rsanchez628@alumno.uned.es

Cuestiones teóricas de la práctica

1.- Coste temporal y espacial del algoritmo.

- Tipos de datos.
    - **Grafo dirigido**.
        - Matriz de enteros. El coste de ir del nodo **i** al nodo **j** se encuentra en la fila **i**, columna **j**.
        - Su implementación consta de un array en el que sus elementos son arrays de números enteros.
        - Espacio en memoria. Tomando 4 bytes como espacio en memoria ocupado por un número entero y la matriz asociada al grafo dirigido como una matriz cuadrada nxn, siendo n el número de nodos del grafo, el espacio en memoria del dato es $4 \times n^2$ bytes.
    - **Matriz de Floyd**. Se utilizan dos matrices para su implementación.
        1. Mínimo recorrido.
            - Matriz de enteros. El coste mínimo de ir del nodo **i** al nodo **j** se encuentra en la fila **i**, columna **j**.
            - Su implementación consta de un array en el que sus elementos son arrays de números enteros.
            - Se calcula mediante el algoritmo de Floyd.
            - Espacio en memoria. Tomando 4 bytes como espacio en memoria ocupado por un número entero y la matriz asociada al grafo dirigido como una matriz cuadrada nxn siendo, n el número de nodos del grafo, el espacio en memoria del dato es $4 \times n^2$ bytes.
        2. Ruta mínima.
            - Matriz de enteros. Guarda el último nodo que se visita antes de llegar al nodo **j** desde el nodo **i.**
            - Su implementación consta de un array en el que sus elementos son arrays de números enteros.
            - Se calcula mediante el algoritmo de Floyd.
            - Espacio en memoria. Tomando 4 bytes como espacio en memoria ocupado por un número entero y la matriz asociada al grafo dirigido como una matriz cuadrada nxn siendo, n el número de nodos del grafo, el espacio en memoria del dato es $4 \times n^2$ bytes.

Coste temporal.

El **algoritmo de Floyd**

```
/*
 * Genera la matriz con el valor más corto entre dos nodos
 * así como la información para generar el camino
 */
public FloydMatrix(DirectedGraph graph, boolean steps) {

    this.graph = graph;
    rm = new RoutesMatrix(graph.dim);
    Integer ancient;

    for (int k=1; k<=graph.dim; k++) {
        for(int i=1; i<=graph.dim; i++) {
            for(int j=1; j<=graph.dim; j++) {
                ancient = this.graph.getWeight(i, j);
                this.graph.setWeight(i, j, this.graph.floydSelector(i, j, k));
                if(ancient!=this.graph.getWeight(i, j) && !(this.graph.getWeight(i, j).equals(DirectedGraph.getMax()))) {
                    rm.add(i, j, k);
                }
            }
        }
        if(steps) {
            System.out.println("k= "+k);
            this.graph.show();
            System.out.println();
        }
    }
    rm.setIntermediateNodes();
}
```

En la inspección del código puede verse como se calcula k veces el camino mínimo para cada nodo de la matriz.

También se puede ver cómo el número de veces k que se hace el cálculo es función de la dimensión de la matriz asociada al grafo dirigido, es decir, función del número de nodos.

```
/*
 * Devuelve los nodos de la matriz k-ésima que usa el método de
 * Floyd
 * @param:  k -> matriz del método de Floyd
 *          i,j -> origen y destino
 * @return: mínima distancia entre los nodos i, y para la matirz k
 */
public Integer floydSelector(Integer i, Integer j, Integer k) {

    Integer compar1 = this.turnVoid(this.getWeight(i, j));
    Integer sum1 = this.turnVoid(this.getWeight(i, k));
    Integer sum2 = this.turnVoid(this.getWeight(k, j));
    Integer compar2 = sum1 + sum2;

    return Math.min(compar1, compar2);
}


/*
 * Se le pasa por parámetro un número y devuelve infinito si el número
 * que se pasó por parámetro era -1
 */
private Integer turnVoid(Integer n) {
    boolean condition = n==-1;
    return (condition)?max:n;
}
```

Así mismo, el cálculo del camino mínimo cada una de las k veces se compone de una serie de operaciones lógicas, aritméticas y asignaciones de coste temporal constante O(1).

Por lo que se hace un cálculo de coste temporal constante O(1), nxn, número de nodos del grafo, n veces → **$O(n^3)$, siendo n el número de nodos del grafo dirigido.**

Para el **cálculo de la ruta de menor coste** se emplea la función recurrente:

```
/*
 * Imprimir recursivo. Método del libro
 */
private List<Integer> generateIN(List<Integer> nodes, Integer i, Integer j) {

    Integer k = m.get(i).get(j).isEmpty();

    if(!(k.equals(getEmpty()))) {
        generateIN(nodes, i, k-1);
        nodes.add(k);
        generateIN(nodes, k-1, j);
    }

    return nodes;

}
```

El coste de la función recurrente depende de cuántas descomposiciones se realicen y del tipo de decrecimiento de las sucesivas llamadas recursivas a los subproblemas.

