



ugr | Universidad
de Granada

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA

Software para el diseño de rutas turísticas con puntos de interés

Autor
Alberto Armijo Ruiz

Directores
David A. Pelta Mochcovsky
Marina Torres Anaya



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
7 de junio de 2018

Software para el diseño de rutas con puntos de interés

Alberto Armijo Ruiz

Palabras clave: Turismo, Rutas, Puntos de Interés, Aplicaciones móviles.

Resumen

El objetivo de este trabajo ha sido diseñar y desarrollar de un prototipo de app para el diseño de rutas turísticas con puntos de interés teniendo en cuenta las preferencias del usuario.

A partir de información de diferentes tipos sobre puntos de interés (obtenida mediante la interacción con varios servidores), las preferencias del usuario y un algoritmo de cálculo de rutas; la app obtiene y dibuja en un mapa la ruta recomendada para el usuario.

Para el desarrollo de la app fueron esenciales los conocimientos y la experiencia en programación adquirida a lo largo del grado, en especial la aportada por las asignaturas “Metaheurística” y “Programación de dispositivos móviles”; las cuales el autor cursó durante el tercer y cuarto curso del Grado en Ingeniería Informática de la Universidad de Granada. Además, fue necesaria la consulta de fuentes bibliográficas para el estudio del problema de diseño de rutas, así como el estudio y uso de herramientas de desarrollo y gestión de proyectos.

Software for the desing of routes with points of interest

Alberto Armijo Ruiz

Keywords:Tourism, Routes, Points of Interest, Mobile Applications.

Abstract

The objective of this work was to design and develop an app's prototipe for the design of turistic routes with points of interest given the user's preferences.

With the information about different types of points of interest (obtained through the interaction with different servers), the user's preferences and an route calculation algorithm; the app gets and draws in a map the recommended route for the user.

For the development of this app were essential the knowledge and experience in programming acquired along the degree, specially those from the subjects "Metaheurísticas" and "Programación de dispositivos móviles"; which the autor attended during the third and fourth year of the "Grado en Ingeniería Informática" of the University of Granada. In addition, it was necessary to consult bibliographic sources for the study of route design problem, as well as the study and use of development and project management tools.

Yo, **Alberto Armijo Ruiz**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 26256219V, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Alberto Armijo Ruiz

Granada a 7 de Junio de 2018 .

D. David A. Pelta Mochcovsky, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento DECSAI de la Universidad de Granada.

Dña. Marina Torres Anaya, Investigadora en el Departamento DECSAI de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado *Software para el diseño de rutas con puntos de interés*, ha sido realizado bajo su supervisión por **Alberto Armijo Ruiz**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 7 de Junio de 2018.

Los directores:

David A. Pelta Mochcovsky
Marina Torres Anaya

Índice general

1. Introducción	15
1.1. Motivación	15
1.2. El Problema de Diseño de Rutas Turísticas	16
1.3. Objetivos	17
2. Planificación y Especificación de Requisitos y Casos de Uso	19
2.1. Planificación	19
2.1.1. Diagrama de Gantt	19
2.1.2. Herramientas utilizadas para la planificación	20
2.2. Especificación de Requisitos	23
2.2.1. Requisitos	23
2.2.2. Casos de uso	25
2.2.3. Fuente de los datos	26
3. Análisis y Diseño	31
3.1. Análisis	31
3.1.1. Diseño de rutas turísticas	31
3.1.2. Heurísticas propuestas	32
3.2. Diseño	36
3.2.1. Diagrama de clases	36
3.2.2. Interfaz de Usuario	45
4. Implementación y Pruebas	49
4.1. Implementación	49
4.1.1. Interfaz de usuario	49
4.1.2. Procesos internos	61
4.1.3. Glosario de términos de Android	70
4.2. Pruebas	71
4.2.1. Caso 1	71
4.2.2. Caso 2	73

5. Conclusiones	75
5.1. Conclusiones	75
5.2. Trabajos futuros	76
Bibliografía	82

Capítulo 1

Introducción

1.1. Motivación

Como consecuencia de la globalización, las naciones están cada vez más conectadas. Esto se debe en gran parte a los ordenadores, a Internet; y desde hace unos años a los dispositivos móviles.

Una consecuencia de esto es el aumento en el turismo mundial, en el cual cada año crece más el número de turistas y los ingresos generados por el turismo. Según los últimos datos ofrecidos por el UNWTO [1], el turismo representa el 10 % del PIB mundial. Según dicha fuente también, el turismo representa el 7 % de las exportaciones mundiales y aporta uno de cada diez puestos de trabajos en todo el mundo. Además, en el año 2016 hubo más de 1200 millones de turistas y se prevé que para 2030 haya 1800 millones de turistas. Este último año ha habido en España unos 82 millones de turistas internacionales, lo que ha generado 87.000 millones de euros, suponiendo un 12.4 % más que el año anterior.

Hoy en día casi todo el mundo cuenta con un móvil, el cual utiliza para todo tipo de cosas: redes sociales (Twitter, Facebook, etc...), consultar su cuenta bancaria, escuchar música, ver películas, editar documentos o incluso pagar con él.

Por todo esto, debería aprovecharse el potencial del turismo y adaptarlo a la tecnología actual; mejorando la calidad de las visitas, adaptándose a sus preferencias y necesidades. Con este propósito existen diferentes tipos de apps dirigidas a los turistas; como por ejemplo aplicaciones para reserva de hoteles, mapas, compra de entradas o generación de itinerarios.

La propuesta de aplicación móvil a desarrollar en este proyecto pretende ofrecer itinerarios personalizados que incluyan rutas asociadas a los intereses del usuario y que maximicen la satisfacción de este. El producto será una aplicación móvil desarrollada en Android que ejecutará un algoritmo en Java basado en el problema Tourist Trip Design Problem. Dicho algoritmo usa una implementación de una heurística voraz (Greedy) y

retornará la mejor solución encontrada en una clase contendora, la cual permite dibujar los diferentes puntos de interés de la solución y la ruta asociada a dichos puntos. Para este proyecto se utilizará como ejemplo la ciudad de Granada.

1.2. El Problema de Diseño de Rutas Turísticas

El *Problema de Diseño de Rutas Turísticas* (Tourist Trip Desing Problem, TTDP) intenta resolver el problema de obtener una ruta que contenga el número mayor posible de puntos de interés sin violar ninguna restricción que el turista tenga; algunas restricciones pueden ser económicas o de tiempo. Este problema fue mencionado por primera vez por Vansteenwegen y Oudheusden en 2007 [2].

Este problema es una extensión del problema llamado *Orienteering Problem* (OP). En el OP, existen localizaciones con un beneficio asociado las cuales deben ser visitadas en una franja de tiempo; el objetivo de este problema es maximizar el beneficio total visitando solamente una vez cada localización. Dicho problema fue introducido en 1984 por Tsiligirides [3].

Con el tiempo han aparecido extensiones del OP, por ejemplo el *Team Orienteering Problem* (TOP), desarrollado en 1996 (Chao et al.) [4]. En dicho problema existen varios equipos los cuales tienen que elegir rutas que no pueden contener puntos de interés seleccionados por otros equipos; el objetivo es maximizar la puntuación total sin exceder el tiempo máximo. Un equipo se puede interpretar como un día dentro de un viaje de múltiples días. Una extensión de este mismo problema es el *Team Orienteering Problem with Time Windows* (TOPTW), en el cual un punto de interés puede visitarse dentro de su ventana de tiempo específica (por ejemplo, el horario de un museo) (Vanteenwegen et al. 2009) [5].

Otra extensión del OP es el *Time Dependent Orienteering Problem* (TDOP), en el cual se considera el tiempo necesario para llegar desde un punto de interés a otro punto de interés (Formin and Lingas 2002) [6]. La combinación de los dos problemas anteriores da resultado al *Time Dependent Orienteering Team Problem with Time Windows* (TD-TOPTW) (Garcia et al. 2010) [7]. Otro problema es el *Multi Constrained TOPTW*, que considera múltiples restricciones como pueden ser un presupuesto además del tiempo (Sylejmani et al. 2012) [8].

Algunas extensiones más recientes son el DPTOP, *TOP with Decreasing Profits*, en el cual se utiliza una función para calcular el valor de un punto de interés según el tiempo (Murat Afsar y Labadie 2013) [9]; y el *Clustered OP* (Angelelli et al. 2014) [10], donde los puntos de interés se asignan en grupos. Cada grupo tiene asignado un valor y este valor forma parte de beneficio total si todos los puntos de interés contenidos en dicho punto son visitados en la ruta.

1.3. Objetivos

Los objetivos concretos que persigue este proyecto son los siguientes:

1. Estudio del estado del arte de los sistemas de planificación de rutas con recomendaciones, y de los sistemas de información sobre puntos de interés en las ciudades.
2. Definición y diseño de modelos y herramientas de recomendación de itinerarios ajustados a las necesidades y preferencias de los turistas.
3. Validación de modelos y métodos, empleando pruebas y datos de puntos de interés en ciudades de interés turístico.
4. Obtención de un prototipo software integrado en una aplicación para dispositivos móviles.
5. Creación de documentación técnica, entregables y memoria final del proyecto.

Capítulo 2

Planificación y Especificación de Requisitos y Casos de Uso

En este capítulo se mostrará la planificación del Trabajo de Fin de Grado. También se describen los requisitos (de la interfaz y de la aplicación), se presentan los diagramas de caso de uso y se describen las fuentes de datos a partir de las cuales la aplicación obtiene la información necesaria.

2.1. Planificación

2.1.1. Diagrama de Gantt

En esta sección se muestra una diagrama de Gantt con la planificación del proyecto dividido en diferentes tareas. Las tareas desarrolladas junto a su planificación temporal se muestran en el diagrama de Gantt de la figura 2.3. Las tareas en las que se ha dividido el proyecto son las siguientes:

- **Investigación sobre el problema:** en esta tarea se realizó un estudio del problema que se resuelve en este Trabajo de Fin de Grado.
- **Investigación sobre herramientas para el desarrollo:** en esta tarea se realizó un estudio sobre las herramientas disponibles para resolver el problema.
- **Análisis de requisitos:** en esta tarea se realizó un análisis de los requisitos de la aplicación a desarrollar.
- **Diseño:** en esta tarea se explicó los diferentes aspectos del diseño de la aplicación.
- **Implementación:** en esta tarea se explicó los aspectos más importantes desarrollado en la aplicación.
- **Evaluación y pruebas:** en esta tarea se realizaron pruebas sobre diferentes casos de uso de la aplicación por parte de un usuario.

2.1.2. Herramientas utilizadas para la planificación

Trello

Trello [11] (<https://trello.com/>) es una herramienta muy utilizada en el ámbito empresarial. Trello consiste en tableros en los cuales se incluyen tareas. Dichas tareas pueden moverse de un tablero a otro y pueden asignarse si es necesario a personas concretas.

Para este Trabajo de Fin de Grado, se ha creado un tablero al que se le ha llamado “TFG” y se han utilizado tres tablas: Lista de tareas, En proceso y Finalizado. En la tabla “Lista de tareas” se encuentran todas las tareas programadas que aún no se han empezado; en la tabla “En proceso” se encuentran todas las tareas que se están desarrollando y en el tabla “Finalizado” se encuentran todas las tareas que se han finalizado.

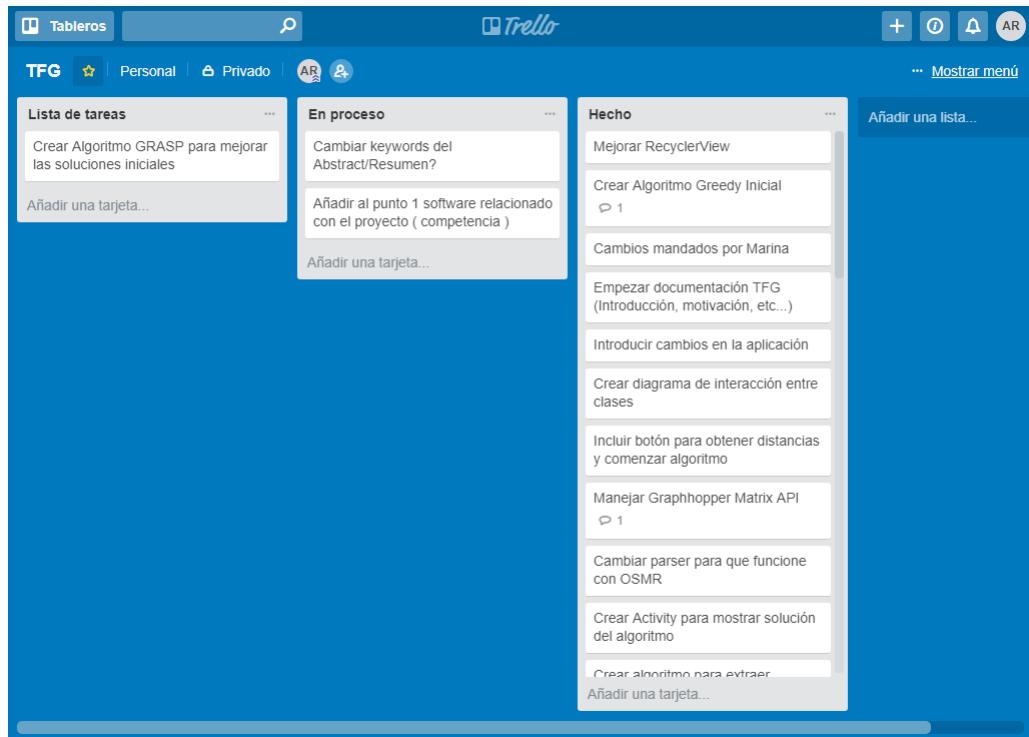
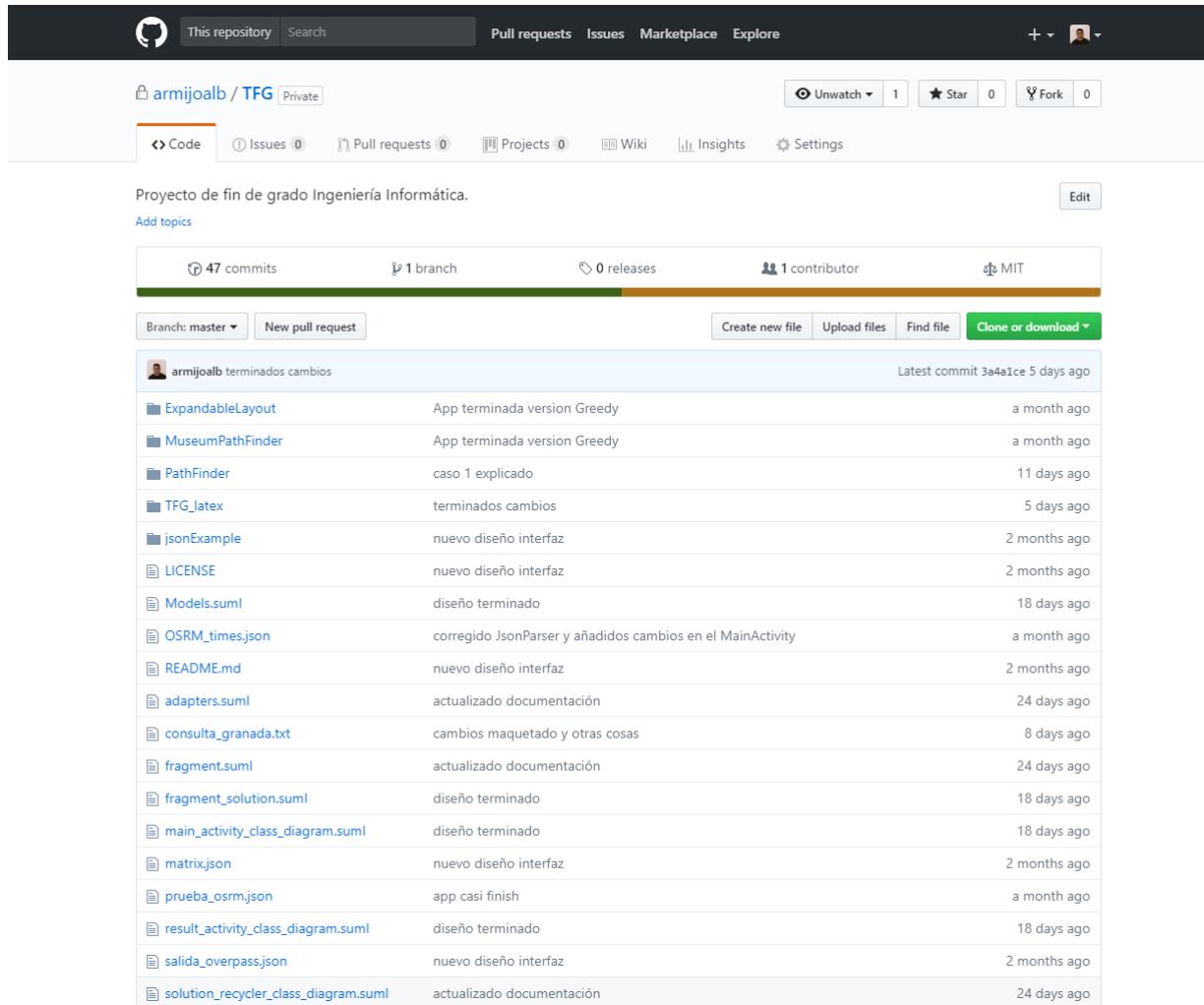


Figura 2.1: Tablero utilizado en Trello para el Trabajo de Fin de Grado.

