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 muestra partes de implementaciones existentes del módulo.

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 [14]. 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) [3] 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 [8] (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 [2], 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 [10]. 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, existe la Fundación OpenStreetMap Chile [2] cumpliendo ese papel. Se presenta la figura 2.1 a modo de ejemplo, donde se observa el mapa del Gran Santiago visualizado en su página web, en el que se pueden notar, entre otras vías, las carreteras más destacadas de la ciudad, como la Circunvalación Américo Vespucio y la Autopista Central.

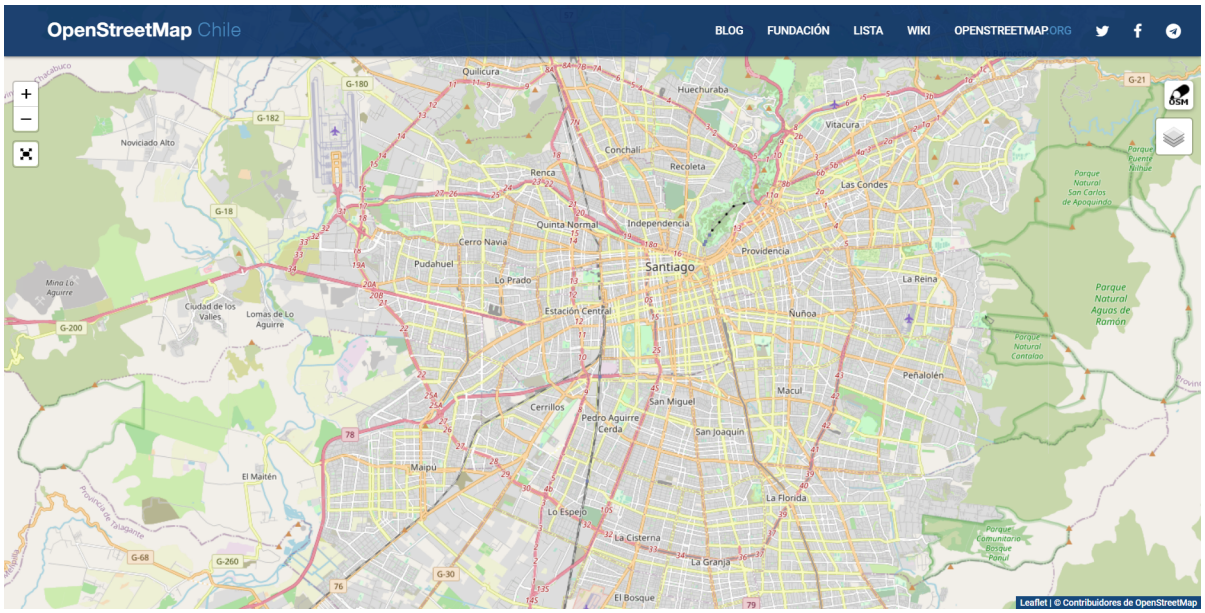

Figura 2.1: Mapa de Santiago en OpenStreetMap. Fuente: openstreetmap.cl .

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 Chile. Además de analizar la información mediante esta página, el portal global de OpenStreetMap cuenta con un buscador que permite hallar direcciones de todo el mundo y visualizar un mapa de la zona [10]. Sin embargo, aparte de utilizar la información mediante visualizaciones web, los datos de OSM también se pueden descargar en distintos formatos para su uso. Esto permite una mayor versatilidad a la hora de crear herramientas que hagan uso de esta información, al poder obtener los datos en el formato más conveniente para trabajar con ellos.

Para integrar estos datos 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** [22] 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.

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 2.2, se muestra el cómo se ven los archivos en GTFS, mostrando, como ejemplo, la información de los paraderos disponibles.

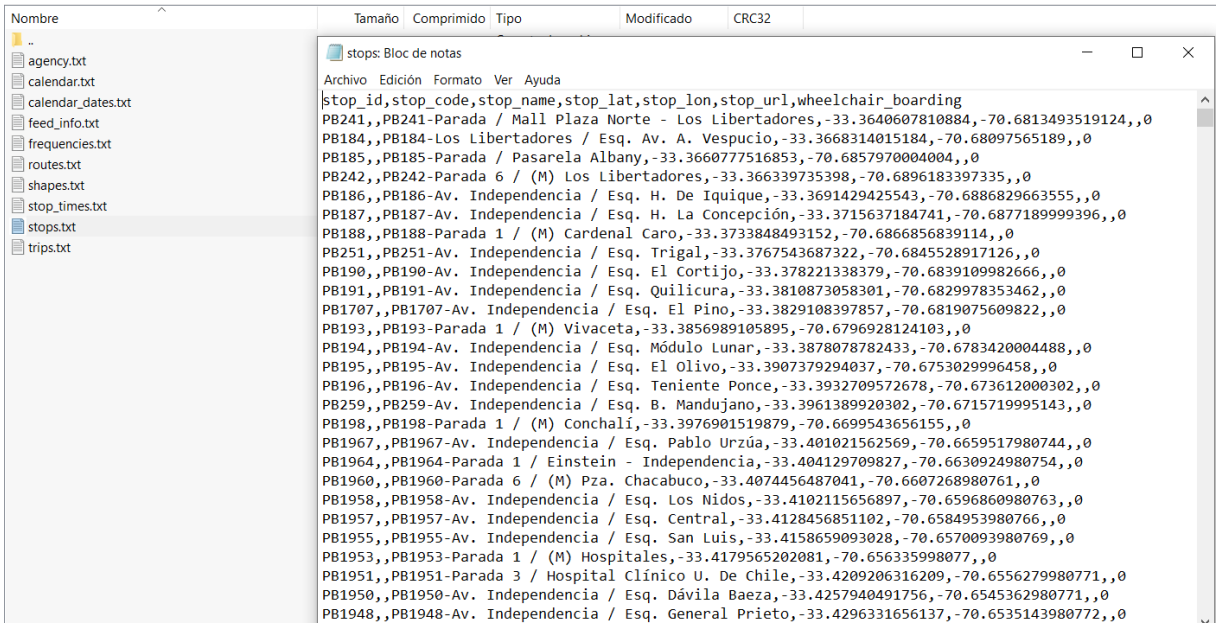

Figura 2.2: Visualización de archivos del feed GTFS para Santiago.

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 [6]. 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 2.3.

