Practica 02

CURSO 2017-2018

<Greedy, Esquema Voraz>

FRANCISCO JAVIER MARQUÉS GAONA
JUAN JOSÉ MÉNDEZ CONTRERAS
JOSE ANDRÉS GONZÁLEZ VENTURA

ESTRUCTURA DATOS Y ALGORITMOS II
2º GRADO EN INGENIERÍA INFORMÁTICA

Departamento de Informática
Universidad de Almería

ÍNDICE

PRESENTACIÓN DEL PROBLEMA

Partimos del mismo supuesto práctico que en la práctica 1, relativo al sistema de metro y metro ligero de la expansión urbana de Nueva Almería, trabajando para la "Metropolitan Transport Authority" (NAMTA).
El trabajo a desarrollar tiene dos partes:
a/ Gestionar el análisis de las roturas de la catenaria descrito en la práctica 1 utilizando en este caso el esquema "greedy", buscando la misma funcionalidad; el objetivo es comprobar las diferencias de eficiencia en los distintos escenarios/casos analizados. Esta aplicación supone simplemente sustituir en la aplicación de la práctica 1 el método D&V por un Greedy con igual funcionalidad, este Greedy es realmente un caso de fuerza bruta, normalmente la FB tiene estructura Greedy.
b/ Construir una aplicación distinta para el cálculo del tiempo de desplazamiento desde una cierta estación de la red (origen) a cualquier otra estación de la red.
Para ello tenemos que profundizar en la topología de la red de metro, según los siguientes parámetros y consideraciones:

- El tamaño de la expansión urbana viene dado por el número de avenidas – m – y de calles – n –. Ver figura 1 de la práctica 1.

- En la figura 1 de esta práctica (figura 2 de la práctica 1) se ve la estructura de las líneas de metro. Se ha considerado una versión simplificada en la que hay sólo 6 líneas (teniendo la 1 tres ramales). La línea 1, en forma de cruz, va por la avenida y calles centrales – caso de ser m y/o n pares, irían por m DIV 2 y n DIV 2 (división entera). La línea 2 va por la periferia de la urbanización, que es rectangular. Las líneas 3 a 6 van por la zona media entre la línea central (1) y la circular (2), estando más cerca de la circular cuando no puede estar equidistante (como se ve en la figura).

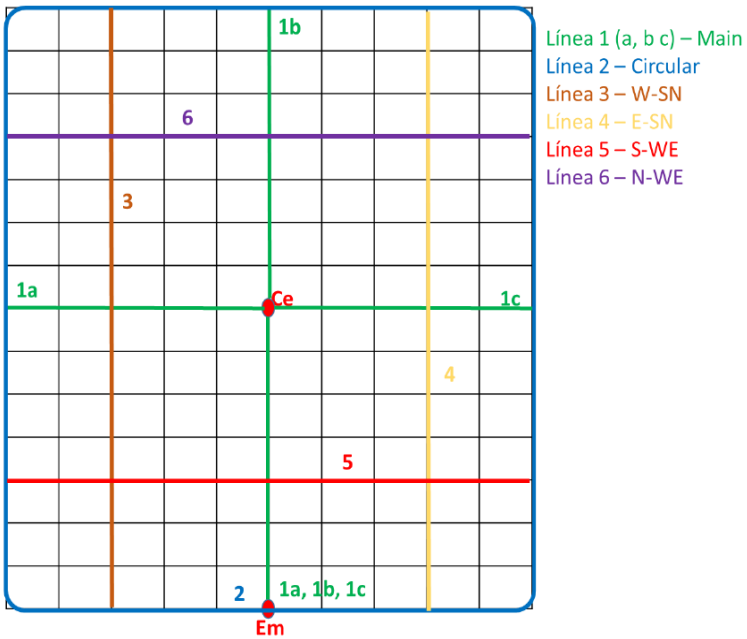


Figura 2. Red de metro y metro ligero de Nueva Almería, con sus distintas líneas.

- Hay estaciones de metro en todos los cruces de las distintas líneas.
- Hay estaciones de metro cada dos cruces avenida-calle en la línea central partiendo de la estación Ce (Central).
- Hay estaciones de metro partiendo de Em (Embarkment) cada dos cruces de avenida-calle en la línea circular – 2 -, saliendo por cada lado hasta llegar a la estación opuesta.
- En las líneas 3 y 4 hay estaciones de metro cada dos cruces partiendo del límite sur (calle 1). En las líneas 5 y 6 hay estaciones de metro cada dos cruces partiendo de la avenida 1.
- Las estaciones (exceptuando Em y Ce que tienen nombre propio) vienen referenciadas por el cruce en que se encuentran (avenida, calle).

Siguiendo estas reglas hay que definir para un par (m,n) las estaciones que existen, por ejemplo para el caso m= 11 y n= 15 las estaciones serían:

- Especiales: Em (6,1) y Ce (6,8), realmente son un caso especial de cruce.

- Cruces: (1,8), (8,15) y (11,8) – línea central/circular -, (3,1), (3,4), (3,8), (3,12) y (3,15) – línea 3/todas -, (9,1), (9,4), (9,8), (9,12) y (9,15) – línea 4/todas -, (1,4), (3,4), (6,4), (9,4) y (11,4) – línea 5/todas -, y finalmente -, (1,12), (3,12), (6,12), (9,12) y (11,12) – línea 6/todas -.

- Estaciones simples (sólo una línea, sin intercambiador):

> Línea 1: (6,6), (6,2), (6,10), (6,14), (4,8), (2,8), (8,8) y (10,8).

> Línea 2: (4,1), (2,1), (1,2), (1,6), (1,10), (1,14), (2,15), (4,15), (6,1), (8,1), (10,2), (11,6), (11,10), (11,14), (8,15) y (6,15).

