E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

# Aplicación GeoWeb de cálculo de rutas sobre cartografía navegable propia con tecnología OpenSource

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alex Salvatierra Vettore

María Aránzazu Jurío Munárriz

Pamplona, 14 de enero de 2023

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

# Índice

# 1. Introducción

En la actualidad, casi toda la sociedad está familiarizada con el uso de tecnologías de la información para tareas cotidianas, como la búsqueda de rutas de viaje o la determinación de tiempos de traslado en coche. Uno de los softwares más utilizados para estas tareas es Google Maps [1], que suele estar preinstalado en dispositivos móviles y vehículos. Además de para uso particular, la planificación de rutas y optimización de tiempos de llegada son herramientas valiosas para profesionales en distintas áreas.

Existen dos tipos de software: propietario y libre (también conocido como open source), que se diferencian en cómo se pueden utilizar y distribuir. El software propietario [2, 3] está protegido por derechos de autor y solo se puede utilizar de acuerdo con los términos establecidos por el desarrollador. Esto a menudo incluye limitaciones en cuanto a cómo se puede utilizar el software, la prohibición de modificación o distribución a terceros, y, a menudo, también impide el acceso al código fuente del software. Además, en algunos casos, es necesario pagar licencias por su uso.

Por otro lado, el software libre [4] se distribuye con una licencia que permite a los usuarios consultar, utilizar, modificar y distribuir el software de manera libre, siempre y cuando se cumplan ciertas condiciones, como citar al autor original y no utilizar el software para fines comerciales sin permiso.

Google Maps [5] es un producto propietario y está sujeto a los términos de uso y políticas de privacidad de Google. Esto significa que al utilizar Google Maps, los usuarios aceptan compartir cierta información, como sus ubicaciones y rutas, con Google y permitir que Google utilice esta información de ciertas maneras, vendiéndola a anunciantes, por ejemplo. Otros ejemplos de software propietario, especialmente los enfocados al sector profesional, requieren de un pago de licencia para su uso.

# 2. Objetivo

Por encargo de la Dirección General de Obras Públicas e Infraestructuras del Gobierno de Navarra, Tracasa Instrumental desarrolló una base de datos de la red de carreteras, caminos y sendas de Navarra. Esta información ofrece una muy alta calidad, con datos mucho más depurados que los de compañías de carácter privado, y constituye una gran inversión pública, que, hoy, ve limitada su explotación debido al software propietario y el coste de sus licencias.

Con el objetivo de explotar esta información sin depender de software propietario, se propone investigar acerca de las aplicaciones de enrutamiento de código abierto sobre mapas propios y desarrollar una aplicación web piloto sobre la mejor alternativa, para, por ejemplo, dar servicio al equipo de emergencias de Navarra, al de acciones medioambientales o extender la funcionalidad del visor web de la Infraestructura de Datos Espaciales de Navarra (IDENA).

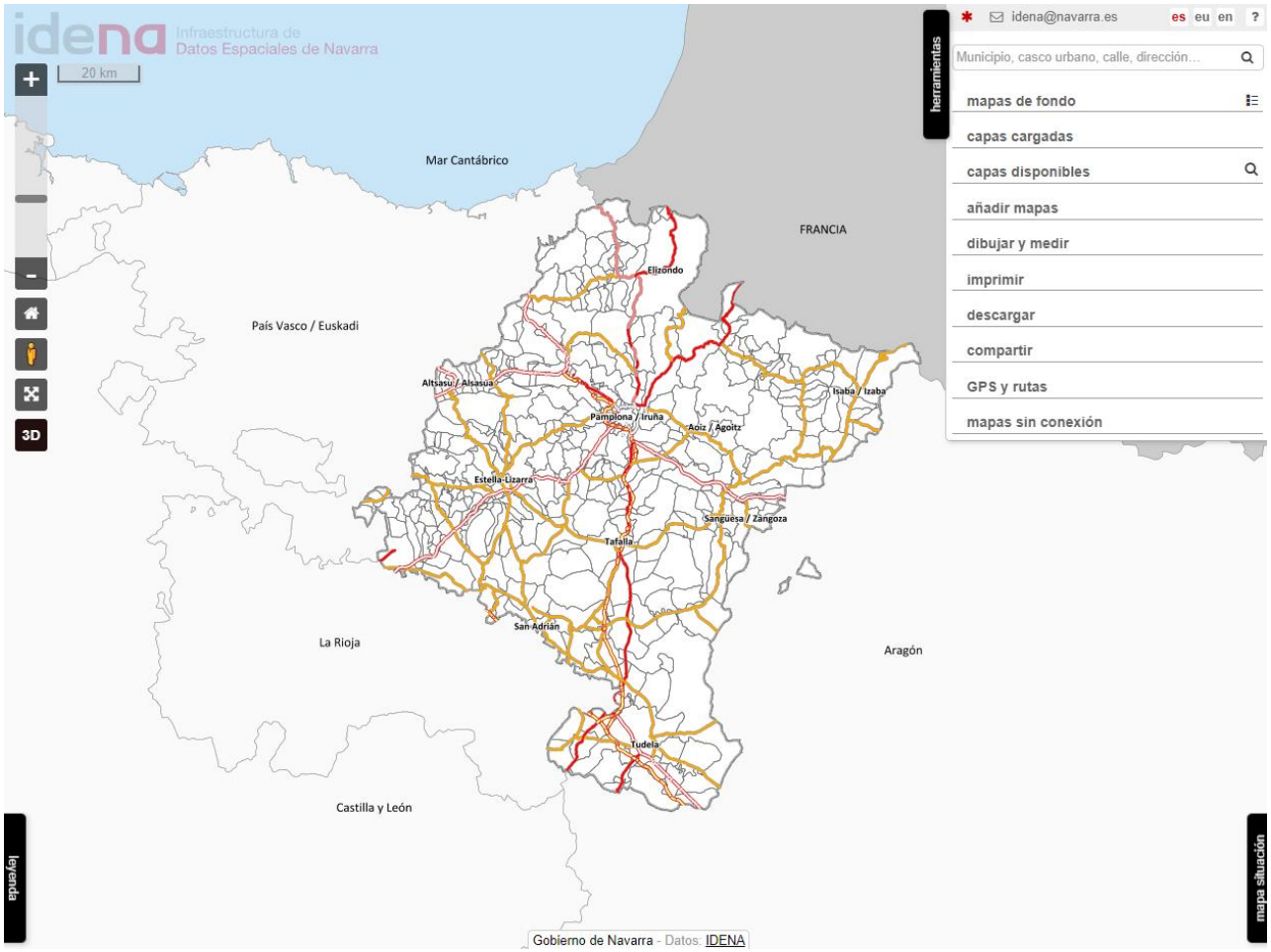



Ilustración 1: Visor web IDENA.

# 3. Datos

Los datos disponibles para este proyecto consisten en toda la cartografía de Navarra. La red de todas sus carreteras, caminos y sendas. Unos datos propios recogidos y elaborados por el equipo de cartografía de Tracasa Instrumental para el Gobierno de Navarra.

Estos datos están almacenados en una base de datos PostgreSQL (o Postgres) [6], un sistema gestor de base de datos relacional orientado a objetos y de código abierto, característica común entre todo el software utilizado.

Están almacenados en diferentes unidades llamadas ejes, en una tabla homónima. Un eje es un tramo de una carretera (o camino o senda) del que se almacena un identificador único, que actuará como clave primaria de la tabla, y diferente información. Entre ella se encuentra:

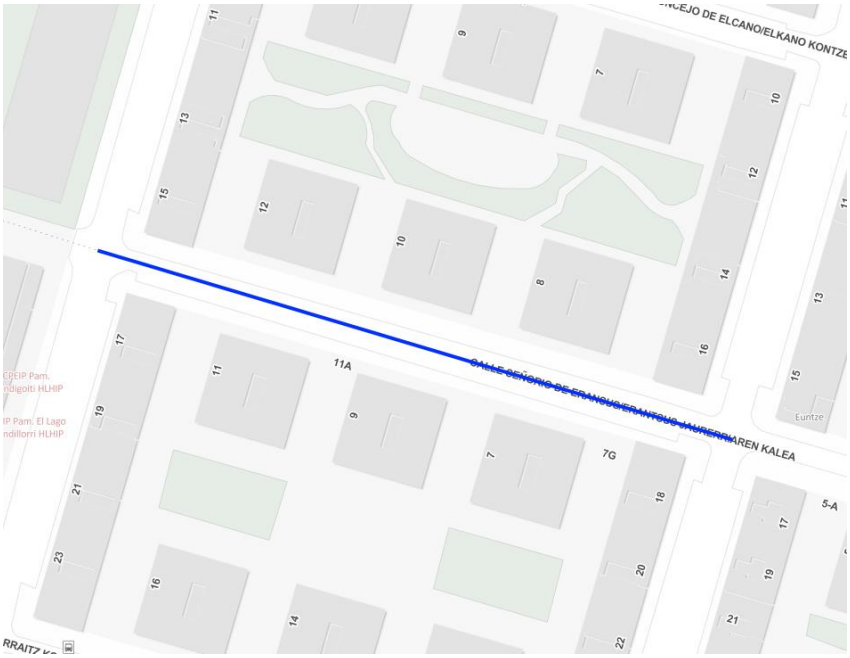



Ilustración 2: En azul: un eje. Mapa base: IDENA

- tipo_via: El tipo de vía. Sus valores son:
  - Vía urbana conexión de carretera
  - Camino
  - Carretera de calzada doble convencional
  - Carretera de calzada doble autopista
  - Vía urbana genérica
  - Carretera de calzada doble autovía
  - Senda
  - Carretera de calzada única área de descanso
  - Conexión entre sentidos de autovías y autopistas

    - Carretera de calzada única
    - Pista

- paviment: Indica si está o no pavimentada. Sus valores son:
    - NPA: Vía no pavimentada
    - PAV: Vía pavimentada

- velocidad: La velocidad máxima permitida. Sus valores dependen de los límites legales establecidos. La velocidad para los ejes por los que no está permitida la circulación de vehículos es de 1.

- sentido: El sentido de la circulación. Sus valores son:
    - UNO
    - DOS

- situacion: La situación geográfica. Sus valores son:
    - VAD: Vado (lugar de un río, arroyo o corriente de agua con fondo firme y poco profundo, por donde se puede pasar)
    - SUP: Vía superficial, sobre el terreno
    - OCU: Vía superficial sobre la que pasa una vía elevada
    - ELE: Vía elevada
    - SUB: Vía subterránea

- codigoc: El código de la carretera, si lo tuviera.

- nombre: El nombre de la carretera o calle, si lo tuviera.

- jerarquia: Valor entero que especifica la prioridad del eje a ser elegido para circular sobre él. Toma valores entre el 1 (mayor prioridad) y el 50 (menor prioridad).

- restri: Especifica restricción de acceso. Sus valores son:
    - 0: Sin restricción
    - 1: Acceso cerrado (puertas, barreras...)
    - 2: Vía solo peatonal o de bicicletas
    - 3: Prohibido excepto vehículos autorizados
    - 4: Propiedad privada
    - 5: Vías verdes
    - 6: Conexiones entre sentidos de autovías / autopistas

- rotonda: Si pertenece a una rotonda.

- peaje: Si pertenece a una carretera con peaje.

- vehiculo: Especifica el vehículo "mínimo" que puede circular por el eje. Sus valores son:
    - 1: Tráiler
    - 2: Camión de monte con remolque
    - 3: Turismo
    - 4: Camión de monte sin remolque
    - 5: Autobomba forestal
    - 6: Todoterreno

Es decir, por un eje que pueda circular un turismo (3), también puede hacerlo un camión de monte sin remolque (4), una autobomba forestal (5) y un todoterreno (6). Si un eje es peatonal este campo será nulo.

- shape_length: La longitud en metros del eje. Sus valores se distribuyen de la siguiente manera:

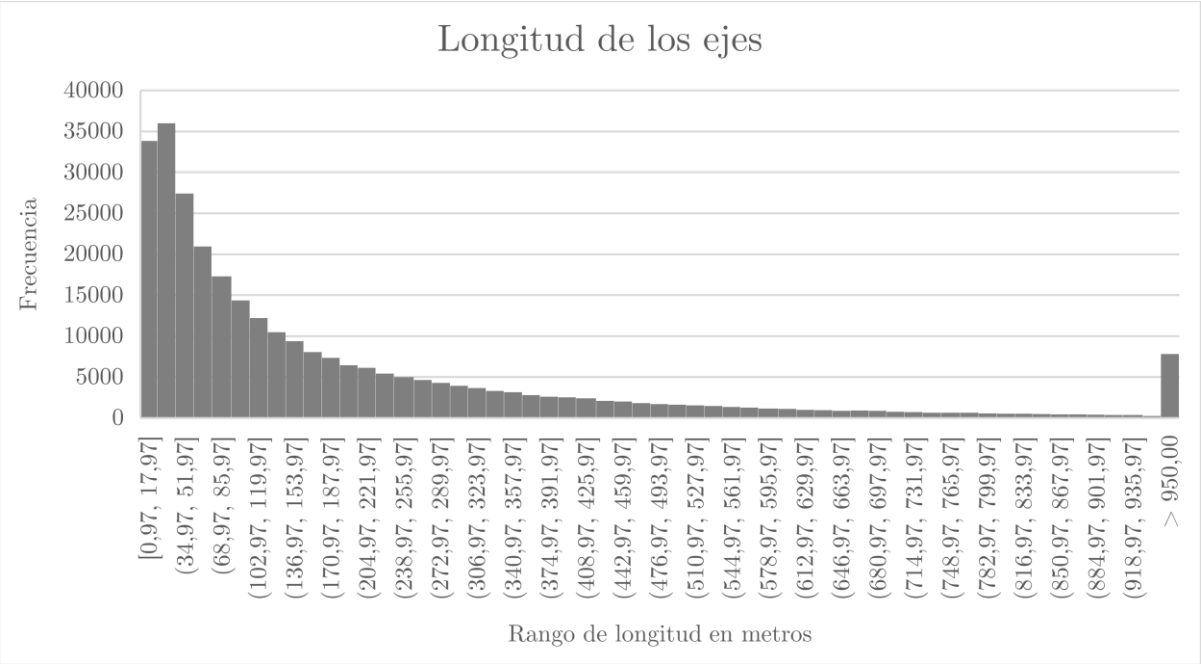



Ilustración 3: Histograma de la longitud de los ejes en metros.

- Valor mínimo: 0.9
- Valor máximo: 16168.20
- Media 203.64
- Moda: 98.16
- Mediana: 2.45
- Desviación estándar: 321.67
- Varianza: 103472.38

La base de datos cuenta con 300.000 ejes, aproximadamente.

Antes de continuar, es necesario explicar algunos conceptos.

## 3.1. Datos espaciales

Esta base de datos tiene una característica que la diferencia de las bases de datos convencionales, en las que es suficiente con utilizar los tipos de datos usuales (números enteros, números reales, caracteres...): Es una base de datos espacial, que como su nombre indica, está destinada a almacenar información espacial.

La información espacial se divide en dos tipos:

**Datos ráster:**

Los datos ráster [7, 8, 9] constan de una matriz de celdas (o píxeles) organizadas en filas y columnas en la que cada celda está asociada con una localización geográfica específica y contiene un valor que representa información del espacio ocupado por ella. Los datos almacenados en formato ráster representan fenómenos del mundo real, como la temperatura, o en el caso más popular, imágenes satelitales y fotografías aéreas.

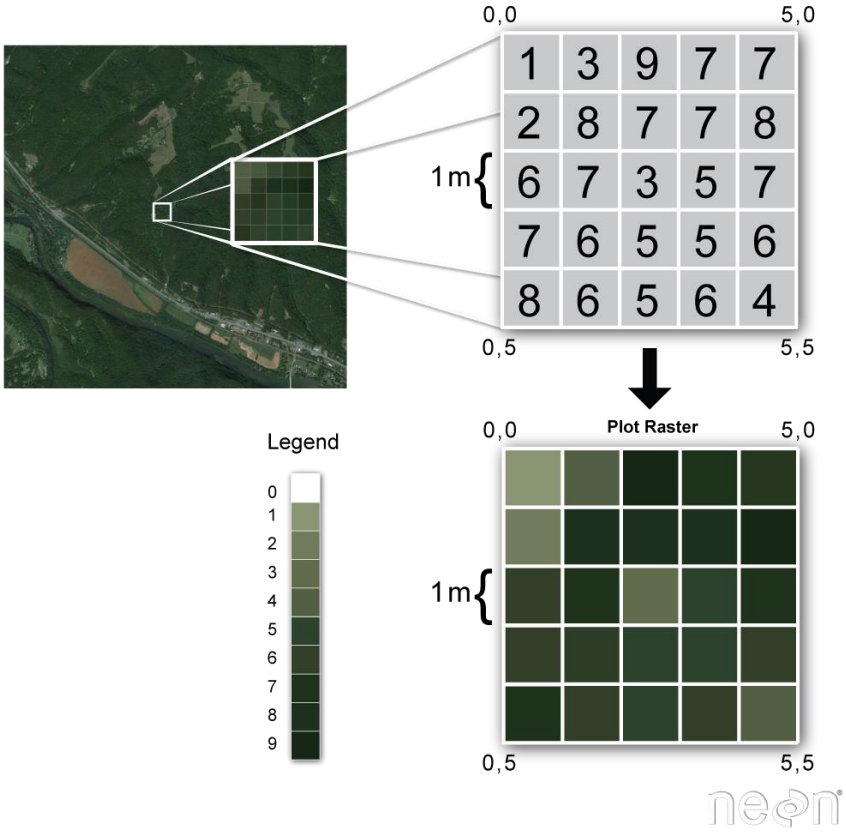



Ilustración 4: Representación ráster. Fuente: Red Nacional de Observatorios Ecológicos

El concepto de ráster es esencialmente el mismo al concepto de imagen digital, con la diferencia de que en un ráster cada píxel contiene información espacial que lo asocia a una ubicación.

