



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AYATORI: CREACIÓN DE MÓDULO BASE PARA PROGRAMAR ALGORITMOS DE
PLANIFICACIÓN DE RUTAS EN PYTHON USANDO GTFS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

FELIPE IGNACIO LEAL CERRO

PROFESOR GUÍA:
EDUARDO GRAELLS GARRIDO

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYAN
HERNÁN SARMIENTO ALBORNOZ

SANTIAGO DE CHILE
2023

Resumen

El presente informe detalla la creación de un módulo en Python para programar algoritmos de planificación de rutas, utilizando la información del transporte público disponible en formato GTFS (General Transit Feed Specification), en el contexto del desarrollo de una Memoria para optar al título de Ingeniero Civil en Computación. La motivación principal es crear una herramienta base que permita desarrollar algoritmos que aporten en el estudio de planificación urbana y de transporte. Para evaluar la utilidad y correcta implementación de esta solución, se implementó una versión simplificada de Connection Scan Algorithm, un algoritmo que utiliza la información en GTFS para calcular la mejor ruta para ir desde un punto A hasta un punto B. De esta implementación, se concluye que la herramienta creada funciona como se esperaba, cumpliendo exitosamente su objetivo.

Dijiste que tenías un sueño, y ahora... ¡se cumplirá! ¡Los sueños y los ideales tienen poder para cambiar el mundo!

-N.

Agradecimientos

A mi mamá, por enseñarme a estudiar, preocuparte por mi futuro, y darme una razón para salir adelante a pesar de todo. A mi papá, por todo tu amor, apoyo y respeto, por enseñarme a priorizar mi vida, por todos tus años de servicio como padre viudo, por nunca dejar de cuidarme, y ser mi ejemplo a seguir. Al amor de mi vida, por ser mi apoyo y compañía principal, por ayudarme a extender las fronteras de mis sueños y esperanzas, por dejarme estar en tu vida, darme alegría y luz en mis peores momentos, reírte de mis chistes, y todo tu amor incondicional.

Gracias Pauli, por amar a mi padre y darle la oportunidad de estar de nuevo felizmente casado, por tu amor y preocupación, y extender lo que entiendo por ‘familia’. Gracias Ita, Renata y Felipe, por nuestra mutua adopción familiar y convertirse en mi abuela y hermanos. A mis tatas, Daniel y Pechita, por sentar las bases familiares que inspiraron mis valores y moral, por consentirme y preocuparse de mí aunque la distancia nos separe. A mi tía Helen, por su amor, apoyo, y preocupación constantes por mi bienestar. A mi tío Leo, por todas las risas y anécdotas que me ayudaron a apreciar la sobremesa en familia y los consejos de los mayores. A mis primos: Amalia, Cristobal, Benja, Naty y Bastian, por su amistad, amor, apoyo, y alegrías varias. Gracias especiales a Nico, mi hermano del alma. Gracias a todo el resto de mi familia extendida por su cariño y buenos deseos. Gracias, tío Alvaro y tía Katy, por aceptarme como su yerno, por su preocupación y cariño, por darme una segunda familia, y por otorgarme la posibilidad de seguir trabajando en mi sueño cuando más lo necesité. Gracias, Florencia y Julieta, por enseñarme a ser un hermano mayor. A toda la familia Luna, por aceptarme como uno más entre los suyos, y por todo el cariño, consejos, y buenas vibras.

A mi curso, 12°B, por acompañarme en la primera parte de mi vida y por los amigos que encontré en ustedes. Gracias a Anime no Seishin Doukoukai, por darme la oportunidad de adquirir responsabilidades incluso con mis hobbies, y por todos los grandes amigos que me permitió hallar. Gracias a Ivancito, Kurisu, Gus y Chelo, por ser mi apoyo en los llantos y mi compañía en las celebraciones. Gracias Gabi, Julio, Gabo, Sofi, Naise, por su amistad. Gracias a Basti, Lucho y Seba, por ser mis primeros amigos en el mundo exterior y mantenerse a mi lado hasta el día de hoy. Gracias a todos aquellos compañeros de carrera con los que he podido compartir y colaborar al ir educándome, destacando especialmente a mi amigo Matías Vergara, y a todos los miembros de Team Michil.

Gracias a todas mis profesoras y profesores, por darme las herramientas para llegar a donde estoy hoy. Gracias, profesor Eduardo, por guiarme en este proceso.

Tabla de Contenido

1. Introducción	1
1.1. Objetivos	3
2. Estado del Arte	4
2.1. OpenStreetMap	4
2.1.1. OSM integrado en algoritmos	5
2.2. GTFS	7
2.2.1. GTFS integrado en algoritmos	10
2.3. Datos: estructura y manejo de la información	10
2.4. Connection Scan Algorithm: un algoritmo de planificación de rutas	11
2.4.1. Utilidad como caso de prueba	13
3. Diseño	14
3.1. Stack tecnológico	14
3.2. Funcionamiento lógico	15
3.2.1. OSMGraph: la clase de OSM	15
3.2.2. GTFSDData: la clase de GTFS	16
3.2.3. Funcionalidades entre clases	17
3.3. Criterio de Evaluación	17
4. Implementación	19
4.1. Clases y métodos	19

4.1.1. Procesamiento de OpenStreetMap	19
4.1.2. Procesamiento de datos en GTFS	22
4.1.3. Funcionalidades de GTFS sobre OSM	26
4.2. Creando un algoritmo	27
5. Resultados	30
5.1. Ejemplos de uso y caso de estudio	30
5.1.1. Mapeo de recorridos	31
5.1.2. Análisis de densidad en paradas	33
5.1.3. Algoritmo de enrutamiento: Connection Scan Algorithm	35
5.2. Evaluación de resultados	38
6. Discusión	39
6.1. Implicancias	39
6.2. Limitaciones	40
6.3. Trabajo Futuro	40
7. Conclusión	42
Bibliografía	44
Anexos	45
Apéndice A. Código de la solución	45
A.1. Clases del módulo	45
A.1.1. OSMGraph	45
A.1.2. GTFSData	50
A.2. Algoritmo de ejemplo: Connection Scan Algorithm	67

Índice de Ilustraciones

2.1. Mapa de la Región Metropolitana en OpenStreetMap. Fuente: openstreetmap.org	5
2.2. Mapa de la ciudad de Helsinki, Finlandia, que destaca sus puntos de interés. Fuente: pyrosm.readthedocs.io	6
2.3. Diagrama relacional de los archivos que componen el formato GTFS. Fuente: medium.com	7
2.4. Diagrama de uso de datos en tiempo real en formato GTFS para una aplicación. Fuente: watrifeed.ml	9
2.5. Diagrama explicativo del funcionamiento de Connection Scan Algorithm. Fuente: “Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys”(R. Kujala et al., 2017), vía sciedirect.com	11
2.6. Posibles caminos para llegar desde FCFM hasta Derecho en transporte público. Fuente: Google Maps.	12
4.1. Diagrama de la interacción entre el usuario y un algoritmo de enrutamiento.	28
5.1. Ejemplo de uso: mapeo de los recorridos de la Zona H de los buses Red.	31
5.2. Ejemplo de uso: mapeo de las líneas del Metro de Santiago.	32
5.3. Plano de la Red de Metro, incluyendo estaciones operativas desde Septiembre de 2020. Fuente: moovitapp.com	32
5.4. Ejemplo de uso: mapa de densidad de la totalidad de paradas del transporte público en Santiago.	33
5.5. Ejemplo de uso: mapa de densidad de las paradas del transporte público en Santiago, donde se detienen 10 o más recorridos.	34
5.6. Ejemplo de uso: mapa de densidad de las paradas del transporte público en Santiago, donde se detiene solo un recorrido.	35

5.7. Ejemplo de uso: generación de ruta entre Conchalí y Santiago un jueves de madrugada.	36
5.8. Ejemplo de uso: generación de ruta entre Plaza de Quilicura y la Casa Central de la Universidad de Chile, previo a la inauguración de la extensión de Línea 3 de Metro.	37

Estructura del Documento

Este informe presenta las distintas etapas del desarrollo de un módulo llamado ‘ayatori’, que contiene la base para programar algoritmos de planificación de rutas, correspondiendo a la Memoria para optar al título de Ingeniero Civil en Computación. El documento está dividido en 7 capítulos distintos, listados a continuación con su respectiva temática:

- **Capítulo 1: Introducción.** Entrega la base contextual y los objetivos del proyecto.
- **Capítulo 2: Estado del Arte.** Presenta los antecedentes del proyecto que componen su Marco Teórico, junto a los conceptos y herramientas a utilizar, para comprender la base teórica del mismo.
- **Capítulo 3: Diseño.** Explica el diseño de la solución propuesta, incluyendo el stack tecnológico a usar, el funcionamiento lógico de la solución, y el criterio de evaluación a considerar, explicitando el caso de estudio a realizarse.
- **Capítulo 4: Implementación.** Expone el desarrollo de las distintas fases de la implementación del módulo.
- **Capítulo 5: Resultados.** Muestra los resultados finales obtenidos al terminar la implementación, a través de ejemplos de uso del módulo, la ejecución del caso de prueba establecido, y la posterior evaluación.
- **Capítulo 6: Discusión.** Desarrolla las discusiones posteriores a la evaluación de resultados, considerando las implicancias y limitaciones de la solución, además de posibles líneas de trabajo futuro.
- **Capítulo 7: Conclusión.** Sintetiza el trabajo realizado y concluye el desarrollo del Trabajo de Título.

Posteriormente, se presenta la bibliografía utilizada y referenciada a lo largo del informe. Además, un anexo que incluye el código de la solución implementada.

Capítulo 1

Introducción

A día de hoy, es normal que las grandes ciudades experimenten cambios constantemente que las hagan crecer. Este fenómeno, común a nivel mundial, está presente también en Chile. Estudiando la situación local, existen múltiples causas asociadas, algunas de estas siendo más globales (como el cambio climático), y otras más específicas, como el importante aumento de la migración tanto interna como externa al país durante los últimos años, y la construcción de nueva infraestructura urbana. Si bien han existido ciertas condiciones que afecten negativamente el florecimiento de las ciudades, como la pandemia del COVID-19, la tendencia general de crecimiento se mantiene. Así es como, en un mundo donde las grandes urbes tienden a crecer exponencialmente, la planificación y buena gestión de las ciudades se ha visto afectada por el auge de estos fenómenos.

Es necesario, entonces, hallar maneras novedosas para comprender y caracterizar, correc-tamente, la vida de los habitantes de las grandes ciudades, tal como la capital de nuestro país, Santiago. En este mismo contexto, una arista muy importante a considerar es la movilidad vial, o el cómo las personas son capaces de movilizarse a través de las calles y avenidas de una ciudad, la cual es un factor determinante de la calidad de vida de sus habitantes. Las grandes ciudades suelen ser el hogar de una gran cantidad de personas, las cuales necesitan transportarse cada día para realizar sus jornadas de trabajo, de estudio, entre otras.

Existen múltiples registros de información que pueden ser utilizados para estudiar la movilidad urbana de ciudades como Santiago. Sin embargo, esto no quita que existan ciertas condiciones que puedan afectar su correcto análisis. Por ejemplo, la *Encuesta Origen Destino* es una herramienta utilizada por los gobiernos para estudiar patrones de viajes de los habitantes de las ciudades, y el gobierno de Chile ha realizado esta encuesta en múltiples ciudades del país durante los últimos años. Esto, evidentemente, incluye también a Santiago, pero la última vez que se realizó fue en el año 2012, hace más de una década atrás [19]. Debido a esto, la información inferida gracias a la encuesta probablemente no represente, de forma correcta, la realidad actual del transporte en la capital, lo cual es una problemática común a esta clase de instrumentos de estudio. Se necesita, luego, una herramienta que permita hacer este trabajo más continuamente, y que represente al comúin de los habitantes de la ciudad.

Con respecto a los medios de movilización, la gente puede tener a su disposición múltiples tipos, tanto públicos como privados. Por ejemplo, se pueden mover a pie, en bicicleta, en auto, o utilizando el transporte público. Siguiendo la idea anterior, para poder caracterizar correctamente la movilidad urbana, es útil hacerlo desde la perspectiva de un medio de transporte que esté disponible para toda la población, así que estudiar el uso del transporte público en Santiago resulta ser una buena opción para este fin. Dentro de la ciudad, el sistema de transporte es llamado Red Metropolitana de Movilidad, e incluye al Metro de Santiago, los buses de Red (antiguamente Transantiago), y el servicio de tren urbano Nos-Estación Central, los cuales son usados por las personas en múltiples combinaciones, generando una cantidad enorme de rutas diferentes. Para almacenar y hacer pública la información del sistema de horarios de esta clase de medios de transporte, existe el formato GTFS (General Transit Feed Specification) [6] que utilizan las agencias de transporte en el mundo para estandarizar la información de, entre otras cosas, los diferentes servicios existentes, sus rutas y paradas respectivas.

Actualmente, existen algoritmos que se han diseñado con el fin de responder a consultas de movilidad. Por ejemplo, Connection Scan Algorithm [12] (CSA) es un algoritmo creado para responder de manera eficiente a consultas en los sistemas de información de horarios del transporte público, recibiendo como entrada una posición de origen y una posición de destino, y generando una secuencia de vehículos que el viajero debe tomar para recorrer una ruta entre ambos puntos. CSA, al igual que otros algoritmos que cumplen un objetivo similar, se alimentan de la información del transporte público disponible, enlazando esta información con los datos cartográficos de la ciudad a estudiar. Por este motivo, ya sea que se desee implementar un algoritmo existente o desarrollar uno nuevo, es crucial contar con una buena base de información y herramientas de programación que permitan la creación exitosa de nuevos mecanismos de estudio.

Este trabajo de título tiene por objetivo principal realizar un módulo en Python llamado Ayatori, que además de contar con la información del transporte en Santiago, contenga todas las definiciones y declaraciones necesarias para poder desarrollar algoritmos de generación de rutas. La idea es que el producto generado permita desarrollar estudios de movilidad vial de forma más actualizada y directa, a través de las herramientas que se puedan desarrollar usándolo como base. La visión a futuro es que puedan realizarse casos de estudio que visualicen el impacto de la ampliación del transporte público disponible sobre los patrones de movilidad de las personas, y contribuir al desarrollo de nuevas tecnologías para programar soluciones de movilidad vial.

1.1. Objetivos

Objetivo General

El objetivo general de este trabajo de título es crear un módulo de trabajo en Python, con el fin de generar una base de programación para desarrollar algoritmos de movilidad, focalizando su uso en Santiago de Chile. Para ello, se utilizan los datos cartográficos de la ciudad provenientes de OpenStreetMap, un proyecto colaborativo de creación de mapas comunitarios [5], y la información del transporte público provista por la Red Metropolitana de Movilidad.

Objetivos Específicos

1. Obtener la información cartográfica de Santiago, además de la información del transporte público (en formato GTFS), y almacenarla en estructuras de datos pertinentes.
2. Enlazar la información de ambas fuentes de datos para ubicar las rutas de transporte en el mapa de Santiago.
3. Programar las definiciones para poder operar sobre estos datos, identificando lo necesario dentro de las estructuras de datos definidas y extrayendo la información que se necesite entregar al usuario.
4. Realizar un caso de prueba, utilizando el módulo para crear una implementación básica de un algoritmo de generación de rutas, además de generar otras visualizaciones útiles para estudiar la movilidad urbana, y así ejemplificar la utilidad del trabajo realizado.

Capítulo 2

Estado del Arte

2.1. OpenStreetMap

OpenStreetMap (OSM) es un proyecto colaborativo cuyo propósito es crear mapas editables y de uso libre [15]. Los mapas, generados mediante la recopilación de información geográfica a través de dispositivos GPS móviles, incluyen detalles sobre las vías públicas (como pasajes, calles y carreteras), paradas de autobuses y diversos puntos de interés. Al ser un proyecto *Open-Source*, el desarrollo de los mapas locales es gestionado por organizaciones voluntarias de contribuyentes; en nuestro país, la Fundación OpenStreetMap Chile [5] cumple ese papel.

OpenStreetMap se encarga de almacenar información sobre los distintos elementos del mapa de las ciudades. Esto incluye, evidentemente, a la red vial que está presente (incluyendo calles, pasajes, carreteras, etc.), pero no se limita solo a ello. El proyecto también almacena datos de los edificios presentes en la ciudad, y además, de múltiples puntos de interés, tales como escuelas, parques y capillas, representados como nodos en la organización interna de la información. Estos nodos también pueden estar englobados en relaciones, permitiendo delimitar las ciudades y otras organizaciones espaciales (como regiones y comunas), volviendo el acceso a la información más versatil.

Existen múltiples mapas online que utilizan la información de OpenStreetMap. El más conocido es el que tiene la propia web de OpenStreetMap [15], desde donde se puede ingresar una ubicación en el buscador de la página para hallarla en el mapa. Se presenta la figura 2.1 a modo de ejemplo, donde se observa el mapa de la Región Metropolitana de Santiago en toda su extensión y delimitada por una línea de color naranja. En este mapa se pueden notar, entre otras cosas, las autopistas principales representadas por líneas de color rojo, tal como la Circunvalación Américo Vespucio.

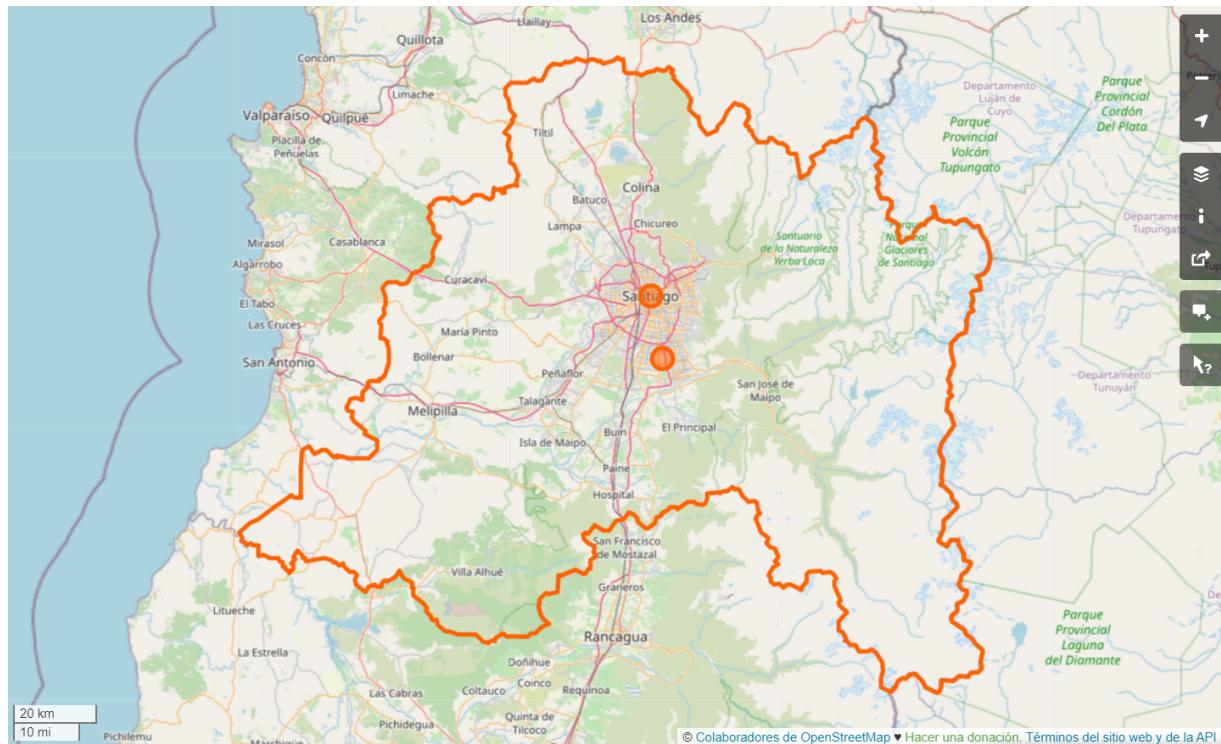


Figura 2.1: Mapa de la Región Metropolitana en OpenStreetMap. Fuente: openstreetmap.org

También existen otros mapas online, que se especializan en un área determinada de los datos disponibles. Por ejemplo, la web de OpenCycleMap destaca la información relevante para ciclistas (como ciclovías y estacionamientos de bicicletas) [2], WheelMap se encarga de mostrar los lugares que son accesibles para usuarios de sillas de ruedas [13], y ÖPNVKarte es un mapa que grafica las redes de transporte público disponibles [23].

2.1.1. OSM integrado en algoritmos

Para poder programar correctamente algoritmos de planificación de rutas, se requiere de la información cartográfica (o sea, mapas) de la ciudad en cuestión para poder ubicar los puntos que las rutas deben conectar. Para este fin, se pueden alimentar de los datos provenientes de OpenStreetMap (OSM), los cuales son utilizados para ubicar las coordenadas de los puntos de origen y destino en un mapa, y de esta forma obtener propiedades como la distancia entre los puntos, además de identificar detalles como las calles aledañas a las ubicaciones buscadas. Aquí también se obtienen las coordenadas de las paradas de los servicios de transporte público, tales como los paraderos de bus y las estaciones del Metro.

Anteriormente en la figura 2.1, se mostró una visualización de estos datos proveniente de la web de OpenStreetMap. Sin embargo, aparte de utilizar la información mediante visualizaciones web, los datos de OSM también se pueden descargar en distintos formatos para su uso offline. Esto permite una mayor versatilidad a la hora de crear herramientas que hagan uso de esta información, y no necesitar de una conexión constante a internet para analizar

los mapas. Esto, claro, implica no trabajar necesariamente con la última versión de los datos, y deja a responsabilidad del usuario el descargar manualmente la información.

En esa arista, existen múltiples aplicaciones que trabajan con mapas offline. Algunas de estas son aplicaciones de dispositivos móviles, como Cruiser [11] en Android, y OsmAnd [28] tanto en Android como en iOS, softwares de navegación GPS que trabajan con datos offline. Además, existen algunos frameworks de programación que permiten generar visualizaciones offline de mapas en las aplicaciones, como es el caso de CartoType, un framework de enrutamiento y renderizado de mapas para programas en C++ [3].

En este caso, para integrar los datos de OSM dentro de un módulo de Python, se puede importar alguna de las librerías existentes que permiten operar con estos datos. Por ejemplo, la librería **pyrosm** [29] funciona como un parser de la información de OSM en Python. **pyrosm** permite descargar la información más actualizada de la ciudad, almacenándola en un grafo dirigido donde cada nodo representan una intersección entre vías o algún lugar de interés, y las aristas entre los nodos representan las vías en sí; el que el grafo sea dirigido responde al sentido de las vías (es diferente una calle que es *doble vía* a una que va en un solo sentido). Trabajar con grafos permite otorgar propiedades a los componentes, como las coordenadas a los nodos (su latitud y longitud) o el largo a las aristas (representando la distancia entre ambos nodos que conecta). Además, de esta forma, la información de OSM se almacena en un formato conveniente para su fácil operación.

pyrosm hace posible acceder a la información de OpenStreetMap y crear visualizaciones con ella. Esto permite crear mapas que grafiquen distintas áreas de los datos, como por ejemplo, los puntos de interés de la ciudad. En la figura 2.2, se muestra un mapa de la ciudad de Helsinki, Finlandia, donde se destacan los diferentes puntos de interés, tales como bancos, tiendas de conveniencia, restaurantes, entre otros.

