ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DΣIM]

Estructura de Datos

Práctica 2 – Grafo Etiquetado y No Dirigido

Curso 2021-22

Estudiante: Cristian Fernández López

Profesor/a: Cristina Llort

Grupo: L3

Fecha de entrega: 20/06/2022

Tabla de Contenido

1. Contenidos

1.1 Introducción

La práctica 2 consiste en la realización de un grafo genérico etiquetado y no dirigido, para ello se implementará las funciones básicas para la construcción y consulta de un grafo. La segunda parte de la práctica es la implementación del grafo genérico para un problema específico, crear una red de carreteras que unen las estaciones de recarga a partir de la base de datos de los puntos de recarga de Cataluña, además habrá que realizar la lectura de un formato JSON para cargar los datos a la aplicación.

En la tercera parte habrá que implementar dos algoritmos que trabajaran con la estructura del grafo de zonas de recarga. Y, por último, en la parte opcional de la práctica se analizará el coste temporal del algoritmo de camino óptimo.

1.2 Organización de Paquetes del Programa

1.2.1 Grafo Genérico

Como se puede en la **figura 1.2.1** se muestra las clases realizadas para el funcionamiento y la comprobación del grafo genérico. Es decir, se han usado para realizar el grafo: 1 interfaz, 3 clases, 2 excepciones. Para realizar el juego de pruebas se ha usado una clase de programa principal. Adicionalmente, para esta parte se necesita de la Tabla Hash realizada en la práctica 1.

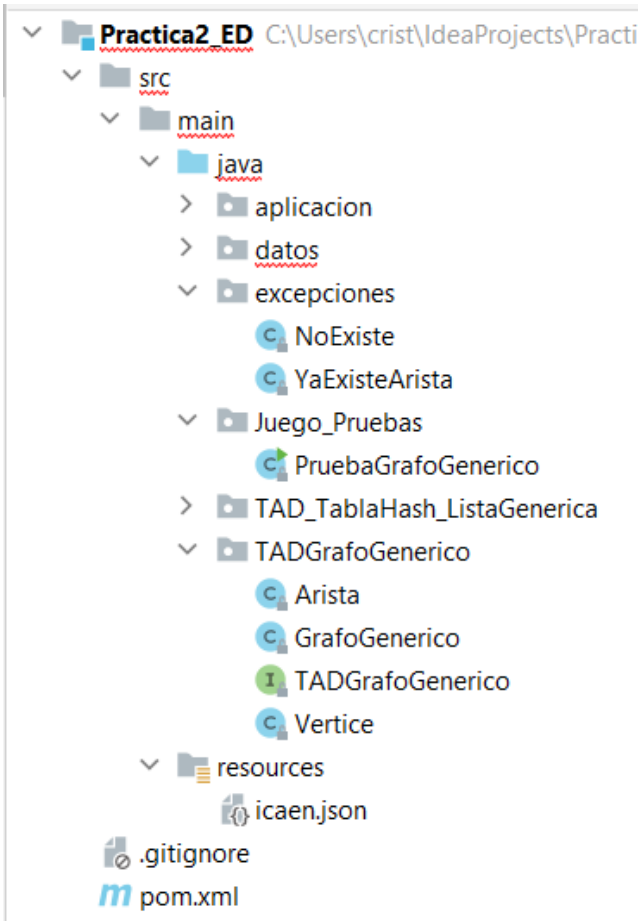


Figura 1.2.1. Estructura de paquetes y clases del Grafo Genérico

A continuación, se realiza una breve descripción del objetivo de cada clase del programa.

***TADGrafoGenerico*:** Interfaz del grafo genérico, la clase contiene los métodos que se marcan en las especificaciones del enunciado de la práctica.

***GrafoGenerico*:** Clase genérica, contiene los datos del grafo agrupa la tabla hash con los vértices y la multilista de adyacencia para las aristas.

***Vertice*:** Clase genérica, que es un nodo que almacena la información de un vértice y además contiene los 2 punteros a las aristas. Una para las aristas horizontales y otra para las aristas verticales.

***Arista*:** Clase genérica, que contiene la etiqueta y que además tiene los punteros a las siguientes aristas horizontales y verticales. Adicionalmente, tiene referencias a los vértices que une.

***NoExiste*:** Excepción propia. Se lanza al pasar por parámetro vértices cuyo valor sea nulo.

***YaExisteArista*:** Excepción propia. Se lanza en algunas funciones cuando no se puede añadir una arista debido a que ya existe la arista que une a esos dos vértices. No se actualiza el valor de la arista, se queda con el valor inicial.

***PruebaGrafoGenerico*:** Juego de Pruebas de la clase Grafo Genérico, se comprueba el correcto funcionamiento de las distintas funciones que proporciona la clase.

1.2.2 Grafo Estaciones y Zonas de Recarga

Para la segunda parte de la práctica, tiene la organización mostrada en la **figura 1.2.2.** Se ha usado 2 clases básicas que almacenan los datos, 1 clase que almacena los datos de las clases anteriores en forma de grafo (haciendo uso del Grafo Genérico realizado en la parte anterior) y 1 clase que es el programa principal que sirve Juego de Pruebas de las funciones y algoritmos del grafo de estaciones.

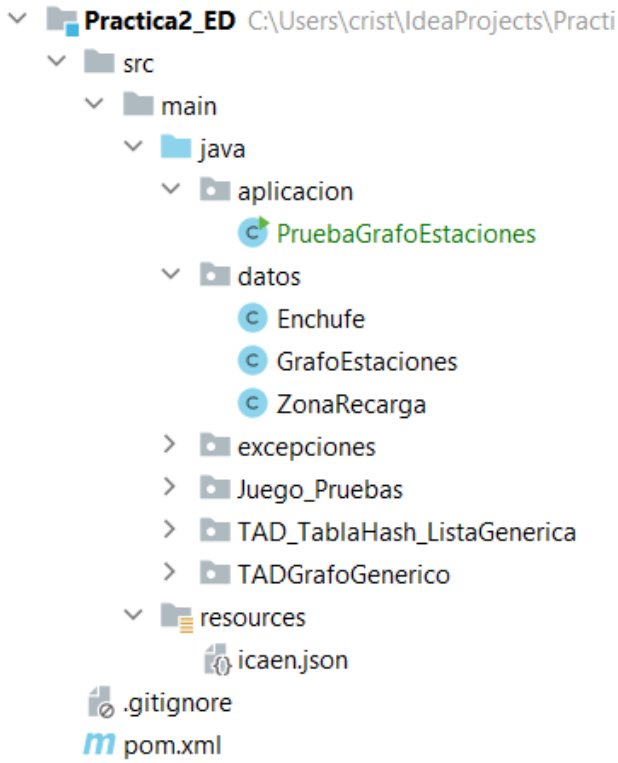


Figura 1.2.2. Estructura de paquetes y clases de la implementación del Grafo Estaciones

A continuación, la explicación de cada clase que se ha usado:

GrafoEstaciones: Clase que contiene el grafo genérico implementando las zonas de recarga. Implementa las funciones necesarias para construir el grafo y además añade los algoritmos pedidos en el enunciado.

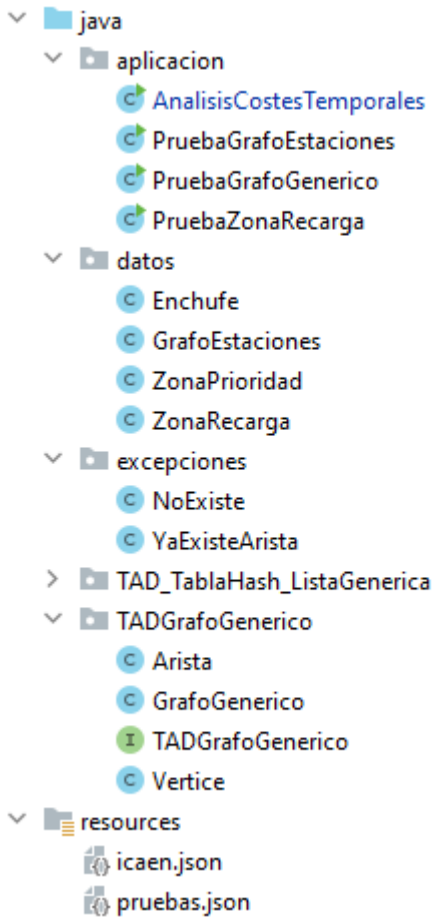
***Enchufe*:** Clase básica que contiene los datos de un enchufe de recarga. Los atributos más importantes que almacena son su id, las coordenadas, y la potencia.

ZonaRecarga: Clase que contiene una lista de los enchufes que comparten las mismas coordenadas. Es usado como nodos del grafo de estaciones. Tiene como id, el id del primer enchufe que se une con esas coordenadas.

PruebaGrafoEstaciones: Juego de Pruebas de la clase GrafoEstaciones

1.2.3 Estructura Final

La estructura final resultante del proyecto es la siguiente:



1.3 Análisis y diseño del programa

1.3.1 Grafo Genérico

A continuación, se detalla algunos de los aspectos más importantes de las decisiones tomadas en la implementación del grafo genérico.

En la cabecera de *GrafoGenerico* (**figura 1.3.1**), se puede observar que se usan 3 parámetro genéricos esto se debe a que como he decidido guardar los vértices en una tabla hash (para que el coste de acceder sea constante) es necesario tener una clave para el valor del vértice. Por tanto, en vez de tener solo dos parámetros genéricos, uno para los datos del vértice (que debería implementar Comparable, para asegurar que tiene un identificador) y otro que sería para guardar la etiqueta de las aristas. He optado por usar 3 genéricos de forma que separemos el identificador del vértice del resto de datos, de esta forma se podría se facilita el uso de muchas de las funciones del grafo ya que no necesitan trabajar con todos los datos del vértice si no que basta con saber el identificador.

```
Clase Grafo Generico, implementa TADGrafoGenerico, Proporciona una implementación para crear un grafo
etiquetado y no dirigido.
Type parameters: <K> – Tipo de Objeto que será la clave o identificador del vertice. Debe implementar
                 Comparable
                 <V> – Tipo de Objeto que será el valor del vertice
                 <A> – Tipo de Objeto que será el valor o etiqueta de la arista
public class GrafoGenerico<K extends Comparable<K>, V, A> implements TADGrafoGenerico<K, V, A> {

    TablaHashGenerica<K, Vertice<K, V, A>> tablaVertices;
```

Figura 1.3.1. Atributos de la clase *GrafoGenerico*

Otra característica adicional son los métodos adicionales que se han añadido a *GrafoGenerico* que no estaban presentes en *TADGrafoGenerico*, en la **figura 1.3.2** se pueden ver**.**

- agregarVertice(): Era necesario de una función que añadiera los vértices uno a uno en el grafo, otra opción podría haber sido pasar al constructor una lista de vértice.
- getMidaTablaVertices(): Es una función necesaria para los algoritmos del grafo estación, se necesita saber cual es el tamaño de la tabla hash para así poder crear las tabla hash auxiliares.
- getClavesVertice(): Función con las que se obtiene una lista con los identificadores de todos los vértices. Es necesaria también para usarla por los algoritmos ya que de esta forma se puede recorrer los vértices presentes en el grafo para crear las estructuras auxiliares.
- valorVertice(): Función que devuelve la información sobre el vertice. Es necesaria por algunas funciones del grafo estaciones para obtener la zona de recarga.
- buscarArista(): Es una función privada que es auxiliar del resto de funciones, se encarga de buscar una arista en la multilista y devuelve una excepción si no lo encuentra. Para hacer la búsqueda con mirar en la lista de aristas horizontales del nodo menor sirve. *En el siguiente párrafo se explica la multilista.

```java
    Función que agrega un vértice
    Params: vertice – - dato que se quiere añadir al grafo
    Throws: NoExiste – - El vértice pasado por parámetro es nulo
public void agregarVertice(K clave, V vertice) throws NoExiste{
    if (vertice == null || clave == null){
        throw new NoExiste( fraseError: "El vértice pasado por parámetro es nulo");
    }
    tablaVertices.insertar(clave, new Vertice<K, V, A>(clave, vertice));
}

    Obtener tamaño tabla hash de vertices
    Returns: la longitud de la tabla hash de vertices del grafo
public int getMidaTablaVertices(){
    return tablaVertices.midaTabla();
}

    Obtener Lista de Claves Vertices
    Returns: una lista con los identificadores de los vertices del grafo
public ListaGenerica<K> getClavesVertices(){
    return tablaVertices.obtenerClaves();
}

    Valor del vertice
    Params: idVertice – - identificador/clave del vertice
    Returns: valor del vertice que tiene ese id
    Throws: ClaveException – - en caso de que no exista el vertice en el grafo
public V valorVertice(K idVertice) throws ClaveException, NullPointerException {
    return tablaVertices.obtener(idVertice).getDatos();
}
```

Figura 1.3.2. Métodos adicionales de la clase GrafoGenerico

Explicación de clase Vértice y Arista:

Para entender como he realizado la implementación del grafo genérico, es necesario explicar que he hecho de una multilista de adyacencia y por tanto las aristas se almacenan en memoria dinámica. A diferencia de los vértices que se almacenan en una tabla hash y por tanto en memoria estática (aunque se redimensiona la tabla hash de forma dinámica a medida que aumenta el número de vértices y se supera el factor de carga). En la multilista de adyacencia, para optimizar el espacio en memoria utilizado al ser un grafo **no dirigido** para evitar repetidos se guarda las aristas en el vértice más pequeño, se guarda en la fila o puntero horizontal, en cambio en el vértice grande se actualiza el puntero vertical o columna. Al añadir la arista lo guardamos al principio de la lista de nodos de forma que el coste de añadir sea constante, (no sé guarda las aristas siguiendo ningún orden, solo se sigue el orden de cuando se han añadido).