GitHub

GitHub [12] (<https://github.com/>) es un servicio que permite almacenar proyectos utilizando el sistema de control de versiones Git. GitHub es gratuito para todas las personas que quieran crear repositorios visibles para todo el mundo; si se desea crear repositorio privados hay que contratar un plan de pago.

Para este Trabajo de Fin de Grado, se ha utilizado GitHub para crear un repositorio que contenga el proyecto e ir subiendo nuevas versiones de este cuando se han completados tareas dentro del proyecto.



The screenshot shows a GitHub repository page for the user 'armijoalb' named 'TFG'. The repository is private. At the top, there are links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right, there are buttons for 'Unwatch', 'Star' (0), and 'Fork' (0). Below the header, there are tabs for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. A description below the tabs reads 'Proyecto de fin de grado Ingeniería Informática.' with an 'Edit' button. The main area shows a summary bar with '47 commits', '1 branch', '0 releases', '1 contributor', and 'MIT' license. Below this is a table of commits:

Author	Commit Message	Date
armijoalb	terminados cambios	Latest commit 3a4a1ce 5 days ago
ExpandableLayout	App terminada version Greedy	a month ago
MuseumPathFinder	App terminada version Greedy	a month ago
PathFinder	caso 1 explicado	11 days ago
TFG_latex	terminados cambios	5 days ago
jsonExample	nuevo diseño interfaz	2 months ago
LICENSE	nuevo diseño interfaz	2 months ago
Models.suml	diseño terminado	18 days ago
OSRM_times.json	corregido JsonParser y añadidos cambios en el MainActivity	a month ago
README.md	nuevo diseño interfaz	2 months ago
adapters.suml	actualizado documentación	24 days ago
consulta_granada.txt	cambios maquetado y otras cosas	8 days ago
fragment.suml	actualizado documentación	24 days ago
fragment_solution.suml	diseño terminado	18 days ago
main_activity_class_diagram.suml	diseño terminado	18 days ago
matrix.json	nuevo diseño interfaz	2 months ago
prueba_osrm.json	app casi finish	a month ago
result_activity_class_diagram.suml	diseño terminado	18 days ago
salida_overpass.json	nuevo diseño interfaz	2 months ago
solution_recycler_class_diagram.suml	actualizado documentación	24 days ago

Figura 2.2: Repositorio de GitHub para el Trabajo de Fin de Grado.

TFG	Fecha de inicio	Fecha final	Timeline	Estado
Investigación sobre el problema	1/11/2017	11/06/2018		Completado
Investigación sobre herramientas para el desarrollo	8/11/2017	15/01/2018		Completado
Ánalisis de requisitos	30/11/2017	7/12/2017		Completado
Diseño	1/12/2017	3/05/2018		Completado
Implementación	7/12/2017	3/05/2018		Completado
Evaluación y pruebas	3/05/2018	1/06/2018		Completado

Figura 2.3: Planificación del proyecto.

2.2. Especificación de Requisitos

2.2.1. Requisitos

En este apartado se resumirán los requisitos funcionales y no funcionales del proyecto, dividiéndolos en los requisitos de la interfaz de usuario y los requisitos internos de la aplicación.

a) Requisitos de la interfaz del usuario

Requisitos funcionales:

- **Mapa de la ciudad elegida por el usuario:** la aplicación debe funcionar con cualquier ciudad, mostrando tanto los alojamientos y POIs que haya en la ciudad con marcadores en el mapa.
- **Lista de alojamientos y POIs:** la aplicación dispondrá de una lista de POIs y de alojamientos entre los que el usuario podrá elegir para generar la ruta.
 - Alojamiento: hostales y hoteles.
 - POIs: museos, miradores, lugares históricos o de culto.
- **Especificación de las rutas:** la aplicación deberá mostrar los POIs que contiene la ruta como marcadores, así como las direcciones que deberá seguir el usuario para llegar desde un punto hasta otro. También mostrará una lista ordenada de dichos puntos en los cuales se especificará la hora de entrada y salida aproximadas de cada POI.
- **Información sobre los marcadores:** la aplicación deberá mostrar información referente a los marcadores mostrados en la interfaz. Dicha información será el nombre de dicho POI.

Requisitos no funcionales:

- **Interfaz:** la interfaz deberá ser atractiva, ligera y lo más intuitiva posible.
- **Plataforma:** la aplicación estará disponible para todos los dispositivos móviles que utilicen el sistema operativo Android a partir de la versión 5.0 (Lollipop).

b) Requisitos de la aplicación

Requisitos funcionales

- **Recepción de peticiones del cliente:** la aplicación será capaz de recibir peticiones que el cliente le envía a través de la interfaz gráfica que contienen los POIs seleccionados por el usuario.

- **Cálculo y retorno de ruta óptima:** a partir de los datos proporcionados por el usuario y la información que se tiene sobre dichos datos; la aplicación calculará y devolverá la ruta que después será mostrada en la interfaz.
- **Publicación de lista de alojamientos y POIs:** la aplicación tendrá acceso a una lista de alojamientos y POIs que el usuario podrá seleccionar desde la interfaz de la aplicación.
- **Publicación de la ruta en una mapa:** la aplicación será capaz de mostrar en la interfaz de usuario los POIs seleccionados en un mapa, así como una lista ordenada de la ruta y el camino para llegar desde un POI hasta el siguiente. El primer punto siempre será el alojamiento seleccionado por el usuario.

Requisitos no funcionales

- **Envío de peticiones de POIs y alojamientos:** la aplicación deberá mandar peticiones a un servidor que es capaz de conectarse a una base de datos para obtener los datos necesarios sobre alojamientos y POIs. Dicha información se devuelve en un fichero .json.
- **Importación de información sobre POIs y alojamientos:** tras obtener los POIs y alojamientos del servidor; la aplicación deberá procesar la información contenida en el fichero .json para mostrarlos en la interfaz.
- **Envío de peticiones a servidor de cálculo de matrices de tiempos:** la aplicación deberá mandar peticiones a un servidor que calculará la distancias entre los puntos seleccionados por el usuario. Dichos tiempos se devuelven en un archivo .json.
- **Importación de matriz de tiempos:** una vez se ha obtenido el archivo .json que contiene la matriz de tiempos, la aplicación debe procesar dicho fichero y guardar la matriz para su uso en el cálculo de rutas.
- **Envío de peticiones para obtener ruta entre los distintos POIs que contiene la solución calculada:** la aplicación deberá enviar peticiones a un servidor que devolverá la ruta a seguir el usuario para llegar a cada uno de los POIs. Dicha información se devuelve en un archivo .json.
- **Importación de direcciones entre POIs:** la aplicación deberá guardar y procesar la información que contiene el archivo .json para después mostrar las direcciones en la interfaz de usuario.

2.2.2. Diagramas de casos de uso

Interfaz de usuario

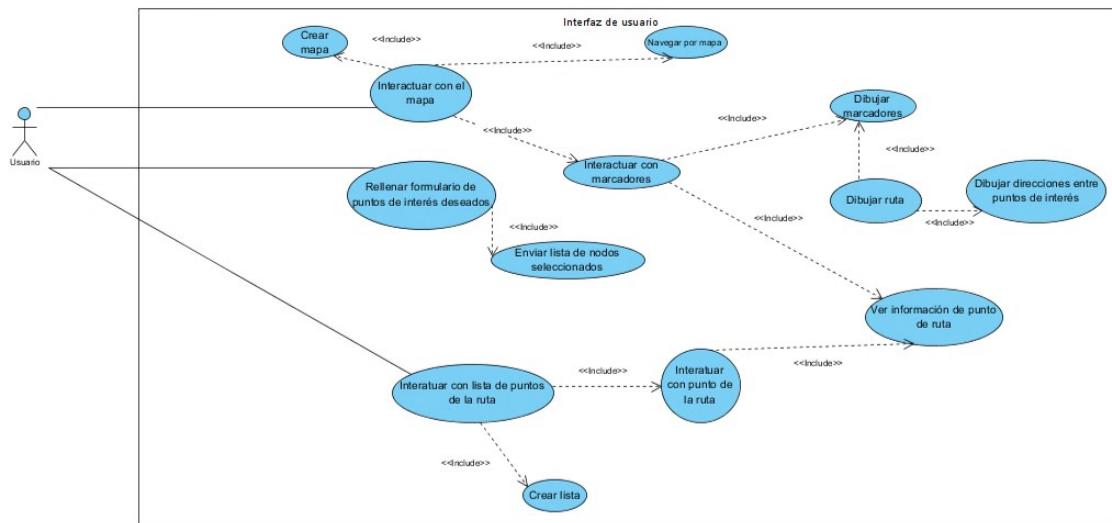


Figura 2.4: Diagrama de casos de uso de la interfaz de usuario.

Aplicación móvil

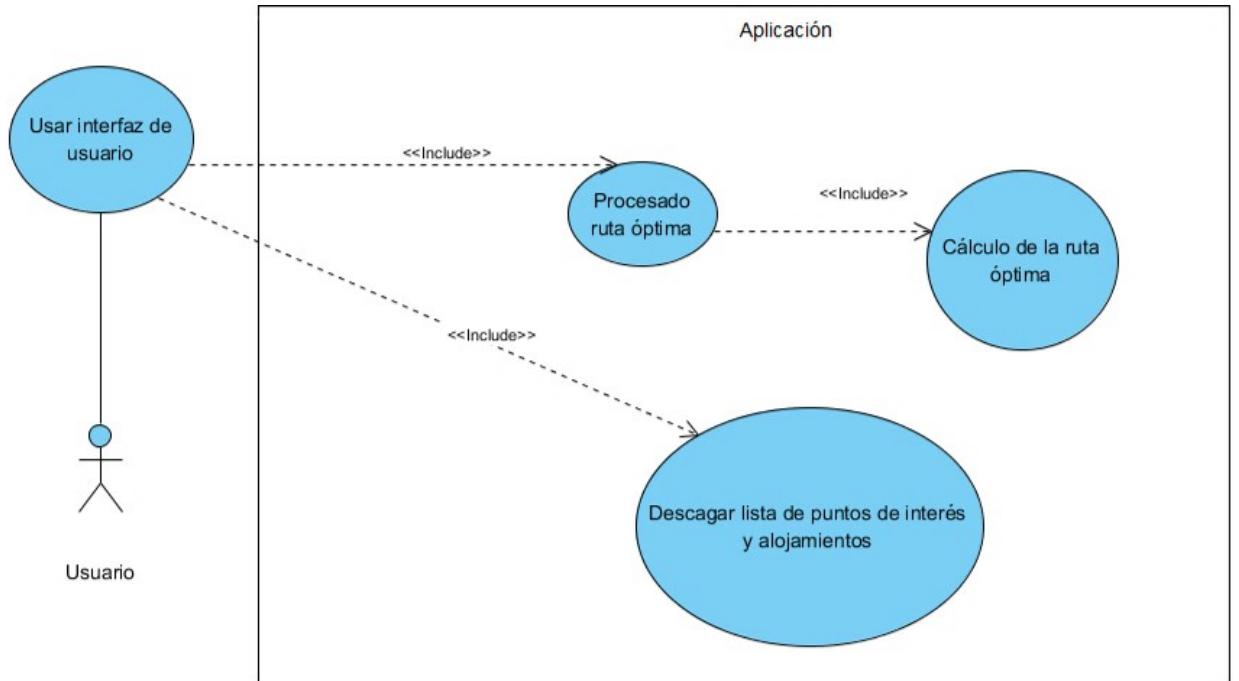


Figura 2.5: Diagrama de casos de uso de la aplicación.

2.2.3. Fuente de los datos

Open Street Map

Ya que el objetivo de la aplicación fue crear rutas a partir de POIs, se requirió obtener una lista con todos los POIs y alojamientos disponibles en una ciudad con la suficiente información para que puedan ser usados para calcular rutas. Tras investigar diferentes posibilidades, se decidió utilizar *OpenStreetMap*, un proyecto colaborativo que permite crear mapas y editar los ya existentes.

OpenStreetMap tiene una web [13] en la cual puedes editar cualquiera de los POIs que contiene el mapa o crear nuevos siempre que estés registrado. Además, cuenta con diferentes tipos de estructuras para representar edificios, carreteras, etc... *OpenStreetMap* utiliza tres tipos diferentes de información:

- Líneas: sirven para describir cualquier tipo de vía, como por ejemplo carreteras.
- Nodos: sirven para definir cualquier tipo de edificio, como por ejemplo farmacias o museos.
- Áreas: sirven para definir el espacio ocupado por un nodo, por ejemplo, un parque. Dichas áreas están formadas un conjunto de nodos que delimitan el área.

Al no estar disponible el acceso a la base de datos que contiene *OpenStreetMap* de forma directa se tuvo que buscar una API para poder obtener los datos.

Overpass API

Para obtener la información necesaria de *OpenStreetMap*, se necesitó buscar una interfaz de programación de aplicaciones (API, Application Programming Interface) que permitiera obtener dicha información. *Overpass* es una API que permite hacer consultas sobre POIs y que utiliza la información de *OpenStreetMap* para devolver la información sobre los POIs. Dichas consultas pueden realizarse mediante urls. Las respuestas obtenidas por *Overpass* se devuelven en formato JSON, dentro de dicho fichero se encuentra un array llamado “elements” que contiene cada uno de los elementos. A continuación, se muestra una parte del fichero que devuelve Overpass para alojamientos y POIs en Granada.

