UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

PROYECTO FIN DE CARRERA

SISTEMA BASADO EN LA LOCALIZACIÓN PARA LA AYUDA A PERSONAS CON NECESIDADES ESPECIALES

Ingeniería de Telecomunicación

Lucía Anglés Alcázar
Febrero 2010

SISTEMA BASADO EN LA LOCALIZACIÓN PARA LA AYUDA A PERSONAS CON NECESIDADES ESPECIALES

AUTORA: Lucía Anglés Alcázar
TUTOR: Pablo A. Haya Coll

Grupo de Herramientas Interactivas Avanzadas (GHIA)
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero 2010

Resumen

En este proyecto se presenta WAI-Routes, un sistema que calcula la ruta óptima entre dos puntos del espacio empleando la red de transporte público de la Comunidad de Madrid, teniendo en cuenta el perfil del usuario. WAI-Routes ha sido diseñado teniendo en mente a usuarios con discapacidad mental, de tal manera que las rutas calculadas se ajustan a las particularidades de los miembros de este colectivo. Para ello, consta de una serie de filtros variados que pueden ser prefijados la primera vez que el usuario use la aplicación o cada vez que éste quiera calcular una ruta.

Palabras clave

Aplicaciones Sensibles a la Localización, Sistema de Estimación de Ruta, Discapacidad Cognitiva, Servicio de Transporte Público.

Abstract

This project presents WAI-Routes, a system whose main goal is to calculate the best route between two points in space using the public transport network of the Comunidad de Madrid regional area, taking into account the user's profile. WAI-Routes has been designed bearing in mind mentally disabled users; routes are calculated with the particularities of the members of this group in mind. In order to achieve this, the system contains a variety of filters than can be set the first time the user uses the application or whenever he/she wants to calculate a new route.

Keywords

Location-Aware Applications, Route Estimation System, Cognitive Impairments, Public Transportation Service.

Agradecimientos

En primer lugar quiero agradecerle a mi tutor, Pablo Haya, todo su esfuerzo y dedicación con este proyecto. Siempre ha tenido tiempo para mi, demostrando una paciencia infinita.

No puedo olvidarme de mis compañeros de laboratorio, siempre dispuestos a ayudarme, que han hecho que mi paso por el AmiLab haya sido una experiencia muy grata.

En segundo lugar quiero dar las gracias a mi familia, que me ha apoyado en todo momento. Papis, gracias por esas infusiones relajantes antes de irme a dormir y por eximirme de limpiar mi parte de la casa en época de exámenes. Hermanitos, gracias por dejarme un considerable margen de suspensos (Samu), por explicarme un montón de cosas extrañas (Dani) y por el contrabando de gominolas (Migue).

También quiero hacer una mención especial a mi tía Adela, que no sólo trajo dos botellas de cava a mi casa cuando aprobé la última asignatura de la carrera, si no que además me ayudó en los aspectos topográficos de este proyecto.

En tercer lugar me gustaría nombrar a Euge (mi fiel compañero de aventuras y gran marujeador), por la de prácticas que hemos hecho juntos y por todos los momentos que hemos compartido, tanto dentro como fuera de la universidad. Alicia, gracias por hacer más divertidos estos últimos años de la carrera.

Per l'ultimo sempre si lascia quello che è migliore. Davide, grazie con tutto il cuore per essere stato al mio lato dal momento in cui ti ho conosciuto, appoggiandomi, animandomi e coccolandomi, sempre. Grazie mille per avere sempre un sorriso per me. Senza di te tutto sarebbe molto più difficile.

Lucía Anglés
Febrero 2010

Índice de contenidos

Índice de figuras

Índice de tablas

1 Introducción

1.1. Motivación

Los últimos avances en hardware realizados en el campo de los teléfonos inteligentes o PDA's han permitido que surjan nuevas aplicaciones, así como nuevos servicios, especialmente aquellos relacionados con la localización y el acceso o gestión de la información. De esta forma se pueden crear herramientas que, utilizando las características anteriores, permitan ayudar a la realización de tareas cotidianas por parte de personas que sufren una discapacidad o minusvalía.

El proyecto HADA [1] tiene como objetivo "desarrollar metodologías y herramientas que integren hipermedia adaptativa y entornos de inteligencia ambiental, con el fin de facilitar el uso de nuevas tecnologías a usuarios con necesidades especiales y específicas, concretamente personas con discapacidades cognitivas y personas mayores". Para ello se ofrecen soluciones adaptadas a las necesidades y requisitos de cada usuario en el entorno del hogar, en los medios de trasporte y dentro de oficinas o aulas.

Según la Federación Española de Síndrome de Down [2], "la autonomía y la independencia pasan necesariamente por el acceso a un puesto de trabajo". Para muchas personas con discapacidad el solo hecho de ir y venir del trabajo puede ser un gran problema, lo que supone una gran barrera a la hora de integrarse mejor en entornos laborales. Así mismo, la incapacidad para emplear con seguridad el transporte público perjudica su calidad de vida, lo cual afecta no solo a su desarrollo profesional si no también a su tiempo libre [3].

Es importante destacar que algunas de estas personas no pueden responder rápidamente a los cambios de circunstancias, como por ejemplo cancelación de trenes y autobuses, retrasos y cortes de tráfico… Todas estas circunstancias pueden producir que la persona necesite más información para ser capaz de tomar decisiones apropiadas.

WAI (Where Am I) pertenece al proyecto HADA centrándose en la necesidad de facilitar el uso de los medios de transporte público que ofrece la Comunidad de Madrid en personas con Síndrome de Down, siguiendo un enfoque similar a Patterson et al [4].

El proyecto WAI está dividido en dos partes:

- WAI-Interface: se encarga de gestionar la interacción con el usuario y guiar al mismo mediante una interfaz intuitiva y adaptativa.

- WAI-Routes: se encarga de calcular la ruta según el punto de origen y destino escogido, y el perfil del usuario.

El proyecto aquí presentado desarrolla lo que correspondería al proyecto WAI-Routes.

WAI-Routes se enmarca dentro de las áreas de investigación de computación basada en localización y sensible al contexto. Un área cuyos trabajos se remontan a principios de los años 90 [5] y en la que durante las últimas décadas han surgido múltiples sistemas [6]. Los servicios basados en localización son percibidos como necesarios y son requeridos por los usuarios [7]. De hecho, en los próximos dos años se prevé en Europa un incremento en las inversiones para el desarrollo de este tipo de servicios [8].

Más concretamente, el sistema empleará la posición (suministrada por el GPS o introducida por el usuario) en que se encuentre el usuario, y por medio de una interfaz visual asistirá al usuario indicándole cómo dirigirse a un destino.

1.2. Objetivos

La finalidad del proyecto (WAI-Routes) consiste en desarrollar un sistema informático que permita guiar al usuario de un punto a otro de Madrid mediante la Red de Transporte Público de la Comunidad de Madrid [9]. Los servicios basados en localización que se piensan proveer se encuadran dentro de los servicios basados en posicionamiento [10]. En particular, se emplean las coordenadas espacio-temporales del mismo. Para ello, el usuario dispondrá de un terminal móvil con conexión a Internet y GPS integrado.

La ruta a seguir por el usuario está constituida por una serie de puntos. Cada punto

de la ruta puede ser una coordenada (latitud, longitud), una calle, una estación de metro, una estación de metro ligero, una estación de cercanías o una parada de autobús, siendo el espacio considerado el abarcado por la Comunidad de Madrid.

Una persona que sufre una discapacidad cognitiva tiene mayores dificultades al realizar tareas mentales que otra persona sin discapacidad. Los principales procesos cognitivos afectados por dichas discapacidades son la memoria, la resolución de problemas, la atención, la lectura, lingüística y comprensión verbal, la comprensión matemática y la comprensión visual. Hay que tener en cuenta que existe bastante diversidad en cuanto a los procesos cognitivos afectados en cada persona, y en cuanto al grado de discapacidad. Por todo ello, WAI-Routes necesita tener en cuenta las necesidades específicas del usuario para la creación de la ruta. Algunos tendrán dificultades para hacer transbordos, a otros les resultará complicado coger el cercanías, otros necesitarán que las estaciones estén adaptadas a personas con movilidad reducida, etc. Todo ello se llevará a cabo mediante un mecanismo de filtrado, a través del cual cada usuario podrá especificar sus preferencias.

1.3. Organización de la memoria

La memoria consta de ocho capítulos y cinco anexos, que serán explicados a continuación.

Capítulo 1. Introducción. Se explica la motivación del proyecto, así como los objetivos del mismo. También se detalla cómo se ha estructurado la memoria.

Capítulo 2. Trabajo relacionado. Se comenta la documentación estudiada referente a proyectos o técnicas relacionadas con WAI, enumerando las similitudes y diferencias con este proyecto. También se explican los diferentes algoritmos de búsqueda del camino mínimo estudiados para calcular la ruta.

Capítulo 3. Arquitectura del sistema. Se explica la arquitectura Cliente-Servidor empleada en el sistema desarrollado y el por qué de esa elección. También se presenta la estructura empleada para almacenar la información de la red de transporte público.

Capítulo 4. Servidor personalizado de rutas. Se detalla la estructura del servidor, así

como las funciones que realiza. Se explica tanto su funcionalidad como su implementación.

Capítulo 5. Cliente ligero. Se detalla la estructura del cliente, así como las funciones que realiza. Se explica tanto su funcionalidad como su implementación. También se especifica el funcionamiento y la implementación de la conexión entre el cliente y el servidor, así como su intercambio de datos.

Capítulo 6. Prototipo de demostración. Se explican las propiedades de Android, así como su arquitectura, los componentes de una aplicación y la jerarquía visual. También se explica un ejemplo de interfaz realizado para probar la aplicación, incluida su implementación.

Capítulo 7. Pruebas y resultados. Se expone las pruebas realizadas durante la ejecución del proyecto, así como los resultados obtenidos de dichas pruebas.

Capítulo 8. Conclusiones y trabajo futuro. Se explican las conclusiones obtenidas después de la realización del proyecto, así como las líneas futuras que puede seguir la aplicación desarrollada.

Anexo I. Guía básica de desarrollo: crear una aplicación en Android. Se detalla paso a paso cómo crear una aplicación en Android.

Anexo II. Medidas del GPS del terminal sobre vértices del centro de Madrid. Se adjuntan las hojas de datos de siete vértices de la Red Topográfica de Madrid, así como las correspondientes medidas tomadas por el GPS del terminal. También se describe como se ha realizado la equivalencia entre grados y metros.

Anexo III. Presupuesto. Aproximación de los costes del proyecto.

Anexo IV. Pliego de condiciones. Condiciones legales del proyecto.

Anexo V. Artículo publicado. Se incluye el artículo titulado "WAI-Routes: a route-estimation system for aiming public transportation users with cognitive impairments", escrito a raíz de este proyecto. Será publicado en la conferencia *Advances in Computer Science and Engineering* (ACSE 2010), que tendrá lugar del 15 al 17 de marzo del presente año en Sharm El Sheikh (Egipto).

2 Trabajo relacionado

En este capítulo se va a explicar el trabajo relacionado con este proyecto. En el primer apartado se van a comparar tres sistemas de guiado para personas con discapacidades frente a WAI-Routes. En el segundo apartado se van a explicar los diferentes algoritmos de búsqueda del camino mínimo estudiados para calcular la ruta.

2.1. Sistemas relacionados con WAI-Routes

Antes de comenzar a desarrollar WAI-Routes se hizo un estudio sobre las técnicas o proyectos relacionados con este proyecto. De los artículos que se leyeron, a continuación se mencionan los más representativos, comentando las similitudes y diferencias con WAI-Routes.

Reppening y Loannidou [11] plantean el uso de un agente de movilidad (*Mobility Agents*) que funciona como eje central de información entre los autobuses, los cuidadores y los usuarios. El agente de movilidad proporciona opciones de destino, recibe las elecciones del usuario, analiza la información de los autobuses con GPS y compara la ubicación de los viajeros con la de dichos autobuses. WAI-Routes sigue un enfoque similar al planteado por estos autores con dos diferencias sustanciales. Por un lado, en WAI-Routes no se tiene en cuenta la posición del medio de transporte pero si la ubicación del usuario respecto a la ruta que debe seguir. Por otro lado, en *Mobility Agents* el cuidador decide cómo se mostrará la ruta al usuario, mientras que en WAI-Routes el cuidador (de haberlo) podría modificar las preferencias a la hora de calcular la ruta y estar al corriente de las trayectorias que siguen los usuarios, pero es el propio sistema el que decide automáticamente cómo se muestra la información y que ruta se debe seguir.

Chang et al. [12] y Tsai [13] proponen usar una PDA que muestra las direcciones en

tiempo real y las instrucciones mostrando fotos en un navegador web, redirigido por etiquetas de códigos QR decodificados que han sido fotografiados por la cámara integrada de la PDA. Por el contrario, WAI-Routes localiza la posición del usuario a través del GPS de la PDA y guía al usuario según un grafo de la red de trasporte público almacenado en un servidor.

Patterson et al. [4] exponen un sistema que no requiere una entrada explícita del usuario, si no que genera su sugerencia de planificación de camino y predicciones de destino de una manera no supervisada a partir de observaciones pasadas del usuario. El sistema registra señales GPS en lugares donde el usuario se detuvo durante un largo período de tiempo y aprende a donde quiere ir el usuario, pero no proporciona destinos a los que el usuario no ha viajado. A diferencia de ello, WAI-Routes no aprende rutas y requiere entradas de usuario que especifiquen las características (especificación de filtros) a tener en cuenta para calcular la ruta según las necesidades de cada usuario. Además permite que el destino sea cualquier punto ubicado dentro de la Comunidad de Madrid.

Por otro lado, hay diversas páginas web que dan información sobre posibles rutas usando el transporte público, como por ejemplo el Sistema de Información de Transportes de la Comunidad de Madrid [9] y Transports Metropolitans de Barcelona [14]. La gran diferencia con WAI-Routes es que éste proporciona muchas más características u opciones a tener en cuenta para calcular la ruta, es decir, da más facilidades al usuario que quiera usar la aplicación.

2.2. Algoritmos de búsqueda del camino mínimo

A la hora de calcular la ruta se han estudiado cinco algoritmos para encontrar el camino más corto: Bellman-Ford, Floyd-Warshall, Johnson, A* y Dijkstra.

Bellman-Ford. [15] Genera el camino más corto desde un nodo a todos los demás nodos de un grafo dirigido ponderado en el que el peso de las aristas puede ser tanto positivo como negativo. Consiste fundamentalmente en calcular la distancia del nodo inicial a los demás sin dar ningún salto, luego dando un salto, después dando dos saltos, etc. En cada iteración del algoritmo, para cada par de nodos (u,v) se hace la siguiente comprobación: si $C(v) > C(u) + C(u,v)$ entonces $C(v) = C(u) + C(u,v)$ y $P(v) = u$., siendo

C(v) el coste desde el nodo inicial al nodo *v* y *P(v)* el nodo predecesor de *v*. Es decir, si el coste del nodo *v* es mayor que el coste *u* mas el coste entre *u* y *v*, entonces el coste de *v* pasa a ser el coste de *u* más el coste entre *u* y *v*. Por lo tanto, el nodo predecesor de *v* será *u*. Este algoritmo necesita realizar *n-1* iteraciones, siendo *n* el número de nodos del grafo.

El algoritmo de Dijkstra resuelve este mismo problema en un tiempo menor pero los pesos de las aristas tienen que ser positivos. Dado que en el grafo empleado no existen pesos negativo se este algoritmo se descartó.

Floyd-Warshall. [16] [17] Encuentra el camino mínimo entre todos los pares de nodos en un grafo ponderado usando una matriz de *n*x*n*, siendo *n* el número de nodos del grafo. Esta matriz contiene las distancias iniciales de todos los nodos a todos los nodos (en caso de no conocer la distancia se toma como ∞). Siendo D*ij* la distancia del nodo *i* al nodo *j*, el algoritmo se calcula de la siguiente manera: para *k*=1 hasta *k*=*n*, para todo nodo *i*,*j* desde 1 hasta *n*, se obtiene la distancia mínima como D*ij* = min(D*ij*, D*ik* + D*kj*). Es decir, la distancia del nodo *i* al nodo *j* es el mínimo entre la distancia de ir de *i* a *j* directamente o de ir de *i* a *j* pasando por el nodo *k*. Una vez calculada la matriz final con todas las distancias mínimas entre todos los pares de nodos no es necesario volver a calcularla, pero si hay un fallo en un nodo o en un enlace no se puede actualizar.

Para WAI-Routes interesa que el grafo pueda actualizarse fácilmente, ya que pueden variar los costes según la hora del día (por ejemplo en hora punta), pueden surgir averías, etc. Por eso este algoritmo también fue descartado.

Johnson. [18] Encuentra el camino más corto entre todos los pares de vértices de un grafo dirigido. Si el grafo es disperso (el número de aristas es mucho menor que el número máximo de aristas en el grafo) este algoritmo es mucho mejor que el de Floyd-Warshall. El algoritmo de Johnson utiliza el algoritmo de Bellman-Ford para transformar el grafo inicial en otro sin pesos negativos, y después aplica Dijkstra sobre este último grafo. En primer lugar se añade un nuevo nodo *q* al grafo, conectado a cada uno de los nodos del grafo por una arista de peso cero. En segundo lugar, se utiliza el algoritmo de Bellman-Ford, empezando por el nuevo nodo *q*, para determinar para cada nodo *v* el coste mínimo *h(v)* del camino de *q* a *v*. Todos los valores *h(v)* son negativos, ya que *q* tiene una arista de unión con cada nodo de peso 0, y por tanto el camino más corto no puede ser mayor que ese valor. En tercer lugar, a las aristas del grafo original se les cambia el peso usando los

valores calculados por el algoritmo de Bellman-Ford de la siguiente forma: una arista de *u* a *v* con coste *c(u, v)*, tendrá un nuevo coste *c(u, v) + h(u) – h(v)*. Con este cambio todos los costes del grafo quedan positivos. Por último, para cada nodo *s* se usa el algoritmo de Dijkstra para determinar el camino más corto entre *s* y los demás nodos, usando el grafo con los costes modificados.

Como se ha mencionado anteriormente, el grafo empleado no tiene pesos negativos, así que no tendría sentido usar este algoritmo pudiendo emplear directamente el algoritmo de Dijkstra.

A*. [19] Encuentra el camino de menor coste entre un nodo inicio y un nodo destino. El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad. Evalúa los nodos combinando *g(n)*, el coste total desde el nodo inicio hasta alcanzar el nodo *n*; y *h(n)*, el coste estimado del camino mínimo desde el nodo *n* hasta el nodo objetivo. Así, el coste mínimo estimado de la solución a través de *n* será *f(n) = g(n) + h(n)*. Para que A* sea óptimo, *h(n)* debe ser una heurística admisible, es decir, *h(n)* nunca debe sobrestimar el coste de alcanzar el objetivo. Además, *h(n)* debe cumplir la condición de consistencia (también llamada monotonía) para evitar que el algoritmo devuelva una solución subóptima.

Este algoritmo es más eficiente que los anteriores pero es más complejo de implementar, dado que no es sencillo hallar una *h(n)* adecuada.

Dijkstra. [20] [21] Calcula todos los caminos de menor coste desde un nodo al resto de los nodos de un grafo dirigido ponderado con costes únicamente positivos. Para WAI-Routes, es mejor que Bellman-Ford, Floyd-Warshall y Johnson; y es más fácil de implementar que A*. Por estos motivos WAI-Routes empleará Dijkstra para calcular la ruta. En el apartado 4.1.1 del capítulo 4 se detallará cómo funciona este algoritmo y cómo se ha aplicado a WAI-Routes.

3 Arquitectura del sistema

La arquitectura de WAI-Routes sigue el modelo clásico de Cliente-Servidor. En la Figura 3.1 se puede observar el esquema de la arquitectura empleada.

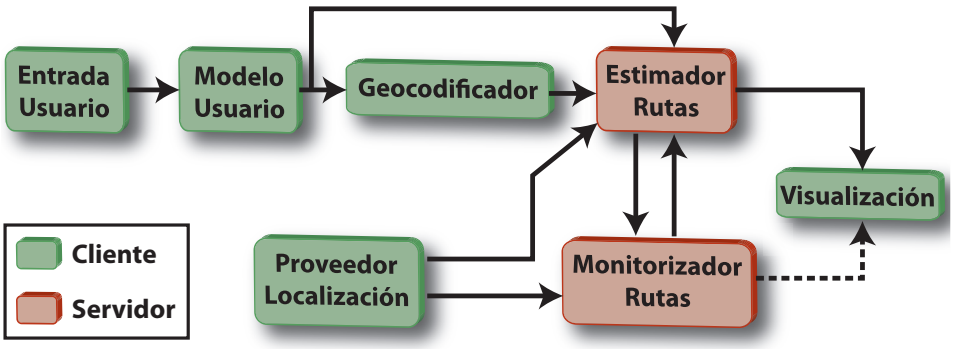


Figura 3.1: Arquitectura de WAI-Routes. Los módulos en verde se incluyen en el cliente, mientras que los módulos en rojo se corresponden con la funcionalidad integrada en el servidor.

Se ha escogido este modelo principalmente porque de esta manera toda la información de la red de transporte público se encuentra en el servidor. Así, en el caso en el que haya algún cambio en dicha red (por ejemplo si hay obras en una línea de metro y está temporalmente fuera de servicio) las modificaciones se hacen directamente en el servidor, ahorrándose el tener que actualizar todos los terminales cada vez que ocurra algo similar. Otra ventaja de emplear esta arquitectura es que facilita la implementación en diferentes tipos de terminales y plataformas, ya que dada una aplicación concreta únicamente hay que adaptar la interfaz de un cliente a otro. Esto es posible debido a que el algoritmo de búsqueda y la información del transporte público permanecen en el servidor.

Se ha podido optar por este modelo gracias a que los terminales móviles actuales tienen la posibilidad de estar siempre conectados a Internet, incluso dentro de la red de metro. Ahora bien, por motivos de eficiencia siempre sería posible realizar una copia local de la información más usada en el terminal, de manera que el cálculo de ciertas rutas no requiera el uso

del servidor.

El servidor está dividido en dos bloques, el ***Estimador de Rutas*** y el ***Monitorizador de Rutas***, que se explicarán con más detalle en el capítulo 4.

A su vez, el cliente está dividido en cinco bloques, la ***Entrada de Usuario***, el ***Modelo de Usuario***, el ***Proveedor de Localización***, el ***Geocodificador*** y la ***Visualización***, que serán analizados en los capítulos 5 y 6.

Para almacenar la red de transporte público de Madrid en el servidor se ha creado un grafo que contiene todas las uniones entre las distintas estaciones de los distintos sistemas de transporte, así como la información relativa a dichas estaciones o uniones. A continuación se va a explicar cómo se ha modelado el grafo de la red de trasporte público usado en la aplicación.

3.1. Red de transporte público

Dentro del servidor se incluye el grafo de la red de transporte público, el cual se compone de tres tipos de elementos: estaciones, líneas y transbordos.

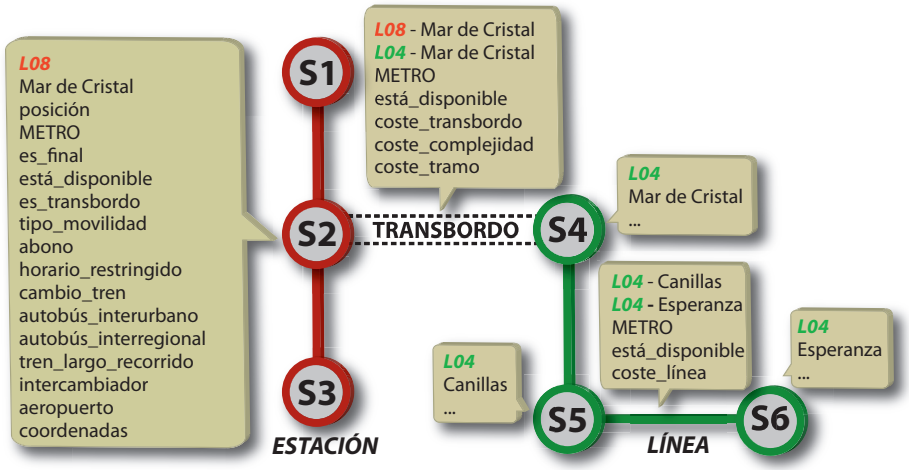


Figura 3.2: Esquema del grafo de la red de transporte público. Las estaciones de la línea 8 aparecen en rojo, mientras que las de la línea 4 están en verde.

En la Figura 3.2 se pueden diferenciar claramente estos tres elementos. Las estaciones vienen representadas por círculos, que son de distinto color según pertenezcan a la línea 8 (en color rojo) o a la 4 (en color verde), como por ejemplo S2 ó S5. Las líneas vienen

representadas como un segmento que une dos estaciones de una misma línea, como por ejemplo el tramo que une S5 y S6. Por último, los transbordos están representados como un segmento con dos líneas discontinuas que une dos estaciones pertenecientes a líneas distintas, como puede ser el tramo que une S2 y S4.

Cada elemento tiene un conjunto de propiedades que lo describe (véase los apartados que vienen a continuación). La información de todos los elementos se almacena en ficheros a los que accederá el servidor para crear el grafo.

Es importante señalar que aunque WAI-Routes ha sido probado con el grafo de la red de transporte público de la Comunidad de Madrid podría usar la red de transporte de cualquier otra ciudad, simplemente modificando los ficheros que contienen la información de las estaciones, líneas y transbordos.

El objetivo inicial del proyecto era tener en cuenta todos los tipos de transporte público disponibles en la Comunidad de Madrid, es decir, metro, metro ligero, cercanías y autobús. Sin embargo, el autobús se tuvo que descartar por la dificultad a la hora de obtener la información necesaria, así como por la complejidad para procesar dicha información y utilizarla para guiar al usuario.

A continuación se detallarán cada uno de los elementos que componen el grafo.

3.1.1. Estaciones

Las estaciones contienen la información de todas las estaciones de metro, metro ligero y cercanías. Cada estación se identifica unívocamente por el tipo y número de la línea junto al nombre de la estación.

La información que define cada estación es:

- Tipo y número de la línea. El tipo puede ser “L” para el metro, “ML” para el metro ligero o “C” para el cercanías.

- Nombre de la estación.

- Posición que ocupa la estación respecto a la línea.
- Si la estación es final de trayecto.
- Si está disponible.
- Si es una estación de transbordo.
- Si está adaptada (total o parcialmente) a personas con movilidad reducida.
- Tipo de abono transportes necesario.
- Si es una estación con horario restringido.
- Si hay que hacer cambio de tren.
- Si es una terminal de autobús interurbano.
- Si es una terminal de autobús interregional.
- Si es una estación de ferrocarril de largo recorrido.
- Si es un intercambiador.
- Si se corresponde con una terminal de aeropuerto y hay que pagar el pertinente suplemento.
- Las coordenadas de la estación (latitud y longitud expresados en grados decimales).

```
(1)          (2)           (3) (4)  (5)  (6)     (7)     (8) (9) (10) (11) (12) (13) (14) (15)     (16)          (17)
L01 PLAZA_DE_CASTILLA 4 NO SI SI PARCIAL A NO NO NO NO NO SI NO 40.465486 -3.689403

ML3 FERIAL_DE_BOADILLA 11 NO SI NO TOTAL B2 NO NO SI NO NO NO NO 40.401947 -3.879333
```

Figura 3.3: Definición de dos estaciones, la superior de metro y la inferior de metro ligero.

En la Figura 3.3 se puede observar un ejemplo de la definición de dos estaciones. La primera línea hace referencia a la estación de Plaza de Castilla(2), e indica que dicha estación está en la línea 1 de metro(1), ocupa la posición 4 dentro de esa línea(3), no

es final de trayecto(4), está disponible(5), es un transbordo(6), está adaptada parcialmente a personas con movilidad reducida(7), se necesita el abono A(8), no es una estación con horario restringido(9), no hay que hacer un cambio de tren(10), no es una terminal de bus interurbano(11) ni interregional(12), no es una estación de tren de largo recorrido(13), es una estación con intercambiador(14), no se corresponde con una terminal de aeropuerto(15) y está ubicada en la coordenada con latitud 40.465486(16) y longitud -3.689403(17). La segunda línea hace referencia a la estación de Ferial de Boadilla, e indica que dicha estación está en la línea 3 de metro ligero, ocupa la posición 11 dentro de esa línea, no es final de trayecto, está disponible, no es un transbordo, está totalmente adaptada a personas con movilidad reducida, se necesita el abono B2, es una terminal de autobús interurbano y está ubicada en la coordenada con latitud 40.401947 y longitud -3.879333.

Una misma estación que pertenece a dos líneas distintas (ya sean de igual tipo o de distinto), se trata como si fueran dos estaciones distintas con iguales parámetros, excepto el número de línea y/o el tipo de línea. También puede variar la posición que ocupa la estación en la línea, si es final de trayecto y la disponibilidad de la estación. Para mayor claridad observar el ejemplo de la Figura 3.4.

(a)

```
L08 NUEVOS_MINISTERIOS 8 SI SI SI PARCIAL A NO NO NO NO NO NO NO 40.44655 -3.691575
L10 NUEVOS_MINISTERIOS 18 NO SI SI PARCIAL A NO NO NO NO NO NO NO 40.44655 -3.691575
```

(b)

```
L06 MENDEZ_ALVARO 10 NO SI SI PARCIAL A NO NO SI SI NO NO NO 40.397275 -3.681961
C05 MENDEZ_ALVARO 13 NO SI SI PARCIAL A NO NO SI SI NO NO NO 40.397275 -3.681961
```

Figura 3.4: (a) Definición de dos nodos de la misma estación pero con distinto número de línea. (b) Definición de dos nodos de la misma estación pero con distinto tipo y número de línea.

Se llama "nodo" a cada renglón que define una estación en el grafo, sin embargo "estación" hace referencia a la estación física. Es decir, la estación de Gran Vía tiene dos nodos, el referente a L01-Gran Vía y el referente a L05-Gran Vía.

A la hora de numerar las estaciones dentro de las líneas se ha empezado por la estación de la línea que está primero más arriba y después más a la izquierda en el plano esquemático de la red. En el caso de las líneas circulares se ha comenzado por la estación que está más arriba y sea más representativa (por ejemplo, en la línea de metro L12 se ha empezado por la estación Puerta del Sur porque tiene transbordo

con otra línea de metro).

Los datos de las estaciones se han obtenido de la página web de Andén 1 (Asociación de amigos del metro de Madrid) [22] y de la del Sistema de Información de Transportes de Madrid ofrecida por el Consorcio de Transportes de Madrid y la Consejería de Transportes e Infraestructuras de la Comunidad de Madrid [9].

Las coordenadas de las estaciones se han obtenido de Google Earth [23], que las suministra en grados, minutos y segundos. Como las coordenadas proporcionadas por el GPS del terminal son recogidas en grados decimales se ha usado el conversor de la FCC [24].

3.1.2. Líneas

Las líneas contienen la información de todas las uniones entre estaciones de la misma línea (mismo número y tipo de línea). Cada línea se identifica unívocamente por la unión de dos identificadores de estación.

Cada línea consta de:

- Nombre de la estación con el tipo y número de la línea por cada una de las dos estaciones que forman la línea.
- Si está disponible.
- Coste asociado a la línea.

```
   (1)              (2)             (3) (4)
L02-GOYA*L02-MANUEL_BECERRA* SI 1

C08-CHAMARTIN*C08-NUEVOS_MINISTERIOS* SI 2
```

Figura 3.5: Definición de dos líneas, la superior de metro y la inferior de cercanías.

En la Figura 3.5 se puede apreciar como se definen dos tipos de líneas distintas. La primera línea hace referencia a la unión entre las estaciones Goya y Manuel Becerra, e indica que ambas estaciones pertenecen a la línea 2 de metro[(1),(2)], que la línea está

disponible[3] y que tiene un coste igual a 1[4]. La segunda línea hace referencia a la unión de las estaciones Chamartín y Nuevos Ministerios, e indica que ambas están en la línea 8 de cercanías, que la línea está disponible y que tiene un coste igual a 2.

El coste asociado a las líneas se explicará en el apartado 3.1.4.

3.1.3. Transbordos

Los transbordos contienen la información de todas las uniones entre estaciones de distintas líneas. Cada transbordo, al igual que las líneas, se identifica unívocamente por la unión de dos identificadores de estación.

La información que define cada transbordo es:

- Nombre de la estación con el tipo y número de la línea por cada una de las dos estaciones que forman el transbordo.
- Si está disponible.
- Tres costes asociados al transbordo.

```
L04-DIEGO_DE_LEON*L05-DIEGO_DE_LEON* SI 3 1 1
L05-PIRAMIDES*C10-PIRAMIDES* SI 3 2 1
```

Figura 3.6: Definición de dos transbordos, el superior entre dos líneas del mismo tipo (metro) y el inferior entre dos líneas de distinto tipo (metro y cercanías).

En la Figura 3.6 se puede observar la definición de dos transbordos. El renglón superior hace referencia a un transbordo entre líneas del mismo tipo (ambas son de metro) y el inferior hace referencia a un transbordo entre líneas de distinto tipo (una es de metro y otra de cercanías). También existen transbordos entre dos estaciones distintas, como se puede apreciar en la Figura 3.7.

```
L03-EMBAJADORES*L05-ACACIAS* SI 3 1 7
```

Figura 3.7: Definición de un transbordo entre dos estaciones distintas.

Los costes asociados a los transbordos serán explicados a continuación, en el apartado 3.1.4.

3.1.4. Costes asociados al grafo

Los costes asociados al grafo se emplean para estimar la mejor ruta. Tal como se ha mencionado previamente existe dos tipos de costes: de línea y de transbordo.

El **coste de la línea** es una estimación de lo que cuesta pasar de una estación a otra dentro de la misma línea (mismo tipo y número de línea). Por ejemplo, el coste entre estaciones de la línea L04 puede ser mayor que el coste entre estaciones de la línea L08 si en ésta última los trenes son más nuevos y van más rápido. O por ejemplo, el coste entre las estaciones de Valdeacederas y Tetuán (línea L01) será menor que el coste entre las estaciones de Chamartín y Nuevos Ministerios (línea C08) porque en la primera línea la distancia entre estaciones es mucho menor (véase Figura 3.8).

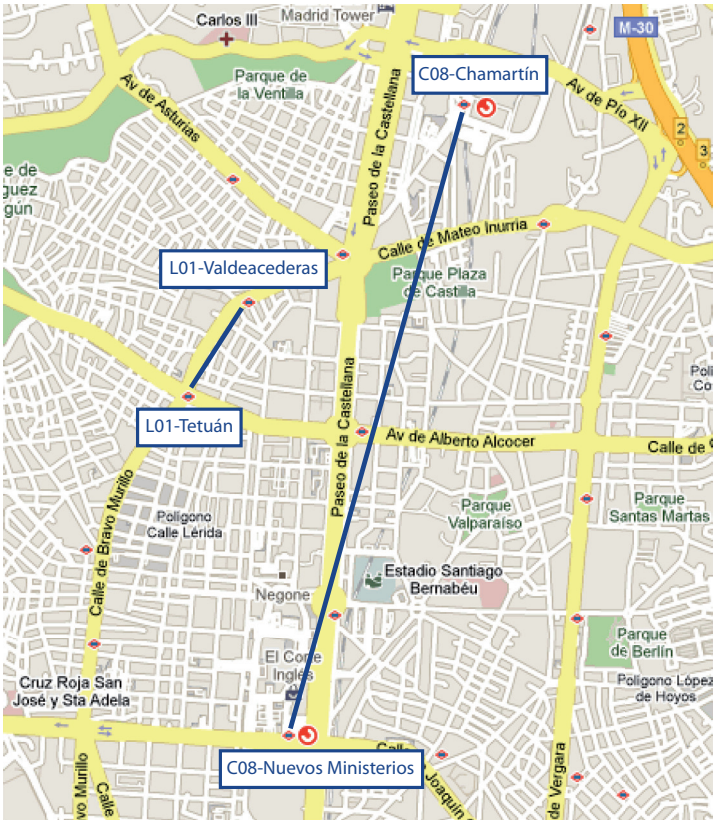


Figura 3.8: Comparativa de distancias entre dos estaciones de la línea L01 (Valdeacederas y Tetuán) y dos estaciones de la línea C08 (Chamartín y Nuevos Ministerios).

El **coste total del transbordo** es la suma de tres costes:

• **Coste asociado al transbordo en sí**. Todos los transbordos tienen asociados un mismo coste fijo que será mayor que el coste de la línea. Ahora bien, el coste de cada transbordo en particular se puede ver incrementado por los dos costes siguientes.

• **Coste asociado a la complejidad del transbordo**. Este coste hace referencia a la complejidad intrínseca que tiene el transbordo. Esta complejidad es independiente del perfil del usuario. Por ejemplo, el coste del transbordo en Cuatro Caminos será mucho mayor que el coste del transbordo en Gran Vía porque en el primer transbordo hay varios niveles y confluyen más líneas, con lo cual es más complejo.

• **Coste asociado al tramo**. Este coste incluye el coste inherente por tener que desplazarse caminando de una estación a otra a través del transbordo. Por ejemplo, el coste del transbordo L02-Noviciado con L10-Plaza de España será mayor que el de L03-Callao con L05-Callao porque en el primer caso hay que recorrer un tramo muy largo que implica gastar alrededor de cinco minutos.

Los costes, tanto de las líneas como de los transbordos, pueden verse modificados por los perfiles de cada usuario a través de los mecanismos de filtrado que se explican en el capítulo 4.

El coste de una línea siempre será menor que el coste total de un transbordo, es decir, los transbordos siempre se penalizan para evitarlos en la medida de lo posible, ya que de una forma u otra suponen una complejidad adicional.

La elección de los costes, en la mayoría de las ocasiones, es algo bastante complejo A la hora de asignarlos hay que tener muchos detalles en cuenta. ¿Quiero la ruta más corta, prefiero la ruta más fácil o me interesa la ruta más rápida?, ¿qué diferencia hay entre los diversos tipos de transporte?, ¿qué se tiene en cuenta para evaluar la dificultad de un transbordo?, ¿qué diferencias existen entre unas líneas y otras?

A la hora de decidir los costes de las líneas y los transbordos se hicieron una serie de pruebas para comprobar qué costes tenían más sentido. Esto se explicará con más detalle en el apartado 7.2 del capítulo 7.

4 Servidor personalizado de rutas

El servidor está dividido en dos bloques, el ***Estimador de Rutas*** y el ***Monitorizador de Rutas***, que se explicarán con más detalle a continuación.