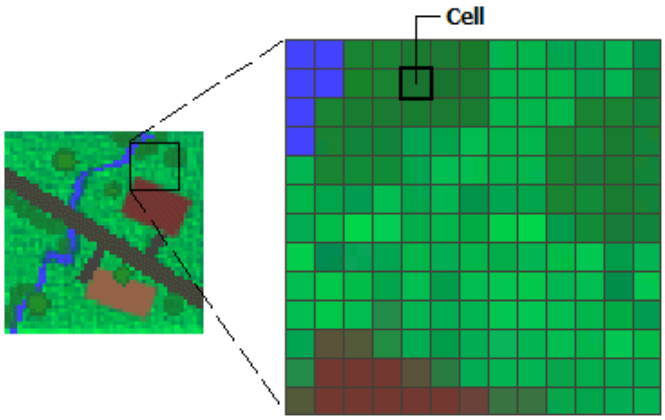


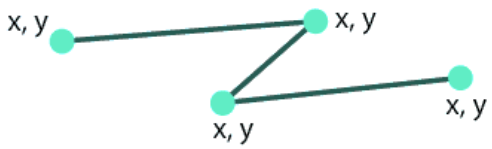

Ilustración 5: Representación ráster. Fuente: ARCGIS

**Datos vectoriales:**

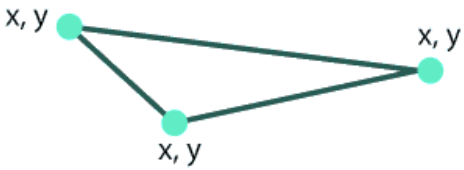
Los datos vectoriales [10, 11] constan de localizaciones o coordenadas geométricas (valores x, y e incluso z) llamados vértices que definen la forma de un objeto espacial. Estos vectores representan información del mundo real. La organización de los vértices define el tipo de vector que compone. Los 3 tipos de datos vectoriales son:

- Punto: Representado por una coordenada. Puede representar un árbol, un edificio...


- Línea: Representada por al menos 2 puntos (o vértices) conectados. Puede representar un río, una **carretera**...




- Polígono: Representado por al menos 3 puntos (o vértices) conectados y cerrados. Puede representar un lago, las fronteras de un país...



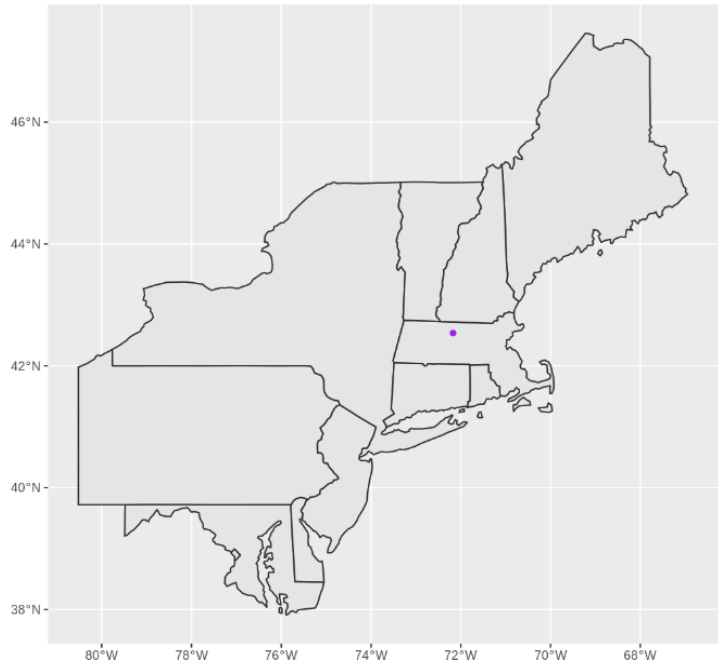



Ilustración 6: Mapa representado mediante datos vectoriales. Fuente: datacarpentry.org

**Diferencias entre ráster y vectorial:**

El dato ráster almacena el píxel mientras que el dato vectorial almacena las coordenadas de sus elementos.

Esencialmente, los dos tipos de dato se utilizan para diferentes casos de uso, pero en general, el dato vectorial tiene mayor capacidad de compactar información utilizando menor volumen de datos, además, son más precisos a la hora de calcular distancias y superficies y permiten tener límites más precisos. El dato ráster es más sencillo, y puede simular la realidad tridimensional de forma más fiable, ya que el dato vectorial es plano.

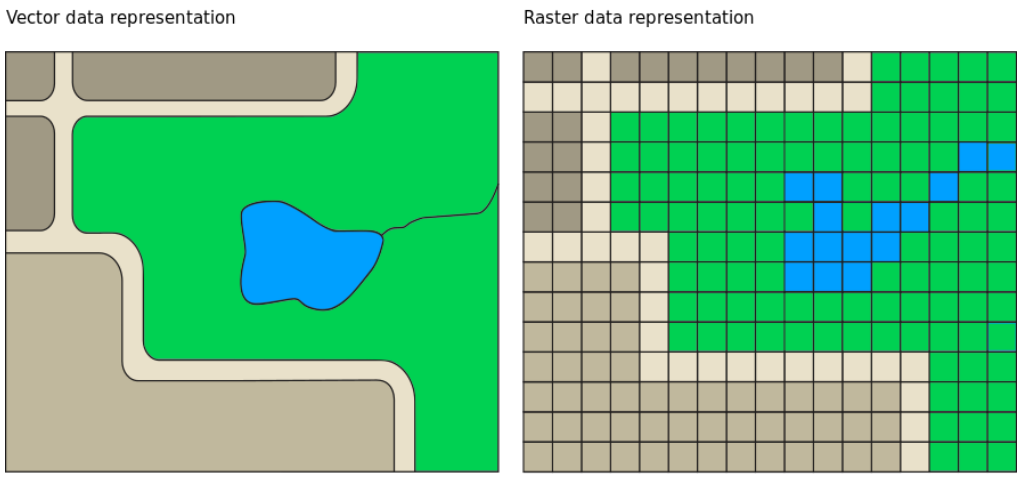



Ilustración 7: Diferencia entre representación vectorial y ráster. Fuente: oracle.com

En adelante, toda referencia a información espacial se referirá a información vectorial, líneas concretamente, debido a la naturaleza de los datos (carreteras, caminos y sendas).

## 3.2. Almacenamiento de información espacial

La organización internacional sin ánimo de lucro Open Geospatial Consortium (OGC) desarrolló el estándar que define el tipo de dato espacial fundamental: Geometry (o geometría) [12], que simplifica la representación de coordenadas geométricas, para que no sea necesario almacenar todas las coordenadas de un objeto, así como operaciones y métodos para manipular y transformar valores geométricos. El dato Geometry almacena la información basándose en un sistema de coordenadas plano (euclidiano). Abstrae y encapsula estructuras espaciales como el perímetro y la dimensión. Se puede entender como una forma o figura.

Además del tipo de dato Geometry, se define el tipo de dato Geography [13], con la única diferencia de que este último almacena la latitud y longitud del dato, sin necesidad de un sistema de coordenadas que lo proyecte a un plano. Pese a ser una representación más sencilla y precisa, el rendimiento del sistema disminuye notablemente al tener que considerar en todas las operaciones la curvatura de la Tierra, por lo que su uso no está extendido.

Nativamente, PostgreSQL, como los demás SGBD, no admiten el tipo de dato geometría ni sus operaciones, por lo que es necesario instalar una extensión que extienda el comportamiento de nuestra base de datos: PostGIS [14]. PostgreSQL con PostGIS es el estándar para las bases de datos espaciales de código abierto.

Ahora, se puede continuar especificando la información que almacena cada eje:

- the_geom: La geometría del eje.

## 3.3. Sistema de Referencia Espacial

Una geometría almacena información basándose en un sistema de coordenadas plano, en dos dimensiones. No es posible representar una superficie de tres dimensiones, la Tierra, en este caso, en dos dimensiones sin deformar la información original. El hecho de pasar una representación de tres dimensiones a un plano se conoce como proyección.

Ilustración 8: Representación de la Tierra en un plano. Fuente: ARCGIS

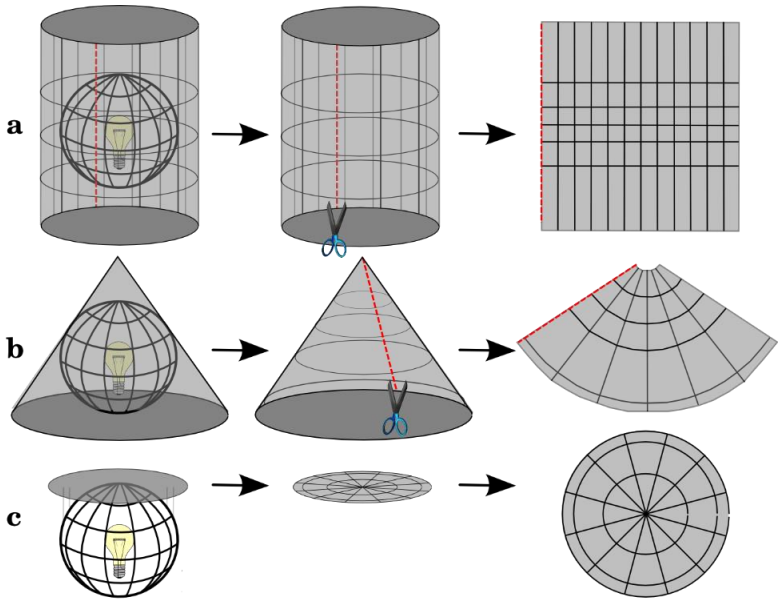



Ilustración 9: Proyección cilíndrica (a), cónica (b) y plana (c). Fuente qgis.org

Todas las proyecciones distorsionan la distancia, la dirección, la escala y el área de alguna manera.

Un SRE, Sistema de Referencia Espacial o Sistema de Coordenadas (CRS, Coordinate Reference Systems o SRS, Spatial Reference System en inglés) [15, 16] define como la información en dos dimensiones se relacionan con su ubicación real en la Tierra. Es algo imprescindible en este tipo de bases de datos, para definir la localización y relación entre geometrías, ya que los datos tratados en este tipo de bases de datos tienen un valor relativo, no un valor absoluto, a diferencia de los tipos de datos convencionales.

Existen dos tipos principales de SRE:

- Sistema de Referencia Espacial Geodésico: Utiliza coordenadas geográficas angulares (longitud y latitud) que referencian directamente a la superficie de la Tierra. Define la localización de un punto en la superficie de la Tierra.

- Sistema de Referencia Espacial Proyectado: Utiliza una proyección matemática para "aplanar" la superficie de la Tierra. Define la forma de representar un puto de la Tierra en una superficie plana.

Ya que en este proyecto se trabaja únicamente con la geografía de Navarra, es relativamente sencillo minimizar las distorsiones de su superficie, que se acentúan al intentar representar toda la esfera en dos dimensiones. Al ser una zona reducida, proyectarla en un plano es más sencillo, es obvio que a mayor superficie se quiera proyectar, mayor es la curvatura de la superficie, que causa mayor distorsión.

El Sistema de Referencia Espacial Proyectado para la zona geográfica que comprende Navarra es el EPSG:25830, sobre el que se proyectan los ejes de la base de datos.

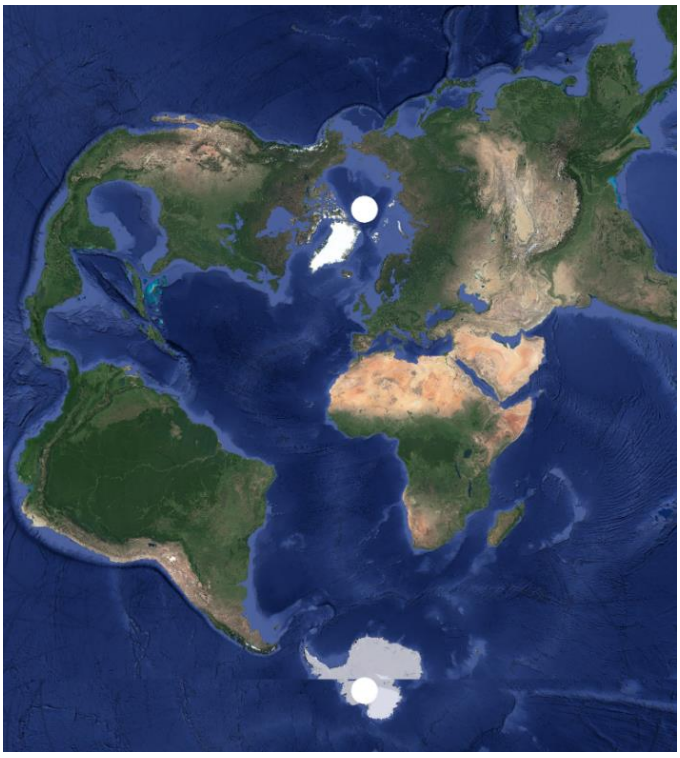


Ilustración 10: Representación de la Tierra usando la proyección EPSG:25830 que reduce la distorsión en Navarra. Imagen: Google Earth

## 3.4. OpenStreetMap

Es la mayor base de datos de mapas libre del mundo. El proyecto OSM (OpenStreetMap) [17, 18] es un proyecto colaborativo que nace en 2004 para crear cartografía gratuita y de libre uso. Su desarrollo es esencial para el crecimiento de los Sistemas de Información Geográfica libres.

Actualmente se distribuye bajo Licencia Abierta de Bases de Datos (ObdL), basada en la licencia Creative Commons. Por lo tanto, cualquier persona tiene derecho a copiar, distribuir, comunicar, transformar y comercializar dicha cartografía con la única condición de decir quien la ha producido y utilizar una licencia igual o compatible en la creación de cualquier producto derivado.

OSM integra en una única base de datos los datos que terceros han liberado, combinando en un único lugar y con un único formato datos de todo tipo y de todos los países del mundo: desde calles y carreteras, hasta edificios y parques, comercios, lugares naturales, rutas de transporte público, tendido eléctrico, y todo lo que uno pueda imaginar. Hoy cuenta con más de 7 mil millones de nodos y 200 millones de ejes.

Pese a que para este proyecto se utilice información propia, el formato de almacenamiento de datos de OpenStreetMap (.osm) es prácticamente un estándar para los mapas de código abierto, y es utilizado en multitud de escenarios. El formato .osm es un archivo en formato .xml.

El dato de OSM está formado por 4 tipos de elementos [19]:

- Nodo: 
  Consiste en un punto en el espacio, y viene definido por su latitud, su longitud y un identificador. Los nodos pueden utilizarse para reflejar elementos puntuales en la cartografía (POI), pero su uso más extendido es el de puntos que se utilizan para definir la geometría de un elemento lineal (vía).

  ```
  <node id='1' lat='37.233335' lon='-115.812772'/>
  ```

- Vía: 
  Una vía es una lista ordenada de nodos. Pueden utilizarse para representar carreteras y caminos.

  ```
  <way id='1'>
      <nd ref='1'/>
      <nd ref='2'/>
      <nd ref='3'/>
  </way>
  ```

- Relación: 
  Una relación es un elemento con al menos la etiqueta *type=** y un grupo de miembros que es una lista ordenada de uno o más nodos, vías y/o relaciones. Se utiliza para definir relaciones lógicas o geográficas entre estos diferentes objetos (p. ej., un lago y su isla, o varias carreteras para una ruta de autobús).

```
<relation id="1">
    <member type="node" ref="1" role="stop"/>
    ...
    <member type="way" ref="2" role="via"/>
</relation>
```

- Etiquetas: 
  
  Una etiqueta consta de dos elementos, una 'clave' y un 'valor'. Las etiquetas describen características específicas de los elementos del mapa: (nodos, vías o relaciones).

```
<node id='83' lat='41.902172' lon='12.453934'>
    <tag k="building" v="cathedral"/>
<node/>

<way id='1'>
    <nd ref='1'/>
    <nd ref='2'/>
    <nd ref='3'/>
    <tag k="name" v="5th Ave"/>
</way>
```

Hasta ahora se han definido los conceptos esenciales de los SIG (Sistema de Información Geográfica) para comprender este proyecto.

