PRÁCTICA OPCIONAL MARP: ALGORITMO DE DIJKSTRA USANDO UN MONTÍCULO SESGADO

Pedro Simarro Guerra

2020/2021

Índice

1. Explicación del problema y estructura del código

En esta práctica se trata el problema de la búsqueda de caminos mínimos sobre un grafo usando el algoritmo de Dijkstra. En este caso se pedía implementar dicho algoritmo utilizando un montículo sesgado y cuya complejidad (vista en clase) es $O((m+n)\log n)$. La implementación se ha realizado en *Java* utilizando la siguiente estructura de clases:

- *Main.java*: esta es la clase principal del programa, la cual se encarga de ejecutar los distintos modos de ejecución de la aplicación.
- *Grafo.java*: esta clase define grafos no dirigidos valorados
- *MinHeap.java*: representa montículos sesgados usando árboles
- *Dijkstra.java*: clase de los objetos resolutores del algoritmo.

Además, se ha usado la librería *Commons CLI* de *Apache* para para el filtrado de las entradas. (https://commons.apache.org/proper/commons-cli/). Es necesario compilar el código con el jar correspondiente para poder usar la aplicación.

2. Uso de la aplicación

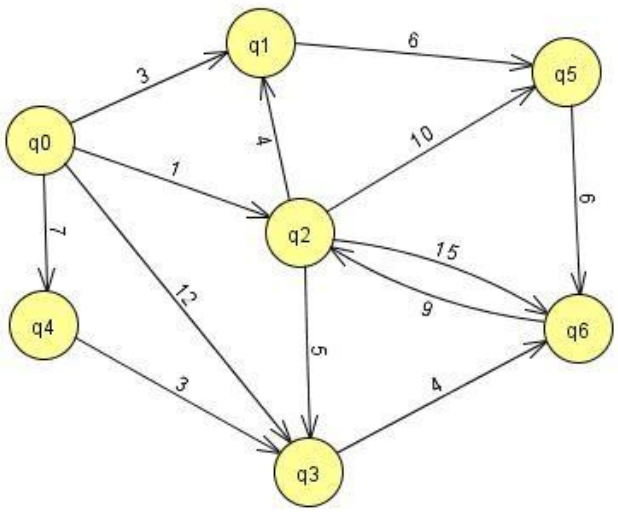
La aplicación tiene distintos modos de ejecución accesibles mediante línea de comandos. Son los siguientes:

- **Modo usando un grafo aleatorio** (opción -r): aquí se genera un grafo aleatorio que puede tener una densidad (usando -d) y un número de vértices (-v) dados. Se muestra dicho grafo mostrando las listas de adyacencia de los vértices y el resultado del algoritmo.
- **Modo usado para generar gráficas de resultados** (opción -g): este modo genera grafos aleatorios con una densidad fija (opción -d) o un número de vértices fijo (-v). Se guardan los resultados de los tiempos de ejecución del algoritmo de Dijkstra en el archivo indicado en la opción -o. Este modo no muestra los grafos debido al tamaño de estos.
- **Modo para ejecutar un grafo definido en un archivo** (opción -i): genera un grafo definido en el archivo que se pasa como argumento, y muestra los resultados de la misma manera que en el primer modo.

3. Ejemplos de ejecución con casos pequeños

A continuación, se muestra la salida por pantalla de la ejecución de la aplicación sobre los siguientes grafos.

- Grafo número 1:

