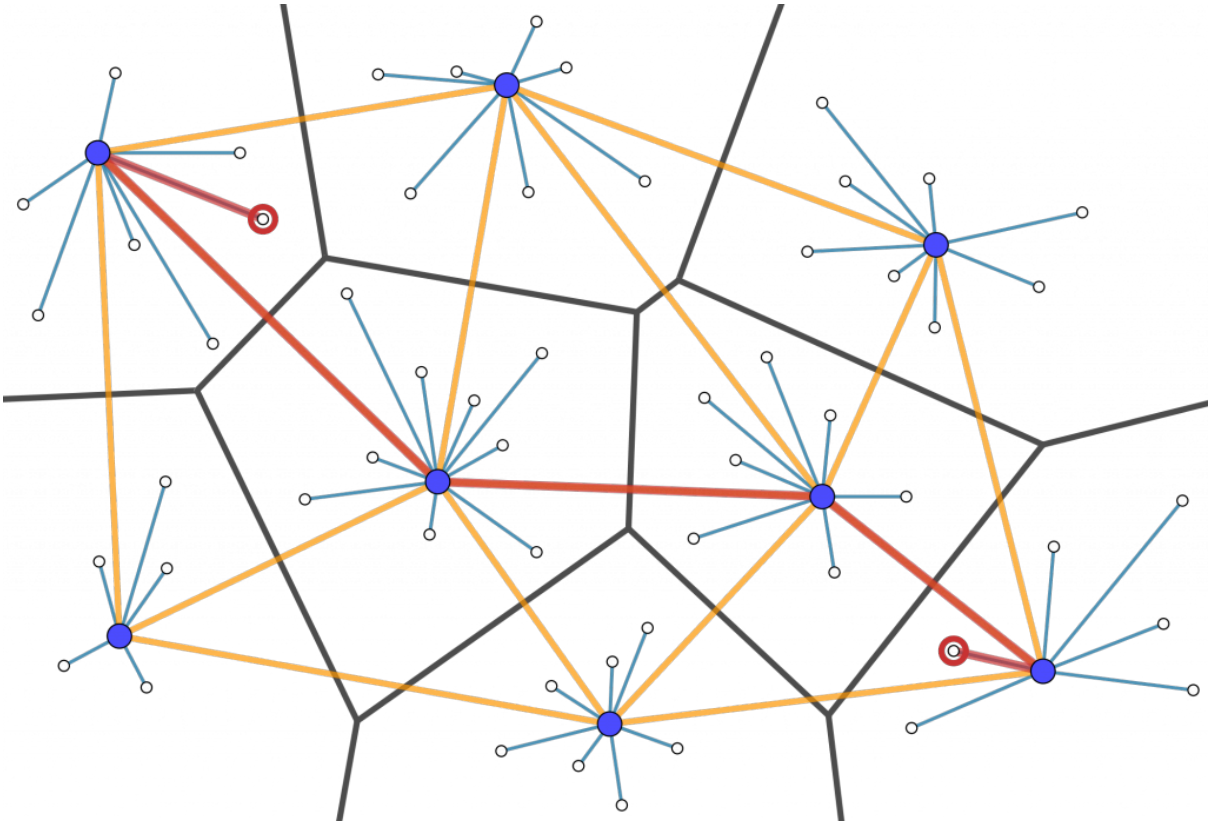
Trabajo practico obligatorio grafos



Materia: Diseño y análisis de algoritmos
Profesor: Rodriguez Guillermo
Alumnos: Agustín Luporini, Mateo Fontaine, Agustín Gimenez, Jonatan Tarragona
Trabajo practico: Flujo de vehículos

UADE
COSTA ARGENTINA

Enunciado

Las emisiones de los automóviles privados son una de las principales fuentes de contaminación en las grandes ciudades. Las autoridades universitarias han decidido contribuir seriamente a la mejora de la calidad del aire prohibiendo el paso de automóviles por el campus y creando una serie de líneas de autobuses eléctricos que permitan acceder a distintos puntos de la Ciudad Universitaria.

Para ello se han estudiado los principales flujos de vehículos entre distintos puntos seleccionados (facultades, estación de metro, etc.) de la Ciudad Universitaria. Los resultados del estudio contienen el número de vehículos que transitan al día entre estos puntos seleccionados (sin tener en cuenta el sentido en el que circulan).

