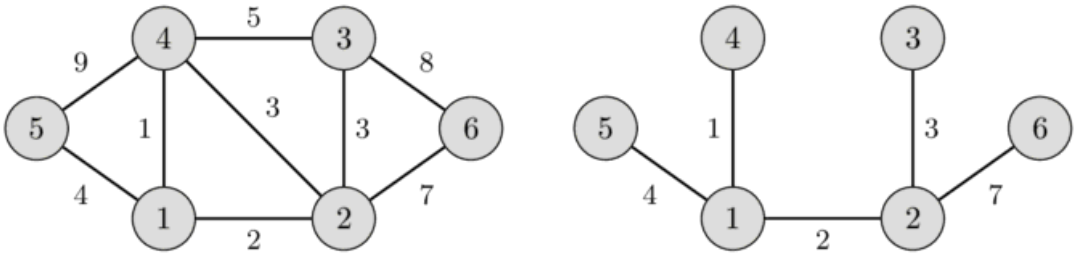
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO DE KRUSKAL

1. Introducción

Dado un grafo no dirigido ponderado. Queremos encontrar un subárbol de este grafo que conecte todos los vértices (es decir, es un árbol de expansión) y tiene el menor peso (es decir, la suma de los pesos de todos las aristas es mínima) de todos los árboles de expansión posibles. Este árbol de expansión se denomina árbol de expansión mínimo.

En la imagen de la izquierda puede ver un grafo no dirigido ponderado, y en la imagen de la derecha puede ver el árbol de expansión mínimo correspondiente.



2. Conocimientos previos

2.1. Árbol de expasión mínima

Dado un grafo conexo y no dirigido, un árbol recubridor, árbol de cobertura o árbol de expansión de ese grafo es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Cada arista tiene asignado un peso proporcional entre ellos, que es un número representativo de algún objeto, distancia, etc.; y se usa para asignar un peso total al árbol recubridor mínimo computando la suma de todos los pesos de las aristas del árbol en cuestión. Un árbol recubridor mínimo o un árbol de expansión mínimo es un árbol recubridor que pesa menos o igual que todos los otros árboles recubridores. Todo grafo tiene un bosque recubridor mínimo.

2.2. *Disjoint Set Union* o *DSU*

Esta estructura de datos proporciona las siguientes capacidades. Se nos dan varios elementos, cada uno de los cuales es un conjunto separado. Una DSU tendrá una operación para combinar dos conjuntos y podrá decir en qué conjunto se encuentra un elemento específico. La versión clásica también introduce una tercera operación, puede crear un conjunto a partir de un nuevo elemento.

Esta estructura de datos proporciona las siguientes capacidades. Se nos dan varios elementos, cada uno de los cuales es un conjunto separado. Una DSU tendrá una operación para combinar dos conjuntos y podrá decir en qué conjunto se encuentra un elemento específico. La versión clásica también introduce una tercera operación, puede crear un conjunto a partir de un nuevo elemento.

3. Desarrollo

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa). El algoritmo de Kruskal es un ejemplo de algoritmo voraz. Funciona de la siguiente manera

- Se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado.
- Se crea un conjunto C que contenga a todas las aristas del grafo.
- Mientras C es no vacío.
    * Eliminar una arista de peso mínimo de C.
    * Si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol.
    * En caso contrario, se desecha la arista

Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

Dada una lista de las aristas del grafo, el primer paso del algoritmo de Kruskal es ordenarlas por peso (usando un quicksort por ejemplo). Luego se van procesando las aristas en el orden de su peso, agregando aristas que no produzcan ciclos en el MST. El algoritmo de Kruskal es de O(A*$\log_2$ A), donde A es el número de aristas del grafo.

El siguiente ejemplo ilustra el funcionamiento del algoritmo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo.

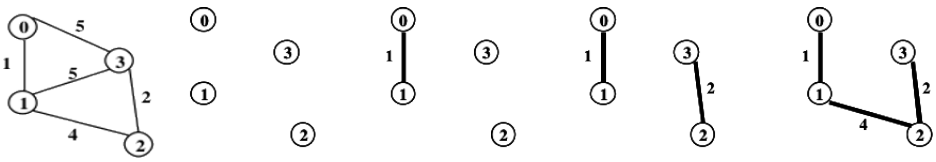


Figura 1: Ejecucción del algoritmo Kruskal.

