

## Actividad I: La clase GRAFO.

Profesor responsable: Sergio Alonso.

Dificultad: media.

Presentación del plan de trabajo: martes, 31 de marzo de 2020.

Fecha límite para la entrega de la práctica: jueves, 23 de abril de 2020, 23:59 horas.

### 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. Crearemos 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. En particular, añadiremos finalmente una sencilla utilidad para el análisis de la conexidad de un grafo no dirigido, que pondrá a prueba la codificación implementada. El programa que pondrá a prueba la clase GRAFO tendrá forma de menú, que interactúa con el usuario para ejecutar las distintas opciones posibles.

### Plan de trabajo

Esta actividad se divide en dos fases. En una primera, la primera semana se trabaja, **es fundamental 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, se podrán usar ficheros en c++ como plantillas, descargables desde el campus virtual, para ser analizados y que podrán ser completados por el alumnado.

Un vez comprobado el buen funcionamiento del constructor y su actualizador, la segunda fase, durante la segunda semana de trabajo, **incorporará un método para el recorrido en profundidad, que será la base para analizar la conexidad de un grafo no dirigido.**

La entrega de esta práctica se podrá realizar en el campus virtual **hasta las 23:59 horas del jueves 23 de abril.**

### 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

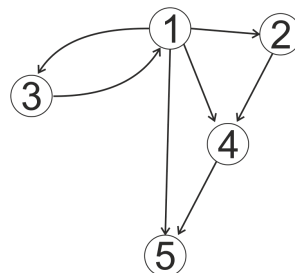
im jm
```

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 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. Si ilustramos mediante un ejemplo, el objetivo sería la codificación del mismo en el fichero de texto, y el análisis de las estructuras en c++ necesarias para almacenar las listas de forma eficiente.

Trabajamos con el grafo dirigido siguiente:



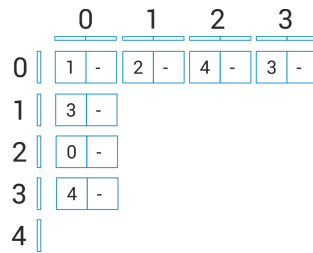
Si codificamos el grafo anterior en un fichero de texto, tal y como se ha expresado, tendríamos:

```
5 7 1
1 2
1 3
1 5
1 4
2 4
3 1
4 5
```

Cuyo conjunto de sucesores sería, por tanto:

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

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:



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**.

Vamos a explicarla con detalle, siempre desde el interior hacia el exterior. Primero definimos el registro, que se denomina ElementoLista, y donde se almacenaría 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 la posibilidad de almacenar un coste.

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

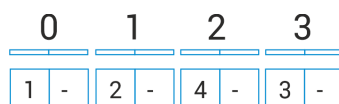
Un ElementoLista podríamos idealizarlo gráficamente como un eslabón con la siguiente forma:



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

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

Por tanto, mediante un vector del tipo LA\_nodo, podemos almacenar la información 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 tendrían la apariencia 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 información de los nodos, siempre restaremos una unidad. Así, información que guardamos en este LA\_nodo, 1, 2, 4 y 3,

corresponde a los nodos, 2, 3, 5 y 4. Esto es, 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`, sería suficiente para almacenar la información 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:

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

#### ¿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`. Por tanto, 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.

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.

#### ¿De qué tipo 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].coste`, que almacenaría el coste del arco (3, `LS[2][0].j+1`), es decir, (3, 1), si lo hubiera.

### **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. Es esta:

```

class GRAFO
{
    unsigned n;           // número de nodos o vértices
    unsigned m;           // número de arcos o aristas
    unsigned dirigido;    // almacena 0 si el grafo es no dirigido, 1 eoc
    vector<LA_nodo> LS;     // lista de sucesores o de adyacencia
    vector<LA_nodo> LP;     // lista de predecesores