El objetivo de este estudio es localizar los trayectos entre puntos seleccionados que utilizan más vehículos y que permiten conectar todos los puntos seleccionados. Diseñar un algoritmo que proporcione estos trayectos. Validar que el algoritmo sea óptimo en base a algoritmos conocidos. Calcular la complejidad temporal.

Entregables

El entregable consiste en la entrega del link del repositorio GitHub con el proyecto y un informe describiendo cada una de las etapas del proyecto (diseño, justificación, pseudocódigo, cálculo de la complejidad temporal).

Diseño

Para solucionar la problemática planteada, diseñamos un algoritmo cuyo objetivo sea construir un árbol de expansión máxima (MST) de un grafo ponderado no dirigido. Lograremos esto seleccionando las aristas de manera que conecten todos los vértices del grafo sin formar ciclos, maximizando la suma de los pesos de las aristas incluidas.

Para lograr esto, usaremos una clase llamada “Kruskal”, en la que se aplicara una versión de dicho algoritmo. Esta clase contara con el método que resolverá el problema, el cual puede dividirse en tres partes mayores:

1) Inicialización y ordenamiento

-Comienza generando una lista de todas las aristas del grafo
-Se ordenan las aristas en orden descendente según sus pesos

2) Union-Find para representar conjuntos

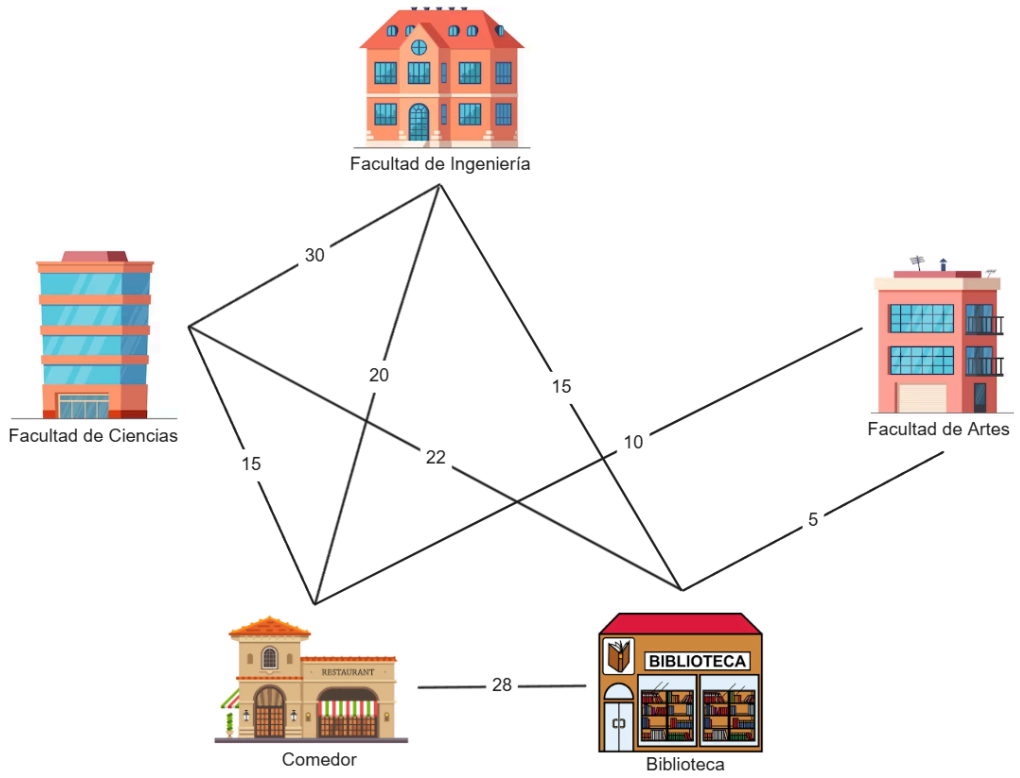
-Para evitar generar ciclos en el algoritmo Kruskal es necesario tener una representación de conjuntos. Para esto usaremos “Union-Find”, una estructura comunmente usada en este tipo de implementación para representar conjuntos de una manera óptima
-Inicialmente, cada vértice pertenece a su propio conjunto, lo que representa que cada vértice es un conjunto independiente

