

# Práctica 3.

# Programación dinámica

---

ESTRUCTURAS DE DATOS Y ALGORITMOS 2

AUTORES: Francisco Javier Marqués Gaona, Germán Ruano García y  
David Subires Parra  
GRADO INGENIERÍA INFORMÁTICA | EDA2

## **Práctica 3. Programación dinámica**

### ***Índice***

• <i>Presentación del problema</i>	<i>Página 2</i>
• <i>Algoritmos de pretratamiento de datos</i>	<i>Página 5</i>
• <i>Desarrollo teórico y análisis de eficiencia</i>	<i>Página 7</i>
• <i>Tipo de problema y características especiales</i>	<i>Página 12</i>
• <i>Juegos de pruebas</i>	<i>Página 13</i>
• <i>Salidas del programa</i>	<i>Página 15</i>

### ***Presentación del problema***

Uno de los resultados de la práctica 2 es un listado de intercambiadores con fugas, indicando el porcentaje de caída de la presión y ordenado según este valor, así como señalando el tipo de fuga.

En esta práctica necesitaremos sólo las referencias de los intercambiadores y los porcentajes de caída, pudiendo obviar el tipo de fuga.

Número de intercambiador	Porcentaje de caída
15	10.00
900	9,46
512	9,46
816	8,23
4	8,21
123	7,11
....	....
322	0,34
215	0,24
519	0,24
711	0,24
1002	0,21
693	0,10

La expansión urbana “New Almería” se ha desarrollado según la típica trama en damero usual en América; en nuestro caso vamos a obviar que se trata de una isla en la que algunas manzanas no están presentes, por lo que trabajaremos con un damero como el de la figura, con  $n$  avenidas y  $m$  calles

Los intercambiadores están ubicados en las calles y avenidas, y se supone que nos van a dar un listado de sus ubicaciones; las ubicaciones van a venir dadas de la siguiente forma:

- Un intercambiador ubicado en la calle  $i$  entre las avenidas  $j$  y  $j+1$  vendrá referenciado como  $CiAj$
- Un intercambiador ubicado en una avenida  $j$  entre las calles  $i$  y  $i+1$  vendrá referenciado como  $AjCi$

Se supone que todos los intercambiadores ubicados en el mismo segmento de calle o avenida tienen la misma referencia, el equipo de trabajo dispone luego de mapas más detallados con su ubicación. En la figura 2 se muestra un ejemplo de varias ubicaciones de intercambiadores.

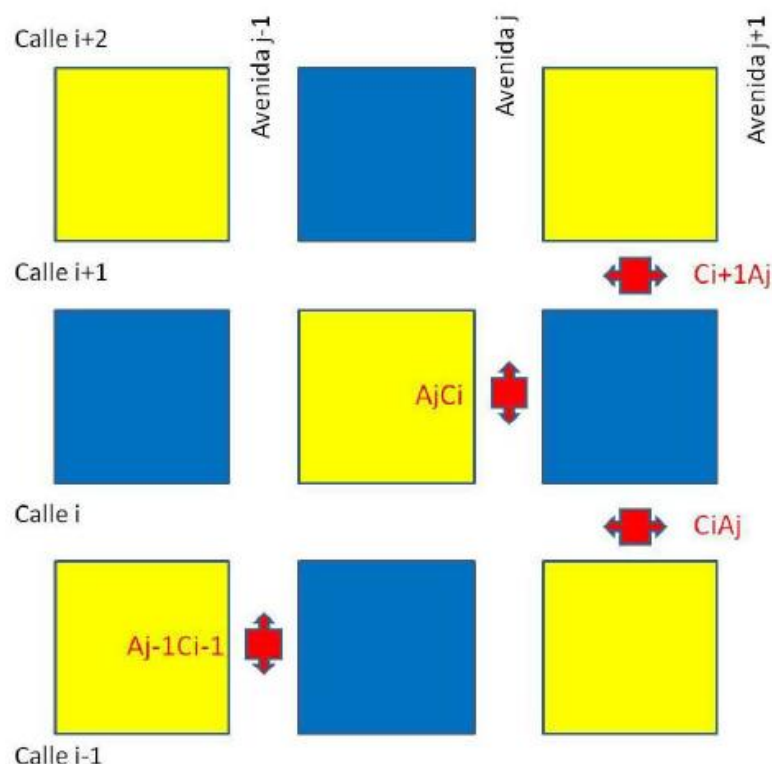


Figura 2. Ejemplo de ubicación de algunos intercambiadores en la trama urbana.

Dada una retícula con  $n$  avenidas y  $m$  calles, siendo  $(n, m)$  dos valores de entrada, debe de generar un archivo que de forma aleatoria asigne una ubicación a los intercambiadores del listado anterior. En la realidad estas ubicaciones nos serían dadas desde el departamento topográfico de la empresa. El formato para anotar las direcciones es libre, pueden almacenarlo de forma que les parezca más oportuna para el posterior procesamiento. El generador de ubicaciones debe de ser capaz de imprimir un archivo de texto con el listado de los intercambiadores y su ubicación (tabla 3).

