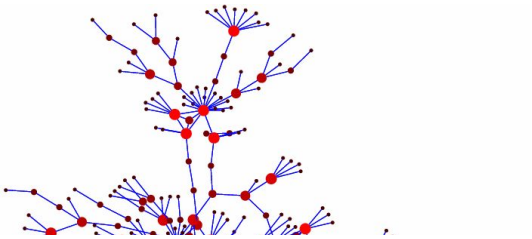


Actividad II: Árboles generadores de mínimo coste: el algoritmo de Kruskal

Profesor responsable: Sergio Alonso.

Dificultad: media.

Presentación del plan de trabajo: lunes, 27 de abril de 2020.

Fecha límite para la entrega de la práctica: jueves, 7 de mayo de 2020, 23:59 horas.

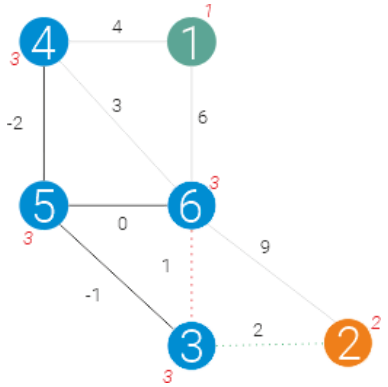
Objetivo

El objetivo de esta práctica es la implementación de un algoritmo capaz de construir el árbol generador de mínimo coste, MST, de un grafo no dirigido con costes o pesos en sus aristas. Para resolver el problema MST usaremos el algoritmo de Kruskal (1956) que lo construye, por lo que debemos inducir un orden entre las aristas según sus costes, para examinarlas en ese orden y decidir si entran o no en la solución, siempre evitando los ciclos.

El Algoritmo de Kruskal

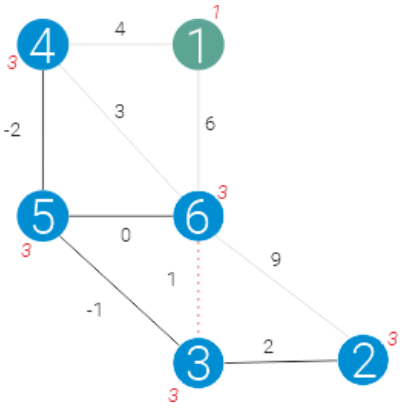
El algoritmo de Kruskal, selecciona las aristas que entran en la solución de forma que no construyan un ciclo con las previamente seleccionadas, además de analizar las candidaturas por orden no decreciente de su peso o coste.

En el algoritmo de Kruskal, se controla de forma muy sencilla cuando una arista (*i*, *j*), formará un ciclo o no con las aristas previamente seleccionadas para la solución. Usa un sencillo sistema de etiquetado de los nodos que están en cada componente conexa. En el siguiente ejemplo, se ilustra cómo, pensemos que ya se han añadido algunas aristas de coste bajo a una solución parcial:



- se han añadido por ahora: (4, 5), de coste -2; (3, 5) de coste -1; (5, 6) de coste 0; *(sí, tienen coste negativo, pero eso no es problema en el MST)*.
- las componentes conexas con las aristas añadidas serían: componente conexa 1, verde, {1}; componente conexa 2, naranja, {2}; componente conexa 3, azul, {3, 4, 5, 6}.

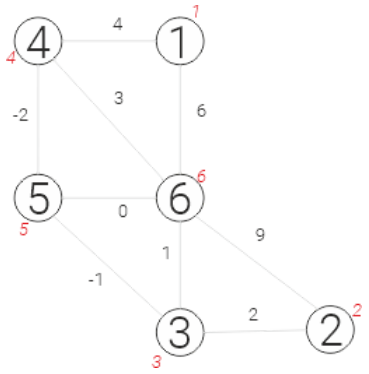
- siguiente arista candidata, (3, 6), forma ciclo, pues los dos nodos están en la misma componente conexa: se rechaza.
- siguiente arista candidata, (2, 3), no forma ciclo, el nodo 2 está en la componente conexa etiquetada con el 2; el nodo 3 está en la componente conexa etiquetada con el 3: se acepta.
- una vez aceptada, se actualizan las componentes conexas y sus etiquetas:



En pseudocódigo, sin mucho detalle, sería:

```
Ordenar las aristas en orden no decreciente según sus costes
T=∅;
Para todo nodo i de V hacer raiz[i] = i;
Mientras en T no haya n-1 aristas hacer
      {
      Sea e=(i,j) la siguiente arista de la lista ordenada;
      Si raiz[i] <> raiz[j] entonces
            {
            T= T ∪ {e};
            kill = raiz[i];
            Para todo nodo k de V hacer
                  {
                  Si raiz[k] = kill entonces raiz[k]=raiz[j]
                  }
            }
      }
```

Por tanto, sólo añadir que la posición de etiquetado inicial de las componentes conexas, cuando no hay ninguna arista aún en la solución, sería el que reflejara que cada nodo está en su propia componente conexa unitaria:



Plan de trabajo

Para implementar el algoritmo de Kruskal para la construcción del MST, hay que resolver tres obstáculos:

- revisión y adaptación de las estructuras para que puedan trabajar con grafos pesados.
- ordenación de las aristas según sus costes de forma lo más eficiente posible.
- controlar las componentes conexas que se van conformando a medida de que se van añadiendo a la solución parcial.

Adaptación de las estructuras y métodos para trabajar con grafos pesados

El grafo de ejemplo, tendría como codificación, la siguiente:

```
6 9 0
4 1 4
4 5 -2
4 6 3
5 6 0
3 5 -1
3 6 1
2 3 2
1 6 6
2 6 9
```

Por tanto, para cada línea de información de las 9 aristas, **no sólo** se leen los dos extremos (recordemos que es un par no ordenado), **sino también el tercer valor** de la línea que es el coste, que es de tipo integer. Por tanto, la lectura de cada una de esas líneas debe cambiar a algo similar a esto:

```
textfile >> (unsigned &) i  >> (unsigned &) j >> (int &) c;
```

rehaciendo el código para que incorpore el dato del coste en las estructuras, tanto si lo que se lee en un grafo no dirigido, como si es el dirigido, que será necesario para la próxima práctica.

Asimismo, **hay que adaptar el método que muestra las listas**, para que también muestra por pantalla los pesos o costes de cada nodo, tanto si es en la lista de adyacencia, como si es en las listas de sucesores o como en la de predecesores.

Ordenación de las aristas según sus costes

La implementación del algoritmo de Kruskal **necesita** que las aristas han de ser analizadas en orden no decreciente de costes. Para ello, proponemos construir un vector con las aristas y sus costes, *manteniendo un orden parcial de interés*, como veremos. Por ello, se ha incluido en el fichero `grafo.h`, a partir de esta práctica la siguiente estructura de datos:

```
typedef struct {
        unsigned extremo1, extremo2;
        int peso;
    } AristaPesada;
```

Y así, cargar la información de la lista de adyacencia del grafo de `LS` a este vector:

```
vector <AristaPesada> Aristas;

/*Cargamos todas las aristas de la lista de adyacencia*/
Aristas.resize(m);
unsigned k = 0;
for (unsigned i = 0; i<n; i++)
      {
      for (unsigned j=0; j<LS[i].size();j++)
            {
            if (i < LS[i][j].j)
                  {
                  Aristas[k].extremo1 = i;
                  Aristas[k].extremo2 = LS[i][j].j;
                  Aristas[k++].peso = LS[i][j].c;
                  }
            }
      };
```

Por tanto, es en el vector `Aristas` en el que buscaremos esa arista de menor coste candidata a entrar en la solución. Estrategias para ello:

- ordenar todo el vector por orden no decreciente de costes (puede usarse cualquier método)
- ir situando en las posiciones de cabeza las aristas con menos coste, a medida que el algoritmo lo necesite.