En este caso, el problema decrece de forma aritmética:

$T(n) = aT(n-b) + cn^k$

a ≡ número de subproblemas en los que se descompone el problema = **1**

b ≡ factor de reducción del tamaño del problema = **1**

cn^k ≡ coste de la función de combinación de las soluciones parciales = **1 → k = 0**

Si **a = 1** → $\theta(n^{k+1})$ con k = 0 → **θ(n)**

Siendo n el número de nodos del grafo dirigido.

El coste de calcularse la ruta mínima para el conjunto del grafo dirigido será

```
/*
 * Cálculo de la ruta mínima para cada ruta posible del
 * grafo dirigido
 */
private void setIntermediateNodes() {
    List<Integer> nodes = new LinkedList<Integer>();
    for(int i=0; i<dim; i++) {
        for(int j=0; j<dim; j++) {

            nodes = generateIN(nodes,i,j);
            this.in.setIntNodes(i+1, j+1, nodes);
            nodes = new LinkedList<Integer>();
        }
    }

}
```

n ≡ nodos origen x n ≡ nodos destino → n^2 ≡ rutas → **O(n^3)** siendo n el número de nodos del grafo.

El algoritmo es de la forma:

1. Cálculo de los pesos mínimos entre nodos. **Algoritmo de Floyd → O(n^3)**
2. Cálculo de la ruta mínima. **Función recursiva →O(n^3)**

Coste temporal: O(n^3)

NOTA 1: aunque en la función recursiva hay dos llamadas a sí misma, la propia función recursiva, una de ellas –según el nodo de origen una u otra- siempre tendrá como resultado el caso base, será de coste constante. A efectos del cálculo del coste temporal, sólo hay una llamada recursiva.

2.- Esquemas que pueden resolver el problema. Idoneidad.

Dentro de los esquemas de algoritmos voraces, el algoritmo de Dijkstrat puede determinar la distancia mínima entre dos nodos de un grafo dirigido como el planteado en el ejercicio.

El **coste temporal** del algoritmo de Dijkstra si:

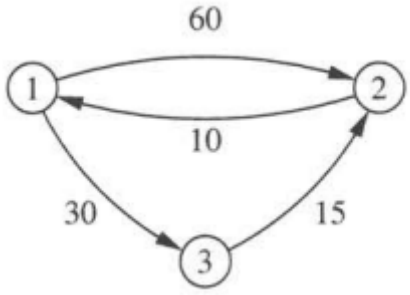
- el grafo es disperso.
- el número de aristas es pequeño y cercano al número de nodos.
- la estructura de datos que se usa en su implementación es un montículo.

Sería **O(n^2 log n)**, con lo que en las condiciones requeridas mejoraría levemente al algoritmo de Floyd usado.

Sea cual sea la implementación que se haga del algoritmo de Dijkstrat, y el tipo de grafo sobre el que se haga el cálculo, se debe almacenar el mismo número de datos que se guardaron con el algoritmo de Floyd por lo que, en cuanto a coste espacial, no hay diferencia entre ambos algoritmos.

Ejemplos de ejecución para distintos tamaños del problema

- No hay grafo / fichero vacío. El programa no se finaliza su ejecución sin dar errores.
- Grafo n=3



Datos de entrada.

0	60	30
10	0	-
-	15	0

Datos de salida.

[1, 1]: 0

[1, 2]: 3: 45

[1, 3]: 30

[2, 1]: 10

[2, 2]: 0

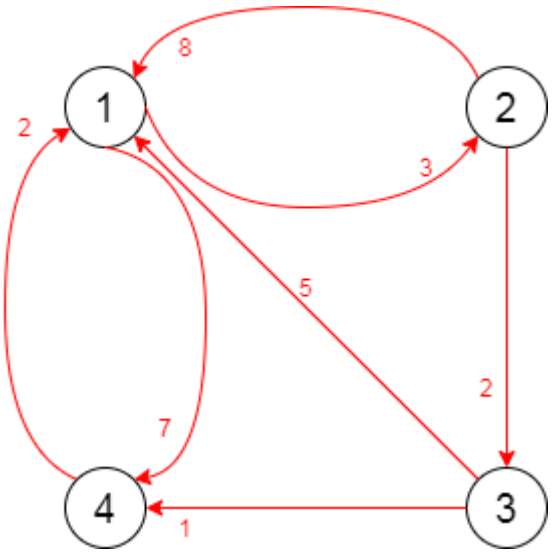
[2, 3]: 1: 40

[3, 1]: 2: 25

[3, 2]: 15

[3, 3]: 0

- Grafo n=4



Datos de entrada.

