Practica 2:
Algoritmo Greedy

Grupo

Alejandro Damaso Pastor

Francisco Escamilla Vizcaino

Índice

Introducción

Características:

Este tipo de algoritmos son usados normalmente para resolver problemas de optimización, como por ejemplo en la obtención de máximos y mínimos, tomando decisiones en función de la información disponible y una vez llegando a una esta solución no se repite.

Su principal ventaja podríamos decir que es que suelen ser fáciles y rápidos de implementar, pero a su vez tenemos que no siempre llegan a la solución óptima.

Elementos

Para poder hacer un enfoque para resolver un problema usando greedy necesitamos los siguientes elementos:

- Conjunto de candidatos: Consiste en los elementos seleccionables.

-Solución parcial: Es una selección de los candidatos a posible solución.

-Función de selección: Determina el mejor candidato del conjunto de candidatos seleccionables.

-Función de factibilidad: Determina si es posible realizar la funcion parcial para alcanzar una solución final.

-Criterio de lo que es una solución: Indica si la solución parcial resuelve el problema.

-Función objetivo: Valor de la solución.

Implementación de clases

Grafo

En esta clase lo que hacemos es generar un grafo y cuando está generado lo escribimos en memoria

```
package org.eda2.practica2.src;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.*;
import java.util.Map.Entry;

public class Grafo {

	public HashMap <String, HashMap<String, Double>> map = new HashMap<String,
HashMap<String, Double>>();

	private boolean dirigido = false;
	private String origen = "";
	private int numV;
```

```
	private int numA;
	private int size;

	public Grafo(String filename) {
		cargarGrafo(filename);
		this.size = this.map.size();
	}

	public Grafo(int numVertices, int densidad) {
		generarGrafo(numVertices, densidad);
		this.size = this.map.size();
	}

	public void generarGrafo(int numVertices, int densidad) {
		map.clear();
		// La key es el indice, el value es el numero de enlaces
		HashMap<Integer, Integer> nodes = new HashMap<Integer, Integer>();
		HashSet<Integer> remain = new HashSet<Integer>();
		for (int i = 0; i < numVertices; i++) {
			nodes.put(i, 0);
			remain.add(i);
		}
		Random r = new Random();
		int minAristas = numVertices - 1;// Número minimo de aristas para un grafo
conexo
		int maxAristas = numVertices * (numVertices - 1) / 2;// Número máximo de
aristas que puede contener el grafo
		// int numAristas = r.nextInt((maxAristas + 1) - minAristas) + minAristas;
		int numAristasPorVertice = (maxAristas - minAristas) * densidad / 100;

		int index = 0;
		while (!remain.isEmpty()) {
			int n = 1;
			if (!map.containsKey("" + index))
				map.put("" + index, new HashMap<String, Double>());

			int numEnlaces = nodes.get(index);
			while (numEnlaces < numAristasPorVertice) {
```

```
                                int indexVecino = (index + n) % numVertices;
                                int nAux = nodes.get(indexVecino);
                                if (nAux >= numAristasPorVertice) {
                                        remain.remove(nAux);
                                        continue;
                                }

                                double weight = r.nextDouble() * 1000;
                                BigDecimal bd = BigDecimal.valueOf(weight);
                                weight = bd.setScale(2,
RoundingMode.HALF_UP).doubleValue();
                                map.get("" + index).put("" + indexVecino, weight);
                                if (!dirigido) {
                                        if (!map.containsKey("" + indexVecino))
                                                map.put("" + indexVecino, new
HashMap<String, Double>());
                                        map.get("" + indexVecino).put("" + index, weight);
                                }
                                nodes.put(indexVecino, nAux++);
                                nodes.put(index, numEnlaces++);
                                n++;
                        }
                        remain.remove(index);
                        index++;
                }

                origen = "" + 0;
        }

        public void cargarGrafo(String filename) {
                map.clear();
                try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
                        String line = "";
                        line = br.readLine().trim();

                        if (Integer.parseInt(line) != 0)
                                dirigido = true;
```

```
                        while (line != null) {
                                line = br.readLine();
                                numV = Integer.parseInt(line);
                                for (int i = 0; i < numV; i++) {
                                        line = br.readLine();
                                        if (i == 0)
                                                origen = line;
                                        map.put(line, new HashMap<String, Double>());
                                }
                                line = br.readLine();
                                numA = Integer.parseInt(line);
                                for (int i = 0; i < numA; i++) {
                                        line = br.readLine();
                                        String[] atributo = line.split(" ");

                                        map.get(atributo[0]).put(atributo[1],
tryParseDouble(atributo[2]));
                                        if (!dirigido)
                                                map.get(atributo[1]).put(atributo[0],
tryParseDouble(atributo[2]));
                                }
                                line = br.readLine();
                        }

                } catch (IOException ioe) {
                        ioe.printStackTrace();
                }
        }

        public void print() {
                for (Entry<String, HashMap<String, Double>> entry : map.entrySet()) {
                        System.out.println(entry.getKey() + ":");

                        int c = 1;
                        for (Entry<String, Double> valueEntry : entry.getValue().entrySet()) {
                                System.out.println("\t" + c++ + ":");
                                System.out.println("\t\tDestino: " + valueEntry.getKey() +
"\n\t\tDistancia: " + valueEntry.getValue());
```

```
                        }
                        if (c == 1)
                                System.out.println("\t" + "No entry");
                }
        }

        static Double tryParseDouble(String text) {
                try {
                        return Double.parseDouble(text);
                } catch (NullPointerException e) {
                        return null;
                }
        }

        public Double getWeight(String v1, String v2) {
                return map.containsKey(v1) && map.get(v1).containsKey(v2) ?
map.get(v1).get(v2) : null;
        }

        public int getSize() {
                return size;
        }

        public String getOrigen() {
                return origen;
        }

        public boolean isDirigido() {
                return dirigido;
        }

        public int getNumA() {
                return numA;
        }

        public int getNumV() {
                return numV;
        }
```

```
}
```