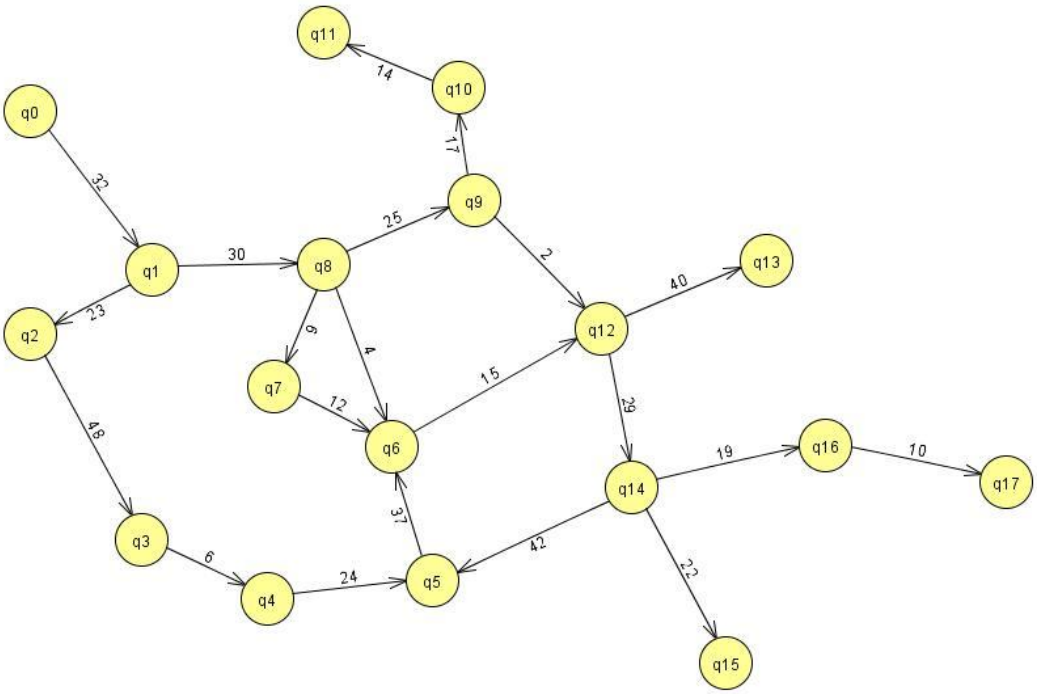


```
(0) : (1, 3,000000), (2, 1,000000), (3, 12,000000), (4, 7,000000),
(1) : (5, 6,000000),
(2) : (1, 4,000000), (5, 10,000000), (6, 15,000000), (3, 5,000000),
(3) : (6, 4,000000),
(4) : (3, 3,000000),
(5) : (6, 6,000000),
(6) : (2, 9,000000),


Numero de aristas = 13
Costes mínimos:
   (0) : 0.0
   (1) : 3.0
   (2) : 1.0
   (3) : 6.0
   (4) : 7.0
   (5) : 9.0
   (6) : 10.0
----
Caminos:
 (1): 0,1
 (2): 0,2
 (3): 0,2,3
 (4): 0,4
 (5): 0,1,5
 (6): 0,2,3,6

Tiempo transcurrido: 0,683 milisegundos
```

- Grafo número 2:



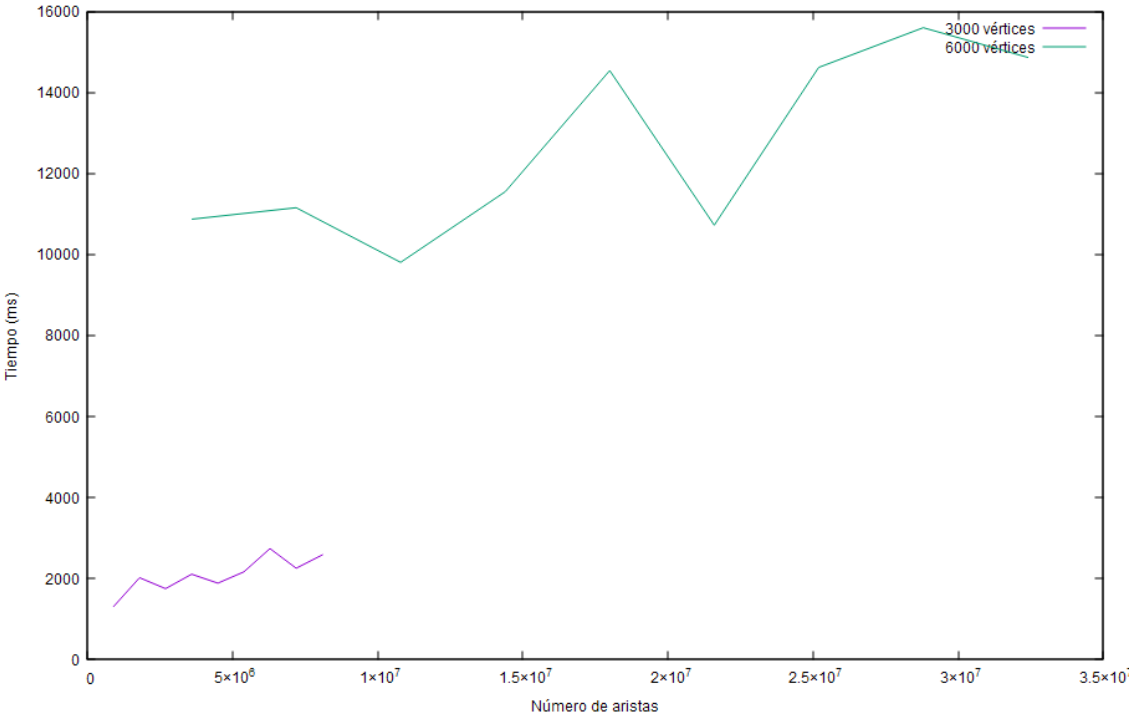
```
(0) : (1, 32,000000),
(1) : (2, 23,000000), (8, 30,000000),
(2) : (3, 48,000000),
(3) : (4, 6,000000),
(4) : (5, 24,000000),
(5) : (6, 37,000000),
(6) : (12, 15,000000),
(7) : (6, 12,000000),
(8) : (7, 6,000000), (6, 4,000000), (9, 25,000000),
(9) : (10, 17,000000), (12, 2,000000),
(10) : (11, 14,000000),
(11) :
(12) : (13, 40,000000), (14, 29,000000),
(13) :
(14) : (5, 42,000000), (15, 22,000000), (16, 19,000000),
(15) :
(16) : (17, 10,000000),
(17) :


Numero de aristas = 21
Costes mínimos:
   (0) : 0.0
   (1) : 32.0
   (2) : 55.0
   (3) : 103.0
   (4) : 109.0
   (5) : 133.0
   (6) : 66.0
   (7) : 68.0
   (8) : 62.0
   (9) : 87.0
   (10) : 104.0
   (11) : 118.0
   (12) : 81.0
   (13) : 121.0
   (14) : 110.0
   (15) : 132.0
   (16) : 129.0
   (17) : 139.0
----
Caminos:
 (1): 0,1
 (2): 0,1,2
 (3): 0,1,2,3
 (4): 0,1,2,3,4
 (5): 0,1,2,3,4,5
 (6): 0,1,8,6
 (7): 0,1,8,7
 (8): 0,1,8
 (9): 0,1,8,9
 (10): 0,1,8,9,10
 (11): 0,1,8,9,10,11
 (12): 0,1,8,6,12
 (13): 0,1,8,6,12,13
 (14): 0,1,8,6,12,14
 (15): 0,1,8,6,12,14,15
 (16): 0,1,8,6,12,14,16
 (17): 0,1,8,6,12,14,16,17

Tiempo transcurrido: 0,809 milisegundos
```

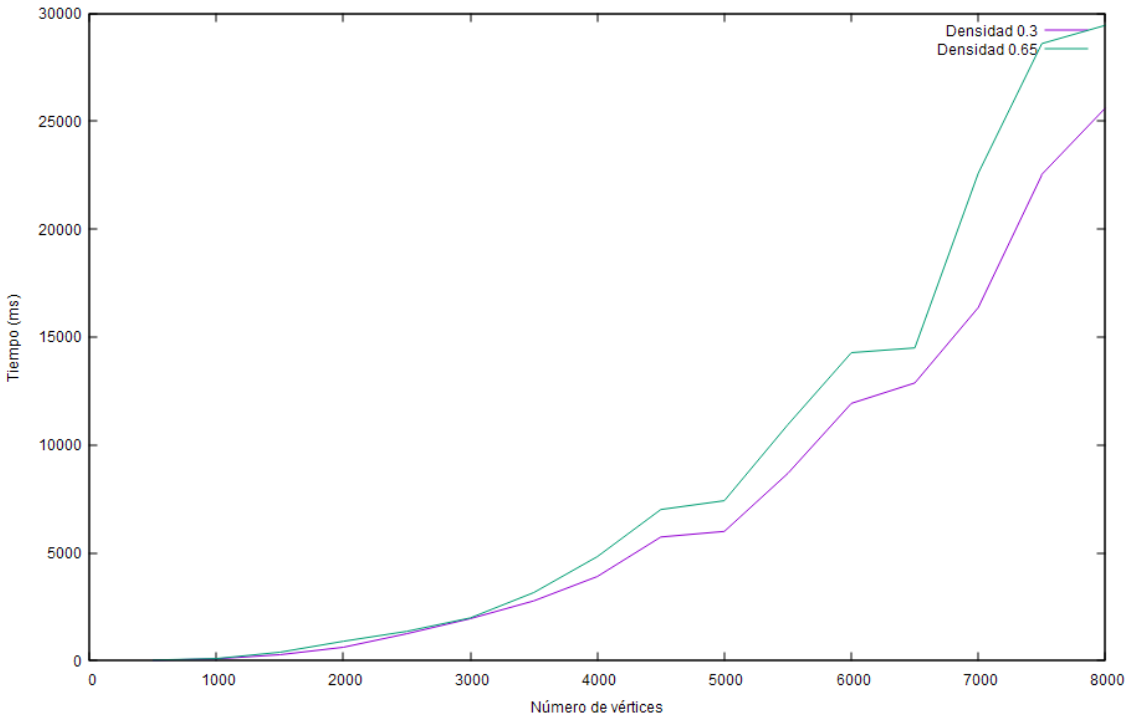
4. Gráficas para casos voluminosos y discusión de los resultados.