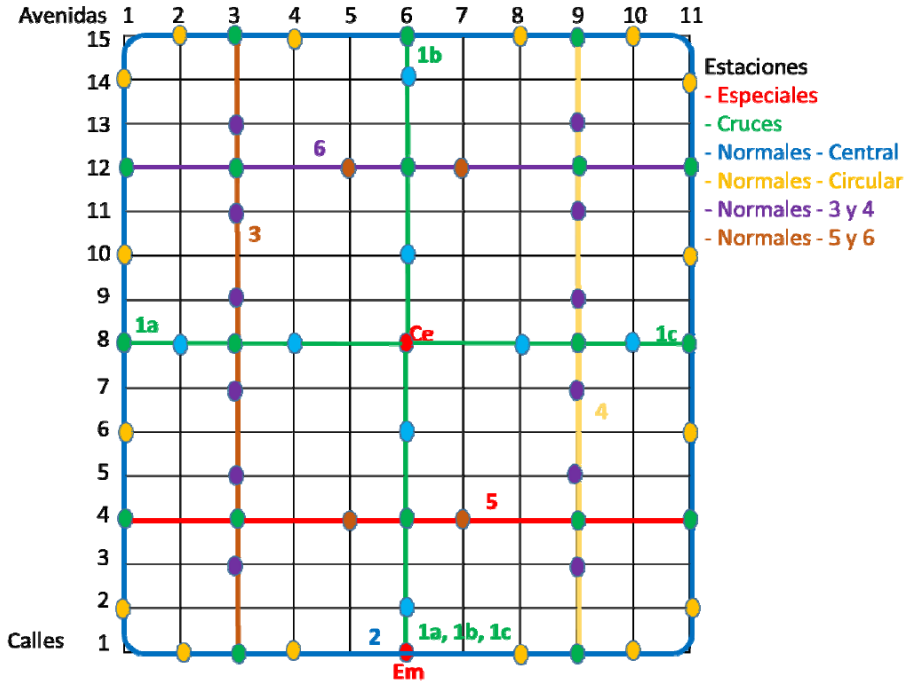
> Línea 3: (3,3), (3,7), (3,9), (3,11) y (3,13)

> Línea 4: (9,3), (9,7), (9,9), (9,11) y (9,13).

> Línea 5: (5,4) y (7,4).

> Línea 6: (5,12) y (7,12).

En la figura 2 se muestra la ubicación de las estaciones según el tipo.



En relación a la definición de las estaciones de la red disponen de dos posibilidades:
a/ Construir un algoritmo que dado (m,n) evalúe todas las estaciones, dando su posición en la cuadrícula.
b/ Para cada caso de prueba, calcular "a mano" las estaciones e introducirlas como datos de entrada, por ejemplo desde un fichero.

Para el cálculo de los tiempos vamos a considerar las siguientes condiciones y simplificaciones:

- Las manzanas son cuadradas, con un lado "l" (ej. 300 pies), tardándose en recorrer esa distancia por término medio un tiempo "t1" (ej. 4 minutos).
- En los intercambios (cambio de línea), se tarda un tiempo medio "t2" (ej. 7 minutos).
- El tiempo de espera medio al llegar a una estación es "t3" (ej. 5 minutos).

Los tiempos se tratan en forma de media, en el caso real habría que modificarlos en función del horario, teniendo en cuenta la frecuencia de los trenes y el tiempo distinto que se tarda en descargar y cargar los pasajeros, según la hora (hora punta, hora plana y hora valle) y las estaciones (unas tienen mayor aglomeración que otras). En nuestro caso vamos a simplificar estos datos tomando valores medios.

El objetivo es calcular para una estación dada (mi, ni) el tiempo que se tarda en llegar a cualquier otra estación de la red, dando a su vez la recomendación de intercambios a hacer (en qué estación cambiar y a qué línea). Se supone que un usuario puede consultar en un panel los tiempos o través del móvil.

Vamos a suponer tiempos únicos como se ha indicado anteriormente, en la realidad habría tiempos distintos en función de la hora y de las estaciones.

Se ha de generar la tabla de enlaces y tiempos para una estación dada (mi,ni) utilizando el esquema "Greedy", detallando si se cumplen las condiciones, y el tipo de Greedy que es, es decir, si obtiene la solución óptima siempre o es aproximado o heurístico. Se han de definir las funciones de optimización, que en algún caso hay que discutir pues pueden no ser evidentes.

GENERACIÓN DE LOS JUEGOS DE PRUEBA

En primer lugar, dado la presentación del problema era necesario implementar una clase que se encargue de generar juegos de prueba, tal y como se define en el guion, generándose la ciudad.

Para esta práctica necesitamos generar la ciudad con sus correspondientes líneas tal y como se describe en el guión de la práctica. Es decir, una línea circular que rodea la ciudad, una línea principal o main que se divide en tres ramas justo a partir del punto céntrico de la ciudad, y 4 líneas más, una en cada punto cardinal (N-S-W-E) a partir de la principal, que atraviesa la ciudad de forma perpendicular.

Seguidamente, utilizamos la clase GenerarCiudad para generar dicha ciudad con un número aleatorio de calles y avenidas, comprendiendo entre 50 y 500 de cada una de ellas.

Las avenidas y calles se pasan como parámetro en el constructor de la clase Ciudad, que se encargará de generar cada una de las líneas de la ciudad con sus respectivos postes.

Además, cuando generamos la ciudad, el método generarFallos() de la clase Ciudad inicializa un fallo en cada una de las líneas de la ciudad de forma aleatoria.

