



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

AYATORI: IMPLEMENTACIÓN EFICIENTE DE CONNECTION SCAN EN GTFS
CON CASO DE ESTUDIO DE MOVILIDAD EN SANTIAGO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

FELIPE IGNACIO LEAL CERRO

PROFESOR GUÍA:
EDUARDO GRAELLS GARRIDO

MIEMBROS DE LA COMISIÓN:
EDUARDO GRAELLS GARRIDO
NELSON BALOIAN TATARYAN
HERNÁN SARMIENTO ALBORNOZ

SANTIAGO DE CHILE
2023

Resumen

El presente informe presenta la creación de la Memoria para optar al Título Profesional de Ingeniero Civil en Computación. A lo largo de este, se detalla la planificación en la creación de un algoritmo computacional (programado en Python) que, entregando coordenadas de inicio y término dentro de una ciudad estipulada, sea capaz de utilizar la información disponible para entregar especificaciones de rutas que unan ambos puntos geográficos. Para evaluar la utilidad y correcta implementación de esta solución, se realizará un caso de estudio para Santiago de Chile.

Las circunstancias en que uno nace no tienen importancia, es lo que uno hace con el don de la vida lo que nos dice quienes somos.

-Mewtwo.

Agradecimientos

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

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 variadas. 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 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.

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 amiguito Mati.

Gracias a todas mis profesoras y profesores, por darme las herramientas para llegar a donde estoy hoy.

Tabla de Contenido

1. Introducción	1
2. Estado del Arte	4
2.1. Connection Scan	4
2.1.1. Implementaciones existentes	6
2.2. OpenStreetMap	6
2.2.1. OSM integrado en Connection Scan	7
2.3. GTFS	8
2.3.1. GTFS integrado en Connection Scan	10
3. Diseño	11
3.1. Stack tecnológico	11
3.2. Funcionamiento lógico	12
3.3. Criterio de Evaluación	13
4. Implementación	14
4.1. Obtención de datos	14
4.1.1. Procesamiento de OpenStreetMap	14
4.1.2. Procesamiento de GTFS	17
4.2. Generación del output	19
4.2.1. Texto	19
4.2.2. Mapa	22

5. Resultados	25
5.1. Usabilidad del programa	25
5.2. Caso de estudio	27
5.3. Evaluación de resultados	27
6. Discusión	28
6.1. Implicancias	28
6.2. Limitaciones	28
6.3. Trabajo Futuro	28
7. Conclusión	29
Bibliografía	31
Anexos	32
Apéndice A. Implementaciones existentes	32
A.1. Aalto University	32
A.1.1. ULTRA: headers	37
Apéndice B. Código de la solución	42
B.1. Creación de grafos	42
B.1.1. OSM	42
B.1.2. GTFS	46
B.2. Creación del mapa	50
B.3. Operación del algoritmo	57

Índice de Ilustraciones

2.1.	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	4
2.2.	Posibles caminos para llegar desde FCFM hasta Derecho en transporte público. Fuente: Google Maps.	5
2.3.	Mapa de Santiago en OpenStreetMap. Fuente: openstreetmap.cl	7
2.4.	Visualización de archivos del feed GTFS para Santiago.	8
2.5.	Diagrama de uso de datos en formato GTFS. Fuente: watrifeed.ml	9
4.1.	Ejemplo de salida del algoritmo (texto). Ruta entre FCFM y Puente Cal y Canto.	22
4.2.	Mapa de Santiago, generado con folium.	22
4.3.	Ejemplo de salida del algoritmo (mapa). Ruta entre FCFM y Puente Cal y Canto.	24
5.1.	Resultado de la implementación del algoritmo: viaje de domingo de madrugada por la Alameda.	26

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 interna y externa al país durante los últimos años, y la construcción de nueva infraestructura urbana. Si bien han existido ciertas condiciones que afecten negativamente el florecimiento de las ciudades (como la pandemia del COVID-19), la tendencia general de crecimiento se mantiene. Así es como, en un mundo donde las grandes urbes tienden a crecer exponencialmente, la planificación y buena gestión de las ciudades se ha visto afectada por el auge de estos fenómenos.

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

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 el patrón de viajes de los habitantes de la ciudad, 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 aquí fue en el año 2012, hace más de una década atrás [17]. 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. 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. La gente utiliza el MetroTren, el Metro de Santiago, y los buses de Red en múltiples combinaciones, generando una cantidad enorme de rutas diferentes para moverse por la ciudad. 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) [4] 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.

Connection Scan [10] (CSA) es un algoritmo creado para responder de manera eficiente a consultas en los sistemas de información de horarios. Este algoritmo es capaz de recibir como entrada una posición de origen y una posición de destino, generando una salida consistente en una secuencia de vehículos que el viajero debe tomar para recorrer una ruta entre ambos puntos. CSA se alimenta de la información del transporte público disponible, así que es útil enlazar la información cartográfica de la ciudad con datos en formato GTFS para que logre operar correctamente.

Este trabajo de título tiene por objetivo principal realizar una implementación de Connection Scan, alimentándolo con la información del transporte en Santiago, para poder analizar los patrones de movilidad disponibles para los habitantes de la ciudad. La idea es que el producto generado permita, mediante este análisis, realizar estudios de movilidad vial de forma más actualizada y directa. 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.

Objetivos

Objetivo General

El objetivo general de este trabajo de título es crear una implementación de Connection Scan Algorithm (CSA) en Python, con el fin de generar un programa que permita trazar rutas entre dos puntos cualesquiera de Santiago de Chile. Para ello, se utilizarán los datos cartográficos de la ciudad provenientes de OpenStreetMap, y la información del transporte público provista por la Red Metropolitana de Movilidad (en formato GTFS).

Objetivos Específicos

1. Obtener la información cartográfica de Santiago, además de la información del transporte público (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. Desarrollar una implementación de CSA en Python, que permita ubicar dos puntos cualesquiera de la ciudad y entregar una serie de trasbordos a realizar para llegar del origen al destino.
4. Utilizar el algoritmo para realizar un caso de estudio que ejemplifique la utilidad del trabajo realizado.

Capítulo 2

Estado del Arte

2.1. Connection Scan

Como fue mencionado anteriormente, Connection Scan Algorithm (CSA) es un algoritmo desarrollado para responder, de manera eficiente, consultas en sistemas de información de horarios [10]. Este algoritmo es capaz de optimizar los tiempos de viaje entre dos puntos determinados de origen y destino, puntos que recibe como entrada, y, siendo alimentado por distintas fuentes de información de transporte, entrega como resultado 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 trasbordos, 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 el siguiente diagrama, se grafica el funcionamiento antes descrito:

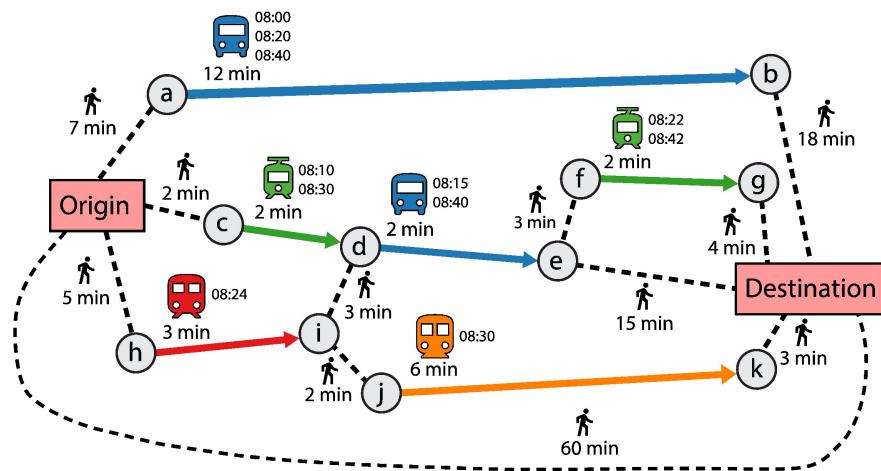


Figura 2.1: Diagrama explicativo del funcionamiento de Connection Scan Algorithm. Fuente: "Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys" (R. Kujala et al., 2017), vía sciencedirect.com .

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 trasbordos 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 trasbordos entre ambos medios de transporte (subir en estación Parque O'Higgins y andar hasta bajar en Puente Cal y Canto, para luego tomar el recorrido 502).

Para graficar el ejemplo dado, se introduce la siguiente figura 2.2, generada utilizando el portal de Google Maps, el servidor web de visualización de mapas de Google [13], por su simplicidad de uso; cabe destacar que Google Maps utiliza otro algoritmo para calcular sus rutas (basado en el algoritmo de búsqueda A*). Las rutas aparecen enumeradas según la lista previa.

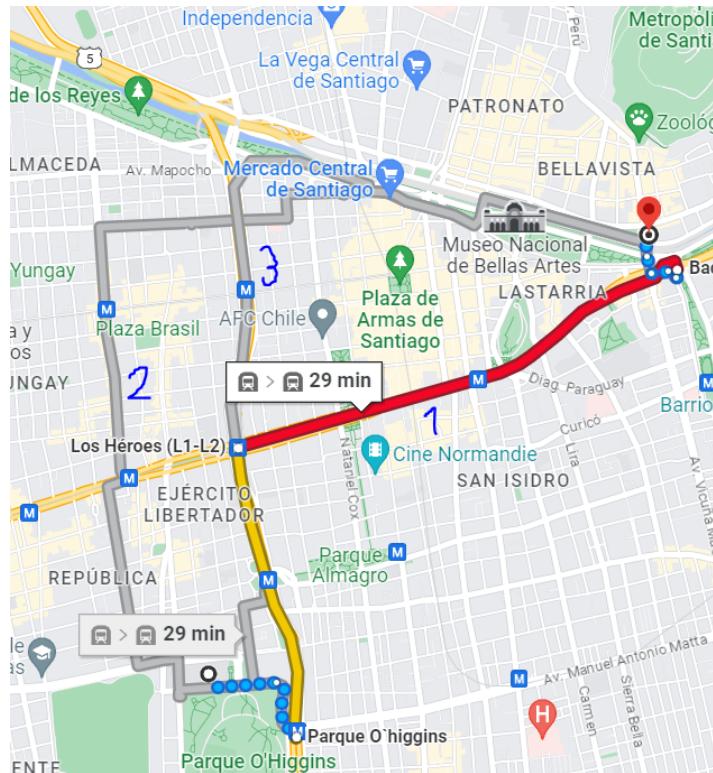


Figura 2.2: Posibles caminos para llegar desde FCFM hasta Derecho en transporte público.
Fuente: Google Maps.