3) Construcción del Árbol de Expansión Máxima

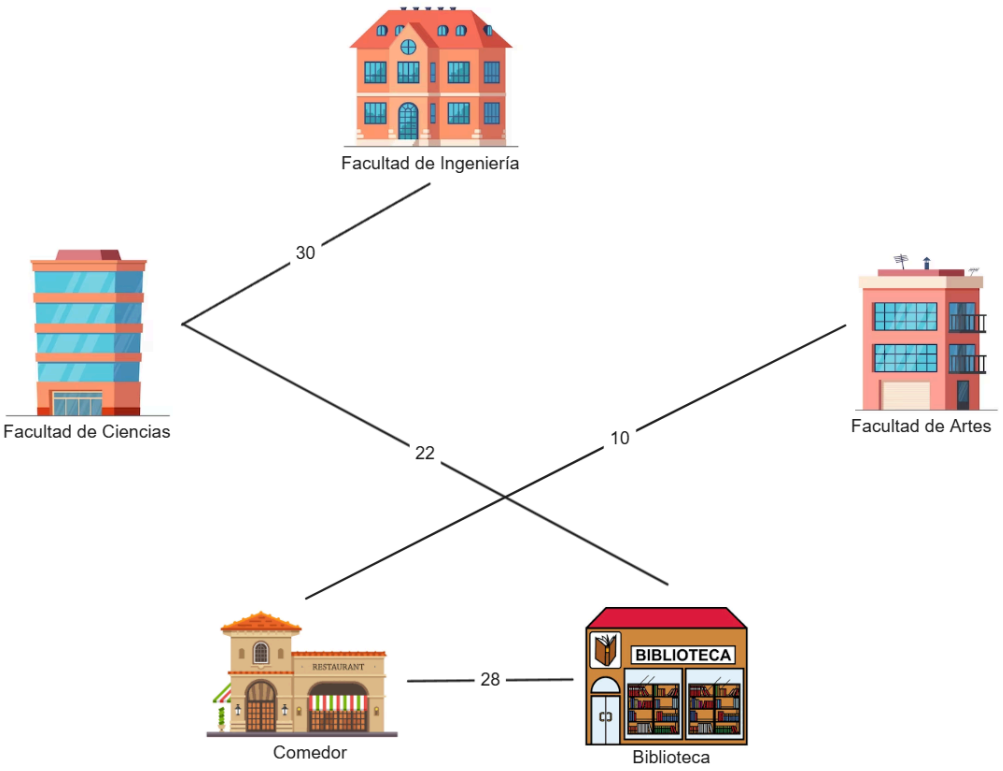
-Se iteran las aristas (en orden descendente de peso)
-Para cada arista, se verifica si los vértices de origen y destino pertenecen al mismo conjunto. Si no es así, significa que añadir esta arista no creará un ciclo, por lo que se realiza la unión de los conjuntos de origen y destino en el Union-Find
-El proceso continúa hasta que se ha formado un grafo que conecta todos los vértices (lo que es igual a un único conjunto)

Ejemplo

Grafo inicial:



Árbol de máxima expansión:



Justificación

La elección del algoritmo de Kruskal para construir un Árbol de Expansión Máxima (MST) se basa en su eficiencia y simplicidad. Kruskal es particularmente adecuado para grafos dispersos, ya que se centra en seleccionar aristas de mayor peso, lo que permite maximizar la suma total de los pesos del MST.

Acompañando la suposición de que se trabajara con un grafo disperso, consideramos adecuado el uso de listas de adyacencia en lugar de matrices de adyacencia. En este tipo de grafos, las listas son más eficientes en términos de espacio, ya que permiten almacenar únicamente las conexiones necesarias entre los vértices.

Para la representación de conjuntos decidimos utilizar la estructura de datos Union-Find, que es esencial para garantizar que el algoritmo funcione de manera óptima y sin ciclos. Esta estructura permite agrupar y gestionar los conjuntos de vértices de forma eficiente. La capacidad de realizar uniones y encontrar representantes de conjuntos en tiempo casi constante es crucial para el rendimiento del algoritmo.

En resumen, la combinación del algoritmo de Kruskal, el uso de listas de adyacencia y la estructura Union-Find proporciona un enfoque eficiente para resolver el problema de construir un árbol de expansión máxima en un grafo ponderado no dirigido. Esta metodología no solo optimiza el uso de recursos, sino que también simplifica la implementación y el mantenimiento del algoritmo.

