UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Optimización coste/duración en viajes por autopista de peaje

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Bataller Herrero, Carles

Tutor: Jordan Lluch, Cristina
Morillas Gómez, Samuel

Curso 2019-2020

Resum

La finalitat d'aquest projecte és el desenvolupament e implementació d'una eina capaç de calcular itineraris per a vehicles d'ús particular entre diverses poblacions. Aquesta eina proporcionarà el major nombre de rutes alternatives possibles, especificant el temps aproximat que ens portarà completar cada ruta i el cost econòmic dels peatges d'aquestes alternatives. Les rutes obtingudes no hauran de ser directament comparables entre si, en termes de costos en peatges i duració, proporcionant d'aquesta forma major capacitat de decisió i un major control del cost de les rutes triades pels usuaris.

Per al càlcul de la rutes s'ha fet ús de l'algorisme de cerca heurística en grafs ponderats A*, que s'ha modificat perquè complisca amb les necessitats concretes d'aquest projecte.

A més s'ha obtingut una base de dades amb dades reals que ens ha permés provar el funcionament del nostre producte en un entorn real.

El sistema s'ha implementat en el *framework* Ruby on Rails, també s'ha fet ús dels llenguatges Coffescript per a la implementació de la interfície i de la llibreria PostGIS per a la base de dades.

Paraules clau: Cerca heurística, autopista de peatge, itinerari, RoR, Ruby

Resumen

La finalidad de este proyecto es el desarrollo e implementación de una herramienta capaz de calcular itinerarios para vehículos de uso particular entre varias poblaciones. Esta herramienta proporcionará el mayor número de rutas alternativas posibles, especificando el tiempo aproximado que nos llevará completar cada ruta y el coste económico de los peajes de estas alternativas. Las rutas obtenidas no deberán de ser directamente comparables entre sí, en términos de costes en peajes y duración, proporcionando de esta forma mayor capacidad de decisión y un mayor control del coste de las rutas elegidas por los usuarios.

Para el cálculo de la rutas se ha hecho uso del algoritmo de búsqueda heurística en grafos ponderados A*, que se ha modificado para que cumpla con las necesidades concretas de este proyecto.

Además se ha obtenido una base de datos con datos reales que nos ha permitido probar el funcionamiento de nuestro producto en un entorno real.

El sistema se ha implementado en el *framework* Ruby on Rails, también se ha hecho uso de los lenguajes Coffescript para la implementación de la interfaz y de la librería PostGIS para la base de datos.

Palabras clave: Búsqueda heurística, autopista de peaje, itinerario, RoR, Ruby

Abstract

The purpose of this project is the development and implementation of a tool capable of calculating itineraries for private vehicles between various populations. This tool will provide the greatest number of alternative routes possible, specifying the approximate time it will take us to complete each route and the economic cost of tolls for these alternatives. The routes obtained will not have to be directly comparable between them, in terms of toll costs and duration, thus providing a greater decision capacity and a greater control of the cost of the routes chosen by the users.

For the calculation of the routes, A* the heuristic search algorithm in weighted graphs has been used, which has been modified to meet the specific needs of this project.

In addition, a database with real data has been obtained which has allowed us to test the operation of our product in a real environment.

The system has been implemented in the Ruby on Rails framework, also the Coffe-script languages have been used for the implementation of the interface and the PostGIS library for the database.

Key words: heuristic search algorithm, toll road, itineraries, RoR, Ruby

Índice general

Índice de figuras

Índice de tablas

Siglas

RF (Requisitos funcionales)
RNF(Requisitos no funcionales)
MVC (Modelo Vista Controlador)
SGBDR (Sistema gestor de bases de datos relacional)
RoR (Ruby on Rails)
HTML (HyperText Markup Language)
CSS (Cascading Style Sheets)
AJAX (Asynchronous JavaScript And XML)
MVC (Modelo-Vista-Controlador)
OSM (Open Street Maps)
SMA* (Simplified Memory Bounded A*)

CAPÍTULO 1
Introducción

En este capítulo vamos a hablar de la motivación y los objetivos de este proyecto.

1.1 Motivación

El motivo que nos impulsó a realizar este proyecto fue que detectamos que la mayoría de buscadores más conocidos de rutas para automóviles entre distintas ubicaciones nos sugerían rutas demasiado genéricas, proporcionándonos muy pocas alternativas a la hora de elegir que ruta queríamos seguir. Esta falta de opciones llega al punto en que algunos de estos buscadores nos permitían elegir entre solo dos opciones, la ruta más corta o la más rápida.

La carencia de variedad se acentúa aún más a la hora de elegir rutas que transcurran en un área de peaje, ya que en la mayoría de casos esta falta de opciones obliga al usuario a elegir entre una opción más rápida con un alto coste en peajes o una opción más corta por carreteras menos prioritarias evitando así todos los peajes. Como se puede ver, se ha descartado completamente la opción de combinar ambas rutas, la más cara y la más económica, para crear una opción más personalizada que se pueda adaptar mejor a las necesidades de un usuario con un presupuesto más humilde, opción que consideramos que sería interesante considerar.

1.2 Objetivos

El objetivo de este proyecto es el desarrollo e implementación de una herramienta de búsqueda de rutas para automóvil que a partir de la Teoría de Grafos sea capaz de mostrarnos una gran variedad de rutas posibles entre dos poblaciones. Además, esta herramienta debe de estar preparada para su despliegue en un servidor en producción. Gracias a esta herramienta el usuario debe de ser capaz de elegir entre gran variedad de rutas válidas, permitiendo al usuario elegir entre múltiples opciones económicas, conociendo exactamente la distancia y el tiempo de viaje de cada ruta. Esta herramienta deberá de ser capaz de calcular rutas en un entorno real y mostrarnos los resultados mediante una interfaz lo más amigable posible.

La interfaz de la herramienta desarrollada deberá de ser compatible con las resoluciones de pantalla de todo tipo de dispositivos, desde dispositivos móviles a ordenadores de sobremesa. Además, será conveniente que la interfaz disponga de un mapa en el que se puedan observar las rutas calculadas.

Es imprescindible que las rutas calculadas ofrezcan la mayor variedad en los precios de peaje posible, mostrando una selección representativa del conjunto de soluciones existente. Idealmente las rutas propuestas no serán comparables entre si, o al menos no directamente. Un ejemplo de esto podría ser un caso en el que el usuario busque las rutas entre dos ciudades, priorizando las rutas más rápidas, y se encuentren como resultado dos rutas diferentes. Si ambas rutas tienen el mismo coste de peaje se mostrará la ruta más rápida, al contrario si la ruta más lenta tiene un coste de peaje más reducido se mostrarán ambas rutas. De esta forma se pretende dar una mayor libertad al usuario para elegir que ruta prefiere tomar y permitirle elegir exactamente que importe desea gastarse en peajes, gracias al elevado número de opciones que se le muestran, mientras se le informa de cuanto tiempo le va a tomar realizar el viaje.

Al no poder descartar la mayoría de posibles rutas mientras se realiza una búsqueda, y por lo tanto tener que comparar tantas posibles soluciones, los tiempos de ejecución de nuestras búsquedas será más elevado que el de un buscador de rutas convencional. Esto nos lleva a recalcar que el objetivo de este proyecto no es rivalizar con el resto de buscadores de rutas en cuanto a rendimiento, lo cual sería prácticamente imposible debido a lo limitados que son los recursos de los que disponemos, sino a ofrecer una mayor variedad de soluciones al usuario, manteniendo un tiempo de cálculo lo más reducido posible.

1.3 Estructura de memoria

En este apartado proporcionaremos una guía sobre los aspectos tratados en cada uno de los capítulos que componen esta memoria.

- **Capítulo 1 Introducción:** En este capítulo se explicarán los motivos que nos han llevado a realizar este proyecto, además también se listarán los objetivos que se pretenden conseguir durante el desarrollo del mismo.
- **Capítulo 2 Estado del arte:** A lo largo de este apartado se explicará el contexto de nuestro proyecto, tanto desde un punto de vista teórico, explicando su vinculación con la teoría de grafos, como funcional, comparando las funcionalidades de los productos que comparten similitudes con el producto a desarrollar.
- **Capítulo 3 Análisis y requisitos:** En este apartado se detallarán las especificaciones del sistema, además se listarán los requisitos identificados, tanto funcionales como no funcionales, y se explicarán los diagramas de clases y de casos de uso de nuestra aplicación.
- **Capítulo 4 Tecnologías y herramientas:** En este apartado se explican los posibles lenguajes y herramientas de programación a utilizar y cuáles han sido elegidas para desarrollar nuestro proyecto.
- **Capítulo 5 Diseño de la aplicación:** En este apartado se explicará la arquitectura de nuestro sistema, junto a la estructura creada de nuestra base de datos y también se explicarán los bocetos iniciales de nuestra aplicación.
- **Capítulo 6 Implementación:** En este capítulo se resumirá la implementación de nuestra aplicación, explicando los pasos realizados para poblar nuestra base de datos y por último se remarcarán los aspectos más destacables del desarrollo de nuestro proyecto.
- **Capítulo 7 Despliegue:** En este apartado se resume el proceso realizado para preparar nuestra aplicación para el despliegue de la misma en un servidor de producción.

- **Capítulo 8 Pruebas:** En este capítulo se identifican las herramientas utilizadas para realizar las pruebas de nuestra aplicación y se explica que tipo de pruebas se han realizado y cómo se han desarrollado.
- **Capítulo 9 Conclusiones:** En este capítulo se analiza el producto obtenido y se compara con los objetivos que se identificaron en la introducción de esta memoria.
- **Capítulo 10 Trabajo futuro:** Para finalizar, en este apartado se listarán las posibles mejoras que se han considerado interesantes y se explicará en que mejorarían estas funcionalidades la aplicación actual.

CAPÍTULO 2
Estado del arte

2.1 Teoría de grafos

La teoría de grafos es una rama de las matemáticas y las ciencias de la computación que estudia las propiedades de los grafos.

Un grafo es un conjunto de elementos denominados nodos los cuales pueden estar relacionados entre ellos, estas relaciones se denominan aristas. Una de las características interesantes de los grafos es su utilidad a la hora de representar gráficamente relaciones binarias entre elementos de un conjunto. De esta forma, los grafos nos permiten la representación y estudio de problemas del mundo real, y nos facilitan la resolución de estos problemas mediante un enfoque matemático.

En nuestra aplicación se ha representado la red de carreteras del estado español en un grafo donde las diferentes carreteras están representadas por aristas y los nodos son los diferentes puntos que las unen. Emulando los diferentes elementos que componen un grafo mediante una estructura lógica que nos facilita el trabajo con dichos elementos y nos permite encontrar una solución al problema propuesto.

Se pueden diferenciar diferentes tipos de grafos según sus propiedades, algunos de los más relevantes por su aplicación en nuestro proyecto son los grafos ponderados y los grafos dirigidos.

Los grafos ponderados son aquellos cuyas aristas tienen un peso asociado, este peso suele representar el coste de transcurrir entre los nodos relacionados por dicha arista. En nuestra aplicación esta característica nos ha servido de ayuda para representar los diferentes costes asociados a un tramo, como la distancia de éste o el coste de su peaje.

Los grafos dirigidos son aquellos que tienen asociada una dirección a sus aristas, por ejemplo permitiéndonos desplazarnos del nodo «A» al «B» pero no del «B» a «A». Este tipo de grafos nos ha servido de ayuda a la hora de operar con carreteras de un único sentido.

2.2 Algoritmos de búsqueda

Los algoritmos de búsqueda definen las estrategias necesarias para encontrar un elemento dentro de una estructura de datos. En nuestra aplicación el algoritmo usado será el encargado de intentar encontrar el destino de nuestra ruta partiendo desde el origen de ésta, navegando dentro del grafo formado por las carreteras del territorio español.

Existen diferentes algoritmos de búsqueda, cada uno tiene sus mecanismos y particularidades propias cuya eficiencia depende de la estructura de datos sobre la que operen y

la distribución de los datos dentro de esta estructura. Debido a esta característica la elección del algoritmo de búsqueda utilizado en cada caso influirá directamente en el tiempo necesario que requerirá terminar dicha búsqueda.

En nuestra aplicación se ha utilizado un algoritmo de búsqueda heurística, el cual se basa en una fórmula matemática para intentar estimar que camino tiene más probabilidades de ser el camino mínimo, aquel cuya suma de los pesos de sus aristas sea la menor posible. La elección del algoritmo de búsqueda se discutirá en mayor profundidad en el apartado 6.2.1.

2.3 Estudio de mercado

Cuando se desarrolla una aplicación es fundamental hacer un análisis del entorno en el que ésta se ubicará, permitiéndonos la identificación de un posible nicho de mercado y facilitando que el producto final obtenido sea lo más óptimo posible, y a ser posible que satisfaga una necesidad real que ningún otro producto de la misma índole sea capaz de satisfacer.

2.3.1. Las cinco fuerzas de Porter

- **Intensidad competitiva - alta** En el sector de la búsqueda de rutas automovilísticas la intensidad competitiva es alta, debido a que ya existen abundantes competidores que se han consolidado a lo largo de los años y que disponen muchos más recursos de los que dispondrá nuestra aplicación.

- **Amenaza de nuevos competidores - media** Aunque si que existe la amenaza de nuevos competidores, se considera que es poco probable que surja un nuevo producto que aporte una nueva funcionalidad propia, ya que la mayoría de ellos hará uso de los recursos aportados por aplicaciones ya disponibles intentando darles un enfoque diferente.

- **Poder del proveedor - medio**

  En este caso se considera al proveedor a aquella entidad que nos suministra la información geográfica necesaria para realizar nuestros cálculos.

  Aunque es cierto la mayor fuente de datos geográficos pertenece a una aplicación competidora como es Google Maps [1] y que el coste económico de crear nuestra propia fuente de datos sería desmesurado, también es cierto que existen diversas fuentes de datos gratuitas y de uso libre como Open Street Maps [24], por lo que se ha considerado que el poder del proveedor en este caso es medio.

- **Poder del comprador - bajo**

  Los usuarios de la aplicación tienen muy poco poder sobre la aplicación, ya que ésta será gratuita para todos los usuarios.

- **Amenaza de los sustitutos - alto**

  Ésta se ha considerado la amenaza más preocupante, ya que debido a la dominancia de las aplicaciones competidoras en el mercado es muy difícil que surjan aplicaciones nuevas que implementen funciones similares.

2.3.2. Análisis de la competencia

Nuestra aplicación web compite con un gran número de aplicaciones que se han ido consolidando a lo largo de los años. Es por esto que nos resulta de gran ayuda comparar las características de estas aplicaciones para así intentar obtener un producto de la mayor calidad posible.

- **Google Maps** Esta aplicación web desarrollada por Google, es posiblemente la plataforma más utilizada entre las aplicaciones de esta índole. Destaca por la rapidez de su servicio y por la gran cantidad de información que nos proporciona. También es destacable la existencia de una aplicación móvil que facilita el uso de esta solución en este tipo de dispositivos.

Google Maps	
Fortalezas	**Debilidades**
Rapidez de respuesta	Poca personalización de las rutas obtenidas
Gran cantidad de información adicional	
Opción de alternar entre vista de satélite y mapa	
Permite realizar búsquedas entre direcciones	
Multiplataforma	

Tabla 2.1: Fortalezas y debilidades de Google Maps

- **Vía Michelin** El buscador de rutas de la Vía Michelin [2] es una aplicación web cuyas características más destacables son la velocidad con la que calcula las rutas, la opción de calcular el coste en combustible necesario para realizar la ruta y la integración de un buscador de hoteles en las ciudades de origen y destino.

Vía Michelin	
Fortalezas	**Debilidades**
Rapidez de respuesta	Poca personalización de las rutas obtenidas
Calculador de coste de combustible	
Buscador de hoteles	
Permite realizar búsquedas entre direcciones	

Tabla 2.2: Fortalezas y debilidades de Vía Michelin

- **DGT** La aplicación móvil de la DGT [3] también cuenta con un buscador de rutas, una de las características más destacables de esta aplicación podría ser la incorporación de un buscador de rutas seguras, en el que te propone rutas alternativas que eviten tramos con mayor índices de accidentes. Además esta aplicación también nos informa en tiempo real de las rutas en los que se ha sufrido de un accidente, también nos informa de la localización de los radares fijos que se encuentran en las rutas elegidas. El mayor inconveniente de la utilización de esta aplicación es la imposición del sistema de firma digital Cl@ve [4] para la utilización de esta aplicación, dificultando el uso de esta.

2.3.3. Nuestra aplicación frente a la competencia

Tras analizar el mercado y algunas de las aplicaciones con las que competiremos, podemos determinar cual será el lugar que ocupará nuestra aplicación en el mercado.

DGT	
Fortalezas	**Debilidades**
Buscador de radares fijos	Imposición del sistema Cl@ve
Buscador de rutas seguras	Plataforma única
Información de accidentes en tiempo real	Uso limitado a España

Tabla 2.3: Fortalezas y debilidades del buscador de la aplicación móvil de la DGT

Al comparar nuestra aplicación con el resto de aplicaciones competidoras podemos afirmar que ninguna implementa la funcionalidad principal en la cual se centra nuestra aplicación, permitir al usuario un mayor margen de decisión sobre las rutas que va a seguir, proporcionándole una solución que se adapte mejor a sus necesidades concretas y por lo tanto proporcionando un un trato más personalizado.