Si utilizamos el ejemplo con Connection Scan, el algoritmo recibiría como entrada las coordenadas del punto de origen y el punto de destino. Luego, revisando la información

del transporte público, calcularía las rutas posibles (como las descritas anteriormente). CSA entonces buscaría el punto óptimo de Pareto con respecto a los trasbordos, y entregaría la ruta “recomendada” para llegar al destino deseado. Evidentemente, como se trabaja con información ‘estática’, esto da por sentados ciertos supuestos, como una velocidad de caminata fija entre trasbordos y la continuidad operativa del servicio en todo momento.

Connection Scan Algorithm precisa, en primera instancia, de 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 requiere de coordenadas de origen y destino, y de que, como se expresa en el ejemplo anterior, la creación de un mapa es una buena forma de graficar el resultado, el algoritmo también requiere de los datos cartográficos del sector o ciudad en cuestión donde se desea realizar el viaje. Trabajando con ambos flujos de información, CSA es capaz de realizar su funcionamiento correctamente.

2.1.1. Implementaciones existentes

Connection Scan Algorithm fue presentado en un paper que fue publicado en marzo de 2017. En el lapso de tiempo hasta este punto, 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 implementaciones realizadas de Connection Scan [28], basadas en el paper que lo define [10]. Destaca, entre estos, el repositorio de ULTRA: UnLimited TRAnsfers for Multimodal Route Planning [24], 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 [21]. Parte de estas implementaciones se pueden analizar en el código provisto en el anexo A.

Analizando el terreno actual, existe más de un ejemplo existente de implementación del algoritmo, en más de un lenguaje de programación. Asimismo, existen programas o frameworks que dicen incluir a Connection Scan internamente, pero no en todos estos casos el algoritmo se encuentra expuesto al público. Estas ideas motivan la decisión de que, en este trabajo de título, el algoritmo sea implementado en Python, como se explica en la sección 3.1.

2.2. OpenStreetMap

OpenStreetMap (OSM) es un proyecto colaborativo cuyo propósito es crear mapas editables y de uso libre [12]. Los mapas, generados utilizando la información geográfica capturada con dispositivos GPS móviles, contienen los caminos de la vía pública (pasajes, calles, carreteras) y diversos puntos de interés, entre otros elementos como las paradas de autobuses. 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 [3] cumpliendo ese papel. Se presenta la figura 2.3 a modo de ejemplo, donde se observa el mapa del Gran Santiago, 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.

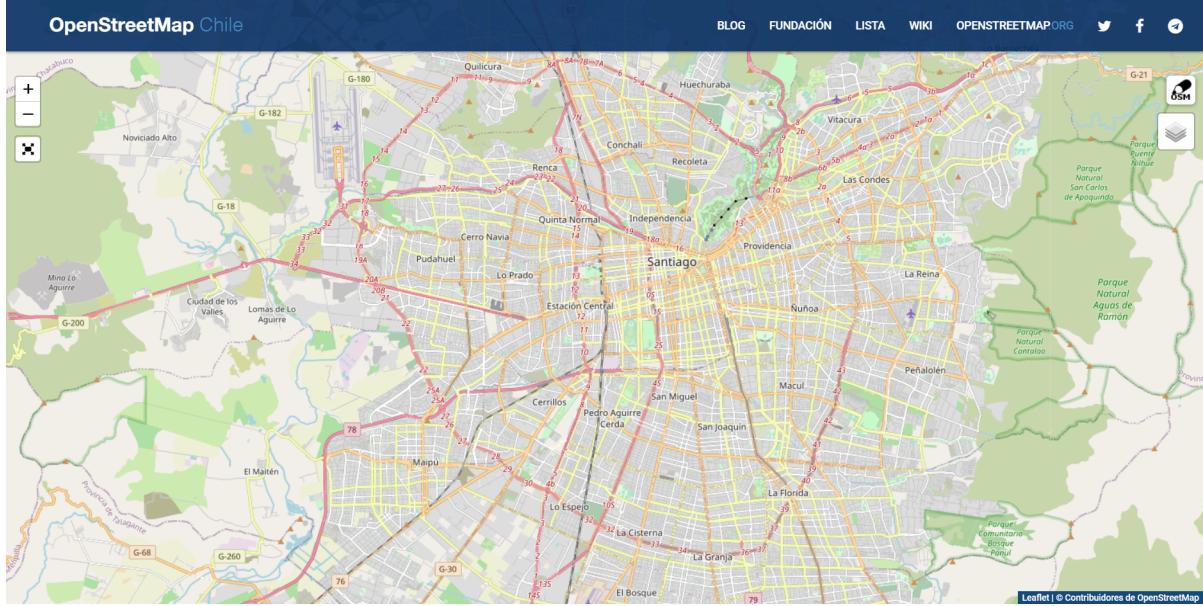


Figura 2.3: Mapa de Santiago en OpenStreetMap. Fuente: openstreetmap.cl .

2.2.1. OSM integrado en Connection Scan

Connection Scan es capaz de alimentarse de los datos provenientes de OpenStreetMap para calcular la salida del algoritmo. Desde OSM se puede obtener la información cartográfica de una ciudad, la cual es utilizada para poder 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.

Existe una librería llamada Pyrosm [25], que 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 los nodos representan las intersecciones entre las vías (además de lugares de interés, como escuelas), 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).

2.3. 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 2.4, se muestra el cómo se ven los archivos en GTFS, mostrando, como ejemplo, la información de los paraderos disponibles.

Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..					
agency.txt					
calendar.txt					
calendar_dates.txt					
feed_info.txt					
frequencies.txt					
routes.txt					
shapes.txt					
stop_times.txt					
stops.txt					
trips.txt					
stops: Bloc de notas					
Archivo Edición Formato Ver Ayuda					
stop_id,stop_code,stop_name,stop_lat,stop_lon,stop_url,wheelchair_boarding					
PB241,,PB241-Parada / Mall Plaza Norte - Los Libertadores,-33.3640607810884,-70.6813493519124,,0					
PB184,,PB184-Los Libertadores / Esq. Av. A. Vespucio,-33.3668314015184,-70.68097565189,,0					
PB185,,PB185-Parada / Pasarela Albany,-33.3660777516853,-70.6857970004004,,0					
PB242,,PB242-Parada 6 / (M) Los Libertadores,-33.366339735398,-70.6896183397335,,0					
PB186,,PB186-Av. Independencia / Esq. H. De Iquique,-33.3691429425543,-70.6886829663555,,0					
PB187,,PB187-Av. Independencia / Esq. H. La Concepción,-33.3715637184741,-70.6877189999396,,0					
PB188,,PB188-Parada 1 / (M) Cardenal Caro,-33.3733848493152,-70.6866856839114,,0					
PB251,,PB251-Av. Independencia / Esq. Trigal,-33.3767543687322,-70.6845528917126,,0					
PB190,,PB190-Av. Independencia / Esq. El Cortijo,-33.378221338379,-70.6839109982666,,0					
PB191,,PB191-Av. Independencia / Esq. Quilicura,-33.3810873058301,-70.6829978353462,,0					
PB1707,,PB1707-Av. Independencia / Esq. El Pino,-33.3829108397857,-70.6819075609822,,0					
PB193,,PB193-Parada 1 / (M) Vivaceta,-33.3856989105895,-70.6796928124103,,0					
PB194,,PB194-Av. Independencia / Esq. Módulo Lunar,-33.3878078782433,-70.6783420004488,,0					
PB195,,PB195-Av. Independencia / Esq. El Olivo,-33.3907379294037,-70.6753029996458,,0					
PB196,,PB196-Av. Independencia / Esq. Teniente Ponce,-33.3932709572678,-70.673612000302,,0					
PB259,,PB259-Av. Independencia / Esq. B. Mandujano,-33.3961389920302,-70.6715719995143,,0					
PB198,,PB198-Parada 1 / (M) Conchalí,-33.3976901519879,-70.6699543656155,,0					
PB1967,,PB1967-Av. Independencia / Esq. Pablo Urzúa,-33.401021562569,-70.6659517980744,,0					
PB1964,,PB1964-Parada 1 / Einstein - Independencia,-33.404129709827,-70.6630924980754,,0					
PB1960,,PB1960-Parada 6 / (M) Pza. Chacabuco,-33.4074456487041,-70.6607268980761,,0					
PB1958,,PB1958-Av. Independencia / Esq. Los Nidos,-33.4102115656897,-70.6596860980763,,0					
PB1957,,PB1957-Av. Independencia / Esq. Central,-33.4128456851102,-70.6584953980766,,0					
PB1955,,PB1955-Av. Independencia / Esq. San Luis,-33.4158659093028,-70.6570093980769,,0					
PB1953,,PB1953-Parada 1 / (M) Hospitales,-33.4179565202081,-70.656335998077,,0					
PB1951,,PB1951-Parada 3 / Hospital Clínico U. De Chile,-33.4209206316209,-70.6556279980771,,0					
PB1950,,PB1950-Av. Independencia / Esq. Dávila Baeza,-33.4257940491756,-70.6545362980771,,0					
PB1948,,PB1948-Av. Independencia / Esq. General Prieto,-33.4296331656137,-70.6535143980772,,0					

Figura 2.4: 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), organismo dependiente del Ministerio de Transportes y Telecomunicaciones, y cuya misión es mejorar la calidad del sistema de transporte público en la ciudad, tiene disponible públicamente esta información, y la actualiza periódicamente [8]; al momento de la entrega de este informe mejorado, la última versión fue configurada para implementarse desde el 8 de julio de 2023. La información 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 los días laborales (lunes a viernes), para los sábados, y 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 2.5.