Display

Muestra por pantalla los resultados de nuestros códigos usando la librería graphstream

```
package org.eda2.practica2.src;

import java.util.Set;
import java.util.ArrayList;
import java.util.HashMap;

import org.graphstream.graph.Graph;
import org.graphstream.graph.Node;
import org.graphstream.graph.implementations.SingleGraph;

public class Display {

	static final String nodestyle = "node {" + "	size: 20px;" + "
text-background-mode:plain;"
			+ "	text-alignment:under;" + "	text-style:bold;" + "}";

	public static void dibujarGrafo(Grafo grafo, HashMap<String, ArrayList<String>>
results) {
		System.setProperty("org.graphstream.ui", "javafx");
		Graph graph = new SingleGraph("EDAland");
		graph.setAttribute("ui.stylesheet", nodestyle);

		Set<String> set = grafo.map.keySet();
		for (String i : set) {
			Node n = graph.addNode(i);
			n.setAttribute("ui.label", i);
			if (i.contains(grafo.getOrigen())) {
				n.setAttribute("ui.style", "fill-color: rgb(0,100,255);");
			} else {
				n.setAttribute("ui.style", "fill-color: rgb(0,0,0);");
			}
		}
```

```
                for (String entry : results.keySet()) {
                        for (String value : results.get(entry)) {
                                graph.addEdge(entry + "-" + value, entry, value);
                        }
                }
                graph.display();
        }
}
```

App

Este es nuestro código main en el cual se ejecuta la aplicación

```
package org.eda2.practica2.src;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;
import java.util.concurrent.TimeUnit;
import java.util.Map.Entry;

public class App {

	private static Grafo grafo;

	private static final String DATASET_PATH = System.getProperty("user.dir") + "\\" +
"dataset\\";
	private static String filename;

	public static void main(String[] args) {
		System.out.println("Selecciona fichero de datos:\n" + "0:
graphPrimKruskal.txt\n" + "1: graphEDAland.txt\n"
					+ "2: graphEDAlandLarge.txt\n" + "3: grafo aleatorio");
		Scanner sc = new Scanner(System.in);
		int input = sc.nextInt();

		switch (input) {
		case 0:
			filename = "graphPrimKruskal.txt";
			break;

		case 1:
			filename = "graphEDAland.txt";
			break;

		case 2:
```

```
                filename = "graphEDAlandLarge.txt";
                break;

            case 3:
                filename = "";
                break;

            default:
                System.out.println("No has elegido ninguna opcion valida");
                sc.close();
                System.exit(0);
                break;
            }

            if (filename.isEmpty()) {
                System.out.println("Selecciona el tamaño del grafo:");
                input = sc.nextInt();
                System.out.println("Selecciona la densidad (%): ");
                int densidad = sc.nextInt();
                grafo = new Grafo(input,densidad);
            } else {
                System.out.println(DATASET_PATH);
                grafo = new Grafo(DATASET_PATH + filename);
            }

            grafo.print();
            System.out.println("Selecciona algoritmo:\n" + "0: Prim\n" + "1: Prim con
PQ\n" + "2: Kruskal");
            input = sc.nextInt();

            long startNano = System.nanoTime();
            HashMap<String, ArrayList<String>> resultado = null;
            switch (input) {
            case 0:
                resultado = Prim.mst(grafo);
                break;

            case 1:
```

```
                        resultado = Prim.mstPQ(grafo);
                        break;

                case 2:
                        resultado = Kruskal.mst(grafo);
                        break;

                default:
                        System.out.println("No has elegido ninguna opcion valida");
                        sc.close();
                        System.exit(0);
                        break;
                }
                long endNano = System.nanoTime();

                if (resultado == null) {
                        System.out.println("No hay resultados.");
                } else {
                        print(resultado);
                        System.out.println("Tiempo de ejecución para algoritmo PrimPQ: " +
(endNano - startNano) + " ns " + " o "
                                        + TimeUnit.MILLISECONDS.convert((endNano -
startNano), TimeUnit.NANOSECONDS) + " ms.");
                        long cost = cost(resultado, grafo);
                        System.out.println("\n\tEl coste es: " + cost);

                        System.out.println("Mostrar visualizacion por pantalla:\n" + "0: No\n"
+ "1: Si\n");
                        input = sc.nextInt();
                        if (input == 1)
                                Display.dibujarGrafo(grafo, resultado);
                }

                sc.close();
        }

        private static long cost(HashMap<String, ArrayList<String>> resultado, Grafo grafo)
{
```

```
                long sum = 0;
                for (Entry<String, ArrayList<String>> entry : resultado.entrySet()) {
                        String key = entry.getKey();
                        ArrayList<String> values = entry.getValue();

                        for (String value : values) {
                                HashMap<String, Double> map = grafo.map.get(value);
                                if (map != null) {
                                        for (Entry<String, Double> adjMapEntry :
map.entrySet()) {
                                                if (adjMapEntry.getKey() == key) {
                                                        sum += adjMapEntry.getValue();
                                                }
                                        }
                                }
                        }
                }
                return sum;
        }

        private static void print(HashMap<String, ArrayList<String>> resultado) {
                System.out.println("\nMST is : ");
                for (Entry<String, ArrayList<String>> entry : resultado.entrySet()) {
                        for (String value : entry.getValue()) {
                                System.out.println("\t" + entry.getKey() + " -- " + value);
                        }
                }
        }
}
```

MST

En esta clase combinamos el Prim, Prim PQ y el Kruskal

En esta parte del código lo que hace es que realiza un seguimiento del camino mas corto de un punto A a un punto B.

Para ello selecciona el punto de partida y se va moviendo a el punto que menos coste le suponga y así sucesivamente hasta llegar al punto de destino.

Con el HashSet guardamos los datos por los que no hemos pasado para después seleccionarlos como prioridad en nuestro recorrido.

