 0 en `errorapertura` si no ha habido incidencia alguna, y 1 si la ha habido. Es necesario, por tanto, añadir la línea `#include <fstream>` al fichero de cabeceras **grafo.h**

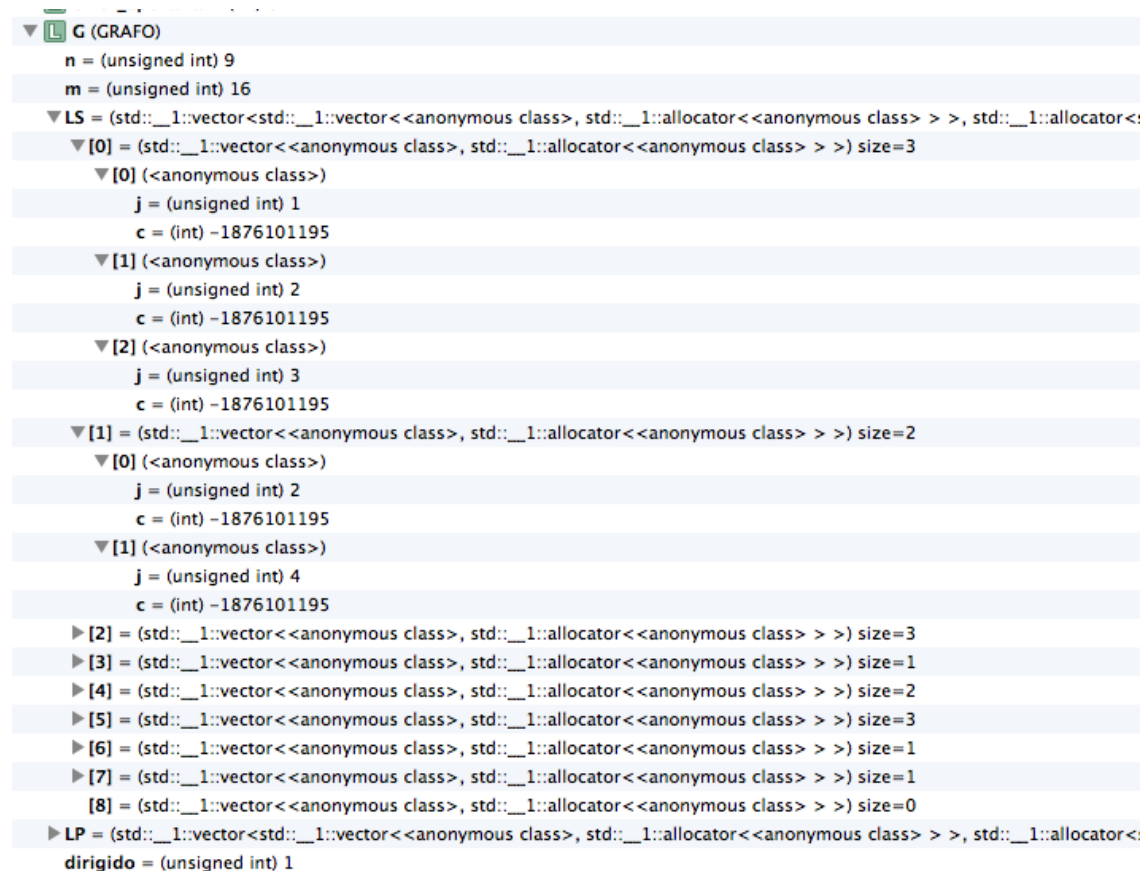
Tras la lectura de la primera línea, ya conocemos el orden del grafo, esto es, el número de nodos, el número de arcos o aristas, lo cual nos indica el número de líneas que aún hemos de leer del fichero, y el tipo de grafo que es: dirigido o no dirigido. Si es dirigido, debemos construir tanto la lista de sucesores como la de predecesores, que se almacenarán en `LS` y

LP, respectivamente. Para gestionar los tipos vector de las listas, debemos incluir en el fichero de cabeceras **grafo.h** la línea `#include <vector>`.

Conocido  $n$  podemos dimensionar **LS** con el método `resize`, esto es, `LS.resize(n)`, obteniendo el acceso a los vectores vacíos, `LS[0]`, `LS[1]`, ..., `LS[n-1]`. Se trata de guardar en `LS[i-1]` la información de los sucesores del nodo  $i$ . Para ello, cada vez que leamos una línea del fichero de texto de la forma  $i\ j$  usamos una variable dummy del tipo `ElementoLista` para asignarle esos valores, y luego el método `push_back` para introducir la información del sucesor, esto es, `LS[i-1].push_back(dummy)`, donde `dummy.j` es  $j-1$ .

