Autores: *Germán Ruano García, Francisco Javier Marqués Gaona y David Subires Parra*

Ejercicio 0 Tema 4

Ejercicio 0.

0. Implementar el esquema algorítmico de Floyd y las opciones alternativas (fuerza bruta).

Se ha de:

- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba generados aleatoriamente (dado el tamaño del problema).
- Almacenar juegos de prueba, resultados y tiempos de ejecución.
- Comparar los resultados de fuerza bruta, de Dijkstra sobre todos los nodos (ejercicio 0 del tema anterior) y Floyd.
- Analizar la eficiencia obtenida empíricamente frente a la teórica.

Esquemas algorítmicos

Fuerza Bruta

```
/* Suponemos que la función pesoArista[i][j] devuelve el coste del camino que va de i a j
   (infinito si no existe).
   También suponemos que n es el número de vértices y pesoArista[i,i]) = 0
   Recorridos es una matriz de ArrayList para almacenar el camino del vértice i al vértice j.
  El método getVecinos(i) devuelve todos los vértices que son vecinos de i
 */


fuerzaBruta(caminos, recorridos)
para todos i < n hacer
        para todos j < n hacer
                peso = 0
                si i == j hacer
                        caminos[i][j] = peso
                        recorridos[i][j] = null
                sino hacer
                        si pesoArista[i][j] es distinto Infinito hacer
                                peso += pesoArista[i][j]
                                camino.añadir(i) camino.añadir(j)
                                caminos[i][j] = peso
                                recorridos[i][j] = camino
                        sino hacer
                                buscarCamino(i,j,peso,camino,caminos,recorridos)
                        fsi
                fsi
        fin para
fin para
```

```
buscarCamino(desde, hasta, peso, camino, caminos, recorridos)


vecinos = getVecinos(from)
si vecinos contiene hasta hacer
        peso += pesoArista[desde][hasta]
        camino.añadir(desde) camino.añadir(hasta)
        si caminos[camino.getPrimer][hasta] > peso hacer
                caminos[camino.getPrimer][hasta] = peso
                recorridos[camino.getPrimer][hasta] =  camino
        fsi
sino hacer
        para v ∈ vecinos hacer
                si camino no contiene v hacer
                        caminoTemp = camino.clonar()
                        caminoTemp.añadir(desde)
                        buscarCamino(v, hasta, peso + pesoArista[desde][v], caminoTemp,
                        camino, recorridos )
                fsi
        fin para
fsi
```

DIJKSTRA

```
DIJKSTRA (Origen u, distancia D, padre P)
    para v ∈ [D] hacer
        distancia[v] = INFINITO
        padre[v] = -1
    distancia[u] = 0
    padre[u] = -1
    adicionar (cola, (u, distancia[u]))
    mientras que cola no es vacía hacer
        n = extraer_mínimo(cola)
        para todos v ∈ adyacencia[n] hacer
            si distancia[v] > distancia[n] + peso (n, v) hacer
                distancia[v] = distancia[n] + peso (n, v)
                padre[v] =n
                adicionar(cola,(v, distancia[v]))
```

FLOYD

```
/* Suponemos que la función pesoArista[i][j] devuelve el coste del camino que va de i a j
   (infinito si no existe).
   También suponemos que n es el número de vértices y pesoArista[i,i]) = 0
 */

 int camino[][];
 /* Una matriz bidimensional. En cada paso del algoritmo, camino[i][j] es el camino mínimo
    de i hasta j usando valores intermedios de (1..k-1). Cada camino[i][j] es inicializado a

 */
```

```
Floyd (camino, padre)
para todos k < n hacer
        para todos i < n hacer
                para todos j < n hacer
                        si   camino[i][j]  >  camino[i][k]+camino[k][j]
                                d[i][j] = camino[i][k]+camino[k][j]
                                padre[i][j] = k
                        fsi
                fin para
        fin para
fin para
```