4.1. Estimador de Rutas

Este módulo es el encargado de encontrar la mejor ruta para guiar al usuario. A la hora de calcular dicha ruta se ha aplicado el algoritmo de Dijkstra, que ha sido modificado según las necesidades del problema a resolver (véase apartado 4.1.1). Esta adaptación del algoritmo se va a denominar Dijkstra-F.

El cálculo de la ruta óptima requiere suministrar un punto de inicio y un punto de destino. Tanto el punto de inicio como el destino final pueden ser unas coordenadas en el espacio (latitud y longitud en grados decimales) introducidas por el usuario u obtenidas por el GPS del terminal, una dirección o una estación.

Si el nodo de inicio o destino, o ambos, son una dirección o una coordenada, el usuario tiene la opción de elegir el número de estaciones cercanas a tener en cuenta desde la posición del usuario, o de determinar un radio a partir del cual se tengan en cuenta o no las estaciones (véase apartado 4.1.2). Si no se especifica ni el número de estaciones cercanas ni el radio, por defecto se tomará la estación más cercana. Si dentro del radio establecido no hay ninguna estación, por defecto se cogerá también la estación más cercana y se avisará al usuario de la distancia que hay hasta la misma. En este caso, en el que hay varios nodos inicio y/o destino, se aplicará el algoritmo de Dijkstra-F de todos los posibles nodos inicio a todos los posibles nodos destino. La ruta escogida será la combinación con menor coste.

Se pueden establecer una serie de restricciones que permiten particularizar la ruta

obtenida. Una primera restricción es calcular la ruta pasando por una serie de estaciones predeterminadas por el usuario. Por ejemplo: "Quiero ir de P1 a P2 pasando por Nuevos Ministerios, ya que cerca de allí tengo que hacer un recado". Se puede pasar por todas las estaciones que quiera el usuario, pero éstas deberán ser introducidas en el orden en el que se quieran recorrer.

Más versátil es la siguiente modificación realizada al algoritmo de Dijkstra, que consiste en que a la hora de actualizar los nodos vecinos se pasan una serie de filtros para ver si las estaciones, líneas y/o transbordos cumplen los requisitos impuestos por el usuario.

Existen dos tipos de filtros, de eliminación y de modificación, que serán explicados en los apartados 4.1.3 y 4.1.4 respectivamente.

4.1.1. Algoritmo de Dijkstra aplicado a WAI-Routes, Dijkstra-F

Tal y como se vio en el apartado 2.2 del capítulo 2, donde se analizaron diversos algoritmos de búsqueda del camino mínimo, el algoritmo de Dijkstra calcula el camino mínimo de un nodo a todos los demás nodos de un grafo dirigido cuyos pesos deben ser únicamente positivos. Evidentemente, el grafo no puede tener nodos aislados.

El algoritmo utiliza una serie de parámetros que se definirán a continuación:

- C(u) ≡ coste mínimo desde el nodo inicial al nodo u.
- C(u,v) ≡ coste desde el nodo u hasta el nodo v.
- P ≡ almacena el nodo predecesor de cada nodo del grafo en el camino más corto.
- E ≡ almacena el conjunto de nodos estables (nodos que ya tienen la distancia más corta respecto al nodo inicial).
- I ≡ almacena el conjunto de nodos inestables (nodos que todavía no tienen la distancia mínima respecto al nodo inicial).
- n ≡ nodo inicial del grafo

El pseudocódigo del algoritmo se puede observar en la Figura 4.1.

```
// Inicializar
C = (∞)
P = ()
E = ()
I = ()

// Algoritmo
añadir n a I
C(n) = 0

Mientras I no esté vacío{

  u = extraerMínimo(I)
  añadir u a E
  eliminar u de I
  actualizarVecinos(u)
}
```

Figura 4.1: Pseudocódigo del algoritmo de Dijkstra.

En el bucle principal del algoritmo se hacen dos llamadas a dos funciones. En la Figura 4.2 se puede observar el pseudocódigo de la función *extraerMínimo* y en la Figura 4.3 el del procedimiento *actualizarVecinos*.

```
extraerMínimo(I){

  // encontrar el nodo de I con el menor
  // coste C y devolverlo
}
```

Figura 4.2: Pseudocódigo de la función *extraerMínimo*.

```
actualizarVecinos(u){

  Por cada nodo v adyacente a u que no esté en E{

    Si C(v) > C(u) + C(u,v){

      C(v) = C(u) + C(u,v)
      P(v) = u
      añadir v a I si antes no estaba en I
    }
  }
}
```

Figura 4.3: Pseudocódigo del procedimiento *actualizarVecinos*.

Para entender bien cómo funciona este algoritmo lo mejor es ver un ejemplo. La Figura 4.4 muestra el desarrollo del algoritmo paso a paso partiendo del grafo inicial (esquina superior izquierda). El nodo *a* será el nodo de partida.

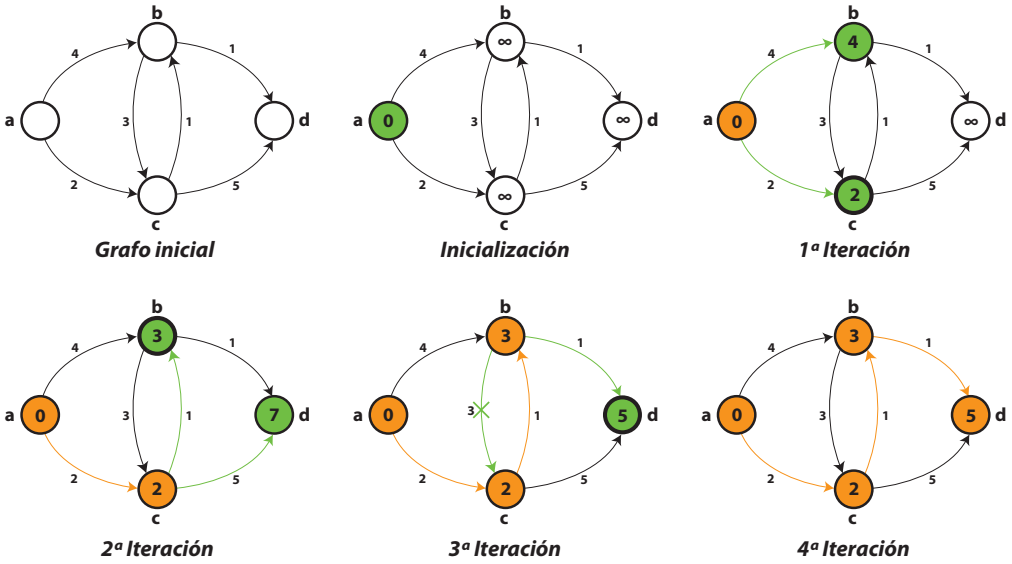


Figura 4.4: Ejemplo de desarrollo del algoritmo de Dijkstra. Los nodos estables están representados en color naranja, mientas que los nodos inestables son de color verde.

Inicialmente, el coste de todos los nodos se pone a ∞ excepto el coste del nodo *a* que se pone a cero (C(*a*) = 0). También se añade el nodo *a* al conjunto de nodos inestables I.

Como I no está vacío, en la **primera iteración** se extrae el nodo con menor coste, que será el nodo *a* puesto que es el único nodo en I. Se añade *a* al conjunto de nodos estables E, se elimina de I y se actualizan sus nodos vecinos. Dichos nodos son *b* y *c*. Dado que C(*b*) = ∞ su coste se modifica como C(*b*) = C(*a*) + C(*a*,*b*) = 0 + 4 = 4, y su nodo predecesor será P(*b*) = *a*. De igual forma, C(*c*) = C(*a*) + C(*a*,*c*) = 0 + 2 = 2 y P(*c*) = *a*. Por último se añaden los nodos *b* y *c* al grupo de nodos inestables I.

En la **segunda iteración** se vuelve a extraer el nodo de I con menor coste (en este caso el nodo *c*), se añade al conjunto de nodos estables E y se elimina de I. A continuación hay que actualizar sus nodos vecinos, que son *b* y *d*.

En la primera iteración del algoritmo se había llegado a la conclusión de que el camino más corto de *a* a *b* era el camino directo. Sin embargo, teniendo a *b* como vecino de *c*, se tiene que C(*b*) = 4 > C(*c*) + C(*c*,*b*) = 2 + 1 = 3. Es decir, es menos costoso llegar a *b* desde *c* que desde *a*. Por lo tanto se tendrá que C(*b*) = 3 y P(*b*) = *c*. De manera similar se tiene que C(*d*) = C(*c*) + C(*c*,*d*) = 2 + 5 = 7 y P(*d*) = *c*. Por último se añade el nodo *d* al grupo de nodos inestables I.

Hasta el momento se tienen los nodos *a* y *c* como nodos estables y *b* y *d* como nodos inestables.

Al igual que se hizo anteriormente, en la **tercera iteración** se debe extraer el nodo de I con menor coste, en este caso *b*, añadirlo a E y eliminarlo de I. El siguiente paso vuelve a ser actualizar los nodos vecinos. Los nodos adyacentes a *b* son *c* y *d*, pero como *c* ya es un nodo estable sólo se tendrá en cuenta el nodo *d*.

En la segunda iteración del algoritmo se había obtenido que el camino más corto de *a* a *d* era pasando por el nodo *c*. Sin embargo, pasando por el nodo *b* se tiene que C(*d*) = 7 > C(*b*) + C(*b*,*d*) = 4 +1 = 5. Por lo tanto se tendrá que C(*d*) = 5 y P(*d*) = *b*.

Finalmente, en la **cuarta iteración** se extrae el último nodo que queda en I, el nodo *d*, y se añade al conjunto de nodos estables E.

Los pasos seguidos al desarrollar el algoritmo y el resultado final pueden verse en la Tabla 4.1.

E	**Nodo *a*** **C(*a*) -- P(*a*)**	**Nodo *b*** **C(*b*) -- P(*b*)**	**Nodo *c*** **C(*c*) -- P(*c*)**	**Nodo *d*** **C(*d*) -- P(*d*)**	**I**
-	0 -- ninguno	∞ -- ?	∞ -- ?	∞ -- ?	a
a	***0 -- ninguno***	4 -- a	2 -- a	∞ -- ?	b,c
a,c	***0 -- ninguno***	3 -- c	**2 -- a**	7 -- c	b,d
a,c,b	***0 -- ninguno***	***3 -- c***	***2 -- a***	5 -- b	d
a,c,b,d	***0 -- ninguno***	***3 -- c***	***2 -- a***	***5 -- b***	-

Tabla 4.1: Pasos seguidos al desarrollar el algoritmo de Dijkstra.

Hasta ahora se ha explicado cómo funciona el algoritmo de Dijkstra de manera convencional. Pero como se ha mencionado anteriormente, WAI-Routes introduce una serie de modificaciones a este algoritmo para adaptarlo a las necesidades y requisitos del usuario. El pseudocódigo usado en la aplicación se puede ver en la Figura 4.5.

Para obtener rutas no interesa calcular la mínima distancia desde el nodo inicial a todos los nodos del grafo, así que la primera modificación consiste en interrumpir el algoritmo en cuanto se llega al nodo destino.

```
// Inicializar
C = (∞)
P = ()
E = ()
I = ()

// Comprobar nodo inicial y final
Si n == nodo destino
  devuelve INI_IGUAL_FIN

// Filtrar nodo inicial
Si filtrar(n) == ERROR
  devuelve ERROR

// Algoritmo
añadir n a I
C(n) = 0

Mientras I no esté vacío{

  Si costeRuta > costeMax
    devuelve COSTE_MAYOR

  u = extraerMínimo(I)
  añadir u a E

  Si u == nodo destino{

    Si costeRuta == costeMax
      devuelve COSTE_IGUAL

    devuelve OK
  }

  si no{

    eliminar u de I
    actualizarVecinos(u)
  }

  devuelve ERROR
}
```

Figura 4.5: Pseudocódigo del algoritmo de Dijkstra-F.

También se ha añadido un control de errores que comprueba si el nodo inicial es igual al nodo final, si el nodo inicial cumple los filtros impuestos por el usuario o si hubo cualquier otro tipo de error.

Por último se ha incorporado un control de costes. Como se mencionó anteriormente, existe la posibilidad de tener varios nodos de inicio y/o varios nodos de destino. Cuando esto ocurre se aplica el algoritmo para cada par de nodos inicio-destino y se escoge la combinación que da menor coste. En el pseudocódigo de la Figura 4.5, la variable *costeMax* contiene el valor mínimo que se tiene de todas las combinaciones inicio-destino estudiadas hasta el momento. La variable *costeRuta* contiene el valor del coste de la ruta actual (de la combinación inicio-destino actual) desde el nodo inicial hasta el nodo por el que va el algoritmo. Este valor va incrementándose en cada iteración del algoritmo. En cuanto el valor de *costeRuta* supere el valor de *costeMax* significará que esa combinación inicio-destino tiene un coste mayor del deseado ya

que se tiene otra combinación inicio-destino más eficiente, así que se interrumpe el algoritmo.

Existe un caso especial en el control de costes, ya que podría ocurrir que dos combinaciones inicio-destino tuvieran el mismo coste. En este caso daría igual coger un camino u otro, y el algoritmo seleccionaría la primera combinación. Sin embargo, si el nodo de inicio y el de destino están en la misma línea, se escoge la combinación que lleve directamente de un nodo al otro. A simple vista se puede pensar que es una condición innecesaria, ya que parece obvio que la mejor ruta sería recorrer la línea que une el inicio con el destino. Sin embargo, hay que tener en cuenta que esa opción puede no ser la óptima si se introducen algunos filtros que hagan variar los costes de las estaciones o líneas que haya entre el inicio y el destino. Por ejemplo, para ir de Argüelles a Pinar de Chamartín podría salir más rentable coger la línea de metro L06 y después la línea de metro L01 en vez de recorrer únicamente la línea de metro L04, la cual tiene muchas paradas entre el inicio y el destino. Precisamente para evitar este tipo de situaciones se ha impuesto la restricción de que si el nodo inicio y destino están en la misma línea se escoja la combinación que lleve directamente de un nodo a otro.

El procedimiento *actualizarVecinos* también ha sufrido alguna modificación, como por ejemplo comprobar al inicio si los nodos, líneas y/o transbordos están disponibles. Si por alguna avería hay una serie de nodos que están fuera de servicio no tiene sentido tenerlos en cuenta a la hora de calcular la ruta. Sin embargo, el cambio más importante es la incorporación de filtros al algoritmo. Una vez se ha comprobado la disponibilidad de los nodos, líneas y transbordos se pasan los filtros seleccionados por el usuario. El uso de estos filtros se verá más adelante, en los apartados 4.1.3 y 4.1.4. Por último se actualizarían los costes tal y como se vió en la Figura 4.3.

4.1.2. Estaciones de inicio y destino, número de estaciones cercanas y radio de estaciones

A la hora de determinar el inicio y el destino de una ruta, el usuario tiene tres posibilidades: a) suministrar el nombre de una estación, b) proporcionar una dirección o c) facilitar una coordenada (obtenida mediante el GPS del terminal sólo para el inicio

o introducida por el usuario).

Si el usuario introduce como inicio y/o destino una estación, el algoritmo usa una función llamada *estacionesPosibles* que devuelve todos los nodos asociados a una estación. Así, si se quiere ir desde Colombia a Plaza de Castilla, la aplicación tendrá que evaluar todas las posibles combinaciones entre los nodos de inicio L08-Colombia y L09-Colombia con los nodos de destino L01-Plaza de Castilla, L09-Plaza de Castilla y L10-Plaza de Castilla.

Si el usuario introduce como inicio y/o destino una dirección, la aplicación mediante el uso del *geocodificador* convertirá esa dirección en una coordenada (esta conversión la realiza el cliente, ya que al servidor sólo se envían como inicio o destino nombres de estaciones o coordenadas). A partir de ahí se hará de igual forma que si se hubiera introducido una coordenada directamente, tal y como se verá a continuación.

Por último, si el inicio y/o el destino viene dado por una coordenada hay tres posibilidades:

- El cliente determina el **número de estaciones** a tener en cuenta desde la coordenada de inicio y/o destino.

- El cliente determina el **radio** a partir del cual se tendrán en cuenta o no las estaciones.

- El cliente no determina ni número de estaciones ni radio, o determina un radio tan pequeño que no llega a abarcar ninguna estación. En este caso se coge la estación más cercana y se le notifica al cliente la distancia hasta dicha estación.

Para calcular las ***num* estaciones más cercanas** a un punto (definido como una coordenada, con su latitud y su longitud en grados decimales) se utiliza la función *estacionesCercanas*, que calcula las distancias euclídeas desde dicho punto hasta todas las estaciones del grafo y escoge las *num* estaciones con la distancia más pequeña. Después se usa la función *estacionesPosibles* para obtener los nodos posibles de esas estaciones.

La distancia euclídea entre dos puntos $A(x_1,y_1)$ y $B(x_2,y_2)$ del plano es la longitud

del segmento de recta que tiene por extremos A y B. Matemáticamente se calcula como:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Para obtener la distancia en metros hay que pasar la longitud y la latitud de grados decimales a radianes. Para ello simplemente se multiplica por π y se divide entre 180.

El pseudocódigo de la función *estacionesCercanas* puede verse en la Figura 4.6. Inicialmente se guardan las *num* estaciones primeras, que no es lo mismo que los *num* primeros nodos (recordar que, por ejemplo, la estación de Goya tiene los nodos L02-Goya y L04-Goya). Después, por cada nueva estación que se estudia se compara su distancia con la mayor distancia de las estaciones que se han guardado. Si la distancia de la nueva estación es menor, se añade a la lista de estaciones cercanas y se elimina el nodo con el que se había comparado. Al final, usando la función *estacionesPosibles*, se obtienen todos los nodos a partir de las *num* estaciones más cercanas.

```
coord: coordenada de inicio o destino
num: numero de estaciones a tener en cuenta
dist: tabla hash de nodos con su distancia
nodos: nodos de las "num" estaciones más cercanas

estacionesCercanas(coord,num){

  Para cada nodo n{

    // Almacenar las "num" estaciones primeras
    Si (tamaño de dist < num) && (dist no contiene(nombre de n)){

      d = distanciaEuclidea(coord,coordenada de n)
      añadir n y d a dist
    }

    si no{

      d = distanciaEuclidea(coord,coordenada de n)
      u = extraerMaximo(dist)

      Si d > distancia de u{

        eliminar u de dist
        añadir n y d a dist
      }

    }

  Para cada nodo n de dist{
    añadir estacionesPosibles(n) a nodos
  }

  devuelve nodos
}
```

Figura 4.6: Pseudocódigo de la función *estacionesCercanas*.

Para calcular las estaciones que están dentro de una circunferencia cuyo **radio** es impuesto por el cliente y cuyo centro es el punto de inicio y/o destino (de nuevo definido como una coordenada, con su latitud y su longitud en grados decimales) se utiliza la función *estacionesRadio*, que calcula las distancias geodésicas desde dicho punto hasta todos los nodos del grafo y escoge aquellos cuya distancia sea menor o igual al radio.

Generalmente, cuando se habla de calcular la distancia entre dos puntos se refiere a la longitud del segmento que une ambos puntos en línea recta. Sin embargo, cuando se quiere calcular la distancia entre dos puntos situados en una esfera (como es la superficie de la Tierra) se habla de la distancia geodésica entre esos dos puntos (para tener en cuenta la curvatura de la Tierra).

Para calcular la distancia geodésica entre dos puntos se han usado las F**órmulas de Vincenty (1975)** [25] para el problema inverso, el cual viene especificado en la Figura 4.7.

Dadas las coordenadas (Φ_1,λ_1) (Φ_2,λ_2) de dos puntos en el elipsoide, encontrar la distancia *s* a lo largo de la geodésica que los une y los acimutes directo α_{12} y recíproco α_{21}.

$$\tan u_1 = (1-f) \tan \Phi_1$$

$$\tan u_2 = (1-f) \tan \Phi_2$$

Partir de la aproximación $\theta = \omega = \lambda_1 + \lambda_2$

Iterar las 7 ecuaciones siguientes hasta que no se produzcan cambios significativos en θ:

1) $\text{sen}^2 \sigma = (\cos u_2 \text{ sen } \theta)^2 + (\cos u_1 \text{ sen } u_2 - \text{sen } u_1 \cos u_2 \cos \theta)^2$
2) $\cos \sigma = \text{sen } u_1 \text{ sen } u_2 + \cos u_1 \cos_2 u \cos \theta$
3) $\tan \sigma = \text{sen } \sigma / \cos \sigma$
4) $\text{sen } \alpha = \cos u_1 \cos u_2 \text{ sen } \theta / \text{sen } \sigma$
5) $\cos 2\sigma_m = \cos \sigma - (2 \text{ sen } u_1 \text{ sen } u_2 / \cos^2 \alpha)$
6) $C = (f/16) \cos^2 \alpha [4 + f (4 - 3 \cos^2 \alpha)]$
7) $\theta = \omega + (1-C) f \text{ sen } \alpha \{\sigma + C \text{ sen } \sigma [\cos 2\sigma_m + C \cos \sigma (-1 + 2 \cos^2 2\sigma_m)]\}$

$$(u^2) = (a^2 - b^2/ b^2) \cos^2 \alpha$$

$$A = 1 + (u^2/16384) \{4096 + u^2 [-768 + u^2 (320 - 175 u^2)]\}$$

$$B = (u^2/1024) \{256 + u^2 [-128 + u^2 (74 - 47 u^2)]\}$$

$$\Delta\sigma = B \text{ sen } \sigma \{\cos 2\sigma_m + (B/4) [\cos \sigma (-1 + 2 \cos^2 2\sigma_m) - (B/6) \cos 2\sigma_m (-3 + 4 \text{ sen}^2 \sigma) (-3 + 4 \cos^2 2\sigma_m)]\}$$

$$s = b * A * (\sigma - \Delta\sigma)$$

$$\tan \alpha_{12} = \cos u_2 \text{ sen } \theta / (\cos u_1 \text{ sen } u_2 - \text{sen } u_1 \cos u_2 \cos \theta)$$

$$\tan \alpha_{21} = \cos u_1 \text{ sen } \theta / (- \cos u_2 \text{ sen } u_1 + \text{sen } u_2 \cos u_1 \cos \theta)$$

Figura 4.7: Problema inverso de Vicenty (1975).

El acimut [26] de una semirrecta es el ángulo que forma dicha semirrecta con la dirección Norte-Sur en el sentido de las agujas del reloj. Si el ángulo se mide hacia el Norte el acimut de denomina topográfico, y si es hacia el Sur se denomina geodésico. En ambos casos el valor del ángulo puede valer de 0º a 360º. También puede verse como el ángulo que forma el plano imaginario que pasa por un punto en el espacio y el centro de la Tierra (plano vertical), con el plano que pasa por el Norte y el Sur geográficos y el cenit (plano meridiano). Para mayor claridad observar los ejemplos de la Figura 4.8.

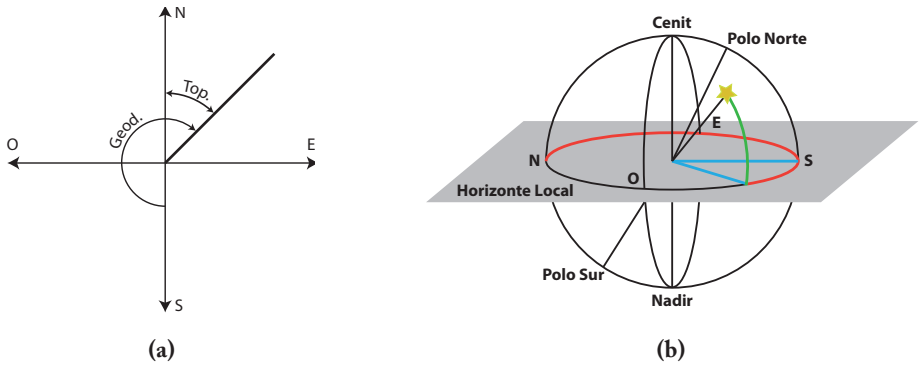


Figura 4.8: a) Acimut topográfico y geodésico de una semirrecta. b) Acimut de un punto en el espacio.

En nuestro caso el cálculo de los acimutes no interesa, lo importante es el cálculo de la distancia *s* a lo largo de la geodésica. Es importante recordar que para que la solución esté en metros las coordenadas tienen que estar en radianes en vez de en grados decimales.

WGS84 [27] [28] es un sistema de coordenadas cartográficas mundial que permite localizar cualquier punto de la Tierra (sin necesitar otro punto de referencia) por medio de tres unidades dadas. WGS84 son las siglas en inglés de *World Geodetic System 84* (Sistema Geodésico Mundial 84). Se trata de un estándar en geodesia, cartografía y navegación que data de 1984. Consiste en un patrón matemático que representa la Tierra por medio de un geoide (un tipo de elipsoide), un cuerpo geométrico más regular que la Tierra, que se denomina WGS 84. Los GPS utilizan este elipsoide, así que es el que se ha escogido para calcular la distancia geodésica. Sus parámetros vienen especificados en la Tabla 4.2.

Parámetro	Notación	Valor
Semieje mayor	a	6378137.0 m
Achatamiento	1/f	298.257223563
Semieje menor	b	6356752.31424 m
Constante gravitacional de la Tierra	GM	$(3986000.9 \pm 0.1) \times 10^8$ m^3/s^2
Velocidad angular de la Tierra	ω	7292115×10^{-11} rad/s

Tabla 4.2: Parámetros del elipsoide WGS 84.

Para resolver el problema inverso de Vicenty sólo se necesitan los valores del semieje mayor, del achatamiento y del semieje menor. A la hora de iterar las siete ecuaciones de la Figura 4.7 se ha escogida una precisión de $1x10^{-11}$. Se ha tomado este valor haciendo pruebas y comprobando que, sin la necesidad de iterar demasiadas veces, se obtenía una buena precisión.

El pseudocódigo de la función *estacionesRadio* se puede ver en la Figura 4.9.

```
coord: coordenada de inicio o destino
nodos: nodos dentro del radio

estacionesRadio(radio){

  Para cada nodo n{

    distancia = distanciaGeo(coord,coordenada de n)

    Si radio ≥ distancia
      añadir n a nodos
  }

  devuelve nodos
}
```

Figura 4.9: Pseudocódigo de la función *estacionesRadio*.

4.1.3. Filtros de eliminación

Los filtros de eliminación (FE) sirven para suprimir estaciones, líneas y/o transbordos que no cumplen una serie de requisitos impuestos por el usuario a través de dichos filtros. Los FE, como su propio nombre indican, eliminan algunas partes del grafo para que no sean tenidas en cuenta a la hora de calcular la ruta. Estos filtros podrían sustituirse por un "que no ocurra tal evento".

Si debido a los FE es imposible calcular la ruta se avisará al cliente del problema.

Por ejemplo, no se puede ir de P1 a P2 sin pasar por la línea de metro L05 cuando P2 es una estación de la línea de metro L05 y no es una estación con transbordo (de ser una estación con transbordo se podría llegar a la misma estación desde otra línea).

Existen diez tipos de filtros de eliminación que serán explicados a continuación.

FE Abono. Elimina la estación que esté fuera de la zona tarifaria del abono que tiene el usuario. Con este filtro se descarta la posibilidad de salir de la zona tarifaria que cubre el abono del usuario, evitando tener que pagar un suplemento por cambio de zona. Por ejemplo: "Quiero ir de P1 a P2 con mi abono B2".

Los abonos que se han tenido en cuenta son el A, B1, B2, B3, C1, C2, E1 y E2 (véase Figura 4.10). Como la aplicación está limitada a la Comunidad de Madrid, los abonos E1 y E2 son tomados como un C2.

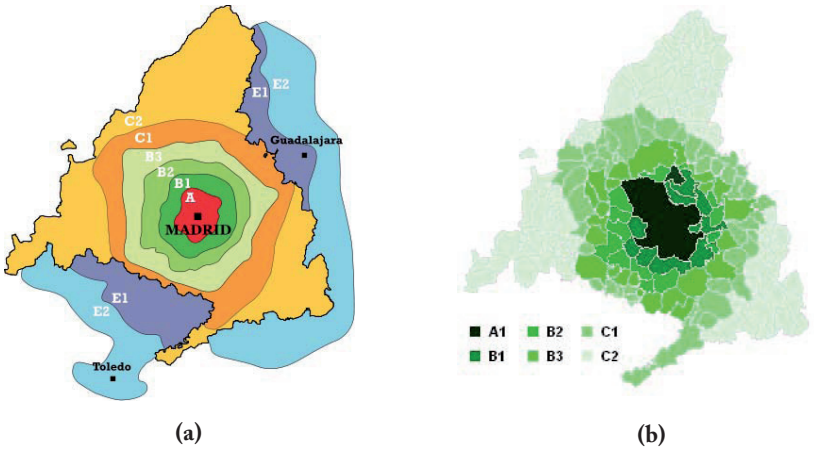


Figura 4.10: Mapa de las zonas tarifarias del transporte público en: (a) Madrid y alrededores . (b) Madrid por municipios. [29]

El pseudocódigo de este filtro puede verse en la Figura 4.11. Inicialmente se crea una tabla con los tipos de abonos asignando un número a cada tipo de abono, de tal forma que el abono que abarque menos zona tenga el número más pequeño y el abono que abarque mayor zona tenga el número más grande. A la hora de filtrar se comprueba si el abono del usuario tiene asociado un número mayor o igual que el del abono del nodo que se está estudiando. Si es mayor o igual indica que el usuario puede pasar con su abono por dicho nodo. En caso contrario el nodo es descartado y

no se tendrá en cuenta a la hora de calcular la ruta.

```
crearTablaAbono {

  tablaAbono = {A, B1, B2, B3, C1, C2}
}

filtrar(nodo){

  nUsuario = numero correspondiente a la posición del abono del usuario según tablaAbono
  nNodo = numero correspondiente a la posición del abono del nodo según tablaAbono

  Si nUsuario ≥ nNodo
    devuelve verdadero

  devuelve falso
}
```

Figura 4.11: Pseudocódigo de FE Abono.

FE Estaciones. Elimina las estaciones indicadas por el usuario de tal forma que la ruta no pasará por ninguna de ellas. Por ejemplo: "Quiero ir de P1 a P2 sin pasar por la estación de Sol porque sé que hay mucha aglomeración de gente".

El pseudocódigo de este filtro puede verse en la Figura 4.12. Se tiene una lista de estaciones a eliminar, así que por cada nodo estudiado se comprueba si el nombre de estación del nodo es igual a alguna de las estaciones que deben ser eliminadas. De ser así, el nodo será descartado y no se usará para calcular la ruta.

```
listaEstaciones: lista de estaciones a eliminar

filtrar(nodo){

  Para cada estación listaEstaciones{

    Si nombre de estación del nodo == estación
      devuelve falso
  }

  devuelve verdadero
}
```

Figura 4.12: Pseudocódigo de FE Estaciones.

FE Línea Metro. Elimina las líneas de metro indicadas por el usuario de tal forma que la ruta no pasará por ninguna estación que pertenezca a dichas líneas. Por ejemplo: "Quiero ir de P1 a P2 sin pasar por la línea de metro L05 porque generalmente va muy lenta".

El pseudocódigo de este filtro puede verse en la Figura 4.13. Igual que con el FE Estaciones, se tiene una lista de las líneas por las que no se quiere pasar. Así, por

cada nodo estudiado se comprueba que no esté en ninguna de las líneas eliminadas.

```
listaLíneaMetro: lista de líneas de metro a eliminar

filtrar(nodo){

  Para cada línea de listaLíneaMetro{

    Si línea del nodo == línea
      devuelve falso
  }

  devuelve verdadero
}
```

Figura 4.13: Pseudocódigo de FE Línea Metro.

FE Línea Metro Ligero. Elimina las líneas de metro ligero indicadas por el usuario de tal forma que la ruta no pasará por ninguna estación que pertenezca a dichas líneas. Por ejemplo: "Quiero ir de P1 a P2 sin pasar por la línea de metro ligero L02 porque no me gusta que las estaciones estén al aire libre". El pseudocódigo es similar al de FE Línea Metro (véase Figura 4.13) sólo que con líneas de metro ligero en vez de líneas de metro.

FE Línea Renfe. Elimina las líneas de cercanías indicadas por el usuario de tal forma que la ruta no pasará por ninguna estación que pertenezca a dichas líneas. Por ejemplo: "Quiero ir de P1 a P2 sin pasar por la línea de cercanías C08 porque sé que por obras hay cortes intermitentes". Igual que en el caso anterior, el pseudocódigo es similar al de FE Línea Metro (véase Figura 4.13) sólo que con líneas de cercanías en vez de líneas de metro.

FE Metro. Elimina todas las estaciones pertenecientes a la red de metro. Sólo se usará el cercanías y el metro ligero. Por ejemplo: "Quiero ir de P1 a P2 sin usar el metro porque me parece demasiado complicado".

```
filtrar(nodo){

  Si tipo de transporte del nodo == "metro"
    devuelve falso

  devuelve verdadero
}
```

Figura 4.14: Pseudocódigo de FE Metro.

El pseudocódigo de este filtro puede verse en la Figura 4.14. Se comprueba si los nodos estudiados pertenecen a cualquiera de las líneas de metro de la red, es decir, si

su tipo de transporte público es “metro”.

FE Metro Ligero. Elimina todas las estaciones pertenecientes a la red de metro ligero. Sólo se usará el metro y el cercanías. Por ejemplo: “Quiero ir de P1 a P2 sin usar el metro ligero porque hoy llueve y sé que la mayoría de las paradas están al aire libre”.

El pseudocódigo es similar al de FE Metro (véase Figura 4.14), sólo que el tipo de transporte será “metro ligero” en vez de “metro”.

FE Renfe. Elimina todas las estaciones pertenecientes a la red de cercanías. Sólo se usará el metro y el metro ligero. Por ejemplo: “Quiero ir de P1 a P2 sin usar el cercanías porque nunca sé en que andén tengo que colocarme”.

De nuevo el pseudocódigo es similar al de FE Metro (véase Figura 4.14), sólo que el tipo de transporte en este caso será “renfe” en vez de “metro”.

FE Movilidad. Elimina las estaciones que no estén total o parcialmente adaptadas a personas con movilidad reducida. Por ejemplo: “Quiero ir de P1 a P2 sin pasar por estaciones que no estén adaptadas totalmente porque voy en silla de ruedas”.

El pseudocódigo de este filtro se puede ver en la Figura 4.15. Este tipo de filtro sólo actúa en las estaciones de inicio y destino y en las estaciones en las que hay que hacer transbordo, porque son las únicas situaciones en las que uno se va a desplazar y necesita que el entorno esté adaptado a sus necesidades.

Como se acaba de mencionar hay dos tipos de movilidad, total o parcial. Según la Asociación de amigos del metro de Madrid, Andén [22], una estación se considera completamente accesible (es decir, totalmente adaptada a personas con movilidad reducida) si una persona en silla de ruedas puede entrar en la estación, subirse a los trenes y hacer todas las combinaciones posibles de transbordos sin necesidad de acudir a ayuda externa (asistentes en la estación, rampas de quita y pon, etcétera). En cualquier estación en la que una de estas acciones no se pueda llevar a cabo, la estación estará parcialmente adaptada o sin ningún tipo de adaptación. Se consideran personas con movilidad reducida a aquellas que van en silla de ruedas, cargan bultos de un cierto volumen o peso, son ciegos total o parcialmente, son sordos total o parcialmente, son ancianos, son mujeres embarazadas, llevan cochecitos o sillas con niños, llevan niños

a cuestas o tienen cualquier dificultad física para desplazarse con normalidad.

```
unión: línea o transbordo que une el nodo actual con el nodo vecino

filtrar(nodo,nodoInicio,nodoDestino,unión){

  Si tipoMovilidad == "total"{

    Si (nodo == nodoInicio || nodo == nodoDestino) && movilidad de nodo == tipoMovilidad)
      devuelve verdadero

    Si la union es una línea
      devuelve verdadero

    Si la union es un transbordo && movilidad de los dos nodos de la unión == tipoMovilidad
      devuelve verdadero

    devuelve falso
  }

  Si tipoMovilidad == "parcial"{

    Si (nodo == nodoInicio || nodo == nodoDestino) && movilidad de nodo == tipoMovilidad) ||
       (nodo == nodoInicio || nodo == nodoDestino) && movilidad de nodo == "total")
      devuelve verdadero

    Si la unión es una línea
      devuelve verdadero

    Si la unión es un transbordo && (movilidad de los dos nodos de la unión == tipoMovilidad
                                     || movilidad de los dos nodos de la unión == "total")
      devuelve verdadero

    devuelve falso
  }

}
```

Figura 4.15: Pseudocódigo de FE Movilidad.

FE Transbordos. Elimina las uniones entre dos estaciones que impliquen un transbordo, es decir, que impliquen tener que bajarse del metro/metro ligero/cercanías para coger otro medio de transporte distinto, o para cambiarse de línea. Por ejemplo: "Quiero ir de P1 a P2 sin hacer transbordos para evitar perderme".

El pseudocódigo de este filtro se puede ver en la Figura 4.16.

```
unión: línea o transbordo que une el nodo actual con el nodo vecino

filtrar(unión){

  Si la unión es un transbordo
    devuelve falso

  devuelve verdadero
}
```

Figura 4.16: Pseudocódigo de FE Transbordos.

4.1.4. Filtros de modificación

Los filtros de modificación (FM) sirven para intentar evitar estaciones, líneas y/o transbordos que no cumplen una serie de requisitos impuestos por el usuario a través de dichos filtros. Los FM, como su propio nombre indican, modifican las estaciones, líneas y/o transbordos penalizándolos para no pasar por ellos a no ser que sea inevitable. Estos filtros podrían sustituirse por un "a ser posible, que no ocurra tal evento".

Para evitar que la ruta pase por una estación, línea y/o transbordo, los FM aumentan el coste de las líneas y los transbordos de tal forma que no salga rentable pasar por ellos. Por otro lado, se debe impedir que una línea y/o transbordo se penalice en más de una ocasión por el mismo filtro. Esto puede ocurrir si al aplicar el algoritmo de Dijkstra-F se estudia en más de una ocasión la misma línea y/o transbordo. Se recuerda que, tal y como se explicó en el apartado 4.1.1, la acción de filtrado se realiza en la función *actualizarVecinos*. Esto quiere decir que cada vez que se actualice una línea y/o transbordo será filtrado. Sin embargo, una misma línea y/o transbordo sí puede ser modificado varias veces por distintos filtros.

Por todo ello se ha creado la clase *ControlModificar*, que guarda la información de los distintos filtros que han modificado una línea y/o transbordo, y el valor de su coste modificado. Es decir, por cada línea y/o transbordo que es modificado por un filtro se crea un objeto de la clase *ControlModificar*. Todos estos objetos son guardados en una *tabla hash* (llamada *modificando*) junto con el identificador de la línea y/o transbordo correspondiente.

