
Práctica 01

Esquema Algorítmico Greedy o Voraz: Pavimentación de caminos



Estructura de Datos y Algoritmos II
Rafael Expósito Mullor
Luis Díaz
Universidad de Almería



INDICE

1. Introducción: Esquema algorítmico voraz (Greedy).
2. Estudio de la implementación.



1. Introducción

Los algoritmos que se obtienen aplicando este esquema se denominan, por extensión, algoritmos voraces. El esquema forma parte de una familia de algoritmos mucho más amplia denominada ALGORITMOS DE BUSQUEDA LOCAL de la que también forman parte, por ejemplo, el método del gradiente, los algoritmos *Hill-Climbing*, los algoritmos genéticos, etc.

Las condiciones que, de forma general, deben cumplir los problemas que son candidatos a ser resueltos usando un algoritmo voraz son las siguientes:

- El problema a resolver ha de ser de optimización [MIN/MAX] y debe existir una función, la **función objetivo**, que es la que hay que minimizar o maximizar.
- Existe un conjunto de valores posibles para cada una de las variables de la función objetivo, su dominio.
- La solución al problema debe ser expresable en forma de secuencia de decisiones y debe existir una función que permita determinar cuándo una secuencia de decisiones es solución para el problema (**función solución**). Se entiende por decisión la asociación a una variable de un valor de su dominio.
- Debe existir una función que permita determinar si una secuencia de decisiones viola o no las restricciones, la **función factible**.

El propósito de un algoritmo voraz es encontrar una solución, es decir, una asociación de valores a todas las variables tal que el valor de la función objetivo sea óptimo.

Para ello sigue un proceso secuencial en el que a cada paso toma una decisión (decide qué valor del dominio le ha de asignar a la variable actual) aplicando siempre el mismo criterio (función de selección).

La decisión es localmente óptima, es decir, ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera un valor mejor, y luego comprueba si la puede incorporar a la secuencia de decisiones que ha tomado hasta el momento, es decir, comprueba que la nueva decisión junto con todas las tomadas anteriormente no violan las restricciones y así consigue una nueva secuencia de decisiones factible.

En el siguiente paso el algoritmo voraz se encuentra con un problema idéntico, pero estrictamente menor, al que tenía en el paso anterior y vuelve a aplicar la misma función de selección para tomar la siguiente decisión. Esta es, por tanto, una técnica descendente.

Pero nunca se vuelve a reconsiderar ninguna de las decisiones tomadas. Una vez que a una variable se le ha asignado un valor localmente óptimo y que hace que la secuencia



de decisiones sea factible, nunca más se va a intentar asignar un nuevo valor a esa misma variable.

A continuación, se presenta el esquema general de un algoritmo voraz en pseudocódigo:

```
Greedy (conjunto de candidatos C): solución S  
  
S = ∅  
while (S no sea una solución y C ≠ ∅) {  
    x = selección(C)  
    C = C - {x}  
    if (S ∪ {x} es factible)  
        S = S ∪ {x}  
}  
  
if (S es una solución)  
    return S;  
else  
    return "No se encontró una solución";
```

2. Estudio de la implementación

Para la resolución del problema de la pavimentación de caminos propuesto en esta práctica se han implementado tres algoritmos voraces: el algoritmo de Prim, Prim con cola de prioridad y el algoritmo de Kruskal.

Cabe mencionar que para la creación del grafo se ha utilizado como estructura de datos un HashMap, puesto que no es necesario ordenar.

2.1. ALGORITMO DE PRIM

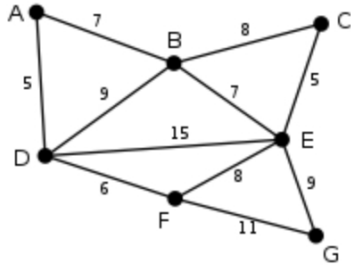
El algoritmo de Prim, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de recubrimiento mínimo. Es decir, es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

Para la implementación del algoritmo se ha tenido en cuenta el siguiente esquema general de funcionamiento:

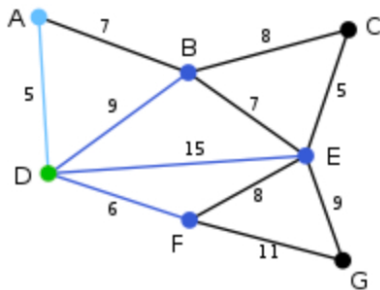
1. Se marca un vértice cualquiera. Será el vértice de partida.
2. Se selecciona la arista de menor peso incidente en el vértice seleccionado anteriormente y se selecciona el otro vértice en el que incide dicha arista.
3. Repetir el paso 2 siempre que la arista elegida enlace un vértice seleccionado y otro que no lo esté. Es decir, siempre que la arista elegida no cree ningún ciclo.