Por contra, también se ha podido observar que nuestra aplicación estará en clara desventaja en cuanto a velocidad en la obtención de respuestas. Y aunque es posible expandir el ámbito en el que opera nuestra aplicación, en este caso el territorio español, también ofrecerá menores prestaciones en comparación a las aplicaciones capaces de buscar rutas entre varios países.

CAPÍTULO 3
Análisis y requisitos

En este capítulo vamos a describir el proyecto a desarrollar, junto a los requisitos tanto funcionales como no funcionales identificados.

3.1 Especificación del sistema

El ámbito de este proyecto es la búsqueda de rutas entre diferentes poblaciones.

Existen múltiples buscadores de rutas, muchos de ellos con un rendimiento excelente y capaces de de buscar rutas por todo el planeta, por lo que este proyecto no pretende competir en cuanto a rendimiento con los productos ya existentes, sino crear una solución que permita al usuario participar más activamente en el proceso de elección de la ruta.

Nuestra aplicación intenta ofrecer el mayor número de opciones posibles, intentando que las soluciones obtenidas cubran el mayor rango de costes de peaje posible, y permitiendo que el usuario pueda elegir la solución que se adapte mejor sus necesidades concretas. De esta forma se fomentará que el usuario conozca toda la información necesaria sobre cada ruta y pueda elegir una alternativa que cumpla sus requerimientos y se ajuste a su presupuesto.

3.2 Requisitos

A continuación se van a describir requisitos de la aplicación, tanto los funcionales como los no funcionales, necesarios para describir el comportamiento de nuestra aplicación y definir el alcance de ésta.

3.2.1. Requisitos funcionales (RF)

- RF1. El resultado de la búsqueda debe de ser una lista de rutas no comparables directamente entre ellas, en términos de coste y duración, por ser cada una mejor según un criterio y peor según otro criterio respecto de las demás.

- RF2. Al encontrarse una nueva ruta se comprobará que no existe ninguna ruta en la lista de resultados con el menor o igual coste económico pero con menores prestaciones (mayor distancia o tiempo de viaje, según las características de la búsqueda), en caso de que exista alguna ruta que cumpla estas condiciones se eliminará de la lista de resultados.

- RF3. Al encontrarse una nueva ruta se comprobará que no existe ninguna ruta en la lista de resultados con el menor o igual coste económico y mayores prestaciones (menor distancia o tiempo de viaje, según las características de la búsqueda), en caso de que exista alguna ruta que cumpla estas condiciones se no se añadirá la nueva ruta a la lista de resultados.
- RF4. Se mostrarán diferentes características relevantes de los resultados obtenidos como velocidad media del trayecto, distancia o tiempo estimado de viaje.
- RF5. Se mostrarán los resultados calculados conforme éstos se vayan encontrando, sin necesidad de que la búsqueda haya concluido.
- RF6. Se podrá visualizar información avanzada de cada resultado, mostrando los diferentes tramos que lo conforman junto a información de cada tramo como velocidad máxima permitida, longitud, etc.
- RF7. Se podrán elegir los puntos de origen y destino de la ruta entre una lista de poblaciones.
- RF8. El usuario podrá detener la búsqueda en caso de haber obtenido ya una ruta que cumpla sus necesidades.
- RF9. En caso de que el usuario detenga una búsqueda se seguirán mostrando los resultados obtenidos hasta que este inicie una nueva búsqueda.
- RF10. Se podrán visualizar las rutas obtenidas en un minimapa.

3.2.2. Requisitos no funcionales (RNF)

- RNF1. La aplicación web deberá estar soportada por los navegadores web más populares, como Google Chrome, Mozilla Firefox o Safari.

3.2.3. Restricciones de la aplicación

- El servidor en el que se despliegue la aplicación deberá de tener más de 3GB de memoria libre para poder almacenar la base de datos de la aplicación.
- Los puntos de origen y de destino de las rutas deberán de ser poblaciones existentes en la base de datos de la aplicación.
- El número de peticiones diferentes que se podrán atender concurrentemente dependerá directamente de las capacidades del servidor.
- Se delimitará un tiempo máximo de búsqueda de 30 minutos, cancelando todas las búsquedas de rutas que lo superen y mostrando los resultados obtenidos al usuario, para evitar el acopio de recursos y garantizar el servicio al mayor número de usuarios posibles.

3.3 Casos de uso

Realizar un diagrama de casos de uso nos resulta de gran utilidad a la hora de iniciar un nuevo proyecto, ya que nos ayuda a visualizar las tareas que será capaz de realizar cada usuario, y por lo tanto las funcionalidades que se deben de desarrollar.

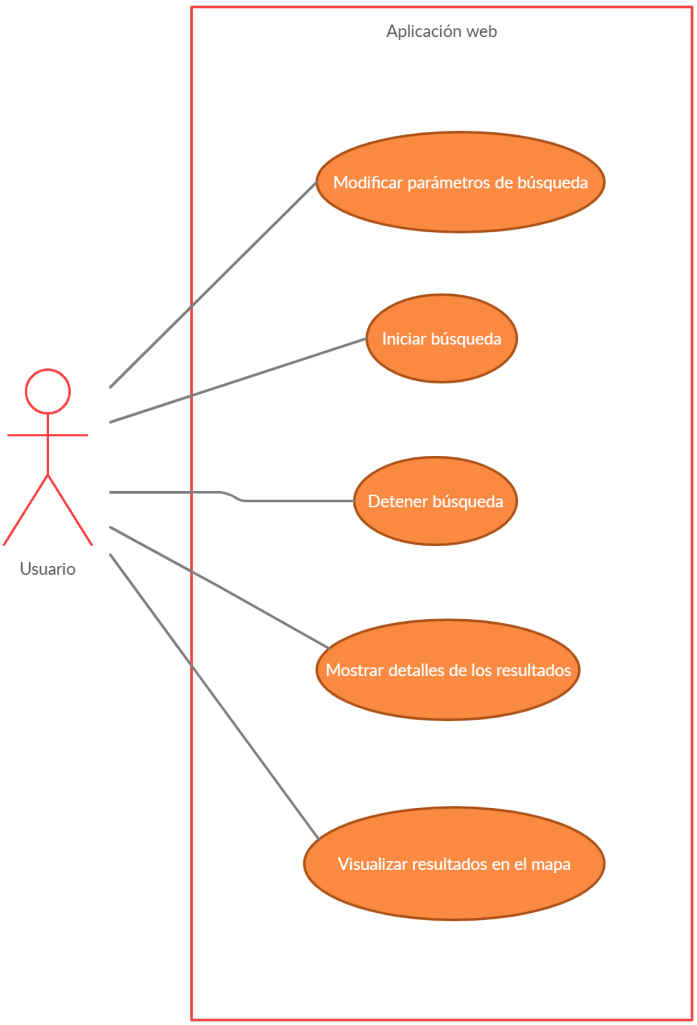

Figura 3.1: Diagrama de casos de uso.

Aunque en nuestro caso el diagrama de casos de uso puede resultar algo simple, debido a que nuestra aplicación no tiene diferentes tipos de usuarios y a que la complejidad de nuestra aplicación no reside en la cantidad de acciones que el usuario puede realizar sino en la complejidad computacional de las acciones realizadas, la utilidad de este diagrama no se ha reducido, ya que nos ha servido para entender mejor el alcance de nuestro proyecto y a no perder el foco en las funcionalidades que se tenían que desarrollar.

3.4 Diagrama de clases

Los diagramas de clases nos permiten representar de forma visual la estructura que forman nuestros Modelos, facilitando la comprensión del funcionamiento de nuestra capa de negocio.

El diagrama de clases suele tener parecidos con la estructura de tablas de nuestra base de datos, ya que será necesario persistir muchos de los recursos que se utilicen en la capa de Negocio, por lo que la realización de este diagrama también nos será de ayuda a la hora de diseñar la estructura de nuestra base de datos.

En la figura 3.2 se puede observar el diagrama de clases realizado, la funcionalidad asociada a las diferentes clases identificadas se explica a continuación:

- **Way** Esta clase es la representación de los tramos que forman una ruta. La clase «Way» está relacionada con la clase «Line», ya que gracias a la información que obtengamos de esta clase seremos capaces de representar visualmente los elementos de tipo «Way». Además, también está relacionada con la clase «Node», representando los objetos de esta clase a los puntos que unen dos elementos del tipo «Way», como por ejemplo un cruce.

  Las funcionalidades que nos aporta esta clase son bastante simples, ya que solamente dispone de métodos de tipo «get» que nos permitirán conocer dentro del marco de otras clases información diversa de objetos de tipo «Way», como la longitud de un tramo o la velocidad máxima permitida en éste.

- **Line** Esta clase se encarga de gestionar la información necesaria para representar objetos de la clase «Way» en un mapa, por lo que su funcionalidad más importante radica en convertir la información almacenada en la base de datos en objetos GeoJSON [19] que puedan ser posteriormente transmitidos a la interfaz y ser representados.

- **Toll** La funcionalidad de esta clase radica en facilitarnos el acceso a los datos de los peajes que almacenaremos en la base de datos, haciendo las conversiones necesarias para permitirnos estimar el coste de un trayecto.

- **Node** Esta clase es la representación de los diferentes puntos que dividen las diferentes vías de transporte en tramos más pequeños, estos elementos pueden ser desde intersecciones, a cambios de dirección e incluso puntos que representen los centros de las poblaciones. La funcionalidades de esta clase están marcadas por la información que se precise en nuestro algoritmo de búsqueda de estos elementos, siendo la información más destacable la distancia de un «Nodo» a otro.

- **Point** Al igual que la clase «Line», la funcionalidad de esta clase es realizar las transformaciones necesarias en la información de nuestra base de datos para permitirnos representar un elemento concreto, que en este caso se trata de los elementos del tipo «Node».

- **Node-Way** El propósito de esta tabla es facilitar la implementación de la relación *many-to-many* entre las clases «Node» y «Way».

- **Path** La clase «Path» nos permite la creación de estructuras de datos en las que almacenar las rutas conforme las vayamos expandiendo, estas estructuras formarán el esquema («Node»,«Way»,«Node»,«Way»,...) y nos aportarán una gran versatilidad a la hora de almacenar las rutas auxiliares con las que trabajemos. Cada vez que añadamos un nuevo elemento a un objeto de la clase «Path», esta clase comprueba que la estructura resultante empieza y acaba con un elemento de tipo «Node», que sigue la estructura anteriormente mencionada alternando elementos de tipo «Node» y «Way» y por último que los elementos consecutivos estén relacionados, o lo que es lo mismo que sean geográficamente adyacentes. Estas comprobaciones evitan que se creen accidentalmente rutas que contienen errores, aportando un mayor grado de fiabilidad a nuestra aplicación.

  Además, la clase «Path» también nos aporta métodos para obtener información de las rutas que creemos, ya que nos permite conocer la longitud de estas rutas, su velocidad media, el tiempo que nos llevaría realizar esta ruta, etc.

- **Results** Esta clase implementa una colección de «Paths» permitiéndonos almacenar los resultados de una búsqueda conforme éstos se vayan calculando e interactuar con ellos cuando sea necesario.

- **Score** La implementación de esta clase es necesaria para facilitar la implementación de un algoritmo de búsqueda heurístico, el funcionamiento de estos algoritmos se explica en mayor detalle en el apartado 6.2. Cada vez que se llegue a un nodo se guardará en la clase «Score» la puntuación de las mejores rutas por las cuales se ha llegado a dicho nodo, de esta forma se permite guardar varios caminos en caso de que una opción no sea claramente mejor que la otra. Además, la clase «Score» también guardará una referencia a los caminos que han obtenido las puntuaciones que almacena, por lo que al encontrar un camino que si que sea claramente mejor que los ya almacenadas será posible eliminar las puntuaciones anteriores y los «Paths» asociadas a estas puntuaciones, reduciendo así el uso de memoria de nuestra aplicación.
- **Search** En esta clase se puede encontrar la implementación de nuestro algoritmo de búsqueda. Además esta clase también implementa diferentes funcionalidades que nos permiten conocer el estado en el que se encuentra la búsqueda y recuperar los resultados de ésta.

3.5 Análisis de seguridad

La implantación de unos requisitos mínimos de seguridad es un requerimiento imprescindible en nuestra aplicación, ya que un posible ciberataque no solo podría infectar los servidores de nuestra aplicación, sino que podría comprometer todos los equipos de nuestros usuarios.

A continuación se listan algunas de las medidas preventivas que se han tomado o que se deberían de tomar una vez se lleve la aplicación a producción:

- **Evitar malas praxis:** Aunque siempre existe el factor humano y por lo tanto siempre se pueden haber cometido errores, sobretodo durante la fase de programación de la aplicación, existe un conjunto de malas prácticas que se deben de evitar, y que el seguimiento de la documentación y la investigación sobre las tecnologías utilizadas pueden ayudar a prevenir. En el apartado 6.3 se detalla una de estas malas prácticas que en caso de no haberse corregido hubiera significado una importante brecha de seguridad en nuestra aplicación.
- **Copia de seguridad de nuestra base de datos:** Es importante en la mayoría de aplicaciones que se realice periódicamente una copia de seguridad de la base de datos, permitiéndonos recuperar todas las interacciones que se han realizado con nuestra aplicación en caso de que nuestra base de datos principal se haya visto comprometida. Al no modificar las interacciones del usuario nuestra base de datos, ya que en nuestra aplicación los usuarios simplemente consultan la información de la base de datos sin crear en ningún momento entradas nuevas en ésta, no será necesario realizar copias de la base de datos frecuentemente. Pero si que será necesaria mantener una copia de seguridad cada vez que modifiquemos nuestra base de datos, ya sea para incluir una mejora como para corregir un error. Preferiblemente estas copias de la base de datos se realizarán en un servidor externo ajeno al servidor de producción.
- **Servidor de base de datos independiente:** Se recomienda que el servidor de nuestra base de datos no esté ubicado en el mismo sistema que el servidor en el que se encuentra la parte de funcionalidad de nuestra aplicación, de esta forma se reduce el riesgo de que ambos servidores fallen al mismo tiempo, reduciendo la complejidad y el tiempo necesario para realizar las reparaciones necesarias.

- **Control de versiones:** Es importante mantener un control de las versiones de la aplicación que despleguemos, este control de versiones se puede realizar mediante un repositorio GIT [32], de esta forma cuando se detecte un defecto introducido accidentalmente, nos será más fácil detectar donde se ha introducido el defecto y repararlo, e incluso podremos utilizar una versión anterior en el servidor de producción mientras se subsana dicho defecto.
- **Copias de seguridad del servidor de producción:** Aunque mantengamos un control de versiones de nuestra aplicación siempre es interesante realizar copias de seguridad de nuestro servidor de producción, de esta forma se reducirá el tiempo necesario para restaurar dicho servidor y se reducirán las configuraciones que tendremos que realizar en caso de error.

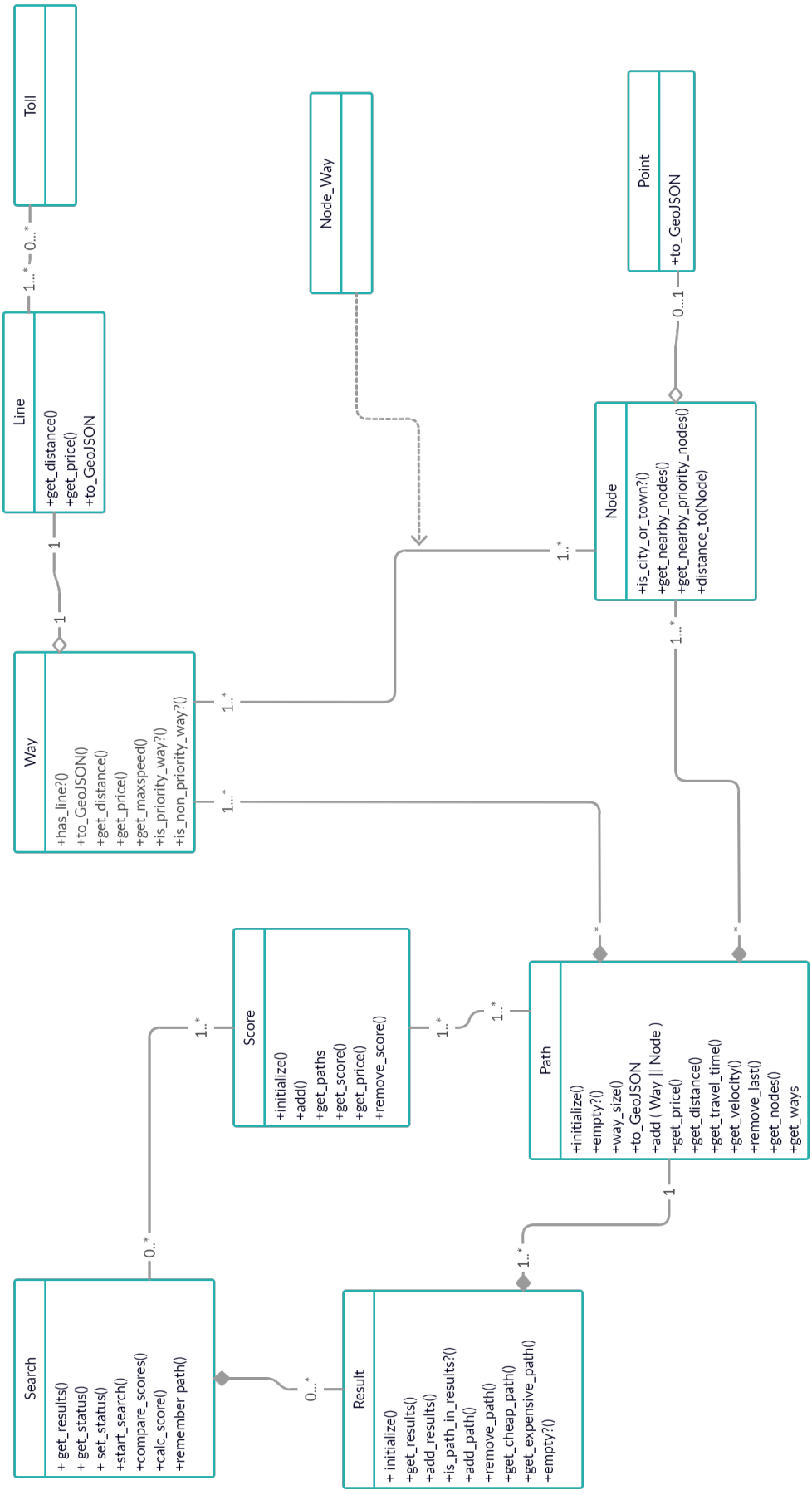

Figura 3.2: Diagrama de clases.