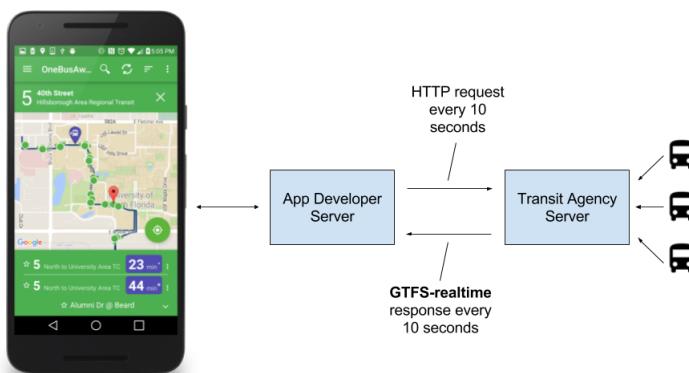


Figura 2.5: Diagrama de uso de datos en formato GTFS. Fuente: watrifeed.ml.

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.

2.3.1. GTFS integrado en Connection Scan

Connection Scan necesita tener a su disposición la información del transporte público para ser capaz de calcular las rutas solicitadas, por lo que es útil entregarle dicha información en formato GTFS. De esta manera, el algoritmo tendrá información de los distintos recorridos disponibles, incluyendo sus rutas, paradas, horarios, y cualquier otra información que se estime necesaria para poder obtener la secuencia de transbordos que formen la ruta a seguir.

Similar al caso de OSM, existe una librería de Python llamada pygtfs [6], la que modela información en formato GTFS en una interfaz de Python. La librería almacena la información del transporte público en una base de datos relacional, tal que pueda ser usada en proyectos programados en este lenguaje.

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 simular de manera correcta el funcionamiento del algoritmo, se debe procesar correctamente la información, tanto los datos cartográficos de Santiago, como la información del transporte público. Para desarrollar el algoritmo, se define lo siguiente:

- El algoritmo será programado en Python.
- La información cartográfica se obtendrá desde OpenStreetMap, procesada mediante la librería Pyrosm [25].
- La información del transporte público se obtendrá en el formato GTFS, procesada mediante la librería Pygtfs [6]. Los datos serán obtenidos previamente, descargando la última versión del sitio web del Directorio de Transporte Público Metropolitano [4].
- Para almacenar y trabajar con la información obtenida, se utilizará la librería graph-tool [7] para trabajar con grafos.

Con respecto al último punto, se decide utilizar 'graph-tool' por ser una librería eficiente para realizar análisis de redes de gran tamaño, tales como la red vial de una ciudad. Al tener una base algorítmica implementada en C++, un lenguaje de programación basado en compilación, el procesamiento de información se realiza más rápidamente, lo que permite trabajar con grandes volúmenes de información de mejor manera. Como consecuencia de utilizar 'graph-tool', al no estar disponible para sistemas operativos Windows, la programación del algoritmo se realiza en Ubuntu, mediante WSL2 (Windows Subsystem for Linux 2 [20]).

Otras librerías que son utilizadas para cumplir de mejor forma los objetivos especificados en la sección 1 responden a la necesidad de procesar, tanto la entrada como la salida del algoritmo, de una forma tal que facilite su funcionamiento para el usuario final. En primer lugar, la librería Nominatim [16] permite que el usuario pueda ingresar como entrada una dirección en palabras, en vez de coordenadas numéricas, lo que facilita el uso del algoritmo 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 [11] 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.

3.2. Funcionamiento lógico

Con el fin de llegar al resultado esperado, Santiago estará representada como una red de capas. La primera vendrá entregada por OpenStreetMap mediante pyrosm, conformada por la infraestructura urbana de la ciudad (calles, edificios, etc.) La segunda provendrá de la información de transporte, proveniendo del GTFS del Directorio de Transporte Público Metropolitano, mediante pygtfs. De esta manera, el algoritmo entregará un conjunto de caminos en esa red, desde un punto de la infraestructura urbana hasta otro punto dentro de la misma, que pueden ser recorridos usando la red de transporte público.

Connection Scan Algorithm requiere de un punto de inicio y uno de destino como entrada para funcionar, y mediante estos calcula la ruta a seguir para efectuar el viaje. Sin embargo, luego de estudiar la estructura de GTFS y la información disponible para Santiago, se concluye que, además de considerar los puntos antes mencionados, el algoritmo debería obtener además una fecha y una hora de inicio del viaje, para entregar una respuesta que represente mejor la realidad del servicio disponible. Dado que no todos los recorridos funcionan en los mismos horarios, y algunos solo funcionan ciertos días de la semana, es provechoso tomar en consideración estas variables temporales. Los archivos que componen el GTFS provisto para Santiago incluyen un listado de días feriados, por lo que también se consideran como excepciones a los recorridos regulares.

La motivación del algoritmo, tal como fue discutida anteriormente, es obtener la 'mejor' ruta para llegar al destino. En este caso, se prioriza la ruta que posea menos transbordos, y dentro de estas, la ruta cuyo recorrido demore menos en llegar desde la hora de inicio. Por lo tanto, el algoritmo debe procesar las paradas cercanas a los puntos de inicio y destino, obtener los recorridos que pasan por ellas, y teniendo la información de sus rutas y tiempos de espera, generar la 'mejor' ruta y entregarla como salida al usuario.

Para poder realizar esto, el funcionamiento del algoritmo sería:

- El usuario corre el programa.
- El programa descarga la información de OpenStreetMap y la procesa, en conjunto con la información de GTFS, y almacena todo en grafos.
- Al correr el algoritmo, el usuario ingresa una fecha y hora para el inicio del viaje, una dirección de origen y una dirección de destino.
- El programa busca las direcciones para obtener sus coordenadas, en la capa de OpenStreetMap.
- Con las coordenadas obtenidas y la información de los paraderos, se buscan los más cercanos y los recorridos que se detienen en ellos.

- Se busca la existencia de un recorrido que conecte ambos puntos. De no existir, se buscan los paraderos en los que se detienen los recorridos que llegan al destino, para revisar si alguno de los recorridos que parten del origen terminan allí y es posible hacer un transbordo. De esta forma, se consigue la secuencia de vehículos a tomar.
- Con los recorridos necesarios obtenidos, se calculan los tiempos de viaje. Se obtiene el tiempo de espera para el primer bus y se calcula cuánto tiempo demora la ruta hasta llegar al destino.
- Finalmente, se entrega la salida del algoritmo. Con las coordenadas de los puntos de origen y destino, además de las coordenadas de los paraderos necesarios para efectuar la ruta, se grafican los pasos a seguir en un mapa. Por otro lado, se entrega el tiempo de viaje como texto.

Como nota aparte, la implicancia de la estructuración propuesta es que, de cambiar la información por la de una ciudad diferente, el algoritmo también sería capaz de entregar una respuesta válida. Esto quiere decir que el algoritmo deberá estar abierto a recibir datos de GTFS y de OpenStreetMap provenientes de cualquier lugar. Evidentemente, como el enfoque de este trabajo de título es estudiar la movilidad vial en Santiago de Chile, el resto de posibilidades escapa de su alcance, y no serán consideradas. Sin embargo, pueden sentar bases para realizar trabajo futuro.

3.3. Criterio de Evaluación

El usuario final del proyecto será cualquier individuo u organización que requiera información sobre cómo movilizarse en Santiago. Por ejemplo, una persona que desee saber la mejor forma de movilizarse desde Padre Hurtado hasta Vitacura, o una entidad gubernamental (como la DTPM) que requiera estudiar las rutas más utilizadas por la gente para repartirles los recursos correspondientes. Por esto, para asegurar el éxito del proyecto, el criterio que se usará para evaluar que el resultado del trabajo cumpla su objetivo, es:

- El usuario final deberá ser capaz de, haciendo uso del algoritmo propuesto como solución, identificar patrones de movilidad a través de Santiago de Chile.

Posterior a la implementación, se planifica realizar un caso de estudio para analizar la movilidad en Santiago. La finalidad es probar la efectividad y eficiencia de la solución desarrollada, y utilizar sus resultados para comparar la situación en la ciudad bajo diferentes condiciones. El caso de estudio definido consiste en estudiar las rutas disponibles en Santiago, antes y después de la implementación de la Línea 3 del Metro (la última en inaugurarse al momento del desarrollo de este trabajo de título). Para efectuarlo, se definirá una serie de puntos de origen y destino que actualmente puedan conectarse realizando viajes en la Línea 3, y se obtendrán tanto rutas que incluyan a la L3, como otras que la saquen de las posibilidades. Luego, se compararán los tiempos estimados de viaje, para así analizar qué tanto efecto tiene esta línea de Metro sobre los trayectos de las personas.

Capítulo 4

Implementación

En el presente capítulo, se detallará la implementación realizada de Connection Scan Algorithm 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 [19].

4.1. Obtención de datos

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 [26], tiene disponible para descarga la información de los distintos países del mundo, incluído Chile [27]. 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 [25], la librería utilizada para procesar la información, permite leer datos de OpenStreetMap en formato PBF e interpretarla en estructuras de GeoPandas [18], 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.

Los datos de OSM se obtienen mediante el método `get_network` de pyrosm, obteniendo así nodos y aristas. Esto se logra ejecutando el siguiente fragmento de código:

```
fp = get_data(
    "Santiago", # Nombre de la ciudad
    update=True,
    directory=OSM_PATH
```

```
)  
  
osm = OSM(fp)  
nodes, edges = osm.get_network(nodes=True)
```

Como fue mencionado anteriormente, los nodos representan tanto puntos de interés de una ciudad, así como intersecciones entre calles. Por su parte, las aristas representan las calles en sí.

