Título de la actividad: Caminos mínimos en un digrafo.

Profesor Responsable: Sergio Alonso | Actividad III | Dificultad: media | Inicio: semana del 13 de mayo de 2019 y corrección la semana siguiente

Objetivo

El objetivo de esta actividad es añadir nuevas funcionalidades al menú de utilidades sobre grafos de las prácticas anteriores. En este caso, para el caso de grafos dirigidos o *digrafos* estudiaremos algoritmos para contruir los caminos mínimos de un nodo del grafo a todos los demás. Los métodos a implementar son el **algoritmo de Dijkstra** y el **algoritmo de Bellman-End-Moore**.

Temporalización

Esta actividad se divide en dos prácticas. En la hora de tutorización, se revisará la estructura del grafo y sus métodos, pues ahora sí es necesario cargar y mostrar los costes o pesos por cada arco. Finalmente, se recordará brevemente el problema de caminos mínimos en un grafo dirigido entre un nodo y todos los demás y se analizará las dificultades de implementación del algoritmo de Dijkstra. Además, plantearemos las dificultades de contar con costes negativos para la resolución de los problemas de caminos mínimos, y se presentará el algoritmo de Bellman-End-Moore, comparándolo con el ya estudiado de Dijkstra. En la última fase, se presentará por parte del estudiante el ejecutable, que será corregido.

Implementación

La estructura del programa a implementar será también la de un menú de opciones, que añadirá a las ya vistas en la actividad anterior, las utilidades de resolución de los problemas de caminos mínimos con los algoritmos mencionados. De nuevo el grafo se lee del fichero, siendo los grafos de esta actividad todos dirigidos y con costes o pesos asociados a los arcos.

La clase `GRAFO` incluye, además, dos nuevos métodos para la resolución de los problemas de caminos mínimos.

```
class GRAFO
 {
    unsigned             n;          // número de nodos o vértices
    unsigned             m;          // número de arcos o aristas
    unsigned             dirigido;   // si el grafo es dirigido, 1
    vector<LA_nodo>      LS;         // lista de sucesores o de adyacencia
    vector<LA_nodo>      LP;         // lista de predecesores
 public:
     unsigned   Es_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();
     void ComponentesConexas();
     void dfs(unsigned i, vector<bool> &visitado);
```

```
    void Dijkstra();
    void BellmanEndFord();
    ~GRAFO();
```

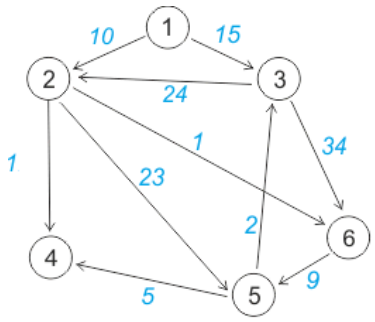
Las definiciones de las estructuras de datos necesarias, las de la clase `GRAFO` y otras comunes, se almacenarán en el fichero de cabeceras **grafo.h**

Es muy conveniente revisar los métodos de carga de datos del fichero de texto, la construcción de las estructuras de listas de los grafos y la salida por pantalla de esta información, para que tengan en cuenta el coste asociado a los arcos o aristas.

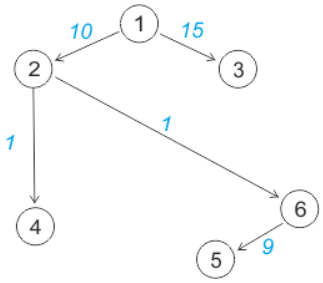
Cómo guardar los caminos mínimos

Usaremos el grafo siguiente para ilustrar los elementos de básicos de los problemas de caminos mínimos, en lo que afecta al desarrollo de la presente actividad. Se encuentra codificado como `grafo2.gr` en la biblioteca de recursos del campus virtual de la asignatura.

```
6 10 1
1 2 10
1 3 15
2 4 1
2 5 23
2 6 1
3 2 24
3 6 34
5 3 2
5 4 5
6 5 9
```



