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 *lite* 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, reirte 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. 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.

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, todo su aceptación y amor. A toda la familia Luna, por aceptarme como uno más entre los suyos, y por todo el cariño, consejos, y buenas vibras.

Gracias a mi curso, 12°B, por acompañarme en la primera parte de mi vida y por los amigos que encontré entre sus filas. 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.

Tabla de Contenido

Índice de Ilustraciones

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, correctamente, 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 implica que dicho estudio se pueda realizar sin inconvenientes notables. 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 [18]. 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 común 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, sería útil verlo 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, esto incluye al Metro de Santiago y los buses de Red (antiguamente Transantiago), 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) [5] 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 múltiples algoritmos que se han diseñado con el fin de responder a consultas de movilidad. Por ejemplo, Connection Scan Algorithm [11] (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 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 visibilicen 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 algorimos de movilidad, focalizando su uso en Santiago de Chile. Para ello, se utilizarán los datos cartográficos de la ciudad provenientes de OpenStreetMap, un proyecto colaborativo de creación de mapas comunitarios [4], 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, 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 [14]. 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 [4] 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 vérsatil.

Existen múltiples mapas online que utilizan la información de OpenStreetMap. El más conocido es el que tiene la propia web de OpenStreetMap [14], 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.

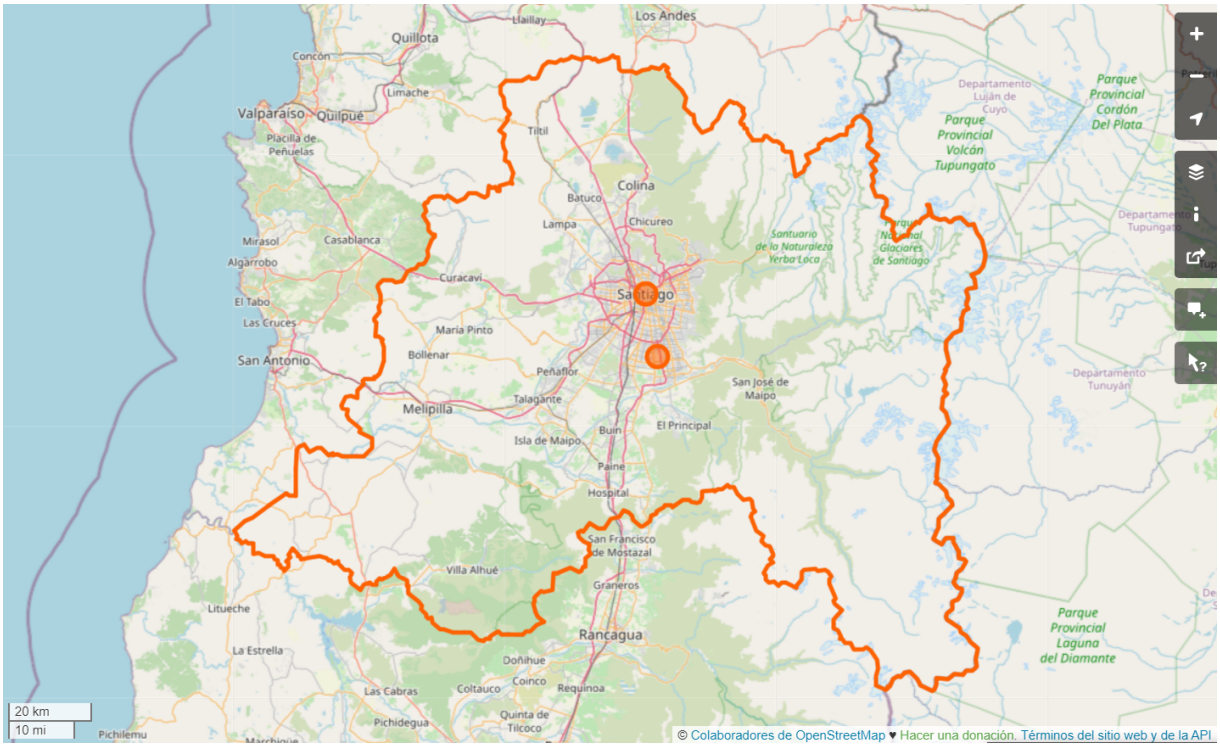

Figura 2.1: Mapa de la Región Metropolitana en OpenStreetMap. Fuente: openstreetmap.org
.

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) [1], WheelMap se encarga de mostrar los lugares que son accesibles para usuarios de sillas de ruedas [12], y ÖPNVKarte es un mapa que grafica las redes de transporte público disponibles [22].

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 [10] en Android, y OsmAnd [27] 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++ [2].

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** [28] 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.

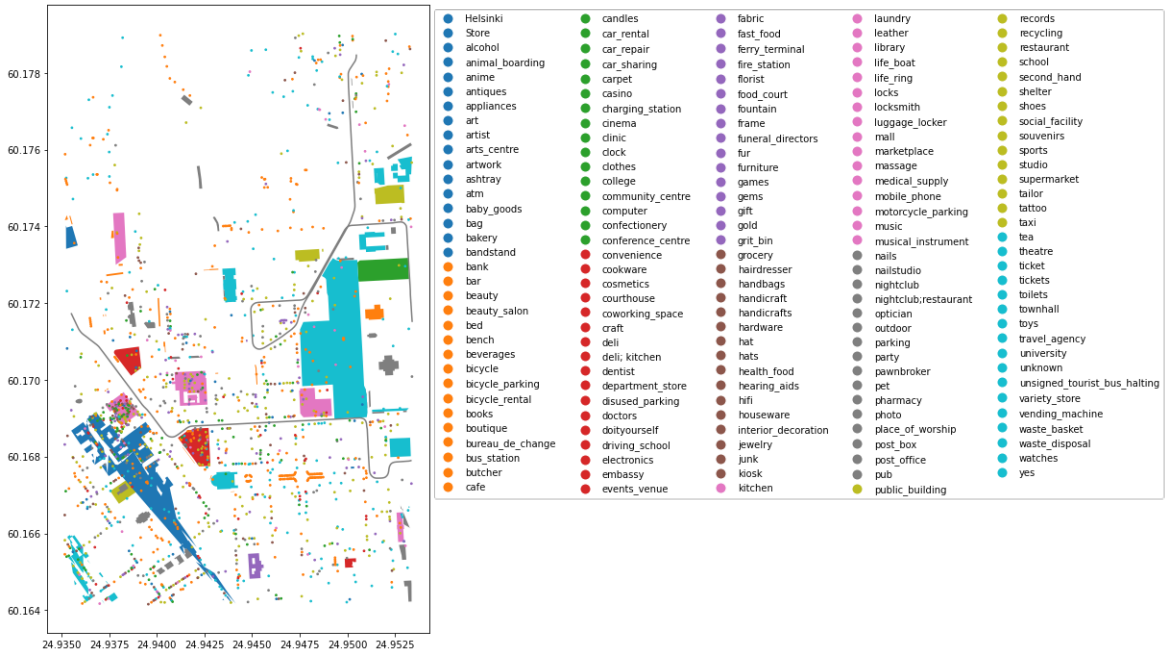

Figura 2.2: Mapa de la ciudad de Helsinki, Finlandia, que destaca sus puntos de interés. Fuente: pyrosm.readthedocs.io.

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:

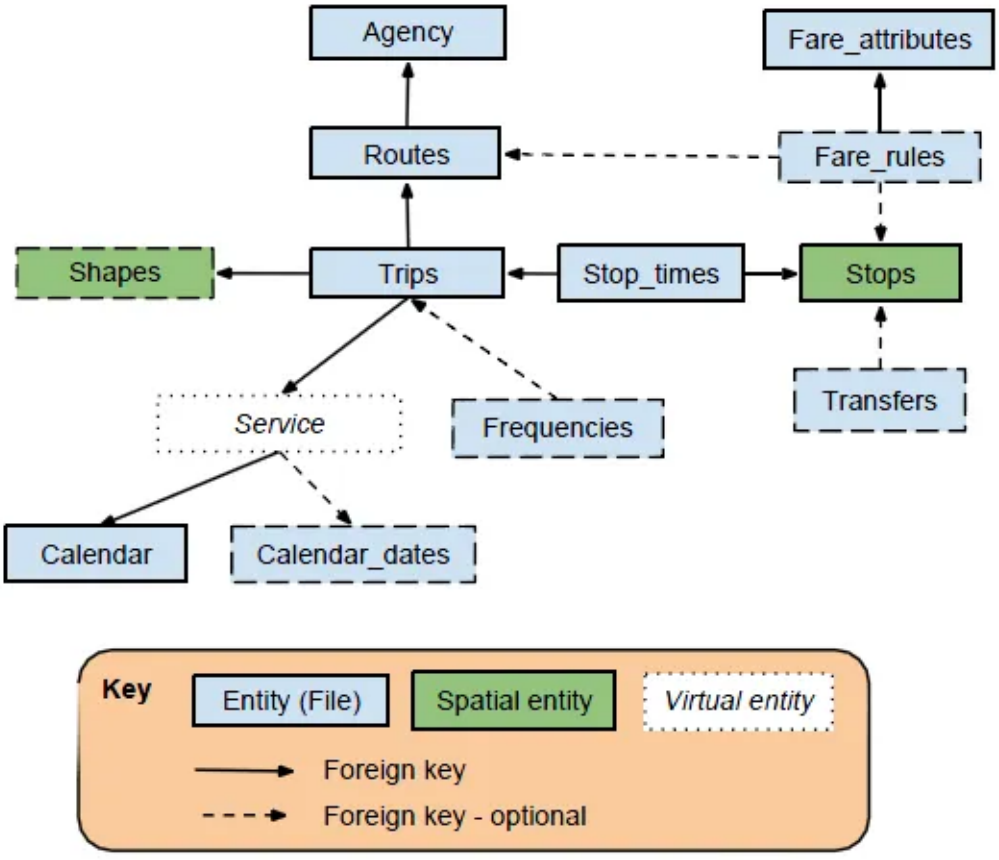

Figura 2.3: Diagrama relacional de los archivos que componen el formato GTFS. Fuente: medium.com .

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 [8]. Al momento de la entrega de este informe, la última versión fue configurada para implementarse desde el 16 de septiembre de 2023.

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, las cuales se describen a continuación:

- **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.