Antes de modificar una línea y/o transbordo se mira si está en *modificando*. Si está es que ya ha sido filtrado anteriormente, así que se mira su correspondiente objeto *ControlModificar* y se comprueba si ya fue modificado por el mismo filtro. Si es así no se vuelve a modificar su coste. Si no fue modificado anteriormente por ese filtro se procede a aumentar su coste y a registrar dicho filtro en *ControlModificar*. Si no está en *modificando* significa que todavía no ha sido modificado por ningún filtro, así que se crea un objeto *ControlModificar* asociado a esa línea y/o transbordo, se modifica su coste y se registra dicho filtro, y por último se añade dicho objeto con su correspondiente identificador de la línea y/o transbordo a *modificando*.

Al igual que con los FE, hay diez tipos de filtros de modificación (FM) que serán detallados a continuación.

FM Abono. Evita las estaciones que están fuera de la zona tarifaria del abono transportes del usuario. Por ejemplo: "A ser posible, quiero ir de P1 a P2 usando sólo mi abono B2".

```
unión: línea o transbordo que une el nodo actual con el nodo vecino
modificando: tabla hash que contiene los identificadores de las líneas y transbordos y su
             objeto controlModificar asociado.

crearTablaAbono {

  tablaAbono = {A, B1, B2, B3, C1, C2}
}

filtrar(nodo,unión,modificando,nodoDestino){

  nUsuario = numero correspondiente a la posición del abono del usuario según tablaAbono
  nNodo = numero correspondiente a la posición del abono del nodo según tablaAbono

  Si (nodo != nodoDestino) && (nUsuario < nNodo){

    Si la unión es una línea && aparece en modificando{

      Si no ha sido modificada anteriormente por FE Abono{
        modificar coste línea en controlModificar de modificando
        modificar control_abono en controlModificar de modificando
      }
    }

    Si la unión es una línea && no aparece en modificando{

      crear objeto de ControlModificar
      modificar coste línea en controlModificar
      modificar control_abono en controlModificar
      añadir controlModificar y el identificador de la línea a modificando
    }

    Si la unión es un transbordo && aparece en modificando{

      Si no ha sido modificado anteriormente por FE Abono{
        modificar coste transbordo en controlModificar de modificando
        modificar control_abono en controlModificar de modificando
      }
    }

    Si la unión es un transbordo && no aparece en modificando{

      crear objeto de ControlModificar
      modificar coste transbordo en controlModificar
      modificar control_abono en controlModificar
      añadir controlModificar y el identificador del transbordo a modificando
    }
  }
}
```

Figura 4.17: Pseudocódigo de FM Abono.

El pseudocódigo de este filtro se puede ver en la Figura 4.17. Al igual que en el caso del FE Abono, inicialmente se crea una tabla con los tipos de abonos asignando un número a cada tipo de abono, de tal forma que el abono que abarque menos zona tenga el número más pequeño y el abono que abarque mayor zona tenga el número más grande.

Después se comprueba que el nodo estudiado no sea el nodo destino y que el abono del usuario tenga asociado un número menor que el del abono del nodo que se está estudiando, lo que significa que está fuera de la zona tarifaria abarcada por el abono del usuario. Sólo en este caso se modifica el coste de la línea y/o transbordo. No tiene sentido evitar un nodo que está fuera de la zona que abarca el abono del usuario si dicho nodo es el destino del usuario.

A continuación se hacen dos distinciones, dependiendo de si se está estudiando una línea o un transbordo. Se comprueba si la línea o el transbordo ha sido modificado anteriormente por el mismo filtro y en caso negativo se aumenta su coste y se registra el filtro que lo ha modificado.

FM Estaciones. Evita una serie de estaciones señaladas por el usuario para no pasar por ellas a no ser que no quede otra manera. Por ejemplo: "Quiero ir de P1 a P2 evitando en la medida de lo posible pasar por la estación de metro Sol".

El pseudocódigo de este filtro es similar al de FM Abono (ver Figura 4.17), sólo que se hacen esas comprobaciones para todas las estaciones que indicó el usuario. Por supuesto, en vez de modificar *control_abono* se modificaría *control_estaciones*.

FM Línea Metro. Evita una serie de líneas de metro indicadas por el usuario. Por ejemplo: "Quiero ir de P1 a P2 intentando no coger la línea de metro L05".

El pseudocódigo de este filtro puede verse en la Figura 4.18, donde se puede comprobar que sigue el esquema general comprobando la *tabla hash modificando* y los objetos de *ControlModificar* tanto para las líneas como para los transbordos.

En este filtro hay una serie de excepciones que hay que comentar, dado que no se modifican toda las líneas que señaló el usuario sin más. Antes de modificar algún coste la línea tiene que cumplir una serie de requisitos.

La primera comprobación es que si el nodo estudiado es el nodo destino tiene que ser un transbordo para que la línea de ese nodo sea filtrada. Si el nodo estudiado es el nodo destino no tendría sentido evitar la línea que llega a él puesto que es el nodo al que queremos llegar. Sin embargo, si el nodo es un transbordo se puede llegar por cualquier otra línea que puede no estar filtrada, así que en este caso si se modifica su

valor. No se ha tenido en cuenta el caso en el que el nodo destino es un transbordo y todas las líneas que llegan a él deben ser filtradas.

```
unión: línea o transbordo que une el nodo actual con el nodo vecino
modificando: tabla hash que contiene los identificadores de las líneas y transbordos y su
             objeto controlModificar asociado.
listaLínea: lista de las líneas de metro a modificar

filtrar(nodo,unión,modificando,nodoDestino){

  Por cada línea lin de listaLínea{

    Si (nodo == nodoDestino && nodo es transbordo) ||
       (nodo != nodoDestino && línea del nodo != línea del nodoDestino) ||
       (nodo != nodoDestino && línea del nodo == línea del nodoDestino && nodoDestino es transbordo{

      Si (tipo de transporte de nodo == "metro") && (línea del nodo == lin){

        Si la unión es una línea && aparece en modificando{

          Si lin no ha sido modificada anteriormente por FM LíneaMetro{
            modificar coste línea en controlModificar de modificando
            modificar control_línea_metro en controlModificar de modificando
          }
        }

        Si la unión es una línea && no aparece en modificando{

          crear objeto de ControlModificar
          modificar coste línea en controlModificar
          modificar control_línea_metro en controlModificar
          añadir controlModificar y el identificador de la línea a modificando
        }

        Si la unión es un transbordo && aparece en modificando{

          Si lin no ha sido modificado anteriormente por FM LíneaMetro{
            modificar coste transbordo en controlModificar de modificando
            modificar control_línea_metro en controlModificar de modificando
          }
        }

        Si la unión es un transbordo && no aparece en modificando{

          crear objeto de ControlModificar
          modificar coste transbordo en controlModificar
          modificar control_línea_metro en controlModificar
          añadir controlModificar y el identificador del transbordo a modificando
        }
      }
    }
  }
}
```

Figura 4.18: Pseudocódigo de FM Línea Metro.

La segunda comprobación es que si el nodo estudiado no es el nodo destino y no está en la misma línea que el nodo destino, se ejecuta el filtro. Ésta es la comprobación más intuitiva, puesto que no hay ningún motivo para no ejecutar el filtro ya que el nodo estudiado no dificulta el camino hasta llegar al nodo destino.

La tercera comprobación es que si el nodo estudiado no es el nodo destino pero está en su misma línea, el nodo destino tiene que ser un transbordo para que se ejecute el filtro. Se supone que si se llega a un nodo que está en la línea del nodo final

la ruta óptima será continuar sobre esa línea hasta llegar al destino. Por este motivo, si el nodo estudiado está en la línea del nodo destino y esta línea debe ser evitada no se modifica el coste de la línea. De no ser así, la ruta daría un rodeo para acabar metiéndose en la línea que se quería evitar. Sin embargo, si el nodo destino es un transbordo si que se ejecuta el filtro, dado que puede llegarse al final de la ruta por otra línea distinta.

FM Línea Metro Ligero. Es similar a FM Línea Metro solo que en esta caso se evitan las líneas de metro ligero señaladas por el usuario. Por ejemplo: "Quiero ir de P1 a P2, a ser posible sin coger la línea de metro ligero ML3".

Su pseudocódigo es igual al de la Figura 4.18, solo que el tipo de transporte en este casó será "metroligero" y la variable "listaLínea" contendrá las líneas de metro ligero que el usuario no desea utilizar.

FM Línea Renfe. De igual forma, penaliza las líneas de cercanías para ir por ellas sólo si es estrictamente necesario. Por ejemplo: "A ser posible quiero ir de P1 a P2 sin pasar por la línea de cercanías C08".

De nuevo, su pseudocódigo es similar al de la Figura 4.18, modificando el tipo de transporte a "renfe" y teniendo en cuenta que la variable "listaLínea" contendrá las líneas de cercanías por las que el usuario no quiere pasar.

FM Metro. Elude todas las estaciones de metro, es decir, dará prioridad al uso del metro ligero y el cercanías. Por ejemplo: "Quiero ir de P1 a P2 intentando no usar la red de metro".

El pseudocódigo de este filtro, al igual que en los casos anteriores, es similar al pseudocódigo de FM Línea Metro (véase Figura 4.18), sólo que en este caso no importa que línea de metro sea, dado que se quieren evitar absolutamente todas las líneas siempre y cuando sean de metro.

FM Metro Ligero. Es exactamente igual que FM Metro sólo que en vez de evitar todas las líneas de metro elude todas las líneas de metro ligero. En este caso se intentará usar solamente las líneas de metro y cercanías. Por ejemplo: "Quiero ir de P1 a P2 evitando en la medida de lo posible el metro ligero".

Su pseudocódigo es igual al de FM Metro cambiando el tipo de transporte a "metroligero".

FM Renfe. Intenta no tomar ninguna de las líneas de cercanías, así que tratará de usar exclusivamente las líneas de metro y metro ligero. Por ejemplo: "Me gustaría ir de P1 a P2 sin usar el cercanías, si es posible".

Su pseudocódigo también es similar al de FM Metro, ya que sólo habría que modificar el tipo de transporte a "renfe".

FM Movilidad. Evita todas las estaciones que no están adaptadas parcial o totalmente a personas con movilidad reducida (en el apartado 4.1.3 se explicó la diferencia entre adaptación parcial y total, y se describió cuándo se considera que una persona tiene movilidad reducida). Por ejemplo: "A ser posible, quiero ir de P1 a P2 teniendo en cuenta que necesito estaciones parcialmente adaptadas a personas con movilidad reducida porque llevo muletas".

En el caso de FE Movilidad (véase apartado 4.1.3), el filtro sólo actúa en las estaciones de inicio y destino y en las estaciones en las que hay que hacer transbordo, porque son las únicas situaciones en las que el usuario se va a desplazar y necesita que el entorno esté adaptado a sus necesidades. Sin embargo, para FM Movilidad no se penalizan las estaciones de inicio y destino porque hay que pasar por ellas obligatoriamente, así que no sería muy lógico intentar evitarlas. Por eso, este filtro sólo penaliza las estaciones con transbordo si la movilidad no es la deseada.

Para que un transbordo no sea filtrado, los dos nodos que lo forman deben de estar adaptados para personas con movilidad reducida. Si el usuario sólo solicita una movilidad parcial, entonces los nodos pueden estar tanto parcial como totalmente adaptados.

En la Figura 4.19 puede verse el pseudocódigo de este filtro. En este caso se ha añadido un procedimiento llamado *filtrarTransbordo* para que el código sea más legible, pero el funcionamiento es exactamente el mismo que en los filtros anteriores.

```
unión: línea o transbordo que une el nodo actual con el nodo vecino
modificando: tabla hash que contiene los identificadores de las líneas y transbordos
             y su objeto controlModificar asociado.

filtrar(unión,modificando){

  Si tipoMovilidad == "total"{

    Si la unión es un transbordo && (movilidad nodo1 de la unión != "total") ||
                                    movilidad nodo2 de la unión != "total"){

      filtrarTransbordo(unión,modificando)
    }
  }

  Si tipoMovilidad == "parcial"{

    Si la unión es un transbordo && (movilidad nodo1 de la unión != "parcial" ||
       movilidad nodo2 de la unión != "parcial" || movilidad nodo1 de la unión != "total") ||
       movilidad nodo2 de la unión != "total"){

      filtrarTransbordo(unión,modificando)

    }
  }
}


filtrarTransbordo(unión,modificando){

  Si el transbordo aparece en modificando{

    Si no ha sido modificado anteriormente por este filtro{
      modificar coste transbordo en controlModificar
      modificar control_movilidad en controlModificar
    }
  }

  Si no{

    crear objeto de ControlModificar
    modificar coste transbordo en controlModificar
    modificar control_movilidad en controlModificar
    añadir transbordo y controlModificar a modificando
  }
}
```

Figura 4.19: Pseudocódigo de FM Movilidad.

FM Transbordos. Intenta reducir el número de transbordos de la ruta. Por ejemplo: "Me gustaría ir de P1 a P2 realizando el menor número de transbordos posibles". Para ello, este filtro aumenta el coste de todas las uniones que sean transbordos y así consigue que la ruta los evite en la medida de lo posible. El pseudocódigo de este filtro puede verse en la Figura 4.20.

```
unión: línea o transbordo que une el nodo actual con el nodo vecino
modificando: tabla hash que contiene los identificadores de las líneas y
             transbordos y su objeto controlModificar asociado.

filtrar(unión,modificando){

  Si la unión es un transbordo && aparece en modificando{

    Si no ha sido modificado anteriormente por este filtro{
      modificar coste transbordo en controlModificar
      modificar control_transbordos en controlModificar
    }
  }

  Si la unión es un transbordo && no aparece en modificando{

    crear objeto de ControlModificar
    modificar coste transbordo en controlModificar
    modificar control_transbordos en controlModificar
    añadir transbordo y controlModificar a modificando
  }
}
```

Figura 4.20: Pseudocódigo de FM Transbordos.

4.1.5. Funcionamiento de los filtros

En los apartados 4.1.3 y 4.1.4 se han explicado los diferentes tipos de filtros que emplea WAI-Routes, centrándose en la funcionalidad e implementación de cada filtro en particular. En este apartado se va a dar a conocer dónde y cómo se integra el mecanismo de filtrado en Dijkstra-F.

Como se mencionó en el apartado 4.1.1, la modificación más representativa hecha al algoritmo de Dijkstra es la incorporación de filtros en la parte de actualizar los nodos vecinos. Primero se comprueba si los nodos, líneas y transbordos están disponibles. De ser así, a continuación se les pasan los filtros designados por el usuario. Si el nodo, línea y /o transbordo estudiado supera el filtro se actualizan sus nodos vecinos. En caso contrario, no se actualizan sus nodos vecinos y además dicho nodo no será tenido en cuenta a la hora de calcular la ruta.

Para realizar la acción de filtrado se ha creado una clase llamada *RealizarFiltrado*. Cuando se crea un objeto de esta clase, en el constructor se le pasan todos los filtros de eliminación y modificación que usará la aplicación en el cálculo de la ruta actual, así como la *tabla hash modificando*, que como se mencionó anteriormente guarda todos los identificadores de las líneas y transbordos que han sido modificados por algún filtro de modificación y la información de el/los filtro/s correspondiente/s que los

modificó (se recuerda que esta información queda guardada en un objeto de la clase *ControlModificar*, que se crea por cada línea y/o transbordo modificado).

La clase *RealizarFiltrado* tiene dos métodos, *filtrando* y *filtradoInicio*, que se explicarán a continuación.

El método *filtrando* se encarga de pasar el nodo y la unión estudiado por todos y cada uno de los filtros, tanto de eliminación como de modificación, que el usuario escogió para calcular la ruta. Este método se ejecuta cada vez que en el algoritmo de Dijkstra-F para calcular la ruta se ejecuta el método *actualizarVecinos* (véase apartado 4.1.1). Es decir, se filtran los nodos vecinos al nodo actual (el cual ya ha sido añadido a la lista de nodos estables) y las uniones entre ellos antes de actualizar sus valores. Si no pasan la acción de filtrado no sólo no son actualizados si no que tampoco se añadirán a la lista de nodos inestables, con lo que serán invisibles para el algoritmo y como consecuencia no serán tenidos en cuenta a la hora de calcular la ruta.

```
listaFE: lista con los filtros de eliminación que se usarán al calcular la ruta
listaFM: lista con los filtros de modificación que se usarán al calcular la ruta
modificando: tabla hash que contiene los identificadores de las líneas y transbordos y su
             objeto controlModificar asociado.
unión: línea o transbordo que une el nodo actual con el nodo vecino
n: nodo actual

filtrando(unión,n,nodoInicio,nodoDestino){

  Por cada filtro fe de ListaFE{

    Si nodo1 de la unión != n && fe.filtrar(nodo1,unión,nodoInicio,nodoDestino) == falso{
      devuelve ERROR
    }

    Si nodo2 de la unión != n && fe.filtrar(nodo2,unión,nodoInicio,nodoDestino) == falso{
      devuelve ERROR
    }
  }

  Por cada filtro fm de ListaFM{

    Si nodo1 de la unión != n{
      fm.filtrar(nodo1,unión,modificando,nodoDestino)
    }

    Si nodo2 de la unión != n{
      fm.filtrar(nodo1,unión,modificando,nodoDestino)
    }
  }

  devuelve OK
}
```

Figura 4.21: Pseudocódigo del método *filtrando*.

El pseudocódigo del método *filtrando* se puede observar en la Figura 4.21. Si el método devuelve "ERROR" significa que el nodo o la unión no han superado algún

filtro imprescindible; en caso contrario devuelve “OK”. Como es lógico, primero se empieza por los filtros de eliminación. En cuanto el método *filtrar* de uno solo de estos filtros (véase apartado 4.1.3) devuelva falso, se suspende la acción de filtrado y se devuelve “ERROR”. No tiene sentido seguir con el resto de los filtros, ya sea de eliminación o de modificación, porque en cuanto el nodo o la unión no cumpla un filtro de eliminación ya será descartado. Una vez se han pasado todos los filtros de eliminación se procede con los filtros de modificación, los cuales siempre devuelven “OK” porque, como se mencionó en el apartado 4.1.4, este tipo de filtros no descartan nodos ni uniones, sólo los evitan.

El método *filtrandoInicio* sólo sirve para filtrar el nodo de partida. Éste es un caso especial porque al ser el primer nodo que se estudia, según el pseudocódigo del algoritmo de Dijkstra que se explicó en el apartado 4.1.1, se añade directamente a la lista de nodos estables sin necesidad de ser actualizado. Por ese motivo, este método es un ligeramente diferente al del filtrado general. Su pseudocódigo puede verse en la Figura 4.22. Puede parecer un poco incoherente querer eliminar o evitar el nodo inicial, ya que a simple vista carece de sentido, pero se recuerda que a la hora de calcular una ruta puede haber varios posibles nodo de inicio. Gracias a este filtrado inicial se podrían eliminar o evitar algunas de esas opciones no deseadas por el usuario.

```
listaFE: lista con los filtros de eliminación que se usarán al calcular la ruta
listaFM: lista con los filtros de modificación que se usarán al calcular la ruta
modificando: tabla hash que contiene los identificadores de las líneas y transbordos y su
             objeto controlModificar asociado.
unión: línea o transbordo que une el nodo actual con el nodo vecino
n: nodo actual

filtrandoInicio(unión,n,nodoInicio,nodoDestino){

  Por cada filtro fe de ListaFE{

    Si fe.filtrar(nodoInicio,unión,nodoInicio,nodoDestino) == falso
      devuelve ERROR
  }

  Por cada filtro fm de ListaFM{

    fm.filtrar(nodoInicio,union,modificando,nodoDestino)
  }

  devuelve OK
}
```

Figura 4.22: Pseudocódigo del método *filtrandoInicio*.

En resumen, el uso de los filtros en la parte de actualizar los nodos vecinos en el algoritmo de Dijkstra-F se puede observar en la Figura 4.23.

```
a: nodo actual
u: unión(a,n)

Por cada nodo vecino n de a{

  Si n está disponible && u está disponible{

    Por cada filtro f{

      Si f es FiltroEliminar && hayQueEliminar(f,n,u)
        n descartado
        break

      Si f es FiltroModificar && hayQueModificar(f,n,u)
        modificarCoste(f,n,u)
    }

    Si noDescartado(n)
      ActualizarVecinosDijkstra(a,n,u)
  }

  Si no
    n descartado
}
```

Figura 4.23: Pseudocódigo del uso de los filtros cuando se actualizan los nodos vecinos en el algoritmo de Dijkstra-F.

4.1.6. Ejemplo de una ruta con y sin filtros

El usuario quiere ir desde su casa, en c/ Arequipa 13, hasta la estación de Islas Filipinas donde tiene una entrevista de trabajo.

Como el inicio de la ruta es una dirección (se traduciría en una coordenada, según lo visto en el apartado 4.1.2) y el usuario no ha especificado el número de estaciones cercanas o el radio de estaciones a tener en cuenta, WAI-Routes escogerá la estación más cercana e indicará la distancia a la que está dicha estación. En este caso la estación de inicio es Mar de Cristal, así que los posibles nodos de inicio serán L04-Mar de Cristal y L08-Mar de Cristal.

El destino de la ruta es Islas Filipinas, y como es una estación sin transbordo el único nodo posible de destino es L07-Islas Filipinas.

WAI-Routes calculará la ruta desde L04-Mar de Cristal hasta L07-Islas Filipinas, y desde L08-Mar de Cristal hasta L07-Islas Filipinas, y escogerá la ruta con menor

coste. El resultado puede verse en la Figura 4.24.

```
# La estación de inicio más cercana está a 69.55 metros #

L08-MAR_DE_CRISTAL
L08-PINAR_DEL_REY
L08-COLOMBIA
L08-NUEVOS_MINISTERIOS
L08-NUEVOS_MINISTERIOS
L06-CUATRO_CAMINOS
L06-GUZMAN_EL_BUENO
L07-GUZMAN_EL_BUENO
L07-ISLAS_FILIPINAS
```

Figura 4.24: Ruta calculada desde c/Arequipa hasta Islas Filipinas, sin usar ningún filtro.

Antes de salir de casa, el usuario se acuerda de que tiene que dejar un libro en la biblioteca, situada en frente de la parada de metro de Esperanza.

En esta ocasión, en vez de salir desde L08-Mar de Cristal la aplicación coge como nodo inicial L04-Mar de Cristal, ya que la estación por la que hay que pasar (Esperanza) está en la línea L04. La nueva ruta viene mostrada en la Figura 4.25.

```
# La estación de inicio más cercana está a 69.55 metros #

L04-MAR_DE_CRISTAL
L04-CANILLAS
L04-ESPERANZA
L04-ARTURO_SORIA
L04-AVDA_DE_LA_PAZ
L04-ALFONSO_XII
L04-PROSPERIDAD
L04-AVDA_DE_AMÉRICA
L07-AVDA_DE_AMÉRICA
L07-GREGORIO_MARAÑÓN
L07-ALONSO_CANO
L07-CANAL
L07-ISLAS_FILIPINAS
```

Figura 4.25: Ruta calculada desde c/Arequipa hasta Islas Filipinas, pasando por Esperanza.

Además, el usuario prefiere no pasar por Avda. de América a no ser que sea imprescindible, porque es un transbordo muy complicado ya que se juntan cuatro líneas de metro.

Para evitar la estación de Avda. de América se emplea el filtro FM Estación. Para ello no hay más remedio que ir desde Mar de Cristal hasta Esperanza por la línea L04 y volver a Mar de Cristal para hacer un transbordo y continuar por la línea L08. En Nuevos Ministerios habría que hacer otro transbordo y en Guzmán el Bueno otro

más. Se recuerda que, a diferencia del filtro de eliminación análogo, ser podría dar el caso de que la ruta pasara por Avda. de América. El itinerario obtenido se muestra en la Figura 4.26.

```
# La estación de inicio más cercana está a 69.55 metros #

L04-MAR_DE_CRISTAL
L04-CANILLAS
L04-ESPERANZA
L04-CANILLAS
L04-MAR_DE_CRISTAL
L08-MAR_DE_CRISTAL
L08-PINAR_DEL_REY
L08-COLOMBIA
L08-NUEVOS_MINISTERIOS
L06-NUEVOS_MINISTERIOS
L06-CUATRO_CAMINOS
L06-GUZMÁN_EL_BUENO
L07-GUZMÁN_EL_BUENO
L07-ISLAS_FILIPINAS
```

Figura 4.26: Ruta calculada desde c/Arequipa hasta Islas Filipinas, pasando por Esperanza y evitando Avda. de América.

Por último, el usuario tiene claro que de ninguna manera quiere coger la línea de metro L06, porque al ser circular le resulta muy complicado saber en que andén debe colocarse, además que algunas estaciones tienen tres andenes en lugar de dos.

El hecho de no pasar por la línea L06 se traduce como un filtro FE Línea Metro. La nueva ruta sería igual que la anterior hasta Nuevos Ministerios, donde en vez de coger la línea L06 cogería la línea L10 hasta Gregorio Marañón. Allí haría un transbordo a la línea L07, la cual llega directamente a la estación de destino Islas Filipinas. La ruta final se puede observar en la Figura 4.27.

```
# La estación de inicio más cercana está a 69.55 metros #

L04-MAR_DE_CRISTAL
L04-CANILLAS
L04-ESPERANZA
L04-CANILLAS
L04-MAR_DE_CRISTAL
L08-MAR_DE_CRISTAL
L08-PINAR_DEL_REY
L08-COLOMBIA
L08-NUEVOS_MINISTERIOS
L10-NUEVOS_MINISTERIOS
L10-GREGORIO_MARAÑON
L07-GREGORIO_MARAÑON
L07-ALONSO_CANO
L07-CANAL
L07-ISLAS_FILIPINAS
```

Figura 4.27: Ruta calculada desde c/Arequipa hasta Islas Filipinas, pasando por Esperanza, evitando Avda. de América y sin usar la línea de metro L06.

Para comprender mejor la selección de rutas véase en la Figura 4.28 el extracto de la red de metro de la Comunidad de Madrid usado en este ejemplo.

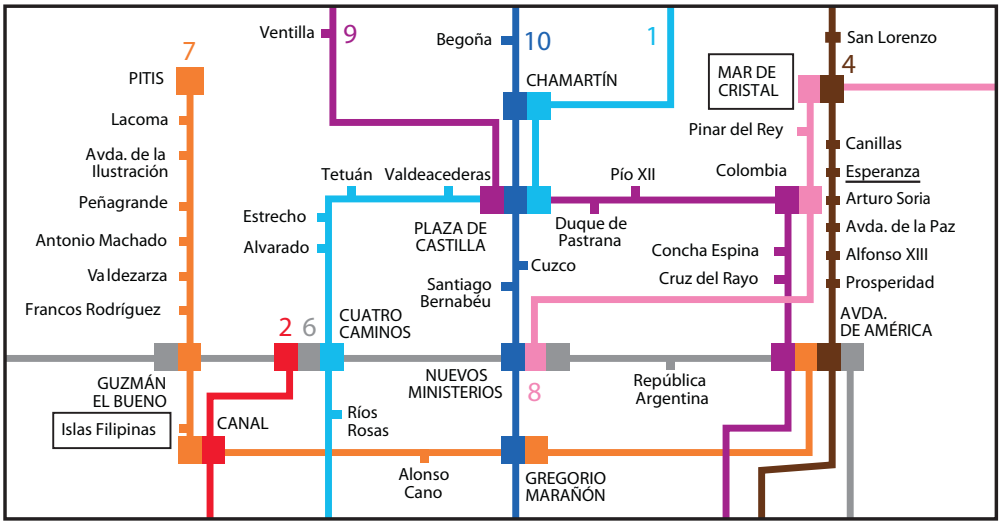


Figura 4.28: Extracto de la red de metro usado en el ejemplo. Las estaciones de inicio y destino están recuadradas, y la estación por la que hay que pasar aparece subrayada.

4.2. Monitorizador de rutas

El seguimiento de rutas no se ha contemplado en los objetivos del proyecto, pero aún así se ha planteado a grandes rasgos cómo debería hacerse.

Este módulo es el encargado de monitorizar la ruta que va siguiendo el usuario para comprobar si va por el camino correcto.

Por el momento no está implementado en WAI-Routes, pero la idea consistiría en recibir periódicamente datos de la localización del usuario (mediante el GPS del terminal) para estimar si la localización actual del usuario se desvía de la localización en la que está la ruta. Si el usuario se confundiera de camino sería avisado de alguna forma (mostrando un mensaje por pantalla, haciendo vibrar el terminal, utilizando una alarma, etc) para que pudiera corregir su trayectoria y evitar así que se perdiera.

4.3. Implementación del servidor

Para implementar el servidor se han utilizado cuatro paquetes: Servlet, Grafo, Filtros y Dijkstra. En las figuras que se muestran en este apartado (diagramas UML) cada clase perteneciente a un paquete tiene su mismo color, y cada paquete tiene un color distinto. Así, el paquete Servlet es de color marrón, el paquete Grafo es de color rojo, el paquete Filtros es de color naranja y por último el paquete Dijkstra es de color morado.

El paquete **Servlet** (véase Figura 4.29) es el servidor en si, el cual se encarga de recibir la información enviada por el cliente, calcular la ruta y responder al cliente. Consta de una sola clase, *CalculaRutaServlet*, que a su vez tiene tres métodos. Cuando el servidor se arranca por primera vez se ejecuta el método *init*, que se encarga de crear un objeto de la clase *CalculaRuta* (perteneciente al paquete Dijkstra, véase Figura 4.32) que creará el grafo de la red de transporte público. El método *doPost* espera recibir el comando *post* por parte del cliente. Cuando esto ocurre se ejecuta el método *processRequest*, el cual recoge el mensaje enviado por el cliente, decodifica el objeto *Consulta* recibido mediante un objeto *JSONDecodificaConsulta* y se lo pasa a *CalculaRuta*. Esta clase obtiene la mejor ruta y devuelve un objeto *Ruta* al método *processRequeset*, el cual codifica dicho objeto usando *JSONCodificaRuta* y envía un mensaje al cliente.

Para saber más sobre las clases *Ruta*, *Consulta*, *JSONCodificaRuta* y *JSONDecodificaConsulta*, pertenecientes al paquete Datos, véase el apartado 5.4.1 del capítulo 5. Para profundizar sobre la codificación y decodificación usando objetos *JSON* véase el apartado 5.2 del capítulo 5.

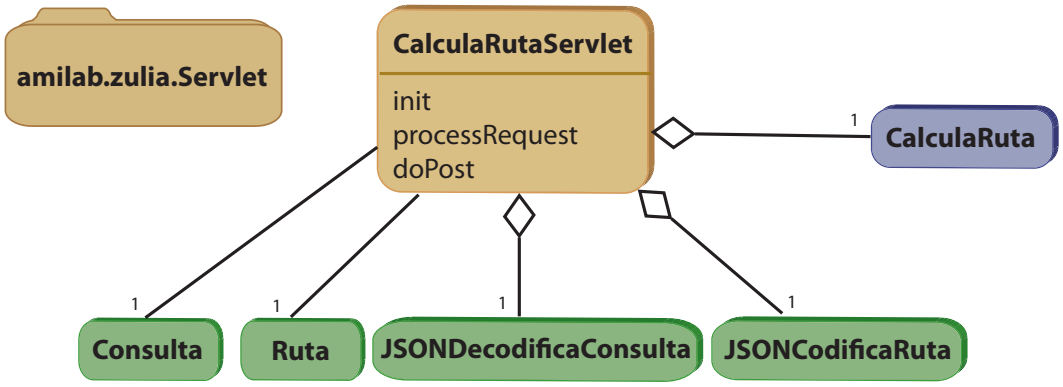


Figura 4.29: Diagrama UML del paquete Servlet (en color marrón), el cual realiza las funciones del servidor. Consta de una sola clase, CalculaRutaServlet, y emplea cuatro clases del paquete Datos (en color verde) y una del paquete Dijkstra (en color morado).

El paquete **Grafo** (véase Figura 4.30) es el encargado de crear el grafo de la red de transporte público, así como calcular los posibles nodos de inicio y destino. Consta de ocho clases: *Coordenada*, *Nodo*, *Linea*, *Transbordo*, *Coste*, *Union*, *Grafo* y *PosiblesIniFin*. La clase *Coordenada* guarda la información de una coordenada, es decir su latitud y longitud. La clase *Nodo* guarda toda la información necesaria sobre una estación (véase apartado 3.1.1 del capítulo 3), así como el nodo desde el cual se llega al nodo actual. Cada *Nodo* desciende de *Hito* (véase el apartado 5.4.1) y tiene una *Coordenada*. *Union* guarda la información entre la unión de dos nodos, incluidos ambos nodos. *Linea* (véase apartado 3.1.2 del capítulo 3) y *Transbordo* (véase apartado 3.1.3 del capítulo 3) heredan de *Union*, añadiendo cada uno sus relativos costes. Cada *Transbordo* usa un *Coste*, que a su vez tiene tres tipos de costes (de transbordo, de complejidad y del tramo que hay que andar), tal y como se vio en el apartado 3.1.4 del capítulo 3.

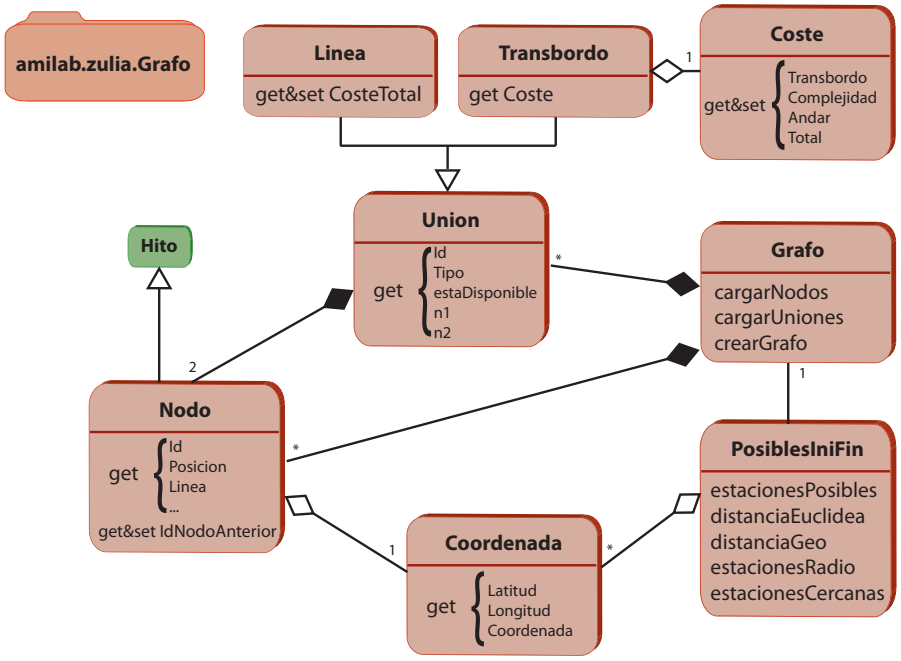


Figura 4.30: Diagrama UML del Paquete Grafo, el cual crea el grafo de la red de transporte público y los posibles nodos de inicio y destino. Consta de ocho clases, Linea, Transbordo, Coste, Union, Nodo, Coordenada, Grafo y PosiblesIniFin.

La clase *Grafo*, mediante los métodos *cargarNodos* y *cargarUniones*, lee la información de los nodos y las uniones de unos ficheros de texto y crean los objetos *Nodo* y *Union* necesarios. Después utiliza el método *crearGrafo* para formar el grafo de la red de transporte público. Por último, *PosiblesIniFin* calcula los posibles nodos de inicio y destino que se pueden tener en cuenta para calcular la ruta, usando cinco métodos (véase el apartado 4.1.2). El método *estacionesPosibles* obtiene todos los nodos dada una serie de

estaciones (se recuerda que por ejemplo la estación de Callao tiene los nodos L03-Callao y L05-Callao). El método *estacionesRadio* obtiene todas las estaciones que están dentro de un radio dado empleando el método *distaciaGeo*, que calcula la distancia geodésica entre dos coordenadas. Por otro lado, el método *estacionesCercanas* obtiene las "numEstaciones" más cercanas usando el método *distanciaEuclídea*, que calcula la distancia en línea recta de dos coordenadas.

El paquete **Filtros** (véase Figura 4.31) es el encargado de crear los filtros y realizar la acción en si de filtrar. Para ello usa veinticuatro clases, entre las cuales están los diez filtros de eliminación y los diez de modificación, *FiltroEliminar*, *FiltroModificar*, *ControlModificar* y *RealizarFiltrado* (véase los apartados 4.1.3, 4.1.4 y 4.1.5). Todos los filtros de eliminación (tales como *FEAbono* y *FETransbordos*) heredan de *FiltroEliminar*. A su vez, todos los filtros de modificación (tales como *FMEstaciones* y *FMMovilidad*) descienden de *FiltroModificar*. Además, cada filtro de modificación usa uno o varios objetos de la clase *ControlModificar*, que guardan la información sobre los filtros de modificación usados y sus costes modificados.

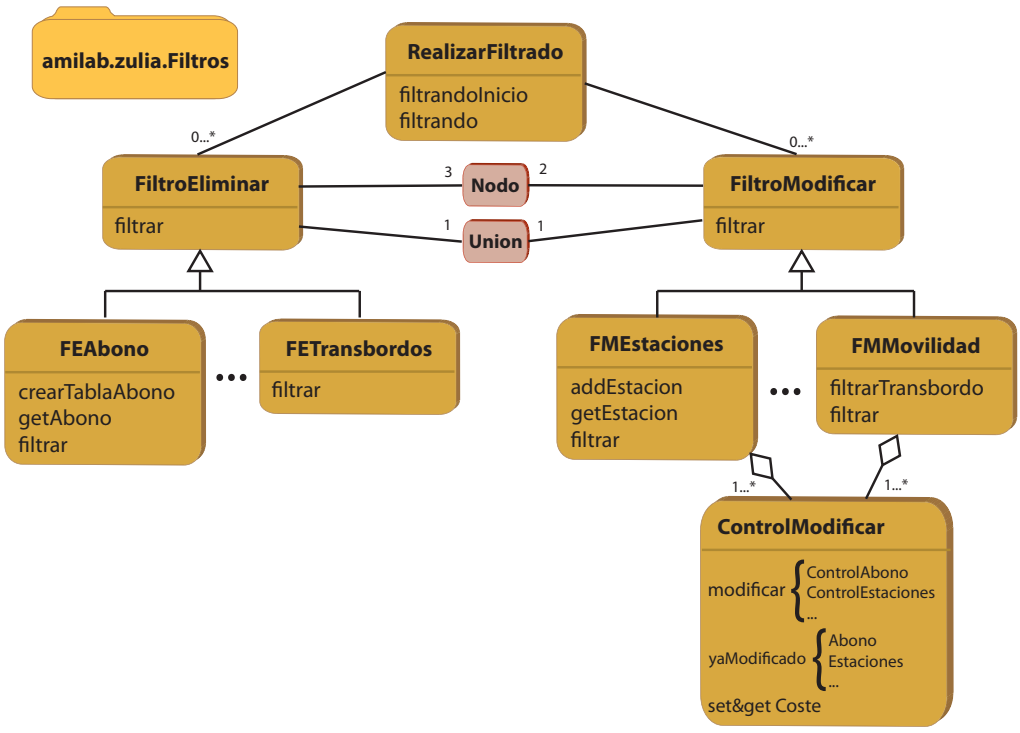


Figura 4.31: Diagrama UML del paquete Filtros, el cual define los diferentes tipos de filtros y realiza la opción de filtrado. Consta de diversas clases, todas ellas en color naranja. Utiliza dos clases del paquete Grafo, representadas en color rojo.