Número de intercambiador	Porcentaje de caída	Ubicación intercambiador
15	10,00	A5C18
900	9,46	C21A3
512	9,46	C26A4
816	8,23	C40A3
4	8,21	A1C12
123	7,11	A2C72
....	....	
322	0,34	C99A8
215	0,24	C72A21
519	0,24	C81A42
711	0,24	A12C42
1002	0,21	A26C76
693	0,10	C12A20

Tabla 3. Ejemplo de tabla con los datos de los intercambiadores y sus ubicaciones.

Otros valores de entrada que tenemos son, el tamaño de las manzanas (bloques), **tm**, el ancho de las calles y avenidas, **an**; en nuestro caso vamos a suponer que las manzanas son cuadradas y el ancho de las calles y avenidas es el mismo. A su vez supondremos que todas las vías son de doble dirección. Todas estas simplificaciones se ponen para evitar centrar la atención en puntos distintos del trozo algorítmico que se desea desarrollar. Se supone que todos los intercambiadores de un tramo de calle están ubicados en el centro del tramo de calle o avenida. Con estos valores podemos calcular la distancia entre cualquier par de intercambiadores.

Podemos calcular la distancia entre cualquier par de intercambiadores utilizando el algoritmo adecuado que deben desarrollar.

Ya tenemos todos los datos necesarios para nuestra práctica. Nuestro objetivo es fijar una ruta para que la brigada de reparación de intercambiadores revise todos los intercambiadores de la lista, pero con las siguientes condiciones:

1. La lista de intercambiadores se segmenta en varias sub-listas, se fijan diversas prioridades según unos límites en los porcentajes de caída. En la tabla 5 se muestra cómo se divide la tabla ejemplo:

Número de intercambiador	Porcentaje de caída	Ubicación intercambiador	Sub-listas
15	10.00	A5C18	Segmento 1 ( $\geq 8\%$ )
900	9,46	C21A3	
512	9,46	C26A4	
816	8,23	C40A3	
4	8,21	A1C12	
123	7,11	A2C72	Segmento 2 <8% y $\geq 5\%$
....	....		
....	....		Segmento f < 0.5%
322	0,34	C99A8	
215	0,24	C72A21	
519	0,24	C81A42	
711	0,24	A12C42	
1002	0,21	A26C76	
693	0,10	C12A20	

Tabla 4. Ejemplo de tabla con los datos de los intercambiadores y sus ubicaciones.

2. Se parte del taller donde tienen el material los técnicos, ubicado en un edificio con dirección (App, Cpp) utilizando para referenciarlo el mismo sistema que hemos utilizado para los intercambiadores; si bien no puede estar en medio de la calle suponemos que la diferencia es despreciable.
3. Se han de revisar primero los intercambiadores del segmento 1, luego los del 2 y así sucesivamente hasta revisar todos los intercambiadores, al final se vuelve al taller. Como los intercambiadores del segmento 1 son los que más peligro ofrecen, se han de recorrer optimizando la distancia recorrida en dicho segmento, luego los del dos, etc. Si hay un recorrido no óptimo en el segmento 1 que termina en un punto tal que obtiene un valor menor para el conjunto Segmento 1 – Segmento 2 (y siguientes), se opta por el que recorre el segmento 1 de forma óptima (con menor distancia en el segmento 1)

4. Se desea realizar el recorrido recorriendo la menor distancia posible, pero revisando los intercambiadores por segmento, sólo se pueden intercambiar los intercambiadores de un segmento. No se puede comenzar a trabajar con un intercambiador de un segmento hasta no haber terminado con el anterior.

Debe generar el listado con los números de los intercambiadores en el orden en que se han de revisar, así como la distancia que se va a recorrer. El punto de partida para recorrer el primer segmento es el taller, para el segundo segmento el último intercambiador del primer segmento, y así sucesivamente, al terminar el último segmento se ha de volver al taller.

## Algoritmos de pretratamiento de datos

En nuestro caso, hemos fijado los valores número de calles, número de avenidas, tamaño manzana y ancho avenidas y calles, dentro de la clase Ubicación.java.

Además, en la clase Reparación.java, hemos fijado los valores SEP1, SEP2 y TALLER. Taller contiene la dirección del taller (el punto C1A1 en nuestro caso), SEP1 indica el porcentaje que debe ser menor o igual que los porcentajes de las averías del subconjunto 1, SEP2 lo mismo pero con el subconjunto 2, y el resto de averías se añadirán al subconjunto 3.

Para la correcta ejecución del algoritmo, necesitamos realizar un tratamiento de datos sobre la salida de la práctica 2 (listas de averías con su porcentaje de fuga). El tratamiento consiste en asignar a cada avería una ubicación aleatoria e implementar la manera de calcular la distancia entre dos puntos cualesquiera.