Los caminos mínimos entre pares de nodos de un grafo dirido, han de cumplir la siguiente condición necesaria y suficiente *los subcaminos que contiene deben ser también mínimos*. Por ello, realmente, al construir un camino minimo entre un par de nodos, estamos contruyendo los caminos mínimos de los nodos que atraviesa. Tal es así, que si usamos lo arcos que participan en los caminos mínimos de un nodo al resto, éstos forman una arborescencia con raiz el nodo origen, es decir, el nodo 1. En el caso del grafo que nos ocupa, sería ésta:



Usemos esta ventaja para facilitar el almacenamiento de la información sobre los caminos mínimos, pues, si bien comprobamos que en la arborescencia, un nodo puede tener más de un sucesor, tiene un predecesor único. Por ello, el vector de nodos `pred`, es suficiente para codificar la arborescencia. De igual forma, hemos de almacenar la distancia acumulada a cada nodo desde el nodo origen, usando también un vector denominado *etiqueta distancia* denominado `d`. La codificación de los valores de la solución del grafo que nos ocupa serían:

nodo	1	2	3	4	5	6
pred	**1**	**1**	**1**	**2**	**6**	**2**
d	**0**	**10**	**15**	**11**	**20**	**11**

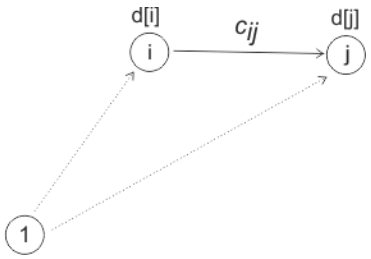
Por ello, el siguiente procedimiento recursivo, recorre el vector `pred`, mostrando los caminos mínimos almacenados:

```
void MostrarCamino(unsigned s, unsigned i, vector<unsigned> pred)
{
   if (i != s)
     {
      MostrarCamino(s,pred[i],pred);
      cout << pred[i]+1 << " -> ";
```

```
    }
}
```

Algoritmos

Los algoritmos para la construcción de caminos mínimos en un grafo desde un nodo a todos los demás, se basan en la búsqueda ordenada de mejoras de los valores de la etiqueta distancia, d, o lo que es lo mismo, la búsqueda de *atajos* a los caminos ya establecidos. En la imagen siguiente se muestra la base de las mejoras de la etiqueta distancia:



¿es el arco (i, j) un atajo para llegar a j desde el nodo 1? Simplemente hay que comparar d[j] con d[i]+cij, siendo cij, el coste o peso del arco (i, j).

El algoritmo de Dijkstra, resuelve el problema de forma óptima cuando no hay costes negativos en los arcos del grafo. El método a incorporar en el fichero **grafo.cpp** tendría el esquema siguiente:

```
void GRAFO::Dijkstra()
{
   vector<bool> PermanentementeEtiquetado;
   vector<int> d;
   vector<unsigned> pred;
   int min;
   unsigned s, candidato;

   //Inicialmente no hay ningun nodo permanentemente etiquetado
   PermanentementeEtiquetado.resize(n,false);
   //Inicialmente todas las etiquetas distancias son infinito
   d.resize(n,maxint);
   //Inicialmente el pred es null
   pred.resize(n,UERROR);

   //Solicitamos al usuario nodo origen
   cout << endl;
   cout << "Caminos minimos: Dijkstra" << endl;
   cout << "Nodo de partida? [1-"<< n << "]: ";
   cin  >> (unsigned &) s;
   //La etiqueta distancia del nodo origen es 0, y es su propio pred
   d[--s]=0; pred[s]=s;
   do
     {
      - Buscamos un nodo candidato a ser permanentemente etiquedado: aquel de
        entre los no permanentemente etiquetados con menor etiqueta
        distancia no infinita.
      - Si existe ese candidato, lo etiquetamos permanentemente y usamos los
        arcos de la lista de sucesores para buscar atajos.
      - Esto lo hacemos mientras haya candidatos

     }
   while (min < maxint);

   cout << "Soluciones:" << endl;
   //En esta parte del código, mostramos los caminos mínimos, si los hay
}
```