# 4. Modelos de enrutamiento

Una aplicación de enrutamiento ofrece, dados unos datos espaciales, una ruta entre dos puntos. Este problema, esencialmente, es un problema matemático escalado al mundo real: Encontrar el camino más corto entre dos nodos de un grafo, donde las calles son las aristas del grafo y sus intersecciones los nodos.

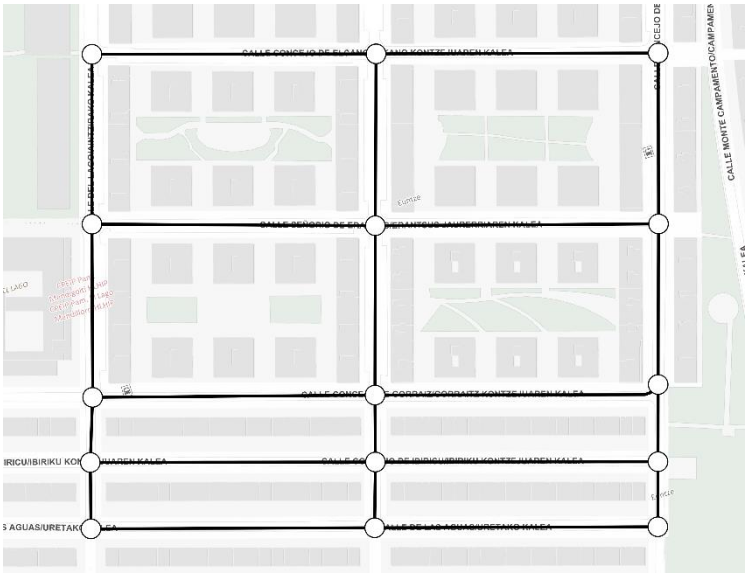


Ilustración 11: Representación de calles e intersecciones como elementos de un grafo. Mapa base: IDENA

Entre los algoritmos que resuelven este problema destacan:

**Algoritmo de Dijkstra:**

El algoritmo de Dijkstra [20] es un algoritmo de búsqueda de rutas mínimas en un grafo con pesos en sus arcos, es decir, un grafo donde cada arco tiene asociado un peso o costo.

El algoritmo fue desarrollado por Edsger Dijkstra en 1956 y es ampliamente utilizado en muchas aplicaciones, como redes de computadoras, sistemas de navegación y en la planificación de rutas de transporte.

El algoritmo de Dijkstra funciona de la siguiente manera:

1. Se elige un nodo inicial y se marca como el nodo actual.
2. Se calcula la distancia desde el nodo inicial hasta todos los demás nodos adyacentes y se almacena en una tabla de distancias. Si el nodo actual no tiene ningún nodo adyacente, entonces el algoritmo termina.
3. Se selecciona el nodo adyacente con la menor distancia y se marca como el nodo actual.

4. Se actualizan las distancias a los nodos adyacentes al nodo actual, si es necesario. Para cada nodo adyacente, si la distancia desde el nodo inicial hasta el nodo adyacente a través del nodo actual es menor que la distancia almacenada actualmente para ese nodo, entonces se actualiza la distancia.
5. Se repiten los pasos 3 y 4 hasta que todos los nodos hayan sido visitados o hasta que se haya encontrado una ruta mínima a todos los nodos.

**Algoritmo A*:**

El algoritmo A* (A estrella) [21] es un algoritmo de búsqueda de rutas óptimas en un grafo. A diferencia del algoritmo de Dijkstra, que sólo tiene en cuenta la distancia recorrida desde el nodo inicial hasta el nodo actual, el algoritmo A* también tiene en cuenta una estimación de la distancia restante hasta el nodo final (también conocida como la heurística). Esto permite que el algoritmo A* encuentre rutas más cortas y eficientes que el algoritmo de Dijkstra en muchos casos.

El algoritmo A* funciona de la siguiente manera:

1. Se elige un nodo inicial y un nodo final.
2. Se crea una lista de nodos abiertos y otra de nodos cerrados. Se añade el nodo inicial a la lista de nodos abiertos y se calcula su costo total, que es la suma de la distancia desde el nodo inicial hasta el nodo actual y la estimación de la distancia restante desde el nodo actual hasta el nodo final (la heurística).
3. Se selecciona el nodo de la lista de nodos abiertos con el menor costo total y se marca como el nodo actual. Si el nodo actual es el nodo final, entonces el algoritmo ha encontrado una ruta óptima y termina. Si no, se añade el nodo actual a la lista de nodos cerrados y se procesan todos sus nodos adyacentes.
4. Para cada nodo adyacente al nodo actual, se calcula su costo total como la suma de la distancia desde el nodo inicial hasta el nodo adyacente a través del nodo actual y la estimación de la distancia restante desde el nodo adyacente hasta el nodo final (la heurística). Si el nodo adyacente ya está en la lista de nodos cerrados, se ignora. Si el nodo adyacente ya está en la lista de nodos abiertos, se comprueba si el nuevo camino es más corto o no. Si es más corto, se actualiza el costo total del nodo adyacente y se actualiza su camino anterior. Si el nodo adyacente no está en ninguna de las dos listas, se añade a la lista de nodos abiertos y se le asigna su costo total y su camino anterior.
5. Se repiten los pasos 3 y 4 hasta que se haya encontrado una ruta óptima o hasta que se haya explorado todo el grafo y no se haya encontrado una ruta válida.

Ambos algoritmos cuentan con una versión bidireccional, la cual explora el grafo desde ambos extremos (desde el nodo inicial y desde el nodo final) simultáneamente.

La principal ventaja de la variante bidireccional es que es más rápido que el algoritmo original en muchos casos. Esto se debe a que reduce la cantidad de nodos que se tienen que explorar para encontrar una ruta mínima. En lugar de explorar todos los nodos desde el nodo inicial hasta el nodo final, el algoritmo bidireccional solo explora aquellos nodos que están a una distancia razonable de ambos extremos. Esto permite

encontrar rutas mínimas mucho más rápido que el algoritmo de en grafos grandes y densos.

Sin embargo, hay algunos casos en los que el algoritmo bidireccional no es tan eficiente como el algoritmo original. Por ejemplo, si el grafo es muy disperso o si hay una gran cantidad de arcos con pesos muy altos, el algoritmo original puede ser más rápido que el algoritmo bidireccional. Además, el algoritmo bidireccional requiere más memoria.

Estos algoritmos son la base de las principales aplicaciones de enrutamiento, que enumeraré a continuación. Como todo el software utilizado para este proyecto, se ha trabajado con aplicaciones de código abierto:

## 4.1. pgRouting

Es una extensión para Postgres con PostGIS [22] que ofrece, entre otras funcionalidades, la de encontrar el camino más corto entre dos nodos de una base de datos espacial. Dispone de diferentes funciones que realizan este enrutamiento a través de consultas a la misma.

Las funciones de enrutamiento que ofrece esta extensión requieren que la base de datos cumpla una serie de requisitos, debe tener una topología correcta o válida. La topología describe la relación entre los nodos y ejes de la red. Asimismo, todo eje debe de tener un nodo inicio y un nodo final, que represente el origen y el destino de las rutas. Para ello, pgRouting ofrece funciones que procesan la base de datos, haciendo de ella una red o grafo con aristas y nodos sobre la que calcular una ruta [23]:

**pgr_createTopology**

Esta función crea una topología a partir de una tabla de ejes con geometría. A partir de esta, crea una tabla de nodos (*ejes*_vertices_pgr) en la que almacena los puntos de inicio y final de los ejes, que serán los nodos de la red. Entre la información que almacena de cada nodo se encuentra:

- id: Identificador del nodo. Actuará como clave primaria.
- the_geom: Geometría del nodo. Será un punto.

Además de crear una tabla que almacene los nodos de la red, añade, a la tabla de los ejes, la siguiente información:

- source: Identificador del nodo origen del eje.
- target: Identificador del nodo destino del eje.

La geometría de los ejes es dirigida, tiene un inicio (origen) y un final (destino). En ejes de un sentido, esta información indicará dicho sentido. La función inserta esta información, pero estos dos campos deben ser añadidos manualmente.

Además, a todo eje de la red deberá asignársele un coste, que represente si ese enlace debiera ser usado en más o menos proporción según unas características que se explican posteriormente.

Este procesamiento previo no sería necesario si se partiera de información de mapas de OpenStreetMap, que cuenta con herramientas que preparan directamente información en formato .osm a base de datos PostGIS con pgRouting, una práctica muy común.

pgRouting ofrece una gran variedad de funciones para el análisis de la red y cálculo de rutas, y su mayor ventaja es que permite una gran personalización al actuar directamente sobre la base de datos. Es el proyecto de cálculo de rutas en código abierto más longevo y establecido, con una gran comunidad que lo mantiene actualizado.

## 4.2. Aplicaciones sobre OSM

Existen múltiples aplicaciones de enrutamiento (desarrolladas principalmente en JAVA) basadas en OpenStreetMap. Son aplicaciones complejas que ofrecen múltiples funcionalidades, pero todas comparten que o bien recogen datos directamente de la base de datos de OpenStreetMap, o bien pueden utilizar archivos .osm. Las primeras no son relevantes para este proyecto, ya que los datos con los que se trabaja no coinciden siempre con los de OpenStreetMap además de que cualquier usuario puede realizar cambios a estos. Sobre las segundas se indagará más adelante, investigando las opciones para que los datos en PostGIS puedan ser explotados por una de estas aplicaciones.

## 4.3. ol-ext

Es una librería JavaScript [24] que ofrece enrutamiento en el cliente (navegador web). Alternativa muy interesante por la naturaleza del proyecto, que es ofrecer un servicio web, sin necesidad de desarrollar un back-end. ol-ext es una extensión a la librería JavaScript OpenLayers, librería de código abierto para la creación de aplicaciones de mapas web.

# 5. Implementación

## 5.1. Aplicación web con pgRouting, GeoServer y OpenLayers

### 5.1.1. Back-end en PostGIS con pgRouting

pgRouting implementa dos algoritmos para resolver el problema del camino más corto entre dos nodos: El algoritmo de Dijkstra y el algoritmo A* , ambos con sus versiones bidireccionales. El desarrollo de esta aplicación se basará en la implementación del algoritmo de Dijkstra [25].

```
pgr_dijktstra(Edges SQL, start vid, end vid)
```

- Edges SQL: Consulta SQL que debe proporcionar el conjunto de registros de los ejes sobre los que se realiza la ruta con los siguientes campos:
  - id: Identificador del eje.
  - source: Nodo origen del eje.
  - target: Nodo destino del eje.
  - cost: Coste de recorrer el eje.
- start vid: Identificador el nodo origen de la ruta.
- end vid: Identificador del nodo destino de la ruta.

Para utilizar esta función es necesario especificar el coste de recorrer los ejes. En la tabla ejes se almacenan, por cada eje, dos formas distintas de representar este coste:

- cost_time: El coste en segundos de recorrer el eje. Se ha calculado en base a la distancia y velocidad media a la que se circula por el eje.
- cost_length: El coste en metros de recorrer el eje. Su valor es el mismo que el campo *shape_length* del eje.

Así, se puede calcular, según el coste que se elija, el camino de menor tiempo (más rápido) o el de menor distancia (más corto). Por lo general, el camino deseado es el que menor tiempo cueste.

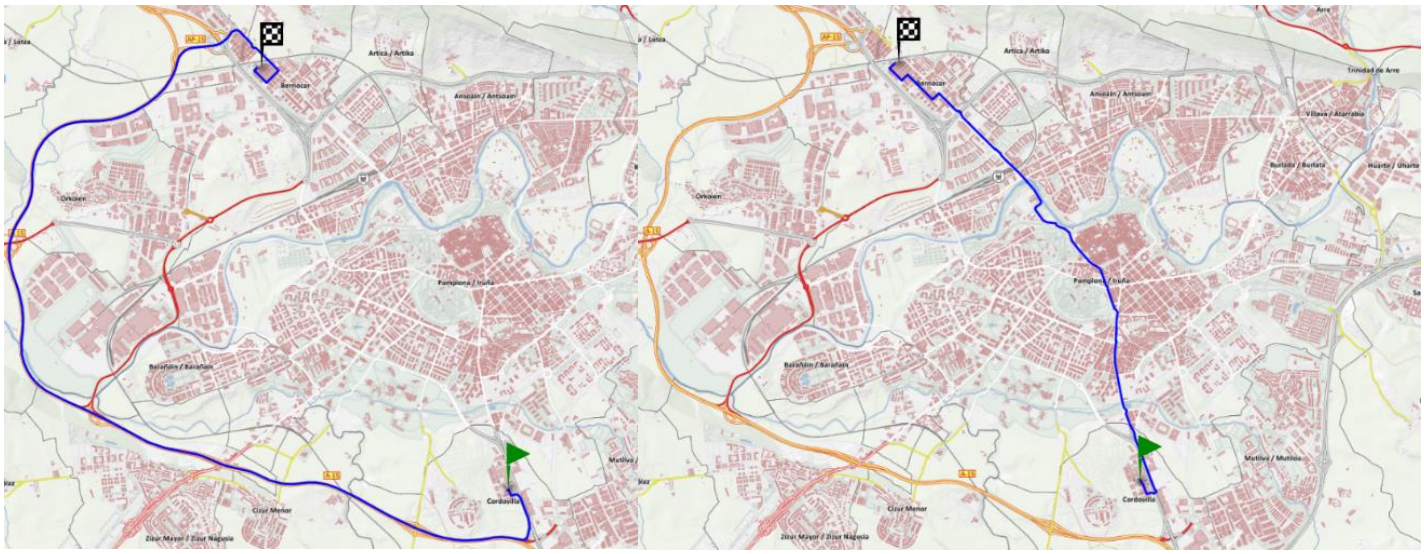


Ilustración 12: A la izquierda: el camino más rápido, a la derecha: el más corto. Mapa base: IDENA

De esta manera, un ejemplo de ejecución de la función `pgr_dijkstra` utilizando el coste temporal para calcular el camino más corto entre el nodo 58229 y el nodo 39198 es:

```
select * from pgr_dijkstra('select id, source, target, cost_time as cost from ejes', 58229, 39198)
```

Esta función devuelve un conjunto de registros que conforman el camino con la siguiente estructura:

- seq: Valor secuencial, comienza en 1.
- node: Identificador del nodo en el camino.
- edge: Identificador del eje recorrido para ir de *node* al siguiente nodo en la secuencia. Toma el valor -1 para el último nodo del camino.
- cost: El coste de ir desde *node* a través de *edge* al siguiente nodo en la secuencia.
- agg_cost: El coste agregado desde el nodo inicial hasta *node*

Para visualizar la ruta es necesario realizar un *join* de este resultado con la tabla ejes, que almacena la geometría que se visualizará, relacionando el *id* de los ejes con el *edge* del resultado:

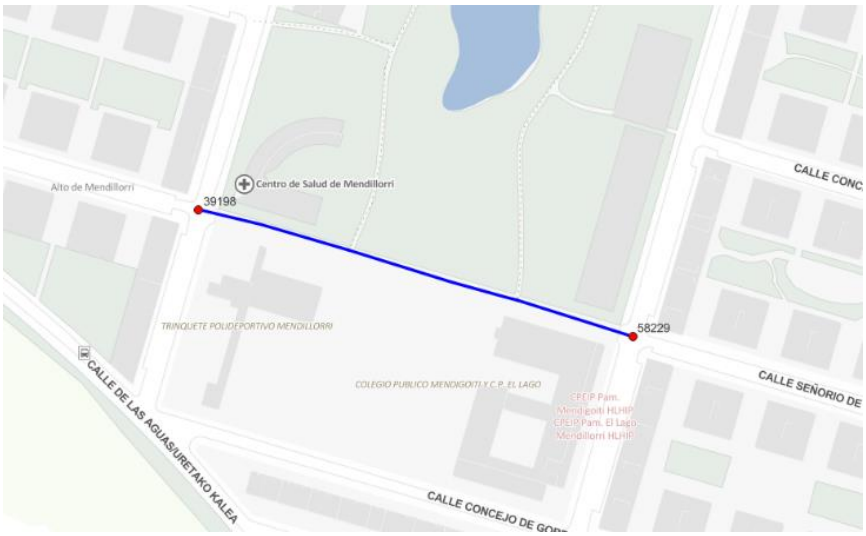



Ilustración 13: Camino más corto entre dos nodos utilizando la función pgr_dijkstra. Mapa base: IDENA

