
# TRABAJO FIN DE CARRERA

**TÍTULO DEL TFC: Rutalc: aplicación de soporte a viajeros en entornos rurales para Android**

**TITULACIÓ: Ingeniería Técnica de Telecomunicaciones, especializada en Sistemas de Telecomunicación**

**AUTOR: Carlos Fuertes del Pozo**

**DIRECTOR: Esther Salamí San Juan**

**FECHA: 24 de abril 2015**

**Título:** Rutalc: aplicación de soporte a viajeros en entornos rurales para Android

**Autor:** Carlos Fuertes del Pozo

**Director:** Esther Salamí San Juan

**Fecha:** 24 de abril 2015

**Resumen**

En este documento se detallan las herramientas y algoritmos empleados para el desarrollo de la aplicación *Rutalc*. Se trata de una aplicación de soporte al turismo rural para terminales que corren bajo el sistema operativo Android.

La aplicación permite al usuario seleccionar la tipología de sitios que desea visitar, en una zona determinada o entre dos localizaciones concretas, y le proporciona un conjunto de puntos que coinciden con el tipo de turismo seleccionado y una serie de caminos que interconectan los diferentes puntos de interés con independencia de su tipología. Estos caminos pueden discurrir bien por asfalto, tierra o por ambos tipos de vía, la cual es escogida también por el usuario.

La aplicación funciona con independencia de la disponibilidad de la red móvil. Trabaja con mapas proporcionados por la plataforma OpenStreetMap y para su visualización se utiliza una librería de terceros que permite la integración de éstos con Android. Los datos utilizados son extraídos de los mapas proporcionados por la plataforma anteriormente mencionada.

El documento detalla las diferentes tecnologías utilizadas para el desarrollo, cómo se ha procesado la información extraída de la base de datos, y los algoritmos utilizados para la elaboración de las diferentes rutas. También se muestran varios ejemplos del resultado obtenido con este desarrollo.

**Title:** Rutalc: support application for travelers in rural environments for Android

**Author:** Carlos Fuertes del Pozo

**Director:** Esther Salamí San Juan

**Date:** April, 24th 2015

## Overview

This document details the tools and algorithms used for the development of *Rutalc*, an agricultural tourism oriented application. It is a mobile application aimed at terminals running under the Android operating system.

The application allows the user to select the type of sites he wants to visit in a given area or between two specific locations, and provides a set of points that match the selected type of tourism and a series of paths that interconnect the different points of interest regardless of their type. These paths may run either on asphalt, dirt or both types of track, which is chosen by the user.

The application operates independently of the availability of the mobile network. It works with maps provided by the OpenStreetMap platform and for viewing a library of third parties is used, which allows the integration of these with Android. Data used are extracted from the maps provided by the above platform.

The document details the different technologies used for development, how information is processed in the database to generate the different routes, and the algorithms used to perform this task. Different examples of results obtained with this development are also shown.

Con estas pocas líneas quisiera dar las gracias a mis padres por empujarme siempre hacia delante, a mi hermano por siempre a su manera hacerme exigir un poco más. A la gente, que durante este tiempo ha tenido que aguantarme oír hablar sobre este proyecto.

Y en especial quiero destacar a mi tutora por brindarme la oportunidad y confianza de poder desarrollar este pequeño sueño que hoy es más real.

# ÍNDICE

## ÍNDICE FIGURAS Y TABLAS

## FIGURAS

# CAPITULO 1. INTRODUCCIÓN

Tras el cambio de paradigma de la sociedad actual nos encontramos con un amplio y variado abanico de herramientas a nuestra disposición, con aplicaciones en ámbitos tan diversos como el entretenimiento, el ocio, la salud, la educación, o la exploración del entorno, y es en este último en el que enmarca el proyecto que presentamos a continuación.

## 1.1. Motivación del proyecto

Desde el año 2001, con la formación de la Asociación de amigos de Alcubilla de Avellaneda (localidad de la provincia de Soria), se han llevado a cabo una serie de iniciativas para revitalizar el pueblo y evitar que éste cayese en el olvido. Como consecuencia de ello, el pasado 8 de agosto recibía el premio provincial de turismo de la Diputación de Soria.

Intentando continuar en esa línea, surge la idea de aprovechar la tecnología existente para poner a la disposición del viajero una serie de información gráfica que le haga disfrutar más de su paso por Alcubilla.

En el año 2006 era impensable plantear que en apenas 2 o 3 años se iba a vivir un cambio tan considerable en el sector de la telefonía móvil que reinventaría totalmente el concepto conocido hasta entonces. Más lejos quedaba el pensar que la gran Nokia terminaría desbancada, ya no por otras compañías como LG, Samsung o Sony, que trabajaban con sistemas operativos similares, si no por la llegada de un tercero con un sistema operativo que lo revolucionaría todo **(Fig. 1.1)**.

De modo que hoy en día una gran parte de la sociedad es poseedora, ya no de un teléfono móvil, sino de un *Smarthphone* (pequeñas computadoras más potentes que algunos ordenadores que por allá en el 2005 se pudiesen tener) con el sistema operativo Android. Aunque no podemos obviar la presencia de diversas alternativas, éstas no tienen tanta penetración como las del robot verde. Por ello y por tratarse de código abierto con una gran comunidad a sus espaldas, se decidió programar una aplicación basada en Android.

Si bien es cierto que en la actualidad gran parte de la sociedad dispone de un *Smarthphone*, no es tan cierto que exista cobertura en todo el territorio nacional, tal y como se muestra en **(Fig. 1.2)**. Por esa razón, y ser el caso que nos atañe en nuestra zona de ensayo, se tomó la decisión de que la aplicación fuera capaz de trabajar de modo *offline*, para que la no disposición de la red no afectase en el funcionamiento de la misma.

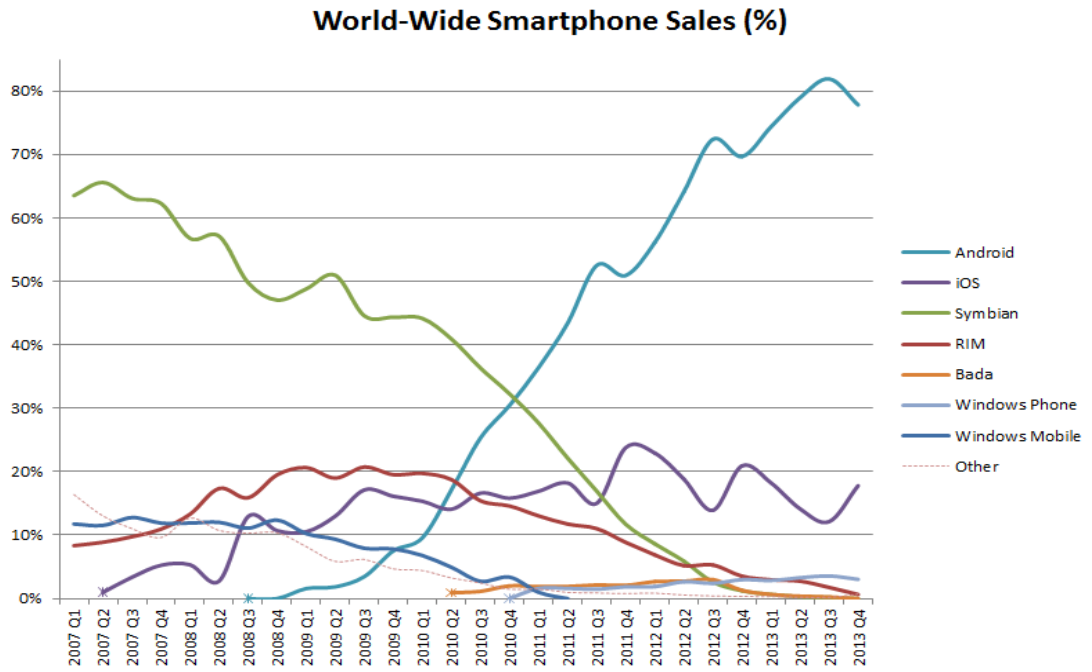



**Fig. 1.1** Evolución de ventas en el sector móvil a nivel global basadas en los diferentes sistemas operativos [1]

Por lo general, cuando un visitante quiere realizar algún tipo de ruta, busca cómo llegar desde un punto de partida a un punto de destino. Si además quiere aprovechar para visitar puntos de interés que se hallan en el trayecto, tales como fuentes, castillos, etc., tiene que hacer una segunda búsqueda.

En dicha situación, sería de utilidad disponer de una aplicación que permitiera al usuario indicar los puntos de origen y destino y el tipo de puntos de interés que le gustaría visitar. El resultado sería el conjunto de caminos que le permitirían llegar del punto origen al punto destino pasando por los diferentes puntos de interés. Para mayor comodidad, la ruta debería mostrarse al usuario pintada sobre el mapa de la zona.

Investigando en la tienda de aplicaciones de GOOGLE para su sistema operativo, se encontraron diferentes opciones, de las cuales ninguna proporcionaba rutas en función de los puntos de interés del lugar, sino que el usuario registra, coge rutas de otros usuarios o se le indica cómo llegar de A a B por el camino más rápido (Oruxmaps, Google Maps, Endomondo, Locus, etc.).

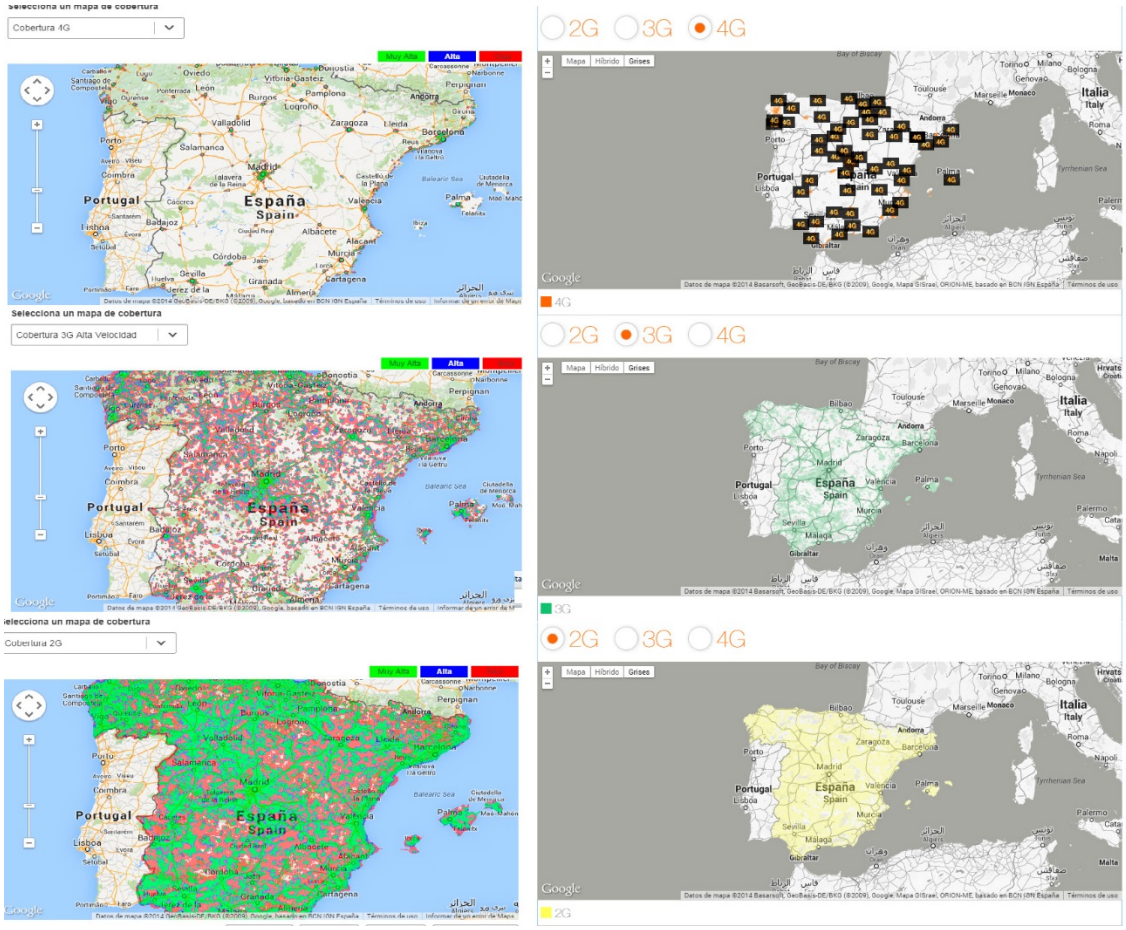



**Fig. 1.2** Mapa de la cobertura móvil en España. De izquierda a derecha y de arriba abajo se ven las operadoras Vodafone y Orange para 4G, 3G y 2G (datos extraídos de [2] y [3])

Por todo ello, se decidió desarrollar una aplicación

- que dé a conocer el entorno,
- adaptada a la realidad de la sociedad,
- que no dependa del estado de la red móvil,
- y que proporcione información interconectada entre sí
- de una manera amigable y útil para el usuario.

Bajo esta premisa nace el proyecto *Rutalc*, una herramienta de apoyo al turista para la exploración de la zona rural de Alcubilla de Avellaneda.

## 1.2. Objetivos a alcanzar con la realización del proyecto

Como se ha planteado en el apartado anterior, el objetivo general del trabajo es desarrollar una aplicación basada en el sistema operativo Android que sea

capaz de mostrar al usuario una serie de rutas óptimas que le permitan explorar una determinada zona, independientemente del estado de la red.

Para conseguirlo, nos planteamos los siguientes retos concretos:

- Obtener los mapas del entorno deseado
- Crear una base de datos (BD) con información extraída de los mapas
- Desarrollar una aplicación en Android que permita:
    - Cargar los datos de la base de datos.
    - Procesarlos según las preferencias del usuario.
    - Elaborar un conjunto de rutas para ir de origen a destino pasando por puntos de interés.
    - Visualizar el mapa de la zona, con representación gráfica de los puntos de interés que sean accesibles y las rutas propuestas.

## 1.3. Estructura del informe

El resto del documento está organizado en los siguientes capítulos:

- Tecnologías utilizadas. En este capítulo se analizan las diferentes tecnologías con las cuales se podía desarrollar el proyecto, cuáles son las que finalmente se han utilizado y el por qué de dicha decisión.

- Algoritmos. En primer lugar, se realiza una breve introducción teórica sobre los términos empleados en la explicación y los algoritmos existentes que cubren problemáticas similares a la que se desea solucionar. A continuación, se expone el algoritmo propuesto para solucionar nuestro problema particular.

- Desarrollo de la aplicación. Se muestra cómo hemos planteado implementar la aplicación en rasgos generales y el diseño de las clases que intervienen.

- Resultados. En este capítulo se muestran los resultados obtenidos, incluyendo capturas de pantalla obtenidas durante la ejecución de la aplicación.

- Conclusiones y trabajos futuros. Finalmente, se resumen las principales conclusiones obtenidas durante el desarrollo del proyecto y se comentan posibles líneas del futuro de la aplicación.