Sería algo así:

i. Vector Aristas cargado de la lista de adyacencia en `LS`: cada fila contiene la posición 0..m-1, los dos extremos (recordemos que el nodo `i` aparece como `i-1`), y el coste `c`:

índice	extremo	extremo	coste	head
0	0	3	4	
1	0	5	6	
2	1	2	2	
3	1	5	9	
4	2	4	-1	
5	2	5	1	
6	3	4	-2	
7	3	5	3	
8	4	5	0	

ii. Posicionamos el valor de la cabeza head a 0, y la arista de menor coste es buscada para que ocupe esa posición, y por tanto, queda:

índice	extremo	extremo	coste	head
0	**3**	**4**	**-2**	**0**
1	0	5	6	
2	0	3	4	
3	1	5	9	
4	1	2	2	
5	2	5	1	
6	2	4	-1	
7	3	5	3	
8	4	5	0	

¿Cómo se ha hecho? Hemos recorrido las posiciones desde la 1 (`head+1`) en adelante, comparando el coste de la arista de la posición `head` con las siguientes; si encontramos una arista de menor coste que en `head`, la intercambiamos; así, al terminar, tendremos la arista de menor coste en la posición `head`. Al ser la de menor coste, es la arista candidata para poder entrar en el MST.

iii. Posicionamos el valor de la cabeza `head` a 1, repitiendo: la arista de menor coste entre las posiciones `1` a `m-1` es buscada para que ocupe esa posición, y por tanto, queda:

índice	extremo	extremo	coste	head
0	3	4	-2	
1	2	4	-1	**1**
2	0	5	6	
3	1	5	9	
4	0	3	4	
5	1	2	2	
6	2	5	1	
7	3	5	3	
8	4	5	0	

¿Cómo se ha hecho? De nuevo hemos recorrido las posiciones desde la 2 (`head+1`) en adelante, comparando el coste de la arista de la posición `head` con las siguientes; si encontramos una arista de menor coste que en `head`, la intercambiamos; así, al terminar, tendremos la arista de menor coste en la posición `head`. Al ser la de menor coste, es la arista candidata para poder entrar en el MST.

iv. Ahora posicionamos el valor de la cabeza `head` a 2, repitiendo: la arista de menor coste entre las posiciones `2` a `m-1` es buscada para que ocupe esa posición, y por tanto, queda:

índice	extremo	extremo	coste	head
0	3	4	-2	
1	2	4	-1	
2	4	5	0	**2**
3	1	5	9	
4	0	5	6	
5	2	4	4	
6	1	2	2	
7	3	5	3	
8	2	5	1	

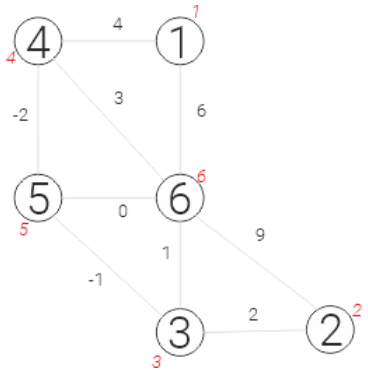
¿Cómo se ha hecho? De nuevo hemos recorrido las posiciones desde la 3 (`head+1`) en adelante, comparando el coste de la arista de la posición `head` con las siguientes; si encontramos una arista de menor coste que en `head`, la intercambiamos; así, al terminar, tendremos la arista de menor coste en la posición `head`. Al ser la de menor coste, es la arista candidata para poder entrar en el MST.

¿Qué estamos haciendo? Estamos ordenando el vector de aristas, sí, pero poco a poco, de una forma que deja siempre el menor de las posiciones restantes al principio, identificando la arista que pasaría a ser candidata para el algoritmo de Kruskal. Date cuenta que si el algoritmo de Kruskal finaliza mucho antes de que se recorran todas las aristas, no hará falta ordenarlo completamente.

Controlando las componentes conexas que se van conformando a medida de que se van añadiendo a la solución parcial

Para controlar que no se formen ciclos, el algoritmo de Kruskal etiqueta a cada nodo con la componente conexa a la que pertenece. Inicialmente, dado que no hay aristas en la solución, cada nodo está en su propia componente conexa que lo tiene a él como único elemento.