En la ilustración 13 se puede observar la ruta de menor coste temporal entre el nodo 58229 y el nodo 39198. En este caso se recorre un eje peatonal. De ahora en adelante, para todas las rutas, se restringirán los ejes a aquellos por los que puedan circular vehículos, añadiendo al parámetro edges SQL `where vehiculo is not null`, para no tener en cuenta los ejes con vehículo nulo. Quedando la query:

```
select * from pgr_dijkstra('select id, source, target, cost_time as cost from
ejes where vehiculo is not null', 58229, 39198)
```

La ruta quedaría de la siguiente manera:

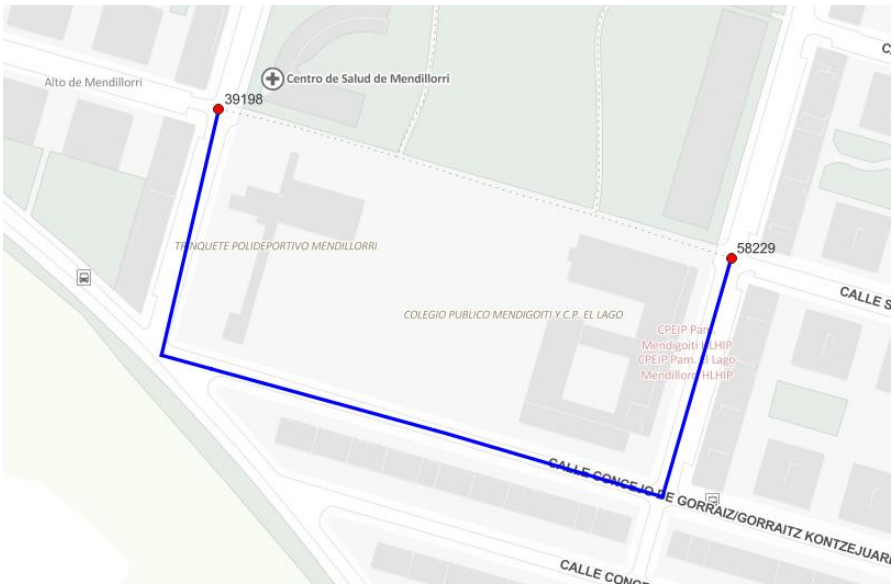



Ilustración 14: Camino más corto en vehículo entre dos nodos utilizando la función pgr_dijkstra. Mapa base: IDENA

Hasta este momento se ha tratado el problema de enrutamiento como si la red de ejes se tratase de un grafo no dirigido, cuando en la realidad el grafo formado por las carreteras es un grafo dirigido, ya que hay tramos de sentido único por los que no se pueden circular en ambas direcciones. La función `pgr_dijkstra` considera enrutar sobre grafos dirigidos, con el parámetro opcional *directed*.

```
pgr_dijktstra(Edges SQL, start vid, end vid, [directed])
```

Para añadir información del grafo dirigido, al parámetro Edges SQL se le proporciona un campo adicional:

- **reverse_cost: Coste de recorrer el eje en sentido contrario.**

La tabla ejes contiene información del coste de recorrer el eje en sentido contrario de forma análoga al coste normal:

- rcost_time
- rcost_length

Para representar un eje de sentido único, ambos costes "inversos" tendrán un valor negativo (-1) para indicar que no se puede tomar dicho camino. En los demás casos, cuando un eje sea bidireccional, el coste normal será el mismo al coste "inverso". Al tener esto en cuenta, la ruta anterior cambia al pasar previamente por ejes de un solo sentido por los que estaba circulando en sentido contrario. Quedando la query:

```
select * from pgr_dijkstra('select id, source, target, cost_time as cost,
rcost_time as reverse_cost from ejes where vehiculo is not null', 58229, 39198,
true)
```

La ruta quedaría de la siguiente manera:

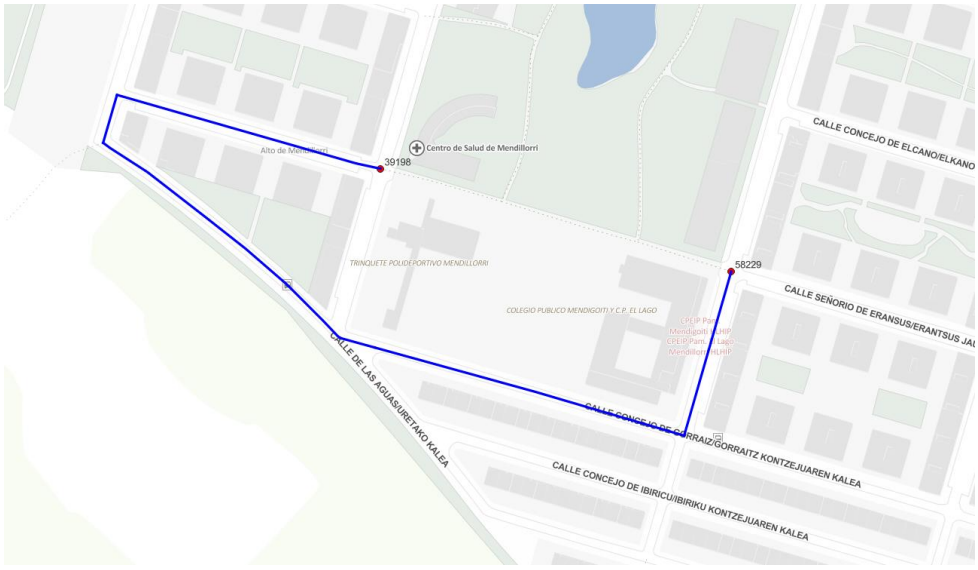



Ilustración 15: Camino más corto en vehículo entre dos nodos utilizando la función pgr_dijkstra en red dirigida. Mapa base: IDENA

Además de circular por los sentidos permitidos, para resolver un problema de enrutamiento en carretera es necesario tener en cuenta las restricciones de giro que existen en ellas: Las prohibiciones de cambio de sentido y de giro, que aportan una información adicional a los sentidos de circulación y que son de obligatorio cumplimiento.

Ilustración 16: Señal de prohibición de giro a la izquierda. Fuente: Google

Estas restricciones de giro están contempladas en pgRouting y existe una función análoga a `pgr_dijkstra` [26] que además de recoger los parámetros previamente mencionados, añade el parámetro *Restrictions SQL*.

```
pgr_trsp(Edges SQL, Restrictions SQL, start vid, end vid, directed)
```

- Restrictions SQL: Consulta SQL que debe proporcionar el conjunto de registros de las restricciones de giro con los siguientes campos:
  - to_cost: Coste de tomar el camino prohibido. Es un valor tan elevado que asegura que nunca se tomará el camino, o negativo.
  - target_id: Identificador de la arista destino del camino que no se permite tomar.
  - via_path: Identificador de la arista origen del camino que no se permite tomar.

La información de las restricciones de giro se almacena en otra tabla con estos campos. La función `pgr_trsp` (turn restricted shortest path) calcula el camino más corto aplicando el algoritmo de Dijkstra, ajustando el coste de recorrer caminos restringidos para que nunca se tomen.

El objetivo de este proyecto es desarrollar un servicio web para el cálculo de rutas. El usuario final no va a disponer de los identificadores de los nodos de la base de datos, ni sería práctico requerirlos, por lo que es necesario desarrollar una función que, dadas unas coordenadas, información pública y accesible, realice la ruta entre los nodos más cercanos a dichas coordenadas. Pero no siempre el elemento del grafo más cercano esas coordenadas será un nodo, sino un punto intermedio de un eje: en la mayoría de las ocasiones el usuario no partirá de una intersección, inicio o fin de eje, sino de un punto de una calle, por lo que habrá que seleccionar uno de los dos nodos (*source* o *target*) del eje del que se parte o se llega como nodo de origen o destino.

Además, se deberá añadir a la ruta, de manera manual, el tramo para llegar desde las coordenadas de origen al nodo origen y desde el nodo destino a las coordenadas destino. Para obtener estos tramos, se calcula la proyección del punto sobre el eje más cercano y se selecciona la porción de la geometría de dicho eje desde el punto proyectado hasta el nodo origen, y de manera análoga para el tramo desde el nodo destino hasta el punto destino. Después, se calcula el coste del tramo en proporción a esta proyección. Estos costes, y la información del eje original que se mantiene (nombre, velocidad...) deben ser añadidos a cada camino.


Ilustración 17: Punto de origen (rombo azul) más cercano a un eje que a un nodo.

A través de diferentes funciones que ofrece PostGIS, se identifica el eje más cercano a unas coordenadas. A continuación, se debe decidir cuál de sus dos nodos debería tomarse como nodo origen o destino. La forma más intuitiva de resolver este problema sería el de seleccionar el origen (o *source*) del eje de origen, y el destino (o *target*) del eje destino. De esta manera, sería seguro que se puede partir del nodo seleccionado como origen y que se puede llegar al nodo seleccionado como destino.

Esta solución es problemática cuando se desee partir o llegar a un punto de una vía de único sentido: Si se toma el origen de esta vía como origen de la ruta, el usuario nunca va a poder llegar al punto donde inicia su ruta, ya que estando en un punto de esa vía de un sentido, debe desplazarse en sentido contrario para llegar al origen de esta. Tampoco va a poder llegar al punto destino, ya que la ruta le dirigirá al destino de la vía de único sentido, desde la cual no es posible circular a otro punto de esa vía.

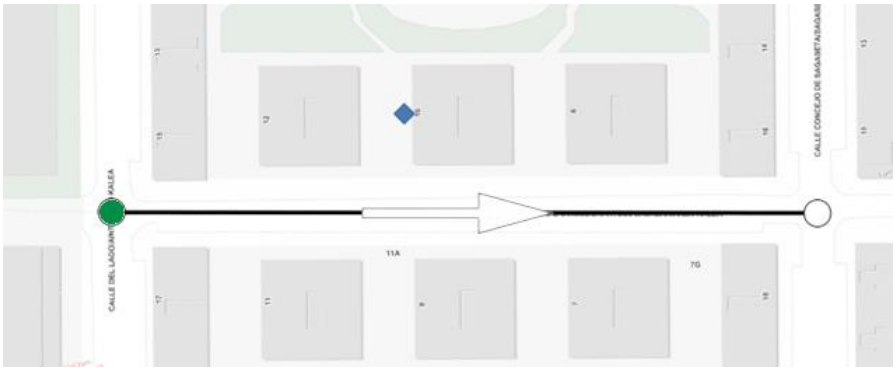


Ilustración 18: Nodo origen de la ruta (verde) inaccesible desde punto de origen (azul) por el sentido único de la vía.

Entonces, es obvio que para poder ofrecer una ruta que permita partir y llegar a las coordenadas especificadas, se debe seleccionar el nodo destino del eje más próximo a la coordenada origen, para poder acceder a él, y el nodo origen del eje más próximo a la coordenada destino, para poder acceder a un punto de este desde él.

Esta solución solventa el problema de las vías de sentido único, pero en las de sentido doble, en las que se puede llegar desde cualquier punto de ella a cualquiera de sus dos nodos y llegar a cualquier punto de ella desde cualquiera de sus dos nodos, no es necesario aplicar dicha solución. Hay dos nodos candidatos a ser seleccionados como origen o destino.

En este escenario, no es posible saber cuál de los dos nodos del eje ofrecerá la mejor ruta a priori. Escoger el nodo más cercano a las coordenadas de origen/destino no asegura que el camino sea más corto que si se eligieran los nodos más lejanos, como muestra la ilustración 19, en la que se parte y se llega a ejes de doble sentido, donde seleccionando el nodo más cercano al origen como nodo origen (ilustración 20) se realiza una ruta más costosa que seleccionando el nodo más lejano (ilustración 21). Seleccionar los nodos de origen y destino condiciona la ruta por completo y la única forma de ofrecer la ruta óptima es probando ambos candidatos, realizando sus respectivas rutas y comparando sus costes.

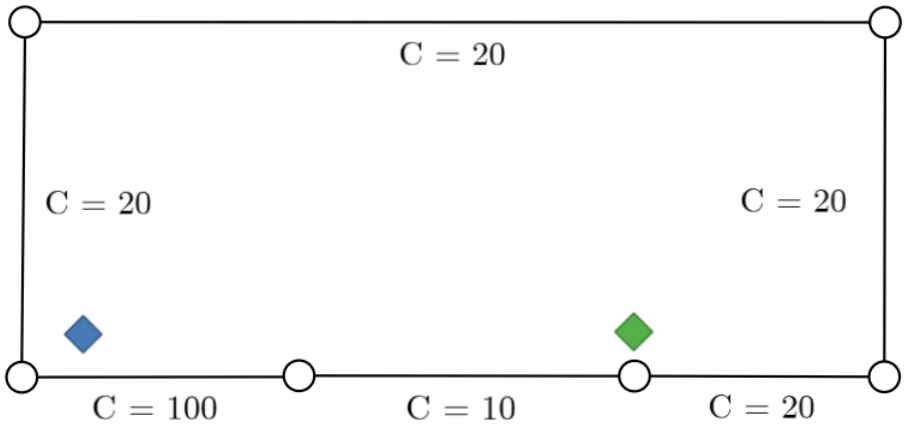



Ilustración 19: Punto de origen (azul). Punto de destino (verde). Todas las aristas son bidireccionales.

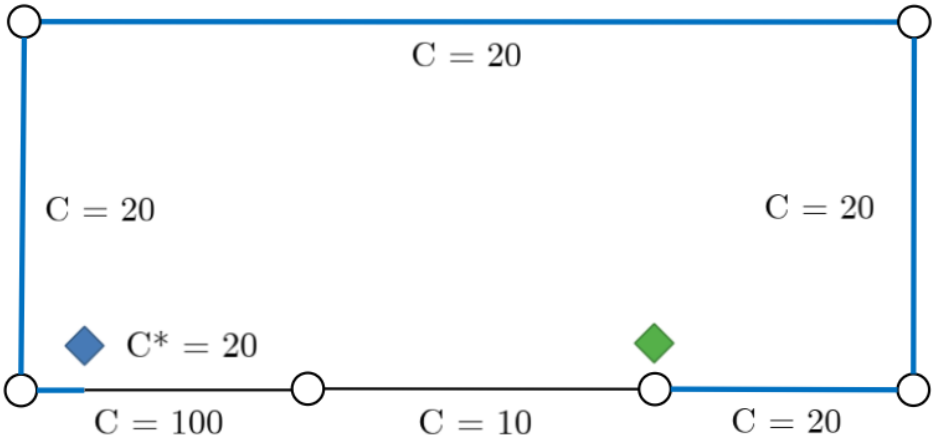



Ilustración 20: Camino de coste 100 sobre el grafo de la ilustración 19. C* representa el coste del tramo de completar la ruta desde el punto de origen hasta el nodo de origen.

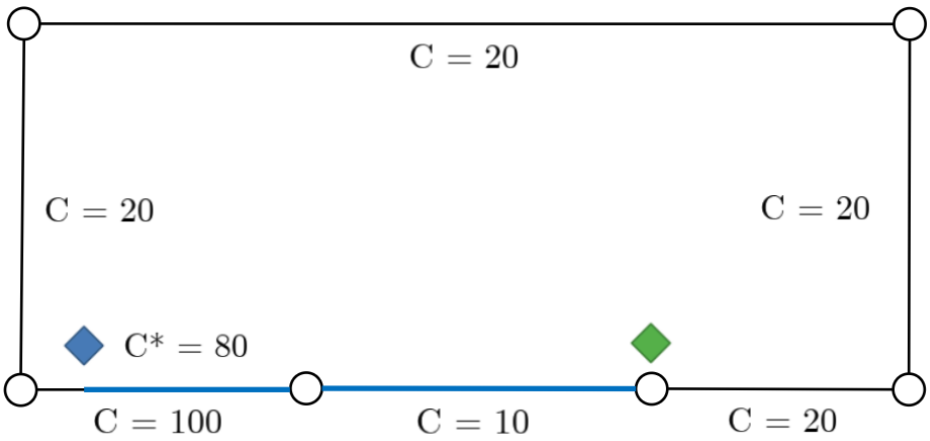



Ilustración 21: Camino de coste 90 sobre el grafo de la ilustración 19. C* representa el coste del tramo de completar la ruta desde el punto de origen hasta el nodo de origen.

pgRouting ofrece una función en estado de beta que solventa el problema de seleccionar nodo permitiendo que, en vez de introducir los nodos de origen y destino, se pueda introducir un punto intermedio de un eje como origen y destino. Esta función no añade al camino el tramo entre las coordenadas y el nodo, pero tiene en cuenta sus costes por lo que se asegura obtener el camino óptimo, aunque haya que añadir estos tramos manualmente. A fecha de finalización de este proyecto esta función ha sido descontinuada.

