PRÁCTICA 2 - ALGORITMOS GREEDY

Índice:

1. Estudio de la implementación

Para ello, lo que hemos realizado ha sido una explicación al estilo JavaDoc de todo el código que hemos empleado, Dicho código se puede encontrar en nuestro repositorio https://github.com/jlp717/ual_eda2_2022/tree/main/practica2/sources , estando explicado al máximo nivel de detalle con el fin de que sea fácilemente legible y entendible (además de diversos juegos de prueba/ JUnit que nos permite comprobar que nuestro código se ejecuta de manera correcta). A continuación, mostraremos el diagrama de clases resultante de nuestro programa:

2. Estudio teórico

Cómo bien sabemos, tenemos tres algoritmos, que corresponden al de Prim, al de Prim con una PQ, y el de Kruskal. Para ello, lo que haremos será calcular el tiempo de ejecución variando los tamaños de entrada (tanto vértices cómo aristas). Para ello, una primera prueba que haremos será probar los tamaños de entrada que vienen en *graphPrimKruskal.txt* , es decir, un total de 6 aristas y 10 vértices/caminos. Posteriormente, con un generador que hemos implementado, introduciendo el nº de aristas nos devuelve un .txt de n aristas (las que hayamos especificado), y lo hemos ido guardando en *Prueba01.txt* hasta *Prueba08.txt*. Los tiempos obtenidos en el archivo que se nos ha facilitado están señalados en un gris oscuro, mientras que los generados por nosotros están en un color gris claro. Finalmente, los resultados han sido:

(1) Tiempos de creación con el generador:

Nº vértices	Tiempo creación generador (ms)
10	1
25	2
50	3
100	5
250	14
500	35
1000	99
2500	389
5000	1494
10000	19383
20000	181755

Para la generación de dichos tiempos lo que hemos hecho ha sido irnos a nuestra clase java *Generador* y dentro de este main, lo que hacemos es llamar al método *grafoNoDirigido* , cuyo primer parámetro de entrada del método corresponde al nº vértices y el segundo el nombre del archivo con el que vamos a guardar dicho archivo .txt. Aunque viene todo esta clase mejor explicada al estilo JavoDoc en nuestro repositorio, el generador de archivos permite una mayor capacidad, no obstante, los ficheros que comparamos tienen limitadas la cantidad de aristas al doble (de vértices), para así evitar el error de heap capacity (en vértices más abundantes) al usar diferentes terminales.

```
public static void main(String[] args) throws FileNotFoundException {
    long startNano = System.nanoTime();
    long startMili = System.currentTimeMillis();
    grafoNoDirigido(10,"Prueba01.txt");
    long endNano = System.nanoTime();
    long endMili = System.currentTimeMillis();
    System.out.println("Tiempo de ejecución: " + (endNano-startNano) + " nanosegundos. || "
}
```

(2) Tiempos de cada algoritmo

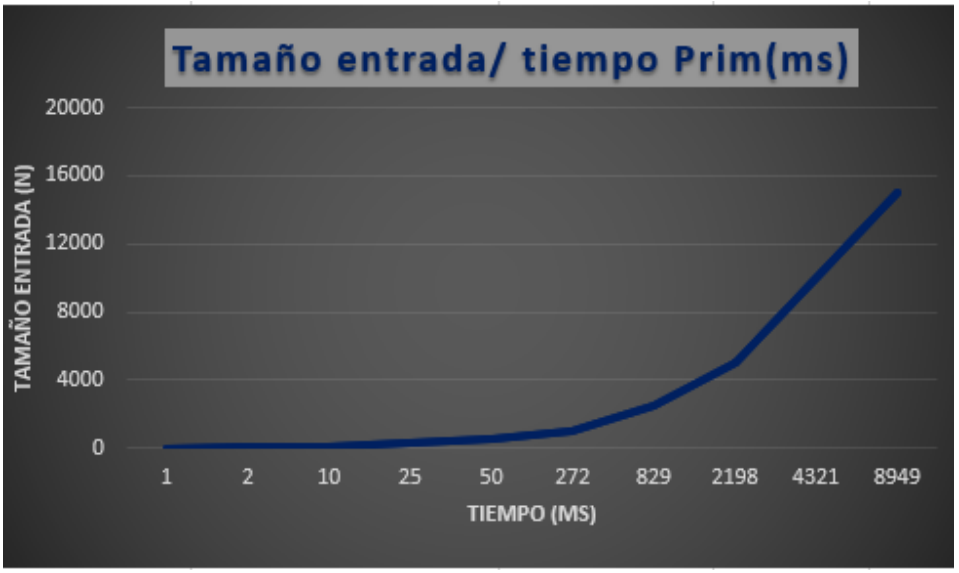
Posteriormente, y ya cómo último paso, para obtener los tiempos que hemos mostrado anteriormente en la tabla los obtenemos del main *MST:*

```java
public static void main(String[] args) {
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\graphPrimKruskal.txt", "A");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba01.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba02.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba03.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba04.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba05.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba06.txt", "0");
    MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba07.txt", "0");
    //MST tree = new MST("C:\\WORKSPACES\\EDA_2022\\practica_2\\src\\Prueba08.txt", "0");

    List<Arista> resultado = tree.prim();
    System.out.println(resultado);
    ContarCostes costes = new ContarCostes(resultado);
    System.out.println(costes);

    List<Arista> resultado1 = tree.primPQ();
    System.out.println(resultado1);
    ContarCostes costes1 = new ContarCostes(resultado1);
    System.out.println(costes1);

    List<Arista> resultado2 = tree.kruskal();
    System.out.println(resultado2);
    ContarCostes costes2 = new ContarCostes(resultado2);
    System.out.println(costes2);
}
```