Este algoritmo fue publicado por primera vez en Proceedings of the American Mathematical Society, pp. 48–50 en 1956, y fue escrito por Joseph Kruskal. A continuación se muestra un seudocódigo del algoritmo.

```
function Kruskal(G)
   Para cada v en V[G] hacer
      Nuevo conjunto C(v)\textleftarrow {v}.
   Nuevo heap Q que contiene todas las aristas de G, ordenando por su peso.
```

```
Defino un arbol T ={ vacio }
// n es el numero total de vertices
Mientras T tenga menos de n-1 vertices hacer
   (u,v) igual Q.sacarMin()
   // previene ciclos en T. agrega (u,v) si u y v estan diferentes
      componentes en el conjunto.
   // Notese que C(u) devuelve la componente a la que pertenece u.
   if C(v) distinto C(u) then
      Agregar arista (v,u) a  T.
      Merge C(v) y C(u) en el conjunto
Return arbol T
```

4. Implementación

4.1. C++

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#define MAX_N 100001
using namespace std;
typedef long long lld;

int n, m;
int numComponents, ret;

struct Edge{
   int a, b;
   int weight;
   bool operator <(const Edge &e) const{
      return (weight < e.weight);
   }
};
Edge E[MAX_N];

struct Node{
   int parent;
   int rank;
};

Node DSU[MAX_N];

inline void MakeSet(int x){
   DSU[x].parent = x; DSU[x].rank = 0;
}

inline int Find(int x){
   if (DSU[x].parent == x) return x;
```

```
   DSU[x].parent = Find(DSU[x].parent);
   return DSU[x].parent;
}

inline void Union(int x, int y){
   int xRoot = Find(x); int yRoot = Find(y);
   if (xRoot == yRoot) return;
   if (DSU[xRoot].rank < DSU[yRoot].rank){
      DSU[xRoot].parent = yRoot;
   }else if (DSU[xRoot].rank > DSU[yRoot].rank){
      DSU[yRoot].parent = xRoot;
   }else{
      DSU[yRoot].parent = xRoot; DSU[xRoot].rank++;
   }
}

inline int Kruskal(){
   int ret = 0, numComponents = n;
   for (int i=0;i<n;i++) MakeSet(i);
   sort(E, E+m);
   for (int i=0;i < m && numComponents > 1;i++){
      if (Find(E[i].a) != Find(E[i].b)){
         Union(E[i].a, E[i].b);
         ret += E[i].weight;
         numComponents--;
      }
   }
   if (numComponents > 1) return -1;
   return ret;
}

int main(){
   n = 7, m = 11;
   E[0].a = 0, E[0].b = 1, E[0].weight = 7;
   E[1].a = 0, E[1].b = 3, E[1].weight = 5;
   E[2].a = 1, E[2].b = 2, E[2].weight = 8;
   E[3].a = 1, E[3].b = 3, E[3].weight = 9;
   E[4].a = 1, E[4].b = 4, E[4].weight = 7;
   E[5].a = 2, E[5].b = 4, E[5].weight = 5;
   E[6].a = 3, E[6].b = 4, E[6].weight = 15;
   E[7].a = 3, E[7].b = 5, E[7].weight = 6;
   E[8].a = 4, E[8].b = 5, E[8].weight = 8;
   E[9].a = 4, E[9].b = 6, E[9].weight = 9;
   E[10].a = 5, E[10].b = 6, E[10].weight = 11;
   printf(" %d\n",Kruskal());
   return 0;
}
```

Esta implementación devuelve la longitud del árbol de expansión mínimo que genera el algo-

ritmo. En caso que no pueda generar el árbol de expasión mínima retorna -1.

4.2. Java

```
import java.io.*;
import java.math.*;
import java.util.*;
import java.lang.*;
import java.util.regex.*;

public class Main {
   private BufferedReader in;
   private PrintWriter out;
   private StringTokenizer st;

   public void solve() throws Exception {
      int nNodos=nextInt(),nEdges=nextInt(),A,B; long weigth;
      Graph graph =new Graph(nNodos);
      for(int i=0;i<nEdges;i++) {
         A = nextInt(); B = nextInt(); weigth = nextLong();
         graph.addEdge(new Edge(A, B, weigth));
      }
      out.printf("\ %d\n",graph.kruskal());
   }