El nodo vértice, guarda el identificador y el valor del vértice, además, se guarda 2 punteros. El puntero arista horizontal como se ha mencionado en el párrafo anterior se guarda una lista de las aristas del vértice con otros vértices que sean **mayores**. En cambio, en el puntero arista vertical se almacena una lista de las aristas del vértice con otros vértices que sean **menores**. En la **figura 1.3.3** se puede ver los atributos del nodo vértice.

```
private K clave;
private V dato;

private Arista<K, V, A> punteroAristaHorizontal;
private Arista<K, V, A> punteroAristaVertical;
```

Figura 1.3.3. Atributos de la clase *Vertice*

Para la clase nodo Arista, se guarda por una parte el valor de la etiqueta, y por otra los punteros. Hay 2 punteros que guardan la siguiente arista, uno para las aristas horizontales y otra para las verticales. Luego, hay otros 2 punteros que son referencias a los vértices que unen. En la **figura 1.3.4** se puede observar.

```
public class Arista<K extends Comparable<K>, V, A> {

    private A dato;

    // Punteros a las siguientes aristas del Vertice
    private Arista<K, V, A> punteroAristaHorizontal;
    private Arista<K, V, A> punteroAristaVertical;

    // Referencias a los vertices que unen
    private Vertice<K, V, A> referVerticeHorizontal; // Es el vertice menor
    private Vertice<K, V, A> referVerticeVertical; // Es el vertice mayor
```

Figura 1.3.4. Atributos de la clase *Arista*

Por último, hay que mencionar que tanto la clase arista y la clase vértice tienen implementado funciones que permiten modificar el valor de los punteros y de esta forma se puede añadir aristas.

1.3.2 Grafo Estaciones y Zonas de Recarga

Para explicar la Tabla Hash, detallaremos las 4 clases que hemos realizado (**figura 1.3.5**):

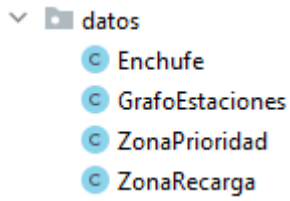


Figura 1.3.5 Clases de los datos para crear GrafoEstaciones

Enchufe. Se puede observar los datos que almacena el enchufe (**figura 1.3.6**). Es una clase que guarda los datos de cada enchufe cargado del fichero JSON. Los métodos de la clase son *Getter* para obtener la información necesitada por otras clases.

```
Clase que almacena los datos de cada enchufe de recarga
public class Enchufe {
    private final int id, id_estacio;
    private final String nom;
    private final String data;
    private final String consum,carrer,ciutat,estat;
    private final String temps, potencia;
    private final String tipus;
    private final double latitud,longitud;
```

```
public interface TADTablaHash<K extends Comparable<K>, T extends Comparable<T>> {
```

Figura 1.3.6 Atributos de la clase Enchufe

ZonaRecarga. Esta clase guarda todos los enchufes que comparten unas mismas coordenadas, en la **figura 1.3.7** se puede ver los atributos de la clase. El identificador de la zona será el id del primer enchufe que se añade a la zona de recarga. Esta clase contiene como funciones:
Distancia()**:** permite calcular la distancia entre dos zonas de recarga. Necesaria para construir el grafo de estaciones.
addEnchufe()**:** amplia la lista de enchufes de la zona de recarga solo en el caso de que tenga las misma coordenadas.
leerJSON(): lee el fichero JSON, es una función estática. Retorna una lista de las zonas de recarga del fichero.
getEnchufeMasPotencia(): es una función que retorna el enchufe con más potencia de la zona es necesario para el algoritmo de camino optimo del grafo estaciones.

```java
Clase que es una zona de recarga que contiene todos los enchufes que tienen las mismas coordenadas
public class ZonaRecarga implements Comparable<ZonaRecarga> {
    private final int id;
    private LinkedList<Enchufe> listaEnchufes;
    private final double latitud, longitud;
```

Figura 1.3.7 Atributos de la clase ZonaRecarga

ZonaPrioridad: Es una clase que se usa para la estructura auxiliar del algoritmo de camino optimo. La razón de que la necesitemos es para el funcionamiento de la cola de prioridad (PriorityQueue de Java). Guarda el identificador y el coste, (el coste es el que da la prioridad, a menor coste mayor prioridad).

GrafoEstaciones: Es una clase que almacena el grafo genérico con los datos de las zonas de recarga. Además, contiene las funciones necesarias para construir el grafo, y los algoritmos pedidos en el enunciado de la práctica. En la **figura 1.3.8** se puede ver los atributos.

```java
Clase que contiene una instancia del GrafoGenerico para guardar las estaciones de recarga. Contiene los
algoritmos camino optimo y zonas no garantizadas
public class GrafoEstaciones {
    private final GrafoGenerico <Integer, ZonaRecarga, Double> grafoEstaciones;
    private ZonaRecarga zonaRecargaInicial; // un vertice perteneciente al grafo por el cual se comienza a hacer un recorrido
```

Figura 1.3.8 Atributos de la clase GrafoEstaciones

Constructor (): Pasando una lista de zonas de recarga por parámetro, construye el grafo. Primero añade los vértices y luego revisa los vértices para añadir las carreteras (aristas) entre los vértices que estén a 40km y en caso de que un vértice no tenga uno a 40km lo une al más cercano.

CaminoOptimo(): Para encontrar la ruta mínima, he aplicado una implementación del algoritmo **Dijkstra**. Para ello he necesitado de usar como estructuras auxiliares una lista con los identificadores de las zonas de recarga, 3 tablas hash con las visitas, costes y predecesores y una cola de prioridad (min-heap). El algoritmo, empieza visitando la estación de origen y a partir de ella va modificando los costes (pero si el coste de la arista es superior a la autonomía no se actualiza) y predecesores de sus adyacentes. A la hora de escoger el siguiente vértice se elige aquella estación que tenga un coste menor y aún no este visitado. Al escoger un vertice se usa una cola de prioridad (min-heap), ya que de esta forma sería de coste logarítmico el obtener el siguiente vertice y no sería necesario recorrer todos lo vértices (coste lineal) para ver cual tiene un coste menor. Para usar la PriorityQueue de Java he tenido que crear la clase ZonaPrioridad.
En caso de que existan dos zonas con el mismo coste se elige aquella que tenga el enchufe con más potencia. El bucle se repite hasta que se alcance el destino o que todos los vértices se hayan visitado. Retorna una excepción, si los parámetros no son válidos o no se ha podido alcanzar el destino. Adicionalmente, esta función la he sobrecargado de forma que pasando por parámetro un puntero (StringBuilder) te añade al String la distancia recorrida. He usado un StringBuilder en vez de la clase String debido a que String es una clase inmutable y por tanto no sirve para obtener el resultado pasándolo por parámetro.

ZonasDistMaxNoGarantizada(): Para encontrar las estaciones a las que no se puede llegar uso un **recorrido en profundidad**. Es necesario como estructura auxiliar una tabla hash con las visitas, una lista de zonas (para crear la tabla hash) y una pila en la que se guarda las zonas adyacentes. En el algoritmo lo que se hace es en cada iteración del bucle se desapila una estación y se marca como visitada luego posteriormente se visita todos sus adyacentes y se apila aquellas estaciones que no hayan sido visitados y que cumplan la condición de que la arista entre la estación actual y el adyacente sea menor o igual a la autonomía. Una vez que la pila quede vacía, se revisa la tabla de visitas para ver que nodos no se han visitado que serán aquellos que no se puedan alcanzar. Retorna una excepción en caso de que los parámetros no sean válidos.

1.4 [OPCIONAL] Análisis de Costes Temporales

- ### 1.4.1 Obtención de los Costes

Para el análisis de costes temporales del camino óptimo del Grafo Estaciones he creado un programa en el paquete *aplicación* que se llama *AnalisisCostesTemporales*. En este programa lo que hago es guardar el tiempo en milisegundos desde que se inicia la función de camino optimo hasta que se termina para cada una de las 5 rutas. También guardamos el número de estaciones que tiene la lista devuelta para relacionar el número de estaciones con el coste temporal, la hipótesis sería que más estación mayor será el coste temporal porque el algoritmo tendrá que repetir más veces el bucle para llegar al destino.

Para obtener un resultado más fiable debido a que el coste temporal varía de múltiples factores es calcular para cada ruta 100 veces cuanto tarda y luego con eso hago una media con la que obtengo el coste temporal medio de cada ruta. Repetimos esto varias veces modificando la autonomía para ver como afecta.

- ### 1.4.2 Razonamiento y Análisis de los Resultados

Como se puede observar en la **figura 1.4.1** y **tabla 1.4.1**, se puede ver que ha medida que aumenta la autonomía de la ruta también aumenta el coste temporal, esto se puede deber a que al aumentar la autonomía aumenta el número de aristas que se tienen en cuenta y por tanto es necesario visitar más adyacentes. En el único que no ocurre esto es en la ruta 4, en la que al aumentar la autonomía se reduce el coste temporal, esto se puede deber a que en la ruta 4 al aumentar la autonomía se alcanza más rápido el nodo de destino posiblemente porque hay más nodos que se dirigen hacia al nodo de destino y por tanto el algoritmo alcanza más eficientemente el nodo de destino. Las rutas tienen costes diferentes porque cada ruta tendrá una mayor o menor densidad de nodos cercanos que estén conectados, cuanto mayor sea la densidad de la aquella parte más se verá afectado el coste temporal por la autonomía.

Relacionado con la autonomía, se puede observar como al aumentarla se reducen el número de estaciones por las que pasa la ruta (**figura y tabla 1.4.2)**. Ya que al aumentar la autonomía se visitan más nodos adyacentes de forma que se puede descartar más nodos que pueden ser menos favorables a alcanzar el camino mínimo.

Aunque el número de estaciones no es un factor que afecte al coste temporal dentro de una misma ruta. Si que afecta al coste temporal, pero de una forma distinta, y es que es el factor que hace que el coste temporal de la ruta sea mayor o menor. En la ruta 2 que es la que menos estaciones (y una menor distancia recorrida) tiene en la ruta con el menor coste temporal, en cambio la ruta 1 y 5 que son las que tiene más estaciones y por tanto un mayor coste temporal. En la **figura 1.4.3** se puede observar esta tendencia, que hay que decir que no es una tendencia lineal clara ya que el coste temporal depende de otros factores como son por ejemplo los nodos cercanos (como se ha mencionado antes).