Para el prim PQ hemos hecho lo mismo solo que hemos establecido una cola de prioridad

Aquí lo que intentamos hacer es ordenar las aristas en orden creciente de peso para así para así la arista que no entre en la solución se deseche.

```
	package org.eda2.practica2.src;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Map.Entry;

public class MST {
	private static class Edge implements Comparable<Edge> {

		public Edge(String origen, String destino, double weight) {
			super();
			this.origen = origen;
			this.destino = destino;
			this.weight = weight;
		}

		@Override
		public int compareTo(Edge o) {
			return Double.compare(this.weight, o.weight);
		}
```

```
                public String origen, destino;
                public double weight;
        }

        private static HashMap<String, String> heap = new HashMap<String, String>();
        private static HashMap<String, Double> weight = new HashMap<String,
Double>();

        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()) {
                        // Obtenemos el valor minimo
                        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);

                // Añadir resultado
                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;
}

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());
```

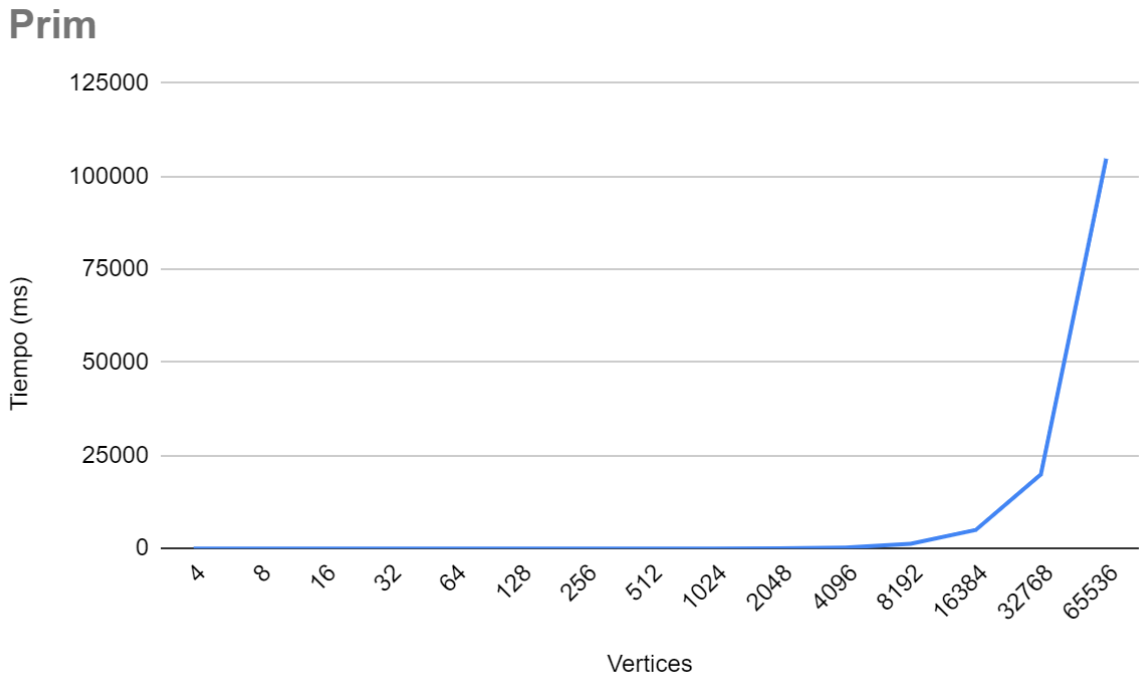
```
                PriorityQueue<Edge> queue = new PriorityQueue<Edge>();

                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;
        }

        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;
        }
}
```

Estudio Teórico

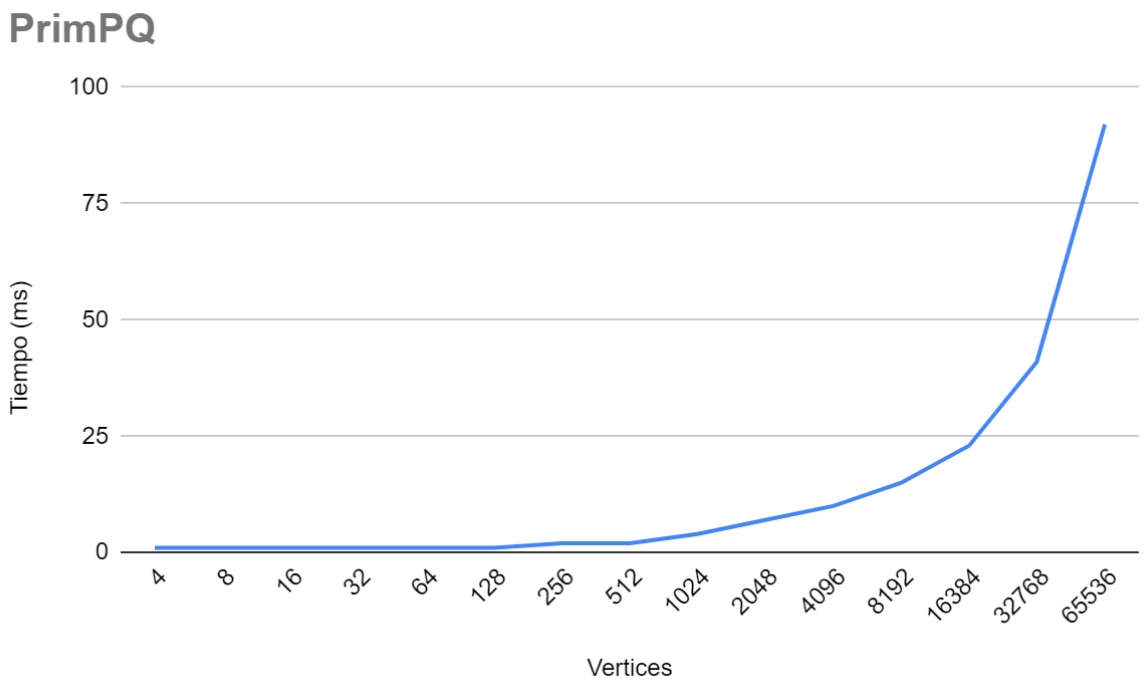
Algoritmo Prim



Vértices	Tiempo (ms)
4	1
8	1
16	1
32	1
64	1
128	2
256	8
512	7
1024	18
2048	76
4096	294
8192	1268
16384	5004
32768	19939