Listado 2.1: Respuesta Overpass sobre alojamientos y POIs en Granada.

```
{
  "version": 0.6,
  "generator": "Overpass API 0.7.54.13 ff15392f",
  "osm3s": {
    "timestamp_osm_base": "2018-02-27T10:23:02Z",
    "copyright": "The data included in this document is from www.openstreetmap.org.  
The data is made available under ODbL."
  },
  "elements": [
    {
      "type": "node",
      "id": 1667155336,
      "lat": 37.1625928,
      "lon": -3.6066354,
      "tags": {
        "name": "Parque de las Ciencias",
        "tourism": "museum"
      }
    },
    ...
    {
      "type": "node",
      "id": 4029579625,
      "lat": 37.1800207,
      "lon": -3.5957336,
      "tags": {
        "name": "Makuto Guesthouse",
        "tourism": "hostel"
      }
    }
  ]
}
```

Para generar dicho fichero de salida, se deben ajustar ciertos parámetros dentro de la petición que se hace a *Overpass*. Para probar diferentes parámetros, se puede utilizar la herramienta de *Overpass* llamada *Overpass-turbo*.

Overpass-turbo es una página web que contiene un mapa interactivo y una ventana donde nos permite escribir consultas a *Overpass*; desde ahí, se puede generar un script que encuentre todos los POIs y alojamientos en una ciudad; después, dicho script puede exportarse como una petición en formato URL para utilizarla sin necesidad de utilizar *Overpass-turbo*. Dentro de dicho script debe especificarse los tipos de nodos que se quieren obtener. La información sobre los diferentes tipos de nodos se encuentra dentro de la documentación de *OpenStreetMap* [14]. El script para obtener la información mostrada arriba es el siguiente.

Listado 2.2: Script para encontrar todos los POIs y alojamientos de una ciudad.

```
[out:json][timeout:100];

(node["place"="city"]["name"="Granada"]["is_in:province"="Granada"]
["is_in:country"="Spain"]);->.ciudad;

node["tourism"="museum"](|around.ciudad:7000);
out;>;out;

node["tourism"="viewpoint"](|around.ciudad:7000);
out;

// Catedrales.
node["building"="cathedral"](|around.ciudad:7000);
out;

// Hoteles.
node["tourism"="hotel"](|around.ciudad:7000);
out;

// Hostales.
node["tourism"="hostel"](|around.ciudad:7000);
out;
```

Open Source Routing Machine

Como la aplicación necesita obtener la matriz de distancias entre los distintos puntos seleccionados por el usuario, se investigó para encontrar una herramienta que permitiera obtener dicha matriz a través de peticiones mediante URL. Tras probar con varias herramientas, se optó por usar *Open Source Routing Machine* (OSRM) [15], la cual es de uso gratuito. OSRM permite hacer peticiones a sus servidores o montar uno propio [16].

OSRM devuelve de sus peticiones un archivo JSON que contiene las distancias entre los puntos especificados en la petición, el tiempo que se tarda en llegar desde un punto

a otro está guardado en un array llamado “durations”; cada elemento de dicho array contiene otro array con los tiempos desde dicho POI al resto de POIs. Un ejemplo de la salida de OSRM es el siguiente:

Listado 2.3: Salida de OSRM.

```
{"durations":  
  [[0,245,165.9,165.9,165.9,44.9,162.3,130.6,130.6,44.9,56.2],  
   [245,0,79.1,79.1,79.1,200.1,82.7,114.4,114.4,200.1,188.8],  
   [165.9,79.1,0,0,0,121,3.6,35.3,35.3,121,109.7],  
   [165.9,79.1,0,0,0,121,3.6,35.3,35.3,121,109.7],  
   [165.9,79.1,0,0,0,121,3.6,35.3,35.3,121,109.7],  
   [44.9,200.1,121,121,121,0,117.4,85.7,85.7,0,11.3],  
   [162.3,82.7,3.6,3.6,3.6,117.4,0,31.7,31.7,117.4,106.1],  
   [130.6,114.4,35.3,35.3,35.3,85.7,31.7,0,0,85.7,74.4],  
   [130.6,114.4,35.3,35.3,35.3,85.7,31.7,0,0,85.7,74.4],  
   [44.9,200.1,121,121,121,0,117.4,85.7,85.7,0,11.3],  
   [56.2,188.8,109.7,109.7,109.7,11.3,106.1,74.4,74.4,11.3,0]],  
  ...  
}
```

Google Maps Directions API

Dado que la aplicación debe mostrar el camino óptimo entre dos puntos seleccionados en la solución, se buscó una herramienta que lo calculara. Tras investigar y probar diferentes herramientas, como por ejemplo la que ofrece OSRM, se optó por utilizar *Google Maps Directions API* ya que obtiene resultados más exactos [17].

Esta herramienta permite hacer peticiones a un servidor y devuelve un archivo JSON que contiene la ruta que se debe seguir. Este archivo se debe procesar para obtener las polilíneas que representan el camino entre dos puntos. Un ejemplo de este archivo es el siguiente.

Listado 2.4: Salida de Google Maps Directions API.

```
{"routes" : [  
  ...  
  "legs" : [  
    {  
      "distance" : {  
        "text" : "2.0 km",  
        "value" : 2044  
      },  
      "duration" : {  
        "text" : "25 min",  
        "value" : 1496  
      },  
    },  
  ]}
```

```
"end_location" : {
  "lat" : 37.1622873,
  "lng" : -3.6068922
},
"start_location" : {
  "lat" : 37.1761203,
  "lng" : -3.6025799
},
"steps" : [
{
  "distance" : {
    "text" : "0.1 km",
    "value" : 115
  },
  "duration" : {
    "text" : "1 min",
    "value" : 75
  },
  "end_location" : {
    "lat" : 37.1757377,
    "lng" : -3.6037831
  },
  "polyline" : {
    "points" : "w}faFbs~TNx@VrAV|@Jb@"
  },
  "start_location" : {
    "lat" : 37.1761203,
    "lng" : -3.6025799
  },
  "travel_mode" : "WALKING"
},
{
  "distance" : {
    "text" : "0.4 km",
    "value" : 354
  },
  "duration" : {
    "text" : "4 min",
    "value" : 252
  },
  "end_location" : {
    "lat" : 37.1738268,
    "lng" : -3.6069271
  },
  "polyline" : {
    "points" : "k{{aFrz~T@Hp@nBP`@T\\j@p@fA|ADF~AtBfAld"
  },
  "start_location" : {
    "lat" : 37.1757377,
    "lng" : -3.6037831
  },
  "travel_mode" : "WALKING"
}
}
```

Capítulo 3

Análisis y Diseño

En este capítulo se presenta el problema de diseño de rutas con POIs y modelización. Para resolver el problema, se proponen inicialmente dos métodos heurísticos que permitirán evaluar el funcionamiento del prototipo. Además se detallarán los aspectos más importantes relacionados con el diseño del prototipo.

3.1. Análisis

3.1.1. Diseño de rutas turísticas

Los *Problemas de Diseño de Rutas Turísticas* (Tourist Trip Design Problems, TTDP) consisten en seleccionar los puntos de interés (points of interest, POI) a visitar por un turista atendiendo a sus restricciones y al beneficio o grado de satisfacción que produce su visita. El turista dispone en su estancia de un día para organizar las visitas a los POIs mediante una ruta de duración limitada. Se parte de un conjunto de puntos disponibles a visitar de los que se conoce, el beneficio o grado de satisfacción, la duración de la visita y el intervalo de tiempo en el que puede realizarse. El beneficio total que se intenta maximizar es la suma de los beneficios obtenidos en cada visita.

En este caso, se trata de un caso particular del problema Team Orienteering Problem with Time Windows (TOPTW) que se ha estudiado en la literatura científica.

Los elementos que forman parte del modelo son:

- Un conjunto de POIs , asociado a un índice $i, i = 1, 2, \dots, n$ y con los siguientes atributos:
 - Una puntuación o beneficio s_i .
 - Un tiempo de duración de la visita r_i .
 - Un intervalo de tiempo $[e_i, l_i]$ dentro del que se puede realizar la visita.
- Un punto de partida de cada una de las rutas denotado por $i = 0$.

- Los tiempos de recorrido entre los pares de puntos $t_{ij}, i, j = 0, 1, \dots, n$.
- Un tiempo máximo T_{max} de duración total de la ruta del día considerando los tiempos de viaje y visita en los POI.
- Una función objetivo $\max \sum_{i=1}^n s_i y_i$

En la que y_i corresponde a una variable de visita para cada uno de los POIs $i, i = 1, \dots, n$. Dicha variable es binaria, teniendo como valor 1 si el punto de interés i es visitado en la ruta y un 0 si no se visita. Esta variable solamente tendrá valor 1 si además está disponible cuando se quiere visitar dicho punto de interés, es decir, si el punto se va a visitar durante el intervalo de tiempo $[e_i, l_i]$ asociado a la variable.

Se consideran POIs museos, miradores, catedrales, mezquitas, etc... de la ciudad que el usuario elija antes de iniciar el algoritmo; además, dichos POIs son considerados para la ruta de un único día.

3.1.2. Heurísticas propuestas

Greedy

Como solución inicial a este problema se ha utilizado una heurística Greedy, Algoritmo 1, ya que es una heurística rápida que puede ejecutar cualquier dispositivo móvil. Dicha heurística Greedy seleccionará siempre el punto más cercano al punto de interés actual; este proceso se repetirá hasta que no puedan introducirse más POIs en la solución, bien porque no haya más POIs por visitar o bien porque no haya más tiempo para visitarlos. El pseudocódigo de dicho algoritmo es el siguiente:

Algoritmo 1 Pseudocódigo algoritmo Greedy.

```

function GREEDY(lista_POI)
    while ( do es posible visitar POI) {
        mejor_poi ← encontrarMásCercano(último_poi_visitado, tiempo_actual)
        tiempo_entrada←tiempo_actual+tiempo_llegada(último_poi_visitado,mejor_poi)
        tiempo_salida ← tiempo_entrada + tiempo_visita_mejor_poi
        solución.añadir(mejor_poi,tiempo_entrada, tiempo_salida)
        tiempo_actual ← tiempo_salida
        eliminar mejor_poi de la lista de POI disponibles
    }
    end while
    return solución
end function

```

El parámetro “lista_POI” es la lista de POIs seleccionados para realizar la búsqueda,

los cuales se van a usar para buscar la ruta en el algoritmo.

La función *encontrarMásCercano(último_poi_visitado, tiempo_actual)* encuentra el punto de interés más cercano al último punto de interés seleccionado y que se pueda visitar en el momento que se sale del último punto de interés, es decir, que el tiempo de entrada a dicho punto de interés esté dentro del intervalo de tiempo permitido para dicho punto de interés. Para ello, recorre un vector que contiene el tiempo necesario para llegar desde el último punto de interés visitado y otro punto; de todos los POIs que estén disponibles, selecciona aquel cuyo tiempo de llegada desde el último punto de interés sea menor. Dicho vector está contenido en una matriz que guarda esta información por cada uno de los POIs. Un punto de interés está disponible si está abierto a la hora en la que el algoritmo lo vaya a seleccionar y si no está incluido en la solución.

Dentro del pseudocódigo se pueden ver tres variables que tienen que ver con el tiempo; la variable “tiempo_actual” guarda el tiempo actual en cada momento del algoritmo, es decir, es una variable que va actualizando su valor cada vez que se selecciona un nuevo punto de interés. La variable “tiempo_entrada” representa el tiempo de entrada aproximado al punto de interés seleccionado; este valor se calcula con el valor de “tiempo_actual” más el tiempo necesario para llegar desde dicho punto hasta el punto seleccionado, representado por la función “tiempo_llegada(último_poi_visitado,mejor_poi)”. Por último, la variable “tiempo_salida” representa el tiempo aproximado de salida de dicho punto de interés, esta variable se calcula como la suma de “tiempo_entrada” y el tiempo aproximado de visita, representado por la variable “tiempo_visita_mejor_poi”.

Una vez termina el algoritmo, devuelve la solución. Dicha solución cuenta con un vector que contiene los identificadores de los POIs seleccionados; además, contiene la hora aproximada de entrada y salida de cada uno de los POIs seleccionados. Además, los POIs contenidos en la solución, se encuentran ordenados por orden en llegada.

GRASP

Como otra posible heurística que resuelva el problema y que no necesite de mucho tiempo de cómputo, se ha propuesto también la metaheurística constructiva GRASP (Greedy Randomized Adaptive Search Procedure), Algoritmo 2. Dicha metaheurística comprende dos fases: una fase constructiva y otra de búsqueda local. En la fase constructiva se genera una solución partiendo de una ruta vacía a la que se va añadiendo POIs desde una Lista Restringida de Candidatos (Restricted Candidate List, RCL en inglés) de forma aleatoria hasta que no se puedan añadir nuevos puntos a la ruta. En la fase de búsqueda local se reemplaza la solución obtenida en la parte constructiva por la mejor de sus soluciones vecinas si existe mejora. Estas dos fases se ejecutan un cierto número de iteraciones.

Algoritmo 2 Pseudocódigo algoritmo GRASP.

```

function GRASP(maxIteraciones,tamRCL)
    leerDatos()
    for  $i = 1$  to maxIteraciones do {
        solución  $\leftarrow$  GRASPFaseConstructiva(tamRCL)
        solución  $\leftarrow$  BúsquedaLocal(solución)
        if solución  $\geq$  mejorSolución then
            mejorSolución  $\leftarrow$  solución
        end if
    }
    end for
    return mejorSolución
end function

```

Para la parte constructiva se muestran en el Algoritmo 3, representado por la función *GRASPFaseConstructiva(tamRCL)*, primero crearemos una lista de candidatos (CL), la cual tiene un tamaño igual al de la lista de POIs que queden disponibles. Dicha lista de candidatos contiene la posición “ i ” dentro de la lista de POIs y la puntuación que tiene dicho punto de interés según las preferencias elegidas por el usuario. La forma de puntuar un punto de interés dependerá también del tiempo necesario para llegar desde el último punto de interés seleccionado y dicho punto de interés. Al igual que para la heurística Greedy, solo se considerarán aquellos POIs que estén disponibles, es decir, que no estén incluidos ya en la solución que se está construyendo y que estén abiertos en el momento que pueden ser seleccionados.

Una vez hemos calculado la lista de candidatos, se crea la lista restringida de candidatos con los POIs mejor valorados, esta es de tamaño “tamRCL”. Una vez hemos obtenido la lista de candidatos restringida, elegimos uno de los POIs de forma aleatoria y lo introducimos dentro de la solución. Este proceso se repite hasta que no se puedan introducir más POIs dentro de la solución.

Algoritmo 3 Pseudocódigo algoritmo GRASPFaseConstructiva.

```
function GRASPFASECONSTRUCTIVA(tamRCL)
    solución.añadir(nodo_salida)
    while es posible visitar POIs do{
        for cada poi en lista.POIs do{
            valor ← f(poi)
            CL.añadir([valor,i])
        }
    end for
    Crear RCL con los mejores tamRCL elementos de CL
    seleccionado ← seleccionarRandom(RCL)
    solución.añadir(lista.POIs[seleccionado.i])
}
end while
return solución
end function
```

En este algoritmo, la función llamada “ $f(\text{POI})$ ” devuelve el valor de dicho POI, este valor esta representado por el tiempo necesario para llegar desde el último punto visitado hasta dicho punto “ poi ”. La variable “ i ” representa la posición del POI dentro de “ lista_POIs ”, utilizada para acceder directamente a la información de dicho POI dentro de “ lista_POIs ”. La variable “ seleccionado ” representa el elemento seleccionado de la lista “ RCL ” (un par que contiene el valor de dicho POI y la posición dentro de “ lista_POIs ”) de forma aleatoria.

Para la fase de optimización se utilizará el algoritmo de búsqueda local, mostrado en el Algoritmo 4, dicho algoritmo busca la mejor solución posible entre la solución actual y los vecinos de esta. Una solución vecina es aquella que intercambie dos POIs dentro de la solución, por ejemplo, el segundo por el cuarto. El procedimiento es el siguiente, se generan todos los posibles vecinos de la solución, y por cada uno de ellos se comprueba si la valoración de dicha solución vecina es mejor que la mejor solución actual; finalmente se devuelve la mejor solución encontrada.