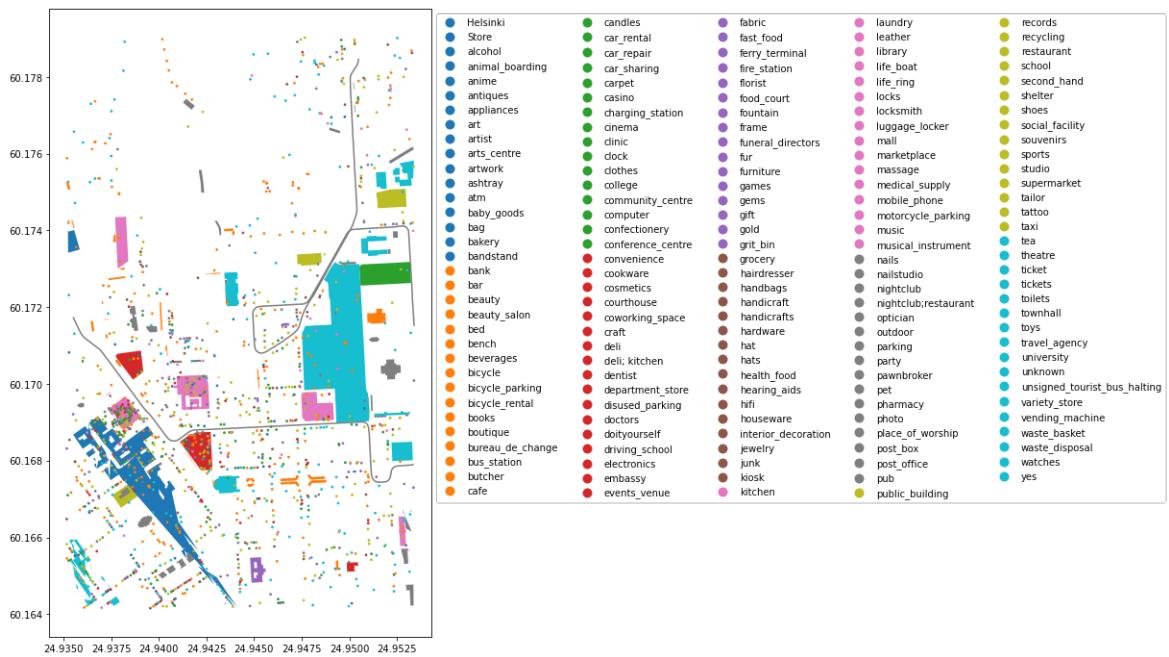


Figura 2.2: Mapa de la ciudad de Helsinki, Finlandia, que destaca sus puntos de interés.
Fuente: pyrosm.readthedocs.io.

Un detalle a destacar es que el gráfico anterior muestra tanta información que la paleta de colores disponible no alcanza a cubrir ni la décima parte de los tipos de edificios mostrados, por lo que cada color se repite una gran cantidad de veces. En el sentido de la visualización de la información, es un punto en contra. Sin embargo, es una muestra clara de la gran cantidad de datos que se incluyen, para cada ciudad, en el proyecto de OpenStreetMap, lo que deriva en una multitud de diferentes aplicaciones prácticas que se pueden explotar.

2.2. GTFS

Las Especificaciones Generales del Suministro de datos para el Transporte público, o en inglés, General Transit Feed Specification (GTFS), son un tipo de especificaciones ampliamente utilizado para definir y trabajar sobre datos de transporte público en las grandes ciudades. Este instrumento consiste en una serie de archivos de texto, recopilados en un archivo ZIP, de manera tal que cada archivo modela un aspecto específico de la información del transporte público, como paradas, rutas, viajes y horarios.

En la figura 2.3, un diagrama relacional muestra cómo estos distintos archivos se relacionan entre ellos, mostrando las diferentes entidades incluidas en el formato GTFS:

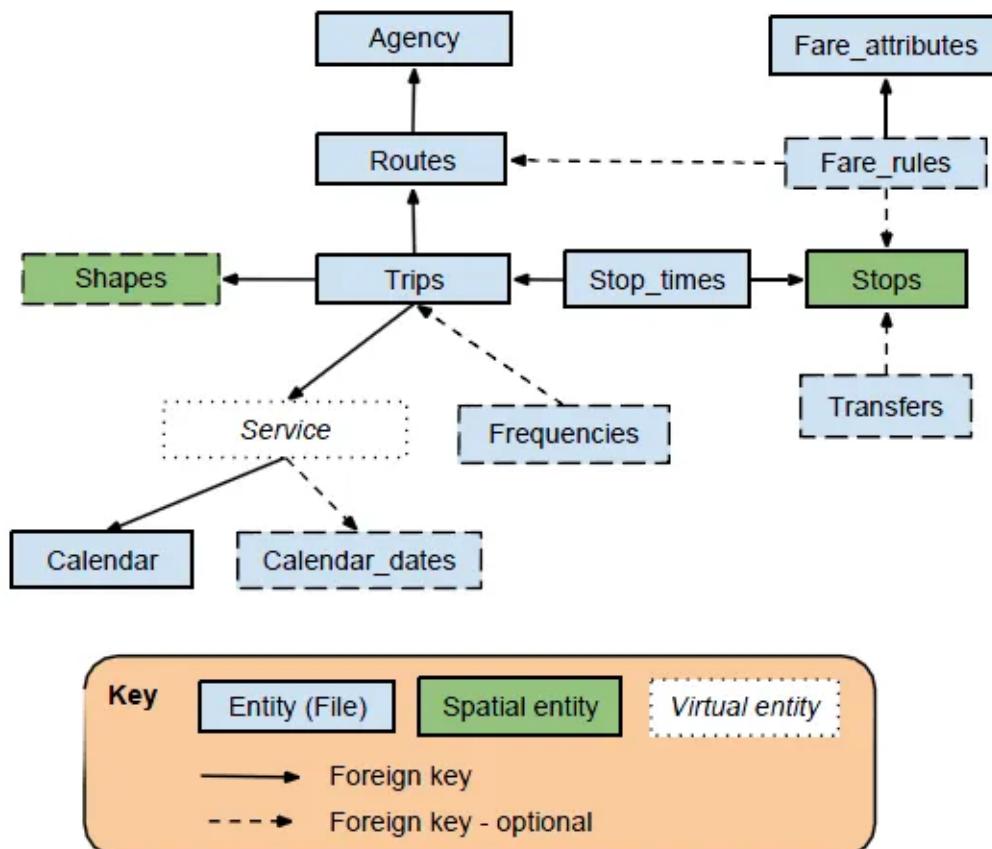


Figura 2.3: Diagrama relacional de los archivos que componen el formato GTFS. Fuente: medium.com .

Se destaca que algunas de las entidades que aparecen en el diagrama, como *Fare_rules* y *Fare_attributes*, no son tablas obligatorias para el formato. En específico, los datos obligatorios que todo grupo de archivos en GTFS debe incluir son *Agency* (que representa a las agencias que proveen vehículos al transporte público), *Stops* (que representa a las paradas de los diferentes servicios del transporte), *Routes* (que representa a las rutas o servicios ofrecidos en la red), *Trips* (que representa a los viajes de cada ruta, como una secuencia de paradas en un tiempo determinado), y *Stop-times* (que representa a los tiempos en los que cada servicio llega y se va de sus paradas).

A nivel global, las organizaciones encargadas de la gestión administrativa del transporte público suelen utilizar este formato para compartir la información. En Santiago, el Directorio de Transporte Público Metropolitano (DTPM) es la entidad encargada de esta tarea. Este organismo, dependiente del Ministerio de Transportes y Telecomunicaciones, y cuya misión es mejorar la calidad del sistema de transporte público en la ciudad, tiene disponible públicamente esta información, y la actualiza periódicamente [9]. Al momento de la entrega de este informe, la última versión fue configurada para implementarse desde el 23 de septiembre de 2023.

Como se mencionó anteriormente, la información en GTFS está contenida en diferentes archivos de texto, con sus valores separados por comas (similar a un CSV). Cada archivo concentra un área específica de los datos. En el caso de la información provista por el DTPM, los archivos incluidos en el *feed* GTFS son los siguientes:

- **Agency:** entrega la información de las diferentes agencias de transporte que alimentan el GTFS. En este caso, se encuentra la Red Metropolitana de Movilidad (que engloba a todos los buses Red, antiguamente Transantiago), el Metro de Santiago, y EFE Trenes de Chile.
- **Calendar Dates:** especifica fechas especiales que alteran el funcionamiento habitual de los recorridos que varían por día. Para la última versión, este archivo contiene todas las fechas de feriados que caen entre lunes y sábado.
- **Calendar:** especifica los diferentes recorridos que varían por día, con su tiempo de validez. Acá se especifican los recorridos de Red para tres formatos diferentes: el cronograma para los días laborales (lunes a viernes), el cronograma para los sábados, y el cronograma para los domingos.
- **Feed Info:** información de la entidad que publica el GTFS.
- **Frequencies:** listado que, para todos los viajes de los recorridos disponibles, incluye sus tiempos de inicio y de término, y el *headway* o tiempo de espera estimado entre vehículos.
- **Routes:** contiene el identificador de cada ruta existente, su agencia, ubicación de origen y destino.
- **Shapes:** lista las diferentes ‘formas’ de los viajes de cada recorrido. Esto incluye el identificador de cada viaje (el recorrido y si acaso es de ida o retorno), y las latitudes y longitudes para cada secuencia posible.
- **Stop Times:** incluye las horas estimadas de llegada para que cada recorrido incluido en el GTFS llegue a cada parada incluida en su trayecto.

- **Stops:** contiene los identificadores, nombres, latitud y longitud de cada parada de transporte.
- **Trips:** contiene todos los viajes diferentes que realiza cada recorrido, señalando el nombre del recorrido, sus días de funcionamiento, si es de ida o retorno, y su dirección de destino.

Siguiendo este formato, los operadores de transporte pueden almacenar y publicar la información pertinente a sus sistemas, para que esta sea utilizada por las personas o entidades que lo estimen conveniente. Por ejemplo, los desarrolladores de aplicaciones que permitan a sus usuarios revisar el estado actual de los servicios de transporte público, con el fin de planificar sus viajes. Un ejemplo de flujo de información en el que estos datos pueden ser utilizados se detalla en el diagrama mostrado en la figura 2.4.

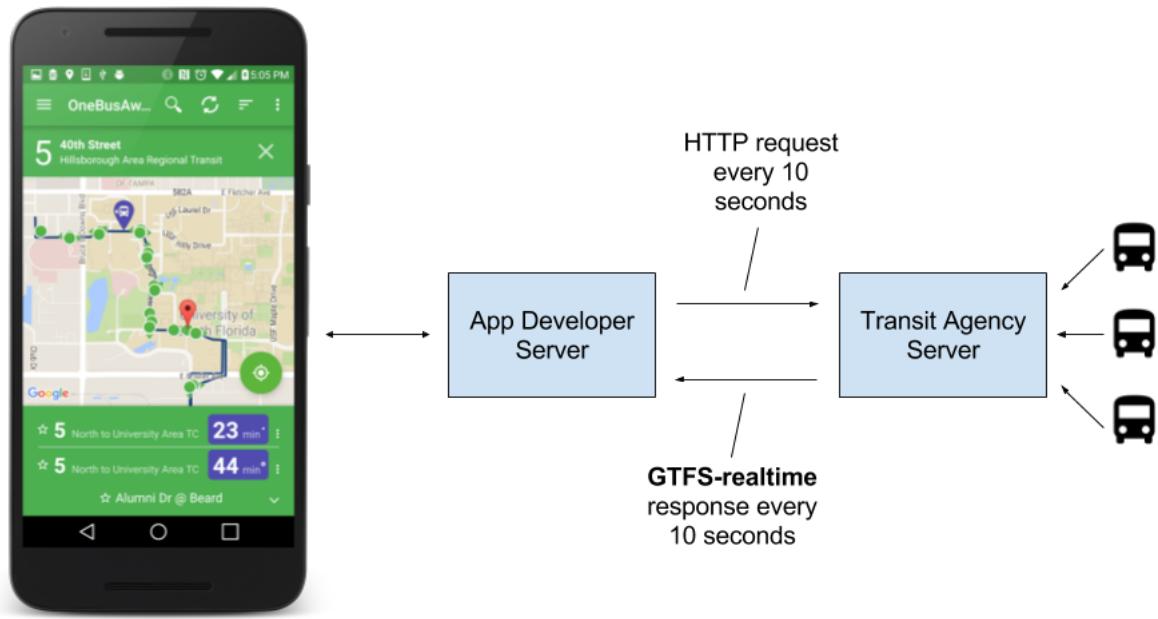


Figura 2.4: Diagrama de uso de datos en tiempo real en formato GTFS para una aplicación.
Fuente: watrifeed.ml .

En este ejemplo, se muestra cómo una aplicación móvil se conecta al servidor que almacena sus datos, el cual hace consultas periódicas al servidor de la agencia de tránsito. Este, al contener la información de los recorridos (en este caso, de buses), responde con información en tiempo real en formato GTFS, que finalmente el servidor de la aplicación interpreta para mostrarle al usuario la ruta en un mapa. Si bien este ejemplo muestra una aplicación con información en vivo, también pueden realizarse aplicaciones que hagan uso de datos *estáticos*, basándose en los cronogramas de viaje previamente definidos.

2.2.1. GTFS integrado en algoritmos

Los algoritmos de planificación de rutas necesitan tener a su disposición la información del transporte público, para ser capaces de calcular las rutas solicitadas. Esto implica que, para los distintos recorridos disponibles, se debe obtener los datos de sus rutas, paradas, horarios, y cualquier otra información que se estime necesaria para poder obtener la mejor ruta a seguir. Para este fin, es útil alimentar al algoritmo con la información del transporte público en formato GTFS, dado que así los datos están organizados de tal manera que son fácilmente accesibles, facilitando la programación y el cálculo de las rutas.

Similar al caso de OSM, se puede importar alguna librería existente que permita operar con los datos. Por ejemplo, la librería **pygtfs** [7] permite modelar archivos GTFS en Python. Esta librería almacena la información del transporte público en una base de datos relacional, tal que pueda ser usada en proyectos programados en este lenguaje de forma directa.

En relación a su organización interna, el objeto *Scheduler* es lo más importante de esta librería, pues representa a la base de datos completa. De esta manera, sus propiedades vienen dadas por las tablas que conforman el formato GTFS, mostradas como atributos. Así, al instanciar un objeto *Scheduler*, podemos acceder a la información pertinente de los cronogramas del transporte público.

2.3. Datos: estructura y manejo de la información

Implementar algoritmos de planificación de rutas requiere trabajar con un gran volumen de datos. Sin ir más lejos, considerando el cómo se definen los nodos y aristas de OSM en **pyrosm** (como se mencionó en la sección 2.1.1), se deduce que, para una ciudad como Santiago, existe un volumen importante de información que debe almacenarse para poder operar con ella. Es por esta razón que es crucial saber elegir una buena herramienta para la creación de las estructuras de datos correspondientes. En esta misma línea, el tipo de estructura de datos a utilizar viene dado, precisamente, por la forma en la que se almacena la información de OSM: grafos. Dicho esto, y dado que existen múltiples librerías que manejan este tipo de estructura en Python, se debe elegir una que se adecúe mejor a las necesidades de este proyecto.

Una alternativa muy utilizada en conjunto a **pyrosm** es **networkx**, un paquete de Python para la creación, manipulación, y estudio de la estructura, dinámica, y funciones de redes complejas [10]. Esta librería está disponible para sistemas operativos Windows mediante **pip**, el sistema de gestión de paquetes de Python. La razón por la cual es ampliamente utilizada es por su simplicidad en el manejo y operación de la información. Sin embargo, su principal problema recae en su rendimiento, pues al estar programada completamente en Python, su desempeño es lento en comparación a otras opciones. Si, además, se toma en cuenta el gran volumen de datos que se requiere almacenar, se infiere que el uso de **networkx** terminará generando un importante *bottleneck* o cuello de botella en el desempeño del algoritmo.

Por los motivos antes mencionados, se decide utilizar una librería diferente para este fin. La opción seleccionada es **graph-tool**, un módulo de Python creado para la manipulación

y análisis de grafos [8]. A diferencia de otras herramientas, **graph-tool** posee la ventaja de tener una base algorítmica implementada en C++, un lenguaje de programación basado en compilación, por lo que su desempeño es mucho más eficiente. Esto permite trabajar con grandes volúmenes de información de mejor manera, por lo que demuestra ser una excelente librería para utilizar en este proyecto. Cabe destacar, eso sí, que **graph-tool** solo se encuentra disponible para sistemas operativos GNU/Linux y MacOS. Como consecuencia, la programación del algoritmo se realiza en Ubuntu, una distribución de GNU/Linux, mediante WSL2 (Windows Subsystem for Linux 2) [22].

2.4. Connection Scan Algorithm: un algoritmo de planificación de rutas

Dentro de los algoritmos diseñados para el fin de planificar rutas de transporte, se encuentra Connection Scan Algorithm (CSA), un algoritmo desarrollado para responder, de manera eficiente, consultas en sistemas de información de horarios [12]. Este algoritmo es capaz de optimizar los tiempos de viaje entre dos puntos determinados de origen y destino, siendo alimentado por distintas fuentes de información de transporte. Como salida, entrega una secuencia de vehículos (como trenes o buses) que un viajero debería tomar para llegar al destino desde el origen establecido. La base teórica tras el algoritmo hace que este analice las opciones disponibles y optimice el número de transbordos, tal que sea Pareto-eficiente, es decir, llegando al punto en el cual no es posible disminuir el tiempo de viaje en un medio de transporte sin tener que aumentar el de otro. En la figura 2.5, se grafica el funcionamiento antes descrito:

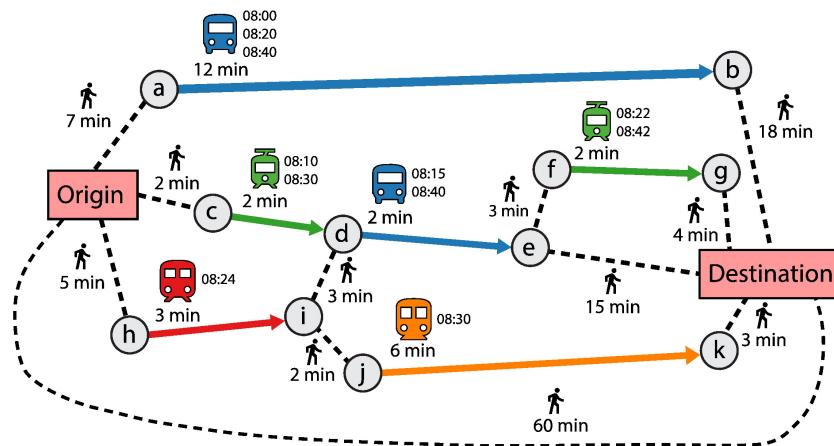


Figura 2.5: Diagrama explicativo del funcionamiento de Connection Scan Algorithm. Fuente: "Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys" (R. Kujala et al., 2017), vía sciencedirect.com .

Por ejemplo, si el punto de origen fuera la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile (Beauchef 850, Santiago), y el destino fuera la Facultad de Derecho de la Universidad de Chile (Pio Nono 1, Providencia), se debieran evaluar los medios de transportes que pueden ser utilizados para ir desde las coordenadas del punto de origen hasta las del punto de destino, y los transbordos necesarios. Posibles rutas podrían abarcar:

1. Una ruta con uso exclusivo del Metro de Santiago (subiendo en estación Parque O'Higgins de Línea 2, combinando en Los Héroes a Línea 1 y bajando en Baquedano).
2. Una ruta con uso exclusivo de buses de Red (tomar el recorrido 121 y luego el recorrido 502).
3. Una ruta que realice transbordos entre ambos medios de transporte (subir al metro en estación Parque O'Higgins y bajar en Puente Cal y Canto, para luego tomar el recorrido 502).

Los recorridos del ejemplo se muestran en la figura 2.6, generada utilizando el portal de Google Maps, el servidor web de visualización de mapas de Google [16], por su simplicidad de uso. Las rutas aparecen enumeradas según la lista previa.

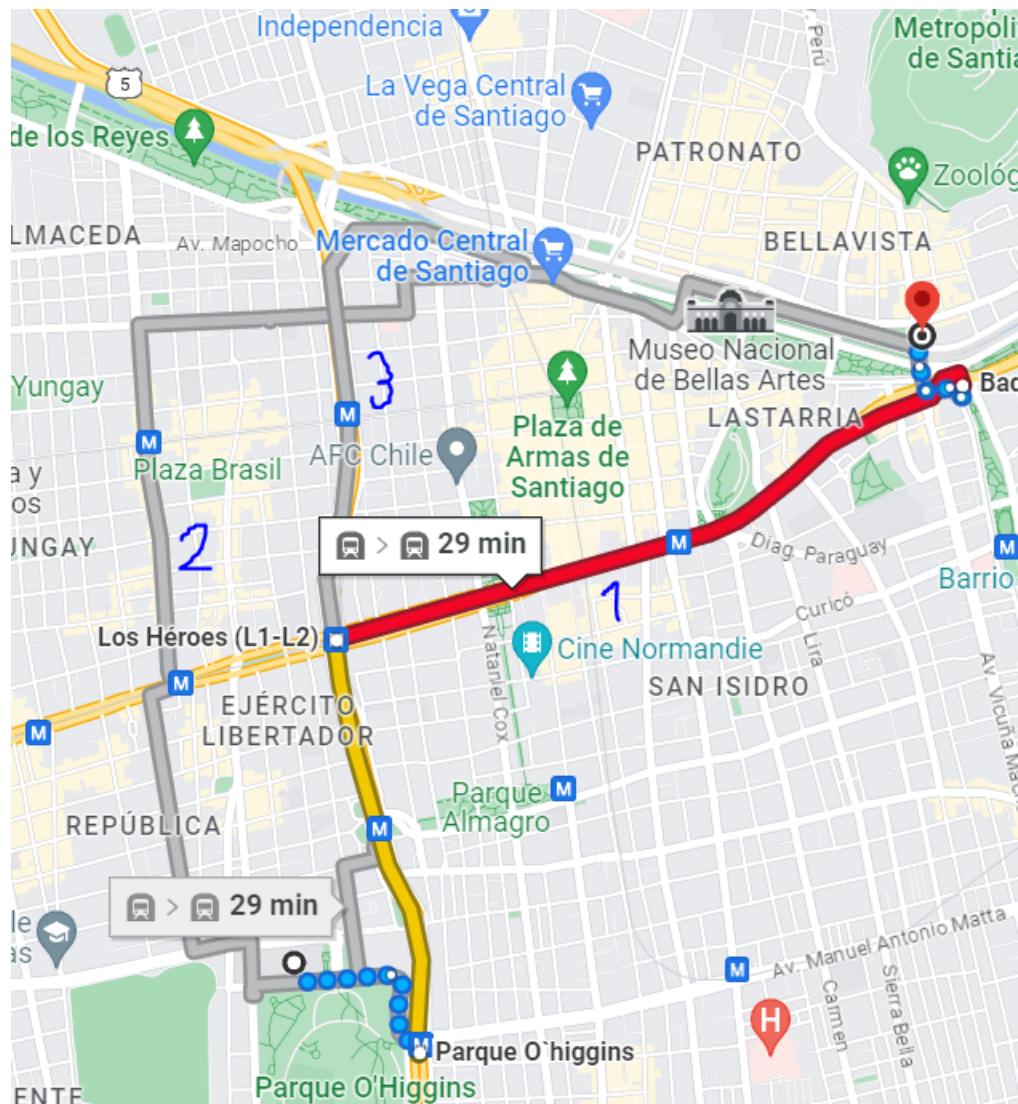


Figura 2.6: Posibles caminos para llegar desde FCFM hasta Derecho en transporte público.
Fuente: Google Maps.