65536	104743

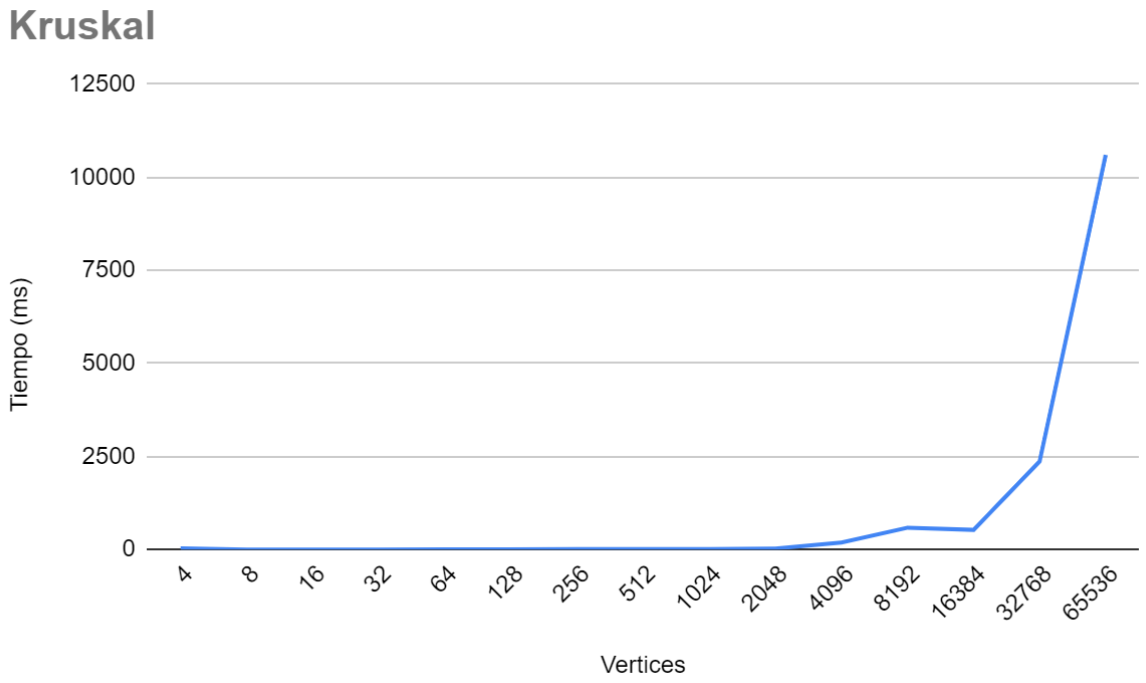
Algoritmo PrimPQ



Vertices	Tiempo (ms)
4	1
8	1
16	1
32	1
64	1
128	1
256	2
512	2
1024	4
2048	7
4096	10
8192	15
16384	23
32768	41

65536	92

Algoritmo Kruskal



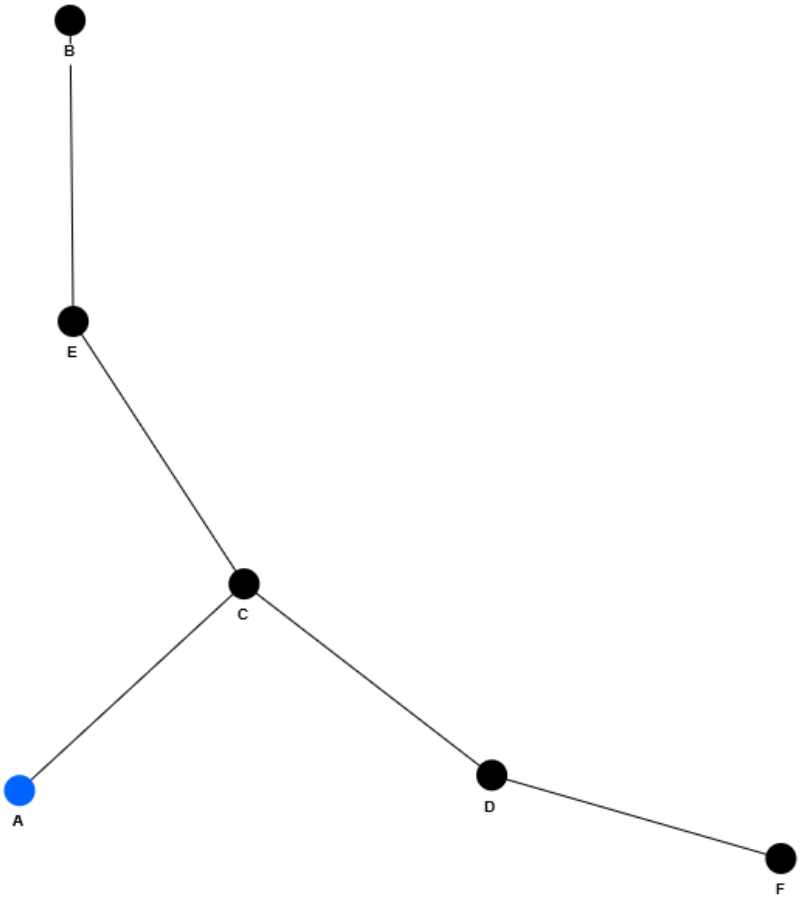
Vértices	Tiempo (ms)
4	33
8	1
16	1
32	2
64	3
128	5
256	10
512	15
1024	16
2048	23
4096	186
8192	589
16384	529
32768	2369