Algoritmo 4 Pseudocódigo algoritmo BúsquedaLocal.

```

function BÚSQUEDALOCAL(solución)
    for all solución_vecina de solución do {
        if solución_vecina > solución then
            solución ← solución_vecina
        end if
    }
    end for
    return solución
end function

```

Para generar un nuevo vecino de la solución dada, se intercambiarán la posición de dos POIs dentro de la solución. Además de intercambiar dichos puntos, hay que actualizar los valores de entrada y de salida de dichos puntos y comprobar que la solución sigue siendo válida. Es decir, que sigue respetando el número de POIs que hay en la solución y que la hora en la que se finaliza la ruta sigue siendo la misma.

3.2. Diseño

3.2.1. Diagrama de clases

En este apartado se mostrarán los diagramas de clases de los elementos más importantes de la aplicación.

En la figura 3.1 se muestra el diagrama de clases del Fragment (ver explicación en el glosario de la Subsección 4.1.3) “TypesFragment”. Dicho diagrama de clases muestra las siguientes clases:

- La clase **TypesFragment**: esta clase es la encargada de comunicar los elementos gráficos que se encuentra dentro de la lista de alojamientos y POIs, también del botón que inicia el algoritmo de búsqueda de la ruta óptima.
- La clase **TypesRecyclerFragment**: es la encargada de manejar la lista de alojamientos y POIs, además devuelve los elementos seleccionados en la lista al iniciar la búsqueda de la ruta. También se encarga de gestionar la interfaz gráfica de la lista.
- La clase **CityNodesViewHolder**: es la encargada de gestionar los elementos gráficos de los POIs o los alojamientos de forma individual.
- La clase **TypeViewHolder**: es la encargada de gestionar los elementos gráficos de los tipos que aparecen en la lista de alojamientos y POIs.
- La clase **CityNode**: clase que contiene toda la información importante sobre un alojamiento o punto de interés que se muestra en la lista.

- La clase **TypeOfNode**: clase que contiene la información sobre un tipo, dicho tipo engloba a POIs del mismo tipo (por ejemplo: tipo Alojamiento).

En la figura 3.2, se muestra el diagrama de clases de la actividad principal de la aplicación. Las clases que se muestran en el diagrama son las siguientes:

- La clase **MapsActivity** es la clase principal y gestiona el mapa que se muestra y se comunica con la clase **TypesFragment** para obtener los nodos seleccionados. Esta clase contiene vectores que almacenan información sobre los marcadores que aparecen en el mapa. Dicha información se utiliza también para obtener la matriz de distancias. También contiene otros vectores para almacenar la información contenida en la solución y se utilizan para mandar la información de la solución a la actividad **ResultActivity**.
- La clase **DownloadFileFromURL**: se encarga de descargar y guardar la información que devuelven las peticiones a los servidores sobre alojamientos, POIs, rutas y matriz de tiempos entre POIs. Esta clase hereda de la clase AsyncTask, lo cual le permite ejecutarse en segundo plano sin que afecte al rendimiento de la interfaz de usuario.
- La clase **jsonProcessor** se encarga de procesar la información que se ha guardado en ficheros tras ser descargada. Esta clase utiliza la clase **JsonParser** para procesar los archivos. Además, también hereda de la clase AsyncTask por lo que se ejecuta en segundo plano.
- La clase **JsonParser** se encarga de procesar archivos JSON y devuelve la información contenida en los archivos en estructuras que la aplicación puede manipular.
- La clase **SendNodes** se encarga de obtener la matrix de distancias entre los POIs seleccionados. Para ello utiliza las clases **DownloadFileFromURL** y **jsonProcessor**, también se ejecuta en segundo plano debido a que hereda de la clase AsyncTask.
- La clase **FindSolution** se encarga de ejecutar en segundo plano el algoritmo de búsqueda de rutas y de mandar la solución a la siguiente actividad. Para ello hace uso de la clase **PathFinder** y la clase **Solution**.
- La clase **PathFinder** es la clase que contiene el algoritmo de búsqueda de rutas. Dicha clase obtiene la solución al problema mediante el algoritmo Greedy detallado en Alg. 1 y devuelve un objeto de la clase **Solution**.
- La clase **Solution** es la que clase que contiene una solución al problema. Esta clase cuenta con un vector que almacena los identificadores de los puntos de la solución, así como dos vectores que almacenan las horas de entrada y de salida de cada uno de los puntos de la solución.

En la figura 3.3, se muestra el diagrama de clases del paquete models; dicho paquete está formado por clases que se utilizan para modelar diferentes estructuras dentro del proyecto. Las clases que aparecen en el diagrama son las siguientes:

- La clase **Solution**: clase que contiene una solución al problema. Contiene vectores para almacenar identificadores y horarios de entrada y salida de los lugares por los que pasa la solución.
- La clase **ModelNode**: clase genérica que se utiliza para poder mostrar elementos tanto de la clase **CityNode** como de la clase **TypeOfNode**; ambas clases están explicadas en la descripción del primer diagrama de clases.
- La clase **SolutionNode**: clase que contiene el nombre, el horario de entrada y salida de un nodo de la solución. Esta clase se utiliza para encapsular los nodos de la solución y acceder a los datos a la hora de mostrar la lista de la solución.

En la figura 3.4 muestra el diagrama de clases de la Activity (ver explicación en el glosario de la Subsección 4.1.3) ResultActivity, dicha Activity se muestra cuando se obtiene una solución. Las clases que se muestran en el diagrama son las siguientes:

- La clase **ResultActivity**: clase que se ocupa de leer los datos mandados por la actividad principal y procesarlos para comunicárselos a la clase **SolutionFragment**.
- La clase **SimpleFragmentPagerAdapter**: clase que se ocupa de ver el número de soluciones que se han encontrado y crear un objeto de la clase **SolutionFragment** para mostrar cada una de ellas.
- La clase **SolutionFragment**: clase que se ocupa de mostrar en un mapa los marcadores de la solución y la lista con la información específica de cada uno de los marcadores. Además se ocupa de calcular el camino entre los distintos marcadores de la solución.

Por último, en la figura 3.5 se muestra el diagrama de clases de la clase **SolutionFragment**. A continuación se describen cada una de las clases que se muestran en el diagrama:

- La clase **SolutionFragment** se ha definido en la figura 3.4 anterior. Es la encargada de gestionar todas las vistas que muestran la solución.
- La clase **FindRoutes**: clase que se encarga de mandar una petición a un servidor para obtener la ruta óptima entre los nodos de la solución y procesarla, después manda la información procesada a la clase **SolutionFragment** para que dibuje la ruta.
- La clase **SolutionRecyclerAdapter** se encarga de manejar la lista con los nodos de la solución y de mostrarlos en una lista.

- La clase **SolutionNode**: clase que contiene la información sobre un nodo de la solución.
- La clase **SolutionNodeViewHolder**: clase que utiliza la información de un objeto de la clase **SolutionNode** y la muestra dentro de la lista que gestiona la clase **SolutionRecyclerAdapter**.

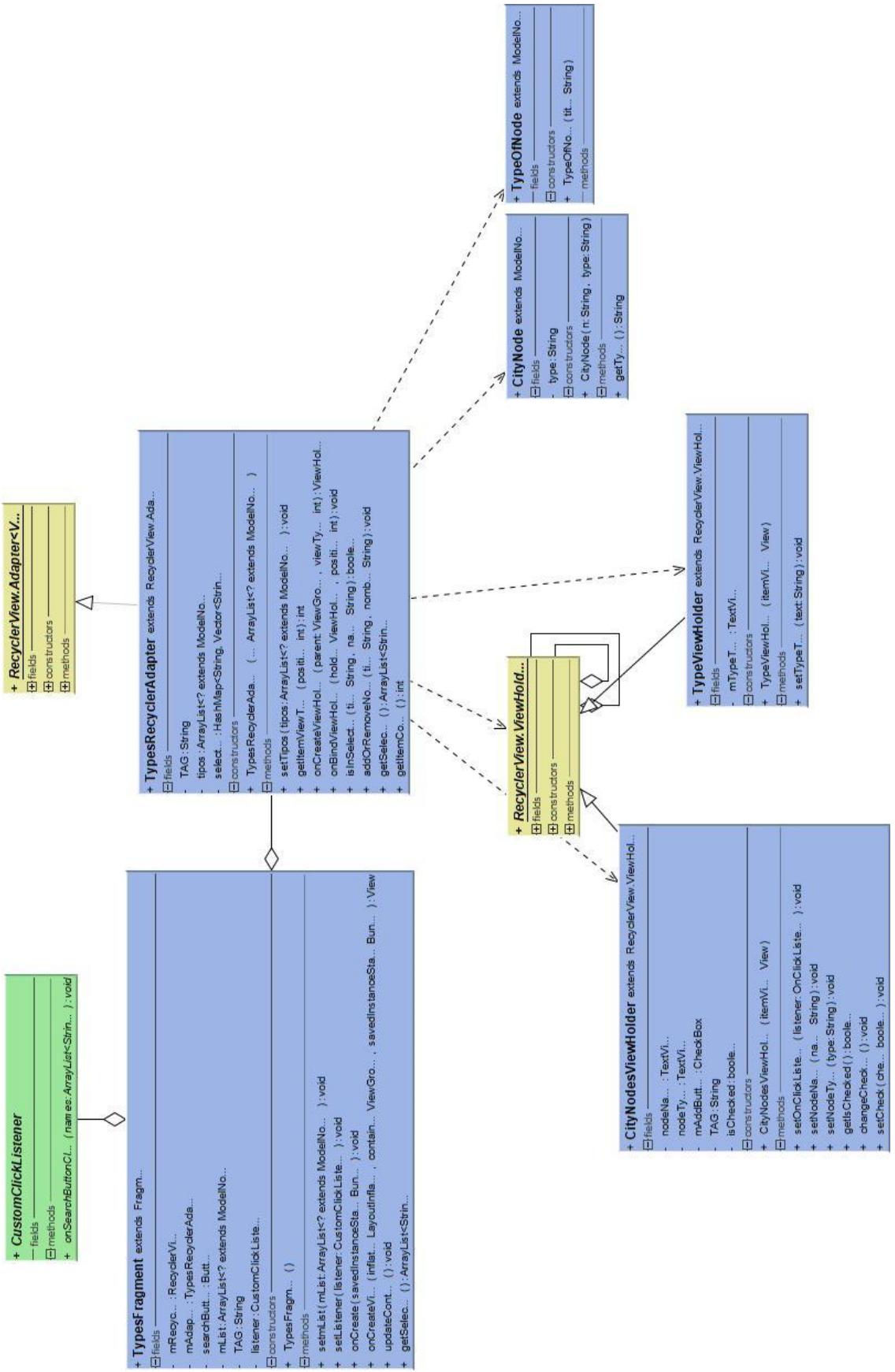


Figura 3.1: Diagrama de clases del **TypesFrament**.