Ejemplificando el caso anterior para resolverlo mediante CSA, el algoritmo recibe como entrada las coordenadas del punto de origen y el punto de destino. Luego, revisando la información del transporte público, calcula las rutas posibles (como las descritas anteriormente). CSA entonces buscaría el punto óptimo de Pareto con respecto a los transbordos, y entregaría la ruta recomendada para llegar al destino deseado. En este caso, al trabajar con información estática, se toman ciertos supuestos, como una velocidad de caminata fija entre transbordos y la continuidad operativa del servicio en todo momento.

Connection Scan Algorithm precisa, en primera instancia, ser capaz de obtener y almacenar la información del transporte público disponible, para así ser capaz de calcular la ruta óptima. Sin embargo, dado que el algoritmo trabaja con las coordenadas de los puntos de origen y destino, es bueno contar también con los datos cartográficos del sector o ciudad en cuestión donde se desea realizar el viaje, con el fin de obtener mejores visualizaciones de los resultados. Trabajando con ambos flujos de información, es posible crear una sólida implementación del algoritmo.

2.4.1. Utilidad como caso de prueba

Desde que Connection Scan Algorithm fue publicado, en marzo de 2017, ha sido implementado en varios formatos y lenguajes de programación. En el sitio web de Papers with Code, un portal que recopila códigos desarrollados sobre la idea central de diferentes papers, se muestran varias de estas implementaciones, enlazadas con su respectiva fuente de origen.

Destaca, entre estos, el repositorio de *ULTRA: UnLimited TRAnsfers for Multimodal Route Planning* [27], un framework desarrollado por el *Karlsruher Institut für Technologie* (KIT) en C++, para realizar planificaciones de viajes que incluyen diferentes medios de transporte. Este framework considera CSA, junto con otros algoritmos, para entregar posibles rutas entre dos puntos de una ciudad. Otra implementación disponible existe en el repositorio creado por Linus Norton, que creó una implementación del algoritmo en TypeScript [24].

Para demostrar la utilidad del módulo Ayatori para programar esta clase de algoritmos, se decide crear una implementación **simplificada** de CSA en Python como caso de prueba, estudiando rutas en Santiago. Además del hecho de que, por su arquitectura, el algoritmo requiera la información cartográfica de la ciudad y de su transporte público (ambas incluidas en Ayatori), la motivación principal para elegir este algoritmo es que, analizando el terreno actual, las implementaciones existentes están programadas en lenguajes diferentes a Python, por lo que existe una arista no explorada. La elección del lenguaje de programación motiva las decisiones posteriores de herramientas y librerías, que se mencionan en las secciones 2.1.1, 2.2.1, y 2.3, explicitadas y resumidas en la sección 3.1 del siguiente capítulo.

Capítulo 3

Diseño

3.1. Stack tecnológico

Basado en lo obtenido del capítulo anterior, se genera un formato para diseñar la solución propuesta. Para poder obtener toda la información necesaria para programar un algoritmo de planificación de rutas, se debe procesar correctamente tanto los datos cartográficos de Santiago, como la información del transporte público. Así, para desarrollar el proyecto, se define lo siguiente:

- El producto objetivo consiste en un módulo para el lenguaje de programación Python.
- La información cartográfica que se incluye en el módulo se obtiene desde OpenStreetMap, procesada mediante la librería **pyrosm** [29].
- La información del transporte público que se incluye en el modulo está almacenada en el formato GTFS, procesada mediante la librería **pygtfs** [7]. Para operar el módulo, los datos de transporte son obtenidos previamente, descargando la última versión desde el sitio web del Directorio de Transporte Público Metropolitano [6].
- Para almacenar y trabajar con la información obtenida, se utiliza la librería **graph-tool** [8] para trabajar con grafos.

Otras librerías que son utilizadas para cumplir de mejor forma los objetivos especificados en la sección 1.1 responden a la necesidad de procesar la información del módulo para facilitar su uso al usuario final. En primer lugar, la librería **Nominatim** [18] permite que el usuario pueda ingresar una dirección en palabras y encontrar su ubicación en el mapa, en vez de trabajar directamente con coordenadas numéricas, lo que facilita el uso de algoritmos y resta la necesidad de obtener las coordenadas de los puntos deseados por otro medio; **Nominatim** permite realizar la geocodificación de estas direcciones, buscando sus coordenadas en los datos de OpenStreetMap. En segundo lugar, la librería **folium** [14] permite visualizar datos cartográficos en un mapa de fácil uso. **folium** está basado en la librería **Leaflet.js** de JavaScript [1], una librería que permite crear mapas interactivos, por lo que aprovecha todas sus características para generar sus visualizaciones. Estas dos librerías son utilizadas en la implementación del caso de prueba para exemplificar el tipo de uso del módulo Ayatori.

3.2. Funcionamiento lógico

Como fue mencionado, el diseño de la solución consiste en un fichero modularizado de funciones que procesan la información de OpenStreetMap y del transporte público disponible en Santiago, en formato GTFS. El objetivo de modularizar la solución es permitir la importación de sus definiciones a ficheros externos, tal que implementen aplicaciones prácticas de estas al crear algoritmos de generación de rutas.

Con el fin de facilitar el uso del módulo, las funcionalidades se almacenan en dos clases diferentes. La primera se encarga del almacenamiento y procesamiento de todos los datos provenientes de OpenStreetMap, mediante **pyrosm**. La segunda clase tiene por objetivo almacenar y procesar la información del transporte público, proveniente de **pygtfs**. De esta manera, ciudades como Santiago estarán representadas como una red de capas, donde una capa estará conformada por la información de la infraestructura urbana (calles, edificios, puntos de interés, etc.), y la otra estará conformada por la red de transporte público existente en ella (servicios, paradas, tiempos de espera, etc.)

3.2.1. OSMGraph: la clase de OSM

Al instanciar la clase **OSMGraph**, se descargan los datos más recientes de OpenStreetMap para Santiago, y se almacenan en un grafo de **graph-tool**. En este grafo, los nodos representan puntos de interés de la ciudad (pudiendo ser edificios o intersecciones), y las aristas representan a las vías, ya sean calles, pasajes o carreteras. Las aristas del grafo son dirigidas, cuya dirección representa el sentido de la vía (diferenciando las vías de un solo sentido de las llamadas *doble vía*).

Cada elemento del grafo generado posee propiedades para almacenar información relevante. En el caso de los nodos, sus propiedades dentro del grafo son:

- **Node ID**: el identificador del nodo dentro de los datos de OpenStreetMap.
- **Graph ID**: el identificador interno del nodo dentro del mismo grafo.
- **Lon**: la longitud de la ubicación asociada al nodo.
- **Lat**: la latitud de la ubicación asociada al nodo.

Por otro lado, las propiedades que poseen las aristas del grafo son:

- **u**: corresponde al vértice desde donde inicia la arista.
- **v**: corresponde al vértice hacia donde se dirige la arista.
- **Length**: corresponde al *tramo* cubierto por la arista, el cual debe ser mayor o igual a 2 para considerarse válida.
- **Weight**: el peso de la arista, que representa el *metraje* cubierto por la misma, es decir, la distancia física entre sus vértices.

La clase **OSMGraph** posee funcionalidades para visualizar los elementos del grafo de OSM, así como también funciones para operar con estos. Dado que toda la información está contenida en un solo grafo de **graph-tool**, **OSMGraph** está diseñada para ser una clase con herencia directa desde la clase **Graph** de esta librería. Esto implica que todos los métodos disponibles en **graph-tool** se pueden utilizar directamente para analizar y visualizar la red.

3.2.2. **GTFSDATA: la clase de GTFS**

Instanciando la clase **GTFSDATA**, se procesan los datos previamente descargados del transporte público (ubicados en un archivo llamado *gtfs.zip* en el mismo directorio, a menos que se indique lo contrario). La información de cada tabla es leída y procesada para cada servicio de transporte disponible, para así, posteriormente, crear un grafo de **graph-tool** independiente para cada servicio y almacenar sus datos. En este grafo, los nodos representan las paradas del servicio, y las aristas enlazan cada parada con la siguiente del recorrido que siga en la misma orientación.

Para cada grafo, sus elementos poseen propiedades para almacenar información, al igual que en el caso de OSM mencionado en la sección 3.2.1. En el caso de los nodos, se tiene:

- **Node ID:** el identificador de la parada.

Por otro lado, las aristas poseen las siguientes propiedades:

- **u:** corresponde al vértice desde donde inicia la arista. En este caso, el identificador de la parada de origen.
- **v:** corresponde al vértice hacia donde se dirige la arista. En este caso, el identificador de la parada de destino.
- **Weight:** el peso de la arista.

Además de esto, se hace necesario crear un diccionario que almacene todos los datos que enlazan a una parada con una ruta en cuestión. Esto es debido a que existe información importante que cobra sentido únicamente al solapar los datos de una parada con los de una ruta. En específico, estos son:

- Orientación: refiere al sentido de la ruta cuando se detiene en una parada en específico. Cada ruta tiene un recorrido de ida y uno de vuelta, y por lo general, solo se detiene en un determinado paradero en uno de los sentidos.
- Número de Secuencia: al realizar una ruta en una orientación dada, el número de secuencia es el valor ordinal de una parada para esa ruta. En palabras simples, representa el orden en el que la ruta pasa por las paradas (la primera parada, la segunda, la tercera, etc.)
- Tiempos de llegada: representa la hora aproximada en la que una ruta llega a una parada.

La orientación y el número de secuencia deben utilizarse para filtrar las rutas que sirven para viajar entre dos puntos del mapa, mientras que los tiempos de llegada son cruciales para elegir la mejor ruta y entregar el resultado. Sin embargo, ninguno de estos datos son inherentes a una parada o a una ruta, pues, por ejemplo, no se puede decir que una ruta *posee* una orientación, sino que pasa por una parada al ir en cierta orientación. Por estos motivos, se opta por usar un diccionario anidado, aparte de los grafos por ruta, para almacenar esta clase de información. Este diccionario se denomina **route_stops**.

Al igual que para el caso anterior, la clase **GTFSData** posee funcionalidades para visualizar los elementos del grafo de GTFS, así como también funciones para operar con estos.

3.2.3. Funcionalidades entre clases

Además de las funcionalidades creadas como métodos dentro de las dos clases previamente mencionadas para operar con la información, es necesario crear funciones adicionales que crucen los datos provistos por OSM y los que están en formato GTFS. Esto permite unificar la información proveniente de ambas fuentes y generar aplicaciones útiles para generar rutas de transporte, tal como obtener los nodos del mapa de OSM a los que corresponden las paradas de una ruta en específico del transporte público, o hallar la lista de paradas que se encuentran cerca de un punto específico del mapa. El generar estas funcionalidades fuera de las clases provistas permite no caer en malas prácticas de diseño como tener que instanciar una clase dentro de otra. La especificación de estas funcionalidades, además de los métodos de cada clase, se ahondan con mayor profundidad en el capítulo 4 del informe (Implementación).

3.3. Criterio de Evaluación

Tal como fue discutido anteriormente, la motivación principal al desarrollar el módulo Ayatori es crear una base de programación para desarrollar algoritmos de generación de rutas, específicamente enfocadas en el uso del transporte público de la ciudad. Para efectos de este Trabajo de Título, se usa a Santiago como ejemplo para mostrar las capacidades del módulo, pero dada la naturaleza de los datos utilizados, si se quisiera estudiar la movilidad de otra ciudad, basta con modificar la procedencia de los datos (específicamente el lugar buscado en OSM y el archivo del transporte público en formato GTFS).

En cualquier caso, considerando que el usuario final del proyecto es cualquier programador que desee desarrollar algoritmos de generación de rutas para estudiar la movilidad vial, se debe definir un criterio de evaluación acorde para valorar la utilidad de la solución creada. En este caso, el criterio es:

- El usuario final deberá ser capaz de crear herramientas (programas o visualizaciones) que permitan realizar estudios de movilidad urbana en Santiago, utilizando como base el módulo Ayatori.

Posterior a la implementación, se realiza un caso de prueba para analizar la utilidad de Ayatori. La finalidad es probar la efectividad de la solución desarrollada, exemplificando la utilidad del módulo y evaluando el cumplimiento del criterio definido anteriormente. El caso de prueba definido consiste en programar una versión simplificada de Connection Scan Algorithm [12] (CSA), que cuente con una visualización gráfica que mapee una ruta en Santiago de Chile, para ir desde un punto a otro de la ciudad utilizando el transporte público disponible (Metro de Santiago o buses Red). Además, la implementación debe hacer uso de la información provista por el módulo para entregarle información adicional al usuario, tal como los tiempos de espera estimados para los siguientes recorridos de la ruta buscada. De forma adicional, también se incluye el realizar visualizaciones con la información provista por GTFS, con el fin de caracterizar el transporte público de Santiago.

Cabe destacar que CSA está definido para considerar transbordos entre distintos recorridos del transporte público. Esta funcionalidad no está implementada en este caso de prueba, por escapar del objetivo general del proyecto (definido en la sección 1.1). Por este motivo, se habla de una versión *simplificada* de CSA, que cumple con calcular la ruta más conveniente considerando distancias y tiempos de espera estimados, pudiendo así demostrar la utilidad práctica de la solución. El desarrollo de este Caso de Prueba está documentado en la sección 5.1 del informe.

Capítulo 4

Implementación

En el presente capítulo, se detalla la implementación realizada del módulo Ayatori y todo el trabajo que corresponde a su desarrollo. El código fuente de la implementación ha sido almacenado en un repositorio de GitHub creado para este fin [21].

4.1. Clases y métodos

4.1.1. Procesamiento de OpenStreetMap

La información almacenada en OpenStreetMap puede ser descargada en formato PBF (Protocolbuffer Binary Format), para luego ser filtrada y procesada según lo necesitado. Geofabrik, un portal comunitario para proyectos relacionados con OpenStreetMap [30], tiene disponible para descarga la información de los distintos países del mundo, incluído Chile [31]. Con esto, es posible obtener la información geoespacial de Santiago y trabajar con ella, para lo cual es necesario procesarla correctamente. En un principio, se pretendía realizar este proceso manualmente, pero se descubrió una mejor alternativa, que permite automatizarlo.

pyrosm [29], la librería utilizada para procesar la información, permite leer datos de OpenStreetMap en formato PBF e interpretarla en estructuras de GeoPandas [20], librería de Python de código abierto para trabajar con datos geoespaciales. Además de esto, **pyrosm** también permite directamente descargar la información de una ciudad y actualizarla en caso de existir una versión anterior en el directorio, permitiendo automatizar este proceso. De esta forma, una vez descargada la información de Santiago, se pueden crear gráficos según se necesite para su representación.

Para realizar este procedimiento, dentro de la clase **OSMGraph** se programa el método **download_osm_file**, que usando el método **get_data** de **pyrosm**, descarga la información de la ciudad especificada. Como salida, entrega el puntero al archivo que contiene los datos cartográficos de dicho lugar. La definición de este método se muestra en el código 4.1:

```

1 def download_osm_file(self, OSM_PATH):
2     fp = pyrosm.get_data(
3         "Santiago", # Nombre de la ciudad
4         update=True,
5         directory=OSM_PATH)
6     return fp

```

Código 4.1: Definición del método **get_osm_data()**.

Por otro lado, se define el método **create_osm_graph**, que utilizando el método anterior, crea un grafo con la información obtenida. Aquí se definen y evalúan las propiedades para cada elemento del grafo, tal y como fue mencionado en la sección 3.2.1. Finalmente, se retorna el grafo creado. De esta manera, la clase **OSMGraph** llama a este método para instancear el grafo como definición interna de la clase. Un fragmento de este método se aprecia en el código 4.2:

```

1 def create_osm_graph(self, OSM_PATH):
2     fp = self.download_osm_file(OSM_PATH) # Descarga datos de OSM
3     osm = pyrosm.OSM(fp)
4     nodes, edges = osm.get_network(nodes=True) # Almacena nodos y aristas
5     en variables
6     graph = Graph() # Crea el grafo vacío
7
8     # Propiedades
9     lon_prop = graph.new_vertex_property("float")
10    lat_prop = graph.new_vertex_property("float")
11    node_id_prop = graph.new_vertex_property("long")
12    graph_id_prop = graph.new_vertex_property("long")
13    u_prop = graph.new_edge_property("long")
14    v_prop = graph.new_edge_property("long")
15    length_prop = graph.new_edge_property("double")
16    weight_prop = graph.new_edge_property("double")
17    (...)

18    return graph

```

Código 4.2: Fragmento del método **create_osm_graph()** que crea el grafo y las propiedades de sus elementos.

Luego de definir lo necesario para que la clase obtenga el grafo con la información proveniente desde OpenStreetMap, se definen métodos adicionales que permiten trabajar con estos datos. Por ejemplo, métodos que imprimen los nodos y aristas del grafo, o aquellos que buscan un nodo utilizando su identificador o coordenadas. El código se puede apreciar en mayor profundidad en el Anexo A de este informe.

Una funcionalidad a destacar para la lógica del módulo es el método **find_nearest_node**, el cual recibe coordenadas de latitud y longitud de un punto deseado, y entrega el índice del nodo del grafo que se encuentra más cercano a esas coordenadas. Esta función es necesaria dado que los nodos de OpenStreetMap están predefinidos y son fijos, por los que en muchos

casos no coinciden *exactamente* con las coordenadas del punto que se desea ubicar, así que se opera con el nodo más cercano. La definición de `find_nearest_node` se puede observar en el código 4.3:

```

1 def find_nearest_node(self, latitude, longitude):
2     query_point = np.array([longitude, latitude])
3
4     # Obtiene las propiedades
5     lon_prop = self.graph.vertex_properties['lon']
6     lat_prop = self.graph.vertex_properties['lat']
7
8     # Calcula las distancias hasta el punto
9     distances = np.linalg.norm(np.vstack((lon_prop.a, lat_prop.a)).T -
10        query_point, axis=1)
11
12     # Encuentra el indice del nodo mas cercano
13     nearest_node_index = np.argmin(distances)
14     nearest_node = self.graph.vertex(nearest_node_index)
15
16     return nearest_node

```

Código 4.3: Definición del método `find_nearest_node()`.

Finalmente, para permitirle al usuario final una operación más fácil sobre los datos, se crea un método adicional que permite entregar la dirección del punto deseado (en palabras, no en coordenadas) y, haciendo uso del método `find_nearest_node` definido anteriormente, entrega el nodo más cercano a la dirección deseada. Este método se denomina `address_locator`, y utiliza los servicios de geocodificación provistos por la librería **Nominatim** [18] para este fin, como fue mencionado en la sección 3.1. La definición de esta funcionalidad se puede apreciar en el código 4.4:

```

1 def address_locator(self, address):
2     geolocator = Nominatim(user_agent="ayatori")
3     while True: # Testeo del estado del servicio
4         try:
5             location = geolocator.geocode(address)
6             break
7         except GeocoderServiceError:
8             i = 0
9             if i < 15:
10                 print("Geocoding service error. Retrying in 5 seconds...")
11                 tm.sleep(5)
12                 i+=1
13             else:
14                 msg = "Error: Too many retries. Geocoding service may be
down. Please try again later."
15                 print(msg)
16                 return
17         if location is not None: # Obtiene las coordenadas para hallar al nodo
correspondiente
18             lat, lon = location.latitude, location.longitude
19             nearest = self.find_nearest_node(self.graph, lat, lon)
20             return nearest
21         msg = "Error: Address couldn't be found."
22         print(msg)

```

Código 4.4: Definición del método `address_locator()`.

El método recibe una dirección (address), suscribe un agente de geocodificación con un nombre (en este caso, *ayatori*), e intenta buscar las coordenadas de la dirección mediante la librería **Nominatim**. Evidentemente, el funcionamiento del algoritmo depende directamente del estado del servicio de **Nominatim**, por lo que si ese servicio se encuentra caído en algún momento, el algoritmo no funcionará. Por esta razón, se previene este caso, intentando acceder al servicio 3 veces; si no está disponible, se imprime un mensaje de error.

4.1.2. Procesamiento de datos en GTFS

El formato GTFS incluye múltiples archivos de texto que almacenan la información del transporte público, organizada según diversos criterios, tal y como se especificó en la sección 2.2. Toda la información viene en un archivo comprimido ZIP, descargado desde la web del DPTM [6]. Este archivo debe descargarse manualmente, y las versiones nuevas salen cada uno o dos meses. Sin embargo, suelen haber relativamente pocas diferencias entre una versión y la siguiente.

pygtfs [7], la librería utilizada para procesar los datos de GTFS, posee un módulo llamado **Schedule**, encargado de gestionar toda la información. Instanciando el método al crear una nueva variable, permite obtener la información de GTFS y enlazarla a ella. Con este objetivo, se crea el método **create_scheduler** para ser lo primero en operarse al trabajar con la clase **GTFSData**. Esto se muestra en el código 4.5:

```

1 def create_scheduler(self, GTFS_PATH):
2     # Crea el scheduler usando el archivo de GTFS
3     scheduler = pygtfs.Schedule(":memory:")
4     pygtfs.append_feed(scheduler, GTFS_PATH)
5     return scheduler

```

Código 4.5: Definición del método **create_scheduler()**.

En este fragmento, se solicita la memoria necesaria para generar la instancia del *scheduler*, y se procesa la información descargada previamente (el archivo *gtfs.zip*). Luego, se llama al método creando una variable interna para la clase (*scheduer*), y así, posteriormente, se puede acceder a la información de cada archivo de GTFS como si fuera un método de esta variable. Por ejemplo, para obtener la información de las paradas, basta con llamar a **scheduler.stops**, y para obtener los servicios o rutas, se llama a **scheduler.routes**.

Para almacenar esta información, tal como se mencionó en la sección 3.2.2, se crea un grafo de **graph-tool** específico para cada recorrido del transporte público disponible en Santiago. Esto quiere decir que cada recorrido de bus Red y cada línea de Metro de Santiago tiene su propio grafo, donde se almacenan sus paradas como nodos y se crean aristas que las unen. Dentro de la clase, se crea como variable interna un diccionario con estos grafos, cuya llave es el identificador del recorrido en cuestión (por ejemplo, '506' o 'L1'), para poder acceder a ellos de manera fácil.