4. El árbol de expansión mínima será encontrado cuando hayan sido seleccionados todos los vértices del grafo.

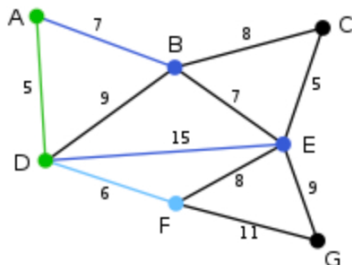
EJEMPLO DETALLADO DE FUNCIONAMIENTO



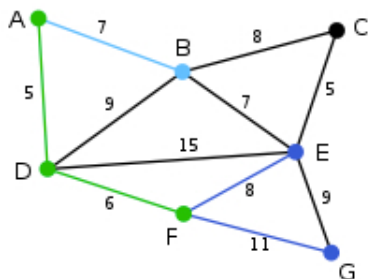
→ Este es el grafo inicial. Los números indican el peso de las aristas. Se elige de manera aleatoria uno de los vértices que será el vértice de partida. En este caso se ha elegido el vértice D.



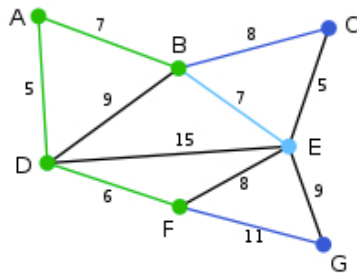
→ Se selecciona la arista de menor peso de entre todas las incidentes en el vértice D, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista AD.



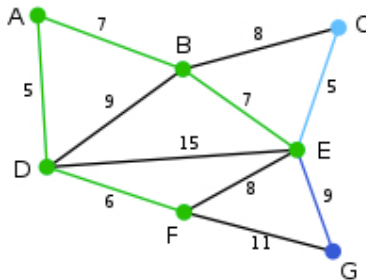
→ Ahora se selecciona la arista de menor peso de entre todas las incidentes en los vértices D y A, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista DF.



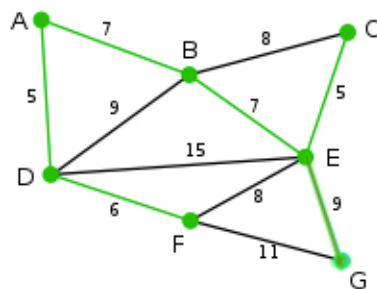
→ Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A y F, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista AB. Llegado a este punto, la arista DB no podrá ser seleccionada, ya que formaría el ciclo ABD.



→ Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A, F y B, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista BE. Llegado a este punto, las aristas DE y EF no podrán ser seleccionadas, ya que formarían los ciclos ABED y ABEFD respectivamente.



→ Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A, F, B y E, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista EC. Llegado a este punto, la arista BC no podrá ser seleccionada, ya que formaría el ciclo BEC.



→ Solo queda disponible el vértice G, por lo tanto se selecciona la arista de menor peso que incide en dicho vértice. Es la arista EG. Como todos los vértices ya han sido seleccionados el proceso ha terminado. Se ha obtenido el árbol de expansión mínima con un peso de 39.

Una vez visto el funcionamiento del algoritmo en profundidad, se muestra la implementación a la que hemos llegado, además de algunas consideraciones que hemos querido señalar.