Cada *FiltroEliminar* usa tres objetos *Nodo* y uno de *Union* que pertenecen al paquete Grafo (véase Figura 4.30), mientras que cada *FiltroModificar* usa dos de *Nodo* y uno de *Union*. La clase *RealizarFiltrado* se encargar de juntar todos los *FiltroEliminar* y todos los *FiltroModificar* (si hubiera) y ejecutarlos usando el método *filtrar*.

El paquete **Dijkstra** (véase Figura 4.32) se encarga de calcular la ruta, para lo cual emplea dos clases, *CalculaRuta* y *Dijkstra-F*. Al crear un objeto de *CalculaRuta* se crea automáticamente el grado de la red de transporte público, usando la clase *Grafo*. Después se usa el método *getPosiblesIniFin* para calcular los posibles nodos de inicio y destino a partir del objeto *Consulta* que le ha suministrado la clase *CalculaRutaServlet*, tal y como se explicó anteriormente. Para ello se usa un objeto de la clase *PosiblesIniFin*. A continuación se emplea el método *getFiltros* para obtener los filtros de eliminación y modificación (si los hubiera), de nuevo a partir de *Consulta*. Por último, se obtiene la ruta (creando un objeto *Ruta*) usando el método *calcularRuta* (y empleando *pasarPorEstaciones* si es necesario). Para ello, este método crea un objeto de la clase *Dijkstra-F*, donde la función *algoritmo* aplica Dijkstra-F mediante los métodos *extraerMin* y *actualizarVecinos* (véase apartado 4.1.1). Para usar los filtros, el método *actualizarVecinos* emplea un objeto *RealizarFiltrado*.

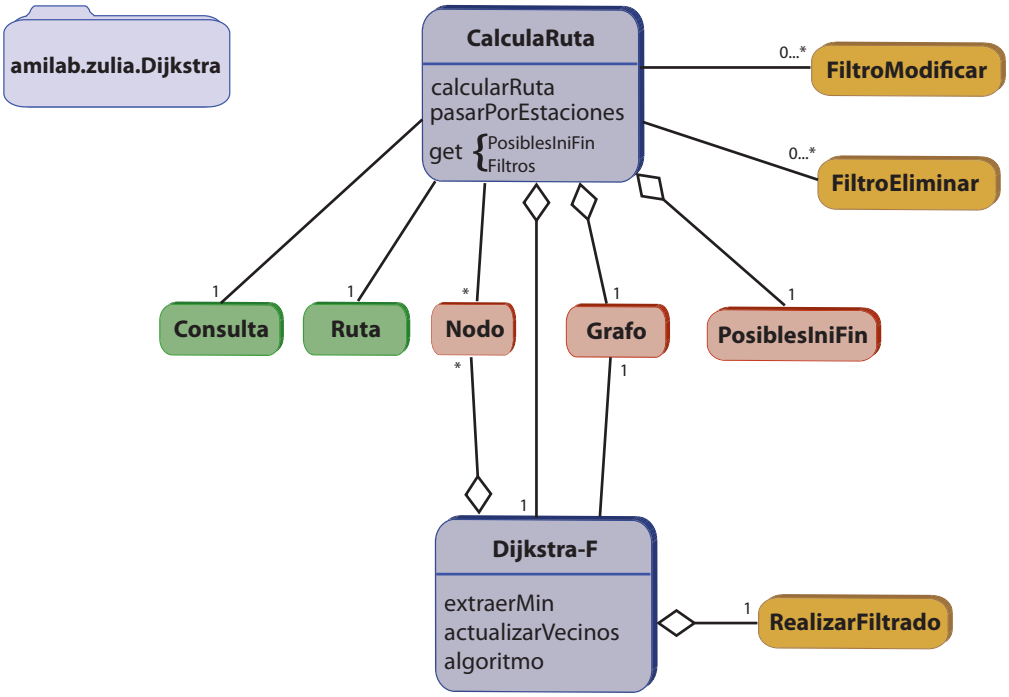


Figura 4.32: Diagrama UML del Paquete Dijkstra, el cual se encarga de calcular la ruta empleando el algoritmo Dijkstra-F. Consta de dos clases, CalculaRuta y Dijkstra-F. Emplea tres clases del paquete Filtros (en color naranja), dos clases del paquete Datos (en color verde) y tres clases del paquete Grafo (en color rojo).

5 Cliente ligero

En el apartado 3.1.4 del capítulo 3 se explicó que tanto las líneas como los transbordos tienen un coste asociado. La idea es que estos costes varíen dinámicamente según los datos de la empresa de transportes, por lo que es más útil disponer de la información en un servidor y no en el terminal móvil. Así, cuando uno de esos costes varíe bastará con actualizar el grafo del servidor en vez de tener que actualizar todos los terminales.

Por este motivo, la funcionalidad del cliente se reduce a obtener los datos del usuario, enviarlos al servidor, recibir la información sobre la ruta calculada por el servidor y mostrarla al usuario por la pantalla del terminal.

Tal y como se vio en el capítulo 3 el cliente consta de cinco bloques, la ***Entrada de Usuario***, el ***Modelo de Usuario***, el ***Proveedor de Localización***, el ***Geocodificador*** y la ***Visualización***. En este capítulo se van a explicar la *Entrada de Usuario* y el *Modelo de Usuario*. Además se detallará cómo se comunican el cliente y el servidor para intercambiar información. El *Proveedor de Localización*, el *Geocodificador* y la *Visualización* se explicarán en el capítulo 6 (Prototipo de demostración).

5.1. Entrada de usuario y modelo de usuario

El usuario debe introducir una serie de parámetros necesarios para calcular la ruta. Tal y como se ha mencionado en capítulo anterior, el usuario como mínimo debe introducir el destino de la ruta (el inicio puede ser obtenido mediante el GPS del terminal). Después tiene la posibilidad de definir una serie de opciones, como pasar por una serie de estaciones, o establecer una serie de filtros. Toda la información suministrada por el usuario (o establecida por el modelo de usuario) se guarda en un objeto de tipo *Consulta*.

Una información relevante que se almacena en el terminal es el modelo de usuario. Éste permite personalizar las consultas realizadas al servidor decidiendo distintas restricciones a la hora de calcular la ruta según las preferencias del usuario. El modelo de usuario se puede verse de dos maneras. Por un lado, el usuario puede definir un perfil por defecto de forma que siempre que use la aplicación automáticamente se creen una serie de filtros. Por otro lado, el modelo de usuario puede ser modificado en cada consulta, pudiendo elegir los filtros que desee ya sea para añadir nuevos o para sobrescribir los que se eligen por defecto.

5.2. JSON, codificación y decodificación de Consulta y Ruta

Tal y como se ha explicado en el apartado anterior, toda la información recopilada en el cliente queda almacenada en un objeto *Consulta*. Este objeto contiene todo lo necesario para calcular la ruta: el inicio, el destino, las opciones y los filtros a emplear. El objeto *Consulta* se codifica en un objeto *JSON*, el cual será enviado al servidor. El servidor decodifica el objeto *JSON* en un objeto *Consulta* y calcula la ruta. La información de la ruta obtenida se almacena en un objeto *Ruta*, que también será codificado en un objeto *JSON* para enviarlo al cliente. Por último, cuando el cliente recibe el objeto *JSON* lo decodifica en un objeto *Ruta*. Para más información sobre el intercambio de datos entre el cliente y el servidor véase el apartado 5.4.

Se ha escogido *JSON* (*JavaScript Object Notation*) [30] para el intercambio de información entre el cliente y el servidor porque es un formato ligero de intercambio de datos, sencillo de leer y escribir para los programadores y sencillo de interpretar y generar para las máquinas. *JSON* es un formato de texto completamente independiente del lenguaje de programación, pero que utiliza convenciones conocidas por los programadores. La principal ventaja respecto a *XML* es que *JSON* es más sencillo de *parsear*, es decir, es más fácil leer o interpretar un documento *JSON* que un documento *XML*.

JSON está constituido por dos estructuras:

JSONObject. Es una colección de pares de nombre/valor y funciona de manera similar a una *tabla hash* definida en el lenguaje de programación Java.

JSONArray. Es una lista ordenada de valores y funciona de manera similar a un *arraylist* en el lenguaje de programación Java.

Un objeto *JSON* siempre comienza con una llave { de apertura y termina con una llave } de cierre. El nombre del objeto es una cadena de caracteres (*string*). Cada nombre viene seguido por dos puntos, a continuación de los cuales se añade el valor. Los pares nombre/valor vienen separados por una coma.

Un *array JSON* comienza con un corchete [de apertura y termina con un corchete] de cierre. Los valores vienen separados por comas.

Los valores pueden ser una cadena de caracteres (*string*), un número, un *array*, un *booleano* (verdadero o falso), *null* u otro objeto.

En la Figura 5.1 y en la Figura 5.4 se pueden ver dos ejemplos del formato *JSON*.

Como se ha explicado al principio de este apartado, el cliente codifica el objeto *Consulta* en un objeto *JSON* para enviárselo al servidor. Para comprender mejor en qué consiste dicha codificación y cómo se emplea en ella *JSON*, véase el ejemplo de la Figura 5.1. En ella viene definido el nodo de inicio (Alvarado), la coordenada de destino (su longitud y latitud en grados decimales), el número de estaciones cercanas (dos en este caso) a tener en cuenta desde la coordenada de destino, un filtro de eliminación (elimina las estaciones de Cuzco y Sol) y dos filtros de modificación (uno para evitar la línea de metro L06 y otro para evitar las estaciones no adaptadas totalmente a personas con movilidad reducida).

```
{
 "inicio":{
       "valor": "ALVARADO",
       "tipo": "estación"},

 "destino":{
       "valor":{
           "longitud": -3.696406,
           "latitud": 40.438261},
       "tipo": "coordenadas"},

 "numEstacionesCercanas": 2,

 "filtros":{

       "filtrosEliminar":{
           "feEstaciones": ["CUZCO","SOL"]},

       "filtrosModificar":{
           "fmLineaMetro": ["L06"],
           "fmMovilidad": "TOTAL"}
       }
}
```

Figura 5.1: Ejemplo de codificación de un objeto *Consulta* a un objeto *JSON*.

En la Figura 5.2 se puede observar el esquema del ejemplo anterior. Aquí se puede apreciar como se forma un objeto *JSON* a partir de varios *JSONObject* y *JSONArray*.

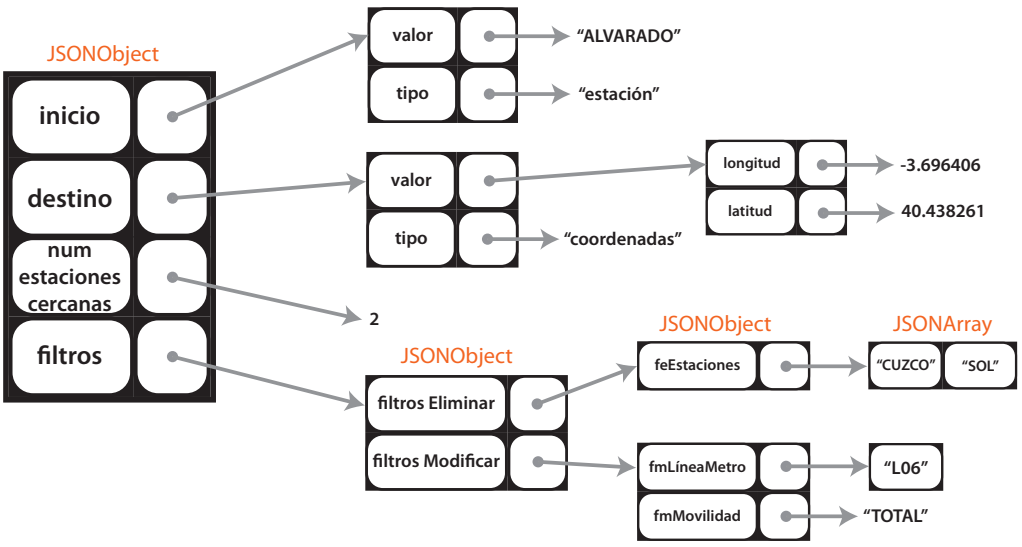


Figura 5.2: Esquema de un objeto *JSON* a partir de los datos de un objeto *Consulta*.

```
JSONObject sms = new JSONObject();

JSONObject ini = new JSONObject();
ini.put("tipo", "estación");
ini.put("valor", "ALVARADO");

sms.put("inicio", ini);

JSONObject coord = new JSONObject();
coord.put("latitud", -3.696406);
coord.put("longitud", 40.438261);

JSONObject fin = new JSONObject();
fin.put("tipo", "coordenadas");
fin.put("valor", coord);

sms.put("destino", fin);

JSONArray estaciones = new JSONArray();
estaciones.put("CUZCO");
estaciones.put("SOL");

JSONObject fe = new JSONObject();
fe.put("feEstaciones", estaciones);

JSONObject f = new JSONObject();
f.put("filtrosEliminar", fe);

sms.put("filtros", f);
```

Figura 5.3: Ejemplo de programación con *JSON* para codificar un objeto *Consulta*.

Por último, en la Figura 5.3 se presenta un ejemplo de cómo se puede programar la codificación de un objeto *Consulta* a *JSON*. Aquí se describe un inicio que es una estación, un destino que es una coordenada y un filtro de eliminación de estaciones. En esta figura se puede apreciar claramente cómo se crea el objeto *JSON* principal, *sms*, a partir de *JSONObject* y *JSONArray*.

De igual forma, el servidor codifica un objeto *Ruta* a un objeto *JSON* para enviarlo al cliente. En la Figura 5.4 se muestra un ejemplo de una codificación de este tipo. Aquí se especifica que el estado de la ruta es "ok", es decir que no hubo ningún problema a la hora de calcularla. También indica que la distancia a la estación de inicio es de 24.87 metros. Por último viene una lista de hitos (véase el apartado 5.4.1) o nodos especificando la ruta, con sus respectivos valores. En este ejemplo, a modo de demostración sólo se expone un hito y un nodo, pero en una ruta real habría varios hitos y/o nodos.

```
{
 "estado": "OK",

 "distanciaEstaciónIni": 24.87,

 "ruta":{[

          { "tipo": "hito",
            "valor":{
                 "longitud": -3.696406,
                 "latitud": 40.438261}
          },

          ...

          { "tipo": "nodo",
            "valor":{
                 "id": "L01-SOL",
                 "línea": "L01",
                 "estación": "SOL",
                 "posición": 15,
                 "tipo": "METRO",
                 "esFinal": "NO",
                 "estáDisponible": "SI",
                 "esTransbordo": "SI",
                 "abono": "A",
                 "movilidad": "PARCIAL",
                 "horarioRestringido": "NO",
                 ...}
          },

          ...

        ]}
}
```

Figura 5.4: Ejemplo de codificación de un objeto *Ruta* a un objeto *JSON*.

En la Figura 5.5 está representado el esquema de la codificación del ejemplo anterior. De nuevo se puede apreciar la estructura que se forma a partir de varios *JSONObject* y *JSONArray*.

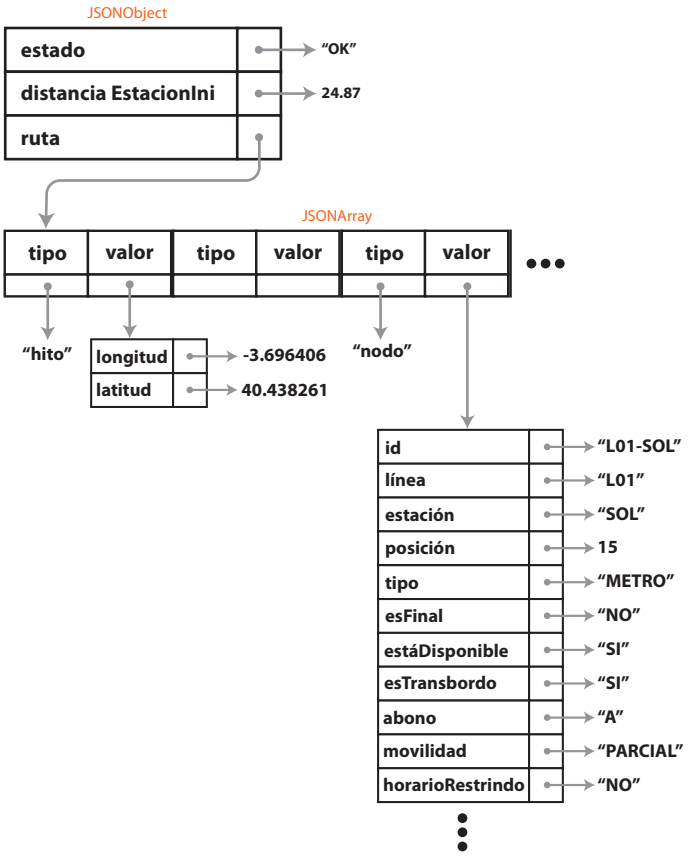


Figura 5.5: Esquema de un objeto *JSON* a partir de los datos de un objeto *Ruta*.

5.2.1. Formato de Consulta y Ruta para JSON

Tal y como se ha mencionado antes, un objeto *JSON* es un par nombre/valor. En este apartado se van a explicar los posibles nombres y valores que pueden tener los objetos *JSON* utilizados para codificar *Consulta* y *Ruta*. Con esta información se podrá realizar el intercambio de información entre el servidor y cualquier tipo de cliente. Nótese que para los nombres nunca se emplean tildes.

Los posibles nombres que puede tener un objeto de tipo ***Consulta*** son:

- *inicio*
- *destino*
- *radioEstacionesCercanas*
- *numEstacionesCercanas*
- *pasarEstaciones*
- *filtros*

Como mínimo, una *Consulta* debe tener obligatoriamente los nombres *inicio* y *destino*.

Tanto *inicio* como *destino* tienen como valor un objeto *JSON* con dos nombres, *tipo* y *valor*. Si el *tipo* es "estacion" el valor del campo *valor* es una cadena de caracteres con el nombre de la estación. Si el *tipo* es "coordenada" el valor del campo *valor* es un objeto JSON con dos nombres, *latitud* y *longitud*. Los valores de ambos son un número decimal que corresponderá con la latitud y la longitud respectivamente expresadas en grados decimales. Si la coordenada está situada en la latitud sur o longitud oeste, el signo del número decimal será negativo.

El nombre *radioEstacionesCercanas* tiene como valor un número entero que expresará el radio en metros.

El nombre *numEstacionesCercanas* tiene como valor un número entero.

El nombre *pasarEstaciones* tienen como valor un *JSONArray* de cadenas de caracteres (*strings*). Cada cadena de caracteres debe contener el nombre de una estación.

Por último, *filtros* tiene como valor un objeto *JSON* con dos posibles nombres, *filtrosEliminar* y *filtrosModificar*. A su vez, el nombre *filtrosEliminar* tiene como valor un objeto *JSON* con diez posibles nombres, uno por cada tipo de filtro. A continuación se muestra una lista con el nombre del filtro seguido de sus posibles valores:

• *feAbono* → “A”, “B1”, “B2”, “B3”, “C1”, “C2”.

• *feEstaciones* → *JSONArray* de cadena de caracteres. Cada cadena de caracteres debe contener el nombre de una estación.

• *feLineaMetro* → *JSONArray* de cadena de caracteres. Cada cadena de caracteres debe contener el identificador de una línea de metro (“L01”, “L02”, etc).

• *feLineaMetroLigero* → *JSONArray* de cadena de caracteres. Cada cadena de caracteres debe contener el identificador de una línea de metro ligero (“ML1”, “ML2”, etc).

• *feLineaRenfe* → *JSONArray* de cadena de caracteres. Cada cadena de caracteres debe contener el identificador de una línea ce cercanías (“C01”, “C02”, etc).

• *feMetro*, *feMetroLigero*, *feRenfe* → “OK”.

• *feMovilidad* → “PARCIAL”, “TOTAL”.

• *feTransbordos* → “OK”.

Para *filtrosModificar* se hace de igual forma que para *filtrosEliminar* solo que los posibles filtros son *fmAbono*, *fmEstaciones*, *fmLineaMetro*, *fmLineaMetroLigero*, *fmLineaMetroRenfe*, *fmMetro*, *fmMetroLigero*, *fmRenfe*, *fmMovilidad* y *fmTransbordos*.

Los posibles nombres de ***Ruta*** son:

• *estado*

• *estadoRadio*

• *distanciaEstacionIni*

• *distanciaEstacionFin*

• *ruta*

Como mínimo una ruta debe de tener obligatoriamente los nombres *estado* y *ruta*.

Tanto *estado* como *estadoRadio* tienen como valor una cadena de caracteres, que puede ser o bien "OK" o bien "ERROR".

Tanto *distanciaEstacionIni* como *distanciaEstacionFin* tienen como valor un número decimal que expresará la distancia en metros.

El nombre *ruta* tiene como valor un *JSONArray* de objetos *JSON*. Cada objeto JSON tiene dos nombres, *tipo* y *valor*. Si el *tipo* es "hito" hace referencia a un objeto *Hito* (véase apartado 5.4.1) y su *valor* es un objeto *JSON* con dos nombres, *longitud* y *latitud*, cuyos valores son un número decimal.

Si el *tipo* es "nodo" hace referencia a un objeto *Nodo* (véase el apartado 4.3 del capítulo 4). Tiene dieciocho nombres, correspondientes a las características que definen cada estación (véase apartado 3.1.1 del capítulo 3). A continuación se muestra una lista con los nombres que describen una estación y sus posibles valores:

- *id* → cadena de caracteres que forman la unión de *linea* y *estacion* con un guión
- *linea* → cadena de caracteres con el identificador de la línea ("L01", "C04", etc).
- *estacion* → cadena de caracteres con el nombre de la estación.
- *posicion* → número entero de la posición que ocupa la estación dentro de la línea.
- *esFinal*, *estaDisponible,*, *esTransbordo* → "SI", "NO".
- *horarioRestringido, cambioTren* → "SI", "NO".
- *abono* → "A", "B1", "B2", "B3", "C1", "C2".
- *movilidad* → "PARCIAL", "TOTAL", "NULA".
- *busInterurbano*, *busInterregional, trenLargoRecorrido* → "SI", "NO".
- *intercambiado* y *aeropuerto* → "SI", "NO".
- *coordenada* → igual que cuando *inicio* (en ***Consulta***) es una coordenada.

5.3. Implementación del cliente

Para implementar el cliente se ha utilizado un único paquete: ClienteRuta.

El paquete **ClienteRuta** (véase Figura 5.6) tiene una sola clase, *ClienteRuta*, que se encarga de enviar un mensaje al servidor con los datos del usuario y de recibir la información enviada por el servidor con la ruta calculada. Para ello emplea el método *doPost*, el cual establece la conexión con el servidor. A continuación codifica los datos del usuario almacenados en un objeto *Consulta* usando *JSONCodificaConsulta*. Después crea el mensaje y lo envía al servidor mediante el comando *post*. Por último decodifica la repuesta del servidor usando *JSONDecodificaRuta* para obtener un objeto *Ruta*.

Para saber más sobre las clases *Ruta*, *Consulta*, *JSONCodificaRuta* y *JSONDecodificaConsulta*, véase el apartado 5.4.1. La codificación y la decodificación empleando objetos *JSON* se ha explicado en el apartado 5.2.

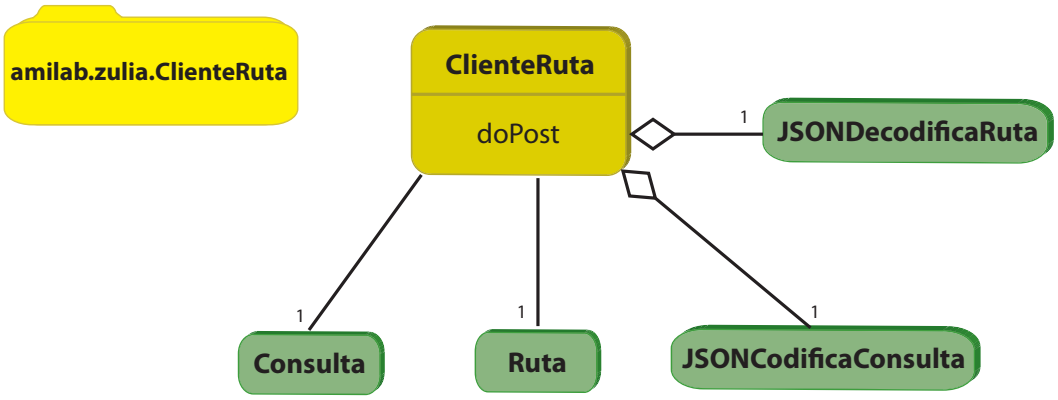


Figura 5.6: Diagrama UML del paquete ClienteRuta, el cual realiza las funciones del cliente. Consta de una sola clase, ClienteRuta (en color amarillo). Utiliza cuatro clases del paquete Datos (en color verde).

5.4. Conexión Cliente-Servidor

En la Figura 5.7 se puede observar el esquema general del funcionamiento de WAI-Routes, cuya arquitectura empleada es Cliente-Servidor. En color rojo se muestran las clases principales: *PruebaIU* y *ClienteRuta* para el cliente; *CalculaRutaServlet* y *CalculaRuta* para el servidor. En color azul se especifican las acciones realizadas por

cada clase, y los rombos de color morado equivalen a las comprobaciones. Por último, las flechas discontinuas en color gris representan el intercambio de datos entre las diversas clases.

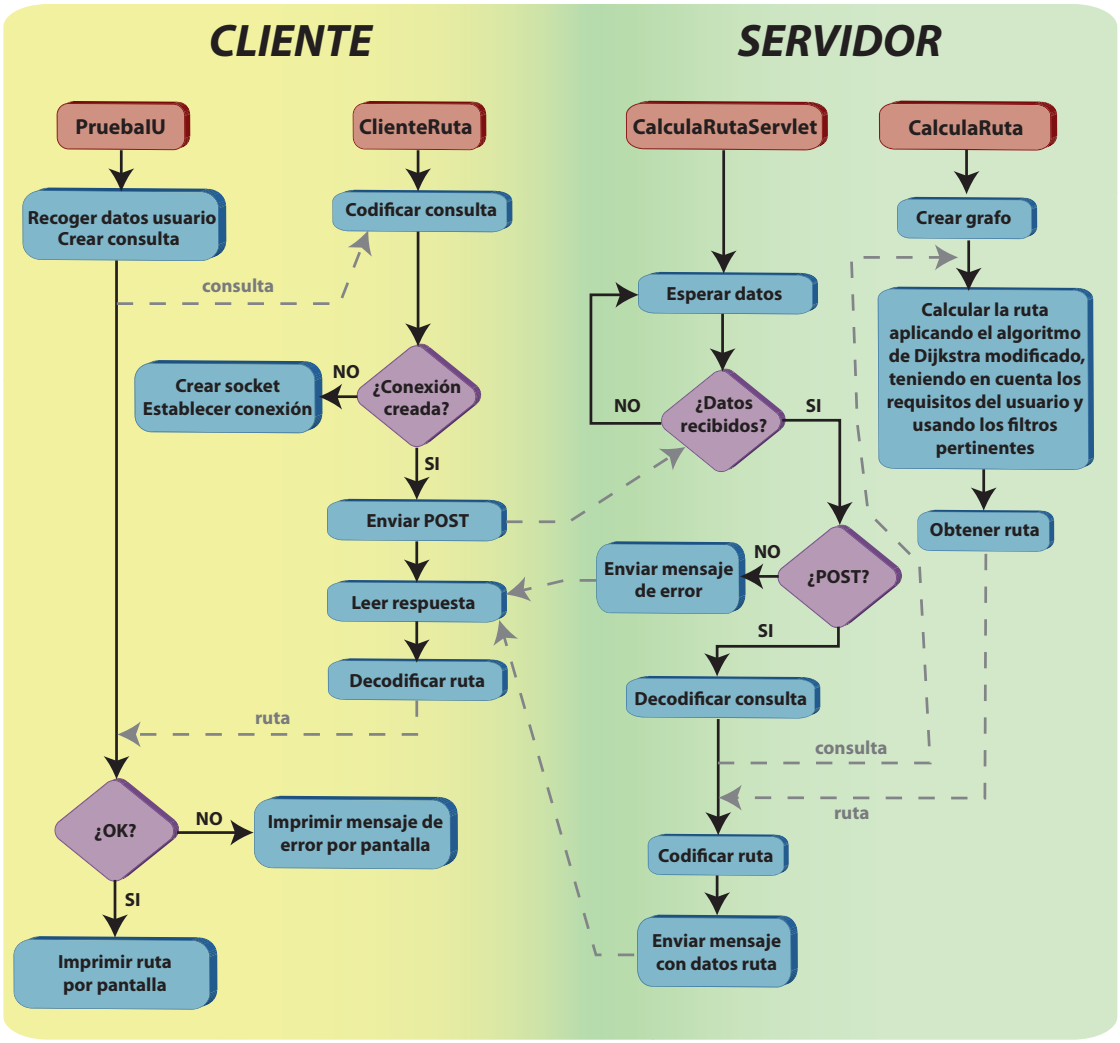


Figura 5.7: Esquema general del funcionamiento de WAI-Routes, basado en la arquitectura Cliente-Servidor.

La información que el usuario va introduciendo en cada ventana de la interfaz de usuario se guarda en un objeto *InfoConsulta*. En *PruebaIU*, con toda la información recopilada en *InfoConsulta* se crea un objeto *Consulta*. La implementación de la *Visualización* del cliente y el ejemplo de interfaz de usuario empleada (usando *InfoConsulta* y *PruebaIU*) será explicado en el apartado 6.3 del capítulo 6.

En *InfoConsulta* sólo se almacena la información suministrada por el usuario, y su contenido se puede modificar fácilmente para añadir y eliminar datos. Por ejemplo, en *InfoConsulta* se guardaría la información de que el usuario quiere añadir un filtro para eli-

minar la estación de Canal. En *Consulta* se almacenan los objetos creados con la información suministrada por *InfoConsulta* (también se almacena información que no puede ser modificada, como lo referente al inicio o al destino), y no existe la posibilidad de eliminar datos. Por ejemplo, en *Consulta* se almacenaría el filtro FE Estación (Canal).

La clase *PruebaIU* envía a la clase *ClienteRuta* el objeto *Consulta*. A continuación *Consulta* es codificado en un objeto *JSON* (véase apartado 5.2), que será lo que el cliente envíe al servidor.

Lo primero que hace el servidor, en la clase *CalculaRutaServlet*, es crear un objeto de tipo *CalculaRuta*, el cual se encarga de crear el grafo de transporte público de la Comunidad de Madrid con los ficheros que contienen la información de las estaciones, líneas y transbordos, tal y como se explicó en el apartado 3.1 del capítulo 3. Esto sólo se hace una vez, cuando se arranca el servidor. Cuando *CalculaRutaServlet* recibe los datos de *ClienteRuta*, si todo es correcto decodifica el objeto *JSON* para obtener el objeto *Consulta*. Este objeto *Consulta*, a su vez, es enviado a la clase *CalculaRuta*.

La clase *CalculaRuta* es la encargada de calcular la ruta tal y como se explicó en el capítulo 4, usando el algoritmo de Dijkstra-F y teniendo en cuenta los requisitos del usuario definidos en el objeto *Consulta*. Cuando se ha calculado el camino a seguir por el usuario, se crea un objeto *Ruta* que almacena un conjunto de hitos (cualquier información relativa al camino, de momento sólo se han considerado coordenadas) y/o nodos que especifican la ruta. También se almacena en *Ruta* una serie de información extra que puede ser útil para el usuario, como por ejemplo la distancia a la estación más cercana o si hubo algún problema a la hora de calcular la ruta. Este objeto *Ruta* será enviado a *CalculaRutaServlet*.

Cuando *CalculaRutaServlet* recibe el objeto *Ruta* de *CalculaRuta*, lo codifica en un objeto *JSON* que será lo que se envíe del servidor al cliente.

Cuando *ClienteRuta* recibe la respuesta de *CalculaRutaServlet*, decodifica el objeto *JSON* para obtener el objeto *Ruta*. A continuación, enviará este objeto a *PruebaIU*.

Finalmente, cuando *PruebaIU* recibe el objeto *Ruta*, si no hubo ningún problema imprime en la pantalla del terminal el camino que debe seguir el usuario, según los datos obtenidos por *Ruta*.

5.4.1. Implementación del intercambio de datos entre cliente y servidor

Para programar el intercambio de datos entre el cliente y el servidor se ha usado un paquete llamado Datos. También se han usado los paquetes Servlet (véase el apartado 4.3 del capítulo 4) y ClienteRuta (véase el apartado 5.3), que ya han sido explicados con anterioridad.

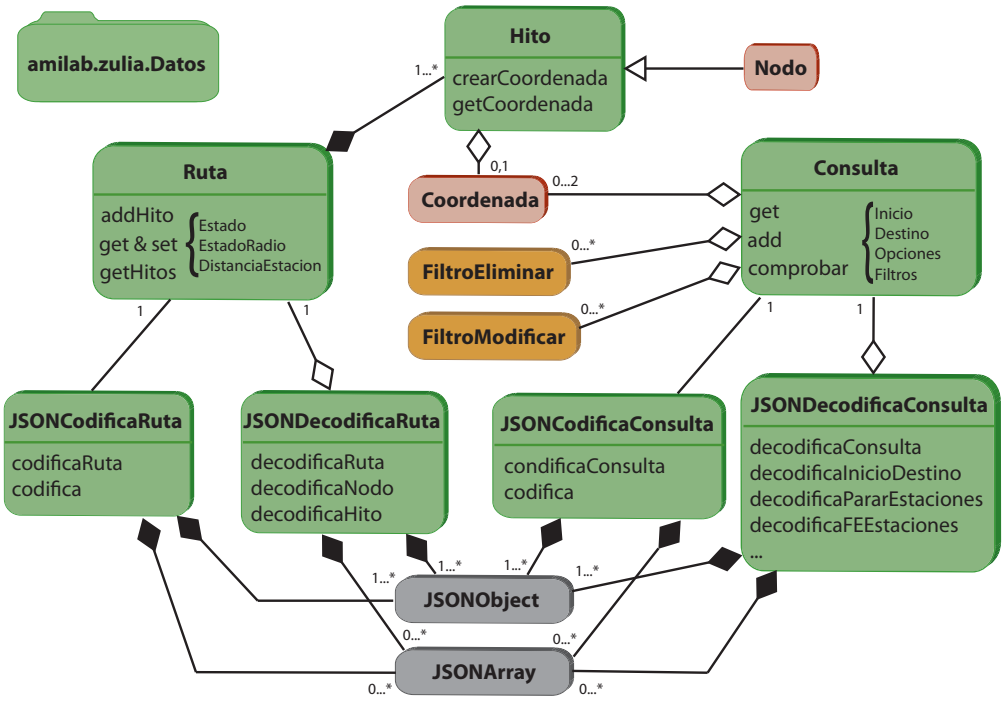


Figura 5.8: Diagrama UML del paquete Datos, el cual se encarga de almacenar, codificar y decodificar la información que se transmite del cliente al servidor y viceversa. Consta de siete clases (en color verde) y utiliza una clase del paquete Grafo (en color rojo), dos clases del paquete Filtros (en color naranja) y dos clases del paquete org.json (en color gris).

El paquete **Datos** (véase Figura 5.8) se encarga de almacenar, codificar y decodificar la información que se envía del cliente al servidor y viceversa. Consta de siete clases: *Hito*, *Ruta*, *JSONCodificaRuta*, *JSONDecodificaRuta*, *Consulta*, *JSONCodificaConsulta* y *JSONDecodificaConsulta*. La clase *Hito* almacena un punto de la ruta, que puede ser un coordenadas o un nodo (también podría ser cualquier lugar u objeto que sirva como referencia). *Ruta* almacena todos los hitos que forman la ruta, así como información adicional importante para el usuario, como por ejemplo la distancia a la estación más cercana o la imposibilidad de calcular una ruta. *Consulta* almacena la información introducida por el usuario para calcular la ruta, como el inicio, el destino, las opciones para calcular la ruta (como por ejemplo pasar por alguna estación

determinada) y los filtros que se deben emplear. *JSONCodificaRuta* recibe un objeto *Ruta* y lo codifica en un objeto *JSON*, que luego convierte en una cadena de caracteres (*string*). *JSONCodificaConsulta* es similar sólo que utilizando un objeto *Consulta*. *JSONDecodificaRuta* recibe una cadena de caracteres que, pasado a un objeto *JSON* se decodifica para obtener un objeto *Ruta*. De nuevo *JSONDecodificaConsulta* es similar sólo que aplicado a un objeto *Consulta*.

El uso de *JSON* en la codificación y decodificación se explicó detenidamente en el apartado 5.2.

6 Prototipo de demostración

El terminal móvil empleado en las pruebas de WAI-Routes (el HTC Dream) utiliza el sistema operativo Android, aunque se podría haber usado cualquier otro. Se ha escogido este terminal por varios motivos: Android está basado en el núcleo de Linux, lo que implica licencia de software libre y fuente abierta; permite a los desarrolladores controlar los recursos del dispositivo por medio de bibliotecas desarrolladas o adaptadas por Google (como por ejemplo el uso de mapas de Google maps); el terminal puede permanecer conectado a Internet y tiene GPS integrado.

Tal y como se mencionó en el capítulo 3, el cliente consta de cinco módulos, la *Entrada de Usuario*, el *Modelo de Usuario*, el *Proveedor de Localización*, el *Geocodificador*, y la *Visualización*.

En este capítulo se va a profundizar en las características y el funcionamiento de Android (apartado 6.1). En segundo lugar se va a explicar el *Proveedor de Localización* y el *Geocodificador*, ambos aplicados a un cliente con el sistema operativo Android (apartado 6.2). Por último, se va a comentar la *Visualización* mediante un ejemplo de una interfaz (apartado 6.3).