Es altamente ineficiente calcular hasta 4 rutas (cuando los puntos de origen y destino se encuentren en puntos intermedios de vías de doble sentido) para mostrar la ruta óptima, por lo que se debe encontrar otra solución: Crear nodos auxiliares. [27]

La idea de esta solución surge de las complicaciones de seleccionar el nodo de origen o de destino, ya que en la realidad un nodo casi nunca es el punto de la red más cercano a las coordenadas de origen o destino. Esta solución pasa por crear unos nodos auxiliares en la proyección de los puntos de origen o destino sobre los ejes.

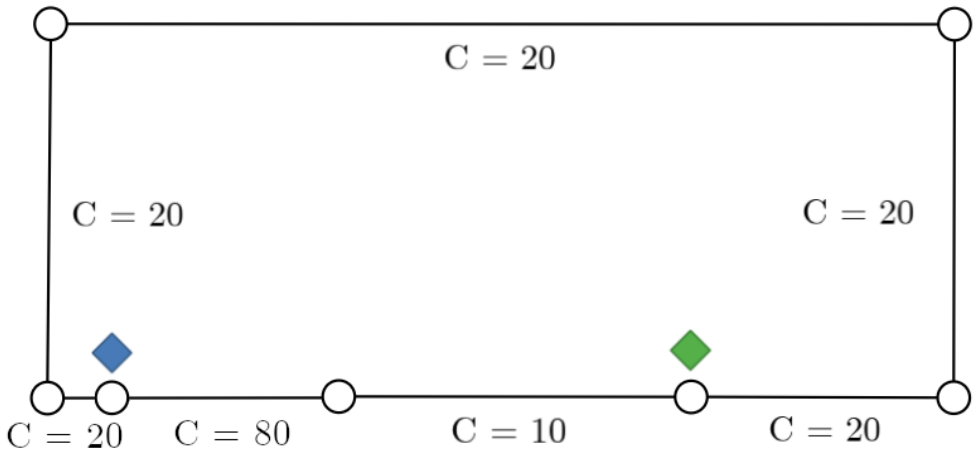



Ilustración 22: Grafo de la ilustración 19 con un nodo auxiliar en la proyección sobre el grafo del punto de origen.

Así, especificando los nodos auxiliares como origen y destino, se asegura encontrar la ruta óptima entre dos puntos. De esta manera, también se soluciona la necesidad de mostrar manualmente el camino desde el punto de origen hacia el primer nodo del camino, ya que este punto es ahora el primer nodo de la ruta.

En la ilustración 19 se mostraba la problemática de no saber que nodo escoger como inicio de la ruta, en las ilustraciones 20 y 21 se mostraban 2 caminos de costes diferentes dependiendo del nodo seleccionado, ambos añadiendo manualmente el tramo entre punto de origen y nodo de origen, y en la ilustración 23 se muestran ambos problemas solucionados realizando una ruta sobre el grafo de la ilustración 22 con nodo auxiliar.

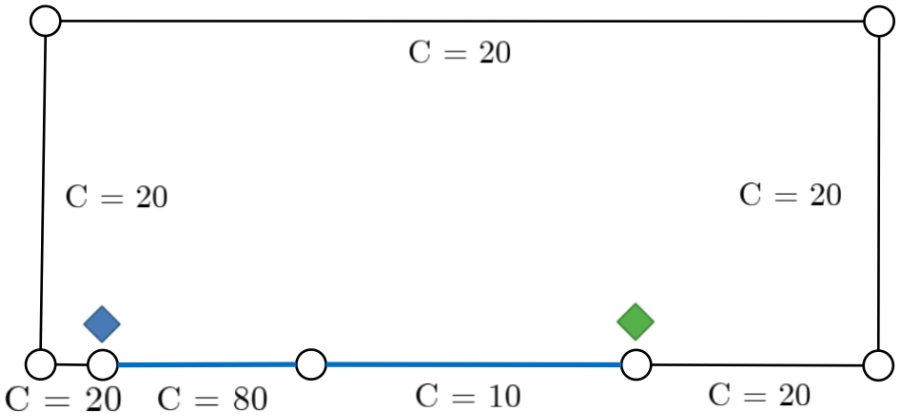



Ilustración 23: Camino óptimo entre el punto de origen (originalmente punto intermedio de un eje) y punto destino.

Los nodos auxiliares, en realidad, son ejes auxiliares. pgRouting enruta sobre los ejes, utilizando los campos que se le proporciona a la función escogida en el parámetro *Edges SQL*: *id*, *source*, *target*, *cost* y *reverse_ cost*. La manera en la que se "introducen" estos nodos en la red es añadiendo a *Edges SQL* unos ejes auxiliares que actúan como si existieran estos nodos auxiliares. Cuando se desee "crear" un nodo auxiliar, se divide el eje sobre el que se crearía dicho nodo en dos ejes de los cuales se modifican sus campos *source* y *target*, haciéndole creer al algoritmo que entre esos dos ejes existe un nodo, del que simplemente se proporcionará un identificador falso, que posteriormente será el nodo origen / destino de la función de enrutamiento. Además, se debe copiar el resto de información del eje original a los 2 ejes auxiliares en los que se parte, exceptuando los costes, los cuales deberán ser modificados según la proporción del eje original que conformen.


Ilustración 24: Eje original con *source* 1 y *target* 2.

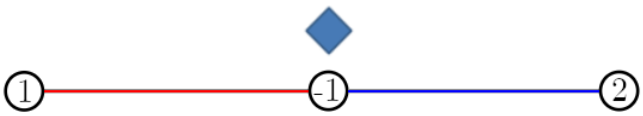



Ilustración 25: En rojo: Eje auxiliar con *source* 1 y *target* -1. En azul: Eje auxiliar con *source* -1 y *target* 2. En este ejemplo los costes de los ejes auxiliares serán la mitad del original.

Los ejes auxiliares se almacenan en la tabla ejes_auxiliares. Para utilizar su información en el enrutamiento se le debe añadir, mediante *union*, a la consulta *Edges SQL* los campos de la nueva tabla, quedándo *Edges SQL*:

```
'select id, source, target, cost_time as cost, rcost_time as reverse_cost from ejes union select id, source, target, cost_time as cost, rcost_time as reverse_cost from ejes_auxiliares'
```

En el ejemplo de la ilustración 25, el nodo origen sería el nodo -1. El mismo proceso aplica para el nodo auxiliar destino. Por cada nueva ruta, se añaden los ejes auxiliares a ejes_auxiliares y se borran de la tabla al finalizar el enrutamiento.

En este punto del desarrollo se obtiene la ruta óptima en carretera en vehículo sean cualquiera las coordenadas de origen y destino sobre la red de ejes, cumpliendo con los sentidos de circulación y restricciones de giro, pudiendo escoger la ruta más rapida (menos tiempo) o la más corta (menos longitud). La base de datos tiene mucha más información la cual se procederá a explotar a continuación.

Dentro de la función que enruta dado unas coordenadas, se va a permitir que el usuario pueda discriminar por el tipo de vehículo, restricción de acceso, pueda seleccionar evitar peaje o una zona dadas unas coordenadas y un radio. Todas estas opciones se especifican en el parámetro *SQL Edges*, de la misma manera que al principio se decidió evitar tramos peatonales, con sentencias *where* para solo enrutar sobre ejes que cumplan las condiciones deseadas.

Estas funcionalidades además de añadirse al enrutamiento con el método de nodos auxiliares se añaden también al enrutamiento con la función descontinuada de pgRouting mencionada previamente, que en el desarrollo del proyecto aún estaba en desarrollo, al no presentar ningún problema y ser más rápida que el método de los nodos auxiliares. Además de estos dos métodos, pgRouting también ofrece una función en beta, ahora descontinuada también, para poder seleccionar diferentes puntos (intermedios de ejes) para realizar una ruta que pase por todos ellos. Para esta última también se añade toda funcionalidad. Las 3 funciones calculan una ruta en alrededor de 2 segundos.

### 5.1.2. Servidor GeoServer

GeoServer [28] es una aplicación de software libre que permite la publicación y el acceso a datos geoespaciales. Está desarrollado en Java y está diseñado para funcionar con una amplia variedad de sistemas de información geográfica (SIG), bases de datos y formatos de datos geoespaciales. Es un servidor OGC (Open Geospatial Consortium) Compliant, lo que significa que cumple con estándares de la industria para el intercambio de datos geoespaciales.

En GeoServer, es posible realizar consultas a vistas SQL utilizando el servicio WFS (Web Feature Service). WFS [29] es un estándar para el intercambio de datos geoespaciales a través de internet, y en el contexto de GeoServer se puede utilizar para realizar consultas a las capas (layers) publicadas en el servidor y obtener los resultados en un formato compatible con la visualización de mapas.

Tras conectar el servidor con la base de datos, se publica la vista SQL deseada como una capa en GeoServer, siendo así posible realizar consultas a esa capa mediante el envío de una solicitud HTTP utilizando el método GET y especificando los parámetros de la consulta en la solicitud. En este caso, se crea una vista con una consulta a la función de enrutamiento que se desee especificando los siguientes parámetros:

- Las coordenadas de inicio y fin de la ruta.
- El vehículo con el que se realiza la ruta.
- Las restricciones de acceso de la ruta.
- Si se realiza la ruta más rápida o la más corta.
- Si se desea evitar peajes.
- Las coordenadas de la zona a evitar y su radio, si se quisiera.

Para la función de enrutamiento con nodos dinámicos `pgr_fromAtoB_dynamic`, con %*parámetro*%:

```
select * from pgr_fromAtoB_dynamic('%coordenadas_origen%', '%coordenadas_destino%',
%vehiculo%, ARRAY[%restricciones%], %coste%, %peaje%, ARRAY[%evita%], %radio%)
```

Que se llamaría con la siguiente solicitud:

```
GET http://localhost/geoserver/wfs?
  service=WFS&
  version=1.0.0&
  request=GetFeature&
  typeName=routing:pgr_fromAtoB_dynamic&
  outputFormat=application/json&
  viewparams=
       'coordenadas_origen':valor coordenadas_origen;
       'coordenadas_destino':valor coordenadas_destino;
       'vehiculo':valor vehiculo;
       'restricciones':valor restricciones;
       'coste':valor coste;
       'peaje':valor peaje;
       'evita':valor evita;
       'radio':valor radio;
```

Y devolvería la información de la ruta en formato GeoJSON, formato de archivo para almacenar y transmitir datos geoespaciales en formato de texto plano, basado en el formato de objetos de JavaScript (JSON).

### 5.1.3. Front-end web con OpenLayers

OpenLayers [30] es una biblioteca de código abierto para el desarrollo de aplicaciones de mapas web. Proporciona un conjunto de herramientas para la integración de mapas en la interfaz de usuario de una página web y permite la visualización y manipulación de datos geoespaciales, está escrito en JavaScript y es compatible con la mayoría de los navegadores web modernos.

Se desarrolla en HTML, CSS y JavaScript una página web que permita al usuario configurar los parámetros de la ruta, así como los puntos de inicio y fin además de la posibilidad de puntos intermedios o zonas a evitar. Según esta información se envía una petición a GeoServer que devuelve en GeoJSON la información de la ruta. A partir de esta información se muestra por pantalla la ruta, así como indicaciones de la misma con el tiempo y distancia a recorrer en total y por calles.

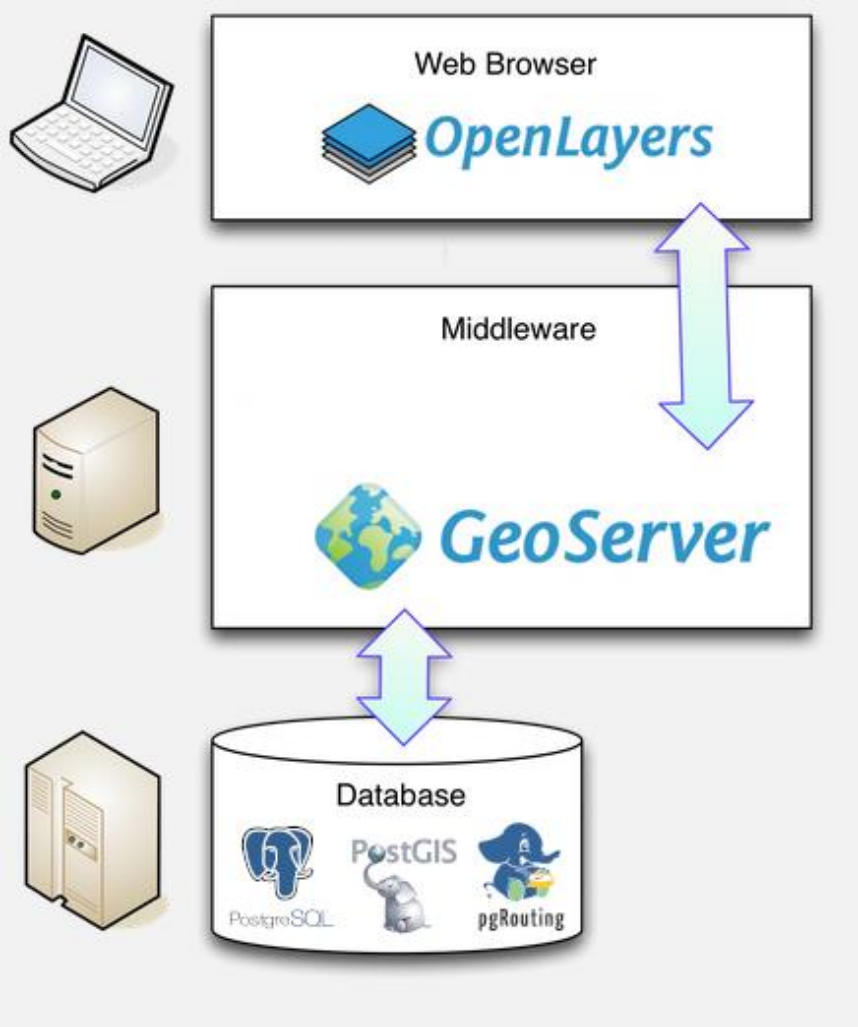



Ilustración 26: Arquitectura de la aplicación web. Fuente: Elaboración propia (editada) y erkancomez

En las siguientes ilustraciones se muestra la aplicación web en funcionamiento: La pantalla de acceso (ilustración 27) muestra el mapa completo de Navarra y un botón en la parte superior derecha que al ser presionado muestra el menú (ilustración 28) que permite modificar los parámetros de la ruta. Se navega sobre el mapa y al hacer click derecho en cualquier punto del mismo aparece el menú (ilustración 29) que ofrece las diferentes acciones que se pueden hacer sobre un punto. Al seleccionar los puntos de inicio y final de una ruta, esta se calcula y se muestra (ilustración 30) junto a información de las carreteras por las que circula y durante cuantos minutos. Al final de estas indicaciones se muestra el tiempo y distancia totales de la ruta. Además de seleccionar los puntos de inicio y final de una ruta, se permite seleccionar zonas a evitar (ilustraciones 31 y 32) y puntos intermedios (ilustraciones 33 y 34). Por último, (ilustraciones 35 y 36) se indica el tramo entre el punto seleccionado y su proyección sobre los ejes.


Ilustración 28: Vista inicial de la página web. Mapa base: IDENA.

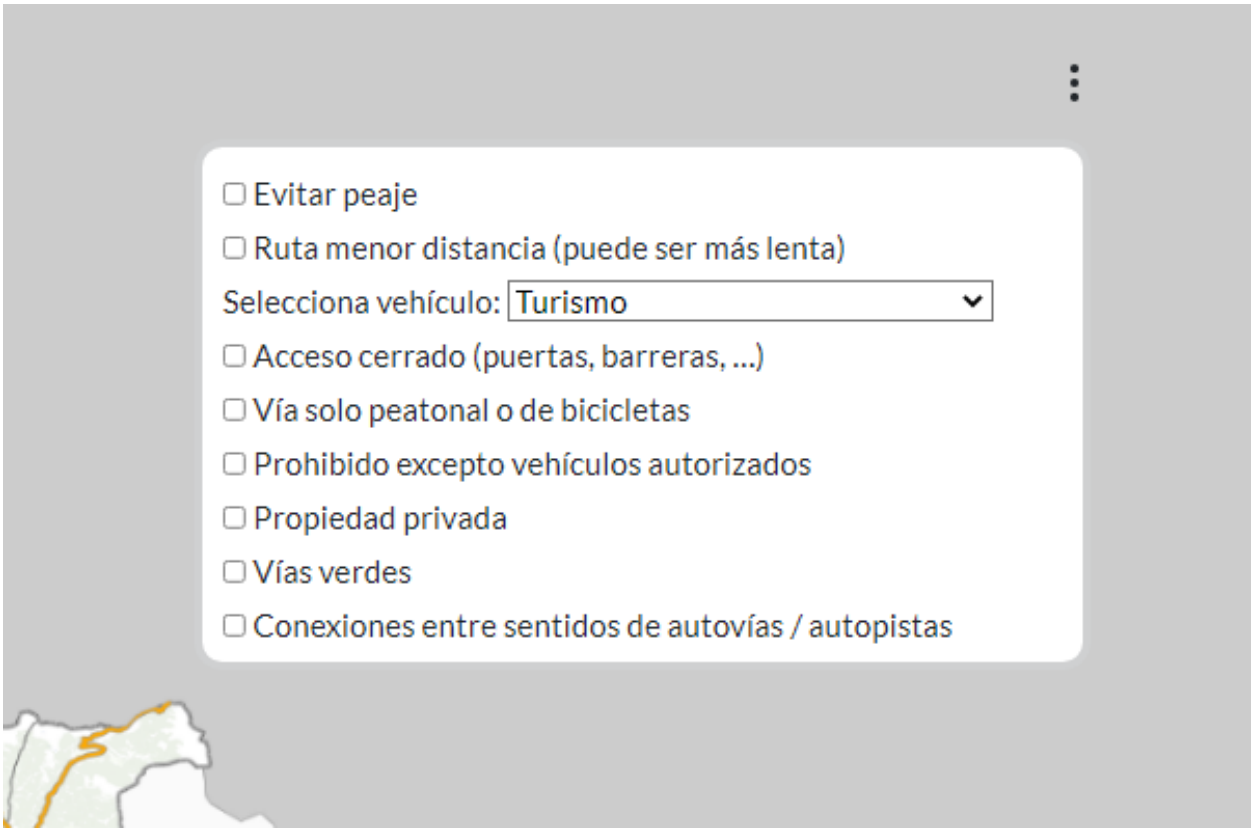



Ilustración 27: Menú desplegable de parámetros de ruta. Mapa base: IDENA.