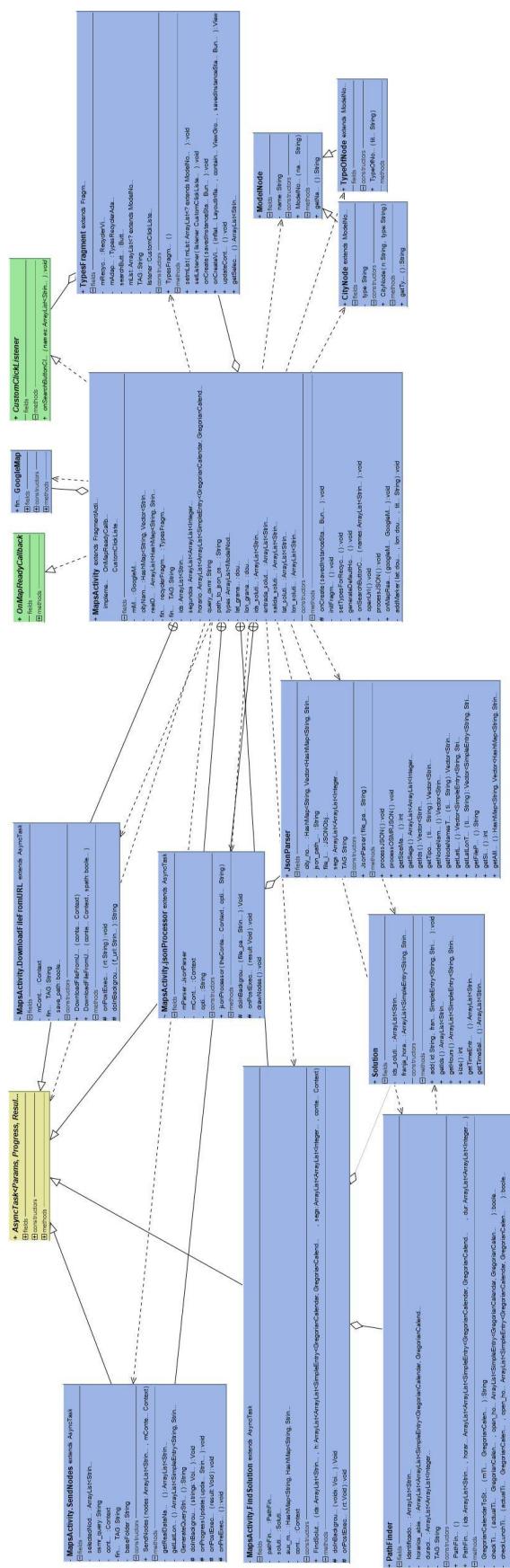
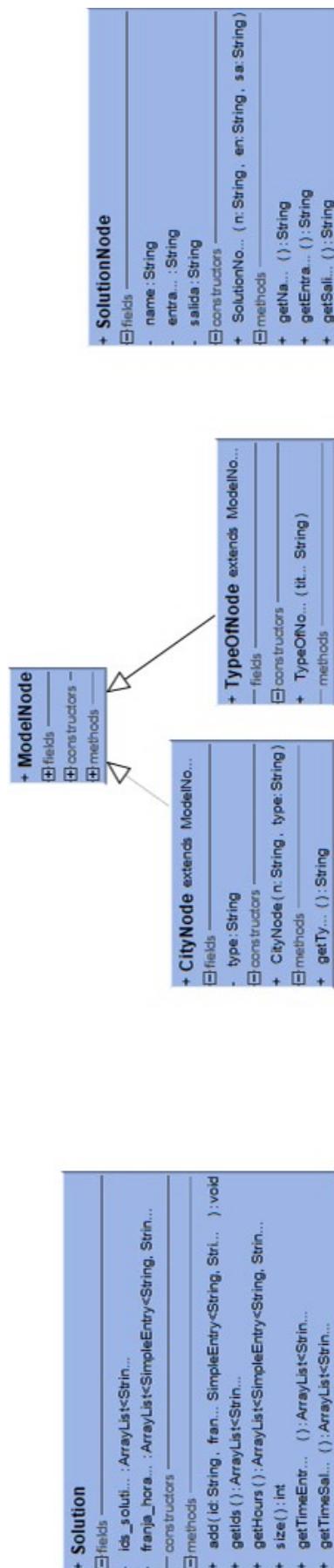
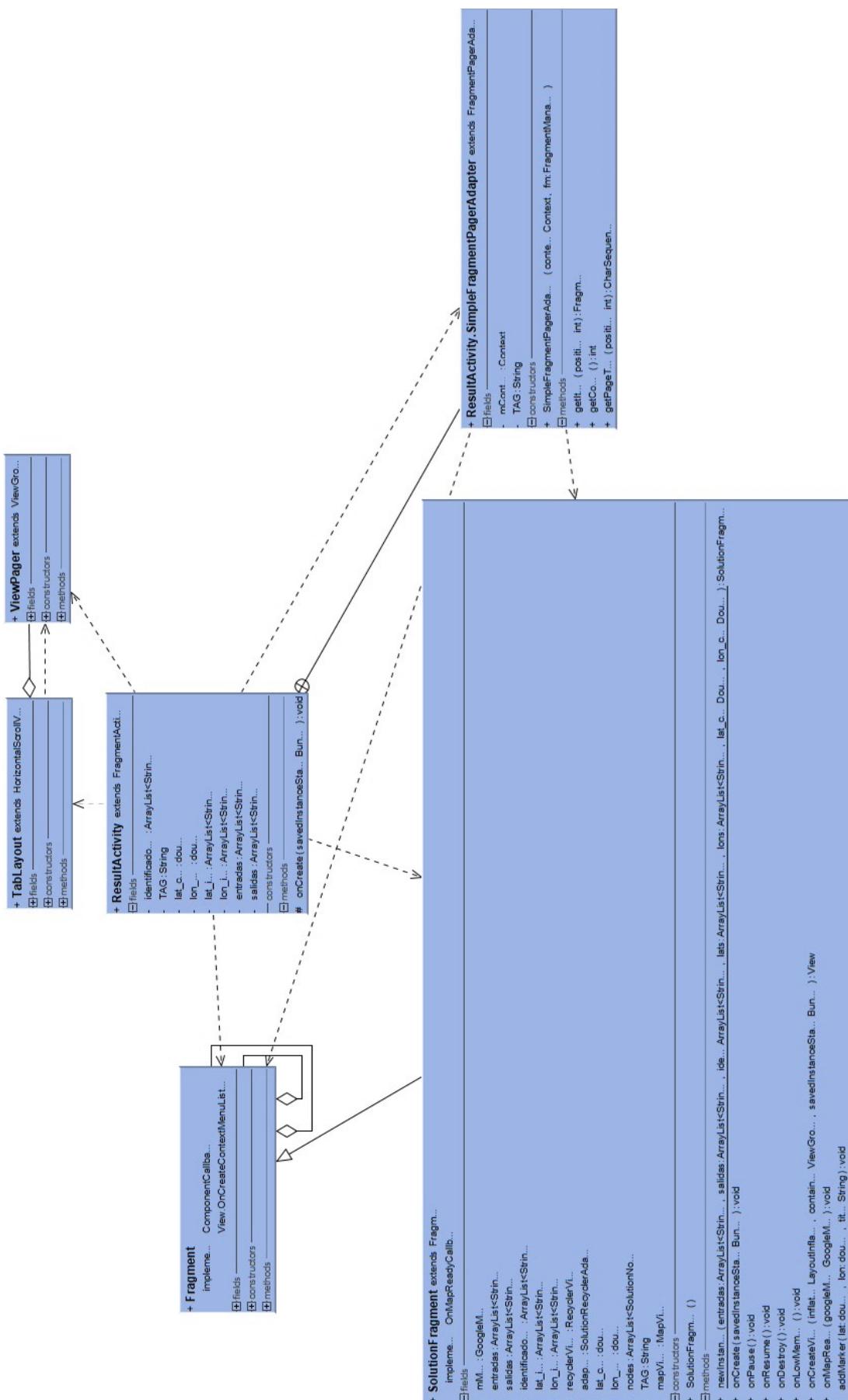
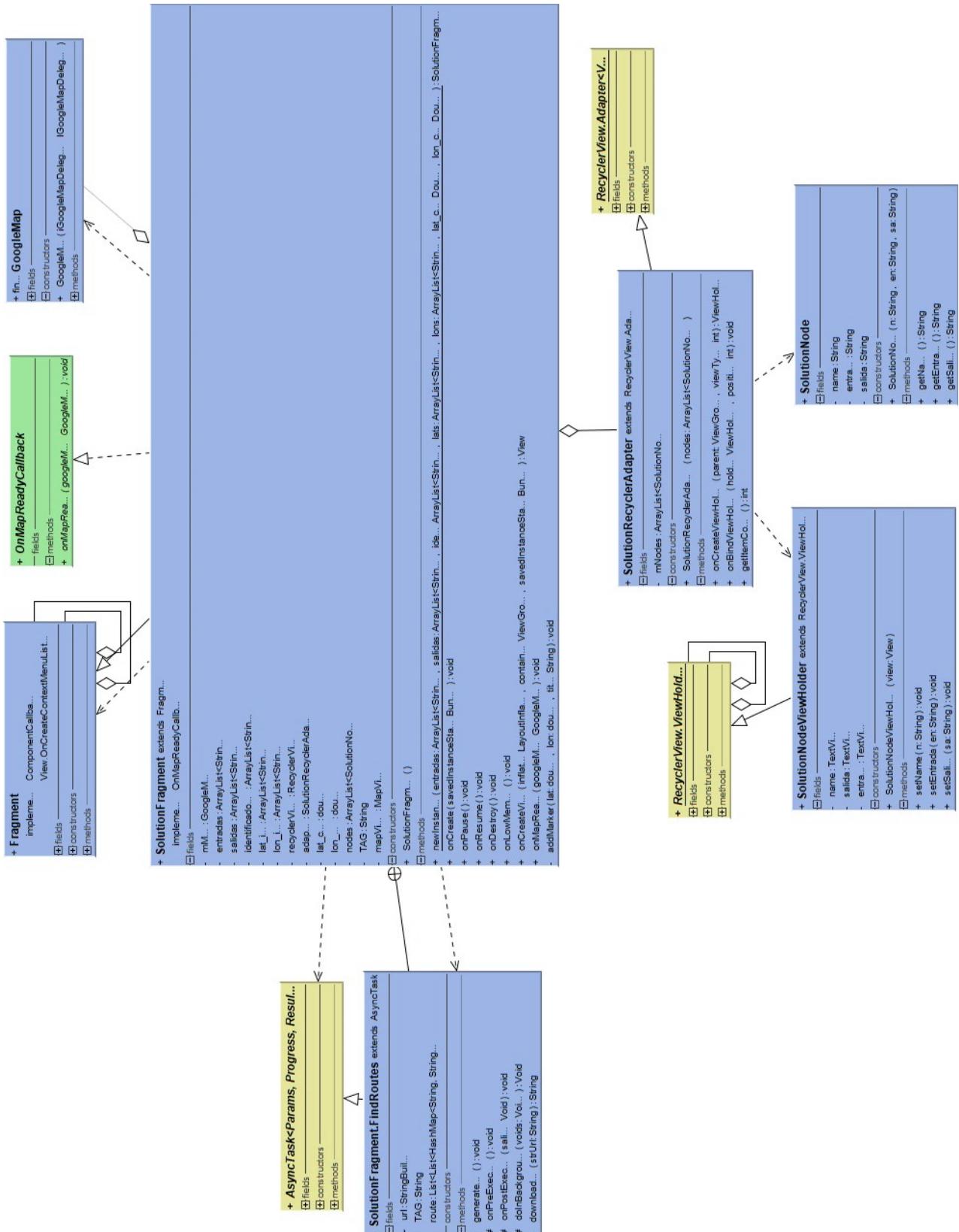


Figura 3.2: Diagrama de clases de la actividad principal 5.2.

Figura 3.3: Diagrama de clases del paquete **models**.

Figura 3.4: Diagrama de clases del activity **ResultActivity**.

Figura 3.5: Diagrama de clases de la clase **SolutionFragment**.

3.2.2. Interfaz de Usuario

A continuación se mostrarán todos los elementos que conforman la interfaz de usuario de la aplicación.

Lo primero que se muestra es la actividad principal que cuenta con un mapa, ver figura 3.2, por el que se puede navegar y una lista de alojamientos y POIs, entre los cuales el usuario puede elegir un alojamiento y el número de POIs que desee; como se muestra en la figura 3.7.

Para poder ver la lista, se debe deslizar la pestaña “Sitios interesantes” hacia arriba, así tendremos acceso a la lista completa, para explorar todos los alojamientos y POIs, debemos hacer scroll hacia arriba sobre la lista. Dicha lista muestra primero los alojamientos y tras estos los POIs, que se organizan en “Museos”, “Miradores” y “Monumentos”.

Dentro la lista, cada uno de los elementos contiene un caja en la que se puede pulsar para seleccionar o deseleccionar dicho elemento. En cada elemento de la lista, se encuentra también una caja, dicha caja permite seleccionar o deseleccionar todos los POIs de ese tipo.

Tras elegir el alojamiento y los POIs que el usuario desee, se ejecuta el algoritmo. Cuando este termina se abre una nueva actividad en la cual se muestran diferentes rutas al problema. Por defecto se muestra la primera ruta, para mostrar otras rutas, se debe pulsar sobre los tabs llamados “Solución X” para mostar otras soluciones.

En la figura 3.8 se muestran con marcadores los POIs seleccionados. Además, en la parte inferior de la pantalla se muestra una lista oculta que contiene la información detallada de la ruta; para poder ver dicha información se debe deslizar hacia arriba en “Descripción ruta final” o pulsar. Dentro de la lista, se muestra en orden de visita los POIs que contiene la ruta, el primer elemento que se muestra es el alojamiento seleccionado; además, cada uno de los POIs muestra la hora aproximada de entrada y de salida de dicho POI.

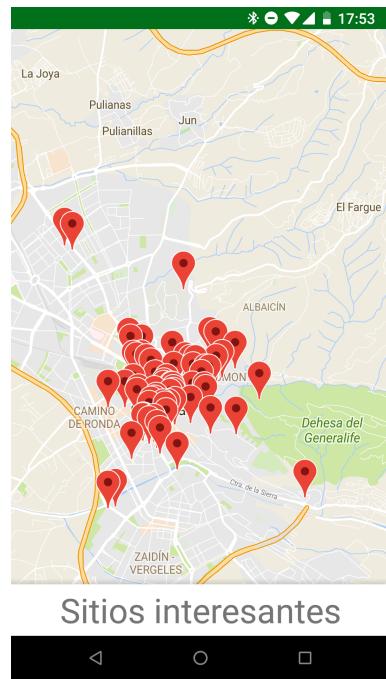


Figura 3.6: Mapa mostrando la localización de los POIs y alojamientos.

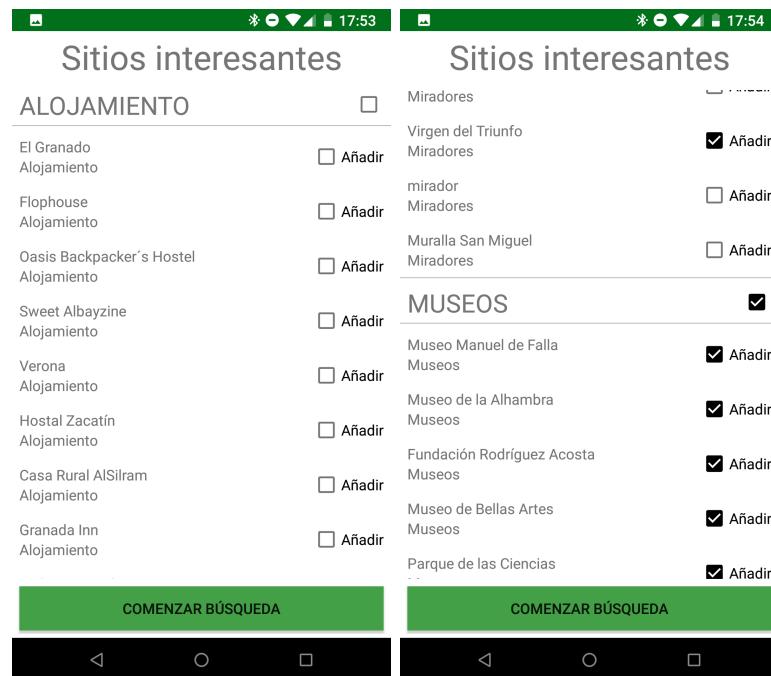


Figura 3.7: Lista de alojamientos y POIs disponibles para seleccionar.

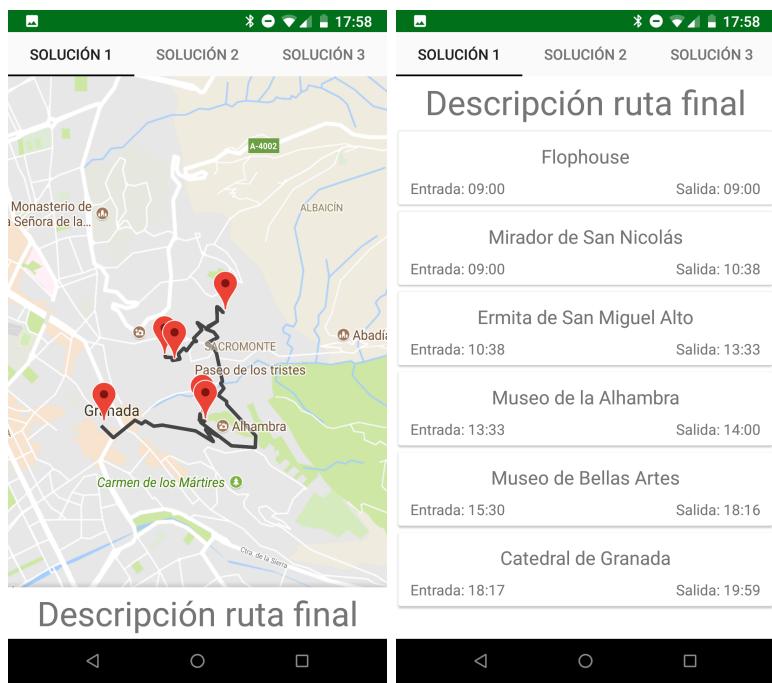


Figura 3.8: A la izquierda, mapa mostrando la solución. A la derecha, la descripción de los POIs.

Capítulo 4

Implementación y Pruebas

En este capítulo se muestran los aspectos más importantes del código de la aplicación, tanto en el lado de la interfaz como los procesos internos. Los métodos se organizan en realización a los requisitos especificados en el análisis. También se muestran diferentes ejemplos de caso de uso de la aplicación, junto con las soluciones que se obtienen.

4.1. Implementación

El código de la aplicación ha sido desarrollado en Java para implementar la funcionalidad; para la interfaz de usuario se ha utilizado XML. Como herramienta de desarrollo se ha utilizado Android Studio, la herramienta que ofrece Google para el desarrollo de aplicaciones en Android. La aplicación se ha compilado para la última versión de Android (SKD 27) [18], aunque es compatible con todos los dispositivos con una versión igual o mayor a la SDK 21 (Android Lollipop, 5.0).

4.1.1. Interfaz de usuario

Mostrar mapa

Con el objetivo de mostrar el mapa se ha utilizado la vista de Google Maps que tiene integrada Android, antes de poder utilizarla se debe crear una clave que permita a la aplicación acceder a los servicios de Google Maps. Para ello, hay que crear al menos un plan estándar desde la página de Google Maps API. Una vez se tenga la clave, hay que crear una vista del mapa dentro de la actividad donde se quiera mostrar dicho mapa y implementar la interfaz *OnMapReadyCallback*, para la cual hay que implementar el método *onMapReady(GoogleMap googleMap)*, el cuál se llama cuando se inicializa un objeto de la clase GoogleMaps y está preparado para usarse dentro de Android. El código necesario para mostrar el mapa es el siguiente.