# CAPITULO 2. Tecnologías utilizadas

Este capítulo se ha esquematizado en tres apartados:

- Sistema operativo. Se hablará sobre las diferentes alternativas que hay en el mercado y diferentes modos de realizar la tarea.
- Software. El apartado anterior hace referencia a los sistemas operativos, mientras que en este apartado se mostrarán los diferentes programas y *plugins* que se pueden utilizar para Android.
- Hardware. Para finalizar se verán que sensores del terminal móvil pueden ser usados para mejorar la experiencia.

## 2.1. Sistema operativo

Antes de empezar, es importante conocer los diferentes sistemas operativos que hay en el mercado, qué ofrecen y cuáles son los más utilizados allí donde vamos a querer lanzar nuestra aplicación.

Para el caso que nos atañe nos fijaremos en la cuota de mercado de los diferentes sistemas operativos en el territorio nacional. Según los datos consultados [4], vemos que el sistema operativo más utilizado es Android, seguido de iOS, WindowsPhone y otros, tal y como se puede ver en la **Fig.2.1.**

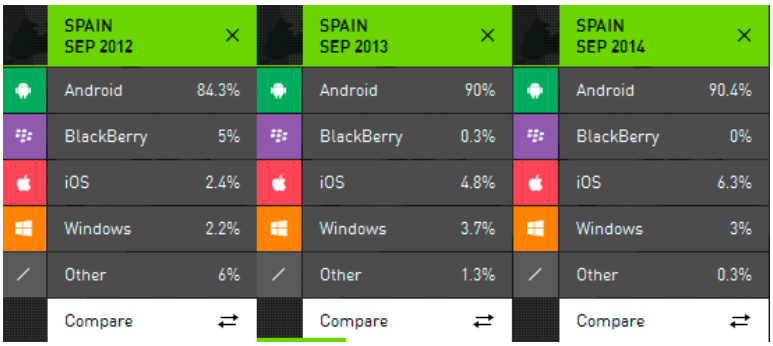



**Fig.2.1** Cuota de mercado de los diferentes sistemas operativos en Septiembre de 2012, 2013 y 2014 en España [4]

De manera que ahora existen dos posibilidades, crear una aplicación nativa o buscar una plataforma que nos permita programar para los diferentes sistemas operativos. En este último caso podríamos usar plataformas como Unity3D, PhoneGp o RubyMotion, por mencionar algunas.

Si se analizan los beneficios de usar el segundo método tendríamos que programando una única vez nos aseguraríamos que la aplicación funcionase

en todas las plataformas que queramos. Pero por otro lado, estaríamos limitados a lo que el programa nos permitiese realizar y al ecosistema de *plugins* que tuviese.

Por el contrario, si se decide programar nativamente, se dispone de toda la documentación oficial, más la gran comunidad que tiene a sus espaldas y una mayor libertad para programar.

Por estos motivos se decide programar nativamente en Android pero con Eclipse y no con Android Studio, por llevar este primero más tiempo y existir más referencias.

Uno de los parámetros a configurar cuando creas una nueva aplicación es la versión mínima del sistema operativo para la cual vas a programar. Tras analizar y ver cuáles eran las versiones más extendidas se decide programar a partir de la 2.3 en adelante, verificado el correcto funcionamiento en un terminal con versión 4.x.x.

## 2.2. Software

En este apartado se hablará sobre los posibles servicios a usar y qué librerías necesitaríamos exportar a nuestro entorno de desarrollo para poder trabajar con dicho servicio.

Por una de las cosas que se caracteriza Android es por ser de código abierto, es por esta razón y haciendo una excepción para el apartado de diseño gráfico, que todas las herramientas seleccionadas son de código abierto y/o libre distribución.

### 2.2.1. Mapas

Como se ha comentado anteriormente, se desea realizar una aplicación que trabajara con mapas. En ese sentido las opciones que existen son bastante limitadas. Las dos opciones más conocidas son Google Maps y OpenStreetMap.

#### Google Maps

Como ya indica el nombre, se trata de la herramienta de Google aplicada a mapas. Se trata de un servicio cerrado en el cual son los propios trabajadores de Google quienes introducen y gestionan los datos que en dicho servicio podemos consultar. En cualquier caso, siempre podemos contactar con ellos para comunicar que hay algún fallo y ellos introducirán dichos cambios.

Existe también una herramienta para introducir datos llamada Map Maker, pero no está disponible para el territorio español a fecha de hoy. Otra alternativa es crearnos nuestras propias capas con Google Maps Engine y luego mirar cómo usar dichas capas en nuestra aplicación.

El otro gran tema a tener en cuenta es la posibilidad de trabajar con los mapas de Google de manera offline. Si bien es cierto que en la propia aplicación de Google Maps se puede, no está tan claro si es posible realizar dicha tarea en nuestra propia aplicación. Y además la licencia de Google Maps no permite usarla en conjunción con otros servicios del mismo estilo [5].

Y por finalizar tenemos el hecho de que el SDK proporcionado para desarrollar en Android no dispone de la librería necesaria para usar sus mapas. Para ello hay que bajarse otra librería, obtener una clave de API y realizar los cambios necesarios en el manifiesto.

**OpenStreetMap**

Es la alternativa más potente. Se trata de “un proyecto colaborativo para crear mapas libres y editables” [6].

Una de las grandes diferencias respecto al anterior servicio es el hecho de que cualquier persona puede ir a la página de OpenStreetMap, registrarse y empezar a editar una determinada zona. Al cabo de unas horas, dichos datos estarían disponibles para su descarga y así luego poder trabajar con ellos en nuestra aplicación.

De entrada eso ya supone un gran avance respecto a Google Maps, ya que para el desarrollo de nuestro proyecto será necesario editar ciertas áreas, descargar la información y luego trabajar con ella.

Por el contrario, en este caso el contrapunto es la no oficialidad de una librería. Las opciones que ahora se tienen son OsmDroid y MapsForge. La diferencia crucial entre ambas librerías es el hecho de que MapsForge utiliza mapas vectoriales binarios, los cuales terminan teniendo un impacto enorme en la memoria de la aplicación ya que pesan menos que las teselas que guarda OsmDroid.

Por otra parte permite utilizar diferentes marcadores para los puntos de interés y dispone de una herramienta que a través de Osmosis nos permite crear los mapas para usarlos de modo offline. También con esta herramienta podemos crear nuestros propios estilos visuales para renderizar los mapas. De todo ello, concluimos que la librería que mejor se adapta a nuestros intereses es Mapsforge.

Resumiendo, tenemos que OpenStreetMap es el servicio que nos permite editar datos, obtenerlos, procesarlos y finalmente trabajar con ellos sobre un

mapa el cual no es necesario que sea de una URL si no que lo podemos tener alojado en nuestro terminal móvil.

### 2.2.2. OSM

OSM es el formato de datos propios de OpenStreetMap. Se trata de un xml que contiene una lista de datos primitivos tales como los nodos, vías y relaciones. Ya bien sea desde su página web u otros lugares nos podemos descargas dicho contenido para luego procesarlo como mejor nos convenga para nuestro uso. En nuestro caso particular, a partir de un fichero ".osm" requerimos obtener un ".map" y un ".sql". Para ello se necesitan tres herramientas:

- **Osm2Sqlite:** Se trata de un script el cual se puede hacer correr tanto en Windows como en Ubuntu el cual nos convierte el xml de OpenStreetMap en una base de datos con la cual poder trabajar en Android.

- **SQLite Manager:** Se trata de una extensión para el navegador FireFox con el cual podremos consultar la base de datos y realizar los cambios necesarios para poder trabajar con dicho fichero en Android.

- **Osmosis:** Consiste en una aplicación de línea de comandos java para procesar los datos OSM [7]. Es una herramienta muy potente, aunque en nuestro caso solo la utilizaremos para instalar el *plugin* que proporciona MapsForge para obtener el mapa vectorial a partir de los datos OSM. Para ello usaremos el *plugin* "mapsforge-map-writer".

### 2.2.3. Gráficos

Por último, está el material utilizado para el aspecto visual de la aplicación, entiéndase iconos y portadas de la aplicación. Para la creación de dicho contenido, se han utilizado las herramientas de Adobe Photoshop e Illustrator.

Como patrón se ha intentado usar el documento de diseño proporcionado por Android.

## 2.3. Hardware

Si bien es cierto que los terminales móviles cada día tienen más sensores, hemos considerado necesario utilizar solo el GPS (Sistema de Posicionamiento Global) para mejorar la experiencia de usuario, y ofrecerle su posición actual dentro del camino que está recorriendo.

# CAPITULO 3. Algoritmos y Grafos

Antes de entrar en materia y comenzar a hablar de qué y cómo se han hecho las cosas, es necesario dar unas nociones básicas de algunos conceptos utilizados para el desarrollo del núcleo de la aplicación. Por ello se hablará sobre algoritmos y grafos.

Si se consulta en la web de la RAE vemos que la definición que nos dan es la siguiente:

*"Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema."*

De manera que se puede decir que el día en que se nos enseñó a resolver ecuaciones de primer grado no estaban haciendo más que enseñarnos un algoritmo de resolución de ecuaciones de primer grado.

Atendiendo la funcionalidad de los mismo podríamos encontrar la siguiente clasificación [8]: cuánticos, búsqueda de raíces, clasificación, factorización de enteros, ordenamiento, precisión arbitraria, recorte, evolutivos, geométricos, SAT, búsqueda, compresión, computación gráfica y de grafos.

La definición es aplicable a cualquier proceso, pero a nosotros ahora mismo nos interesan aquellos algoritmos que nos proporcionan soluciones factibles a problemas de optimización, es decir, algoritmos de grafos. En función de las necesidades que se tengan y de lo que se quiera obtener se usará un tipo u otro de algoritmo de grafo.

## 3.1. Grafos

Un grafo es la representación gráfica de la unión de una serie de puntos. Un ejemplo sería la imagen de la topología de una red o el mapa de carreteras. Las carreteras se conocen como aristas mientras que los pueblos a los que se llega por dicha carretera se denominan vértices.

De modo que, si volvemos a la definición de los términos que dan pie a este apartado, tenemos que un grafo es la representación gráfica de un conjunto de vértices con sus correspondientes aristas. En la **Fig. 3.1** se pueden apreciar algunos ejemplos.

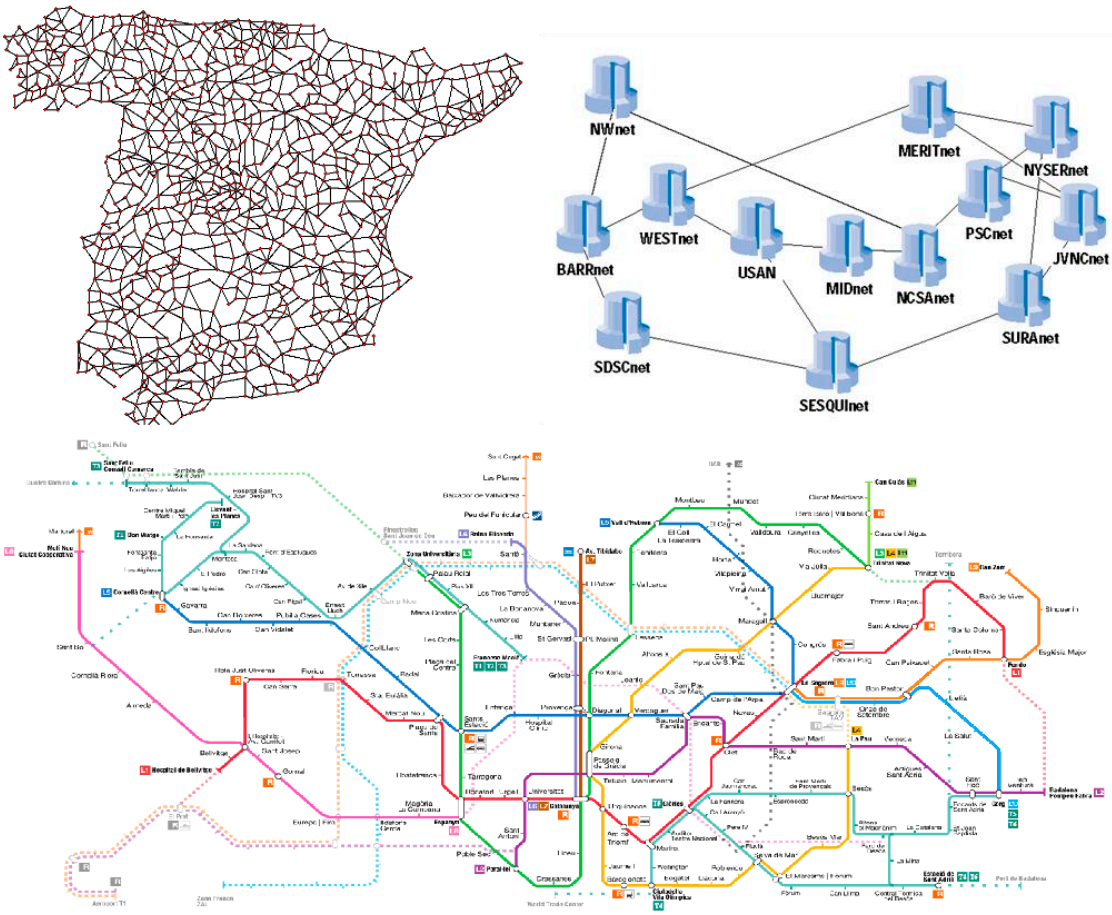



**Fig. 3.1** Ejemplo de grafos, de izquierda a derecha y de arriba abajo: grafo de las carreteras de España [9], grafo de la topología NSFnet [10] y grafo del metro de Barcelona [11]

Queda un único elemento no mencionado con relación a las aristas y vértices, la valencia. La valencia nos indica el número de aristas que tiene un nodo.

Atendiendo a la conexión entre las aristas y los vértices que conecta, podemos distinguir los siguientes tipos de grafos, no excluyentes entre sí:

- Digrafos: son aquellos en los cuales las aristas están dirigidas (**Fig. 3.2** derecha).
- Multigrafos: son aquellos en los cuales hay más de una arista que conecta dos mismos vértices (**Fig. 3.2** izquierda).

