[Fecha]

Practica 02

EDA2

HAMZA EL FALLAH
NADA DIOUANE

Contenido

1. Objetivo de la practica

Utilizar el método algorítmico greedy para solucionar el problema. Hacer la comparación de los algoritmos implementados:

- Cualitativa
- Cuantitativamente

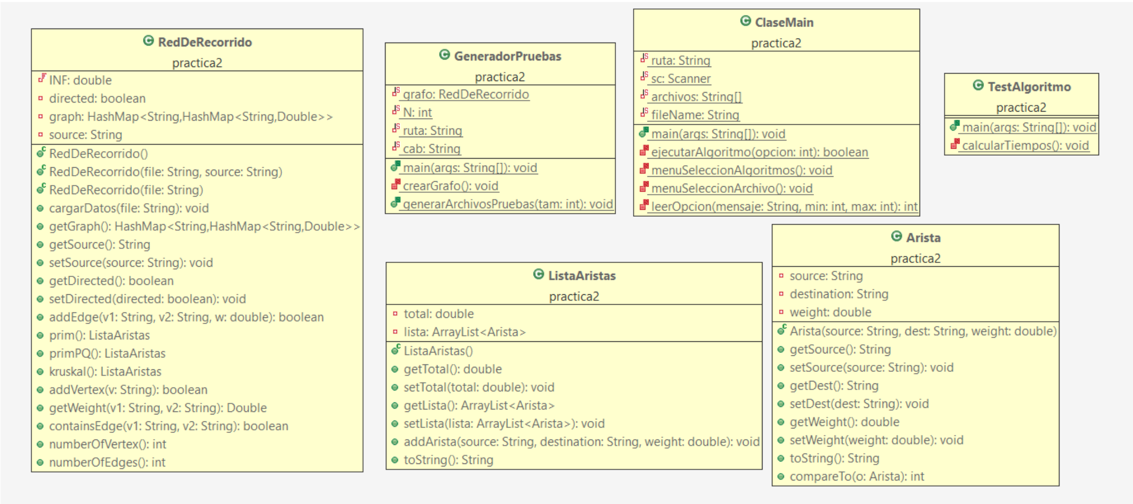
Realizar una comparación desde el punto de vista teórico y práctico y un análisis de la eficiencia de las soluciones aportadas

2. Implementación

Para la realización de esta práctica contamos con una red de rutas y proponemos implementar dos soluciones utilizando el esquema voraz.

Planteamos el problema en forma de gráfico donde los vértices son ciudades y las aristas son las líneas que las unen y forman una red de caminos.

3. Diagrama de clases



4. Clase Main

Tiene dos menús para mostrar resultados en la consola, uno para seleccionar archivos y otro para seleccionar algoritmos. Se ejecutan archivos que contienen gráficos y otros datos de lista que operan con los tres algoritmos.

5. Clase Aristas

En la clase clase Aristas encontramos un vértice de destino y otro de origen junto a un peso asociado a la arista que los conectan.

El método compareTo hace la comparación según el peso de la arista, en caso de tener el mismo peso.

```
@Override
public int compareTo(Arista o) {
    int comp = Double.compare(this.weight, o.weight);

    return comp ==0 ? (this.source+"-"+this.destination).compareTo(o.source+"-"+o.destination) : comp;

}
```

6. Clase ListaAristas

Método addArista

Acepta dos parámetros de cadena que se refieren a los vértices inicial y final y el peso asociado con el borde. Cree un objeto de borde, cree una instancia y agréguelo a conjAristas, que es un ArrayList de tipo Arista. El total se actualiza sumando los pesos de los bordes agregados a la estructura.

```
public void addArista(String source, String destination, double weight){
    this.total += weight;
    lista.add(new Arista(source, destination, weight));
}
```