Por último, guardamos el archivo ciudad.txt, que contiene:

- Número de calles generadas
- Número de avenidas generadas
- Distancia entre los postes
- Posición del punto em (punto de embarque)
- Posición del punto ce (punto central)
- Número de líneas generadas
- Confirmación de las líneas generadas

CLASE CIUDAD

Nuestra clase ciudad tiene un total de 8 atributos, dos constantes denominadas Tamaniocalle y Tamanioavenida(double) que tienen un valor de 50, numCalles(int), numAvenidas(int), distPostes(int), eM(Ubicación), cE(Ubicación), lineas (Arraylist <línea>). Debemos aclarar que Ubicación se trata de una clase genearada por nosotros donde se almacena la posición.

CONSTRUCTOR:

En cuanto al constructor, recibe el nombre del fichero que generamos con anterioridad. Recibe como parámetros el número de calles y avenidas. Además, se genera la distancia de postes de forma aleatoria y se inicializa la ubicación de eM y cE.

Finalmente, se llama al método "generarLineasCiudad()" para generar la ciudad, como hemos descrito anteriormente.

```
public Ciudad (int nCalles, int nAvenidas) {
    this.numCalles = nCalles;
    this.numAvenidas = nAvenidas;
    distPostes = (int) (Math.random() * 5);
    eM = new Ubicacion (1, nAvenidas/2);
    cE = new Ubicacion (nCalles/2, nAvenidas/2);
    lineas = new ArrayList<Linea>();
    generarLineasCiudad ();
}
```

METODOS:

- *generarLineasCiudad():* Este método será el encargado de registrar cada una de las ubicaciones de los postes de las respectivas líneas en las calles y avenidas correspondientes y se genera un *Arraylist* de postes que se añadirán a cada una de las líneas.
- *generarFallos():* Para cada una de las líneas de la ciudad, se introduce un fallo en uno de los postes de forma aleatoria.
- *guardarCiudad(String directorio):* Se almacena la ciudad en un archivo .txt para su posterior uso en el cálculo de los tiempos.

CLASE LINEA

Nuestra clase Linea tiene un total de 5 atributos, nombreLinea (String), distancia (double), numeroPostes(int), longitud(double), postes (Arraylist-postes).

CONSTRUCTOR:

En cuanto al constructor, recibe el nombre del fichero que generamos con anterioridad. Asignamos a cada atributo los valores de las líneas de dichos ficheros.

METODOS:

- *Close(int i):* Este método devuelve el estado del poste i, es decir, si hay o no hay corriente.
- *EncontrarErrorDyV(int ini, int fin):* Basándonos en el esquema de divide y vencerás de división, realiza una búsqueda binaria hasta encontrar el poste por cual no pasa corriente siempre que su predecesor sí tenga corriente.

```
private Poste encontrarErrorDyV(int ini, int fin)
    if (ini >= fin) {
        //if (postes[ini].close())
        if (close(ini) && ini != postes.size()-1)
            return postes.get(ini+1);
        return postes.get(ini);
    }
    int med = (ini + fin) / 2;
    //if (postes[med].close()) {
    if (close(med)) {
        return encontrarErrorDyV (med + 1, fin);
    }
    return encontrarErrorDyV(ini, med-1);
}
```

- *EncontrarErrorVoraz():* Basándonos en el esquema Voraz, buscamos el poste del error y lo devolvemos.

```
public Poste encontrarErrorVoraz () {
    int posicion = 0;
    Poste solucion = null;
    while (solucion == null) {
        if (posicion >= postes.size())
            return null;
        Poste x = postes.get(posicion);
        posicion++;
        if ( ! x.close())
            solucion = x;
    }
    return solucion;
}
```

CLASE COMPROBARLINEAS

Nuestra clase ComprobarLineas se ha creado con el propósito de calcular los tiempos de para encontrar el poste de error, se realizan un total de 10 repeticiones para obtener el tiempo medio y además se calcula tanto el tiempo del algoritmo DyV como el del algoritmo Voraz.

Se imprime por consola lo siguiente:

```
<terminated> ComprobarLineas (1) [Java Application] C:\Program Files\Java\jre1.8.0_144\bin\javaw.exe (31 de may. de 2018 17:59:30)
Linea              N° postes       Error       TiempoDyV         TiempoVZ
Linea1a                12704        4053            7307           478391
Linea1b                 5408        3411            4639           309129
Linea1c                12729        8650           14163           212531
Linea2                 50841       28252            3366           441402
Linea3                  5408        5179            1724           185600
Linea4                  5408        4852            1642           181495
Linea5                 20025        8770            2504           346487
Linea6                 20025        5452            1600            11987
```

Y se guardan los datos en un archivo de texto, que se utilizará para crear las gráficas con Excel.

CLASE VERTICE

Esta clase se encarga de representar los vértices del grafo para luego utilizar el algoritmo Dijkstra que recorrerá los caminos y calculará los tiempos. Tiene un total de 3 atributos, nombre(String), ubicacion(Ubicacion) y cruce (boolean).

CONSTRUCTOR:
En cuanto al constructor, recibe como parámetro nombre, ubicación y cruce, y los inicializa.

METODOS:
- *isCruce():* Comprueba si un vértice (poste), es un cruce con otra línea.

CLASE ARISTA

Esta clase se encarga de representar las aristas del grafo (de un poste a otro) para luego utilizar el algoritmo Dijkstra que recorrerá los caminos y calculará los tiempos. Tiene un total de 5 atributos, tiempoArista (static double), origen (Vertice), destino (Vertice), longitud (double) y nombreLinea (string).

CONSTRUCTOR:
En cuanto al constructor, recibe como parámetro origen, destino, longitud y nombreLiena, y los inicializa.

CLASE CAMINO

Esta clase se encarga de representar los caminos del grafo formados por un conjunto vértices y aristas, además de calcular el tiempo de dicho camino. Tiene un total de 6 atributos, 3 de ellos constantes que son tiempoManzana (double), tiempoIntercambio (double), y tiempoEstación (double). Por otra parte tenemos listaVertices (Arraylist <Vertice>), listaArista (Arraylist <Arista>) y tiempo (double).