Adicionalmente, se crea el diccionario **route_stops** con la información cruzada entre rutas y paradas, como los tiempos de llegada de los recorridos. Este diccionario anidado, o diccionario de diccionarios, tiene como primera llave el identificador de la ruta, y luego posee

un diccionario para cada parada por la que pasa dicha ruta. Toda la gestión del almacenamiento de estos datos, tanto en grafos como en diccionarios, se lleva a cabo en el método `get_gtfs_data`, del que se puede apreciar un fragmento en el código 4.6:

```

1 def get_gtfs_data(self):
2     sched = self.scheduler # Instancia del Scheduler
3     for route in sched.routes:
4         graph = Graph(directed=True) # Se crea un grafo por recorrido
5         node_id_prop = graph.new_vertex_property("string")
6         u_prop = graph.new_edge_property("object")
7         v_prop = graph.new_edge_property("object")
8         weight_prop = graph.new_edge_property("int")
9         (...)

10        # Se almacena la informacion en route_stops
11        self.route_stops[route.route_id][stop_id] = {
12            "route_id": route.route_id,
13            "stop_id": stop_id,
14            "coordinates": stop_coords[route.route_id][stop_id],
15            "orientation": "round" if orientation == "I" else "return",
16            "sequence": sequence,
17            "arrival_times": []
18        }
19        (...)

20        self.graphs[route.route_id] = graph # Se agrega el grafo al
21        diccionario
22        (...)

23        for route_id, graph in self.graphs.items():
24            weight_prop = graph.new_edge_property("int")
25            for e in graph.edges():
26                weight_prop[e] = 1
27            graph.edge_properties["weight"] = weight_prop
28            data_dir = "gtfs_routes" # Se declara el directorio para almacenar
29            los grafos
30            if not os.path.exists(data_dir):
31                os.makedirs(data_dir)
32            graph.save(f"{data_dir}/{route_id}.gt")
33        (...)

34    return self.graphs, self.route_stops, self.special_dates

```

Código 4.6: Fragmento del método `get_gtfs_data()`.

Accediendo a estas estructuras de datos, es posible crear múltiples funcionalidades que sean de utilidad para generar rutas de viaje. Por ejemplo, `get_near_stop_ids`, para obtener los identificadores de las paradas cercanas a un punto del mapa. Este método recibe como entrada una tupla de coordenadas y un margen numérico. Iterando sobre los elementos del diccionario `route_stops`, obtiene las coordenadas de cada parada, y revisa si está a una distancia cercana de las coordenadas entregadas, cercanía dada por el margen entregado. La existencia de este margen permite, en la práctica, modificar la distancia máxima hasta la cual una parada se considera *cercana* a los puntos del mapa. En el código 4.7, se puede observar la definición de este método:

```

1 def get_near_stop_ids(self, coords, margin):
2     stop_ids = []
3     orientations = []
4     for route_id, stops in self.route_stops.items():
5         for stop_info in stops.values():
6             stop_coords = stop_info["coordinates"]
7             distance = self.haversine(coords[1], coords[0], stop_coords
8 [1], stop_coords[0])
9             if distance <= margin:
10                 orientation = stop_info["orientation"]
11                 stop_id = stop_info["stop_id"]
12                 if stop_id not in stop_ids:
13                     stop_ids.append(stop_id)
14                     orientations.append((stop_id, orientation))
15
16 return stop_ids, orientations

```

Código 4.7: Definición de `get_near_stop_ids()`.

Como se ve en la definición del método, para calcular la distancia entre el punto y las paradas, se usa la función `haversine`, definida en el código 4.8:

```

1 def haversine(self, lon1, lat1, lon2, lat2):
2     R = 6372.8 # Radio de la Tierra en km
3     dLat = radians(lat2 - lat1)
4     dLon = radians(lon2 - lon1)
5     lat1 = radians(lat1)
6     lat2 = radians(lat2)
7     a = sin(dLat / 2)**2 + cos(lat1) * cos(lat2) * sin(dLon / 2)**2
8     c = 2 * asin(sqrt(a))
9     return R * c

```

Código 4.8: Definición de `haversine()` para calcular la distancia entre dos puntos.

La fórmula Haversine, o fórmula del semiverseno en español, es una ecuación que calcula la distancia entre dos puntos de una esfera, en base a su longitud y latitud. Esta fórmula es ampliamente utilizada en la navegación astronómica, pues permite calcular de forma fidedigna la distancia entre dos puntos del planeta.

Otra función importante que ha sido creada utilizando `route_stops` es `connection_finder`. Esta obtiene el diccionario y los identificadores de dos paradas como entrada, y luego de revisar todos los recorridos, entrega el listado de aquellos que se detienen en ambas paradas, es decir, los recorridos que pueden tomarse para ir de la primera parada a la segunda. La implementación de `connection_finder` se puede observar en el código 4.9:

```

1 def connection_finder(self, stop_id_1, stop_id_2):
2     connected_routes = []
3     for route_id, stops in self.route_stops.items():
4         stop_ids = [stop_info["stop_id"] for stop_info in stops.values()]
5
6         if stop_id_1 in stop_ids and stop_id_2 in stop_ids:
7             connected_routes.append(route_id)
8
9 return connected_routes

```

Código 4.9: Definición del método `connection_finder()`.

Tal como fue mencionado en la sección 3.2.2, uno de los datos relevantes que se deben conseguir accediendo a *route_stops* son los tiempos de llegada de los recorridos. Para poder considerar los tiempos de espera al realizar cálculos de la mejor ruta en el desarrollo de algoritmos, es crucial saber cuánto tiempo tardará el recorrido en llegar a una parada en específico, pues esto puede ayudar a definir casos donde hay más de una parada o recorrido útiles para llegar al destino deseado. En este caso, eso motiva la creación del método `get_arrival_times`, que se puede apreciar en el código 4.10:

```

1 def get_arrival_times(self, route_id, stop_id, source_date):
2     frequencies = pd.read_csv("frequencies.txt")
3     route_frequencies = frequencies[frequencies["trip_id"].str.startswith(
4         route_id)] # Obtiene las frecuencias de la ruta
5
6     day_suffix = self.get_trip_day_suffix(source_date)
7
8     stop_route_times = []
9     bus_orientation = ""
10    for _, row in route_frequencies.iterrows():
11        start_time = pd.Timestamp(row["start_time"])
12        if row["end_time"] == "24:00:00": # Normalizacion
13            end_time = pd.Timestamp("23:59:59")
14        else:
15            end_time = pd.Timestamp(row["end_time"])
16        headway_secs = row["headway_secs"]
17        round_trip_id = f"{route_id}-I-{day_suffix}"
18        return_trip_id = f"{route_id}-R-{day_suffix}"
19        round_stop_times = pd.read_csv("stop_times.txt").query(f"trip_id.
20 str.startswith('{round_trip_id}') and stop_id == '{stop_id}'")
21        return_stop_times = pd.read_csv("stop_times.txt").query(f"trip_id.
22 str.startswith('{return_trip_id}') and stop_id == '{stop_id}'")
23        if len(round_stop_times) == 0 and len(return_stop_times) == 0:
24            return
25        elif len(round_stop_times) > 0:
26            bus_orientation = "round"
27            stop_time = pd.Timestamp(round_stop_times.iloc[0]["arrival_time"])
28        elif len(return_stop_times) > 0:
29            bus_orientation = "return"
30            stop_time = pd.Timestamp(return_stop_times.iloc[0]["arrival_time"])
31            for freq_time in pd.date_range(start_time, end_time, freq=f"{headway_secs}s"):
32                freq_time_str = freq_time.strftime("%H:%M:%S")
33                freq_time = datetime.strptime(freq_time_str, "%H:%M:%S")
34                stop_route_time = datetime.combine(datetime.min, stop_time.
35 time()) + timedelta(seconds=(freq_time - datetime.min).seconds)
36                if stop_route_time not in stop_route_times:
37                    stop_route_times.append(stop_route_time)
38                    stop_time += pd.Timedelta(seconds=headway_secs)
39
40    return bus_orientation, stop_route_times

```

Código 4.10: Definición del método `get_arrival_times()`.

Un detalle a destacar de la definición del método anterior es que, además de tomar un identificador de parada y un identificador de ruta como entrada, considera la fecha del viaje para obtener los tiempos de llegada. La razón tras esto reside en que existen rutas que no operan todos los días de la semana, o algunas que sí lo hacen, pero con frecuencias de llegada distinta dependiendo del día, por lo que es necesario tomar en cuenta este detalle. De la misma manera, existen recorridos que solamente operan a ciertas horas del día, por lo que ambos datos se toman en consideración para crear el algoritmo de prueba, proceso especificado posteriormente en este capítulo.

De manera similar, se definen múltiples métodos dentro de la clase **GTFSData** para acceder a los múltiples archivos que constituyen la información del transporte público, permitiendo operar con ellos para programar algoritmos de generación de rutas. Con esto, se puede realizar un uso correcto de los datos provistos por ambas capas de información. Si bien, en esta sección se omite el código completo, gran parte de este puede revisarse en el anexo A.

4.1.3. Funcionalidades de GTFS sobre OSM

Además de la implementación de las funcionalidades internas de las clases anteriormente descritas, se definen aquellas que requieran utilizar información cruzada proveniente de ambas. Esto permite trabajar con las ciudades como un conjunto de dos capas enlazadas (mapa y red de transporte) y obtener datos tales como el listado de nodos de OpenStreetMap a los que corresponden las paradas de un recorrido en específico, útil para graficar rutas en el mapa. Esta funcionalidad se implementa en el método **find_route_nodes**, que se puede apreciar a continuación en el código 4.11:

```

1 def find_route_nodes(osm_graph, gtfs_data, route_id, desired_orientation):
2     (...)

3     stops = gtfs_data.route_stops.get(route_id, {}) # Obtiene las paradas
4         del recorrido

5     trip_stops = [stop_info for stop_info in stops.values() if stop_info[""
6         orientation"] == desired_orientation] # Filtra aquellas que coincidan
7         con la orientacion declarada

8     route_nodes = []
9     for stop_info in trip_stops:
10         # Halla los nodos correspondientes al recorrido
11         stop_coords = stop_info["coordinates"]
12         route_node = osm_graph.find_nearest_node(stop_coords[1],
13             stop_coords[0])
14         route_nodes.append(route_node)

15     return route_nodes

```

Código 4.11: Definición del método **find_route_nodes()**.

Otra funcionalidad interesante y útil es la definida por el método **find_nearest_stops**. Esta obtiene una dirección que busca en el grafo de **OSMGraph** para obtener sus coordenadas, y luego llama al método **get_near_stop_ids** de **GTFSData** (código 4.7) para obtener

las paradas cercanas y la orientación de los recorridos. A continuación, en el código 4.12 se muestra la definición de este método:

```

1 def find_nearest_stops(osm_graph, gtfs_data, address, margin):
2     graph = osm_graph.graph
3     v = osm_graph.address_locator(graph, str(address))
4     v_lon = graph.vertex_properties['lon'][v]
5     v_lat = graph.vertex_properties['lat'][v]
6     v_coords = (v_lon, v_lat)
7     nearest_stops, orientations = gtfs_data.get_near_stop_ids(v_coords,
8         margin)
9     return nearest_stops, orientations

```

Código 4.12: Definición del método `find_nearest_stops()`.

Con esto, la implementación del módulo Ayatori queda definida.

4.2. Creando un algoritmo

Para probar las capacidades del módulo, se implementa una versión *simplificada* de Connection Scan Algorithm utilizando las funciones disponibles. La implementación es tal que el programa solicita que el usuario ingrese una dirección de origen, una dirección de destino, una fecha y una hora de inicio del viaje, para luego hallar los puntos en el mapa de Santiago dentro del grafo de OpenStreetMap. Entonces, cruzando esta información con la provista por GTFS, busca paradas cercanas a estos puntos y revisa los servicios que las conectan, analizando a la vez sus tiempos de llegada aproximados. Finalmente, habiendo decidido la mejor ruta, entrega al usuario las instrucciones sobre qué servicio tomar, dónde subir y bajar del vehículo, y el tiempo de viaje aproximado que demora llegar hasta el destino señalado, todo acompañado de un mapa que grafica la ruta. Para poder realizar esto, y tal como se menciona en la sección 3.1, se utilizan funciones de las librerías **Nominatim** y **folium**. Adicionalmente, se utiliza la librería **datetime** para procesar la fecha y hora.

En el código 4.13 a continuación, se muestra la función que procesa los inputs del usuario para hacer funcionar el algoritmo de ejemplo:

```

1 def algorithm_commands():
2     now = datetime.now() # Hora y fecha actuales
3     today = date.today() # Fecha actual
4     today_format = today.strftime("%d/%m/%Y")
5     moment = now.strftime("%H:%M:%S") # Formateo
6     used_time = datetime.strptime(moment, "%H:%M:%S").time()
7
8     source_date = input(
9         "Enter the travel's date, in DD/MM/YYYY format (press Enter to use
10        today's date) : ") or today_format
11     print(source_date)
12     source_hour = input(
13         "Enter the travel's start time, in HH:MM:SS format (press Enter to
14        start now) : ") or used_time
15     if source_hour != used_time:
16         source_hour = datetime.strptime(source_hour, "%H:%M:%S").time()

```

```

15     print(source_hour)
16
17     source_example = "Beauchef 850, Santiago"
18     while True:
19         source_address = input(
20             "Enter the starting point's address, in 'Street #No, Province'
21             format (Ex: 'Beauchef 850, Santiago'):" or source_example
22         if source_address.strip() != '':
23             break
24
25     destination_example = "Campus Antumapu Universidad de Chile, Santiago"
26     while True:
27         target_address = input(
28             "Enter the ending point's address, in 'Street #No, Province'
29             format (Ex: 'Campus Antumapu Universidad de Chile, Santiago'):" or
30     destination_example
31         if target_address.strip() != '':
32             break
33
34     # La ultima entrada del algoritmo fija el rango de distancia entre los
35     # puntos de origen/destino y las paradas cercanas a revisar
36     best_route_map = connection_scan_lite(source_address, target_address,
37     source_hour, source_date, 0.2)
38
39     if not best_route_map:
40         print("Something went wrong. Please try again later.")
41         return
42
43
44     return best_route_map

```

Código 4.13: Función de comandos para operar el algoritmo de ejemplo.

La interacción entre el usuario y el algoritmo de enrutamiento se muestra en el diagrama de la figura 4.1, donde se destacan las dos capas incluidas por la solución (OSM y GTFS):

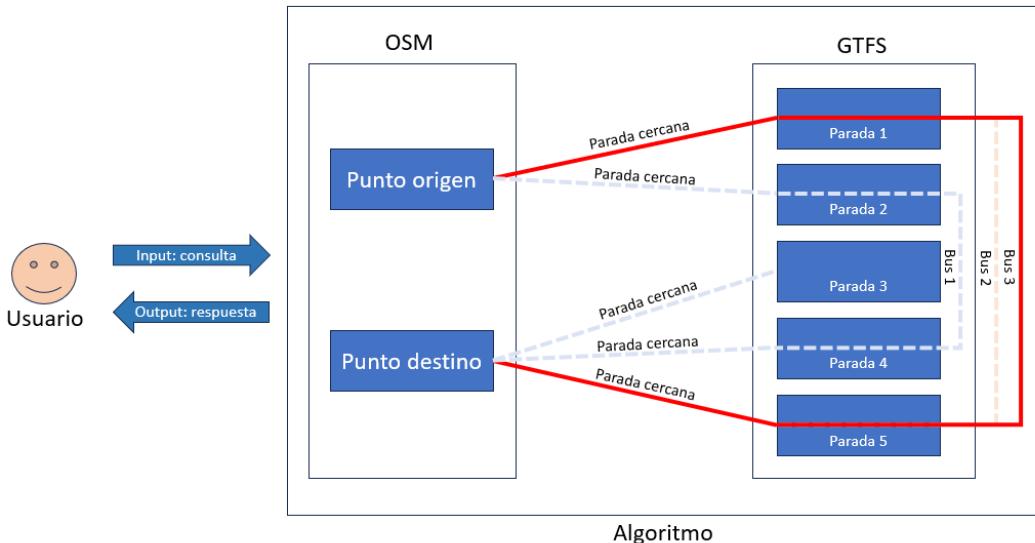


Figura 4.1: Diagrama de la interacción entre el usuario y un algoritmo de enrutamiento.

Como se muestra en la figura, el usuario interactúa con el algoritmo al hacer una consulta y entregarla como *input*. Luego, el algoritmo cruza la información contenida en las dos capas del modelo. Primero, encontrando en el mapa de OpenStreetMap los puntos de origen y destino que se desea conectar, para luego, con sus coordenadas, buscar la ubicación de todas las paradas cercanas a estos puntos en los datos de GTFS. Al hallar estas paradas, se buscan recorridos que pasen por alguna parada cercana al origen y luego por alguna parada cercana al destino. Como puede existir el caso de que exista más de algún recorrido que cumpla esta condición, se filtra con respecto al tiempo, prefiriendo el recorrido que menos tiempo demore en llegar. Finalmente, con esto se elige la mejor ruta, y se le retorna al usuario como *output* del programa.

Para entregarle al usuario la información completa de salida, la implementación está diseñada para retornar una salida de texto y otra salida visual. El texto corresponde a las instrucciones necesarias para realizar la ruta, lo que incluye el ID de la parada en la que hay que subirse al recorrido definido como mejor ruta (con su respectivo identificador), el ID de la parada donde se debe bajar del recorrido, y además, la hora aproximada de llegada de los próximos buses y el tiempo estimado de viaje completo. Por otro lado, la parte visual corresponde a un mapa que cruza la información de OSM y GTFS para graficar la mejor ruta, destacando los puntos de origen y destino, las paradas de subida y bajada del recorrido, y el recorrido en sí.

La función **connection_scan_lite** realiza el trabajo descrito, cruzando la información de OSM y GTFS. Así, se define la mejor ruta (destacada en rojo en la figura) por sobre todas las conexiones posibles (representadas por líneas punteadas en color gris). Por su extensión, no se incluye el código de la función en esta sección del informe, pero está disponible en el Anexo A.

Capítulo 5

Resultados

En el presente capítulo, se describen y analizan los resultados obtenidos luego de la implementación de la solución. Para mostrar los resultados, se presentan ejemplos de uso del módulo, y la realización del caso de estudio definido anteriormente, con la posterior evaluación de resultados.

5.1. Ejemplos de uso y caso de estudio

Para demostrar los diferentes usos posibles de la solución, se pueden crear diferentes visualizaciones de la información provista, tanto por OpenStreetMap como por la Red Metropolitana de Movilidad, en formato GTFS. A continuación, se muestran algunos ejemplos de los análisis que pueden realizarse.

5.1.1. Mapeo de recorridos

Usando la función `map_route_stops`, se genera un mapa que grafica la secuencia de paradas completa de un recorrido en específico. Esto permite, por ejemplo, comprender la longitud del recorrido y qué comunas abarca. En la figura 5.1, se muestra el uso de esta función para mapear los recorridos de buses Red correspondientes a la ‘Zona H’:

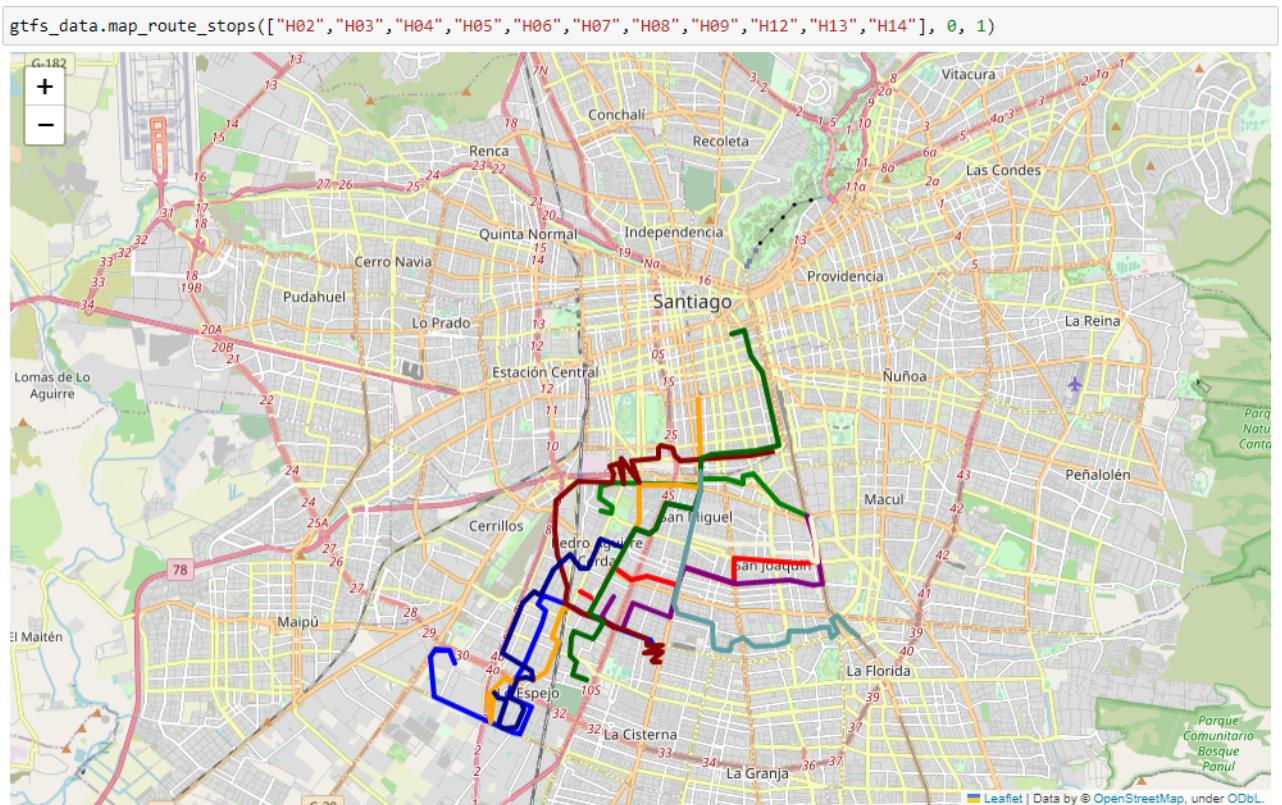


Figura 5.1: Ejemplo de uso: mapeo de los recorridos de la Zona H de los buses Red.

En la figura, cada recorrido de esta zona (que incluye a los recorridos ‘HXX’) se representa como una línea con un color diferente sobre el mapa de Santiago. Analizando en detalle, se puede apreciar que los buses de esta zona abarcan, principalmente, a las comunas de Lo Espejo, Pedro Aguirre Cerda, San Miguel, y San Joaquín, conectándolas mediante uno de sus recorridos con Santiago Centro.