- Resultados para grafos con número de vértices fijos y densidad variable:



Esta gráfica muestra la importancia del número de vértices a la hora de ejecutar el algoritmo. Como se puede observar a pesar de que, en las primeras ejecuciones del algoritmo sobre el grafo de 6000 vértices, el número de aristas es igual que en el caso del grafo de 3000, el tiempo de ejecución se incrementa sustancialmente. Las gráficas acaban cuando el grafo llega prácticamente a su conexión completa.

- Resultados para grafos con densidad fija y número de vértices variable:



En este caso, aumentar la densidad de los grafos no genera grandes cambios en el tiempo de ejecución, dejando claro, una vez más, que el tiempo de ejecución depende sobre todo del número de vértices. Lo cual tiene sentido por la complejidad $O((m+n)\log n)$. El número de vértices aumenta gradualmente en esta ejecución, sin embargo, la proporción de aristas para los grafos es constante, fijando de alguna manera el número de aristas en términos del número de vértices. Esta densidad viene dada por

$$Densidad(G) = \frac{|A|}{|V| \cdot (|V|-1)}$$

siendo A el conjunto de aristas del grafo y V el conjunto de vértice.

5. Discusión de la complejidad el algoritmo

```
public void ejecuta(){
        //Se inicializa las variables del problema
        Iterator<Grafo.InfoArista> it;
        int N = this.grafo.getN();
        MinHeap M = new MinHeap(N);

        //Se inicializa el resultado del vertice 0 (inicio)
        costeMin[0] = 0f;
        predecesor[0] = null;

        //Se inicializan los resultados para el resto de vertices
        for(int i = 1; i < N; ++i){
            costeMin[i] = Float.MAX_VALUE;
            predecesor[i] = -1;
        }

        //Las aristas directas del vertice 0 a sus adyacentes,
        // actualizan los resultados de estos
        it = this.grafo.itAdyacentes(0);
        while(it.hasNext()){
            Grafo.InfoArista a = it.next();
            costeMin[a.getDest()] = a.getVal();
            predecesor[a.getDest()] = 0;
            M.insertar(a.getVal(), a.getDest());
        }

        while(!M.vacio()){
            Integer elegido = M.peek();
            M.borrar();

            if(elegido != null){
                it = this.grafo.itAdyacentes(elegido);
                while(it != null && it.hasNext()){
                    Grafo.InfoArista arista_adj = it.next();
                    int adj = arista_adj.getDest();
                    float coste = costeMin[elegido] + arista_adj.getVal();

                    if(coste < costeMin[adj]){
                        costeMin[adj] = coste;
                        predecesor[adj] = elegido;
                        modifica(M, adj, coste);
                    }
                }
            }

        }

    }
```

Teniendo en cuenta la implementación del algoritmo veamos la complejidad del mismo paso a paso:

- El coste de creación del montículo es logarítmico en el orden de los vértices.
- Los vectores de los resultados tienen un coste de creación lineal en el orden los vértices.
- La inserción de los vértices adyacentes al vértice 0 está es en el peor de los casos $n \cdot log\ n$ si este estuviera conectado a todos.
- En el bucle principal, mirar si el montículo está vacío es constante; se realizan como mucho n borrados y en el peor de los casos se hacen m modificaciones al montículo, tantas como aristas haya.

Si sumamos todos los costes, y asumiendo la complejidad de las operaciones de un montículo sesgado el coste total en el peor de los casos es $n + 3(n \cdot \log n) + (m \cdot \log n)$, que está en el orden de $O(m + n(\log n))$, coste propuesto en clase para esta implementación si obviamos el coste de las inicializaciones.