Estos datos se almacenan en un grafo de graph-tool, siguiendo la misma estructura de nodos y aristas. A ambos grupos se les otorgan propiedades en el grafo para permitir acceder a su información de manera consistente. A los nodos, se les dan propiedades para almacenar sus coordenadas (una para su latitud, y otra para su longitud), además de propiedades para almacenar sus identificadores (un id interno del grafo, y su id otorgado en la estructura de OpenStreetMap). Por otro lado, las aristas tienen propiedades para marcar los nodos que conectan (una para el nodo de origen y otra para el nodo de destino), y una extra para representar la distancia que mide la arista. Iterando por todos los nodos y aristas existentes, se almacenan con sus propiedades respectivas, y termina la obtención de datos de OpenStreetMap.

Estudiando el grafo obtenido, se observó que este grafo es dirigido, es decir, sus aristas poseen una dirección fija y conectan los nodos siguiendo esta lógica. En el trabajo posterior, se observaron problemas para trabajar con el grafo dirigido, puesto que la búsqueda de paraderos cercanos a los puntos de origen y destino se dificultaba por esta razón. Al tener que moverse por las calles para llegar al paradero, las caminatas de transbordo se veían afectadas por la dirección de las aristas que conectaban los nodos, generando un comportamiento innecesario y complejo de sortear. Por este motivo, se decidió trabajar con una copia no dirigida del grafo original, poseyendo la misma estructura y propiedades, con la única diferencia de perder la dirección de las aristas. Así, se genera una versión no dirigida tomando como base el grafo inicial, recorriéndolo y copiando su estructura.

Naturalmente, sobre este grafo se gestionan las búsquedas de direcciones para los puntos de origen y destino. Para realizar esta gestión, se crea la función **address_locator**:

```
def address_locator(graph, address):  
    geolocator = Nominatim(user_agent="ayatori")  
    while True:  
        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:
    long, lati = location.longitude, location.latitude
    nearest = find_nearest_node(graph, lati, long)
    return nearest

msg = "Error: Address couldn't be found."
print(msg)

```

Esta función recibe el grafo creado con la información de OpenStreetMap (graph) y una dirección (address), suscribe un agente de geocodificación con un nombre (en este caso, 'ayatori'), e intenta buscar las coordenadas de la dirección. 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, la función previene este caso, intentando acceder al servicio 3 veces; si no está disponible, muestra el error correspondiente. Por otro lado, también puede existir el caso en el que la dirección no se encuentre (por ejemplo, por estar mal escrita), para cuyo caso también se gestiona el mensaje de error correspondiente. Si la dirección se logra encontrar, la función obtiene sus coordenadas, y luego entrega el nodo más cercano. La razón de esto es que la dirección buscada puede no coincidir exactamente con un nodo del grafo de OpenStreetMap, así que lo más correcto sería entregar el nodo más cercano. Esto se realiza con la función `find_nearest_node`:

```

def find_nearest_node(graph, latitude, longitude):

    query_point = np.array([longitude, latitude])

    # Obtains vertex properties: 'lon' and 'lat'
    lon_prop = graph.vertex_properties['lon']
    lat_prop = graph.vertex_properties['lat']

    # Calculates the euclidean distances between the node's coordinates and the
    # consulted address's coordinates
    distances = np.linalg.norm(np.vstack((lon_prop.a, lat_prop.a)).T -
                               query_point, axis=1)

    # Finds the nearest node's index
    nearest_node_index = np.argmin(distances)
    nearest_node = graph.vertex(nearest_node_index)

    return nearest_node

```

Así, se obtienen las coordenadas de los puntos de origen y destino, y se puede comenzar a calcular la mejor ruta para efectuar el viaje.

4.1.2. Procesamiento de 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 2.3. Toda la información viene en un archivo comprimido ZIP, descargado desde la web del DPTM [4]. Este archivo debe descargarse manualmente, y las versiones nuevas salen cada uno o dos meses. Sin embargo, las diferencias entre versiones suelen tener relativamente pocas diferencias.

Pygtfs [6], la librería utilizada para procesar los datos de GTFS, posee un módulo llamado **Schedule**, encargado de gestionar toda la información. Esto permite realizar lo siguiente:

```
# Create a new schedule object using a GTFS file
sched = pygtfs.Schedule(":memory:")
pygtfs.append_feed(sched, "gtfs.zip")
```

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 puede acceder a la información de cada archivo de texto como si fuese un método de *sched*. Por ejemplo, para obtener la información de las paradas desde 'stops.txt', se puede hacer **sched.stops**, y para obtener los servicios o rutas, se puede hacer **sched.routes**.

Al igual que en el caso anterior, estos datos se almacenan en un grafo de graph-tool. En este caso, se guardan en los nodos los identificadores de las paradas, y las aristas conectan las paradas entre sí, generando la red de transporte. La información de GTFS es leída para llenar este grafo, y además, un diccionario de información denominado **route_stops**, estructura que mapea los identificadores de rutas a la lista de identificadores de paradas, guardando información relevante como valor. La estructura se define en el código como muestra el siguiente fragmento:

```
route_stops[route.route_id][stop_id] = {
    "route_id": route.route_id,
    "stop_id": stop_id,
    "coordinates": stop_coords[route.route_id][stop_id],
    "orientation": "round" if orientation == "I" else "return",
    "sequence": sequence,
    "arrival_times": []
}
arrival_time = (datetime.min + stop_times[i].arrival_time).time()
```

Donde se ha iterado sobre cada ruta, viaje y parada, para obtener la información señalada. Así, se guardan los identificadores de ruta y parada, las coordenadas de la parada, la orientación que tiene la ruta al pasar por esa parada (es decir, si se trata de su viaje de ida o de su viaje de regreso), el número de la secuencia de la ruta correspondiente a la parada, y sus tiempos de llegada.

Utilizando el diccionario **route_stops**, se pueden crear variadas funciones que sean de utilidad para generar la ruta del viaje. Por ejemplo, **get_stop_ids**, para obtener las paradas

cercanas a un punto del mapa. La función se puede apreciar en el siguiente fragmento de código:

```
def get_stop_ids(route_stops, coords, margin):
    stop_ids = []
    orientations = []
    for route_id, stops in route_stops.items():
        for stop_info in stops.values():
            stop_coords = stop_info["coordinates"]
            distance = haversine(coords[1], coords[0], stop_coords[1],
                                  stop_coords[0])
            if distance <= margin:
                orientation = stop_info["orientation"]
                stop_ids.append(stop_info["stop_id"])
                orientations.append((stop_info["stop_id"], orientation))
    return stop_ids, orientations
```

Esta función recibe como entrada el diccionario en cuestión, una tupla de coordenadas, y un margen numérico. Iterando sobre los elementos del diccionario, obtiene las coordenadas de cada parada, y revisa si está a una distancia cercana de las coordenadas entregadas, cercanía dada por el margen entregado. Como se ve, esta distancia se calcula usando la función **haversine**, definida en el siguiente fragmento de código:

```
def haversine(lon1, lat1, lon2, lat2):
    R = 6372.8 # Earth radius in kilometers
    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)
    a = sin(dLat / 2)**2 + cos(lat1) * cos(lat2) * sin(dLon / 2)**2
    c = 2 * asin(sqrt(a))
    return R * c
```

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 siguiente fragmento de código:

```
def connection_finder(route_stops, stop_id_1, stop_id_2):
    connected_routes = []
    for route_id, stops in 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
```

Con esto, se puede realizar un uso correcto de los datos provistos por ambas capas de información para ejecutar el algoritmo. Si bien, en esta sección se omite el código completo, gran parte de este puede revisarse en el anexo B. Luego, queda analizar cómo utilizar estas herramientas para generar la salida del programa.

4.2. Generación del output

Tal como se mencionó en la sección 3.2 del capítulo de Diseño, el *output* del algoritmo está conformado por dos partes, explicadas a continuación.

4.2.1. Texto

La primera parte del output está en formato de texto, que consiste en las instrucciones que el usuario debe seguir para realizar la 'mejor ruta' entre su punto de origen y su punto de destino. Estas instrucciones incluyen los identificadores de los recorridos a tomar, los identificadores de los paraderos de subida y bajada, y el tiempo aproximado del viaje. El tiempo se calcula con la diferencia entre la hora a la que pasa el siguiente bus del recorrido deseado y la hora actual, a lo que se le agrega el tiempo aproximado que demorará en llegar al paradero de destino. Esto se puede realizar al leer y cruzar la información de dos archivos de GTFS: *stop_times.txt* y *frequencies.txt*, lo que permite saber los rangos de tiempo en los que opera cada servicio y la frecuencia de salida de nuevos buses dentro de estos, además del tiempo que demora un bus desde que inicia el recorrido y hasta que llega a una parada en específico. Las funciones *get_arrival_times*, *get_time_until_next_bus*, y *get_travel_time* se crean con esta finalidad, como se muestra en el siguiente fragmento:

```
def get_arrival_times(route_id, stop_id, source_date):
    # Read the frequencies.txt file
    frequencies = pd.read_csv("frequencies.txt")

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

    # Get the day suffix
    day_suffix = get_trip_day_suffix(source_date)

    # Get the arrival times for the stop for each trip
    stop_route_times = []
    bus_orientation = ""
    for _, row in route_frequencies.iterrows():
        start_time = pd.Timestamp(row["start_time"])
```

```

if row["end_time"] == "24:00:00":
    end_time = pd.Timestamp("23:59:59")
else:
    end_time = pd.Timestamp(row["end_time"])
headway_secs = row["headway_secs"]
round_trip_id = f"{route_id}-I-{day_suffix}"
return_trip_id = f"{route_id}-R-{day_suffix}"
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

def get_time_until_next_bus(arrival_times, source_hour, source_date):
    arrival_times_remaining = []
    for a_time in arrival_times:
        if a_time.time() >= source_hour:
            arrival_times_remaining.append(a_time)
#arrival_times_remaining = [time for time in arrival_times if time.time() >=
#                           source_hour]
    if len(arrival_times_remaining) == 0:
        return None
    else:
        # Sort the remaining arrival times in ascending order
        arrival_times_remaining.sort()

        # Get the datetime objects for the next three buses
        next_buses = []
        for i in range(min(3, len(arrival_times_remaining))):
            next_arrival_time = arrival_times_remaining[i]

```

```

        next_bus = datetime.combine(next_arrival_time.date(),
                                     next_arrival_time.time())
        next_buses.append(next_bus)

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

    return time_until_next_buses

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

```

Se destaca que uno de los detalles importantes a tener en cuenta a la hora de calcular los tiempos es el sentido u orientación del recorrido, pues de no considerar esto, se podría obtener un tiempo de llegada que no se apegue a la realidad. Cada recorrido del transporte público de Santiago posee un viaje de ida y un viaje de regreso, los cuales difieren en paradas y tiempos relativos para todos los buses (solo exceptuándose el Metro), por lo que no basta con buscar un recorrido que sirva para realizar un transbordo, sino que su orientación coincida con el viaje que se desea realizar. Desafortunadamente, no existe una columna en algún archivo que posea esta información directamente. Sin embargo, los identificadores del archivo *frequencies.txt* son *strings* compuestos que incluyen, además del identificador del recorrido y del tipo de viaje según el día de la semana, un identificador que señala la orientación del bus, pudiendo filtrar esta información correctamente.