Listado 4.1: código necesario para mostrar el mapa en la aplicación.

```
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback ,  
CustomClickListener {  
    private GoogleMap mMap;  
    ...  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        /*  
         * Inicializar otras vistas de la actividad  
         */  
  
        // Obtenemos mapa y esperamos a que esté preparado.  
        SupportMapFragment mapFragment = (SupportMapFragment)  
            getSupportFragmentManager()  
            .findFragmentById(R.id.map);  
        mapFragment.getMapAsync(this);  
        initFragment();  
  
        Log.e("map: ", "mapa puesto");  
    }  
  
    /*  
     * Otros procesos de la actividad  
     */  
  
    @Override  
    public void onMapReady(GoogleMap googleMap) {  
        mMap = googleMap;  
        // Cambiamos estilo del mapa.  
        try {  
            boolean success = googleMap.setMapStyle(  
                MapStyleOptions.loadRawResourceStyle(  
                    this, R.raw.style_json));  
  
            if (!success) {  
                Log.e("map_style", "Style parsing failed.");  
            }  
        } catch (Resources.NotFoundException e) {  
            Log.e("map_style", "Can't find style. Error: ", e);  
        }  
  
        // Cambiamos posición del mapa para mostrar la ciudad de Granada.  
        LatLng granada = new LatLng(lat_granada, lon_granada );  
        mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(granada ,13));  
    }  
}
```

El archivo de la vista de la actividad sería el siguiente.

Listado 4.2: Código XML de la vista de la actividad principal.

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.sothree.slidinguppanel.SlidingUpPanelLayout
        xmlns:sothree="http://schemas.android.com/apk/res-auto"
        android:id="@+id/sliding_panel"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="bottom"
        sothree:layout_constraintBottom_toBottomOf="parent"
        sothree:layout_constraintEnd_toEndOf="parent"
        sothree:layout_constraintStart_toStartOf="parent"
        sothree:layout_constraintTop_toTopOf="parent"
        sothree:umanoDragView="@+id/recyclerLayout"
        sothree:umanoOverlay="true"
        sothree:umanoPanelHeight="?android:attr actionBarSize"
        sothree:umanoShadowHeight="6dp">

        <!-- MAIN CONTENT -->
        <fragment
            android:id="@+id/map"
            android:name="com.google.android.gms.maps.SupportMapFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            />

        <!-- SLIDING LAYOUT -->
        <FrameLayout
            android:id="@+id/recyclerLayout"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

    </com.sothree.slidinguppanel.SlidingUpPanelLayout>
</android.support.constraint.ConstraintLayout>
```

Lista de POIs/alojamiento y lista solución

Para crear una lista en Android existen diferentes estructuras, la más reciente y más versátil es crear un *RecyclerView*, esta clase está disponible en la librería *Android Support Library*.

Para utilizar un *RecyclerView* en Android, se debe utilizar un objeto de la clase *RecyclerView* en la actividad o Fragment en el que se vaya a mostrar la lista. Además, hay que crear una clase que herede de la clase *RecyclerView.Adapter* para indicar el

comportamiento de la lista, dentro de esta clase se debe sobreescribir al menos los métodos *onCreateViewHolder()*, que es llamado cuando se crea un nuevo objeto en la lista; *onBindViewHolder()*, que es llamado cuando se va a mostrar un objeto en pantalla de la lista y se le asocia una posición dentro de la lista; y la función *getItemCount()*, que devuelve el número de elementos de la lista.

Una vez se haya creado este objeto se debe de indicar al *RecyclerView* cual es el *Adapter* que se va a utilizar, esto se hace con el método *RecyclerView.setAdapter(Clase_Adapter)*. También es necesario especificar el layout que tendrá el *RecyclerView*, para ello hay que utilizar el método *RecyclerView.setLayoutManager(tipo_layout_manager)*; existen diferentes tipos de layout posibles para el *RecyclerView*, pero si se quiere mostrar una lista se debe utilizar como “*tipo_layout_manager*” un objeto de la clase *LinearLayoutManager*; para este objeto hay que especificar el contexto al crear el objeto, para ello se debe utilizar la función *getActivity()*, si se va a utilizar dentro de un Fragment, o “this” si se va a utilizar en una Activity. El código necesario para esto sería el siguiente.

Listado 4.3: Código de una clase que contiene un RecyclerView.

```
public class TypesFragment extends Fragment {

    private RecyclerView mRecycler;
    private TypesRecyclerAdapter mAdapter;
    private Button searchButton;
    private ArrayList<?extends ModelNode> mList = new ArrayList<>();

    public TypesFragment() {
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {

        // Obtenemos la vista del layout.
        View recyclerViewLayout = inflater.inflate(R.layout.fragment_types,
            container, false);

        // Obtenemos el botón y el RecyclerView.
        searchButton = (Button) recyclerViewLayout.findViewById(R.id.busqueda);
        mRecycler = (RecyclerView) recyclerViewLayout.findViewById(R.id.recycler);

        // Configuramos el RecyclerView.
        mAdapter = new TypesRecyclerAdapter(mList);
        mRecycler.setLayoutManager(new LinearLayoutManager(getActivity()));
        mRecycler.setAdapter(mAdapter);

        // Añadir funcionalidad al botón.
        searchButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                listener.onSearchButtonClick(getSelected());
            }
        });
        return recyclerViewLayout;
    }

}
```

Listado 4.4: Código para crear una lista en Android.

```

public class TypesRecyclerAdapter extends
    RecyclerView.Adapter<RecyclerView.ViewHolder> {

    private String TAG = TypesRecyclerAdapter.class.getSimpleName();
    private ArrayList<?extends ModelNode> tipos = new ArrayList<>();
    private HashMap<String, Vector<String>> selected = new HashMap<>();
    private ArrayList<String> type_selected = new ArrayList<>();

    // Constructor.
    public TypesRecyclerAdapter(ArrayList<?extends ModelNode> all) {
        this.tipos = all;
    }

    public void setTipos(ArrayList<?extends ModelNode> tipos){
        this.tipos = tipos;
    }

    // Función para distinguir entre diferentes tipos de clases dentro de un
    // RecyclerView.
    @Override
    public int getItemViewType(int position) {
        int pos=0;
        if(tipos.get(position).getClass().toString().contains("CityNode")){
            pos = 2;
        }
        return pos;
    }

    // Función llamada cuando se crea un nuevo elemento dentro del RecyclerView.
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
    {
        View view;
        switch (viewType){
            // Caso tipo de nodo.
            case 0:
                view =
                    LayoutInflater.from(parent.getContext()).inflate(R.layout.types_view
parent, false);
                return new TypeViewHolder(view);

            // Caso nodo concreto.
            case 2:
                view =
                    LayoutInflater.from(parent.getContext()).inflate(R.layout.nodes_view
parent, false);
                return new CityNodesViewHolder(view);
        }
        return null;
    }

    // Función llamada cuando se va a dibujar un elemento en el RecyclerView.
    @Override
    public void onBindViewHolder(final RecyclerView.ViewHolder holder, final int
position) {
        // Seleccionamos tipo de vista y establecemos información que se quiere
        mostrar
    }
}

```

```
switch (holder.getItemViewType()){
// Caso tipo de nodo.
case 0:
    final TypeViewHolder typeViewHolder = (TypeViewHolder) holder;
    final TypeOfNode typeNode = (TypeOfNode) tipos.get(position);
    typeViewHolder.setTypeText(typeNode.getName());
    typeViewHolder.setCheck(isInTypeSelected(typeNode.getName()));
    typeViewHolder.setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View v){
            typeViewHolder.changeChecked();
            addOrRemoveType(typeNode.getName());
        }
    });
    break;

// Caso nodo concreto.
case 2:
    final CityNode cityNode = (CityNode) tipos.get(position);
    final CityNodesViewHolder node = (CityNodesViewHolder) holder;
    node.setNodeName(cityNode.getName());
    node.setNodeType(cityNode.getType());
    node.setCheck(isInSelected(cityNode.getType(),cityNode.getName()));
    node.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            node.changeChecked();
            addOrRemoveNode(cityNode.getType(),
                            cityNode.getName());
        }
    });
    break;
}

...
// Función que devuelve número de elementos en el RecyclerView.
@Override
public int getItemCount() {
    return tipos.size();
}
```

Listado 4.5: Código XML de la vista de una lista de elementos.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/typesLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".fragment.TypesFragment"
    android:background="@color/white"
    >

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginBottom="8dp"
        android:layout_marginTop="8dp"
        android:clickable="true"
        android:focusable="true"
        app:layout_constraintBottom_toTopOf="@+id/busqueda"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        tools:listitem="@layout/types_viewholder" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="0dp"
        android:layout_height="?android:attr actionBarSize"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:text="@string/textViewSlideUpPanel"
        android:textSize="32sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:gravity="center"/>

    <Button
        android:id="@+id/busqueda"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:background="@color/searchGreen"
        android:text="@string/startSearch"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />
</android.support.constraint.ConstraintLayout>
```

Una vez se tiene creada y definida la lista, se debe crear una clase más para especificar la vista y el comportamiento individual de cada uno de los elementos de dicha lista. Esta nueva clase debe heredar de la clase *RecyclerView.ViewHolder*; esta debe tener además un fichero XML que defina la estructura de la vista de dicha clase. El código necesario para crear dicha clase es el siguiente.

Listado 4.6: Código para crear un elemento de una lista en Android.

```
public class CityNodesViewHolder extends RecyclerView.ViewHolder {
    private TextView nodeName;
    private TextView nodeType;
    private CheckBox mAddButton;
    private String TAG = CityNodesViewHolder.class.getSimpleName();
    private boolean isChecked = false;

    public CityNodesViewHolder(View itemView){
        super(itemView);

        // Obtenemos vistas.
        nodeName = (TextView) itemView.findViewById(R.id.nodeName);
        mAddButton = (CheckBox) itemView.findViewById(R.id.addButton);
        nodeType = (TextView) itemView.findViewById(R.id.type_name);
        mAddButton.setChecked(isChecked);
    }
    ...
}
```

Listado 4.7: Código XML de la vista de un elemento de la lista.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/node_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/nodeName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintEnd_toStartOf="@+id/addButton"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <CheckBox
        android:id="@+id/addButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/add_checkbox"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/type_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginStart="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/nodeName" />

</android.support.constraint.ConstraintLayout>
```

Mostrar diferentes soluciones

Para mostrar diferentes soluciones se optó por mostrar cada una de las diferentes soluciones en un mapa distinto, para ello, se debe crear una vista del mapa y una lista para mostrar la solución por cada solución; este proceso está explicado en los apartados anteriores, por lo que no se va a entrar en más detalle en este. Para poder navegar por

las diferentes vistas de las soluciones, hay que crear una vista que permita seleccionar la vista de la solución que se quiera en cada momento, para esto se puede utilizar la clase *TabLayout*; dicha clase debe ir acompañada de un objeto de la clase *ViewPager*. Ambas clases están contenidas en la biblioteca por defecto de Android.

La clase *TabLayout* solo se utiliza para definir tabs que se muestran en pantalla, y cada uno de ellos hace referencia a una vista diferente. Para añadir comportamiento a cada uno de los tabs, se debe preparar al *TabLayout* para ser usado por la clase *ViewPager*, esto se hace utilizando el método *TabLayout.setupWithViewPager(viewpager_obj)*.

Para definir el comportamiento del objeto de la clase *ViewPager*, hay que crear una clase que defina dicho comportamiento y especificárselo al objeto de la clase *ViewPager* con el método *ViewPager.setAdapter(adapter)*. Una de las posibles formas de hacer esto es crear una clase interna a la clase en la que se utiliza el *ViewPager* que herede de la clase *FragmentViewPager*. Dentro de esta clase se deben sobreescribir al menos los métodos *getItem()*, que determina el Fragment que se debe utilizar para cada tab; *getCount()*, que devuelve el número de tabs; y *getPageTitle()*, que determina el nombre de cada uno de los tabs según su posición.

El siguiente código se encarga de hacer lo descrito en los párrafos anteriores.

Listado 4.8: Código de una vista con tabs dentro de ella.

```
public class ResultActivity extends FragmentActivity {

    /*
     * Atributos de la clase
     */

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /*
         * Inicializar de otras vistas y atributos
         */

        ViewPager viewPager = (ViewPager) findViewById(R.id.viewpager);
        SimpleFragmentPagerAdapter adapter = new
            SimpleFragmentPagerAdapter(this, getSupportFragmentManager());
        viewPager.setAdapter(adapter);

        TabLayout tabLayout = (TabLayout) findViewById(R.id.sliding_tabs);
        tabLayout.setupWithViewPager(viewPager);
    }

    public class SimpleFragmentPagerAdapter extends FragmentPagerAdapter {

        private Context mContext;
        private String TAG =
            SimpleFragmentPagerAdapter.class.getSimpleName();
    }
}
```

```
public SimpleFragmentPagerAdapter(Context context,
    FragmentManager fm) {
    super(fm);
    mContext = context;
}

// Esta función determina el Fragment de cada uno de los tabs.
@Override
public Fragment getItem(int position) {
    Log.i(TAG, "position " + position);
    if (position == 0) {
        return
            SolutionFragment.newInstance(entradas, salidas, identificadores);
    } else if (position == 1){
        return
            SolutionFragment.newInstance(entradas, salidas, identificadores);
    } else if (position == 2){
        return
            SolutionFragment.newInstance(entradas, salidas, identificadores);
    } else{
        return null;
    }
}

// Esta función determina el número de tabs.
@Override
public int getCount() {
    return 3;
}

// Esta función determina el título de cada uno de los tabs.
@Override
public CharSequence getPageTitle(int position) {
    // Generar título dependiendo de la posición.
    switch (position) {
        case 0:
            return "Sol 1";
        case 1:
            return "Sol 2";
        case 2:
            return "Sol 3";
        default:
            return null;
    }
}
}
```

Listado 4.9: Código XML de la vista con tabs.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/result_constraint"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <android.support.design.widget.TabLayout
        android:id="@+id/sliding_tabs"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:tabMode="fixed" />

    <android.support.v4.view.ViewPager
        android:id="@+id/viewpager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/white" />

</LinearLayout>
```

4.1.2. Procesos internos

En este apartado se describen las funciones de envío de peticiones a servidores, procesado de las respuestas de los servidores y el cálculo de la ruta óptima.

Peticiones a servidores

La aplicación cuenta con procesos que se encargan de mandar peticiones a servidores para obtener información sobre POIs, distancias entre puntos, o camino óptimo entre dos puntos. Para enviar dichas peticiones, se debe especificar una URL para la cual el servidor pueda devolver una respuesta, una vez se ha obtenido la respuesta se guarda en un archivo o en un String para procesarlo. La función que descarga las respuestas de servidores se puede encontrar en la función *doInBackground()* de la clase *DownloadFileFromURL*.

Listado 4.10: Función para enviar peticiones a servidores y guardar respuesta.

```

@Override
protected String doInBackground(String... f_url) {
    int count;
    String baseFolder = mContext.getFilesDir().getAbsolutePath();
    File file = new File(baseFolder + File.separator + f_url[1]);
    Log.i(TAG, "starting download");

    try {
        URL url = new URL(f_url[0]);
        HttpURLConnection connection = (HttpURLConnection)
            url.openConnection();

        // Descargar el archivo
        InputStream input = new BufferedInputStream(url.openStream(),
            8192);

        file.getParentFile().mkdirs();
        OutputStream output = new FileOutputStream(file);

        Log.i(TAG, "file_out: "+"File opened");
        Log.i(TAG, "file_out:"+ "File saved in: " +
            mContext.getFilesDir().getAbsolutePath());

        byte data[] = new byte[1024];

        long total = 0;

        while ((count = input.read(data)) != -1) {
            total += count;
            // Escribimos los datos en el archivo.
            output.write(data, 0, count);
        }

        // Limpiamos salida.
        output.flush();

        // Cerramos los flujos y la conexión a internet.
        output.close();
        input.close();
        connection.disconnect();

        Log.i(TAG, "file_out:"+ "Finished writting output");

    } catch (Exception e) {
        Log.e(TAG, e.getMessage());
    }

    // Devolvemos PATH del archivo.
    return baseFolder + File.separator + f_url[1];
}

```

Cuando la función termina, devuelve el “PATH” donde está guardado el archivo que contiene la respuesta del servidor. El parámetro que se le pasa a la función *doInBackground(String... f_url)* se trata de un vector que contiene como primer elemento la URL que la función debe abrir y guardar el contenido; el segundo elemento contiene el nombre del archivo que contendrá la respuesta del servidor.