   Main() throws Exception {
               in = new BufferedReader(new InputStreamReader(System.in));
               out = new PrintWriter(System.out);
               eat(""); solve(); in.close(); out.close();
   }

   private void eat(String str) {
      st = new StringTokenizer(str);
   }

   private String next() throws Exception {
      while (!st.hasMoreTokens()) {
         String line = in.readLine();
         if (line == null) return null;
         eat(line);
      }
      return st.nextToken();
   }

   private int nextInt() throws Exception {
      return Integer.parseInt(next());
   }

   private long nextLong() throws Exception {
```

```
    return Long.parseLong(next());
}

private double nextDouble() throws Exception {
    return Double.parseDouble(next());
}

public static void main(String[] args) throws Exception { new Main();}

private class Graph{
    private List<Edge> edges;
    private int [] DSU ;
    private int nodes;

    public Graph(int _nodes) {
         edges = new ArrayList<Edge>();
         DSU = new int [_nodes+10];
         for (int i = 1; i <= _nodes; i++) DSU[i] = i;
            this.nodes =_nodes;
    }
    public void addEdge(Edge e) { edges.add(e); }
    public int root(int x) {
                      return x == DSU[x] ? x : (DSU[x] = root(DSU[x]));
    }
    public void unite(int a, int b) {
        a = root(a); b = root(b);
        if (a != b) DSU[a] = b;
    }
    public long kruskal() {
        Collections.sort(edges);
        int nedges = edges.size();
        int ncomponents = this.nodes;
        long answer = edges.get(0).cost;
                for(int i=0;i<nedges && ncomponents > 1;i++){
           Edge e = this.edges.get(i);
           if(this.root(e.nodeA)!=this.root(e.nodeB)) {
              this.unite(e.nodeA, e.nodeB); answer+=e.cost;
           }
        }
        if(ncomponents > 1) return -1;
        return answer;
    }
}
private class Edge implements Comparable<Edge>{
    public int nodeA;
    public int nodeB;
    public long cost;

    public Edge(int _a,int _b, long _cost) {
        this.nodeA=_a; this.nodeB=_b; this.cost=_cost;
```

```
        }
        @Override
        public int compareTo(Edge e1) {
            if(this.cost < e1.cost) return -1;
            else if(this.cost > e1.cost) return 1;
            else return 0;
        }
    }
}
```

5. Complejidad

Para determinar la complejidad del algoritmo vamos a definir m como el número de aristas mientras el número de nodos o vértices será n. El algoritmo de Kruskal muestra una complejidad O(m log m) o, equivalentemente, O(m log n), cuando se ejecuta sobre estructuras de datos simples. Los tiempos de ejecución son equivalentes porque:

- m es a lo sumo n^2 y $\log n^2 = 2 \log n$ es O($\log n$)
- ignorando los vértices aislados, los cuales forman su propia componente del árbol de expansión mínimo, n $\leq$ 2m, así que log n es O(log m).

Se puede conseguir esta complejidad de la siguiente manera: primero se ordenan las aristas por su peso usando una ordenación por comparación (comparison sort) con una complejidad del orden de O(m log m); esto permite que el paso .[el]iminar una arista de peso mínimo de C"se ejecute en tiempo constante. Lo siguiente es usar una estructura de datos sobre conjuntos disjuntos (disjoint-set data structure) para controlar qué vértices están en qué componentes. Es necesario hacer orden de O(m) operaciones ya que por cada arista hay dos operaciones de búsqueda y posiblemente una unión de conjuntos. Incluso una estructura de datos sobre conjuntos disjuntos simple con uniones por rangos puede ejecutar las operaciones mencionadas en O(m log n). Por tanto, la complejidad total es del orden de O(m log m) = O(m log n)

6. Aplicaciones

Como es evidente el algoritmo no permite el hallar árbol de expansión mínima de un grafo si este fuera inicalmente conexo sino hallaríamos el árbol de expasión mínima de cada componente conexa de que se compone el grafo.

El algoritmo de Kruskal también ha sido aplicado para hallar soluciones en diversas áreas como es el diseño de redes de transporte, telecomunicaciones, TV por cable, sistemas distribuidos, interpretación de datos climatológicos, visión artificial, entre otros.

7. Ejercicios propuestos

La siguiente lista de ejercicios de pueden resolver aplicando el algoritmo de Kruskal original o con alguna moficación:

- DMOJ - Irrigando los Campos
- DMOJ -Sin Heno
- DMOJ - Los corredores de MegaBit
- DMOJ - Conectando Islas