También se puede utilizar esta función para graficar los recorridos de las diferentes líneas del Metro de Santiago. Así, se puede visualizar el Plano de la Red de Metro en un mapa a escala real, lo que se presenta a continuación en la figura 5.2:

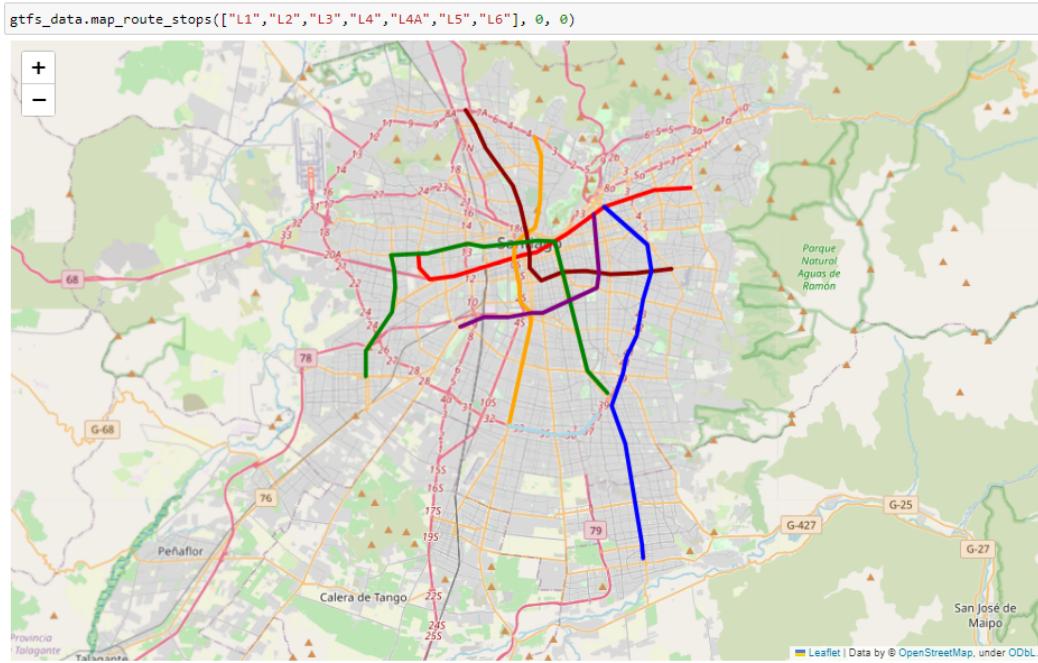
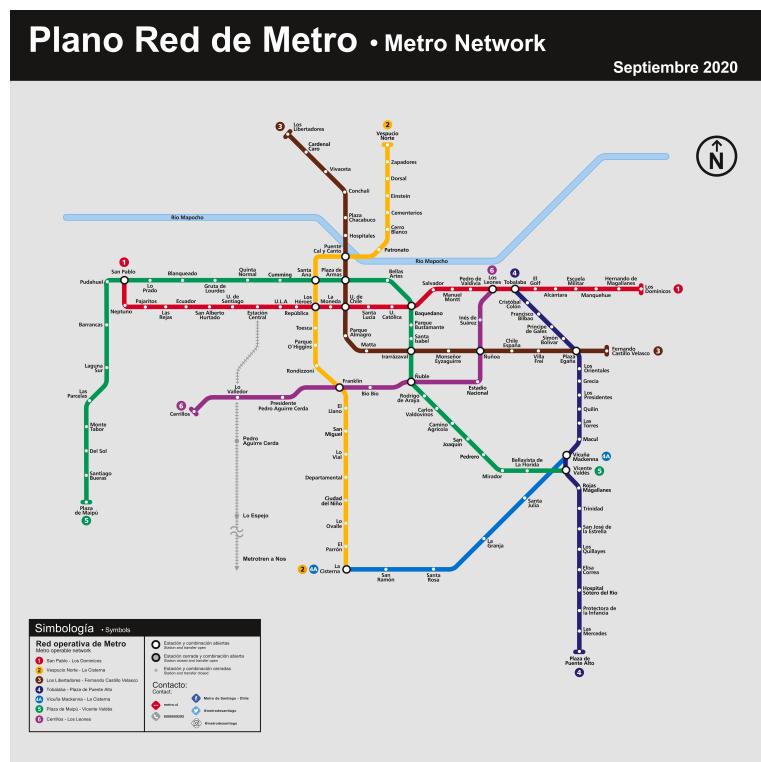


Figura 5.2: Ejemplo de uso: mapeo de las líneas del Metro de Santiago.

Como referencia, en la figura 5.3 se muestra el plano oficial provisto por Metro S.A., con las estaciones que se encuentran operativas al momento de la entrega de este informe:



Analizar la Red de Metro a escala real permite comprender mejor la extensión de la misma, y qué tan grande es la porción de la ciudad que está conectada por sus estaciones. Observando el mapa, se aprecia que el centro de Santiago está bastante interconectado por el Metro, pero algunos sectores, como el Nor-Poniente y el Sur de Santiago, carecen de conectividad en la red. Esto puede motivar proyectos de extensión de la Red, para beneficiar a los sectores que lo necesitan. Curiosamente, justo para la fecha límite de entrega de este informe (25 de septiembre de 2023), está programada la inauguración de la extensión de la Línea 3 del Metro hacia la comuna de Quilicura¹, lo que beneficiará a la conectividad del sector Nor-Poniente de la ciudad.

5.1.2. Análisis de densidad en paradas

Otro análisis que puede realizarse corresponde a estudiar las paradas del servicio. Por ejemplo, se puede estudiar la *densidad* de los recorridos en Santiago, es decir, analizar las paradas en las que se detiene una mayor cantidad de recorridos y observar si se presenta algún patrón notable. La función `calculate_stop_density` cruza la información de OSM y GTFS para marcar en el mapa las distintas paradas disponibles, generando una especie de *mapa de calor*.

Para efectos de comparación, en la figura 5.4 se presenta un mapa de densidad de todas las paradas del servicio de transporte público disponibles en Santiago, donde un color más rojizo representa paradas más densas (i.e. donde se detienen más recorridos), y así bajando paulatinamente hasta llegar al color morado.

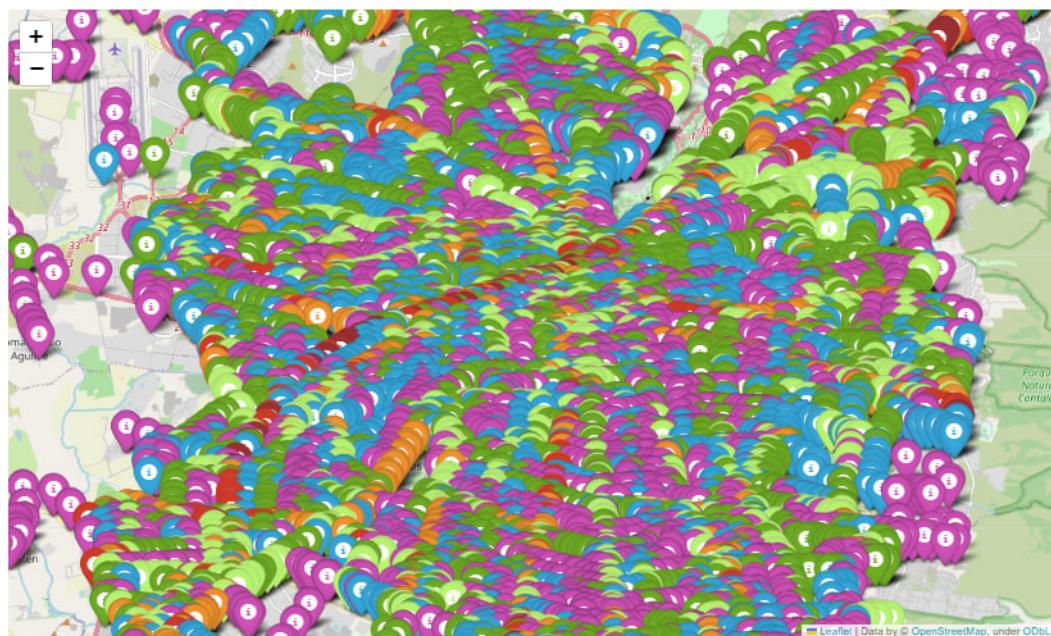


Figura 5.4: Ejemplo de uso: mapa de densidad de la totalidad de paradas del transporte público en Santiago.

¹Filtrado por el ingeniero Louis de Grange en X (<https://x.com/louisdegrange/status/1705262473073373598>).

Como es evidente, la cantidad de paradas en Santiago es tan grande que en la figura no se puede apreciar bien la información, mostrándose como una visualización bastante caótica. Sin embargo, se puede notar que en los sectores periféricos existe una mayor cantidad de paradas donde se detienen menos recorridos, al existir más marcadores de color morado que se pueden apreciar a simple vista.

Para poder apreciar mejor dónde se encuentran las paradas más densas, se modifica la función para filtrar las paradas y dejar sólo aquellas que presenten una cantidad destacable de recorridos (en este caso específico, 10 o más). En la figura 5.5, se observa esto en un nuevo mapa de Santiago:

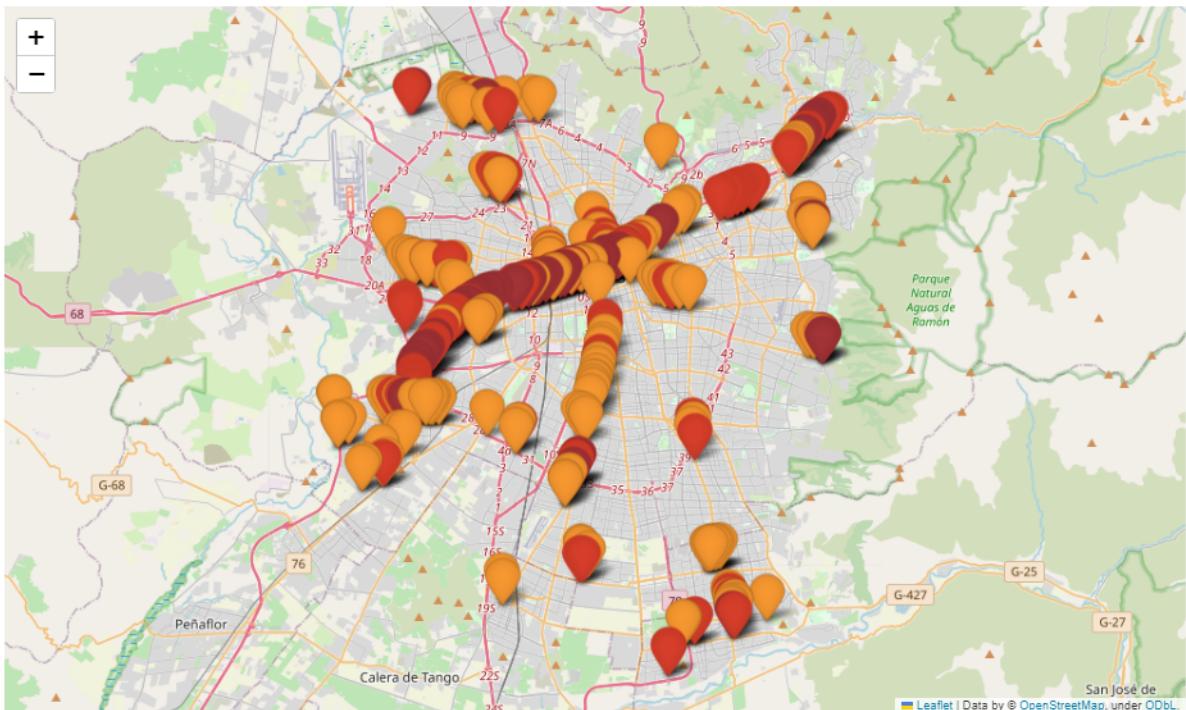


Figura 5.5: Ejemplo de uso: mapa de densidad de las paradas del transporte público en Santiago, donde se detienen 10 o más recorridos.

Aquí, se puede apreciar claramente cuál es el eje que mayor densidad de recorridos tiene, que corresponde al eje Pajaritos - Alameda - Providencia - Apoquindo, atravesando varias comunas (Maipú, Estación Central, Santiago Centro, Providencia, entre otras) y concentrando la mayor cantidad de gente que se moviliza por Santiago a diario y utiliza el transporte público. Además de este eje, también se aprecia que algunas de las avenidas más famosas y transitadas de la capital poseen una gran densidad de recorridos, como es el caso de la Avenida San Pablo en Pudahuel, Gran Avenida en San Miguel, o Avenida Grecia en Peñalolén. Sumado a esto, algunos sectores alejados del centro de Santiago también concentran una densidad considerable de recorridos, coincidiendo con los sectores céntricos de las comunas más periféricas de la ciudad, como es el caso de la Plaza de Quilicura en el sector Nor-Poniente, o la Plaza de Puente Alto en el sector Sur-Oriente.

Por otro lado, si se quiere analizar la cantidad de paradas que concentran la menor densidad de recorridos, se modifica nuevamente la función para aplicar este filtro, y dejar específicamente solo aquellas paradas donde se detiene un único recorrido. Esto se puede apreciar en la figura 5.6 a continuación:

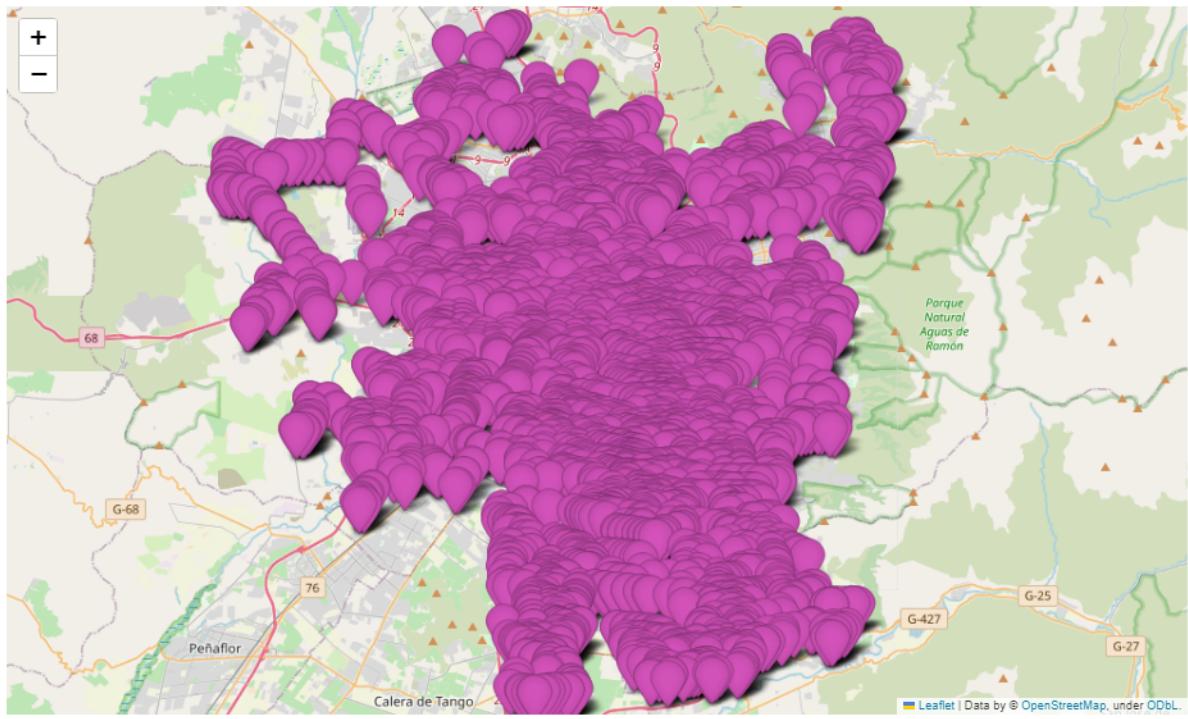


Figura 5.6: Ejemplo de uso: mapa de densidad de las paradas del transporte público en Santiago, donde se detiene solo un recorrido.

Como se aprecia en la figura, existe una cantidad considerable de paradas que alojan a un único recorrido de la red de transporte. Estas están presentes en toda la ciudad, sin enfocarse en ningún sector en específico. Si bien no existe ningún patrón, el tener presente que una gran mayoría de las paradas en Santiago se utilizan únicamente para un servicio, puede ser útil para planificar la creación de paradas nuevas o extensiones de servicios existentes.

5.1.3. Algoritmo de enrutamiento: Connection Scan Algorithm

Finalmente, para demostrar la utilidad del módulo, se realizan pruebas sobre la versión de Connection Scan Algorithm que fue implementada. Primero, en el escenario presentado en la siguiente figura, se busca una ruta para viajar entre una dirección en la comuna de Conchalí (Avenida Diagonal José María Caro #1553), y una dirección en Santiago Centro (Avenida Libertador Bernardo O'Higgins #706). Con respecto al momento del viaje, se decide una fecha arbitraria correspondiente a un día de semana (jueves 20 de julio), durante la madrugada (01:00 AM). La idea es demostrar que se puede crear una herramienta que filtre correctamente los diferentes tipos de recorrido; en este caso, los recorridos nocturnos. El resultado se presenta en la figura 5.7 a continuación:

Enter the travel's date, in DD/MM/YYYY format (press Enter to use today's date) : 20/07/2023
 20/07/2023
 Enter the travel's start time, in HH:MM:SS format (press Enter to start now) : 01:00:00
 01:00:00
 Enter the starting point's address, in 'Street #No, Province' format (Ex: 'Beauchef 850, Santiago'):Av. Diagonal Cardenal José María Caro 1553, Conchali
 Enter the ending point's address, in 'Street #No, Province' format (Ex: 'Campus Antumapu Universidad de Chile, Santiago'):Avenida Libertador Bernardo O'Higgins 706, Santiago
 Both addresses have been found.
 Processing...

Routes have been found.
 Calculating the best route and getting the arrival times for the next buses...

To go from: 1551, Avenida Diagonal Cardenal José María Caro, Conchali, Provincia de Santiago, Región Metropolitana de Santiago, 8710022, Chile
 To: 706, Avenida Libertador Bernardo O'Higgins, Barrio París-Londres, Santiago, Provincia de Santiago, Región Metropolitana de Santiago, 8330069, Chile

The best option is to walk for 1 minutes and 7 seconds to stop PB968, and take the route B31N.

The next bus arrives at 01:28.

The other two next buses arrives in:

87 minutes, 52 seconds (02:27)

147 minutes, 52 seconds (03:27)

You will get off the bus on stop PA91 after 24 minutes and 38 seconds.

After that, you need to walk for 2 minutes and 11 seconds to arrive at the target spot.

Total travel time: 55 minutes, 49 seconds. You will arrive your destination at 01:55.

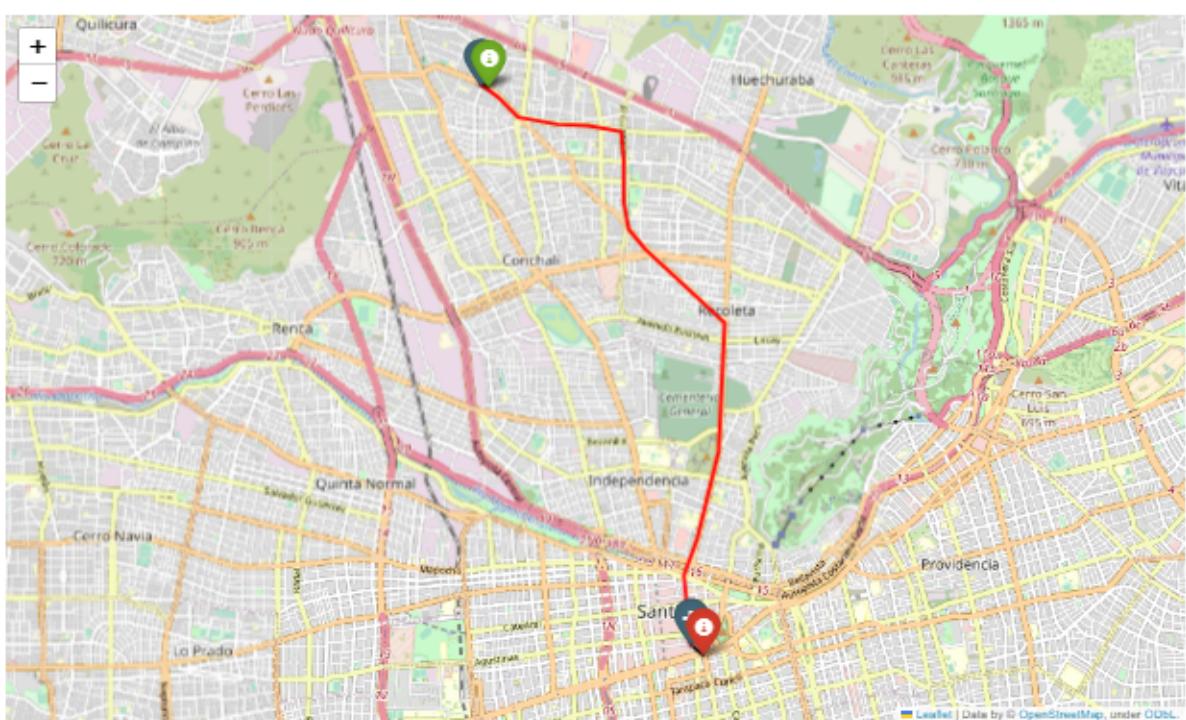


Figura 5.7: Ejemplo de uso: generación de ruta entre Conchalí y Santiago un jueves de madrugada.

Como se aprecia en la imagen, la salida de texto le indica al usuario cuál es la mejor opción, que en este caso corresponde a caminar a la parada PB968 para tomar el recorrido B31N. Considerando que la hora de inicio es a las 01:00, el siguiente bus del recorrido llegará a las 01:28. Luego de viajar aproximadamente por 25 minutos, el usuario deberá bajarse en la parada PA91 y caminar un par de minutos para llegar al destino, a las 01:55 (hora estimada). Con respecto al mapa, el marcador de color verde indica el punto de inicio, y el marcador de color rojo indica el punto de destino. Las paradas de subida y bajada están marcadas de color gris, y el recorrido del bus se indica como una línea roja, formada al unir las ubicaciones de las paradas intermedias.

Otro ejemplo útil a futuro es el que se presenta en la figura 5.8, que genera una ruta entre la Plaza de Quilicura y la Casa Central de la Universidad de Chile, en el centro de Santiago.

```

Enter the travel's date, in DD/MM/YYYY format (press Enter to use today's date) :
25/09/2023
Enter the travel's start time, in HH:MM:SS format (press Enter to start now) : 10:00:00
10:00:00
Enter the starting point's address, in 'Street #No, Province' format (Ex: 'Beauchef 850, Santiago'):Punto Centro, Quilicura
Enter the ending point's address, in 'Street #No, Province' format (Ex: 'Campus Antumapu Universidad de Chile, Santiago'):Unive
rsidad de Chile, Santiago
Both addresses have been found.
Processing...

Routes have been found.
Calculating the best route and getting the arrival times for the next buses...

To go from: Mall Arauco Quilicura, Avenida Bernardo O'Higgins, Doctor Dussert, Quilicura, Provincia de Santiago, Región Metropo
litana de Santiago, 8700000, Chile
To: Librería y Editorial Universitaria, 1050, Avenida Libertador Bernardo O'Higgins, Barrio París-Londres, Santiago, Provincia
de Santiago, Región Metropolitana de Santiago, 8331009, Chile

The best option is to walk for 1 minutes and 33 seconds to stop PB406, and take the route 314.
The next bus arrives at 10:03.
The other two next buses arrives in:
3 minutes, 36 seconds (10:03)
4 minutes, 6 seconds (10:04)

You will get off the bus on stop PA598 after 43 minutes and 35 seconds.
After that, you need to walk for 1 minutes and 39 seconds to arrive at the target spot.
Total travel time: 48 minutes, 18 seconds. You will arrive your destination at 10:48.