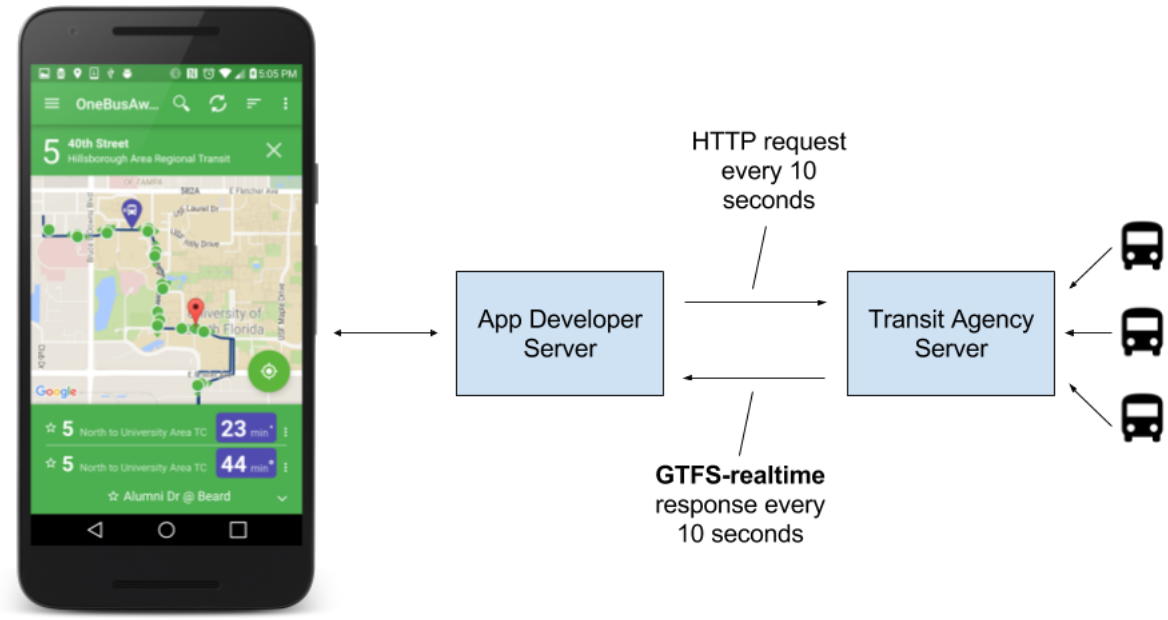

Figura 2.3: Diagrama de uso de datos en tiempo real en formato GTFS para una aplicación. Fuente: watrifeed.ml .

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** [4] 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.

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 [7]. 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 [5]. 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) [17].

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 [8]. 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 2.4, se grafica el funcionamiento antes descrito:

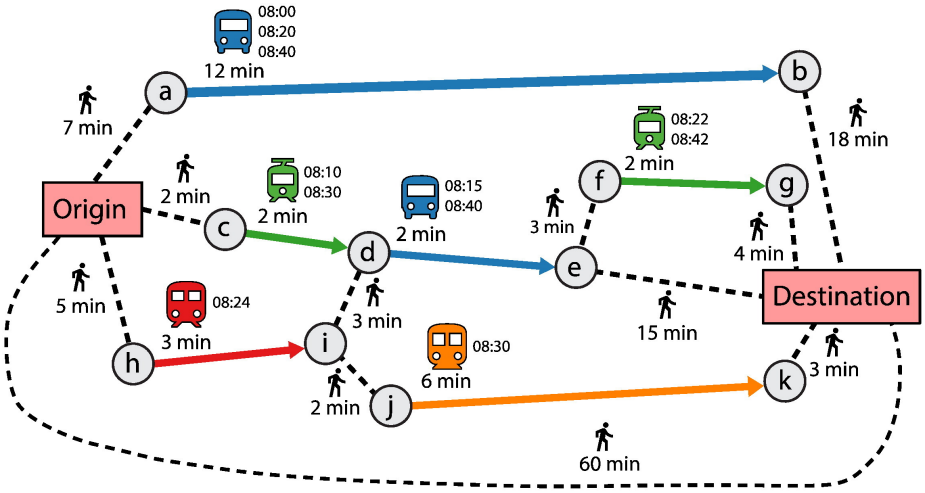

Figura 2.4: Diagrama explicativo del funcionamiento de Connection Scan Algorithm. Fuente: "Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys"(R. Kujala et al., 2017), vía sciencedirect.com .

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 2.5, generada utilizando el portal de Google Maps, el servidor web de visualización de mapas de Google [11], por su simplicidad de uso. Las rutas aparecen enumeradas según la lista previa.

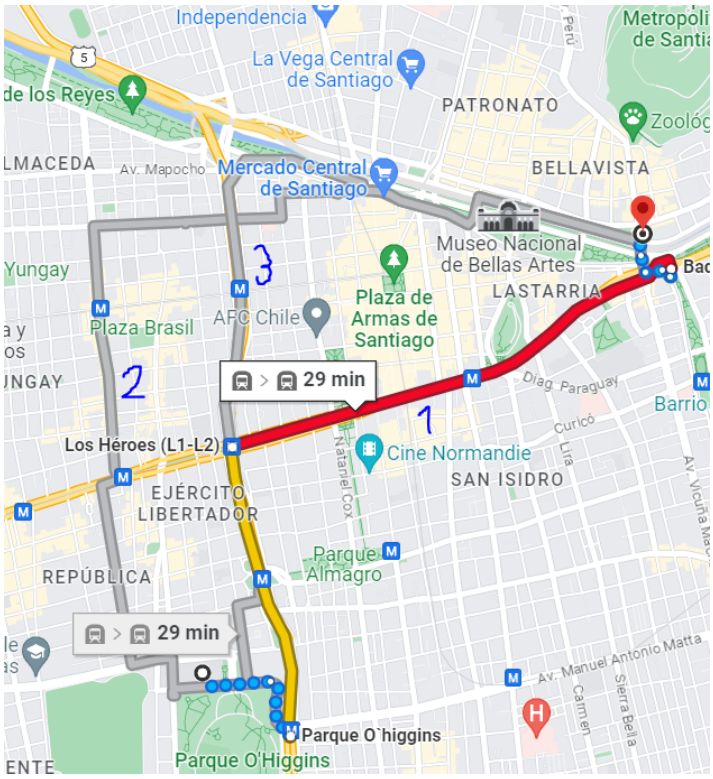

Figura 2.5: Posibles caminos para llegar desde FCFM hasta Derecho en transporte público. Fuente: Google Maps.

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* [21], 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 [18].

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** [22].
- La información del transporte público que se incluye en el modulo está almacenada en el formato GTFS, procesada mediante la librería **pygtfs** [4]. 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 [3].
- Para almacenar y trabajar con la información obtenida, se utiliza la librería **graph-tool** [5] 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** [13] 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** [9] 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

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. OSMData: la clase de OSM

Al instanciar la clase **OSMData**, 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 **OSMData** posee funcionalidades para visualizar los elementos del grafo de OSM, así como también funciones para operar con estos.

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 obtener información útil para generar rutas de transporte, tal como 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 [8], 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 [16].

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 [23], tiene disponible para descarga la información de los distintos países del mundo, incluído Chile [24]. 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 [22], la librería utilizada para procesar la información, permite leer datos de OpenStreetMap en formato PBF e interpretarla en estructuras de GeoPandas [15], 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 **OSMData** 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 **OSMData** 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** [13] 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 [3]. 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 [4], 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 B.