CAPÍTULO 4
Tecnologías y herramientas

En esta sección vamos a describir varias tecnologías, tanto aquellas que se han utilizado como las que se podrían haber llegado a usar.

4.1 Posibles tecnologías

Antes de empezar a desarrollar la aplicación web, se analizaron los diferentes *frameworks* que podrían ser utilizados para desarrollar el producto deseado:

- **Symfony:** Uno de los *frameworks* de PHP más utilizados. Symfony [5] hace uso del patrón Modelo-Vista-Controlador (MVC), proporcionando al programador una experiencia *full-stack*. Además este *framework* es compatible con todos los sistemas gestores de bases de datos relacionales (SGBDR) más extendidos, permitiéndonos utilizar el que más se adapte a nuestras necesidades, e incluso modificar o cambiar a otro distinto en cualquier punto del desarrollo con muy poco esfuerzo.
- **Angular:** *Framework* basado en TypeScript comúnmente conocido por ser utilizado para crear aplicaciones web de una sola página. Angular [6] destaca por su alta velocidad y rendimiento, gracias a su sistema de renderizado asíncrono, en el que el cliente solo carga el código que es estrictamente necesario para la acción que va a realizar.
- **Spring:** Spring [7] es un *framework* ampliamente utilizado para crear servicios web basado en Java. La principal ventaja de la utilización de este *framework* es el diseño modular de éste, permitiéndonos centrarnos en las funcionalidades que estamos desarrollando en cada momento sin preocuparnos por el resto. Otra ventaja de este diseño modular es la facilidad para el testeo que nos aporta.

  Un detalle a tener en cuenta de Spring [7] es que este *framework* nos ofrece muy poca abstracción en las conexiones a la base de datos, lo que nos obliga a depender de librerías externas como Hibernate [8], que facilitan crear las relaciones entre los modelos de nuestra aplicación y los objetos alojados en nuestra base de datos.
- **Django:** Django [9] es un *framework* basado en Python que hace especial énfasis en el principio «No te repitas.» - Foote, Steven (2014) [10], convirtiéndolo en un *framework* que ofrece un desarrollo rápido y propicia que se obtenga un código claro como resultado. Además Django [9] nos ofrece un alto nivel de seguridad gracias al alto grado de configuración de su *middleware* que nos permite, entre otras opciones, descartar peticiones a direcciones erróneas y que da soporte por defecto al uso de sesiones y usuarios, permitiéndonos no depender de herramientas externas para dichos fines.

Aunque cualquiera de estas tecnologías hubiera sido una buena opción, se optó por utilizar el *framework* Ruby on Rails (RoR)[11], ya que consideramos que es un *framework* con mucho potencial pero que pese a contar con muchas características interesantes su uso no está tan extendido como otros *frameworks* ya mencionados.

4.2 Tecnologías utilizadas

A continuación se van a presentar algunas de las tecnologías utilizadas junto con un breve resumen de su función en la solución desarrollada.

4.2.1. *HyperText Markup Language*

HyperText Markup Language (HTML) [12] lenguaje básico en la creación de cualquier pagina web, siendo compatible con cualquier navegador web y con multitud de plataformas como móvil, ordenador o *tablet*. Este lenguaje es el encargado de definir y estructurar el contenido que se mostrará al usuario final.

El funcionamiento de HTML [12] se basa en etiquetas, que identifican el tipo del contenido que se va a mostrar. Esta característica nos permite que elementos externos a la propia pagina web como imágenes, vídeos o sonido no se incrusten directamente en el código, sino que se introduzcan mediante las rutas en las que se encuentran estos recursos.

4.2.2. *Cascading Style Sheets*

Cascading Style Sheets o más conocido como CSS [13] es un lenguaje de diseño creado específicamente para definir y especificar la apariencia de un lenguaje marcado, en nuestro caso HTML [12]. Una de las ventajas que nos aporta el uso de CSS [13] es la capacidad de diferenciar el contenido, del aspecto con el que éste se visualizará, permitiéndonos así utilizar la misma hoja de estilos CSS [13] para diferentes documentos HTML [12].

4.2.3. *Bootstrap*

Bootstrap [14] es una biblioteca multiplataforma de código abierto que nos facilita enormemente la creación de interfaces web. Esta biblioteca nos ofrece una colección de plantillas de diferentes elementos web, como botones, tablas, formularios, etc. Y nos permite modificarlos para que se adapten a nuestras necesidades, ahorrándonos las molestias de la implementación de estos elementos desde cero.

4.2.4. CoffeeScript

CoffeeScript[15] es un lenguaje de programación que se compila a JavaScript, por lo que las funciones que se realizan en este lenguaje se podrían realizar de forma idéntica en JavaScript y viceversa. Las principal ventaja que nos ofrece CoffeeScript[15] frente a su predecesor son una mayor brevedad y legibilidad en el código, sin ninguna repercusión en el rendimiento, reduciendo la extensión del código a un tercio de su versión en JavaScript, según palabras de su creador[16].

4.2.5. JQuery

JQuery[17] es una biblioteca de código abierto desarrollada en JavaScript. Una de las principales funcionalidades que nos aporta esta biblioteca es la posibilidad de seleccionar diferentes elementos de nuestra interfaz y modificar su apariencia y comportamiento durante la ejecución de nuestra página web, dotando a la interfaz de un mayor dinamismo e interactividad. Otra característica destacable de esta biblioteca son las facilidades que nos aporta para crear llamadas AJAX [18] lo que nos permite refrescar algunos elementos de la interfaz en tiempo de ejecución, como tablas o listas de resultados, permitiéndonos mostrar nuevos resultados conforme éstos se van calculando.

4.2.6. *Asynchronous JavaScript And XML*

Asynchronous JavaScript And XML o más conocido como AJAX [18] es una técnica de desarrollo web que nos permite actualizar algunos elementos de nuestra pagina web sin necesidad de recargarla. Esta característica es posible manteniendo una conexión asíncrona entre el cliente y el servidor, solicitando el cliente de forma periódica, y en segundo plano, los nuevos datos al servidor y actualizando los apartados correspondientes en caso de que haya cambios en los datos obtenidos.

4.2.7. *Geographic JavaScript Object Notation*

Geographic JavaScript Object Notation o más conocido como GeoJSON [19] es un estándar abierto diseñado específicamente para representar elementos geográficos sencillos. Este estándar no solo nos permite intercambiar información geográfica de los elementos sino que también nos permite añadir información adicional que consideremos relevante, por ejemplo en el caso de una ciudad podríamos añadir la población, fecha de fundación, etc.

4.2.8. Leaflet

Leaflet [20] es una biblioteca de código abierto desarrollada en JavaScript. Esta biblioteca nos ofrece una forma simple y eficiente de añadir un mapa interactivo a nuestra aplicación web, permitiéndonos añadir multitud de elementos a estos mapas y siendo compatible con múltiples herramientas que extienden su funcionalidad base.

4.2.9. PostgreSQL

PostgreSQL [21] es un sistema de gestión de bases de datos relacional (SGBDR) de código abierto. PostgreSQL [21] nos ofrece gran variedad de tipos de datos nativos, proporcionándonos la opción de guardar datos como direcciones IP, objetos geométricos o *arrays* sin necesidad de convertirlos a otros formatos. Otra de las características relevantes de este SGBDR es la alta concurrencia que soporta, ya que permite que mientras un proceso escribe en una tabla otros procesos accedan a la misma taba sin necesidad de bloqueos.

4.2.10. PostGIS

PostGIS [22] es un módulo que extiende PostgreSQL [21] haciéndolo compatible con objetos geográficos, permitiéndonos almacenar objetos espaciales y aportándonos las funciones necesarias para operar con ellos.

4.2.11. Ruby On Rails

Ruby on Rails [11] también conocido como RoR es un *framework* de desarrollo web basado en Ruby. RoR [11] es una opción viable para desarrollar una página web con el mínimo de configuración posible y escribiendo menos código que con otros *frameworks*. Además implementa el patrón Modelo-Vista-Controlador (MVC), permitiéndonos crear un código más cohesionado y más fácil de testear. Es la tecnología principal en la que se basa nuestra solución, que se encarga que conectar el resto de tecnologías y en la que se ejecutan la mayoría de cálculos algorítmicos.

Una de las características más interesantes de RoR [11] es el uso del patrón *Active Record* [23] como característica base en su diseño, esta característica nos permite abstraernos de las conexiones con la base de datos, mapeando nuestras instrucciones en Ruby a las instrucciones SQL correspondientes.

Otra característica destacable de RoR [11] es la facilidad con la que nos permite añadir nuevas bibliotecas, o gemas como se denominan dentro del *framework*, ya que solamente tendremos que especificar que gema y que versión queremos instalar y el *frameworks* se encargará de instalar todas las dependencias necesarias por nosotros.

CAPÍTULO 5
Diseño de la aplicación

Después de haber introducido las tecnologías utilizadas en el apartado anterior, en esta sección vamos a detallar la estructura utilizada en nuestro proyecto.

5.1 Arquitectura del sistema

Como ya se ha dicho anteriormente, una de las principales características de las aplicaciones desarrolladas en RoR [11] es la implementación del patrón Modelo-Vista-Controlador (MVC), y nuestra aplicación no es una excepción. La finalidad de este patrón es la de separar los diferentes componentes de la aplicación, lo que nos resulta de gran ayuda durante el desarrollo de la misma y nos facilita identificar donde se ejecutará cada fragmento de código, ya que como podremos observar en la figura 5.1 no todos los componentes se ejecutan en la mima capa del sistema. Los componentes que define el patrón MVC son los siguientes:

- **Modelo** En este componente se encuentra la lógica de nuestra aplicación, es donde se halla nuestro algoritmo de búsqueda y es el componente encargado de comunicarse con la capa de almacenamiento de datos.
- **Controlador** Es el encargado de gestionar las peticiones del usuario, decidiendo que acciones de los modelos se van a ejecutar para satisfacer estas peticiones y que vistas se van a mostrar al usuario como respuesta.
- **Vista** En este componente se encuentran los recursos necesarios para dar soporte visual al usuario, facilitando así el uso de la aplicación y mejorando en general la experiencia del usuario.

En la figura 5.1 se puede observar una representación visual de la estructura del sistema, donde se pueden identificar los elementos que la componen, junto a las principales tecnologías utilizadas en cada uno de ellos y en que capa de nuestro sistema se ejecutan.

El flujo de comunicación más común entre los diferentes componentes de nuestra aplicación empezaría en el cliente, donde el usuario realizaría una petición, ya sea interactuando con las vistas o introduciendo directamente una ruta asociada a nuestra aplicación. Esta petición realizada en la capa del cliente se transmitirá a la capa del servidor, más concretamente al controlador asociado a la vista o la ruta correspondiente.

El controlador será el encargado de comprobar de que la petición es correcta, verificando que le origen y el destino de la ruta son poblaciones diferentes, que los parámetros de la búsqueda son correctos, etc. Si la petición contiene errores se informará al usuario

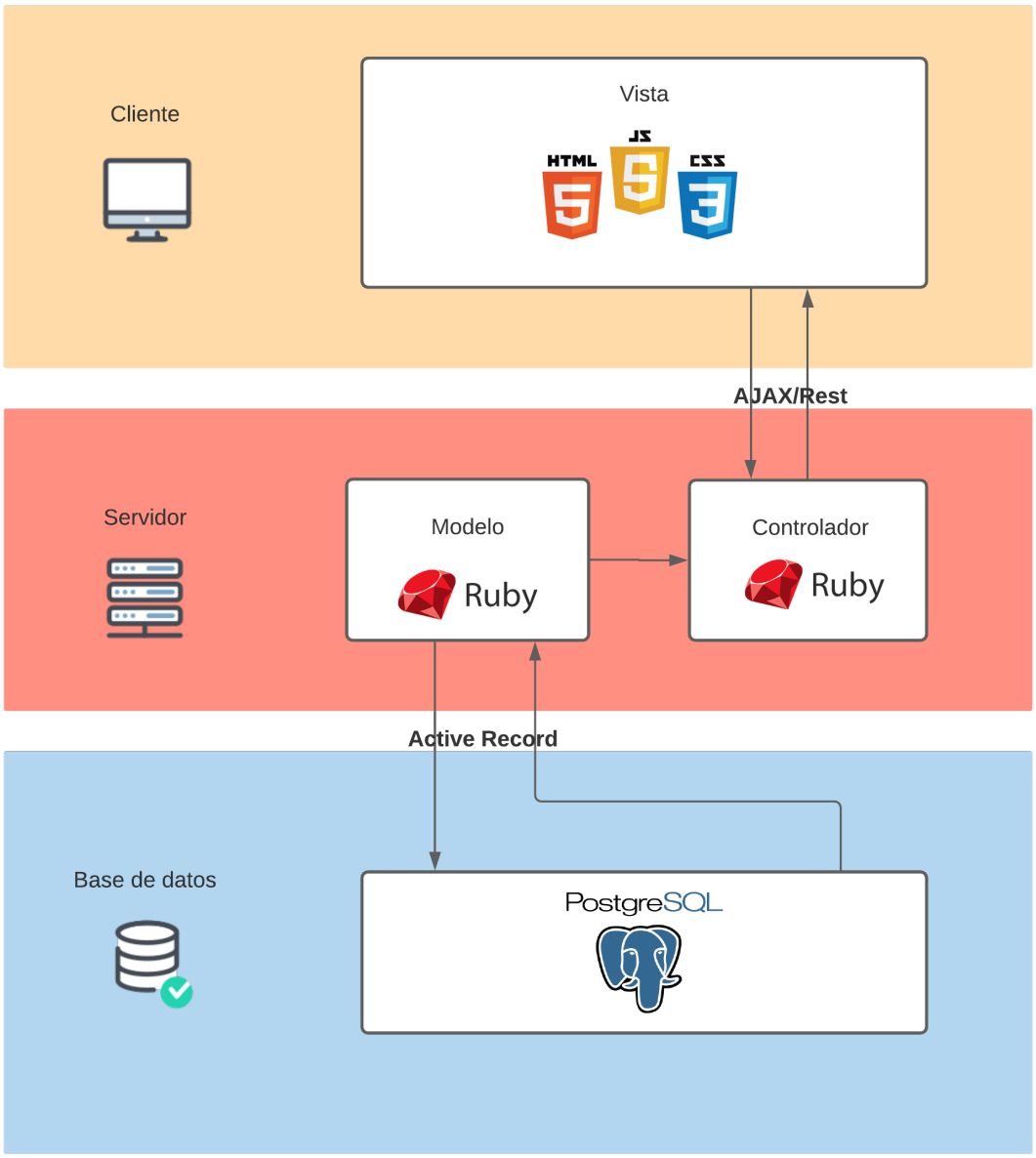

Figura 5.1: Diagrama de la arquitectura del sistema de la aplicación web.

del error mediante un mensaje informativo en la vista correspondiente para que éste se pueda corregir en las siguientes peticiones. En caso de que la petición sea válida, ésta se transmitirá a los modelos donde, en el caso concreto de nuestra aplicación, se creará un nuevo proceso que será el encargado de buscar las rutas correspondientes, realizando las peticiones necesarias a la capa de la base de datos.

Al empezar el proceso de búsqueda en los modelos, se notificará al usuario que su petición está siendo gestionada. También se actualizarán las vistas frecuentemente mostrando las posibles rutas encontradas a su petición hasta que la búsqueda termine o hasta que el usuario la detenga mediante la interfaz, actualizando una última vez las vistas, mostrando así todas las rutas alternativas encontradas.