- **1.4.3 Gráficas y Tabla de los Costes Computacionales**

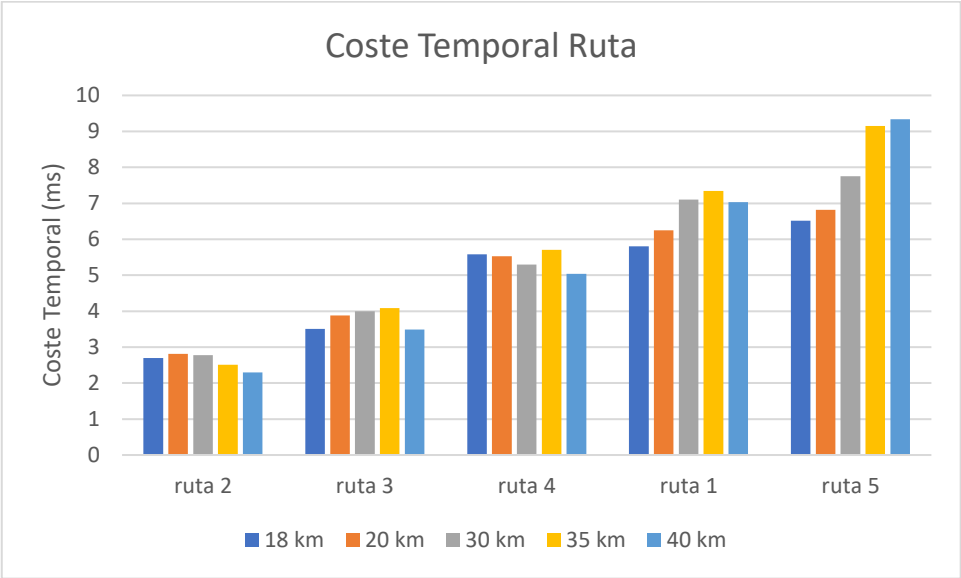


Figura 1.4.1 Coste Temporal de las rutas pedidas

	RUTA 2	RUTA 3	RUTA 4	RUTA 1	RUTA 5
18 KM	2.7	3.51	5.58	5.8	6.52
20 KM	2.81	3.88	5.53	6.25	6.82
30 KM	2.78	4	5.3	7.1	7.75
35 KM	2.51	4.09	5.71	7.34	9.15
40 KM	2.3	3.49	5.04	7.03	9.34

Tabla 1.4.1 Coste Temporal de las rutas pedidas

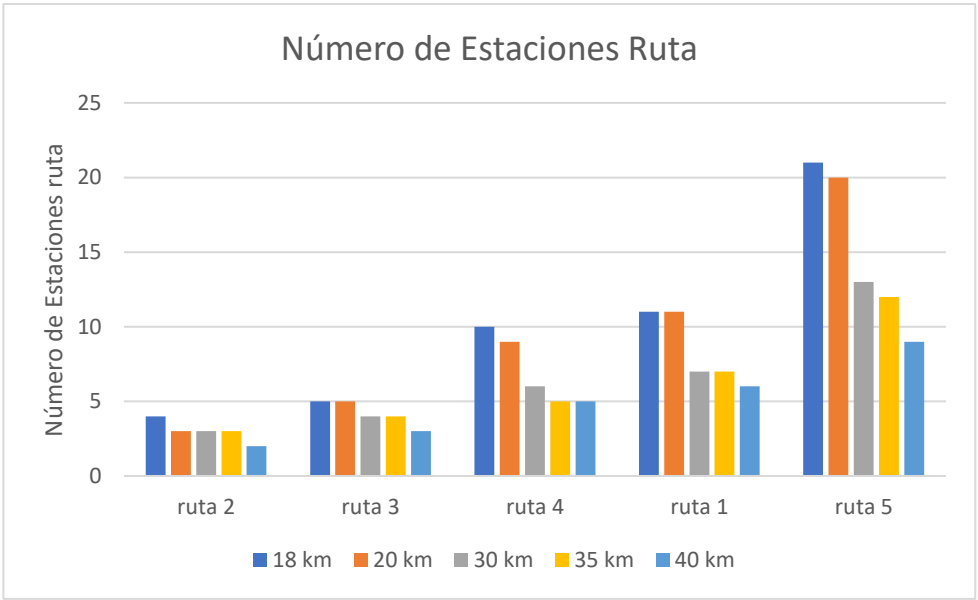


Figura 1.4.2 Número de Estaciones de las rutas pedidas

	RUTA 2	RUTA 3	RUTA 4	RUTA 1	RUTA 5
18 KM	4	5	10	11	21
20 KM	3	5	9	11	20
30 KM	3	4	6	7	13
35 KM	3	4	5	7	12
40 KM	2	3	5	6	9

Tabla 1.4.2 Número de estaciones de las rutas pedidas

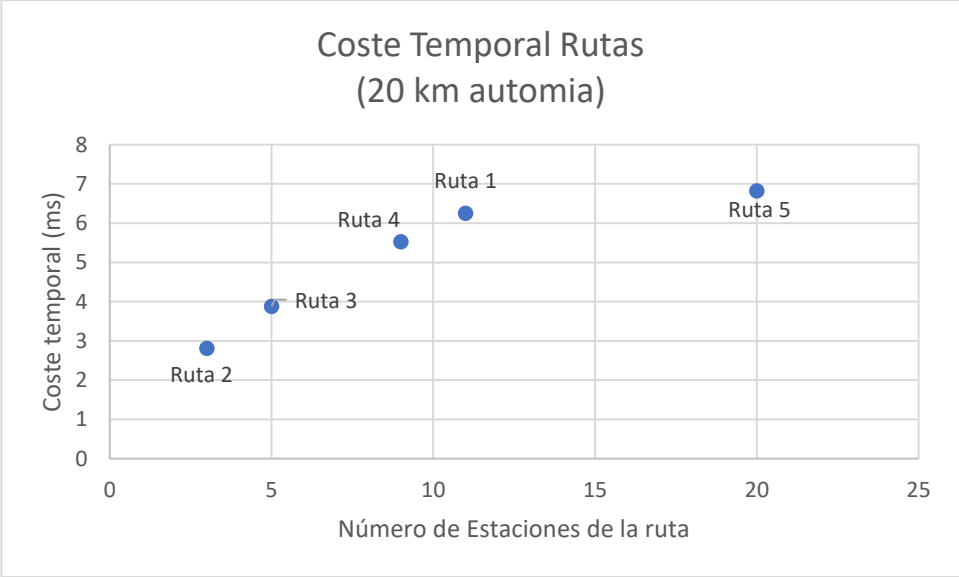


Figura 1.4.3 Coste Temporal de las rutas pedidas a 20 km de autonomía

2. Juego de Pruebas

2.1 Grafo Genérico

El juego de pruebas de la clase *GrafoGenerico,* para probarlo usamos como vértices Strings de números y como etiqueta de las aristas números enteros.

PRUEBAS	Objetivo	Input	Salida	Esperado
Probamos el Constructor	Probamos el constructor para ver si el grafo se inicia correctamente	El tamaño inicial del Grafo. Valor 6	Un grafo vacío con una tabla hash para los vértices de un tamaño de 6 posiciones	Correcto
Prueba Inserción Vertice	Añadir Vértices que son correctos	Vertice "1", "2", "3" y "4"	Un grafo con los vértices del input	Correcto
	Intentar añadir Vertice nulo	Vertice nulo	EXCEPCIÓN	Correcto
Prueba Agregar Arista	Añadimos arista de forma normal. El primer parámetro es el vertice menor y el segundo es el vertice mayor	Vertice "1" como primer parámetro. Vertice "2" como segundo parámetro	Un grafo en el que 1 y 2 están conectados. La arista se encuentra en el puntero horizontal del vertice menor "1"	Correcto
	Añadimos arista poniendo en el primer parámetro el vertice mayor y en el segundo el vertice menor	Vertice "3" como primer parámetro. Vertice "2" como segundo parámetro	Un grafo en el que 3 y 2 están conectados. La arista se encuentra en el puntero horizontal del vertice menor "2"	Correcto
	Intentamos añadir una arista ya presente en el grafo	Insertar arista entre "1" y "2"	EXCEPCIÓN	Correcto
Prueba Existe Arista	Comprobamos si existe una arista que se encuentra en el grafo	Primero probamos con buscar la arista 1<-->3 y luego su inversa 3<-->1	Valor True	Correcto
	Comprobamos si existe una arista al mismo vertice	Buscamos arista 1<-->1	Valor False	Correcto
Prueba Valor Arista	Buscamos valor arista de una que existe	Arista 1<-->2	El valor de la arista es 100	Correcto
	Intentamos buscar valor arista de una que no existe	Arista 2<-->4	EXCEPCIÓN	Correcto

Prueba Adyacentes	Buscamos adyacentes del vertice menor del grafo	Vertice 1	Lista [3, 2]	Correcto
	Buscamos adyacentes del vertice intermedio del grafo	Vertice 2	Lista [1, 3]	Correcto
	Buscamos adyacentes de un vertice no conectado del grafo	Vertice 4	Lista [ ]	Correcto
	Buscamos adyacentes de un vertice que no existe en el grafo	Vertice 5	EXCEPCIÓN	Correcto

2.2 Prueba Zonas de Recarga y Enchufes:

Probamos las funciones de las clases que almacenan los datos cargados del fichero JSON:

PRUEBAS	Objetivo	Input	Salida	Esperado
Prueba lectura fichero JSON	Leemos un fichero JSON de estaciones reducido	Fichero JSON reducido	Una lista con las zonas de recarga del fichero	Correcto
	Leemos el fichero JSON de estaciones completo	Fichero JSON completo	Una lista con las zonas de recarga del fichero	Correcto
Prueba de lectura JSON - No repetidos	Buscamos identificadores repetidos en la lista reducida de zonas cargada	Lista reducida Zonas Recarga	No hay repetidos	Correcto
	Buscamos identificadores repetidos en la lista completa de zonas cargada	Lista completa Zonas Recarga	No hay repetidos	Correcto
Prueba Enchufe Más Potencia	Buscamos el enchufe con más potencia de las Zonas de Recargas de la lista	Lista reducida Zonas Recarga	Una lista con el enchufe más potente de cada zona de recarga	Correcto
Prueba Distancia	Calculamos distancia de zona 1 (Tarragona) a si mismo	Lista reducida Zonas Recarga	0.0 km	Correcto
	Calculamos distancia zona 1 (Tarragona) a zona 7 (Cambrils Oeste)		17.943 km	Correcto
	Calculamos distancia zona 1 (Tarragona) a zona 8 (Cambrils Este)		18.030 km	Correcto
	Calculamos distancia zona 7 (Cambrils Oeste) a zona 8 (Cambrils Este)		0.127 km	Correcto

2.3 Prueba Grafo Estaciones:

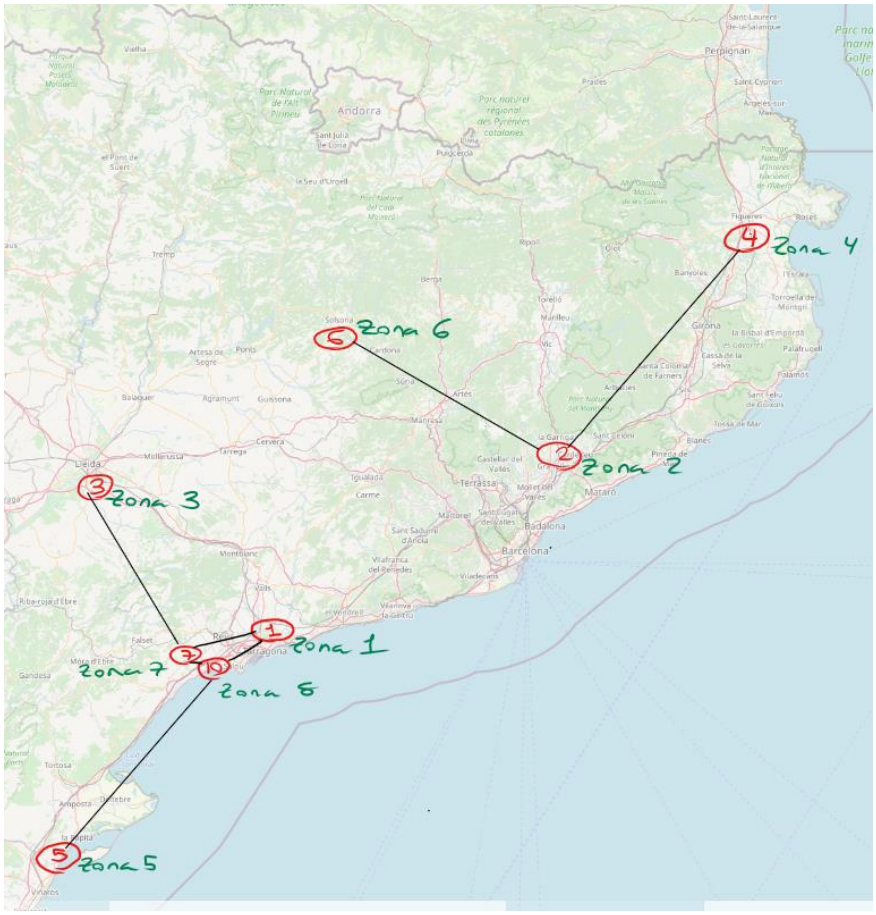
Probamos las funciones y algoritmos de la estructura del Grafo Estaciones:

La tabla Resultante es:

PRUEBAS	Objetivo	Input	Salida	Esperado
Probamos el Constructor	Probamos el constructor para ver si el grafo se inicia correctamente	Una lista de zonas de carga reducido	Obtenemos un grafo con partes no conexas. Los datos introducidos contienen zonas que están a más de 40km que no se conectan con la red principal del grafo debido a que se conectan a la zona más cercana que no pertenece a la red principal	Un grafo conexo.
		Una lista de zonas de carga completo		
Prueba Camino Optimo	Obtenemos una ruta entre zonas cercanas	Zona Área Cambrils rápida y zona Área Cambrils semirapida, están a 127 metros. Autonomia de 100km	Una lista con la zona de origen y de destino sin pasar por zonas intermediarias. Distancia 127 m	Correcto
	Obtenemos una ruta entre zonas lejanas y con autonomía baja y otra con autonomía alta	Zona Área Cambrils rápida y zona Gandesa. Autonomia de 20km y 50km	20km -> Una lista en la que se pasa por dos intermediarios. 51.780 km 50km -> Una lista en la que se pasa por un intermediario. 50.354 km	Correcto
	Obtenemos una ruta entre zonas muy lejanas y con autonomía media y otra alta	De Alcanar hasta Figueres autonomía de 150km y de 30km	150km -> 10 intermediarios 282.723 km 30km -> 14 intermediarios 286,911 km	Correcto
	Intentamos usar parámetros no validos	Ponemos autonomía negativa, destino que no se encuentra en el grafo y un origen que es un valor nulo	EXCEPCIÓN	Correcto
	Intentamos usar como zona de destino una igual a la de origen	De Alcanar hasta Alcanar	Una lista en la que solo aparece la estación Alcanar. Recorrido 0.0km	Correcto
	Intentamos usar una ruta lejana, pero con una autonomía muy baja	De Alcanar a Figueres con autonomía de 10km	EXCEPCIÓN: no se ha podido alcanzar el destino	Correcto

Prueba Distancia Zona Máxima no Garantizada	Partiendo de una misma zona de origen 9165 (Cambrils), probamos distintas autonomías	Autonomía de 10000 km	Una lista de 3 zonas que no se han podido alcanzar. Son zonas que no están conectadas a la red principal	Correcto
		Autonomía de 40 km	Una lista de 4 zonas que no se pueden alcanzar	Correcto
		Autonomía de 20 km	Una lista de 25 zonas que no se han podido alcanzar	Correcto
	Tomamos una zona de la red principal con autonomía media	zona 9904800 (Móra d'Ebre). Autonomía de 25 km	Una lista de 10 zonas que no se han podido alcanzar	Correcto
	Usamos como origen la zona una zona no conectada a la red	33852430 (Almedinilla, Córdoba). Autonomia de 5000 km	Una larga lista con muchas zonas de recarga no conectadas	Correcto, el grafo debería ser conexo
	Intentamos pasar parámetros no validos	Ponemos autonomía de 0, destino que no se encuentra en el grafo y un origen que es un valor nulo	EXCEPCIÓN	Correcto

El grafo resultante obtenido del fichero JSON reducido es el siguiente.



Como se puede observar tanto el grafo reducido como el grafo completo tiene zonas que no están conectadas a la red principal. Esto se debe a que los datos proporcionados no cumplen la condición de que conectándose a la zona más cercana cuando no haya ninguna a 40km, esto no garantiza la obtención de un grafo conexo como se afirma en el enunciado de la práctica.

3. Algoritmo de la práctica

TADGrafoGenerico

```
package TADGrafoGenerico;

import excepciones.NoExiste;
import excepciones.YaExisteArista;

import java.util.LinkedList;

public interface TADGrafoGenerico <K, V, A> {

   /**
    * Función agregar Arista
    * @param vertice1 - un vértice
    * @param vertice2 - otro vértice
    * @param arista - union entre ambos vertices
    * @throws NoExiste - cuando no existe uno de los vertices
    */
   void agregarArista(K vertice1, K vertice2, A arista) throws NoExiste,
YaExisteArista;

   /**
    * Función que comprueba si existe la arista
    * @return true - existe la arista. false - no existe
    */
   boolean existeArista(K vertice1, K vertice2);

   /**
    * Función para saber el valor de la arista de dos vertices pasados por
parámetro
    * @return el valor de la arista
    * @throws NoExiste - cuando la arista no existe en el grafo
    */
   A valorArista(K vertice1, K vertice2) throws NoExiste;

   /**
    * Función que busca los vertices conectados directamente al vértice pasado
por referencia
    * @return una lista con todos los vertices adyacentes
    * @throws NoExiste -cuando no se ha podido crear la lista
    */
   LinkedList<V> adyacentes(K vertice) throws NoExiste;


}
```

Vertice

```
package TADGrafoGenerico;


import TAD_TablaHash_ListaGenerica.Nodo;

public class Vertice<K extends Comparable<K>, V, A> implements
Comparable<Vertice<K, V, A>> {

   private K clave;
   private V dato;

   private Arista<K, V, A> punteroAristaHorizontal;
   private Arista<K, V, A> punteroAristaVertical;


   // ************ ↓↓↓ MÉTODOS ↓↓↓ ***********

   public Vertice(K clave, V dato){
      this.clave = clave;
      this.dato = dato;
      punteroAristaHorizontal = null;
      punteroAristaVertical = null;
   }

   /**
    * Metodo que comprueba si hay un siguiente nodo arista
    * @return true si hay nodo, false si es null
    */
   public boolean hasNextHorizontal() {
      return punteroAristaHorizontal != null;
   }

   /**
    * Metodo next
    * @return el puntero al siguiente nodo Arista Horizontal
    */
   public Arista<K, V, A> getPunteroAristaHorizontal() {
      return punteroAristaHorizontal;
   }
   /**
    * Metodo next
    * @return el puntero al siguiente nodo Arista Vertical
    */
   public Arista<K, V, A> getPunteroAristaVertical() {
      return punteroAristaVertical;
   }

   /**
    * Setter
    * @param aristaHorizontal - nuevo puntero al primer nodo Arista Horizontal
    */
   public void setNodoHorizontal(Arista<K, V, A> aristaHorizontal){
      this.punteroAristaHorizontal = aristaHorizontal;
   }

   /**
    * Setter
    * @param aristaVertical - nuevo puntero al primer nodo Arista Vertical
    */
   public void setNodoVertical(Arista<K, V, A> aristaVertical){
```

```
        this.punteroAristaVertical = aristaVertical;
    }

    /**
     * Metodo para obtener la instancia a los datos
     * @return la instancia de los datos del nodo
     */
    public V getDatos() {
        return dato;
    }

    @Override
    public int compareTo(Vertice<K, V, A> vertix) {
        return clave.compareTo(vertix.clave);
    }

    //colores para la impresión por consola
    public static final String ANSI_GREEN = "\u001B[32m";
    public static final String ANSI_BLUE = "\u001B[34m";
    public static final String ANSI_PURPLE = "\u001B[35m";
    public static final String ANSI_RESET = "\u001B[0m";

    @Override
    public String toString() {
        return "Vertice{" +
                ANSI_PURPLE + "dato=" + dato +
                ANSI_BLUE + ", punteroAristaHorizontal=" + punteroAristaHorizontal +
                ANSI_GREEN + ", punteroAristaVertical=" + punteroAristaVertical +
ANSI_RESET +
                '}';
    }
}
```

Arista

```
package TADGrafoGenerico;

public class Arista<K extends Comparable<K>, V, A> {

   private final A dato;

   // Punteros a las siguientes aristas del Vertice
   private Arista<K, V, A> punteroAristaHorizontal;
   private Arista<K, V, A> punteroAristaVertical;

   // Referencias a los vertices que unen
   private Vertice<K, V, A> referVerticeHorizontal; // Es el vertice menor
   private Vertice<K, V, A> referVerticeVertical; // Es el vertice mayor


   // ************ ↓↓↓ MÉTODOS ↓↓↓ ***********

   public Arista(A dato){
      this.dato = dato;
      punteroAristaHorizontal = null;
      punteroAristaVertical = null;
      referVerticeHorizontal = null;
      referVerticeVertical = null;
   }

   /**
    * Metodo que comprueba si hay un siguiente nodo arista
    * @return true si hay nodo, false si es null
    */
   public boolean hasNextHorizontal() {
      return punteroAristaHorizontal != null;
   }


   /**
    * Getter
    * @return el puntero al siguiente nodo Arista Horizontal, la nueva arista
tiene el mismo vertice que el menor de esta arista
    */
   public Arista<K, V, A> getPunteroAristaHorizontal() {
      return punteroAristaHorizontal;
   }
   /**
    * Getter
    * @return el puntero al siguiente nodo Arista Vertical, la nueva arista tiene
el mismo vertice que el vertice mayor de esta arista
    */
   public Arista<K, V, A> getPunteroAristaVertical() {
      return punteroAristaVertical;
   }

   /**
    * Setter
    * @param aristaHorizontal - nuevo puntero al siguiente nodo Arista Horizontal
(ambas están unidas al vertice menor de esta arista)
    */
   public void setNodoHorizontal(Arista<K, V, A> aristaHorizontal){
      this.punteroAristaHorizontal = aristaHorizontal;
   }
```

```
    /**
     * Setter
     * @param aristaVertical - nuevo puntero al siguiente nodo Arista Vertical
(ambas están unidas al vertice mayor de esta arista)
     */
    public void setNodoVertical(Arista<K, V, A> aristaVertical){
        this.punteroAristaVertical = aristaVertical;
    }

    /**
     * Setter
     * @param referVerticeHorizontal - nueva referencia de la arista a un vertice
menor al que está unido
     */
    public void setReferVerticeHorizontal(Vertice<K, V, A> referVerticeHorizontal)
{
        this.referVerticeHorizontal = referVerticeHorizontal;
    }

    /**
     * Setter
     * @param referVerticeVertical - nueva referencia de la arista al vertice
mayor al que está unido
     */
    public void setReferVerticeVertical(Vertice<K, V, A> referVerticeVertical) {
        this.referVerticeVertical = referVerticeVertical;
    }

    /**
     * Metodo para obtener la instancia a los datos
     * @return la instancia de los datos del nodo
     */
    public A getDatos() {
        return dato;
    }

    /**
     * Getter
     * @return los datos del vertice menor que une la arista
     */
    public V getValorReferMenor(){return referVerticeHorizontal.getDatos();}

    /**
     * Getter
     * @return los datos del vertice mayor que une la arista
     */
    public V getValorReferMayor(){return referVerticeVertical.getDatos();}

    /**
     * Getter
     * @return la referencia a uno de los vertices
     */
    public Vertice<K, V, A> getReferVerticeHorizontal() {
        return referVerticeHorizontal;
    }

    /**
     * Getter
     * @return la referencia a otro de los vertices
     */
    public Vertice<K, V, A> getReferVerticeVertical() {
```

```
        return referVerticeVertical;
    }

    @Override
    public String toString() {
        // Imprimimos el objeto y cuál es la arista siguiente
        V referHorizMenor = punteroAristaHorizontal == null? null:
punteroAristaHorizontal.getValorReferMenor();
        V referHorizMayor = punteroAristaHorizontal == null? null:
punteroAristaHorizontal.getValorReferMayor();
        V referVertMenor = punteroAristaVertical == null? null:
punteroAristaVertical.getValorReferMenor();
        V referVertMayor = punteroAristaVertical == null? null:
punteroAristaVertical.getValorReferMayor();

        return "Arista{" +
                "dato=" + dato +
                ",\n\t VerticeHorizontal=" + referVerticeHorizontal.getDatos() +
                ",\n\t VerticeVertical=" + referVerticeVertical.getDatos() +
                ",\n\t AristaHorizontal=(" + referHorizMenor + "<-->" +
referHorizMayor +
                "),\n\t AristaVertical=(" + referVertMenor + "<-->" + referVertMayor
+
                ")\n\t}";
    }
}
```

