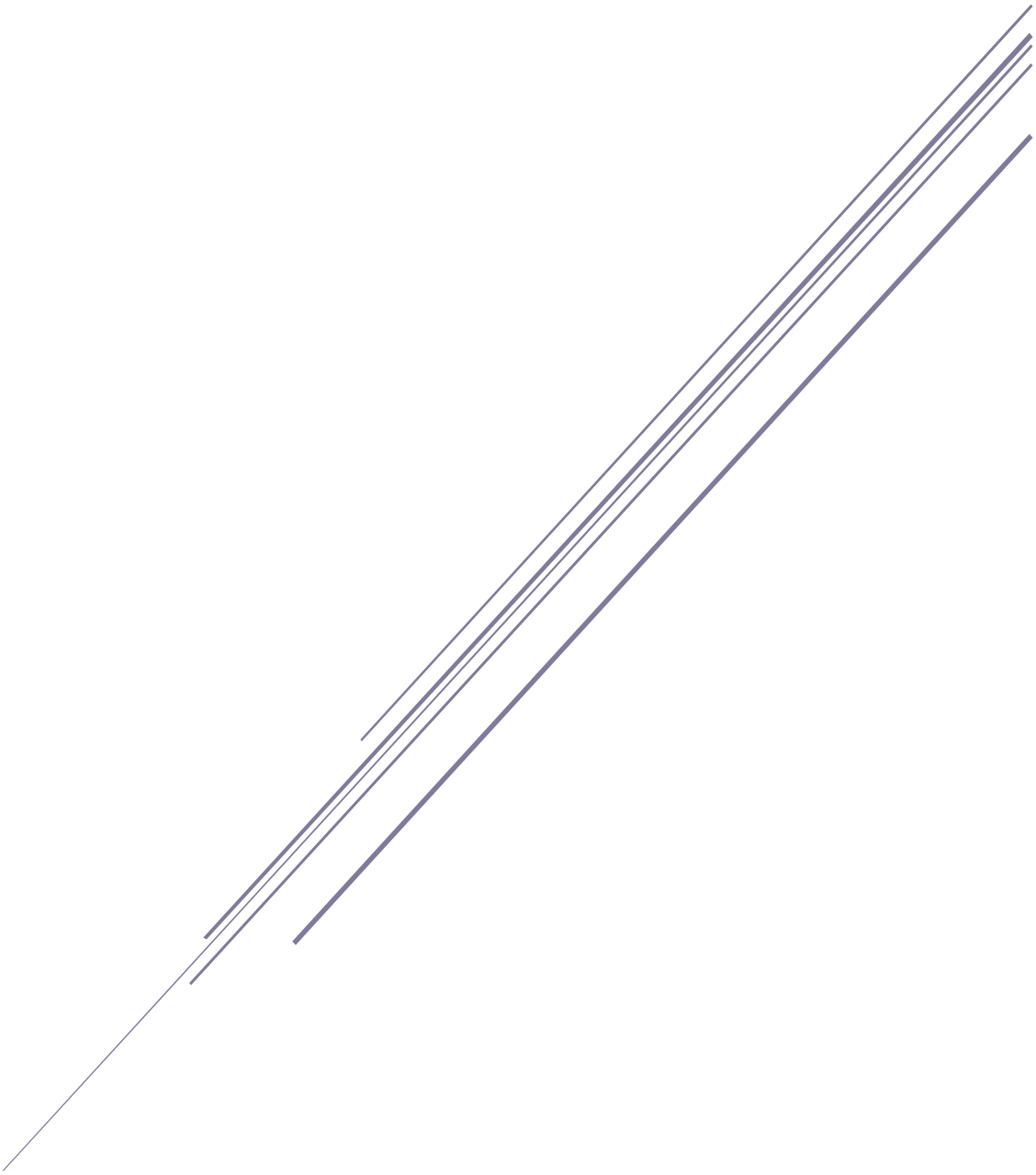
PRÁCTICA 2

EDA2



Universidad de Almería

Alejandro Rodríguez Moreno, Ángel López Capel y José Luis Pérez Lopez

1 ESTUDIO DE LA IMPLEMENTACIÓN

1.1 GRAFO

El grafo lo hemos definido como un HashMap que guarda el nombre de el nodo inicio y el nombre del nodo destino y peso en otro HashMap anidado.