7. Clase RedDeCarretera

Método cargarDatos

Recibe la ruta completa del archivo y verifica si es un gráfico dirigido y lee la cantidad de vértices, siempre que la línea sea un vértice, la coloca en la estructura como una clave llamando al método addVertex. Una vez que termina de leer los vértices, lee el número de aristas y agrega las aristas a sus pesos asociados mediante el método addEdge. Si no se establece ningún vértice de origen, se elige uno al azar.

```java
    public void cargarDatos(String file) {
        Scanner sc = null;
        String line = "";
        String[] tokens = null;
        try {
            sc = new Scanner(new File(file));
        }catch (IOException e) {
            System.out.println(e.getMessage());
        }
        int contV = 0;
        int contEd = 0;
        String cab = sc.nextLine();
        if(cab.equals("0")) {
            this.directed = false;
        }
        if(cab.equals("1")){
            this.directed = true;
        }
        int numVertex = Integer.parseInt(sc.nextLine());
        int numEdge = 0;
        while (sc.hasNextLine()){
            line = sc.nextLine().trim();
            if (line.isEmpty()) continue;
            if(contV++ < numVertex && line.length() == 1) {
                this.addVertex(line);
            }if(contV == numVertex) {
                numEdge = Integer.parseInt(sc.nextLine());
                contEd++;
                continue;
            }
            if(contEd != 0) {
                if(contEd++ <= numEdge) {
                    tokens = line.split(" ");
                    this.addEdge(tokens[0], tokens[1], Double.parseDouble(tokens[2]));
                }
            }
        }
        sc.close();
        if(this.source == null) {
            ArrayList<String> aux = new ArrayList<String>(this.graph.keySet());
            Collections.shuffle(aux);
            this.source = aux.get(0);
        }
```

Método prim

El algoritmo de Prim consiste en encontrar el árbol generador mínimo del grafo. Este árbol es el árbol donde todos los vértices del gráfico están conectados y la unión es mínima, es decir, cuando se suman todos los costos de un vértice a otro, obtenemos el coste mínimo.

Aplica iterativamente las propiedades del árbol de expansión de costo mínimo, agregando un borde en cada paso. Use un conjunto de vértices procesados U y, en cada paso, elija el borde más pequeño que conecta un vértice de U con su otro complemento.

Ajustar las propiedades del árbol consta de dos pasos, recuerda que no puedes crear bucles. Los pasos son, elija un vértice inicial y vaya a seleccionar vértices adyacentes y tome el borde con el menor peso, hasta que se visiten todos los vértices.

Implementación en código

El método prim primero verifica que el vértice de origen existe. A continuación, se crean cuatro estructuras, dos HashMaps. El primero (peso) tiene como clave el vértice y como valor su peso asociado. La segunda (rama) tiene el vértice de destino adyacente como su clave y el vértice de origen como su valor. También tenemos un HashSet (restante). También tenemos un EdgeSet (resultado) que almacena los bordes del árbol de expansión mínimo del vértice fuente seleccionado (fuente). Establecemos un String desde inicializado a "nulo" para el vértice "desde". El primer ciclo "for" itera sobre un conjunto de vértices (claves) de la estructura del gráfico y los agrega a la estructura restante. Luego elimina el vértice de origen.

El segundo ciclo "for" itera sobre los vértices agregados previamente a la estructura restante y obtiene los pesos relativos de los bordes entre el vértice de origen y sus vecinos. Si son adyacentes, el peso no está vacío, en una estructura ramificada, agrega vértices adyacentes a la fuente (aristas entre vértices y sus vecinos). Agregue vértices adyacentes y pesos asociados con el origen a Pesos. Si no son adyacentes, agregue vértices a la rama estableciendo vértices en nulo. En pesos establece los pesos al infinito (INF). Finalmente, cuando el vértice de destino es el mismo que el vértice de origen, el peso se establece en cero.