Así, se puede procesar lo necesario para entregar esta parte de la salida del algoritmo: el usuario lee en pantalla el recorrido del bus que debe tomar, en qué parada subirse y en cuál bajarse. Adicionalmente, se le dice a qué hora pasa el próximo bus del recorrido, además de los dos posteriores a ese, junto al tiempo de viaje sobre el bus y el tiempo total que demorará el trayecto. A modo de ejemplo, en la siguiente imagen se muestra la salida del algoritmo, en la parte de texto, cuando se busca una ruta para ir desde la Facultad de Ciencias Físicas y Matemáticas hasta Puente Cal y Canto, a las 10:00 durante un día de semana:

To go from: Universidad de Chile (Facultad de Ciencias Físicas y Matemáticas), 850, Avenida Almirante Blanco Encalada, Barrio El Ejército, Santiago, Provincia de Santiago, Región Metropolitana de Santiago, 8370456, Chile
 To: Puente Cal y Canto, Avenida Cardenal José María Caro, Centro Histórico, Santiago, Provincia de Santiago, Región Metropolitana de Santiago, 8320012, Chile

The best option is to take the route 509 on stop PA451. The next bus arrives at 10:02.
 The other two next buses arrives in:
 3 minutes, 30 seconds (10:03)
 4 minutes, 50 seconds (10:04)

You will get off the bus on stop PA396 after 12 minutes and 20 seconds.
 Total travel time: 14 minutes, 40 seconds. You will arrive your destination at 10:14.

Figura 4.1: Ejemplo de salida del algoritmo (texto). Ruta entre FCFM y Puente Cal y Canto.

Se puede apreciar cómo se señala que la mejor opción es subirse al recorrido 509 en la parada PA451 hasta bajarse en la parada PA396, junto a los tiempos respectivos de viaje.

4.2.2. Mapa

La segunda parte de la salida consiste en el mapa interactivo que grafica los puntos relevantes de la ruta calculada. Este mapa debe ser un complemento para el texto, destacando los puntos de origen y destino, y las paradas pertinentes. De esta forma, el usuario final puede entender mejor la información otorgada.

Como fue mencionado anteriormente, los mapas son generados usando la librería folium. En la siguiente figura, se muestra un mapa de Santiago generado con esta librería:

```
santiago_coords = [-33.4489, -70.6693]
m = folium.Map(location=santiago_coords, zoom_start=12)
display(m)
```

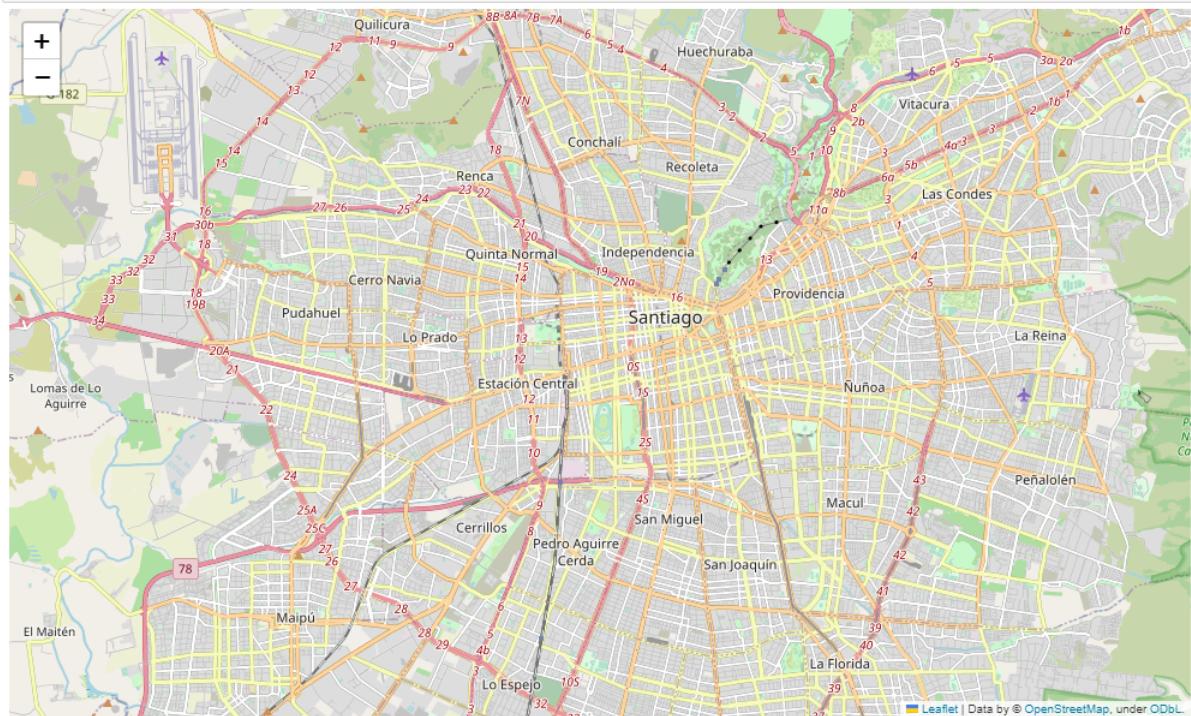


Figura 4.2: Mapa de Santiago, generado con folium.

El método **folium.Map** crea el mapa, centrado en una locación dada por una tupla de coordenadas (en este caso, Santiago), y con un nivel configurable de zoom en la imagen. Al ser interactivo, este nivel de zoom es modifiable utilizando los botones de la esquina superior izquierda o la rueda del mouse, mientras que se puede cambiar el foco al clickear y arrastrar.

Para destacar los puntos importantes, folium posee métodos como **folium.Marker**, que agrega un marcador a una ubicación específica. Este es configurable estéticamente con respecto al color e ícono, y además se le puede dar un mensaje para mostrar al clickear el marcador, como **popup**. En este caso, lo que se muestra para los puntos de inicio y destino corresponde a la información completa de la dirección buscada, proveniente de Nominatim. Por otro lado, para las paradas, se señalan los identificadores del recorrido y la parada en cuestión.

Usando estas herramientas en conjunto, se puede generar el mapa correctamente para entregarlo como salida del algoritmo. La función creada para gestionar la salida, tanto el texto como el mapa, se llama **create_transport_map**. El código completo se puede revisar en el anexo B, pero en el siguiente fragmento se señalan las partes correspondientes a la generación del mapa:

```
# 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):
    ...
    # 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
    ...

    for stop_id in near_source_stops:
        if stop_id in valid_source_stops:
            ...
            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 service in valid_target:
    if service == best_option[0]:
        ...
        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)
...
# Set the optimal zoom level for the map
fit_bounds(selected_path, m)

return m

```

De esta forma, se procesa la información para generar el mapa, señalando los puntos relevantes de la ruta. A modo de ejemplo, continuando lo mostrando en la subsección anterior, en la siguiente imagen se muestra la salida del algoritmo, en la parte del mapa, al realizar la misma ruta entre la FCFM y Puente Cal y Canto:

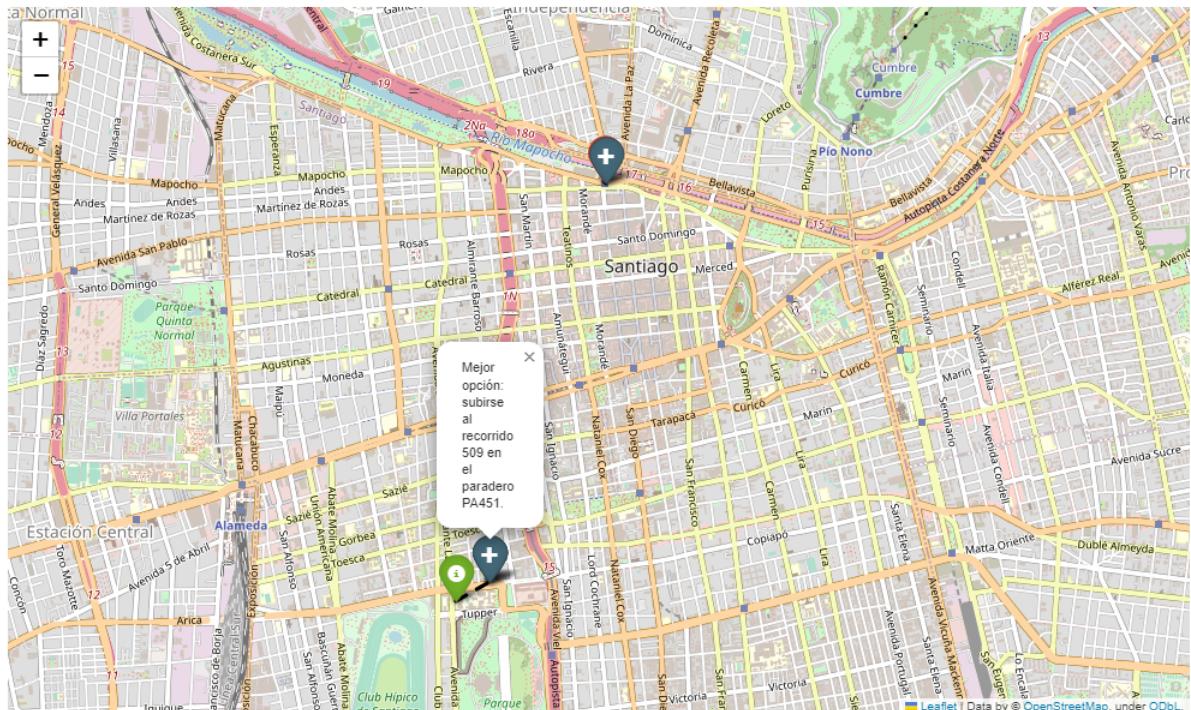


Figura 4.3: Ejemplo de salida del algoritmo (mapa). Ruta entre FCFM y Puente Cal y Canto.