1.2 ALGORITMO DE PRIM

Este algoritmo trata de conseguir el árbol de recubrimiento mínimo del grafo que le pasamos como entrada. El algoritmo empieza con una estructura de datos con un solo nodo escogido aleatoriamente de entre todos los del grafo y selecciona el nodo vecino cuya arista tenga el peso más bajo, y lo añade a la estructura de datos hasta que todos los nodos del árbol estén presentes.

```
public static HashMap<String, ArrayList<String>> prim(Grafo grafo) {

        HashSet<String> remain = new HashSet<String>();

        for (String v : grafo.map.keySet())
            remain.add(v);
        remain.remove(grafo.getOrigen());

        heap.clear();
        weight.clear();
        for (String v : remain) {

            Double w = grafo.getWeight(grafo.getOrigen(), v);
            if (w != null) {
                heap.put(v, grafo.getOrigen());
                weight.put(v, w);

            }

            else {
                heap.put(v, null);
                weight.put(v, Double.MAX_VALUE);
            }
        }

        heap.put(grafo.getOrigen(), grafo.getOrigen());

        weight.put(grafo.getOrigen(), 0.0);

        HashMap<String, ArrayList<String>> resultado = new HashMap<String,
ArrayList<String>>();
        while (!remain.isEmpty()) {
```

```
            double minValue = Double.MAX_VALUE;
            String minKey = "";
            for (String v : remain) {
                double w = weight.get(v);
                if (w < minValue) {
                    minValue = w;
                    minKey = v;
                }
            }

            if (minKey.isEmpty())
                break;

            remain.remove(minKey);

            String origen = heap.get(minKey);
            if (!resultado.containsKey(origen)) {
                resultado.put(origen, new ArrayList<String>());
            }
            resultado.get(origen).add(minKey);

            for (String v : remain) {
                Double w = grafo.getWeight(minKey, v);
                if (w != null && w < weight.get(v)) {
                    weight.put(v, w);
                    heap.put(v, minKey);
                }
            }
        }
        return resultado;
    }
```

1.2.1 Variación con Cola de Prioridad

Esta variación tiene la misma funcionalidad que la versión sin cola de prioridad. La única diferencia es que la cola de prioridad es un *least-first-out*. Esto quiere decir que el primer elemento en salir será el de menor valor, algo muy útil para este algoritmo.

```
public static HashMap<String, ArrayList<String>> primPQ(Grafo grafo) {

        HashSet<String> remain = new HashSet<String>();
        for (String v : grafo.map.keySet())
            remain.add(v);
        remain.remove(grafo.getOrigen());

        PriorityQueue<Edge> queue = new PriorityQueue<Edge>();
```

```
        HashMap<String, ArrayList<String>> resultado = new HashMap<String,
ArrayList<String>>();
        String origen = grafo.getOrigen();
        String destino = "";
        while (!remain.isEmpty()) {
            for (Entry<String, Double> entry : grafo.map.get(origen).entrySet()) {
                destino = entry.getKey();
                if (remain.contains(destino)) {
                    queue.add(new Edge(origen, destino, entry.getValue()));
                }
            }
            Edge edge = null;
            do {
                edge = queue.poll();
                destino = edge.destino;
            } while (!remain.contains(destino));
            origen = edge.origen;

            remain.remove(destino);

            if (!resultado.containsKey(origen)) {
                resultado.put(origen, new ArrayList<String>());
            }
            resultado.get(origen).add(destino);

            origen = destino;
        }

        return resultado;
    }
```

1.3 ALGORITMO DE KRUSKAL

Este algoritmo se inicializa igual que el de Prim, pero en su funcionamiento ordena todos los nodos de menor a mayor, después coge el menor y comprueba si se crea algún ciclo, si no se genera, lo añade a la lista de nodos escogidos para el MST y repite el proceso hasta que termina.

```
public static HashMap<String, ArrayList<String>> kruskal(Grafo grafo) {

        HashMap<String, Double> remain = new HashMap<String, Double>();
        for (String v : grafo.map.keySet())
            remain.put(v, Double.MAX_VALUE);
        remain.remove(grafo.getOrigen());

        heap.clear();
```