A reanudación pasamos al centro primordial del notación que consiste en un onda while recorre remain mientras que no está vacía Está composición en su interno por dos bucles “for” El primeramente recorre remain obteniendo el mínimo valor desde la estructura weights y el punto medianero agregado Si devuelve el punto de mínimo valor se elimina mentado punto de la estructura remain Se obtiene el punto de procedencia desde branch y se añade en la estructura result la saliente dentro el punto de partido (aux) y de destino (from) y su valor agregado El siguiente onda for recorre la estructura remain obteniendo el valor de la saliente asociada a los vértices (from yto) si el valor es mínimo lo actualiza en la estructura weights y actualiza la saliente en la estructura branches Por final devuelve el conglomerado de aristas del arbusto de revestimiento mínimo.

```java
    public ListaAristas prim(){
        if(source == null || !this.graph.containsKey(source)) return null;

        HashMap<String, Double> weights = new HashMap<String, Double>();
        HashMap<String, String> branches = new HashMap<String, String>();
        HashSet<String> remain = new HashSet<String>();
        ListaAristas result = new ListaAristas();
        String from = null;

        for (String v : this.graph.keySet()) {
            remain.add(v);
        }
        remain.remove(source);

        for (String v : remain) {
            Double weight = getWeight(source, v);
            if(weight != null) {
                branches.put(v, source);
                weights.put(v, weight);
            }else {
                branches.put(v, null);
                weights.put(v, INF);
            }

        }
        branches.put(source, source);
        weights.put(source, 0.0);

        while(!remain.isEmpty()) {

            double min = INF;
            from = null;
            for (String v : remain) {
                double weight = weights.get(v);
                if(weight < min) {
                    min = weight;
                    from = v;
                }
            }

            if(from == null) break;
            remain.remove(from);
            String aux = branches.get(from);
            result.addArista(aux, from, getWeight(aux, from));

            for (String to : remain) {
                Double weight = getWeight(from, to);
                if(weight != null && weight < weights.get(to)) {
                    weights.put(to, weight);
                    branches.put(to, from);
                }
            }
        }

```

Método primPQ con cola de prioridad

Implementación en código

primPQ verifica que exista el vértice de origen. Después hace la creación tres estructuras.

Primero es un HashSet tipo string (**remain**).

Segundo es (**pq**) tipo Arista se trata de una cola de prioridad.

ListaAristas (**result**) donde se almacenan las aristas del árbol de recubrimiento mínimo a partir del vértice de origen elegido (source).

```java
    public ListaAristas primPQ(){
        String source = this.source;
        if(source == null || !this.graph.containsKey(source)) return null;

        HashSet<String> remain = new HashSet<String>();
        PriorityQueue<Arista> pq = new PriorityQueue<Arista>();
        ListaAristas result = new ListaAristas();
        String from = source;
        String to;
        Arista aux;
        Double weight;

        for (String v : this.graph.keySet()) {
            remain.add(v);
        }
        remain.remove(source);

        while(!remain.isEmpty()) {
            for (Entry<String, Double> it : this.graph.get(from).entrySet()) {
                to = it.getKey();
                weight = it.getValue();
                if(remain.contains(to)) {
                    aux = new Arista(from, to, weight);
                    pq.add(aux);
                }
            }
            do {
                aux = pq.poll();
                from = aux.getSource();
                to = aux.getDest();
                weight = aux.getWeight();
            } while(!remain.contains(to));
            remain.remove(to);
            result.addArista(aux.getSource(), aux.getDest(), aux.getWeight());
            from = to;
        }
        return result;
    }
```

Método kruskal

Implementación en código

El método coloca el vértice como punto de partida y comprueba que no está vacío y que existe

Se declaran tres sistemas de árboles, uno para la rama y otro para los vértices adyacentes y el peso asociado. El tercer sistema compuesto almacena el conjunto de aristas de la solución.

El método está compuesto tres bucles principales:

-Primero bucle pasa por todos los vértices del diagrama de estructura general y manténgalos en forma fijando la primera carga a infinito.

-Se declara una variable booleana para identificar el punto de partida o el vértice de origen

-El segundo bucle recorre remain mientras que no esté vacía, saca de la estructura el primer vértice y está compuesto a su vez por dos bucles for, donde el primero recorre el conjunto de vértices y sus pesos de remain para encontrar la arista con menor peso y después se elimina la arista de la estructura (vértice y peso asociado).