0	3	-	7
8	0	2	-
5	-	0	1
2	-	-	0

Datos de salida

[1, 1]: 0

[1, 2]: 3

[1, 3]: 2: 5

[1, 4]: 2,3: 6

[2, 1]: 3,4: 5

[2, 2]: 0

[2, 3]: 2

[2, 4]: 3: 3

[3, 1]: 4: 3

[3, 2]: 4,1: 6

[3, 3]: 0

[3, 4]: 1

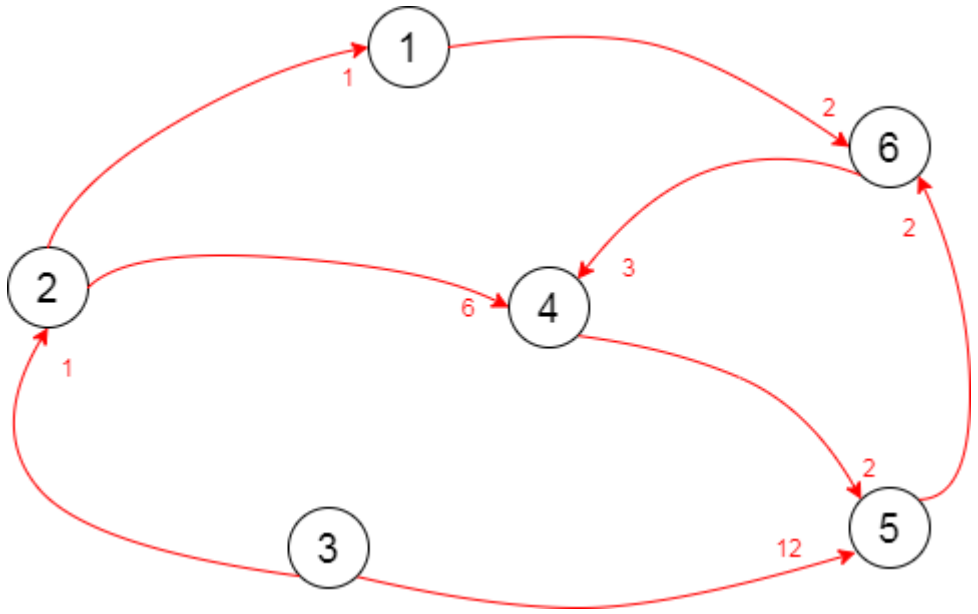
[4, 1]: 2

[4, 2]: 1: 5

[4, 3]: 1,2: 7

[4, 4]: 0

- Grafo n=5



Datos de entrada

0	-	-	-	-	2
1	0	-	6	-	-
-	1	0	-	12	-
-	-	-	0	2	-
-	-	-	-	0	2
-	-	-	3	-	0

Datos de salida

[1, 1]: 0

[1, 2]: ∞

[1, 3]: ∞

[1, 4]: 6: 5

[1, 5]: 6,4: 7

[1, 6]: 2

[2, 1]: 1

[2, 2]: 0

[2, 3]: ∞

[2, 4]: 6

[2, 5]: 4: 8

[2, 6]: 1: 3

[3, 1]: 2: 2

[3, 2]: 1

[3, 3]: 0

[3, 4]: 2: 7

[3, 5]: 2,4: 9

[3, 6]: 2,1: 4

[4, 1]: ∞

[4, 2]: ∞

[4, 3]: ∞

[4, 4]: 0

[4, 5]: 2

[4, 6]: 5: 4

[5, 1]: ∞

[5, 2]: ∞

[5, 3]: ∞

[5, 4]: 6: 5

[5, 5]: 0

[5, 6]: 2

[6, 1]: ∞

[6, 2]: ∞

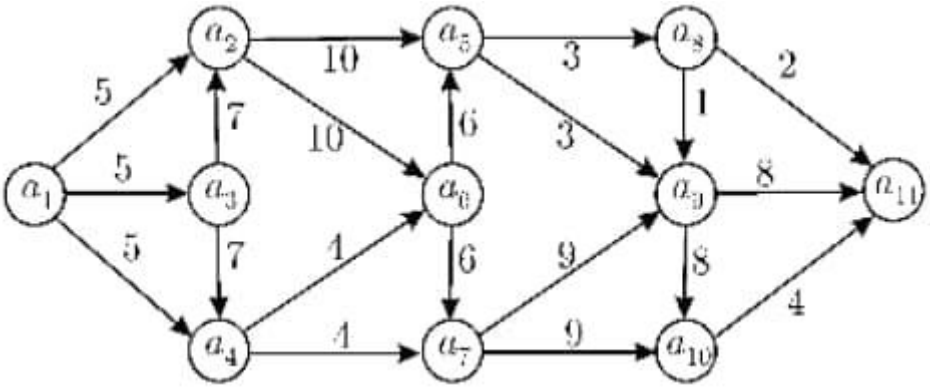
[6, 3]: ∞

[6, 4]: 3

[6, 5]: 4: 5

[6, 6]: 0

- Grafo n=11



Datos de entrada

0	5	5	5	-	-	-	-	-	-	-
-	0	-	-	10	10	-	-	-	-	-
-	7	0	7	-	-	-	-	-	-	-
-	-	-	0	-	4	4	-	-	-	-
-	-	-	-	0	-	-	3	3	-	-
-	-	-	-	6	0	6	-	-	-	-
-	-	-	-	-	-	0	-	9	9	-
-	-	-	-	-	-	-	0	1	-	2
-	-	-	-	-	-	-	-	0	8	8
-	-	-	-	-	-	-	-	-	0	4
-	-	-	-	-	-	-	-	-	-	0

Datos de salida

[1, 1]: 0

[1, 2]: 5

[1, 3]: 5

[1, 4]: 5

[1, 5]: 2: 15

[1, 6]: 4: 9

[1, 7]: 4: 9

[1, 8]: 2,5: 18

[1, 9]: 2,5: 18

[1, 10]: 4,7: 18

[1, 11]: 2,5,8: 20

[2, 1]: ∞