Para asignar a cada avería una ubicación aleatoria simplemente instanciamos una nueva ubicación y la asignamos a la avería, el constructor que instancia la ubicación ya lo hace con una ubicación aleatoria, dentro de los rangos permitidos:

```
public Ubicacion () {  
    calle = (int) (Math.random()*m) + 1;  
    avenida = (int) (Math.random()*n) + 1;  
    if (Math.random() <= 0.5)  
        nombre = "A"+avenida+"C"+calle;  
    else  
        nombre = "C"+calle+"A"+avenida;  
}
```

Para calcular la distancia entre dos puntos cualesquiera, hemos usado el método descrito en el PDF de la práctica:

```
public double calcularDistancia (Ubicacion otra) {  
    // la primera letra de cada direccion es diferente  
    if (this.nombre.charAt(0) != otra.nombre.charAt(0)) {  
        double distancia = (Math.abs(this.avenida - otra.avenida) +  
            Math.abs(this.calle - otra.calle)) * (tm+an);  
        if ((this.avenida - otra.avenida) * (this.calle - otra.calle) <= 0)  
            distancia = distancia + 1 * (tm+an);  
        return distancia;  
    }  
    else {  
        //las dos comienzan por la misma letra  
        if (this.nombre.charAt(0) == 'A') {  
            if (this.calle == otra.calle)  
                return (Math.abs(this.avenida - otra.avenida) +  
                    Math.abs(this.calle - otra.calle) + 1) * (tm+an);  
            else  
                return (Math.abs(this.avenida - otra.avenida) +  
                    Math.abs(this.calle - otra.calle)) * (tm+an);  
        }  
        else {  
            if (this.avenida == otra.avenida)  
                return (Math.abs(this.avenida - otra.avenida) +  
                    Math.abs(this.calle - otra.calle) + 1) * (tm+an);  
            else  
                return (Math.abs(this.avenida - otra.avenida) +  
                    Math.abs(this.calle - otra.calle)) * (tm+an);  
        }  
    }  
}
```

Ambos métodos están dentro de la clase **Ubicación.java**

## Desarrollo teórico y análisis de eficiencia de algoritmo principal

El algoritmo que genera el recorrido de forma óptima es un método privado de la clase Reparación, (**CalculaRutaR**). Este método es llamado por el método público **CalculaRuta**. Este último se encarga de dividir la lista de averías en 3 sub-conjuntos, en nuestro caso, en el primer sub-conjunto se encuentran las averías con un porcentaje de fuga mayor o igual a 7, en el segundo sub-conjunto averías con un porcentaje mayor o igual a 3 y en el tercer y último sub-conjunto el resto de averías.

El primer sub-conjunto se resuelve con el algoritmo minimizando el recorrido teniendo en cuenta que empezamos en el taller y terminamos en el taller también, ya que no sabemos cuál será el próximo punto del subconjunto 2, en el caso del subconjunto 2 se ejecuta teniendo en cuenta que el recorrido empieza desde el último punto del recorrido 2 y termina en el mismo punto también, y por último, en el subconjunto 3, se realiza teniendo en cuenta que el punto de partida es el último punto del subconjunto 2 y que la ruta debe de finalizar lo más próxima posible al taller.

El problema puede ser resuelto mediante programación dinámica, puesto que cumple el principio de optimalidad de Bellman, puesto que toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas.

Vamos a **calcular** la eficiencia del algoritmo **CalculaRutaR**:

**Esquema algorítmico:**

Método CalculaRutaR(Avería desde, Lista S, Avería finalRuta): entero {

Entero masCorto, distancia

Avería masCortoA

**Si** S es vacío **entonces**

Tabla[desde, {}] = distancia[desde, finalRuta]

TablaJ[desde, {}] = finalRuta

devuelve distancia[desde, finalRuta]

**Sino**

**Si** tabla contiene (desde, S) **entonces**

**devuelve** mapa[desde, S]

**Sino**

masCorto = infinito

masCortoA = null

**para todo** av en S **hacer**

distancia = distancia[desde, av] + CalculaRutaR(j, S \ {j})

**si** distancia < masCorto **entonces**

masCorto = distancia

masCortoA = av

**fsi**

**fpara**

mapa[desde, S] = masCorto

mapaJ[desde, S] = masCortoA

**devuelve** masCorto

**fsi**

**fsi**

}



**Eficiencia:**

Cálculo de  $\text{CalculaRutaR}(i, \{ \}) = n-1$  consultas a la tabla

Cálculo de los  $\text{CalculaRutaR}(i, S) = \binom{n-1}{k} k$  sumas en total

Cálculo de  $\text{CalculaRutaR}(1, S \setminus \{1\}) = n-1$  sumas

Eficiencia total

$$O(2(n-1) + \sum_{k=1}^{n-1} (n-1) k \binom{n-2}{k}) = O(n^2 2^n)$$

Por lo tanto, el algoritmo está en orden de  **$O(2^n)$**

Para comprobar que la ejecución del algoritmo es correcta y que reutiliza los datos de la tabla en la que se van almacenando hemos desarrollado el siguiente ejemplo:

Hemos añadido un contador a cada “parte” por la que puede pasar la ejecución del algoritmo, para obtener el número de veces y el porcentaje de veces que la ejecución pasa por cada parte:

```
private double calculaRutaR(Averia desde, ArrayList<Averia> conjunto){
    double masCorto,distancia;
    Averia masCortoA;
    EstructuraAverialista eal;
    n1++;
    if(conjunto.isEmpty()){
        double temp = desde.calcularDistancia(finalRuta);
        eal = new EstructuraAverialista(desde, new ArrayList<Averia>());
        mapa.put(eal, temp);
        mapaJ.put(eal, finalRuta);
        n2++;
        return temp;
    }else{
        if(mapa.get(new EstructuraAverialista(desde, conjunto)) != null){
            n3++;
            return mapa.get(new EstructuraAverialista(desde, conjunto));
        }
        else{
            n4++;
            masCorto = Double.MAX_VALUE;
            masCortoA = null;

            for(Averia av : conjunto){
                ArrayList<Averia> conjuntoMenosDesde = (ArrayList<Averia>) conjunto.clone();
                conjuntoMenosDesde.remove(av);
                distancia = desde.calcularDistancia(av) + calculaRutaR(av, conjuntoMenosDesde);

                if(distancia < masCorto){
                    masCorto = distancia;
                    masCortoA = av;
                }
            }

            eal = new EstructuraAverialista(desde, conjunto);
            mapa.put(eal, masCorto);
            mapaJ.put(eal, masCortoA);
            return masCorto;
        }
    }
}
```

N1 = número total de veces que se ejecuta recursivamente el algoritmo.

N2 = número de veces que el algoritmo acaba en el caso base

N3 = número de veces que el algoritmo finaliza reutilizando datos almacenados en la tabla

N4 = número de veces que se ejecuta la parte más pesada del algoritmo, en la que no conocemos el camino de coste mínimo de un punto a otro, por lo tanto realizamos tantas llamadas recursivas `CalculaRutaR` como nodos haya en el conjunto, sin contar el nodo desde.

Y el resultado que hemos obtenido es el siguiente:

N1 =	8912914	100 %
N2 =	272	0.003 %
N3 =	7798546	87.497 %
N4 =	1114096	12.499 %

Estos datos se han obtenido con el juego de pruebas **ubicaTestN.txt**

Como podemos observar, casi el 90% de las instancias del problema se resuelven accediendo a un dato almacenado previamente en la tabla.

La “**tabla**” de nuestro algoritmo de programación dinámica es un mapa, en concreto un **HashMap<EstructuraAveriaLista, doublé>**, para lo cual hemos implementado la clase **EstructuraAveriaLista**, que, como su propio nombre indica, contiene una avería (que sería el punto de partida) y una lista de averías (que sería el conjunto de puntos a recorrer. Además, para que la estructura funcione correctamente, hemos añadido el método `hashCode` a la clase **EstructuraAveriaLista**:

```
@Override
public int hashCode() {
    return desde.hashCode() + conjunto.hashCode();
}
```

Esta estructura optimiza la memoria consumida frente a una matriz de tamaño  $N \times M$ , donde  $N$  es el número de averías a reparar y  $M$  es el número de subconjuntos diferentes posibles de  $M$ , de tamaño  $N - 1$ . Esta estructura nos permite almacenar/obtener el coste mínimo del recorrido partiendo desde un vértice  $i$  hasta recorrer el conjunto  $S$  (`calculaRutaR(i, S)`).

Además, aparte de almacenar/obtener el coste del recorrido, necesitamos saber qué avería es la siguiente a recorrer, de acuerdo con el coste mínimo del recorrido almacenado en el mapa, para lo cual hemos implementado el mapa **J**. Consiste en otro mapa, **HashMap<EstructuraAveriaLista, Averia>**

Por último, para obtener el recorrido solución, hemos implementado el método `recorridoSolucion(Averia inicio, ArrayList<Averia> conjunto)`.