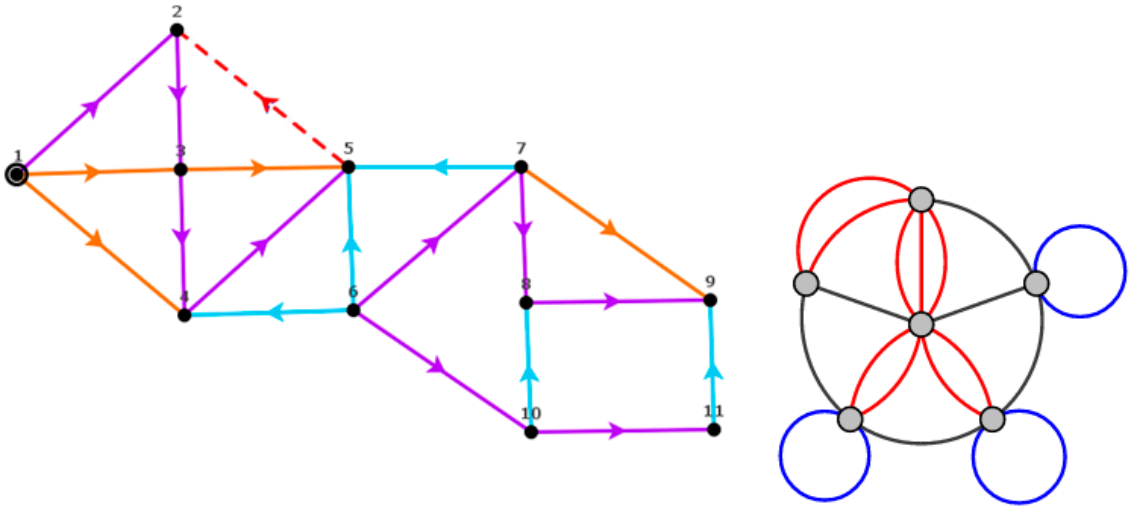



**Fig. 3.2** Ejemplo de digrafo [12] (izquierda) y multigrafo [13] (derecha)

En ocasiones, la resolución de un problema de grafos pasa por obtener un árbol, ya sea minimal o maximal, que no es más que un grafo conexo (que existe al menos una arista que una los vértices) sin ciclos, de modo que se tiene algo como lo de la **Fig. 3.3.**

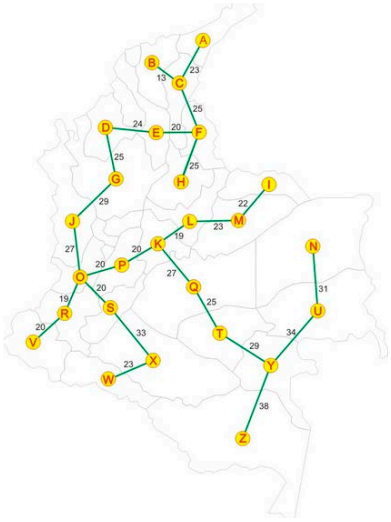



**Fig. 3.3** Ejemplo de árbol minimal [14]

## 3.2. Algoritmos de Grafo

En nuestro caso, necesitamos un algoritmo que dado una serie de puntos nos proporcione la ruta más corta que pase por todos o la mayor parte de puntos posibles hasta llegar a nuestro destino.

Este es el enunciado de nuestro problema, y para hallar la solución no usaremos un solo algoritmo. El problema a resolver se asemeja al conocido como el problema del agente viajero, que consiste en como un comercial puede pasar por todos los pueblos y volver al punto de partida sin pasar dos veces por el mismo sitio. Su resolución se realiza mediante el algoritmo de Christofides. Nuestro problema, a pesar de ser similar al que se acaba de mencionar, difiere en algunos putos que se nombran a continuación:

- En nuestra aplicación se permite que el usuario parta de una localidad A y quiera llegar a otra localidad B, siendo A y B localidades distintas. En cambio, en el algoritmo de Christofides el punto de origen y destino coinciden.

- La otra diferencia tiene que ver con cómo construye Christofides la solución para poder recorrer el grafo. Dicho algoritmo requiere poder balancear las valencias de los nodos hacia un estado par. Pero el hecho de que los caminos de la zona suelen terminar en callejones sin salida y que muchas veces existe una única vía de acceso, provoca que no sea posible hacer el balanceo. Por eso decimos que por la topografía de la zona, no es posible aplicar directamente el algoritmo de Christofides.

Por todo ello, se decide utilizar el algoritmo de Christofides como referencia y punto de apoyo para poder avanzar en la elaboración de nuestro propio algoritmo.

### 3.2.1. Algoritmo de Dijkstra

Este algoritmo fue descrito por Edsger Dijkstra en 1959. Dicho algoritmo viene a dar solución al problema del camino más corto.

El primer requisito para poder aplicar el algoritmo que a continuación se explica, es que sea un grafo conectado (que no existan nodos sin conectar entre sí) con aristas de peso positivo.

Para poder empezar a resolver es necesario conocer el punto de partida. A partir de este nodo se mira el coste generado para llegar a los nodos adyacentes y se guarda este dato. Una vez que se tiene esta información se busca el que tiene el coste más bajo.

Encontrado el nuevo nodo se repite la operación inicial, mirar cuánto cuesta llegar a los nodos adyacentes, pero con la novedad de que ya nunca más se volverá a consultar cuánto cuesta llegar a los nodos de los que venimos. Y además sumaremos al coste lo que nos costó llegar desde el nodo actual.

Veamos un ejemplo. Imaginemos que hemos ido de A hasta B y el peso la arista AB tiene un coste de 10. Estando ahora en B, miramos cuánto cuesta llegar al nodo C. Si la arista BC tiene un coste 5, el coste que guardaremos

será 10+5, ya que interesa saber el camino que proporciona el coste mínimo hasta llegar a destino.

Una vez explorados todos los nodos adyacentes se vuelve a seleccionar el que tiene menor peso y se repite todo el procedimiento. Así hasta que se hayan visitado todos los nodos.

En la **Fig. 3.4** se puede apreciar el proceso arriba mencionado representado gráficamente. En este caso, para ir de "*s*" a "*z*" el camino sería "*syz*".

Como se puede apreciar, esta solución solo nos da el camino más corto para llegar (siguiendo la nomenclatura de la **Fig. 3.4**) de "*s*" a cualquier punto, pero no de cualquier punto a cualquier punto. Para ello se tendría que recalcular variando el punto de inicio.

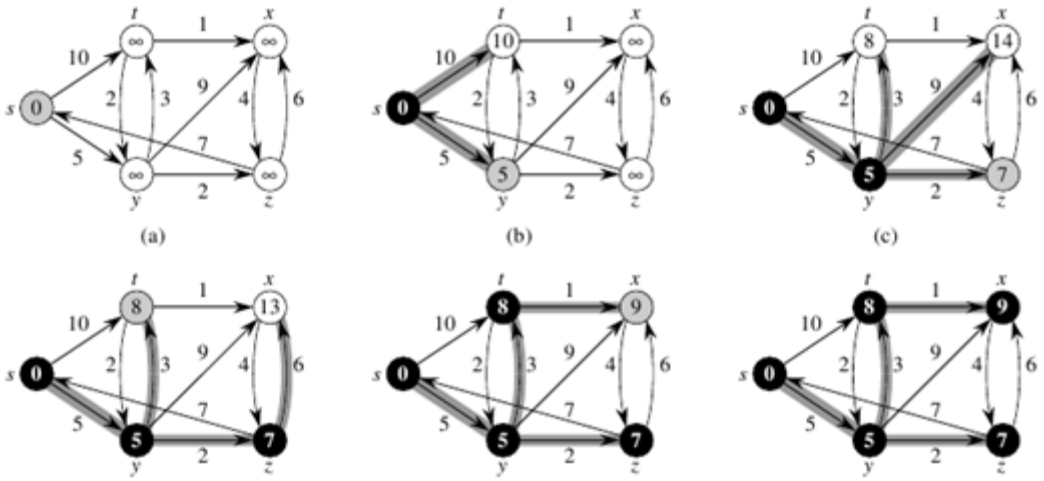



**Fig. 3.4** Ejemplo de resolución de grafo mediante el algoritmo de Dijkstra [15]

### 3.2.2. Algoritmo de Kruskal

Debe su nombre a Joseph Kruskal, que lo publicó por primera vez en 1956. Este algoritmo se encarga de extraer un árbol que conecte todos los nodos del mismo por sus aristas de menor peso cuando se le proporciona un grafo conectado y ponderado (aristas con pesos). Es decir, nos proporciona el árbol recubridor minimal

Como se ha comentado hace un momento, es requisito que el grafo sea ponderado y conectado. Es decir, que las aristas tengan un peso y no exista ningún nodo inaccesible.

El primer paso del algoritmo consiste en ordenar las aristas de menor a mayor peso. Una vez ordenas se van cogiendo las aristas y cada vez que se añade

una se comprueba que no forma un ciclo. Si forma un ciclo se desecha. Una vez que se ha evaluado todo el grafo, se obtendrá el árbol de expansión mínima.

En la **Fig. 3.5** se puede contemplar la aplicación gráfica del algoritmo de Kruskal. Primero se toma la arista h-g, como no forma ciclo con nadie se toma la siguiente, i-c, como sigue sin formar ciclo también se agrega al árbol. El proceso se repite hasta llegar al paso entre la quinta y sexta imagen. La arista i-g tiene menor peso que h-i y c-d, que tienen el mismo peso. La primera de ellas se descarta porque formaría un ciclo. Se analizan h-i y c-d, pero con la primera se produce un ciclo, y por ello se escoge c-d. Finalmente cuando el algoritmo termina se tiene el árbol de la última imagen, el cual garantiza poder llegar desde un nodo cualquiera a otro con el coste mínimo.

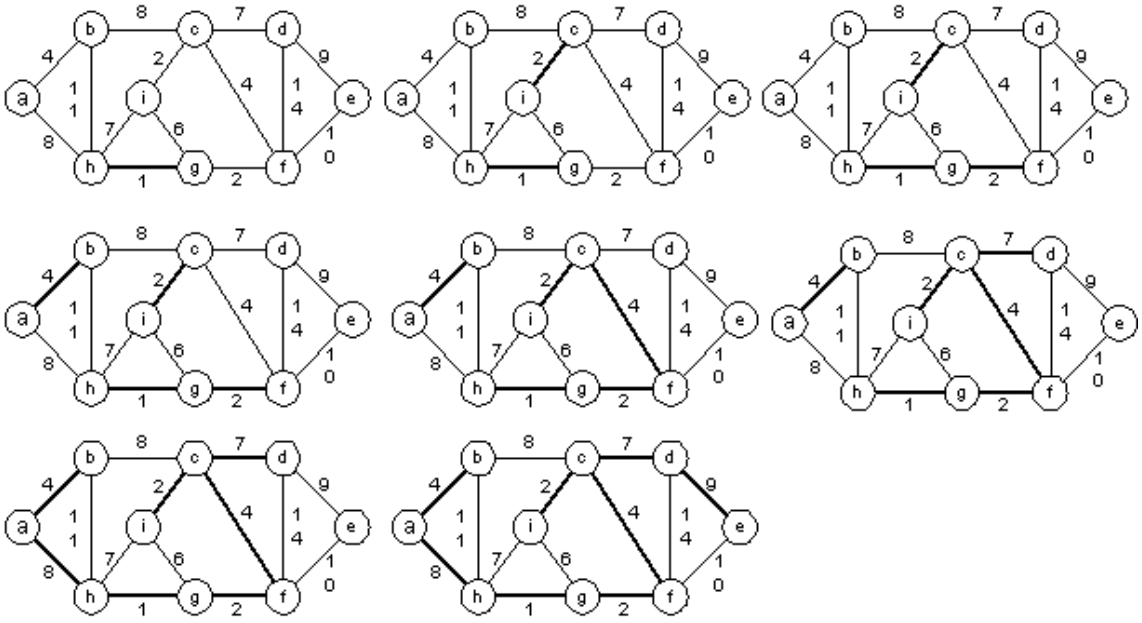


**Fig. 3.5** Ejemplo de resolución de grafo mediante el algoritmo de Kruskal [16]

## 3.3. Algoritmo propuesto

En los algoritmos anteriormente mencionados partimos de que el grafo ya está construido. En nuestro caso no se dispone de un grafo sobre el cual empezar a actuar. Por este motivo el primer paso de nuestro algoritmo es construir el grafo con todos los datos que el usuario especifique.

### 3.3.1. Filtro de datos

Para extraer la información que necesitamos de la base de datos se construyen tres filtros de datos. El primero de ellos se aplica en las dos consultas que se realizan, mientras que los filtros restantes se utilizan para buscar los puntos de interés a visitar y las vías por las que transitar.

El primer filtro nos proporciona lo que denominamos la *zona de exploración*. Se trata de un cuadrado que limita el área que vamos a cubrir. Dicho cuadrado está centrado en el punto medio entre origen y destino (que coincide con origen y destino si ambos son el mismo punto), y la longitud de los lados se calcula como la distancia aproximada que desea recorrer el usuario más un diez por ciento de margen. Esto nos da unos valores mínimos y máximos de latitud y longitud que se utilizan cómo intervalos de la consulta.

De este modo se agiliza la consulta y los cálculos posteriores, ya que toda la información que se descarga consiste en un identificador asociado a unas coordenadas de latitud y longitud.

Destacar que no se permite que esta zona de exploración, que puede ser estipulada por el usuario, pueda ser inferior a la distancia entre los puntos de origen y destino. De igual modo, la distancia por defecto de la zona de exploración es de diez kilómetros.

Después tenemos el filtro para los diferentes tipos de rutas. OpenStreetMap dispone de una serie de etiquetas con sus posibles valores, con esta información podemos saber lo que se desea buscar. Este filtro lo que hace es pasarle a la consulta las etiquetas correspondientes a los tipos de puntos de interés que queremos descargar.

Finalmente tenemos el filtro encargado de determinar el tipo de caminos por los que ir. Igual que en el caso anterior, los caminos están etiquetados con un valor que indica del tipo de vía. Este filtro indica los tipos de vía que nos interesan.

De modo que como resultado de las consultas a la base de datos se obtienen dos listas: una con la información de los puntos de interés de tipología indicada por el usuario que quedan dentro de la zona de exploración, y otra con información de los caminos del tipo indicado por el usuario que quedan también dentro de la zona de exploración.

### 3.3.2. Tratamiento de datos

Para poder construir el grafo, antes se han de tratar los datos que obtenemos de OpenStreetMap, ya que los caminos codificados en general no se corresponden directamente con aristas del grafo.

Por ejemplo, en la **Fig. 3.6** se han resaltado dos caminos, la vía 196018481 y la vía 196018458. Según se aprecia en la figura, los caminos se cruzan entre sí

pero no están seccionados. Es decir, no hay una correspondencia directa entre los caminos (dos en este caso) y las aristas del grafo (cuatro). Esto se debe a que es el usuario quien introduce los datos, y éste no tiene por qué hacerlo siguiendo algún criterio. En la **Fig. 3.7** se puede ver como nosotros y otros usuarios hemos introducido datos. Para este caso nuestra tarea será romper cada vía unitaria en dos y así pasaremos de tener dos caminos a cuatro.

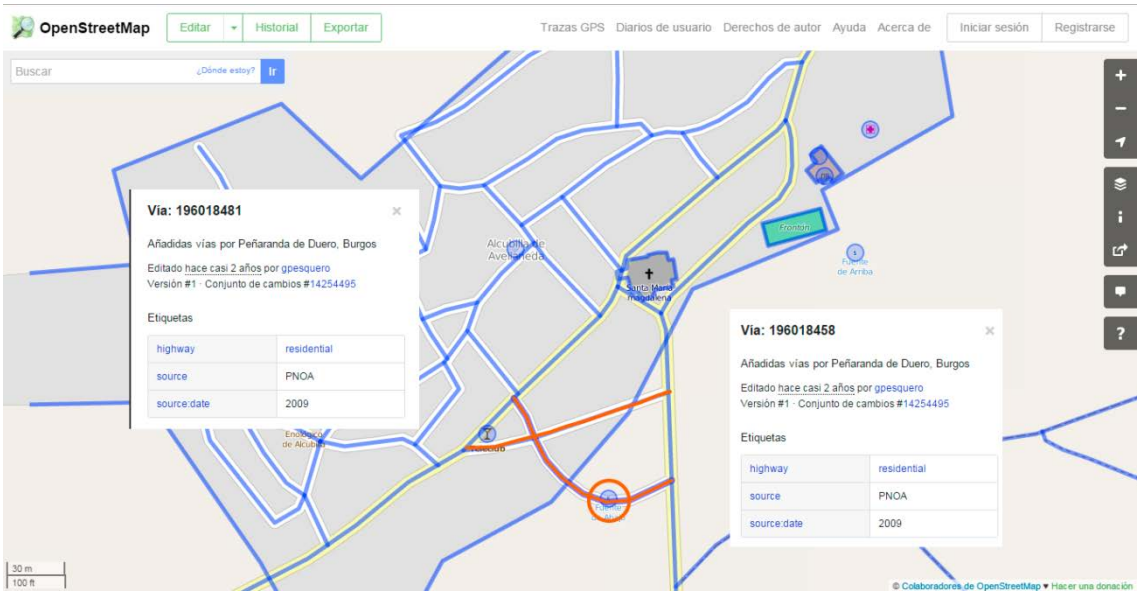



**Fig. 3.6** Ejemplo de cómo están guardados dos caminos en OpenStreetMap

De modo que la tarea general consiste en tomar los caminos existentes y fraccionarlos en tantas partes como sean necesarias para terminar teniendo caminos cuyos extremos coincidan con cruces de caminos.