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 **OSMData** 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
```

4.2. Creando un algoritmo

En proceso...

Capítulo 5

Resultados

5.1. Ejemplos de uso del programa

5.2. Caso de estudio

5.3. Evaluación de resultados

Capítulo 6

Discusión

6.1. Implicancias

6.2. Limitaciones

6.3. Trabajo Futuro

Capítulo 7

Conclusión

Bibliografía

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

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

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

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

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

[6] 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.

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

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

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

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

[22] 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.

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

[24] 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.

[25] 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

Implementaciones existentes

A.1. Aalto University

```
class ConnectionScan(AbstractRoutingAlgorithm):
    """
    A simple implementation of the Connection Scan Algorithm (CSA) solving the first arrival problem
    for public transport networks.

    http://i11www.iti.uni-karlsruhe.de/extra/publications/dpsw-isftr-13.pdf
    """

    def __init__(self, transit_events, seed_stop, start_time,
                 end_time, transfer_margin, walk_network, walk_speed):
        """
        Parameters
        ----------
        transit_events: list[Connection]
        seed_stop: int
            index of the seed node
        start_time : int
            start time in unixtime seconds
        end_time: int
            end time in unixtime seconds (no new connections will be scanned after this time)
        transfer_margin: int
            required extra margin required for transfers in seconds
        walk_speed: float
            walking speed between stops in meters / second
        walk_network: networkx.Graph
```

```
            each edge should have the walking distance as a data attribute
    ("d_walk") expressed in meters
        """
        AbstractRoutingAlgorithm.__init__(self)
        self._seed = seed_stop
        self._connections = transit_events
        self._start_time = start_time
        self._end_time = end_time
        self._transfer_margin = transfer_margin
        self._walk_network = walk_network
        self._walk_speed = walk_speed

        # algorithm internals
        self.__stop_labels = defaultdict(lambda: float('inf'))
        self.__stop_labels[seed_stop] = start_time

        # trip flags:
        self.__trip_reachable = defaultdict(lambda: False)

    def get_arrival_times(self):
        """
        Returns
        -------
        arrival_times: dict[int, float]
            maps integer stop_ids to floats
        """
        assert self._has_run
        return self.__stop_labels

    def _run(self):
        self._scan_footpaths(self._seed, self._start_time)
        for connection in self._connections:
            departure_time = connection.departure_time
            if departure_time > self._end_time:
                return
            from_stop = connection.departure_stop
            to_stop = connection.arrival_stop
            arrival_time = connection.arrival_time
            trip_id = connection.trip_id
            reachable = False
            if self.__trip_reachable[trip_id]:
                reachable = True
            else:
                dep_stop_reached = self.__stop_labels[from_stop]
                if dep_stop_reached + self._transfer_margin <=
    departure_time:
                    self.__trip_reachable[trip_id] = True
                    reachable = True
            if reachable:
                self._update_stop_label(to_stop, arrival_time)
                self._scan_footpaths(to_stop, arrival_time)

    def _update_stop_label(self, stop, arrival_time):
        current_stop_label = self.__stop_labels[stop]
        if current_stop_label > arrival_time:
            self.__stop_labels[stop] = arrival_time
```

```
80
81      def _scan_footpaths(self, stop_id, walk_departure_time):
82          """
83          Scan the footpaths originating from stop_id
84
85          Parameters
86          ----------
87          stop_id: int
88          """
89          for _, neighbor, data in self._walk_network.edges(nbunch=[stop_id
    ], data=True):
90              d_walk = data["d_walk"]
91              arrival_time = walk_departure_time + d_walk / self._walk_speed
92              self._update_stop_label(neighbor, arrival_time)
93
94
95  class ConnectionScanProfiler(AbstractRoutingAlgorithm):
96      """
97      Implementation of the profile connection scan algorithm presented in
98
99      http://i11www.iti.uni-karlsruhe.de/extra/publications/dpsw-isftr-13.
    pdf
100     """
101
102     def __init__(self,
103                  transit_events,
104                  target_stop,
105                  start_time=None,
106                  end_time=None,
107                  transfer_margin=0,
108                  walk_network=None,
109                  walk_speed=1.5,
110                  verbose=False):
111         """
112         Parameters
113         ----------
114         transit_events: list[Connection]
115             events are assumed to be ordered in DECREASING departure_time
    (!)
116         target_stop: int
117             index of the target stop
118         start_time : int, optional
119             start time in unixtime seconds
120         end_time: int, optional
121             end time in unixtime seconds (no connections will be scanned
    after this time)
122         transfer_margin: int, optional
123             required extra margin required for transfers in seconds
124         walk_speed: float, optional
125             walking speed between stops in meters / second.
126         walk_network: networkx.Graph, optional
127             each edge should have the walking distance as a data attribute
     ("distance_shape") expressed in meters
128         verbose: boolean, optional
129             whether to print out progress
130         """
```

```
        AbstractRoutingAlgorithm.__init__(self)

        self._target = target_stop
        self._connections = transit_events
        if start_time is None:
            start_time = transit_events[-1].departure_time
        if end_time is None:
            end_time = transit_events[0].departure_time
        self._start_time = start_time
        self._end_time = end_time
        self._transfer_margin = transfer_margin
        if walk_network is None:
            walk_network = networkx.Graph()
        self._walk_network = walk_network
        self._walk_speed = float(walk_speed)
        self._verbose = verbose

        # algorithm internals

        # trip flags:
        self.__trip_min_arrival_time = defaultdict(lambda: float("inf"))

        # initialize stop_profiles
        self._stop_profiles = defaultdict(lambda: NodeProfileSimple())
        # initialize stop_profiles for target stop, and its neighbors
        self._stop_profiles[self._target] = NodeProfileSimple(0)
        if graph_has_node(walk_network, target_stop):
            for target_neighbor in walk_network.neighbors(target_stop):
                edge_data = walk_network.get_edge_data(target_neighbor,
target_stop)
                walk_duration = edge_data["d_walk"] / self._walk_speed
                self._stop_profiles[target_neighbor] = NodeProfileSimple(
walk_duration)

    def _run(self):
        # if source node in s1:
        previous_departure_time = float("inf")
        connections = self._connections  # list[Connection]
        n_connections = len(connections)
        for i, connection in enumerate(connections):
            # basic checking + printing progress:
            if self._verbose and i % 1000 == 0:
                print(i, "/", n_connections)
            assert (isinstance(connection, Connection))
            assert (connection.departure_time <= previous_departure_time)
            previous_departure_time = connection.departure_time

            # get all different "accessible" / arrival times (Pareto-
optimal sets)
            arrival_profile = self._stop_profiles[connection.arrival_stop]
  # NodeProfileSimple