IMPLEMENTACIÓN RESULTANTE

```
public List<Arista> prim() {
    long startNano = System.nanoTime();
    long startMili = System.currentTimeMillis();

    if (origen == null || !this.grafo.containsKey(origen))
        throw new RuntimeException("No puede ser nulo.");

    HashMap<String, Double> pesos = new HashMap<String, Double>();
    HashMap<String, String> aristas = new HashMap<String, String>();
    HashSet<String> restantes = new HashSet<String>();
    List<Arista> resultado = new ArrayList<Arista>();
    String desde = null;

    for (String vertice : this.grafo.keySet()) {
        restantes.add(vertice);
    }
    restantes.remove(origen);
```



```

for (String vertice : restantes) {
    Double peso = getPeso(origen, vertice);
    if (peso != null) {
        aristas.put(vertice, origen);
        pesos.put(vertice, peso);
    } else {
        aristas.put(vertice, null);
        pesos.put(vertice, INFINITO);
    }
}

aristas.put(origen, origen);
pesos.put(origen, 0.0);

while (!restantes.isEmpty()) {
    double min = INFINITO;
    desde = null;
    for (String v : restantes) {
        double peso = pesos.get(v);
        if (peso < min) {
            min = peso;
            desde = v;
        }
    }
    if (desde == null)
        break;

    restantes.remove(desde);
    String aux = aristas.get(desde);
    resultado.add(new Arista(aux, desde, getPeso(aux, desde)));

    for (String hasta : restantes) {
        Double peso = getPeso(desde, hasta);
        if (peso != null && peso < pesos.get(hasta)) {
            pesos.put(hasta, peso);
            aristas.put(hasta, desde);
        }
    }
}

long endNano = System.nanoTime();
long endMili = System.currentTimeMillis();
System.out.println("Tiempo de ejecución para algoritmo Prim: " + (endNano -
startNano) + " nanosegundos. | " + (endMili - startMili) + " milisegundos.");
return resultado;
}

```

Función de Selección del algoritmo



CONSIDERACIONES

1. Todo lo necesario para medir tiempos, tanto en nanosegundos como milisegundos, se ha añadido ya junto a la implementación del propio algoritmo al considerar que era más cómodo para nosotros.
2. La estructura de datos elegida para almacenar los resultados es un ArrayList.
3. La función de selección del algoritmo es lo que engloba el bucle while en el código.

2.2. ALGORITMO DE PRIM CON PQ

La diferencia más significativa con el algoritmo anterior es que en este caso usaremos como estructura auxiliar una cola de prioridad en la que se almacenarán aristas, además de modificarse el núcleo principal del algoritmo, la función de selección, para hacer uso de la mencionada cola de prioridad.

La prioridad en este caso será el coste y se entenderá que la arista con mayor prioridad será la de menor coste.

IMPLEMENTACIÓN RESULTANTE

```
public List<Arista> primPQ() {
    long startNano = System.nanoTime();
    long startMili = System.currentTimeMillis();
    String origen = this.origen;
    if (origen == null || !this.grafo.containsKey(origen))
        throw new RuntimeException("No puede ser nulo.");

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

    PriorityQueue<Arista> colaDePrioridad = new PriorityQueue<Arista>();
    List<Arista> resultado = new ArrayList<Arista>();
    String desde = origen;
    String hasta;
    Double peso;
    Arista aux;

    for (String vertice : this.grafo.keySet()) {
        restantes.add(vertice);
    }
    restantes.remove(origen);
    while (!restantes.isEmpty()) {
        for (Entry<String, Double> iterador : this.grafo.get(desde).entrySet()) {
            hasta = iterador.getKey();
            peso = iterador.getValue();
            if (restantes.contains(hasta)) {
                aux = new Arista(desde, hasta, peso);
            }
        }
    }
}
```




```

        colaDePrioridad.add(aux);
    }
}
do {
    aux = colaDePrioridad.poll();
    desde = aux.getOrigen();
    hasta = aux.getDestino();
    peso = aux.getPeso();
} while (!restantes.contains(hasta));

restantes.remove(hasta);
resultado.add(new Arista(aux.getOrigen(), aux.getDestino(), aux.getPeso()));
desde = hasta;
}
long endNano = System.nanoTime();
long endMili = System.currentTimeMillis();
System.out.println("Tiempo de ejecución para algoritmo PrimPQ: " +
    (endNano - startNano) + " nanosegundos. || " + (endMili - startMili) + "
    milisegundos.");
return resultado;
}