Pero aún queda un grado más de fragmentación. Por ejemplo, para la vía 196018458 de la **Fig. 3.6** se puede apreciar una circunferencia naranja que corresponde a un posible punto de interés del tipo Fuente. Si no existiese un camino para llegar a dicho punto, el punto de interés sería descartado. Por tanto, cuando se realiza la operación de fragmentación también se tienen en cuenta estos puntos de interés como si fuesen intersecciones. En caso de que el punto de interés quede ligeramente apartado del camino, o no coincida con un punto exacto del camino, se toma el punto del camino más cercano. De manera que, finalmente, las dos vías iniciales quedarían en cinco caminos.

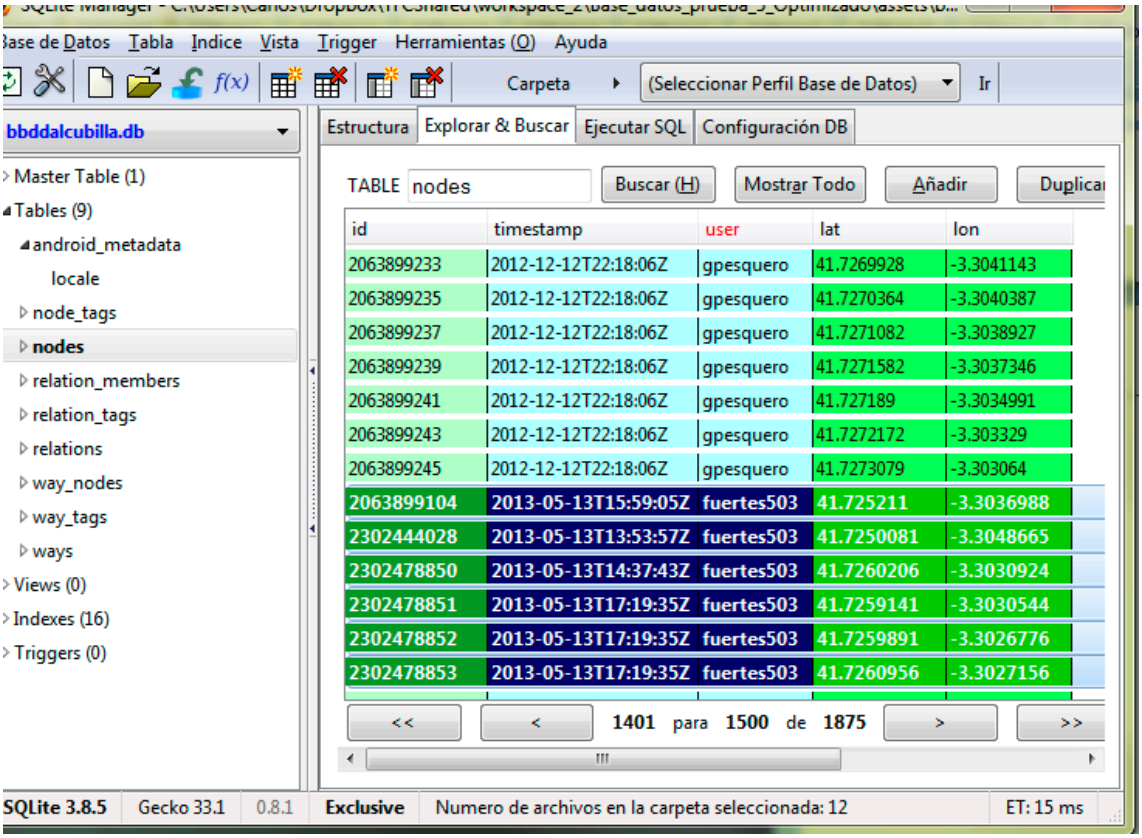



**Fig. 3.7** Base de datos utilizada con mis contribuciones y las de otros usuarios

### 3.3.3. Construcción de los grafos

Con estos datos se tendría un primer grafo donde los nodos son los puntos de cruces de caminos y los puntos de interés (incluyendo origen y destino). Pero lo que se quiere construir es un grafo donde los puntos de interés sean los vértices.

Es en este punto donde entra el algoritmo de Dijkstra. Se comprueba de qué punto de interés a qué punto de interés se puede llegar sin pasar por ninguno intermedio a través del recorrido más corto. La solución estará constituida por los fragmentos que se han creado en el paso anterior.

Una vez que se tiene el conjunto de fragmentos, descartando la coordenada origen del primer fragmento y la coordenada destino del último fragmento, se comprueba si alguna de las coordenadas origen o destino de los diferentes fragmentos que conforman la nueva arista corresponde con la coordenada de alguno de los puntos de interés. Si coincidiese, esto significaría que cuando se ha intentado llegar de A hasta C, se ha pasado por B, ya que la coordenada de B está en alguno de los fragmentos.

### 3.3.4. Ruta resultante

Una vez repetida la operación para todos los puntos de interés, se tendría formado el grafo final sobre el que actuar. Sobre éste se aplicaría Kruskal para sacar el árbol recubridor minimal, el cual nos indicará como llegar a cualquier punto usando siempre el recorrido más rápido.

De modo que al finalizar tendríamos el grafo con los caminos más cortos para llegar a los puntos de interés y el árbol que indica cual es el mejor camino a utilizar. Al usuario se le mostraran los dos, para en caso de no querer volver por el mismo camino pueda escoger la segunda mejor opción.

Recapitulando, hemos utilizado dos algoritmos para la resolución de nuestro problema: por un lado Dijkstra, que nos ha proporcionado el trayecto más corto para llegar de un punto de interés a otro, y Kruskal, que nos ha proporcionado los caminos más óptimos para recorrer.

# CAPITULO 4. Desarrollo de la aplicación

Hasta ahora hemos hablado del contexto y de las herramientas que vamos a necesitar antes de empezar a programar. En este apartado nos centraremos en ver como deseamos que vaya a funcionar el programa para posteriormente determinar qué clases vamos a necesitar y cómo vamos a querer mostrar la información.

## 4.1. Análisis

En la **Fig. 4.1** se presenta el diagrama de flujo deseado para el funcionamiento de la aplicación. Como se puede observar, tras el evento de ingreso en la aplicación se le muestra al usuario una primera interfaz desde la cual podrá continuar hacia delante o por el contrario salir de la misma. Mientras se está en esta ventana se aprovecha para volcar los datos del mapa en un fichero, en caso de que no exista, para que la aplicación pueda trabajar con ellos.

Si se decide continuar, se mostrará una nueva interfaz destinada a que el usuario introduzca los datos con los que desea confeccionar la ruta. Nuevamente el usuario tiene dos opciones, continuar hacia delante calculando el resultado o volver hacía la primera ventana.

Suponiendo que se haya escogido la primera opción, aparecerá un mapa, el cual se carga desde el dispositivo, en el que se visualizará el resultado obtenido de calcular la ruta en función de los datos introducidos por el usuario. En esta ventana, al igual que en las anteriores, se podrá volver atrás o presionando en el botón de menú acceder a una serie de opciones. Éstas serían salir de la aplicación o activar la localización GPS para mostrar la ubicación actual del usuario.

## 4.2. Diseño de clases

Al tratarse de una aplicación para móvil, el diseño de la interfaz de usuario (UI) es un punto muy importante a día de hoy. En los últimos tiempos se ha ido viendo como los propios propietarios de los sistemas operativos móviles han ido haciendo más hincapié a los desarrolladores para que cuiden más el aspecto visual de sus creaciones. Un ejemplo de esto se puede ver con la nueva apuesta de Android por “Material design”.

En este proyecto no nos hemos centrado tanto en mimar todo lo necesario el aspecto gráfico, tarea que dejamos como trabajo futuro, sino en la implementación de los algoritmos.

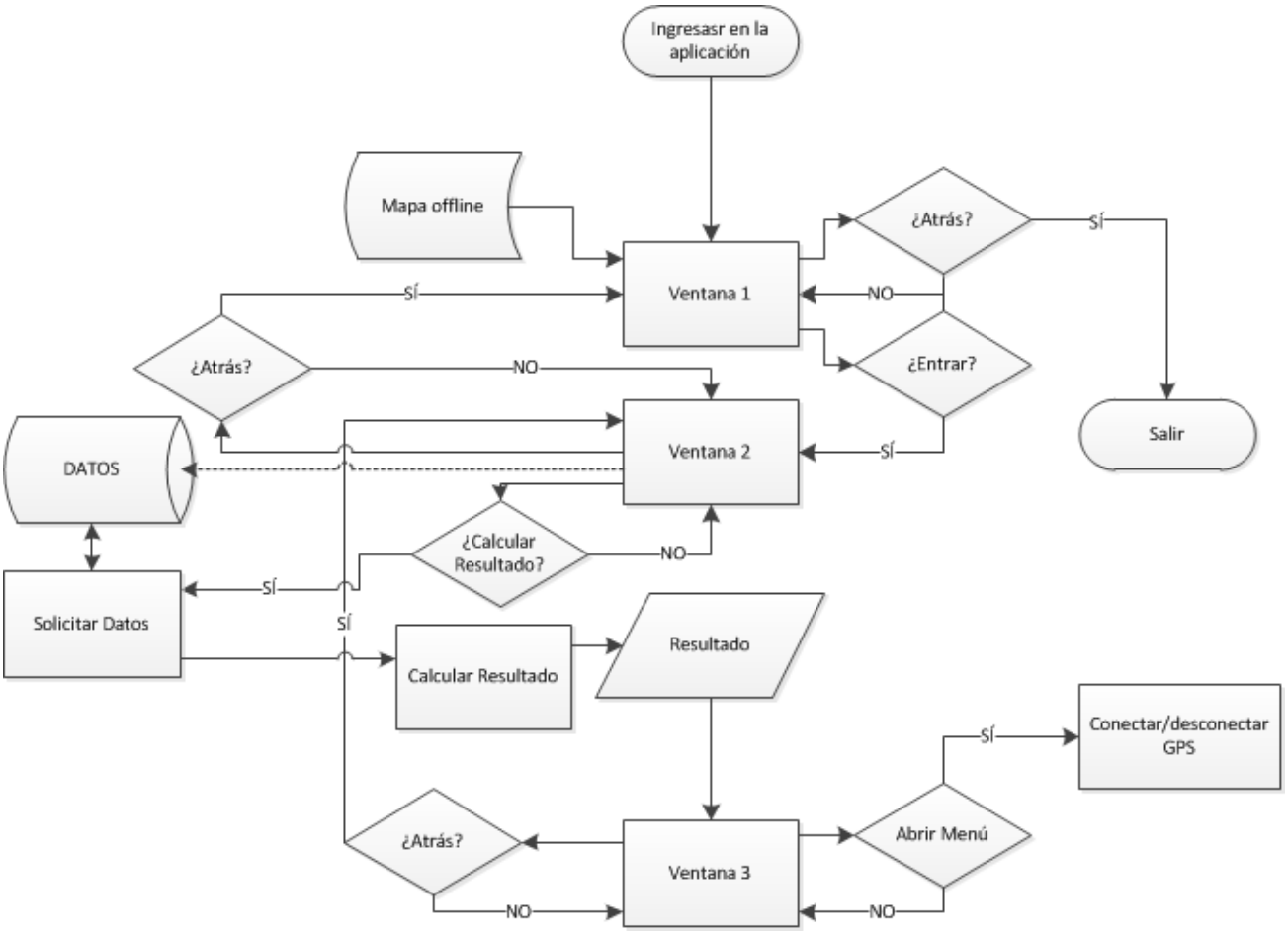



**Fig. 4.1** Flujograma de la aplicación

Se han creado un total de veintiuna clases, las cuales las podemos catalogar en las siguientes seis categorías.

- Funcionalidades UI
- Base de datos
- Estructuras de datos
- Grafos
- Utilidades

A continuación veremos qué clases encontramos dentro de cada categoría y cuál es la función que desempeñan.

### 4.2.1. Funcionalidades UI

Este apartado hace referencia a todas aquellas clases dentro del programa que gestionan las lógicas de los *layouts*, también nombrados en este documento como vistas. Por ejemplo si al pulsar un determinado botón de la pantalla el

color de fondo cambiase, esta lógica se encontraría en las clases que a continuación comentaremos.

**PortadaActivity**

Esta clase se vincula con la primera vista que observaremos al ingresar en la aplicación. Momento en el que se comprueba qué *path* le corresponde y verifica si existe el fichero del mapa, el cual tiene un nombre en concreto. Si no existe, crea un fichero al que volcar los datos binarios del mapa. También es el encargado de lanzar la vista asociada a StartActivity.

**StartActivity**

Conecta con la base de datos, para que llegado el momento se le puedan solicitar los datos necesarios para el cálculo de la ruta a recorrer. Una de las solicitudes que se lleva a cargo al iniciarse esta *Activity* es el nombre de las localidades desde las cuales se puede partir o llegar. El resultado de la misma se muestra en los menús desplegables. Además es el encargado de alojar las lógicas que limitan la consulta tanto en área de trabajo como en tipología de ruta. Finalmente, dispone del método que solicita el resultado y lo envía mediante una clase global a la parte del programa donde gestionar los datos y procesarlos de un determinado modo para mostrarlo al usuario. Este método concluye lanzando la vista para MapaRutasActivity.

**MapaRutasActivity**

Gestiona los datos del mapa de OpenStreetMap y configura la visualización de los datos recibidos mediante la clase global. En su interior podemos encontrar los métodos encargados de la interacción usuario-mapa como serían el GPS o qué se debe hacer cuando se realiza un toque largo sobre alguno de los iconos.

De aquí en adelante la estructura que usaremos para enseñar las diferentes clases que hemos utilizado es la siguiente:

- Nombre de la clase
- Descripción genérica de la misma.
- Estructura de la clase.
- Breve explicación de los métodos.

### 4.2.2. Base de datos

Todas las partes del programa son importantes pues si fallase solo una el resto vería comprometida su efectividad para el desarrollo del proceso. Pero

esto no es del todo cierto ya que si no tuviésemos mapas donde pintar la información ya encontraríamos otra manera de proporcionar dicha información. No obstante si no hay información no hay nada que mostrar por muchas maneras de hacerlo que tengamos. Por ello se puede decir que es el primer gran punto crítico del desarrollo con el que nos encontramos al empezar.