            # Three possibilities:

            # 1. earliest arrival time (Profiles) via transfer
            earliest_arrival_time_via_transfer = arrival_profile.
```

```
    evaluate_earliest_arrival_time_at_target(
183                 connection.arrival_time, self._transfer_margin
184             )
185 
186             # 2. earliest arrival time within same trip (equals float('inf
    ') if not reachable)
187             earliest_arrival_time_via_same_trip = self.
    __trip_min_arrival_time[connection.trip_id]
188 
189             # then, take the minimum (or the Pareto-optimal set) of these
    three alternatives.
190             min_arrival_time = min(earliest_arrival_time_via_same_trip,
191                                    earliest_arrival_time_via_transfer)
192 
193             # If there are no 'labels' to progress, nothing needs to be
    done.
194             if min_arrival_time == float("inf"):
195                 continue
196 
197             # Update information for the trip
198             if earliest_arrival_time_via_same_trip > min_arrival_time:
199                 self.__trip_min_arrival_time[connection.trip_id] =
    earliest_arrival_time_via_transfer
200 
201             # Compute the new "best" pareto_tuple possible (later: merge
    the sets of pareto-optimal labels)
202             pareto_tuple = LabelTimeSimple(connection.departure_time,
    min_arrival_time)
203 
204             # update departure stop profile (later: with the sets of
    pareto-optimal labels)
205             dep_stop_profile = self._stop_profiles[connection.
    departure_stop]
206             updated_dep_stop = dep_stop_profile.
    update_pareto_optimal_tuples(pareto_tuple)
207             # if the departure stop is updated, one also needs to scan the
     footpaths from the departure stop
208             if updated_dep_stop:
209                 self._scan_footpaths_to_departure_stop(connection.
    departure_stop,
210                                                        connection.
    departure_time,
211                                                        min_arrival_time)
212 
213     def _scan_footpaths_to_departure_stop(self, connection_dep_stop,
    connection_dep_time, arrival_time_target):
214         """ A helper method for scanning the footpaths. Updates self.
    _stop_profiles accordingly"""
215         for _, neighbor, data in self._walk_network.edges(nbunch=[
    connection_dep_stop],
216                                                           data=True):
217             d_walk = data['d_walk']
218             neighbor_dep_time = connection_dep_time - d_walk / self.
    _walk_speed
219             pt = LabelTimeSimple(departure_time=neighbor_dep_time,
    arrival_time_target=arrival_time_target)
```

```
            self._stop_profiles[neighbor].update_pareto_optimal_tuples(pt)

    @property
    def stop_profiles(self):
        """
        Returns
        -------
        _stop_profiles : dict[int, NodeProfileSimple]
            The pareto tuples necessary.
        """
        assert self._has_run
        return self._stop_profiles
```

A.1.1. ULTRA: headers

```
    namespace CSA {

template<bool PATH_RETRIEVAL = true, typename PROFILER = NoProfiler>
class CSA {

public:
    constexpr static bool PathRetrieval = PATH_RETRIEVAL;
    using Profiler = PROFILER;
    using Type = CSA<PathRetrieval, Profiler>;
    using TripFlag = Meta::IF<PathRetrieval, ConnectionId, bool>;

private:
    struct ParentLabel {
        ParentLabel(const StopId parent = noStop, const bool
   reachedByTransfer = false, const TripId tripId = noTripId) :
            parent(parent),
            reachedByTransfer(reachedByTransfer),
            tripId(tripId) {
        }

        StopId parent;
        bool reachedByTransfer;
        union {
            TripId tripId;
            Edge transferId;
        };
    };

public:
    CSA(const Data& data, const Profiler& profilerTemplate = Profiler()) :
        data(data),
        sourceStop(noStop),
        targetStop(noStop),
        tripReached(data.numberOfTrips(), TripFlag()),
        arrivalTime(data.numberOfStops(), never),
        parentLabel(PathRetrieval ? data.numberOfStops() : 0),
        profiler(profilerTemplate) {
        AssertMsg(Vector::isSorted(data.connections), "Connections must be
    sorted in ascending order!");
```

```
38          profiler.registerPhases({PHASE_CLEAR, PHASE_INITIALIZATION,
    PHASE_CONNECTION_SCAN});
39          profiler.registerMetrics({METRIC_CONNECTIONS, METRIC_EDGES,
    METRIC_STOPS_BY_TRIP, METRIC_STOPS_BY_TRANSFER});
40          profiler.initialize();
41      }
42
43      inline void run(const StopId source, const int departureTime, const
    StopId target = noStop) noexcept {
44          profiler.start();
45
46          profiler.startPhase();
47          AssertMsg(data.isStop(source), "Source stop " << source << " is
    not a valid stop!");
48          clear();
49          profiler.donePhase(PHASE_CLEAR);
50
51          profiler.startPhase();
52          sourceStop = source;
53          targetStop = target;
54          arrivalTime[sourceStop] = departureTime;
55          relaxEdges(sourceStop, departureTime);
56          const ConnectionId firstConnection = firstReachableConnection(
    departureTime);
57          profiler.donePhase(PHASE_INITIALIZATION);
58
59          profiler.startPhase();
60          scanConnections(firstConnection, ConnectionId(data.connections.
    size()));
61          profiler.donePhase(PHASE_CONNECTION_SCAN);
62
63          profiler.done();
64      }
65
66      inline bool reachable(const StopId stop) const noexcept {
67          return arrivalTime[stop] < never;
68      }
69
70      inline int getEarliestArrivalTime(const StopId stop) const noexcept {
71          return arrivalTime[stop];
72      }
73
74      template<bool T = PathRetrieval, typename = std::enable_if_t<T ==
    PathRetrieval && T>>
75      inline Journey getJourney() const noexcept {
76          return getJourney(targetStop);
77      }
78
79      template<bool T = PathRetrieval, typename = std::enable_if_t<T ==
    PathRetrieval && T>>
80      inline Journey getJourney(StopId stop) const noexcept {
81          Journey journey;
82          if (!reachable(stop)) return journey;
83          while (stop != sourceStop) {
84              const ParentLabel& label = parentLabel[stop];
85              if (label.reachedByTransfer) {
```

```
86                  const int travelTime = data.transferGraph.get(TravelTime,
    label.transferId);
87                  journey.emplace_back(label.parent, stop, arrivalTime[stop]
     - travelTime, arrivalTime[stop], label.transferId);
88              } else {
89                  journey.emplace_back(label.parent, stop, data.connections[
    tripReached[label.tripId]].departureTime, arrivalTime[stop], label.
    tripId);
90              }
91              stop = label.parent;
92          }
93          Vector::reverse(journey);
94          return journey;
95      }
96
97      inline std::vector<Vertex> getPath(const StopId stop) const noexcept {
98          return journeyToPath(getJourney(stop));
99      }
100
101     inline std::vector<std::string> getRouteDescription(const StopId stop)
     const noexcept {
102         return data.journeyToText(getJourney(stop));
103     }
104
105     inline const Profiler& getProfiler() const noexcept {
106         return profiler;
107     }
108
109 private:
110     inline void clear() {
111         sourceStop = noStop;
112         targetStop = noStop;
113         Vector::fill(arrivalTime, never);
114         Vector::fill(tripReached, TripFlag());
115         if constexpr (PathRetrieval) {
116             Vector::fill(parentLabel, ParentLabel());
117         }
118     }
119
120     inline ConnectionId firstReachableConnection(const int departureTime)
    const noexcept {
121         return ConnectionId(Vector::lowerBound(data.connections,
    departureTime, [](const Connection& connection, const int time) {
122             return connection.departureTime < time;
123         }));
124     }
125
126     inline void scanConnections(const ConnectionId begin, const
    ConnectionId end) noexcept {
127         for (ConnectionId i = begin; i < end; i++) {
128             const Connection& connection = data.connections[i];
129             if (targetStop != noStop && connection.departureTime >
    arrivalTime[targetStop]) break;
130             if (connectionIsReachable(connection, i)) {
131                 profiler.countMetric(METRIC_CONNECTIONS);
132                 arrivalByTrip(connection.arrivalStopId, connection.
```