CONSTRUCTOR:
En cuanto al constructor, recibe como parámetros lista (Arraylist <Vertice>) y grafo (Grafo). Se inicializa la lista de aristas introduciendo todas las aristas existentes en el grafo y se calcula el tiempo de esa arista.

METODOS:

- *calcularLista(Grafo grafo):* Recorre todos los vértices del grafo, seleccionando las aristas entre cada vértice y añadiéndola a la lista de aristas.
- *calcularTiempo():* Como su nombre indica, calcula el tiempo de una determinada arista, teniendo en cuenta el TIEMPOMANZANA, y si se trata de un cruce o no también se ha de estimar el TIEMPOINTERCAMBIO o TIEMPOESTACION.

```
private double calcularTiempo() {
    tiempo = 0;
    for (int i=0; i<listaVertices.size(); i++) {
        Arista a = listaAristas.get(i);
        tiempo += TIEMPOMANZANA * a.getLongitud();
        if (i < listaVertices.size()-1) {
            if (a.getDestino().isCruce())
                tiempo += TIEMPOINTERCAMBIO;
            else
                tiempo += TIEMPOESTACION;
        }
    }
    return tiempo;
}
```

CLASE GRAFO

Esta clase se encarga de inicializar un grafo o mapa a partir de las calles, avenidas y postes con su ubicación. Tiene un total de 6 atributos, con una constante denominada MAX, grafo (TreeMap<Vertice, TreeMap<Vertice, Arista>>), distancia (TreeMap<Vertice, Double>), previo (TreeMap<Vertice, Vertice>), numeroCalles (int) y numeroAvenidas (int).

CONSTRUCTOR:

En cuanto al constructor recibe como parámetro de entrada el nombre del fichero de donde se recogen los datos, recorremos cada uno de los vértices y construimos el mapa de adyacencias.

METODOS:

- *getArista(Vertice origen, Vertice destino):* Devuelve la arista comprendida entre dos vértices, devolviendo nulo en caso de que éstos no existan.
- *dijkstra(Vertice origen):* Dado un determinado vértice o punto de origen, encuentra los caminos de coste mínimo (menor tiempo) a todos los vértices restantes, almacenándolos en el TreeMap<Vertice, Double> distancia.

```
private void dijkstra (Vertice origen) {
    TreeSet<Vertice> s = new TreeSet<Vertice>();
    s.add (origen);
    distancia = new TreeMap<Vertice, Double> ();
    previo = new TreeMap<Vertice, Vertice> ();
    for (Vertice v : grafo.keySet()) {
        Arista a = getArista (origen, v);
        if (a == null) {
            distancia.put(v, MAX);
            previo.put(v, null);
        }
        else {
            distancia.put(v, a.getLongitud());
            previo.put(v, origen);
        }
    }
    TreeSet<Vertice> vmenoss = new TreeSet<Vertice> (grafo.keySet());
    vmenoss.remove(origen);
    while (vmenoss.size() > 0) {
        Vertice w = extraerVertice (vmenoss, distancia);
        System.out.println("Despues de extraer: "+w);
        s.add(w);
        vmenoss.remove(w);
        for (Vertice v : vmenoss) {
            Arista a = getArista (w, v);
            if (a != null) {
                if (distancia.get(w) + a.getLongitud() < distancia.get(v)) {
                    distancia.put(v, distancia.get(w) + a.getLongitud());
                    System.out.println("Cambiando previo de "+v+" a "+w);
                    previo.put(v, w);
                }
            }
```

- *extraerVertice(TreeSet<Vertice> vMenos, TreeMap<Vertice, Double> distancia2):* Recorre cada uno de los vértices de vMenos, seleccionando aquel cuya distancia sea la mínima.

```
public TreeMap<Vertice, ArrayList<Vertice>> obtenerCaminos
                        (Vertice origen) {
    dijkstra (origen);
    TreeMap<Vertice, ArrayList<Vertice>> salida =
                new TreeMap<Vertice, ArrayList<Vertice>>();
    for (Vertice v : grafo.keySet()) {
        //System.out.println("mirando: "+v);
        if (v.equals(origen)) {
            //System.out.println("comparando "+v+" con "+origen);
            salida.put(v, null);
        }
        else {
            ArrayList<Vertice> lista = new ArrayList<Vertice> ();
            lista.add(origen);
            //System.out.println("antes: "+lista);
            hacerCamino (origen, v, lista);
            lista.add(v);
            //System.out.println("despues: "+lista);
            salida.put(v, lista);
        }
```

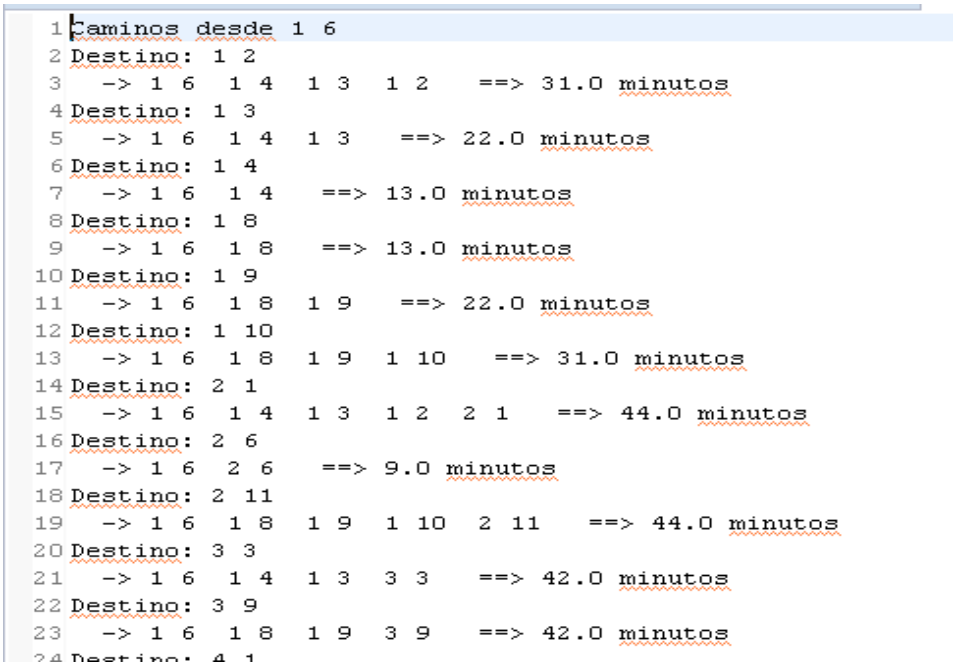
- *obtenerCaminos(Vertice origen):* Obtiene todos los caminos mínimos desde el vértice de origen al resto de vértices.
- *hacerCamino(Vertice origen, Vertice v, Arraylist<Vertice> lista):* Muestra el camino desde un vértice de origen a un vértice de destino.