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

Para el grafo de ejemplo, almacenaremos los sucesores y predecesores a los nodos del grafo, y la forma visual de almacenamiento de la clase `GRAFO` la puedes ver a continuación:



3

Se observa en la imagen anterior, la estructura de vector de vectores que tiene **LS**. Los vectores que tiene en su interior son de tamaño variable, dependiendo lógicamente del número de sucesores del nodo. Análogamente con **LP**, en el caso de grafos dirigidos. Así, localizamos en `LS[0]` los sucesores del nodo 1, en las posiciones, `LS[0][0]`, `LS[0][1]` y `LS[0][2]`. Para conocer los valores de esos sucesores, accedemos al registro, esto es, `LS[0][0].j` almacena el valor 1, por lo que se refiere al nodo 2. Esto lo comprobamos al instante, puesto que 2 es sucesor del nodo 1.

Finalmente, es importante saber que, en el caso de grafos no dirigidos, la adyacencia es simétrica. Esto es, si leemos del fichero de texto la arista  $(i, j)$ , debemos situar  $j$  como adyacente de  $i$ , y también a  $i$  como adyacente de  $j$ , excepto en el caso de que se trate de un bucle. Toda esa información se almacena en **LS**, y **LP** permanece sin uso.

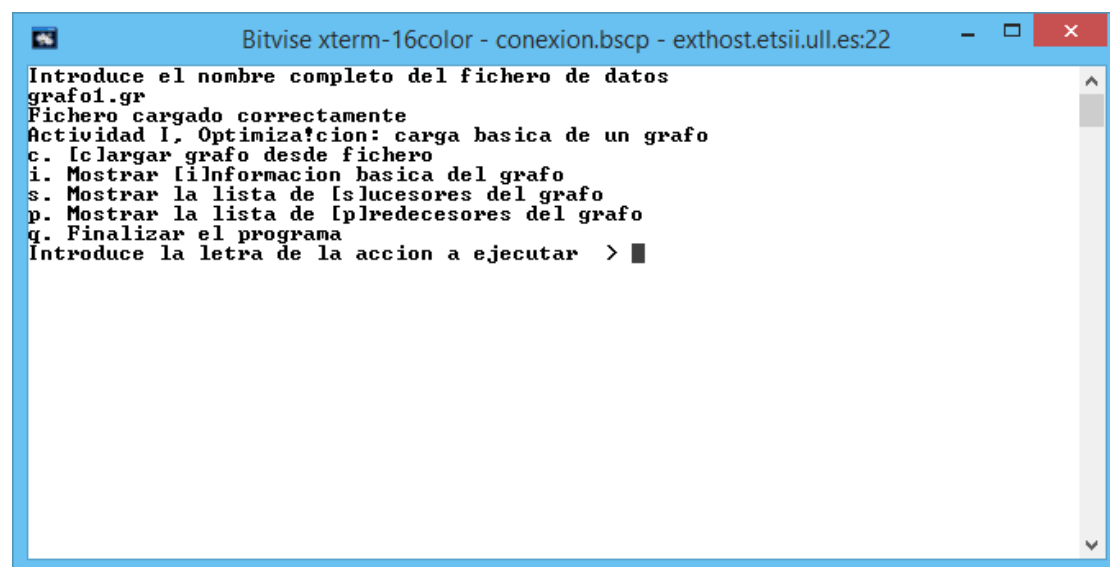
## Ficheros de la actividad

Ya hemos citado la necesidad de trabajar con un fichero de cabeceras **grafo.h** en el que se definan las estructuras, constantes, y se carguen el resto de ficheros de cabecera. Son también de interés para la serie de programas de prácticas de optimización combinatoria, que se incluyan en el fichero las constantes siguientes:

```
const unsigned UERROR = 65000;  
const int maxint = 1000000;
```

Ambas serán importantes en el control de los algoritmos.