Pseudocódigo

```
function encontrarMST(grafo):
    // Parte 1: Inicialización y ordenamiento
    aristas;                    //obtener todas las aristas de grafo
    aristas.sort(reverse);  //ordenar aristas en orden ascendente por peso

     // Parte 2: Inicialización del Union-Find
    unionFind;                //conjunto disjunto para todos los vértices del grafo

    // Parte 3: Construcción del MST
     mst;                        //lista vacía para almacenar las aristas del MST

    for arista in aristas:
            //salir cuando haya un solo conjunto
            if(mst.size() == grafo.cantVertices() - 1){
            continue
            }

            arista.getOrigen                  //obtener vértice de origen de la arista
            arista.getDestino               //obtener vértice de destino de la arista

            if (!mismoCojunto(origen, destino){
            mst.add(arista)                  //agregar arista al mst
                    unir(origen, destino)            //unir conjuntos
            }
    return mst
```

Análisis de complejidad temporal

a = Arista
v = Vértices

```
public List<Arista> encontrarMST(Grafo grafo) {
List<Arista> aristas = grafo.obtenerAristas();                    → TBloque1: O (v + a)
Collections.sort(aristas);                                        → TBloque2: O (a log a)

UnionFind unionFind = new UnionFind(grafo.getListaAdyacencia().keySet());  → TBloque3: O (v)

List<Arista> mst = new ArrayList<>();                             → TBloque4: O (v)

for (Arista arista : aristas) {                                   → TBloque5: O (a)
    if (mst.size() == grafo.cantVertices() - 1){
        break;
    }
    Vertice origen = arista.getOrigen();
    Vertice destino = arista.getDestino();

    if (!unionFind.mismoConjunto(origen, destino)) {
        mst.add(arista);
        unionFind.unir(origen, destino);
    }
}
 return mst;
}
```

TEncontrrMST = TBloque1 + TBloque2 + TBloque3 +TBloque4 + TBloque5
TEncontrrMST = O(v + a) + O(a log a) + O(v) + O(v) + O(a) = **O(a log a)**

TEncontrrMST = O(a log a).
Como suele ocurrir en algoritmos de la técnica Greedy y en el algoritmo de Kruskal, el costo temporal corresponde a O(a log a), que es el costo del ordenamiento de las aristas.

Cálculos auxiliares:

TBloque1

```
public List<Arista> obtenerAristas() {
      Set<Arista> aristasSet = new HashSet<>();                   //O(1)
      for (List<Arista> lista : listaAdyacencia.values()) {       →TBloque1
         for (Arista arista : lista) {
            aristasSet.add(arista);
         }
      }
      return new ArrayList<>(aristasSet);
   }
```

T(TBloque1) = Tcond + (Tcond + Ts1) * iteraciones
T(TBloque1) = 1 + (1 + Ts1 + 1) * v = 1 + (2 + 1 + 3s) * v = 1 + 3v + 3av = **O(v + a)**
Ts1 = 1 + (1 + Ts2) * a = 1 + (1 + 1 + 1) * a = 1 + 3a = O(a)
Ts2 = O(1)

Aclaración TBloque1:
*En este método no se multiplica el costo del bucle externo por el costo del bucle interno (como se suele hacer en FOR anidados) ya que si bien el primer FOR itera V veces (es decir por todos los vértices), el FOR interno recorre en total A veces, es decir que no recorre todas las aristas en cada iteración, sino que recorre todas las aristas una sola vez en lo que termina el FOR externo. Es por esto que en lugar de tener un costo de O(V * A), termina teniendo un costo de O(V + A), que es el costo de recorrer ambas estructuras*

TBloque2
El costo del ordenamiento *Collections.sort()* en java es de $O(n \log n)$

TBloque3
UnionFind unionFind = new UnionFind(grafo.getListaAdyacencia().keySet());

TBloque5

```
for (Arista arista : aristas) {                          →TBloque1
    if (mst.size() == grafo.cantVertices() - 1){         →TBloque2
        break;
    }
    Vertice origen = arista.getOrigen();                 //O(1)
    Vertice destino = arista.getDestino();               //O(1)

    if (!unionFind.mismoConjunto(origen, destino)) {     →TBloque3
        mst.add(arista);
        unionFind.unir(origen, destino);
    }
}
 return mst;
}
```

TBloque1 = Tcond + (Tcond + Ts1) * iteraciones = ***O(a)***
TBloque1 = 1 + (1 + Ts1) * a = 1 + (1 + 1) * a = 1 + 2a = O(a)

Ts1= TBloque1 + 1 + 1 + TBloque3 = 1 + 1 + 1 + 1 = O(1)

TBloque2 = O(1)

TBloque3 = O(1). *Teniendo presente la implementación de UnionFind, el costo de "unir" se considera constante ya que su crecimiento es muy lento para cualquier número razonable de vértices.*