- *calcularTiempo(Arraylist<Vertice> lista):* Calcula el tiempo de un vértice a otro, sumando tantos tiempos de manzanas como haya, además de sumar el tiempo de espera de la estación y el posible tiempo de intercambio de línea, si lo hubiese.

```
public double calcularTiempo(ArrayList<Vertice> lista) {
    double tiempo = 0, tManzana=Camino.TIEMPOESTACION, tIntercambio=Camino.TIEMPOINTERCAMBIO,
            tEstacion=Camino.TIEMPOMANZANA;
    //estacion inicial: t3
    tiempo = tiempo + tEstacion;
    Vertice anterior = lista.get(0);
    Arista aAnterior = null;
    //para el resto de estaciones
    for(int i=1; i<lista.size(); i++) {
        Vertice actual = lista.get(i);
        Arista aSiguiente = getArista(anterior, actual);
        tiempo = tiempo + aSiguiente.getLongitud()*tManzana;
        if (i < lista.size() - 1)
            tiempo = tiempo + tEstacion;
        if (aAnterior != null) {
            if ( ! aAnterior.getNombreLinea().equals(aSiguiente.getNombreLinea()))
                tiempo = tiempo + tIntercambio;
        }
        aAnterior = aSiguiente;
        anterior = actual;
    }
    return tiempo;
}
```