public:
    GRAFO(char nombrefichero[], int &errorapertura);
    void actualizar (char nombrefichero[], int &errorapertura);
    unsigned Es_dirigido(); //devuelve 0 si el grafo es no dirigido, 1 eoc
    void Info_Grafo();
    void Mostrar_Listas(int l);
    void ListaPredecesores();
    ~GRAFO();
};

```

### El constructor del objeto GRAFO

El constructor del objeto GRAFO,

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

se encarga de leer el problema del fichero de texto de nombre nombrefichero de tipo ifstream y de asignar los atributos y las estructuras de la clase GRAFO, construyendo un objeto que denominaremos G, devolviendo 0 en errorapertura si no ha habido incidencia alguna, y 1 si la ha habido.

#### ¿Cómo abrir y leer de un fichero de texto?

Usaremos la librería fstream para poder gestionar ficheros de entrada de datos, ifstream. Así, las primeras líneas de código del constructor deberán 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 conversión implícita el número de nodos, arcos y el
    atributo dirigido en la primera línea de fichero de texto con la
    información 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 línea, ya conocemos la dimensión 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.

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, que podemos llamar `dummy`, del tipo `ElementoLista` para asignarle esos valores. Luego, con el método `push_back`, introducimos la información del sucesor en `LS[i-1]`, esto es, `LS[i-1].push_back(dummy)`, donde `dummy.j` es `j-1`.

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, 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.**

#### ¿Qué ocurre con los grafos no dirigidos?

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`, pero también a `i` como adyacente de `j`, **excepto en el caso de que se trate de un bucle**. Es decir, en el caso de  $(i, i)$  sólo hay que realizar una inserción con el `push_back`. La información de la lista de adyacencia se almacena en `LS`, y `LP` permanece sin uso.

### Primera fase de la actividad: métodos de la clase GRAFO

Los métodos necesarios para esta primera fase son:

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

Es el método constructor, y sus parámetros ya han sido comentados: el nombre del fichero de texto donde está el grafo, así como una variable puente para situar si ha habido error en la apertura. Esta variable puente servirá para controlar el error e interactuar con el usuario para que introduzca el nombre correcto del fichero.

```
~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);
```

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.

```
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: 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. 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.

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 opcional que recorre la lista de sucesores cargada de un fichero de texto para un grafo dirigido, y construye la lista de predecesores. Esto es, para un grafo dirigido, puede construirse la lista de sucesores y predecesores a medida que se lee, línea a línea, los arcos del grafo desde el fichero de texto. O, construir sólo la lista de sucesores y, con este procedimiento, recorrerla e ir construyendo la lista de predecesores sin acudir a la lectura del fichero por segunda vez.

### Primera fase de la actividad: ficheros de la práctica

Para poder avanzar e implementar en esta primera fase la carga de los ficheros de grafos problema, la construcción del objeto grafo y su visualización y comprobación, son necesarios tres ficheros:

#### grafo.h

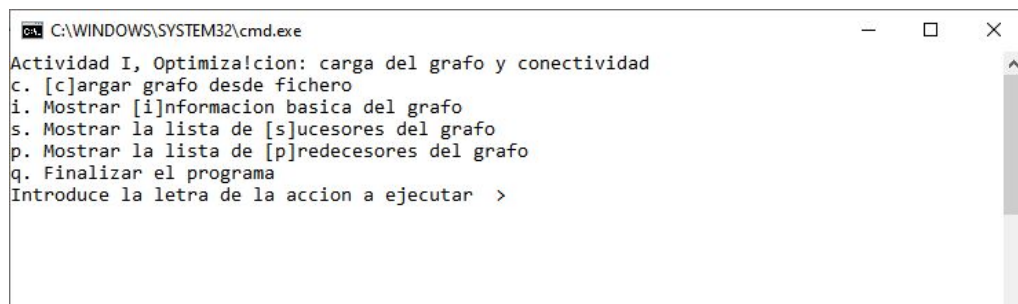
Fichero de cabecera de las prácticas. Contiene las instrucciones incluye de las bibliotecas que necesitaremos, así como constantes. Puede descargarse desde el campus virtual.

#### grafo\_.cpp

Fichero con la implementación de los métodos del objeto definido en el fichero anterior que es una plantilla a ser completada por el alumnado.

#### pg1\_.cpp

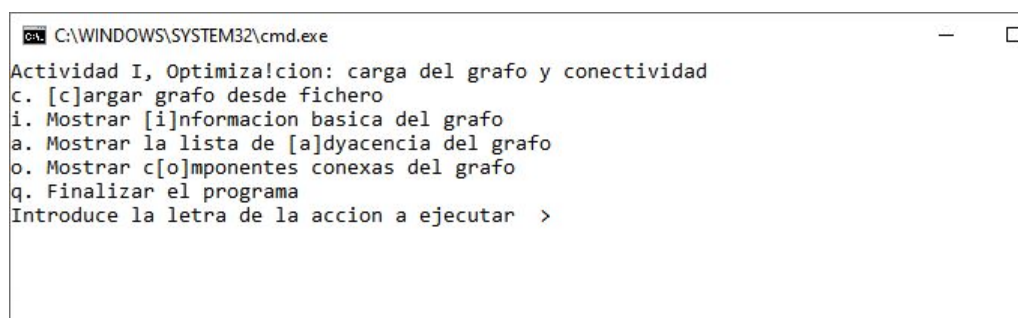
Fichero plantilla con la implementación del programa principal que usa las definiciones de la clase GRAFO y sus métodos en un sencillo menú de interacción con el usuario. Este programa, visualmente debe mostrar un menú de opciones para el grafo dirigido:



```

C:\WINDOWS\SYSTEM32\cmd.exe
Actividad I, Optimiza!cion: carga del grafo y conectividad
c. [c]argar grafo desde fichero
i. Mostrar [i]nformacion basica del grafo
s. Mostrar la lista de [s]ucesores del grafo
p. Mostrar la lista de [p]redecesores del grafo
q. Finalizar el programa
Introduce la letra de la accion a ejecutar >
  