Capítulo 5

Resultados

En el presente capítulo, se describen los resultados obtenidos al terminar la implementación del algoritmo.

5.1. Usabilidad del programa

Al correr el programa, el algoritmo le solicita al usuario que ingrese los inputs para las 4 entradas del algoritmo; en orden, la fecha, la hora, la dirección de origen, y la dirección de destino. El programa permite ingresar valores por defecto, que consisten en la fecha y hora actuales del sistema, el Campus Beauchef de la Universidad de Chile como punto de origen, y el Campus Antumapu de la Universidad de Chile como punto de destino. La ventana de salida mostrará el estado actual del programa, imprimiendo mensajes cuando se encuentren las direcciones, cuando se esté calculando la mejor ruta, y el resultado final, incluyendo los paraderos de origen, de destino, los recorridos a tomar, y el tiempo final del viaje. Además, se mostrará el mapa interactivo que graficará la ruta.

En la siguiente figura, se muestra como ejemplo la salida del algoritmo al buscar la mejor ruta en un viaje entre dos puntos de la Alameda (Av. Libertador Bernardo O'Higgins #5121 y Av. Libertador Bernardo O'Higgins #1460), a realizarse el 16/07/2023 a las 01:00:

```

Enter the travel's date, in DD/MM/YYYY format (press Enter to use today's date) :
16/07/2023
Enter the travel's start time, in HH:MM:SS format (press Enter to start now) : 01:00:00
01:00:00
Enter the starting point's address, in 'Street #No, Province' format (Ex: 'Beauchef 850, Santiago'):avenida libertador bernardo o'higgins 5121, santiago
Enter the ending point's address, in 'Street #No, Province' format (Ex: 'Campus Antumapu Universidad de Chile, Santiago'):avenida libertador bernardo o'higgins 1460, santiago
Both addresses have been found.
Processing...

```

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

```

To go from: 5119, Avenida Libertador Bernardo O'Higgins, Estación Central, Provincia de Santiago, Región Metropolitana de Santiago, 9190847, Chile
To: @ElectoralPDC, 1460, Avenida Libertador Bernardo O'Higgins, Barrio Cívico, Santiago, Provincia de Santiago, Región Metropolitana de Santiago, 8330164, Chile
The best option is to take the route 401 on stop PI400. The next bus arrives at 01:09.
The other two next buses arrives in:
39 minutes, 30 seconds (01:39)
69 minutes, 30 seconds (02:09)
DEBUG
40 50

```

You will get off the bus on stop PA338 after 13 minutes and 9 seconds.
Total travel time: 22 minutes, 39 seconds. You will arrive your destination at 01:22.

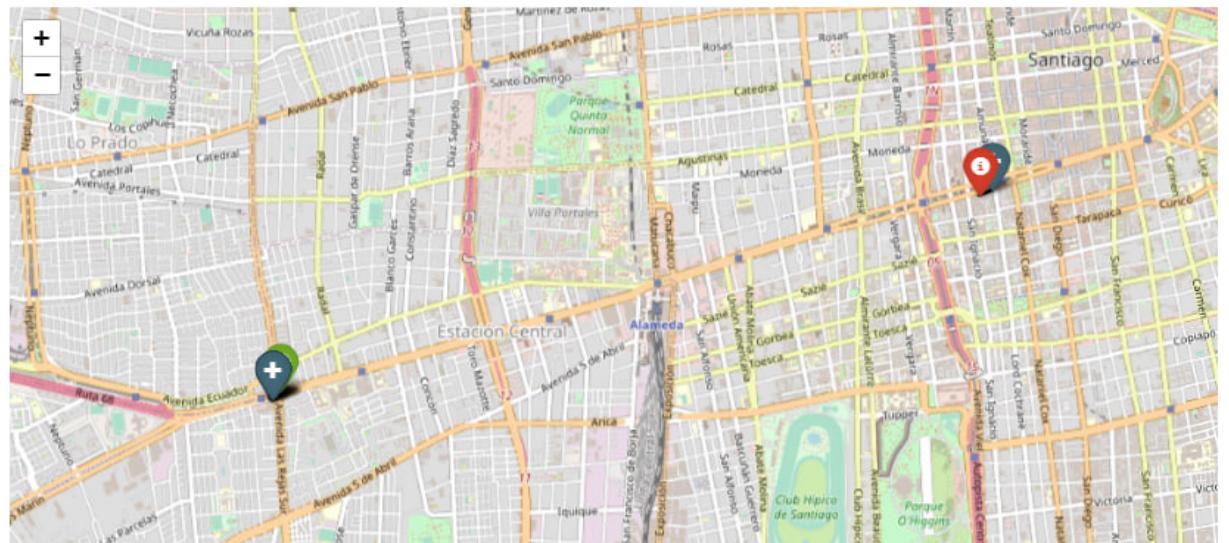


Figura 5.1: Resultado de la implementación del algoritmo: viaje de domingo de madrugada por la Alameda.

Los marcadores del mapa son clickeables. El verde representa la posición de origen, y el mensaje popup entrega la dirección. El rojo, por su parte, representa la posición de destino, y también entrega la dirección en el mensaje popup. Los marcadores grises representan los paraderos de subida y bajada del recorrido, y su mensaje popup muestra los identificadores de parada y recorrido respectivos.

5.2. Caso de estudio

Sección a completar...

5.3. Evaluación de resultados

Sección a completar...

Capítulo 6

Discusión

En este capítulo, habiendo ya presentado los resultados obtenidos, se procede a realizar una discusión sobre las implicancias y limitaciones del proyecto.

6.1. Implicancias

Sección a completar...

6.2. Limitaciones

Sección a completar...

6.3. Trabajo Futuro

Sección a completar...

Capítulo 7

Conclusión

El trabajo de título tuvo por objetivo generar una implementación de Connection Scan Algorithm en Python, con la finalidad de ser utilizado para obtener rutas entre dos puntos de Santiago, siguiendo una secuencia de transbordos de servicios del transporte público. Su motivación fue el problema de estudiar la movilidad vial en una ciudad, algo bastante más crítico de lo que parece a primera vista. Si bien existen múltiples herramientas para cumplir este objetivo, una de sus carencias comunes es la rápida obsolescencia de la información entregada, dado que al momento de terminar de procesar y analizar los datos obtenidos, suele ocurrir que se hayan generado cambios en el entorno que generen un sesgo negativo, como un cambio en la ruta de un servicio de autobús o la apertura de nuevas estaciones de Metro.

El resultado obtenido fue una implementación adecuada de CSA, cumpliendo así el objetivo principal del proyecto. La ruta obtenida como salida del algoritmo optimiza la cantidad de trasbordos a realizar, entregando así la mejor secuencia disponible. Dado que la implementación considera la información cartográfica actual de Santiago (proveniente de OpenStreetMap), así como las especificaciones vigentes del transporte público (en formato GTFS y provistas por el Directorio de Transporte Público Metropolitano), se necesita estar constantemente actualizando la alimentación del algoritmo para que entregue una ruta válida y actualizada; sin embargo, esta acción se puede hacer de forma sencilla, y permite la continuidad operativa de esta implementación. Aun así, dado que el algoritmo se alimenta de información estática (porque no recibe datos en vivo), al momento de ocurrir una eventualidad o emergencia, la implementación podría quedar igualmente sesgada bajo un supuesto de normalidad permanente en el estado del servicio. Esta línea de pensamiento puede ser utilizada para motivar trabajo futuro en esta área, mejorando la implementación para obtener información en directo del transporte público disponible.

En definitiva, esta implementación de Connection Scan Algorithm en Python representa un paso significativo hacia la mejora de la planificación de rutas de transporte público en Santiago. La información generada por el algoritmo puede aportar datos concretos a las organizaciones encargadas de administrar los servicios de transporte, abriendo nuevas posibilidades para el estudio y análisis de la movilidad urbana. Finalmente, la capacidad de adaptación y actualización sencilla del algoritmo sienta las bases para futuras investigaciones y desarrollos en busca de una movilidad más eficiente y resiliente en la ciudad.