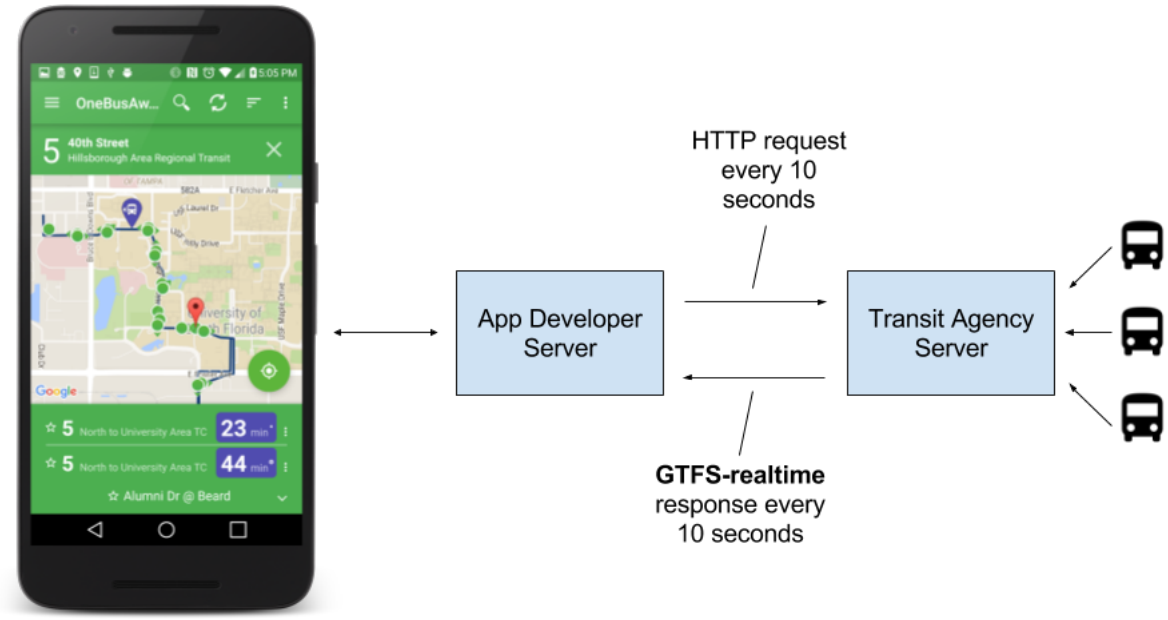

Figura 2.4: Diagrama de uso de datos en tiempo real en formato GTFS para una aplicación. Fuente: watrifeed.ml .

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 con la información programada de los recorridos.

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** [6] 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 los datos de una de las tablas.

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 [9]. 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 [7]. 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) [21].

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 [11]. 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:

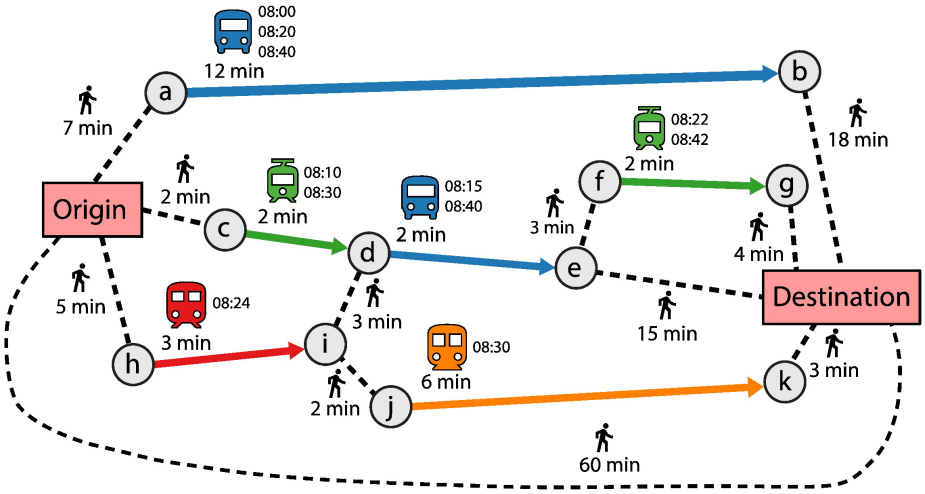

Figura 2.5: Diagrama explicativo del funcionamiento de Connection Scan Algorithm. Fuente: "Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys" (R. Kujala et al., 2017), vía sciencedirect.com .

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 [15], por su simplicidad de uso. Las rutas aparecen enumeradas según la lista previa.

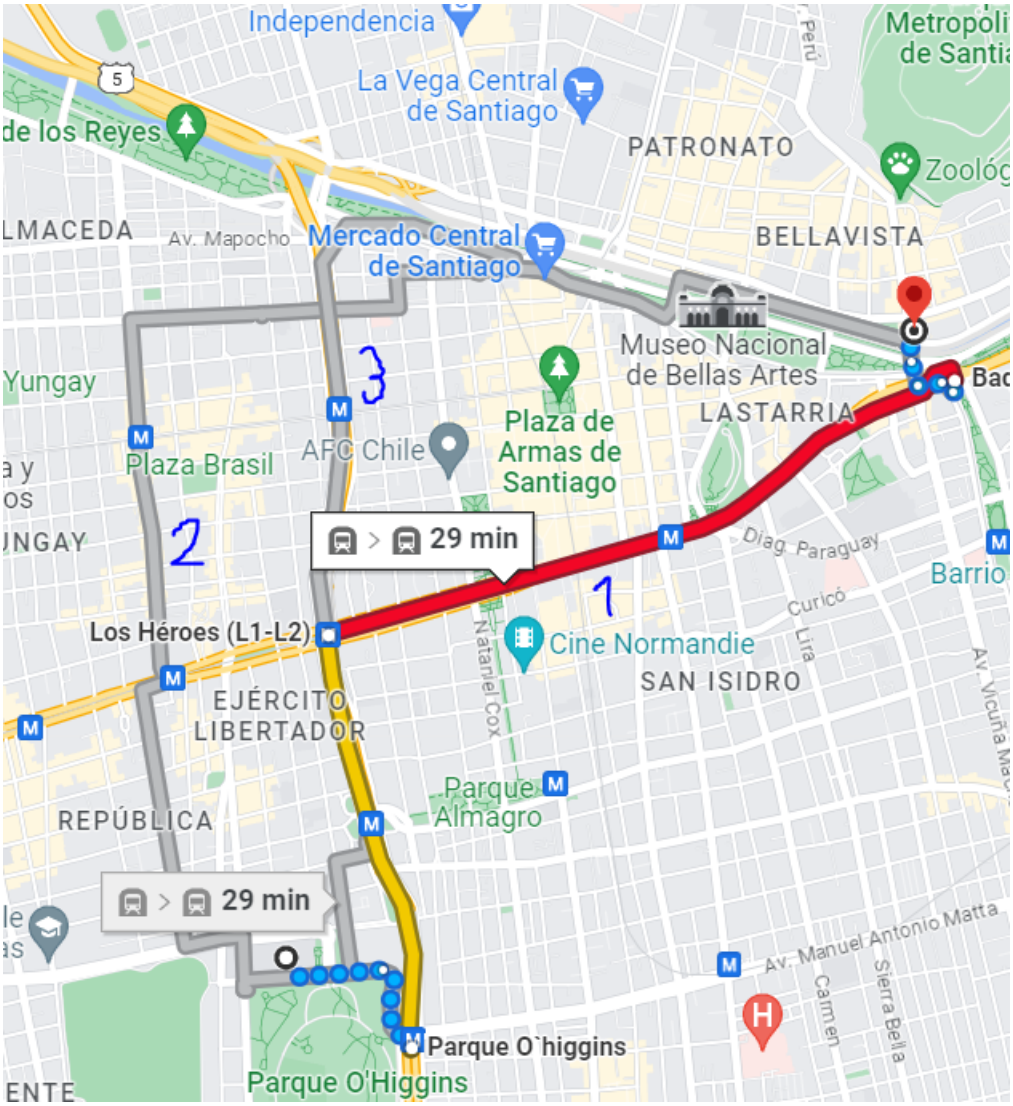

Figura 2.6: Posibles caminos para llegar desde FCFM hasta Derecho en transporte público. Fuente: Google Maps.

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* [26], 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 [23].

Para demostrar la utilidad del módulo Ayatori para programar esta clase de algoritmos, se decide crear una implementación **básica** 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** [28].
- La información del transporte público que se incluye en el modulo está almacenada en el formato GTFS, procesada mediante la librería **pygtfs** [6]. 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 [5].
- Para almacenar y trabajar con la información obtenida, se utiliza la librería **graph-tool** [7] 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 de forma tal que facilite su funcionamiento para el usuario final, a la hora de definir la entrada y la salida de los algoritmos de generación de rutas. En primer lugar, la librería **Nominatim** [17] permite que el usuario pueda ingresar como entrada una dirección en palabras, en vez de 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** [13] permite visualizar datos cartográficos en un mapa de fácil uso. **folium** está basado en la librería **Leaflet.js** de JavaScript, por lo que aprovecha todas sus características para generar un mapa interactivo. Estas dos librerías son utilizadas en la implementación del caso de prueba para ejemplificar 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. Inicialmente, acá le damos peso 1 a todas las aristas.

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 programar un algoritmo de generación de rutas de transporte público, utilizando únicamente la información provista por el módulo Ayatori, y obtener resultados útiles para realizar un estudio de movilidad.

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, ejemplificando la utilidad del módulo y evaluando el cumplimiento del criterio definido anteriormente. El caso de prueba definido consiste en programar una versión *lite* de Connection Scan Algorithm [11], 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.

Cabe destacar que la definición de CSA considera 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 *lite* de CSA, que cumple con calcular la ruta más conveniente considerando distancias y tiempos de espera estimados, siendo suficiente para demostrar la utilidad del módulo. El desarrollo de este Caso de Prueba está documentado en la sección 5.2 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 [20].