Ilustración 29: Menú puntos de ruta (click derecho). Mapa base: IDENA.


Ilustración 30: Visualización de ruta con indicaciones. Mapa base: IDENA.

Ilustración 31: Visualización de zona a evitar. Mapa base: IDENA.

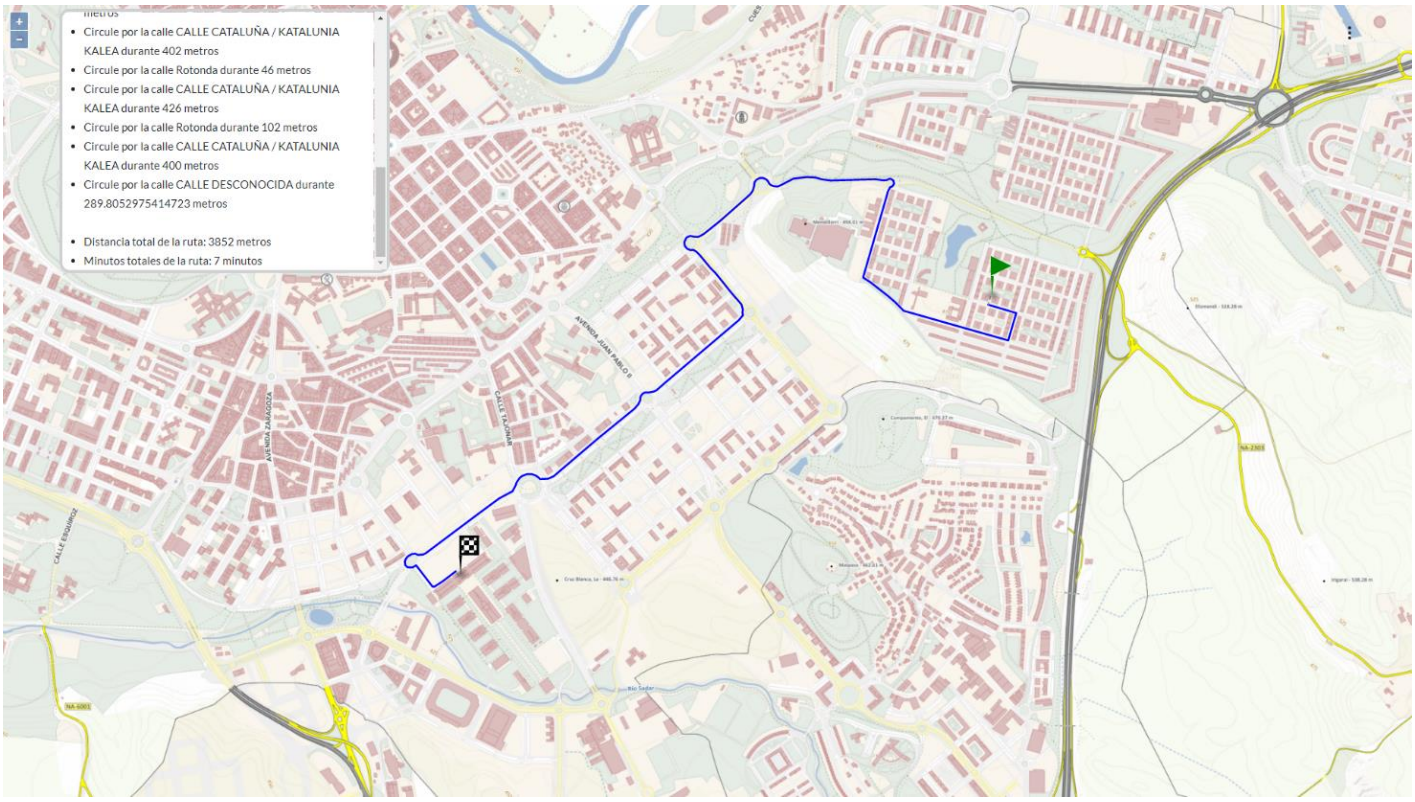



Ilustración 32: Ruta de la ilustración 30 evitando zona de la ilustración 31. Mapa base: IDENA.

Ilustración 33: Selección de punto intermedio de ruta. En gris: Punto seleccionado sobre el mapa. En azul: su proyección en la red de ejes. Mapa base: IDENA.


Ilustración 34: Ruta de la ilustración 30 pasando por el punto intermedio de la ilustración 32. Mapa base: IDENA.

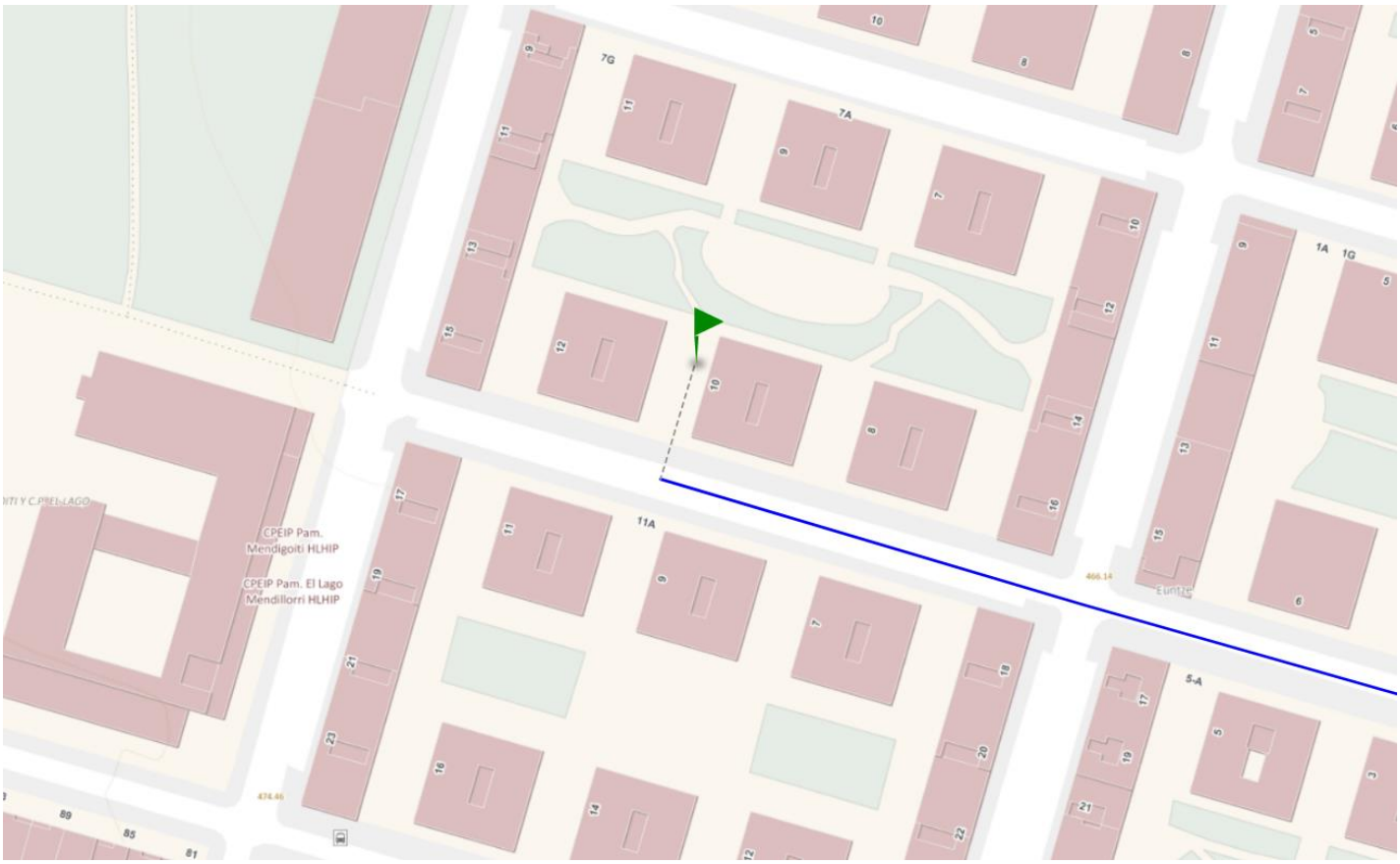



Ilustración 35: Punto de origen y su proyección en la red de ejes. Mapa base: IDENA

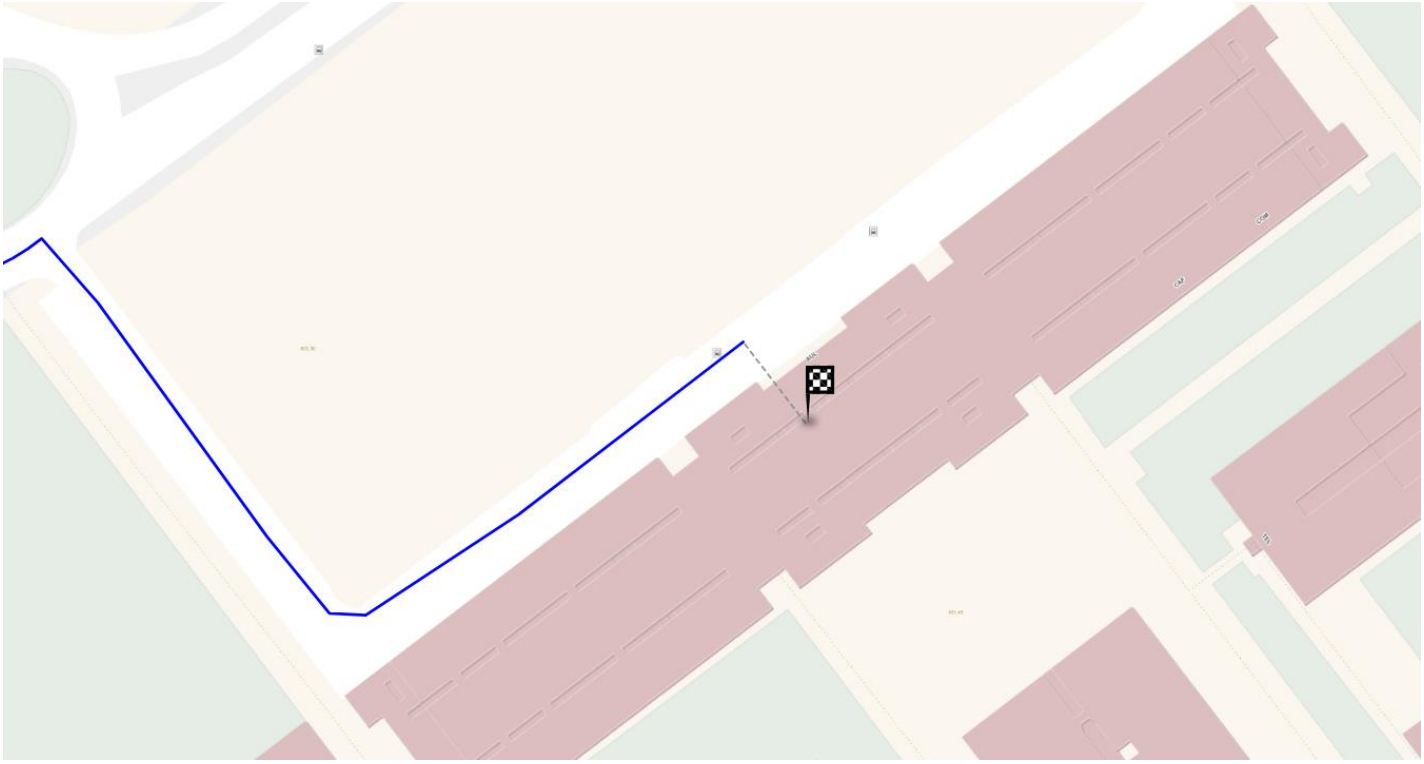


Ilustración 36: Punto de destino y su proyección en la red de ejes. Mapa base: IDENA

## 5.2 Migración a formato OSM y GraphHopper

Como se ha comentado previamente, existen multitud de aplicaciones de enrutamiento para datos OpenStreetMap. En este apartado se estudiará la migración de datos en PostGIS a formato OSM para poder explotarlos con una de estas aplicaciones.

La transformación de PostGIS a OSM no es trivial, se tratan de dos formatos muy dispares. La información espacial de los ejes se almacena como geometría en PostGIS y como XML de identificadores de nodos en OSM. Cada eje se transformará a una vía OSM.

Al ser OpenStreetMap prácticamente un estándar en el sector cartográfico de código abierto, existen aplicaciones para importar o exportar datos desde múltiples formatos, expuestas en su propia web.

| File type | Extension | To import into OSM | To export from OSM |
|---|---|---|---|
| Adobe Illustrator | .ai | no | osm2ai.pl |
| ArcGIS (by ESRI) | several | ExportToOSM.py | ArcGIS |
| E00 | e00 | partial | partial |
| Dafif | ? | dafif2osm | no |
| DXF (by AutoCad) | .dxf | DXF_import (JOSM plugin), Dxf2gpx | GeoConverter, Spatial Manager Desktop™, GDAL/OGR, QGIS, mygeodata cloud |
| GADM | ? | gadm2osm | no |
| Garmin maps | .img | no | OSM Map on Garmin |
| Geobase | ? | geobase2osm | no |
| GeoJSON | .geojson | osm-and-geojson, geojsontoosm, geojson2osm-es6 | OSM2GEO, osmtogeojson, Gdal ogr2ogr, osmium export |
| GeoPackage | .gpkg | no | Gdal ogr2ogr, osm-export-tool |
| GML | ? | gml2osm | XSLT, Python, GeoConverter |
| Google Earth (Keyhole) Markup Language | .kml | JOSM/Plugins/OpenData | OSM in Google Earth, Osm2KML, GeoConverter, osm-export-tool. |
| GPX | .gpx | Import as layer in main site, josm, gpx2csv->csv2osm | OGR, QGIS, GeoConverter |
| MP (Polish format) | .mp | mp2osm PFM2OSM | Osm2mp |
| Persistance of Vision Raytracer | .pov | no | |
| Shapefile | .shp | JOSM/Plugins/OpenData | GDAL, QGIS, GeoConverter, osm-export-tool |
| Smallworld (GE Smallworld datastore) | .ds | no | OpenStreetMap Plug-In |

Ilustración 37: Aplicaciones de importación y exportación de diferentes formatos a OSM. Fuente: OpenStreetMap

En primer lugar, se deberá transformar la información almacenada en la base de datos a un formato que disponga de una aplicación de importación a OSM. PostGIS ofrece funciones que exportan tablas con información espacial a una variedad de formatos, siendo los más populares GeoJSON, GML y KML. Además de transformar la información espacial (geometría), para poder explotar como se desea la base de datos, es necesario transformar más información (nombre, velocidad, sentido...) de los ejes. Esta información será almacenada como etiquetas OSM. Únicamente exportando a GeoJSON se pueden almacenar otros campos además de la geometría, por lo que este será el formato elegido. GeoJSON almacena la geometría como el conjunto de coordenadas que la componen.

Como se puede observar en la ilustración 37, OpenStreetMap ofrece tres aplicaciones, escritas en JavaScript, para importar un archivo GeoJSON a OSM:

- osm-and-geojson
- geojsontoosm
- geojson2osm-es6

La primera no soporta el tipo de geometría de los ejes, por lo que se prueban las otras dos y otra aplicación, geojson2osm, que aparece como *fork* de la primera de la lista. Tras la realización de pruebas, con porciones reducidas de los ejes, tan solo geojson2osm y geojsontoosm obtienen los resultados deseados, la segunda en mucho menor tiempo.

