ALGORTIMO DE KRUSKAL

QUÉ ES EL ALGORITMO DE KRUSKAL

Este algoritmo nos proporciona un grafo acíclico, conexo y ponderado de coste mínimo, o, lo que es lo mismo, un árbol generador de coste mínimo, a partir de un grafo dado.

Nos devolverá un grafo ponderado del tipo tCoste.

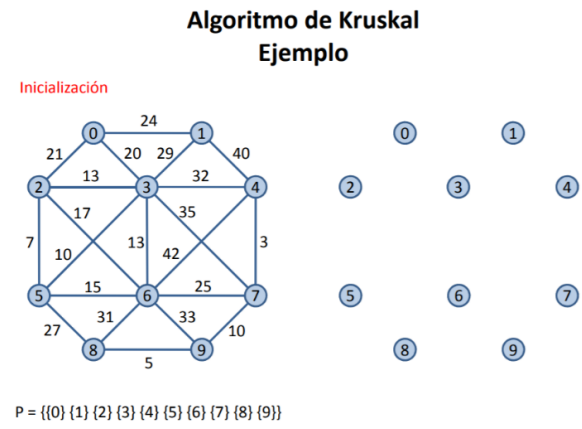
Este algoritmo es de orden $O(n^2 \cdot \log(n))$.

Con qué tipo de grafos podemos usar este algoritmo

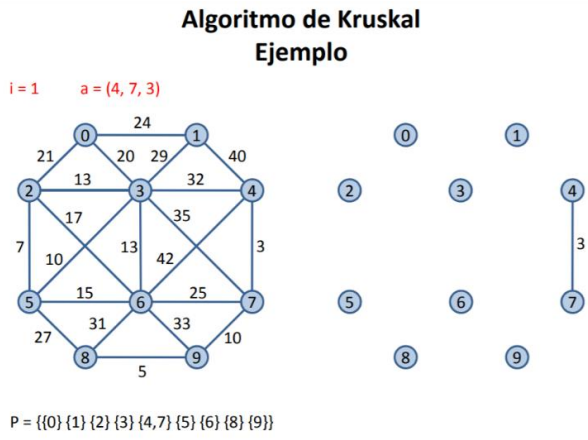
i. Ponderado
ii. No dirigido
iii. Conexo

CÓMO FUNCIONA EL ALGORITMO DE KRUSKAL

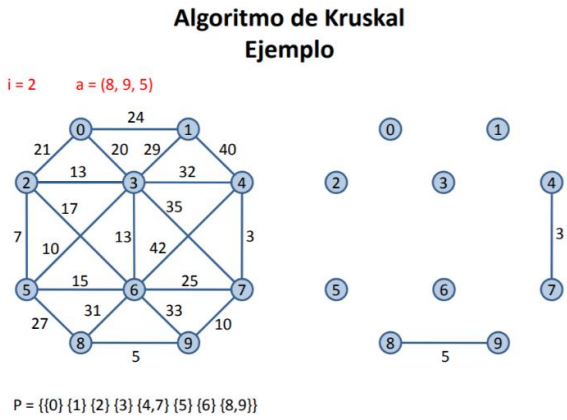
El algoritmo recibe un grafo no dirigido ponderado. Veremos qué hace este algoritmo mediante el ejemplo de las transparencias de teoría.



A la izquierda tenemos el grafo recibido en la función, a la derecha los n vértices que tiene el grafo y abajo a la izquierda los diferentes subgrafos que hay en el grafo de la derecha.



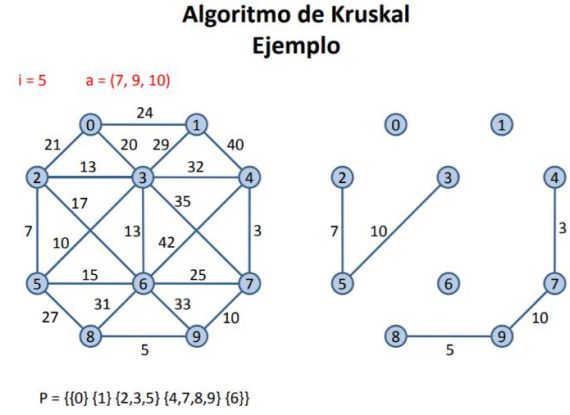
En primer lugar, buscaremos la arista de menor coste y uniremos los vértices que intervienen en esa arista. En P, eliminamos el {7} y lo incluimos en {4}, quedando así {4,7}.



Luego, buscamos la siguiente arista con el menor coste, en este caso la arista tiene un coste de 5. Al igual que antes, unimos los vértices que participan en esa arista y uniremos los vértices en P, de tal forma que eliminamos {9} y lo incluimos en {8}.