Procesado de respuestas de servidor

Para el procesado de la información que devuelven los servidores, se ha creado la clase *JsonParser*. Esta clase procesa los archivos y guarda la información en estructuras para poder utilizarlas después. Para procesar los datos, se debe pasar el “PATH” del archivo, para abrirlo y meterlo en un String que después se utilizará para inicializar un objeto del tipo *JSONObject*; o se puede pasar el String que contiene el fichero directamente.

Procesado de alojamientos y POIs:

Para el procesado del archivo que contiene la información sobre alojamientos y POIs existe el método *processJSON()*, este método no tiene parámetros, por lo que hay que crear el objeto de la clase *JsonParser* indicando el “PATH” del archivo.

Dentro de este método se recorre el array llamado “elements”, por cada uno de los elementos del array se crea un objeto *JSONObject* y se obtiene la información sobre el identificador del nodo, latitud, longitud, nombre y tipo de nodo en un hashmap. El código es el siguiente.

Listado 4.11: Función para procesar información sobre POIs y alojamientos.

```

public void processJSON() throws JSONException{
    JSONArray elements = file_info.getJSONArray("elements");
    for(int i=0; i < elements.length(); i++){
        JSONObject node = elements.getJSONObject(i);

        // Obtenemos datos del archivo.
        String id = node.optString("id");
        String latitud = node.optString("lat");
        String longitud = node.optString("lon");
        JSONObject tags = node.getJSONObject("tags");
        String name = tags.optString("name", "desconocido");

        String tipo = tags.optString("tourism");
        if( tipo.equals("") ){
            tipo = tags.optString("amenity");
        }

        HashMap<String, String> aux_hashMap = new HashMap<>();
        // Utilizamos un hashMap auxiliar.
        if( !name.equals("desconocido") ) {
            aux_hashMap.put("id", id);
            aux_hashMap.put("lat", latitud);
            aux_hashMap.put("lon", longitud);
            aux_hashMap.put("name", name);

            if(!city_nodes.containsKey(tipo)){
                // Metemos un nuevo tipo.
                System.out.println("nuevo tipo: " + tipo);
                Vector<HashMap<String, String>> v_aux = new Vector<>();
                v_aux.add(aux_hashMap);
                city_nodes.put(tipo, v_aux);
            }else{
                // Añadimos a un tipo existente.
                System.out.println("adding new map to "+ tipo);
                city_nodes.get(tipo).add(aux_hashMap);
            }
        }
    }
}

```

Procesado de la matriz de tiempos:

Para el procesado de la matriz se ha implementado el método *processOSRMJSON()*, dicho método necesita que antes se inicialice el objeto de la clase *JsonParser* para que funcione correctamente. Este método procesa el array llamado “durations” donde va guardando en una matriz el tiempo necesario para llegar desde un punto a otro en segundos. El código de dicho método es el siguiente.

Listado 4.12: Función para procesar matriz de tiempos entre puntos.

```
public void processOSMRJSON() throws JSONException{  
  
    JSONArray times = file_info.getJSONArray("durations");  
    ArrayList<Integer> tim = new ArrayList<>();  
    // Recorremos la matriz.  
    for(int i=0; i < times.length(); i++){  
        JSONArray aux_t = times.getJSONArray(i);  
        for(int j=0; j < aux_t.length(); j++){  
            // Obtenemos distancia.  
            int dist_time = aux_t.getInt(j);  
            Log.i(TAG,dist_time+"");  
            tim.add(dist_time);  
        }  
  
        segs.add(tim);  
        tim = new ArrayList<>();  
    }  
  
}
```

Procesado de caminos entre puntos de la solución:

Para el procesado de caminos entre dos puntos se creó el método *parseRoutes(String geo_coord)*. Este método recorre el array llamado “steps”, dentro de este array se toma el String “points” que está contenido en el objeto “polyline”, después este String se debe decodificar [19]. Para entrar en el array “steps” se debe primero entrar en el array “routes” y dentro de este en “legs”.

El array “routes” contiene todas las rutas posibles entre los puntos que se le indiquen; las rutas se dividen en etapas, que representan la ruta entre dos puntos; dichas etapas se representan con los arrays “legs”. Dentro de una etapa puede haber más de un paso a seguir, dichos pasos se representan con los arrays “steps”.

El código de la función de procesado de rutas y de decodificar las polilíneas es el siguiente.

Listado 4.13: Función para procesar los caminos entre los puntos de una ruta.

```

public List<List<HashMap<String ,String>>> parseRoutes(String geo_coord) throws
JSONException{
    JSONObject object = new JSONObject(geo_coord);
    Log.i(TAG,object.toString());
    List<List<HashMap<String , String>>> routes = new ArrayList<>() ;
    JSONArray jRoutes;
    JSONArray jLegs;
    JSONArray jSteps;

    try {

        jRoutes = object.getJSONArray("routes");

        // Recorremos rutas.
        for(int i=0;i<jRoutes.length();i++){
            jLegs = (
                (JSONObject)jRoutes.get(i)).getJSONArray("legs");
            List path = new ArrayList<>();
            // Recorremos Legs.
            for(int j=0;j<jLegs.length();j++){
                jSteps = (
                    (JSONObject)jLegs.get(j)).getJSONArray("steps");
                    // Recorremos cada uno de los pasos.
                    for(int k=0;k<jSteps.length();k++){
                        String polyline = "";
                        polyline =
                            ((String)((JSONObject)((JSONObject)
jSteps.get(k)).get("polyline")).get("points"));
                        List<LatLng> list = decodePoly(polyline);

                        for(int l=0;l<list.size();l++){
                            HashMap<String , String> hm = new
                                HashMap<>();
                            hm.put("lat",
                                Double.toString((list.get(l)).latitude)
                                );
                            hm.put("lng",
                                Double.toString((list.get(l)).longitude)
                                );
                            path.add(hm);
                        }
                    }
                    routes.add(path);
                }
            }

        } catch (JSONException e) {
            e.printStackTrace();
        }catch (Exception e){
    }

    return routes;
}

```

Listado 4.14: Función para decodificar polilíneas.

```

private ArrayList<LatLng> decodePoly(String encoded) {

    ArrayList<LatLng> poly = new ArrayList<LatLng>();
    int index = 0, len = encoded.length();
    int lat = 0, lng = 0;

    while (index < len) {
        int b, shift = 0, result = 0;
        do {
            b = encoded.charAt(index++) - 63;
            result |= (b & 0x1f) << shift;
            shift += 5;
        } while (b >= 0x20);
        int dlat = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
        lat += dlat;

        shift = 0;
        result = 0;
        do {
            b = encoded.charAt(index++) - 63;
            result |= (b & 0x1f) << shift;
            shift += 5;
        } while (b >= 0x20);
        int dlng = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
        lng += dlng;

        LatLng p = new LatLng(((double) lat / 1E5),
            (((double) lng / 1E5)));
        poly.add(p);
    }

    return poly;
}

```

Cálculo de ruta

Para el cálculo de la ruta óptima se ha creado el método *obtainGreedySolution(starting_time)*; esta función implementa un algoritmo Greedy para calcular la ruta. Para ejecutar el algoritmo se debe de inicializar un objeto de la clase *PathFinder* pasándole como argumentos la lista de identificadores de los puntos seleccionados, la matriz de tiempos entre puntos y los horarios en los que cada POI está abierto.

Una vez comienza el algoritmo, el primer nodo, que corresponde con el alojamiento que el usuario ha seleccionado, se introduce dentro de la solución y se inicializa una variable que contiene el tiempo actual en el algoritmo. Esta variable se inicializa con el valor del parámetro “*starting_time*”. Después, se selecciona el punto más cercano a este

que se pueda visitar y se introduce en la solución; también se añade al tiempo actual el tiempo que se tarda en realizar la visita y el tiempo necesario para llegar hasta dicho punto. Finalmente se actualiza el punto actual, que pasa a ser el punto seleccionado. Esta operación se repite hasta que no quedan más puntos por seleccionar o hasta que se llega a la hora límite, esta hora por defecto es las 20:00.

Una vez se termina el algoritmo, se devuelve la solución, un objeto de la clase *Solution*; que contiene un subconjunto de los puntos que se habían sido seleccionados por el usuario. La implementación del algoritmo se muestra en la siguiente figura.

Listado 4.15: Función para encontrar la ruta entre los puntos seleccionados.

```
public Solution obtainGreedySolution(GregorianCalendar starting_time){
    Solution m_solution = new Solution();
    GregorianCalendar finish_time = new GregorianCalendar(1,1,1,20,0,0);
    GregorianCalendar current_time = new GregorianCalendar();
    current_time = starting_time;
    GregorianCalendar aux_time = new GregorianCalendar();
    aux_time = current_time;
    Vector<Integer> solution = new Vector<Integer>();
    Vector<String> ids_solution = new Vector<String>();
    Vector<Integer> non_added = new Vector<Integer>();
    for(int i=0; i < identificadores.size(); i++){
        non_added.add(i);
    }
    Vector<Integer> valid = new Vector<Integer>(non_added);
    boolean added = false;
    Integer id = 0;
    Integer visita = 0;
    GregorianCalendar aux_greg = new GregorianCalendar();

    if(identificadores.size() <= 0){
        return m_solution;
    }

    solution.add(id);
    non_added.remove(id);
    valid.remove(id);
    m_solution.add(identificadores.get(0),new SimpleEntry<String,
                    String>(GregorianCalendarToString(starting_time),
                    GregorianCalendarToString(starting_time)));

    // Continuamos hasta que no queden sitios por visitar o no quede más
    // tiempo.
    while( current_time.before(finish_time) && !non_added.isEmpty()){
        added = false;
        valid = non_added;

        // Repetimos hasta que consigamos añadir un nuevo elemento.
        while(!added && !valid.isEmpty()){
            // Calculamos el museo al cual tardamos menos en llegar
            // desde donde estamos.
            Integer pos = findNearest(id,valid);
            aux_time = (GregorianCalendar)current_time.clone();
            aux_time.add(GregorianCalendar.SECOND,duracion.get(id).get(pos));
            if(aux_time.after(finish_time)){
                break;
            }
            added = true;
            valid.remove(pos);
            solution.add(id);
            non_added.remove(id);
            id = pos;
        }
    }
}
```

```
// Comprobamos si está abierto.
if( checkTime( aux_time, horarios_abierto.get(pos-1)) ) {
    current_time.add(
        GregorianCalendar.SECOND,duracion.get(id).get(pos));
    aux_greg = (GregorianCalendar)
        current_time.clone();

    visita =
        ThreadLocalRandom.current().nextInt(60,180+1);
    current_time.add(GregorianCalendar.MINUTE,visita
    );

    current_time =
        checkIfNotClosed(current_time,horarios_abierto.get(pos-1));

    id = pos;
    solution.add(id);
    added = true;
    non_added.remove(id);
    valid.remove(id);

    m_solution.add(identificadores.get(id),new
        SimpleEntry<>(GregorianCalendarToString(aux_greg),
        GregorianCalendarToString(current_time)));

}

// Comprobamos si está cerrado por horario de comida.
else if(checkLunchTime(aux_time,
    horarios_abierto.get(pos-1))){
    if(horarios_abierto.get(pos-1).size() > 1) {
        current_time = horarios_abierto.get(pos -
            1).get(1).getKey();
    }else{
        current_time =
            horarios_abierto.get(pos-1).get(0).getKey();
    }
    aux_greg = (GregorianCalendar)
        current_time.clone();

    visita =
        ThreadLocalRandom.current().nextInt(60,180+1);
    current_time.add(GregorianCalendar.MINUTE,visita
    );

    current_time =
        checkIfNotClosed(current_time,horarios_abierto.get(pos-1));

    id = pos;
    solution.add(id);
    added = true;
    non_added.remove(id);
    valid.remove(id);

    m_solution.add(identificadores.get(id),new
        SimpleEntry<>(GregorianCalendarToString(aux_greg),
        GregorianCalendarToString(current_time)) );
}

}
```

```

        // Lo borramos ya que no es válido en este momento.
        else{
            valid.remove(pos);
        }

    }

    if(valid.isEmpty() && !added){
        non_added.clear();
    }

}

// Devolvemos la solución.
return m_solution;
}

```

Dentro de esta función se encuentran funciones para comprobar si el tiempo es válido. La función *checkTime(tiempo_actual, horarios_punto)* comprueba si el valor del parámetro “tiempo_actual” está dentro del horario del POI, si está dentro devuelve true, sino false.

La función *checkLunchTime(tiempo_actual, horario_punto)* comprueba si el parámetro “tiempo_actual” está entre el horario de mañana del punto y el horario de tarde del punto, si es así devuelve true, sino false. Esta función se utiliza para cambiar el valor de la variable “tiempo_actual” hasta que el punto seleccionado esté abierto.

4.1.3. Glosario de términos de Android

En este apartado se describen algunos términos de Android utilizados en el apartado anterior.

- **Activity:** es una componente de la aplicación que contiene una pantalla con la que un usuario puede interactuar. Cada actividad tiene una pantalla asociada a una interfaz de usuario. Normalmente la pantalla suele ocupar toda la pantalla.
- **Fragment:** representa un comportamiento o parte de la interfaz de usuario de una Activity. Dentro de una Activity se pueden combinar varios Fragments. Al igual que una Activity, tiene un ciclo de vida propio y recibe sus propios eventos de entrada. Podría definirse como una “subactividad”.
- **View:** representa el bloque básico para representar cualquier elemento básico de la interfaz de usuario en Android.
- **ViewGroup:** es un tipo especial de View que puede contener Views a su vez. Este tipo de View es utilizada como base de un Layout y otro contenedores complejos dentro de Android.

- **Layout:** define la estructura para el diseño de una interfaz de usuario, como la UI de una actividad o un widget de una aplicación.

4.2. Pruebas

Para los ejemplos que se muestran en este apartado se ha utilizado la ciudad de Granada; que ha sido la ciudad elegida para desarrollar el prototipo.

4.2.1. Ruta sobre museos en Granada

En el caso de que un usuario quiera visitar museos, lo primero que debe hacer es abrir la aplicación y seleccionar un alojamiento; para ello debe levantar o pulsar sobre la pestaña llamada “Sitios interesantes”. Una vez se ha seleccionado el alojamiento; el usuario deberá buscar en la lista el tipo llamado “Museos”, al lado de esta hay una casilla en la que deberá pulsar para que se seleccionen todos los museos. El resultado se puede ver en la figura 4.1.

Una vez ha seleccionado los museos deberá pulsar en el botón “Comenzar búsqueda”. Cuando se haya terminado la búsqueda de la ruta, se cargará otro mapa en el que se mostrarán los museos seleccionados en la ruta; si el usuario pulsa en alguno de los marcadores se mostrará el nombre y el horario de entrada y salida del museo aproximada. Si quiere ver la ruta entera en detalle, puede pulsar o deslizar hacia arriba la pestaña “Descripción ruta final”.

Si nos fijamos en la ruta seleccionada que se muestra en las figuras 4.2; se puede ver que el algoritmo selecciona el primer museo que se encuentra más cerca del alojamiento seleccionado, después va seleccionando el museo más cercano al que se ha visitado; cumpliendo con la heurística que se ha implementado.