A continuación vamos a mostrar un **ejemplo** del **contenido** de la **tabla g** (mapa) y tabla J para solucionar el problema **g (Taller, {1,2,3}, Taller) [g(InicioRuta, Conjunto, finalRuta)]**:

Tabla L

Column1	1	2	3	Taller	
1	0	1	1	1	5
2	1	0	1	1	6
3	1	1	0	0	7
TALLER	7	6	5	0	

mapaG

Key	Value
1 {}	Taller
2 {}	Taller
3 {}	Taller
1 {2}	2
1 {3}	3
2 {1}	1
2 {3}	3
3 {1}	1
3 {2}	2
1 {2, 3}	3
2 {1, 3}	3
3 {1, 2}	2
Taller {1, 2, 3}	3

mapaJ

Column1	Column2
1 {}	5
2 {}	6
3 {}	7
1 {2}	7
1 {3}	8
2 {1}	6
2 {3}	8
3 {1}	6
3 {2}	7
1 {2, 3}	8
2 {1, 3}	7
3 {1, 2}	7
Taller {1, 2, 3}	12

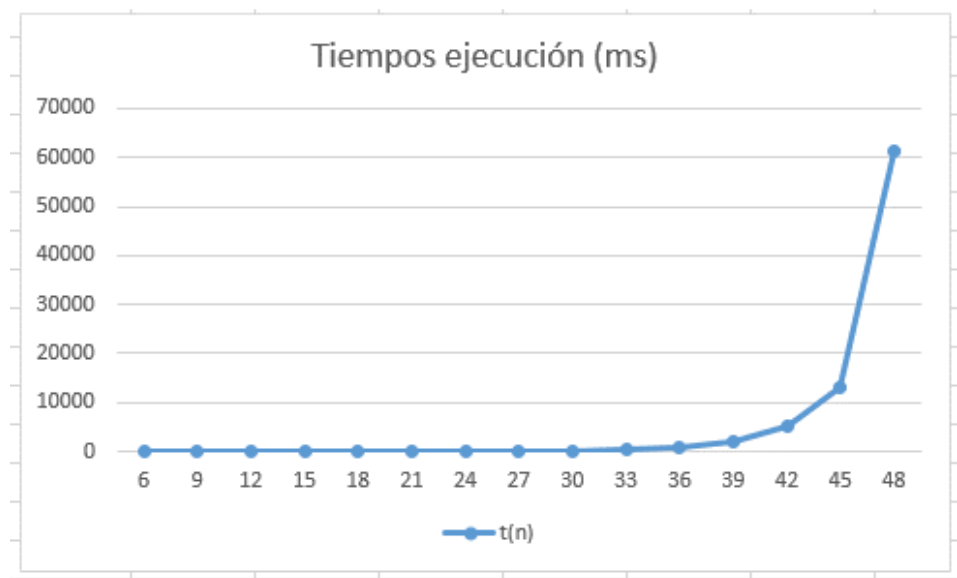
Para este sencillo ejemplo, el **coste** del ciclo hamiltoniano más corto sería **12** con el **recorrido** **{Taller, 3, 2, 1, Taller}** empezando y terminando el recorrido en el taller y recorriendo todos los puntos del subconjunto.

Además, con este ejemplo, podemos justificar que este tipo de problema se puede solucionar mediante programación dinámica ya que una solución se construye a partir de soluciones de problemas más pequeños, por ejemplo:

$$\begin{aligned}
 g(\text{Taller}, \{1, 2, 3\}, \text{Taller}) &= \min( \text{distancia}[\text{Taller}][1] + g(1, \{2, 3\}, \text{Taller}), \\
 &\quad \text{distancia}[\text{Taller}][2] + g(2, \{1, 3\}, \text{Taller}), \\
 &\quad \text{distancia}[\text{Taller}][3] + g(3, \{1, 2\}, \text{Taller}) ) \\
 &= \min( 7 + 8, 6 + 7, 5 + 7 ) = \min(15, 13, 12) = 12
 \end{aligned}$$

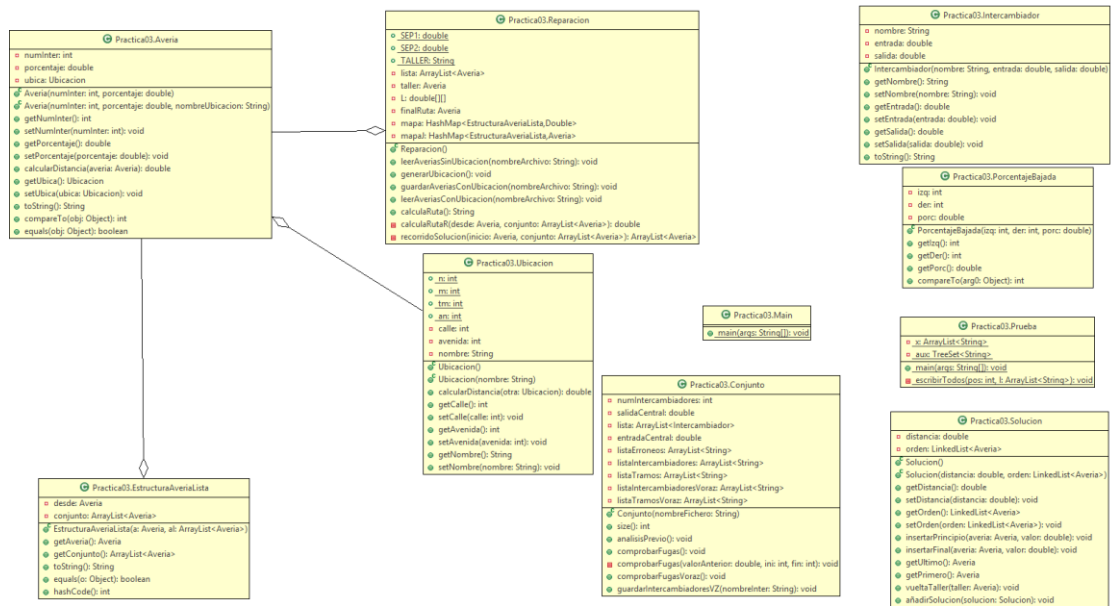
Una vez obtenida la eficiencia teórica del algoritmo, vamos a calcular la eficiencia empírica:

n	t(n)
6	1
9	1
12	1
15	2
18	2
21	5
24	16
27	48
30	140
33	359
36	983
39	2137
42	5299
45	13202
48	61241



Como podemos observar, la eficiencia teórica y la eficiencia empírica coinciden. Estos tiempos han sido obtenidos con los juegos de prueba ubicaX.txt

Diagrama de clases de la práctica:



## Tipo de problema y características especiales

En esta práctica nos encontramos con un caso especial del **problema del viajante**. Sus características especiales son que, siendo  $n$  el número de puntos a recorrer, dividimos  $n$  en tres subconjuntos dependiendo la prioridad de pasar por cada punto, primero los puntos con un porcentaje de pérdida superior a 7%, después otro subconjunto con porcentaje de pérdida superior a 3% y por último los puntos restantes. Finalmente “unimos” el recorrido, partiendo del taller, recorrido obtenido para el subconjunto 1, 2, 3 y por último vuelta a taller.

## Juegos de pruebas

Para generar los diversos juegos de pruebas hemos implementado las siguientes líneas de código en la clase **Main.java**:

```
for (int i=1; i<=25; i++) {  
    String nombreInter = directorio + "intervz"+i+".txt";  
    String nombreUbica = directorio + "ubica"+i+".txt";  
    Reparacion repara = new Reparacion ();  
    repara.leerAveriasSinUbicacion(nombreInter);  
    repara.generarUbicacion ();  
    repara.guardarAveriasConUbicacion (nombreUbica);  
}  
  
for (int i=1; i<=25; i++) {  
    String nombreUbica = directorio + "ubica"+i+".txt";  
    Reparacion repara = new Reparacion ();  
    repara.leerAveriasConUbicacion(nombreUbica);  
}
```

Lo que hacen estas líneas, es, a partir de la salida de la práctica 02, la cual generaba informes con listas de intercambiadores con pérdidas ( idIntercambiador, PorcentajePérdida ), y mediante el método leerAveriasSinUbicacion() de la clase Reparación.java, leer estos datos y generar para cada tupla una ubicación aleatoria. Después mediante el método guardarAveriasConUbicación() almacenamos en disco la tupla ( idIntercambiador, PorcentajePérdida, Ubicación ).

El algoritmo resuelve sin problemas los juegos de pruebas generados partiendo de los resultados de la práctica 02, pero al ser todos los listados de un tamaño parecido, no genera unas gráficas representativas sobre el rendimiento de éste mismo. Por lo tanto, para la representación gráfica del rendimiento, hemos modificado los juegos empezando desde un tamaño y aumentando en tres averías este tamaño consecutivamente para los siguientes juegos.

Todas las salidas de los juegos de pruebas se entregan adjuntos a este informe.

Así hemos podido llegar a la conclusión de que el algoritmo genera un error de memoria a partir de un tamaño de problema  $n = 51$ . En la siguiente foto se muestra el error generado:

```
ated> Main (4) [Java Application] C:\Program Files (x86)\Java\jre1.8.0_45\bin\javaw.e
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.HashMap.resize(Unknown Source)
at java.util.HashMap.putVal(Unknown Source)
at java.util.HashMap.put(Unknown Source)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:183)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:173)
at Practica03.Reparacion.calculaRutaR(Reparacion.java:123)
at Practica03.Main.main(Main.java:72)
```

Esto se podría solucionar modificando la estructura que almacena la “tabla”, cambiando el `HashMap<EstructuraAveriaLista, Double>` por este otro

`HashMap<Averia, <HashMap<ArrayList<Averia>, Double>>>`

Esto nos permitiría dividir la carga de memoria, en lugar de almacenar todo en un solo `HashMap`, lo almacenaríamos todo en `N HashMap`, donde `N` es el tamaño del problema a resolver. Aunque no lo hemos llevado a la práctica por falta de tiempo.