El principal problema que presentan estas aplicaciones es que ninguna está diseñada para una transformación de datos tan grande como la de la base de datos con la que se está tratando, con alrededor de unas 300.000 entradas. El caso de uso de estas aplicaciones es el de migrar pequeños archivos para poder subirlos a OpenStreetMap y la transformación de todos los datos duraría días. Esto no resulta una limitación para el proyecto, ya que es un proceso que se realiza tan solo una vez y se entiende que tratar una cantidad de datos tan grande es un proceso costoso.

Además de necesitar transformar la información de los ejes, para realizar un correcto enrutamiento es necesario tener en cuenta las restricciones de giro, por lo que también se deberá transformar esta información. No hay ninguna herramienta que realice esta transformación por lo que será necesario desarrollar una.

Como ya se ha visto, en la base de datos las restricciones de giro se almacenan como los identificadores de los ejes destino y origen de la restricción de giro además del coste de tomarlo, mientras que en OSM se almacenan como una relación cuyos miembros son los identificadores de las vías destino y origen, el nodo (o vía) a través del cual se toma el camino restringido y el tipo de restricción de giro.

OpenStreetMap soporta diferentes tipos de restricciones de giro según su dirección: Prohibido girar a la izquierda, derecha, seguir de frente o cambiar de sentido. Esto presenta un problema en la migración de datos ya que en la base de datos no se almacena esta información de las restricciones de giro, y no todas surgen de estas prohibiciones de dirección, en muchos casos una restricción de giro indica una prohibición de acceder a un eje a través de otro, sin tener que ser una de estas 4 restricciones. Además de estas, OpenStreetMap ofrece, de manera ambigua, la restricción de prohibición de acceso, apuntando que a día de hoy no hay consenso para su uso, pero que la mayoría de las aplicaciones de enrutamiento la soportan. Etiquetar todas las restricciones de giro como de prohibición de acceso, pese a ser información menos rigurosa (se etiquetan todas las restricciones igual, aunque realmente se traten de prohibiciones de giro a izquierda, por ejemplo), solventaría el problema de migrar las restricciones sin conocer la dirección del giro.

La transformación de las restricciones de giro presenta otros dos problemas: En primer lugar, en el proceso de transformar los ejes (Postgis) a vías (OSM) no se mantiene el identificador del eje como identificador de la vía, necesario para almacenar la restricción, y, en segundo lugar, no se tiene la información del nodo a través del cual se toma el camino restringido.

En la transformación de GeoJSON a OSM, en primer lugar, se crean nodos por cada coordenada que forme un eje, ya que, en OSM, una vía la componen nodos, no coordenadas, por lo que se transforman las coordenadas a nodos cuyo identificador se asigna de manera secuencial: es información que se genera en el proceso y no existe en la base de datos. A continuación, se procesa cada eje creando una vía con sus nodos correspondientes, a la cual también se le asigna un identificador que no se conoce.

Como se ha comentado, el archivo GeoJSON que se transforma almacena más información de los ejes que la espacial. Para poder transformar las restricciones de giro a OSM es necesario poder obtener el identificador de la vía OSM a partir del identificador del eje en la base de datos. Por lo que se almacena en el GeoJSON dicho identificador de cada eje y se crea un diccionario en el script de transformación de datos, para así, partiendo de los identificadores de los ejes que forman una restricción de giro, poder obtener los identificadores de las respectivas vías OSM. Para obtener el nodo a través del cual se toma el camino, se crea otro diccionario que, en el proceso de asignar a una vía sus nodos, almacene los identificadores de estos. Así, se realiza la traducción de los identificadores de los ejes que forman la restricción de giro en PostGIS a los identificadores de sus vías OSM correspondientes, y se toma como nodo de la restricción OSM la intersección de los nodos de ambas vías, que será el nodo a través del cual se toma el camino de una a otra.

De esta forma, modificando la aplicación de migración de GeoJSON a OSM, se obtiene la información de los ejes deseada además de las restricciones de giro en formato OSM. Además de esto, se optimiza dicha aplicación para minimizar el tiempo de ejecución. Hasta este momento, se han realizado todas las pruebas con zonas reducidas del mapa de la base de datos, por el elevado coste de procesar los datos.

A continuación, con un proceso de transformación que almacena toda la información deseada en OSM, se procede a probar una de las aplicaciones que enrutan sobre estos datos: GraphHopper.

GraphHopper [31] es una aplicación open-source de enrutamiento escrita en JAVA. Entre otras funcionalidades, implementa los algoritmos de enrutamiento Dijksta y A*, ambas en sus versiones bidireccionales. Soporta las restricciones de giro y enrutamiento para diferentes vehículos, destaca por su velocidad y uso eficaz de la memoria y ofrece enrutamiento a través de peticiones HTTP a su API o mediante una interfaz web. Es por esto último, además de por ser una de las aplicaciones más populares y con mayor soporte, que se ha seleccionado, al incorporar un front-end para realizar pruebas de enrutamiento.

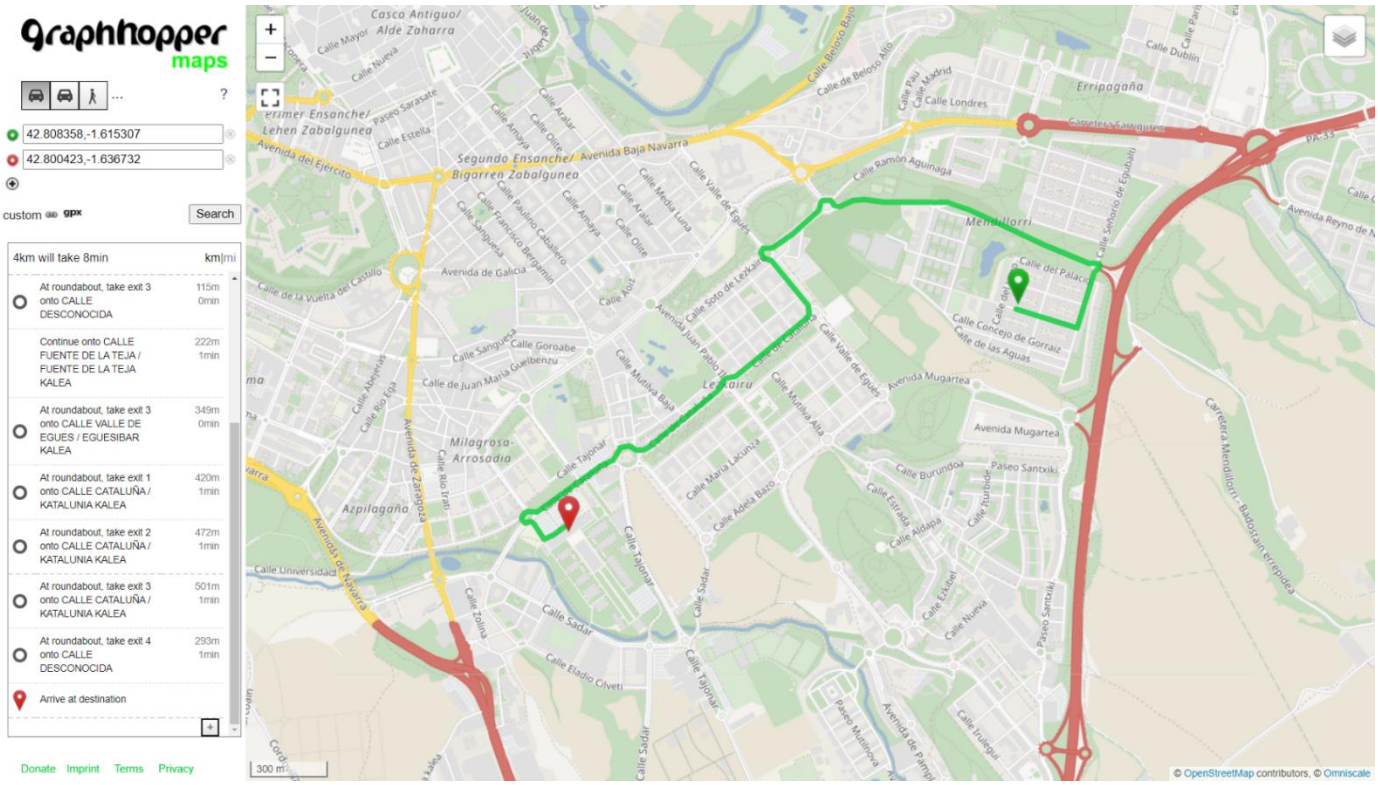



Ilustración 38: Ruta utilizando GraphHopper.

A través de diferentes pruebas, se comprueba que GraphHopper calcula las rutas de manera muy rápida y es muy eficiente en memoria. Además, presenta indicaciones detalladas. Se observa también un problema que presenta la migración de datos: se realizan rutas que no deberían ser posibles a través de vías a diferente nivel, como se observa en la ilustración 39.

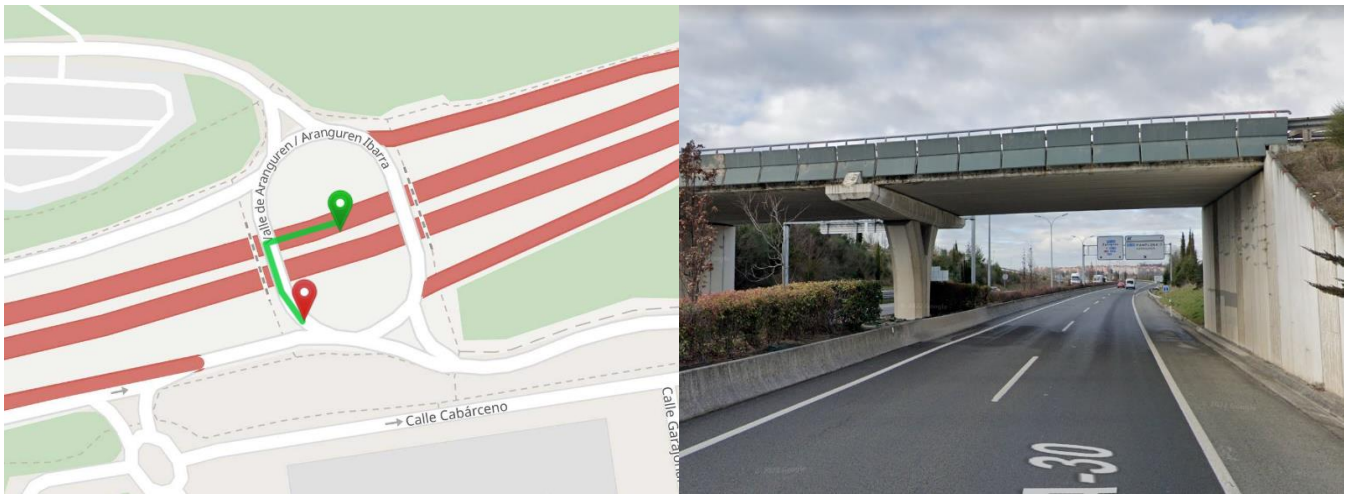



Ilustración 39: Ruta utilizando GraphHopper que circula por vías de distinto nivel.

Esto se debe a que, contrario a pgRouting, donde los nodos son el inicio y final de los ejes, en OSM todas las coordenadas que forman una geometría son nodos, por lo que, si dos vías comparten una coordenada en su geometría, comparten un nodo, luego OSM lo considerará una intersección de vías pese a que estén a distinto nivel. La solución a este problema es modificar el script de transformación de datos para que a la hora de asignar los nodos que conforman una vía, no añadir aquellos nodos intermedios que ya formen parte de otra vía. Esto es posible ya que la aplicación de transformación no crea dos nodos diferentes para la misma coordenada, por lo que en estos casos el mismo nodo intermedio es parte de dos vías. Esta solución significa una pérdida despreciable de información ya que perder un nodo no altera de manera significativa la geometría de una vía.

Ahora sí, con toda la información deseada en formato OSM de manera que las rutas se realizan correctamente, se procede a realizar la migración de todos los ejes. Tras días de ejecución, el script encargado de transformar el GeoJSON deja de funcionar debido a un error de memoria. Los objetos que estaba intentando procesar son demasiado grandes para ser procesados en JavaScript, por lo que era necesario buscar una alternativa a los programas ofrecidos en la web de OpenStreetMap, todos escritos en este lenguaje.

JOSM (Java OpenStreetMap Editor) [32] es una aplicación de software libre programada en Java para la edición de datos OpenStreetMap. Esta aplicación permite transformar, entre otros, archivos GeoJSON a OSM (pese a no ofrecerse como tal en la ilustración 37). Al contrario que utilizando el script, el cual se puede modificar de forma sencilla, por ejemplo, para eliminar nodos en intersecciones no deseadas o crear restricciones de giro, JOSM realiza este proceso de manera interna. Al ser un programa de código abierto sería posible modificar este proceso, pero resultaría una solución más compleja, por lo que se opta por realizar los cambios necesarios directamente al archivo OSM.

Como se ha comentado, el formato OSM es un XML. Se desarrolla en Python con librerías que permiten parsear XML un programa para resolver los problemas que presentaba la migración: Intersecciones no deseadas y no contar la información necesaria para crear las restricciones de giro. Para el primer problema se realiza la misma solución presentada previamente, de eliminar nodos intermedios de vías que ya hayan sido utilizados como nodos de otras, y para la segunda se toma una solución más sencilla: En vez de realizar un diccionario para traducir los identificadores de PostGIS a identificadores de vías OSM, se modifica el identificador de la vía OSM para que tome su equivalente en PostGIS, que estaba almacenado como una etiqueta de la vía. Por último, se vuelve a crear un diccionario de los nodos de una vía para obtener el nodo necesario para escribir la restricción de giro.

Así, transformando el archivo GeoJSON a OSM utilizando JOSM y tratando posteriormente el archivo OSM se obtienen todos los ejes junto a sus restricciones de giro. Este proceso resulta ser muy rápido y en cuestión de minutos se realiza la migración de unos 300.000 ejes con sus respectivas restricciones. El enrutamiento en GraphHopper con toda la base de datos transformada sigue siendo igual de rápido y eficiente en memoria que las pruebas realizadas con datos reducidos.

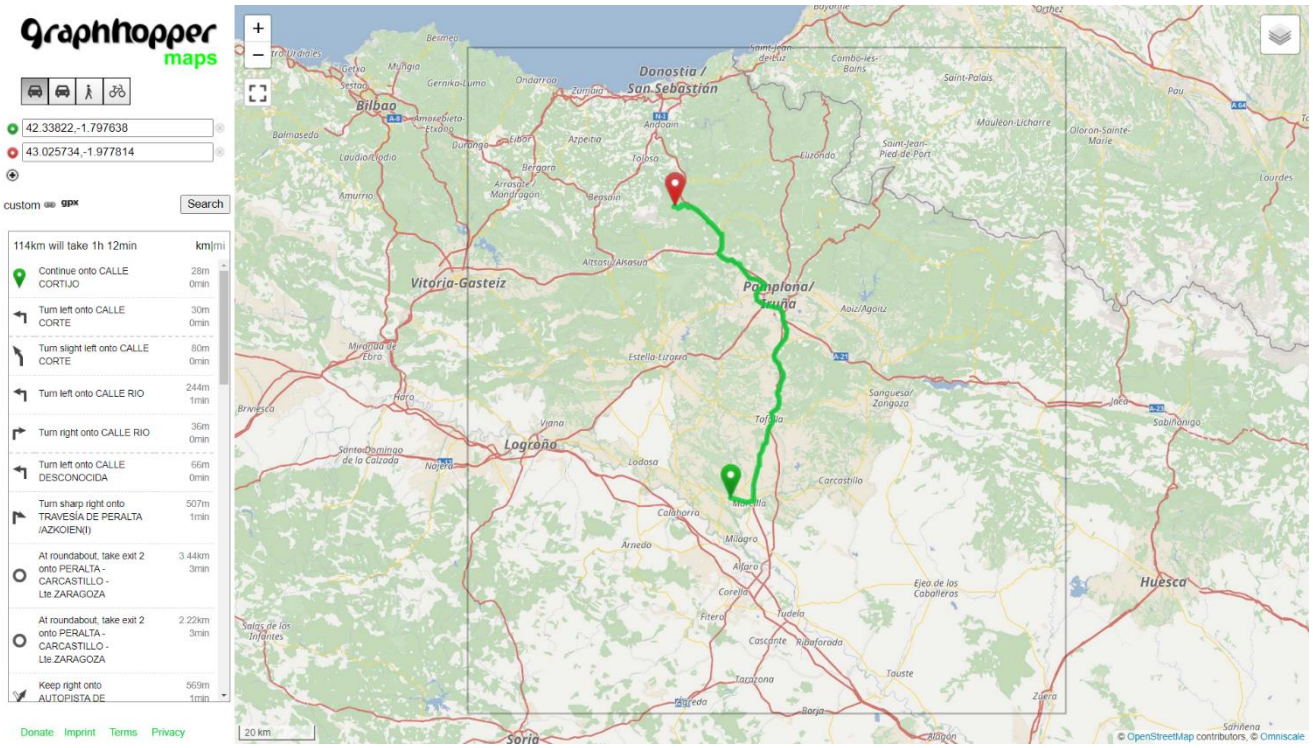



Ilustración 40: Ruta utilizando GraphHopper sobre todos los ejes originales.