[2, 2]: 0

[2, 3]: ∞

[2, 4]: ∞

[2, 5]: 10

[2, 6]: 10

[2, 7]: 6: 16

[2, 8]: 5: 13

[2, 9]: 5: 13

[2, 10]: 5,9: 21

[2, 11]: 5,8: 15

[3, 1]: ∞

[3, 2]: 7

[3, 3]: 0

[3, 4]: 7

[3, 5]: 2: 17

[3, 6]: 4: 11

[3, 7]: 4: 11

[3, 8]: 2,5: 20

[3, 9]: 2,5: 20

[3, 10]: 4,7: 20

[3, 11]: 2,5,8: 22

[4, 1]: ∞

[4, 2]: ∞

[4, 3]: ∞

[4, 4]: 0

[4, 5]: 6: 10

[4, 6]: 4

[4, 7]: 4

[4, 8]: 6,5: 13

[4, 9]: 6,5: 13

[4, 10]: 7: 13

[4, 11]: 6,5,8: 15

[5, 1]: ∞

[5, 2]: ∞

[5, 3]: ∞

[5, 4]: ∞

[5, 5]: 0

[5, 6]: ∞

[5, 7]: ∞

[5, 8]: 3

[5, 9]: 3

[5, 10]: 9: 11

[5, 11]: 8: 5

[6, 1]: ∞

[6, 2]: ∞

[6, 3]: ∞

[6, 4]: ∞

[6, 5]: 6

[6, 6]: 0

[6, 7]: 6

[6, 8]: 5: 9

[6, 9]: 5: 9

[6, 10]: 7: 15

[6, 11]: 5,8: 11

[7, 1]: ∞

[7, 2]: ∞

[7, 3]: ∞

[7, 4]: ∞

[7, 5]: ∞

[7, 6]: ∞

[7, 7]: 0

[7, 8]: ∞

[7, 9]: 9

[7, 10]: 9

[7, 11]: 10: 13

[8, 1]: ∞

[8, 2]: ∞

[8, 3]: ∞

[8, 4]: ∞

[8, 5]: ∞

[8, 6]: ∞

[8, 7]: ∞

[8, 8]: 0

[8, 9]: 1

[8, 10]: 9: 9

[8, 11]: 2

[9, 1]: ∞

[9, 2]: ∞

[9, 3]: ∞

[9, 4]: ∞

[9, 5]: ∞

[9, 6]: ∞

[9, 7]: ∞

[9, 8]: ∞

[9, 9]: 0

[9, 10]: 8

[9, 11]: 8

[10, 1]: ∞

[10, 2]: ∞

[10, 3]: ∞

[10, 4]: ∞

[10, 5]: ∞

[10, 6]: ∞

[10, 7]: ∞

[10, 8]: ∞

[10, 9]: ∞

[10, 10]: 0

[10, 11]: 4

[11, 1]: ∞

[11, 2]: ∞

[11, 3]: ∞

[11, 4]: ∞

[11, 5]: ∞

[11, 6]: ∞

[11, 7]: ∞

[11, 8]: ∞

[11, 9]: ∞

[11, 10]: ∞

[11, 11]: 0