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 [29], tiene disponible para descarga la información de los distintos países del mundo, incluído Chile [30]. 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 [28], la librería utilizada para procesar la información, permite leer datos de OpenStreetMap en formato PBF e interpretarla en estructuras de GeoPandas [19], 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:

```
def download_osm_file(self, OSM_PATH):
    fp = pyrosm.get_data(
        "Santiago", # Nombre de la ciudad
        update=True,
        directory=OSM_PATH)
    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:

```
def create_osm_graph(self, OSM_PATH):
    fp = self.download_osm_file(OSM_PATH) # Descarga datos de OSM
    osm = pyrosm.OSM(fp)
    nodes, edges = osm.get_network(nodes=True) # Almacena nodos y aristas
   en variables
    graph = Graph() # Crea el grafo vacio

    # Propiedades
    lon_prop = graph.new_vertex_property("float")
    lat_prop = graph.new_vertex_property("float")
    node_id_prop = graph.new_vertex_property("long")
    graph_id_prop = graph.new_vertex_property("long")
    u_prop = graph.new_edge_property("long")
    v_prop = graph.new_edge_property("long")
    length_prop = graph.new_edge_property("double")
    weight_prop = graph.new_edge_property("double")
    (...)

    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 profundidad en el anexo 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:

```
def find_nearest_node(self, latitude, longitude):
    query_point = np.array([longitude, latitude])

    # Obtiene las propiedades
    lon_prop = self.graph.vertex_properties['lon']
    lat_prop = self.graph.vertex_properties['lat']

    # Calcula las distancias hasta el punto
    distances = np.linalg.norm(np.vstack((lon_prop.a, lat_prop.a)).T -
   query_point, axis=1)

    # Encuentra el indice del nodo mas cercano
    nearest_node_index = np.argmin(distances)
    nearest_node = self.graph.vertex(nearest_node_index)

    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** [17] para este fin, como fue mencionado en la sección 3.1. La definición de esta funcionalidad se puede apreciar en el código 4.4:

```
def address_locator(self, address):
    geolocator = Nominatim(user_agent="ayatori")
    while True: # Testeo del estado del servicio
        try:
            location = geolocator.geocode(address)
            break
        except GeocoderServiceError:
            i = 0
            if i < 15:
                print("Geocoding service error. Retrying in 5 seconds...")
                tm.sleep(5)
                i+=1
            else:
                msg = "Error: Too many retries. Geocoding service may be
   down. Please try again later."
                print(msg)
                return
    if location is not None: # Obtiene las coordenadas para hallar al nodo
     correspondiente
        lat, lon = location.latitude, location.longitude
        nearest = self.find_nearest_node(self.graph, lat, lon)
        return nearest
    msg = "Error: Address couldn't be found."
    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 [5]. 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 [6], 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 aceder 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:

```
def get_gtfs_data(self):
    sched = self.scheduler # Instancia del Scheduler
    for route in sched.routes:
        graph = Graph(directed=True) # Se crea un grafo por recorrido
        node_id_prop = graph.new_vertex_property("string")
        u_prop = graph.new_edge_property("object")
        v_prop = graph.new_edge_property("object")
        weight_prop = graph.new_edge_property("int")
        (...)
        # Se almacena la informacion en route_stops
        self.route_stops[route.route_id][stop_id] = {
            "route_id": route.route_id,
            "stop_id": stop_id,
            "coordinates": stop_coords[route.route_id][stop_id],
            "orientation": "round" if orientation == "I" else "return",
            "sequence": sequence,
            "arrival_times": []
        }
        (...)
        self.graphs[route.route_id] = graph # Se agrega el grafo al
    diccionario
    (...)
    for route_id, graph in self.graphs.items():
        weight_prop = graph.new_edge_property("int")
        for e in graph.edges():
            weight_prop[e] = 1
        graph.edge_properties["weight"] = weight_prop
        data_dir = "gtfs_routes" # Se declara el directorio para almacenar
    los grafos
        if not os.path.exists(data_dir):
            os.makedirs(data_dir)
        graph.save(f"{data_dir}/{route_id}.gt")
    (...)
    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:

```python
def get_near_stop_ids(self, coords, margin):
    stop_ids = []
    orientations = []
    for route_id, stops in self.route_stops.items():
        for stop_info in stops.values():
            stop_coords = stop_info["coordinates"]
            distance = self.haversine(coords[1], coords[0], stop_coords
    [1], stop_coords[0])
            if distance <= margin:
                orientation = stop_info["orientation"]
                stop_id = stop_info["stop_id"]
                if stop_id not in stop_ids:
                    stop_ids.append(stop_id)
                    orientations.append((stop_id, orientation))
    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:

```python
def haversine(self, lon1, lat1, lon2, lat2):
    R = 6372.8  # Radio de la Tierra en km
    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)
    a = sin(dLat / 2)**2 + cos(lat1) * cos(lat2) * sin(dLon / 2)**2
    c = 2 * asin(sqrt(a))
    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:

```python
def connection_finder(self, stop_id_1, stop_id_2):
    connected_routes = []
    for route_id, stops in self.route_stops.items():
        stop_ids = [stop_info["stop_id"] for stop_info in stops.values()]

        if stop_id_1 in stop_ids and stop_id_2 in stop_ids:
            connected_routes.append(route_id)
    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:

```
def get_arrival_times(self, route_id, stop_id, source_date):
    frequencies = pd.read_csv("stop_times.txt")
    route_frequencies = frequencies[frequencies["trip_id"].str.startswith(route_id)] # Obtiene las frecuencias de la ruta

    day_suffix = self.get_trip_day_suffix(source_date)

    stop_route_times = []
    bus_orientation = ""
    for _, row in route_frequencies.iterrows():
        start_time = pd.Timestamp(row["start_time"])
        if row["end_time"] == "24:00:00": # Normalizacion
            end_time = pd.Timestamp("23:59:59")
        else:
            end_time = pd.Timestamp(row["end_time"])
        headway_secs = row["headway_secs"]
        round_trip_id = f"{route_id}-I-{day_suffix}"
        return_trip_id = f"{route_id}-R-{day_suffix}"
        round_stop_times = pd.read_csv("stop_times.txt").query(f"trip_id.str.startswith('{round_trip_id}') and stop_id == '{stop_id}'")
        return_stop_times = pd.read_csv("stop_times.txt").query(f"trip_id.str.startswith('{return_trip_id}') and stop_id == '{stop_id}'")
        if len(round_stop_times) == 0 and len(return_stop_times) == 0:
            return
        elif len(round_stop_times) > 0:
            bus_orientation = "round"
            stop_time = pd.Timestamp(round_stop_times.iloc[0]["arrival_time"])
        elif len(return_stop_times) > 0:
            bus_orientation = "return"
            stop_time = pd.Timestamp(return_stop_times.iloc[0]["arrival_time"])
        for freq_time in pd.date_range(start_time, end_time, freq=f"{headway_secs}s"):
            freq_time_str = freq_time.strftime("%H:%M:%S")
            freq_time = datetime.strptime(freq_time_str, "%H:%M:%S")
            stop_route_time = datetime.combine(datetime.min, stop_time.time()) + timedelta(seconds=(freq_time - datetime.min).seconds)
            if stop_route_time not in stop_route_times:
                stop_route_times.append(stop_route_time)
            stop_time += pd.Timedelta(seconds=headway_secs)

    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:

```
def find_route_nodes(osm_graph, gtfs_data, route_id, desired_orientation):
    (...)
    stops = gtfs_data.route_stops.get(route_id, {}) # Obtiene las paradas del recorrido

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

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

    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:

```
def find_nearest_stops(osm_graph, gtfs_data, address, margin):
    graph = osm_graph.graph
    v = osm_graph.address_locator(graph, str(address))
    v_lon = graph.vertex_properties['lon'][v]
    v_lat = graph.vertex_properties['lat'][v]
    v_coords = (v_lon, v_lat)
    nearest_stops, orientations = gtfs_data.get_near_stop_ids(v_coords,
   margin)
    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 *lite* 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:

```
def algorithm_commands():
    now = datetime.now() # Hora y fecha actuales
    today = date.today() # Fecha actual
    today_format = today.strftime("%d/%m/%Y")
    moment = now.strftime("%H:%M:%S") # Formateo
    used_time = datetime.strptime(moment, "%H:%M:%S").time()

    source_date = input(
        "Enter the travel's date, in DD/MM/YYY format (press Enter to use
   today's date) : ") or today_format
    print(source_date)
    source_hour = input(
        "Enter the travel's start time, in HH:MM:SS format (press Enter to
    start now) : ") or used_time
    if source_hour != used_time:
        source_hour = datetime.strptime(source_hour, "%H:%M:%S").time()
```

```
    print(source_hour)

    source_example = "Beauchef 850, Santiago"
    while True:
        source_address = input(
            "Enter the starting point's address, in 'Street #No, Province'
    format (Ex: 'Beauchef 850, Santiago'):") or source_example
        if source_address.strip() != '':
            break

    destination_example = "Campus Antumapu Universidad de Chile, Santiago"
    while True:
        target_address = input(
            "Enter the ending point's address, in 'Street #No, Province'
    format (Ex: 'Campus Antumapu Universidad de Chile, Santiago'):")or
    destination_example
        if target_address.strip() != '':
            break

    # La ultima entrada del algoritmo fija el rango de distancia entre los
    puntos de origen/destino y las paradas cercanas a revisar
    best_route_map = connection_scan_lite(source_address, target_address,
    source_hour, source_date, 0.2)

    if not best_route_map:
        print("Something went wrong. Please try again later.")
        return

    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):

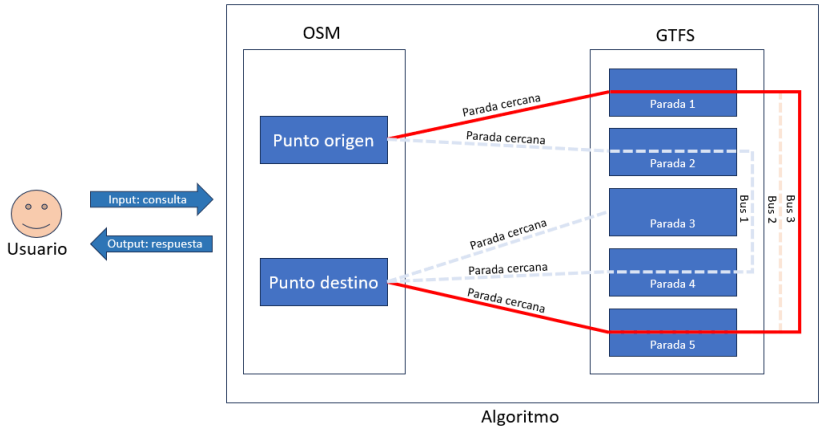

Figura 4.1: Diagrama de la interacción entre el usuario y un algoritmo de enrutamiento.