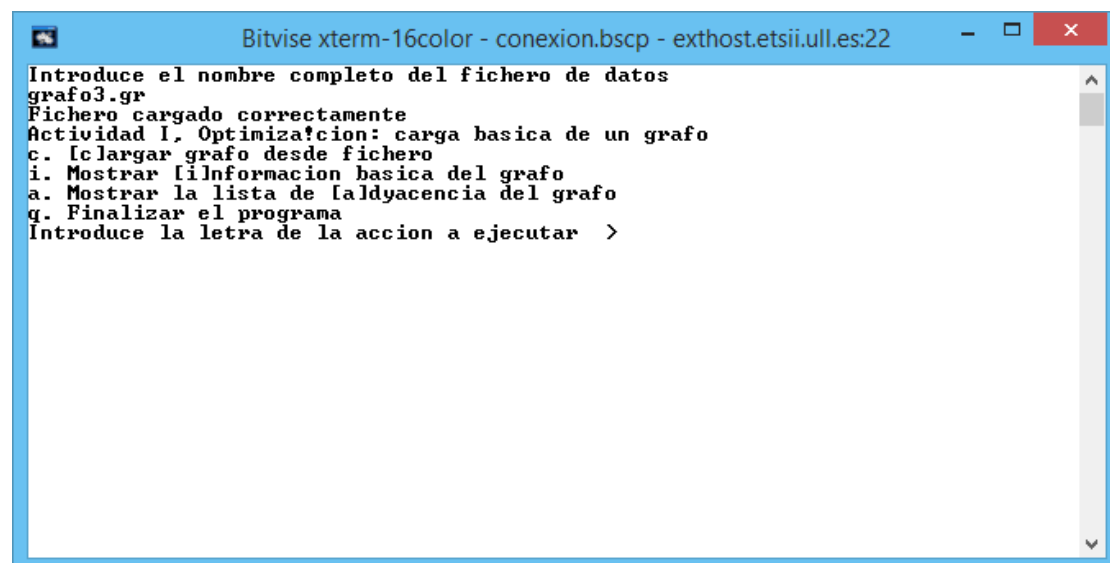
Además, la actividad deberá contenerse en dos ficheros más. El primero con el fichero **grafo.cpp**, que contendrá el código para los métodos, funciones y procedimientos de la clase GRAFO. El segundo, con el programa principal `main`, que será un simple gestor tipo menú y que estará en un fichero denominado **pg1.cpp**. Las pantallas con las opciones dependerán del tipo de grafo que sea, así, para un grafo dirigido, mostrará:



```
Bitwise xterm-16color - conexion.bsccp - exthost.etsii.ull.es:22  
Introduce el nombre completo del fichero de datos  
grafo1.gr  
Fichero cargado correctamente  
Actividad I, Optimiza!cion: carga basica de un grafo  
c. lclargar grafo desde fichero  
i. Mostrar [i]nformacion basica del grafo  
s. Mostrar la lista de [s]lucadores del grafo  
p. Mostrar la lista de [p]rededores del grafo  
q. Finalizar el programa  
Introduce la letra de la accion a ejecutar > █
```

4

Mientras que, para un grafo no dirigido, mostrará,



```
Bitwise xterm-16color - conexion.bsccp - exthost.etsii.ull.es:22  
Introduce el nombre completo del fichero de datos  
grafo3.gr  
Fichero cargado correctamente  
Actividad I, Optimiza!cion: carga basica de un grafo  
c. lclargar grafo desde fichero  
i. Mostrar [i]nformacion basica del grafo  
a. Mostrar la lista de [a]dyacencia del grafo  
q. Finalizar el programa  
Introduce la letra de la accion a ejecutar >
```

## Métodos de la clase GRAFO

Como se ha indicado, los métodos necesarios para esta actividad son:

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

Es el método constructor, y sus parámetros son, el nombre del fichero de texto donde está el grafo, así como una variable puente para situar si ha habido error en la apertura. Su función ya ha sido explicada con detalle.

```
~GRAFO();
```

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

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

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.

```
void Info_Grafo();
```

Muestra la información básica de un grafo: parámetros y 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.

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, deberán caber en una línea, y toda la información, en una pantalla.

```
void ListaPredecesores();
```

Es un método que recorre la lista de sucesores cargada de un fichero de texto para un grafo dirigido, y construye la lista de predecesores.

## Evaluación

5

Para superar esta actividad, los procedimientos de carga de las listas de sucesores, predecesores y adyacencia, para los dos tipos de grafos, deben estar correctas. El profesor podrá valorar la limpieza, documentación del código y la optimización de los recursos para puntuar por encima del apto.