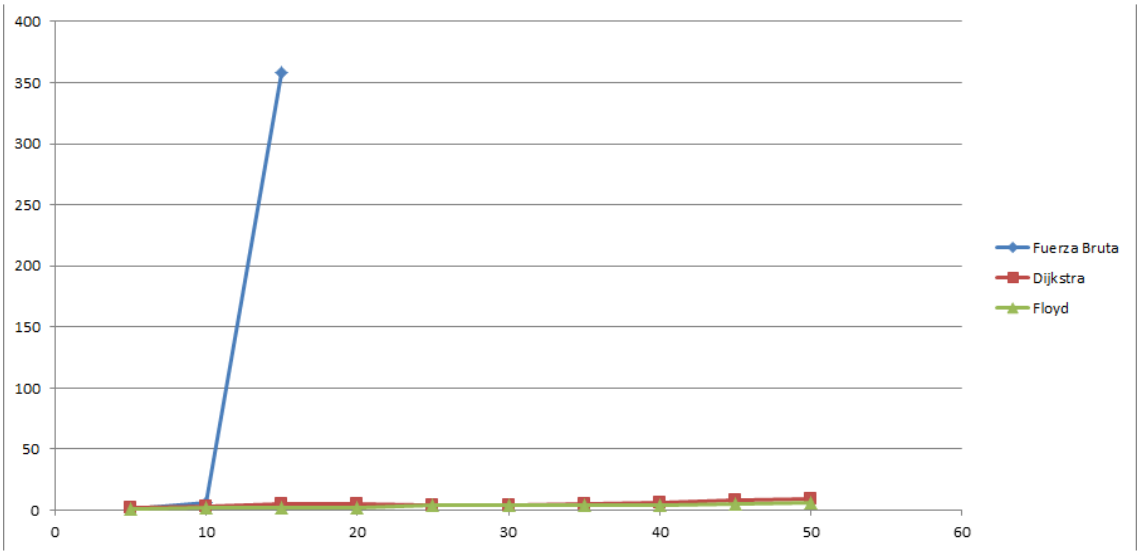
Selección de estructuras de datos adecuadas

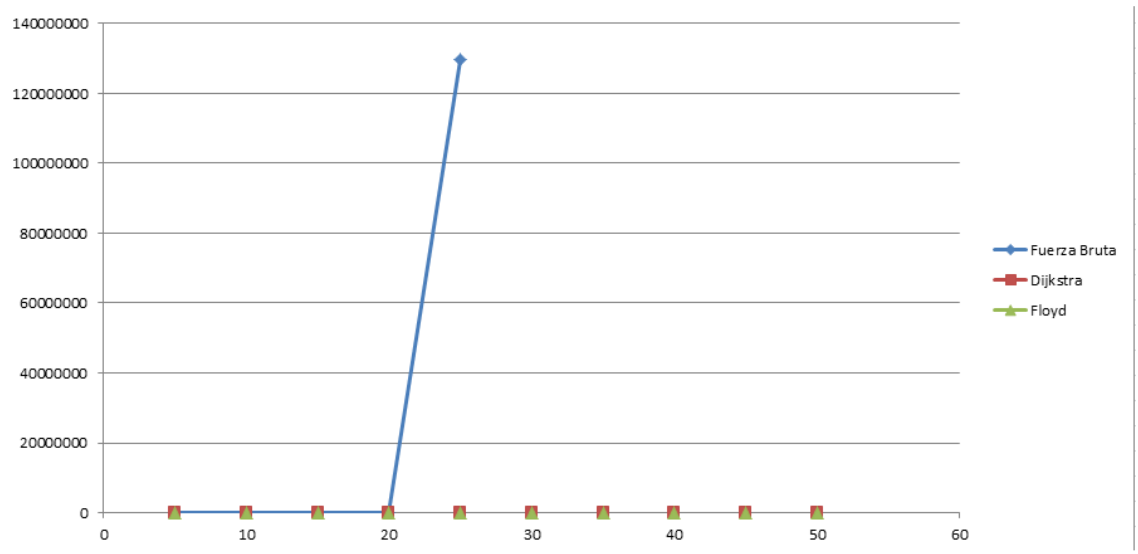
En los tres algoritmos, hemos usado matrices de enteros para representar los caminos y los nodos anteriores o padres. Puesto que las operaciones de acceder a una posición en concreto son de orden constante y se recorren rápidamente. En el algoritmo de fuerza bruta hemos usado un ArrayList<Integer> para representar el camino, esto consume algo más de memoria que hacerlo mediante una matriz de enteros, pero es mucho más sencillo mostrar los caminos posteriormente.
Además, en el algoritmo Dijkstra, se usa una cola de prioridad para poder obtener el vértice que menos esfuerzo nos cuesta recorrer (greedy).