## Salidas del programa

La salida del programa tiene el siguiente formato, como se requería:

Número de averías: N

Coste recorrido: C

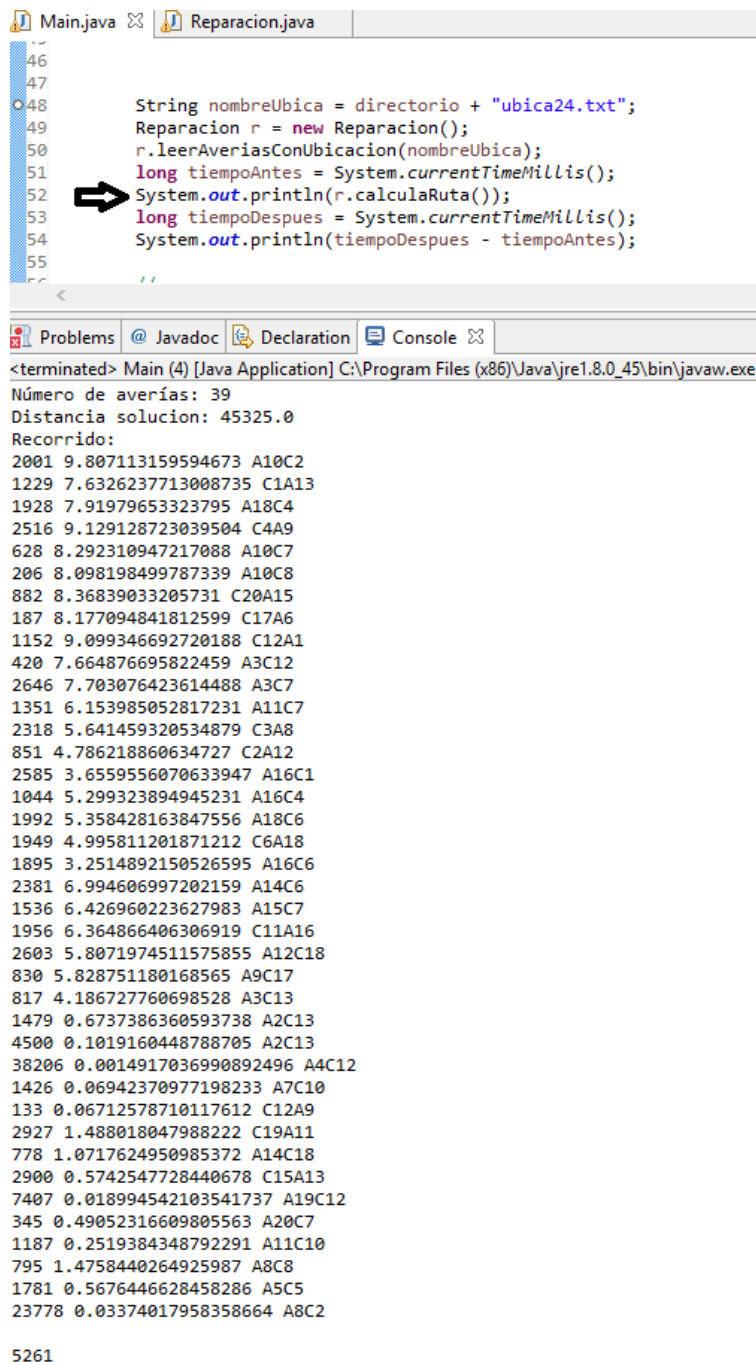
Recorrido:

....

idIdentifiacor, porcentajePerdida, ubicación

idIdentifiacor, porcentajePerdida, ubicación

.....



```

46
47
48 String nombreUbica = directorio + "ubica24.txt";
49 Reparacion r = new Reparacion();
50 r.leerAveriasConUbicacion(nombreUbica);
51 long tiempoAntes = System.currentTimeMillis();
52 System.out.println(r.calculaRuta());
53 long tiempoDespues = System.currentTimeMillis();
54 System.out.println(tiempoDespues - tiempoAntes);
55