5.2 Estructura de datos

Para almacenar la información de las diferentes vías de transporte y poblaciones que utiliza nuestra aplicación para calcular las rutas solicitadas se ha optado por el uso de PostgreSQL [21]. Se ha decidido utilizar esta tecnología tanto por las sinergias que este gestor de bases de datos presenta con RoR [11] como por la posibilidad de migrar los datos de la organización Open Street Maps [24], fuente de la que se han obtenido los datos necesarios para poblar nuestra base de datos y cuyo proceso de migración se explicará con mayor detalle en el apartado 6.1.

Al tratarse PostgreSQL [21] de un sistema de gestión de bases de datos relacional orientado a objetos, una representación visual de la estructura de nuestra base de datos nos resulta de gran ayuda para entender el funcionamiento de la misma.

El diseño de la estructura de nuestra base de datos está inspirado en el diseño utilizado por la organización OSM [24], ya que utilizaremos los datos de esta organización como fuente de datos y éstas similitudes nos facilitarán enormemente la migración de los datos. Aunque, como es lógico, nuestro diseño también cuenta con características propias destinadas a satisfacer nuestros objetivos y mejorar el rendimiento de la aplicación en el mayor grado posible.

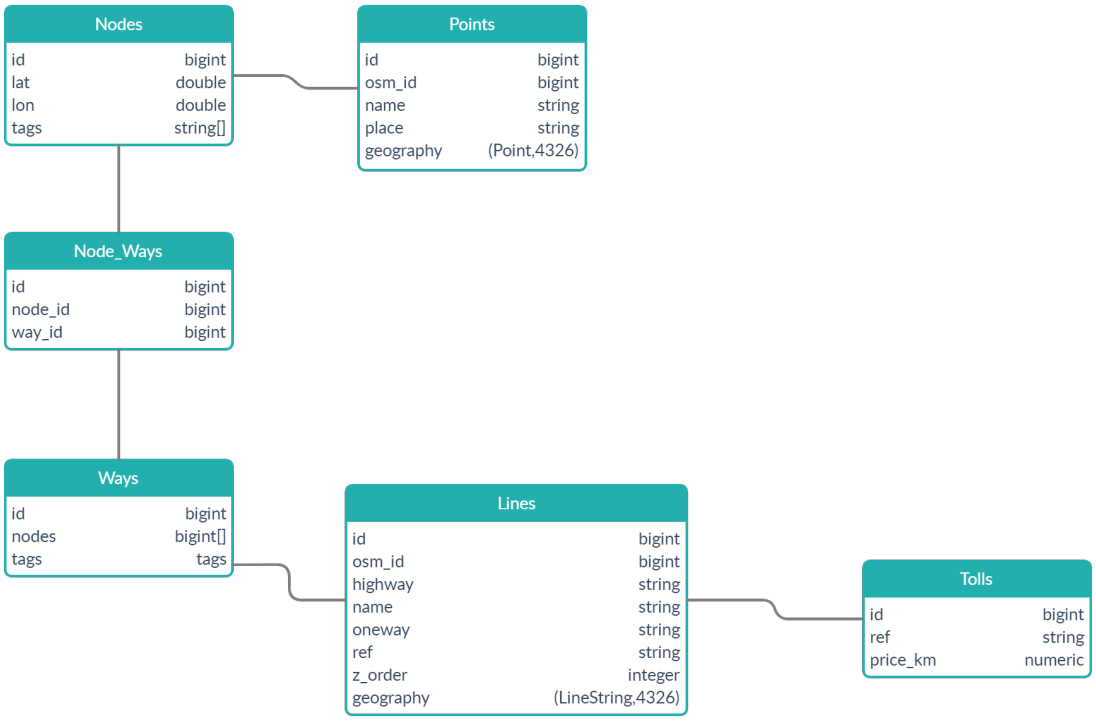

Figura 5.2: Diagrama de la arquitectura de la base de datos de la aplicación web.

Como se puede observar en la figura 5.2, nuestra base de datos consta de seis tablas, entre las cuales destacan la tabla «Ways» y la tabla «Nodes», ya que en estas tablas se almacenan los datos de los diferentes elementos que componen la red de carreteras de la región en la que opera nuestra aplicación, más concretamente la vía de carreteras del territorio español. El resto de tablas que se pueden observar nos ofrece información adicional de los elementos «Nodes» y «Ways», esta información nos permite representar los elementos mencionados visualmente en un mapa o conocer con que otros elementos están comunicados. En la tabla «Ways» se almacenan los datos de los diferentes tramos de

las vías de transporte, mientras que en la tabla «Nodes» se almacenan las poblaciones y los puntos de unión entre los tramos de la vías de transporte anteriormente mencionadas.

- **Ways:** Si observamos detalladamente los elementos que forman la tabla «Ways» podemos observar que está compuesta por tres elementos. El elemento «id» es un identificador único que nos facilita el uso y la identificación de los diferentes objetos de la tabla. El elemento «nodes» es una lista de los identificadores de los «Nodes» con los cuales está relacionado cada objeto de la tabla «Ways». Y por último, el elemento «tags» es una lista de las diferentes características de la vía, en la cual podemos encontrar información de gran utilidad como el tipo de vía, la velocidad máxima permitida, etc.

- **Nodes:** En la tabla «Nodes», al igual que en el resto de tablas podemos encontrar el elemento «id», también se pueden observar los elementos «lat» y «lon» que representan la latitud y longitud asociada al nodo. Por último, el elemento «tags», al igual que en la tabla «Ways», consiste en una lista de características del elemento, que en este caso nos permite identificar de que tipo de nodo se trata, un cruce, un desvío, un centro urbano, etc.

- **Node-Ways:** En la figura 5.2 también podemos observar que las tablas «Node» y «Ways» están directamente relacionadas con la tabla «Node-Ways». Esta tabla se trata de una tabla de asociación que nos sirve para implementar una relación *many-to-many* de forma eficiente, reduciendo así el tiempo de ejecución de las consultas a la base de datos que pretendan obtener los «Ways» que están relacionados con un «Node» concreto o viceversa. Mediante este diseño se reduce en gran medida el tiempo de ejecución total de la búsqueda respecto a otras estrategias, teniendo en cuenta que este tipo de consultas se realizan cientos de veces por cada petición de un usuario.

  La tabla «Node-Ways» está formada simplemente por tuplas que contienen un «id» de un «Nodo» y un «id» de un «Way», por lo que si un «Nodo» está relacionado con cien «Ways» serán necesarias cien entradas en esta tabla para representar dichas relaciones. Esta característica aumenta considerablemente el tamaño de la base de datos, pero dado que mejora drásticamente el rendimiento de la aplicación se ha considerado adecuada.

- **Points:** La tabla «Points» contiene información necesaria para representar visualmente los «Nodes» en un mapa. En nuestra aplicación solamente se ha considerado relevante representar los «Nodes» que hacen referencia a una población, más concretamente a su centro urbano, por lo que el número de entradas de esta tabla será bastante menor que el número de entradas de la tabla «Nodes».

  La tabla «Points» está compuesta por el elemento «osm-id», el cual es una clave ajena de la tabla «Nodes», lo que relaciona directamente estas tablas y nos permite asegurarnos de que no existe ninguna entrada en la tabla «Points» que no esté representada también en la tabla «Nodes». El elemento «name» almacena el nombre de la población, mientras que el elemento «place» nos indica que tipo de población es, si un pueblo, una ciudad, etc, permitiéndonos organizar mejor los diferentes poblaciones en caso de que se quiera mostrar una lista de éstas al usuario.

  Por último, el elemento más destacable de esta tabla es el elemento «geography» el cual es un elemento de tipo «Point» de la extensión PostGIS [22], dicha extensión nos ofrece varias funcionalidades para almacenar elementos geográficos. El elemento «geography» es, más concretamente, un elemento «(Point,4326)» haciendo estos

últimos dígitos alusión al sistema de referencia espacial [25] usado en nuestra aplicación. Este atributo contiene información de la ubicación geográfica del «Node» por lo que es el elemento que nos permite representarlo visualmente en un mapa.

- **Lines:** La tabla «Lines» contiene información necesaria para representar visualmente los «Ways» en un mapa. A diferencia de la tabla «Nodes» todos los elementos de la tabla «Ways» deben de ser representables, por lo que el número de entradas de la tabla «Lines» será idéntico al número de entradas de la tabla «Ways».

  La tabla «Lines» está compuesta por el elemento «osm-id», el cual es una clave ajena de la tabla «Ways», lo que relaciona directamente estas tablas y nos permite asegurarnos de que no existe ninguna entrada en la tabla «Lines» que no esté representada también en la tabla «Ways». El elemento «highway» nos indica el tipo de vía al que hace referencia por ejemplo una autovía, una carretera segundaría, etc. El elemento «ref» simplemente nos indica las siglas de la vía a la que pertenece el tramo al que hace referencia cada entrada en la tabla «Lines» si éste pertenece a una vía numerada como la «Ap-7» o la «V-31». En los casos en los que los tramos no pertenecen a una vía de este tipo por ejemplo en el caso de una vía de servicio, el elemento «ref» estará vació y será el elemento «name» el que contendrá una descripción breve del tramo y para que se utiliza éste.

  Por último, el elemento más destacable de esta tabla es el elemento «geography» el cual es un elemento de tipo «LineString» de la extensión PostGIS [22], dicha extensión nos ofrece varias funcionalidades para almacenar elementos geográficos. El elemento «geography» es más concretamente un elemento «(LineString,4326)» haciendo estos últimos dígitos alusión al sistema de referencia espacial [25] usado en nuestra aplicación y contiene información de la ubicación geográfica del «Way», por lo que es el elemento que nos permite representarlo visualmente en un mapa.

- **Tolls:** La tabla «Tolls» nos ofrece información útil a la hora de calcular los costes de peajes de cada ruta. La tabla «Tolls» está compuesta por un elemento «id» que nos permite operar con las diferentes entradas de esta tabla. También podemos observar la presencia del elemento «ref», que contiene las siglas de la vía a la que hace referencia cada entrada de la tabla «Tolls», y nos permite relacionarla con todos los elementos de las tablas «Lines» que tengan el mismo valor su atributo «ref» o con los «Ways» que contengan esta referencia en su atributo «tags». Por último, el elemento «price-km» contiene el precio medio de peaje por kilómetro asociado a cada carretera de peaje.

5.3 Diseño inicial

En este apartado vamos a explicar el diseño de interfaz elegido, mostrando algunos de los prototipos realizados y explicando los motivos que nos han llevado a elegir este diseño.

Para el diseño de nuestra aplicación se ha optado por usar un diseño *single-page application*[26], ya que este tipo de diseño nos permite crear una interfaz muy simple con tan solo los elementos imprescindibles por la que el usuario podrá navegar muy fácilmente, intentando que la interfaz tenga el mayor grado de usabilidad y el menor tiempo de aprendizaje posible.

Como se puede ver en la figura 5.3 ha optado por un diseño en el que el formulario de búsqueda de una ruta esté siempre visible, añadiendo a esta vista elementos adicionales o modificando algunos ya presentes en caso de que ya se esté efectuando una búsqueda o que se estén mostrando los resultados de ésta.

Figura 5.3: Prototipo de la vista inicial

Si observamos la figura 5.4 podremos ver que los elementos que se modifican al empezar una nueva búsqueda son el botón de «Buscar ruta» que pasa a mostrar el texto «Detener búsqueda» y permite al usuario detener la búsqueda ya iniciada. También se añade un nuevo *spinner* junto con el texto «Buscando» para indicar al usuario que su búsqueda se está realizando. Además, una vez que se encuentra una ruta que satisface las condiciones de la búsqueda, se añade un mapa y una tabla expandible que contiene todas las soluciones encontradas hasta el momento, si hacemos *click* en una ruta mostrada en la tabla, ésta se mostrará en el mapa y se expandirá para que podamos visualizar los detalles de los tramos que la conforman.

Por último, cuando una búsqueda se finaliza o se detiene, desaparece el *spinner* que nos indicaba que nuestra búsqueda se estaba realizando, y la tabla que contiene las soluciones encontradas se mantiene hasta que se inicie una nueva búsqueda.

Uno de los objetivos de este proyecto es que nuestra aplicación sea utilizable en todo tipo de dispositivos, desde ordenadores a otros terminales con periféricos de menor tamaño como móviles o tabletas. Por lo que para completar este objetivo con éxito es necesario que todas las interfaces anteriormente mencionadas implementen un diseño adaptativo o *Responsive design* [27] que adapte el tamaño y la posición de los diferentes elementos de la interfaz al tamaño de la pantalla en la que se están visualizando.

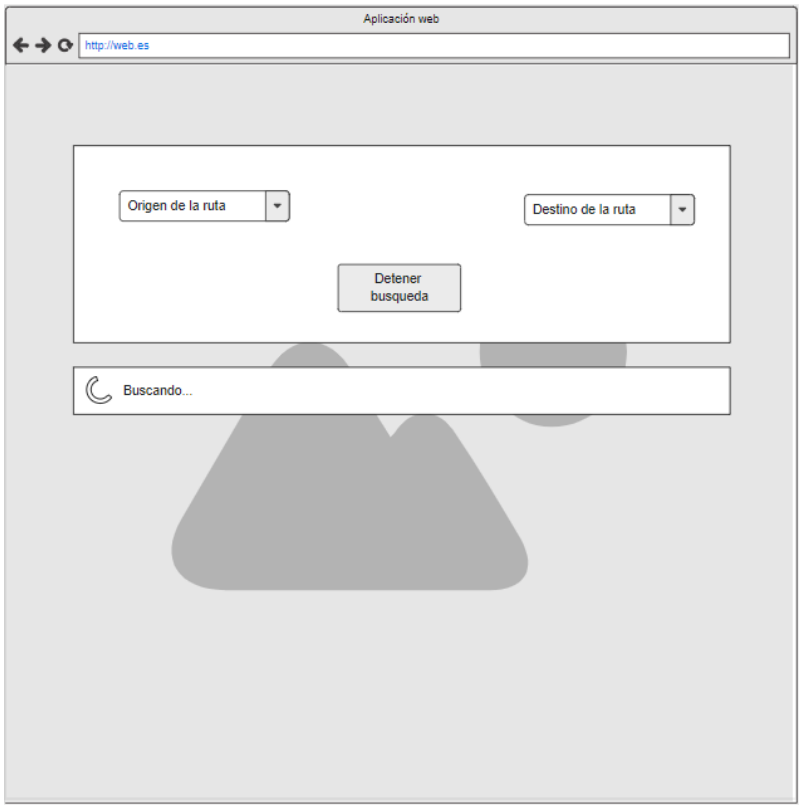

Figura 5.4: Prototipo de la vista al iniciar una búsqueda

Figura 5.5: Prototipo de la vista al encontrar resultados

CAPÍTULO 6
Implementación

Desde los primeros momentos en los que empezamos a implementar la aplicación se identificaron dos grandes hitos.

El primero de ellos fue poblar nuestra base de datos, ya que esto implicaba encontrar una fuente de datos que contuviera datos reales y en la que pudiéramos aplicar las transformaciones necesarias para que fuera compatible con la estructura deseada para nuestra base de datos.

El segundo hito que identificamos fue el diseño de un algoritmo de búsqueda capaz de lidiar con un gran número de datos reales y que nos ofreciera un rendimiento aceptable para el tipo de producto que se estaba desarrollando.

En este capítulo vamos a abordar como se desarrollaron estos hitos y que soluciones alternativas se descartaron, junto a los motivos por los cuales éstas se descartaron.

6.1 Implementación de la base de datos

En esta sección se van a detallar los pasos realizados para la obtención de nuestra base de datos poblada y funcional, desde el proceso de creación, a los últimos ajustes realizados, pasando por la población de la misma.

6.1.1. Creación de la base de datos

Como ya se ha dicho anteriormente, la creación de la base de datos para una aplicación en RoR [11] es una tarea bastante simple, ya que este *framework* nos permite la modificación de una base de datos mediante *scripts* escritos en lenguaje Ruby. Estos *scripts* serán numerados y guardados dentro del proyecto, y sera el propio RoR [11] el encargado de ejecutar los *scripts* si la modificaciones que describen no se han aplicado ya a la base de datos asignada.

Antes de crear o ejecutar los *scipts* necesarios para crear la estructura de nuestra base de datos, será necesario la configuración de la conexión de nuestra aplicación con nuestro servidor de bases de datos PostgreSQL [21]. Esta configuración se realiza en el archivo «condig/database.yml» dentro de nuestra aplicación. En este archivo se introducirán tanto la dirección IP del servidor de base de datos como las credenciales de acceso a dicho servidor, también sera necesario especificar el nombre de la base de datos que se va a utilizar.

Una vez realizada la configuración descrita podremos crear la base de datos, ya sea mediante la interfaz de PostgreSQL [21] o mediante el uso del comando de RoR [11]

«rails db:create». Por último, antes de la creación de la estructura de la base de datos, será necesario añadir la extensión PostGIS [22], necesaria para el uso de algunos de los tipos de datos que se utilizarán para almacenar elementos geográficos.