Bibliografía

- [1] Geoff Boeing. Osmnx. Documentación disponible en <https://osmnx.readthedocs.io/en/stable/>. Revisado el 2023/03/07.
- [2] Bicineta Chile. Mapa de Ciclovías de la Región Metropolitana. Disponible en <https://www.bicineta.cl/ciclovias>. Revisado el 2023/03/07.
- [3] Fundación OpenStreetMap Chile. Mapa de OpenStreetMap Chile. Información disponible en <https://www.openstreetmap.cl>. Revisado el 2023/03/07.
- [4] GTFS Community. General Transit Feed Specification. Disponible en <https://gtfs.org>. Revisado el 2023/06/28.
- [5] Richard Darst. gtfspy. Repositorio disponible en <https://github.com/CxAalto/gtfspy>. Revisado el 2023/03/07.
- [6] Yaron de Leeuw. pygtfs. Repositorio disponible en <https://github.com/jarondl/pygtfs>. Revisado el 2023/06/29.
- [7] Tiago de Paula Peixoto. graph-tool: Efficient network analysis with python. Documentación disponible en <https://graph-tool.skewed.de>. Revisado el 2023/07/17.
- [8] Directorio de Transporte Público Metropolitano. GTFS Vigente. Disponible en <https://www.dtpm.cl/index.php/gtfs-vigente>. Revisado el 2023/03/07. Última versión: 2023/02/27.
- [9] NetworkX developers. Networkx - network analysis in python. Documentación disponible en <https://networkx.org>. Revisado el 2023/03/07. Última versión: 2023/01/07.
- [10] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithms*, 23(1.7):1–56, 2018.
- [11] Filipe Fernandes. Folium. Repositorio disponible en <https://github.com/python-visualization/folium>. Revisado el 2023/07/17.
- [12] OpenStreetMap (Global). Mapa de OpenStreetMap. Información disponible en <https://www.openstreetmap.org>. Revisado el 2023/03/07.
- [13] Google. Google Maps. Disponible en <https://www.google.com/maps/>.
- [14] Eduardo Graells-Garrido. Aves: Análisis y Visualización, Educación y Soporte. Repositorio disponible en <https://github.com/zorzaletalerrante/aves>. Revisado el 2023/03/07.
- [15] Eduardo Graells-Garrido, Daniela Opitz, and Francisco Rowe. A Generalisable Data Fusion Framework to Infer Mode of Transport Using Mobile Phone Data. Paper presentado para su publicación, 2022.

- [16] Sarah Hoffmann. Nominatim. Disponible en <https://nominatim.org>. Revisado el 2023/07/17.
- [17] Universidad Alberto Hurtado. Actualización y recolección de información del sistema de transporte urbano, IX Etapa: Encuesta Origen Destino Santiago 2012. Encuesta origen destino de viajes 2012. Disponible en <http://www.sectra.gob.cl/biblioteca/detalle1.asp?mf=3253> (2012). Revisado el 2023/03/07. Última versión lanzada el 2014.
- [18] Kelsey Jordahl. Geopandas. Documentación disponible en <https://geopandas.org>. Revisado el 2023/03/07. Última versión: 2022/12/10.
- [19] Felipe Leal. CC6909-Ayatori (Repositorio del Trabajo de Título). Repositorio disponible en <https://github.com/Lysorek/CC6909-Ayatori>. Revisado el 2023/03/07.
- [20] Microsoft. Windows subsystem for linux. Documentación disponible en <https://learn.microsoft.com/en-us/windows/wsl/>. Revisado el 2023/07/17.
- [21] Linus Norton. Connection Scan Algorithm (implementación en TypeScript). Repositorio disponible en <https://github.com/planarnetwork/connection-scan-algorithm>. Revisado el 2023/06/29.
- [22] Data Reportal. Digital 2021 Report for Chile. Disponible en <https://datareportal.com/reports/digital-2021-chile> (2021/02/11). Revisado el 2023/03/07.
- [23] Audrey Roy and Cookiecutter community. Cookiecutter: Better project templates. Documentación disponible en <https://cookiecutter.readthedocs.io/en/stable/>. Revisado el 2023/03/07.
- [24] Jonas Sauer. ULTRA: UnLimited TRAnsfers for Multimodal Route Planning. Repositorio disponible en <https://github.com/kit-algo/ULTRA>. Revisado el 2023/03/07.
- [25] 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.
- [26] Jochen Topf and Frederik Ramm. Geofabrik. Disponible en <https://www.geofabrik.de>. Revisado el 2023/03/07.
- [27] 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.
- [28] 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

    def _scan_footpaths(self, stop_id, walk_departure_time):
        """
        Scan the footpaths originating from stop_id

        Parameters
        -----
        stop_id: int
        """
        for _, neighbor, data in self._walk_network.edges(nbunch=[stop_id],
                                                       data=True):
            d_walk = data["d_walk"]
            arrival_time = walk_departure_time + d_walk / self._walk_speed
            self._update_stop_label(neighbor, arrival_time)

    class ConnectionScanProfiler(AbstractRoutingAlgorithm):
        """
        Implementation of the profile connection scan algorithm presented in
        http://i11www.iti.uni-karlsruhe.de/extra/publications/dpsw-isftr-13.pdf
        """

        def __init__(self,
                     transit_events,
                     target_stop,
                     start_time=None,
                     end_time=None,
                     transfer_margin=0,
                     walk_network=None,
                     walk_speed=1.5,
                     verbose=False):
        """
        Parameters
        -----
        transit_events: list[Connection]
            events are assumed to be ordered in DECREASING departure_time (!)
        target_stop: int
            index of the target stop
        start_time : int, optional
            start time in unixtime seconds
        end_time: int, optional

```

```

        end time in unixtime seconds (no connections will be scanned after
        this time)
    transfer_margin: int, optional
        required extra margin required for transfers in seconds
    walk_speed: float, optional
        walking speed between stops in meters / second.
    walk_network: networkx.Graph, optional
        each edge should have the walking distance as a data attribute
        ("distance_shape") expressed in meters
    verbose: boolean, optional
        whether to print out progress
    """
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(
            connection.arrival_time, self._transfer_margin
        )

    # 2. earliest arrival time within same trip (equals float('inf') if
        # not reachable)
    earliest_arrival_time_via_same_trip =
        self.__trip_min_arrival_time[connection.trip_id]

    # then, take the minimum (or the Pareto-optimal set) of these three
        # alternatives.
    min_arrival_time = min(earliest_arrival_time_via_same_trip,
                           earliest_arrival_time_via_transfer)

    # If there are no 'labels' to progress, nothing needs to be done.
    if min_arrival_time == float("inf"):
        continue

    # Update information for the trip
    if earliest_arrival_time_via_same_trip > min_arrival_time:
        self.__trip_min_arrival_time[connection.trip_id] =
            earliest_arrival_time_via_transfer

    # Compute the new "best" pareto_tuple possible (later: merge the sets
        # of pareto-optimal labels)
    pareto_tuple = LabelTimeSimple(connection.departure_time,
                                   min_arrival_time)

    # update departure stop profile (later: with the sets of
        # pareto-optimal labels)
    dep_stop_profile = self._stop_profiles[connection.departure_stop]
    updated_dep_stop =
        dep_stop_profile.update_pareto_optimal_tuples(pareto_tuple)

```

```

        # if the departure stop is updated, one also needs to scan the
        # footpaths from the departure stop
    if updated_dep_stop:
        self._scan_footpaths_to_departure_stop(connection.departure_stop,
                                                connection.departure_time,
                                                min_arrival_time)

    def _scan_footpaths_to_departure_stop(self, connection_dep_stop,
                                          connection_dep_time, arrival_time_target):
        """ A helper method for scanning the footpaths. Updates
            self._stop_profiles accordingly"""
        for _, neighbor, data in
            self._walk_network.edges(nbunch=[connection_dep_stop],
                                    data=True):
            d_walk = data['d_walk']
            neighbor_dep_time = connection_dep_time - d_walk / self._walk_speed
            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!");
        profiler.registerPhases({PHASE_CLEAR, PHASE_INITIALIZATION,
            PHASE_CONNECTION_SCAN});
        profiler.registerMetrics({METRIC_CONNECTIONS, METRIC_EDGES,
            METRIC_STOPS_BY_TRIP, METRIC_STOPS_BY_TRANSFER});
        profiler.initialize();
    }

    inline void run(const StopId source, const int departureTime, const StopId
        target = noStop) noexcept {
        profiler.start();

        profiler.startPhase();
        AssertMsg(data.isStop(source), "Source stop " << source << " is not a
            valid stop!");
        clear();
        profiler.donePhase(PHASE_CLEAR);

        profiler.startPhase();
        sourceStop = source;
        targetStop = target;
        arrivalTime[sourceStop] = departureTime;
        relaxEdges(sourceStop, departureTime);
        const ConnectionId firstConnection =
            firstReachableConnection(departureTime);
    }
}

```

```

    profiler.donePhase(PHASE_INITIALIZATION);

    profiler.startPhase();
    scanConnections(firstConnection, ConnectionId(data.connections.size()));
    profiler.donePhase(PHASE_CONNECTION_SCAN);

    profiler.done();
}

inline bool reachable(const StopId stop) const noexcept {
    return arrivalTime[stop] < never;
}

inline int getEarliestArrivalTime(const StopId stop) const noexcept {
    return arrivalTime[stop];
}

template<bool T = PathRetrieval, typename = std::enable_if_t<T ==
         PathRetrieval && T>>
inline Journey getJourney() const noexcept {
    return getJourney(targetStop);
}

template<bool T = PathRetrieval, typename = std::enable_if_t<T ==
         PathRetrieval && T>>
inline Journey getJourney(StopId stop) const noexcept {
    Journey journey;
    if (!reachable(stop)) return journey;
    while (stop != sourceStop) {
        const ParentLabel& label = parentLabel[stop];
        if (label.reachedByTransfer) {
            const int travelTime = data.transferGraph.get(TravelTime,
                label.transferId);
            journey.emplace_back(label.parent, stop, arrivalTime[stop] -
                travelTime, arrivalTime[stop], label.transferId);
        } else {
            journey.emplace_back(label.parent, stop,
                data.connections[tripReached[label.tripId]].departureTime,
                arrivalTime[stop], label.tripId);
        }
        stop = label.parent;
    }
    Vector::reverse(journey);
    return journey;
}

inline std::vector<Vertex> getPath(const StopId stop) const noexcept {
    return journeyToPath(getJourney(stop));
}

```