```

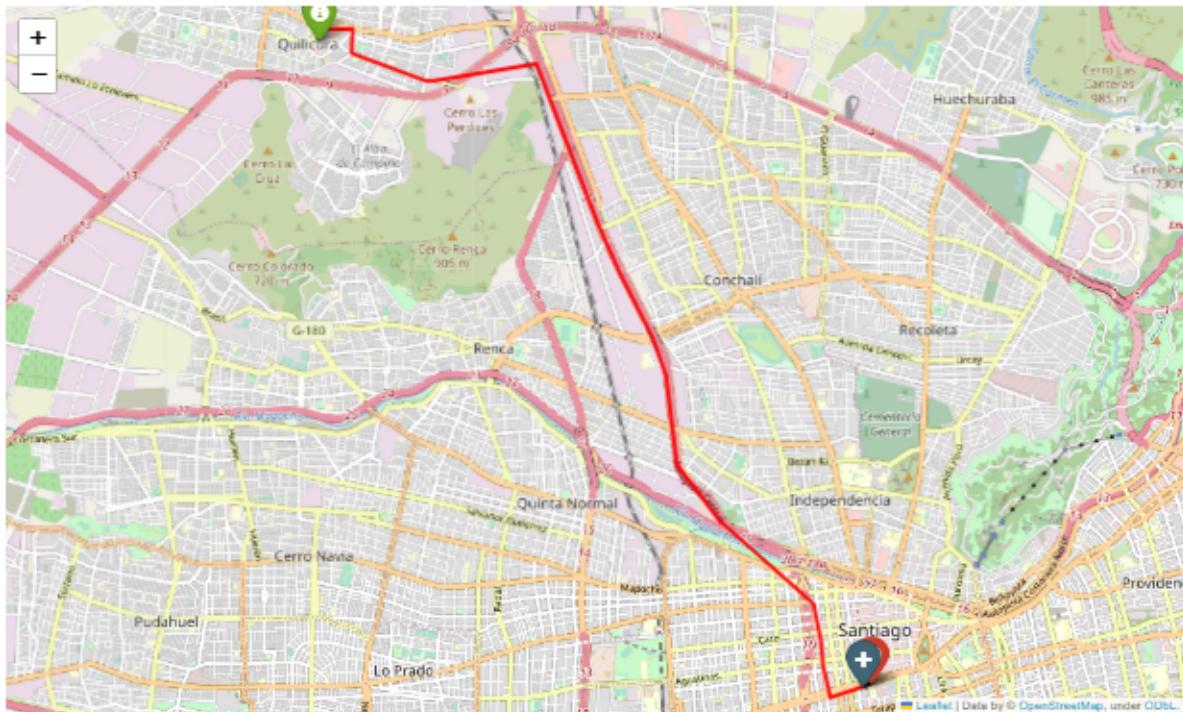


Figura 5.8: Ejemplo de uso: generación de ruta entre Plaza de Quilicura y la Casa Central de la Universidad de Chile, previo a la inauguración de la extensión de Línea 3 de Metro.

Como ya fue mencionado, el lunes 25 de septiembre es la inauguración de las nuevas estaciones de la Línea 3 del Metro, incluyendo precisamente a la nueva estación terminal, Plaza Quilicura. Como Universidad de Chile también pertenece a la Línea 3, próximamente se podrá realizar una ruta utilizando el Metro entre estos dos puntos. Por lo tanto, será útil comparar qué tan rápido será hacer el viaje en Metro en comparación a la mejor ruta calculada en los buses Red, que en este caso corresponde a tomar el recorrido 314.

5.2. Evaluación de resultados

Analizando los resultados, se ve que el módulo Ayatori es capaz de funcionar correctamente para permitir operar con la información del mapa de Santiago y del transporte público disponible. Se pueden crear visualizaciones que grafiquen diferentes secciones de la información incluida, creando funciones auxiliares que crucen la información de ambas capas. Incluso, se pueden implementar algoritmos de enrutamiento, generando rutas a través de la ciudad.

Como fue mencionado en la sección 3.3 del capítulo de Diseño, el criterio específico que fue elegido para evaluar el cumplimiento de los objetivos del proyecto consiste en que el usuario sea capaz de *crear herramientas para realizar estudios de movilidad* en Santiago. Esto implica que el usuario debe ser capaz de usar el módulo como base para generar visualizaciones que le permitan identificar patrones de movilidad, utilizando la información disponible para caracterizar las diferentes partes del sistema de transporte público (como recorridos o paradas), y así poder inferir información sobre el cómo se movilizan las personas en la ciudad. Para este caso, como se mostró en la sección anterior, esto se consigue exitosamente, habiendo generado varias visualizaciones diferentes e implementando una versión simplificada del algoritmo de enrutamiento Connection Scan Algorithm. Por lo tanto, al haberse satisfecho el criterio de evaluación, esto indica que se cumple correctamente el objetivo del proyecto, y los resultados se evalúan satisfactoriamente.

Capítulo 6

Discusión

En este capítulo, habiendo ya presentado los resultados obtenidos, se procede a realizar una discusión sobre las implicancias y limitaciones del proyecto. Además, se discuten ideas para continuar con el desarrollo del proyecto a modo de Trabajo Futuro.

6.1. Implicancias

La creación de la solución permitirá a potenciales usuarios crear herramientas y algoritmos que permitan realizar simulaciones de viajes en el sistema de transporte público disponible. Como se vio en el capítulo anterior, esta implementación logra su objetivo y es capaz de cruzar la información cartográfica y del transporte en Santiago, pudiendo analizarla y utilizarla para programar. Dado que el repositorio del proyecto estará disponible de forma pública para cualquiera que lo desee usar, esto permitirá que futuros potenciales usuarios puedan utilizar esta solución sin necesidad de una plataforma externa. Además, esto permitirá que puedan modificar las funcionalidades del módulo según su conveniencia.

Una gran implicancia derivada del último punto es que el código de la solución podría ser modificado para estudiar el transporte público en otras ciudades de Chile, o incluso, del extranjero, cambiando la ciudad al llamar a la función `get_network` de `pyrosm` (como se muestra en la sección 4.1.1) para obtener la información cartográfica , y utilizando un archivo en formato GTFS con los datos del transporte público de la misma al usar el módulo `Schedule` de `pygtfs` (como se muestra en la sección 4.1.2). La razón de esto es que la lógica tras el funcionamiento del código no está ligada directamente a los datos de Santiago, sino que necesita, simplemente, que la información desde OSM y desde GTFS sea provista correctamente y corresponda a la misma ciudad. Por ello, se abre un mundo de posibilidades para extender lo discutido en el desarrollo de este Trabajo de Título.

6.2. Limitaciones

Si bien la implementación de la solución logró sus objetivos al cumplir el criterio de evaluación estipulado, esto no le resta de tener ciertas limitaciones en su estado actual. Por ejemplo, una de sus limitantes principales es que los datos ingresados para el transporte público son todos de naturaleza *estática*, es decir, información previamente ingresada que, en teoría, debiera ser representativa de la realidad. Sin embargo, en la práctica, suelen ocurrir inconvenientes constantemente que generan que la realidad diste bastante de la teoría. El caso más directo de ver recae en los tiempos de espera de los buses, los que corresponden a tiempos aproximados que responden al cronograma mandatorio que los buses deben de seguir. Por este motivo, se dejan de lado variables que pueden existir al momento de realizar los viajes, como embotellamientos, malfuncionamiento del bus, u otras similares que puedan causar retrasos. Al no poseer información en tiempo real, los algoritmos programados sobre el módulo Ayatori funcionan bajo un supuesto de *continuidad operativa* constante, y al entregar una respuesta basada en este supuesto, pueden presentar un sesgo importante, especialmente en horarios complejos para la movilidad, como las horas punta.

Por otro lado, dado que el módulo requiere de información de transporte público entregada en el formato GTFS, una limitación directa es que la responsabilidad de utilizar la última versión disponible recae en el usuario, puesto que él es quien debe preocuparse de ingresar manualmente la última versión disponible cada vez que aparece una nueva actualización. El uso de una versión desactualizada puede llevar a otro sesgo, pudiendo llevar a, por ejemplo, omitir un recorrido nuevo que llegue a ser la solución óptima para obtener la mejor ruta. Además, en el caso de extender el funcionamiento para otra ciudad, basta que el organismo encargado del transporte no entregue la información en formato GTFS para que la simulación no pueda realizarse.

Además, los datos provistos por la solución están orientados al desarrollo de algoritmos que busquen realizar enrutamientos exclusivamente usando el transporte público disponible. Dado el diseño que posee, el módulo podría extenderse para calcular rutas utilizando otros medios de transporte, pero actualmente esa información no está considerada. Esto genera una arista explotable dentro de la implementación.

6.3. Trabajo Futuro

Inspirado directamente por las limitaciones señaladas anteriormente, se pueden derivar múltiples aristas a desarrollar para un posible trabajo a futuro sobre el módulo desarrollado. Además, durante el desarrollo del proyecto, existieron varias ideas de desarrollo e implementación que quedaron fuera del enfoque del Trabajo de Título, pero que pueden motivar una línea de trabajo adicional para el futuro. Las ideas presentes son:

- Desarrollar una versión que obtenga constantemente la información del transporte público en tiempo real, proveniente de bases como los GPS integrados en los buses o datos provistos por los mismos usuarios. Esto ya existe en otras plataformas que poseen objetivos similares, por lo que en teoría debiera ser posible agregarlo a esta implementación.

Esto permitiría que el algoritmo entregue una respuesta mucho más cercana a la realidad, y que pierda el sesgo de *continuidad operativa* antes mencionado, al menos de forma parcial.

- Añadir nuevas fuentes de datos cartográficos adicionales a OpenStreetMap, para adecuarse a otros medios de transporte fuera del transporte público. Por ejemplo, obtener información de las ciclovías disponibles para un usuario que desee movilizarse en bicicleta (pudiendo provenir también de proyectos comunitarios, como OpenCycleMap [2] o Bicineta Chile [4]). Esto puede aumentar las aristas estudiadas y desarrollar estudios de movilidad mucho más enriquecedores.
- Desarrollar una versión completamente *offline* de esta implementación, que no dependa de servicios externos ni conexión a Internet. Esto implica forzar al usuario a descargar previamente la información cartográfica de la ciudad (al igual que con *gtfs.zip*), con el fin de que se pueda acceder a la información en todo momento.
- Desarrollar un método de actualización de datos del transporte público, permitiendo automatizar la obtención de la información. Esto permitirá al usuario librarse de una responsabilidad para el funcionamiento del programa, mejorando la usabilidad del mismo. Esta idea puede extenderse a desarrollar una versión que permita entregar la información del transporte público en un formato distinto a GTFS, y/o que permita su traducción para usarlo como tal.
- Realizar una revisión completa a la implementación para mejorarla u optimizarla. Si bien el proyecto fue realizado con el cuidado de generar una implementación lo más limpia y eficiente posible, siempre puede existir lugar a mejoras. Además, dado que el funcionamiento del algoritmo está garantizado con el stack tecnológico discutido, en sus versiones disponibles a septiembre de 2023, a futuro será necesario actualizar las dependencias y librerías a las últimas versiones y realizar un chequeo general para verificar que todo siga funcionando correctamente.

Se espera que este proyecto pueda continuar su desarrollo en el futuro, ya que es un buen aporte como herramienta para el estudio de movilidad urbana, y posee un gran potencial explotable.

Capítulo 7

Conclusión

El trabajo de título tuvo por objetivo crear un módulo de Python que unificara la información cartográfica de OpenStreetMap con los datos de cronograma del transporte público en formato GTFS, con la finalidad de ser utilizado para crear algoritmos de enrutamiento que generen rutas entre dos puntos de Santiago. Su motivación fue el problema de estudiar la movilidad vial en una ciudad, algo bastante más crítico de lo que parece a primera vista. Toda herramienta a utilizar para realizar estudios de movilidad, con enfoque en el transporte público, requiere de tener los datos necesarios para poder estudiar la ciudad y los cronogramas de viajes del transporte, por lo que tener un módulo con esta información para poder programar sobre él se formuló como una buena apuesta de solución.

El resultado obtenido fue la creación del módulo Ayatori en Python, permitiendo desarrollar algoritmos de movilidad, cumpliendo así el objetivo principal del proyecto. Dado que la implementación considera la información cartográfica actual de Santiago (proveniente de OpenStreetMap), así como las especificaciones vigentes del transporte público (en formato GTFS y provistas por el Directorio de Transporte Público Metropolitano), se necesita estar constantemente actualizando los datos del módulo para permitir que opere con la información vigente; sin embargo, esta acción se puede hacer de forma sencilla, y permite la continuidad operativa de la solución. Aun así, dado que el módulo se alimenta de información estática (porque no recibe datos en vivo), al momento de ocurrir una eventualidad o emergencia, la implementación podría quedar igualmente sesgada bajo un supuesto de normalidad permanente en el estado del servicio. Esta línea de pensamiento puede ser utilizada para motivar trabajo futuro en esta área, mejorando la implementación para obtener información en directo del transporte público disponible.

En definitiva, la creación de este módulo representa un paso significativo hacia la mejora de la planificación de rutas de transporte público en Santiago. El producto de los algoritmos creados a partir de él genera valor y aportes en datos concretos a las organizaciones encargadas de administrar los servicios de transporte, abriendo nuevas posibilidades para el estudio y análisis de la movilidad urbana. Finalmente, la capacidad de adaptación y actualización sencilla de la base de datos del módulo sienta las bases para futuras investigaciones y desarrollos en busca de una movilidad más eficiente y resiliente en la ciudad.

Bibliografía

- [1] Volodymyr Agafonkin. Leaflet - a JavaScript library for interactive maps. Disponible en <https://leafletjs.com>. Revisado el 2023/09/25.
- [2] Andy Allan. OpenCycleMap - the OpenStreetMap Cycle Map. Disponible en <https://www.opencyclemap.org>. Revisado el 2023/09/25.
- [3] Graham Asher. CartoType — Powerful Software — Beautiful Maps. Disponible en <https://www.cartotype.com>. Revisado el 2023/09/25.
- [4] Bicineta Chile. Mapa de Ciclovías de la Región Metropolitana. Disponible en <https://www.bicineta.cl/ciclovias>. Revisado el 2023/09/25.
- [5] Fundación OpenStreetMap Chile. Mapa de OpenStreetMap Chile. Disponible en <https://www.openstreetmap.cl>. Revisado el 2023/09/25.
- [6] GTFS Community. General Transit Feed Specification. Disponible en <https://gtfs.org>. Revisado el 2023/09/25.
- [7] Yaron de Leeuw. pygtfs. Repositorio disponible en <https://github.com/jarondl/pygtfs>. Revisado el 2023/09/25.
- [8] Tiago de Paula Peixoto. graph-tool: Efficient network analysis with python. Documentación disponible en <https://graph-tool.skewed.de>. Revisado el 2023/09/25.
- [9] Directorio de Transporte Público Metropolitano. GTFS Vigente. Disponible en <https://www.dtpm.cl/index.php/gtfs-vigente>. Revisado el 2023/09/25. Última versión: 2023/09/23.
- [10] NetworkX developers. Networkx - network analysis in python. Documentación disponible en <https://networkx.org>. Revisado el 2023/09/25. Última versión: 2023/04/04.
- [11] devemux86. Cruiser - Map and Navigation Platform. Disponible en <https://github.com/devemux86/cruiser>. Revisado el 2023/09/25.
- [12] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithms*, 23(1.7):1–56, 2018.
- [13] Sozialhelden e.V. WheelMap - Busca lugares accesibles para sillas de ruedas. Disponible en <https://www.wheelmap.org>. Revisado el 2023/09/25.
- [14] Filipe Fernandes. Folium. Repositorio disponible en <https://github.com/python-visualization/folium>. Revisado el 2023/09/25.
- [15] OpenStreetMap (Global). Mapa de OpenStreetMap. Disponible en <https://www.openstreetmap.org>. Revisado el 2023/09/25.

- [16] Google. Google Maps. Disponible en <https://www.google.com/maps/>.
- [17] Eduardo Graells-Garrido. Aves: Análisis y Visualización, Educación y Soporte. Repositorio disponible en <https://github.com/zorzalerrante/aves>. Revisado el 2023/09/25.
- [18] Sarah Hoffmann. Nominatim. Disponible en <https://nominatim.org>. Revisado el 2023/09/25.
- [19] Universidad Alberto Hurtado. Actualización y recolección de información del sistema de transporte urbano, IX Etapa: Encuesta Origen Destino Santiago 2012. Encuesta origen destino de viajes 2012. Disponible en <http://www.sectra.gob.cl/biblioteca/detalle1.asp?mf=3253> (2012). Revisado el 2023/09/25. Última versión lanzada el 2014.
- [20] Kelsey Jordahl. Geopandas. Documentación disponible en <https://geopandas.org>. Revisado el 2023/09/25. Última versión: 2023/09/15.
- [21] Felipe Leal. CC6909-Ayatori (Repositorio del Trabajo de Título). Repositorio disponible en <https://github.com/Lysorek/CC6909-Ayatori>. Revisado el 2023/09/25.
- [22] Microsoft. Windows subsystem for linux. Documentación disponible en <https://learn.microsoft.com/en-us/windows/wsl/>. Revisado el 2023/09/25.
- [23] Melchior Moos. ÖPNVKarte. Disponible en <https://www.pnvkarte.de>. Revisado el 2023/09/25.
- [24] Linus Norton. Connection Scan Algorithm (implementación en TypeScript). Repositorio disponible en <https://github.com/planarnetwork/connection-scan-algorithm>. Revisado el 2023/09/25.
- [25] Data Reportal. Digital 2021 Report for Chile. Disponible en <https://datareportal.com/reports/digital-2021-chile> (2021/02/11). Revisado el 2023/09/25.
- [26] Audrey Roy and Cookiecutter community. Cookiecutter: Better project templates. Documentación disponible en <https://cookiecutter.readthedocs.io/en/stable/>. Revisado el 2023/09/25.
- [27] Jonas Sauer. ULTRA: UnLimited TRAnsfers for Multimodal Route Planning. Repositorio disponible en <https://github.com/kit-algo/ULTRA>. Revisado el 2023/09/25.
- [28] Victor Shcherb. OsmAnd - Offline Maps and Navigation. Disponible en <https://github.com/osmandapp/OsmAnd>. Revisado el 2023/09/25.
- [29] Henrikki Tenkanen. Pyrosm: Read OpenStreetMap data from Protobuf files into GeoDataFrame with Python, faster. Repositorio disponible en <https://github.com/HTenkanen/pyrosm>. Revisado el 2023/09/25.
- [30] Jochen Topf and Frederik Ramm. Geofabrik. Disponible en <https://www.geofabrik.de>. Revisado el 2023/09/25.
- [31] Jochen Topf and Frederik Ramm. Geofabrik download server - chile. Disponible en <https://download.geofabrik.de/south-america/chile.html>. Revisado el 2023/09/25. Última versión: 2023/09/24.
- [32] Papers with Code. Connection Scan Algorithm implementations. Disponible en <https://cs.paperwithcode.com/paper/connection-scan-algorithm>. Revisado el 2023/09/25.

ANEXOS

Apéndice A

Código de la solución

A.1. Clases del módulo

A.1.1. OSMGraph

```
1 import pyrosm
2 import numpy as np
3 import time as tm
4 from graph_tool.all import Graph
5 from geopy.exc import GeocoderServiceError
6 from geopy.geocoders import Nominatim
7
8 class OSMGraph(Graph):
9     def __init__(self, OSM_PATH='.'):
10         self.node_coords = {}
11         self.graph = self.create_osm_graph(OSM_PATH)
12
13     def download_osm_file(self, OSM_PATH):
14         """
15             Downloads the latest OSM file for Santiago.
16
17             Parameters:
18                 OSM_PATH (str): The directory where the OSM file will be saved
19
20             Returns:
21                 str: The path to the downloaded OSM file.
22
23             fp = pyrosm.get_data(
24                 "Santiago",
```

```

25         update=True,
26         directory=OSM_PATH
27     )
28
29     return fp
30
31 def create_osm_graph(self, OSM_PATH):
32     """
33     Creates a graph-tool's graph using the downloaded OSM data for
34     Santiago.
35
36     Returns:
37         graph: osm data converted to a graph
38     """
39     # Download latest OSM data
40     fp = self.download_osm_file(OSM_PATH)
41
42     osm = pyrosm.OSM(fp)
43
44     nodes, edges = osm.get_network(nodes=True)
45
46     graph = Graph()
47
48     # Create vertex properties for lon and lat
49     lon_prop = graph.new_vertex_property("float")
50     lat_prop = graph.new_vertex_property("float")
51
52     # Create properties for the ids
53     # Every OSM node has its unique id, different from the one given
54     # in the graph
55     node_id_prop = graph.new_vertex_property("long")
56     graph_id_prop = graph.new_vertex_property("long")
57
58     # Create edge properties
59     u_prop = graph.new_edge_property("long")
60     v_prop = graph.new_edge_property("long")
61     length_prop = graph.new_edge_property("double")
62     weight_prop = graph.new_edge_property("double")
63
64     vertex_map = {}
65
66     print("GETTING OSM NODES...")
67     for index, row in nodes.iterrows():
68         lon = row['lon']
69         lat = row['lat']
70         node_id = row['id']
71         graph_id = index
72         self.node_coords[node_id] = (lat, lon)
73
74         vertex = graph.add_vertex()
75         vertex_map[node_id] = vertex
76
77         # Assigning node properties
78         lon_prop[vertex] = lon
79         lat_prop[vertex] = lat
80         node_id_prop[vertex] = node_id

```

```

79         graph_id_prop[vertex] = graph_id
80
81     # Assign the properties to the graph
82     graph.vertex_properties["lon"] = lon_prop
83     graph.vertex_properties["lat"] = lat_prop
84     graph.vertex_properties["node_id"] = node_id_prop
85     graph.vertex_properties["graph_id"] = graph_id_prop
86
87     print("DONE")
88     print("GETTING OSM EDGES...")
89
90     for index, row in edges.iterrows():
91         source_node = row['u']
92         target_node = row['v']
93
94         if row["length"] < 2 or source_node == "" or target_node == "":
95             continue # Skip edges with empty or missing nodes
96
97         if source_node not in vertex_map or target_node not in
vertex_map:
98             print(f"Skipping edge with missing nodes: {source_node} ->
{target_node}")
99             continue # Skip edges with missing nodes
100
101         source_vertex = vertex_map[source_node]
102         target_vertex = vertex_map[target_node]
103
104         if not graph.vertex(source_vertex) or not graph.vertex(
target_vertex):
105             print(f"Skipping edge with non-existent vertices: {source_vertex} -> {target_vertex}")
106             continue # Skip edges with non-existent vertices
107
108         # Calculate the distance between the nodes and use it as the
weight of the edge
109         source_coords = self.node_coords[source_node]
110         target_coords = self.node_coords[target_node]
111         distance = np.linalg.norm(np.array(source_coords) - np.array(
target_coords))
112
113         e = graph.add_edge(source_vertex, target_vertex)
114         u_prop[e] = source_node
115         v_prop[e] = target_node
116         length_prop[e] = row["length"]
117         weight_prop[e] = distance
118
119         graph.edge_properties["u"] = u_prop
120         graph.edge_properties["v"] = v_prop
121         graph.edge_properties["length"] = length_prop
122         graph.edge_properties["weight"] = weight_prop
123
124         print("OSM DATA HAS BEEN SUCCESSFULLY RECEIVED")
125         return graph
126
127     def get_nodes_and_edges(self):