```

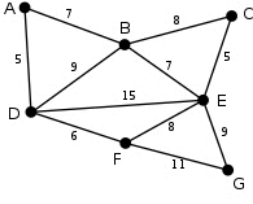
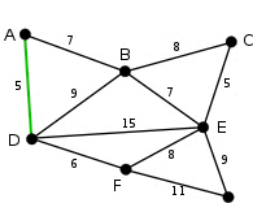
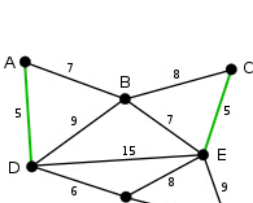
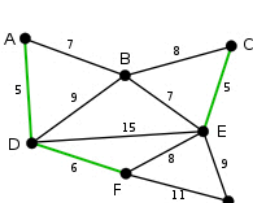
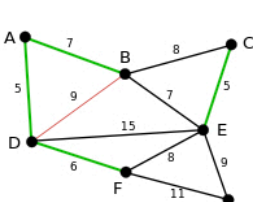
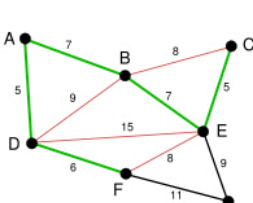
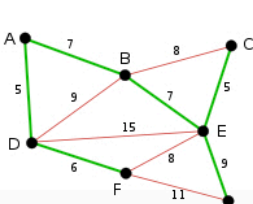
2.3. ALGORITMO DE KRUSKAL

Al igual que con el algoritmo de Prim, el algoritmo de Kruskal dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de recubrimiento mínimo. Es decir, es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

Para la implementación del algoritmo se ha tenido en cuenta el siguiente esquema de funcionamiento:

1. Se selecciona, de entre todas las aristas restantes, la de menor peso siempre que no cree ningún ciclo.
2. Se repite el paso 1 hasta que se hayan seleccionado $|V| - 1$ aristas.

En la siguiente página puede observarse lo que sería un ejemplo detallado de funcionamiento:

	<p>Este es el grafo inicial. Los números indican el peso de las aristas. Se elige de manera aleatoria uno de los vértices que será el vértice de partida. En este caso se ha elegido el vértice D.</p>
	<p>Se selecciona, de entre todas las aristas restantes, la de menor siempre que no cree ningún ciclo. Las aristas de menor peso son las aristas AD y CE (5). Se ha seleccionado de manera aleatoria la arista AD.</p>
	<p>Se selecciona, de entre todas las aristas restantes, la de menor siempre que no cree ningún ciclo. Ésta es la arista CE.</p>
	<p>Seleccionamos DF, con peso 6, que es la siguiente arista de menor peso que no forma ciclos.</p>
	<p>De las aristas restantes, las de menor peso son las aristas AB y BE, de peso 7. AB se elige aleatoriamente, y se añade al conjunto de las aristas seleccionadas. De este modo, la arista DB no puede ser seleccionada ya que formaría el ciclo ADB. Por tanto la marcamos en rojo.</p>
	<p>Siguiendo el proceso seleccionamos la arista BE con peso 7. Además marcamos en rojo las aristas BC, DE y FE ya que formarían los ciclos BCE, DEBA, FEBAD respectivamente.</p>
	<p>Por último se selecciona la arista EG de peso 9. Como han sido seleccionadas un número de aristas igual al número de vértices menos uno, el proceso ha terminado. Se ha obtenido el árbol de expansión mínima con un peso de 39.</p>



IMPLEMENTACIÓN DEL ALGORITMO

```

public List<Arista> kruskal() {
    String origen = this.origen;

    if (origen == null || !this.grafo.containsKey(origen))
        throw new RuntimeException("No puede ser nulo.");

    TreeMap<String, String> aristas = new TreeMap<String, String>();
    TreeMap<String, Double> restantes = new TreeMap<String, Double>();
    List<Arista> resultadoFinal = new ArrayList<Arista>();

    for (String vertice : grafo.keySet()) {
        restantes.put(vertice, INFINITO);
    }
    boolean esPrimero = true;
    while (!restantes.isEmpty()) {
        String keyMinima = restantes.firstKey();
        if (esPrimero) {
            keyMinima = origen;
            esPrimero = false;
        }

        Double valorMinimo = INFINITO;
        for (Entry<String, Double> entrada : restantes.entrySet()) {
            if (entrada.getValue() < valorMinimo) {
                valorMinimo = entrada.getValue();
                keyMinima = entrada.getKey();
            }
        }
        restantes.remove(keyMinima);
        for (Entry<String, Double> iterador : grafo.get(keyMinima).entrySet()) {
            String hasta = iterador.getKey();
            Double dActual = getPeso(aristas.get(hasta), hasta);
            dActual = dActual == null ? INFINITO : dActual;
            if (restantes.containsKey(hasta) && iterador.getValue() < INFINITO &&
                iterador.getValue() < dActual) {
                restantes.put(hasta, iterador.getValue());
                aristas.put(hasta, keyMinima);
            }
        }
    }
    for (Entry<String, String> iterador : aristas.entrySet()) {
        String hasta = iterador.getKey();
        String desde = iterador.getValue();
        resultadoFinal.add(new Arista(hasta, desde, getPeso(hasta, desde)));
    }
}

```

Función de selección.