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 su extensión, no se incluye su código en esta sección, 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.

5.1. Ejemplos de uso

5.2. Caso de estudio

5.3. Evaluación de resultados

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

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, pero en la práctica, existe más de un inconveniente para que esto no ocurra tan así. El caso más directo de ver recae en los tiempos de espera de los buses. Estos tiempos son aproximados, pues responden al cronograma mandatorio que los buses deben de seguir, pero dejan de lado variables que pueden existir al momento de realizar los viajes: los retrasos, que pueden ocurrir por múltiples razones, como fallas humanas, embotellamientos, malfuncionamiento del bus, entre otras. Al no poseer información en tiempo real, los algoritmos programados sobre el módulo Ayatori funcionan bajo un supuesto de *continuidad operativa perfecta*, y al entregar una respuesta basada en este supuesto, pueden presentar un sesgo importante.

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 perfecta*, 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 [1] o Bicineta Chile [3]). 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. Esto aseguraría la disponibilidad operativa del módulo.
- 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

Bibliografía

[1] Andy Allan. OpenCycleMap - the OpenStreetMap Cycle Map. Disponible en `https://www.opencyclemap.org`. Revisado el 2023/09/14.

[2] Graham Asher. CartoType — Powerful Software — Beautiful Maps. Disponible en `https://www.cartotype.com`. Revisado el 2023/09/17.

[3] Bicineta Chile. Mapa de Ciclovías de la Región Metropolitana. Disponible en `https://www.bicineta.cl/ciclovias`. Revisado el 2023/03/07.

[4] Fundación OpenStreetMap Chile. Mapa de OpenStreetMap Chile. Disponible en `https://www.openstreetmap.cl`. Revisado el 2023/03/07.

[5] GTFS Community. General Transit Feed Specification. Disponible en `https://gtfs.org`. Revisado el 2023/06/28.

[6] Yaron de Leeuw. pygtfs. Repositorio disponible en `https://github.com/jarondl/pygtfs`. Revisado el 2023/06/29.

[7] Tiago de Paula Peixoto. graph-tool: Efficient network analysis with python. Documentación disponible en `https://graph-tool.skewed.de`. Revisado el 2023/07/17.

[8] Directorio de Transporte Público Metropolitano. GTFS Vigente. Disponible en `https://www.dtpm.cl/index.php/gtfs-vigente`. Revisado el 2023/07/22. Última versión: 2023/07/08.

[9] NetworkX developers. Networkx - network analysis in python. Documentación disponible en `https://networkx.org`. Revisado el 2023/07/22. Última versión: 2023/04/04.

[10] devemux86. Cruiser - Map and Navigation Platform. Disponible en `https://github.com/devemux86/cruiser`. Revisado el 2023/09/17.

[11] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithmics*, 23(1.7):1–56, 2018.

[12] Sozialhelden e.V. WheelMap - Busca lugares accesibles para sillas de ruedas. Disponible en `https://www.wheelmap.org`. Revisado el 2023/09/14.

[13] Filipe Fernandes. Folium. Repositorio disponible en `https://github.com/python-visualization/folium`. Revisado el 2023/07/17.

[14] OpenStreetMap (Global). Mapa de OpenStreetMap. Disponible en `https://www.openstreetmap.org`. Revisado el 2023/03/07.

[15] Google. Google Maps. Disponible en `https://www.google.com/maps/`.

[16] Eduardo Graells-Garrido. Aves: Análisis y Visualización, Educación y Soporte. Repositorio disponible en `https://github.com/zorzalerrante/aves`. Revisado el 2023/03/07.

[17] Sarah Hoffmann. Nominatim. Disponible en https://nominatim.org. Revisado el 2023/07/17.

[18] 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?mfn=3253 (2012). Revisado el 2023/03/07. Última versión lanzada el 2014.

[19] Kelsey Jordahl. Geopandas. Documentación disponible en https://geopandas.org. Revisado el 2023/03/07. Última versión: 2022/12/10.

[20] Felipe Leal. CC6909-Ayatori (Repositorio del Trabajo de Título). Repositorio disponible en https://github.com/Lysorek/CC6909-Ayatori. Revisado el 2023/03/07.

[21] Microsoft. Windows subsystem for linux. Documentación disponible en https://learn.microsoft.com/en-us/windows/wsl/. Revisado el 2023/07/17.

[22] Melchior Moos. ÖPNVKarte. Disponible en https://www.pnvkarte.de. Revisado el 2023/09/14.

[23] Linus Norton. Connection Scan Algorithm (implementación en TypeScript). Repositorio disponible en https://github.com/planarnetwork/connection-scan-algorithm. Revisado el 2023/06/29.

[24] Data Reportal. Digital 2021 Report for Chile. Disponible en https://datareportal.com/reports/digital-2021-chile (2021/02/11). Revisado el 2023/03/07.

[25] Audrey Roy and Cookiecutter community. Cookiecutter: Better project templates. Documentación disponible en https://cookiecutter.readthedocs.io/en/stable/. Revisado el 2023/03/07.

[26] Jonas Sauer. ULTRA: UnLimited TRAnsfers for Multimodal Route Planning. Repositorio disponible en https://github.com/kit-algo/ULTRA. Revisado el 2023/03/07.

[27] Victor Shcherb. OsmAnd - Offline Maps and Navigation. Disponible en https://github.com/osmandapp/OsmAnd. Revisado el 2023/09/17.

[28] Henrikki Tenkanen. Pyrosm: Read OpenStreetMap data from Protobuf files into GeoDataFrame with Python, faster. Repositorio disponible en https://github.com/HTenkanen/pyrosm. Revisado el 2023/03/07.

[29] Jochen Topf and Frederik Ramm. Geofabrik. Disponible en https://www.geofabrik.de. Revisado el 2023/03/07.

[30] Jochen Topf and Frederik Ramm. Geofabrik download server - chile. Disponible en https://download.geofabrik.de/south-america/chile.html. Revisado el 2023/03/07. Última versión: 2023/03/06.

[31] Papers with Code. Connection Scan Algorithm implementations. Disponible en https://cs.paperswithcode.com/paper/connection-scan-algorithm. Revisado el 2023/03/07.

ANEXOS

Apéndice A

Código de la solución

A.1. Clases del módulo

A.1.1. OSMGraph

```
import pyrosm
import numpy as np
import time as tm
from graph_tool.all import Graph
from geopy.exc import GeocoderServiceError
from geopy.geocoders import Nominatim

class OSMGraph:
    def __init__(self, OSM_PATH='.'):
        self.node_coords = {}
        self.graph = self.create_osm_graph(OSM_PATH)

    def download_osm_file(self, OSM_PATH):
        """
        Downloads the latest OSM file for Santiago.

        Parameters:
            OSM_PATH (str): The directory where the OSM file will be saved
    .

        Returns:
            str: The path to the downloaded OSM file.
        """
        fp = pyrosm.get_data(
            "Santiago",
```

```
            update=True,
            directory=OSM_PATH
        )

        return fp

    def create_osm_graph(self, OSM_PATH):
        """
        Creates a graph-tool's graph using the downloaded OSM data for Santiago.

        Returns:
            graph: osm data converted to a graph
        """
        # Download latest OSM data
        fp = self.download_osm_file(OSM_PATH)

        osm = pyrosm.OSM(fp)

        nodes, edges = osm.get_network(nodes=True)

        graph = Graph()

        # Create vertex properties for lon and lat
        lon_prop = graph.new_vertex_property("float")
        lat_prop = graph.new_vertex_property("float")

        # Create properties for the ids
        # Every OSM node has its unique id, different from the one given in the graph
        node_id_prop = graph.new_vertex_property("long")
        graph_id_prop = graph.new_vertex_property("long")

        # Create edge properties
        u_prop = graph.new_edge_property("long")
        v_prop = graph.new_edge_property("long")
        length_prop = graph.new_edge_property("double")
        weight_prop = graph.new_edge_property("double")

        vertex_map = {}

        print("GETTING OSM NODES...")
        for index, row in nodes.iterrows():
            lon = row['lon']
            lat = row['lat']
            node_id = row['id']
            graph_id = index
            self.node_coords[node_id] = (lat, lon)

            vertex = graph.add_vertex()
            vertex_map[node_id] = vertex

            # Assigning node properties
            lon_prop[vertex] = lon
            lat_prop[vertex] = lat
            node_id_prop[vertex] = node_id
```

```
            graph_id_prop[vertex] = graph_id

        # Assign the properties to the graph
        graph.vertex_properties["lon"] = lon_prop
        graph.vertex_properties["lat"] = lat_prop
        graph.vertex_properties["node_id"] = node_id_prop
        graph.vertex_properties["graph_id"] = graph_id_prop

        print("DONE")
        print("GETTING OSM EDGES...")

        for index, row in edges.iterrows():
            source_node = row['u']
            target_node = row['v']

            if row["length"] < 2 or source_node == "" or target_node == ""
:
                continue # Skip edges with empty or missing nodes

            if source_node not in vertex_map or target_node not in
vertex_map:
                print(f"Skipping edge with missing nodes: {source_node} ->
 {target_node}")
                continue  # Skip edges with missing nodes

            source_vertex = vertex_map[source_node]
            target_vertex = vertex_map[target_node]

            if not graph.vertex(source_vertex) or not graph.vertex(
target_vertex):
                print(f"Skipping edge with non-existent vertices: {
source_vertex} -> {target_vertex}")
                continue  # Skip edges with non-existent vertices

            # Calculate the distance between the nodes and use it as the
weight of the edge
            source_coords = self.node_coords[source_node]
            target_coords = self.node_coords[target_node]
            distance = np.linalg.norm(np.array(source_coords) - np.array(
target_coords))

            e = graph.add_edge(source_vertex, target_vertex)
            u_prop[e] = source_node
            v_prop[e] = target_node
            length_prop[e] = row["length"]
            weight_prop[e] = distance

        graph.edge_properties["u"] = u_prop
        graph.edge_properties["v"] = v_prop
        graph.edge_properties["length"] = length_prop
        graph.edge_properties["weight"] = weight_prop

        print("OSM DATA HAS BEEN SUCCESSFULLY RECEIVED")
        return graph

    def get_nodes_and_edges(self):
```

```
        """
        Returns a tuple containing two lists: one with the nodes and another with the edges.
        """
        nodes = list(self.graph.vertices())
        edges = list(self.graph.edges())
        return nodes, edges

    def print_graph(self):
        """
        Prints the vertices and edges of the graph.
        """
        print("Vertices:")
        for vertex in self.graph.vertices():
            print(f"Vertex ID: {int(vertex)}, lon: {self.graph.vertex_properties['lon'][vertex]}, lat: {self.graph.vertex_properties['lat'][vertex]}")

        print("\nEdges:")
        for edge in self.graph.edges():
            source = int(edge.source())
            target = int(edge.target())
            print(f"Edge: {source} -> {target}")

    def find_node_by_coordinates(self, lon, lat):
        """
        Finds a node in the graph based on its coordinates (lon, lat).

        Parameters:
            lon (float): the longitude of the node.
            lat (float): the latitude of the node.

        Returns:
            vertex: the vertex in the graph with the specified coordinates, or None if not found.
        """
        for vertex in self.graph.vertices():
            if self.graph.vertex_properties["lon"][vertex] == lon and self.graph.vertex_properties["lat"][vertex] == lat:
                return vertex
        return None

    def find_node_by_id(self, node_id):
        """
        Finds a node in the graph based on its id.

        Parameters:
            node_id (long): the id of the node.

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