-El segundo bucle interno recorre los vértices adyacentes del vértice de la arista de menor coste del grafo general (graph) y obtiene el peso de la arista que une dicho vértice con cada uno de sus adyacentes si no lo son establece el peso a infinito actualiza en remain el vértice con menor peso y en branches la arista con menor peso.

-EL tercer bucle principal recorre el conjunto de aristas en branches ordenadas por el menor peso y las añade en result junto con sus pesos correspondientes.

-Por último, devuelve el resultado del árbol de recubrimiento mínimo.

```java
    public ListaAristas kruskal(){
        String source = this.source;
        if(source == null || !this.graph.containsKey(source)) return null;

        TreeMap<String, String> branches = new TreeMap<String, String>();
        TreeMap<String, Double> remain = new TreeMap();
        ListaAristas result = new ListaAristas();

        for (String v : graph.keySet()) {
            remain.put(v, INF);
        }
        boolean isFirst = true;
        while(!remain.isEmpty()) {
            String minKey = remain.firstKey();
            if(isFirst) {
                minKey = source;
                isFirst = false;
            }
            Double minValue = INF;
            for (Entry<String, Double> e : remain.entrySet()) {
                if(e.getValue() < minValue) {
                    minValue = e.getValue();
                    minKey = e.getKey();
                }
            }
            remain.remove(minKey);

            for (Entry<String, Double> it : graph.get(minKey).entrySet()) {
                String to = it.getKey();
                Double dActual = getWeight(branches.get(to), to);
                dActual = dActual == null ? INF : dActual;

                if(remain.containsKey(to) && it.getValue() < INF && it.getValue() < dActual) {
                    remain.put(to, it.getValue());
                    branches.put(to, minKey);
                }
            }
        }

        for (Entry<String, String> it : branches.entrySet()) {
            String to = it.getKey();
            String from = it.getValue();
            result.addArista(from, to, getWeight(from, to));

        }
        return result;
    }
```

8. Estudio teórico

Para un grafo con n nodos y x aristas:

- Prim: O(n^2)
- Kruskal O(x log n)

Para un grafo denso se cumple que x → n (n -1)/2

- Prim puede ser mejor depende del valor de las constantes ocultas
- Kruskal: O(n^2 log n)

Para un grafo disperso se cumpla que: x→ n

- Prim es menos eficiente
- Kruskal: O(n log n)

Para una implementación simple de Prim, utilizando una matriz de adyacencia o una representación de lista de adyacencia de un grafo, haciendo una búsqueda lineal de la matriz de peso para encontrar el borde ponderado mínimo para agregar, requiere el tiempo de ejecución de $O(n^2)$.

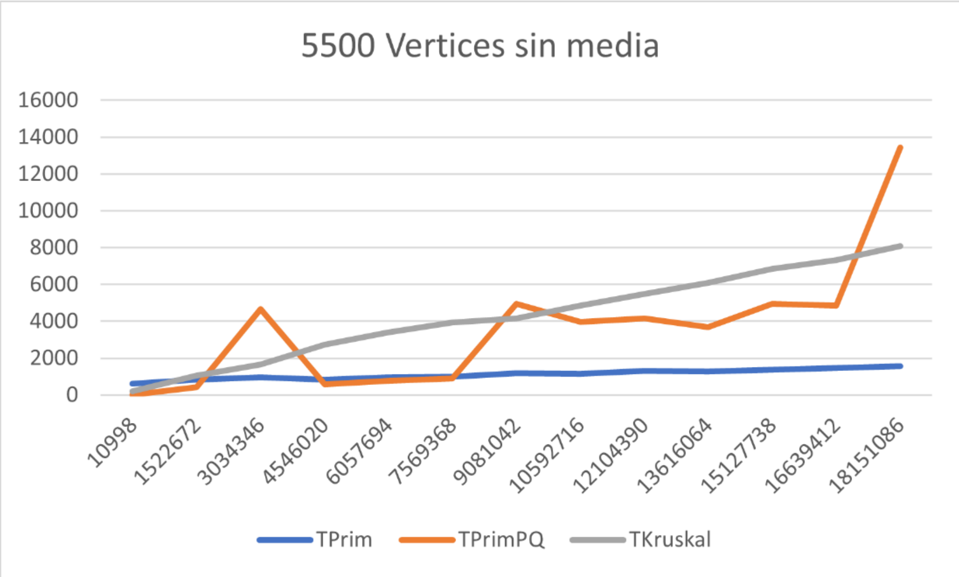
Sin embargo, este tiempo de ejecución se puede mejorar aún más utilizando la pila para realizar la búsqueda de los bordes menos probables en el bucle interno del algoritmo.