A continuación se incluye uno de los *scripts* utilizados para implementar la estructura de nuestra base de datos, más concretamente este *script* es el encargado de la creación de la tabla «Lines».

```
1  class CreateLines < ActiveRecord::Migration[5.2]
2    def change
3      create_table :lines do |t|
4      t.bigint :osm_id, foreign_key: true
5        t.string :highway
6        t.string :name
7        t.string :oneway
8        t.string :ref
9        t.integer :z_order
10       t.line_string :way,   geographic: true, srid: 4326
11     end
12     add_foreign_key :lines, :ways, column: :osm_id, primary_key: "id"
13   end
14 end
15
```

Figura 6.1: *Script* encargado de crear la tabla «Lines»

Como se puede observar en la figura 6.1, el *script* nos permite definir los diferentes elementos de la tabla, también nos permite identificar los tipos de datos de los atributos, los nombres de estos atributos e incluso, como se puede observar en la línea doce, nos permite definir las claves ajenas de esta tabla. Otra característica de estos *scripts* es que nos permiten crear atributos de tipos no incluidos en la versión base de PostgreSQL [21], como se puede ver en la línea diez, donde creamos un atributo de un tipo propio de la extensión PostGIS [22].

Por último, una vez desarrollados los *scripts* necesarios, los podemos ejecutar mediante el comando de RoR [11] «rails db:migrate», el cual aplicará las modificaciones descritas en dichos *scripts* a nuestra base de datos.

6.1.2. Obtención de los datos de la base de datos

Para obtener datos reales con los que poblar nuestra base de datos usamos como fuente de datos la organización OSM [24]. Esta organización obtiene sus datos de una forma similar a Wikipedia [28], ya que son los usuarios voluntarios los que de forma desinteresada realizan aportes para mantener los datos actualizados. Una vez moderados y aceptados estos aportes pasan a formar parte de la base de datos de esta organización, y pueden ser utilizados de forma gratuita.

Para obtener los datos de la OSM [24] podemos acceder directamente a la página web de esta organización, donde podremos descargar en un archivo de formato «osm» toda la información del área geográfica que seleccionemos, en nuestro caso obtuvimos toda la información del territorio español.

Una vez descargado el archivo en formato «osm» hicimos uso de la herramienta de código abierto «osm2pgsql». Esta herramienta nos permite transferir la información del archivo «osm» a una base de datos de PostgreSQL [21] mediante el uso de comandos en

la consola. Este proceso requiere el procesado de un gran número de datos, por lo que en casos de regiones geográficas extensas el proceso tardará varios días en finalizar.

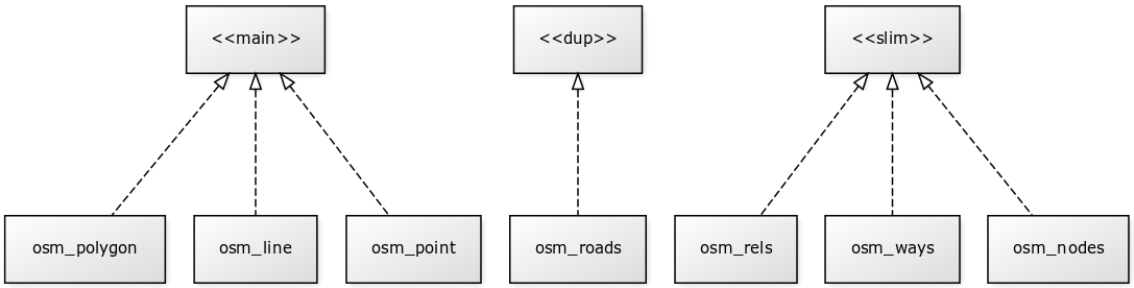

Figura 6.2: Esquema de la base de datos obtenida de la OSM

Una vez ha finalizado este proceso dispondremos de una base de datos con información de todo el territorio español, en la cual podremos consultar la información que necesitemos y realizar las modificaciones que consideremos necesarias.

6.1.3. Migración de datos

Después de obtener una base de datos poblada con datos reales con la que podemos interactuar y en la que podemos realizar cualquier consulta, debemos de transferir estos datos a la base de datos de nuestra aplicación.

Antes de empezar a realizar las migraciones correspondientes para transferir los datos de una base de datos a otra, se debe de tener en cuenta que la base de datos creada a partir de los datos de la OSM [24] contiene información muy variada, y gran parte de ella no nos resultara útil. Por lo tanto, antes de realizar la migración de los datos es necesario filtrarlos, manteniendo solo aquellos que nos resulten de utilidad, así reduciremos el tiempo necesario para la migración posterior e incluso el tiempo de ejecución de la mayoría de consultas realizadas en la aplicación final.

Como ya se ha dicho anteriormente la base de datos de la OSM [24] contiene información muy variada, desde rutas de autobús, a limites de provincias o calles peatonales. Para filtrar los datos que deseábamos sin corromper la base de datos original, se creó un nuevo *schema* que replicaba la estructura de la base de datos original, permitiéndonos así filtrar los datos manteniendo la fuente de éstos.

```
INSERT INTO test.osm_line(osm_id, highway, name, oneway, ref,
z_order, way)
      SELECT
            planet_osm_line.osm_id,
            planet_osm_line.highway,
            planet_osm_line.name,
            planet_osm_line.oneway,
            planet_osm_line.ref,
            planet_osm_line.z_order,
            planet_osm_line.way
      FROM
      public.planet_osm_line
INNER JOIN test.osm_ways
ON public.planet_osm_line.osm_id = test.osm_ways.id;
```

Figura 6.3: Fragmento de las instrucciones SQL destinadas a filtrar la base de datos de OSM

En este filtrado de datos se obtuvo toda la información correspondiente a las vías de transporte automovilístico, desde autovías a vías de servicio o calles urbanas. Mediante esta operación no solo se rescataron todos los elementos de la tabla «Ways» que hacían referencia directa a estas vías, sino que también se obtuvieron los elementos de la tabla «Nodes» relacionadas con estas vías, como intersecciones, cruces, cambios de vía, etc. También se migraron los datos de las poblaciones con mayor número de habitantes, obteniendo un total de novecientas setenta poblaciones. Una vez aislados los elementos «Ways» y «Nodes» nos resulta muy fácil obtener los datos necesarios para representarlos visualmente ya que, al igual que en nuestro diseño de la base de datos, éstos estaban directamente relacionados.

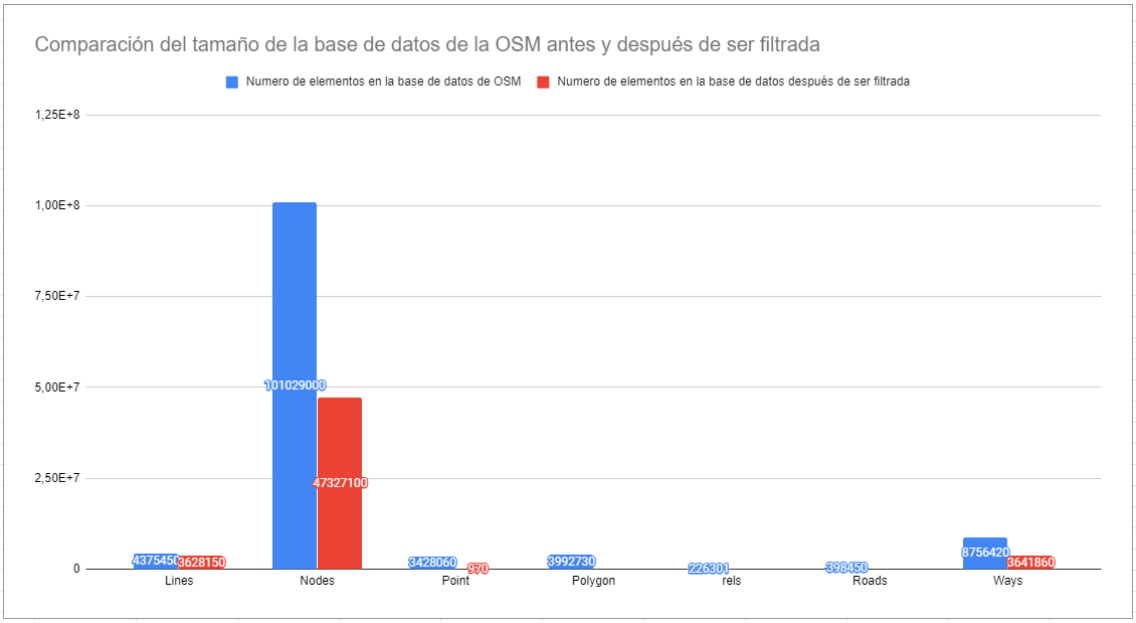

Figura 6.4: Comparación del tamaño de la base de datos antes y después de su filtrado

Como cabría esperar, el filtrado de estos datos también fue una operación costosa computacionalmente, debido al elevado número de datos, pero después de realizar estas operaciones ya tenemos una base de datos que contiene solamente aquella información que deseamos lista para ser transformada en la estructura de nuestra base de datos final. Como se puede observar en la figura 6.4 la reducción del tamaño de la base de datos fue drástica, el mayor número de elementos descartados corresponde a los elementos almacenados en la tabla «Nodes», donde se pasa de tener más de cien millones de elementos a cerca de cincuenta millones. Esta reducción se debe a que solo se conservaron los elementos relacionados con vías de transporte y centros urbanos. También se puede observar que algunas de las tablas pasan a estar vacías ya que la información que contenían no nos era de utilidad. Otra particularidad que podemos observar en la figura es que el mayor número de elementos de las tablas «Lines» y «Ways» se han mantenido, esto es debido a que la mayoría de la información almacenada en estas tablas corresponde a los tramos de la red de carreteras del territorio nacional, datos de los que hace uso nuestra aplicación, y solamente se han desechado los datos correspondientes a carriles bici, vías peatonales o vías ferroviarias.

El siguiente paso que se realizó fue la migración de los datos a la base de datos de nuestra aplicación, para esto fueron necesarias varias transformaciones, debido a que las estructuras de ambas bases de datos comparten similitudes pero no son exactamente iguales. Estas transformaciones se realizaron junto a la migración mediante instrucciones SQL. Algunas de las transformaciones más importantes que se realizaron fueron el cambio del SRID [25] de los objetos geográficos, como «Points» o «Lines» ya que el SRID

[25] utilizado en la base de datos de origen no era compatible con las herramientas que utilizábamos para representar los elementos visualmente en un mapa.

Otra transformación importante fue la creación de la tabla «Node-Ways» en base a las entradas de las tablas «Node» y «Ways» de la base de datos de origen. Esta nueva tabla nos permite un mayor rendimiento en las consultas a la base de datos que pretendan obtener los «Ways» que están relacionados con un «Node» concreto o viceversa. Al haber filtrado los datos previamente, la base de datos con la que operamos tiene un tamaño significativamente más reducido que la base de datos que obtuvimos de la organización OSM [24], por lo que el tiempo de ejecución de esta migración también será menor que el resto de operaciones realizadas previamente.

Por último, la última tabla en la que no tenemos información en nuestra base de datos era la tabla «Tolls», en esta tabla se almacena la información necesaria para calcular el precio del peaje de las rutas propuestas a los usuarios. Dado que la base de datos de la organización OSM [24] no nos ofrecía información sobre el precio de estos peajes, recurrimos a otra fuente de datos, más concretamente a la página oficial de Abertis [30], empresa encargada de la gestión de la mayoría de autopistas españolas, en esta página web podemos consultar los precios de cada uno de los tramos de sus autopistas para todas las modalidades de vehículos. En esta ocasión, estos datos se insertaron mediante instrucciones simples de inserción SQL, ya que el número de datos a añadir era reducido y no disponíamos de una base de datos con esta información.

Una vez realizadas éstas operaciones ya disponíamos de una base de datos con información de toda la extensión de la red de carreteras del territorio español y con un gran número de poblaciones con las que nuestra aplicación puede operar.

6.2 Algoritmo de búsqueda

El segundo gran hito que debíamos de completar en nuestro proyecto fue el diseño e implementación de un algoritmo de búsqueda capaz de buscar diferentes caminos en un entorno real, intentando conseguir unos tiempos de ejecución aceptables.

6.2.1. Elección del algoritmo de búsqueda

Para la elección del algoritmo de búsqueda que se iba a usar se barajaron diferentes tipos de algoritmos cuyo diseño podríamos adaptar a nuestras necesidades:

- **Algoritmo de Floyd-Warshall**: El algoritmo de Floyd-Warshall es un algoritmo basado en la teoría de grafos, diseñado para encontrar el camino mínimo en un grafo ponderado. Este algoritmo destaca por su capacidad de encontrar el camino mínimo entre cualquier par de nodos del grafo en una única ejecución.

  A simple vista este algoritmo nos resultó atractivo debido a la posibilidad de realizar una ejecución y guardar en la base de datos sus resultados. De este modo, cada vez que un usuario nos realizara una petición ya dispondríamos de antemano de un camino supuestamente válido, el cual podríamos expandir o usar como base para otros algoritmos e intentar encontrar un abanico más grande de alternativas que ofrecer al usuario.

  Una vez se indagó más en el funcionamiento de este algoritmo, rápidamente nos dimos cuenta de que el uso de este algoritmo era inviable. Esto es debido a que es imprescindible para su ejecución la creación de una estructura de datos en memoria del tamaño del número de nodos al cuadrado, y teniendo en cuenta que en nuestra

base de datos contamos con información de cerca de cincuenta millones de nodos el coste en memoria sería inasumible. Al igual que sería inasumible el tiempo de ejecución de este algoritmo debido al altísimo número de iteraciones que tendría que realizar antes de proporcionarnos una solución. Otro impedimento sería el coste en memoria física de guardar las soluciones que este algoritmo nos proporcionaría en caso de que nos fuera posible aplicarlo en nuestra aplicación.

Por estos motivos se decidió descartar el uso de este algoritmo, ya que solamente nos sería útil en casos con un número de datos mucho más reducido, y por lo tanto no sería recomendable su aplicación en nuestro proyecto.

- **Algoritmo de Dijkstra** El algoritmo de Dijkstra es posiblemente el algoritmo de búsqueda más conocido, este algoritmo nos permite conocer la distancia mínima de un nodo origen determinado al resto de nodos.

  El uso de este algoritmo se implementó en un entorno de pruebas con una base de datos con un número de nodos reducido. En esta implementación se usaba el algoritmo de Dijkstra para encontrar el camino más corto, para luego buscar otros caminos alternativos y compararlos entre sí mediante un algoritmo propio, obteniendo resultados satisfactorios.

  Pero al igual que con el algoritmo de Floyd-Warshall, el uso de memoria de este algoritmo crece exponencialmente con el número de nodos con los que se opera, dificultando enormemente su uso en nuestro proyecto. Esta característica junto al hecho de que los objetivos de este algoritmo no encajaban con nuestras necesidades nos obligó a descartar el uso de este algoritmo y a buscar otras alternativas.

- **Algoritmo de búsqueda A*** El algoritmo A* es un algoritmo heurístico diseñado para encontrar el camino mínimo entre dos nodos. Los algoritmos heurísticos se basan en una formula matemática para puntuar los diferentes nodos que están a nuestro alcance en función de los caminos que llevan a ellos, y de esta forma expandir aquellos nodos más prometedores, ya que son los que más probabilidades tienen de ser parte del camino mínimo. El coste en memoria de este algoritmo crece exponencialmente conforme se van explorando nuevos nodos, ya que será necesario guardar tanto la puntuación de este nodo como el camino que nos ha llevado a él.

  Aunque este algoritmo presenta algunos problemas también presenta algunas ventajas que nos resultaron relevantes, por lo que se decidió tenerlo en cuenta a falta de una alternativa mejor.

- **Algoritmo de búsqueda SMA*** El algoritmo *Simplified Memory Bounded A** es una variación del algoritmo A*, el cual se caracteriza por guardar en memoria solamente un número determinado de nodos, guardando en los padres la mejor puntuación de los hijos borrados.

  Este algoritmo nos ofrece la posibilidad de usar un algoritmo heurístico con un coste de memoria acotado, permitiéndonos teóricamente su ejecución en servidores con prestaciones muy modestas, aunque esto afecte gravemente al tiempo de ejecución de este algoritmo, ya que en caso de que la predicción heurística falle y el camino expandido no sea el mínimo es posible que se tengan que volver a expandir los nodos ya olvidados, afectando negativamente al rendimiento de la aplicación. Otra posible complicación podría surgir cuando el número de nodos que conforman el camino hasta el destino es mayor que el número de nodos máximo que se pueden guardar en memoria, ya que el algoritmo sería incapaz de encontrar una solución. Siendo así un algoritmo que solo es óptimo y completo si la memoria asignada es lo suficientemente elevada.

Después de valorar las diferentes opciones descritas se optó por implementar el algoritmo A*, ya que era la opción que mayor flexibilidad nos ofrecía, también era la opción que consideramos que tenía un mayor margen para realizar las modificaciones necesarias para adaptar su funcionamiento a las necesidades de nuestro proyecto, éstas modificaciones están descritas en detalle en el apartado 6.2.3.

Una de las características de este algoritmo que más atractiva nos pareció fue la posibilidad de obtener soluciones sin la necesidad de explorar la totalidad de las rutas posibles, permitiéndonos obtener resultados en tiempos de ejecución más cortos.