Y cómo dijimos anteriormente, hemos ido guardando los archivos desde *Prueba01.txt* hasta *Prueba08.txt* . Con lo cual, vamos ejecutando y abriendo cada uno de los archivos de texto, obtendremos los tiempos siguientes:

Prim:

Nº vértices	Tiempo Prim(ms)
6	1
10	1
25	1
50	2
100	2
250	10
500	25
1000	50
2500	272
5000	829
10000	2198
15000	4321
20000	8949

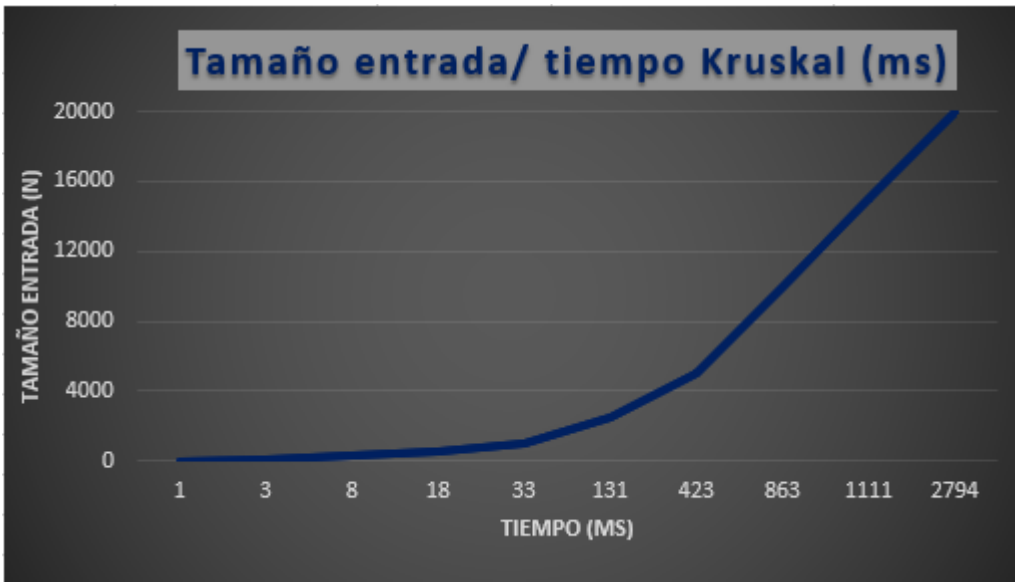


Prim con PQ:

Nº vértices	Tiempo PrimPQ(ms)
6	1
10	1
25	1
50	1
100	2
250	3
500	5
1000	8
2500	16
5000	23
10000	35
15000	25
20000	47

Kruskal:

Nº vértices	Tiempo Kruskal(ms)
6	1
10	1
25	2
50	2
100	3
250	8
500	18
1000	33
2500	131
5000	423
10000	863
15000	1111
20000	2794



Se ha podido observar la mejora MUY significativa que se ha producido entre cada uno de los algoritmos. Por ejemplo, para el caso de Prim, es el más costoso de todos, y es porque en cada pasada tiene que ir comprobando elemento por elemento (en este caso nodo por nodo). Sin embargo, al tener una PQ, este proceso se aligera mucho y el tiempo se ve drásticamente reducido

(1) ¿el resultado de la ejecución de cada algoritmo es único?. No tiene el porqué serlo, y de hecho ya lo hemos estado viendo que los resultados no son únicos, especialmente para tamaños de entrada bajos, puesto que son pocos los elementos que hay que comprobar, por lo que suelen ser de duración parecida. A partir de tamaños de entrada más altos, hemos comentado anteriormente que el Prim con una PQ es mucho más eficiente

(2) ¿el resultado de la ejecución de los dos algoritmos debe ser el mismo?, ¿por qué?. Recapitulando con lo que comentamos anteriormente, no tienen porque ser el mismo resultado, pero en el caso de que sean pocos elementos (por lo general), suelen coincidir muchos tiempos de estos tres algoritmos.