Analizar eficiencia obtenida empíricamente frente a la teórica

Estos son los datos obtenidos empíricamente

Aristas	Fuerza Brut	Dijkstra	Floyd
5	1	2	1
10	6	3	2
15	358	5	2
20	208983	5	2
25	129617561	4	4
30		4	4
35		5	4
40		6	4
45		8	5
50		9	6





- *Algoritmo Fuerza Bruta (*$\Theta(k^n)$*):*

Se trata de un algoritmo de fuerza bruta que contiene dos bucles y en el interior de ellos comprueba si la solución es inmediata, en caso de no serlo, llama a una función recursiva. Esta función puede ser llamada k^n veces, puesto que para cada vértice (no explorado previamente) se realiza tantas llamadas recursivas como vecinos tenga, con el solapamiento de operaciones que esto conlleva. Por lo tanto su orden es de t(n) $\in \Theta(n^2 * k^n) \rightarrow \Theta(k^n)$

- Algoritmo Dijkstra ($\Theta(n^3)$):

El Algoritmo de Dijkstra realiza $\Theta(n^2)$ operaciones (sumas y comparaciones) para determinar la longitud del camino más corto desde un vértice pasado por parámetro hasta todos los demás vértices de un grafo ponderado simple, conexo y dirigido con n vértices. Y como ejecutamos Dijkstra tantas veces como vértices tiene el grafo, el orden es de t(n) $\in \Theta(n * n^2) \rightarrow \Theta(n^3)$

- *Algoritmo Floyd (*$\Theta(n^3)$*):*

El orden de complejidad temporal se debe al triple bucle anidado, en cuyo interior hay tan sólo operaciones constantes. Por lo tanto su orden es de t(n) $\in \Theta(n * n * n) \rightarrow \Theta(n^3)$

Conclusión: El algoritmo de Fuerza Bruta es inviable, Dijkstra y Floyd a pesar de tener el mismo orden de complejidad, el interior del bucle Floyd es más simple, por lo tanto la constante oculta es más pequeña, lo que nos reporta tiempos inferiores que Dijkstra.

Salidas obtenidas por los algoritmos para el juego de pruebas **grafo0.txt**:
Nota: Para ver las salidas de este o cualquier otro juego de prueba hay que modificar las siguientes líneas, dentro de **Main.java**:

```
g.caminosMinimosDijkstra();
g.caminosMinimosFuerzaBruta();
g.caminosMinimosFloyd();
```

Quedando de la siguiente manera:

```
System.out.println(g.caminosMinimosDijkstra());
System.out.println(g.caminosMinimosFuerzaBruta());
System.out.println(g.caminosMinimosFloyd());
```

Salida Fuerza Bruta

[Peso camino de 0 a 1-> 2

, Camino: [0, 1]

, Peso camino de 0 a 2-> 5

, Camino: [0, 1, 2]

, Peso camino de 0 a 3-> 9

, Camino: [0, 1, 2, 3]

, Peso camino de 0 a 4-> 8

, Camino: [0, 1, 4]

, Peso camino de 1 a 0-> 12

, Camino: [1, 2, 3, 0]

, Peso camino de 1 a 2-> 3

, Camino: [1, 2]

, Peso camino de 1 a 3-> 7

, Camino: [1, 2, 3]

, Peso camino de 1 a 4-> 6

, Camino: [1, 4]

, Peso camino de 2 a 0-> 9

, Camino: [2, 3, 0]

, Peso camino de 2 a 1-> 11

, Camino: [2, 3, 0, 1]

, Peso camino de 2 a 3-> 4

, Camino: [2, 3]

, Peso camino de 2 a 4-> 17

, Camino: [2, 3, 0, 1, 4]