Otra de las ventajas que nos ofrecía este algoritmo era un mayor control de la memoria utilizada en el proceso de búsqueda, ya que el uso de memoria del algoritmo A* es mucho menor en el cálculo de rutas relativamente cortas, mientras que el uso de memoria de algoritmos como Dijstra es indistinto a la longitud de las rutas calculadas. Esta característica nos resultó sumamente interesante ya que nos facilitaba la coexistencia de diferentes procesos de búsqueda en un mismo servidor.

Aunque algunas de las características del algoritmo SMA* nos resultaran atractivas, las posibles consecuencias negativas que éstas tenían en el rendimiento de la aplicación, junto al aumento de la complejidad del código respecto al algoritmo A* nos hicieron decantarnos por este último.

6.2.2. Implementación del algoritmo de búsqueda

Al ser el algoritmo A* un algoritmo heurístico, uno de los elementos más importantes que teníamos que determinar era la formula heurística que se iba a utilizar, ya que la calidad de esta heurística influiría directamente en la complejidad computacional de nuestro algoritmo, y por lo tanto en el rendimiento que ofreciera nuestra aplicación. Para nuestra aplicación se encontraron dos posibles formulas heurísticas:

- **Heurística del camino más corto** en base a esta heurística la puntuación de cada nodo se calcula mediante la siguiente formula *f(nodo actual)= distancia recorrida hasta llegar al nodo actual + distancia en linea recta desde el nodo actual al nodo destino*
- **Heurística del camino más rápido** en esta formula heurística la puntuación de cada nodo se calcula mediante la formula *f(nodo actual)= (distancia recorrida hasta llegar al nodo actual / velocidad máxima media del camino recorrido) + (distancia en linea recta desde el nodo actual al nodo destino/ velocidad media del conjunto de vías del sistema)*

Debido a la naturaleza de estas dos heurísticas se comprobó que en la mayoría de los escenarios la heurística del camino más corto expande todos los posibles caminos viables de una forma más equitativa que la otra alternativa. Por lo que esta opción tarda más en encontrar su primera solución, pero a cambio en los casos en los que existe más de un solución, que son los casos en los que cobra relevancia nuestra aplicación, una vez encontrado el primer camino los siguientes caminos relevantes se encuentran en un margen de tiempo mucho menor que la otra heurística.

Por otra parte, la heurística del camino más rápido por lo general suele expandir los posibles caminos dando prioridad a uno concreto, generalmente un camino que transcurre por una vía de alta velocidad, y expandiéndolo hasta llegar al destino, para luego pasar a expandir otro posible camino. Debido a este comportamiento, el primer resultado se encuentra en poco tiempo, si lo comparamos con la heurística del camino más corto, pero el cálculo de las siguientes rutas se dilata más en el tiempo que en su contra-parte.

Una vez examinadas ambas heurísticas nos fue complicado compararlas, debido a la gran variedad de posibles escenarios que se podrían presentar en nuestra aplicación y

al hecho que de media los tiempos que tardaban ambas heurísticas en encontrar un número relevante de soluciones era equiparable, y confluye aún más conforme aumenta el número de soluciones posibles. Debido a estas observaciones, y aunque en el caso de que solamente se buscara una solución, la heurística del camino más rápido es indudablemente mejor, para nuestra aplicación, en la que se busca obtener varios resultados de una forma eficiente nos ha sido imposible decantarnos por una heurística en concreto, por lo que se ha decidido mantener ambas y permitir al usuario decidir cual de las dos prefiere usar en cada búsqueda.

Figura 6.5: Comparación de rendimiento de las Heurísticas en el caso concreto de la ruta Sitges-Castelldefels

Decididas las heurísticas que se iban a utilizar, la implementación del proyecto continuó según nuestro diagrama de clases inicial. El único contratiempo que nos impedía tener nuestro algoritmo completamente funcional, aunque a falta de algunas modificaciones que nos permitieran buscar múltiples resultados, fue que un gran número de nodos que hacían referencia a núcleos urbanos estaban inconexos con el resto del grafo formado por nuestra base de datos. Este contratiempo nos impedía buscar rutas entre las poblaciones representadas por estos nodos, ya que dichos nodos se usaban como origen y destino de nuestras rutas. Este incidente se solucionó desarrollando un método que dado un centro urbano busca los nodos próximos a éste, concretamente a menos de trescientos metros, y preferentemente pertenecientes a vías prioritarias como avenidas, y los añadiera a la lista de nodos disponibles para ser expandidos. De esta forma se subsanó el problema que nos imposibilitaba realizar búsquedas con ciertas poblaciones como origen de la ruta.

Al mismo tiempo se modificó la condición para establecer que una ruta había llegado a su destino, estableciendo como condición que el nodo actual estuviera a menos de trescientos metros del centro urbano de la ciudad de destino. De esta forma no solo se corrigió este problema sino que se mejoró ligeramente el rendimiento de la aplicación, ya que en la mayoría casos el centro urbano de la ciudad se encontraba en el casco antiguo de la ciudad, y lidiar con este tipo de geografía podría ser tedioso para nuestro algoritmo, ya que se trata de calles muy cortas, muchas veces sin salida o con gran variedad de intersecciones.

6.2.3. Modificación del algoritmo de búsqueda

Uno de las principales características que deseábamos en nuestra aplicación era la variedad de posibilidades que queríamos que nos ofrecieran sus soluciones, por lo que el hecho de que el diseño original del algoritmo A* nos ofreciera una única solución interfería gravemente con nuestros objetivos. Así que una de las primeras modificaciones de se realizaron en el algoritmo fue la modificación de su condición de terminación, es decir cuando el algoritmo se detenía porque consideraba que ya había cumplido con su cometido.

Las nuevas condiciones de terminación que se introdujeron fueron las siguientes: que no haya ningún camino por explorar que sea más barato que la solución más barata encontrada y que no haya ningún camino más rápido o corto que la solución más rápida o corta encontrada, o que no haya ningún nodo por expandir que sea conexo con el nodo origen. De esta forma se convirtió la anterior condición de terminación, que el nodo actual sea el nodo destino, en la condición para añadir el camino actual a la lista de soluciones, aunque antes de ser añadida definitivamente las posibles soluciones se comparan entre ellas para comprobar que no haya una solución mejor que otra, y en el caso de que dos soluciones comparables entre ellas se eliminara la peor ruta de la lista.

Estos cambios en las condiciones de terminación extendieron considerablemente el tiempo máximo de ejecución de cada búsqueda en nuestra aplicación, aunque el tiempo a partir del cual empiezan a encontrarse resultados era el mismo. Este factor aumentaba el riesgo de que algunas de las peticiones de los clientes nunca fueran atendidas, debido a que ya hubiera una petición previa que estuviera consumiendo recursos y que tardara demasiado tiempo en liberarlos. Este bloqueo podría ocurrir debido a que al recibir una petición se crea un nuevo proceso que inicia una búsqueda, esta búsqueda en caso de rutas largas como podrían ser «A Coruña-Murcia» tendrían un tiempo de ejecución máximo muy elevado, de incluso días, ya que se explorarían gran parte de las carreteras del territorio español en todas sus combinaciones posibles hasta cumplir con las condiciones de terminación.

Otro agravante a este problema era que el proceso creado por una petición de un usuario no se detenía, ni aunque el usuario cerrara su navegador web, hasta que se cumplieran las condiciones de terminación, por lo que un número pequeño de usuarios podrían saturar nuestro servidor por días sin ni siquiera pretenderlo. Para evitar estos bloqueos se fijó un tiempo máximo de vida de los procesos de búsqueda después del cual éstos se suspenderían. También se han implementado las funcionalidades necesarias para permitir al usuario detener sus búsquedas cuando éstas ya han obtenido los resultados deseados. Aunque estas características no eliminan el riesgo de bloqueo de los servidores, sí que limitan el tiempo de espera máximo que los usuarios tendrán que esperar para realizar su búsqueda en caso de que no haya recursos disponibles.

Para intentar disminuir el consumo de memoria de nuestra aplicación, también se ha modificado como se almacenan los caminos recorridos hasta cada nodo, ya que según el diseño más común del algoritmo A* al expandir un nodo se guarda en memoria el camino completo hasta llegar a este nodo. Esta característica en nuestra aplicación aún aumentaba más si cabe el consumo de memoria, ya que al haber modificado nuestro algoritmo para obtener múltiples soluciones, también se guardaban múltiples caminos para cada nodo. Para disminuir el consumo de memoria de nuestro algoritmo se ha hecho uso clase de la «Score», en la cual se guarda la puntuación de un nodo expandido por un camino concreto, y se ha modificado la forma en la que se guardan los caminos recorridos para que solo se guarde la referencia del nodo actual expandido, junto con el último fragmento de camino por el que se ha expandido y el nodo origen de este fragmento. De esta forma por cada «Score» creada, es decir por cada nodo expandido de una forma

concreta, solo se guardará en memoria un elemento «Way», que hace referencia al último fragmento de camino expandido, y dos elementos del tipo «Node», mientras que de la forma convencional se habrían guardado varias instancias de cada tipo de elemento.

Adicionalmente también se ha desarrollado un nuevo método necesario para el correcto funcionamiento de nuestro algoritmo. Ya que al encontrarse ahora los caminos recorridos hasta el nodo destino divididos en pequeños fragmentos, es necesario un método que realice el recorrido inverso del algoritmo de búsqueda, recorriendo los nodos que nuestra estructura de datos nos indique que se han expandido hasta llegar al destino y reconstruyendo el camino recorrido. Este método se ejecutará cada vez que se encuentre una nueva solución, su coste es lineal y su talla es el número de nodos que componen la solución encontrada, por lo que su impacto en el rendimiento de la aplicación será ínfimo.

La última modificación que se realizó en el algoritmo fue la implementación de una lista de prioridad, para así mejorar el rendimiento de la aplicación cuando se realiza una búsqueda que transcurriera en un espacio urbano extenso. Esta mejora se consideró necesaria ya que al analizar el comportamiento de nuestro algoritmo de búsqueda se observó que éste empleaba demasiado tiempo transitando e intentando buscar nuevas rutas por pequeñas calles dentro de grandes ciudades. Considerando que estos tramos de rutas urbanas tienen menor importancia en nuestra aplicación que los tramos de rutas interurbanas, que nos resultan de mayor interés debido a que es en estos tramos donde aparecen los peajes y donde las diferentes alternativas cobran mayor relevancia.

Por estos motivos se implementó una lista de prioridad, en la que se añadirían los nodos expandidos a partir de vías consideradas como prioritarias, como avenidas, calles principales, carreteras interurbanas, etc. Mientras, el resto de nodos expandidos a partir de vías consideradas no prioritarias se añadirían a una lista de baja prioridad, siendo eliminados dichos nodos de las listas en las que se encuentren al ser expandidos. De esta forma se expanden los nodos normalmente hasta encontrar una vía prioritaria, expandiendo los nodos que componen la vía hasta explorar todas las opciones que esta nos ofrece, o lo que es lo mismo hasta que la lista prioritaria quede vacía, empezando a explorar nuevamente los nodos ubicados en la lista de baja prioridad.

Por último, también se añadió la característica de que los nodos que se encuentren a menos de una distancia determinada del destino, concretamente a menos de seiscientos metros, también se añadirían a esta lista de prioridad. Favoreciendo la completa exploración de las opciones más expandidas y reduciendo el tiempo medio de cálculo de los primeros resultados.

6.3 Implementación de los controladores y la interfaz

Al tener nuestra página web un diseño de *Single-page Application* [26] en la que principalmente se muestra una página web de con una vista base, en la cual se van añadiendo o ocultando elementos dinámicamente, solamente nos fue necesaria la implementación de un controlador que gestionara las funcionalidades de estos elementos.

El controlador es el encargado de suministrar la información que necesitan los diferentes elementos que se nos mostrarán por pantalla a las vistas correspondientes. Un ejemplo que ilustra esta funcionalidad de forma clara en nuestra aplicación son los dos elementos *dropbox*, encargados de mostrarnos todas las poblaciones disponibles en nuestra base de datos. Esta funcionalidad es posible ya que antes de que estos dos elementos sean *renderizados*, nuestro controlador ha pedido a los modelos correspondientes la in-

formación de estas poblaciones y la ha transmitido a la vista correspondiente para que puedan ser mostradas.

```
77      def load_groups
78          @grouped_options = [["Ciudades",Node.cities.order_by_name.collect{ |c| [c.point.name, c.id] }],
79                                          ["Pueblos", Node.towns.order_by_name.collect{ |t| [t.point.name, t.id ] }] ]
80      end
```

Figura 6.6: Fragmento de código del controlador encargado de recuperar la lista de poblaciones existentes en la base de datos

Otra de las funcionalidades de las que se encarga nuestro controlador es de gestionar las peticiones que el usuario nos hace llegar a partir de los diferentes elementos de la interfaz web. La gestión de estos eventos nos permite realizar las comprobaciones que consideremos oportunas y gestionarlos en función de los resultados obtenidos. Verbigracia, cuando nos llega una petición de buscar una ruta se comprueba, entre otras comprobaciones, que la población de origen y la de destino no son la misma población, esta es una comprobación simple, pero nos permite informar inmediatamente por interfaz de este error y se evita iniciar un proceso de búsqueda cuyos resultados podrían ser imprevisibles.

```
77      def load_groups
78          @grouped_options = [["Ciudades",Node.cities.order_by_name.collect{ |c| [c.point.name, c.id] }],
79                                          ["Pueblos", Node.towns.order_by_name.collect{ |t| [t.point.name, t.id ] }] ]
80      end
```

Figura 6.7: Fragmento de código del controlador encargado de recuperar la lista de poblaciones existentes en la base de datos

Uno de los elementos fundamentales en nuestra interfaz son las instrucciones JQuery [17], estas instrucciones nos permiten seleccionar diferentes elementos de nuestra vista en tiempo de ejecución y operar con ellos de múltiples formas, modificando su comportamiento o su apariencia. JQuery [17] además nos facilita la implementación de nuestro diseño *Single-page Application* [26], ya que nos facilita la creación de llamadas AJAX [18] permitiéndonos la actualización de secciones enteras de nuestra vista sin la necesidad de refrescar la página completa.

La utilización JQuery[17] en RoR[11] es muy simple, ya que solamente es necesaria la creación de un nuevo archivo con la terminación «js.erb» y el nombre de la acción del controlador con la que queremos que se ejecute. Utilizando el ejemplo anterior, si un usuario inicia una petición en la que la ciudad origen y la ciudad de destino resultan ser la misma, el controlador ejecutará la acción «load error», que a su vez ejecuta las ordenes de JQuery[17] que aparecen en la figura 6.6 y que aparecen en el archivo «load-error.js.erb» refrescando la vista parcial encargada de mostrar los errores en las peticiones al usuario.

```
1   $('#error_div').html('<%= j render partial: "error" %>');
```

Figura 6.8: Instrucción JQuery contenida en el archivo load-error.js.erb

Otra característica importante de nuestra interfaz es el mapa interactivo que esta incluye. Este mapa se ha conseguido implementar gracias al uso de la biblioteca Leaflet [20]. El funcionamiento del mapa es el siguiente, cuando se hace *click* sobre una ruta en la tabla de resultados de nuestra interfaz, el mapa elimina la ruta que estuviera mostrando anteriormente, representa la nueva ruta y por último centra el mapa y modifica el *zoom* de éste de forma que se pueda ver la ruta completa sin que ningún elemento de la ruta quede fuera de la imagen.

El comportamiento descrito se implementó en CoffeeScript [15], y la mayor dificultad que se obtuvo fue la obtención de los datos en formato GeoJSON [19] de las rutas que se

querían representar. Esta característica presentó dificultades durante su implementación, ya que era necesario establecer una comunicación entre nuestras instrucciones CoffeeScript [15] con los modelos, para así obtener los datos GeoJSON [19]. Este tipo de comunicaciones está altamente desaconsejado, ya que suponen una grave brecha de seguridad en nuestra aplicación, permitiendo la ejecución de código de terceros y la potencial infección de nuestros sistemas y los de nuestros clientes con software malintencionado.

Dado que esta era una praxis que se quería evitar se recurrió a otra solución más segura, la cual consiste en una vez encontrada una ruta y al mismo tiempo que ésta es representada en la tabla de resultados, nuestra interfaz también crea un nuevo elemento HTML [12], el cual no se muestra visualmente en nuestra interfaz, y que contiene los datos GeoJSON [19] convertidos a texto plano. Una vez estos elementos existen en nuestras vistas podemos acceder a ellos mediante una instrucción JQuery[17] para posteriormente reconvertirlos a formato GeoJSON [19] y así poder representarlos.

```
$(document).on 'click', '[id*="path-button-"]', ->
    name = "path-lines-"+this.id.substring(12)
    id_filter = '[id*="'+name+'"]'
    mymap = $('#map')
    if mymap?
        $(".leaflet-clickable").remove()
        origen_coords = $("#origen_geojson").val()
        destino_coords = $("#destino_geojson").val()
        map.fitBounds([ origen_coords.split(','), destino_coords.split(',') ])
        myLayer = L.geoJson().addTo(map)
        for line_div in $(id_filter)
            line = JSON.parse(line_div.value)
            myLayer.addData(line)
```

Figura 6.9: Fragmento del código en lenguaje CoffeeScript encargado de representar elementos en el mapa

Por último, otra característica destacable de nuestra interfaz es la implementación del diseño *Responsive design* [27], el cual permite que nuestra interfaz se adapte a todo tipo de pantallas y resoluciones. Esta funcionalidad se consigue principalmente modificando nuestras definiciones de estilos en los archivos CSS [13] y substituyendo en los casos en los que sea necesario los tamaños de los elementos HTML [12] que estén en valores absolutos por valores que hagan referencia a porcentajes de tamaño de los elementos que los contienen, o a porcentajes del tamaño de la ventana del navegador web. También será necesario definir el tamaño mínimo y máximo que estos elementos pueden tener. De esta forma los diferentes elementos que conformen nuestra interfaz se adaptarán mejor a diferentes resoluciones, por lo que obtendremos una interfaz que será utilizable en un mayor número de dispositivos.