6.1. Android

6.1.1. Introducción

Android ([31], [32], [33], [34] y [35]) es una plataforma de software libre para dispositivos móviles basada en el núcleo de Linux, que incluye un sistema operativo, una API de desarrollo en Java (capa intermedia *middleware*, un software que realiza las funciones de conversión o transformación de un sistema a otro) y una serie de aplicaciones. Una de sus características principales es que permite a los desarrolladores controlar los recursos de los dispositivos por medio de bibliotecas desarrolladas o adaptadas por Google. Para ello, el Android SDK provee las herramientas y APIs

necesarias para comenzar a desarrollar aplicaciones usando el lenguaje de programación Java.

El lanzamiento de la plataforma Android, inicialmente desarrollada por Google, se realizó el 5 de noviembre de 2007 junto con la fundación de la *Open Handset Alliance*, un consorcio de 48 compañías de hardware, software y telecomunicaciones comprometidas a la promoción de estándares abiertos para dispositivos móviles.

Actualmente, Google ha publicado la mayoría del código fuente de Android bajo la licencia de software Apache, una licencia de software libre y de fuente abierta.

6.1.2. Arquitectura

Los principales componentes de la plataforma Android vienen reflejados en la Figura 6.1, y a continuación serán explicados con más detalle.

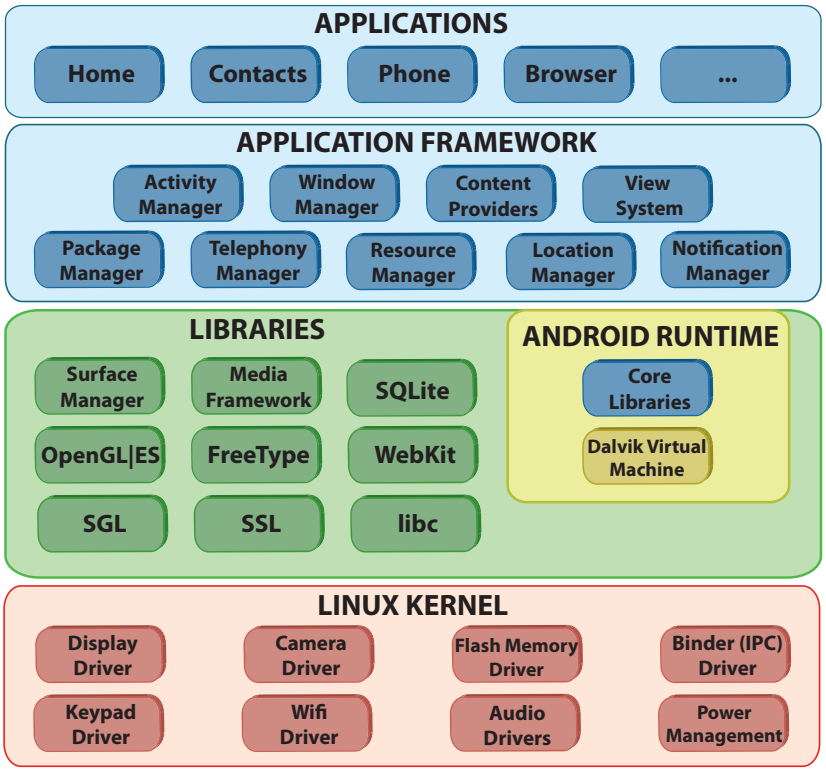


Figura 6.1: Arquitectura de Android.

Aplicaciones (*applications*). Android provee un conjunto de aplicaciones básicas por defecto que incluyen un cliente de e-mail, un programa de SMS, un calendario, mapas, un navegador, contactos y otros. Es importante matizar que todas las aplica-

ciones han sido escritas utilizando el lenguaje de programación Java.

Infraestructura de las aplicaciones (*application framework*). Los desarrolladores tienen acceso completo a las mismas APIs del *framework* usadas para construir las aplicaciones por defecto. La arquitectura está diseñada para simplificar la reutilización de componentes, de tal forma que cualquier aplicación puede publicar sus capacidades. De esta manera, cualquier otra aplicación puede hacer uso de dichas capacidades (sujeto a restricciones de seguridad del *framework* impuestas por Android). Este mismo mecanismo permite que cualquier componente pueda ser reemplazado a voluntad del usuario.

Subyacente a todas las aplicaciones existe un conjunto de servicios que incluyen lo siguiente:

Un conjunto de **vistas** (*Views*) que pueden ser usadas para construir una aplicación, incluyendo listas, matrices, casillas para entrada de texto, botones e incluso un navegador web.

Varios **proveedores de contenidos** (*Content Providers*) que suministran a las aplicaciones la capacidad para acceder a datos de otras aplicaciones, o bien compartir sus propios datos.

Un **administrador de recursos** (*Resource Manager*) que provee gráficos y archivos de disposición (*layout*). Incluye ficheros XML, PNG y JPEG.

Un **administrador de notificaciones** (*Notification Manager*) que proporciona a las aplicaciones la capacidad para desplegar mensajes de alerta personalizados en la barra de estado.

Un **administrador de actividades** (*Activity Manager*) que gestiona el ciclo de vida de las aplicaciones y provee un mecanismo de navegación entre las aplicaciones.

Para entender mejor cómo se usan los administradores citados anteriormente, así como las vistas y los proveedores de contenido, en el Anexo I se incluye una guía básica de desarrollo donde se explica paso a paso cómo se crea una aplicación para Android usando este tipo de servicios.

Por otro lado, en el siguiente apartado se explican los componentes de una aplicación, como las actividades y los proveedores de contenidos.

Bibliotecas (*libraries*). Android incluye un conjunto de librerías escritas en C/C++ usadas por varios componentes del sistema, a las que se accede a través del lenguaje de programación Java. Estas capacidades se exponen a los desarrolladores a través de la infraestructura de las aplicaciones (*framework*) de Android.

En la Tabla 6.1 se muestran las principales características de Android y las bibliotecas asociadas a ellas.

Características	Bibliotecas	Detalles
Soporte multimedia: audio, vídeo y fotografía	Media Libraries	MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF
Almacenamiento estructurado de datos	SQLite	
Motor gráfico 2D	SGL	
Gráficos 3D	3D libraries	
Administración de acceso de los subsistemas de pantalla	Surface Manager	
Navegador integrado basado en WebKit	LibWebCore	
Digitalizador de fuentes para mapas de bits y vectores	FreeType	
Redes de datos		Bluetooth, EDGE, 3G y WiFi
Soporte para telefonía GSM		
APIs de acceso a recursos		Cámara, GPS, brújula y acelerómetro
Pantalla táctil		

Tabla 6.1: Características y bibliotecas de Android.

Por último cabe destacar que Android tiene una tienda de aplicaciones similar al *App Store* del iPhone, que permite a los desarrolladores poner sus aplicaciones en el mercado, ya sean gratuitas o de pago. Su nombre es *Android Market*, y la gran diferencia con la tienda de Apple es que el *Android Market* no necesita la aprobación previa por parte de Google, lo que permite desarrollar cualquier tipo de aplicación y no solo las que Google considere oportunas.

Entorno de ejecución de Android (*Android runtime*). Como se ha dicho anteriormente, Android incluye un conjunto de librerías básicas que proveen la mayor parte de las funcionalidades disponibles en las librerías básicas del lenguaje de programación Java.

Cada aplicación Android se ejecuta dentro de su propio proceso, con su propia instancia de la Máquina Virtual Dalvik (el código de la aplicación se ejecuta de forma aislada). Dicha MV ha sido diseñada de tal forma que el dispositivo puede correr múltiples máquinas virtuales de manera relativamente eficiente. Dalkiv ejecuta archivos en el formato *Dalvik Executable* (*.dex*), el cual está optimizado para emplear memoria mínima. La MV ejecuta archivos de clases compiladas por el compilador de Java que han sido transformadas al formato *.dex* por la herramienta incluida *dx*.

Por otro lado, la MV Dalvik depende del núcleo de Linux (se describe a continuación) para todas las subyacentes funcionalidades, tales como lanzar hilos y gestionar la memoria de bajo nivel.

Núcleo de Linux (*Linux kernel*). Android depende de la versión de Linux 2.6 para los servicios básicos del sistema, tales como seguridad, administración de memoria, administración de procesos, servicios de red y *drivers*. Además, el núcleo de Linux también actúa como una capa de abstracción entre el hardware y el resto de la plataforma de software.

6.1.3. Componentes de una aplicación

Por defecto, cada aplicación se ejecuta en su propio proceso. Android inicia el proceso cuando se ejecuta cualquier código de la aplicación, y se cierra cuando ya no es necesario y los recursos del sistema son requeridos por otras aplicaciones.

La aplicación debe ser capaz de iniciar una solicitud de proceso cuando sea necesario. Por lo tanto, a diferencia de la mayoría de las aplicaciones en otros sistemas, las aplicaciones de Android no disponen de un único punto de entrada para toda la aplicación (no hay función *main*, por ejemplo). Más bien se tienen **componentes esenciales** que el sistema puede instanciar y ejecutar, según sea necesario.

Hay cuatro tipos de componentes, aunque una aplicación no requiere utilizar todos ellos:

Actividades (*activities*). Es el componente más usado en las aplicaciones para

Android, ya que cada actividad presenta una interfaz de usuario visual para cada tarea que se quiera realizar. Por ejemplo, una aplicación que se encargue de gestionar mensajes de texto puede tener una actividad que muestra una lista de contactos para enviar mensajes y una segunda actividad para escribir el mensaje a los contactos elegidos. A pesar de trabajar juntas para formar una interfaz de usuario coherente, cada actividad es independiente de la otra.

Lo común es que una aplicación contenga varias pantallas, y cada una de ellas debe ser implementada como una actividad. Es decir, la navegación de pantallas se hace iniciando una nueva actividad (más adelante se explicará en qué consiste esto). En algunos casos una actividad puede devolver un valor a la actividad anterior.

Cada actividad se implementa como una subclase de la clase *Activity*, que hará que la clase que desciende de *Activity* cree una ventana para colocar la interfaz de usuario. El contenido visual de la ventana es proporcionado por una jerarquía de vistas (por ejemplo botones y campos de texto).

Cuando se abre una ventana, la anterior se pone en pausa (véase apartado 6.1.6, ciclo de vida de una actividad) y se agrega a un historial. Posteriormente, el usuario puede navegar hacia ventanas anteriores invocando las ventanas almacenadas en dicho historial. Como es de esperar, las ventanas también pueden ser eliminadas del historial cuando resulta inapropiado su almacenamiento. Android mantiene un historial para cada aplicación que sea activada desde la pantalla de inicio.

Servicios (*services*). No tienen una interfaz de usuario, sino que se ejecutan en un segundo plano por un período de tiempo indefinido. Un ejemplo sería un programa que reproduce archivos de música desde una lista de canciones, mientras el usuario realiza otras actividades. En este caso, el programa podría iniciar un servicio con la lista a reproducir y de esa forma emitir la música sin necesidad de usar la pantalla. El sistema mantendrá el servicio de reproducción de música corriendo hasta que finalice. Una vez que la aplicación está conectada al servicio, la comunicación entre la aplicación y el servicio se realiza a través de la interfaz que el servicio expone. Para el ejemplo del reproductor de música, esta interfaz podría permitir parar una canción o saltar a una nueva.

Cada servicio extiende de la clase *Service*. Al igual que las actividades y el resto de los componentes, los servicios se ejecutan en el hilo principal del proceso de solicitud. Para no ser bloqueados por otros componentes de la interfaz, a menudo generan hilos para tareas que consumen tiempo.

Receptor de difusión (*broadcast receiver*). Es un componente que recibe y reacciona ante los anuncios de difusión. Muchos son originados por el sistema (por ejemplo anuncios de que ha cambiado la zona horaria o que la batería está baja), pero las aplicaciones también pueden iniciar un anuncio de difusión, permitiendo que otras aplicaciones sepan, por ejemplo, que algunos datos han sido descargados en el dispositivo y están disponibles para su uso.

Una aplicación puede tener cualquier número de receptores de difusión para responder a cualquier anuncio que considere importante.

Al igual que los servicios, no muestran una interfaz de usuario. Sin embargo pueden iniciar una actividad en respuesta a la información que reciben, o pueden utilizar un *NotificationManager* para alertar al usuario.

Proveedor de contenidos (*content providers*). Un proveedor de contenidos especifica un conjunto de datos de una aplicación que está a disposición de otras aplicaciones.

Cada proveedor de contenidos extiende de la clase *ContentProvider*, que implementa métodos que permiten a otras aplicaciones recuperar y almacenar datos, ya sea en archivos, en una base de datos *SQLite* o en cualquier otro mecanismo.

Los proveedores de contenidos se activan cuando son objeto de una solicitud de un *ContentResolver*, que es una clase que ofrece a las aplicaciones el acceso a un contenedor.

Para acceder al proveedor de contenidos de otra aplicación, como por ejemplo una agenda telefónica, hay que solicitar un permiso. Hay que tener en cuenta que para acceder a un archivo no es necesario usar un proveedor de contenidos.

6.1.4. Activar componentes → Intenciones

A diferencia de los proveedores de contenido, las actividades, los servicios y los receptores de difusión son activados por mensajes asíncronos llamados intenciones (*intents*).

El sistema de intenciones permite que una aplicación interactué con otra, o que una actividad interactué con otras actividades, de manera que se pueda tener aplicaciones usando características de otras aplicaciones. Esto es un concepto interesante que puede conducir a una visión del teléfono menos centrada en la aplicación y más centrada en las características. En lugar de desarrollar una aplicación completa, los desarrolladores pueden crear características específicas que realicen una tarea concreta. Estas características pueden ser usadas por otras aplicaciones para crear nuevas e interesantes funcionalidades.

Las intenciones están divididas principalmente en cuatro categorías:

Intenciones de actividad (*Activity Intents*). Son intenciones usadas para llamar a actividades fuera de la aplicación. Sólo una actividad puede manejar la intención. Por ejemplo, para un navegador web, se necesita abrir la *Web Browser Activity* para mostrar una página.

Intenciones de difusión (*Broadcast Intents*). Son intenciones que son enviadas en difusión para que múltiples actividades los manejen. Un ejemplo de una intención de difusión sería un mensaje enviado sobre el nivel de batería. Cualquier actividad puede procesar esta intención y actuar consecuentemente, por ejemplo cancelando una actividad si el nivel de batería baja a un cierto nivel.

Espera de intenciones (*PendingIntent*). Son una descripción de una intención y una acción a realizar (véase el Anexo I, apartado I.VIII).

Intenciones de filtro (*IntentFilter*). Pueden ajustarse a (filtrar) acciones, categorías, y datos (ya sea a través de su tipo, esquema, y/o ruta) en un *Intent*. También incluyen un valor de prioridad que es usado para ordenar múltiples filtros que se ajusten a dichos criterios.

Así, una intención puede ser utilizada con el método *startActivity* para poner en marcha una actividad, con un *broadcastIntent* para enviarlo a todos los *BroadcastReceiver* interesados, y con el método *startService* para comunicarse con un servicio.

Además, una intención también sirve para moverse de una ventana a otra iniciando una nueva actividad (véase el Anexo I, apartado I.V). Las dos partes más importantes de una intención son la acción y los datos (expresados como una URI) sobre los cuales se actuará. Por ejemplo, para ver la información de contacto de una persona podría ser necesario crear una intención con la acción "view" y los datos definidos como una URI que representa a esa persona.

6.1.5. Jerarquía visual

Una actividad por si sola no presenta nada en la pantalla del terminal, para ello es necesario diseñar una interfaz de usuario. Ésta se compone de vistas (*views*) y grupos de vistas (*viewgroups*), que son las clases que se usan para crear la interfaz entre el usuario y la plataforma Android.

Vistas. Una vista es un objeto cuya clase es *android.view.View*. Es una estructura de datos cuyas propiedades contienen la información específica del área rectangular del componente asociado. La vista tiene una disposición (la clase *Layout*, ya implementada, provee los tipos más comunes de disposición de pantalla), un foco, una barra de desplazamiento, etc.

La clase *View* se emplea como clase base para los *widgets*, que son componentes estandarizados que facilitan la creación de una interfaz gráfica. La lista de *widgets* disponibles incluye barras de desplazamiento, elementos de un menú, casillas de verificación, indicadores de progreso, botones, campos de texto, cronómetros, calendarios…

Grupos de vistas. Un grupo de vistas es un objeto de la clase *android.view.Viewgroup*. Su función es contener y controlar la lista de vistas y de otros grupos de vistas. Permiten añadir estructuras a la interfaz y acumular elementos complejos en la pantalla del terminal.

La clase *Viewgroup* es útil como base de la clase *Layout*. Estas disposiciones proporcionan una manera de construir una estructura para una lista de vistas.

Árbol estructurado de la interfaz de usuario. En la plataforma Android se define una actividad de la interfaz de usuario usando un árbol de nodos Vista y Grupo de Vistas, como se puede apreciar en la Figura 6.2. El árbol puede ser tan simple o complejo como sea necesario, y se puede desarrollar usando los *widgets* y las disposiciones (*layouts*) que Android proporciona o creando vistas propias.

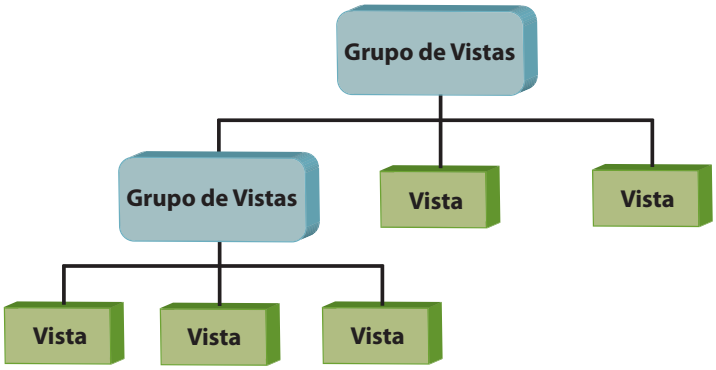


Figura 6.2: Árbol estructurado de la interfaz de usuario.

Como se ha dicho anteriormente, cada grupo de vistas es el responsable de tomar medidas sobre el espacio que tiene, preparando a sus hijos y llamando al método *Draw* por cada hijo que se muestra a si mismo. El hijo hace una petición sobre el tamaño y la localización del padre, pero el objeto padre toma la última decisión sobre el tamaño que cada hijo puede tener.

Parámetros de la disposición (*layout params*). Especifican cómo un hijo indica su posición y su tamaño. Todos los grupos de vistas usan como clase anidada una extensión de *ViewGRoup.LayoutParams*. Esta subclase contiene los tipos de propiedades que definen la posición y el tamaño de un hijo. En la Figura 6.3 se puede apreciar la distribución y organización de las disposiciones, las vistas y los parámetros de disposición.

Hay que reseñar que cada subclase *LayoutParams* tiene su propia sintaxis para cambiar los valores. Cada elemento hijo debe definir unos parámetros de disposición que sean apropiados para su padre, aunque se podrían definir diferentes parámetros de disposición para sus hijos.

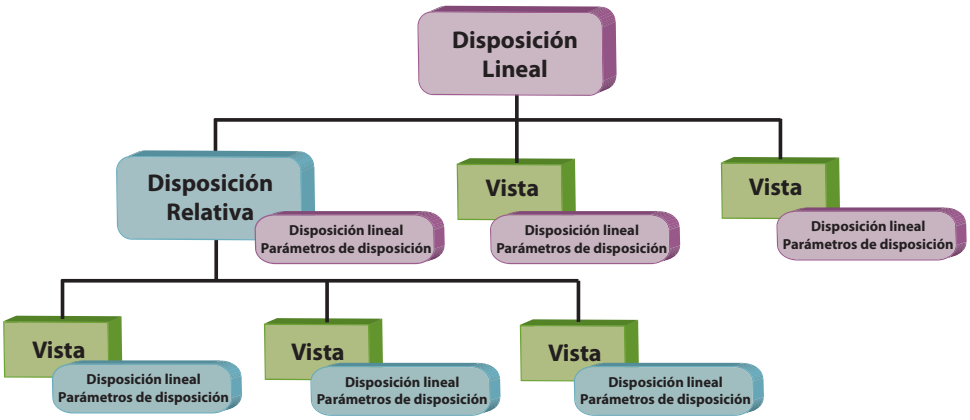


Figura 6.3: Disposiciones, vistas y parámetros de disposición.

Todos los grupos de vistas incluyen anchura y altura. La mayoría también incluyen márgenes y bordes. Se puede especificar un valor cualquiera para la altura y la anchura, pero es más común definir la vista de tal forma que sus medidas tengan un tamaño relativo al contenedor padre (por ejemplo el máximo tamaño permitido) o que ocupen lo que sea necesario (por ejemplo, en un cuadro de texto que el tamaño sea proporcional al texto escrito).

6.1.6. Ciclo de vida de una actividad

Los componentes de la aplicación tienen un ciclo de vida, empezando cuando las instancias de Android responden a través de intenciones y terminando cuando las instancias son destruidas. Entre medias, las instancias pueden estar activas o inactivas o, en el caso de las actividades, pueden ser visibles para el usuario o invisibles.

A continuación se analizará el ciclo de vida de las actividades, incluyendo los estados que pueden adquirir, los métodos que notifican las transiciones entre los estados y el efecto de los estados sobre la posibilidad de que el proceso pueda ser terminado y la instancia destruida.

El ciclo de vida de una actividad tiene esencialmente tres estados:

Está **activa** (*active*), o ejecutándose, cuando está en el primer plano de la pantalla (en la parte superior de la pila de actividades para la tarea actual). La actividad es el foco de las acciones del usuario.

Está **pausada** (*paused*) si la actividad ha perdido el foco pero sigue siendo visible para el usuario. Es decir, que otra actividad se encuentra en la parte superior de la misma actividad pero es transparente, o bien no cubre la pantalla completa, por lo que la actividad pausada puede mostrarse. Una actividad pausada está completamente viva (conserva todos los estados, la información de los miembros y sigue fiel al gestor de ventanas), pero puede ser eliminada en situaciones extremas de baja memoria.

Está **parada** (*stopped*) si está completamente oculta por otra actividad. Todavía conserva los estados y la información de los miembros. Sin embargo, ya no es visible para el usuario por lo que su ventana está oculta, y a menudo será eliminada por el sistema cuando la memoria se necesite en otro lugar.

Si una actividad se ha pausado o parado, el sistema puede eliminarla de la memoria, ya sea pidiendo que termine (llamando al método *finish*) o simplemente matando su proceso. Cuando se muestra de nuevo para el usuario, se debe reiniciar y restaurar completamente a su estado anterior.

Las transiciones de un estado a otro en una actividad le son notificadas mediante las llamadas a los métodos que aparecen en la Figura 6.4.

```
void onCreate(Bundle savedInstanceState)
void onStart()
void onRestart()
void onResume()
void onPause()
void onStop()
void onDestroy()
```

Figura 6.4: Métodos empleados para las transiciones de estado de una actividad.

Todas las actividades deben implementar *onCreate* para realizar la configuración inicial cuando el objeto es instanciado por primera vez.

En conjunto, estos siete métodos definen el ciclo de vida completo de una actividad. Tal como se muestra en la Figura 6.5, se pueden diferenciar tres bucles anidados que se pueden controlar mediante la aplicación de esos siete métodos:

El **tiempo de vida pleno** (*entire lifetime*) de una actividad ocurre entre la primera llamada a *onCreate* y la única llamada final a *onDestroy*. La actividad hace toda su

configuración inicial global en el estado *onCreate*, y libera todos los recursos restantes en *onDestroy*. Por ejemplo, si se tiene un hilo de fondo para descargar los datos de la red, se puede crear ese hilo en *onCreate* y luego detenerlo en *onDestroy*.

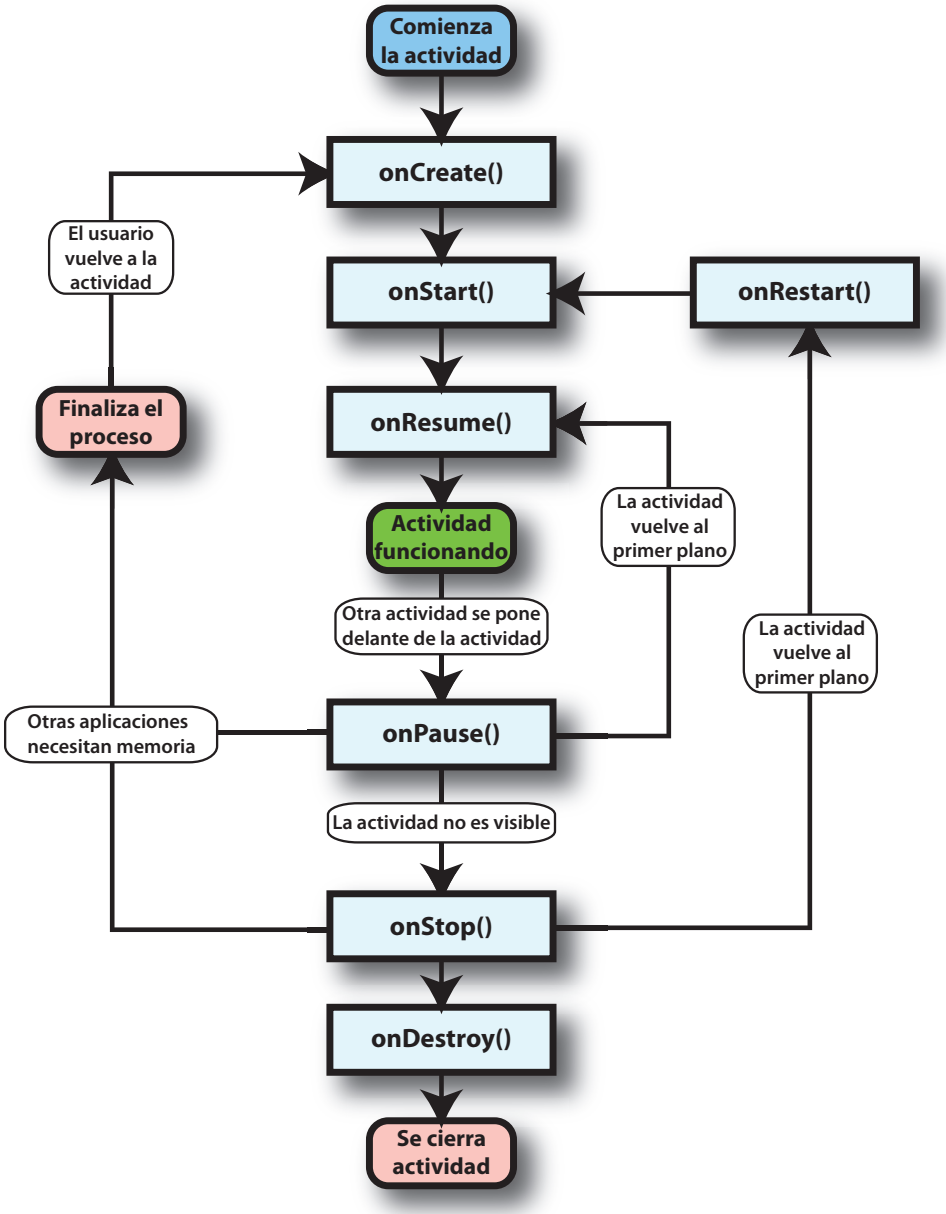


Figura 6.5: Ciclo de vida de una actividad.

El **tiempo de vida visible** (*visible lifetime*) de una actividad transcurre entre una llamada a *onStart* y una llamada a *onStop*. Durante este tiempo el usuario puede ver la actividad en pantalla, aunque puede que no sea en primer plano e interactuando con el usuario. Entre estos dos métodos se pueden mantener los recursos que se necesiten para mostrar la actividad del usuario. Por ejemplo, se puede registrar un receptor de

difusión en *onStart* para controlar los cambios que afectan a la interfaz de usuario, y dejar de registrar en *onStop* cuando deje de haber cambios visibles para el usuario. Los métodos *onStart* y *onStop* pueden ser llamados varias veces, ya que la actividad alterna entre ser visible y oculta para el usuario.

El **tiempo de vida en primer plano** (*foreground lifetime*) de una actividad ocurre entre una llamada a *onResume* y una llamada a *onPause*. Durante este tiempo la actividad se encuentra por encima de todas las demás actividades en la pantalla, y es la que interactúa con el usuario. Una actividad puede estar a menudo cambiando de estado entre la pausa y la reanudación. Por ejemplo, se puede llamar a *onPause* cuando el dispositivo se quede en suspensión o cuando se ha iniciado una nueva actividad, y se puede llamar a *onResume* cuando una actividad recibe una nueva intención. Por este motivo, el código de estos dos métodos debería ser bastante ligero.

El diagrama de la Figura 6.5 ilustra estos bucles y los caminos que puede tomar una actividad entre los diferentes estados.

6.2. Proveedor de Localización y Geocodificación

Android proporciona un gestor de localización (*Location Manager*) que suministra al sistema proveedores de localización (*Location Providers*), entre ellos el GPS. Además, dicho gestor utiliza métodos tales como *getLastKnownLocation*, que devuelve un objeto de la clase *Location* que a su vez tiene dos métodos para obtener la latitud y la longitud de la ubicación del usuario. Tanto la latitud como la longitud vienen expresados en grados decimales. En el caso en el que resulta imposible encontrar la ubicación, ambos parámetros toman el valor *null*. Para ahondar más sobre este tema se recomienda ver el apartado I.VI del Anexo I (Guía básica de desarrollo: crear una aplicación en Android), donde se explica cómo utilizar los servicios basados en la localización suministrados por Android.

De igual forma, Android puede utilizar una clase llamada *Geocoder* para encontrar una dirección a partir de una coordenada y viceversa. Esta clase utiliza el método *getFromLocationName* para obtener la coordenada correspondiente a una dirección. Para más detalles véase el apartado I.VII del Anexo I, donde se explica detalladamente cómo

se puede utilizar el *geocodificador*.

6.3. Ejemplo de interfaz y Visualización

La obtención de los datos necesarios introducidos en el terminal por el usuario y la visualización de la ruta obtenida formarían parte de WAI-Interface, tal y como se explicó en el capítulo 1. Sin embargo, con el solo objetivo de probar la aplicación se ha creado una interfaz sencilla en WAI-Routes, que va a ser explicada a continuación.

En la Figura 6.6 se puede observar el esquema de ventanas o pantallas que forman la interfaz de usuario de WAI-Routes. Las ventanas destacadas en color rojo indican que necesariamente hay que pasar por ellas para poder calcular la ruta, es decir que se necesita elegir un inicio, un destino y en la parte de opciones se debe pedir que se calcule (o muestre) la ruta. Las ventanas con líneas discontinuas indican que son una opción de la ventana superior pero que no tiene interfaz.

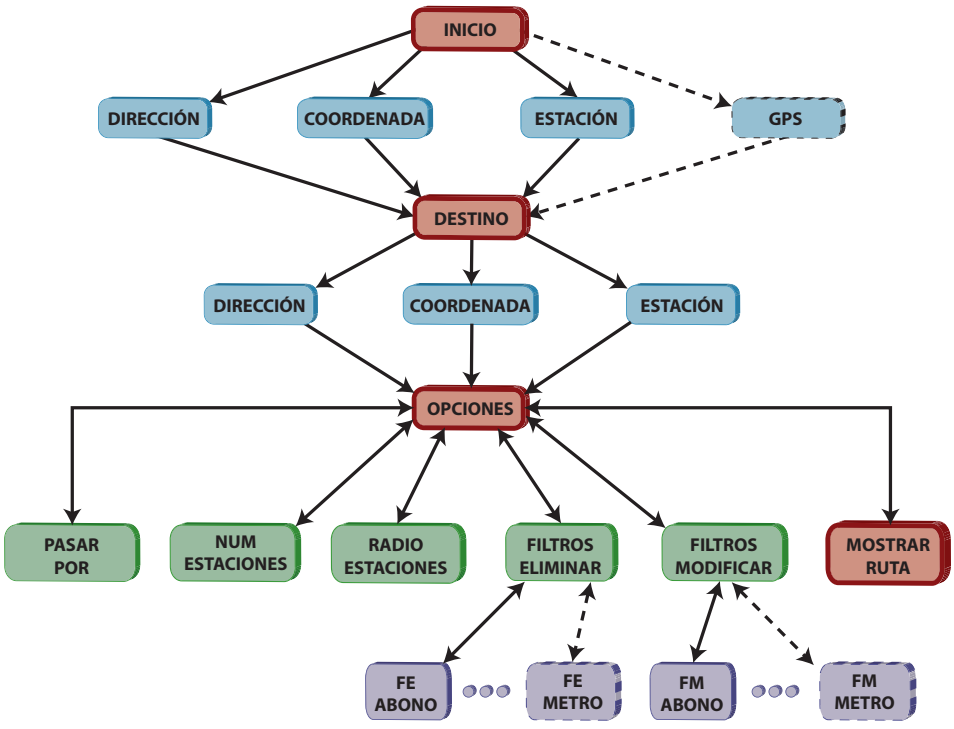


Figura 6.6: Esquema de ventanas de la interfaz de usuario de WAI-Routes.

En *Inicio* y *Destino* se pide al usuario que elija la opción, pulsando el botón correspondiente (*Button*), que desea para introducir los datos de comienzo y fin de la ruta. Estas

opciones son introducir una dirección, una coordenada y una estación (en el caso de *Inicio* también se puede usar la opción de *GPS*, que daría la coordenada en la que está situado el usuario en ese instante).

En *Dirección* se muestra un campo de texto (*EditText*) para que el usuario escriba, a través del teclado del terminal, la dirección de inicio o destino. Cuando se ha introducido la dirección correctamente, el usuario debe pulsar un botón de "ok".

En *Coordenada* se muestran dos campos de texto para que el usuario introduzca la latitud y la longitud de la coordenada. Para que el usuario no se confunda entre ambos parámetros, en el campo de la latitud ya viene escrito un "40." y en el de la longitud un "-3.", datos que corresponden con las coordenadas dentro de la Comunidad de Madrid. De nuevo, cuando se ha introducido todo correctamente el usuario debe pulsar el botón de "ok".

En *Estación* en un principio se optó por mostrar una lista con todas las estaciones posibles ordenadas alfabéticamente para que el usuario pudiera marcar la que quisiera, dado que parecía más simple que escribir la estación, y evitaba el que se produjeran errores de escritura. Sin embargo esto no resultaba demasiado práctico a la hora de hacer pruebas, dado que hay demasiadas estaciones (sólo en la red de metro hay más de 250) y resultaba muy tedioso llegar hasta la estación deseada. Finalmente se optó por usar una vista que completa automáticamente la palabra que está escribiendo el usuario (*AutoCompleteTextView*) y que coincide con alguna de las estaciones. Si hay varias posibilidades, el usuario puede elegir en una lista muy reducida la opción que desea. El usuario nunca debe olvidar pulsar el botón de "ok".

El caso de *GPS* es distinto a los casos anteriores, ya que como el usuario no debe introducir ningún dato no existe ninguna pantalla para esta opción. En *Inicio*, al pulsar el botón de *GPS* se guarda la información de que se usará el GPS y se pasa directamente a *Destino*.

En *Opciones* el usuario puede escoger los requisitos que quiere imponer a la hora de calcular la ruta, que pueden ser pasar por alguna estación (*Pasar Por*), elegir el número de estaciones cercanas (*Num Estaciones*) o el radio de estaciones a tener en cuenta (*Radio Estaciones*) cuando el inicio, el destino o ambos sean una coordenada e imponer al-

gún filtro de eliminación (*Filtros Eliminar*) o de modificación (*Filtros Modificar*). Cuando ya se hayan escogido las opciones deseadas, si se quiso seleccionar alguna, se debe pulsar el botón "calcular ruta" para obtener la ruta calculada por el servidor, la cual será mostrada por pantalla (*Mostrar Ruta*).

En *Pasar Por* el usuario puede escoger todas las estaciones por las que desea pasar a lo largo de la ruta. Para ello hay dos botones, uno para añadir y otro para eliminar estaciones. También existe un tercer botón de "ok". La lista de estaciones sigue el mismo modelo que se explicó en la ventana de *Estación*. Es importante matizar que la ruta pasará por esas estaciones en el orden en el que las introdujo el usuario.

En *Num Estaciones* se muestra una lista con posibles opciones sobre el número de estaciones cercanas a tener en cuenta. Sólo se puede marcar una de las opciones (*Spinner*). Cuando el usuario haya realizado su elección deberá pulsar el botón de "ok".

En *Radio Estaciones* se muestra una lista similar a la de *Num Estaciones*. Las diversas opciones expresan el radio en metros y de nuevo sólo se puede marcar una de las opciones. Para volver a la ventana de *Opciones*, el usuario deberá pulsar el botón de "ok" como en los casos anteriores.

En *Filtros Eliminar* hay un botón por cada tipo de filtro. Algunos de los botones dan lugar a otra ventana con interfaz, como por ejemplo *FE Abono*, donde aparecerá una lista con los tipos de abono que se emplean en la red de transporte público de la Comunidad de Madrid. Sin embargo, hay otros botones que no necesitan crear otra interfaz, como por ejemplo *FE Metro*. Cada vez que se pulse este botón se añadirá o se cancelará el filtro, y aparecerá en pantalla un mensaje que avisará del evento.

El caso de *Filtros Modificar* es exactamente igual al de *Filtros Eliminar*.

Por último está la opción de *Mostrar Ruta*. En la pantalla del terminal aparecerá la ruta que debe seguir el usuario para llegar a su destino, y la información necesaria para ello. El proceso desde que en la ventana de *Opciones* se selecciona *Mostrar Ruta* y en la propia ventana de *Mostrar Ruta* aparecen los datos de la misma ha sido explicado en el apartado 5.4.

En la Figura 5.12 se puede observar algún ejemplo de las ventanas creadas en la interfaz de WAI-Routes para solicitar los datos de la ruta al usuario. En la figura (a) se

muestran las cuatro opciones que hay para introducir el inicio. En la figura (b) se puede seleccionar cualquier estación como inicio de la ruta. En la figura (c) está la opción de indicar el destino mediante una coordenada. En la figura (d) se muestran todos los filtros de eliminación existentes.

Figura 6.7: Interfaz de WAI-Routes para seleccionar el inicio (a), elegir la estación de inicio (b), escribir la coordenada de destino (c) y añadir un filtro de eliminación (d).

En la Figura 6.8 se muestran más ejemplos de la interfaz de WAI-Routes, esta vez relacionados con los filtros de la aplicación. En la figura (a) se pueden observar todas las líneas de la red de metro para marcar aquellas por las que no se quiere pasar, es decir en un *FE Línea Metro*. La figura (b) muestra el momento en el que se ha añadido un *FE Renfe* para no utilizar el cercanías. La figura (c) puede corresponderse tanto para *FE Abono* como para *FM Abono*. En la figura (d) se puede apreciar el *FM Estaciones*, para evitar pasar Bambú y Nuevos Ministerios.