```

```

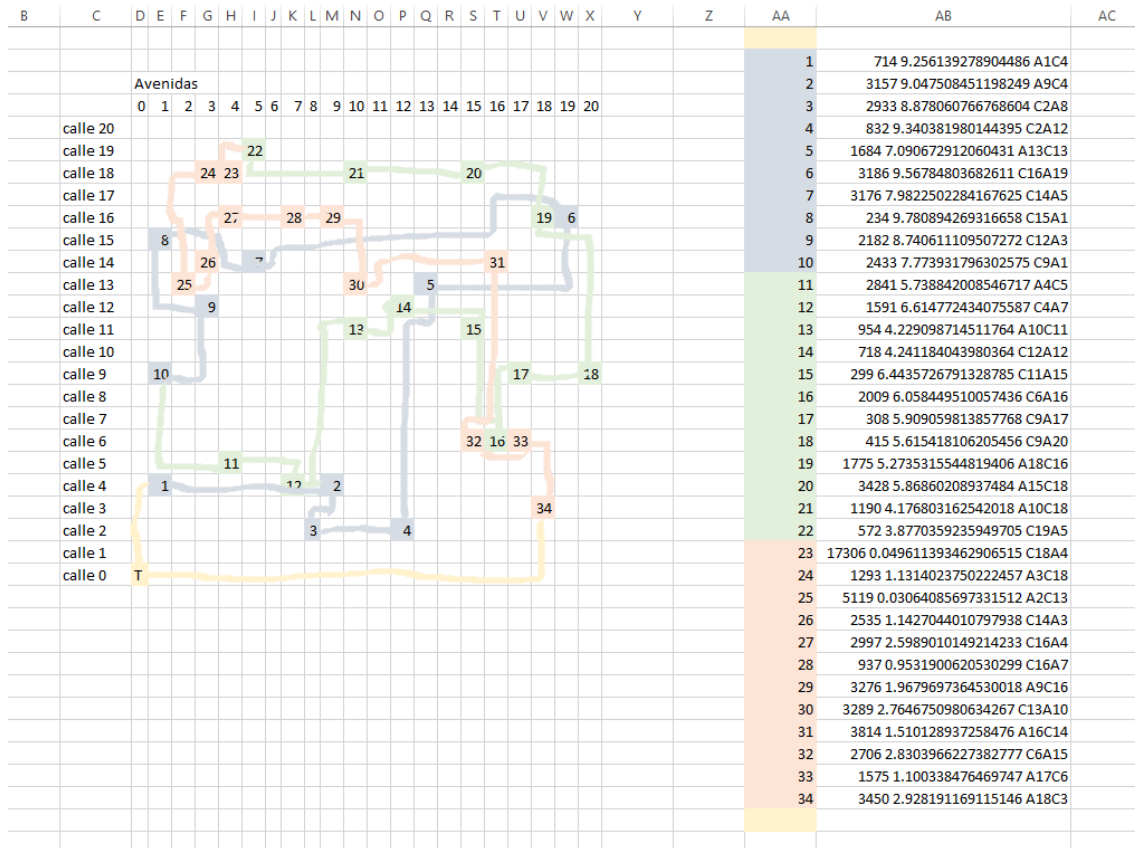
<terminated> Main (4) [Java Application] C:\Program Files (x86)\Java\jre1.8.0_45\bin\javaw.exe
Número de averías: 39
Distancia solución: 45325.0
Recorrido:
2001 9.807113159594673 A10C2
1229 7.6326237713008735 C1A13
1928 7.91979653323795 A18C4
2516 9.129128723039504 C4A9
628 8.292310947217088 A10C7
206 8.098198499787339 A10C8
882 8.36839033205731 C20A15
187 8.177094841812599 C17A6
1152 9.099346692720188 C12A1
420 7.664876695822459 A3C12
2646 7.703076423614488 A3C7
1351 6.153985052817231 A11C7
2318 5.641459320534879 C3A8
851 4.786218860634727 C2A12
2585 3.6559556070633947 A16C1
1044 5.299323894945231 A16C4
1992 5.358428163847556 A18C6
1949 4.995811201871212 C6A18
1895 3.2514892150526595 A16C6
2381 6.994606997202159 A14C6
1536 6.426960223627983 A15C7
1956 6.364866406306919 C11A16
2603 5.8071974511575855 A12C18
830 5.828751180168565 A9C17
817 4.186727760698528 A3C13
1479 0.6737386360593738 A2C13
4500 0.1019160448788705 A2C13
38206 0.0014917036990892496 A4C12
1426 0.06942370977198233 A7C10
133 0.06712578710117612 C12A9
2927 1.488018047988222 C19A11
778 1.0717624950985372 A14C18
2900 0.5742547728440678 C15A13
7407 0.018994542103541737 A19C12
345 0.49052316609805563 A20C7
1187 0.2519384348792291 A11C10
795 1.4758440264925987 A8C8
1781 0.5676446628458286 A5C5
23778 0.03374017958358664 A8C2
5261

```



Todas las salidas de los juegos de prueba se entregan junto a este informe, los juegos de prueba y el código de la práctica.

Además, por ultimo, hemos representado gráficamente el recorrido que nos devuelve el algoritmo para el juego de prueba **ubicaTest.txt** prueba



Como podemos observar, el recorrido del subconjunto uno y el dos es casi un ciclo hamiltoniano, a falta de una arista, esto es porque en estos recorridos se ha minimizado la distancia en función de un único punto, es decir, los recorridos uno y dos se calculan teniendo en cuenta que se empieza y acaba en el mismo punto, en el caso del subconjunto uno hemos establecido como punto de partida y de fin el taller y en el caso del subconjunto dos el ultimo punto del recorrido del subconjunto uno. En cambio, el recorrido tres es distinto, no se asemeja a un ciclo hamiltoniano, más bien a un camino, esto es porque en este recorrido se ha minimizado la distancia estableciendo como punto de partida del recorrido el ultimo punto del recorrido del subconjunto dos, y como punto de final de recorrido el taller.