```
                    i+=1
                else:
                    msg = "Error: Too many retries. Geocoding service may be down. Please try again later."
                    print(msg)
                    return
        if location is not None:
            long, lati = location.longitude, location.latitude
            nearest = self.find_nearest_node(self.graph, lati, long)
            return nearest
        msg = "Error: Address couldn't be found."
        print(msg)
```

A.1.2. GTFSData

```
import pygtfs
import os
import pandas as pd
from math import *
from datetime import datetime, date, time, timedelta
from graph_tool.all import Graph

class GTFSData:
    def __init__(self, GTFS_PATH='gtfs.zip'):
        self.scheduler = self.create_scheduler(GTFS_PATH)
        self.graphs = {}
        self.route_stops = {}
        self.special_dates = []
        self.graphs, self.route_stops, self.special_dates = self.get_gtfs_data()

    def create_scheduler(self, GTFS_PATH):
        """
        Creates the scheduler for the class, using the GTFS file, located in the given path directory.

        Parameters:
        GTFS_PATH (PATH): the path where the GTFS file is located.

        Returns:
        pygtfs.Schedule: the scheduler object
        """
        # Create a new schedule object using a GTFS file
        scheduler = pygtfs.Schedule(":memory:")
        pygtfs.append_feed(scheduler, GTFS_PATH)
        return scheduler

    def get_gtfs_data(self):
        """
        Reads the GTFS data from a file and creates a directed graph with its info, using the 'pygtfs' library. This gives
        the transit feed data of Santiago's public transport, including "Red Metropolitana de Movilidad" (previously known
```

```
        as Transantiago), "Metro de Santiago", "EFE Trenes de Chile", and "Buses de Acercamiento Aeropuerto".

        Returns:
            graphs: GTFS data converted to a dictionary of graphs, one per route.
            route_stops: Dictionary containing the stops for each route.
            special_dates: List of special calendar dates.
        """
        sched = self.scheduler

        # Get special calendar dates
        for cal_date in sched.service_exceptions: # Calendar_dates is renamed in pygtfs
            self.special_dates.append(cal_date.date.strftime("%d/%m/%Y"))

        stop_id_map = {} # To assign unique ids to every stop
        stop_coords = {}

        for route in sched.routes:
            graph = Graph(directed=True)
            stop_ids = set()
            trips = [trip for trip in sched.trips if trip.route_id == route.route_id]

            # Create a new vertex property for node_id
            node_id_prop = graph.new_vertex_property("string")

            # Create edge properties
            u_prop = graph.new_edge_property("object")
            v_prop = graph.new_edge_property("object")
            weight_prop = graph.new_edge_property("int")
            graph.edge_properties["weight"] = weight_prop
            graph.edge_properties["u"] = u_prop
            graph.edge_properties["v"] = v_prop

            added_edges = set() # To keep track of the edges that have already been added

            for trip in trips:
                stop_times = trip.stop_times
                orientation = trip.trip_id.split("-")[1]

                for i in range(len(stop_times)):
                    stop_id = stop_times[i].stop_id
                    sequence = stop_times[i].stop_sequence

                    if stop_id not in stop_id_map:
                        vertex = graph.add_vertex()
                        stop_id_map[stop_id] = vertex
                    else:
                        vertex = stop_id_map[stop_id]

                    stop_ids.add(vertex)

                    # Assign the node_id property to the vertex
```

```
                    node_id_prop[vertex] = stop_id

                    if i < len(stop_times) - 1:
                        next_stop_id = stop_times[i + 1].stop_id

                        if next_stop_id not in stop_id_map:
                            next_vertex = graph.add_vertex()
                            stop_id_map[next_stop_id] = next_vertex
                        else:
                            next_vertex = stop_id_map[next_stop_id]

                        edge = (vertex, next_vertex)
                        if edge not in added_edges: # Check if the edge has already been added
                            e = graph.add_edge(*edge)
                            graph.edge_properties["weight"][e] = 1
                            graph.edge_properties["u"][e] = node_id_prop[vertex]
                            graph.edge_properties["v"][e] = node_id_prop[next_vertex]
                            added_edges.add(edge) # Add the edge to the set of added edges

                        if route.route_id not in stop_coords:
                            stop_coords[route.route_id] = {}

                        if stop_id not in stop_coords[route.route_id]:
                            stop = sched.stops_by_id(stop_id)[0]
                            stop_coords[route.route_id][stop_id] = (stop.stop_lon, stop.stop_lat)

                            if route.route_id not in self.route_stops:
                                self.route_stops[route.route_id] = {}

                            self.route_stops[route.route_id][stop_id] = {
                                "route_id": route.route_id,
                                "stop_id": stop_id,
                                "coordinates": stop_coords[route.route_id][stop_id],
                                "orientation": "round" if orientation == "I" else "return",
                                "sequence": sequence,
                                "arrival_times": []
                            }

                    arrival_time = (datetime.min + stop_times[i].arrival_time).time()

                    if stop_id in self.route_stops[route.route_id]:
                        self.route_stops[route.route_id][stop_id]["arrival_times"].append(arrival_time)

            # Assign the node_id property to the graph
            graph.vertex_properties["node_id"] = node_id_prop

            self.graphs[route.route_id] = graph
```

```
133
134             stops_by_direction = {"round_trip": [], "return_trip": []}
135             for trip in trips:
136                 stop_times = trip.stop_times
137                 stops = [stop_times[i].stop_id for i in range(len(
    stop_times))]
138
139                 if trip.direction_id == 0:
140                     stops_by_direction["round_trip"].extend(stops)
141                 else:
142                     stops_by_direction["return_trip"].extend(stops)
143
144             round_trip_stops = set(stops_by_direction["round_trip"])
145             return_trip_stops = set(stops_by_direction["return_trip"])
146
147             for stop_id in round_trip_stops:
148                 if stop_id in stop_coords[route.route_id]:
149                     if stop_id in self.route_stops[route.route_id]:
150                         self.route_stops[route.route_id][stop_id]["
    orientation"] = "round"
151                     else:
152                         self.route_stops[route.route_id][stop_id] = {
153                             "route_id": route.route_id,
154                             "stop_id": stop_id,
155                             "coordinates": stop_coords[route.route_id][
    stop_id],
156                             "orientation": "round",
157                             "sequence": sequence,
158                             "arrival_times": []
159                         }
160
161             for stop_id in return_trip_stops:
162                 if stop_id in stop_coords[route.route_id]:
163                     if stop_id in self.route_stops[route.route_id]:
164                         self.route_stops[route.route_id][stop_id]["
    orientation"] = "return"
165                     else:
166                         self.route_stops[route.route_id][stop_id] = {
167                             "route_id": route.route_id,
168                             "stop_id": stop_id,
169                             "coordinates": stop_coords[route.route_id][
    stop_id],
170                             "orientation": "return",
171                             "sequence": sequence,
172                             "arrival_times": []
173                         }
174
175         for route_id, graph in self.graphs.items():
176             weight_prop = graph.new_edge_property("int")
177
178             for e in graph.edges():
179                 weight_prop[e] = 1
180
181             graph.edge_properties["weight"] = weight_prop
182             #graph.edge_properties["u"] = graph.new_edge_property("object
    ")
```

```
            #graph.edge_properties["v"] = graph.new_edge_property("object")

            data_dir = "gtfs_routes"
            if not os.path.exists(data_dir):
                os.makedirs(data_dir)

            graph.save(f"{data_dir}/{route_id}.gt")

        print("GTFS DATA RECEIVED SUCCESSFULLY")

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

    def get_route_graph(self, route_id):
        """
        Given a route_id, returns the vertices and edges for the corresponding graph.

        Parameters:
        route_id (str): The ID of the route.

        Returns:
        tuple: A tuple containing the vertices and edges of the graph. The vertices are a list of node IDs, and the edges are a list of tuples containing the source and target node IDs.
        """
        if route_id not in self.graphs:
            print(f"Route {route_id} does not exist.")
            return None

        graph = self.graphs[route_id]
        vertices = []
        for v in graph.vertices():
            node_id = graph.vertex_properties["node_id"][v]
            if node_id != '' and node_id is not None:
                vertices.append(node_id)

        edges = []
        for e in graph.edges():
            u = graph.edge_properties["u"][e]
            v = graph.edge_properties["v"][e]
            if u is not None and v is not None:
                edges.append((u, v))

        return vertices, edges

    def get_route_graph_vertices(self, route_id):
        """
        Given a route_id, returns the vertices for the corresponding graph.

        Parameters:
        route_id (str): The ID of the route.