65536	10605

Estudio Experimental

graphPrimKruskal.txt

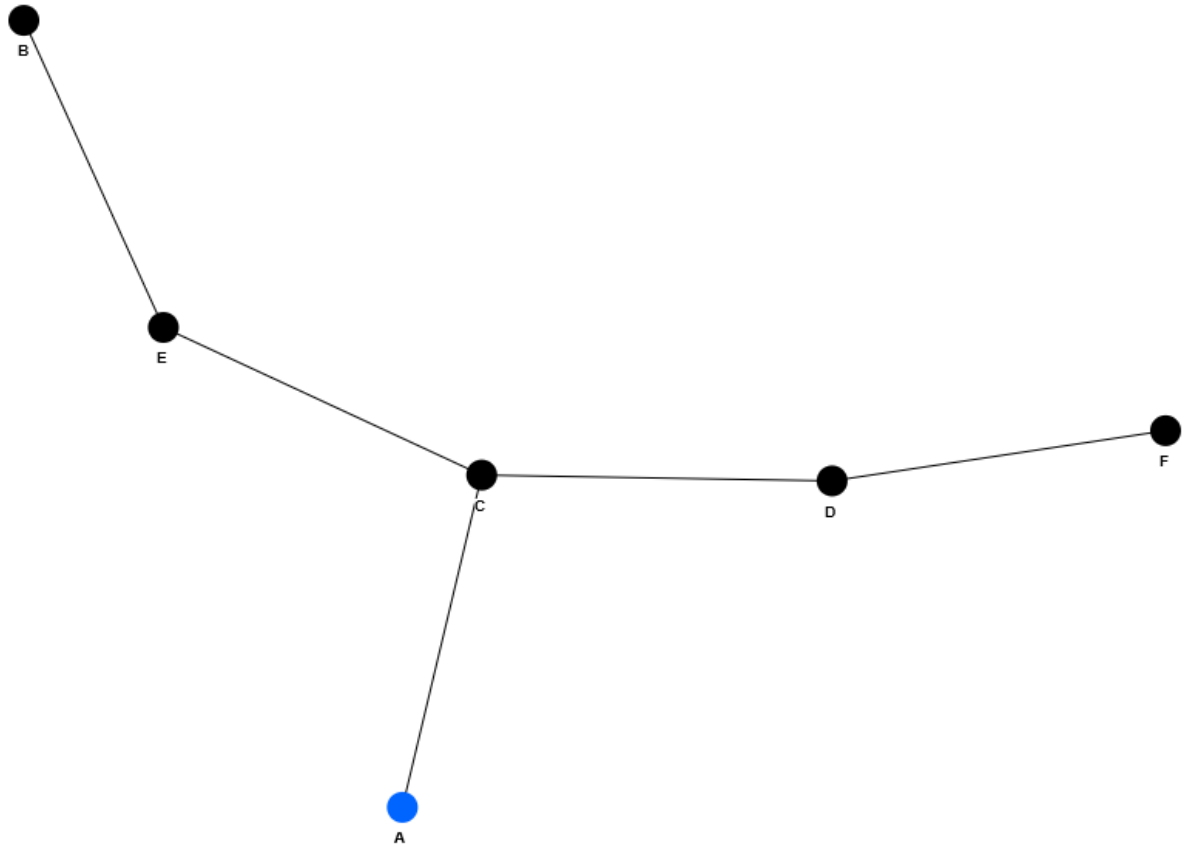
Prim



Tiempo de ejecución para algoritmo: 878400 ns o 0 ms.

Nª Vértices: 6 Nª Aristas: 10
El coste es: 313

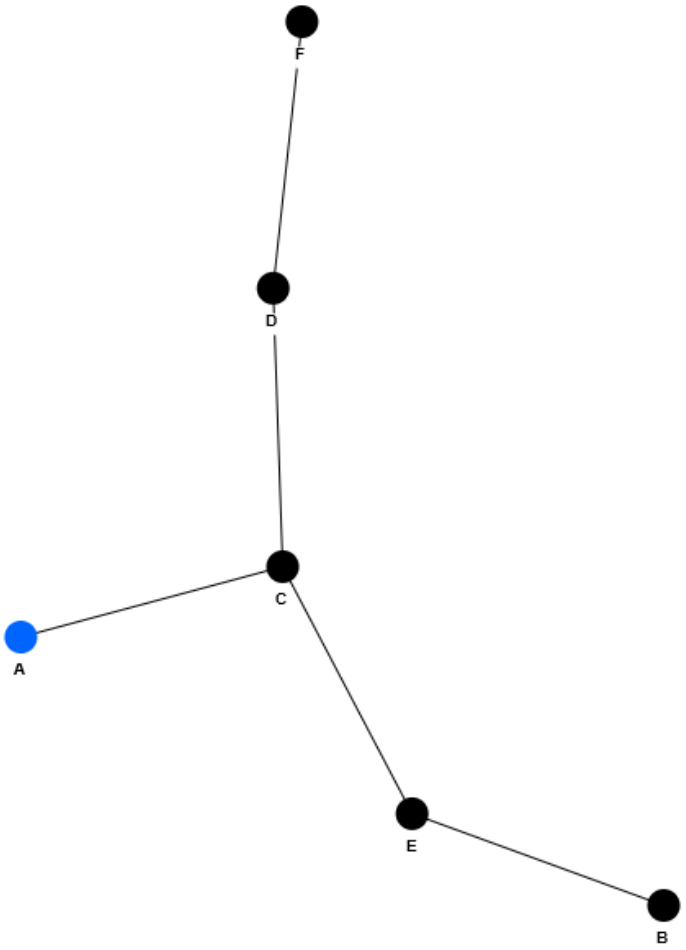
Prim PQ



Tiempo de ejecución para algoritmo: 1396000 ns o 1 ms.