Realizamos este proceso sucesivamente hasta que nos quede un grafo conexo acíclico, cómo sabemos que es acíclico:

Tras realizar este proceso varias veces llegamos a este punto:



Acabamos de colocar las aristas de coste 10. El siguiente valor de las aristas es 13, por lo que vemos, hay dos aristas, esto es casualidad, no tiene el porqué tener que coincidir con otras aristas. Veamos, si colocamos la arista que une los vértices 3 y 6 **no hay un ciclo**; sin embargo, si escogemos la arista de los vértices 2 y 3 **sí hay un ciclo**. Para saber si hay un ciclo o no, nos debemos preguntar si podemos volver a ese vértice sin ir tocar dos veces la misma arista, por ejemplo, en el caso del vértice 2 y 3, si unimos esos dos vértices podemos volver al vértice 3 sin pisar dos veces la misma arista, o dicho con otras palabras, existe un camino hacia el vértice 3 que cada arista es diferente ya que podemos ir del vértice 3 al 2, del 2 al 5 y del 5 volver al 3, y eso significa que es cíclico, mientras que en el caso de los vértices 3 y 6 no es cíclico porque, si partimos del vértice 6, vamos al vértice 3, pero para volver al vértice 6 debemos, obligatoriamente, volver a pasar por la misma arista.

CÓDIGO C++

Este algoritmo usa el TAD Partición.

TAD Partición

Especificación TAD Partición

Definición:

Una partición del conjunto de enteros C = {0, 1,…, n−1} es un conjunto de subconjuntos disjuntos cuya unión es el conjunto total C.

Operaciones:

Particion(int n);

Post: Construye una partición de subconjuntos unitarios del intervalo de enteros [0, n−1].

void unir (int a, int b);

Pre: 0 ≤ a, b ≤ n−1, a y b son los representantes de sus clases y a ≠ b.

Post: Une el subconjunto del elemento a y el del elemento b en uno de los dos subconjuntos arbitrariamente. La partición queda con un miembro menos.

int encontrar(int x) const;

Pre: 0 ≤ x ≤ n−1.

Post: Devuelve el representante del subconjunto al que pertenece el elemento x

Implementaciones

1. Vector de pertenencia
    a. Las operaciones Partición() y unir() ∈ $O(n)$ y la operación encontrar() ∈ $O(1)$.
2. Listas de elementos
    a. La operación encontrar() ∈ $O(1)$ y la operación unir() ∈ $O(n)$.
3. Bosque de árboles
    a. La operación encontrar() ∈ $O(n)$ y la operación unir() ∈ $O(1)$.
    b. Si tenemos un control de altura por tamaño o por altura las operaciones obtienen unos nuevos ordenes: encontrar() ∈ $O(\log n)$ y unir() ∈ $O(1)$.

Implementación del TAD Partición mediante bosque de árboles

```
/*-------------------------------------------------*/
/* particion.h */
/*-------------------------------------------------*/
#ifndef PARTICION_H
#define PARTICION_H

#include <vector>

class Particion {
public:
   Particion(int n): padre(n, -1) {}
   void unir(int a, int b);
   int encontrar(int x) const;
private:
```

```
   mutable std::vector<int> padre;
};
#endif // PARTICION_H
```

```
/*-------------------------------------------------------*/
/* particion.cpp */
/*-------------------------------------------------------*/
/* Implementación de la clase Particion: */
/* Bosque de árboles con unión por altura y búsqueda con */
/* compresión de caminos. */
/*-------------------------------------------------------*/
#include "particion.h"

// El árbol con mayor altura se convierte en subárbol del otro.
void Particion::unir(int a, int b)
{
   if (padre[b] < padre[a])
      padre[a] = b;
   else {
      if (padre[a] == padre[b])
         padre[a]--; // El árbol resultante tiene un nivel más.
      padre[b] = a;
   }
}

int Particion::encontrar(int x) const
{
   int y, raiz = x;
   while (padre[raiz] > -1)
      raiz = padre[raiz];
   // Compresión del camino de x a raíz: Los nodos
   // del camino se hacen hijos de la raíz.
   while (padre[x] > -1) {
      y = padre[x];
      padre[x] = raiz;
      x = y;
   }
   return raiz;
}
```

Algoritmo de Kruskal

```
template <typename T> class GrafoP {
public:
    typedef T tCoste;
    typedef size_t vertice;
    struct arista {
           vertice orig, dest;
           tCoste coste;
           explicit arista(vertice v=vertice(), vertice w=vertice(), tCoste
           c=tCoste()): orig(v), dest(w), coste(c) {}
           // Orden de aristas para Prim y Kruskall
           bool operator <(const arista& a) const { return coste < a.coste; }
    };
    // resto de miembros de la clase GrafoP<T> ...
};

#include "particion.h"
#include "apo.h"

template <typename tCoste>
GrafoP<tCoste> Kruskall(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    typedef typename GrafoP<tCoste>::arista arista;
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;

    const size_t n = G.numVert();
    GrafoP<tCoste> g(n); // Árbol generador de coste mínimo.
    Particion P(n); // Partición inicial de los vértices de G.
    Apo<arista> A(n*(n-1)/2);// Aristas de G ordenadas por coste.

    // Copiar aristas del grafo G en el APO A.
    for (vertice u = 0; u <= n-2; u++)
           for (vertice v = u+1; v <= n-1; v++)
                  if (G[u][v] != INFINITO)
                         A.insertar(arista(u, v, G[u][v]));

    size_t i = 1;
    while (i <= n-1) { // Seleccionar n-1 aristas.
           arista a = A.cima(); // arista de menor coste
           A.suprimir();
           vertice u = P.encontrar(a.orig);
           vertice v = P.encontrar(a.dest);
           if (u != v) { // extremos de a pertenecen a distintos componentes
                  P.unir(u, v);
                  // Incluir la arista a en el árbol g.
                  g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste;
                  i++;
           }
    }
    return g;
}
```