Existen dos alternativas, coger los datos de la nube o tenerlos en nuestro terminal. Revisando las especificaciones iniciales queda patente que la primera alternativa no es una opción, por tanto se trabajará con la información en local y para ello usaremos una base de datos (BD).

Android nos proporciona SQLite para poder realizar nuestras consultas. Dado que nosotros no vamos a crear una BD, sino que trabajaremos con una obtenida de los servidores de OpenStreetMap, hemos de realizar algunas modificaciones para que nuestros terminales puedan trabajar con ellas.

El primer paso será crear una nueva tabla llamada “android_metadata” en la que se le especificará en una columna que llamaremos “locale” el idioma del sistema con el que trabajaremos. En la **Fig. 4.2** se puede observar el resultado

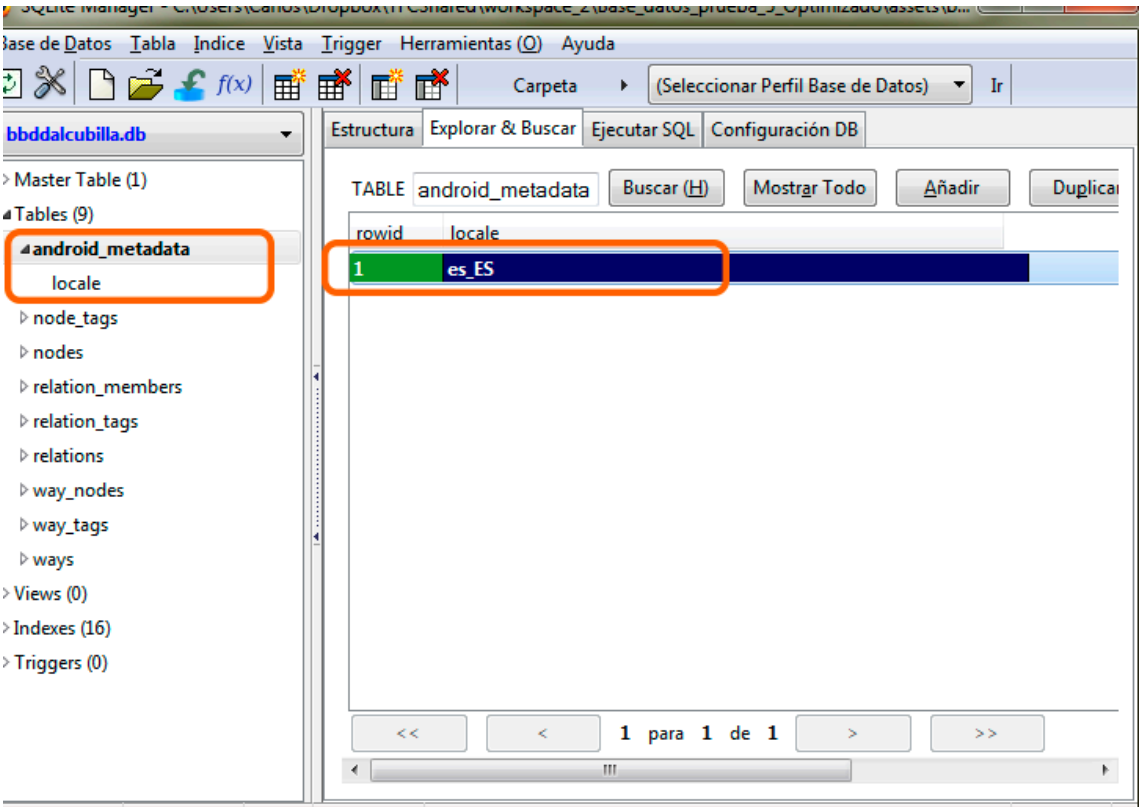



**Fig. 4.2** Base de datos utilizada con la tabla “android_metadata” añadida

Tras hacer esta pequeña modificación añadimos el fichero en la carpeta *Assets* de nuestro proyecto.

Por último solo tendríamos que crear las clases que necesitemos para trabajar con la base de datos. Por lo general, necesitaremos una única clase que extienda de *SQLiteOpenHelper,* la cual será la responsable de alojar los métodos relacionados con la gestión de la BD.

En Android los datos obtenidos de la consulta se nos guardarán en un cursor. Dicho cursor no deja de ser una especie de puntero el cual nos da acceso de lectura y escritura a la base de datos. El orden de los elementos del cursor es el que hayamos seguido a la hora de especificar los elementos que queremos que nos devuelva en el SELECT.

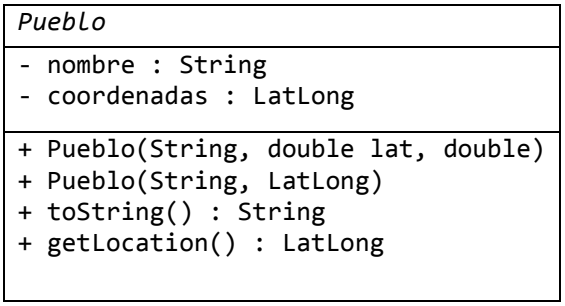
En nuestra aplicación hemos utilizado una clase llamada DBHelper que extenderá de *SQLiteOpenHelper* donde se realizará la conexión y copia de nuestra BD y una clase llamada DBAdapter que extenderá de *Services,* la cual será la encargada de lanzar las consultas y guardar los datos en los objetos que posteriormente explicaremos.

### 4.2.3. Estructuras de Datos

Las siguientes clases contienen las definiciones de los objetos que almacenarán información obtenida de la consulta a la base de datos.

#### Pueblo

Clase que contiene la definición y métodos de consulta para las localidades:




- *toString()*: Devuelve el nombre de la localidad.
- *getLocation()*: Devuelve las coordenadas geográficas de la localidad.

#### InterestPoint

Es la clase que contiene la definición y métodos para aquellos lugares que el usuario puede llegar a visitar.

| *InterestPoint* |
| --- |
| name : String<br>type : String<br>coord : LatLong<br>coordWay : LatLong |
| + InterestPoint(String, String, LatLong)<br>+ InterestPoint(String, String, LatLong, LatLong)<br>+ getName() : String<br>+ getType() : String<br>+ getCoord() : LatLong<br>+ getCoordWay() :LatLong<br>+ setCoordWay(LatLong) : void<br>+ getLat() : double<br>+ getLon() : double<br>+ getValencias() : int<br>+ setValencias(int) : void |

- *getName()* : Devuelve el nombre del punto de interés (IP).
- *getType()*: Devuelve el tipo del IP.
- *getCoord()*: Devuelve las coordenadas exactas del IP.
- *getCoordWay()*: Devuelve las coordenadas del objeto WayPoint más cercano al IP.
- *setCoordWay(LatLong)*: Establece las coordenadas del WayPoint más cercano.
- *getLat()*: Devuelve la coordenada latitud del IP.
- *getLon()*: Devuelve la coordenada longitud del IP.

### Way

Clase para representar los caminos, que consisten en una lista de puntos o WayPoints (ver 4.2.3.4).

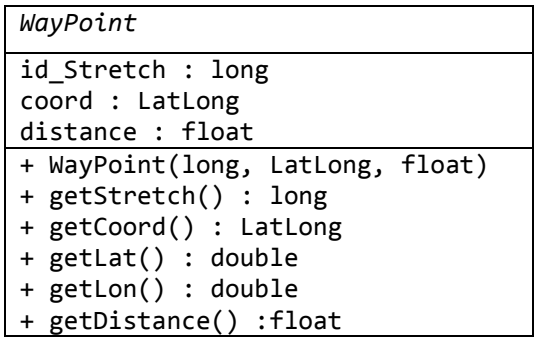
| *Way* |
| --- |
| id_Group : String<br>type : String<br>wayDistance : float<br>listPoints : ArrayList<WayPoint> |
| + Way(String, String, ArrayList<WayPoint>, float)<br>+ getId() : String<br>+ setId(String) : void<br>+ getType() : String<br>+ numNodes() : int<br>+ getPoint(int) : WayPoint<br>+ firstPoint() : WayPoint<br>+ lastPoint() : WayPoitn<br>+ addPoint(WayPoint) : void<br>+ getListPoints() : ArrayList<WayPoint><br>+ setList(ArrayList<WayPoint>) : void<br>+ getWayDistance() : float<br>+ setWayDistance(float) : void<br>+ splitWay(int) : Way |

- *getId()*: Devuelve el identificador del Way.
- *setId(String)*: Asigna un identificador al Way.
- *getType()*:Devuelve el tipo de Way.
- *numNodes()*:Devuelve el número de puntos que conforman el Way.
- *getPoint(int)*: Devuelve el WayPoint que ocupa la posición especificada.
- *firstPoint()*: Devuelve el primer WayPoint de la lista.
- *lastPoint()*: Devuelve el ultimo WayPoint de la lista.
- *addPoint(WayPoint)*: Añade un nuevo WayPoint a la lista.
- *getListPoints()*: Devuelve la lista que contiene los WayPoint.
- *setList(ArrayList<WayPoint>)*: Establece la lista de WayPoint.
- *getWayDistance()*: Devuelve la longitud del Way en metros.
- *setWayDistance(float)*: Establece la longitud del Way en metros.
- *splitWay(int)*: Fragmenta el Way por el WayPoint que ocupa la posición especificada. Devuelve el fragmento de Way substraído.

**WayPoint**

Si inspeccionamos los campos de la clase Way encontramos un ArrayList de tipo WayPoint llamado listPoint del cual no hemos mencionado nada. Para ello necesitamos observar la base de datos. En ella vemos que un way está conformado por varios nodos tal y como podemos ver en la **Fig. 4.3**.

Esta clase es la definición de un tipo de nodos orientados a conformar un camino. Cabría destacar que la distancia que guarda cada nodo hace referencia al espacio que se recorre desde el primer nodo de listPoint hasta sí mismo. De ese modo, el elemento cero de dicha lista tendrá una distancia cero.




- *getStretch()*: Devuelve el identificador de tramo.
- *getCoord()*: Devuelve las coordenadas geográficas.
- *getLat()*: Devuelve la coordenada latitud.
- *getLon()*:Devuelve la coordenada longitud.
- *getDistance()*: Devuelve la distancia del WayPoint respecto al WayPoint inical del Way.

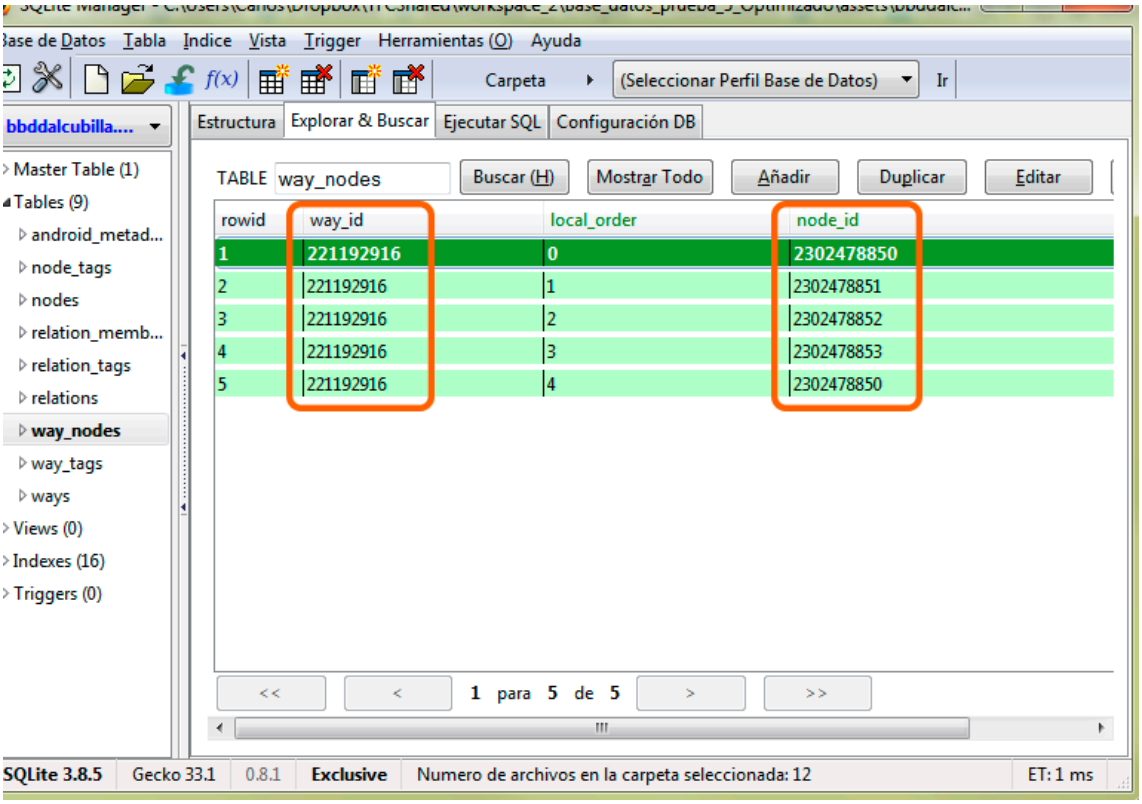



**Fig. 4.3** Base de datos utilizada en la aplicación. Ejemplo de contenido para la tabla way_nodes

### 4.2.4. Grafos

En estas clases se haya la lógica principal de nuestra aplicación.

#### NodeG

Los nodos, a diferencia de las aristas, son iguales para los dos grafos que usamos. Es por esa razón que no tenemos dos tipos diferentes de nodos, como luego veremos que sí le sucede a las aristas. Está clase nodo bien puede ser usada por puntos de interés, pueblos o extremos de vías seccionadas o sin seccionar.

Se tiene que entender como las tiendas de un centro comercial, espacios que pueden ser ocupados por cualquier tipo de tienda, los únicos requisitos son que ese elemento disponga de algo que vender, en nuestro caso eso se traduce a la existencia de coordenadas y por otro lado que se hallen dentro del centro comercial, para nosotros eso significa que dispongamos de una relación de nodos a los cuales llegar. También podría no tenerla, pero entonces no formaría parte del grafo conectado que deseamos hallar; sería lo mismo que tener el centro comercial con todas las tiendas y a un kilometro una zapatería, si no sabemos llegar, no podremos comprarnos unos nuevos zapatos.

| *NodeG* |
| --- |
| nodeName : int<br>coordNode : LatLong<br>listEdges : ArrayList<EdgeG><br>neighbor : int<br>neighborEdge : int<br>bag : float<br>visited : boolean |
| + NodeG(int, LatLong)<br>+ resetParameters() : void<br>+ getNodeName() : int<br>+ setNodeName(int) : void<br>+ getCoordNode() : LatLong<br>+ setEdge(EdgeG) : void<br>+ getNeighbor() : int<br>+ setNeighbor(int) : void<br>+ getNeighborEdge() : int<br>+ setNeighborEdge(int) : void<br>+ getBag() : float<br>+ setBag(float) : void<br>+ isVisited() : boolean<br>+ setVisited(boolean) : void<br>+ getEdges() : ArrayList<EdgeG><br>+ setEdges(ArrayList<EdgeG>) |

- *resetParameters()*: Resetea los campos que participan en el cálculo de Dijkstra.
- *getNodeName()*: Devuelve el nombre del nodo actual.
- *setNodeName(int)*: Asigna un nombre concreto al nodo actual.
- *getCoordNode()*: Devuelve las coordenadas GPS del nodo actual.
- *setEdge(EdgeG)*: Añade una nueva arista a la lista de aristas del nodo.
- *getNeighbor()*: Devuelve la posición que guarda en listNodeG el nodo más próximo a el nodo actual.
- *setNeighbor(int)*: Establece la posición que guarda en listNodeG el nodo más cercano a éste.
- *getNeighborEdge()*: Devuelve la posición que guarda en listNodeG la arista con menos coste por la que viajar para ir del nodo vecino a éste.
- *setNeighborEdge(int)*: Establece la posición que guarda en listNodeG la arista con menos coste por la que viajar para ir del nodo vecino a nosotros mismo.
- *getBag()*:Devuelve el peso acumulado hasta llegar a este nodo.
- *setBag(float)*: Establece el peso acumulado hasta llegar a este nodo.
- *isVisited()*: Indica si el nodo ya ha sido visitado.
- *setVisited(boolean)*: Determina que el nodo ya ha sido visitado.
- *getEdges()*: Devuelve la lista de vecindades.
- *setEdges(ArrayList<EdgeG>)*: Establece toda una lista de vecindades.