        Returns:
        list: A list containing the vertices of the graph. The vertices
```

```
    are a list of node IDs.
234         """
235         if route_id not in self.graphs:
236             print(f"Route {route_id} does not exist.")
237             return None
238 
239         graph = self.graphs[route_id]
240         vertices = [graph.vertex_properties["node_id"][v] for v in graph.
    vertices()]
241 
242         return vertices
243 
244     def get_route_graph_edges(self, route_id):
245         """
246         Given a route_id, returns the edges for the corresponding graph.
247 
248         Parameters:
249         route_id (str): The ID of the route.
250 
251         Returns:
252         list: A list containing the edges of the graph.
253         """
254         if route_id not in self.graphs:
255             print(f"Route {route_id} does not exist.")
256             return None
257 
258         graph = self.graphs[route_id]
259         edges = [(graph.edge_properties["u"][e], graph.edge_properties["v"
    ][e]) for e in graph.edges()]
260 
261         return edges
262 
263     def get_route_coordinates(self, route_id):
264         round_trip_stops = []
265         return_trip_stops = []
266         for stop_info in self.route_stops[route_id].values():
267             if stop_info["orientation"] == "round":
268                 round_trip_stops.append(stop_info)
269             elif stop_info["orientation"] == "return":
270                 return_trip_stops.append(stop_info)
271 
272         round_trip_stops.sort(key=lambda stop: stop["sequence"])
273         return_trip_stops.sort(key=lambda stop: stop["sequence"])
274 
275         round_trip_coords = [(stop_info["coordinates"][1], stop_info["
    coordinates"][0]) for stop_info in round_trip_stops]
276         return_trip_coords = [(stop_info["coordinates"][1], stop_info["
    coordinates"][0]) for stop_info in return_trip_stops]
277 
278         return round_trip_coords, return_trip_coords
279 
280     def haversine(self, lon1, lat1, lon2, lat2):
281         """
282         Calculate the great circle distance between two points on the
    earth (specified in decimal degrees).
283 
```

```
        Parameters:
        lon1 (float): Longitude of the first point in decimal degrees.
        lat1 (float): Latitude of the first point in decimal degrees.
        lon2 (float): Longitude of the second point in decimal degrees.
        lat2 (float): Latitude of the second point in decimal degrees.

        Returns:
        float: The distance between the two points in kilometers.
        """
        R = 6372.8  # Earth radius in kilometers
        dLat = radians(lat2 - lat1)
        dLon = radians(lon2 - lon1)
        lat1 = radians(lat1)
        lat2 = radians(lat2)
        a = sin(dLat / 2)**2 + cos(lat1) * cos(lat2) * sin(dLon / 2)**2
        c = 2 * asin(sqrt(a))
        return R * c

    def get_stop_coords(self, stop_id):
        """
        Given a stop ID, returns the coordinates of the stop with the
    given ID.
        If the stop ID is not found, returns None.

        Parameters:
        stop_id (int): The ID of the stop to get the coordinates for.

        Returns:
        tuple: A tuple of two floats representing the longitude and
    latitude of the stop with the given ID.
        None: If the stop ID is not found.
        """
        for route_id, stops in self.route_stops.items():
            for stop_info in stops.values():
                if stop_info["stop_id"] == stop_id:
                    return stop_info["coordinates"]
        return None

    def get_near_stop_ids(self, coords, margin):
        """
        Given a tuple of coordinates and a margin, returns a list of stop
    IDs
        that are within the specified margin of the given coordinates,
    along with their orientations.

        Parameters:
        coords (tuple): A tuple of two floats representing the longitude
    and latitude of the coordinates to search around.
        margin (float): The maximum distance (in kilometers) from the
    given coordinates to include stops in the result.

        Returns:
        tuple: A tuple of two lists. The first list contains the stop IDs
    that are within the specified margin of the given coordinates.
        The second list contains tuples of stop IDs and their orientations
    .
```

```
        """
        stop_ids = []
        orientations = []
        for route_id, stops in self.route_stops.items():
            for stop_info in stops.values():
                stop_coords = stop_info["coordinates"]
                distance = self.haversine(coords[1], coords[0],
    stop_coords[1], stop_coords[0])
                if distance <= margin:
                    orientation = stop_info["orientation"]
                    stop_id = stop_info["stop_id"]
                    if stop_id not in stop_ids:
                        stop_ids.append(stop_id)
                        orientations.append((stop_id, orientation))
        return stop_ids, orientations

    def get_route_stop_ids(self, route_id):
        """
        Given a route ID, returns a list of stop IDs for the stops on the
    given route.

        Parameters:
        route_id (int): The ID of the route to get the stops for.

        Returns:
        list: A list of stop IDs for the stops on the given route.
        """
        stops = self.route_stops.get(route_id, {})
        return stops.keys()

    def route_stop_matcher(self, route_id, stop_id):
        """
        Given a route ID, and a stop ID, returns True if the stop ID is on
     the given route,
        and False otherwise.

        Parameters:
        route_id (int): The ID of the route to check.
        stop_id (int): The ID of the stop to check.

        Returns:
        bool: True if the stop ID is on the given route, False otherwise.
        """
        stop_list = self.get_route_stop_ids(route_id)
        return (stop_id in stop_list)

    def is_route_near_coordinates(self, route_id, coordinates, margin):
        """
        Given a route ID, a tuple of coordinates, and a margin, returns
    True if the route
        has a stop within the specified margin of the given coordinates,
    and False otherwise.

        Parameters:
        route_id (int): The ID of the route to check.
        coordinates (tuple): A tuple of two floats representing the
```

```
    longitude and latitude of the coordinates to search around.
383         margin (float): The maximum distance (in kilometers) from the
    given coordinates to include stops in the result.
384 
385         Returns:
386         bool: True if the route has a stop within the specified margin of
    the given coordinates, False otherwise.
387         """
388         for stop_info in self.route_stops[route_id].values():
389             stop_coords = stop_info["coordinates"]
390             distance = self.haversine(coordinates[1], coordinates[0],
    stop_coords[1], stop_coords[0])
391             if distance <= margin:
392                 return route_id
393         return False
394 
395     def get_bus_orientation(self, route_id, stop_id):
396         """
397         Checks and confirms the bus orientation, while visiting a stop, in
     the GTFS data files.
398 
399         Parameters:
400         route_id (str): The route or service's ID to check.
401         stop_id (str): The visited stop ID.
402 
403         Returns:
404         str or list: The bus orientation(s) associated with the route_id
    and stop_id. None if nothing is found.
405         """
406         stop_times = pd.read_csv("stop_times.txt")
407         filtered_stop_times = stop_times[(stop_times["trip_id"].str.
    startswith(route_id)) & (stop_times["stop_id"] == stop_id)]
408 
409         orientations = []
410         for trip_id in filtered_stop_times["trip_id"]:
411             orientation = trip_id.split("-")[1]
412             if orientation == "I" and "round" not in orientations:
413                 orientations.append("round")
414             elif orientation == "R" and "return" not in orientations:
415                 orientations.append("return")
416 
417         if len(orientations) == 0:
418             return None
419         elif len(set(orientations)) == 1:
420             return orientations[0]
421         else:
422             return orientations
423 
424     def connection_finder(self, stop_id_1, stop_id_2):
425         """
426         Finds all routes that have stops at both given stop IDs.
427 
428         Parameters:
429         stop_id_1 (str): The ID of the first stop to check.
430         stop_id_2 (str): The ID of the second stop to check.
431 
```

```
        Returns:
        list: A list of route IDs that have stops at both given stop IDs.
        """
        connected_routes = []
        for route_id, stops in self.route_stops.items():
            stop_ids = [stop_info["stop_id"] for stop_info in stops.values()]

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

    def get_routes_at_stop(self, stop_id):
        """
        Finds all routes that have a stop at the given stop ID.

        Parameters:
        stop_id (str): The ID of the stop to check.

        Returns:
        list: A list of route IDs that have a stop at the given stop ID.
        """
        routes = [route_id for route_id in self.route_stops.keys() if stop_id in self.get_route_stop_ids(route_id) and self.connection_finder(stop_id, stop_id)]
        return routes

    def is_24_hour_service(self, route_id):
        """
        Determines if the given route has a 24-hour service.

        Parameters:
        route_id (str): A string representing the ID of the route.

        Returns:
        bool: True if the route has a 24-hour service, False otherwise.
        """
        # Read the frequencies for the route
        frequencies = pd.read_csv("frequencies.txt")
        route_str = str(route_id) + "-"
        route_frequencies = frequencies[frequencies["trip_id"].str.startswith(route_str)]

        # Check if any frequency has a start time of "00:00:00" and an end time of "24:00:00"
        has_start_time = False
        has_end_time = False
        for _, row in route_frequencies.iterrows():
            start_time = row["start_time"]
            end_time = row["end_time"]
            if start_time == "00:00:00":
                has_start_time = True
            if end_time == "24:00:00":
                has_end_time = True

        return has_start_time and has_end_time
```

```
483
484     def check_night_routes(self, valid_services, is_nighttime):
485         """
486         Filters the given list of route IDs to only include night routes
    if is_nighttime is True.
487
488         Parameters:
489         valid_services (list): A list of route IDs to filter.
490         is_nighttime (bool): True if it is nighttime, False otherwise.
491
492         Returns:
493         list: A list of route IDs that are night routes if is_nighttime is
     True, or all route IDs otherwise.
494         """
495         if is_nighttime:
496             #nighttime_routes = [route_id for route_id in valid_services
    if route_id.endswith("N")]
497             nighttime_routes = [route_id for route_id in valid_services if
     route_id.endswith("N") or self.is_24_hour_service(route_id)]
498             if nighttime_routes:
499                 return nighttime_routes
500             else:
501                 return None
502         else:
503             daytime_routes = [route_id for route_id in valid_services if
    not route_id.endswith("N")]
504             if daytime_routes:
505                 return daytime_routes
506             else:
507                 return None
508
509     def is_nighttime(self, source_hour):
510         """
511         Determines if the given hour is during the nighttime.
512
513         Parameters:
514         source_hour (datetime.time): The hour to check.
515
516         Returns:
517         bool: True if the hour is during the nighttime, False otherwise.
518         """
519         start_time = time(0, 0, 0)
520         end_time = time(5, 30, 0)
521         if start_time <= source_hour <= end_time:
522             return True
523         else:
524             return False
525
526     def is_holiday(self, date_string):
527         """
528         Checks if a given date is a holiday.
529
530         Parameters:
531         date_string (str): A string representing the date in the format "
    dd/mm/yyyy".
532
```

```
        Returns:
        bool: True if the date is a holiday, False otherwise.
        """
        # Local holidays
        if date_string in self.special_dates:
            return True
        date_obj = datetime.strptime(date_string, "%d/%m/%Y")

        # Weekend days
        day_of_week = date_obj.weekday()
        if day_of_week == 5 or day_of_week == 6:
            return True
        return False

    def is_rush_hour(self, source_hour):
        """
        Determines if the given hour is during rush hour.