Nª Vertices: 6 Nª Aristas: 10
El coste es: 313

Kruskal

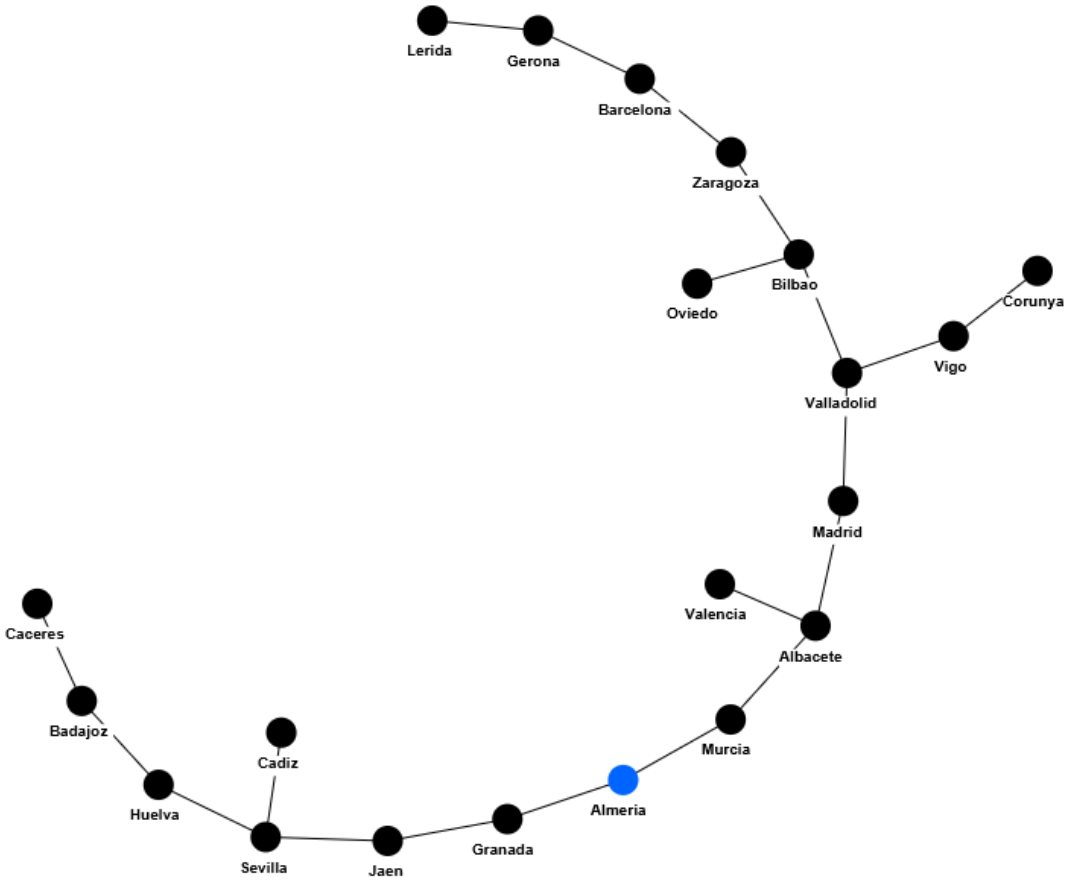


Tiempo de ejecución para algoritmo: 34944500 ns o 34 ms.

Nª Vertices: 6 Nª Aristas: 10
El coste es: 313

graphEDAland.txt

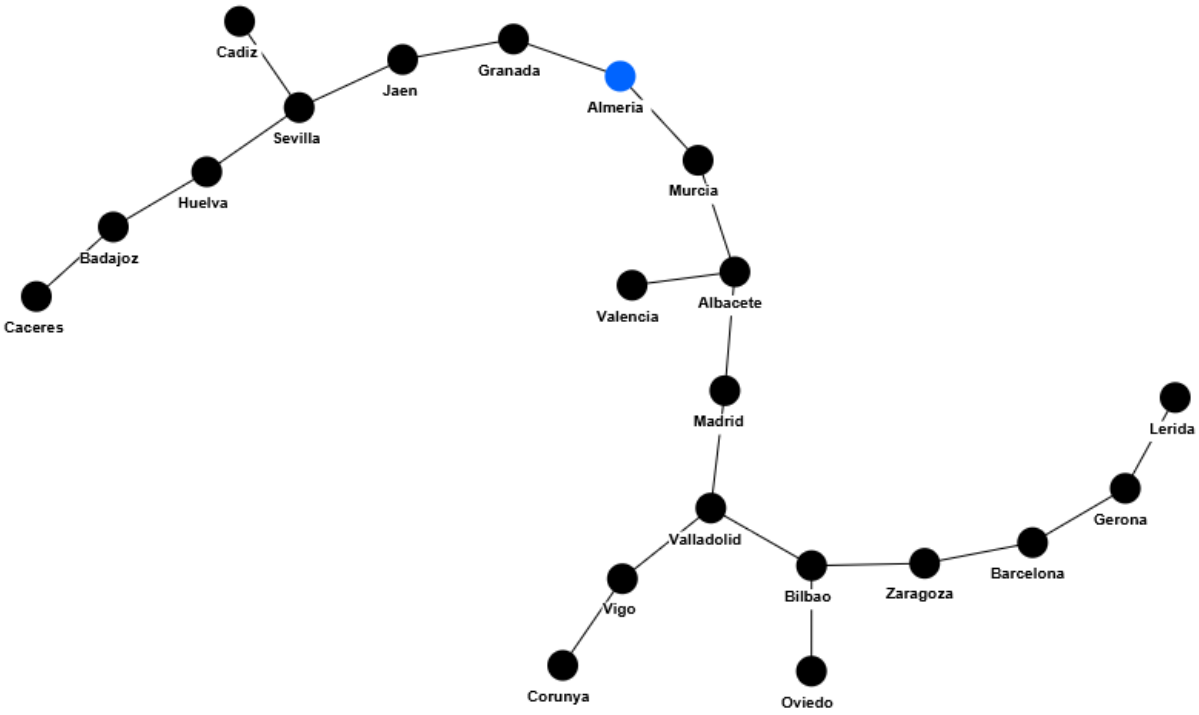
Prim



Tiempo de ejecución para algoritmo: 1109100 ns o 1 ms.