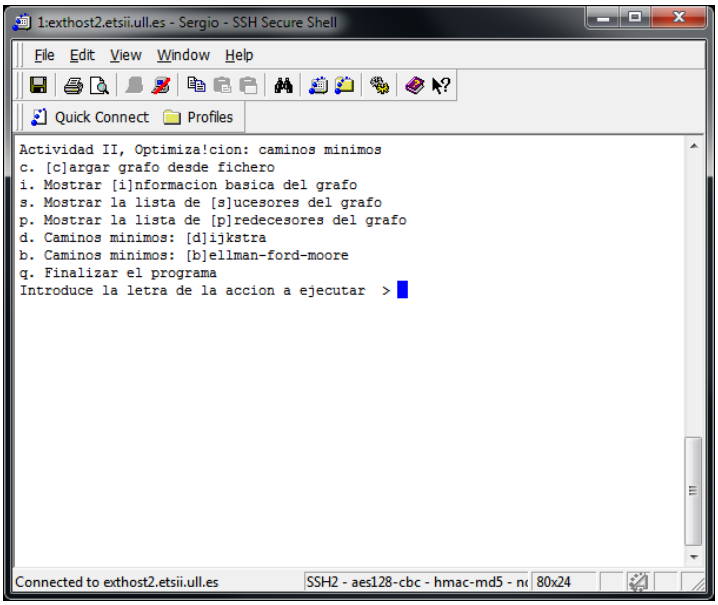
El **algoritmo de Bellman-End-Moore** usa otra estrategia que deriva en mayor número de comparaciones que el algoritmo anterior, pero permite dar soluciones cuando hay costes

negativos en los arcos, y es, además, capaz de detectar cuando el grafo contiene un ciclo de coste negativo. El método a incorporar en el fichero **grafo.cpp** tendría el esquema siguiente:

```
void GRAFO::BellmanEndMoore()
{
   vector<int> d;
   vector<unsigned> pred;
   unsigned s, numeromejoras = 0;
   bool mejora;

   //Idem que en el algoritmo de Dijkstra
   d.resize(n,maxint);
   pred.resize(n,UERROR);

   cout << endl;
   cout << "Caminos minimos: Bellman - End - Moore" << endl;
   cout << "Nodo de partida? [1-"<< n << "]: ";
   cin  >> (unsigned &) s;
   d[--s]=0; pred[s]=s;

do
      {
// recorremos todos los arcos, y para cada (i, j), buscamos si d[j] > d[i] +
cij, y actualizamos d y pred
//si al menos en una ocasion ha mejorado una etiqueta distancia, no hemos
terminado; contabilizamos los ciclos en los que ha habido mejoras
      }
while ((numeromejoras < n) && (mejora == true));
//para salir del bucle, si mejora es true, pues hay un ciclo, pues hemos
realizado n+1 veces la relajacion con mejora; si mejora es false, pues
tenemos solucion

//Mostramos los caminos mínimos que se puedan haber encontrado, o advertimos
de la existencia de un ciclo de coste negativo.

}
```


Ficheros

De igual forma que en la actividad anterior, además del fichero de cabecera grafo.h y con el código de desarrollo de sus métodos y procedimientos, **grafo.cpp**, se trabajará con **pg2.cpp** que incluye el programa principal `main`, que será un simple gestor tipo menú. Las pantallas con las opciones dependerán del tipo de grafo que sea, así, para un grafo dirigido, mostrará:



Mientras que, para un grafo no dirigido, no cambiará respecto a la actividad anterior.

Evaluación

Para superar esta actividad, los métodos para los algoritmos deberán estar correctamente implementados y deberán funcionar con los grafos de prueba. El profesor podrá valorar la limpieza, documentación del código y la optimización de los recursos para puntuar por encima del apto.