, Peso camino de 3 a 0-> 5

, Camino: [3, 0]

, Peso camino de 3 a 1-> 7

, Camino: [3, 0, 1]

, Peso camino de 3 a 2-> 10

, Camino: [3, 0, 1, 2]

, Peso camino de 3 a 4-> 13

, Camino: [3, 0, 1, 4]

, Peso camino de 4 a 0-> 12

, Camino: [4, 3, 0]

, Peso camino de 4 a 1-> 14

, Camino: [4, 3, 0, 1]

, Peso camino de 4 a 2-> 17

, Camino: [4, 3, 0, 1, 2]

, Peso camino de 4 a 3-> 7

, Camino: [4, 3]

]

Salida Dijkstra

[Peso camino de 0 a 1-> 2

, Camino: [0, 1]

, Peso camino de 0 a 2-> 5

, Camino: [0, 1, 2]

, Peso camino de 0 a 3-> 9

, Camino: [0, 1, 2, 3]

, Peso camino de 0 a 4-> 8

, Camino: [0, 1, 4]

, Peso camino de 1 a 0-> 12

, Camino: [1, 2, 3, 0]

, Peso camino de 1 a 2-> 3

, Camino: [1, 2]

, Peso camino de 1 a 3-> 7

, Camino: [1, 2, 3]

, Peso camino de 1 a 4-> 6

, Camino: [1, 4]

, Peso camino de 2 a 0-> 9

, Camino: [2, 3, 0]

, Peso camino de 2 a 1-> 11

, Camino: [2, 3, 0, 1]

, Peso camino de 2 a 3-> 4

, Camino: [2, 3]

, Peso camino de 2 a 4-> 17

, Camino: [2, 3, 0, 1, 4]

, Peso camino de 3 a 0-> 5

, Camino: [3, 0]

, Peso camino de 3 a 1-> 7

, Camino: [3, 0, 1]

, Peso camino de 3 a 2-> 10

, Camino: [3, 0, 1, 2]

, Peso camino de 3 a 4-> 13

, Camino: [3, 0, 1, 4]

, Peso camino de 4 a 0-> 12

, Camino: [4, 3, 0]

, Peso camino de 4 a 1-> 14

, Camino: [4, 3, 0, 1]

, Peso camino de 4 a 2-> 17

, Camino: [4, 3, 0, 1, 2]

, Peso camino de 4 a 3-> 7

, Camino: [4, 3]

]

Salida Floyd

[Peso camino de 0 a 1-> 2

, Camino: [0, 1]

, Peso camino de 0 a 2-> 5

, Camino: [0, 1, 2]

, Peso camino de 0 a 3-> 9

, Camino: [0, 1, 2, 3]

, Peso camino de 0 a 4-> 8

, Camino: [0, 1, 4]

, Peso camino de 1 a 0-> 12

, Camino: [1, 2, 3, 0]

, Peso camino de 1 a 2-> 3

, Camino: [1, 2]

, Peso camino de 1 a 3-> 7

, Camino: [1, 2, 3]

, Peso camino de 1 a 4-> 6

, Camino: [1, 4]

, Peso camino de 2 a 0-> 9

, Camino: [2, 3, 0]

, Peso camino de 2 a 1-> 11

, Camino: [2, 3, 0, 1]

, Peso camino de 2 a 3-> 4

, Camino: [2, 3]

, Peso camino de 2 a 4-> 17

, Camino: [2, 3, 0, 1, 4]

, Peso camino de 3 a 0-> 5

, Camino: [3, 0]

, Peso camino de 3 a 1-> 7

, Camino: [3, 0, 1]

, Peso camino de 3 a 2-> 10

, Camino: [3, 0, 1, 2]

, Peso camino de 3 a 4-> 13

, Camino: [3, 0, 1, 4]

, Peso camino de 4 a 0-> 12

, Camino: [4, 3, 0]

, Peso camino de 4 a 1-> 14

, Camino: [4, 3, 0, 1]

, Peso camino de 4 a 2-> 17

, Camino: [4, 3, 0, 1, 2]

, Peso camino de 4 a 3-> 7

, Camino: [4, 3]

]