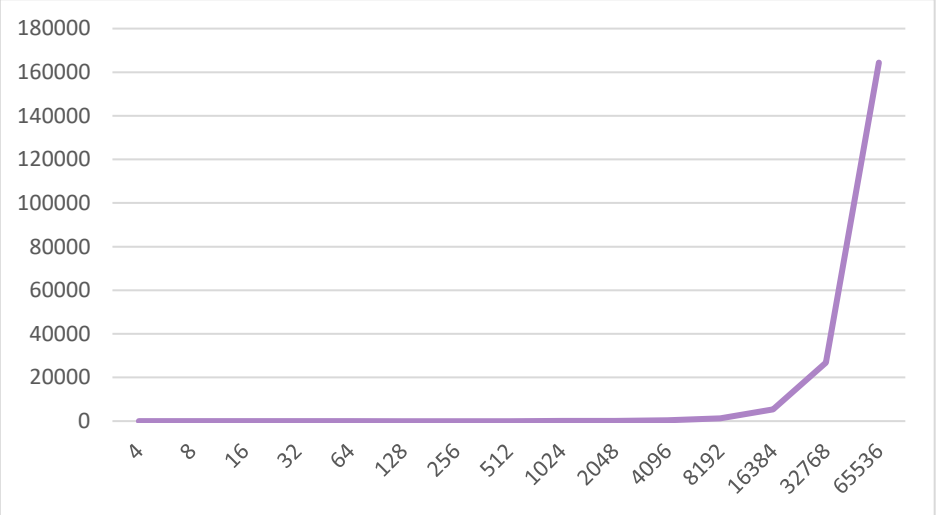
```
        String minKey = grafo.getOrigen();
        String to, from;
        boolean firstLoop = true;
        HashMap<String, ArrayList<String>> resultado = new HashMap<String,
ArrayList<String>>();
        while (!remain.isEmpty()) {
            // Obtenemos el valor minimo
            double minValue = Double.MAX_VALUE;
            if (firstLoop)
                firstLoop = false;
            else
                minKey = remain.keySet().stream().findFirst().toString();

            for (Entry<String, Double> entry : remain.entrySet()) {
                if (entry.getValue() < minValue) {
                    minValue = entry.getValue();
                    minKey = entry.getKey();
                }
            }

            remain.remove(minKey);

            for (Entry<String, Double> entry : grafo.map.get(minKey).entrySet()) {
                to = entry.getKey();
                Double weight = grafo.getWeight(heap.get(to), to);

                weight = (weight == null) ? Double.MAX_VALUE : weight;

                if (remain.containsKey(to) && entry.getValue() < Double.MAX_VALUE &&
entry.getValue() < weight) {
                    remain.put(to, entry.getValue());
                    heap.put(to, minKey);
                }
            }
        }
        // Añadir resultado
        for (Entry<String, String> entry : heap.entrySet()) {
            to = entry.getKey();
            from = entry.getValue();
            if (!resultado.containsKey(from))
                resultado.put(from, new ArrayList<String>());
            resultado.get(from).add(to);
        }
        return resultado;
    }
```

2 ESTUDIO TEÓRICO

Para realizar este estudio generamos grafos aleatorios,

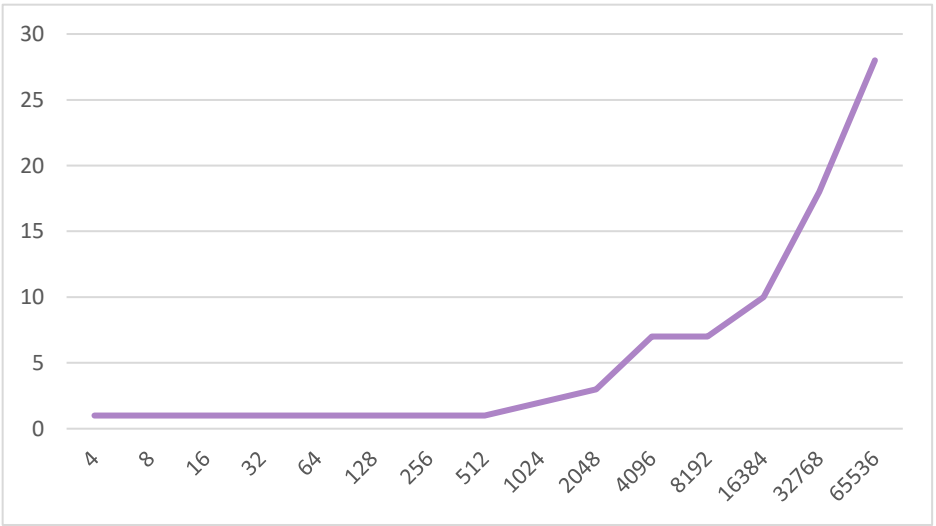
2.1 PRIM

Nº vértices	Tiempo (ms)
4	1
8	1
16	1
32	1
64	1
128	2
256	13
512	12
1024	36
2048	134
4096	342
8192	1307
16384	5445
32768	26874
65536	164397