(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?. Va a depender de cómo sea el grafo, pero podría darse el caso que por ejemplo quisiéramos ir de un nodo/vértices A hacia uno F y encontramos un camino único mínimo con el que llegar, pues sí encontraremos solución. Si por el contrario, hay múltiples soluciones que conducen al mismo camino, habrá varios caminos válidos, por lo que no podremos dar una solución clara. En resumen, si para ir de A a F tenemos más de un camino que pasa por el mismo nº de nodos (cómo el coste va a ser el mismo), no podremos dar solución, pero sí en caso contrario.

3.Estudio experimental

Primeramente, empezaremos analizando los tres algoritmos anteriormente presentados con los tres archivos de texto (.txt) que se nos han proporcionado. Los tiempos que obtenemos han sido:

(1) graphEDAland.txt

En este archivo de texto, los nodos son ciudades de España, por lo que el proceso es igual que el realizado anteriormente pero con nombre para los nodos. A continuación vamos a mostrar una salida de lo que obtenemos al ejecutar el main de nuestra clase MST:

```
Tiempo de ejecución para algoritmo Prim: 2321700 nanosegundos. || 2 milisegundos.
[Almeria --> Granada| peso=173.0 , Granada --> Jaen| peso=99.0 , Almeria --> Murcia|

Coste Total --> 4117.0

Tiempo de ejecución para algoritmo PrimPQ: 1572000 nanosegundos. || 2 milisegundos.
[Almeria --> Granada| peso=173.0 , Granada --> Jaen| peso=99.0 , Almeria --> Murcia|

Coste Total --> 4117.0

Tiempo de ejecución para algoritmo Kruskal: 2559600 nanosegundos. || 3 milisegundos.
[Albacete --> Murcia| peso=150.0 , Badajoz --> Huelva| peso=234.0 , Barcelona --> Za

Coste Total --> 4117.0
```

(2) graphEDAlandLargue.txt

Volvemos a realizar el mismo proceso pero esta con la ruta de este otro archivo de texto, y los resultados han sido:

```
Tiempo de ejecución para algoritmo Prim: 194093500 nanosegundos. || 193 milisegundos.
[1 --> 2| peso=15.0 , 2 --> 3| peso=12.0 , 2 --> 4| peso=12.0 , 4 --> 7| peso=16.0 , 7 --

Coste Total --> 17315.0

Tiempo de ejecución para algoritmo PrimPQ: 18527900 nanosegundos. || 19 milisegundos.
[1 --> 2| peso=15.0 , 2 --> 4| peso=12.0 , 2 --> 3| peso=12.0 , 4 --> 7| peso=16.0 , 7 --

Coste Total --> 17315.0

Tiempo de ejecución para algoritmo Kruskal: 141026800 nanosegundos. || 141 milisegundos.
[10 --> 18| peso=12.0 , 100 --> 102| peso=11.0 , 1000 --> 827| peso=20.0 , 1001 --> 831|

Coste Total --> 17315.0
```

Por último, cómo ya hicimos anteriormente, estos han sido los resultados obtenidos para cada uno de los algoritmos con tamaños 5000, 10000, 15000 y 20000, y volvemos a observar la enorme diferencia que hay de tiempos, incluyendo únicamente una estructura de cola de prioridad o PriorityQueue

Nº vértices	Tiempo Prim(ms)
5000	829
10000	2198
15000	4321
20000	8949

Nº vértices	Tiempo PrimPQ(ms)
5000	23
10000	35
15000	25
20000	47

Nº vértices	Tiempo Kruskal(ms)
5000	423
10000	863
15000	1111
20000	2794

En resumen, tanto para los archivos generados cómo para los de nuestra red EDAland, conseguimos encontrar los caminos mínimos pero sin duda, para trabajar con una carga importante de datos, nuestra mejor recomendación sería emplear el algoritmo de Prim con la cola de prioridad.