```

```

128     """
129         Returns a tuple containing two lists: one with the nodes and
130         another with the edges.
131     """
132     nodes = list(self.graph.vertices())
133     edges = list(self.graph.edges())
134     return nodes, edges
135
136     def print_graph(self):
137         """
138             Prints the vertices and edges of the graph.
139         """
140         print("Vertices:")
141         for vertex in self.graph.vertices():
142             print(f"Vertex ID: {int(vertex)}, lon: {self.graph.
143             vertex_properties['lon'][vertex]}, lat: {self.graph.vertex_properties['
144             lat'][vertex]}")
145
146         print("\nEdges:")
147         for edge in self.graph.edges():
148             source = int(edge.source())
149             target = int(edge.target())
150             print(f"Edge: {source} -> {target}")
151
152     def find_node_by_coordinates(self, lon, lat):
153         """
154             Finds a node in the graph based on its coordinates (lon, lat).
155
156             Parameters:
157                 lon (float): the longitude of the node.
158                 lat (float): the latitude of the node.
159
160             Returns:
161                 vertex: the vertex in the graph with the specified coordinates
162                 , or None if not found.
163             """
164             for vertex in self.graph.vertices():
165                 if self.graph.vertex_properties["lon"][vertex] == lon and self.
166                 graph.vertex_properties["lat"][vertex] == lat:
167                     return vertex
168             return None
169
170     def find_node_by_id(self, node_id):
171         """
172             Finds a node in the graph based on its id.
173
174             Parameters:
175                 node_id (long): the id of the node.
176
177             Returns:
178                 vertex: the vertex in the graph with the specified id, or None
179                 if not found.
180             """
181             for vertex in self.graph.vertices():
182                 if self.graph.vertex_properties["node_id"][vertex] == node_id:
183                     return vertex

```

```

178     return None
179
180     def find_nearest_node(self, latitude, longitude):
181         """
182             Finds the nearest node in the graph to a given set of coordinates.
183
184             Parameters:
185                 latitude (float): the latitude of the coordinates.
186                 longitude (float): the longitude of the coordinates.
187
188             Returns:
189                 vertex: the vertex in the graph closest to the given
190                 coordinates.
191
192         query_point = np.array([longitude, latitude])
193
194         # Obtains vertex properties: 'lon' and 'lat'
195         lon_prop = self.graph.vertex_properties['lon']
196         lat_prop = self.graph.vertex_properties['lat']
197
198         # Calculates the euclidean distances between the node's
199         # coordinates and the consulted address's coordinates
200         distances = np.linalg.norm(np.vstack((lon_prop.a, lat_prop.a)).T -
201             query_point, axis=1)
202
203
204         # Finds the nearest node's index
205         nearest_node_index = np.argmin(distances)
206         nearest_node = self.graph.vertex(nearest_node_index)
207
208
209         return nearest_node
210
211     def address_locator(self, address):
212         """
213             Finds the given address in the OSM graph.
214
215             Parameters:
216                 address (str): The address to be located.
217
218             Returns:
219                 int: The ID of the nearest vertex in the graph.
220
221             Raises:
222                 GeocoderServiceError: If there is an error with the geocoding
223                 service.
224
225         geolocator = Nominatim(user_agent="ayatori")
226         while True:
227             try:
228                 location = geolocator.geocode(address)
229                 break
230             except GeocoderServiceError:
231                 i = 0
232                 if i < 15:
233                     print("Geocoding service error. Retrying in 5 seconds
234 ..."))
235                     tm.sleep(5)

```

```

229             i+=1
230         else:
231             msg = "Error: Too many retries. Geocoding service may
232             be down. Please try again later."
233             print(msg)
234         return
235     if location is not None:
236         lat, lon = location.latitude, location.longitude
237         nearest = self.find_nearest_node(lat, lon)
238         return nearest
239     msg = "Error: Address couldn't be found."
240     print(msg)

```

A.1.2. GTFSData

```

1 import pygtfs
2 import os
3 import pandas as pd
4 from math import *
5 from datetime import datetime, date, time, timedelta
6 from graph_tool.all import Graph
7
8 class GTFSData:
9     def __init__(self, GTFS_PATH='gtfs.zip'):
10         self.scheduler = self.create_scheduler(GTFS_PATH)
11         self.graphs = {}
12         self.route_stops = {}
13         self.special_dates = []
14         self.stops = set()
15         self.graphs, self.route_stops, self.special_dates = self.
16         get_gtfs_data()
17         self.stops = self.get_stop_ids()
18
19     def create_scheduler(self, GTFS_PATH):
20         # Create a new schedule object using a GTFS file
21         scheduler = pygtfs.Schedule(":memory:")
22         pygtfs.append_feed(scheduler, GTFS_PATH)
23         return scheduler
24
25     def get_gtfs_data(self):
26         """
27             Reads the GTFS data from a file and creates a directed graph with
28             its info, using the 'pygtfs' library. This gives
29             the transit feed data of Santiago's public transport, including "
30             Red Metropolitana de Movilidad" (previously known
31             as Transantiago), "Metro de Santiago", "EFE Trenes de Chile", and
32             "Buses de Acercamiento Aeropuerto".
33
34             Returns:
35                 graphs: GTFS data converted to a dictionary of graphs, one per
36                 route.
37                 route_stops: Dictionary containing the stops for each route.
38                 special_dates: List of special calendar dates.
39             """

```

```

35     sched = self.scheduler
36
37     # Get special calendar dates
38     for cal_date in sched.service_exceptions: # Calendar_dates is
39         renamed in pygtfs
40             self.special_dates.append(cal_date.date.strftime("%d/%m/%Y"))
41
42     stop_id_map = {} # To assign unique ids to every stop
43     stop_coords = {}
44
45     for route in sched.routes:
46         graph = Graph(directed=True)
47         stop_ids = set()
48         trips = [trip for trip in sched.trips if trip.route_id ==
49         route.route_id]
50
51         # Create a new vertex property for node_id
52         node_id_prop = graph.new_vertex_property("string")
53
54         # Create edge properties
55         u_prop = graph.new_edge_property("object")
56         v_prop = graph.new_edge_property("object")
57         weight_prop = graph.new_edge_property("int")
58         graph.edge_properties["weight"] = weight_prop
59         graph.edge_properties["u"] = u_prop
60         graph.edge_properties["v"] = v_prop
61
62         added_edges = set() # To keep track of the edges that have
63         already been added
64
65         for trip in trips:
66             stop_times = trip.stop_times
67             orientation = trip.trip_id.split("-")[1]
68
69             for i in range(len(stop_times)):
70                 stop_id = stop_times[i].stop_id
71                 sequence = stop_times[i].stop_sequence
72
73                 if stop_id not in stop_id_map:
74                     vertex = graph.add_vertex()
75                     stop_id_map[stop_id] = vertex
76                 else:
77                     vertex = stop_id_map[stop_id]
78
79                 stop_ids.add(vertex)
80
81                 # Assign the node_id property to the vertex
82                 node_id_prop[vertex] = stop_id
83
84                 if i < len(stop_times) - 1:
85                     next_stop_id = stop_times[i + 1].stop_id
86
87                     if next_stop_id not in stop_id_map:

```

```

88                     next_vertex = stop_id_map[next_stop_id]
89
90                     edge = (vertex, next_vertex)
91                     if edge not in added_edges: # Check if the edge
92                         has already been added
93                         e = graph.add_edge(*edge)
94                         graph.edge_properties["weight"][e] = 1
95                         graph.edge_properties["u"][e] = node_id_prop[
96                             vertex]
97                         graph.edge_properties["v"][e] = node_id_prop[
98                             next_vertex]
99                         added_edges.add(edge) # Add the edge to the
100                        set of added edges
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133

```

```

134         else:
135             stops_by_direction["return_trip"].extend(stops)
136
137             round_trip_stops = set(stops_by_direction["round_trip"])
138             return_trip_stops = set(stops_by_direction["return_trip"])
139
140             for stop_id in round_trip_stops:
141                 if stop_id in stop_coords[route.route_id]:
142                     if stop_id in self.route_stops[route.route_id]:
143                         self.route_stops[route.route_id][stop_id][
144                             "orientation"] = "round"
145                     else:
146                         self.route_stops[route.route_id][stop_id] = {
147                             "route_id": route.route_id,
148                             "stop_id": stop_id,
149                             "coordinates": stop_coords[route.route_id][
150                                 stop_id],
151                             "orientation": "round",
152                             "sequence": sequence,
153                             "arrival_times": []
154                         }
155
156             for stop_id in return_trip_stops:
157                 if stop_id in stop_coords[route.route_id]:
158                     if stop_id in self.route_stops[route.route_id]:
159                         self.route_stops[route.route_id][stop_id][
160                             "orientation"] = "return"
161                     else:
162                         self.route_stops[route.route_id][stop_id] = {
163                             "route_id": route.route_id,
164                             "stop_id": stop_id,
165                             "coordinates": stop_coords[route.route_id][
166                                 stop_id],
167                             "orientation": "return",
168                             "sequence": sequence,
169                             "arrival_times": []
170                         }
171
172             for route_id, graph in self.graphs.items():
173                 weight_prop = graph.new_edge_property("int")
174
175                 for e in graph.edges():
176                     weight_prop[e] = 1
177
178                 graph.edge_properties["weight"] = weight_prop
179
180                 data_dir = "gtfs_routes"
181                 if not os.path.exists(data_dir):
182                     os.makedirs(data_dir)
183
184                 graph.save(f"{data_dir}/{route_id}.gt")
185
186                 print("GTFS DATA RECEIVED SUCCESSFULLY")
187
188             return self.graphs, self.route_stops, self.special_dates

```

```

186     def get_stop_ids(self):
187         stop_set = set()
188         for route_id, stops in self.route_stops.items():
189             for stop_id in stops:
190                 stop_set.add(stop_id)
191         return stop_set
192
193     def get_route_graph(self, route_id):
194         """
195             Given a route_id, returns the vertices and edges for the
196             corresponding graph.
197
198             Parameters:
199                 route_id (str): The ID of the route.
200
201             Returns:
202                 tuple: A tuple containing the vertices and edges of the graph. The
203                     vertices are a list of node IDs, and the edges are a list of tuples
204                     containing the source and target node IDs.
205         """
206
207         if route_id not in self.graphs:
208             print(f"Route {route_id} does not exist.")
209             return None
210
211         graph = self.graphs[route_id]
212         vertices = []
213         for v in graph.vertices():
214             node_id = graph.vertex_properties["node_id"][v]
215             if node_id != '' and node_id is not None:
216                 vertices.append(node_id)
217
218         edges = []
219         for e in graph.edges():
220             u = graph.edge_properties["u"][e]
221             v = graph.edge_properties["v"][e]
222             if u is not None and v is not None:
223                 edges.append((u, v))
224
225         return vertices, edges
226
227     def get_route_graph_vertices(self, route_id):
228         """
229             Given a route_id, returns the vertices for the corresponding graph
230
231             Parameters:
232                 route_id (str): The ID of the route.
233
234             Returns:
235                 list: A list containing the vertices of the graph. The vertices
236                     are a list of node IDs.
237         """
238
239         if route_id not in self.graphs:
240             print(f"Route {route_id} does not exist.")
241             return None
242
243

```

```

237     graph = self.graphs[route_id]
238     vertices = [graph.vertex_properties["node_id"][v] for v in graph.
239     vertices()]
240
241     return vertices
242
243 def get_route_graph_edges(self, route_id):
244     """
245         Given a route_id, returns the edges for the corresponding graph.
246
247     Parameters:
248         route_id (str): The ID of the route.
249
250     Returns:
251         list: A list containing the edges of the graph.
252     """
253     if route_id not in self.graphs:
254         print(f"Route {route_id} does not exist.")
255         return None
256
257     graph = self.graphs[route_id]
258     edges = [(graph.edge_properties["u"][e], graph.edge_properties["v"]
259     )[e]) for e in graph.edges()]
260
261     return edges
262
263 def map_route_stops(self, route_list, stops_flag, orientation_flag):
264     """
265         Create a map showing the stops visited on the round trip for the
266         specified routes.
267
268     Parameters:
269         route_list (list): A list of route IDs.
270         stops_flag (bool): A flag indicating whether to display the stops
271         on the map.
272
273     Returns:
274         folium.Map: A map object showing the stops and routes.
275     """
276     # Map the stops visited on the round trip
277     map = folium.Map(location=[-33.45, -70.65], zoom_start=12)
278
279     # List of valid colors
280     map_colors= ['red', 'orange', 'darkred', 'blue', 'lightblue', ,
281     green', 'purple', 'lightred', 'beige',
282             'darkblue', 'darkgreen', 'cadetblue', 'darkpurple', ,
283     white', 'pink', 'lightgreen',
284             'gray', 'black', 'lightgray']
285
286     color_id = 0
287     for route_id in route_list:
288         # Get the stops for the specified route
289         stops = self.route_stops.get(route_id, {})
290
291         # Filter the stops that are visited on the round trip
292         if orientation_flag:

```

```

287         trip_stops = [stop_info for stop_info in stops.values() if
288     stop_info["orientation"] == "round"]
289     else:
290         trip_stops = [stop_info for stop_info in stops.values() if
291     stop_info["orientation"] == "return"]
292
293     # Sort the stops by their sequence number in the trip
294     trip_stops = sorted(trip_stops, key=lambda x: x['sequence'])
295
296     folium.PolyLine(locations=[[stop_info["coordinates"][1],
297     stop_info["coordinates"][0]] for stop_info in trip_stops],
298                         color=map_colors[color_id], weight=4).add_to(
299     map)
300
301     if stops_flag:
302         for stop_info in trip_stops:
303             folium.Marker(location=[stop_info["coordinates"][1],
304     stop_info["coordinates"][0]], popup=stop_info["stop_id"],
305                         icon=folium.Icon(color='lightgray',
306     icon='minus')).add_to(map)
307
308     color_id+=1
309
310     return map
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333

```

```

334     Returns:
335     float: The distance between the two points in kilometers.
336     """
337     R = 6372.8 # Earth radius in kilometers
338     dLat = radians(lat2 - lat1)
339     dLon = radians(lon2 - lon1)
340     lat1 = radians(lat1)
341     lat2 = radians(lat2)
342     a = sin(dLat / 2)**2 + cos(lat1) * cos(lat2) * sin(dLon / 2)**2
343     c = 2 * asin(sqrt(a))
344     return R * c
345
346 def get_stop_coords(self, stop_id):
347     """
348         Given a stop ID, returns the coordinates of the stop with the
349         given ID.
350         If the stop ID is not found, returns None.
351
352     Parameters:
353         stop_id (int): The ID of the stop to get the coordinates for.
354
355     Returns:
356         tuple: A tuple of two floats representing the longitude and
357             latitude of the stop with the given ID.
358         None: If the stop ID is not found.
359     """
360     for route_id, stops in self.route_stops.items():
361         for stop_info in stops.values():
362             if stop_info["stop_id"] == stop_id:
363                 return stop_info["coordinates"]
364     return None
365
366 def get_near_stop_ids(self, coords, margin):
367     """
368         Given a tuple of coordinates and a margin, returns a list of stop
369         IDs
370         that are within the specified margin of the given coordinates,
371         along with their orientations.
372
373     Parameters:
374         coords (tuple): A tuple of two floats representing the longitude
375             and latitude of the coordinates to search around.
376         margin (float): The maximum distance (in kilometers) from the
377             given coordinates to include stops in the result.
378
379     Returns:
380         tuple: A tuple of two lists. The first list contains the stop IDs
381             that are within the specified margin of the given coordinates.
382             The second list contains tuples of stop IDs and their orientations
383
384     """
385     stop_ids = []
386     orientations = []
387     for route_id, stops in self.route_stops.items():
388         for stop_info in stops.values():
389             stop_coords = stop_info["coordinates"]

```

```

382             distance = self.haversine(coords[1], coords[0],
383                                         stop_coords[1], stop_coords[0])
384             if distance <= margin:
385                 orientation = stop_info["orientation"]
386                 stop_id = stop_info["stop_id"]
387                 if stop_id not in stop_ids:
388                     stop_ids.append(stop_id)
389                     orientations.append((stop_id, orientation))
390     return stop_ids, orientations
391
392     def get_route_stop_ids(self, route_id):
393         """
394             Given a route ID, returns a list of stop IDs for the stops on the
395             given route.
396
397             Parameters:
398                 route_id (int): The ID of the route to get the stops for.
399
400             Returns:
401                 list: A list of stop IDs for the stops on the given route.
402             """
403
404     def route_stop_matcher(self, route_id, stop_id):
405         """
406             Given a route ID, and a stop ID, returns True if the stop ID is on
407             the given route,
408             and False otherwise.
409
410             Parameters:
411                 route_id (int): The ID of the route to check.
412                 stop_id (int): The ID of the stop to check.
413
414             Returns:
415                 bool: True if the stop ID is on the given route, False otherwise.
416             """
417
418     def is_route_near_coordinates(self, route_id, coordinates, margin):
419         """
420             Given a route ID, a tuple of coordinates, and a margin, returns
421             True if the route
422                 has a stop within the specified margin of the given coordinates,
423             and False otherwise.
424
425             Parameters:
426                 route_id (int): The ID of the route to check.
427                 coordinates (tuple): A tuple of two floats representing the
428                     longitude and latitude of the coordinates to search around.
429                 margin (float): The maximum distance (in kilometers) from the
430                     given coordinates to include stops in the result.
431
432             Returns:
433                 bool: True if the route has a stop within the specified margin of

```

```

    the given coordinates, False otherwise.
431
432     """
433     for stop_info in self.route_stops[route_id].values():
434         stop_coords = stop_info["coordinates"]
435         distance = self.haversine(coordinates[1], coordinates[0],
436         stop_coords[1], stop_coords[0])
437         if distance <= margin:
438             return route_id
439     return False
440
441     def get_bus_orientation(self, route_id, stop_id):
442         """
443             Checks and confirms the bus orientation, while visiting a stop, in
444             the GTFS data files.
445
446             Parameters:
447                 route_id (str): The route or service's ID to check.
448                 stop_id (str): The visited stop ID.
449
450             Returns:
451                 str or list: The bus orientation(s) associated with the route_id
452             and stop_id. None if nothing is found.
453         """
454         stop_times = pd.read_csv("stop_times.txt")
455         filtered_stop_times = stop_times[(stop_times["trip_id"].str.
456         startswith(route_id)) & (stop_times["stop_id"] == stop_id)]
457
458         orientations = []
459         for trip_id in filtered_stop_times["trip_id"]:
460             orientation = trip_id.split("-")[1]
461             if orientation == "I" and "round" not in orientations:
462                 orientations.append("round")
463             elif orientation == "R" and "return" not in orientations:
464                 orientations.append("return")
465
466         if len(orientations) == 0:
467             return None
468         elif len(set(orientations)) == 1:
469             return orientations[0]
470         else:
471             return orientations
472
473     def connection_finder(self, stop_id_1, stop_id_2):
474         """
475             Finds all routes that have stops at both given stop IDs.
476
477             Parameters:
478                 stop_id_1 (str): The ID of the first stop to check.
479                 stop_id_2 (str): The ID of the second stop to check.
480
481             Returns:
482                 list: A list of route IDs that have stops at both given stop IDs.
483         """
484         connected_routes = []
485         for route_id, stops in self.route_stops.items():
486             stop_ids = [stop_info["stop_id"] for stop_info in stops.values]

```

```

()]

482     if stop_id_1 in stop_ids and stop_id_2 in stop_ids:
483         connected_routes.append(route_id)
484     return connected_routes

486

487     def get_routes_at_stop(self, stop_id):
488         """
489             Finds all routes that have a stop at the given stop ID.
490
491             Parameters:
492                 stop_id (str): The ID of the stop to check.
493
494             Returns:
495                 list: A list of route IDs that have a stop at the given stop ID.
496         """
497         routes = [route_id for route_id in self.route_stops.keys() if
498 stop_id in self.get_route_stop_ids(route_id) and self.connection_finder
499 (stop_id, stop_id)]
500         return routes

501

502     def is_24_hour_service(self, route_id):
503         """
504             Determines if the given route has a 24-hour service.
505
506             Parameters:
507                 route_id (str): A string representing the ID of the route.
508
509             Returns:
510                 bool: True if the route has a 24-hour service, False otherwise.
511         """
512
513         # Read the frequencies for the route
514         frequencies = pd.read_csv("frequencies.txt")
515         route_str = str(route_id) + "-"
516         route_frequencies = frequencies[frequencies["trip_id"].str.
517                                         startswith(route_str)]

518
519         # Check if any frequency has a start time of "00:00:00" and an end
520         # time of "24:00:00"
521         has_start_time = False
522         has_end_time = False
523         for _, row in route_frequencies.iterrows():
524             start_time = row["start_time"]
525             end_time = row["end_time"]
526             if start_time == "00:00:00":
527                 has_start_time = True
528             if end_time == "24:00:00":
529                 has_end_time = True

530
531         return has_start_time and has_end_time

532
533     def check_night_routes(self, valid_services, is_nighttime):
534         """
535             Filters the given list of route IDs to only include night routes
536             if is_nighttime is True.
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
999

```

```

532     Parameters:
533         valid_services (list): A list of route IDs to filter.
534         is_nighttime (bool): True if it is nighttime, False otherwise.
535
536     Returns:
537         list: A list of route IDs that are night routes if is_nighttime is
538             True, or all route IDs otherwise.
539         """
540         if is_nighttime:
541             #nighttime_routes = [route_id for route_id in valid_services
542             if route_id.endswith("N")]
543             nighttime_routes = [route_id for route_id in valid_services if
544                 route_id.endswith("N") or self.is_24_hour_service(route_id)]
545             if nighttime_routes:
546                 return nighttime_routes
547             else:
548                 return None
549             else:
550                 daytime_routes = [route_id for route_id in valid_services if
551                     not route_id.endswith("N")]
552                 if daytime_routes:
553                     return daytime_routes
554                 else:
555                     return None
556
557     def is_nighttime(self, source_hour):
558         """
559             Determines if the given hour is during the nighttime.
560
561             Parameters:
562                 source_hour (datetime.time): The hour to check.
563
564             Returns:
565                 bool: True if the hour is during the nighttime, False otherwise.
566             """
567             start_time = time(0, 0, 0)
568             end_time = time(5, 30, 0)
569             if start_time <= source_hour <= end_time:
570                 return True
571             else:
572                 return False
573
574     def is_holiday(self, date_string):
575         """
576             Checks if a given date is a holiday.
577
578             Parameters:
579                 date_string (str): A string representing the date in the format "
580                     dd/mm/yyyy".
581
582             Returns:
583                 bool: True if the date is a holiday, False otherwise.
584             """
585             # Local holidays
586             if date_string in self.special_dates:
587                 return True

```

```

583     date_obj = datetime.strptime(date_string, "%d/%m/%Y")
584
585     # Weekend days
586     day_of_week = date_obj.weekday()
587     if day_of_week == 5 or day_of_week == 6:
588         return True
589     return False
590
591 def is_rush_hour(self, source_hour):
592     """
593         Determines if the given hour is during rush hour.
594
595     Parameters:
596         source_hour (datetime.time): The hour to check.
597
598     Returns:
599         bool: True if the hour is during rush hour, False otherwise.
600     """
601     am_start_time = time(5, 30, 0)
602     am_end_time = time(9, 0, 0)
603     pm_start_time = time(17, 30, 0)
604     pm_end_time = time(21, 0, 0)
605     if am_start_time <= source_hour <= am_end_time or pm_start_time <=
606     source_hour <= pm_end_time:
607         return True
608     else:
609         return False
610
611 def check_express_routes(self, valid_services, is_rush_hour):
612     """
613         Filters the given list of route IDs to only include express routes
614         if is_rush_hour is True.
615
616     Parameters:
617         valid_services (list): A list of route IDs to filter.
618         is_rush_hour (bool): True if it is rush hour, False otherwise.
619
620     Returns:
621         list: A list of route IDs that are express routes if is_rush_hour
622         is True, or all route IDs otherwise.
623     """
624     if is_rush_hour:
625         return valid_services
626     else:
627         regular_hour_routes = [route_id for route_id in valid_services
628         if not route_id.endswith("e")]
629         return regular_hour_routes
630
631 def get_trip_day_suffix(self, date):
632     """
633         Based on the given date, gets the corresponding trip day suffix
634         for the trip IDs.
635
636     Parameters:
637         date (date): The date to be checked.