3. Estudio Teórico

Una vez analizados ambos algoritmos, el uso de uno u otro va a estar condicionado por el tipo de grafo que tratemos.

La complejidad del algoritmo de Prim es siempre de orden $O(n^2)$, mientras que el orden de complejidad del algoritmo de Kruskal no solo depende del número de vértices, sino también del número de arcos o aristas.

De esta forma, en teoría, para grafos densos el número de aristas es cercano a $n(n-1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2 \log n)$, peor que la complejidad de Prim.

Sin embargo, para grafos dispersos en los que el número de aristas es próximo a n , el algoritmo de Kruskal es $O(n \log n)$, comportándose probablemente de forma más eficiente que el de Prim.

Preguntas:

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

- Si se aplican los algoritmos sobre grafos cuyas aristas tienen todas pesos diferentes, entendemos que sí, el resultado es único.

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

- El coste total (distancia) debe ser el mismo puesto que en los dos casos estamos obteniendo un árbol de recubrimiento mínimo, pero el camino seguido no tiene por qué.

3. 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, un árbol de recubrimiento mínimo no permite determinar el camino mínimo entre dos ciudades.

4. Estudio Experimental

En primer lugar, vamos a comprobar el resultado obtenido al ejecutar los algoritmos con los archivos de datos proporcionados en la práctica:

- Archivo graphEDAland.txt, dónde los nodos son ciudades españolas:

```
1 0
2 21
3 Almeria
4 Granada
5 Cadiz
6 Huelva
7 Sevilla
8 Jaen
9 Murcia
10 Caceres
11 Badajoz
12 Albacete
13 Valencia
14 Madrid
15 Barcelona
16 Girona
17 Lerida
18 Zaragoza
19 Bilbao
20 Oviedo
21 Valladolid
22 Vigo
23 Corunya
24 29
25 Almeria Granada 173
```

```
Tiempo de ejecución para algoritmo Prim: 2321700 nanosegundos. ||
[Almeria --> Granada| peso=173.0 , Granada --> Jaen| peso=99.0 , A
Coste      --> 4117.0

Tiempo de ejecución para algoritmo PrimPQ: 1572000 nanosegundos. |
[Almeria --> Granada| peso=173.0 , Granada --> Jaen| peso=99.0 , A
Coste      --> 4117.0

Tiempo de ejecución para algoritmo Kruskal: 2559600 nanosegundos.
[Albacete --> Murcia| peso=150.0 , Badajoz --> Huelva| peso=234.0
Coste      --> 4117.0
```

- Archivo graphEDAlandLange, donde los nodos son números:

```
1 0
2 1053
3 1
4 2
5 3
6 4
7 5
8 6
9 7
10 8
11 9
```

```
Tiempo de ejecución para algoritmo Prim: 194093500 nanosegundos.
[1 --> 2| peso=15.0 , 2 --> 3| peso=12.0 , 2 --> 4| peso=12.0 , 4
Coste Total --> 17315.0

Tiempo de ejecución para algoritmo PrimPQ: 18527900 nanosegundos.
[1 --> 2| peso=15.0 , 2 --> 4| peso=12.0 , 2 --> 3| peso=12.0 , 4
Coste Total --> 17315.0

Tiempo de ejecución para algoritmo Kruskal: 141026800 nanosegundo
[10 --> 18| peso=12.0 , 100 --> 102| peso=11.0 , 1000 --> 827| pe
Coste Total --> 17315.0
```



Por último, vamos a comparar los tiempos de ejecución de los algoritmos implementados.

Para ello se ha implementado un generador de grafos aleatorios, en el cual, hacemos una llamada al método que hemos nombrado *grafoAleatorio* al que se le pasa por parámetro el número de vértices que queremos que contenga en grafo y el nombre del archivo donde se va a almacenar.