Nª Vértices: 21 Nª Aristas: 29
El coste es: 13911

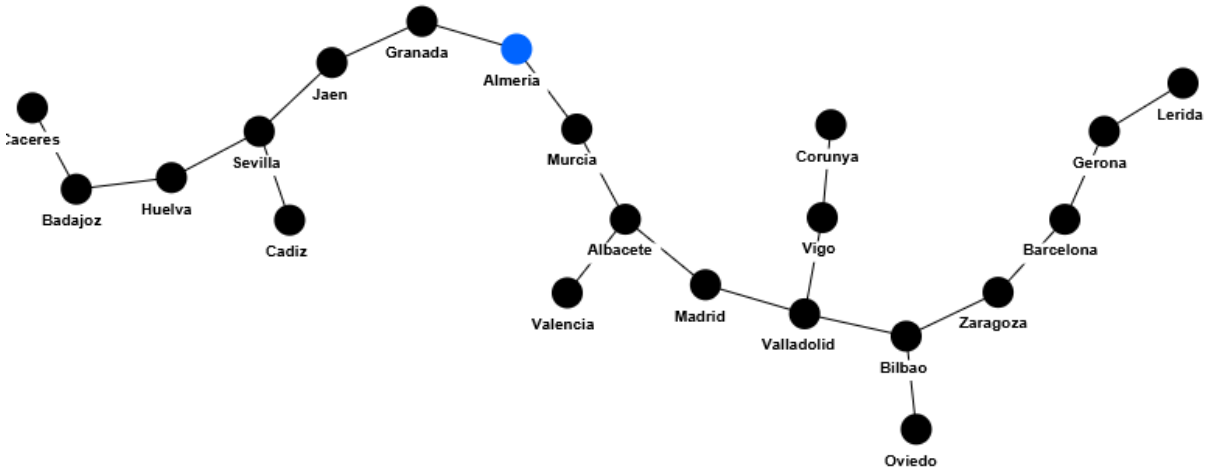
Prim PQ



Tiempo de ejecución para algoritmo: 1541400 ns o 1 ms.

Nª Vertices: 21 Nª Aristas: 29
El coste es: 13911

Kruskal

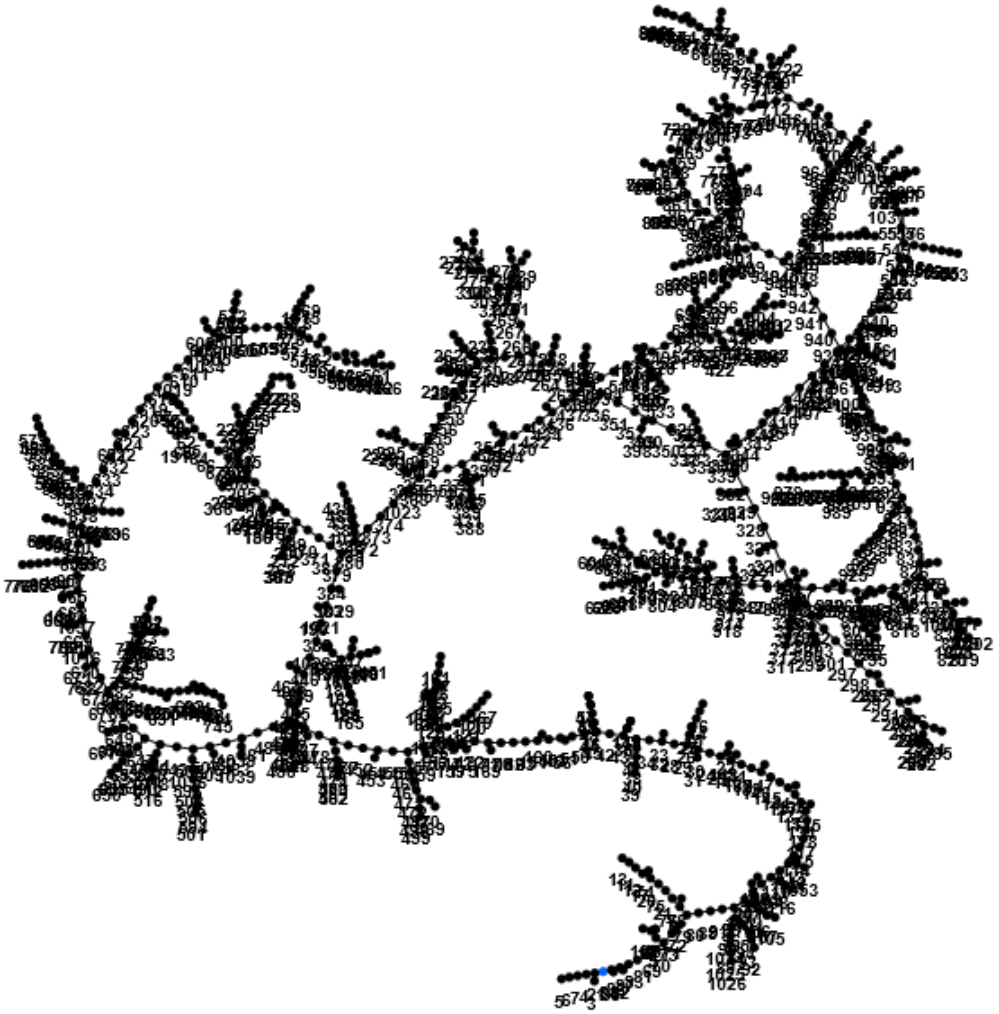


Tiempo de ejecución para algoritmo: 34776700 ns o 34 ms.