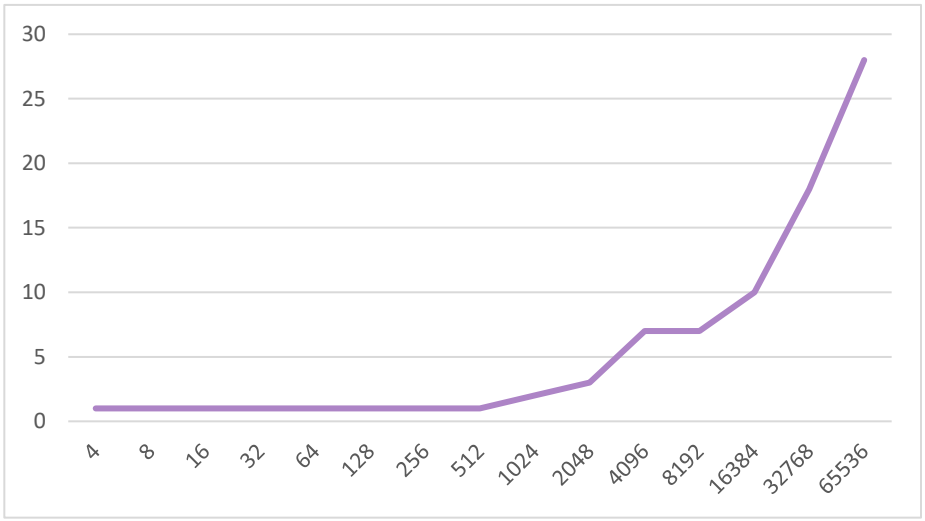
2.2 PRIM COLA DE PRIORIDAD

Nº vértices	Tiempo (ms)
4	1
8	1
16	1
32	1
64	1
128	1
256	1
512	1
1024	2
2048	3
4096	7
8192	7
16384	10
32768	18
65536	28



2.3 KRUSKAL

Nº vértices	Tiempo (ms)
4	1
8	1
16	1
32	1
64	2
128	3
256	6
512	11
1024	18
2048	59
4096	88
8192	352
16384	1554
32768	7587
65536	28945



3 ESTUDIO EXPERIMENTAL

3.1 GRAPHPRIMKRUSKAL.TXT

3.1.1 Prim

Tiempo de ejecución: **2** ms | Coste total: 313

MST:

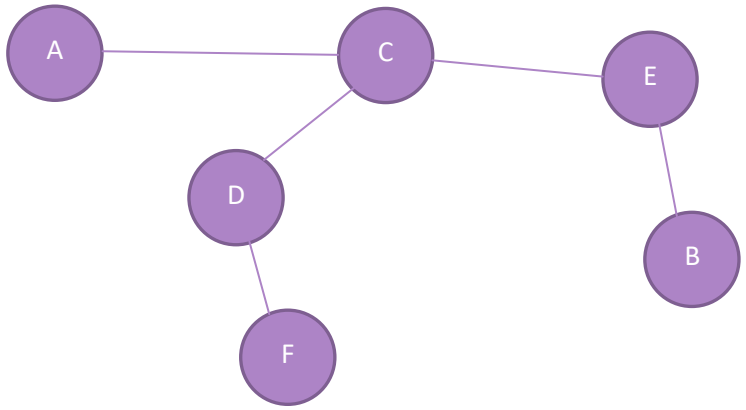
A – C

C – E

C – D

D – F

E – B



3.1.2 Prim cola de prioridad

Tiempo de ejecución: **6** ms | Coste total: 313

MST:

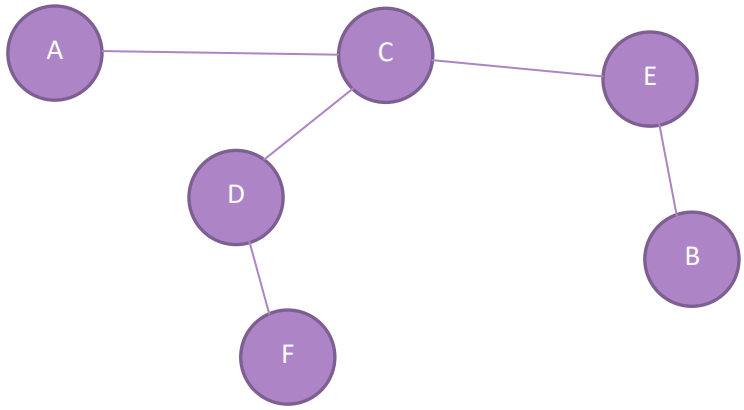
A – C

C – E

C – D

D – F

E – B



3.1.3 Kruskal

Tiempo de ejecución: **1** ms | Coste total: 313

MST:

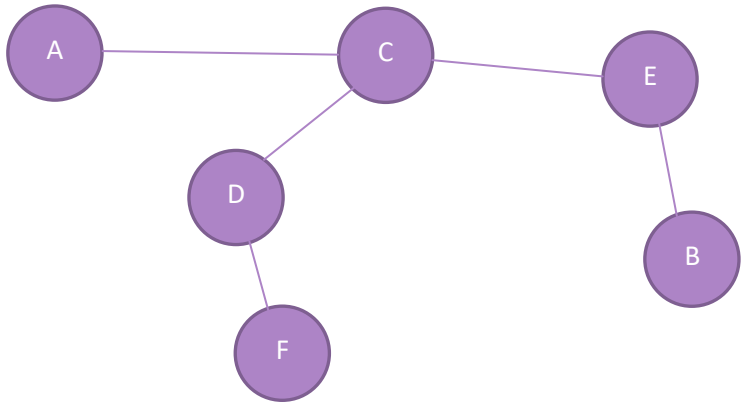
A – C

C – E

C – D

D – F

E – B



3.2 GRAPHEDALAND.TXT

3.2.1 Prim

Tiempo de ejecución: **1** ms | Coste total: 13911

MST:

Huelva - Badajoz

Gerona - Lérida

Jaén - Sevilla

Barcelona - Gerona

Badajoz - Cáceres

Granada - Jaén

Albacete - Valencia

Albacete - Madrid

Bilbao - Oviedo

Bilbao - Zaragoza

Sevilla - Huelva

Sevilla - Cádiz

Valladolid - Bilbao

Valladolid - Vigo

Zaragoza - Barcelona

Vigo - Coruña

Madrid - Valladolid

Almería - Granada

Almería - Murcia

Murcia - Albacete

3.2.2 Prim cola de prioridad

Tiempo de ejecución: **2** ms | Coste total: 13911

MST:

Huelva - Badajoz

Gerona - Lérida

Jaén - Sevilla

Barcelona - Gerona

Badajoz - Cáceres

Granada - Jaén

Albacete - Valencia

Albacete - Madrid

Bilbao - Oviedo

Bilbao - Zaragoza

Sevilla - Huelva

Sevilla - Cádiz

Valladolid - Bilbao

Valladolid - Vigo

Zaragoza - Barcelona

Vigo - Coruña

Madrid - Valladolid

Almería - Granada

Almería - Murcia

Murcia - Albacete

3.2.3 Kruskal

Tiempo de ejecución: **1** ms | Coste total: 13911

MST:

Huelva - Badajoz

Gerona - Lérida

Jaén - Sevilla

Barcelona - Gerona

Badajoz - Cáceres

Granada - Jaén

Albacete - Valencia

Albacete - Madrid

Bilbao - Oviedo

Bilbao - Zaragoza

Sevilla - Huelva

Sevilla - Cádiz

Valladolid - Bilbao

Valladolid - Vigo

Zaragoza - Barcelona

Vigo - Coruña

Madrid - Valladolid

Almería - Granada

Almería - Murcia

Murcia - Albacete

3.3 GRAPHEDALANDLARGE.TXT

3.3.1 Prim

Tiempo de ejecución: **52** ms | Coste total: 85638

MST: *Demasiado grande*

3.3.2 Prim cola de prioridad

Tiempo de ejecución: **4** ms | Coste total: 85638

MST: *Demasiado grande*

3.3.3 Kruskal

Tiempo de ejecución: **29** ms | Coste total: 85638

MST: *Demasiado grande*

¿El resultado de la ejecución de cada algoritmo es único?

Como podemos comprobar, no.

¿El resultado de la ejecución de los dos algoritmos debe ser el mismo?

No tiene por qué, aunque si el grafo tiene pocos nodos, por lo general la solución será siempre la misma.

Si el peso de las aristas fuese la distancia entre dos ciudades, con la estructura resultante, ¿podemos determinar el camino mínimo entre dos pares de ciudades cualquiera?

No, ya que no sabemos por qué nodos pasa.

Bibliografía:

Apuntes de clase

https://es.wikipedia.org/wiki/Algoritmo_de_Prim

https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal

https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm#introduction_to_kruskal_algorithm

https://www.javatpoint.com/prim-algorithm

https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/