```

Y este otro para el grafo no dirigido, aún cuando la opción o es la segunda fase de esta práctica:



```

C:\WINDOWS\SYSTEM32\cmd.exe
Actividad I, Optimiza!cion: carga del grafo y conectividad
c. [c]argar grafo desde fichero
i. Mostrar [i]nformacion basica del grafo
a. Mostrar la lista de [a]dyacencia del grafo
o. Mostrar c[o]mponentes conexas del grafo
q. Finalizar el programa
Introduce la letra de la accion a ejecutar >
  
```

Asimismo, y también útil para esta primera fase, ponemos a disposición del alumnado, un ejecutable de esta primera actividad para poder realizar pruebas.

### Segunda fase: recorridos en grafos

Un recorrido sobre un grafo es una estrategia que permite visitar sus nodos de una forma sistemática y ordenada. Se pretende que, dada cierta filosofía de orden o prioridad que sea de interés, se visiten todos los nodos de manera eficiente: todos visitados y sin repetir visita.

La analogía más cercana a la filosofía de un recorrido es la búsqueda de un tesoro en un laberinto: debemos asegurarnos de no dejar zona por estudiar, pero sin repetir zonas porque nos harían entrar en ciclos en nuestra búsqueda. Por ello, precisamente, es útil llevar una memoria de los sitios visitados y de aquellos que dejamos pendientes por investigar. Siguiendo con la analogía, 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  $i$  sería el siguiente:

```
{inicialización}
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 vacío 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 pseudocódigo se pretende simplificar la estrategia de un recorrido, y no se define, por tanto, la gestión de los nodos que han de ser revisados y que están en la estructura general ToDo. ToDo puede ser, desde una bolsa opaca donde vamos introduciendo los nodos a ser estudiados sin priorización de salida alguna, o, en cambio, puede ser gestionado como una pila o como una cola.

En el caso de una **cola**, donde el nodo más antiguo es el primero en salir, hablamos de un recorrido en amplitud, *breadth search first*, *bfs*.

En el caso de una **pila**, el último nodo que entre es el primero en salir a ser estudiado, se trataría de un recorrido en profundidad o *deep first search*, *dfs*.

En el material dispuesto en el campus virtual sobre los recorridos, tema 6, aparece la expresión del procedimiento de *RecorridoEnAmplitud* con detalle (transparencia 3), mientras que la expresión del procedimiento general *RecorridoEnProfundidad* en su versión recursiva, con detalle (transparencia 6).

Los recorridos, cualquiera de ellos, son necesarios para visitar los nodos de cada una de las componentes conexas de un grafo dirigido, y en la segunda fase de esta



actividad, usaremos la expresión de *dfs* en su versión recursiva implementada en c++. Se puede encontrar en el fichero grafo\_.cpp y es:

```
void GRAFO::dfs(unsigned i, vector<bool> &visitado)
{
    visitado[i] = true;
    cout << i+1 << ", ";
    for (unsigned j=0; j<LS[i].size(); j++)
        if (visitado[LS[i][j].j] == false)
            dfs(LS[i][j].j, visitado);
}
```

Teniendo en cuenta que el recorrido en profundidad sobre un grafo no dirigido desde el nodo 1, visitará todos los nodos conectados a él, esto es, la componente conexa de ese nodo, podremos reiterar el recorrido sobre nodos no visitados hasta visitarlos todos. Cada vez que identifiquemos un nodo aún no visitado desde el que iniciar el recorrido, tendremos una componente conexa más.

Por tanto, ha de escribirse tal método para el análisis de la conexidad de un grafo no dirigido y añadirla como opción de este tipo de grafos. El método será capaz de identificar los nodos de cada una de la componentes conexas del grado y, en el caso de que sólo sea una, expresar que el grafo es conexo. La cabecera de este método es:

```
void GRAFO::ComponentesConexas()
```

Puede consultarse la expresión del procedimiento en la transparencia 12 del tema 6.

## Evaluación

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. Por tanto, el programa principal que gestiona las opciones de carga y presentación de los contenidos de las listas según el tipo de grafo debe funcionar correctamente, siendo capaz de poder cambiar entre varios ficheros de grafos sin tener que reinicializar el ejecutable.

Para poder acceder al apto + debe haberse implementado, eficientemente, la segunda fase de esta actividad.