En nuestra aplicación la implementación del diseño *Responsive design* se ha visto favorecida por el uso de la biblioteca Bootstrap [14], ya que la mayoría de elementos que esta librería nos ofrece ya están preparados para su uso en este tipo de diseños. Además, esta librería cuenta con sus propias hojas de estilo que facilitan el uso de tamaños en base a porcentajes, por lo que nos centramos en aplicar estos estilos a nuestros elementos HTML [12], reduciendo drásticamente el número de modificaciones a realizar en nuestras hojas de estilos.

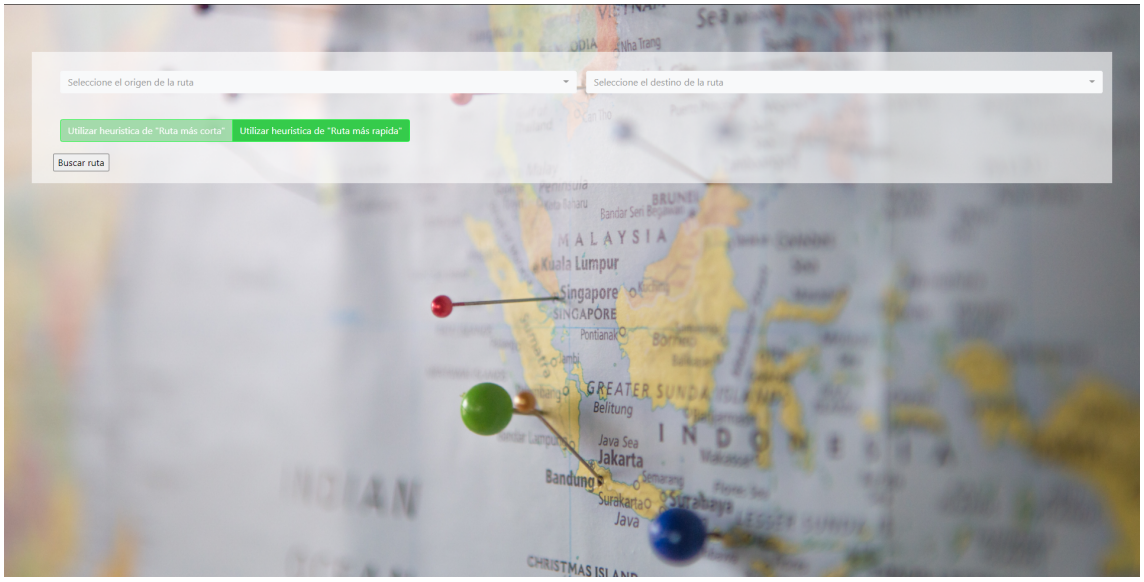

Figura 6.10: Ejemplo de visualización de nuestra aplicación en ventanas anchas

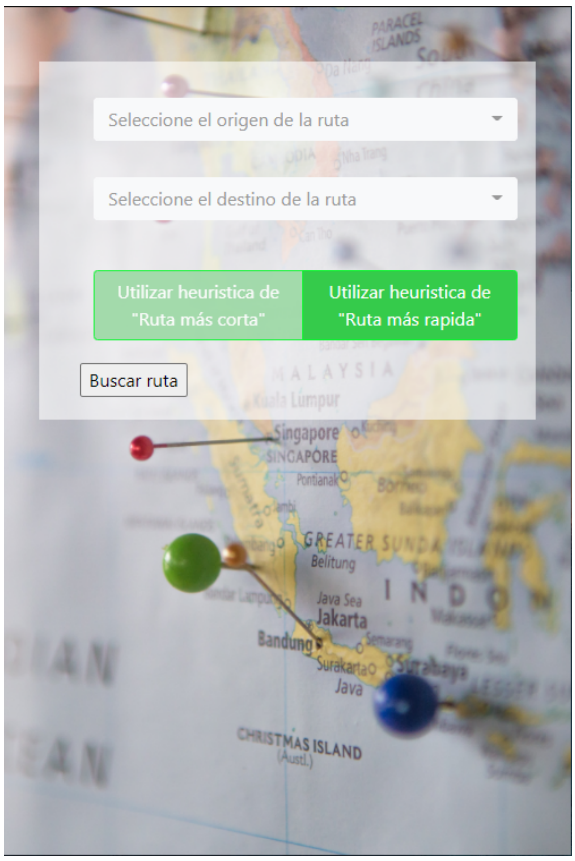

Figura 6.11: Ejemplo de visualización de nuestra aplicación en ventanas estrechas

CAPÍTULO 7
Despliegue

7.1 Configuración del servidor y despliegue

Uno de los objetivos que se querían alcanzar al empezar este proyecto era crear una aplicación que pudiera funcionar en un entorno de producción, así que una vez terminado el desarrollo, el único paso pendiente era desplegar nuestra aplicación en un servidor en producción.

Para desplegar nuestro proyecto hicimos uso de varias herramientas no utilizadas hasta ahora. Una de las herramientas más destacables es Capistrano [31], un *software* de código abierto que nos permite automatizar los despliegues de nuestras aplicaciones de RoR [11]. El proceso de instalación de Capistrano [31] es muy simple gracias al sistema de gemas de RoR [11] donde con simplemente escribir en la consola de nuestro proyecto la instrucción «bundle exec cap install» se instalará esta herramienta y sus dependencias.

Una de las principales ventajas que nos aporta el uso de Capistrano [31], es el alto grado de personalización de esta herramienta, ya que con solamente modificar su archivo de configuración podremos definir el proceso que seguirán todos nuestros despliegues.

Este flujo de despliegue nos permite asegurarnos que la versión que despleguemos en el servidor funcionará según lo esperado, y además, nos facilita enormemente los despliegues, ya que este procedimiento se ejecuta en el servidor deseado con sólo escribir un comando.

En el caso de que se esté realizando el primer despliegue de la aplicación en el servidor, Capistrano [31] se encargará de instalar la versión de Ruby que le indiquemos, utilizando el gestor de versiones que prefiramos. Además se encargará de instalar un gran número de las librerías o recursos de terceros de los que depende nuestra aplicación, gracias una vez más al sistema de gemas de Ruby. En caso de que se quiera desplegar una nueva versión de nuestra aplicación, Capistrano [31] también no resultará de gran ayuda, ya que se encargará de ejecutar las pruebas descritas en nuestra aplicación para asegurar el correcto funcionamiento de ésta, y recuperará la versión anterior en caso de que estas pruebas no se superen, impidiendo que despleguemos por error versiones que contengan defectos. También se encargará de reiniciar el servidor web de nuestro servidor, junto a otras múltiples tareas de mantenimiento, una vez haya finalizado el despliegue, para que así se apliquen los cambios realizados.

En la figura 7.2, podemos observar un fragmento del archivo de configuración de Capistrano [31]. En este fragmento se puede observar como se especifica la dirección de nuestro repositorio de GIT, elemento imprescindible ya que es la fuente de la que se descargará el código de nuestra aplicación en el servidor. Podemos observar como se especifican el número de versiones desplegadas que se guardarán en nuestro servidor, y

```
deploy
  deploy:starting
    [before]
      deploy:ensure_stage
      deploy:set_shared_assets
    deploy:check
  deploy:started
  deploy:updating
    git:create_release
    deploy:symlink:shared
  deploy:updated
    [before]
      deploy:bundle
    [after]
      deploy:migrate
      deploy:compile_assets
      deploy:normalize_assets
  deploy:publishing
    deploy:symlink:release
  deploy:published
  deploy:finishing
    deploy:cleanup
  deploy:finished
    deploy:log_revision
```

Figura 7.1: Estructura del flujo de despliegue de Capistrano

que nos serán útiles en caso de que se despliegue una versión que contenga errores, o que se decida retroceder a una versión anterior por cualquier motivo. También se puede observar como se ha modificado el gestor de versiones de Ruby, cambiando el gestor de versiones por defecto Rbenv [33] por RVM [34]. Por último, en las últimas lineas de código de esta figura podemos observar como se modifica una tarea ya definida por Capistrano [31], aunque la misma estructura nos valdría para crea una nueva tarea que se ejecutará en el flujo anteriormente mencionado.

Otra de las herramientas usadas dignas de mención podría ser Unicorn [35] un servidor web diseñado exclusivamente para aplicaciones RoR [11] que trabajar conjuntamente con Nginx [36], otro servidor web que en nuestro proyecto se encarga de hacer de intermediario entre las peticiones recibidas por los usuarios y Unicorn [35]. Nginx [36] también se encarga de equilibrar la carga de trabajo entre los diferentes trabajadores del servidor Unicorn [35].

Un inconveniente derivado de la actualización de la versión que se encuentra en producción en nuestro servidor viene dado por la necesidad de reiniciar el servidor web de Unicorn [35] una vez ha terminado el despliegue realizado en Capistrano [31]. Este reinicio suele provocar un periodo de tiempo en el que nuestra aplicación no estará disponible para los usuarios, e incluso la cancelación de las búsquedas de rutas que estén teniendo lugar. Este inconveniente se puede solucionar mediante un pequeño ajuste en la configuración de Unicorn [35], consiguiendo lo que se denomina como *Zero Downtime Deployment* [37], es decir un despliegue en el que nuestra aplicación no deja de estar disponible en ningún momento.

Esta característica es posible debido al uso de «trabajadores» que implementa Unicorn [35], siendo un «trabajador» un proceso que se encarga de gestionar las peticiones que le son asignadas, en nuestro caso encontrar rutas entre dos localidades. Para conseguir un

despliegue con *Zero Downtime Deployment* [37] es necesario modificar la forma en la que estos «trabajadores» son creados o eliminados, ya que por defecto el código detenía los «trabajadores» que tenían instanciada la vieja versión de la aplicación, para crear nuevos «trabajadores» que instancian la nueva versión una vez ha terminado el despliegue. La modificación de este código se encarga de mantener activos los «trabajadores» que están ocupados, es decir que están atendiendo peticiones de los clientes, estos trabajadores se irán renovando paulatinamente conforme terminen de gestionar sus peticiones pendientes.

De esta forma es posible realizar un despliegue en el que ninguna petición de ningún usuario es cancelada, y consiguiendo que nuestra aplicación este disponible de forma ininterrumpida. Este tipo de despliegues será posible siempre que el cambio de versión no precise de una migración en la base de datos, en los casos en los que esto ocurra será inevitable interrumpir el funcionamiento habitual de la aplicación mientras se realiza dicha migración.

Una vez configurados los elementos ya mencionados es posible realizar el despliegue mediante Capistrano [31]. Como ya se ha mencionado anteriormente, esta herramienta si esta bien configurada nos facilita muchísimo el trabajo, siendo el único detalle que debemos gestionar la población de nuestra base de datos, ya que aunque Capistrano [31] se encarga de crear la base de datos y crear la estructura de esta, no importará los datos que hemos obtenido mediante el proceso detallado en el apartado 6.1. Por lo que para importar nuestros datos deberemos hacer una copia de nuestra base de datos e importarla a la base de datos de nuestro servidor. Esta tarea es sumamente sencilla si se conoce el funcionamiento de PostgreSQL [21], ya que su interfaz visual nos permite realizar esta operación en muy pocos pasos, y también la podremos realizar por la línea de comandos si así lo preferimos con una instrucción del tipo «pg restore» [38].

7.2 Resultado final y problemas surgidos en el despliegue

Como se puede ver en la figura 7.4 la aplicación desplegada es capaz de realizar búsquedas y proporcionar varias soluciones a las rutas solicitadas, siempre que estas soluciones existan. Las soluciones encontradas se muestran con un fondo negro en la tabla izquierda de la figura, donde también se observan algunas de sus características principales. Además, una vez pulsamos en una de estas rutas, la información de ésta se expande mostrando los diferentes tramos que la forman, como podemos observar en los elementos con el fondo blanco en la tabla. Al pulsar en una de las rutas, ésta también se muestra en el mapa visible a la derecha de la captura, mapa con el que podemos interactuar haciendo *zoom* o desplazando el centro de éste a nuestro antojo.

El principal problema que hemos tenido en el despliegue, ha venido dado por lo limitados que eran los recursos del servidor en el que se ha desplegado la aplicación, haciendo imposible que éste alojara nuestra base de datos completa, por lo que solo se ha desplegado un fragmento muy reducido de esta base de datos, reduciendo enormemente las poblaciones entre las que se pueden buscar rutas y el abanico de posibles opciones calculadas. Pero considerando que este contratiempo se podría solucionar con un servidor con mayores prestaciones, ya que tanto el código fuente de nuestra aplicación como la estructura de la base de datos serían idénticos a las utilizados en un servidor con mayores prestaciones. Por lo tanto valoramos que el despliegue ha sido exitoso.

```ruby
set :rails_env, fetch(:stage)
set :rvm1_ruby_version, '2.3.2'

set :application, 'tfg'
set :full_app_name, deploysecret(:full_app_name)

set :server_name, deploysecret(:server_name)
set :repo_url, 'https://bitbucket.org/carles_bataller/tfg.git'

set :revision, `git rev-parse --short #{fetch(:branch)}`.strip

set :log_level, :info
set :pty, true
set :use_sudo, false

set :linked_files, %w{config/database.yml config/secrets.yml}
set :linked_dirs, %w{log tmp public/system public/assets}

set :keep_releases, 5

set :local_user, ENV['USER']

set :delayed_job_workers, 2
set :delayed_job_roles, :background

set(:config_files, %w(
  log_rotation
  database.yml
  secrets.yml
  unicorn.rb
))

set :whenever_roles, -> { :app }

namespace :deploy do
  before :starting, 'rvm1:install:rvm'  # install/update RVM
  before :starting, 'rvm1:install:ruby' # install Ruby and create gemset
  before :starting, 'install_bundler_gem' # install bundler gem

  after :publishing, 'deploy:restart'
  after :published, 'delayed_job:restart'
  after :published, 'refresh_sitemap'

  after :finishing, 'deploy:cleanup'
end

task :install_bundler_gem do
  on roles(:app) do
    execute "rvm use #{fetch(:rvm1_ruby_version)}; gem install bundler"
  end
end
```

Figura 7.2: Fragmento del archivo de configuración de Capistrano

```ruby
old_pid = "#{server.config[:pid]}.oldbin"
if File.exists?(old_pid) && server.pid != old_pid
  begin
    Process.kill("QUIT", File.read(old_pid).to_i)
  rescue Errno::ENOENT, Errno::ESRCH
    # someone else did our job for us
  end
end
```

Figura 7.3: Fragmento de la configuración de Unicorn que hace posible el despliegue *Zero-Downtime*

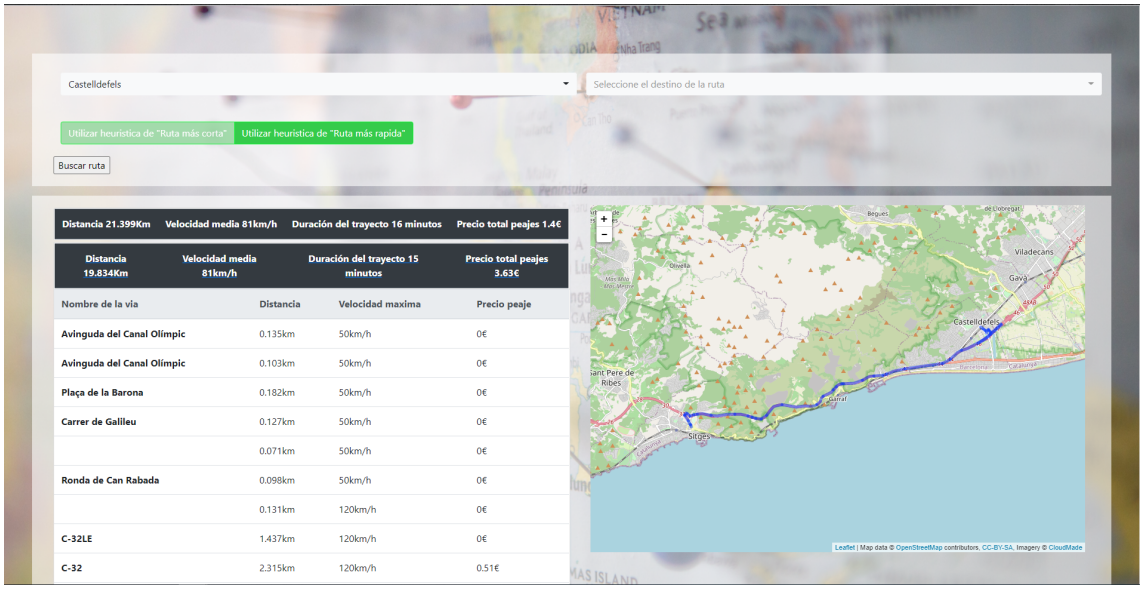

Figura 7.4: Captura de la aplicación en un servidor en producción

CAPÍTULO 8
Pruebas

La realización de pruebas es una tarea imprescindible en el desarrollo de cualquier aplicación, ya que estas pruebas nos ayudarán a identificar los posibles defectos de nuestro producto y nos facilitarán la tarea de corregirlos. Es conveniente que las pruebas realizadas cubran el mayor porcentaje del código desarrollado posible, o en su defecto el mayor número de funcionalidades posible.