GrafoGenerico

```
package TADGrafoGenerico;

import TAD_TablaHash_ListaGenerica.ClaveException;
import TAD_TablaHash_ListaGenerica.ElementoNoEncontrado;
import TAD_TablaHash_ListaGenerica.ListaGenerica;
import TAD_TablaHash_ListaGenerica.TablaHashGenerica;
import datos.ZonaRecarga;
import excepciones.NoExiste;
import excepciones.YaExisteArista;

import java.util.LinkedList;

/**
 * Clase Grafo Generico, implementa TADGrafoGenerico, Proporciona una
implementación para crear un grafo etiquetado y no dirigido.
 * @param <K> Tipo de Objeto que será la clave o identificador del vertice. Debe
implementar Comparable
 * @param <V> Tipo de Objeto que será el valor del vertice
 * @param <A> Tipo de Objeto que será el valor o etiqueta de la arista
 */
public class GrafoGenerico<K extends Comparable<K>, V, A> implements
TADGrafoGenerico<K, V, A> {

   TablaHashGenerica<K, Vertice<K, V, A>> tablaVertices;

   /**
    * Constructor de la clase GrafoGenerico
    * @param mida - longitud de la tabla hash de los vertices (posteriormente se
redimensiona automáticamente)
    */
   public GrafoGenerico(int mida){
      tablaVertices = new TablaHashGenerica<>(mida);
   }

   /**
    * Función que agrega un vértice
    * @param vertice - dato que se quiere añadir al grafo
    * @throws NoExiste - El vértice pasado por parámetro es nulo
    */
   public void agregarVertice(K clave, V vertice) throws NoExiste{
      if (vertice == null || clave == null){ // Se lanza excepcion si los
parámetros son nulos
         throw new NoExiste("El vértice pasado por parámetro es nulo");
      }
      tablaVertices.insertar(clave, new Vertice<K, V, A>(clave, vertice));
   }



   public void agregarArista(K vertice1, K vertice2, A arista) throws NoExiste,
YaExisteArista {
      try {
         // Comprobamos los vertices
         Vertice<K, V,A> nodoMenor = tablaVertices.obtener(vertice1);
         Vertice<K, V,A> nodoMayor = tablaVertices.obtener(vertice2);

         // Añadimos la arista siempre al vértice menor
         if (nodoMenor.compareTo(nodoMayor)>0){ // nodoMenor es mayor que el
nodoMayor, entonces los intercambiamos
            nodoMenor = tablaVertices.obtener(vertice2);
```

```
            nodoMayor = tablaVertices.obtener(vertice1);
         }

         //Comprobamos si ya existe la arista
         if(existeArista(vertice1, vertice2)){
            throw new YaExisteArista("Ya existe una arista entre ambos
vertices");
         }

         // Creamos la arista y la unimos
         Arista<K, V, A> nodoArista = new Arista<K, V, A>(arista);
         nodoArista.setReferVerticeHorizontal(nodoMenor); // Lo unimos al vertice
menor
         nodoArista.setReferVerticeVertical(nodoMayor); // Lo unimos al vertice
mayor

         // Apuntamos a las siguientes aristas
         nodoArista.setNodoHorizontal(nodoMenor.getPunteroAristaHorizontal());
         nodoArista.setNodoVertical(nodoMayor.getPunteroAristaVertical());

         // Modificamos los punteros de los nodos vertice
         nodoMenor.setNodoHorizontal(nodoArista);
         nodoMayor.setNodoVertical(nodoArista);

      } catch (ClaveException e) {
         throw new NoExiste("No existe alguno de los vertices al añadir arista");
      }
   }


   public boolean existeArista(K vertice1, K vertice2) {
      boolean existe = true;
      try{ // Buscamos la arista en la multilista y si no la encuentra es que no
existe
         buscarArista(vertice1, vertice2);
      } catch (NoExiste e) {
         existe = false;
      }
      return existe;
   }


   public A valorArista(K vertice1, K vertice2) throws NoExiste {
      return buscarArista(vertice1, vertice2).getDatos(); // Buscamos la arista
en la multilista y devuelve el valor
   }

   private Arista<K, V, A> buscarArista(K vertice1, K vertice2) throws NoExiste{
      boolean existe = false;
      Arista<K, V, A> aristaHorizontal = null;

      try { // Obtenemos los vertices
         Vertice<K, V, A> nodoMenor = tablaVertices.obtener(vertice1);
         Vertice<K, V, A> nodoMayor = tablaVertices.obtener(vertice2);

         // Identificamos el nodoMenor y nodoMayor
         if (nodoMenor.compareTo(nodoMayor)>0){ // nodoMenor es mayor que el
nodoMayor, entonces los intercambiamos
            nodoMenor = tablaVertices.obtener(vertice2);
            nodoMayor = tablaVertices.obtener(vertice1);
         }
```

```
            // El nodo menor será el que tenga la arista en su puntero Horizontal
            if (nodoMenor.hasNextHorizontal()) {
               aristaHorizontal = nodoMenor.getPunteroAristaHorizontal();
               existe =
aristaHorizontal.getReferVerticeVertical().compareTo(nodoMayor) == 0;

               // Se busca en la lista de aristas horizontales el nodo, si existe se
vuelve true se retorna se ha encontrado si no se lanza excepcion
               while (!existe && aristaHorizontal.hasNextHorizontal()){
                  aristaHorizontal = aristaHorizontal.getPunteroAristaHorizontal();
// Guardamos la siguiente arista
                  existe =
aristaHorizontal.getReferVerticeVertical().compareTo(nodoMayor) == 0; //Miramos
si es la arista que buscamos
               }
            }

            if (!existe){ // En caso de que la arista exista devolvemos como es
               throw new NoExiste("No existe la arista entre los vertices
"+vertice1+" <---> "+vertice2);
            }
         } catch (ClaveException e) {/* No existe alguno de los vertices*/
            throw new NoExiste("No existe alguno de los vertices");
         }
         return aristaHorizontal;
      }



      public LinkedList<V> adyacentes(K vertice) throws NoExiste {
         LinkedList<V> listaVertices = new LinkedList<>();

         try {// Obtenemos el vertice
            Vertice<K, V, A> nodoInicio = tablaVertices.obtener(vertice);

            // Guardamos las primeras aristas del vertice, de las cuales partiremos
para hacer el recorrido de ambas listas
            Arista<K, V, A> aristaHorizontal =
nodoInicio.getPunteroAristaHorizontal();
            Arista<K, V, A> aristaVertical = nodoInicio.getPunteroAristaVertical();

            // Cuando el vértice es mayor que el otro vértice al que está unido
            while (aristaVertical != null){

listaVertices.add(aristaVertical.getReferVerticeHorizontal().getDatos());
               aristaVertical = aristaVertical.getPunteroAristaVertical();
            }

            // Cuando el vértice es menor que el otro vértice al que está unido
            while (aristaHorizontal != null){

listaVertices.add(aristaHorizontal.getReferVerticeVertical().getDatos());
               aristaHorizontal = aristaHorizontal.getPunteroAristaHorizontal();
            }

         } catch (ClaveException  e) {// No existe alguno de los vertices
            throw new NoExiste("No existe el vertice " + vertice);
         }

         return listaVertices;
```

```
    }

    /**
     * Obtener tamaño tabla hash de vertices
     * @return la longitud de la tabla hash de vertices del grafo
     */
    public int getMidaTablaVertices(){
        return tablaVertices.midaTabla();
    }

    /**
     * Obtener Lista de Claves Vertices
     * @return una lista con los identificadores de los vertices del grafo
     */
    public ListaGenerica<K> getClavesVertices(){
        return tablaVertices.obtenerClaves();
    }

    /**
     * Valor del vertice
     * @param idVertice - identificador/clave del vertice
     * @return valor del vertice que tiene ese id
     * @throws ClaveException - en caso de que no exista el vertice en el grafo
     */
    public V valorVertice(K idVertice) throws ClaveException, NullPointerException
{
        return tablaVertices.obtener(idVertice).getDatos();
    }

    @Override
    public String toString() {
        return "GrafoGenerico{" +
                "tablaVertices=\n" + tablaVertices +
                '}';
    }
}
```

NoExiste

```
package excepciones;

import java.io.Serial;

public class NoExiste extends Exception {

   @Serial
   private static final long serialVersionUID = 1L;

   public NoExiste (String fraseError){
      super(fraseError);
   }
}
```

YaExisteArista

```
package excepciones;

import java.io.Serial;

public class YaExisteArista extends Exception {

   @Serial
   private static final long serialVersionUID = 1L;

   public YaExisteArista(String fraseError){
      super(fraseError);
   }
}
```

PruebaGrafoGenerico

```
package aplicacion;

import TADGrafoGenerico.GrafoGenerico;
import excepciones.NoExiste;
import excepciones.YaExisteArista;

public class PruebaGrafoGenerico {
   public static void main(String[] args) {

      // Probamos constructor
      System.out.println("Probamos constructor");
      GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
      System.out.println(grafo);

      // Probamos Inserción Vértice
      pruebaInsercionVertice();

      // Probamos función Añadir Arista
      pruebaAgregarArista();

      // Probamos función Existe Arista
      pruebaExisteArista();

      // Probamos función Valor Arista
      pruebaValorArista();

      // Probamos función Adyacentes
      pruebaAdyacentes();
   }

   private static void pruebaAdyacentes() {
      System.out.println("\n----------PRUEBA ADYACENTES--------------");
      GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
      agregarVertices(grafo);
      agregarAristas(grafo);

      System.out.println("\nEstado Inicial Grafo:");
      System.out.println(grafo);

      try {
         System.out.println("\nVertices adyacentes a 1");
         System.out.println("\tLongitud Lista: " + grafo.adyacentes("1").size());
         System.out.println(grafo.adyacentes("1"));

         System.out.println("\nVertices adyacentes a 2");
         System.out.println("\tLongitud Lista: " + grafo.adyacentes("2").size());
         System.out.println(grafo.adyacentes("2"));

         System.out.println("\nVertices adyacentes a 3");
         System.out.println("\tLongitud Lista: " + grafo.adyacentes("3").size());
         System.out.println(grafo.adyacentes("3"));

         System.out.println("\nVertices adyacentes a 4");
         System.out.println("\tLongitud Lista: " + grafo.adyacentes("4").size());
         System.out.println(grafo.adyacentes("4"));

         System.out.println("\nIntentamos listar vertices adyacentes a 5");
         System.out.println("\tLongitud Lista: " + grafo.adyacentes("5").size());
```

```
            System.out.println(grafo.adyacentes("5"));
        } catch (NoExiste e) {
            System.out.println(e.getMessage());
        }
    }

    private static void pruebaValorArista() {
        System.out.println("\n----------PRUEBA VALOR ARISTA--------------");
        GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
        agregarVertices(grafo);
        agregarAristas(grafo);

        System.out.println("\nEstado Inicial Grafo:");
        System.out.println(grafo);

        try {
            System.out.println("\nArista 1 <--> 2");
            System.out.println(grafo.valorArista("1", "2"));

            System.out.println("\nArista 2 <--> 3");
            System.out.println(grafo.valorArista("3", "2"));

            System.out.println("\nArista 2 <--> 4");
            System.out.println(grafo.valorArista("2", "4"));

        } catch (NoExiste e) {
            System.out.println(e.getMessage());
        }
    }


    private static void agregarVertices(GrafoGenerico<String, String, Integer>
grafo) {
        try {
            grafo.agregarVertice("1", "1");
            grafo.agregarVertice("2", "2");
            grafo.agregarVertice("3", "3");
            grafo.agregarVertice("4", "4");
        } catch (NoExiste e) {
            e.printStackTrace(); //ERROR
        }
    }

    private static void agregarAristas(GrafoGenerico<String, String, Integer>
grafo) {
        try {
            grafo.agregarArista("1", "2", 100);
            grafo.agregarArista("3", "2", 75);
            grafo.agregarArista("1", "3", 90);
        } catch (NoExiste | YaExisteArista e) {
            e.printStackTrace();
        }
    }

    private static void pruebaAgregarArista() {
        System.out.println("\n----------PRUEBA AÑADIR ARISTA--------------");
        GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
        agregarVertices(grafo);
        try {
```

```
            System.out.println("\nAñadimos arista entre vertice 1 y 2");
            grafo.agregarArista("1", "2", 100);
            System.out.println(grafo);

            System.out.println("\nIntentamos añadir vertice entre 3 y 2");
            grafo.agregarArista("3", "2", 75); // Añadimos arista, vertice 3 -> es
el mayor, vertice 2 -> el menor
            System.out.println(grafo);

            System.out.println("\nAñadimos arista a los vertices 1 y 3");
            grafo.agregarArista("1", "3", 90); // Añadimos arista, vertice 1 -> es
el mayor, vertice 3 -> el menor
            System.out.println(grafo);

            System.out.println("\nIntentamos añadir arista entre vertice 1 y 2");
            grafo.agregarArista("2", "1", 50); // Intentamos sobrescribir
            System.out.println("ERROR" + grafo);
         } catch (NoExiste | YaExisteArista e) {
            System.out.println(e.getMessage());
         }
      }

      private static void pruebaExisteArista() {
         System.out.println("\n----------PRUEBA EXISTE ARISTA--------------");
         GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
         agregarVertices(grafo);
         agregarAristas(grafo);

         System.out.println("\nEstado Inicial Grafo:");
         System.out.println(grafo);

         System.out.println("\nComprobamos si existe arista entre 1 <--> 3");
         System.out.println("1-->3 "+grafo.existeArista("1", "3"));
         System.out.println("3-->1 "+grafo.existeArista("3", "1"));

         System.out.println("\nComprobamos si existe arista entre 1 <--> 2");
         System.out.println("1-->2 "+grafo.existeArista("1", "2"));
         System.out.println("2-->1 "+grafo.existeArista("2", "1"));

         System.out.println("\nComprobamos si existe arista entre 1 <--> 1");
         System.out.println(grafo.existeArista("1", "1"));

         System.out.println("\nComprobamos si existe arista entre 1 <--> 4");
         System.out.println("1-->4 "+grafo.existeArista("1", "4"));
         System.out.println("4-->1 "+grafo.existeArista("4", "1"));
      }

      private static void pruebaInsercionVertice() {
         GrafoGenerico<String, String, Integer> grafo = new GrafoGenerico<String,
String, Integer>(6);
         System.out.println("\n----------PRUEBA AÑADIR VERTICE--------------");
         System.out.println("Añadimos vertice 1, 2, 3 y 4");
         agregarVertices(grafo);
         System.out.println(grafo);
         System.out.println("Intentamos añadir un vertice nulo");
         try{
            grafo.agregarVertice(null, null);
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }
```

```
        }

}
```