### EdgeG

Siguiendo con la alegoría anterior, las aristas vendrían a ser esos trozos de pavimento que nos permiten llegar hasta una determinada tienda. Lo que se requiere conocer son los extremos de las aristas y la distancia que se va recorrer hasta llegar al otro lado.

| *EdgeG* |
| --- |
| nameEdge : int<br>origen : int<br>destino : int<br>peso : float |
| + EdgeG(int, int, int, float)<br>+ getNameEdge() : int<br>+ setNameEdge(int) : void<br>+ getOrigen() : int<br>+ setOrigen(int) : void<br>+ getDestino() : itn<br>+ setDestino(int) : void<br>+ getPeso() : float<br>+ setPeso(float) : void |

- getNameEdge(): Devuelve el nombre de la arista.
- setNameEdge(int): Establece el nombre de la arista.
- getOrigen(): Devuelve el extremo origen de la arista.
- setOrigen(int): Establece el extremo origen de la arista.
- getDestino(): Devuelve el extremo destino de la arista.
- setDestino(int): Establece el extremo destino de la arista.
- getPeso(): Devuelve el valor de la arista.
- setPeso(float): Establece el valor de la arista.

### EdgeK

Si imaginamos que dos personas van al centro comercial, la primera desea recorrer una serie de tiendas mientras que la segunda solo desea visitar una de cada tres tiendas que visita la primera persona, tendríamos que la persona uno tiene tres aristas del tipo EdgeG mientras que la segunda dispone de una única EdgeK constituida por las tres EdgeG de la primera. En nuestro caso esto nos resulta de utilidad para guardar el resultado de aplicar Dijkstra para hallar la ruta corta que nos lleva de un IP a otro.

| *EdgeK* |
| --- |
| origen : int<br>destino : int<br>camino : ArrayList<Way><br>length : float |

```
+ EdgeK()
+ size() : int
+ length() : float
+ add(Way) : void
+ get(int) : Way
+ getOrigen() : int
+ getDestino() : int
+ setOrDes(int, int) : void
```

- size(): Devuelve el número de elementos que contiene camino.
- length(): Devuelve los metros que tiene la arista.
- add(Way): Añade un Way a camino.
- get(int): Devuelve un way determinado alojado en camino.
- getOrigen(): Devuelve el extremo origen.
- getDestino(): Devuelve el extremo destino.
- setOrDes(int,int): Establece origen y destino de la arista.

**GraphD**

Clase que contiene el grafo general con todos los posibles caminos, extremos y sitios a visitar. Contiene los métodos para aplicar el algoritmo de Dijkstra.

| *GraphD* |
| --- |
| listNodesG : ArrayList<NodeG><br>listEdgeG : ArrayList<EdgeG><br>listWays : ArrayList<Way><br>nameNode : int<br>nameEdge : int |
| + GraphD(ArrayList<Way>)<br>+ shortestWay(LatLong, LatLong) : EdgeK<br>- relaxation(NodeG, NodeG, EdgeG, PriorityQueue<NodeG>) : void<br>- getNeighborPosNode(LatLong) : int |

- shortestWay(): Devuelve la ruta más corta para llegar de una coordenada a otra.
- relaxation(): Determina qué nodos forman parte de la solución o no.
- getNeighborPosNode(LatLong): Devuelve el elemento más cercano a las coordenadas especificadas.

**GraphK**

Clase que contiene el grafo con todos los posibles caminos a recorrer con todos los IP que se podrán visitar. Contiene los métodos para aplicar el algoritmo de Kruskal.

| *GraphK* |
| --- |
| - grafoDijkstra : GraphD<br>- listInterestPoints : ArrayList<InterestPoint><br>- listEdgeK : ArrayList<EdgeK><br>+ MST : ArrayList<EdgeK><br>- MAX : int<br>- padre : int[]<br>- rango : int[]<br>- valencia : int[] |
| + GraphK(ArrayList<InterestPoint>, ArrayList<Way>)<br>- fixEdgK() : void<br>- makeSet(int) : void<br>- find(int) : int<br>unionbyRank(int, int) : void<br>- sameComponent(int, int) : boolean<br>- kruskal() : ArrayList<EdgeK><br>+ getMst() : ArrayList<EdgeK><br>+ getPreMst() : ArrayList<EdgeK> |

- getMst(): Devuelve la ruta optima a recorrer.
- getPreMst(): Devuelve GraphK.

### Route

Clase encargada de obtener la ruta con todas las especificaciones del usuario.

| *Route* |
| --- |
| origen : LatLong<br>destino : LatLong<br>listWays : ArrayList<Way><br>listInterestPoints : ArrayList<InterestPoint> |
| + Route(ArrayList<Way>, ArrayList<InterestPoint>, LatLong, LatLong)<br>- ipCoordWay(InterestPoint) : int[]<br>- fixIpWays() : void<br>- fixIdStretch() : void<br>- fixCrossWays() : void<br>+ buildSolution() : ArrayList<ArrayList<EdgeK>> |

- buildSolution() : Método encargado de obtener la ruta a recorrer por el usuario.
- fixIpWays(): Método encargado de seccionar un camino en dos siendo el punto de corte las coordenadas más próximas a un punto de interés.
- fixIdStretch(): Método encargado de resetear el id de tramo para todos los puntos del way.
- fixCrossWays(): Método encargado se seccionar los caminos cuando estos cruzan con otros.

### 4.2.5. Utilidades

En este apartado se han agrupo el resto de clases que dan soporte a la aplicación

#### GlobalVariable

Clase que sirve de puente para pasar información entre StartActivity y MapaRutasActivity.

| *GlobalVariable* |
| --- |
| - rutaOD : ArrayList<ArrayList<EdgeK>><br>- listIP : ArrayList<InterestPoint><br>- listWays : ArrayList<Way><br>- O : LatLong<br>- F : LatLong |
| + setGraph(ArrayList<ArrayList<EdgeK>>) : void<br>+ getGraph() : ArrayList<ArrayList<EdgeK>><br>+ setListInterestPoints(ArrayList<InterestPoint>) : void<br>+ getListInterestPoints() : ArrayList<InterestPoint><br>+ setListWays(ArrayList<Way>) : void<br>+ getListWays() : ArrayList<Way><br>+ setCoordIF(LatLong, LatLong) :void<br>+ getCoordIF() : LatLong[] |

- setGraph(ArrayList<ArrayList<EdgeK>>): Guarda la ruta resultante.
- getGraph(): Devuelve ruta resultante.
- setListInterestPoints(ArrayList<InterestPoint>): Guarda la lista de IP en listIP.
- getListInterestPoints(): Devuelve listIP.
- setListWays(ArrayList<Way>): Guarda la lista de caminos en listWays.
- getListWays(): Devuelve listWays.
- setCoordIF(LatLong, LatLong): Guarda las coordenadas de origen y destino.
- getCoordIF(): Devuelve las coordenadas de origen y destino.

#### Utils

Clase encargada de dar más funcionalidades a los iconos que se van a pintar sobre el mapa.

| *Utils* |
| --- |
| |
| - Utils()<br>+ enableHome(Activity) : void<br>+ setBackground(View, Drawable) : void<br>createMarker(Context, int, LatLong) : Marker<br>createPaint(int, int, Style) : Paint<br>createTappableMarker(Context, int, String, float, float, LatLong) : Marker<br>createTileRendererLayer(TileCache, MapViewPosition, File, XmlRenderTheme, boolean) : TileRenderLayer<br>viewToBitmap(Context, View) : Bitmap |

- createPaint(int, int, style): Método encargado de dar color y estilos. Utilizado para pintar los caminos.
- createMarker(Context, int, LatLong): Método encargado de crear un simple icono para los mapas.
- createTappableMarker(Context, int, String, float, float, LatLong): Método encargado de crear un icono, para el mapa, con información que será lanzada cuando el usuario mantenga una pulsación larga sobre el. Dicho método gestiona el evento.
- createTileRendererLayer(TileCache, MapViewPosition, File, XmlRenderTheme, boolean): Método encargado de generar y gestionar las capas sobre las que se pinta el mapa y sus iconos.

## 4.3. Implementación

Al inicio de este capítulo se mostraba el flujograma general de la aplicación, para que se tuviese una idea de que estaba haciendo el programa. Ahora una vez que ya se han visto las diferentes clases que hay en el programa, se procede a enseñar a través de la **Fig. 4.4** en que parte encaja cada clase dentro del esquema de la **Fig. 4.1**.

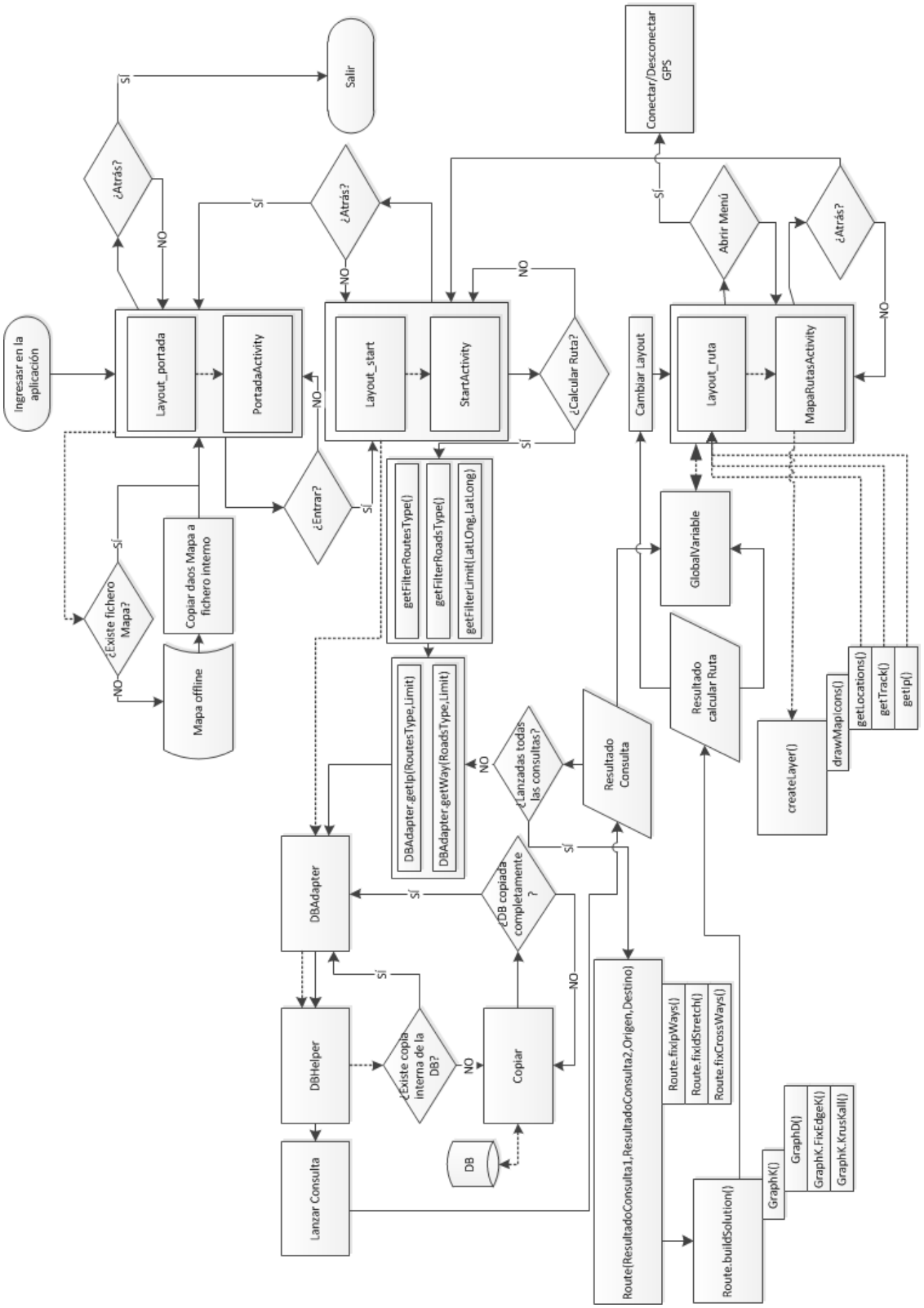



**Fig. 4.4** Flujograma específico de la aplicación

Lo primero que cabe destacar es que cada vista se constituye por un objeto tipo *Layout* el cual contiene todo el aspecto visual y una clase que hereda de *Activity* que contiene todas las lógicas a ejecutar cuando el usuario interaccione con los elementos del *layout* (**Fig. 4.5**).

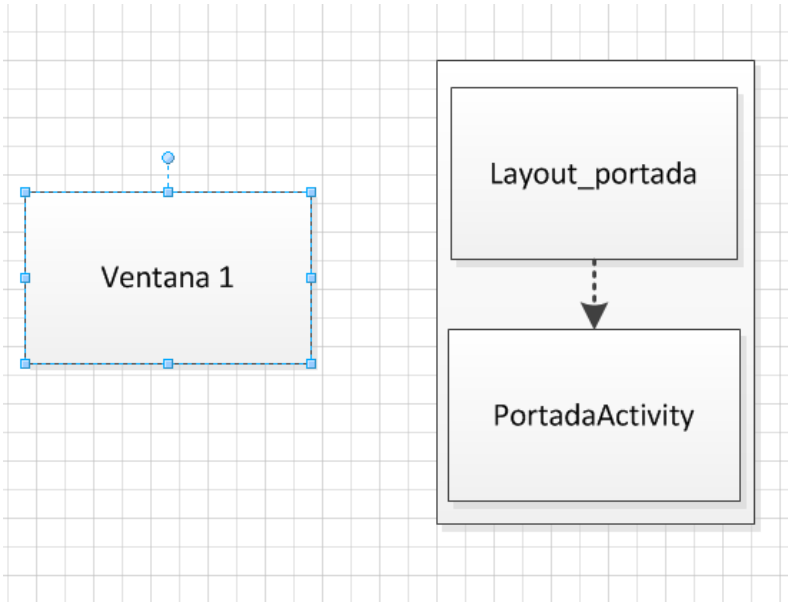



**Fig. 4.5** Modelo general (izquierda) y detallado (derecha) de una vista

Al igual que en el esquema original se han implementado tres vistas. En la primera vista y de modo automático al iniciar la aplicación, se copian los datos del fichero que contiene la información del mapa a otro fichero interno para uso exclusivo de la aplicación, siempre y cuando no existiese con anterioridad, tal y como se muestra en la **Fig. 4.6**. Paralelamente a esto el programa se queda esperando a que el usuario decida seguir avanzando o por el contrario salirse de la misma.