```
    arrivalTime, connection.tripId);
133             }
134         }
135     }
136 
137     inline bool connectionIsReachableFromStop(const Connection& connection
    ) const noexcept {
138         return arrivalTime[connection.departureStopId] <= connection.
    departureTime - data.minTransferTime(connection.departureStopId);
139     }
140 
141     inline bool connectionIsReachableFromTrip(const Connection& connection
    ) const noexcept {
142         return tripReached[connection.tripId] != TripFlag();
143     }
144 
145     inline bool connectionIsReachable(const Connection& connection, const
    ConnectionId id) noexcept {
146         if (connectionIsReachableFromTrip(connection)) return true;
147         if (connectionIsReachableFromStop(connection)) {
148             if constexpr (PathRetrieval) {
149                 tripReached[connection.tripId] = id;
150             } else {
151                 suppressUnusedParameterWarning(id);
152                 tripReached[connection.tripId] = true;
153             }
154             return true;
155         }
156         return false;
157     }
158 
159     inline void arrivalByTrip(const StopId stop, const int time, const
    TripId trip) noexcept {
160         if (arrivalTime[stop] <= time) return;
161         profiler.countMetric(METRIC_STOPS_BY_TRIP);
162         arrivalTime[stop] = time;
163         if constexpr (PathRetrieval) {
164             parentLabel[stop].parent = data.connections[tripReached[trip
    ]].departureStopId;
165             parentLabel[stop].reachedByTransfer = false;
166             parentLabel[stop].tripId = trip;
167         }
168         relaxEdges(stop, time);
169     }
170 
171     inline void relaxEdges(const StopId stop, const int time) noexcept {
172         for (const Edge edge : data.transferGraph.edgesFrom(stop)) {
173             profiler.countMetric(METRIC_EDGES);
174             const StopId toStop = StopId(data.transferGraph.get(ToVertex,
    edge));
175             const int newArrivalTime = time + data.transferGraph.get(
    TravelTime, edge);
176             arrivalByTransfer(toStop, newArrivalTime, stop, edge);
177         }
178     }
179 
```

```
180     inline void arrivalByTransfer(const StopId stop, const int time, const
        StopId parent, const Edge edge) noexcept {
181         if (arrivalTime[stop] <= time) return;
182         profiler.countMetric(METRIC_STOPS_BY_TRANSFER);
183         arrivalTime[stop] = time;
184         if constexpr (PathRetrieval) {
185             parentLabel[stop].parent = parent;
186             parentLabel[stop].reachedByTransfer = true;
187             parentLabel[stop].transferId = edge;
188         }
189     }
190
191 private:
192     const Data& data;
193
194     StopId sourceStop;
195     StopId targetStop;
196
197     std::vector<TripFlag> tripReached;
198     std::vector<int> arrivalTime;
199     std::vector<ParentLabel> parentLabel;
200
201     Profiler profiler;
202 };
203 }
```

Apéndice B

Código de la solución

B.1. Obtención de la información

B.1.1. OSM

```
1 def get_osm_data():
2     """
3     Obtains the required OpenStreetMap data using the 'pyrosm' library.
      This gives the map info of Santiago.
4
5     Returns:
6         graph: osm data converted to a graph
7     """
```

```
    # Download latest OSM data
    fp = get_data(
        "Santiago",
        update=True,
        directory=OSM_PATH
    )

    osm = 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
        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...")
```

```
63
64      for index, row in edges.iterrows():
65          source_node = row['u']
66          target_node = row['v']
67
68          if row["length"] < 2 or source_node == "" or target_node == "":
69              continue # Skip edges with empty or missing nodes
70
71          if source_node not in vertex_map or target_node not in vertex_map:
72              print(f"Skipping edge with missing nodes: {source_node} -> {
    target_node}")
73              continue  # Skip edges with missing nodes
74
75          source_vertex = vertex_map[source_node]
76          target_vertex = vertex_map[target_node]
77
78          if not graph.vertex(source_vertex) or not graph.vertex(
    target_vertex):
79              print(f"Skipping edge with non-existent vertices: {
    source_vertex} -> {target_vertex}")
80              continue  # Skip edges with non-existent vertices
81
82          # Calculate the distance between the nodes and use it as the
    weight of the edge
83          source_coords = node_coords[source_node]
84          target_coords = node_coords[target_node]
85          distance = abs(source_coords[0] - target_coords[0]) + abs(
    source_coords[1] - target_coords[1])
86
87          e = graph.add_edge(source_vertex, target_vertex)
88          u_prop[e] = source_node
89          v_prop[e] = target_node
90          length_prop[e] = row["length"]
91          weight_prop[e] = distance
92
93      graph.edge_properties["u"] = u_prop
94      graph.edge_properties["v"] = v_prop
95      graph.edge_properties["length"] = length_prop
96      graph.edge_properties["weight"] = weight_prop
97
98      print("OSM DATA HAS BEEN SUCCESSFULLY RECEIVED")
99      return graph
100
101 # OSM Graph
102 osm_graph = get_osm_data()
103
104 def make_undirected(graph):
105     """
106     Given a directed graph, returns an undirected version of the graph.
107
108     Parameters:
109     graph (Graph): A directed graph. In this specific case, the osm graph.
110
111     Returns:
112     Graph: An undirected version of the graph.
113     """
```

```
    undirected_graph = Graph(directed=False)
    vprop_map = graph.new_vertex_property("object")

    # Create vertex properties for lon and lat
    lon_prop = undirected_graph.new_vertex_property("float")
    lat_prop = undirected_graph.new_vertex_property("float")
    node_id_prop = undirected_graph.new_vertex_property("long")
    graph_id_prop = undirected_graph.new_vertex_property("long")

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

    undirected_vertex_map = {}

    for v in graph.vertices():
        new_v = undirected_graph.add_vertex()
        vprop_map[new_v] = v
        lon = graph.vertex_properties["lon"][v]
        lat = graph.vertex_properties["lat"][v]
        node_id = graph.vertex_properties["node_id"][v]
        graph_id = graph.vertex_properties["graph_id"][v]

        undirected_vertex_map[node_id] = new_v
        #print("NODO {} EN GRAFO {}".format(node_id, graph_id))

        # Assigning node properties
        lon_prop[new_v] = lon
        lat_prop[new_v] = lat
        node_id_prop[new_v] = node_id
        graph_id_prop[new_v] = graph_id