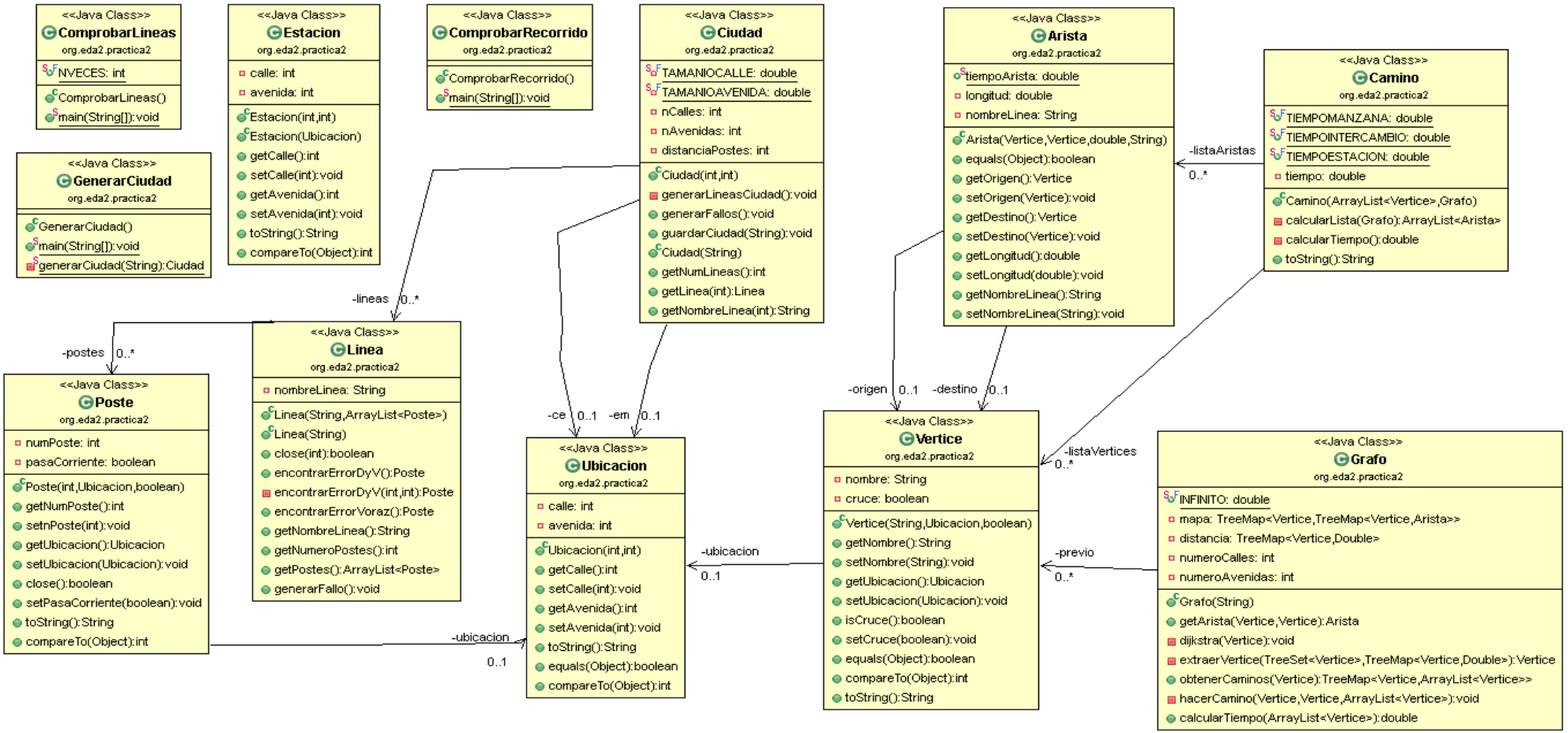
CLASE COMPROBARRECORRIDO

Nuestra clase ComprobarRecorrido se ha creado con el propósito de obtener todos los caminos posibles desde un vértice de origen a uno de destino, para ello hacemos uso del método *obtenerCaminos* de la clase "Grafo" que a su vez usa el método de Dijkstra para encontrar los caminos mínimos. Una vez que tenemos los caminos al destino dilucidamos el tiempo de cada uno de ellos usando el método *calcularTiempo* también de la clase "Grafo". Imprimimos dichos resultados por pantalla y los guardamos en un fichero txt denominado tablaRecorrido.txt, tal y como se muestra a continuación:



```
Caminos desde 1 6
Destino: 1 2
  -> 1 6  1 4  1 3  1 2   ==> 31.0 minutos
Destino: 1 3
  -> 1 6  1 4  1 3   ==> 22.0 minutos
Destino: 1 4
  -> 1 6  1 4   ==> 13.0 minutos
Destino: 1 8
  -> 1 6  1 8   ==> 13.0 minutos
Destino: 1 9
  -> 1 6  1 8  1 9   ==> 22.0 minutos
Destino: 1 10
  -> 1 6  1 8  1 9  1 10   ==> 31.0 minutos
Destino: 2 1
  -> 1 6  1 4  1 3  1 2  2 1   ==> 44.0 minutos
Destino: 2 6
  -> 1 6  2 6   ==> 9.0 minutos
Destino: 2 11
  -> 1 6  1 8  1 9  1 10  2 11   ==> 44.0 minutos
Destino: 3 3
  -> 1 6  1 4  1 3  3 3   ==> 42.0 minutos
Destino: 3 9
  -> 1 6  1 8  1 9  3 9   ==> 42.0 minutos
Destino: 4 1
```

DIAGRAMA DE CLASES



ANÁLISIS TEÓRICO DEL TIEMPO DEL ALGORITMO GREEDY

Comenzaremos calculando el análisis teórico del tiempo de encontrar el poste de error usando el algoritmo greedy:

```
public Poste encontrarErrorVoraz () {
    int posicion = 0;
    Poste solucion = null;
    while (solucion == null) {
        if (posicion >= postes.size())
            return null;
        Poste x = postes.get(posicion);
        posicion++;
        if ( ! x.close())
            solucion = x;
    }
    return solucion;
}
```

El mecanismo general del cálculo de la complejidad podría resumirse como:

- 1) Sea **n**: número de elementos de C; **m**: número de elementos de una solución
- 2) Repetir como máximo n veces y como mínimo m
  - Comprobar si el valor actual es solución: f(m). Normalmente O(1) ó O(m).
  - Selección de un elemento entre los candidatos: g(n). Entre O(1) y O(n).
  - Tiempo de la función factible, parecida a la solución: h(m)
  - La unión de un nuevo elemento a la solución puede requerir de otras operaciones de cálculo: j(n,m)

Tiempo de ejecución genérico es:

$$t(n) \in \theta(n^{*}(f(m) + g(n) + h(m)) + m*j(n, m))$$

En nuestro caso:

- Tiempo calcular solución: f(m) => O(1)
- Selección de un elemento: g(n) => O(1)
- Tiempo función factible: h(m) => O(1)
- Tiempo de unión de un elemento a la solución: O(1)

Por lo que tenemos:

$$t(n) \in \theta(n^{*}(\theta(1) + \theta(1) + \theta(1)) + m*\theta(1) = \theta(n)$$

Para el problema b(dada una estación, calcular el tiempo que se tarda en llegar a cualquier otra estación de la red) el análisis teórico de eficiencia sería el siguiente:

```
private void dijkstra (Vertice origen) {
     TreeSet<Vertice> s = new TreeSet<Vertice>();
     s.add (origen);
     distancia = new TreeMap<Vertice, Double> ();
     previo = new TreeMap<Vertice, Vertice> ();
     for (Vertice v : grafo.keySet()) {
          Arista a = getArista (origen, v);
          if (a == null) {
               distancia.put(v, MAX);
               previo.put(v, null);
          }
          else {
               distancia.put(v, a.getLongitud());
               previo.put(v, origen);
          }
     }
     TreeSet<Vertice> vmenoss = new TreeSet<Vertice> (grafo.keySet());
     vmenoss.remove(origen);
     while (vmenoss.size() > 0) {
          Vertice w = extraerVertice (vmenoss, distancia);
          System.out.println("Despues de extraer: "+w);
          s.add(w);
          vmenoss.remove(w);
          for (Vertice v : vmenoss) {
               Arista a = getArista (w, v);
               if (a != null) {
                    if (distancia.get(w) + a.getLongitud() < distancia.get(v)) {
                         distancia.put(v, distancia.get(w) + a.getLongitud());
                         System.out.println("Cambiando previo de "+v+" a "+w);
                         previo.put(v, w);
                    }
               }
          }
```

- a) Fase de inicialización:
  - Creación del conjunto y ejecución n veces de diversas operaciones constantes: O(n)

- b) Fase de selección:
  - Nº de ejecuciones: n(n-1)/2-1 $\in$ O(n^2)

- c) Fase de contabilizar lo tratado:
  - N supresiones a lo largo del algoritmo: O(n)

- d) Fase de re-cálculo de las distancias mínimas:
  - Queda O(n^2) al igual que la selección

} O(n^2)

Finalmente tendríamos:

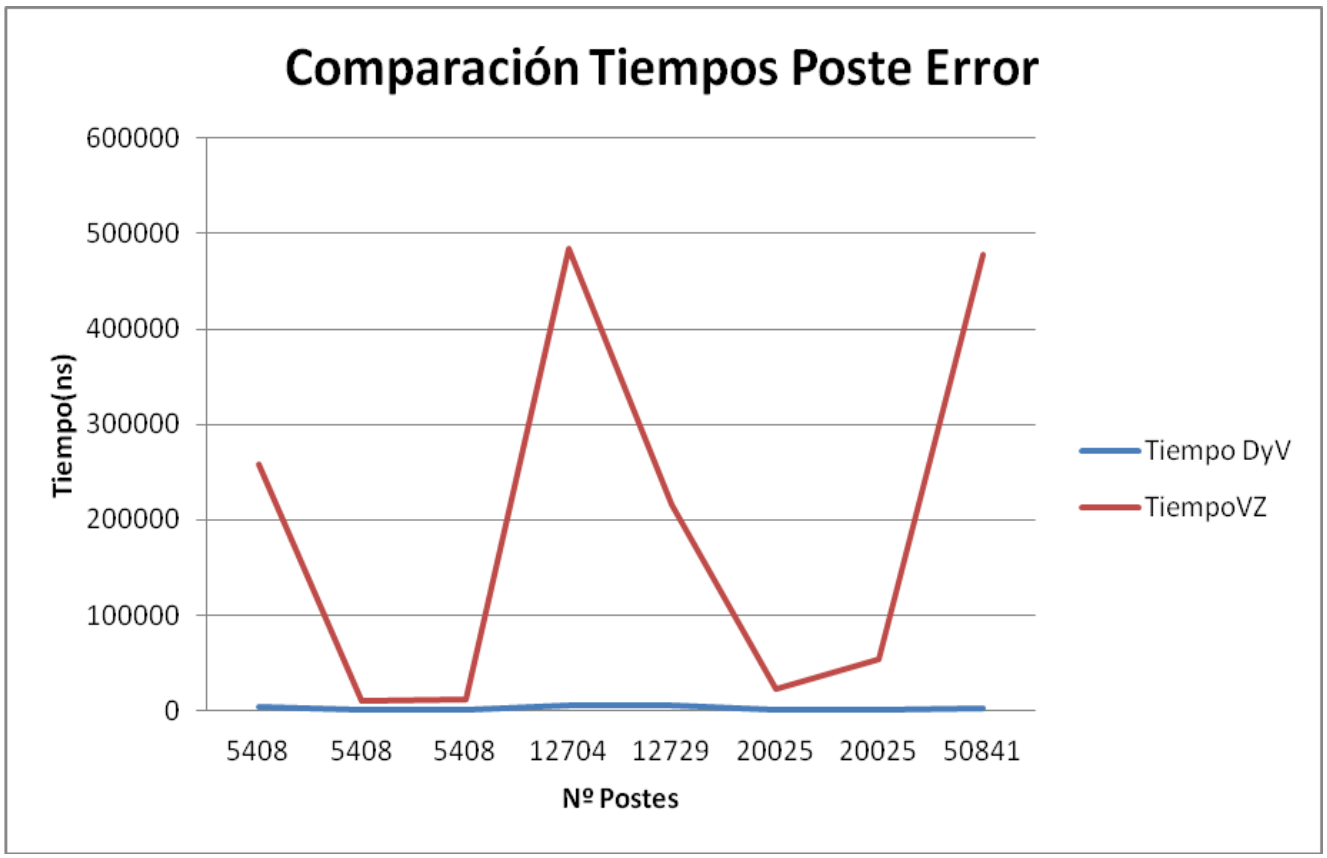
$$t(n) \in \theta(n^2)$$

COMPARACIÓN DE TIEMPOS ENCONTRAR POSTE DE ERROR ENTRE EL ALGORITMO DYV Y GREEDY

Realizamos un total de diez ejecuciones midiendo los tiempos medios de cada uno de ellos. A continuación se muestran los tiempos medios de las diez ejecuciones tanto para el algoritmo DyV como para el Greedy:

Linea	Nº Postes	Error	Tiempo DyV	TiempoVZ
Linea2b	5408	3411	3571	257689
Linea 3	5408	5179	1231	10961
Linea 3	5408	4852	1519	12315
Linea1a	12704	4053	6527	483851
Linea1c	12729	8650	6404	216062
Linea 5	20025	8770	1518	22620
Linea 6	20025	5452	1806	53697
Linea 2	50841	28252	2463	478596

En la siguiente gráfica podemos observar la diferencia de tiempos en la búsqueda del poste de error entre el algoritmo de divide y vencerás y el voraz. Como podemos observar el tiempo que tarda el algoritmo DyV es sustancialmente menor al Greedy.



PRECONDICIONES Y POSTCONDICIONES

- Precondiciones:

  - *GenerarCiudad:*
    - El número de calles y avenidas tiene un rango entre 50 y 500.
    - Las manzanas de la ciudad serán cuadradas con tamaño de calles y avenidas igual a 50 metros.
    - La distancia entre los postes de cada línea oscilará entre 0 y 5 metros de longitud.
    - Para cada una de las líneas generadas, se creará un fallo de forma aleatoria en alguno de sus postes.
    - A cada poste de las líneas, se le asigna 1 si pasa corriente o 0 si no pasa dicha corriente, según la posición de error calculada previamente.
    - Se generarán tantos archivos .txt como líneas se hayan generado además del propio .txt de la ciudad.
    - La estación de embarque (em), siempre se encontrará en la ubicación numerada como calle 1 y en la avenida central de la ciudad (numAvenidas/2).
    - La estación central (ce) siempre se encontrará en el punto céntrico de la ciudad (numCalles/2, numAvenidas/2).