Para facilitar la realización de las pruebas se ha utilizado la herramienta RSpec [40]. Esta tecnología se especializa en el testeo de aplicaciones desarrolladas en el lenguaje Ruby, y nos ofrece las herramientas necesarias para realizar las pruebas de los diferentes módulos que componen nuestra aplicación, además también nos facilita la organización de las pruebas realizadas y la ejecución de las mismas. Otra característica de RSpec [40] es que nos aporta una sintaxis diferente para los diferentes tipos de pruebas que vayamos a realizar, por lo que si observamos detalladamente los test de dos tipos de pruebas distintas, por ejemplo las pruebas de la interfaz y las de los modelos, podremos observar diferencias en cómo se realizan éstas, aunque en ambos casos la estructura que siguen es similar.

También se ha hecho uso de la herramienta FactoryBot [41], la cual nos facilita la creación de los objetos con los que interactuar y realizar nuestras pruebas. Esta funcionalidad es posible mediante el uso de clases fábrica que nos facilitan la creación de objetos genéricos de una clase determinada y con unos parámetros determinados, reduciendo el número de líneas de código necesarias para realizar la mayoría de pruebas.

En esta aplicación se han realizado diferentes tipos de pruebas, las cuales van dirigidas a probar aspectos concretos de nuestra aplicación. La pruebas realizadas han sido las siguientes:

- **Pruebas unitarias** Las pruebas unitarias van dirigidas a comprobar el funcionamiento individual de los elementos que forman nuestra aplicación. En nuestra aplicación se han dividido las pruebas unitarias en tres apartados según la naturaleza de estos componentes.

    1. **Pruebas de interfaz** La realización de este tipo de pruebas de forma correcta puede llegar a tener un alto grado de dificultad, debido a lo complejo que resulta imitar mediante código la interacción del usuario con los diferentes elementos de la interfaz. Por lo que para realizar estas pruebas se ha optado por comprobar que los diferentes elementos de la interfaz se muestren cuando el estado de la aplicación sea el indicado, comprobando que los atributos de estos elementos sean los correctos. Un ejemplo de este tipo de pruebas podría ser la comprobación de que se muestra *feedback* al usuario mediante el enunciado correspondiente cuando éste inicia la búsqueda de una ruta.

```
it 'Display "Loading" div when a search is started' do

  render template: "main/_loading.html.erb",
         locals: { cookies['rendered_status'] == "loading"}

  within '.d-flex' do
    expect(page).to have_selector(".d-flex > strong:nth-child(1)")
  end

end
```

Figura 8.1: Fragmento de test de interfaz encargado de comprobar que se notifica al usuario al iniciar la búsqueda de una ruta

2. **Pruebas de controladores** Este tipo de pruebas son las encargada de testear las funcionalidades propias de los controladores. Más concretamente en las pruebas realizadas se ha comprobado que existen las rutas asociada a cada método del controlador y que éstos responden a un tipo de llamadas HTML [12] concreto, comprobando que esta respuesta cumple con los parámetros requeridos. Este tipo de pruebas es equivalente a las pruebas que podríamos realizar mediante software como Postman [42], SoapUI [43], etc, ya que este tipo de software nos permite realizar peticiones HTML [12] a nuestra aplicación y analizar sus respuestas.

```
it 'load_error returns empty alert when no error is given' do
  get :load_error, alert: nil
  expect(response).to be_ok
  expect(Ahoy::Event.where(:alert.nil?)).to eq true
end
```

Figura 8.2: Fragmento de test de controlador del método encargado de gestionar los mensajes de error cuando se genera una búsqueda de rutas con parámetros inválidos

3. **Pruebas de modelos** Este tipo de pruebas son las encargadas de verificar el correcto funcionamiento de los métodos implementados en las clases de nuestra capa de negocio. La correcta realización de estas pruebas es de vital importancia para asegurar la fiabilidad de los resultados obtenidos en nuestra aplicación, ya que es en esta capa donde se realizan la mayoría de cálculos algorítmicos de nuestra aplicación y cualquier defecto podría alterar el correcto funcionamiento de éstos.

```
it "returns distance in km" do
    origen_node = create(:node, lat: 0.0, lon: 0.0 )
    destino_node = create(:node, lat: 1.0, lon: 1.0 )
    expected_result = 157.40

    expect(origen_node.distance(destino_node).to eq expected_result
end
```

Figura 8.3: Fragmento de test del modelo «Node» que se encarga de probar el correcto funcionamiento del método que calcula la distancia entre dos nodos

- **Pruebas Funcionales** Las pruebas funcionales se encargan de probar funcionalidades concretas de nuestra aplicación. Cada funcionalidad puede afectar a varios módulos de nuestro proyecto, por lo que estas pruebas comprueban el funcionamiento general de la aplicación y la correcta integración entre los diferentes elementos de ésta. En la figura 8.4 se muestra una de las pruebas realizadas para la funcionalidad de «Buscar rutas», el código muestra como se crean los elementos necesarios para

realizar la prueba, y comprueba que al iniciar una búsqueda mediante la interfaz y la aplicación encuentra un resultado, éste se muestra al usuario.

```
scenario 'Search ends and show results' do
  node_origen = create(:node)
  point_origen = create(:point, node_origen)
  node_destino = create(:node)
  point_destino = create(:point, node_destino)
  way_aux = create(:way, node_origen, node_destino)
  line = create(:line, way_aux )

  visit index_path

  fill_in "origin_selector", with: point_origen.name
  fill_in "destiny_selector", with: point_destino.name

  click_button 'Buscar ruta'

  sleep 20

  expect(page).to have_current_path(index_path)
  expect(page).to have_content "accordionResults"
end
```

Figura 8.4: Fragmento de test de la funcionalidad «Buscar ruta»

CAPÍTULO 9
Conclusiones

Tal como se ha descrito en esta memoria, al realizar este proyecto se fijaron unos objetivos a los que se deseaba llegar.

Entre estos objetivos, podemos identificar como objetivo principal la creación de una herramienta capaz de trabajar con datos reales que nos permita buscar rutas entre distintas ubicaciones y que ésta nos propusiera un amplio abanico de soluciones, no comparables directamente entre ellas, entre las que elegir.

También se identificaron otros objetivos como que la aplicación desarrollada fuera compatible con todo tipo de dispositivos o que la aplicación estuviera lista para su despliegue en producción. Gracias al trabajo realizado descrito a lo largo de este documento, y aunque desafortunadamente no ha sido posible el despliegue de la aplicación con una versión completa de la base de datos obtenida, se puede considerar que todos estos objetivos se han completado satisfactoriamente, aunque esto no significa que no haya espacio para posibles mejoras.

9.1 Problemas, dificultades y errores cometidos.

Como ya se ha descrito anteriormente uno de los problemas surgidos durante el desarrollo de este proyecto fue provocado por la falta de recursos, ya que no disponer de un servidor con las suficientes prestaciones nos impidió el despliegue de nuestra aplicación con una versión completa de nuestra base de datos, por lo que la versión completa solamente es accesible en nuestro equipo local.

Una de las mayores dificultades que ha supuesto este proyecto ha sido trabajar con una cantidad tan elevada de datos, ya que entre otras cosas esto ha ralentizado considerablemente el desarrollo de la aplicación, debido a que el tiempo total necesario para el procesado de los diferentes pasos realizados hasta obtener una base de datos funcional se podría contar en semanas. Otra de las complicaciones derivadas de el uso de una cantidad tan grande de datos fue la dificultad a la hora de identificar que casos producían un error en nuestro algoritmo, ya que intentar comprender porque un elemento concreto producía un error con un número tan alto de variables era un tarea complicada.

Por último, uno de los errores que se considera que se han cometido durante este desarrollo ha sido no implementar el *backend* como un elemento propio y desarrollarlo de esta forma como una API independiente, desligándolo del *frontend*, ya que esto habría facilitado la gestión de las peticiones de los usuarios y habría propiciado la implementación de un sistema de colas para la ejecución de estas peticiones.

9.2 Relación con los estudios cursados

Entre los conocimientos adquiridos en la titulación que nos han facilitado la realización del proyecto cabría destacar la introducción al uso de bases de datos relacionales, materia explicada en la asignatura «Bases de Datos», también sería destacable la formación recibida en las materias de algorítmica y teoría de grafos, materias que se estudiaron en las asignaturas «Matemática Discreta» y «Estructuras de Datos y Algoritmos».

Aunque también se podrían destacar algunos conocimientos adquiridos de forma autónoma que quizá habría sido interesante adquirir en esta titulación, como el funcionamiento base de un *framework* MVC, una introducción al uso del lenguaje CSS o el funcionamiento de un servidor web.

CAPÍTULO 10
Trabajo futuro

A pesar de que el resultado general es satisfactorio, hay algunas características que se podrían mejorar, ya sea mediante el desarrollo de nuevas funcionalidades o mediante la mejora de las ya desarrolladas. A continuación se listan algunas de las propuestas que se han considerado de mayor interés:

- Optimización de los tiempos de búsqueda de rutas, aunque nuestro algoritmo de búsqueda es funcional, éste siempre se podría mejorar para así mejorar el rendimiento general de nuestra aplicación.

- Implementación de una cola de peticiones, una de las posibles mejoras más interesantes podría ser la creación de una cola de peticiones. En esta cola se almacenarían las peticiones que no puedan ser atendidas en el momento por falta de recursos disponibles, y que se irían satisfaciendo conforme se vayan finalizando las peticiones que consumían dichos recursos. De esta forma se mejoraría el funcionamiento general de la aplicación y se podría ofrecer mayor información a los usuarios sobre cuando se podrían satisfacer sus peticiones.

- Implementación de API pública, otra funcionalidad que podría ser interesante es la posibilidad de aceptar peticiones sin la necesidad del uso de una interfaz, esta característica es posible gracias a la implementación de una API pública a la que otras aplicaciones de terceros podrían realizar peticiones. Esta funcionalidad sería interesante ya que facilitaría la integración de nuestro producto con otras aplicaciones.

- Disociación entre el *frontend* y el *backend*, en nuestra aplicación el *frontend* y el *backend* están diseñadas como dos partes de un mismo módulo, podría ser interesante la implementación de estas dos partes de nuestro software como dos módulos independientes que estuvieran relacionados, ya que entre otras cosas facilitaría la implementación de las dos mejoras anteriores.

- Interfaz de administración. Por último, también sería interesante la creación de una interfaz de administración, en esta interfaz, los usuarios administradores podrían consultar las búsquedas realizadas recientemente por cualquier usuario, los resultados obtenidos, etc. Se considera que esta funcionalidad sería interesante a la hora de optimizar nuestra aplicación, ya que podríamos consultar que peticiones no se han resuelto correctamente o cuales han tardado en resolverse más de lo esperado, lo que nos ofrecería una excelente herramienta para descubrir posibles defectos y subsanarlos.

Finalmente, recalcar que aunque todas estas mejoras podrían resultar atractivas requerirían una gran número de cambios en nuestra aplicación, por lo que sería necesaria una gran cantidad de tiempo para su implementación.

Bibliografía

[1] Google Maps Consultado el 16 de julio de 2020 en https://www.google.es/maps/.

[2] Vía Michelin Consultado el 16 de julio de 2020 en https://www.viamichelin.es/.

[3] DGT Consultado el 16 de julio de 2020 en http://www.dgt.es/es/.

[4] Pagina oficial del sistema Cl@ve Consultado el 16 de julio de 2020 en https://clave.gob.es/clave_Home/clave.html.

[5] Symfony Consultado el 16 de julio de 2020 en https://symfony.es/documentacion/.

[6] Angular Consultado el 16 de julio de 2020 en https://angular.io/.

[7] Spring Consultado el 16 de julio de 2020 en https://spring.io/projects/spring-framework.

[8] Hibernate Consultado el 20 de julio de 2020 en https://hibernate.org/.

[9] Django Consultado el 16 de julio de 2020 en https://www.djangoproject.com/.

[10] No te repitas Foote, Steven (2014). Learning to Program. Addison-Wesley Professional. p. 336. ISBN 9780133795226

[11] Ruby on Rails Consultado el 16 de julio de 2020 en https://rubyonrails.org/.

[12] HyperText Markup Language Consultado el 17 de julio de 2020 en https://developer.mozilla.org/es/docs/Web/HTML.

[13] Cascading Style Sheets Consultado el 17 de julio de 2020 en https://developer.mozilla.org/es/docs/Web/CSS.

[14] Bootstrap Consultado el 17 de julio de 2020 en https://getbootstrap.com/.

[15] CoffeeScript Consultado el 17 de julio de 2020 en https://coffeescript.org/.

[16] Read Write Hack. Interview with Jeremy Ashkenas Wayback Machine 7 de Junio de 2011

[17] JQuery Consultado el 18 de julio de 2020 en https://api.jquery.com/.

[18] AJAX Consultado el 18 de julio de 2020 en https://developer.mozilla.org/es/docs/Web/Guide/AJAX.

[19] GeoJSON Consultado el 18 de julio de 2020 en https://geojson.org/.

[20] Leaflet Consultado el 18 de julio de 2020 en https://leafletjs.com/.

[21] PostgreSQL Consultado el 18 de julio de 2020 en https://www.postgresql.org/.

[22] PostGIS Consultado el 18 de julio de 2020 en https://postgis.net/.

[23] Active Record Consultado el 18 de julio de 2020 en https://guides.rubyonrails.org/active_record_basics.html.

[24] Pagina principal de la organización Open Street Maps Consultado el 20 de julio de 2020 en https://www.openstreetmap.org/.

[25] Pagina del sistema de referencia espacial 4326 en la organización Spartial Reference Consultado el 20 de julio de 2020 en https://spatialreference.org/ref/epsg/4326/.

[26] Pagina de Wikipedia dedicada al diseño *Single-page* Consultado el 20 de julio de 2020 en https://es.wikipedia.org/wiki/Single-page_application.

[27] Pagina de Wikipedia dedicada al diseño adaptativo Consultado el 20 de julio de 2020 en https://en.wikipedia.org/wiki/Responsive_web_design.

[28] Pagina de colaboración de Wikipedia Consultado el 20 de julio de 2020 en https://en.wikipedia.org/wiki/Wikipedia:Contributing_to_Wikipedia.

[29] Pagina de GIT de la herramienta de código abierto osm2pgsql Consultado el 20 de julio de 2020 en https://github.com/openstreetmap/osm2pgsql.

[30] Pagina oficial de Abertis España Consultado el 20 de julio de 2020 en https://www.autopistas.com/tarifas-y-descuentos/tarifas/.

[31] Pagina de GIT de la herramienta de código abierto Capistrano Consultado el 20 de julio de 2020 en https://github.com/capistrano/rails.

[32] Pagina principal de la tecnología GIT Consultado el 20 de julio de 2020 en https://git-scm.com/.

[33] Pagina de Git del gestor de versiones de Ruby, Rbenv Consultado el 20 de julio de 2020 en https://github.com/rbenv/rbenv.

[34] Pagina principal del gestor de versiones de Ruby, RVM Consultado el 20 de julio de 2020 en https://rvm.io/.

[35] Pagina principal servidor web Unicorn Consultado el 20 de julio de 2020 en https://yhbt.net/unicorn/.

[36] Pagina principal servidor web Nginx Consultado el 20 de julio de 2020 en https://www.nginx.com/.

[37] Articulo sobre Zero Downtime Deployment Consultado el 20 de julio de 2020 en https://dzone.com/articles/zero-downtime-deployment.

[38] Documentación del comando pg restore en PostgreSQL Consultado el 20 de julio de 2020 en https://www.postgresql.org/docs/9.2/app-pgrestore.html.

[39] Dirección URL de la aplicación desarrollada en un servidor en Producción Consultado el 20 de julio de 2020 en https://obscure-temple-38046.herokuapp.com/.

[40] Pagina de Git de la herramienta de pruebas RSpec Consultado el 20 de julio de 2020 en https://github.com/rspec/rspec-rails.

[41] Pagina de Git de la herramienta FactoryBot Consultado el 20 de julio de 2020 en https://github.com/thoughtbot/factory_bot.

[42] Pagina oficial de la herramienta Postman Consultado el 20 de julio de 2020 en https://www.postman.com/.

[43] Pagina oficial de la herramienta SoapUI Consultado el 20 de julio de 2020 en https://www.soapui.org/.