        Parameters:
        source_hour (datetime.time): The hour to check.

        Returns:
        bool: True if the hour is during rush hour, False otherwise.
        """
        am_start_time = time(5, 30, 0)
        am_end_time = time(9, 0, 0)
        pm_start_time = time(17, 30, 0)
        pm_end_time = time(21, 0, 0)
        if am_start_time <= source_hour <= am_end_time or pm_start_time <=
    source_hour <= pm_end_time:
            return True
        else:
            return False

    def check_express_routes(self, valid_services, is_rush_hour):
        """
        Filters the given list of route IDs to only include express routes
    if is_rush_hour is True.

        Parameters:
        valid_services (list): A list of route IDs to filter.
        is_rush_hour (bool): True if it is rush hour, False otherwise.

        Returns:
        list: A list of route IDs that are express routes if is_rush_hour
    is True, or all route IDs otherwise.
        """
        if is_rush_hour:
            return valid_services
        else:
            regular_hour_routes = [route_id for route_id in valid_services
    if not route_id.endswith("e")]
            return regular_hour_routes

    def get_trip_day_suffix(self, date):
        """
```

```
        Based on the given date, gets the corresponding trip day suffix for the trip IDs.

        Parameters:
        date (date): The date to be checked.

        Returns
        str: A string with the trip day suffix.
        """
        date_object = datetime.strptime(date, "%d/%m/%Y")
        day_of_week = date_object.weekday()

        if day_of_week < 5:
            trip_day_suffix = "L"
        elif day_of_week == 5:
            trip_day_suffix = "S"
        else:
            trip_day_suffix = "D"

        return trip_day_suffix

    def get_arrival_times(self, route_id, stop_id, source_date):
        """
        Returns the arrival times for a given route and stop.

        Parameters:
        route_id (str): A string representing the ID of the route.
        stop_id (str): A string representing the ID of the stop.

        Returns:
        tuple: A tuple containing a string representing the bus orientation ("round" or "return") and a list of datetime objects representing the arrival times.
        """
        # Read the frequencies.txt file
        frequencies = pd.read_csv("stop_times.txt")

        # Filter the frequencies for the given route ID
        route_frequencies = frequencies[frequencies["trip_id"].str.startswith(route_id)]

        # Get the day suffix
        day_suffix = self.get_trip_day_suffix(source_date)

        # Get the arrival times for the stop for each trip
        stop_route_times = []
        bus_orientation = ""
        for _, row in route_frequencies.iterrows():
            start_time = pd.Timestamp(row["start_time"])
            if row["end_time"] == "24:00:00":
                end_time = pd.Timestamp("23:59:59")
            else:
                end_time = pd.Timestamp(row["end_time"])
            headway_secs = row["headway_secs"]
            round_trip_id = f"{route_id}-I-{day_suffix}"
            return_trip_id = f"{route_id}-R-{day_suffix}"
```

```
637             round_stop_times = pd.read_csv("stop_times.txt").query(f"
    trip_id.str.startswith('{round_trip_id}') and stop_id == '{stop_id}'")
638             return_stop_times = pd.read_csv("stop_times.txt").query(f"
    trip_id.str.startswith('{return_trip_id}') and stop_id == '{stop_id}'")
639             if len(round_stop_times) == 0 and len(return_stop_times) == 0:
640                 return
641             elif len(round_stop_times) > 0:
642                 bus_orientation = "round"
643                 stop_time = pd.Timestamp(round_stop_times.iloc[0]["
    arrival_time"])
644             elif len(return_stop_times) > 0:
645                 bus_orientation = "return"
646                 stop_time = pd.Timestamp(return_stop_times.iloc[0]["
    arrival_time"])
647             for freq_time in pd.date_range(start_time, end_time, freq=f"{
    headway_secs}s"):
648                 freq_time_str = freq_time.strftime("%H:%M:%S")
649                 freq_time = datetime.strptime(freq_time_str, "%H:%M:%S")
650                 stop_route_time = datetime.combine(datetime.min, stop_time
    .time()) + timedelta(seconds=(freq_time - datetime.min).seconds)
651                 if stop_route_time not in stop_route_times:
652                     stop_route_times.append(stop_route_time)
653                 stop_time += pd.Timedelta(seconds=headway_secs)
654 
655         return bus_orientation, stop_route_times
656 
657 
658     def get_time_until_next_bus(self, arrival_times, source_hour,
    source_date):
659         """
660         Returns the time until the next three buses.
661 
662         Parameters:
663         arrival_times (list): A list of datetime objects representing the
    arrival times of the buses.
664         source_hour (datetime.time): The source hour to compare with the
    arrival times.
665         source_date (datetime.date): The source date to check if there are
     buses remaining.
666 
667         Returns:
668         list: A list of tuples representing the time until the next three
    buses in minutes and seconds.
669         """
670         arrival_times_remaining = []
671         for a_time in arrival_times:
672             if a_time.time() >= source_hour:
673                 arrival_times_remaining.append(a_time)
674         #arrival_times_remaining = [time for time in arrival_times if time
    .time() >= source_hour]
675         if len(arrival_times_remaining) == 0:
676             return None
677         else:
678             # Sort the remaining arrival times in ascending order
679             arrival_times_remaining.sort()
680 
```

```
            # Get the datetime objects for the next three buses
            next_buses = []
            for i in range(min(3, len(arrival_times_remaining))):
                next_arrival_time = arrival_times_remaining[i]
                next_bus = datetime.combine(next_arrival_time.date(), next_arrival_time.time())
                next_buses.append(next_bus)

            if next_buses is None:
                print("No buses remaining for the specified date.")
            else:
                # Calculate the time until the next three buses
                time_until_next_buses = []
                for next_bus in next_buses:
                    time_until_next_bus = (next_bus - datetime.combine(next_bus.date(), source_hour)).total_seconds()
                    minutes, seconds = divmod(time_until_next_bus, 60)
                    time_until_next_buses.append((int(minutes), int(seconds)))

                return time_until_next_buses

    def timedelta_to_hhmm(self, td):
        """
        Converts a timedelta object to a string in HHMM format.

        Parameters:
        td (timedelta): The timedelta object to be converted.

        Returns:
        str: A formated string with the time.
        """
        total_seconds = int(td.total_seconds())
        hours = total_seconds // 3600
        minutes = (total_seconds % 3600) // 60
        return f"{hours:02d}:{minutes:02d}"

    def timedelta_separator(self, td):
        """
        Separates a timedelta object into minutes and seconds.

        Parameters:
        td (timedelta): A timedelta object representing a duration of time.

        Returns:
        tuple: A tuple containing the number of minutes and seconds in the timedelta object. The minutes and seconds are both integers.
        """
        total_seconds = td.total_seconds()
        minutes = int(total_seconds // 60)
        seconds = int(total_seconds % 60)
        return minutes, seconds

    def get_travel_time(self, trip_id, stop_ids):
        """
```

```
        Returns the travel time between two stops for a given trip.

        Parameters:
        trip_id (str): A string representing the ID of the trip.
        stop_ids (list): A list of two strings representing the IDs of the
     stops.

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

    def get_trip_sequence(self, route_id, stop_id):
        """
        Given a dictionary of routes and stops, a route ID and a stop ID,
    gets the trip sequence number corresponding to the stop.

        Parameters:
        route_id (str): The route or service's ID.
        stop_id (str): The stop's ID.

        Returns:
        str: A string representing the sequence number.
        """
        seq = self.route_stops[route_id][stop_id]["sequence"]
        return seq

    def walking_travel_time(self, stop_coords, location_coords, speed):
        """
        Calculates the walking travel time between a location and a stop,
    given a speed value.

        Parameters:
        stop_coords (tuple): A tuple with the stop's coordinates.
        location_coords (tuple):  A tuple with the location's coordinates.
        speed (float): The walking speed value.

        Returns.
        float: The time (in seconds) that represents the travel time.
        """
        distance = self.haversine(stop_coords[0], stop_coords[1],
    location_coords[0], location_coords[1])
        time = round((distance / speed) * 3600,2)
        return time

    def parse_metro_stations(self, stops_file):
        """
        Parses the Metro Stations data, creating a dictionary with their
    names.
```

```
781
782         Parameters:
783         stops_file (File): The GTFS file with the stop data (stops.txt).
784
785         Returns:
786         dict: A dictionary with the names of the stations.
787         """
788         subway_stops = {}
789         with open(stops_file, 'r') as f:
790             for line in f:
791                 stop_id, _, stop_name, _, _, _, _ = line.strip().split(',')
    )
792                 if stop_id.isdigit():
793                     subway_stops[stop_id] = stop_name
794         return subway_stops
795
796     def is_metro_station(self, stop_id, route_dict):
797         """
798         Checks if a stop is a Metro station.
799
800         Parameters:
801         stop_id (str): The stop's ID to be checked.
802         route_dict (dict): The dictionary with the Metro stations names.
803
804         Returns:
805         str or None: A string with the stop ID if the stop is a Metro
    station, or None if it isn't.
806         """
807         try:
808             route_num = int(stop_id)
809             return route_dict[stop_id]
810         except ValueError:
811             return None
```

A.2. Algoritmo de ejemplo: Connection Scan Algorithm

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

```
    departure_time (time): the time at which the travel should start.
    departure_date (date): the date on which the travel should be done.
    margin (float): margin of distance between the nodes and the valid stops.