En nuestro caso, para los algoritmos Prim y PrimPQ, tenemos un orden de $O(n^2)$. Esto debido a que cada uno usa tres bucles de orden n y uno de ellos, el bucle while, tiene otro bucle dentro que también será de orden n, siendo un bucle anidado, que tendrá el orden $O(n^2)$.

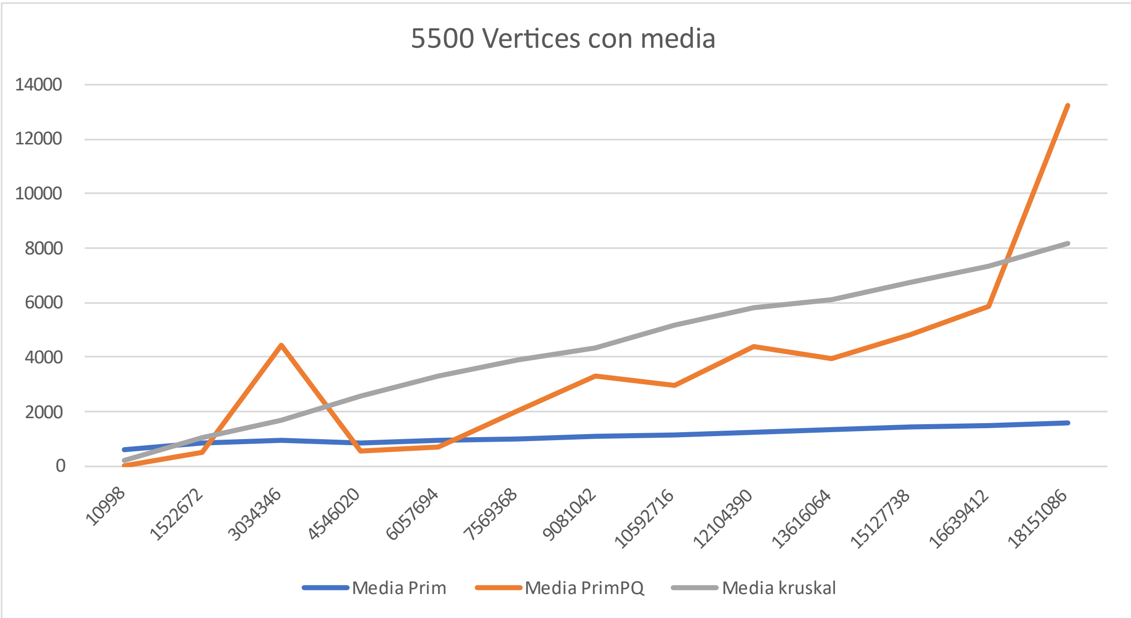
9. Estudio empírico

Se muestran picos en el gráfico, cuando se hace la ejecución de nuestro código, pero no afectan la representación de los resultados. Sin embargo, el gráfico final se genera con un promedio de 4 ejecuciones para obtener los mejores resultados.

- **Nv=5500** Sin media



- **Nv=5500** Con media



Lo que podemos notar es con un numero de aristas pequeño podemos observar una ejecución mejor del algoritmo Prim utilizando cola de prioridad, pero a partir de las 10000000 aristas observamos cómo pasa a ser mejor el algoritmo de Prim sin el uso de cola de prioridad.

para valores pequeños el algoritmo de kruskal es mejor que el Prim sin cola de prioridad pero no puede hacer la representación para valores más grandes porque no se ha podido llevar a cabo la ejecución en el ordenador.