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


    for e in graph.edges():
        source, target = e.source(), e.target()
        source_node = graph.edge_properties["u"][e]
        target_node = graph.edge_properties["v"][e]
        lgt = graph.edge_properties["length"][e]
        wt = graph.edge_properties["weight"][e]

        if lgt < 2 or source_node == "" or target_node == "":
            continue # Skip edges with empty or missing nodes

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

```
168
169         source_vertex = undirected_vertex_map[source_node]
170         target_vertex = undirected_vertex_map[target_node]
171
172         if not undirected_graph.vertex(source_vertex) or not
    undirected_graph.vertex(target_vertex):
173             print(f"Skipping edge with non-existent vertices: {
    source_vertex} -> {target_vertex}")
174             continue  # Skip edges with non-existent vertices
175
176         e = undirected_graph.add_edge(source_vertex, target_vertex)
177         u_prop[e] = source_node
178         v_prop[e] = target_node
179         length_prop[e] = lgt
180         weight_prop[e] = wt
181
182     undirected_graph.edge_properties["u"] = u_prop
183     undirected_graph.edge_properties["v"] = v_prop
184     undirected_graph.edge_properties["length"] = length_prop
185     undirected_graph.edge_properties["weight"] = weight_prop
186
187     return undirected_graph
188
189 # Convertir el grafo en no dirigido
190 undirected_graph = make_undirected(osm_graph)
```

B.1.2. GTFS

```
1     def get_gtfs_data():
2     """
3     Reads the GTFS data from a file and creates a directed graph with its
    info, using the 'pygtfs' library. This gives
4     the transit feed data of Santiago's public transport, including "Red
    Metropolitana de Movilidad" (previously known
5     as Transantiago), "Metro de Santiago", "EFE Trenes de Chile", and "
    Buses de Acercamiento Aeropuerto".
6
7     Returns:
8         graphs: GTFS data converted to a dictionary of graphs, one per
    route.
9         route_stops: Dictionary containing the stops for each route.
10        special_dates: List of special calendar dates.
11    """
12    # Create a new schedule object using a GTFS file
13    sched = pygtfs.Schedule(":memory:")
14    pygtfs.append_feed(sched, "gtfs.zip")
15
16    # Get special calendar dates
17    special_dates = []
18    for cal_date in sched.service_exceptions: # Calendar_dates is renamed
    in pygtfs
19        special_dates.append(cal_date.date.strftime("%d/%m/%Y"))
20
21    # Create a graph per route
```

```
    graphs = {}
    stop_id_map = {}  # To assign unique ids to every stop
    stop_coords = {}
    route_stops = {}
    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]

        weight_prop = graph.new_edge_property("int")  # Propiedad para
almacenar los pesos de las aristas

        for trip in trips:
            stop_times = trip.stop_times

            # Get the orientation of the trip
            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()  # Add empty vertex
                    stop_id_map[stop_id] = vertex
                else:
                    vertex = stop_id_map[stop_id]  # Obtain existing
vertex

                stop_ids.add(vertex)

                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()  # Add an empty
vertex for the next stop
                        stop_id_map[next_stop_id] = next_vertex  # Assign
the vertex
                    else:
                        next_vertex = stop_id_map[next_stop_id]  # Obtain
the vertex

                    e = graph.add_edge(vertex, next_vertex)  # Add an edge
 between the vertexes
                    weight_prop[e] = 1

                    # Store the coordinates of each stop for this route
                    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)

```

```
                        # Store the sequence of each stop for this route
                        if route.route_id not in route_stops:
                            route_stops[route.route_id] = {}
                        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": []
                        }

                # Get the arrival time for the current stop
                arrival_time = (datetime.min + stop_times[i].arrival_time)
.time()

                # Check if the stop ID is already in the dictionary
                if stop_id in route_stops[route.route_id]:
                    # If the stop ID is already in the dictionary, append
the arrival time
                    route_stops[route.route_id][stop_id]["arrival_times"].
append(arrival_time)
                #else:
                    # If the stop ID is not in the dictionary, create a
new entry with the arrival time
                #    route_stops[route.route_id][stop_id] = {
                #        "route_id": route.route_id,
                #        "stop_id": stop_id,
                #        "coordinates": stop_coords[route.route_id][
stop_id],
                #        "visited_on_round_trip": True if orientation == "
I" else False,
                #        "visited_on_return_trip": True if orientation ==
"R" else False,
                #        "sequence": sequence,
                #        "arrival_times": [arrival_time]
                #    }

        graphs[route.route_id] = graph
        # Group the stops by direction to get the stops visited on the
round trip and the return trip
        stops_by_direction = {"round_trip": [], "return_trip": []}
        for trip in trips:
            stop_times = trip.stop_times
            stops = [stop_times[i].stop_id for i in range(len(stop_times))
]

            # Determine the direction of the trip
            if trip.direction_id == 0:
                stops_by_direction["round_trip"].extend(stops)
            else:
                stops_by_direction["return_trip"].extend(stops)


```

```
        # Get the unique stops visited on the round trip and the return trip
        round_trip_stops = set(stops_by_direction["round_trip"])
        return_trip_stops = set(stops_by_direction["return_trip"])

        for stop_id in round_trip_stops:
            if stop_id in stop_coords[route.route_id]:
                if stop_id in route_stops[route.route_id]:
                    route_stops[route.route_id][stop_id]["orientation"] = "round"
                else:
                    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",
                        "sequence": sequence,
                        "arrival_times": []
                    }
        for stop_id in return_trip_stops:
            if stop_id in stop_coords[route.route_id]:
                if stop_id in route_stops[route.route_id]:
                    route_stops[route.route_id][stop_id]["orientation"] = "return"
                else:
                    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": "return",
                        "sequence": sequence,
                        "arrival_times": []
                    }

    print("DONE")
    print("STORING ROUTE GRAPHS...")

    # Store graphs into a file
    for route_id, graph in graphs.items():
        weight_prop = graph.new_edge_property("int")  # Crear una nueva propiedad de peso de arista

        for e in graph.edges():  # Iterar sobre las aristas del grafo
            weight_prop[e] = 1  # Asignar el peso 1 a cada arista

        graph.edge_properties["weight"] = weight_prop  # Asignar la propiedad de peso al grafo

        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 graphs, route_stops, special_dates

# GTFS Graph
gtfs_graph, route_stops, special_dates = get_gtfs_data()
```