Sería el caso de esta imagen que ya hemos usado:



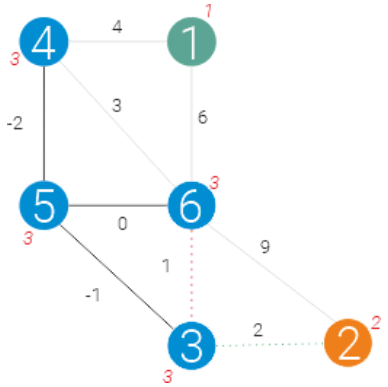
Con lo que tendríamos, inicialmente que:

```
/*Inicializamos el registro de componentes conexas: cada nodo está en su componente conexa*/
vector <unsigned> Raiz;
Raiz.resize(n);
for (unsigned q = 0;q < n; q++)
      {
      Raiz[q]=q;
      };
```

Y, por tanto, en el vector `Raiz`, para cada nodo sería, inicialmente:

posición	0	1	2	3	4	5
valor	0	1	2	3	4	5

Añadiendo tres aristas, las de menor coste por orden, tendríamos esta solución parcial:

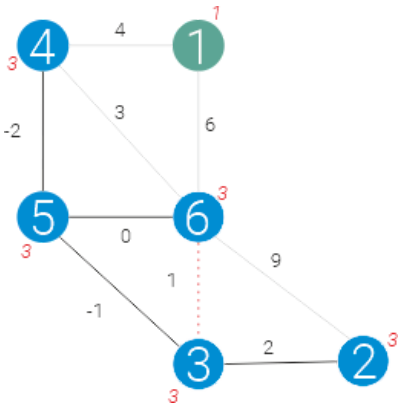


En donde las componentes conexas son, {3, 4, 5, 6), {2}, y {1}. Por tanto, el vector Raiz sería algo como este:

posición	0	1	2	3	4	5
valor	0	1	2	2	2	2

La arista candidata, (3, 6), tiene sus nodos en las componentes conexas, `Raiz[3-1]` y `Raiz[6-1]`, que tienen como valor, 2, ambas. Por tanto, sus nodos están en la misma componente conexa, y no se puede añadir a la solución parcial, pues crearía un ciclo.

La siguiente arista candidata, (2, 3), tiene sus nodos en las componentes conexas, `Raiz[2-1]` y `Raiz[3-1]`, que tienen como valor, 1 y 2, respectivamente. En este caso, sus nodos están en componente conexas distintas, y sí se puede añadir a la solución parcial, pues no crearía un ciclo. La añadimos. En ese caso, hay que actualizar `Raiz` para que refleje que las componentes conexas {2} y {3, 4, 5, 6} se han unido.



Y el vector `Raiz`, sería:

posición	0	1	2	3	4	5
valor	0	2	2	2	2	2

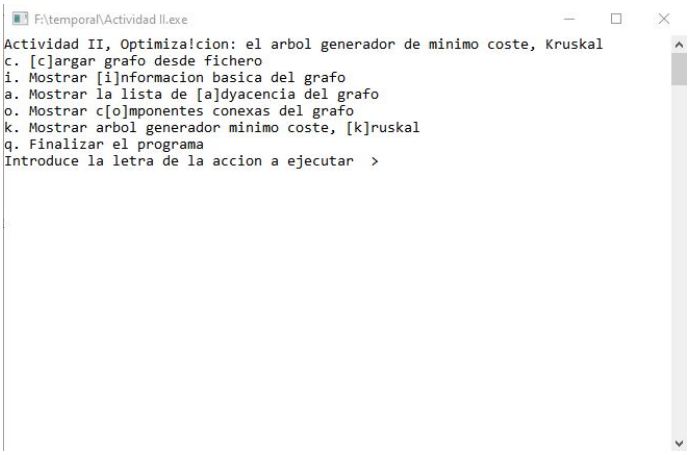
Sólo habría que reemplazar cualquier valor de `Raiz[2-1]` por `Raiz[3-1]`.