Figura 6.8: Interfaz de WAI-Routes para FE Línea Metro (a), FE Renfe (b), elegir el tipo de abono (c) y FM Estaciones.

6.3.1. *Implementación de la Visualización y la interfaz de usuario*

Para la implementación de la *Visualización* (véase capítulo 3) y la interfaz de usuario de ejemplo se ha utilizado un paquete llamado PruebaIU.

En la Figura 6.6 se mostró el esquema de ventanas que forman la interfaz. Cada ventana corresponde a una clase (o mejor dicho una actividad) distinta, que pertecen al paquete PruebaIU. Este paquete emplea dos clases más llamadas *InfoConsulta* y *PruebaIU*, que pueden verse en la Figura 6.9.

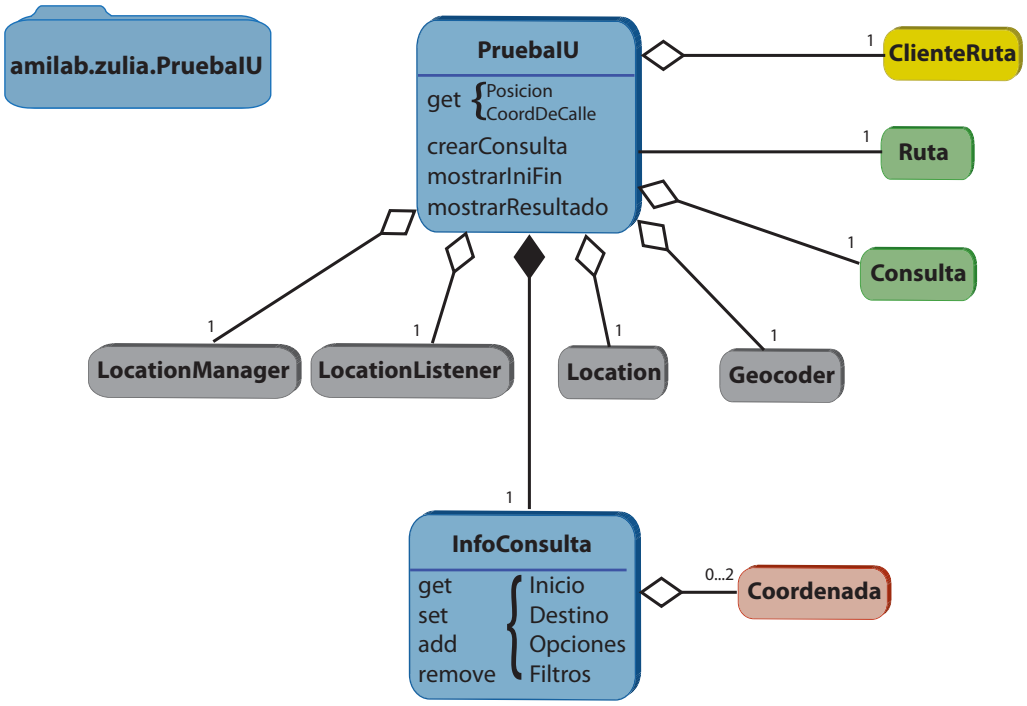


Figura 6.9: Diagrama UML del paquete PruebaIU, el cual se encarga de recoger la información del usuario y mostrar la ruta en la pantalla del terminal. Consta de dos clases, PruebaIU e InfoConsulta (en color azul). Utiliza una clase del paquete Grafo (en color rojo), dos clases del paquete Datos (en color verde), una clase del paquete ClienteRuta (en color amarillo) y cuatro clases específicas de Android (en color gris).

El paquete **PruebaIU** es el encargado de recoger los datos introducidos por el usuario para enviarlos al servidor, y de recoger la ruta calculada para mostrarla por la pantalla del terminal.

La clase *InfoConsulta*, tal y como se comentó en el apartado 5.4, se va transmitiendo de ventana en ventana recopilando la información que introduce el usuario

a través del terminal. Esta clase puede mostrar, modificar, añadir y eliminar datos relativos al inicio, al destino, a las opciones de la ruta (tales como pasar por una serie de estaciones o definir un radio de estaciones) y a los filtros que deberán ser usados al calcular la ruta. Esta clase utiliza la clase *Coordenada* del paquete Grafo (véase el apartado 4.3 del capítulo 4).

La clase *PruebaIU* emplea el método *crearConsulta* para crear un objeto *Consulta* a partir de la información almacenada en *InfoConsulta*. Este método puede usar *getPosicion* para calcular la posición actual del usuario mediante el GPS del terminal, empleando los objetos *Location*, *LocationManager* y *LocationListener*. También puede usar *getCoordDeCalle* para calcular la coordenada a partir de una dirección introducida por el usuario, empleando un objeto *Geocoder* (véase el apartado 6.2). Una vez se tiene el objeto *Consulta*, la clase *PruebaIU* crea un objeto *ClienteRuta* (véase el apartado 5.3 del capítulo 5) al que pasara dicho objeto *Consulta* y del que recibirá un objeto *Ruta*. Con los datos de este último objeto y usando los métodos *mostrarIniFin* y *mostrarResultado*, se muestra por la pantalla del terminal la ruta que debe seguir el usuario, así como una serie de información relevante para el mismo.

7 Pruebas y resultados

En este capítulo se van a detallar las pruebas realizadas para el proyecto y se van a comentar sus resultados. Se han llevado a cabo tres pruebas, una para comprobar la posición del GPS del terminal, otra para analizar la asignación de costes en el grafo de la red de transporte público y la última para comparar el cálculo de rutas de otros sistemas.

Se recuerda que el terminal móvil empleado en las pruebas de WAI-Routes es el HTC Dream, el cual utiliza el sistema operativo Android (véase el apartado 6.1 del capítulo 6).

7.1. Mediciones GPS

A través de la página web del Ayuntamiento de Madrid, en el Área de Urbanismo e Infraestructuras [36], se puede acceder a la información de la Red Topográfica de Madrid. Se han escogido las hojas de datos de siete vértices del centro de Madrid para realizar las pruebas de medición. En cada hoja de datos se especifica la situación del vértice, sus coordenadas ED50 y ETRS89, las referencias para encontrar el vértice y dos fotografías y dos croquis del lugar. Además, cada vértice está señalizado por un clavo reglamentario en el suelo (véase Figura 7.1).

Figura 7.1: Clavo reglamentario de un vértice medido por la Red Topográfica de Madrid.

Para obtener los valores del GPS del terminal se ha creado una pequeña aplicación de manera similar a lo mostrado en el apartado I.VI del Anexo I. Por cada vértice se han tomado cinco medidas, separadas en intervalos de tres minutos.

Tanto las hojas de los vértices de la Red Topográfica de Madrid como sus correspondientes medidas tomadas por el GPS del terminal se muestran en el apartado II.I del Anexo II.

En la Figura 7.2 se muestran las tablas con los resultados de las pruebas realizadas. Cada tabla pertenece a un vértice distinto.

En la primera fila se puede observar la medida del vértice según la Red Topográfica de Madrid (RTM). Este valor corresponde a las coordenadas ETRS89 (coordenadas UTM) pasadas a coordenadas geográficas mediante un conversor [37], teniendo en cuenta que el huso correspondiente a Madrid es 30 [38].

En la segunda fila se indica la media de las medidas tomadas, definida como la suma de las muestras dividido entre el número de muestras.

En la tercera fila se puede observar la desviación típica de las medidas tomadas por el GPS del terminal. La desviación típica muestral mide la separación que presentan las muestras respecto a su media. Se calcula como la raíz cuadrada de la varianza, que es la media de las desviaciones de las muestras con respecto a la media elevadas al cuadrado:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}}$$

En la última fila se muestra el error en metros de la media de las medidas tomadas respecto a la medida de la RTM. El error se ha calculado sabiendo que un grado de latitud corresponde a 111.325 km, y que un grado de longitud en la latitud *X* corresponde a cos(*X*) * 111.325 km (véase el apartado II.II del Anexo II). Este error es un dato orientativo, dado que la medida de la RTM no es una medida exacta. Se ha procurado que las medidas tomadas por el GPS del terminal se aproximaran lo más posible a las de la RTM, colocando el terminal justo encima del clavo reglamentario a una altura aproximada de 1,30 m. La forma de tomar las medidas también afecta al error de medición de los vértices.

Vértice 6949	**Longitud**	**Latitud**
RTM	-3,70385052	40,414343054
Media GPS	-3,70382998	40,414463306
Desviación GPS	$1{,}825 \times 10^{-5}$	$1{,}78473 \times 10^{-4}$
Error RTM-GPS	1,74177918 m	13,386998160 m

Vértice 7678	**Longitud**	**Latitud**
RTM	-3,70334600	40,41413206
Media GPS	-3,70340333	40,41406664
Desviación GPS	$2{,}981 \times 10^{-5}$	$5{,}400 \times 10^{-5}$
Error RTM-GPS	4,85913867 m	7,28297870 m

Vértice 8061	**Longitud**	**Latitud**
RTM	-3,699715419	40,41983922
Media GPS	-3,699806638	40,41975332
Desviación GPS	$1{,}10914 \times 10^{-4}$	$5{,}701 \times 10^{-5}$
Error RTM-GPS	7,731047846 m	9,56379850 m

Vértice 8070	**Longitud**	**Latitud**
RTM	-3,711365636	40,418408966
Media GPS	-3,711583312	40,418543312
Desviación GPS	$3{,}3352 \times 10^{-5}$	$3{,}8362 \times 10^{-5}$
Error RTM-GPS	18,449055825 m	14,955995387 m

Vértice 8072	**Longitud**	**Latitud**
RTM	-3,70928613	40,41795017
Media GPS	-3,70923665	40,41781996
Desviación GPS	$2{,}738 \times 10^{-5}$	$2{,}736 \times 10^{-5}$
Error RTM-GPS	4,19404768 m	14,49604693 m

Vértice 8082	**Longitud**	**Latitud**
RTM	-3,707769063	40,41557533
Media GPS	-3,707713312	40,41560332
Desviación GPS	$4{,}3116 \times 10^{-5}$	$6{,}708 \times 10^{-5}$
Error RTM-GPS	4,725408233 m	3,11529764 m

Vértice 9353	**Longitud**	**Latitud**
RTM	-3,707683208	40,415191680
Media GPS	-3,707749984	40,415216652
Desviación GPS	$3{,}9079 \times 10^{-5}$	$3{,}1173 \times 10^{-5}$
Error RTM-GPS	5,659845805 m	2,779998036 m

Figura 7.2: Datos obtenidos a partir de los vértices de la Red Topográfica de Madrid y las medidas realizadas mediante el GPS del terminal.

Además, también habría que tener en cuenta el error correspondiente a las conversiones de coordenadas y las condiciones meteorológicas en el momento de tomar las medidas (se puede dejar de recibir la señal de algunos satélites y por tanto perder precisión en la medida de la posición).

Según los datos de la Figura 7.2 el error orientativo del GPS del terminal varía aproximadamente entre dos y quince metros. Aunque quince metros no es una distancia demasiado grande, si puede ser un impedimento para personas con discapacidad cognitiva que no sean capaces de ubicarse y orientarse bien en el espacio.

Dado que el valor de todas las desviaciones calculadas ronda entre valores $x10^{-4}$ y $x10^{-5}$ se llega a la conclusión de que todas las muestras tomadas por el GPS se desvían relativamente poco respecto a la media. La mayoría de las medidas de un mismo vértice tienen los cuatro primeros decimales iguales.

7.2. Asignación de costes

Tal y como se mencionó en el apartado 3.1.4 del capítulo 3, la asignación de costes del grafo de la red de transporte público es compleja.

Inicialmente se partió de la idea de que todas las líneas era iguales con un coste igual a 1 y de que todos los transbordos también eran iguales con un coste igual a 4. Si se quiere ir de Argüelles a Pinar de Chamartín, la ruta más lógica sería recorrer toda la línea de metro L04. Sin embargo, la aplicación escogía como la ruta más adecuada tomar la línea de metro L06 de Argüelles a Cuatro Caminos y en esa estación cambiar a la línea de metro L01 hasta Pinar de Chamartín. Como la línea L04 tiene tantas paradas entre el inicio y el destino de la ruta, para que la aplicación escogiera recorrer la línea de metro L04 sin hacer transbordos había que subir el coste total del transbordo de 4 a 9. Es difícil poner el límite de paradas por una misma línea que equivaldrían a hacer un transbordo. No tiene sentido penalizar tanto los transbordos cuando el usuario tiene la opción de elegir un filtro que, o bien elimine los transbordos o bien reduzca el número de ellos (véase capítulo 4).

El problema de la asignación de costes no sólo afecta a la hora de crear el grafo para diferenciar estaciones, líneas y/o transbordos. A la hora de emplear los filtros de modifi-

cación (véase el apartado 4.1.4 del capítulo 4) también se modifican los costes, aumentando su valor para evitar pasar por una estación, línea y/o transbordo. Si se sumaba un coste muy pequeño a los filtros de modificación, por ejemplo un coste de 6, era bastante probable que la ruta obtenida no evitara las estaciones, líneas y/o transbordos porque no saliera rentable.

Para una correcta asignación de coste se debe hacer un estudio en profundidad sobre la red de transporte público. Este aspecto queda como trabajo futuro para mejorar la aplicación.

7.3. Comparativa en el cálculo de rutas

Las rutas obtenidas por WAI-Routes han sido comparadas con otros dos sistemas, el ofrecido por la página web del Sistema de Información de Transportes de Madrid [9] y el propuesto por la página web del Metro de Madrid [39].

En total se han comparado siete rutas, tal y como se puede observar en la Figura 7.3. Las cuatro primeras rutas son desde una estación hasta otra estación, la penúltima es desde una dirección hasta otra dirección y la última es desde una dirección hasta una estación. WAI-Routes ha calculado la ruta sin tener en cuenta ningún tipo de filtro. El Sistema de Información de Transportes de Madrid (*ctm-madrid*) calcula tres posibles rutas, la óptima, la que tiene menos transbordos y la más rápida. El concepto "óptimo" en este caso es un tanto ambiguo, ¿para quién es óptima esa ruta?, ¿en qué se basa?. Por otro lado, el Metro de Madrid (*metromadrid*) obtiene otros tres tipos de rutas, la más rápida, la que tiene menos transbordos y la que tiene menor longitud. El símbolo **||** significa que la ruta es igual a la ruta de la fila superior, y el símbolo **=** implica que la ruta es igual a la obtenida por WAI-Routes.

En la Figura 7.3 - (a)-(d)-(g) todas las rutas calculadas son igual a la de WAI-Routes exceptuando la ruta con menor longitud ofrecida por *metromadrid* (es el único de los tres sistemas evaluados que tiene esa opción). En (a) y (d) la ruta con menor longitud tiene un transbordo más que la de WAI-Routes. En (g), estando aproximadamente a la misma distancia de las estaciones Francos Rodriguez y Estrecho, la ruta con menor longitud de metromadrid escoge Estrecho como inicio y realiza dos transbordos más que WAI-Routes.

(a)

PORTAZGO → ALTO DE EXTREMADURA		
WAI-Routes	Sin filtros	Portazgo (L01) - Pacífico (L01-L06) - Alto de Extremadura (L06)
ctm-madrid	Óptima	=
	Min. Transbordos	=
	Rápida	=
metromadrid	Rápida	=
	Min. Transbordos	=
	Min. Longitud	Portazgo (L01) - Sol (L01-L03) - Argüelles (L03-L06) - Alto de Extremadura (L06)

(b)

SOL → CIUDAD LINEAL		
WAI-Routes	Sin filtros	Sol (L01) - Gran Vía (L01-L05) - Ciudad Lineal (L05)
ctm-madrid	Óptima	Caminar hasta Gran Vía (L05) - Ciudad Lineal (L05)
	Min. Transbordos	Caminar hasta Gran Vía (L05) - Ciudad Lineal (L05)
	Rápida	=
metromadrid	Rápida	=
	Min. Transbordos	=
	Min. Longitud	Sol (L02) - Ventas (L02-L05) - Ciudad Lineal (L05)

(c)

MANUEL DE FALLA → HORTALEZA		
WAI-Routes	Sin filtros	Manuel de Falla (L10) - Chamartín (L10-L01) - Pinar de Chamartín (L01-L04) - Hortaleza (L04)
ctm-madrid	Óptima	Manuel de Falla (L10) - Las Tablas (L10-ML1) - Pinar de Chamartín (ML1-L04) - Hortaleza (L04)
	Min. Transbordos	II
	Rápida	II
metromadrid	Rápida	II
	Min. Transbordos	II
	Min. Longitud	II

(d)

SAN BLAS → PRADO DEL REY		
WAI-Routes	Sin filtros	San Blas (L07) - Gregorio Marañón (L07-L10) - Colonia Jardín (L10-ML2) - Prado del Reyl (ML2)
ctm-madrid	Óptima	=
	Min. Transbordos	=
	Rápida	=
metromadrid	Rápida	=
	Min. Transbordos	=
	Min. Longitud	San Blas (L07) - Pueblo Nuevo (L07-L05) - Alonso Martínez (L05-L10) - Colonia Jardín (L10-ML2) - Prado del Reyl (ML2)

(e)

LISTA → ABRANTES		
WAI-Routes	Sin filtros	Lista (L04) - Diego de León (L04-L06) - Plaza Elíptica (L06-L11) - Abrantes (L11)
ctm-madrid	Óptima	Caminar hasta Diego de León (L06) - Plaza Elíptica (L06-L11) - Abrantes (L11)
	Min. Transbordos	II
	Rápida	=
metromadrid	Rápida	Lista (L04) - Goya (L04-L02) - Manuel Becerra (L02-L06) - Plaza Elíptica (L06-L11) - Abrantes (L11)
	Min. Transbordos	=
	Min. Longitud	Lista (L04) - Goya (L04-L02) - Manuel Becerra (L02-L06) - Plaza Elíptica (L06-L11) - Abrantes (L11)

(f)

C/ VÁLGAME DIOS → C/ SOTOMAYOR		
WAI-Routes	Sin filtros	Chueca (L05) - Diego de León (L05-L06) - Guzmán el Bueno (L06)
ctm-madrid	Óptima	=
	Min. Transbordos	=
	Rápida	=
metromadrid	Rápida	Gran Vía (L01) - Cuatro Caminos (L01-L06) - Guzmán el Bueno (L06)
	Min. Transbordos	II
	Min. Longitud	II

(g)

C/ WAD RAS → CARTAGENA		
WAI-Routes	Sin filtros	Francos Rodriguez (L07) - Cartagena (L07)
ctm-madrid	Óptima	=
	Min. Transbordos	=
	Rápida	=
metromadrid	Rápida	=
	Min. Transbordos	=
	Min. Longitud	Estrecho (L01) - Cuatro Caminos (L01-L06) - Avda. de América (L06-L07) - Cartagena (L07)

Figura 7.3: Comparativa de rutas calculadas por WAI-Routes, la página web del Consorcio de Transportes de Madrid y la página web del Metro de Madrid.

En la Figura 7.3 - (b) la ruta óptima y la que tiene menos transbordos ofrecida por *ctm-madrid* hace andar al usuario hasta la estación de Gran Vía a pesar de haberse especificado que el punto de partida es la estación de Sol, lo cual no tiene demasiado sentido (aunque por andar se elimine un transbordo). De nuevo también varía la ruta con menor longitud de *metromadrid*, aunque en este caso con el mismo número de transbordos que WAI-Routes. En la Figura 7.3 - (e) sucede lo mismo sólo que esta vez la ruta con menor longitud tiene un transbordo más que la obtenida por WAI-Routes.

En la Figura 7.3 - (c) todas las rutas calculadas son distintas a la de WAI-Routes. Parece más lógico coger la línea ML1 que la línea L01, ya que de la segunda forma parece que el recorrido es más largo. Sin embargo, WAI-Routes ha escogido la segunda opción porque para un coste igual a todas las líneas y un coste igual a todos los transbordos, cogiendo la línea L01 se pasa por una estación menos, por lo que el coste es menor.

En la Figura 7.3 - (f), tanto WAI-Routes como *ctm-madrid* escogen como estación de inicio Chueca, mientras que *metromadrid* toma como inicio la estación de Gran Vía. Es más lógico empezar la ruta en Chueca dado que es la estación más cercana al punto de inicio, y además el número de transbordos es igual a la ruta que empieza en Gran Vía.

Por último, en la Tabla 7.1 se muestra una comparativa de las opciones que ofrece cada uno de los tres sistemas estudiados, teniendo en cuenta el uso de los filtros en WAI-Routes. Se recuerda que *ctm-madrid* hace referencia a la página web del Sistema de Información de Transportes de Madrid y *metromadrid* se corresponde con la página web del Metro de Madrid.

Observando la Figura 7.3 se llega a la conclusión de que WAI-Routes, sin emplear filtros, calcula rutas similares a las ofrecidas por *ctm-madrid* y *metromadrid*. Sin embargo, si se observa la Tabla 7.1 se puede apreciar claramente que WAI-Routes es mucho más versátil que los otros dos sistemas, ya que ofrece muchas más posibilidades. De hecho, ninguno de los otros dos sistemas está orientado a personas con discapacidades cognitivas. Por el contrario, WAI-Routes carece de alguna opción interesante como por ejemplo mostrar el tiempo estimado del viaje. Esta opción no se ha llevado a cabo porque es muy difícil de estimar el tiempo de viaje, ya que habría que saber con que frecuencia salen los trenes, cuando se tarda de estación a estación, habría que tener en cuenta el momento del día ya que por ejemplo en hora punta pasan más trenes, etc. Tampoco se ha llevado a

cabo la posibilidad de empezar o terminar la ruta en un lugar de interés, aunque tal cual está programado WAI-Routes esta modificación no sería demasiado compleja de realizar. Por último, no se ha tenido en cuenta el cálculo de rutas con menor longitud porque se ha considerado que no es un parámetro importante a la hora de viajar, y mucho menos para personas con limitaciones cognitivas.

	WAI-Routes	ctm-madrid	metromadrid
Pasar por estaciones intermedias	✓	✗	✗
Tener en cuenta radio o número de estaciones cercanas	✓	(1)	✗
Eliminar estaciones y/o líneas	✓	✗	✗
Eliminar tipo de transporte	✓	(2)	✗
Tener en cuenta estaciones adaptadas	✓	✗	✗
Tener en cuenta tipo de abono transportes	✓	✗	✗
Eliminar transbordos	✓	✗	✗
Minimizar transbordos	✓	✓	✓
Evitar estaciones y/o líneas	✓	✗	✗
Evitar tipo de transporte	✓	✗	✗
Origen/Destino como estación y/o dirección	✓	✓	✓
Origen/Destino como lugar de interés	✗	✓	✓
Origen/Destino como una coordenada	✓	✗	✗
Origen con GPS	✓	✗	✗
Tiempo estimado del viaje	✗	✓	✓
Ruta más rápida	✗	✓	✓
Ruta con menor longitud	✗	✗	✓

(1) *Quizá tenga en cuenta estaciones cercanas pero el usuario no puede establecerlo.*

(2) *Opciones:*
- *sólo metro/ML/cercanías*
- *sólo autobús*
- *sólo cercanías*
- *todo*

Tabla 7.1: Tabla comparativa de las opciones ofrecidas por tres sistemas de cálculo de rutas usando el transporte público de Madrid: WAI-Routes, *ctm-madrid* y *metromadrid*.

8 Conclusiones y trabajo futuro

8.1. Conclusiones

WAI-Routes calcula una ruta entre dos puntos de la Comunidad de Madrid usando el transporte público, y teniendo en cuenta el perfil del usuario. Esta aplicación ha sido específicamente creada para ayudar a usuarios con limitaciones cognitivas a ir de un lugar a otro. Con este propósito se ha tenido en cuenta el modelo de usuario y el uso de una serie de filtros. Estos filtros pueden ser prefijados la primera vez que el usuario use la aplicación o cada vez que el usuario quiera calcular una ruta.

Es importante recordar que aunque WAI-Routes ha sido probado con el grafo de la red de transporte público de la Comunidad de Madrid, también podría ser utilizado para cualquier otra red de transporte de otra ciudad. Simplemente se deberían modificar los archivos que contienen la información de las estaciones, las líneas y los transbordos. Así mismo, este sistema se puede utilizar con cualquier otro cliente que no sea el empleado en las pruebas de este proyecto.

Para comprobar el funcionamiento de WAI-Routes se implementó una interfaz de prueba, tal y como se explicó en el apartado 6.3 del capítulo 6. Así mismo, se comparó la obtención de rutas del sistema desarrollado con otros dos sistemas similares (véase el apartado 7.3 del capítulo 7). Se llegó a la conclusión de que WAI-Routes es más versátil y ofrece muchas más opciones, principalmente porque el sistema desarrollado está enfocado a personas con discapacidades cognitivas.

También se hicieron una serie pruebas con un terminal real para comprobar las medidas que se obtenían con el GPS (véase el apartado 7.1 del capítulo 7). Estas medidas se compararon con otras realizadas por la Red Topográfica de Madrid. Se estimó que el GPS obtenía las medidas con un error orientativo que varía desde los dos a los quince metros.

Como el terminal usado (el HTC Dream) emplea el sistema operativo Android, inicialmente se desarrolló una aplicación de prueba llamada *BotonAlerta*. Esta aplicación consiste en que si en un momento determinado el usuario se pierde, al pulsar un botón se envía un SMS a una persona de confianza (un familiar, un cuidador, etc) con la ubicación del usuario. Si éste se desplaza 10 metros desde su posición o transcurre media hora, el terminal vuelve a enviar un SMS con la nueva ubicación. Mientras tanto, en la pantalla del terminal aparece un mensaje tranquilizador para el usuario, indicándole que en seguida alguien vendrá a ayudarle. A raíz de esta aplicación de prueba se ha desarrollado un tutorial sobre como crear aplicaciones en Android (véase Anexo I).

Con motivo de este proyecto se ha escrito un artículo titulado "WAI-Routes: a route-estimation system for aiming public transportation users with cognitive impairments". Será publicado en la conferencia *Advances in Computer Science and Engineering* (ACSE 2010), que tendrá lugar del 15 al 17 de marzo del presente año en Sharm El Sheikh (Egipto).

8.2. Trabajo futuro

WAI-Routes sólo se centra en el cálculo de la ruta. El siguiente paso sería el desarrollo de WAI-Interface, que se encargaría de la interacción con el usuario. El sistema emplearía la posición en que se encuentre el usuario, y por medio de una interfaz adaptativa y muy intuitiva, le asistirá (según las necesidades específicas de cada usuario) indicándole cómo dirigirse a un destino, o mostrándole los recursos cercanos de que dispone.

Respecto a WAI-Routes, una de las mejoras que se podrían llevar a cabo sería generar varias rutas posibles en lugar de una sola. Cada ruta sería registrada con una clasificación que reflejaría las preferencias del usuario. Esta opción podría llevarse a cabo de dos formas distintas. La opción más sencilla sería no descartar las diferentes combinaciones de inicio-destino para escoger únicamente la de menor coste (véase el apartado 4.1.1 del capítulo 4), ya que esto impide la obtención de varias rutas. La segunda opción consistiría en la utilización del mecanismo de filtrado. Una vez que se ha calculado una ruta se crearía un filtro de eliminación. Este filtro eliminaría el tramo desde el último transbordo realizado hasta el lugar de destino. A continuación se crearía una nueva ruta con dicho filtro. Este

paso de podría repetir tantas veces como fuera necesario para obtener las diferentes rutas. El ranking asociado a cada ruta aparecería en función del coste de cada ruta.

Adicionalmente, sería interesante cotejar qué tareas tiene que realizar el usuario en ese lugar y en ese instante concreto (por ejemplo comprar el billete). En caso de que existiera alguna, el usuario podría ser advertido con distintos grados de insistencia dependiendo de la prioridad de la misma y de las limitaciones del usuario. Además, el sistema podría realizar recomendaciones sobre siguientes tareas a realizar, resúmenes de las tareas completadas y no completadas a lo largo del día, rutas seguidas por el usuario… Finalmente, una funcionalidad a añadir sería la posibilidad de permitir a los cuidadores o familiares que accedan a las trayectorias de las personas bajo sus responsabilidad, teniendo en consideración las cuestiones relativas a la privacidad de los usuarios del sistema. Para ello, habría que implementar el monitorizador de rutas de manera similar a como se explicó en el apartado 4.2 del capítulo 4.

Otra mejora útil sería ampliar la red de transporte público completando la red de cercanías (actualmente no está completa por falta de información) y añadiendo la red de autobuses de la Comunidad de Madrid.

Bibliografía

[1] HADA, Hipermedia Adaptativa para la atención a la Diversidad en entornos de inteligencia Ambiental (TIN2007-64718). http://hada.ii.uam.es.

[2] Federación Española de Síndrome de Down. http://www.sindromedown.net/.

[3] Consolvo, S.; Roessler, P.; Shelton, B.; Lamarca, A.; Schilit, B. and Bly, S. (2004). "Technology for care networks of elders". Pervasive Computing, IEEE, Vol. 3, No. 2, pp. 22-29.

[4] Patterson, D. J.; Liao, L.; Gajos, K.; Collier, M.; Livic, N.; Olson, K.; Wang, S.; Fox, D. and Kautz, H. (2004). "Opportunity Knocks: A System to Provide Cognitive Assistance with Transportation Services." UbiComp 2004: Ubiquitous Computing, pp. 433-450.

[5] Want, R.; Hopper, A.; Falco, V. and Gibbons, J. (1992) "The Active Badge location system, ACM Transactions on Information Systems, 10, 1, pp. 91-102.

[6] Hightower, J. and Borriello, G. (2001) "Location Systems for Ubiquitous Computing", IEEE Computer, 34, 8, pp. 57-66.

[7] Junglas, I. A. and Watson, R. T. (2008) "Location-based services". Commun. ACM 51, 3 (Mar. 2008), pp. 65-69.

[8] Barkhuus, L. and Dey, A. (2003) "Location-Based Services for Mobile Telephony: a study of user's privacy concerns". In Proceedings of INTERACT'03, pp. 709-712.

[9] Sistema de Información de Transportes de Madrid. http://www.ctm-madrid.es/.

[10] Berg Insight. (2006) "Strategic Analysis of the European Mobile LBS Market". BRG1352742.

[11] Repenning, A. and Ioannidou, A. (2006). "Mobility agents: guiding and tracking public transportation users". In Proceedings of the Working Conference on Advanced Visual interfaces (Venezia, Italy, May 23 - 26, 2006). AVI '06. ACM, New York, NY, pp. 127-134.

[12] Chang, Y., Tsai, S., Chang, Y., and Wang, T. (2007). "A novel wayfinding system based on geo-coded qr codes for individuals with cognitive impairments". In Proceedings of the

9th international ACM SIGACCESS Conference on Computers and Accessibility (Tempe, Arizona, USA, October 15 - 17, 2007). Assets '07. ACM, New York, NY, 231-232.

[13] Tsai, S. (2007). "WADER: a novel wayfinding system with deviation recovery for individuals with cognitive impairments". In Proceedings of the 9th international ACM SIGACCESS Conference on Computers and Accessibility (Tempe, Arizona, USA, October 15 - 17, 2007). Assets '07. ACM, New York, NY, 267-268.

[14] Transports Metropolitans de Barcelona (TMB). http://www.tmb.cat/vullanar/es_ES/vullanar.jsp.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, second edition. MIT Press and McGraw-Hill, 2001. Section 24.1: The Bellman-Ford algorithm, pp.588-592.

[16] "Lecture 12: Shortest paths (continued)" (PDF). Network Flows and Graphs. Department of Industrial Engineering and Operations Research, University of California, Berkeley. 7 October 2008. http://www.ieor.berkeley.edu/~ieor266/Lecture12.pdf.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, second edition. MIT Press and McGraw-Hill, 2001. Section 25.2: The Floyd–Warshall algorithm, pp.629-635.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, second edition. MIT Press and McGraw-Hill, 2001. Section 25.3: Johnson's algorithm for sparse graphs, pp.636-640.

[19] Stuart Russell and Peter Norvig. Artificial intelligence, a modern approacha, second edition. Prentice Hall, 2003. Section 4.1: Informed (heuristic) search strategies, pp.94-104.

[20] McGill University. School of Computer Science. Winter 1997. Class Notes for 308-251B. Data structures and algorithms. Topic #29: Shortest path algorithms. http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic29/.

[21] Stuart Russell and Peter Norvig. Artificial intelligence, a modern approacha, second edition. Prentice Hall, 2003. Section 24.9: Dijkstra algorithm, pp. 595-601.

[22] Andén 1 | Asociación de amigos del metro de Madrid: http://www.anden1.org/.

[23] Google Earth: http://earth.google.es/.

[24] Federal Comunications Commission (FCC): http://www.fcc.gov/.

[25] T. Vicenty. "Direct and inverse solutions of seodesics on the ellipsoid with application of nested equations". Survey Review XXII, Vol. 176, April 1975.

[26] F. Vazquez Maure y J. Martín López. Lectura de Mapas. Ministerio de Obras Públicas y Urbanismo, Instituto Geográfico Nacional. Madrid, 1987. Capítulo IV: Elementos geográficos en el plano horizontal, pp. 67-72.

[27] Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/WGS84

[28] NIMA. Technical report TR8350.2. Department of Defense World Geodetic System 1984.

[29] Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/Municipios_del_Abono_Transportes_de_Madrid.

[30] JSON: http://www.json.org/.

[31] J. F. di Marzio. Android. A programmer's guide. Mc Graw Hill, 2008.

[32] R. Rogers, J. Lombardo, Z. Mednieks, B. Meike. Android. Application development. O' Reilly, 2009.

[33] Reto Meier. Professional. Android. Application development. Wrox, 2009.

[34] Android developers: http://developer.android.com/index.html.

[35] Comunidad oficial de Android en español: http://www.android-spa.com/.

[36] Ayuntamiento de Madrid, Área de Infraestructuras: http://www.munimadrid.es/

[37] Conversor del Centro de Investigación en Geografía Aplicada (CIGA) de la Pontificia Universidad Católica del Perú (PUCP): http://www.atlascajamarca.info/conversor/.

[38] Página Española de los GPS: http://www.elgps.com/documentos/utm/coordenadas_utm.html.

[39] Metro de Madrid: http://www.metromadrid.es/es/index.html.

Glosario

Android SDK. Kit de desarrollo software que incluye las APIs y herramientas necesarias para programar e implementar todo tipo de aplicaciones para Android utilizando el lenguaje de programación JAVA.

API. *Application Programming Interface*. Interfaz de programación de aplicaciones, es el conjunto de funciones y procedimientos que provee un sistema operativo, una aplicación o una biblioteca que definen cómo invocar desde un programa un servicio que éstos prestan. En otras palabras, una API representa una interfaz de comunicación entre componentes software.

Códigos QR. *Quick Response Barcode*. Sistema para almacenar información en una matriz de puntos o un código de barras bidimensional. Se caracteriza por tres cuadrados que se encuentran en las esquinas y que permiten detectar la posición del código al lector.

Framework. Es una estructura de soporte definida, en la cual otro software puede ser organizado y desarrollado. Son diseñados con la intención de facilitar el desarrollo de aplicaciones software.

Geocodificador. Instrumento de búsqueda de coordenadas geográficas (a menudo expresadas en latitud y longitud) y de otros datos geográficos, tales como calles o códigos postales.

GPS. *Global Position System*. Es un sistema global de navegación por satélite que permite determinar la posición de un objeto en todo el mundo.

JSON. *JavaScript Object Notation*. Formato ligero de intercambio de datos, completamente independiente del lenguaje de programación.

Layout. Disposición. Define la posición en el espacio (y su representación gráfica) de los componentes de un sistema.

Middleware. Software que conecta componentes software o aplicaciones para que puedan intercambiar datos entre ellas.

PDA. *Personal Digital Assistant*. Dispositivo portátil de tamaño muy reducido (de bolsillo) que tiene su propio sistema operativo, lleva programas instalados, permite intercambiar información con ordenadores convencionales, tiene Internet y GPS.

SMS. *Short Message Service*. Permite enviar y recibir mensajes de texto de hasta 160 caracteres a teléfonos móviles.

Tabla hash. Estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (valores) almacenados a partir de una clave.

UML. *Unified Modeling Language*. Lenguaje de Modelamiento Unificado, es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software.

URI. *Uniform Resource Identifier*. Cadena de caracteres que permiten acceder a recursos (páginas) web.

UTM. *Universal Transverse Mercator*. Es un sistema de coordenadas basado en la proyección cartográfica transversa de Mercator. A diferencia del sistema de coordenadas geográficas, expresadas en longitud y latitud, las magnitudes en el sistema UTM se expresan en metros.

XML. *Extensible Markup Language*. Lenguaje extensible de etiquetas que permite definir la gramática de lenguajes específicos. En realidad no es un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades.

Anexo I. Guía básica de desarrollo: crear una aplicación en Android

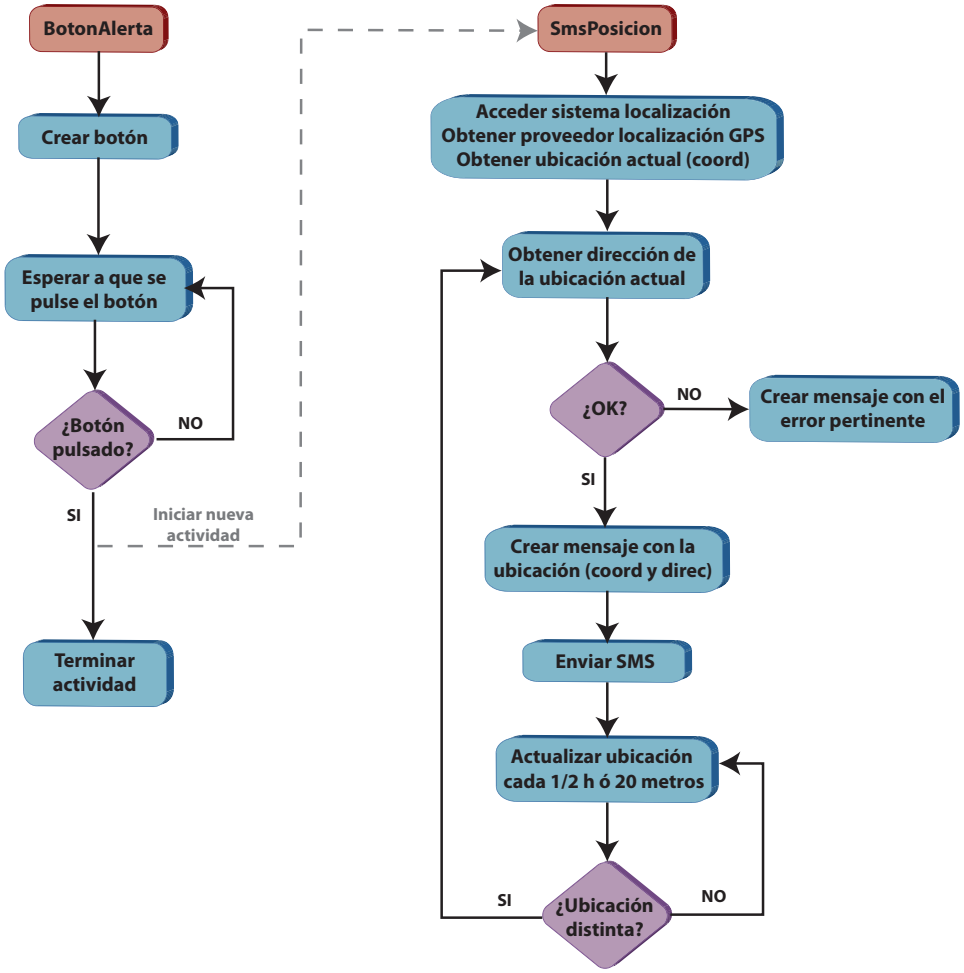


Figura I-1: Esquema del ejemplo de la aplicación con sus dos clases, *BotonAlerta* y *SmsPosicion*, sus respectivas funcionalidades y la relación entre ellas.