B.2. Creación del mapa

```
# Define the function to set the optimal zoom level for the map
def fit_bounds(points, m):
    """
    Fits the map bounds to a given set of points.

    Parameters:
    points (list): A list of points in the format [(lat1, lon1), (lat2, lon2), ...].
    m (folium.Map): A folium map object.
    """
    df = pd.DataFrame(points).rename(columns={0:'Lat', 1:'Lon'})[['Lat', 'Lon']]
    sw = df[['Lat', 'Lon']].min().values.tolist()
    ne = df[['Lat', 'Lon']].max().values.tolist()
    m.fit_bounds([sw, ne])

# Define the function to create a map that shows the correct public transport services to take from a source to a target
def create_transport_map(route_stops, selected_path, source_date, source_hour, margin):
    """
    Creates a map that shows the correct public transport services to take from a source to a target.

    Parameters:
    route_stops (dict): A dictionary that maps route IDs to a list of stop IDs.
    selected_path (list): A list of points in the format [(lat1, lon1), (lat2, lon2), ...].
    nighttime_flag (bool): A flag indicating whether to include nighttime routes.
    rush_hour_flag (bool): A flag indicating whether to include express routes during rush hour.
    margin (float): The margin in kilometers around the given addresses.

    Returns:
    folium.Map: A folium map object.
    """
    # Note: The margin represents the kilometers around the given addresses.
    # Example: a margin of 0.1 represents 0.1 km, or 100 meters.

    geolocator = Nominatim(user_agent="ayatori")
    source_lat = selected_path[0][0]
    source_lon = selected_path[0][1]
    target_lat = selected_path[-1][0]
```

```
    target_lon = selected_path[-1][1]
    source = geolocator.reverse((source_lat,source_lon))
    target = geolocator.reverse((target_lat,target_lon))

    # 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)

    # Add markers for the nearest stop from the source and target points
    source_coords = (selected_path[0][1], selected_path[0][0])
    near_source_stops, source_orientations = find_nearest_stops(source, margin)

    target_coords = (selected_path[-1][1], selected_path[-1][0])
    near_target_stops, target_orientations = find_nearest_stops(target, margin)

    fixed_orientation = None
    valid_services = set()
    for source_stop_id in near_source_stops:
        for target_stop_id in near_target_stops:
            services = connection_finder(route_stops, source_stop_id, target_stop_id)
            for service in services:
                source_orientation = get_bus_orientation(service, source_stop_id)
                target_orientation = get_bus_orientation(service, target_stop_id)
                source_sequence = int(get_trip_sequence(route_stops, service, source_stop_id))
                target_sequence = int(get_trip_sequence(route_stops, service, target_stop_id))
                if source_sequence > target_sequence:
                    continue
                if isinstance(source_orientation, list) and isinstance(target_orientation, list):
                    # If both source and target orientations are lists, check if any of the values match
                    valid_orientation = any(x in target_orientation for x in source_orientation) or any(x in source_orientation for x in target_orientation)
                    if valid_orientation and service not in valid_services:
                        valid_services.add(service)
                        fixed_orientation = [x for x in source_orientation if x in target_orientation][0] if [x for x in source_orientation if x in target_orientation] else source_orientation[0]
```

```
73                elif source_orientation == target_orientation and service
    not in valid_services:  # Check if both stops are visited in the same
    orientation
74                    valid_services.add(service)
75                    fixed_orientation = target_orientation
76                elif isinstance(source_orientation, list) and
    target_orientation in source_orientation and service not in
    valid_services:
77                    valid_services.add(service)
78                    fixed_orientation = target_orientation
79                elif isinstance(target_orientation, list) and
    source_orientation in target_orientation and service not in
    valid_services:
80                    valid_services.add(service)
81                    fixed_orientation = source_orientation
82
83     #print("ORIENTATION FIXED: {}".format(fixed_orientation))
84
85     if len(valid_services) == 0:
86         print("Error: There are no available services right now to go to
    the desired destination.")
87         print("Possible reasons: no routes that have stops near the source
     and target addresses.")
88         print("You can try changing the search margin and try again.")
89         return
90
91     nighttime_flag = is_nighttime(source_hour)
92     rush_hour_flag = is_rush_hour(source_hour)
93     holiday_flag = is_holiday(source_date)
94     if holiday_flag:
95         rush_hour_flag = 0
96
97     # Nighttime check
98     daily_time_services = check_night_routes(valid_services,
    nighttime_flag)
99
100    if daily_time_services is None:
101        print("Error: There are no available services right now to go to
    the desired destination.")
102        print("Possible reasons: Source hour is during nighttime.")
103        print("Please take into account that nighttime goes between
    00:00:00 and 05:30:00.")
104        return
105
106    # Rush hour check
107    valid_services = check_express_routes(daily_time_services,
    rush_hour_flag)
108
109    valid_services = list(set(valid_services))
110
111    valid_source_stops = [stop_id for stop_id in near_source_stops if any(
    route_id in valid_services for route_id in route_stops.keys() if
    stop_id in route_stops[route_id])]
112    valid_source_stops = list(set(valid_source_stops))
113    valid_target_stops = [stop_id for stop_id in near_target_stops if any(
    route_id in valid_services for route_id in route_stops.keys() if
```

```
    stop_id in route_stops[route_id])]
114     valid_target_stops = list(set(valid_target_stops))
115 
116     # Give info
117     print("")
118     print("Routes have been found.")
119     print("Calculating the best route and getting the arrival times for
    the next buses...")
120 
121     best_option = None
122     best_option_times = None
123     source_time = timedelta(hours=source_hour.hour, minutes=source_hour.
    minute, seconds=source_hour.second)
124 
125     valid_orientations = set(source_orientations)
126 
127     best_option_orientation = None
128 
129     valid_target = []
130     for target_stop in valid_target_stops:
131         target_routes = get_routes_at_stop(route_stops, target_stop)
132         valid_target.extend(target_routes)
133     valid_target = list(dict.fromkeys(valid_target))
134 
135     for stop_id in valid_source_stops:
136         routes_at_stop = get_routes_at_stop(route_stops, stop_id)
137         valid_stop_services = [stop_id for stop_id in valid_services if
    stop_id in routes_at_stop]
138         for valid_service in valid_stop_services:
139             a_t = get_arrival_times(valid_service, stop_id, source_date)
140             if a_t is not None and a_t[0] == fixed_orientation:
141                 orientation = a_t[0]
142                 flag = False
143                 for target_stop_id in valid_target_stops:
144                     flag = False
145                     target_stop_routes = get_routes_at_stop(route_stops,
    target_stop_id)
146                     if valid_service in target_stop_routes:
147                         target_orientation = get_bus_orientation(
    valid_service, target_stop_id)
148                         if a_t[0] != target_orientation:
149                             flag = True
150                             continue
151 
152                 if flag:
153                     continue
154 
155                 if valid_service not in valid_target:
156                     continue
157 
158                 arrival_times = a_t[1]
159                 #print(arrival_times)
160                 time_until_next_buses = get_time_until_next_bus(
    arrival_times, source_hour, source_date)
161 
162                 if not time_until_next_buses:
```

```
                    print("Error: There are no available services right now to go to the desired destination.")
                    print("Possible reasons: There are no buses left today. Maybe the source hour is too close to the ending time for the service.")
                    return


                # Print the time until the next three buses in the desired format
                for i in range(len(time_until_next_buses)):
                    minutes, seconds = time_until_next_buses[i]
                    waiting_time = timedelta(minutes=minutes, seconds=seconds)
                    arrival_time = source_time + waiting_time
                    time_string = timedelta_to_hhmm(arrival_time)

                    target_orientation = get_bus_orientation(valid_service, target_stop_id)

                    # Update the best option
                    if (best_option is None or (arrival_time < best_option[2])) and orientation == fixed_orientation:
                        best_option = (valid_service, stop_id, arrival_time, waiting_time)
                        best_option_times = time_until_next_buses
                        best_option_orientation = orientation

    if best_option is None:
        print("Error: There are no available services right now to go to the desired destination.")
        print("Possible reasons: maybe a small margin. You can try using a bigger one")
        return

    # Print the best option
    arrival_time = None

    print("")
    print("To go from: {}".format(source))
    print("To: {}".format(target))
    best_arrival_time_str = timedelta_to_hhmm(best_option[2])
    print("The best option is to take the route {} on stop {}. The next bus arrives at {}.".format(best_option[0], best_option[1], best_arrival_time_str))
    print("The other two next buses arrives in:")
    for i in range(len(best_option_times)):
        if i == 0:
            continue
        minutes, seconds = best_option_times[i]
        waiting_time = timedelta(minutes=minutes, seconds=seconds)
        arrival_time = source_time + waiting_time
        time_string = timedelta_to_hhmm(arrival_time)
        print(f"{minutes} minutes, {seconds} seconds ({time_string})")