  - *EncontrarError* (*Voraz*):
    - Debe existir el conjunto solución inicializado a null.
    - Si el poste es localizado, éste se añade a la solución, y en caso de no ser localizado, se devuelve null.

  - *GenerarLineasCiudad*:
    - El número de avenidas será igual al tamaño de la avenida dividido entre la distancia de los postes.
    - El número de calles será igual al tamaño de la calle dividido entre la distancia de los postes.
    - Debe de haber estaciones en las siguientes posiciones:
      - En todos los cruces de las distintas líneas
      - Cada dos cruces avenida-calle en la línea central partiendo de ce.
      - Cada dos cruces avenida-calle en la línea circular partiendo de em.
      - Partiendo del límite sur (calle 1), cada dos cruces en las líneas 3 y 4.
      - Partiendo del límite oeste (avenida 1), cada dos cruces en las líneas 5 y 6.
      - En los puntos em y ce.

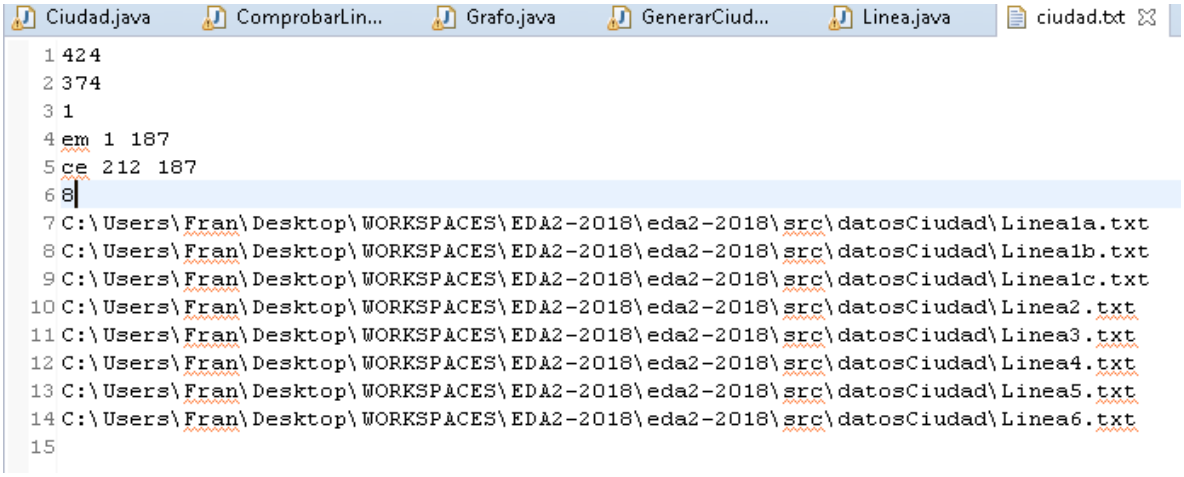
- *Grafo*:
  - Necesitaremos un TreeMap<Vertice, TreeMap<Vertice, Arista>> mapa para representar las líneas de metro.
  - El algoritmo solo funcionará correctamente si los datos que recibe han sido pre-procesados.
  - Asignamos un tiempo de 4 minutos para recorrer una manzana, de 7 minutos para los cambios de línea y de 5 minutos para el tiempo en llegar a una estación, importante a la hora de calcular el tiempo en ir de una estación (origen) a otra (destino).

➢ Postcondiciones:

- *Comprobar Lineas:*
  - Muestra por pantalla la diferencia de tiempos para encontrar el poste de error de cada una de líneas utilizando el algoritmo DyV frente al Greedy.
  - Se genera un solo archivo denominado *"Comparación.txt"* que almacena las diferencias de tiempos anteriormente mencionadas.
  - La primera columna del archivo de tiempo corresponderá al nombre de la Línea.
  - La segunda columna del archivo de tiempo corresponderá al total de postes de la línea.
  - La tercera columna del archivo de tiempo corresponderá al poste de error.
  - La cuarta columna del archivo de tiempo corresponderá al tiempo que tarda el algoritmo DyV en calcular el poste de error.
  - La quinta columna del archivo de tiempo corresponderá al tiempo que tarda el algoritmo Greedy en calcular el poste de error.

- *Encontrar error(Voraz):*
  - Encontrar error se ejecutará correctamente siempre y cuando exista un poste por el cual no pasa corriente si no dará una excepción y mostrará el mensaje: *"No se ha encontrado error"*
  - Debe encontrar solo un poste de error.
  - Devolverá el poste de error indicando su posición.

- *GenerarCiudad:*
  - Se guardará un archivo denominado *"ciudad.txt"* dentro del paquete datosCiudad que contendrá el nº de calles y avenidas, la distancia entre postes las ubicaciones de las estaciones Em y Ce, la cantidad de líneas y por último cada una las direcciones de los archivos txt de las respectivas líneas. El contenido del archivo es el siguiente:



- *GenerarLineasCiudad*:
  - A la hora de inicializar una ciudad se llamará al método generarLineasCiudad() que guardará el contenido de cada una de las líneas en un fichero txt denominados: "Linea1a.txt", "Linea1b.txt", "Linea1c.txt", "Linea2.txt", "Linea3.txt", etc.
  - El contenido de cada uno de estos ficheros será, en primer lugar el nº total de postes de dicha línea y a continuación cada uno de los postes con su ubicación y true (si pasa corriente) o false (en caso de que no pase corriente).

- *ComprobarRecorrido:*
  - Tras calcular todos los caminos desde un vértice de origen a otro de destino (de una estación a otra dada), se mostrará por pantalla cada uno de los caminos y el tiempo que tarda cada uno de ellos. Además de mostrar otra información como la iteración del algoritmo para seguir los pasos que realiza. Será fundamental guardar dichos caminos y alojarlos en un archivo de texto denominado *"tablaRecorrido.txt"*.