Para entender cómo se programa en Android, a continuación se explica paso a paso el desarrollo de una aplicación. El programa se llama *BotonAlerta* y su objetivo es avisar a un cuidador o persona de confianza de la localización del usuario cuando éste se ha perdido, de tal forma que puedan ir a buscarlo. La aplicación funcionará del siguiente modo: cuando se pulse un botón se enviará automáticamente un SMS a un destinatario predefinido anteriormente con la localización exacta del usuario (dirección y coordenadas). Además, cada vez que el usuario se aleje 20 metros de su ubicación actual se volverá a mandar otro SMS con la nueva posición. Una vez que el botón ha sido pulsado, se cambia a otra pantalla en la que aparece un pequeño texto avisando al usuario de que en seguida vendrán a recogerle, y cada

vez que se envía un SMS aparece un aviso que notificará al usuario que el envío se realizó correctamente o que hubo algún tipo de incidente.

En la Figura I-1 se puede observar un esquema con los componentes de la aplicación y las relaciones entre ellos.

I.I. Requisitos del sistema y puesta en marcha

Inicialmente, hay que asegurarse que el sistema cumple con los requisitos necesarios para poder utilizar Android.

Los sistemas operativos soportados por Android son:

- Windows XP o Vista.
- Linux (probado en Linux Ubuntu Dapper Drake).
- Mac OS X 10.4.8 o superior (sólo x86).

Por otro lado, los entornos de desarrollo que se pueden utilizar son:

- Eclipse IDE
  - Eclipse 3.3 (Europa), 3.4 (Ganymede).
  - Eclipse JDT Plugin.
  - JDK 5 ó JDK 6 (JRE por sí solo no es suficiente).
  - No es compatible con Gnu Compiler para Java (gcj).
- Otros
  - JDK 5 o JDK 6 (JRE por sí solo no es suficiente).
  - Apache Ant 1.6.5 o superior para Linux y Mac, 1.7 o superior para Windows.
  - No es compatible con Gnu Compiler para Java (gcj).

Si el JDK ya está instalado, se debe verificar que la versión cumple con los requerimientos descritos. Generalmente, las distribuciones Linux incluyen JDK 1.4 o Gnu Compiler para Java, los cuales no tienen soporte en el entorno de desarrollo de Android.

El entorno de desarrollo Eclipse ya dispone de un *plugin* para Android, y por ello se recomienda usarlo para la programación en Java. Para instalar este *plugin* se pueden ver las instrucciones detalladas paso a paso en la página oficial de *Android Developers*.

Finalmente, para realizar aplicaciones en Android también es necesario instalarse el *Android SDK*. De nuevo, las instrucciones detalladas para ello se encuentran en la página oficial de *Android Developers*.

I.II. Crear un nuevo proyecto

En primer lugar se tiene que crear un nuevo proyecto. Para ello, se selecciona en el menú principal *File>New>Project>Android Project*. A continuación aparecerá una ventana como la de la Figura I-2 en la que se escribirá el nombre del proyecto, del paquete, de la actividad y de la aplicación.

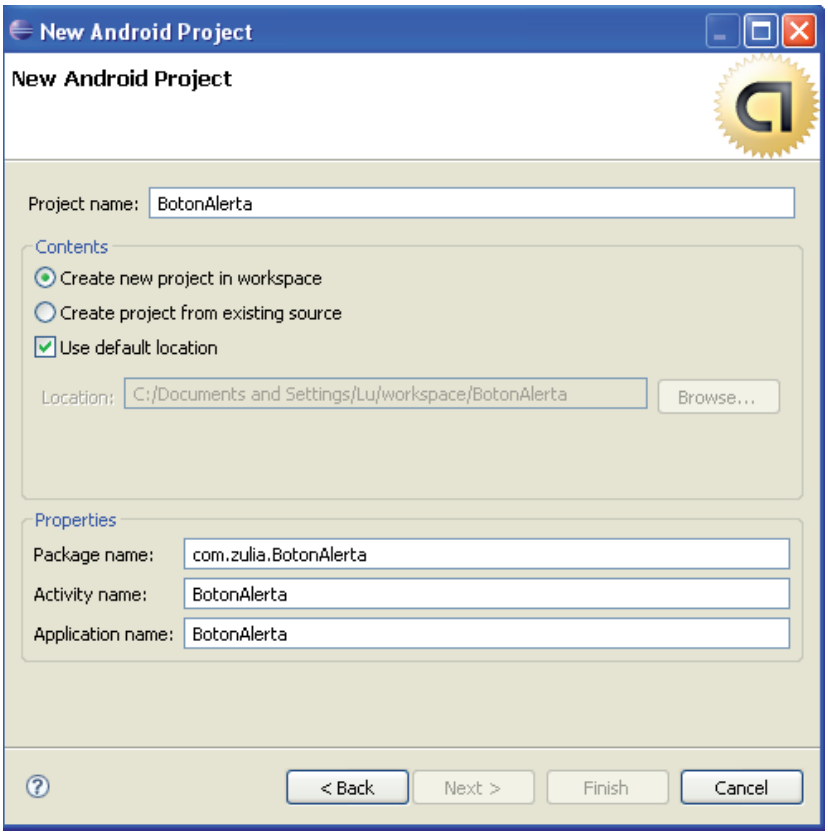


Figura I-2: Crear un nuevo proyecto en Android.

Eclipse genera automáticamente una serie de archivos, entre los cuales los más importantes son:

- El directorio *src* contiene todos los ficheros fuente del proyecto:

  - *BotonAlerta.java*: clase principal que contiene el método *onCreate* y que se puede editar.

  - *R.java*: fichero generado automáticamente que no debe ser modificado.

- El directorio res es donde se encuentran los archivos de recursos del proyecto:

  - Carpeta *layout>main.xml*: información XML para la creación de vistas.

  - Carpeta *values>strings.xml*: información XML para constantes.

  - *AndroidManifest.xml*.

Es muy importante comprender para qué sirve el *Andoird Manifest*. Este fichero informa a Android sobre los componentes de la aplicación, es decir, contiene la información que debe tener el sistema antes de ejecutar el código de cualquier aplicación. Por ejemplo el nombre del paquete, los componentes, los permisos que necesita la aplicación o que necesitan otros para interactuar con los componentes de la aplicación, librerías requeridas...

Las actividades, servicios y proveedores de contenidos que no se declaran en el manifiesto no son visibles para el sistema y, en consecuencia, nunca se ejecutan. Sin embargo, los receptores de difusión pueden ser declarados en el manifiesto o se pueden crear dinámicamente en el código (como objetos *BroadcastReceiver*).

Más adelante se explicará con más detalle en qué consisten los archivos *main.xml* y *strings.xml*.

Si se abre el archivo *BotonAlerta.java*, el código generado automáticamente por Eclipse será el mostrado en la Figura I-3.

```
package amilab.zulia.BotonAlerta;

import android.app.Activity;
import android.os.Bundle;

public class BotonAlerta extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Figura I-3: Código generado automáticamente por Eclipse al crear una nueva clase.

Si se usa Eclipse Galileo es necesario crear un ADV (Android Device Virtual) para poder simular la aplicación realizada. Para ello hay que ir a la pestaña *Windows* y seleccionar *Preferences*, o cliquear directamente en un icono con forma de teléfono. Dependiendo de lo que se quiera hacer se elegirá una u otra opción, por ejemplo para usar mapas se necesita la opción que pone "Google APIs".

I.III. Funcionamiento general de una actividad

La clase con la que se va a trabajar inicialmente es *BotonAlerta.java*, que como se ve en la Figura I-3 es una actividad (desciende de la clase *Activity*). Esta clase tiene un método *onCreate* que será el que se ejecute cuando se inicie la aplicación. En él debe figurar el código necesario para ejecutar el diseño XML de los archivos mencionados en el apartado anterior.

La sentencia *super.onCreate(savedInstanceState)* crea una ventana nueva en la pantalla de Android. Por otro lado, la sentencia *setContentView(R.layout.main)* añade a la actividad que hay creada en esta clase toda la información aportada por el archivo XML indicado (en este caso *main.xml*).

I.IV. Crear un botón y añadirle funcionalidad

Ya se tiene una ventana en la que se pueden introducir vistas, así que ahora hay que indicar a la disposición de la ventana que vistas se van a utilizar.

Siempre es importante trabajar con nombres significativos, por eso antes de comenzar se recomienda cambiar el nombre *main.xml* por *boton.xml* mediante botón derecho, *Refactor*>*Rename*. Como anteriormente se tenía asignado *main.xml*, en *BotonAlerta* hay que cambiar *R.layout.main* por *R.layout.boton*.

Al abrir *boton.xml* se puede observar que ya viene creado una disposición lineal con orientación vertical (*LineraLayout*) y una vista (*TextView*) que contiene un texto (véase Figura I-4).

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <TextView
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:text="@string/hello" />

</LinearLayout>
```

Figura I-4: Disposición lineal en XML con una vista que contiene un texto.

Como inicialmente no queremos imprimir ningún texto por pantalla, hay que eliminar el *TextView* y crear un botón tal y como se muestra en la Figura I-5.

```
<Button
       android:id="@+id/botonParaAlerta"
       android:layout_width="fill_parent"
       android:layout_height="80px"
       android:layout_marginTop="20dip"
       android:text="PULSAR"
       android:textSize="15pt"
       android:gravity="center" />
```

Figura I-5: Definición de un botón con sus parámetros en XML.

De esta forma se define una clase *Button*, dentro de la disposición definida, que ya anteriormente en BotonAlerta.java ha sido asignado a la actividad principal.

Los parámetros "width" y "height" indican el tamaño del botón y son obligatorios, el resto son opcionales. En este caso se han añadido unos cuantos parámetros más: "marginTop" es el margen que hay desde la parte superior de la pantalla hasta el botón, "text" es el texto del botón y "textSize" su tamaño, y "gravity" indica que en este caso el botón estará centrado. Por último, "id" es un identificador del botón para poder referenciar al

botón que se añadirá en el método *onCreate* de la actividad principal.

A continuación se añade un botón en *BotonAlerta* y se le asigna el botón creado anteriormente en XML mediante el "id" (véase Figura I-6).

```
Button boton = (Button) findViewById(R.id.botonParaAlerta);
```

Figura I-6: Asignación de un botón creado en XML al botón usado en una clase.

Por último hay que asignarle funcionalidad al botón. Para ello se le asocia una función de escucha, que se ejecutará cuando el botón sea pulsado. La función tiene que ser de tipo *onClickListener*. Dentro de ella se debe incluir el método *onClick*, donde se introducirá el código que se debe ejecutar cada vez que se pulse el botón (véase Figura I-7).

```
boton.setOnClickListener(new View.OnClickListener()
{
    public void onClick(View v)
    {
        // Código que se ejecutará cada vez que se pulse el botón
    }
});
```

Figura I-7: Asignación de funcionalidad a un botón.

I.V. Crear una ventana y pasar de una ventana a otra

Hasta ahora se ha creado el "botón del pánico" y se ha explicado cómo darle funcionalidad. Como se ha mencionado anteriormente, cuando el usuario pulse el botón aparecerá un mensaje tranquilizador avisando a dicho usuario de que en seguida vendrán a recogerle. En este apartado se va a explicar cómo se crea la segunda pantalla y cómo se pasa de la primera pantalla en la que se verá el botón a la segunda pantalla en la que se verá el mensaje de texto.

Inicialmente hay que crear una nueva clase, que en este caso se llamará *SmsPosicion.java* (también será una actividad, igual que *BotonAlerta.java*), y su correspondiente archivo XML llamado *sms.xml*. Este último fichero lo modificamos simplemente para añadir algún parámetro al *TextView* y cambiar el texto que se mostrará por pantalla (véase Figura I-8).

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="vertical"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent">

     <TextView
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_marginTop="20dip"
          android:text="Quédese quieto, enseguida vendrán a buscarle"
          android:textSize="10pt"
          android:gravity="center_horizontal"/>

</LinearLayout>
```

Figura I-8: Disposición lineal en XML con una vista que contiene un texto para la clase *SmsPosición*.

En algunas ocasiones, en lugar de asignar el texto directamente es preferible que se obtenga de una variable, por ejemplo si la cadena de texto va a ser constante o para evitar que dicha cadena sea modificada por alguna función de la aplicación. Para ello hay que ir al directorio *res* y modificar el archivo *strings.xml* de la carpeta *values* (véase apartado I.II).

Inicialmente hay una variable *String* creada que corresponde al título de la aplicación, y aparece en la parte superior de la pantalla del terminal cuando se ejecuta el programa. Para crear otra variable se crea un <*string*>, y se le asigna un nombre (hará las veces de identificador) y un valor (véase Figura I-9).

```
<string name="textoPantalla"> Quédese quieto, enseguida vendrán a buscarle.</string>
```

Figura I-9: Constante que contiene una cadena de texto.

Una vez que se ha creado la variable hay que asignarla al cuadro de texto (*TextView*) que mostrará dicho mensaje por pantalla. Para ello hay que modificar en *sms.xml* el parámetro "text" tal y como se muestra en la Figura I-10.

```
android:text="@string/textoPantalla"
```

Figura I-10: Asignación de una constante a un cuadro de texto.

Esto indica que el parámetro "text" tiene que tomar el valor del *string* cuyo identificador es *textoPantalla*.

Hasta ahora se tiene una actividad *BotonAlerta* que muestra por pantalla un botón, el

cual realizará alguna acción cuando sea pulsado; y una segunda actividad *SmsPosicion* que de momento no hace nada.

El siguiente paso es hacer que, cuando se pulse el botón, se pase de la ventana creada por *BotonAlerta* a la ventana creada por *SmsPosicion*.

Para pasar de una ventana a otra hay que pasar de una actividad a otra actividad, y eso se hace creando una intención (*intent*) con el cual se cargará la clase *SmsPosicion.java*. Después se asigna a la actividad la intención que se ha creado y por último se termina la ejecución de la actividad llamando al método *finish*. De esta forma, aparecerá por pantalla la nueva asignación del intento, es decir, la nueva ventana creada por la nueva actividad. Éste será el código que habrá que introducir dentro del método *onClick* para que se ejecute cuando el botón sea pulsado (véase Figura I-11).

```
Intent intent = new Intent();
intent.setClass(botonAlerta.this, SmsPosicion.class);
startActivity(intent);
finish();
```

Figura I-11: Iniciar una nueva actividad mediante una intención

Para poder crear muchos de los objetos mencionados anteriormente es necesario incluir una serie de librerías. Se recomienda añadirlas automáticamente mediante *control+shift+O*.

Por último hay que añadir la nueva actividad creada en el archivo general XML llamado *AndroidManifest.xml*, del cual se ha hablado en el apartado I.II. Para ello, entre las etiquetas <*application*> y <*application*> se incluye el código que se muestra en la Figura I-12.

```
<activity android:name=".SmsPosicion"android:label="@string/app_name">
      <intent-filter>
          <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
</activity>
```

Figura I-12: Declaración de una actividad en el *AndroidManifest*.

El funcionamiento de la aplicación, hasta el momento, se puede observar en la figuras mostrada a continuación. En la Figura I-13 - a) se puede apreciar el botón correspondiente a la pantalla inicial y en la Figura I-13 - b) se muestra la segunda pantalla con el texto que aparecerá después de pulsar el botón.

(a) (b)

Figura I-13: Captura de la aplicación antes (a) y después (b) de pulsar el botón.

I.VI. Sistemas basados en localización

Location-based services (LBS) es un término que se utiliza para describir las diferentes tecnologías usadas para encontrar la localización actual del dispositivo. Los dos elementos LBS principales son:

LocationManager. Provee servicios basados en localización, como por ejemplo obtener la ubicación actual o establecer alertas de proximidad para la detección de movimiento dentro y fuera de un área específica.

LocationProviders. Cada uno representa una tecnología de localización diferente usada para determinar la ubicación actual del dispositivo, como por ejemplo el GPS.

Antes de poder utilizar el *LocationManager* es necesario añadir un permiso al *AndroidManifest* para soportar el acceso al LBS hardware. Puede escribirse o antes de la etiqueta <*application*> o después de la etiqueta <*application*> (véase Figura I-14).

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Figura I-14: Definición de un permiso para soportar el acceso al LBS hardware.

El acceso al sistema de localización (dentro de la función *onCreate*) se puede observar en la Figura I-15.

```
String context = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager)getSystemService(context);
```

Figura I-15: Acceso al sistema de localización.

`LOCATION_SERVICE` es una constante que se usa con *getSystemService* para recuperar un LocationManager que controle la actualización de la localización.

De manera general, se puede instanciar un proveedor específico (en este caso GPS) tal y como se muestra en la .Figura I-16

```
String providerName = LocationManager.GPS_PROVIDER;
LocationProvider gpsProvider;
gpsProvider = locationManager.getProvider(providerName);
```

Figura I-16: Obtención del proveedor de localización GPS.

`GPS_PROVIDER` es una constante que contiene el nombre del proveedor de localización GPS. El método *getProvider* devuelve la información asociada con un proveedor de localización.

Por otro lado, si en vez de obtener un proveedor específico nos interesa obtener una lista de los proveedores habilitados, se haría con el código de la Figura I-17.

```
boolean enabledOnly = true;
List<String> providers = locationManager.getProviders(enabledOnly);
```

Figura I-17: Obtención de todos los proveedores de localización disponibles.

Sin embargo, para esta aplicación vamos a escoger un único proveedor que cumpla una serie de requisitos. Los LBS permiten elegir el tipo de proveedor de localización que se quiere usar y además se pueden imponer una serie de criterios que debe cumplir dicho proveedor. Para especificar estos requisitos (precisión, no es necesaria la altitud ni el rumbo, que el proveedor esté autorizado a tener un coste asociado, bajo consumo de energía…) se usa la clase *Criteria*. El código usado puede verse en la Figura I-18.

```
// Criterios que debe cumplir el proveedor

criteria.setAccuracy(Criteria.ACCURACY_FINE);
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setCostAllowed(true);
criteria.setPowerRequirement(Criteria.POWER_LOW);

// Devuelve el nombre del mejor proveedor que cumple los criterios anteriores
String provider = locationManager.getBestProvider(criteria, true);
```

Figura I-18: Obtención de un proveedor que cumpla unos requisitos.

Para obtener la última ubicación conocida por el proveedor se utiliza el código de la Figura I-19.

```
Location location = locationManager.getLastKnownLocation(provider);
```

Figura I-19: Obtención de la última ubicación conocida.

Se debe tener en cuenta que *getLastKnownLocation* no solicita la ubicación al proveedor para actualizar la posición actual. Si el dispositivo no se ha actualizado recientemente la posición actual de este valor puede estar fuera de fecha.

Por último, se usa el método *requestLocationUpdates* para obtener las actualizaciones cada vez que cambia la ubicación actual. Al método se le pasa el proveedor, un tiempo en milisegundos que indica el mínimo intervalo de tiempo para notificaciones, una distancia en metros que indica el mínimo intervalo de distancia para notificaciones y un *LocationListener*. En este caso, el tiempo mínimo es de media hora y la distancia mínima 20 metros. Se han cogido estos valores porque la idea es que al pulsar el botón y enviar el SMS con la ubicación de usuario, el receptor del SMS ayude de alguna forma al usuario o vaya recogerlo si es necesario. Para ello necesitamos saber si desde que envió el SMS se ha desplazado a algún sitio. El código necesario puede verse en la Figura I-20.

```
// Actualizar ubicacion
locationManager.requestLocationUpdates(provider,1800000,20,locationListener);
```

Figura I-20: Obtención de actualizaciones cuando varía la localización actual.

LocationListener se usa para recibir notificaciones de *LocationManager* cuando la ubicación ha cambiado. Los métodos de *LocationListener* son llamados si éste ha sido registrado con el gestor de servicios de localización usando el método anterior.

Ahora falta añadir el código para el *LocationListener* (véase Figura I-21), que se incluirá por ejemplo después del método *onCreate*.

```
private final LocationListener locationListener = new LocationListener(){

        public void onLocationChanged(Location location) {
                // Actualizar la aplicación basada en la nueva ubicación
                updateWithNewLocation(location);

        }

        public void onProviderDisabled(String provider){
                // Actualizar la aplicación basada en la nueva ubicación
                updateWithNewLocation(location);

        }

        public void onProviderEnabled(String provider){ }

        public void onStatusChanged(String provider,int status,Bundle extras){ }

};
```

Figura I-21: Definición de *LocationListener* para cuando la ubicación actual cambia.

El método *updateWithNewLocation* se encargará de actualizar el texto del SMS que se enviará con la información de la nueva ubicación. Será explicado en el siguiente apartado.

No hay que olvidar añadir cuando sea necesario las librerías oportunas, usando *control+shift+O*.

I.VII. Obtener el nombre de una calle a partir de sus coordenadas

En el método *updateWithNewLocation* se tiene que obtener la longitud y la latitud de la localización, para poder obtener con esas datos la dirección correspondiente.

Toda esa información será guardada en un *String*, que será el texto del SMS que se enviará.

El esquema general de *updateWithNewLocation* se muestra en la Figura I-22.

El primer *String* (*latLong*) contendrá las coordenadas y el segundo (*direc*) la dirección con esas coordenadas. La variable *message* es una variable global que contendrá el texto del SMS a enviar.

```
String latLong;
String direc= "Dirección no enconrtada";

if (location != null) {
    // Actualizar valores con la nueva ubicación
}

si no
    latLong = "Localización no encontrada";

message = "¡ME HE PERDIDO!\n\n Estoy en:\n" + direc + "\n\n Y mis coordenadas son:\n"
          + latLong + "\n\n";
```

Figura I-22: Esquema general del método *updateWithNewLocation*.

A continuación se explicará el código que actualizará los valores con la nueva ubicación, y que se introducirá justo después de haber comprobado que la variable que contiene la localización no tiene un valor nulo.

Inicialmente se obtienen las coordenadas a través de la localización (véase Figura I-23).

```
double latitude = location.getLatitude();
double longitude = location.getLongitude();

latLong = "Lat: " + latitude + "\nLong: " + longitude;
```

Figura I-23: Obtención de la latitud y la longitud.

A continuación se obtiene la dirección correspondiente a dichas coordenadas. Para ello se usa la clase *Geocoder*, (véase Figura I-24) que puede encontrar una dirección a través de unas coordenadas y viceversa. A esta clase se le pasa un *Context* (en este caso el contexto de la clase en si) y un *Locale*, que representa un idioma/país y dicta las convenciones particulares para presentar la información.

```
Geocoder latCalle = new Geocoder(this, Locale.getDefault());
```

Figura I-24: Creación de un geocodificador.

La función que hace la conversión de coordenadas a una dirección es *getFromLocation*, y devuelve una lista de objetos *Address* con todas las posibles soluciones encontradas. Los valores devueltos pueden ser obtenidos por medio de una búsqueda de red. A esta función se le pasa la latitud, la longitud y el número de resultadas máximos que quiere que muestre (en este caso sólo queremos un resultado).

En la Figura I-25 se muestra el código, donde se puede apreciar cómo se obtiene el nombre de la localidad, el código postal, el nombre del país.

```
try {
       List<Address> adLatCalle = latCalle.getFromLocation(latitude,longitude,1);

       StringBuilder sbLatCalle = new StringBuilder();

       if (adLatCalle.size() > 0) {

           Address address = adLatCalle.get(0);

           for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
                sbLatCalle.append(address.getAddressLine(i)).append("\n");

           if (address.getLocality() != null)
                sbLatCalle.append(address.getLocality()).append("\n");

           if (address.getPostalCode() != null)
                 sbLatCalle.append(address.getPostalCode()).append("\n");

            sbLatCalle.append(address.getCountryName());

            direc = sbLatCalle.toString();
         }

       } catch (IOException ioe) {}
        catch (IllegalArgumentException iae) {}
```

Figura I-25: Obtención de una dirección a partir de una coordenada.

Como en los apartados anteriores, de nuevo habría que añadir una serie de librerías.

I.VIII. Enviar SMS

Igual que en el caso de los sistemas basados en localización, lo primero que hay que hacer es añadir el permiso necesario para enviar mensajes cortos (SMS) en el *AndroidManifest* (véase Figura I-26).

```
<uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
```

Figura I-26: Definición de un permiso para enviar mensajes cortos (SMS).

Inicialmente se declaran cuatro variables globales (todas son strings), que contendrán el texto del SMS (mencionado anteriormente), el número de teléfono del destinatario, y dos acciones; una para SMS enviados y otra para SMS entregados (véase Figura I-27).

```
private String message;
private final String phoneNumber = "5556";

private final String SENT = "SMS_SENT";
private final String DELIVERED = "SMS_DELIVERED";
```

Figura I-27: Creación de variables para el texto del SMS, el número de teléfono del destinatario, una acción de SMS enviado y otra de SMS entregado.

Para hacerlo de manera más clara se han creado dos funciones. La primera es *controlSMS*, se llama en el método *onCreate* (es decir, se ejecuta en cuanto se crea la actividad *SmsPosicion*) y como su propio nombre indica se encarga de controlar el envío y la entrega de SMS. La segunda función es *sendSMS* y es la que se encarga de realizar el envío propiamente dicho.

En la Figura I-28 se incluye el código de este segundo método, y después se comentará su funcionamiento.

```
private void sendSMS(String phoneNumber, String message){

   SmsManager sms = SmsManager.getDefault();

   PendingIntent sentPI = PendingIntent.getBroadcast(this, 0, new Intent(SENT), 0);

   PendingIntent deliveredPI = PendingIntent.getBroadcast(this, 0, new Intent(DELIVERED), 0);

    try {

         sms.sendTextMessage(phoneNumber, null, message, sentPI, deliveredPI);

     } catch (NullPointerException npe) {}
       catch (IllegalArgumentException iae){}
}
```

Figura I-28: Código del método *sendSMS* para realizar el envío de SMS.

Para el envío de SMS es necesario tener un *SmsManager*, que es un gestor de operaciones SMS como enviar datos, textos y mensajes SMS. Para obtener una instancia por defecto se llama al método estático *getDefault*.

La acción en sí de enviar un SMS se puede ver en la Figura I-29.

```
sms.sendTextMessage(phoneNumber, null, message, sentPI, deliveredPI);
```

Figura I-29: Código para enviar un SMS.

A este método hay que pasarle (por orden de izquierda a derecha) el destinatario del mensaje, el centro de servicio de direcciones (si es *null* coge por defecto SMSC), el cuerpo

del mensaje a enviar y dos *PendingIntent* (*sentPI* y *deliveredPI*).

Como se mencionó en el apartado 6.1.4 del capítulo 6, un *PendingIntent* es una descripción de una intención y un acción a realizar. En este caso se crean dos instancias de esta clase con *getBroadcast*, para realizar una emisión en difusión. Más adelante se explicará con más detalle la función de estos dos *PendingIntent*.

En la Figura I-30 se puede observar el código de la otra función antes mencionada, y a continuación se comentará su funcionamiento.

```
private void controlarSMS() {

   // Cuando el SMS ha sido enviado...
   registerReceiver(new BroadcastReceiver(){

      public void onReceive(Context arg0, Intent arg1) {

         switch (getResultCode()){

           case Activity.RESULT_OK:
              Toast.makeText(getBaseContext(), "SMS enviado", Toast.LENGTH_SHORT).show();
              break;
           case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
              Toast.makeText(getBaseContext(), "Fallo genérico", Toast.LENGTH_SHORT).show();
              break;
           case SmsManager.RESULT_ERROR_NO_SERVICE:
              Toast.makeText(getBaseContext(), "Sin servicio", Toast.LENGTH_SHORT).show();
              break;
           case SmsManager.RESULT_ERROR_NULL_PDU:
              Toast.makeText(getBaseContext(), "PDU vacía", Toast.LENGTH_SHORT).show();
              break;
           case SmsManager.RESULT_ERROR_RADIO_OFF:
              Toast.makeText(getBaseContext(), "Radio apagada", Toast.LENGTH_SHORT).show();
              break;
           }
      }
   }, new IntentFilter(SENT));



   // Cuando el SMS ha sido entregado...
   registerReceiver(new BroadcastReceiver(){

      public void onReceive(Context arg0, Intent arg1) {

        switch (getResultCode()){

           case Activity.RESULT_OK:
               Toast.makeText(getBaseContext(), "SMS entregado", Toast.LENGTH_SHORT).show();
                break;
           case Activity.RESULT_CANCELED:
               Toast.makeText(getBaseContext(), "SMS no entregado", Toast.LENGTH_SHORT).show();
               break;
         }
       }
    }, new IntentFilter(DELIVERED));
 }
```

Figura I-30: Código del método controlarSMS para controlar el envío y la entrega del SMS.

El código de las dos funciones mencionadas anteriormente utiliza un objeto *PendingIntent* (*sentPI*) para supervisar el proceso de envío. Cuando un mensaje SMS es enviado, el primer evento *BroadcastReceiver* de *onReceive* se activará. Ahí es donde

se puede comprobar el estado del proceso de envío. El segundo objeto *PendingIntent* (*deliveredPI*) supervisa el proceso de entrega. El segundo evento *BroadcastReceiver* de *onReceive* se activará cuando el SMS se haya entregado correctamente. Al probarlo en el emulador es imposible saber si el SMS fue entregado, pero en un terminal real si funcionará.

A *registerReceiver* se le pasa un *BroadcastReceiver* y un *IntentFilter* (véase el apartado 6.1.4 del capítulo 6), y lo que hace es registrar un *BroadcastReceiver* para ser ejecutado en el hilo de la aplicación principal. El método *onReceive* será llamado con cualquier *broadcast* recibido cuya acción coincida con el filtro (en este caso cualquier intención cuya acción sea `SENT` o `DELIVERED`).

El método *getResultCode* recupera el código actual según lo establecido por el receptor.

Un *toast* es una vista que contiene un mensaje rápido para el usuario. Cuando la vista se muestra al usuario, aparece como una vista "flotante" sobre la aplicación.

Lo último que queda por hacer es añadir la llamada al método *sendSMS* en *onLocationChanged* para que se envíe un SMS cada vez que la localización cambie. Hay que tener cuidado y colocar dicha llamada después de la llamada a *updateWithNewLocation*, para que se actualice el cuerpo del SMS antes de ser enviado.

I.IX. Probar la aplicación en el emulador

Para probar la aplicación con el emulador hay que mandarle una señal GPS a éste. Para ello se puede usar la perspectiva DDMS que ofrece Eclipse (pestaña *Windows*, opción *Open Perspective*), la cual tiene un apartado en el que se puede enviar la latitud y la longitud del GPS al emulador. Lo malo es que esta opción no siempre funciona.

La otra alternativa es usar el comando *telnet* para conectarse con el emulador. *Telnet* (TELecommunication NETwork) es un protocolo de red que sirve para acceder a otra máquina mediante una red. En la Figura I-31 se muestra como usar el comando *telnet* en una consola de comandos. Evidentemente, antes de ejecutar dicho comando el emulador debe de estar ejecutándose. El número de puerto del emulador al que hay que conectarse

suele ser 5554, pero puede verse en la parte superior del *Android Emulator*.

```
$ telnet localhost 5554
```

Figura I-31: Conectarse con el emulador.

Una vez que estemos en la *Android Console* se usan los comandos *geo* y *fix* para simular la señal GPS, añadiendo a continuación primero la longitud y luego la latitud (véase Figura I-32). Cada vez que se quiera cambiar la ubicación simplemente hay que copiar la línea mostrada en la Figura I-32 y cambiar los valores de la longitud y la latitud.