Nª Vertices: 21 Nª Aristas: 29
El coste es: 13911

graphEDAlandLarge.txt

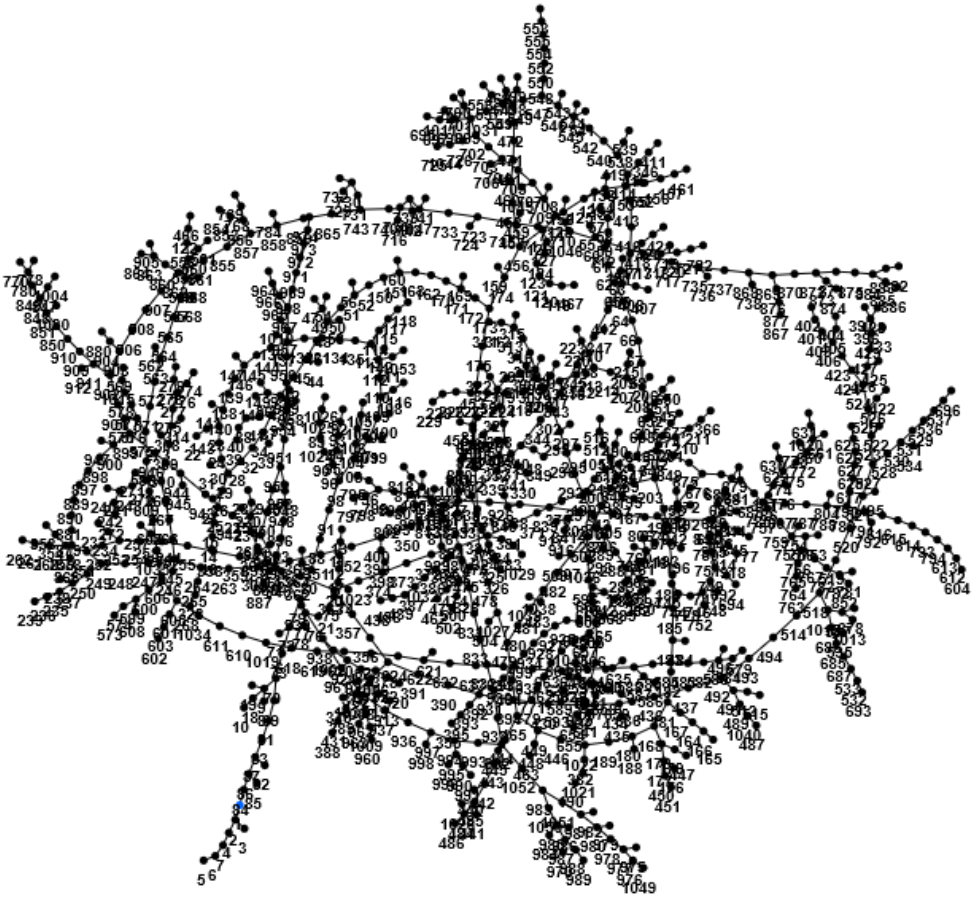
Prim



Tiempo de ejecución para algoritmo: 42573000 ns o 42 ms.

Nª Vertices: 1053 Nª Aristas: 2017
El coste es: 85638

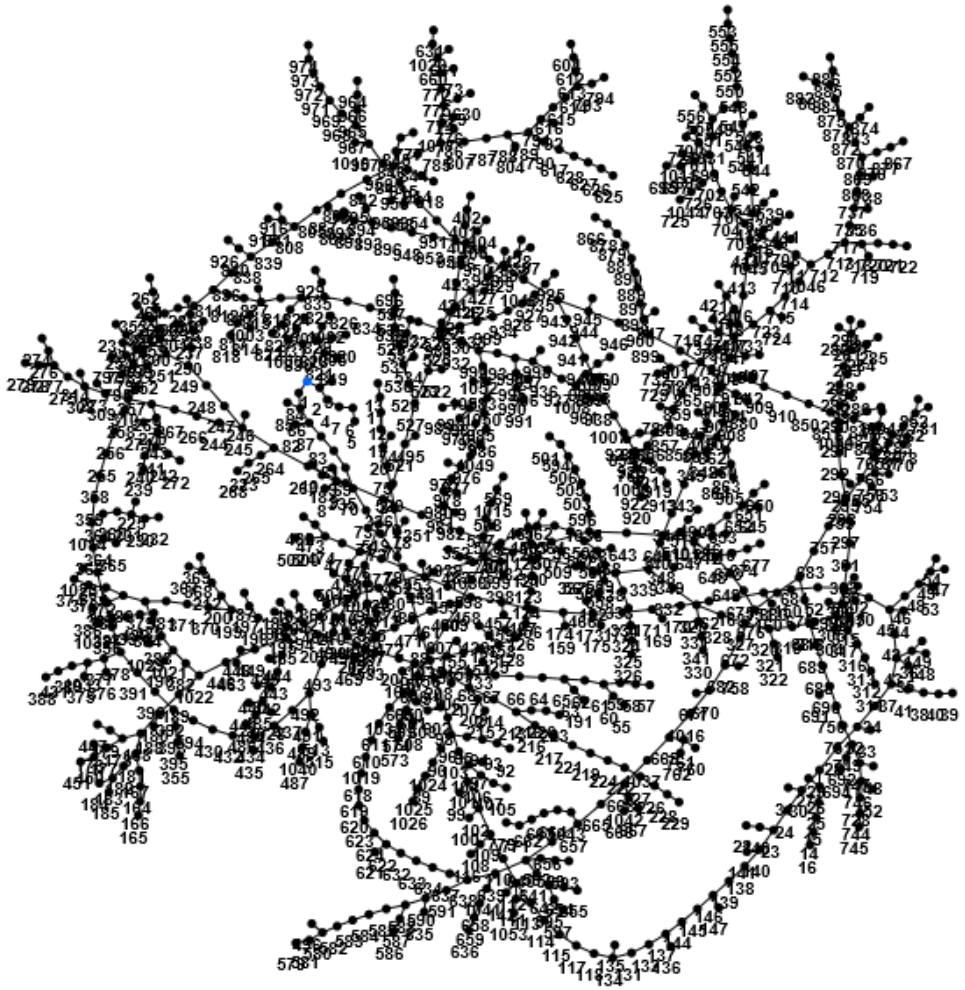
Prim PQ



Tiempo de ejecución para algoritmo: 5594000 ns o 5 ms.

Nª Vertices: 1053 Nª Aristas: 2017
El coste es: 85638

Kruskal



Tiempo de ejecución para algoritmo: 75276200 ns o 75 ms.

Nª Vertices: 1053 Nª Aristas: 2017
El coste es: 85638

- **¿El resultado de la ejecución de cada algoritmo es único?**
  Como se puede apreciar en nuestro caso nos sale cada algoritmo diferente

- **¿El resultado de la ejecución de los dos algoritmos debe ser el mismo?, ¿por qué?**
  Es muy complicado que eso ocurra, pero no es imposible, ya que se pueden dar casos en los que sí suceda.

- **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 de coger los pesos directamente no conocemos por dónde pasa, si no que nos haría un camino directo, ya que por ejemplo si el camino es a, b y c, queremos que nos diga que pasa por el b.

BIBLIOGRAFÍA

- https://graphstream-project.org/