Prácticas Grafos

1. Representación y Recorridos

Utilizando la representación de grafos mediante listas de adyacencia que se proporciona en el fichero **grafos.h** implementar en C las siguientes funciones

Implementar una función que cree el grafo de la figura 1 (diapositiva 28) con el siguiente prototipo

tipoGrafo *creaGrafo28()

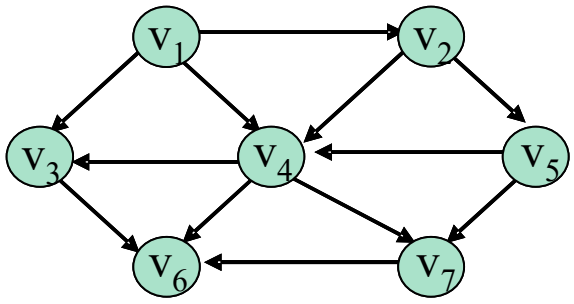
Implementar una función que inicie correctamente el directorio de vértices de un grafo cualquiera teniendo en cuenta que la única entrada del directorio que contiene información correcta del grafo es la correspondiente a las listas de adyacencia de cada uno de sus vértices, es decir, la información sobre las aristas del grafo

void iniciar(tipoGrafo *g)

Codificar las funciones que implemente los recorridos en profundidad y en amplitud de un grafo. Todos los vértices deben ser visitados en el recorrido independientemente del tipo de grafo (conexo o no) y del vértice inicial. Los prototipos a utilizar son los siguientes

void profundidad(int vInicio, tipoGrafo *g)

void amplitud(int vInicio, tipoGrafo *g)



– figura 1 -

```
GRAFO  (Orden 7)

      alc gEnt oTop dist  ant
    +----+----+----+----+----+
  1 |  0 |  0 |  0 |  * |  0 | -> 2 -> 3 -> 4
  2 |  0 |  1 |  0 |  * |  0 | -> 4 -> 5
  3 |  0 |  2 |  0 |  * |  0 | -> 6
  4 |  0 |  3 |  0 |  * |  0 | -> 3 -> 6 -> 7
  5 |  0 |  1 |  0 |  * |  0 | -> 4 -> 7
  6 |  0 |  3 |  0 |  * |  0 |
  7 |  0 |  2 |  0 |  * |  0 | -> 6
    +----+----+----+----+----+
```

- figura 2 -

Ayuda para la realización de la práctica

En el fichero **grafos.c** se proporciona una función que permite ver el contenido de la estructura que representa al grafo en memoria y que para el grafo de la práctica obtiene como resultado la salida que se muestra en la figura 2

Nota: debido a que los vértices del grafo de la diapositiva 28 se han identificado de 1 a 7 se ha optado por no utilizar la celda 0 del array para simplificar la comprensión de la práctica. Sin embargo no debe suponer ningún problema identificar los vértices de 0 a 6 y utilizar todas las celdas del directorio de vértices.

2. Grafos DIRIGIDOS. Ordenación topológica

Implementar una función que clasifique los vértices de un grafo de forma que si existe un camino del vértice v al vértice w, v aparece antes que w en la clasificación. Ampliar los algoritmos vistos en teoría para que la función permitan determinar si el grafo es cíclico, en ese caso la función debe devolver -1 indicando que no es posible la clasificación topológica. El prototipo de la función será el siguiente:

int ordenTop(tipoGrafo *g)

3. Grafos DIRIGIDOS. Caminos mínimos

Implementar las funciones que permiten obtener los caminos mínimos, si existen, desde un vértice inicial del grafo al resto de los vértices del grafo, para

a) Grafos no ponderados: algoritmos de caminos mínimos

b) Grafos ponderados: algoritmo de Dijkstra

Codificar las funciones adecuadas que permitan mostrar los caminos mínimos reales como una secuencia de vértices y el valor de sus distancias.

a) Entre dos vértices cualesquiera de un grafo

b) De un vértice inicial al resto de los vértices del grafo

4. Grafos NO DIRIGIDOS. Árbol de expansión mínimo

Obtener el árbol de expansión mínimo de un grafo no dirigido g, es decir, el subgrafo de g que conecta todos sus vértices con coste total mínimo y en el que se han eliminado todos los ciclos

a) Algoritmo de Prim

tipoGrafo *prim(tipoGrafo *g)

b) Algoritmo de Kruskal

tipoGrafo *kruskal(tipoGrafo *g)

NOTAS

- La mayoría de los algoritmos que se estudian en este tema tienen dos versiones, una básica y otra mejorada. Las versiones mejoradas se implementan incorporando una cola o una cola de prioridad. Aunque no se solicita ninguna versión en concreto se recomienda implementar al menos un algoritmo en alguna de sus versiones mejoradas, de forma que se trabaje con estos TADs.

- En concreto las versiones mejoradas de los algoritmos de Dijkstra y Prim necesitan un TAD Montículo cuyos elementos tienen como clave el valor de la distancia entre vértices (criterio de prioridad: distancia mínima) y como información el valor que identifica al vértice ajustado.
- El TAD montículo que se necesita en el algoritmo de Kruskal varia ligeramente. Se tienen que ordenar las aristas siendo el criterio de prioridad su peso. Por tanto el campo de información debe almacenar el valor de la arista correspondiente al peso de su campo clave.
- El algoritmo de Kruskal necesita además el TAD estructura partición. Se recomienda usar la implementación con árboles, aunque cualquiera de las implementaciones estudiadas sirve. Sólo hay que tener en cuenta que los elementos del conjunto sean los identificadores de los vértices (incluir o excluir el cero según se haya usado o no para identificar los vértices)

Normas generales

En la realización de las prácticas se deben seguir los siguientes criterios:

1. Es **obligatorio** utilizar los tipos y prototipos siempre que se proporcionan en el fichero cabecera. Si no se proporcionan debe crearse el fichero cabecera correspondiente o añadir al proporcionado los prototipos de todas las funciones que se implementen.
2. La codificación de las funciones se realizará en un fichero fuente diferente al del programa principal.
3. Para visualizar el correcto funcionamiento de las operaciones implementadas se debe crear un fichero de prueba desde el cual se llamará a las funciones implementadas.
4. Se debe crear el correspondiente Makefile para la correcta creación del fichero ejecutable.