```
$ geo fix -3.456435 40.475644
```

Figura I-32: Simular señal GPS en el emulador.

Para comprobar que se envían los SMS es necesario abrir otro emulador. Para ello hay que ir a la carpeta del SDK, ir a *Tools* y ejecutar *Emulator*. Es importante haber puesto en la variable *phoneNumber* el valor 5556 (el nuevo emulador).

En el emulador 5554 ejecutamos la aplicación *BotonAlerta*, y nada más haber enviado un SMS se verá lo que se muestra en la Figura I-33 - (a).

(a) **(b)** **(c)**

Figura I-33: Prueba de la aplicación en el emulador. a) Notificación de mensaje enviado. b) Notificación de mensaje recibido. c) Mensaje recibido.

El emulador 5556 es el que recibe los SMS. Efectivamente, en la Figura I-33 - (b), en la parte superior izquierda de la pantalla del teléfono se puede apreciar la notificación de la llegada de un SMS nuevo.

Si se abre el SMS recibido se podrá comprobar que aparece la dirección donde nos encontramos y sus correspondientes coordenadas, tal y como se muestra en la Figura I-33 - (c).

Anexo II. Medidas del GPS del terminal sobre vértices del centro de Madrid

II.I. Hojas de datos de los vértices de la Red Topográfica de Madrid

A continuación se muestran las hojas de los vértices estudiados y las medidas realizadas sobre dichos vértices. Estas hojas han sido obtenidas mediante la Red Topográfica de Madrid, Área de Gobierno de Urbanismo, Vivienda e Infraestructura. En las tablas también se muestra la medida de referencia pero en coordenadas geográficas. Éstas corresponden a las coordenadas ETRS89 de la hoja de datos (coordenadas UTM).

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 6949	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
440393.417	4474192.195	658.292	440283.999	4473984.696	709.317

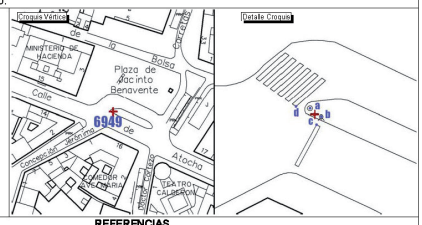
SITUACIÓN:
Empotrado en bordillo de isleta, en una de las esquinas de la plaza Jacinto Benavente embocando la calle Atocha y la calle Concepción Jerónima.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

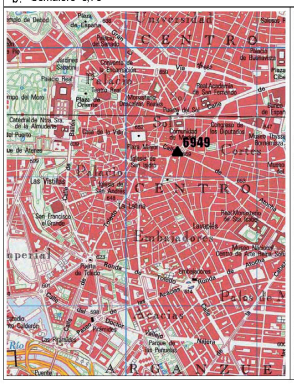
7678

9351



REFERENCIAS

a: Señal 0,98
b: Semáforo 0,73
c: Esquina marca vial 3,14
d: Esquina marca vial 2,43



Vértice: 6949		
Longitud: -3.7038505295333		
Latitud: 40.41434305450074		
Medida	**Longitud**	**Latitud**
1	-3.7038333	40.41445
2	-3.7038333	40.4147666
3	-3.7038333	40.41438333
4	-3.70385	40.4143
5	-3.7038	40.4144166

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 7678	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
440436.037	4474168.434	658.583	440326.618	4473960.935	709.607

SITUACIÓN:
Empotrado en bordillo de acera, en una de las esquinas de la Plaza de Jacinto Benavente, embocanco la calle Atocha, frente a la cafeteria Maestro Churrero y las taquillas del teatro Calderón.

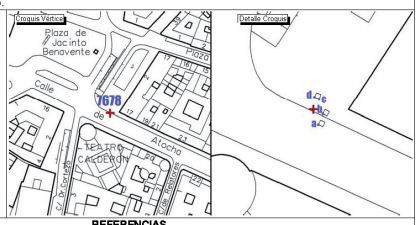
SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

7679

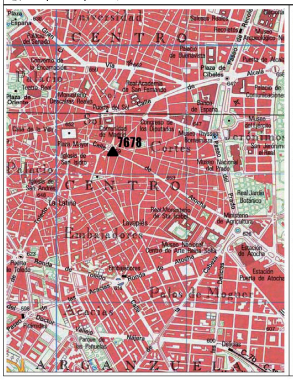
6949

9351



REFERENCIAS

a: Esquina sumidero 0,88
b: Esquina arqueta 0,69
c: Esquina registro alumbrado 1,17
d: Esquina registro alumbrado 1,24



Vértice: 7678		
Longitud: -3.7033460021570517		
Latitud: 40.414132060873285		
Medida	Longitud	Latitud
1	-3.70345	40.4140166
2	-3.70338333	40.4141
3	-3.70338333	40.4141
4	-3.70338333	40.4141166
5	-3.70341666	40.414

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 8061	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
440749.071	4474799.491	656.624	440639.663	4474591.983	707.643

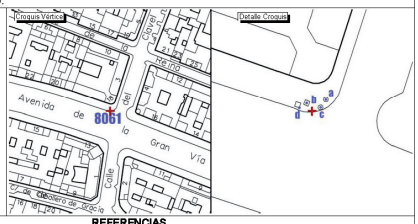
SITUACIÓN:
Empotrado en bordillo de acera de la C/ Gran Vía frente al nº 18 esquina con la C/ del Clavel.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

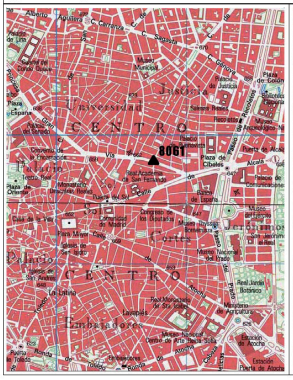
8062

8146



REFERENCIAS

a: Esquina reg. Telefónica 1.91
b: Boca de riego 0.44
c: Bolardo 0.44
d: Esquina alcorque 0.78



Vértice: 8061		
Longitud: -3.699715419825762		
Latitud: 40.419839228812165		
Medida	Longitud	Latitud
1	-3.69961666	40.4197
2	-3.6998	40.41985
3	-3.69988333	40.41975
4	-3.6998666	40.4197333
5	-3.6998666	40.4197333

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 8070	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
439759.443	4474648.609	639.416	439650.031	4474441.116	690.435

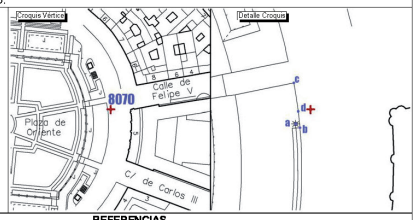
SITUACIÓN:
Empotrado en esquina de adoquín en la Plaza de Oriente entrando por la C/ Felipe V.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

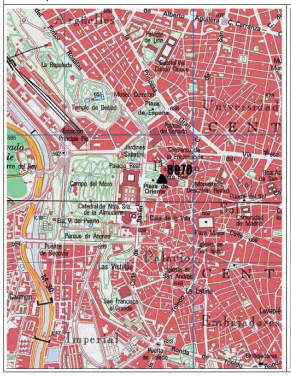
7773

8071



REFERENCIAS

a: Farola 3.28
b: Esquina bordillo 3.67
c: Esquina bordillo 4.76
d: Normal a bordillo 1.97



Vértice: 8070		
Longitud: -3.711365636601412		
Latitud: 40.41840896665635		
Medida	Longitud	Latitud
1	-3.7115333	40.41848333
2	-3.7116	40.41858333
3	-3.71161666	40.4185666
4	-3.7116	40.41855
5	-3.7115666	40.4185333

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 8072	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
439935.455	4474596.268	638.514	439826.042	4474388.772	689.537

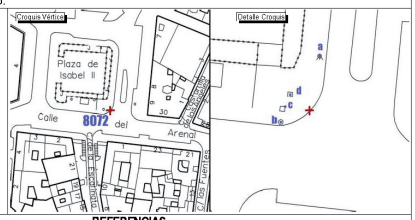
SITUACIÓN:
En el bordillo de la Plaza de Isabel II, en la C/ Arenal.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

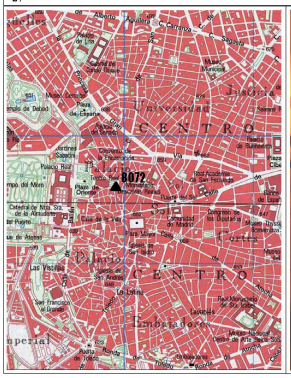
8071

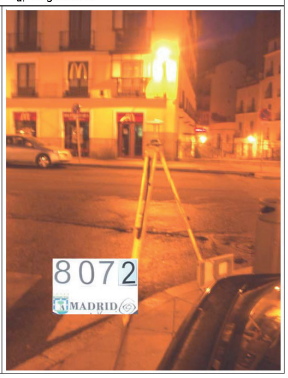
8073



REFERENCIAS

a: Semaforo 5.16
b: Señal 2.74
c: Esquina registro 2.30
d: Registro 3.20





Vértice: 8072		
Longitud: -3.7092861338974226		
Latitud: 40.41795017376093		
Medida	Longitud	Latitud
1	-3.70928333	40.4178666
2	-3.70921666	40.4178166
3	-3.70921666	40.4178166
4	-3.7092333	40.4178
5	-3.7092333	40.4178

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 8082	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
440062.052	4474331.630	651.958	439952.636	4474124.134	702.983

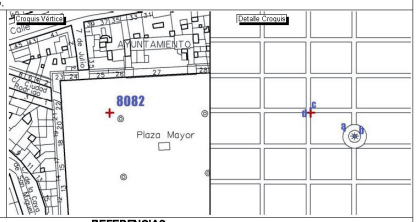
SITUACIÓN:
En la Plaza Mayor, en la esquina del empedrado blanco, entrando desde el acceso del Mercado de la Cebada.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

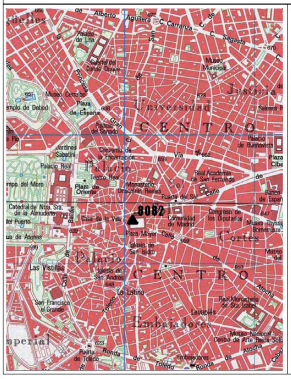
8078

9353



REFERENCIAS

a: Normal a banco 5.85
b: Farola 7.25
c: Esquina empedrado blanco 1.80
d: Esquina empedrado blanco 1.55



Vértice: 8082		
Longitud: -3.7077690633720364		
Latitud: 40.415575336190095		
Medida	Longitud	Latitud
1	-3.70765	40.4157
2	-3.7077333	40.4156333
3	-3.7077	40.4156
4	-3.70771666	40.4155333
5	-3.7077666	40.41555

madrid
ÁREA DE GOBIERNO DE URBANISMO, VIVIENDA E INFRAESTRUCTURAS

RED TOPOGRAFICA DE MADRID

VÉRTICE: 9353	RED G.P.S.	Nº HOJA M.T.N. : 559			
COORDENADAS ED50			COORDENADAS ETRS89		
X	Y	Z Orto.	X	Y	Altitud ELP.
440068.996	4474288.985	651.917	439959.579	4474081.490	702.941

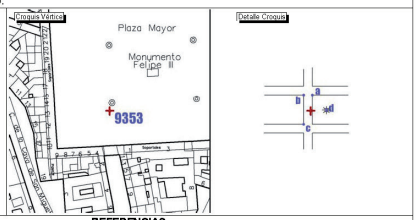
SITUACIÓN:
En la Plaza Mayor, dando vista a la C/ Toledo.

SEÑAL: Clavo Reglamentario.

VÉRTICES VISIBLES

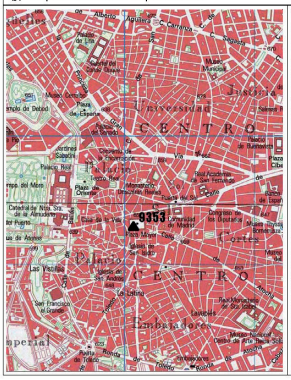
9352

6948



REFERENCIAS

a: Esquina cambio color empedrado 4.57
b: Esquina cambio color empedrado 4.60
c: Esquina cambio color empedrado 2.12
d: Normal a bordillo farola 3.22



Vértice: 9353		
Longitud: -3.7076832083183593		
Latitud: 40.415191680088675		
Medida	Longitud	Latitud
1	-3.70778333	40.4152333
2	-3.70778333	40.41525
3	-3.70771666	40.4152333
4	-3.7077666	40.41518333
5	-3.7077	40.41518333

II.II. Equivalencia entre grados y distancias

La distancia que representa un grado de latitud varía poco moviéndose del Ecuador hacia los polos, ya que todos los meridianos tiene aproximadamente el mismo largo. Suponiendo que la Tierra es esférica y todos sus meridianos miden igual que el Ecuador, se puede calcular la longitud de un grado de latitud dividiendo el largo de un meridiano (40.076 km) entre 360. De ahí se obtiene que cada grado de latitud corresponde a 111.325 km.

Sin embargo, la distancia que representa un grado de longitud varía dependiendo de la latitud, dado que los paralelos cada vez se hacen más pequeños a medida que se acercan a los polos (alejándose del Ecuador). La circunferencia de un paralelo ubicado en la latitud X es $\cos(x) * 40.076$ km. Teniendo en cuenta que los 40.076 km representan la circunferencia del Ecuador, la distancia equivalente a un grado de longitud en la latitud X es $\cos(X) * 111.325$ km.

Anexo III. Presupuesto

1) Ejecución Material

- Compra de ordenador personal (Software incluido).....1.100 €
- Compra de terminal HTC Dream libre.....350 €
- Material de oficina.....140 €
- Total de ejecución material.....1.590 €

2) Gastos generales

- 16 % sobre Ejecución Material.....254,4 €

3) Beneficio Industrial

- 6 % sobre Ejecución Material.....95,4 €

4) Honorarios Proyecto

- 720 horas a 15 € / hora.....10.800 €

5) Material fungible

- Gastos de impresión.....200 €
- Encuadernación.....60 €

6) Subtotal del presupuesto

- Subtotal Presupuesto.....12.650 €

7) I.V.A. aplicable

- 16% Subtotal Presupuesto .....2.024 €

8) Total presupuesto

- Total Presupuesto.....14.674 €

Madrid, Febrero de 2010
El Ingeniero Jefe de Proyecto

Fdo.: Lucía Anglés Alcázar
Ingeniera Superior de Telecomunicación

Anexo IV. Pliego de condiciones

Este documento contiene las condiciones legales que guiarán la realización, en este proyecto, de un *sistema de localización y guía para personas con necesidades especiales*. En lo que sigue, se supondrá que el proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dicho sistema. Dicha empresa ha debido desarrollar una línea de investigación con objeto de elaborar el proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente pliego.

Supuesto que la utilización industrial de los métodos recogidos en el presente proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes:

Condiciones generales.

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el proyecto a concurso se reserva el derecho a declararlo desierto.

2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.

3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si este se hubiera fijado.

4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.

5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará

obligado a aceptarla.

6. El contratista tiene derecho a sacar copias a su costa de los planos, pliego de condiciones y presupuestos. El Ingeniero autor del proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.

7. Se abonará al contratista la obra que realmente ejecute con sujeción al proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra se haya ajustado a los preceptos de los pliegos de condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados. Por consiguiente, el número de unidades que se consignan en el proyecto o en el presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con ma-

yores dimensiones cualquier otra parte de las obras, o en general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.

17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.

18. Si el contratista al efectuar el replanteo, observase algún error en el proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del proyecto.

19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.

20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.

21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa, por retraso de la ejecución siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.

22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.

23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad "Presupuesto de Ejecución de Contrata" y anteriormente llamado "Presupuesto de Ejecución Material" que hoy designa otro concepto.

Condiciones particulares.

La empresa consultora, que ha desarrollado el presente proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiendo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo, pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.

2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente proyecto, bien para su publi-

cación o bien para su uso en trabajos o proyectos posteriores, para la misma empresa cliente o para otra.

3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.

4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.

5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del proyecto, nombre del Ingeniero Director y de la empresa consultora.

6. Si el proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.

7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el proyecto inicial del que resulta el añadirla.

8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.

9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este proyecto, deberá comunicarlo a la empresa consultora.

10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente proyecto para la realización de otras aplicaciones.

11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los proyectos presentados por otros.

12. El Ingeniero Director del presente proyecto, será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.

Anexo V.
Artículo publicado

WAI-ROUTES: A ROUTE-ESTIMATION SYSTEM FOR AIMING PUBLIC TRANSPORTATION USERS WITH COGNITIVE IMPAIRMENTS

Lucía Anglés-Alcázar, Pablo A. Haya, Rosa M. Carro
Escuela Politécnica Superior, Universidad Autónoma de Madrid
C/ Francisco Tomás y Valiente 11
28049 Madrid, Spain
lucia.angles@estudiante.uam.es, pablo.haya@uam.es, rosa.carro@uam.es

ABSTRACT
This article presents WAI-Routes, a system whose main goal is to calculate the best route between two coordinates in space using the public transport network of the *Comunidad de Madrid* regional area, taking into account the user's profile. WAI-Routes has been designed bearing in mind mentally disabled users; routes are calculated with the particularities of the members of this group in mind. In order to achieve this, the system contains a variety of filters than can be set the first time the user uses the application or whenever he/she wants to calculate a new route.

KEYWORDS
Location-Aware Applications, Route Estimation System, Cognitive impairments, Public Transportation Service.

1. Motivation

Recent hardware-related advances made in the field of mobile devices, have allowed to support new interaction styles as well as new services, especially those related to information search, access and management. Therefore, new tools can be created, starting either from scratch or from existing ones, to help people with disabilities or handicaps to perform everyday tasks.

The HADA project [1] aims to develop methodologies and tools that integrate adaptive hypermedia and environmental intelligence to facilitate the use of new technologies to users with special and specific needs and, more specifically, to users with cognitive disabilities along with the elderly. The main aim is to offer solutions adapted to the needs and requirements of each individual within different contexts: at home, while travelling and in working/studying environments. According to the Spanish Federation of Down Syndrome [2], 'autonomy and independence necessarily entail the access to a job'. For many people with cognitive disabilities, travelling to their workplace and coming back, can be a major problem that may constitute a great barrier for them to integrate in working environments. Likewise, the inability to use public transport in a safe way, may damage their life quality which can affect not only their studies/work, but also the use of their spare time [3].

WAI (Where Am I) has been developed within the framework of the HADA project. It focuses on facilitating the use of public means of transport of Madrid. More specifically, its main goal is to guide users with cognitive disabilities (such as Down syndrome) from one point to another through the network of public transport in the *Comunidad de Madrid* regional area [4]. The project is divided into two parts:

- WAI-interface: is responsible for managing the interaction between the system and the user, and guides him/her through adaptive and intuitive interfaces.

- WAI-Routes: calculates the most suitable path to go from one place to another, depending on the origin and destination points, as well as to the user's profile.

This article focuses on describing WAI-Routes. WAI-Routes fits into sensitive and location-based computing context research areas. Location-based services are provided and integrated with follow-up services and positioning-based ones [5]. In particular, space-time coordinates are used. To this purpose, users must carry mobile terminals with integrated GPS and Internet access.

WAI-Routes will use the user location (position provided by GPS or entered by the user) and, by means of a visual interface, will assist the user to reach his/her destination place. In addition, it will track the path followed by the user with as many details as possible.

The path to be followed by the user will consist of a number of points. Each point can be a coordinate (latitude, longitude), a street, a subway station, tram station or a RENFE suburban railway station. The whole space considered, is that of the *Comunidad de Madrid* regional area.

Individuals with cognitive limitations have greater difficulty when performing mental tasks than people without limitations. Some cognitive processes affected by such limitations are memorising, troubleshooting, paying attention, reading, linguistics and verbal understanding, as well as mathematical and visual comprehension.

It must be taken into account that the cognitive processes affected by cognitive limitations can be considerably diverse for each individual. They can vary depending not only on those limitations, but also on the limitation degree. Moreover, some users may have difficulties in catching transfer trains, others may find it difficult to catch RENFE suburban trains, others may require stations adapted to people with reduced mobility, etc. Therefore, WAI-Routes must consider each user's specific needs for the creation of the route to be proposed. The mechanism used to customize each query will be further explained in section 3.2.1.

2. Related work

Reppening and Loannidou [6] raise the use of a Mobility Agent that serves as an information central axis between buses, carers and users. The Mobility Agent provides destination options, receives user choices, analyzes GPS information about the buses and compares the traveller's location to the bus' location. WAI-Routes follows a similar approach to this in some aspects, but there are two substantial differences between both tools. On the one hand, regarding the path to be followed, in WAI-Routes the position of the means of transport is not taken into account, but the user's position indeed is. On the other hand, in Mobility Agent, the carer decides how the route will be shown to the user, while in WAI-Routes, the carer (if existent) can modify the user's preferences when calculating the path chosen and can be aware of the paths followed by the user, but it is the system itself which automatically decides the route to be followed and how the information is displayed.

Chang et al. [7] and Tsai [8] propose the use of a PDA that displays addresses and instructions in real-time, showing photos in a Web browser, redirected by QR decoded code labels that have been photographed by the PDA's integrated camera. Conversely, WAI-Routes locates the user's position from the PDA's GPS and guides the user following a graph of the public transport network stored in a server.

Patterson et al. [9] present a system that does not require an explicit entry from the user, but rather generates a suggestion for a route plan and a destination prediction in a non-supervised way, considering the user's previous observations. The system registers GPS signals in places where the user has stopped for a long time, and it learns where the user wants to go, but it does not provide destinations to which the user has not travelled. In contrast, WAI-Routes does not learn routes; it requires information specifying the criteria to be taken into account for the calculation of the route, from the person responsible for the system, depending on the special needs of each user; this information will be translated into filters to be used at runtime.

Finally, there are several websites that provide information about possible routes from one place to another when using public transport, such as the *Comunidad de Madrid* regional area Transport Information System [4] or Transports Metropolitans de Barcelona [10]. These websites are not adaptive in the sense that routes calculated, consider neither specific/special user features/needs nor other criteria to help people with cognitive limitations, whilst WAI-Routes does (see section 3.2.1).

3. System architecture

WAI-Routes follows the classical Client-Server model. A sketch of the architecture used can be seen in Figure 1.

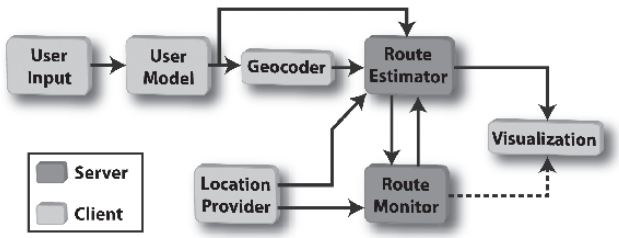


Figure 1. WAI-Routes architecture. Modules in light gray form the client, while modules in dark gray correspond to the functionality integrated within the server.

This model has been chosen mainly because, this way, all the information of the public transport network is on the server. Therefore, if any value of that network should be updated (for example, if a subway line is temporarily out of service), it can be directly updated on the server, without having to update all the terminals whenever something similar happens. The fact that current mobile terminals have the potential to always be connected to Internet, even in the subway network, makes it possible to choose this model. However, it is always possible to make a local copy of the terminal's mostly used information, and to connect to the server from time to time to obtain potential updates, so that the calculation of certain routes does not necessarily require the use of the server.

Following is the description of the main parts of the architecture of WAI- Routes.

3.1. Public transport network

The server stores the public transport network graph, which consists of three main parts: stations, lines, and transfers.

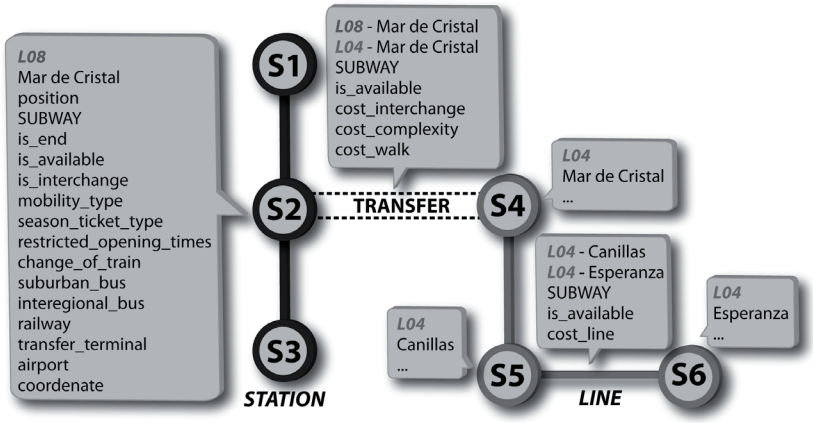


Figure 2. Public transport network graph scheme. Line 8 stations appear in dark gray, while line 4 stations are shown in light gray.

• Stations: contains information on all subway, tram and RENFE-suburban railway stations. The information related to each station is: line type and number; station name; position of the station within the line; type of transport; whether it is the end of the line; whether it is available; whether it is a transfer station; whether it is (totally or partially) adapted to people with reduced mobility; type of season ticket required; whether it has restricted opening hours; whether it is a transfer station; whether it is also a bus station and if so, whether it is interurban or interregional; whether it is also a long-distance railway station; whether it corresponds to an airport terminal (and it is necessary to pay for the corresponding supplement); as well as its coordinates. Each station is identified by the line type and number along with the name of the station.

• Lines: contains information on all links between stations of the same line. Each line consists of: name of the station, along with the line type and number, for the two stations that make up the line; type of transport; whether it is available; and the cost associated to the trip between two stations. Each trip between two stations, as is the case of lines, is identified by the union of two station identifiers.

• Transfers: contains information on all joints between different line stations. The information that defines each transfer is: name of the station with line type and number, for the two stations forming the transfer; type of transport; whether it is available; and three transfer related costs (see section 3.1.1). Each transfer is identified by the union of two station identifiers.

3.1.1. Graph-related costs

The graph-related costs are used to estimate the best route. As mentioned previously, there are two types of costs: **line costs** and **transfer costs**.

The **line cost** is an estimation of the cost of passing from one station to another within the same line (same type and line number). For example, if we look at Figure 2, the cost between L04 line stations can be greater than the cost between L08 line stations, if in the latter, trains are newer and faster. Or, for example, the cost between stations *Cruz del Rayo* and *Concha Espina* (L09) will be greater than the cost between the stations of *Colón* and *Serrano* (L04), because in the first line the distance between stations is much greater.

The **total transfer cost** is the addition of three costs:

• **Cost related to transfer itself**. All transfers have associated the same fixed cost, which is greater than the cost of the line. This cost may be increased for each transfer by the following two costs:

• **Cost related to the transfer complexity**. This cost refers to the intrinsic complexity of transfers. This complexity is independent of the user's profile. For example, the cost of transfer in *Cuatro Caminos* will be much greater than the cost of transfer in *Gran Vía*, because in the first transfer there are several levels (with stairs and elevators) and more lines converge, making the transfer more complicated.

• **Cost associated with the stretch**. This cost includes the inherent cost from walking from one station to another when transferring lines. For example, the transfer cost from L02-*Noviciado* to L10-*Plaza de España* will be greater than the one from L03-*Callao* to L05-*Callao*, because in the first case it is necessary to walk a very long stretch, which takes around five minutes.

These fixed costs can later be modified for each user's profile through the filtering mechanisms used during route calculation, described in section 3.2.1.

The cost of a line will always be smaller than the total cost

of a transfer, that is, transfers are always "penalized", so that they are avoided as much as possible when calculating routes, since they always add extra complexity, specially for individuals with limitations.

3.2. Personalized route server

The server is divided into two blocks, the Route Estimator and the Route Monitor, which are explained in a more detailed way below.

3.2.1. Route Estimator

This module is responsible for creating the path to guide the user. When calculating this path, Dijkstra's algorithm has been applied, but instead of calculating the shorter routes from an initial node to all the others, only the paths that reach the target node are calculated.

The optimum-route calculation requires providing a starting point and a destination point. Both the starting point and the final destination can either be coordinates in space, or a certain street or a specific station. If the starting node or the destination is an address or a coordinate, the user has the option to choose the number of nearby stations to take into account, or the stations that are within a radius defined by him/her. If neither the number of nearby stations nor the radius is specified, the nearest station is considered by default. In the case that there is no station within the established radius, the nearest station will be considered by default, and the existing distance to it, will be notified to the user. In the case that there are multiple starting/ destination nodes, the algorithm mentioned above will be applied from all the possible starting nodes to all the possible destination nodes. The route selected will be the combination that has the lowest cost.

The user can set a series of restrictions that allow him/her to define the parts of the route to be followed. Routes are calculated following these restrictions. For example, the user can set stations to be included in the route: 'I want to go from P1 to P2 passing through *Nuevos Ministerios*, as I have an errand to run nearby'.

The following modification made to the algorithm of Dijkstra is more versatile: it consists of applying a variety of filters when updating the nearby nodes, to see if stations meet with the requirements imposed by the user.

There are two types of filters: elimination filters and modification ones. Elimination filters remove the stations that do not pass through the filter, so as not to be taken into account when calculating the route, while modification filters only penalize lines and transfers by increasing their cost, so that the user does not need to pass though them (if possible). The use of elimination filters can be understood as 'such an event should not take place', and the modification filters as 'if possible, such an event should not take place'.

Types of elimination filters (EF):

• Season ticket EF: removes the stations located outside the user's season ticket rate zone. This filter rules out the possibility of getting out of the rate zone covered by the user's season ticket, and saves the user from paying the supplement due to a zone change. For example: 'I want to go from P1 to P2 with my B2 season ticket'.

• Stations EF: removes the stations that have been selected by the user so that the route will not pass through any of them. For example: 'I want to go from P1 to P2 without going through *Sol* subway station, because I know it is too crowded with people'.

• Subway line EF: removes the subway lines indicated by the user, so that the route will not pass through any station belonging to these lines. For example: 'I want to go from P1 to P2 without going through the L05 subway line because it is usually very slow and trains are too old'.

• Tram line EF: removes the tram lines indicated by the user so as not to be taken into consideration when calculating the route. For example: 'I want to go from P1 to P2 without using tram line ML1 because it takes too long to arrive'.

• RENFE line EF: removes the suburban lines indicated by the user so that the route will not pass through any station belonging to these lines. For example: 'I want to go from P1 to P2 without passing through the C08 suburban line because I know there are intermittent closures due to building works'.

• Subway EF: removes all the stations belonging to the subway network. Only the suburban rail and the tram will be used. For example: 'I want to go from P1 to P2 without using the subway because I think it is too complicated'.

• Tram EF: removes all the stations belonging to the tram network. Only the subway lines and the RENFE suburban railways will be used. For example: 'I want to go from P1 to P2 without using the tram because today it is raining and I know that most of the rail stops are in the open air'.

• RENFE EF: deletes all the stations belonging to the suburban network. Only the subway and the tram will be used. For example: 'I want to go from P1 to P2 without using suburban lines because I never know which platform I should wait on'.

• Mobility EF: eliminates the stations that are not totally or partially adapted to people with reduced mobility. This type of filter only acts when starting, reaching destination and making transfers. For example: 'I want to go from P1 to P2 without going through stations that are not totally

adapted because I am in a wheelchair'.

• Transfers EF: removes all the joints between two stations involving a train transfer, i.e. involving going out from a certain subway/ tram/ suburban rail to catch another different means of transport. For example: 'I want to go from P1 to P2 without transferring in order not to get lost between two stations'.

Types of modification filters (MF): these are the same as the elimination filters, with the only difference that instead of deleting stations, they try to avoid them:

• Season ticket MF: 'If possible, I want to go from P1 to P2 using only my B2 season ticket'.

• Stations MF: 'I want to go from P1 to P2 avoiding, as far as possible, passing through Sol subway station'.

• Subway line MF: 'I want to go from P1 to P2 trying not to go on the L05 subway line'.

• Tram line MF: 'I would like to go from P1 to P2 without using tram line ML1 unless it is impossible'.

• RENFE line MF: 'If possible, I would like to go from P1 to P2 without passing through the C08 suburban line'.

• Subway MF: 'I want to go from P1 to P2 trying not to use the subway network'.

• Tram MF: 'I want to go from P1 to P2 avoiding the tram service, as far as possible'.

• RENFE MF: 'I would like to go from P1 to P2 by not using RENFE suburban railways if possible'.

• Mobility MF: 'If possible, I want to go from P1 to P2 taking into account that I need stations which are partially adapted to people with reduced mobility'.

• Transfers MF: 'I want to go from P1 to P2 minimizing the number of transfers'.

If due to the elimination filters, it is impossible to calculate the route, the user will be alerted about the problem. It will be the case, for example, that one cannot go from P1 to P2 without passing through L05 subway line, since P2 is a station of L05 subway line, and P2 is not a transfer station (if it were one, it would be possible to get there through another line).

Regarding modification filters, there are a number of exceptions in which the line or the transfer will not be penalized. For example, it is pointless to increase the cost of a subway line to prevent passing through it, if the destination node is located on that line and it is not a transfer station, as it is inevitable to pass through that line. Neither a line nor a transfer that have already been amended in some other step of the algorithm are modified.

In addition, before filtering the stations, lines, or tranfers, their availability is checked, in order to discard the possibility of the subway/ tram/ suburban railway lines being closed. In short, the use of filters when updating the nearby nodes at the Dijkstra modified algorithm can be seen in Figure 3.

```
u: current node
l: link(u,v)
for each neighbour v of u:
  if isAvailable(v) && isAvailable(l)
    for each filter f:
      if f is EF && hasToBeRemoved(f,v)
        v discarded
        break
      if f is MF && hasToBeModified(f,v,l)
        modifyCost(f,v,l)
    if notDiscarded(v)
      updateDijkstra'sNeighbours(u,v,l)
  else
    v discarded
```

Figure 3. Filter use pseudocode when updating the nearby nodes in Dijkstra algorithm.

Once the route has been calculated, a *Route* object is created, in which the path information is recorded, be it stations or coordinates. Then, the *Route* object is encoded using the *JSON* standard and sent to the client. When the client receives it, it reverts the process decoding the *Query* into a *Route* object, and with such information, it will display relevant data on the screen.

3.2.2. Route monitor

This module is responsible for monitoring the path the user is following in order to verify whether he/she is on the right track. To do this, it periodically receives data from the user's location (using the GPS Terminal) and estimates if the user's current location deviates from the path location.

3.3. Thin client

As mentioned above, transfers and lines have an associated cost. These costs vary dynamically according to the transport data flow, so it is more useful to have the information in a server rather than in the mobile terminal. This way, when one of these costs varies, it will be enough to update the server graph instead of updating all working terminals.

The mobile terminal used in WAI-Routes tests (HTC Dream), uses the operating system Android, although any other could have been used. This terminal has been chosen for several reasons: Android is based on the Linux kernel, which implies a free software license and open source; it

allows developers to control the device resources through libraries developed or adapted by Google (such as the use of Google maps); the terminal can remain connected to the Internet and has integrated GPS.

The user model is a key element stored in the terminal. This makes it possible to customize the queries raised to the server, and to decide different restrictions when calculating the route according to the user's preferences. The user model can be included in two ways. On the one hand, users or, even better, their carers can describe the profile to be used by default, so that whenever they use the application, it automatically creates a series of filters according to this profile. On the other hand, the user model can be modified in each query, so that new filters can be added and existing ones can be either overwriten or removed. Following is the explanation of how the client works, considering that the user modifies his/her profile dynamically.

As starting and target points the user must initially enter a coordinate, an address, a subway station, a tram station or a suburban station. If an address has been entered, a geocoding is performed, in order to obtain the coordinates of that address (only coordinate or station names are sent to the server). If no starting point has been entered, the exact location of the user given by the GPS terminal is used.

As mentioned in paragraph 3.2.1, if the startup or destination are coordinates, the user has the ability to choose the number of nearby stations that he/she wants to take into account when calculating the route, or the radio (in meters) he/she wants to cover for a possible start-up or destination station. If no option is selected, the nearest station will be used by default.

Then the user can choose if he/she wants to calculate a normal path from the startup to the destination, or if he/she wants to sort out a route through a series of stations. In the second case, the user must specify the names of the stations he/she wants to pass through (the line or type of transport to which it belongs is not specified).

Finally, the user must choose the features he/she wants the route to meet, i.e. he/she should select the filters he/she needs. All the information compiled to calculate the path is included in a *Query* object. Next, it is encoded in a *JSON* object, which is sent to the server. The server decodes the *JSON* object into a *Query* object, and the data are used to calculate the route. The *JSON* format has been chosen for the information exchange between the client and the server because it is a light information exchange format, it is easy to read and written by programmers, and it is easy to interpret and generate by machines.

To better understand how to use the *JSON* encoding, see the example in Figure 4, which defines the starting node (*Alvarado* station), the destination (a point defined by a coordinate), the number of nearby stations to take into account from the destination node coordinate (1), an elimination filter (it removes *Cuzco* and *Sol* stations) and two modification filters (one to avoid L06 subway line, and another to avoid those stations that are not completely adapted to people with reduced mobility).

```
{
 "from":{
       "value": "ALVARADO",
       "type": "station"},

 "to":{
       "value":{
           "longitude":-3.696406,
           "latitude":40.438261},
       "type":"coordinates"},

 "numberOfNearbyStations":1,

 "filters":{

       "EliminationFilters":{
           "stationsEF":["CUZCO","SOL"]},

       "ModificationFilters":{
           "subwayLineMF": ["L06"],
           "mobilityMF": "TOTAL"}
       }
}
```

Figure 4. JSON sketch, where the starting point is a station, the destination is a coordinate and the nearest station to that coordinate is selected. In addition, another three filters are defined: an elimination and two other modifications.

4. Examples of filtered and no-filtered routes

Let us suppose that the user wants to go from c/ *Arequipa 13* to *Islas Filipinas* station. WAI-Routes takes the nearest station and calculates the route shown in Figure 5.

To better understand the route selection, see the piece of subway network of the *Comunidad de Madrid* regional area used in Figure 6.

```
# The nearest init station is at 69.55 meters #

L08-MAR_DE_CRISTAL
L08-PINAR_DEL_REY
L08-COLOMBIA
L08-NUEVOS_MINISTERIOS
L08-NUEVOS_MINISTERIOS
L06-CUATRO_CAMINOS
L06-GUZMAN_EL_BUENO
L07-GUZMAN_EL_BUENO
L07-ISLAS_FILIPINAS
```

Figure 5. Normal route, without using filters, from a street to a subway station.

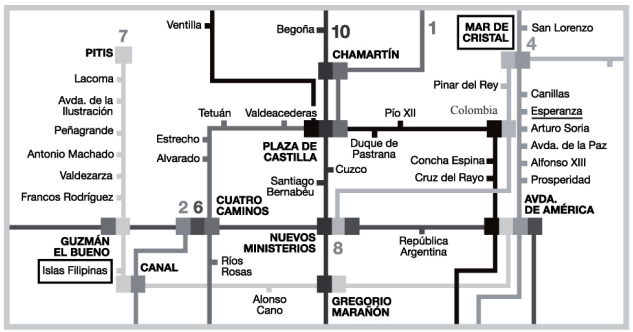


Figure 6. Piece of the subway network. Starting and destination stations appear inside a box and passing station underlined.

Now let us imagine that the user has to return a book to the nearest library, which is in *Esperanza*, he/she does not want to pass through *Avenida de América* because it is a very complicated transfer (four subway lines together) and he/she does not want to use the subway L06 line either because, being a circular line, it is very difficult to know on which platform he/she should be placed.

This time, instead of leaving from *Mar de Cristal* station in L08 line, the starting is from L04 line, since *Esperanza* station is in this line. Without EF (*Avda. de América*), the route would have reached this transfer and then it would have used line L07 to *Islas Filipinas*. With EF (*Avda. de América*) but without the Subway line EF (L06), the path would have been as shown in Figure 7 to *Nuevos Ministerios*; there, line L06 would have been used to *Guzman el Bueno* and finally, line L07 would have been used to *Islas Filipinas*. The final result with all the restrictions imposed by the user can be seen in Figure 7.

```
# The nearest init station is at 69.55 meters #

L04-MAR_DE_CRISTAL
L04-CANILLAS
L04-ESPERANZA
L04-CANILLAS
L04-MAR_DE_CRISTAL
L08-MAR_DE_CRISTAL
L08-PINAR_DEL_REY
L08-COLOMBIA
L08-NUEVOS_MINISTERIOS
L10-NUEVOS_MINISTERIOS
L10-GREGORIO_MARAÑON
L07-GREGORIO_MARAÑON
L07-ALONSO_CANO
L07-CANAL
L07-ISLAS_FILIPINAS
```

Figure 7. Route with: Pass Through (Esperanza), Station EF (Avda. de América) and Subway Line EF (L06).

5. Conclusions and future work

WAI-Routes calculates a route between two points of the *Comunidad de Madrid* regional area to help users when travelling by public transport. This application has been specifically conceived to help users with cognitive limitations to go from one place to another. To this purpose, a user's profile, as well as a series of filters, are considered. These filters can be set once, the first time the user interacts with the application, or modified whenever the user wants to calculate a different route.

It is important to notice that, although WAI-Routes has been tested with the public transport network graph of the *Comunidad de Madrid* regional area, it can be used for any other city transport network, after simply modifying the files describing the graph.

WAI-Routes only focuses on path calculation. The next step will be to develop WAI-interface, which will support the interaction between the system and the users. We plan to build adaptive and very intuitive interfaces to guide each user according to his/her special/specific needs.

One of the improvements in the system that we are planning to include, is to generate several route candidates, instead of just one. Each candidate will be registered with a ranking that reflects the route meeting the initial user's preferences. This implementation is very straightforward using the filter mechanism. Once a route is calculated, a new elimination filter is created. This filter eliminates the stretch from the last transfer or intersection made, to the destination place. Then a new route is re-calculated. These steps can be repeated as many times as required in order to obtain different candidates. The ranking associated to each candidate will appear depending on the cost of the route.

Additionally, it would be interesting to check what tasks should be done by the user at each place and time (i.e., buying the ticket). He/she could be warned about this, with different degrees of emphasis, depending on the priority of the task and the user's limitations. Furthermore, the system could make suggestions on the subsequent tasks to be performed, it could create summaries of the routes followed by the user and it could report the completed and incompleted tasks made within a day, and so on. Finally, the carers or family members could access the routes followed by users under their responsibility (although privacy issues should also be considered).

Acknowledgements

This work was funded by the Spanish Ministry of Science and Innovation through the HADA project (TIN2007-64718). The authors would also like to thank Miss Natalia Ruiz Corres and Mr David Burgos Amador for their support during the elaboration of the paper, and to Dr. Manuel García-Herranz for his inspiring thoughts that have contributed to improve it.

References

[1] HADA, Hipermedia Adaptativa para la atención a la Diversidad en entornos de inteligencia Ambiental (TIN2007-64718). http://hada.ii.uam.es.

[2] Federación Española de Síndrome de Down. http://www.sindromedown.net/.

[3] Consolvo, S., Roessler, P., Shelton, B., Lamarca, A., Schilit, B. and Bly, S. (2004). "Technology for care networks of elders". Pervasive Computing, IEEE, Vol. 3, No. 2, pp. 22-29.

[4] Sistema de Información de Transportes de Madrid. http://www.ctm-madrid.es/.

[5] Barkhuus, L. and Dey, A. (2003) "Location-Based Services for Mobile Telephony: a study of user's privacy concerns". In Proceedings of INTERACT'03, pp. 709-712.

[6] Repenning, A. and Ioannidou, A. (2006). "Mobility agents: guiding and tracking public transportation users". In Proceedings of the Working Conference on Advanced Visual interfaces (Venezia, Italy, May 23 - 26, 2006). AVI '06. ACM, New York, NY, pp. 127-134.

[7] Chang, Y., Tsai, S., Chang, Y., and Wang, T. (2007). "A novel wayfinding system based on geo-coded qr codes for individuals with cognitive impairments". In Proceedings of the 9th international ACM SIGACCESS Conference on Computers and Accessibility (Tempe, Arizona, USA, October 15 - 17, 2007). Assets '07. ACM, New York, NY, 231-232.

[8] Tsai, S. (2007). "WADER: a novel wayfinding system with deviation recovery for individuals with cognitive impairments". In Proceedings of the 9th international ACM SIGACCESS Conference on Computers and Accessibility (Tempe, Arizona, USA, October 15 - 17, 2007). Assets '07. ACM, New York, NY, 267-268.

[9] Patterson, D. J., Liao, L., Gajos, K., Collier, M., Livic, N., Olson, K., Wang, S., Fox, D. and Kautz, H. (2004). "Opportunity Knocks: A System to Provide Cognitive Assistance with Transportation Services." UbiComp 2004: Ubiquitous Computing, pp. 433-450.

[10] Transports Metropolitans de Barcelona (TMB). http://www.tmb.cat/vullanar/es_ES/vullanar.jsp.