Enchufe

```
package datos;

/**
 * Clase que almacena los datos de cada enchufe de recarga
 */
public class Enchufe {
   private final int id, id_estacio;
   private final String nom;
   private final String data;
   private final String consum,carrer,ciutat,estat;
   private final String temps, potencia;
   private final String tipus;
   private final double latitud,longitud;

   Enchufe( int id, int id_estacio, String nom, String data, String consum,
String carrer, String ciutat, String estat, String temps,
          String potencia, String tipus, double latitud, double longitud){
      this.id = id;
      this.id_estacio = id_estacio;
      this.nom = nom;
      this.data = data;
      this.consum = consum;
      this.carrer = carrer;
      this.ciutat = ciutat;
      this.estat = estat;
      this.temps = temps;
      this.potencia = potencia;
      this.tipus = tipus;
      this.latitud = latitud;
      this.longitud = longitud;

   }

   /**
    * Getter
    * @return la latitud (coordenadas) del enchufe
    */
   public double getLatitud() {
      return latitud;
   }

   /**
    * Getter
    * @return la longitud (coordenadas) del enchufe
    */
   public double getLongitud() {
      return longitud;
   }

   /**
    * Getter
    * @return el identificador del enchufe
    */
   public int getId() {
      return id;
   }

   /**
    * Getter de la potencia del enchufe
    * @return la potencia del enchufe
```

```
     * @throws NumberFormatException - cuando no se tiene informacion sobre la
potencia del enchufe
     */
    public double getPotencia(){
        return Double.parseDouble(potencia);
    }

    /**
     * Getter
     * @return el nombre del enchufe
     */
    public String getNom(){
        return nom;
    }

    @Override
    public String toString() {
        return "\n\tEnchufe{" +
                "id='" + id + '\'' +
                ", id_estacio='" + id_estacio + '\'' +
                ", nom='" + nom + '\'' +
                ", data='" + data + '\'' +
                ", consum='" + consum + '\'' +
                ", carrer='" + carrer + '\'' +
                ", ciutat='" + ciutat + '\'' +
                ", estat='" + estat + '\'' +
                ", temps='" + temps + '\'' +
                ", potencia='" + potencia + '\'' +
                ", tipus='" + tipus + '\'' +
                ", latitud='" + latitud + '\'' +
                ", longitud='" + longitud + '\'' +
                '}';
    }
}
```

ZonaRecarga

```
package datos;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.LinkedList;
import java.util.Scanner;

import com.google.gson.Gson;

/**
 * Clase que es una zona de recarga que contiene todos los enchufes que tienen
las mismas coordenadas
 */
public class ZonaRecarga implements Comparable<ZonaRecarga> {
   private final int id;
   private final LinkedList<Enchufe> listaEnchufes;
   private final double latitud, longitud;

   public ZonaRecarga (Enchufe enchufe){
      this.id = enchufe.getId();
      listaEnchufes = new LinkedList<>();
      listaEnchufes.add(enchufe);
      this.latitud = enchufe.getLatitud();
      this.longitud = enchufe.getLongitud();
   }

   /**
    * Función que añade un enchufe a la zona de recarga, tiene que tener las
mismas coordenadas
    * @param enchufe - enchufe a añadir
    */
   public void addEnchufe (Enchufe enchufe)  {
      if (equalsCoordenadas(enchufe.getLatitud(), enchufe.getLongitud())){
         listaEnchufes.add(enchufe);
      }
   }

   /**
    * Función que calcula la distancia euclidiana entre las coordenadas de dos
zonas de recarga
    * @param zonaRecarga - zona de recarga sobre la que se quiere calcular la
distancia respecto a la zona de recarga actual
    * @return la distancia (double) entre dos zonas de recarga
    * @throws NullPointerException - excepcion si zona de recarga es nulo
    */
   public double distancia(ZonaRecarga zonaRecarga) {
      if (zonaRecarga == null){
         throw new NullPointerException();
      }

      final double  R = 6378.137; // Constante radio ecuatorial de la tierra

      // Calculamos la latitud y longitud
      double latitudA = zonaRecarga.latitud * Math.PI/180;
      double longitudA = zonaRecarga.longitud * Math.PI/180;
      double latitudB = this.latitud * Math.PI/180;
      double longitudB = this.longitud * Math.PI/180;

      // Calculamos la variación de la latitud y longitud
      double variacionLatitud = latitudB-latitudA;
```

```
        double variacionLongitud = longitudB-longitudA;

        variacionLatitud = Math.sin(variacionLatitud/2);
        variacionLongitud = Math.sin(variacionLongitud/2);

        double resultado = variacionLatitud * variacionLatitud +
Math.cos(latitudA)*Math.cos(latitudB) * variacionLongitud * variacionLongitud;

        return 2 * R * Math.atan2(Math.sqrt(resultado), Math.sqrt(1-resultado));
    }


    /**
     * Función que lee un archivo JSON con los enchufes de recarga
     * @param path - ruta del archivo JSON
     * @return una lista con las zonas de recarga (conjunto de enchufes con las
mismas coordenadas
     * @throws FileNotFoundException - problemas al no poder encontrar el fichero
     */
    public static LinkedList<ZonaRecarga> leerJson(String path) throws
FileNotFoundException {
        LinkedList<ZonaRecarga> listaZonas = new LinkedList<>();
        ZonaRecarga zona;
        Enchufe enchufe;
        Scanner reader = new Scanner(new File(path));
        int index;
        boolean zonaEncontrada;

        // cargamos el texto
        reader.useDelimiter("]");
        String s = reader.next();
        s += "]";


        // configuramos para leer los enchufes
        String objeto;
        s = s.replace("[", "");
        s = s.replace("]", "");
        Scanner leer = new Scanner(s);
        leer.useDelimiter("},");

        while (leer.hasNext()){
            // Ajustamos el string
            objeto = leer.next();
            objeto = objeto.replace("}", "");
            objeto += "}";

            // Pasamos de JSONObject a un Objeto Java
            enchufe = new Gson().fromJson(objeto, Enchufe.class);

            index = 0;
            zonaEncontrada = false;
            while (!zonaEncontrada && index < listaZonas.size()){
                zona = listaZonas.get(index);
                if (zona.equalsCoordenadas(enchufe.getLatitud(),
enchufe.getLongitud())){ //Añadimos enchufe a zona existente
                    zonaEncontrada = true;
                    zona.addEnchufe(enchufe);
                }
                index++;
            }
```

```
            if(!zonaEncontrada){ //Añadimos nueva zona zona
                zona = new ZonaRecarga(enchufe);
                listaZonas.add(zona);
            }
        }

        leer.close();
        reader.close();
        return listaZonas;
    }

    public int getId() {
        return id;
    }

    public double getLatitud() {
        return latitud;
    }

    public double getLongitud() {
        return longitud;
    }

    /**
     * Compara si la zona tiene esas coordenadas
     * @param latitud de la zona a comparar
     * @param longitud de la zona a comparar
     * @return true si son las mismas coordenadas o false si son diferentes
     */
    public boolean equalsCoordenadas(double latitud, double longitud){
        return latitud == this.latitud && longitud == this.longitud;
    }

    public Enchufe getEnchufeMasPotencia(){
        Enchufe enchufeMasPotencia = listaEnchufes.get(0);
        for (Enchufe enchufe:listaEnchufes) {
            try {
                if (enchufe.getPotencia() > enchufeMasPotencia.getPotencia()){
                    enchufeMasPotencia = enchufe;
                }
            } catch (NumberFormatException e){
                // No hacer nada
            }
        }
        return enchufeMasPotencia;
    }

    @Override
    public int compareTo(ZonaRecarga zona) {
        return this.id - zona.id; // negativo si este objeto es menor, 0 si son
iguales, positivo si es mayor a la zona pasasado por parametro
    }

    @Override
    public String toString() {
        return "\nZonaRecarga{" +
                "id=" + id +
                ", latitud=" + latitud +
                ", longitud=" + longitud +
                ", listaEnchufes=" + listaEnchufes +
                '}';
```

```
        }
    }
```

PruebaZonaRecarga

```
package aplicacion;

import datos.ZonaRecarga;

import java.io.FileNotFoundException;
import java.util.LinkedList;

public class PruebaZonaRecarga {
   public static void main(String[] args) throws FileNotFoundException{
      // Probamos Lectura Fichero JSON
      LinkedList<ZonaRecarga> listaPruebas
=ZonaRecarga.leerJson("src/main/resources/pruebas.json");
      LinkedList<ZonaRecarga> lista  =
ZonaRecarga.leerJson("src/main/resources/icaen.json");

      pruebaJSON(lista, listaPruebas);

      pruebaPotencia(listaPruebas);

      pruebaDistancia(listaPruebas);
   }

   private static void pruebaDistancia(LinkedList<ZonaRecarga> listaPruebas) {
      System.out.println("\n----------PRUEBA DISTANCIA COORDENADAS DE LAS ZONAS--
----------");
      String frase = "";
      frase += "\nDistancia zona 1 a si mismo --> " +
listaPruebas.get(0).distancia(listaPruebas.get(0));
      frase += "\nDistancia zona 7 (Cambrils Oeste) a 8 (Cambrils Este) --> " +
listaPruebas.get(6).distancia(listaPruebas.get(7));

      frase += "\n\nMiramos distancia a las zonas a las que se unira en el grafo
la zona 1 (Tarragona)";
      frase += "\nDistancia zona 1 a 7 (Cambrils Oeste) --> " +
listaPruebas.get(0).distancia(listaPruebas.get(6));
      frase += "\nDistancia zona 1 a 8 (Cambrils Este) --> " +
listaPruebas.get(0).distancia(listaPruebas.get(7));
      frase += "\nDistancia zona 1 a 2 (Granollers (BCN)) --> " +
listaPruebas.get(0).distancia(listaPruebas.get(1));

      frase += "\n\nMiramos a que zona se unirá la zona 5 (Les Cases d'Alcanar)";
      frase += "\nDistancia zona 5 a 7 (Cambrils Oeste)--> " +
listaPruebas.get(4).distancia(listaPruebas.get(6));
      frase += "\nDistancia zona 5 a 8 (Cambrils Este)--> " +
listaPruebas.get(4).distancia(listaPruebas.get(7));
      frase += "\nDistancia zona 5 a 1 (Tarragona)--> " +
listaPruebas.get(4).distancia(listaPruebas.get(0));
      frase += "\nDistancia zona 5 a 3 (Lleida)--> " +
listaPruebas.get(1).distancia(listaPruebas.get(2));

      frase += "\n\nMiramos a que zona se unirá la zona 6 (Solsona)";
      frase += "\nDistancia zona 6 a 2 (Granollers (BCN))--> " +
listaPruebas.get(5).distancia(listaPruebas.get(1));
      frase += "\nDistancia zona 6 a 3 (Lleida)--> " +
listaPruebas.get(5).distancia(listaPruebas.get(2));
      frase += "\nDistancia zona 6 a 4 (Figueres)--> " +
listaPruebas.get(5).distancia(listaPruebas.get(3));

      System.out.println(frase);
   }
```

```
   private static void pruebaPotencia(LinkedList<ZonaRecarga> listaPruebas) {
      System.out.println("\n----------PRUEBA ENCHUFE MAS POTENCIA--------------
");
      String frase = "";
      for (ZonaRecarga vertice: listaPruebas) {
         frase += vertice.getId()+ "/ " + vertice.getEnchufeMasPotencia() + "
\n";
      }
      System.out.println(frase);
   }

   private static void pruebaJSON(LinkedList<ZonaRecarga>
lista,LinkedList<ZonaRecarga> listaPruebas) {
      System.out.println("\n----------PRUEBA LEER JSON--------------");
      // Listamos los id del fichero
      System.out.println("Lista de Zonas de Prueba leidas");
      listarIdZonas(listaPruebas);
      // No imprimimos la lista completa porque es muy larga
      //System.out.println("Lista Completa de Zonas leidas");
      //listarIdZonas(lista);


      // Comprobamos si hay algun id repetido
      System.out.println("Miramos repetidos en las lista");
      System.out.println("Hay repetidos lista Prueba: "+ repes(listaPruebas));
      System.out.println("Hay repetidos lista Completa: "+ repes(lista));

   }

   private static void listarIdZonas(LinkedList<ZonaRecarga> lista) {
      String frase = "";
      int i = 0;
      for (ZonaRecarga vertice: lista) {
         frase += i+ "/ " + vertice.getId() + " \n";
         i += 1;
      }
      System.out.println(frase);
   }

   private static boolean repes(LinkedList<ZonaRecarga> lista){
      boolean repe = false;
      for (int i = 0; i< lista.size(); i++){
         for(int j = 0; j< lista.size();j++){
            if(i != j) {
               if (lista.get(i).getId() == lista.get(j).getId()) {
                  repe = true;
               }
               if (lista.get(i).equalsCoordenadas(lista.get(j).getLatitud(),
lista.get(j).getLongitud())) {
                  repe = true;
               }
            }
         }
      }
      return repe;
   }
}
```