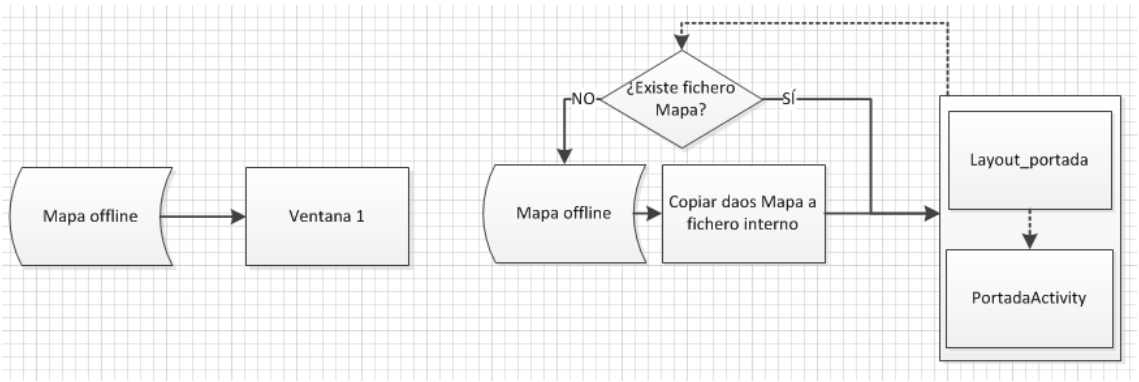



**Fig. 4.6** Modelo general (izquierda) y definitivo (derecha) de la vista Portada

Ya en la segunda vista se puede volver a ver que de modo automático se comprueba si se ha copiado la base de datos en un fichero interno, para uso de la aplicación, o todavía lo tiene pendiente en cuyo caso realizará dicha tarea (**Fig. 4.7**).

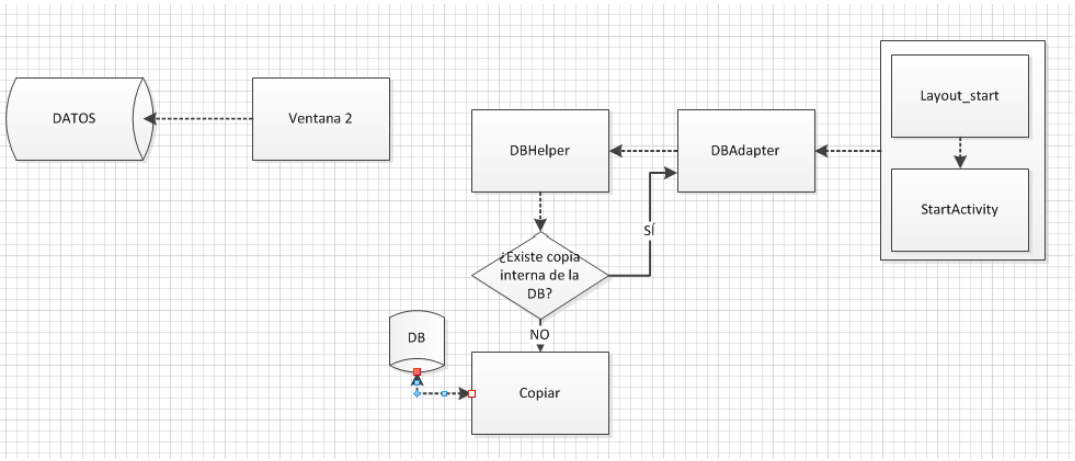



**Fig. 4.7** Modelo general (izquierda) y definitivo (derecha) de la copia de la base de datos en la vista StartActivity

La vista se quedará esperando a que o bien marquemos una serie de parámetros para calcular la ruta o volvamos a la vista anterior. En el caso de pulsar el botón para calcular la ruta, StartActivity nos proporcionará los filtros necesarios para pasárselos a DBAdapter y obtener tanto los puntos de interés como los caminos que sea amoldan a esas condiciones y guardarlos en una variable Global (**Fig. 4.8**).

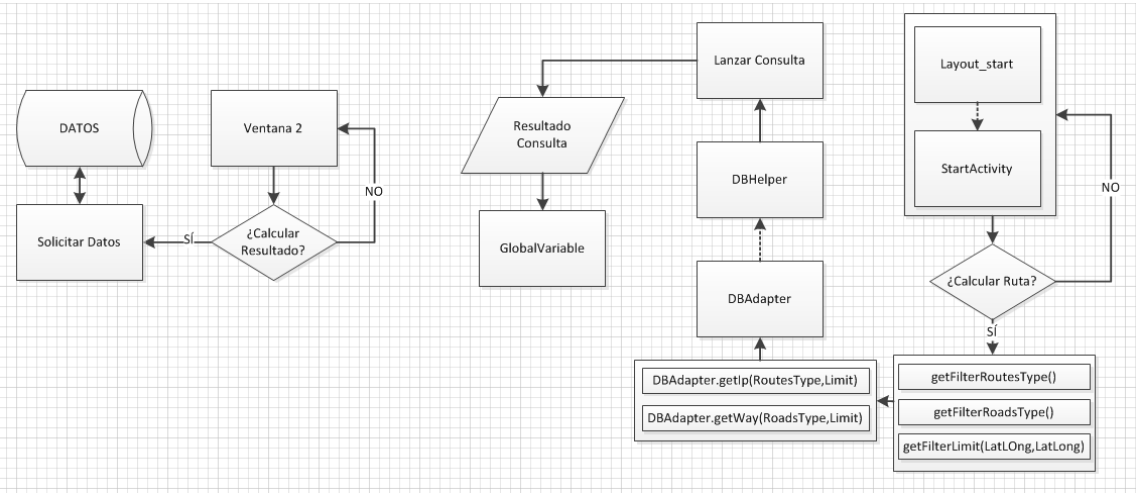



**Fig. 4.8** Modelo general (izquierda) y definitivo (derecha) de la realización de la consulta a la base de datos en la vista StartActivity

Con esta información el programa pasa la información a la clase Route, que se encargará de arreglar los caminos tal y como explicábamos en la sección 3.4. Una vez listos los caminos se invoca el método “buildSoltion” de la clase Route para que realice el algoritmo que hemos detallado en el capítulo 3. Operación tras la cual se guarda el resultado en la variable global y se procede a solicitar una nueva vista (**Fig. 4.9**).

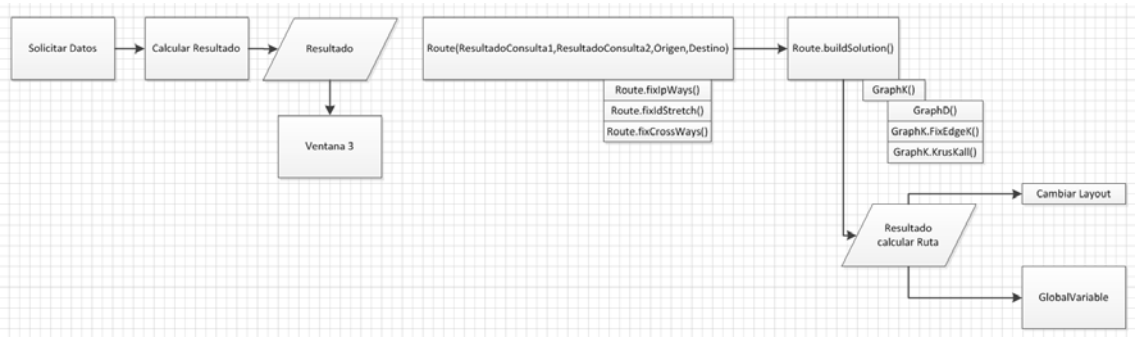



**Fig. 4.9** Modelo general (izquierda) y definitivo (derecha) del cálculo de la ruta en la vista StartActivity

Ya con todos los datos, al iniciar la ventana 3, el programa le solicita a la variable global una serie de datos necesarios para poder pintar la ruta sobre el mapa. Del mismo modo se generan los elementos necesarios para que el usuario puede ver un mapa con iconos que indiquen el tipo de punto de interés, con los cuales se puede interactuar, y la ruta a seguir (Fig. 4.10).

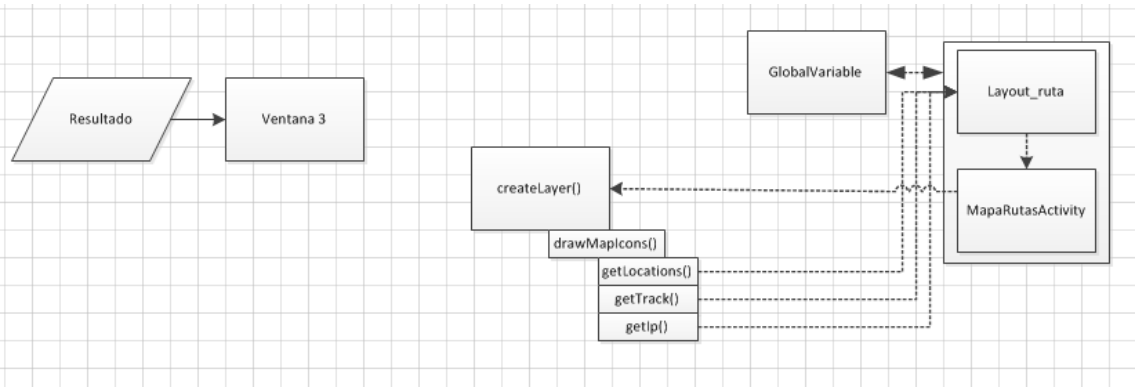



**Fig. 4.10** Modelo general (izquierda) y definitivo (derecha) de la visualización de la ruta en la vista MapaRutasActivity

En cuanto a la navegación dentro de la aplicación, queda del siguiente modo. Mediante el botón de retroceso se puede acceder a la vista anterior. Pero una vez que se llega a la vista inicial, entonces se saldría de la aplicación. Como añadido se tiene que desde la vista tres se puede activar el GPS para guiar al usuario sobre el mapa.

# CAPITULO 5. Resultados

A continuación vamos a exponer varias figuras donde se mostrará tanto el flujo dentro de la aplicación como el resultado que se obtiene de configurar diferentes opciones.

En el capítulo anterior hemos hablado de las clases implicadas en el proceso de la aplicación y se ha mostrado cual sería el flujo que sigue internamente, pero en ningún momento se ha hecho demasiada referencia a las vistas y las diferentes opciones que tiene el usuario para interactuar con la aplicación. En este apartado vamos aprovechar, para además de mostrar los resultados obtenidos, enseñar cuales son las posibles configuraciones.

## 5.1. Iniciar la aplicación

Al ingresar en la aplicación nos encontramos con la imagen de la **Fig. 5.1**, momento en el cual la aplicación aprovecha para gestionar los archivos de mapas. Una vez el usuario clica sobre el botón “Start” se abrirá la ventana de configuración (**Fig. 5.2**).


**Fig. 5.1** Pantalla inicial de la aplicación

## 5.2. Configurar la ruta

En la **Fig. 5.2** se muestra una captura de la pantalla de configuración. En la parte superior, el usuario puede seleccionar el tipo de camino por el cual realizar la ruta. Por tratarse de una aplicación orientada a entornos rurales, se ha considerado tratar por separado las carreteras a los caminos. No obstante siempre se pueden escoger una mezcla de ambas. Ejemplos de los diferentes resultados se pueden apreciar en la **Fig. 5.3** y **Fig. 5.4.**

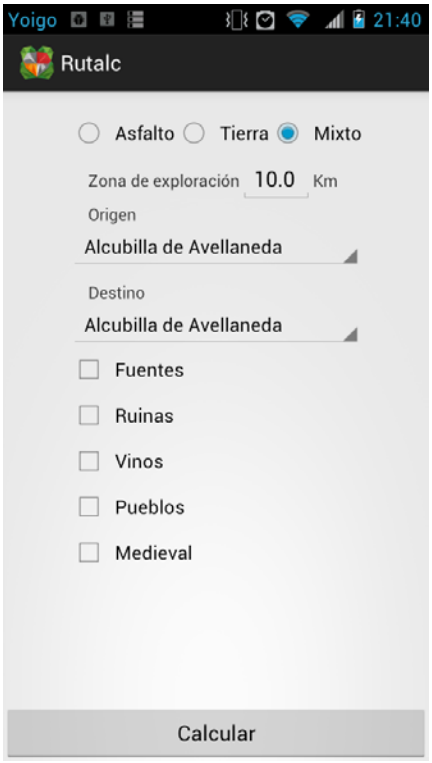



**Fig. 5.2** Pantalla de configuración de ruta

En el centro de la pantalla hay dos desplegables para seleccionar los puntos de origen y destino respectivamente. Estos desplegables se cargan con la lista de poblaciones extraída de la base de datos. Hay también un campo para introducir un valor numérico que corresponde al diámetro de la zona de exploración. Por defecto este valor se configura a diez kilómetros. Si el usuario introduce un valor inferior a la distancia media entre origen y destino, el programa descarta la introducida por el usuario y se queda con dicho valor. Internamente el programa añade un diez por ciento de margen al calcular la zona de exploración, tal y como se ha explicado en el apartado 3.3.1. En el apartado 5.3 se pueden ver ejemplos de diferentes resultados según la zona de exploración.