    Returns:
    folium.Map: the map of the best travel route. It returns None if no routes are found.
    """
    # Getting the nodes corresponding to the addresses
    graph = osm_graph.graph
    source_node = osm_graph.address_locator(graph, source_address)
    target_node = osm_graph.address_locator(graph, target_address)

    # Instance of the route_stops dictionary
    route_stops = gtfs_data.route_stops

    if source_node is not None and target_node is not None:
        # Convert source and target node IDs to integers
        source_node_graph_id = graph.vertex_properties["graph_id"][source_node]
        target_node_graph_id = graph.vertex_properties["graph_id"][target_node]

        print("Both addresses have been found.")
        print("Processing...")

        geolocator = Nominatim(user_agent="ayatori")

        route_info = available_route_finder(graph, gtfs_data, source_node_graph_id, target_node_graph_id, departure_time, departure_date, margin, geolocator)

        selected_path = route_info[0]
        source = route_info[1]
        target = route_info[2]
        valid_source_stops = route_info[3]
        valid_target_stops = route_info[4]
        valid_services = route_info[5]
        fixed_orientation = route_info[6]
        near_source_stops = route_info[7]
        near_target_stops = route_info[8]

        # Create a map that shows the correct public transport services to take from the source to the target
        m = folium.Map(location=[selected_path[0][0], selected_path[0][1]], zoom_start=13)

        # Add markers for the source and target points
        folium.Marker(location=[selected_path[0][0], selected_path[0][1]], popup="Origen: {}".format(source), icon=folium.Icon(color='green')).add_to(m)
        folium.Marker(location=[selected_path[-1][0], selected_path[-1][1]], popup="Destino: {}".format(target), icon=folium.Icon(color='red')).add_to(m)

```

```
56          print("")
57          print("Routes have been found.")
58          print("Calculating the best route and getting the arrival times
    for the next buses...")
59
60          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)
61
62          best_option = best_option_info[0]
63          initial_delta_time = best_option_info[1]
64          best_option_times = best_option_info[2]
65          initial_source_time = best_option_info[3]
66          valid_target = best_option_info[4]
67          best_option_orientation = best_option_info[5]
68
69          if best_option is None:
70              print("Error: There are no available services right now to go
    to the desired destination.")
71              print("Possible reasons: the valid routes are not available at
     the specified date or starting time.")
72              print("Please take into account that some routes have trips
    only during or after nighttime, which goes between 00:00:00 and
    05:30:00")
73              return
74
75          arrival_time = None
76
77          source_stop = best_option[1]
78
79          # Parse Metro stations's names
80          metro_stations_dict = gtfs_data.parse_metro_stations("stops.txt")
81          possible_metro_name = gtfs_data.is_metro_station(best_option[1],
    metro_stations_dict)
82          if possible_metro_name is not None:
83              source_stop = possible_metro_name
84
85          walking_minutes, walking_seconds = gtfs_data.timedelta_separator(
    initial_delta_time)
86
87          print("")
88          print("To go from: {}".format(source))
89          print("To: {}".format(target))
90          best_arrival_time_str = gtfs_data.timedelta_to_hhmm(best_option
    [2])
91          print("")
92          if possible_metro_name is not None: # Changes the printing to
    adapt for the use of Metro
93              print("The best option is to walk for {} minutes and {}
    seconds to {} Metro station, and take the line {}.".format(
    walking_minutes, walking_seconds, source_stop, best_option[0]))
94              print("The next train arrives at {}.".format(
    best_arrival_time_str))
95              print("The other two next trains arrives in:")
96          else:
97              print("The best option is to walk for {} minutes and {}
```

```
    seconds to stop {}, and take the route {}.".format(walking_minutes,
    walking_seconds, source_stop, best_option[0]))
98          print("The next bus arrives at {}.".format(
    best_arrival_time_str))
99          print("The other two next buses arrives in:")
100
101         # Format and prints the times
102         for i in range(len(best_option_times)):
103             if i == 0:
104                 continue
105             minutes, seconds = best_option_times[i]
106             waiting_time = timedelta(minutes=minutes, seconds=seconds)
107             arrival_time = initial_source_time + waiting_time
108             time_string = gtfs_data.timedelta_to_hhmm(arrival_time)
109             print(f"{minutes} minutes, {seconds} seconds ({time_string})")
110
111         # Base Coordinates
112         source_lat = selected_path[0][0]
113         source_lon = selected_path[0][1]
114         target_lat = selected_path[-1][0]
115         target_lon = selected_path[-1][1]
116
117
118         for stop_id in near_source_stops:
119             if stop_id in valid_source_stops:
120                 # Filters the data for selecting the best source option
    for its mapping
121                 stop_coords = gtfs_data.get_stop_coords(route_stops, str(
    stop_id))
122                 routes_at_stop = gtfs_data.get_routes_at_stop(route_stops,
     stop_id)
123                 valid_stop_services = [stop_id for stop_id in
    valid_services if stop_id in routes_at_stop]
124
125                 for service in valid_stop_services:
126                     if service == best_option[0] and stop_id ==
    best_option[1]:
127                         # Maps the best option to take the best option's
    service
128                         folium.Marker(location=[stop_coords[1],
    stop_coords[0]],
129                                       popup="Mejor opcion: subirse al recorrido {}
     en la parada {}.".format(best_option[0], best_option[1]),
130                                       icon=folium.Icon(color='cadetblue', icon='
    plus')).add_to(m)
131                         initial_distance = [(selected_path[0][0],
    selected_path[0][1]),(stop_coords[1], stop_coords[0])]
132                         folium.PolyLine(initial_distance,color='black',
    dash_array='10').add_to(m)
133
134         for stop_id in near_target_stops:
135             if stop_id in valid_target_stops:
136                 # Filters the data for the possible target stops
137                 stop_coords = gtfs_data.get_stop_coords(route_stops, str(
    stop_id))
138                 routes_at_stop = gtfs_data.get_routes_at_stop(route_stops,
```

```
 stop_id)
139                 valid_stop_services = [stop_id for stop_id in
    valid_services if stop_id in routes_at_stop]
140
141         target_orientation = None
142         for service in valid_target:
143             if service == best_option[0]:
144                 # Generates the trip id to get the approximated travel
    time
145                 if fixed_orientation == "round":
146                     trip_id = service + "-I-" + gtfs_data.
    get_trip_day_suffix(departure_date)
147                 else:
148                     trip_id = service + "-R-" + gtfs_data.
    get_trip_day_suffix(departure_date)
149
150                 best_travel_time = None
151                 selected_stop = None
152                 for stop_id in valid_target_stops:
153                     # Calculates the travel time while taking the service
154                     bus_time = gtfs_data.get_travel_time(trip_id, [
    best_option[1], stop_id])
155                     target_stop_routes = gtfs_data.get_routes_at_stop(
    route_stops, stop_id)
156                     target_orientation = gtfs_data.get_bus_orientation(
    best_option[0], stop_id)
157                     if service in target_stop_routes and bus_time >
    timedelta() and (best_travel_time is None or bus_time <
    best_travel_time):
158                         # Checking the correct orientation
159                         if fixed_orientation in target_orientation:
160                             # Updates the selected target stop and travel
    time
161                             best_travel_time = bus_time
162                             selected_stop = stop_id
163
164                 # Gets the coordinates for the target stop
165                 selected_stop_coords = gtfs_data.get_stop_coords(
    route_stops, selected_stop)
166                 # Separates the best travel time for the printing
167                 minutes, seconds = gtfs_data.timedelta_separator(
    best_travel_time)
168
169                 # Gets the sequence number for the source and target stops
170                 seq_1 = route_stops[best_option[0]][best_option[1]]["
    sequence"]
171                 seq_2 = route_stops[best_option[0]][selected_stop]["
    sequence"]
172
173                 # Store the coordinates of the visited stops for their
    mapping
174                 visited_stops = []
175
176                 # Iterate over the stops of the selected route
177                 for stop_id, stop_info in route_stops[best_option[0]].
    items():
```

```
                    # Check if the stop sequence number is between seq_1 and seq_2
                    seq_number = stop_info["sequence"]
                    this_orientation = gtfs_data.get_bus_orientation(best_option[0], stop_id)
                    if best_option_orientation in this_orientation and seq_1 <= seq_number <= seq_2:
                        # Append the coordinates of the stop to the visited_stops list
                        lat = stop_info["coordinates"][0]
                        lon = stop_info["coordinates"][1]
                        visited_stops.append((seq_number, (lon, lat)))

                # Sorts the visited stops and gets their coordinates
                visited_stops_sorted = sorted(visited_stops, key=lambda x: x[0])
                visited_stops_sorted_coords = [x[1] for x in visited_stops_sorted]

                # Checks if the stop is a Metro Station (they are stored as a number)
                possible_metro_target_name = gtfs_data.is_metro_station(selected_stop, metro_stations_dict)

                if possible_metro_target_name is not None:
                    selected_stop = possible_metro_target_name

                print("")
                if possible_metro_name is not None: # Changes the message
                    print("You will get off the train on {} station after {} minutes and {} seconds.".format(selected_stop, minutes, seconds))
                else:
                    print("You will get off the bus on stop {} after {} minutes and {} seconds.".format(selected_stop, minutes, seconds))

                # Maps the best option to get off the best option's service
                folium.Marker(location=[selected_stop_coords[1], selected_stop_coords[0]],
                              popup="Mejor opcion: bajarse del recorrido {} en la parada {}.".format(best_option[0], selected_stop),
                              icon=folium.Icon(color='cadetblue', icon='plus')).add_to(m)
                ending_distance = [(selected_path[-1][0], selected_path[-1][1]),(selected_stop_coords[1], selected_stop_coords[0])]
                folium.PolyLine(ending_distance,color='black',dash_array='10').add_to(m)

                # Create a polyline connecting the visited stops
                folium.PolyLine(visited_stops_sorted_coords, color='red').add_to(m)

                # Gets the coordinates for the target stop and target location
                final_stop_coords = (selected_stop_coords[1], selected_stop_coords[0])
```

```
                final_location_coords = (target_lat, target_lon)

                # Calculates the walking time between the target stop and location
                end_walking_time = gtfs_data.walking_travel_time(final_stop_coords, final_location_coords, 5)
                end_delta_time = timedelta(seconds=end_walking_time)
                end_walk_min, end_walk_sec = gtfs_data.timedelta_separator(end_delta_time)

                # Time walking to stop + waiting the bus + riding the bus + walking to target destination
                total_time = initial_delta_time + best_option[3] + best_travel_time + end_delta_time
                minutes, seconds = gtfs_data.timedelta_separator(total_time)

                # Parses the time for the printing
                destination_time = initial_source_time + total_time
                time_string = gtfs_data.timedelta_to_hhmm(destination_time)
                print(f"After that, you need to walk for {end_walk_min} minutes and {end_walk_sec} seconds to arrive at the target spot.")
                print(f"Total travel time: {minutes} minutes, {seconds} seconds. You will arrive your destination at {time_string}.")

        # Set the optimal zoom level for the map
        fit_bounds(selected_path, m)

        return m
    else:
        # Empty return
        return
```