```

```
    # Map the options
    for stop_id in near_source_stops:
        if stop_id in valid_source_stops:
            stop_coords = get_stop_coords(route_stops, str(stop_id))
            routes_at_stop = get_routes_at_stop(route_stops, stop_id)
            valid_stop_services = [stop_id for stop_id in valid_services
if stop_id in routes_at_stop]

            for service in valid_stop_services:
                if service == best_option[0] and stop_id == best_option
[1]:
                    folium.Marker(location=[stop_coords[1], stop_coords
[0]],
                            popup="Mejor opcion: subirse al recorrido {} en
el paradero {}.".format(best_option[0], best_option[1]),
                            icon=folium.Icon(color='cadetblue', icon='plus')
).add_to(m)
                    initial_distance = [(selected_path[0][0],
selected_path[0][1]),(stop_coords[1], stop_coords[0])]
                    folium.PolyLine(initial_distance,color='black',
dash_array='10').add_to(m)

    for stop_id in near_target_stops:
        if stop_id in valid_target_stops:
            stop_coords = get_stop_coords(route_stops, str(stop_id))
            routes_at_stop = get_routes_at_stop(route_stops, stop_id)
            valid_stop_services = [stop_id for stop_id in valid_services
if stop_id in routes_at_stop]

    target_orientation = None
    for service in valid_target:
        if service == best_option[0]:
            if fixed_orientation == "round":
                trip_id = service + "-I-" + get_trip_day_suffix(
source_date)
            else:
                trip_id = service + "-R-" + get_trip_day_suffix(
source_date)

            best_travel_time = None
            selected_stop = None
            for stop_id in valid_target_stops:
                bus_time = get_travel_time(trip_id, [best_option[1],
stop_id])
                target_stop_routes = get_routes_at_stop(route_stops,
stop_id)
                target_orientation = get_bus_orientation(best_option[0],
stop_id)
                if service in target_stop_routes and bus_time > timedelta
() and (best_travel_time is None or bus_time < best_travel_time):
                    if fixed_orientation == target_orientation:
                        best_travel_time = bus_time
                        selected_stop = stop_id

            selected_stop_coords = get_stop_coords(route_stops,
selected_stop)
```

```
            minutes, seconds = timedelta_separator(best_travel_time)

            print("DEBUG")
            seq_1 = route_stops[best_option[0]][best_option[1]]["sequence"]
            seq_2 = route_stops[best_option[0]][selected_stop]["sequence"]
            print(seq_1, seq_2)
            print("")
            print("You will get off the bus on stop {} after {} minutes and {} seconds.".format(selected_stop, minutes, seconds))

            folium.Marker(location=[selected_stop_coords[1], selected_stop_coords[0]],
                          popup="Mejor opcion: bajarse del recorrido {} en el paradero {}.".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)

            total_time = best_option[3] + best_travel_time
            minutes, seconds = timedelta_separator(total_time)

            destination_time = source_time + total_time
            time_string = timedelta_to_hhmm(destination_time)
            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
```

B.3. Operación del algoritmo

```
def connection_scan(graph, source_address, target_address, departure_time, departure_date):
    """
    The Connection Scan algorithm is applied to search for travel routes from the source to the destination,
    given a departure time and date. By default, the algorithm uses the current date and time of the system.
    However, you can specify a different date or time if needed.

    Args:
        graph (graph): the graph used to visualize the travel routes.
        source_address (string): the source address of the travel.
        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.
```

```
13
14      Returns:
15          list: the list of coordinates of the travel connections needed to
        arrive at the destination.
16      """
17      node_id_mapping = create_node_id_mapping(graph)
18
19      source_node = address_locator(graph, source_address)
20      target_node = address_locator(graph, target_address)
21
22      if source_node is not None and target_node is not None:
23          # Convert source and target node IDs to integers
24          source_node_graph_id = graph.vertex_properties["graph_id"][
        source_node]
25          target_node_graph_id = graph.vertex_properties["graph_id"][
        target_node]
26
27          print("Both addresses have been found.")
28          print("Processing...")
29          #print("SOURCE NODE: {}. TARGET NODE: {}.".format(
        source_node_graph_id, target_node_graph_id))
30
31          path = [source_node_graph_id, target_node_graph_id]
32          path_coords = []
33          for node in path:
34              lon, lat = graph.vertex_properties["lon"][node], graph.
        vertex_properties["lat"][node]
35              path_coords.append((lat, lon))
36          #print(path)
37
38          return path_coords
39      else:
40          return
41
42
43  def csa_commands():
44      """
45      Process the inputs given by the user to run the Connection Scan
        Algorithm.
46      """
47
48      # System's date and time
49      now = datetime.now()
50      dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
51      #print("Fecha y hora actuales =", dt_string)
52
53      # Date formatting
54      today = date.today()
55      today_format = today.strftime("%d/%m/%Y")
56
57      # Time formatting
58      moment = now.strftime("%H:%M:%S")
59      used_time = datetime.strptime(moment, "%H:%M:%S").time()
60
61      # User inputs
62      # Date and 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 address
    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() != '':
            #print("Direccion de Destino ingresada: " + target_address)
            break

    # Destination address
    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() != '':
            #print("Direccion de Destino ingresada: " + target_address)
            break

    start = tm.time()

    path_coords = connection_scan(undirected_graph, source_address, target_address, source_hour, source_date)

    if path_coords:
        map = create_transport_map(route_stops, path_coords, source_date, source_hour, 0.2)
        if map:
            display(map)

        end = tm.time()
        exec_time = round((end-start) / 60,3)
        print("MAP IS READY. EXECUTION TIME: {} MINUTES".format(exec_time))
        return path_coords

    else:
        print("")
        print("Something went wrong. Please try again later.")
        return

selected_path = csa_commands()
```