GrafoEstaciones

```
package datos;

import java.util.LinkedList;
import java.util.Objects;
import java.util.PriorityQueue;
import java.util.Stack;

import TADGrafoGenerico.GrafoGenerico;
import TAD_TablaHash_ListaGenerica.*;
import excepciones.*;

/**
 * Clase que contiene una instancia del GrafoGenerico para guardar las estaciones
de recarga. Contiene los algoritmos camino optimo y zonas no garantizadas
 */
public class GrafoEstaciones {
   private final GrafoGenerico <Integer, ZonaRecarga, Double> grafoEstaciones;
   private ZonaRecarga zonaRecargaInicial; // un vertice perteneciente al grafo
por el cual se comienza a hacer un recorrido

   public GrafoEstaciones (LinkedList<ZonaRecarga> listaZonasRecarga){
      grafoEstaciones = new GrafoGenerico<Integer, ZonaRecarga,
Double>(listaZonasRecarga.size()*2);

      // Añadimos los vertices
      for (ZonaRecarga zonaRecarga : listaZonasRecarga) {
         try {
            grafoEstaciones.agregarVertice(zonaRecarga.getId(), zonaRecarga);
            if (zonaRecargaInicial == null) {
               zonaRecargaInicial = zonaRecarga; // El vertice inicial de
recorridos se asigna al primer nodo del grafo
            }
         } catch (NoExiste e) {
            e.printStackTrace(); //ERROR zona de recarga es null
         }
      }
      // Añadimos las aristas tras añadir los vertices
      for (ZonaRecarga zonaRecarga : listaZonasRecarga) {
         try {
            addCarreteras(zonaRecarga, grafoEstaciones, listaZonasRecarga);
         } catch (NoExiste e) {
            e.printStackTrace();
         }
      }
   }

   private void addCarreteras(ZonaRecarga newEstacion, GrafoGenerico<Integer,
ZonaRecarga, Double> grafoEstaciones, LinkedList<ZonaRecarga> listaZonasGrafo)
throws NoExiste{
      if (zonaRecargaInicial != null) { // En caso de que no sea el primer nodo
del grafo, añadimos aristas
         // Recorremos la lista de vertices del grafo
         ZonaRecarga estacionMasCercana = null;
         boolean conectado = false;
         double distancia;

         for (ZonaRecarga vertice : listaZonasGrafo) {
            if (vertice != newEstacion) {
               distancia = vertice.distancia(newEstacion);
               if (distancia <= 40) { // Añadimos carreteras entre estaciones a
```

```
menos de 40km
                    try {
                       grafoEstaciones.agregarArista(vertice.getId(),
newEstacion.getId(), distancia);
                    } catch (YaExisteArista e) {
                       //Nada
                    }
                    conectado = true;
                } else if (estacionMasCercana == null || distancia <
newEstacion.distancia(estacionMasCercana)) {
                    estacionMasCercana = vertice;
                }
             }
          }

          // Añadimos el más cercano si no hay ninguno a 40km
          if (!conectado && estacionMasCercana!=null) {
             try {
                grafoEstaciones.agregarArista(newEstacion.getId(),
estacionMasCercana.getId(), newEstacion.distancia(estacionMasCercana));
             } catch (YaExisteArista e) {
                //Nada
             }
          }
       }
    }

    /**
     * Camino Optimo
     * @param id_origen estacion de recarga inicial
     * @param id_destino estacion de recarga final
     * @param autonomia distancia máxima del coche sin recargar
     * @param distancia puntero que guardara el resultado de la distancia total
     * @return una lista que contiene todos los nombres de los puntos de carga por
los que hay que pasar para llegar al destino
     * @exception NoExiste no se ha podido crear la lista de camino optimo
     */
    public LinkedList<String> caminoOptimo(String id_origen, String id_destino,
int autonomia, StringBuilder distancia) throws NoExiste{
       // Comprobamos parametros validos
       if(id_origen == null || id_destino == null || autonomia <= 0) {
          throw new NoExiste("Parametros no validos introducidos en funcion:
camino optimo");
       }

       // Obtenemos los vertices del grafo
       Integer destino = Integer.parseInt(id_destino);
       Integer origen = Integer.parseInt(id_origen);
       ListaGenerica<Integer> listaZonas = grafoEstaciones.getClavesVertices();
//Lista de las claves

       // Creamos las tablas auxiliares de Dijkstra
       int mida = grafoEstaciones.getMidaTablaVertices();
       TablaHashGenerica<Integer, Boolean> tablaVisitas = new
TablaHashGenerica<>(mida);
       TablaHashGenerica<Integer, Double> tablaCostes = new
TablaHashGenerica<>(mida);
       TablaHashGenerica<Integer, Integer> tablaPredecesores =new
TablaHashGenerica<>(mida);
       PriorityQueue<ZonaPrioridad> colaPrioridad = new PriorityQueue<>();
```

```
      // Inicializamos las tablas auxiliares
      for (Integer idZona: listaZonas) {
         tablaVisitas.insertar(idZona, false); // Marcamos todas como no
visitadas
         tablaCostes.insertar(idZona, null); // Marcamos a todas como su coste
correspondiente
         tablaPredecesores.insertar(idZona, null); // Marcamos a todas como que
no tienen predecesor
      }

      // Comprobamos la existencia del nodo de origen y el nodo de destino en el
grafo
      try {
         tablaVisitas.buscar(origen);
         tablaVisitas.buscar(destino);
      } catch (ElementoNoEncontrado e) {
         throw new NoExiste("El nodo de origen o de destino no se encuentra en el
grafo");
      }

      // Empezamos por el nodo inicial
      Integer vertice = origen;
      tablaCostes.insertar(vertice, 0.0); // Coste de la arista inicial es 0
      colaPrioridad.add(new ZonaPrioridad(vertice, 0.0)); // Añadimo el nodo a la
cola de prioridad


      try{ // Bucle Dijkstra
         while(!destino.equals(vertice) && !colaPrioridad.isEmpty()) {
            // Elegimos el siguiente vertice
            vertice = extraerMinimo(colaPrioridad);

            // Lo marcamos como visitados
            tablaVisitas.insertar(vertice, true);

            // Para cada adyacente al vertice comprobamos si mejora la distancia
            for (ZonaRecarga zona:grafoEstaciones.adyacentes(vertice)) {
               if (!tablaVisitas.obtener(zona.getId())){ // Si no esta visto
miramos si actualizar coste
                  Double pesoActual = tablaCostes.obtener(zona.getId());
                  Double costeArista = grafoEstaciones.valorArista(vertice,
zona.getId());
                  if (costeArista <= autonomia) { // Descartamos las aristas por
las que no puede pasar el coche
                     Double pesoNuevo = tablaCostes.obtener(vertice) +
costeArista;
                     if (pesoActual == null || pesoActual > pesoNuevo) { // Si
mejora el coste, actualizamos el coste y predecesor
                        tablaCostes.insertar(zona.getId(), pesoNuevo);
                        tablaPredecesores.insertar(zona.getId(), vertice);

                        colaPrioridad.add(new ZonaPrioridad(zona.getId(),
pesoNuevo));
                     }
                  }
               }

            }

         }
      } catch (ClaveException e) {
```

```
            e.printStackTrace(); // Error
        }

        if (!destino.equals(vertice)){
            throw new NoExiste("No se ha podido alcanzar el destino con un coche con
esa autonomia");
        }

        // Generamos la ruta
        LinkedList<String> ruta = new LinkedList<>();

        try { // Obtenemos el nombre del enchufe con mayor potencia de la zona de
recarga
            if (distancia != null) {
                distancia.append(String.format("%.3f",
tablaCostes.obtener(destino)));
            }

ruta.add(grafoEstaciones.valorVertice(destino).getEnchufeMasPotencia().getNom());
            while (!Objects.equals(destino, origen)){ //Comparamos valores
                destino = tablaPredecesores.obtener(destino);
                ruta.add(0,
grafoEstaciones.valorVertice(destino).getEnchufeMasPotencia().getNom());
            }
        } catch (ClaveException e) {
            e.printStackTrace();
        }

        return ruta;
    }


    /**
     * Camino Optimo
     * @param id_origen estacion de recarga inicial
     * @param id_destino estacion de recarga final
     * @param autonomia distancia máxima del coche sin recargar
     * @return una lista que contiene todos los nombres de los puntos de carga por
los que hay que pasar para llegar al destino
     * @exception NoExiste no se ha podido crear la lista de camino optimo
     */
    public LinkedList<String>  caminoOptimo(String id_origen, String id_destino,
int autonomia) throws NoExiste {
        return caminoOptimo(id_origen, id_destino, autonomia, null);
    }


    private Integer extraerMinimo(PriorityQueue<ZonaPrioridad> colaPrioridad)
throws NoExiste {
        // Inicializamos bucle
        Integer verticeElegido = null;

        //Elegimos el vertice de la cola de prioridad si 2 nodos tiene misma
distancia se elige el de mayor potencia
        ZonaPrioridad nodoMenor = colaPrioridad.poll(); // Extraemos y quitamos el
más pequeño
        ZonaPrioridad nodoSegundo = colaPrioridad.peek(); // Miramos el segundo más
pequeño
        try {

            if (nodoSegundo != null) {
```

```
            if (nodoMenor.getCoste() == nodoSegundo.getCoste()) {
               double potenciaMenor =
grafoEstaciones.valorVertice(nodoMenor.getId()).getEnchufeMasPotencia().getPotenc
ia();
               double potenciaSec =
grafoEstaciones.valorVertice(nodoSegundo.getId()).getEnchufeMasPotencia().getPote
ncia();
               if (potenciaMenor >= potenciaSec){
                  verticeElegido = nodoMenor.getId();
               } else{
                  verticeElegido = nodoSegundo.getId();
                  colaPrioridad.poll(); // Entonces quitamos el que tiene mayor
potencia
                  colaPrioridad.add(nodoMenor); // Volvemos a añadir el anterior
nodo que hemos quitado
               }
            } else{
               verticeElegido = nodoMenor.getId();
            }
         } else{
            verticeElegido = nodoMenor.getId();
         }
      } catch (ClaveException e) {
         e.printStackTrace();
      }


      return  verticeElegido;
   }

   /**
    * Zonas con Distancia Máxima No Garantizada
    * @param id_origen estacion de recarga inicial
    * @param autonomia distancia máxima del coche que puede recorrer sin recargar
    * @return lista que contiene aquellas zonas de recarga que no cumplen la
condicion de estar enlazadas con almentos otra zona de recarga a una distancia
maxima determinada por la autonomia
    * @throws NoExiste no se ha podido crear la lista
    */
   public LinkedList<String> zonasDistMaxNoGarantizada(String id_origen, int
autonomia) throws NoExiste{
      // Comprobamos parametros validos
      if(id_origen == null || autonomia <= 0) {
         throw new NoExiste("Parametros no validos introducidos en funcion:
distancia maxima no garantizada");
      }

      // Creamos variables y usamos una pila y una tabla hash como estructura
auxiliar del recorrido en profundidad
      Integer origen = Integer.parseInt(id_origen);
      Stack<Integer> pilaZonasAdyacentes = new Stack<>();
      TablaHashGenerica<Integer, Boolean> tablaVisitas = new
TablaHashGenerica<>(grafoEstaciones.getMidaTablaVertices());

      // Inicializamos las estructuras auxiliares
      ListaGenerica<Integer> listaZonas = grafoEstaciones.getClavesVertices();
      for(Integer vertice: listaZonas){
         tablaVisitas.insertar(vertice, false);
      }
      pilaZonasAdyacentes.add(origen);
```

```
        // Bucle de exploracion en anchura
        Integer zona;
        while (!pilaZonasAdyacentes.isEmpty()){
            // Extraemos una zona de la pila
            zona = pilaZonasAdyacentes.pop();
            tablaVisitas.insertar(zona, true); //Marcamos como visitado los nodos

            for (ZonaRecarga adyacente : grafoEstaciones.adyacentes(zona)){
                try {// Si el adyacente no esta visitado y su arista es menor que la
autonomia se añade a la pila
                    if (!tablaVisitas.obtener(adyacente.getId()) &&
grafoEstaciones.valorArista(zona, adyacente.getId()) <= autonomia){
                        pilaZonasAdyacentes.add(adyacente.getId());
                    }
                } catch (ClaveException e) {
                    e.printStackTrace();
                }
            }
        }

        // Recorremos la lista de vertices del grafo en busca de nodos no
visitados, esos seran los que no se han podido visitar
        LinkedList<String> listaZonasNoGarantizadas = new LinkedList<>();
        for (Integer vertice: listaZonas){
            try {
                if(!tablaVisitas.obtener(vertice)){
                    listaZonasNoGarantizadas.add(vertice.toString());
                }
            } catch (ClaveException e) {
                e.printStackTrace();
            }
        }

        return listaZonasNoGarantizadas;
    }

    @Override
    public String toString() {
        return "GrafoEstaciones{" +
                "grafoEstaciones=" + grafoEstaciones +
                '}';
    }
}
```