Métodos a incluir

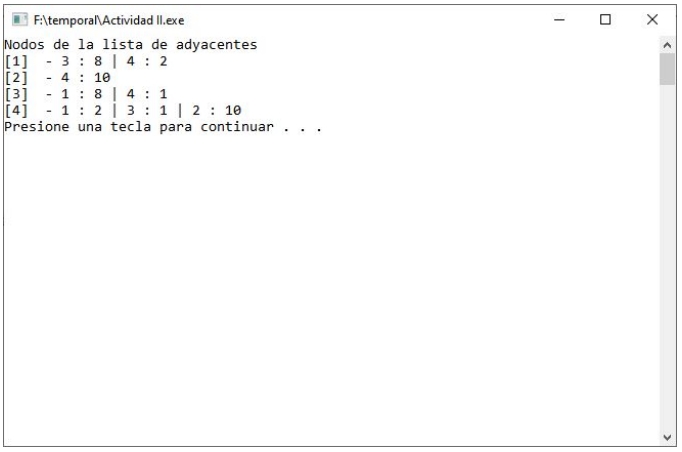
Si bien hay que actualizar los métodos de carga de la información de los ficheros de entrada, para que se trabaje con grafos pesados, además de modificar los métodos que muestran su contenido para que también muestren los costes, el único método que hay que implementar es el algoritmo de Kruskal,

```
void GRAFO::Kruskal();
```

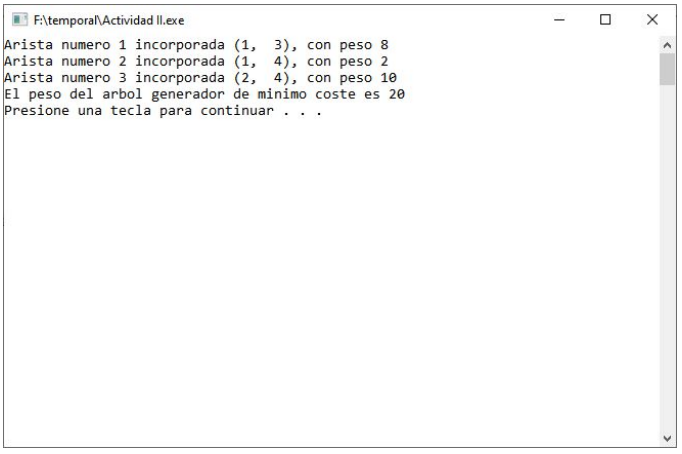
incluyendo la opción en el menú de grafos no dirigidos para que se construya el MST mediante este algoritmo:



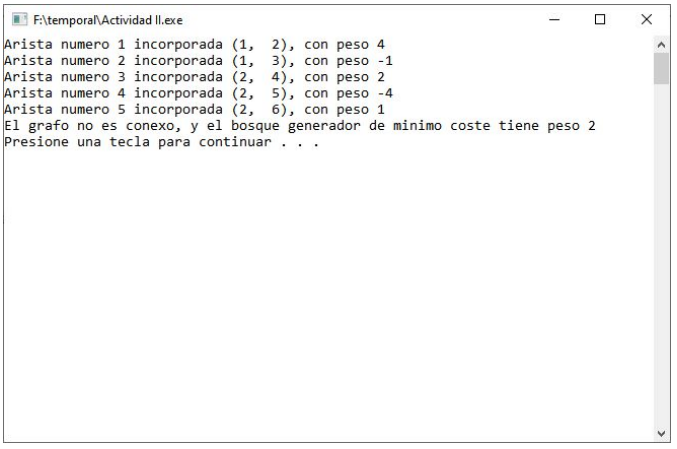
Se han modificado los procedimientos de carga y se ha añadido la exposición de los pesos en pantalla:



La resolución se muestra por pantalla: tanto las aristas que forman parte del MST como el peso total del mismo,



Incluso, si el grafo de partida no es conexo, es capaz de construir el bosque de árboles generadores de mínimo coste:



Evaluación

Para superar esta actividad, tanto la modificación de la estructura y los métodos para que pueda trabajarse con grafos pesados, como la implementación del algoritmo de Kruskal para la construcción del MST, debe funcionar correctamente.

Para poder acceder al apto+ debe haberse implementado, eficientemente, la ordenación parcial de las aristas, así como asegurarse de que el método funciona para grafos no conexos. Esta condición es necesaria, aunque no suficiente para la máxima calificación, en la que intervendrán aspectos como la documentación y limpieza del código así como la incorporación de mejoras en el mismo.