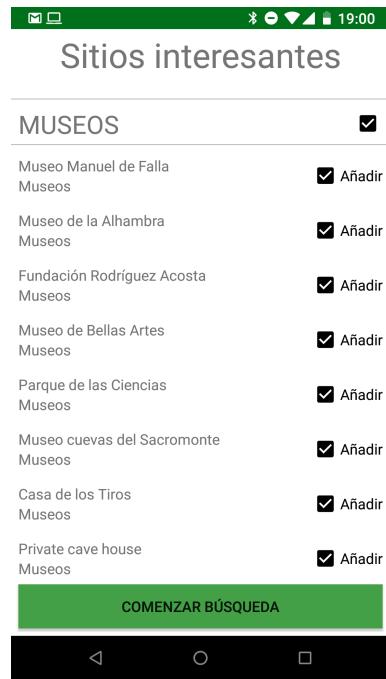


Figura 4.1: Selección de todos los museos disponibles dentro de la aplicación.

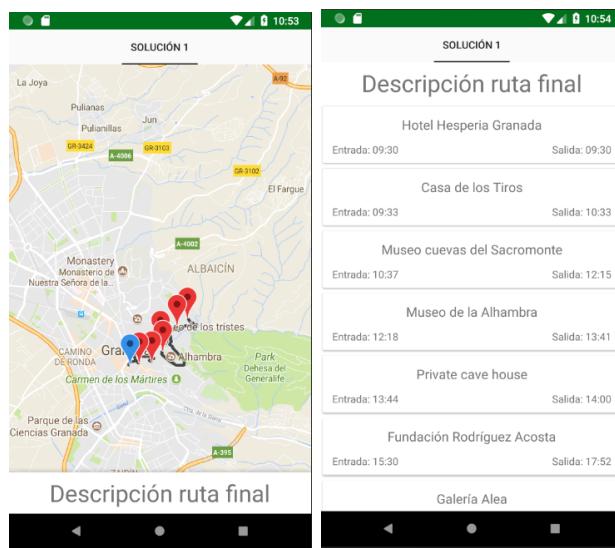


Figura 4.2: Ruta obtenida para el alojamiento seleccionado y todos los museos seleccionados.

4.2.2. Ruta sobre un conjunto específico de POIs

Si se da el caso de que el usuario quiera visitar unos POIs también puede hacerlo; para este ejemplo se van a seleccionar el mirador de San Nicolás, la Hermita de San Miguel Alto, la Catedral de Granada y el Parque de las Ciencias. Para seleccionar dichos puntos, el usuario debe buscarlos dentro de la lista; al igual que se muestran en las figuras 4.3.

Después, el usuario deberá iniciar la búsqueda para pulsar el botón “Comenzar búsqueda”. Tras esto, se mostrará el resultado, dicho resultado es el que se puede ver en las figuras 4.4. Al ser pocos POIs, la solución selecciona todos los POIs.

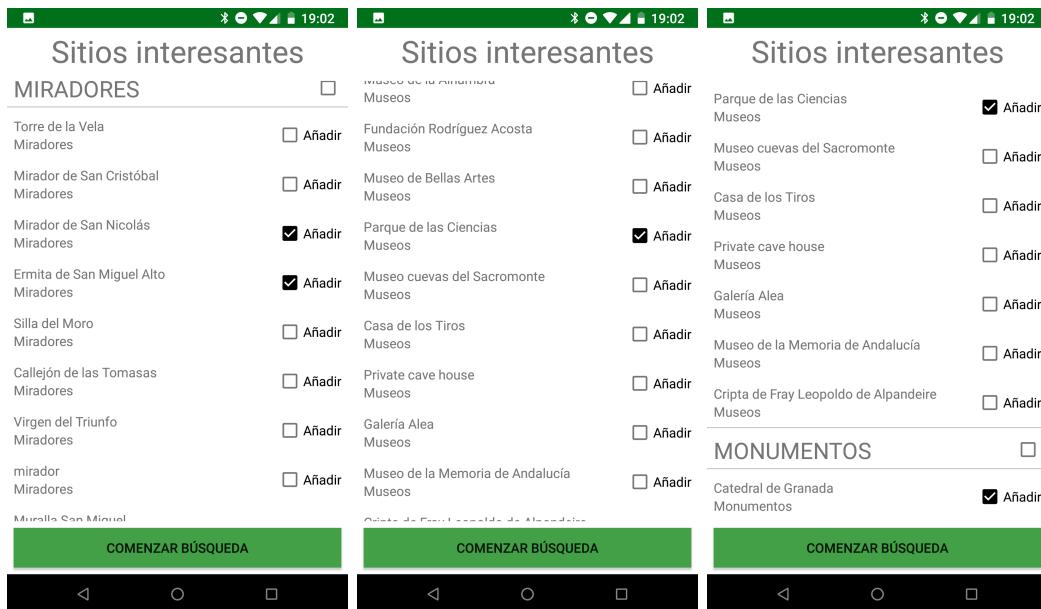


Figura 4.3: Selección de un subconjunto específico de POIs.

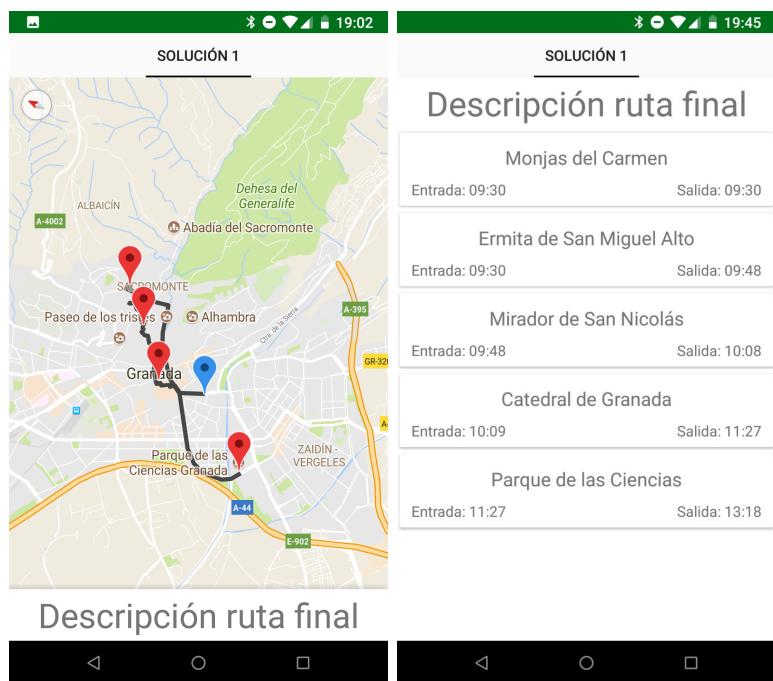


Figura 4.4: Ruta calculada para el conjunto de POIs.

Capítulo 5

Conclusiones

En este capítulo se describen las conclusiones posteriores a la realización del Trabajo de Fin de Grado, con el fin de revisar los objetivos que fueron establecidos en los distintos apartados durante su realización y para comprobar la satisfacción de cada uno de ellos.

También se introducen algunas líneas de trabajo futuro, en los que se podría extender o mejorar el funcionamiento de este trabajo.

5.1. Conclusiones

En el tercer capítulo de este Trabajo de Fin de Grado se establecieron una serie de objetivos, cuyos resultados y conclusiones sobre su cumplimiento son los siguientes:

- Se ha llevado a cabo una recopilación lo suficientemente completa sobre la historia de los sistemas recomendadores y destancando algunas herramientas recientes en este ámbito.
- Se ha elegido y definido un modelo matemático (TOPTW) y uno heurístico (Greedy) funcionales para la generación de itinerarios de rutas de puntos de interés, ajustados a las preferencias del usuario.
- Se ha empleado una muestra de datos reales y fiables de puntos de interés y alojamientos (*OpenStreetMap*) con los cuales se han obtenido resultados satisfactorios al probarlos en el sistema recomendador.
- Se ha conseguido crear una aplicación para móvil funcional e intuitiva la es capaz de generar rutas de puntos de interés y representarlas en un mapa interactivo.
- Se ha documentado el código desarrollado tanto en la parte gráfica de dicha aplicación como en los procesos internos de la misma. También se han documentado el resto de tareas realizadas durante el Trabajo de Fin de Grado. Con todo esto se ha formado esta memoria final.

Por tanto, se han logrado satisfacer todos los objetivos propuestos, se puede concluir de que este Trabajo de Fin Grado se ha terminado con éxito.

En su realización fue necesario estudiar y aprender a usar *OpenStreetMap*, *Overpass*, *OSRM*, *Google Maps API* y *Google Directions API*. Igualmente, se profundizó en otras ya conocidas como Java y el sistema operativo Android.

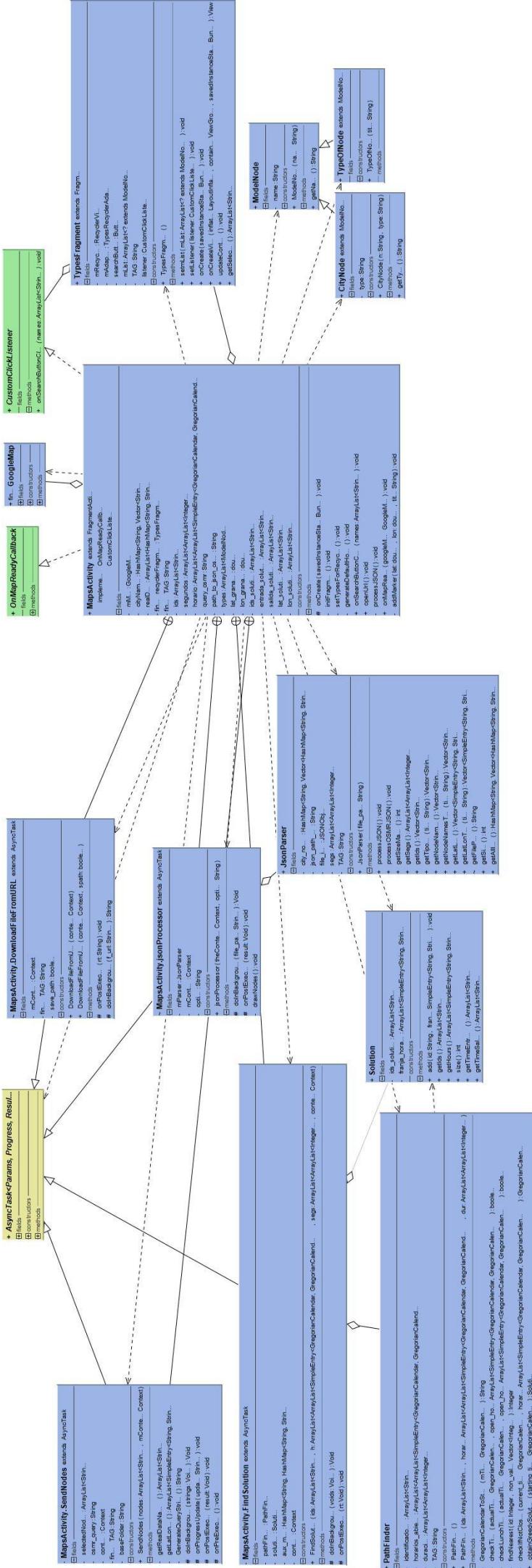
En este Trabajo de Fin de Grado se ha podido apreciar el potencial que tienen las aplicaciones turísticas y el gran impacto positivo que pueden suponer para los turistas y los sectores relacionados con el turismo. Además, construir dicha aplicación fue una tarea muy enriquecedora e interesante para el autor; habiéndole dado experiencia y conocimientos muy valiosos.

5.2. Líneas de trabajo futuro

Para finalizar, en esta sección se muestran una lista de tareas que podrían ser líneas de trabajo futuro para este sistema desarrollado durante el Trabajo de Fin de Grado. Estas consisten en nuevas características y funcionalidades que podrían estudiarse e implementarse para mejorar este producto. Dichas líneas de trabajo son:

- **Integración con transportes urbanos de las ciudades:** para mejorar aún más la experiencia del usuario se podrían indicar medios de transportes urbanos, como autobuses o metro, que el usuario pueda utilizar si quiere para llegar a diferentes puntos de interés de la ruta.
- **Mejorar el algoritmo de búsqueda de rutas:** se podría mejorar las soluciones mostradas por la aplicación utilizando heurísticas mejores que la utilizada en este proyecto, por ejemplo una heurística GRASP. Debido a la falta de tiempo, se decidió no implementarla.
- **Añadir más idiomas:** al estar la aplicación dirigida al turismo, y como gran parte de los turistas que visitan un país son extranjeros, sería oportuno añadir soporte para más idiomas en la aplicación, como por ejemplo Inglés o Chino.
- **Extender las ciudades a visitar:** se podría meter soporte para más ciudades, ya que la aplicación en este momento es un prototipo funcional en la ciudad de Granada. Debido al poco tiempo restante, se decidió no implementar esta característica.

Anexo



Bibliografía

- [1] UNWTO. Panoráma OMT del turismo internacional. *UNWTO*, 2017.
- [2] Vansteenwegen P and Van Oudheusden D. The mobile tourist guide: an OR opportunity. *OR Insight*, 20:21–17, 2007.
- [3] Tsiligirides T. Heuristic methods applied to orienteering. *J Oper Res Soc*, 35:797–809, 1984.
- [4] Chao IM, Golden BL, and Wasil EA. The team orienteering problem. *Eur J Oper Res*, 88:464–474, 1996.
- [5] Vansteenwegen P, Souffriau W, Vanden Berghe G, and Van Oudheusden D. Iterated local search for the team orienteering problem with time windows. *Comput Oper Res*, 36:3281–3290, 2009.
- [6] Fomin FV and Lingas A. Approximation algorithms for time-dependent orienteering. *Inform Process Lett*, 83:57–62, 2002.
- [7] Garcia A, Arbelaitz O, Linaza MT, Vansteenwegen P, and Souffriau W. Personalized tourist route generation. *Proceedings of the 10th international conference on Current trends in web engineering*, page 486–497, 2010.
- [8] Sylejmani K, Dorn J, and Musliu N. A tabu search approach for multi constrained team orienteering problem and its application in touristic trip planning. *12th international conference on hybrid intelligent systems (HIS)*, page 300–305, 2012.
- [9] Murat Afsar H and Labadie N. Team orienteering problem with decreasing profits. *Electron Notes Discrete Math*, 41:285–293, 2013.
- [10] Angelelli E, Archetti C, and Vindigni M. The clustered orienteering problem. *Eur J Oper Res*, 238:404–414, 2014.
- [11] Trello. Página oficial de Trello. <https://trello.com/>.
- [12] Github. Página oficial de GitHub. <https://github.com/>.
- [13] OpenStreetMap. OpenStreetMap. <https://www.openstreetmap.org/#map=6/40.007/-2.488>.

- [14] OpenStreetMap. Documentación de OpenStreetMap. <https://wiki.openstreetmap.org/wiki/Tags>.
- [15] OSRM. OSMR Project. <http://project-osrm.org/>.
- [16] OSRM. Github OSRM. <https://github.com/Project-OSRM/osrm-backend>.
- [17] Google. Google Maps Directions API. <https://developers.google.com/maps/documentation/directions/intro?hl=es-419>.
- [18] Google. Android Oreo. <https://developer.android.com/about/versions/oreo/android-8.0>.
- [19] Google. Tutorial para decodificar polilínea Google Directions API. <https://developers.google.com/maps/documentation/utilities/polylinealgorithm?hl=es-419>.
- [20] Overpass. Overpass Turbo. <https://overpass-turbo.eu/>.