Se procede, a continuación, a añadir más funcionalidades al enrutamiento con GraphHopper, como el de permitir enrutar con diferentes vehículos. La aplicación ofrece esta funcionalidad, pero no permite utilizar la información de la etiqueta *vehiculo* de una vía, campo que en la base de datos indica qué vehículo puede circular por un eje. Únicamente permite crear diferentes vehículos que circulen por vías según cierta información de la vía [33], principalmente de etiquetas estandarizadas de OSM, la etiqueta *vehiculo* no lo es. De nuevo, al ser una aplicación de código abierto podría modificarse para dar soporte a la etiqueta *vehiculo*, pero ese no es el objetivo de este proyecto. Por el mismo motivo, tampoco ofrece GraphHopper la posibilidad de discriminar ejes por sus restricciones de acceso, de nuevo, almacenadas en el archivo OSM como etiquetas no estandarizadas. Este problema surge de intentar explotar toda la información de unos ejes que no fueron desarrollados originalmente para ser utilizados en formato OSM.

## 5.3 Aplicación web con ol-ext

Esta alternativa es, sin duda, la más sencilla de implementar. ol-ext, extensión de OpenLayers, implementa el algoritmo de Dijkstra en JavaScript, haciendo posible el enrutamiento en cliente sin necesidad de un back-end.

Ilustración 41: Ruta con ol-ext. Fuente: ol-ext

El mapa sobre el que se realiza el enrutamiento se almacena en un archivo GeoJSON. Al ser una aplicación JavaScript diseñada para ser ejecutada en navegador, este enrutamiento es el que menos funcionalidades ofrece, pero al tratarse de una alternativa tan sencilla sigue siendo relevante para el proyecto. La implementación de Dijkstra de ol-ext soporta ejes de uno y dos sentidos, y permite personalizar el coste de cada uno: esta sería toda la información que se puede explotar para la creación de una ruta, aunque, de nuevo, como se trata de una aplicación de código abierto, es posible desarrollar nuevas funcionalidades.

Se exporta desde PostGIS un archivo GeoJSON de los ejes con los campos deseados, en este caso, *sentido* y *cost_time*. Al no permitir este enrutamiento seleccionar vehículos o restricciones de acceso, se decide exportar únicamente los ejes transitables por turismos y de acceso público. Para la realización de pruebas se descarga la página web ejemplo de la documentación mostrada en la ilustración 41 y se especifica este archivo GeoJSON como fuente del mapa.

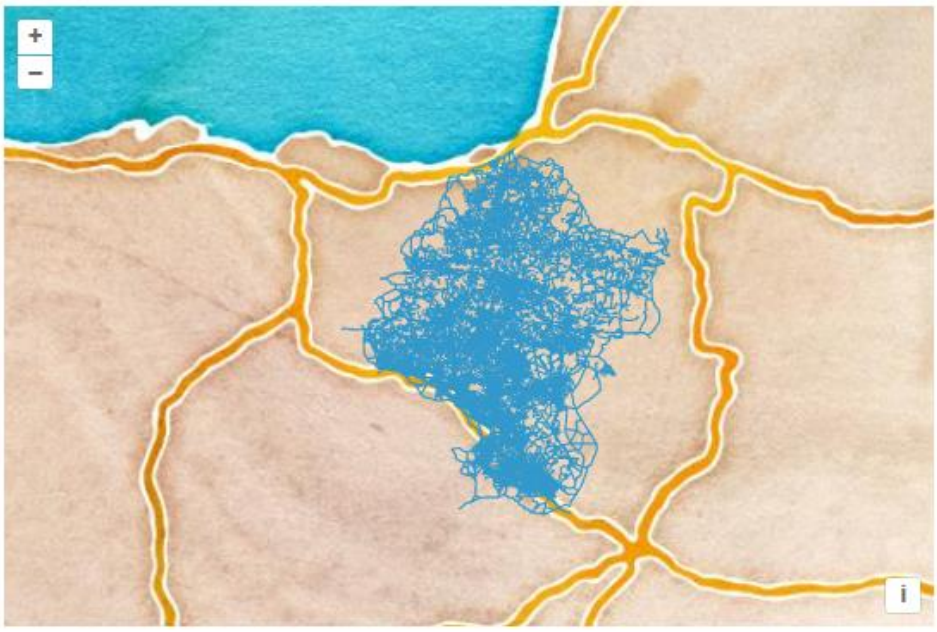


Ilustración 42: Mapa de la ilustración 41 con los ejes de la base de datos.

Con el GeoJSON de ejes cargado, la interacción con el mapa se vuelve extremadamente lenta. Navegar sobre él o hacer zoom llevan segundos en completarse, haciendo la página web inviable. Esto se debe, a que esta aplicación no está diseñada para una red tan compleja. El archivo de la red utilizada para el ejemplo de la ilustración 41 pesa, aproximadamente, unos 10 megabytes, 10 veces menos que la de los ejes de PostGIS.

Se intenta, a continuación, reducir el tamaño del archivo que almacena la red. Se eliminan todos los espacios del archivo que lo almacena y se reducen los nombres de los campos de coste y sentido a una letra, minimizando el tamaño de este manteniendo la totalidad de la información.

Esta reducción de peso no es suficiente para lograr que la web sea usable, así que se procede a reducir la precisión de las coordenadas, que cuentan con 9 decimales. Para el objetivo de la página web, que no es el de mostrar un mapa riguroso y detallado, no es necesaria tanta precisión. De esta manera, la reducción de tamaño es notable, pero sigue sin ser suficiente para que la interacción con la web sea aceptable, ni suprimiendo tanto detalle que la información deja de ser correcta, como se observa en la ilustración 43, en la que los ejes han perdido su localización original. Como se ha comentado, la red tan solo cuenta con los ejes de acceso público transitables por turismo, por lo que reducir aún más el número de ejes no es una opción.

Siendo demasiado pesado para la utilización de la web la red de ejes mínimos y con información imprecisa, se descarta ol-ext como alternativa a desarrollar una web para el cálculo de rutas sobre la cartografía de la base de datos.

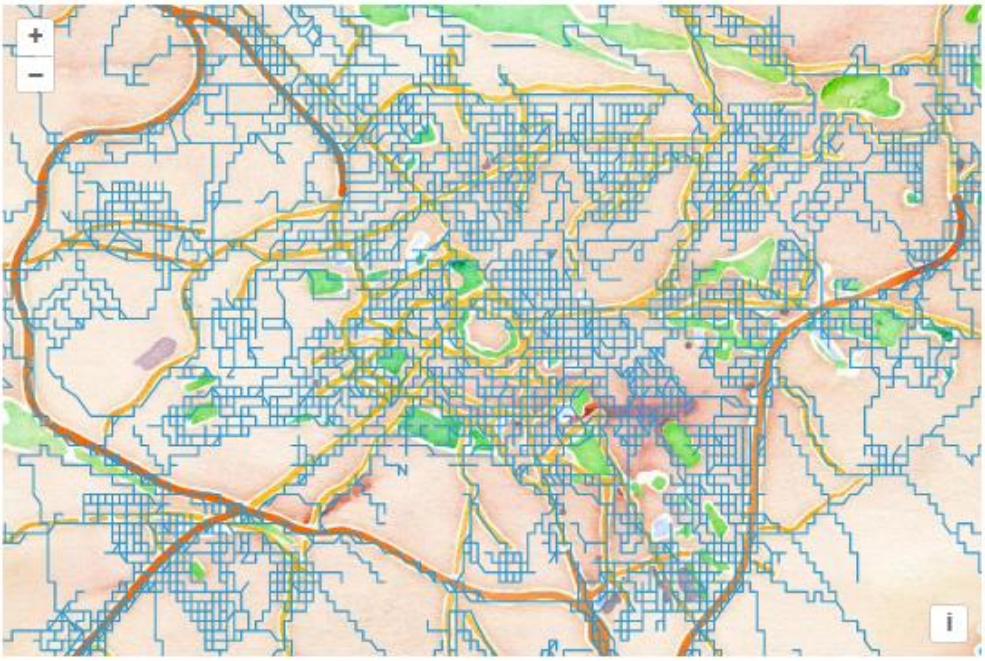


Ilustración 43: Red de ejes con coordenadas de 3 decimales.

# 6. Conclusión y líneas futuras

Se han desarrollado dos aplicaciones web que cumplen con el objetivo de ofrecer un servicio web de cálculo de rutas sobre cartografía propia utilizando únicamente software de código abierto de manera exitosa, ambas con la posibilidad de explotar la información de la base de datos.

La realizada con pgRouting permite realizar rutas según una gran variedad de parámetros y opciones. Sobre esta, al tratar directamente sobre la base de datos, se podría añadir aún más opciones que permitan hacer uso de más información, como, por ejemplo, realizar rutas únicamente sobre vías pavimentadas, o de doble sentido, no circular por túneles, etc. Las opciones son tantas como información se tenga de los ejes. De la misma aplicación web, se podría añadir indicaciones más detalladas de la ruta, como hace GraphHopper, mostrando los giros y las salidas de las rotondas. Para realizar esto se podría analizar la geometría de los ejes que forman una ruta y obtener los grados de giro de ejes sucesivos. También, se podría mejorar la interacción de la página web, añadiendo una barra de búsqueda para no tener que localizar en el mapa las ubicaciones. Esta funcionalidad se estudió implementar, utilizando el buscador que ofrece el visor de IDENA, pero se descartó por los problemas que suponían integrar diferentes librerías que mostraron incompatibilidades.

Acerca de la migración al formato OSM, esta ofrece una gran variedad de aplicaciones. A partir de las herramientas desarrolladas para este proyecto es posible utilizar grandes bases de datos PostGIS, incluso con restricciones de giro, para cualquier servicio que soporte datos OSM. En este proyecto se han realizado pruebas de enrutamiento con GraphHopper, el cual, al ser un programa de código abierto, puede ser modificado para explotar toda la información deseada, por ejemplo, para añadir diferentes vehículos o más parámetros de ruta, y muestra un gran rendimiento y eficiencia.

Resulta más sencilla de integrar la aplicación desarrollada con PostGIS y pgRouting, al crearse las rutas a través de peticiones web estandarizadas, que brindan interoperabilidad y la posibilidad de ser utilizada por terceros, mientras que GraphHopper utiliza peticiones propias de la aplicación. Por otra parte, el enrutamiento sobre el dato transformado a formato OpenStreetMap es más rápido y eficiente, además de ofrecer multitud de posibilidades debido a la gran variedad de aplicaciones desarrolladas sobre OSM, como la estudiada en este proyecto, GraphHopper, la cual ofrece indicaciones de navegación detalladas.

En mi opinión, la aplicación sobre pgRouting es la mejor alternativa por los beneficios de utilizar peticiones estandarizadas que, pese a ser menos rápida, permite explotar toda la información de la base de datos de manera más sencilla y cuya funcionalidad puede ser extendida en el futuro. La migración a dato OSM abre nuevas posibilidades que van más allá de este proyecto, como el desarrollo de aplicaciones móviles o sin acceso a internet.

# 7. Bibliografía


[1] L. Ceci, "Most popular navigation apps in the U.S. 2021, by downloads", *Statista*, 2021. [En línea]. Disponible en: https://www.statista.com/statistics/865419/most-popular-us-mapping-apps-ranked-by-reach/

[2] "Software libre y Software privativo", *UPV*, 2020. [En línea]. Disponible en: https://www.upv.es/entidades/I2T/info/890613normalc.html

[3] "Software Propietario Y Software Libre: ¿Qué Debemos Saber?", *Ceesa*, 2021. [En línea]. Disponible en: https://ceesa.com/software-propietario-y-software-libre/

[4] "Software libre y de código abierto", *Wikipedia*, 2022. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Software_libre_y_de_código_abierto

[5] "Google Maps Platform Terms of Service", *Google*, 2020. [En línea]. Disponible en: https://cloud.google.com/maps-platform/terms

[6] "PostgreSQL: The World's Most Advanced Open Source Relational Database", *PostgreSQL*, 2022. [En línea]. Disponible en: https://www.postgresql.org/

[7] Leah Wasser, Chris Holdgraf, "Lesson 1. What is Raster Data", *EarthLab*, 2020. [En línea]. Disponible en: https://www.earthdatascience.org/courses/use-data-open-source-python/intro-raster-data-python/fundamentals-raster-data/

[8] Priya Pedamkar, "Raster Data", *educba*. [En línea]. Disponible en: https://www.educba.com/raster-data/

[9] "Introduction to Raster Data", *datacarpentry*. [En línea]. Disponible en: https://datacarpentry.org/organization-geospatial/01-intro-raster-data/

[10] "Introduction to Vector Data", *datacarpentry*. [En línea]. Disponible en: https://datacarpentry.org/organization-geospatial/02-intro-vector-data/index.html

[11] Leah Wasser, Chris Holdgraf, "Lesson 1. GIS in Python: Introduction to Vector Format Spatial Data - Points, Lines and Polygons", *EarthLab*, 2020. [En línea]. Disponible en: https://www.earthdatascience.org/courses/use-data-open-source-python/intro-vector-data-python/spatial-data-vector-shapefiles/

[12] *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture*, ISO 19125 SWG, 2011. [En línea]. Disponible en: https://www.ogc.org/standards/sfa

[13] "4.1. GIS Objects", *PostGIS*. [En línea]. Disponible en: https://postgis.net/docs/manual-2.0/using_postgis_dbmanagement.html#RefObjectgeometr

[14] "About PostGIS", *PostGIS*. [En línea]. Disponible en: https://postgis.net/

[15] "8. Coordinate Reference Systems", *QGIS*. [En línea]. Disponible en: https://docs.qgis.org/3.22/en/docs/gentle_gis_introduction/coordinate_reference_systems.html

[16] “Coordinate Reference Systems”, *datacarpentry*. [En línea]. Disponible en: https://datacarpentry.org/organization-geospatial/03-crs/index.html

[17] “About OpenStreetMap”, *OpenStreetMap*. [En línea]. Disponible en: https://www.openstreetmap.org/about

[18] “History of OpenStreetMap”, *OpenStreetMap*, 2022. [En línea]. Disponible en: https://wiki.openstreetmap.org/wiki/History_of_OpenStreetMap

[19] “Elements”, *OpenStreetMap*, 2023. [En línea]. Disponible en: https://wiki.openstreetmap.org/wiki/Elements

[20] “Dijkstra's algorithm”, *Wikipedia*, 2023. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[21] “A* search algorithm”, *Wikipedia*, 2022. [En línea]. Disponible en: https://en.wikipedia.org/wiki/A*_search_algorithm

[22] “About pgRouting”, *pgRouting*. [En línea]. Disponible en: https://pgrouting.org/

[23] “pgr_createTopology”, *pgRouting*. [En línea]. Disponible en: https://docs.pgrouting.org/3.1/en/pgr_createTopology.html

[24] “ol-ext”, *GitHub*, 2022. [En línea]. Disponible en: https://github.com/Viglino/ol-ext

[25] “pgr_dijkstra”, *pgRouting*. [En línea]. Disponible en: https://docs.pgrouting.org/3.1/en/pgr_dijkstra.html

[26] “pgr_trsp - Turn Restriction Shortest Path (TRSP)”, *pgRouting*. [En línea]. Disponible en: https://docs.pgrouting.org/3.1/en/pgr_trsp.html

[27] Toni Hernández Vallés, “Cálculo de rutas òptimas con pgRouting en entornos web”, *DUGiMedia*, 2022. [En línea]. Disponible en: http://diobma.udg.edu/handle/10256.1/6852

[28] “What is Geoserver?”, *GeoServer*. [En línea]. Disponible en: https://geoserver.org/about/

[29] “Web Feature Service”, *Open Geospatial Consortium*, 2014. [En línea]. Disponible en: https://www.ogc.org/standards/wfs

[30] “OpenLayers”, *GitHub*, 2022. [En línea]. Disponible en: https://github.com/openlayers/openlayers

[31] “GraphHopper”, *GitHub*, 2022. [En línea]. Disponible en: https://github.com/graphhopper/graphhopper

[32] “Java OpenStreetMap Editor”, *JOSM*, 2022. [En línea]. Disponible en: https://josm.openstreetmap.de/

[33] “Custom Models”, *GitHub*, 2022. [En línea]. Disponible en: https://github.com/graphhopper/graphhopper/blob/master/docs/core/custom-models.md