```

```

634     Returns
635     str: A string with the trip day suffix.
636     """
637     date_object = datetime.strptime(date, "%d/%m/%Y")
638     day_of_week = date_object.weekday()
639
640     if day_of_week < 5:
641         trip_day_suffix = "L"
642     elif day_of_week == 5:
643         trip_day_suffix = "S"
644     else:
645         trip_day_suffix = "D"
646
647     return trip_day_suffix
648
649 def get_arrival_times(self, route_id, stop_id, source_date):
650     """
651     Returns the arrival times for a given route and stop.
652
653     Parameters:
654     route_id (str): A string representing the ID of the route.
655     stop_id (str): A string representing the ID of the stop.
656     source_date (str): A string representing the date of the travel.
657
658     Returns:
659     tuple: A tuple containing a string representing the bus
660     orientation ("round" or "return") and a list of datetime objects
661     representing the arrival times.
662     """
663
664     # Read the frequencies.txt file
665     frequencies = pd.read_csv("frequencies.txt")
666
667     # Filter the frequencies for the given route ID
668     route_frequencies = frequencies[frequencies["trip_id"].str.
669     startswith(route_id)]
670
671     # Get the day suffix
672     day_suffix = self.get_trip_day_suffix(source_date)
673
674     # Get the arrival times for the stop for each trip
675     stop_route_times = []
676     bus_orientation = ""
677     for _, row in route_frequencies.iterrows():
678         start_time = pd.Timestamp(row["start_time"])
679         if row["end_time"] == "24:00:00":
680             end_time = pd.Timestamp("23:59:59")
681         else:
682             end_time = pd.Timestamp(row["end_time"])
683         headway_secs = row["headway_secs"]
684         round_trip_id = f"{route_id}-I-{day_suffix}"
685         return_trip_id = f"{route_id}-R-{day_suffix}"
686         round_stop_times = pd.read_csv("stop_times.txt").query(f"
687         trip_id.str.startswith('{round_trip_id}') and stop_id == '{stop_id}'")
688         return_stop_times = pd.read_csv("stop_times.txt").query(f"
689         trip_id.str.startswith('{return_trip_id}') and stop_id == '{stop_id}'")
690         if len(round_stop_times) == 0 and len(return_stop_times) == 0:

```

```

685         return
686     elif len(round_stop_times) > 0:
687         bus_orientation = "round"
688         stop_time = pd.Timestamp(round_stop_times.iloc[0][
689             "arrival_time"])
690     elif len(return_stop_times) > 0:
691         bus_orientation = "return"
692         stop_time = pd.Timestamp(return_stop_times.iloc[0][
693             "arrival_time"])
694     for freq_time in pd.date_range(start_time, end_time, freq=f"{headway_secs}s"):
695         freq_time_str = freq_time.strftime("%H:%M:%S")
696         freq_time = datetime.strptime(freq_time_str, "%H:%M:%S")
697         stop_route_time = datetime.combine(datetime.min, stop_time.
698             time()) + timedelta(seconds=(freq_time - datetime.min).seconds)
699         if stop_route_time not in stop_route_times:
700             stop_route_times.append(stop_route_time)
701         stop_time += pd.Timedelta(seconds=headway_secs)
702
703     return bus_orientation, stop_route_times
704
705
706
707     def get_time_until_next_bus(self, arrival_times, source_hour,
708     source_date):
709         """
710             Returns the time until the next three buses.
711
712             Parameters:
713                 arrival_times (list): A list of datetime objects representing the
714                 arrival times of the buses.
715                 source_hour (datetime.time): The source hour to compare with the
716                 arrival times.
717                 source_date (datetime.date): The source date to check if there are
718                 buses remaining.
719
720             Returns:
721                 list: A list of tuples representing the time until the next three
722                 buses in minutes and seconds.
723
724         arrival_times_remaining = []
725         for a_time in arrival_times:
726             if a_time.time() >= source_hour:
727                 arrival_times_remaining.append(a_time)
728             #arrival_times_remaining = [time for time in arrival_times if time.
729             #.time() >= source_hour]
730         if len(arrival_times_remaining) == 0:
731             return None
732         else:
733             # Sort the remaining arrival times in ascending order
734             arrival_times_remaining.sort()
735
736             # Get the datetime objects for the next three buses
737             next_buses = []
738             for i in range(min(3, len(arrival_times_remaining))):
739                 next_arrival_time = arrival_times_remaining[i]
740                 next_bus = datetime.combine(next_arrival_time.date(),

```

```

    next_arrival_time.time())
731        next_buses.append(next_bus)

732    if next_buses is None:
733        print("No buses remaining for the specified date.")
734    else:
735        # Calculate the time until the next three buses
736        time_until_next_buses = []
737        for next_bus in next_buses:
738            time_until_next_bus = (next_bus - datetime.combine(
739                next_bus.date(), source_hour)).total_seconds()
740            minutes, seconds = divmod(time_until_next_bus, 60)
741            time_until_next_buses.append((int(minutes), int(
742                seconds)))
743
744    return time_until_next_buses

745 def timedelta_to_hhmm(self, td):
746     """
747     Converts a timedelta object to a string in HHMM format.
748
749     Parameters:
750         td (timedelta): The timedelta object to be converted.
751
752     Returns:
753         str: A formated string with the time.
754     """
755     total_seconds = int(td.total_seconds())
756     hours = total_seconds // 3600
757     minutes = (total_seconds % 3600) // 60
758     return f"{hours:02d}:{minutes:02d}"

759 def timedelta_separator(self, td):
760     """
761     Separates a timedelta object into minutes and seconds.
762
763     Parameters:
764         td (timedelta): A timedelta object representing a duration of time
765
766     Returns:
767         tuple: A tuple containing the number of minutes and seconds in the
768             timedelta object. The minutes and seconds are both integers.
769     """
770     total_seconds = td.total_seconds()
771     minutes = int(total_seconds // 60)
772     seconds = int(total_seconds % 60)
773     return minutes, seconds

774 def get_travel_time(self, trip_id, stop_ids):
775     """
776     Returns the travel time between two stops for a given trip.
777
778     Parameters:
779         trip_id (str): A string representing the ID of the trip.
780         stop_ids (list): A list of two strings representing the IDs of the

```

```

    stops.

782
    Returns:
783     timedelta: A timedelta object representing the travel time.
784     """
785
786     stop_times = pd.read_csv("stop_times.txt").query(f"trip_id.str.
787     startswith('{trip_id}') and stop_id in {stop_ids}")
788     if len(stop_times) < 2:
789         return None
790     arrival_times = [datetime.strptime(arrival_time, "%H:%M:%S") for
791     arrival_time in stop_times["arrival_time"]]
792     travel_time = arrival_times[1] - arrival_times[0]
793     return travel_time

794
795     def get_trip_sequence(self, route_id, stop_id):
796         """
797             Given a dictionary of routes and stops, a route ID and a stop ID,
798             gets the trip sequence number corresponding to the stop.
799
800             Parameters:
801                 route_id (str): The route or service's ID.
802                 stop_id (str): The stop's ID.
803
804             Returns:
805                 str: A string representing the sequence number.
806                 """
807
808             seq = self.route_stops[route_id][stop_id]["sequence"]
809             return seq

810
811     def walking_travel_time(self, stop_coords, location_coords, speed):
812         """
813             Calculates the walking travel time between a location and a stop,
814             given a speed value.
815
816             Parameters:
817                 stop_coords (tuple): A tuple with the stop's coordinates.
818                 location_coords (tuple): A tuple with the location's coordinates.
819                 speed (float): The walking speed value.
820
821             Returns:
822                 float: The time (in seconds) that represents the travel time.
823                 """
824
825             distance = self.haversine(stop_coords[0], stop_coords[1],
826             location_coords[0], location_coords[1])
827             time = round((distance / speed) * 3600,2)
828             return time

829
830     def parse_metro_stations(self, stops_file):
831         """
832             Parses the Metro Stations data, creating a dictionary with their
833             names.
834
835             Parameters:
836                 stops_file (File): The GTFS file with the stop data (stops.txt).
837
838             Returns:

```

```

831     dict: A dictionary with the names of the stations.
832     """
833     subway_stops = {}
834     with open(stops_file, 'r') as f:
835         for line in f:
836             stop_id, _, stop_name, _, _, _, _ = line.strip().split(',')
837         )
838         if stop_id.isdigit():
839             subway_stops[stop_id] = stop_name
840     return subway_stops
841
842     def is_metro_station(self, stop_id, route_dict):
843         """
844             Checks if a stop is a Metro station.
845
846             Parameters:
847                 stop_id (str): The stop's ID to be checked.
848                 route_dict (dict): The dictionary with the Metro stations names.
849
850             Returns:
851                 str or None: A string with the stop ID if the stop is a Metro
852                 station, or None if it isn't.
853             """
854             try:
855                 route_num = int(stop_id)
856                 return route_dict[stop_id]
857             except ValueError:
858                 return None

```

A.2. Algoritmo de ejemplo: Connection Scan Algorithm

```

1 def connection_scan_lite(source_address, target_address, departure_time,
2                           departure_date, margin):
3     """
4         The Connection Scan Algorithm is applied to search for travel routes
5         from the source to the destination,
6         given a departure time and date. By default, the algorithm uses the
7         current date and time of the system.
8         However, you can specify a different date or time if needed. The
9         margin value let's the user determine
10        the range on which a stop is considered as "near" to the source or
11        target addresses.
12        Note: this is a "lite" version of CSA that maps possible routes
13        without doing any transfers.
14
15        Parameters:
16            source_address (string): the source address of the travel.
17            target_address (string): the destination address of the travel.
18            departure_time (time): the time at which the travel should start.
19            departure_date (date): the date on which the travel should be done.
20            margin (float): margin of distance between the nodes and the valid
21            stops.

```

```

16     Returns:
17     folium.Map: the map of the best travel route. It returns None if no
18     routes are found.
19     """
20
21     # Getting the nodes corresponding to the addresses
22     source_node = osm_graph.address_locator(source_address)
23     target_node = osm_graph.address_locator(target_address)
24
25     # Instance of the route_stops dictionary
26     route_stops = gtfs_data.route_stops
27
28     if source_node is not None and target_node is not None:
29         # Convert source and target node IDs to integers
30         source_node_graph_id = osm_graph.graph.vertex_properties["graph_id"]
31         [source_node]
32         target_node_graph_id = osm_graph.graph.vertex_properties["graph_id"]
33         [target_node]
34
35         print("Both addresses have been found.")
36         print("Processing...")
37
38         geolocator = Nominatim(user_agent="ayatori")
39
40         route_info = available_route_finder(osm_graph, gtfs_data,
41         source_node_graph_id, target_node_graph_id, departure_time,
42         departure_date, margin, geolocator)
43
44         selected_path = route_info[0]
45         source = route_info[1]
46         target = route_info[2]
47         valid_source_stops = route_info[3]
48         valid_target_stops = route_info[4]
49         valid_services = route_info[5]
50         fixed_orientation = route_info[6]
51         near_source_stops = route_info[7]
52         near_target_stops = route_info[8]
53
54         # Create a map that shows the correct public transport services to
55         # take from the source to the target
56         m = folium.Map(location=[selected_path[0][0], selected_path
57         [0][1]], zoom_start=13)
58
59         # Add markers for the source and target points
60         folium.Marker(location=[selected_path[0][0], selected_path[0][1]],
61         popup="Origen: {}".format(source), icon=folium.Icon(color='green')).add_to(m)
62         folium.Marker(location=[selected_path[-1][0], selected_path
63         [-1][1]], popup="Destino: {}".format(target), icon=folium.Icon(color='red')).add_to(m)
64
65         print("")
66         print("Routes have been found.")
67         print("Calculating the best route and getting the arrival times
68         for the next buses...")
69
70         best_option_info = find_best_option(osm_graph, gtfs_data,

```

```

selected_path, departure_time, departure_date, valid_source_stops,
valid_target_stops, valid_services, fixed_orientation)

60
61     best_option = best_option_info[0]
62     initial_delta_time = best_option_info[1]
63     best_option_times = best_option_info[2]
64     initial_source_time = best_option_info[3]
65     valid_target = best_option_info[4]
66     best_option_orientation = best_option_info[5]

67
68     if best_option is None:
69         print("Error: There are no available services right now to go
70             to the desired destination.")
71         print("Possible reasons: the valid routes are not available at
72             the specified date or starting time.")
73         print("Please take into account that some routes have trips
74             only during or after nighttime, which goes between 00:00:00 and
75             05:30:00")
76         return
77
78     arrival_time = None
79
80     source_stop = best_option[1]

81     # Parse Metro stations's names
82     metro_stations_dict = gtfs_data.parse_metro_stations("stops.txt")
83     possible_metro_name = gtfs_data.is_metro_station(best_option[1],
84     metro_stations_dict)
85     if possible_metro_name is not None:
86         source_stop = possible_metro_name

87     walking_minutes, walking_seconds = gtfs_data.timedelta_separator(
88     initial_delta_time)

89
90     print("")
91     print("To go from: {}".format(source))
92     print("To: {}".format(target))
93     best_arrival_time_str = gtfs_data.timedelta_to_hhmm(best_option
94     [2])
95     print("")
96     if possible_metro_name is not None: # Changes the printing to
97         adapt for the use of Metro
98         print("The best option is to walk for {} minutes and {}"
99             "seconds to {} Metro station, and take the line {}.".format(
100             walking_minutes, walking_seconds, source_stop, best_option[0]))
101         print("The next train arrives at {}.".format(
102             best_arrival_time_str))
103         print("The other two next trains arrives in:")
104     else:
105         print("The best option is to walk for {} minutes and {}"
106             "seconds to stop {}, and take the route {}.".format(walking_minutes,
107             walking_seconds, source_stop, best_option[0]))
108         print("The next bus arrives at {}.".format(
109             best_arrival_time_str))
110         print("The other two next buses arrives in:")

```

```

100     # Format and prints the times
101     for i in range(len(best_option_times)):
102         if i == 0:
103             continue
104         minutes, seconds = best_option_times[i]
105         waiting_time = timedelta(minutes=minutes, seconds=seconds)
106         arrival_time = initial_source_time + waiting_time
107         time_string = gtfs_data.timedelta_to_hhmm(arrival_time)
108         print(f"{minutes} minutes, {seconds} seconds ({time_string})")
109
110     # Base Coordinates
111     source_lat = selected_path[0][0]
112     source_lon = selected_path[0][1]
113     target_lat = selected_path[-1][0]
114     target_lon = selected_path[-1][1]
115
116
117     for stop_id in near_source_stops:
118         if stop_id in valid_source_stops:
119             # Filters the data for selecting the best source option
120             for its_mapping:
121                 stop_coords = gtfs_data.get_stop_coords(str(stop_id))
122                 routes_at_stop = gtfs_data.get_routes_at_stop(stop_id)
123                 valid_stop_services = [stop_id for stop_id in
124                                         valid_services if stop_id in routes_at_stop]
125
126                 for service in valid_stop_services:
127                     if service == best_option[0] and stop_id ==
128                         best_option[1]:
129                         # Maps the best option to take the best option's
130                         service
131                         folium.Marker(location=[stop_coords[1],
132                                         stop_coords[0]],
133                                         popup="Mejor opcion: subirse al recorrido {}"
134                                         en la parada {}.format(best_option[0], best_option[1]),
135                                         icon=folium.Icon(color='cadetblue', icon='
136                                         plus')).add_to(m)
137                         initial_distance = [(selected_path[0][0],
138                                         selected_path[0][1]), (stop_coords[1],
139                                         stop_coords[0])]
140                         folium.PolyLine(initial_distance, color='black',
141                                         dash_array='10').add_to(m)
142
143                         for stop_id in near_target_stops:
144                             if stop_id in valid_target_stops:
145                                 # Filters the data for the possible target stops
146                                 stop_coords = gtfs_data.get_stop_coords(str(stop_id))
147                                 routes_at_stop = gtfs_data.get_routes_at_stop(stop_id)
148                                 valid_stop_services = [stop_id for stop_id in
149                                         valid_services if stop_id in routes_at_stop]
150
151                                 target_orientation = None
152                                 for service in valid_target:
153                                     if service == best_option[0]:
154                                         # Generates the trip id to get the approximated travel
155                                         time
156                                         if fixed_orientation == "round":

```

```

145             trip_id = service + "-I-" + gtfs_data.
146             get_trip_day_suffix(departure_date)
147         else:
148             trip_id = service + "-R-" + gtfs_data.
149             get_trip_day_suffix(departure_date)
150
151             best_travel_time = None
152             selected_stop = None
153             for stop_id in valid_target_stops:
154                 # Calculates the travel time while taking the service
155                 bus_time = gtfs_data.get_travel_time(trip_id, [
156                     best_option[1], stop_id])
157                 target_stop_routes = gtfs_data.get_routes_at_stop(
158                     stop_id)
159                 target_orientation = gtfs_data.get_bus_orientation(
160                     best_option[0], stop_id)
161                 if service in target_stop_routes and bus_time >
162                     timedelta() and (best_travel_time is None or bus_time <
163                     best_travel_time):
164                     # Checking the correct orientation
165                     if fixed_orientation in target_orientation:
166                         # Updates the selected target stop and travel
167                         time
168                         best_travel_time = bus_time
169                         selected_stop = stop_id
170
171                         # Gets the coordinates for the target stop
172                         selected_stop_coords = gtfs_data.get_stop_coords(
173                             selected_stop)
174                         # Separates the best travel time for the printing
175                         minutes, seconds = gtfs_data.timedelta_separator(
176                             best_travel_time)
177
178                         # Gets the sequence number for the source and target stops
179                         seq_1 = route_stops[best_option[0]][best_option[1]][""
180                             "sequence"]
181                         seq_2 = route_stops[best_option[0]][selected_stop]["
182                             "sequence"]
183
184                         # Store the coordinates of the visited stops for their
185                         mapping
186                         visited_stops = []
187
188                         # Iterate over the stops of the selected route
189                         for stop_id, stop_info in route_stops[best_option[0]].
190                             items():
191                             # Check if the stop sequence number is between seq_1
192                             and seq_2
193                             seq_number = stop_info["sequence"]
194                             this_orientation = gtfs_data.get_bus_orientation(
195                                 best_option[0], stop_id)
196                             if best_option_orientation in this_orientation and
197                                 seq_1 <= seq_number <= seq_2:
198                                 # Append the coordinates of the stop to the
199                                 visited_stops list
200                                 lat = stop_info["coordinates"][0]

```

```

183                 lon = stop_info["coordinates"][1]
184                 visited_stops.append((seq_number, (lon, lat)))
185
186             # Sorts the visited stops and gets their coordinates
187             visited_stops_sorted = sorted(visited_stops, key=lambda x:
188                 x[0])
189             visited_stops_sorted_coords = [x[1] for x in
190                 visited_stops_sorted]
191
192             # Checks if the stop is a Metro Station (they are stored
193             # as a number)
194             possible_metro_target_name = gtfs_data.is_metro_station(
195                 selected_stop, metro_stations_dict)
196
197             if possible_metro_target_name is not None:
198                 selected_stop = possible_metro_target_name
199
200                 print("")
201                 if possible_metro_name is not None: # Changes the message
202                     print("You will get off the train on {} station after
203                         {} minutes and {} seconds.".format(selected_stop, minutes, seconds))
204                 else:
205                     print("You will get off the bus on stop {} after {}
206                         {} minutes and {} seconds.".format(selected_stop, minutes, seconds))
207
208                 # Maps the best option to get off the best option's
209                 # service
210                 folium.Marker(location=[selected_stop_coords[1],
211                     selected_stop_coords[0]],
212                     popup="Mejor opcion: bajarse del recorrido {} en la
213                     parada {}.".format(best_option[0], selected_stop),
214                     icon=folium.Icon(color='cadetblue', icon='plus')).add_to(m)
215
216                 ending_distance = [(selected_path[-1][0], selected_path
217                     [-1][1]),(selected_stop_coords[1], selected_stop_coords[0])]
218                 folium.PolyLine(ending_distance,color='black',dash_array=
219                     [10]).add_to(m)
220
221
222                 # Create a polyline connecting the visited stops
223                 folium.PolyLine(visited_stops_sorted_coords, color='red').add_to(m)
224
225                 # Gets the coordinates for the target stop and target
226                 # location
227                 final_stop_coords = (selected_stop_coords[1],
228                     selected_stop_coords[0])
229                 final_location_coords = (target_lat, target_lon)
230
231                 # Calculates the walking time between the target stop and
232                 # location
233                 end_walking_time = gtfs_data.walking_travel_time(
234                     final_stop_coords, final_location_coords, 5)
235                 end_delta_time = timedelta(seconds=end_walking_time)
236                 end_walk_min, end_walk_sec = gtfs_data.timedelta_separator
237                     (end_delta_time)
238
239
240

```

```

221             # Time walking to stop + waiting the bus + riding the bus
222             + walking to target destination
223             total_time = initial_delta_time + best_option[3] +
best_travel_time + end_delta_time
224             minutes, seconds = gtfs_data.timedelta_separator(
total_time)
225
226             # Parses the time for the printing
227             destination_time = initial_source_time + total_time
time_string = gtfs_data.timedelta_to_hhmm(destination_time)
228             print(f"After that, you need to walk for {end_walk_min}
minutes and {end_walk_sec} seconds to arrive at the target spot.")
229             print(f"Total travel time: {minutes} minutes, {seconds}
seconds. You will arrive your destination at {time_string}.")
230
231             # Set the optimal zoom level for the map
232             fit_bounds(selected_path, m)
233
234             return m
235         else:
236             # Empty return
237             return

```