En las siguientes imágenes se puede observar el main de la clase generador y un extracto del archivo de texto generado, así como una medida del tiempo tardado en generar dicho archivo.

```
public static void main(String[] args) throws FileNotFoundException {
    long startNano = System.nanoTime();
    long startMili = System.currentTimeMillis();
    grafoAleatorio(1000, "Datos02.txt");
    long endNano = System.nanoTime();
    long endMili = System.currentTimeMillis();
    System.out.println("Tiempo de ejecución: " + (endNano-startNano) +
```

Datos02.txt

```
1 0
2 1000
3 0
4 1
5 2
6 3
7 4
8 5
9 6
10 7
11 8
12 9
13 10
14 11
15 12
16 13
17 14
```

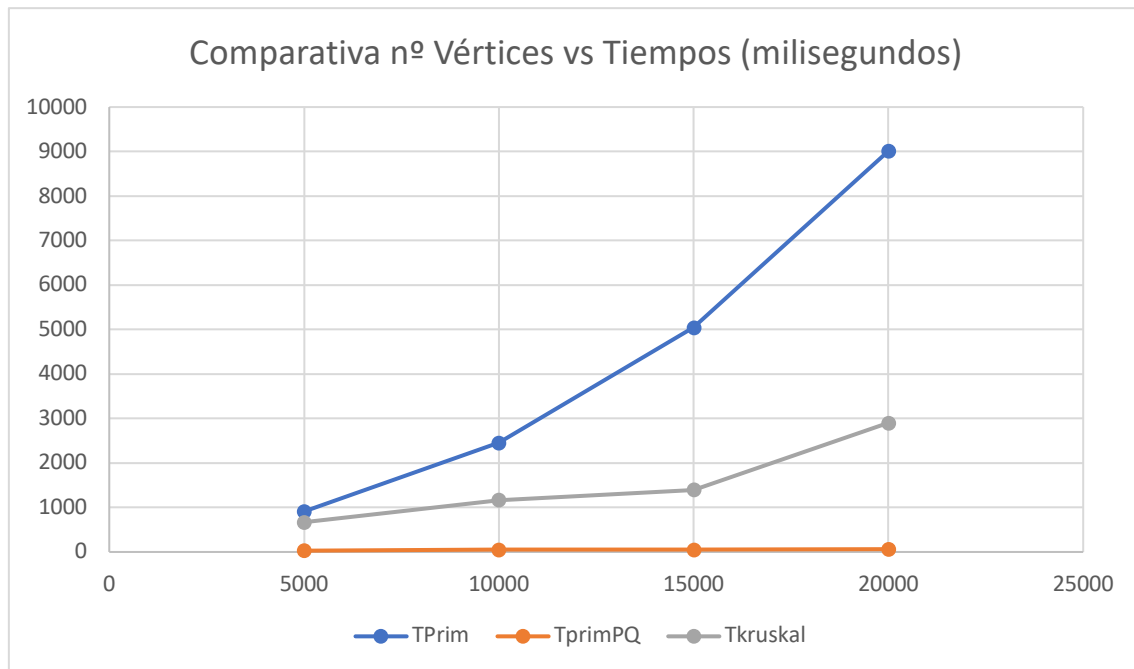
<terminated> GeneradorGrafos [Java Application] /Users/rafaexpósito/.p2/pool/plugins/org.eclipse.justj.op
Tiempo de ejecución: 106880959 nanosegundos. || 107 milisegundos.

Una vez ejecutados los algoritmos para grafos de 5000, 10000, 15000 y 20000 vértices, tal y como se sugiere en el enunciado de la práctica, obtenemos los siguientes resultados:

N	TPrim	TprimPQ	Tkruskal
5000	910	29	669
10000	2457	48	1167
15000	5042	52	1393
20000	9012	63	2902

Nota: Tiempos en milisegundos

Pudiendo ver estos resultados plasmados en la siguiente gráfica:



Como se describió en el análisis teórico, el algoritmo de Prim, con mayor orden de complejidad, es el que peores resultados ha obtenido.

La mejora más significativa la vemos al usar una cola de prioridad en el algoritmo de Prim, puesto que nos ahorramos ir comprobando elemento a elemento.

5. Fuentes Bibliográficas

[1] Algoritmos voraces, facultad de informática, UPC, disponible en <https://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf>

[2] Análisis y diseño de algoritmos Greedy, Universidad de Granada, disponible en <http://elvex.ugr.es/decsai/algorithms/slides/4%20greedy.pdf>

[3] Complejidad algorítmica, algoritmo de Kruskal, disponible en <https://sites.google.com/site/complejidadalgoritmicaes/kruskal>