Algorítmica

Algoritmos voraces

Álvaro Maximino Linares Herrera

Kruskal

La tercera práctica que voy a realizar es el algoritmo de Kruskal.

Aquí tenemos el pseudocódigo:

```
function Kruskal(G)
   Para cada v en V[G] hacer
     Nuevo conjunto C(v) ← {v}.
   Nuevo heap Q que contiene todas las aristas de G, ordenando por su peso.
   Defino un arbol T ← Ø
   // n es el número total de vértices
   Mientras T tenga menos de n-1 vertices hacer
     (u,v) ← Q.sacarMin()
     // previene ciclos en T. agrega (u,v) si u y v estan diferentes componentes
en el conjunto.
     // Nótese que C(u) devuelve la componente a la que pertenece u.
     if C(v) ≠ C(u) then
       Agregar arista (v,u) to T.
       Merge C(v) y C(u) en el conjunto
   Return arbol T
```

Kruskal lo que hace es devolvernos el árbol en el que todos los vértices están, no hay un vértice que no se incluya en el árbol. El árbol que nos devuelve tiene todas las aristas que suman el mínimo.

Funciona de la siguiente manera:

- Creamos un bosque, B, que contenga todos los vértices, que serán árboles separados.

- Creamos un conjunto, C, que contenga todas las aristas del grafo.

- Mientras C es no vacío:

    - Eliminamos una arista de valor mínimo del conjunto C.

    - Si esa arista conecta dos árboles diferentes se añade al bosque, combinando ambos árboles en uno solo.

    - Si no es así la desechamos.

- Al final tendremos un bosque con un solo componente que contendrá lo que buscamos, el árbol de expansión mínima.

Aquí tenemos el código entero. Usamos la matriz de adyacencia para saber que nodos están conectados. Usare una matriz de tipo C y para el resto de matrices vectores de la std, incluidos en la librería vector. En la matriz de adyacencia uso la constante INF con valor bastante grande para indicar cuales no están conectados. En el código que adjunto aquí tengo el ejemplo que sale en la wikipedia, para comprobar si el código es correcto o no. En la función Imprimir muestro que nodos están conectados, como la matriz solución es triangular para que no muestre datos repetidos, que el nodo 3 este conectado con el 5 y el 5 con el 3, le pongo un if(i<=j) en el bucle que recorre la matriz para que me muestre solo la parte que está por encima de la diagonal principal.

```
#include <iostream>
#include <vector>

using namespace std;

int cn=7; //cantidad de nodos
const int INF=999999;
int adyacencia[7][7]={
                    {INF,7,INF,5,INF,INF,INF},
                    {7,INF,8,9,7,INF,INF},
                    {INF,8,INF,INF,5,INF,INF},
                    {5,9,INF,INF,15,6,INF},
                    {INF,7,5,15,INF,8,9},
                    {INF,INF,INF,6,8,INF,11},
                    {INF,INF,INF,INF,8,11,INF}}; //matriz de adyacencia

// Devuelve la matriz de adyacencia del árbol mínimo.
vector< vector<int> > kruskal(){
    vector< vector<int> > arbol(cn);
    vector<int> pertenece(cn); // indica a que árbol pertenece el nodo
 //arbol es la solucion
//cn cantidad de nodos, filas y columnas matriz cuadrada simetrica
//arcos los arcos que ahora mismo tenemos
    for(int i = 0; i < cn; i++){
       arbol[i]= vector<int>(cn,INF);
       pertenece[i] = i;
    }

    int nodoA;
    int nodoB;
    int arcos = 1;
    while(arcos < cn){
        // Encontrar  el arco mínimo que no forma ciclo y guardar los nodos y la distancia.
        int min = INF;
        for(int i = 0; i < cn; i++)
            for(int j = 0; j < cn; j++)
                if(min > adyacencia[i][j] && pertenece[i] != pertenece[j]){ //pertenece[i]!=pertenece[j], si tienen el
mismo valor significa que estan conectados
                 min = adyacencia[i][j];
```

```
                nodoA = i;
                nodoB = j;
            }

        // Si los nodos no pertenecen al mismo árbol agrego el arco al árbol mínimo.
        if(pertenece[nodoA] != pertenece[nodoB]){
            arbol[nodoA][nodoB] = min;
            arbol[nodoB][nodoA] = min;

            // Todos los nodos del árbol del nodoB ahora pertenecen al árbol del nodoA.
                int temp = pertenece[nodoB];
                pertenece[nodoB] = pertenece[nodoA];
                for(int k = 0; k < cn; k++)
                        if(pertenece[k] == temp)
                                pertenece[k] = pertenece[nodoA];

            arcos++;
        }
    }
    return arbol;
}

void Imprimir(vector< vector<int> > mat, int fc){
for(int i=0; i<fc; i++){
 for(int j=0; j<fc; j++){
            if(i<=j)
            if(mat[i][j]!=INF){
                        cout << "El nodo " << i << " está conectado con el nodo " << j << endl;
            }
 }
}
}

int main(){

vector < vector<int> > solucion=kruskal();
Imprimir(solucion,cn);
}
```