PruebaGrafoEstaciones

```
package aplicacion;

import datos.GrafoEstaciones;
import datos.ZonaRecarga;
import excepciones.NoExiste;

import java.io.FileNotFoundException;
import java.util.LinkedList;

public class PruebaGrafoEstaciones {
   public static void main(String[] args) throws FileNotFoundException {
      // Cargamos los datos de JSON al grafo
      LinkedList<ZonaRecarga> listaZonas =
ZonaRecarga.leerJson("src/main/resources/icaen.json");
      LinkedList<ZonaRecarga> listaZonasReducida =
ZonaRecarga.leerJson("src/main/resources/pruebas.json");

      // Añadimos las estaciones al grafo
      System.out.println("\n----------PRUEBA CONSTRUCTOR--------------");
      System.out.println("Pasamos al constructor los datos a guardar en el
grafo");
      GrafoEstaciones grafoEstaciones = new GrafoEstaciones(listaZonas);
      GrafoEstaciones grafoReducido = new GrafoEstaciones(listaZonasReducida);
      System.out.println(grafoReducido); // Imprimimos solo el grafo reducido
porque es más facil de ver

      // Prueba de camino optimo
      pruebaCaminoOptimo(grafoEstaciones);

      // Algoritmo de zonas no alcanzables
      pruebaDistMaxNoGarantizada(grafoEstaciones, grafoReducido);
   }

   private static void pruebaDistMaxNoGarantizada(GrafoEstaciones
grafoEstaciones, GrafoEstaciones grafoReducido) {
      System.out.println("\n----------PRUEBA ZONAS DISTANCIA MAXIMA NO
GARANTIZADA--------------");
      try {
         System.out.println("En el grafo reducido escogemos el nodo 1 (Tarragona)
y miramos a que zonas no se puede acceder con una autonomia de 300km");
         System.out.println(grafoReducido.zonasDistMaxNoGarantizada("1", 300) +
"son zonas que no estan conectadas al grafo");

         System.out.println("\nEscogemos como origen la zona 9165 (Cambrils), las
zonas que no se pueden alcanzar");
         System.out.println("Probamos autonomia de 10000
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("9165", 10000));
         System.out.println("Probamos autonomia de    40
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("9165", 40));
         System.out.println("Probamos autonomia de    39
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("9165", 39));
         System.out.println("Probamos autonomia de    20
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("9165", 20));

         System.out.println("\nTomamos como origen la zona 9904800 (Móra
d'Ebre)");
         System.out.println("Probamos autonomia de 25
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("9904800", 25));

         System.out.println("\nUsamos como origen la zona 33852430 (Almedinilla,
```

```
Córdoba), es una zona no conectada a la red");
            System.out.println("Probamos autonomia de 5000
km:"+grafoEstaciones.zonasDistMaxNoGarantizada("33852430", 5000));
         } catch (NoExiste e) {
            e.printStackTrace();
         }


         try {
            System.out.println("\nIntentamos usar un nodo no presente en el grafo");

System.out.println("ERROR:"+grafoEstaciones.zonasDistMaxNoGarantizada("1",
5000));
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }

         try {
            System.out.println("\nIntentamos usar un vertice nulo");

System.out.println("ERROR:"+grafoEstaciones.zonasDistMaxNoGarantizada(null,
5000));
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }

         try {
            System.out.println("\nIntentamos usar una autonomia de 0");

System.out.println("ERROR:"+grafoEstaciones.zonasDistMaxNoGarantizada("33852430",
0));
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }
      }

      private static void pruebaCaminoOptimo(GrafoEstaciones grafoEstaciones) {
         System.out.println("\n----------PRUEBA CAMINO OPTIMO--------------");
         LinkedList<String> ruta;
         StringBuilder distancia;
         try {
            // Probamos una ruta entre dos zonas de recarga cercanas
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta entre dos zonas de recarga
cercanas (9165-->9168)");
            ruta = grafoEstaciones.caminoOptimo("9165", "9168", 100, distancia);
            System.out.println(distancia + " km --> " +ruta);

            // Probamos una ruta entre dos zonas lejanas y con autonomia baja
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta entre dos zonas lejanas y con
autonomia baja (9165-->34252288) 20km");
            ruta = grafoEstaciones.caminoOptimo("9165", "34252288", 20, distancia);
            System.out.println(distancia + " km --> " +ruta);

            // Probamos una ruta entre dos zonas lejanas y con autonomia media
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta entre dos zonas lejanas y con
autonomia media (9165-->34252288) 50km");
            ruta = grafoEstaciones.caminoOptimo("9165", "34252288", 50, distancia);
            System.out.println(distancia + " km --> " +ruta);
```

```
            // Probamos una ruta entre dos zonas muy lejanas con autonomia alta 150
km
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta entre dos zonas muy lejanas y
con autonomia alta (13361299-->7562018) 150km");
            ruta = grafoEstaciones.caminoOptimo("13361299", "7562018", 150,
distancia);
            System.out.println(distancia + " km --> " +ruta);

            // Probamos una ruta entre dos zonas muy lejanas con autonomia baja 30
km
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta entre dos zonas muy lejanas y
con autonomia media (13361299-->7562018) 30km");
            ruta = grafoEstaciones.caminoOptimo("13361299", "7562018", 30,
distancia);
            System.out.println(distancia + " km --> " +ruta);
         } catch (NoExiste e) {
            System.out.println("ERROR"); // Fallo
         }

         try {// Probamos una ruta pasando como parametro una autonomia no valida
            System.out.println("\nProbamos una ruta pasando como parametro una
autonomia no valida");
            ruta = grafoEstaciones.caminoOptimo("13361299", "7562018", -5);
            System.out.println("ERROR" + ruta); // ERROR
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }

         try {// Probamos una ruta pasando como parametro una zona de destino no
presente en el grafo
            System.out.println("\nProbamos una ruta pasando como parametro una zona
de destino no presente en el grafo (9165->1)");
            ruta = grafoEstaciones.caminoOptimo("9165", "1", 30);
            System.out.println("ERROR" + ruta); // ERROR
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }

         try {// Probamos una ruta pasando como parametro una zona de destino nulo
            System.out.println("\nProbamos una ruta pasando como parametro una zona
de destino nulo");
            ruta = grafoEstaciones.caminoOptimo(null, "13361299", 30);
            System.out.println("ERROR" + ruta); // ERROR
         } catch (NoExiste e) {
            System.out.println(e.getMessage());
         }

         try {// Probamos una ruta pasando como parametro una zona de destino igual
al origen
            distancia = new StringBuilder("-Distancia Recorrida: ");
            System.out.println("\nProbamos una ruta pasando como parametro una zona
de destino igual al origen (13361299-->13361299) 30km");
            ruta = grafoEstaciones.caminoOptimo("13361299", "13361299", 30,
distancia);
            System.out.println(distancia + " km -->" + ruta);
         } catch (NoExiste e) {
            System.out.println("ERROR"); // ERROR
         }
```

```
      try {
         System.out.println("\nProbamos una ruta entre dos zonas distantes y con
autonomia muy baja (13361299-->34252288) 10km");
         ruta = grafoEstaciones.caminoOptimo("13361299", "34252288", 10);
         System.out.println("ERROR" + ruta);
      } catch (NoExiste e) {
         System.out.println(e.getMessage());
      }

   }
}
```

AnálisisCostesTemporales:

```
package aplicacion;

import datos.GrafoEstaciones;
import datos.ZonaRecarga;
import excepciones.NoExiste;

import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.LinkedList;

public class AnalisisCostesTemporales {
   static long inicial;
   static long[] contador = new long[5];
   static final int AUTONOMIA = 40; // MODIFICAR ESTE VALOR

   public static void main(String[] args) throws FileNotFoundException {

      LinkedList<ZonaRecarga> listaZonas =
ZonaRecarga.leerJson("src/main/resources/icaen.json");
      GrafoEstaciones grafoEstaciones = new GrafoEstaciones(listaZonas);

      // Inicializamos el vector a 0
      Arrays.fill(contador, 0);

      int[] nodos = new int[5];
      try {
         StringBuilder distancia = new StringBuilder("[");
         nodos[0] = grafoEstaciones.caminoOptimo("9794082", "7562169", AUTONOMIA,
distancia).size();
         distancia.append(" , ");
         nodos[1] = grafoEstaciones.caminoOptimo("9794082", "35349720",
AUTONOMIA, distancia).size();
         distancia.append(" , ");
         nodos[2] = grafoEstaciones.caminoOptimo("35349720", "7562243",
AUTONOMIA, distancia).size();
         distancia.append(" , ");
         nodos[3] = grafoEstaciones.caminoOptimo("35349720", "29786231",
AUTONOMIA, distancia).size();
         distancia.append(" , ");
         nodos[4] = grafoEstaciones.caminoOptimo("7562086", "7562247", AUTONOMIA,
distancia).size();
         distancia.append("]");
         System.out.println("Las distancias de las rutas son: \n"+distancia);
      } catch (NoExiste e) {
         e.printStackTrace();
      }


      for (int i = 0; i < 100; i++){
         medirTiempo(grafoEstaciones);
      }

      for(int i = 0; i < contador.length; i++){
         System.out.println("Ruta " + (i+1) + ": tiene " + nodos[i] + "
estaciones-> "+ contador[i]/100.0);
      }

   }

   private static void medirTiempo(GrafoEstaciones grafoEstaciones) {
```

```
 /*
Ruta 1: 9794082 (Molins de Rei) --> 7562169 (Tortosa)
   * Origen:
   "latitud": "41.412473739646",
   "longitud": "2.014127862566"
   * Destí:
   "latitud": "40.794775",
   "longitud": "0.525542" */

   try {
      iniciar();
      grafoEstaciones.caminoOptimo("9794082", "7562169", AUTONOMIA, null);
      contador[0] += finalizar();
   } catch (NoExiste e) {
      e.printStackTrace();
   }

   /*
   Ruta 2: 9794082 (Molins de Rei) --> 35349720 (Argentona)
      * Origen:
      "latitud": "41.412473739646",
      "longitud": "2.014127862566"

      * Destí:
      "latitud": "41.5555823",
      "longitud": "2.4005556" */

   try {
      iniciar();
      grafoEstaciones.caminoOptimo("9794082", "35349720", AUTONOMIA, null);
      contador[1] += finalizar();
   } catch (NoExiste e) {
      e.printStackTrace();
   }

   /*
   Ruta 3: 35349720 (Argentona) --> 7562243 (Sant Feliu de Guíxols)
      * Origen:
      "latitud": "41.5555823",
      "longitud": "2.4005556"

      * Destí:
      "latitud": "41.780674",
      "longitud": "3.022077"*/

   try {
      iniciar();
      grafoEstaciones.caminoOptimo("35349720", "7562243", AUTONOMIA, null);
      contador[2] += finalizar();
   } catch (NoExiste e) {
      e.printStackTrace();
   }

   /*
   Ruta 4: 35349720 (Argentona) --> 29786231 (Montblanc)
      * Origen:
      "latitud": "41.5555823",
      "longitud": "2.4005556"

      * Destí:
      "latitud": "41.375768",
```

```
            "longitud": "1.163327"*/

        try {
            iniciar();
            grafoEstaciones.caminoOptimo("35349720", "29786231", AUTONOMIA, null);
            contador[3] += finalizar();
        } catch (NoExiste e) {
            e.printStackTrace();
        }

        /*
        Ruta 5: 7562086 (Tortosa) --> 7562247 (Figueres)
            * Origen:
            "latitud": "40.814151",
            "longitud": "0.515161"

            * Destí:
            "latitud": "42.268984",
            "longitud": "2.966869"     */

        try {
            iniciar();
            grafoEstaciones.caminoOptimo("7562086", "7562247", AUTONOMIA, null);
            contador[4] += finalizar();
        } catch (NoExiste e) {
            e.printStackTrace();
        }
    }


    private static void iniciar(){
        inicial = System.currentTimeMillis();
    }

    private static long finalizar(){
        return System.currentTimeMillis() - inicial;
    }
}
```