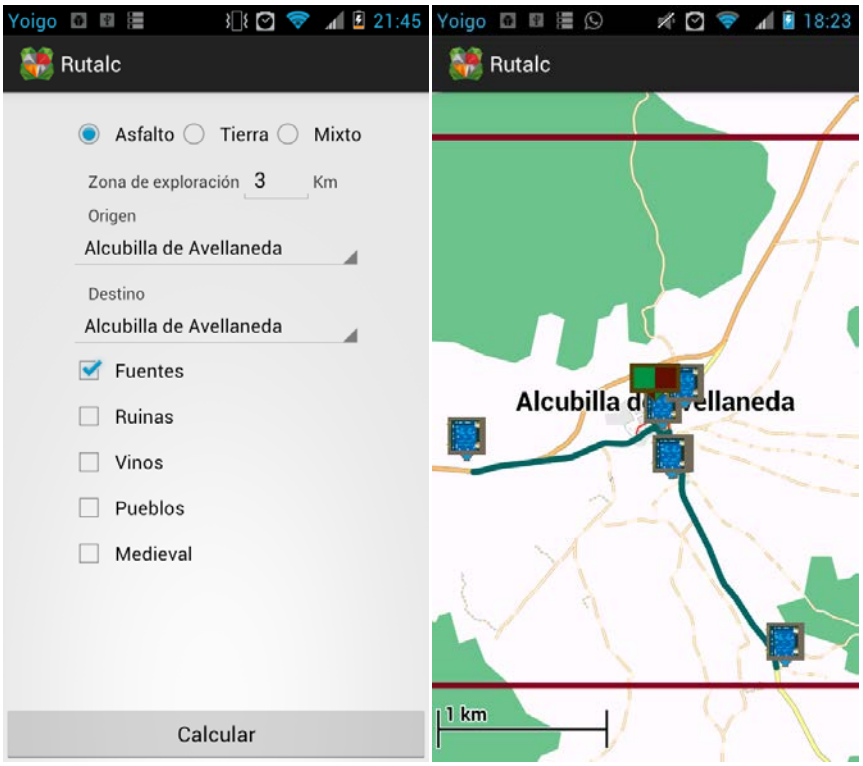



**Fig. 5.3** Ruta por caminos de asfalto

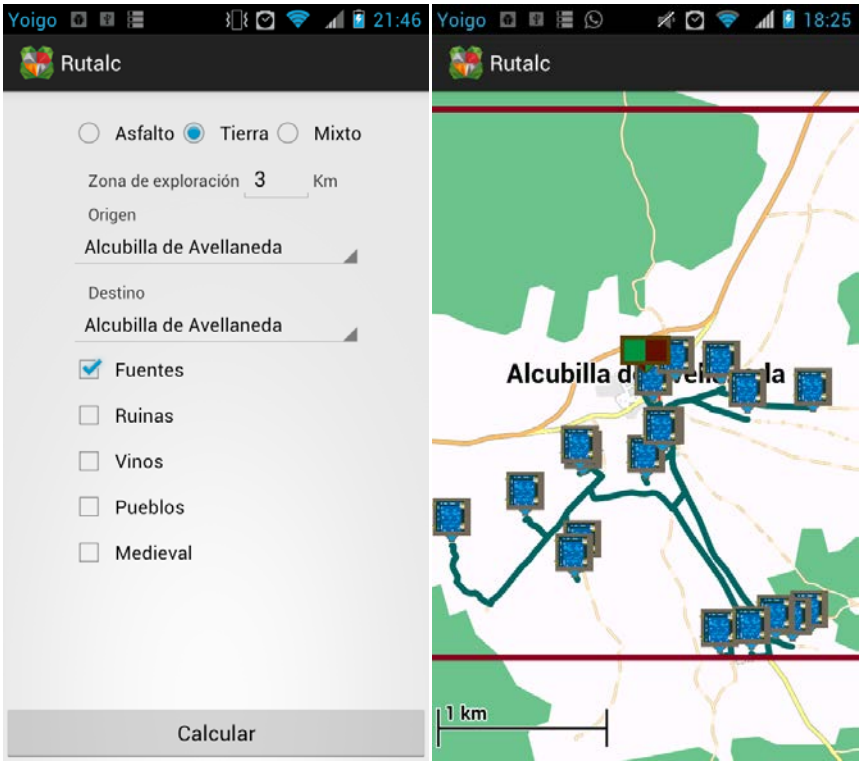



**Fig. 5.4** Ruta por caminos rurales

Por último, se ofrecen al usuario cinco posibles tipos de ruta entre las que escoger: fuentes, ruinas, vinos, pueblos y medieval. El usuario es libre de seleccionar una, varias o ninguna.

Una vez configurada la ruta, para ver el resultado, el usuario deberá pulsar el botón “Calcular” que se encuentra en la parte inferior de la pantalla.

## 5.3. Visualizar el resultado

En la pantalla de visualización, el recorrido que se puede seguir se representa en dos colores, verde para indicar aquella ruta más óptima y en rojo aquellos caminos que no son los más óptimos pero serían los segundos más óptimos y le da al usuario la libertad de decidir volver sobre sus propios pasos o escoger una variante (**Fig. 5.5**).


**Fig. 5.5** Resultado con tipo de ruta pueblos, vino y medieval. Visualización de los caminos más óptimos y caminos alternativos.

Ya dentro del mapa, a través del botón menú se puede activar o desactivar la funcionalidad de escucha de servicios de posicionamiento para capturar nuestro progreso a la hora de recorrer la ruta. Añadida a esta pequeña ayuda, también se puede mantener pulsado sobre alguno de los diferentes iconos para ver cuál es la distancia hasta llegar a destino o para regresar al punto de partida (**Fig. 5.6**).

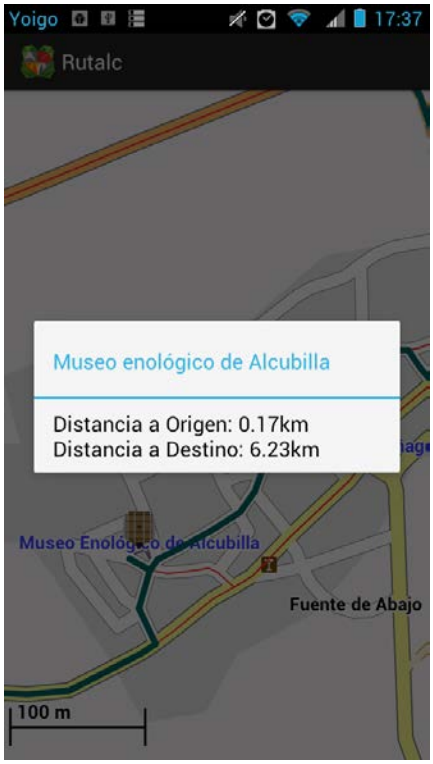



**Fig. 5.6** Ventana emergente que informa de la distancia entre el punto de interés y los puntos de origen y destino

En las siguientes figuras (**Fig. 5.7** y **Fig. 5.8**) se muestran diferentes configuraciones de origen/destino y como el programa gestiona la zona de exploración.

En el primer caso se deja que el sistema configure la zona de exploración mínima para realizar una ruta del tipo pueblo por caminos mixtos. Mientras que en el segundo caso se aumenta la zona de exploración a quince kilómetros.


**Fig. 5.7** Zona de exploración mínima con caminos mixtos y tipo turismo pueblos

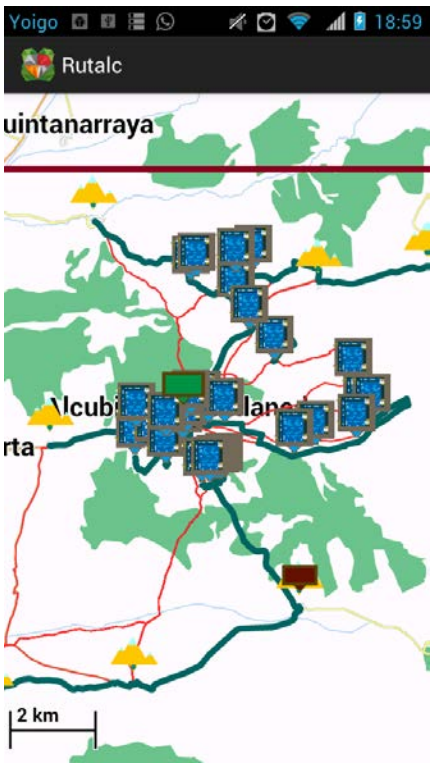



**Fig. 5.8** Zona de exploración de 15km con caminos mixtos y tipo turismo pueblos y fuentes

# CAPITULO 6. Conclusiones y trabajos futuros

El objetivo inicial de este proyecto ha consistido en crear una herramienta de exploración de entornos rurales, accesible para el mayor número de personas y que fuese móvil. Como requisito adicional, tenía que ser lo suficientemente versátil como para que no se viese limitada a la disponibilidad de la señal móvil.

Como resultado de ello lo que se ha obtenido ha sido una aplicación para dispositivos móviles basados en el sistema operativo Android capaz de operar sin necesidad de cobertura móvil, cumpliendo de este modo las especificaciones iniciales.

Tras este desarrollo inicial con Android hemos podido constatar la complejidad del sistema a la hora de atacarlo desde cero, sin ningún tipo de conocimiento previo y con librerías de código abierto no oficiales para servicios de terceros.

Si bien es cierto que la comunidad que lo respalda es grande, hay diversos campos donde la información de la que se dispone no es todo lo extensa que uno querría.

A pesar de tratarse de un sistema operativo de software libre, hemos podido ver que tener acceso total a los sensores del terminal móvil no quiere decir tener acceso a todas las utilidades que Android trae de serie, desde el punto de vista del consumidor. Como es el caso de los mapas de Google, los cuales no vienen dentro del SDK de desarrollo.

A consecuencia de las políticas asociadas al uso del servicio mencionado anteriormente, se optó por usar los mapas de un tercero el cual nos diese la posibilidad de trabajar con ellos de manera offline y sin ningún tipo de limitación.

Todo lo anterior se traduce en un desarrollo lento y limitado al avance de las librerías extras que estemos usando, motivo por el cual tuvimos que cambiar en dos ocasiones la librería encargada de la gestión de mapas. En un primer caso, la decisión fue debida al bajo rendimiento que ofrecía la librería y por problemas con el manejo de mapas en local. Posteriormente se tuvo que actualizar la librería utilizada porque la versión con la que trabajábamos contenía fallos graves de implementación que afectaban a nuestro proyecto, concretamente para poder renderizar más elementos de indicación sobre los mapas.

Con la realización del proyecto hemos podido ver cuál es el proceso de mapeo de zonas rurales, inclusión de datos en servicios como OpenStreetMap frente a Google Maps y cómo tratarlos para posteriormente sacar la base de datos con el contenido de los mismos y crear un mapa acorde al formato que entiende la librería que utilizamos. Asimismo, hemos visto la manera que tiene Android de trabajar con bases de datos y nos hemos montado nuestro sistema de consultas a una base de datos local.

Además, gracias al desarrollo de este trabajo se ha mejorado la capacidad de y comprensión de documentación específica, como *javadocs*, elemento muy importante para el desarrollo de software.

Aunque el resultado obtenido es satisfactorio, ofrece todavía muchas vías de mejora, tanto a nivel de diseño, como de funcionalidades o de ampliación de datos.

Desde el punto de punto de vista de diseño, tenemos la opción actualizar el contenido hacia *Material Design* y realizar iconos más acordes al tema visual que tenga el programa.

En la actualidad descargamos los mapas, los transformamos y extraemos los datos desde un ordenador, posteriormente modificamos unos parámetros de la base de datos obtenida e introducimos ambos ficheros dentro del paquete de la aplicación. Una mejora interesante consistiría en automatizar estos procesos y que los realizase la propia aplicación, teniendo en cuenta la localización del usuario y si esta es considerablemente diferente de la que disponía en el momento de la descarga anterior. La zona a descargar sería una extensión lo suficientemente grande como para que el usuario no perciba la falta de conexión.

Otra opción interesante, en el campo de los *wearable*s, sería realizar una extensión donde se mostrase una brújula que nos orientase hacia el próximo punto de interés y que nos indicase información sobre el mismo.

Finalmente, en cuanto a los datos, se deberían buscar nuevos tipos de puntos de interés y ampliar la base de datos con más información relevante de la zona. Asimismo, el proyecto resulta claramente extensible a cualquier otra zona de la que se disponga la información.

# Referencias


[1] Smartmo. World Wide Smartphone Sales Share, 2012, http://en.wikipedia.org/wiki/Mobile_operating_system [Consulta: 01 de Octubre de 2014]

[2] Mapa de la cobertura móvil en España, http://www.vodafone.es/conocenos/es/cobertura-y-tiendas/cobertura/consulta-de-cobertura-movil/ [Consulta: 01 de Octubre de 2014]

[3] Mapa de la cobertura móvil en España, http://lared.orange.es/cobertura_movil.html [Consulta: 01 de Octubre de 2014]

[4] Cuota de mercado de los diferentes sistemas operativos para Septiembre de 2012, 2013 y 2014 en España, 2014, http://www.kantarworldpanel.com/global/smartphone-os-market-share/ [Consulta: 23 de Octubre de 2014]

[5] Offline use of google maps, 2013, http://stackoverflow.com/questions/16115391/offline-use-of-google-maps [Consulta: 23 de Octubre de 2014]

[6] Mayorga, Alex. OpenStreetMap, 2013, http://es.wikipedia.org/wiki/OpenStreetMap [Consulta: 23 de Octubre de 2014]

[7] http://wiki.openstreetmap.org/wiki/Osmosis [Consulta: 23 de Octubre de 2014]

[8] Algoritmos de búsqueda, 2013, http://es.wikipedia.org/wiki/Categor%C3%ADa:Algoritmos_de_b%C3%BAsqueda [Consulta: 6 de Noviembre de 2014]

[9] Grafo carreteras de España, http://dis.um.es/~ginesgm/retoviajante.html [Consulta: 6 de Noviembre de 2014]

[10] NSFnet 1989, 2006, http://revista-redes.rediris.es/html-vol11/Vol11_9.htm [Consulta: 6 de Noviembre de 2014]

[11] Metro de Barcelona, http://upload.wikimedia.org/wikipedia/commons/6/6c/Met2.svg [Consulta: 6 de Noviembre de 2014]

[12] Digrafos, http://www.dma.fi.upm.es/gregorio/grafos/IAGraph/images/DFS_fin.png [Consulta: 6 de Noviembre de 2014]

[13] Iniesto Díaz, Abraham y Delgado Núñez, Juan Carlos. Ejemplo de multigrafo,

http://upload.wikimedia.org/wikipedia/commons/c/c9/Multi-pseudograph.svg
[Consulta: 6 de Noviembre de 2014]

[14]González, Ramón. Árbol minimal, 2011, http://esteban-gzz.blogspot.com.es/2011_07_01_archive.htm [Consulta: 6 de Noviembre de 2014]

[15] Iniesto Díaz, Abraham y Delgado Núñez, Juan Carlos. Ejempo resolución Dijkstra, http://www.dma.fi.upm.es/gregorio/grafos/IAGraph/images/dijkstra.png
[Consulta: 13 de Noviembre de 2014]

[16] Hernández Peñalver, Gregorio. Ejemplo resolución Kruskal, http://pioneer.netserv.chula.ac.th/~skrung/2301232/images/kruskal.gif
[Consulta: 13 de Noviembre de 2014]

# BIBLIOGRAFÍA


Groussard, T, *JAVA7 Los fundamentos del lenguaje Java*, Ediciones ENI, Cornellà de Llobregat.

Pérochon, S, Android *Guía de desarrollo de aplicaciones para Smartphones y Tabletas*, Ediciones ENI, Cornellà de Llobregat.

Pergar, JM, MapsForge OpenStreetMap en Android, http://compraventa.ebay.es/

*Árbol de expansión mínima. Algoritmo de Kruskal*, http://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/

*Camino más corto: Algoritmo de Dijkstra*, http://jariasf.wordpress.com/2012/03/19/camino-mas-corto-algoritmo-de-dijkstra/

*Osmosis*, http://wiki.openstreetmap.org/wiki/Osmosis

*Map Features*, http://wiki.openstreetmap.org/wiki/Map_Features

*MapsForge*, https://github.com/mapsforge/mapsforge/

S. Mendoza, M, *Usar nuestra propia Base de Datos SQLite en Android*, http://blog.netrunners.es/usar-nuestra-propia-base-de-datos-sqlite-en-android/

*Análisis del contenido de las base de datos de OpenStreetMap*, http://blog.openalfa.com/analisis-del-contenido-de-la-base-de-datos-de-openstreetmap

Página para desarrolladores Android, http://developer.android.com/guide/index.html