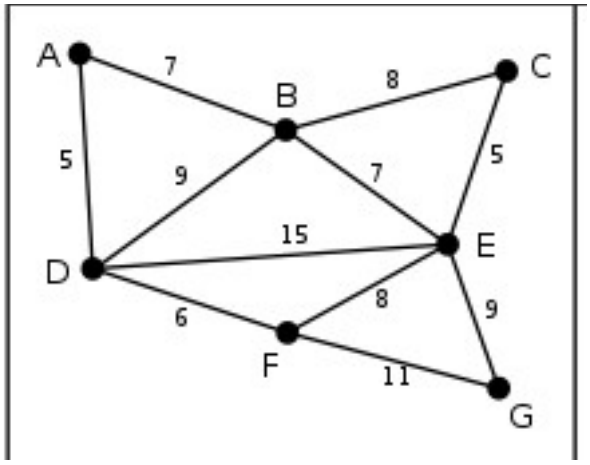
Orden de eficiencia

El orden de complejidad del algoritmo viene determinado por el número de aristas y de vértices, como es el mismo, n, tendremos que el algoritmo tiene una complejidad de orden O(n log n), si fuesen diferentes por ejemplo vértices m y aristas n, el orden sería O(m log n).

Primero ordenamos las aristas con un orden de O(m log m) así hacemos que al eliminar una arista de C se ejecute en un tiempo constante. Para ver que vértices estén en qué componentes lo hacemos con un orden O(m) ya que para cada arista hay una operación de búsqueda y otra de unión en caso de que sea mínimo. Por tanto el orden de complejidad es de O(m log n).

Ejecuciones

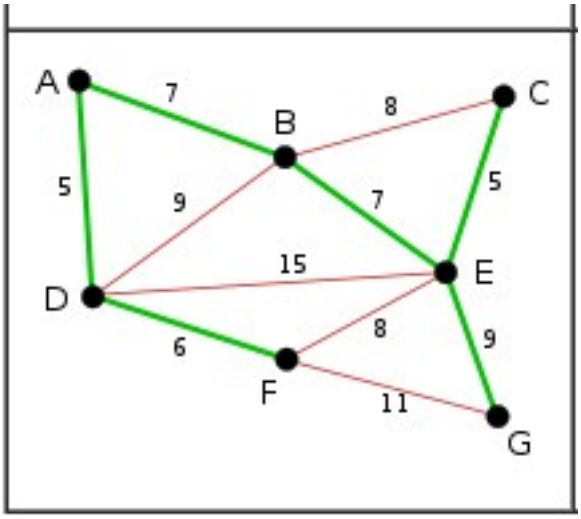
Para el ejemplo que viene en el código que he puesto antes, sacado de la wikipedia, vemos que el grafo sería el siguiente:



Que en el código meteríamos como:

{INF,7,INF,5,INF,INF,INF},
{7,INF,8,9,7,INF,INF},
{INF,8,INF,INF,5,INF,INF},
{5,9,INF,INF,15,6,INF},
{INF,7,5,15,INF,8,9},
{INF,INF,INF,6,8,INF,11},
{INF,INF,INF,INF,8,11,INF}

La solución viene representada en la web como sigue:



Donde vemos que:

El nodo A esta conectado con B y con D. El nodo B esta conectado con E. El nodo C esta conectado con E. El nodo D esta conectado con F. El nodo E esta conectado con G.

Ahora veamos si el programa realiza bien el algoritmo de Kruskal.

```
>g++ -o kruskal kruskal.cpp
>./kruskal
El nodo 0 está conectado con el nodo 1
El nodo 0 está conectado con el nodo 3
El nodo 1 está conectado con el nodo 4
El nodo 2 está conectado con el nodo 4
El nodo 3 está conectado con el nodo 5
El nodo 4 está conectado con el nodo 6
```

Como vemos si que nos da la solución correcta.

Veamos ahora otro ejemplo:

{INF,6,7,3,2},

{6,INF,INF,INF,INF},

{7,INF,INF,3,5},

{3,INF,3,INF,9},

{2,INF,5,9,INF}

Donde la cantidad de nodos es 5, luego la matriz de adyacencia es de 5x5.

```
>g++ -o kruskal kruskal.cpp
>./kruskal
El nodo 0 está conectado con el nodo 1
El nodo 0 está conectado con el nodo 3
El nodo 0 está conectado con el nodo 4
El nodo 2 está conectado con el nodo 3
```

Como vemos nos devuelve los nodos que están conectados sin ningún tipo de problema y correctamente.

Veamos otro ejemplo de ejecución:

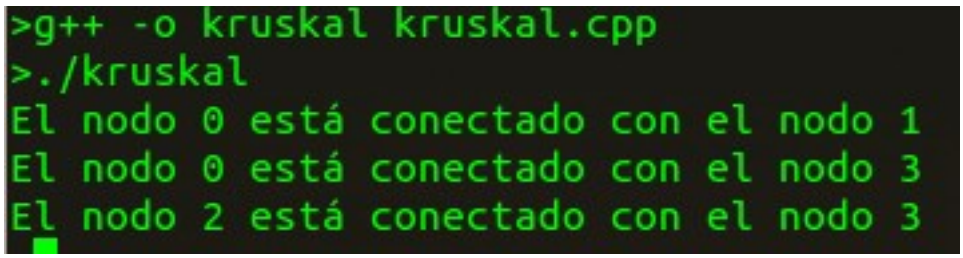
{INF,2,INF,3},

{2,INF,INF,INF},

{INF,INF,INF,3},

{3,INF,3,INF}

Donde la cantidad de nodos es 4, entonces tenemos que la matriz de adyacencia es de tamaño 4x4.



Como vemos nos vuelve a devolver el grafo de expansión mínima.