```

    inline std::vector<std::string> getRouteDescription(const StopId stop) const
        noexcept {
        return data.journeyToText(getJourney(stop));
    }

    inline const Profiler& getProfiler() const noexcept {
        return profiler;
    }

private:
    inline void clear() {
        sourceStop = noStop;
        targetStop = noStop;
        Vector::fill(arrivalTime, never);
        Vector::fill(tripReached, TripFlag());
        if constexpr (PathRetrieval) {
            Vector::fill(parentLabel, ParentLabel());
        }
    }

    inline ConnectionId firstReachableConnection(const int departureTime) const
        noexcept {
        return ConnectionId(Vector::lowerBound(data.connections, departureTime,
            [] (const Connection& connection, const int time) {
                return connection.departureTime < time;
            }));
    }

    inline void scanConnections(const ConnectionId begin, const ConnectionId end)
        noexcept {
        for (ConnectionId i = begin; i < end; i++) {
            const Connection& connection = data.connections[i];
            if (targetStop != noStop && connection.departureTime >
                arrivalTime[targetStop]) break;
            if (connectionIsReachable(connection, i)) {
                profiler.countMetric(METRIC_CONNECTIONS);
                arrivalByTrip(connection.arrivalStopId, connection.arrivalTime,
                    connection.tripId);
            }
        }
    }

    inline bool connectionIsReachableFromStop(const Connection& connection) const
        noexcept {
        return arrivalTime[connection.departureStopId] <=
            connection.departureTime -
            data.minTransferTime(connection.departureStopId);
    }
}

```

```

inline bool connectionIsReachableFromTrip(const Connection& connection) const
    noexcept {
    return tripReached[connection.tripId] != TripFlag();
}

inline bool connectionIsReachable(const Connection& connection, const
    ConnectionId id) noexcept {
    if (connectionIsReachableFromTrip(connection)) return true;
    if (connectionIsReachableFromStop(connection)) {
        if constexpr (PathRetrieval) {
            tripReached[connection.tripId] = id;
        } else {
            suppressUnusedParameterWarning(id);
            tripReached[connection.tripId] = true;
        }
        return true;
    }
    return false;
}

inline void arrivalByTrip(const StopId stop, const int time, const TripId
    trip) noexcept {
    if (arrivalTime[stop] <= time) return;
    profiler.countMetric(METRIC_STOPS_BY_TRIP);
    arrivalTime[stop] = time;
    if constexpr (PathRetrieval) {
        parentLabel[stop].parent =
            data.connections[tripReached[trip]].departureStopId;
        parentLabel[stop].reachedByTransfer = false;
        parentLabel[stop].tripId = trip;
    }
    relaxEdges(stop, time);
}

inline void relaxEdges(const StopId stop, const int time) noexcept {
    for (const Edge edge : data.transferGraph.edgesFrom(stop)) {
        profiler.countMetric(METRIC_EDGES);
        const StopId toStop = StopId(data.transferGraph.get(ToVertex, edge));
        const int newArrivalTime = time + data.transferGraph.get(TravelTime,
            edge);
        arrivalByTransfer(toStop, newArrivalTime, stop, edge);
    }
}

inline void arrivalByTransfer(const StopId stop, const int time, const StopId
    parent, const Edge edge) noexcept {
    if (arrivalTime[stop] <= time) return;
    profiler.countMetric(METRIC_STOPS_BY_TRANSFER);
    arrivalTime[stop] = time;
    if constexpr (PathRetrieval) {

```

```

        parentLabel[stop].parent = parent;
        parentLabel[stop].reachedByTransfer = true;
        parentLabel[stop].transferId = edge;
    }
}

private:
    const Data& data;

    StopId sourceStop;
    StopId targetStop;

    std::vector<TripFlag> tripReached;
    std::vector<int> arrivalTime;
    std::vector<ParentLabel> parentLabel;

    Profiler profiler;
};

}

```

Apéndice B

Código de la solución

B.1. Creación de grafos

B.1.1. OSM

```

def get_osm_data():
    """
    Obtains the required OpenStreetMap data using the 'pyrosm' library. This
    gives the map info of Santiago.

    Returns:
        graph: osm data converted to a graph
    """
    # 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...")

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

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

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

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

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

    # Calculate the distance between the nodes and use it as the weight of the edge
    source_coords = node_coords[source_node]
    target_coords = node_coords[target_node]
    distance = abs(source_coords[0] - target_coords[0]) +
               abs(source_coords[1] - target_coords[1])

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

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

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

# OSM Graph
osm_graph = get_osm_data()

def make_undirected(graph):

```

```

"""
Given a directed graph, returns an undirected version of the graph.

Parameters:
graph (Graph): A directed graph. In this specific case, the osm graph.

Returns:
Graph: An undirected version of the graph.
"""

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

    source_vertex = undirected_vertex_map[source_node]
    target_vertex = undirected_vertex_map[target_node]

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

    e = undirected_graph.add_edge(source_vertex, target_vertex)
    u_prop[e] = source_node
    v_prop[e] = target_node
    length_prop[e] = lgt
    weight_prop[e] = wt

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

return undirected_graph

# Convertir el grafo en no dirigido
undirected_graph = make_undirected(osm_graph)

```

B.1.2. GTFS

```

def get_gtfs_data():
    """
    Reads the GTFS data from a file and creates a directed graph with its info,
    using the 'pygtfs' library. This gives

```

the transit feed data of Santiago's public transport, including "Red Metropolitana de Movilidad" (previously known as Transantiago), "Metro de Santiago", "EFE Trenes de Chile", and "Buses de Acercamiento Aeropuerto".

Returns:

- graphs: GTFS data converted to a dictionary of graphs, one per route.
- route_stops: Dictionary containing the stops for each route.
- special_dates: List of special calendar dates.

```
"""  
# Create a new schedule object using a GTFS file  
sched = pygtfs.Schedule(":memory:")  
pygtfs.append_feed(sched, "gtfs.zip")  
  
# Get special calendar dates  
special_dates = []  
for cal_date in sched.service_exceptions: # Calendar_dates is renamed in  
    pygtfs  
    special_dates.append(cal_date.date.strftime("%d/%m/%Y"))  
  
# 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"
            }

```

```

        "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]
            elif source_orientation == target_orientation and service not in
                  valid_services: # Check if both stops are visited in the same
                  orientation
                valid_services.add(service)
                fixed_orientation = target_orientation
            elif isinstance(source_orientation, list) and target_orientation
                  in source_orientation and service not in valid_services:
                valid_services.add(service)
                fixed_orientation = target_orientation
            elif isinstance(target_orientation, list) and source_orientation
                  in target_orientation and service not in valid_services:
                valid_services.add(service)
                fixed_orientation = source_orientation

#print("ORIENTATION FIXED: {}".format(fixed_orientation))

if len(valid_services) == 0:
    print("Error: There are no available services right now to go to the
          desired destination.")
    print("Possible reasons: no routes that have stops near the source and
          target addresses.")
    print("You can try changing the search margin and try again.")

```

```

        return

nighttime_flag = is_nighttime(source_hour)
rush_hour_flag = is_rush_hour(source_hour)
holiday_flag = is_holiday(source_date)
if holiday_flag:
    rush_hour_flag = 0

# Nighttime check
daily_time_services = check_night_routes(valid_services, nighttime_flag)

if daily_time_services is None:
    print("Error: There are no available services right now to go to the
          desired destination.")
    print("Possible reasons: Source hour is during nighttime.")
    print("Please take into account that nighttime goes between 00:00:00 and
          05:30:00.")
    return

# Rush hour check
valid_services = check_express_routes(daily_time_services, rush_hour_flag)

valid_services = list(set(valid_services))

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])]
valid_source_stops = list(set(valid_source_stops))
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])]
valid_target_stops = list(set(valid_target_stops))

# Give info
print("")
print("Routes have been found.")
print("Calculating the best route and getting the arrival times for the next
      buses...")

best_option = None
best_option_times = None
source_time = timedelta(hours=source_hour.hour, minutes=source_hour.minute,
                        seconds=source_hour.second)

valid_orientations = set(source_orientations)

best_option_orientation = None

valid_target = []
for target_stop in valid_target_stops:

```

```

target_routes = get_routes_at_stop(route_stops, target_stop)
valid_target.extend(target_routes)
valid_target = list(dict.fromkeys(valid_target))

for stop_id in valid_source_stops:
    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 valid_service in valid_stop_services:
        a_t = get_arrival_times(valid_service, stop_id, source_date)
        if a_t is not None and a_t[0] == fixed_orientation:
            orientation = a_t[0]
            flag = False
            for target_stop_id in valid_target_stops:
                flag = False
                target_stop_routes = get_routes_at_stop(route_stops,
                                                       target_stop_id)
                if valid_service in target_stop_routes:
                    target_orientation = get_bus_orientation(valid_service,
                                                               target_stop_id)
                    if a_t[0] != target_orientation:
                        flag = True
                        continue

                if flag:
                    continue

            if valid_service not in valid_target:
                continue

            arrival_times = a_t[1]
            #print(arrival_times)
            time_until_next_buses = get_time_until_next_bus(arrival_times,
                                                             source_hour, source_date)

            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.

```

Returns:

```

list: the list of coordinates of the travel connections needed to arrive
      at the destination.
"""

```

```

node_id_mapping = create_node_id_mapping(graph)

```

```

source_node = address_locator(graph, source_address)
target_node = address_locator(graph, target_address)

```

```

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

    print("Both addresses have been found.")
    print("Processing...")
    #print("SOURCE NODE: {}. TARGET NODE: {}".format(source_node_graph_id,
    #                                                 target_node_graph_id))

    path = [source_node_graph_id, target_node_graph_id]
    path_coords = []
    for node in path:
        lon, lat = graph.vertex_properties["lon"][node],
                   graph.vertex_properties["lat"][node]
        path_coords.append((lat, lon))
    #print(path)

    return path_coords
else:
    return

```



```

def csa_commands():
    """
    Process the inputs given by the user to run the Connection Scan Algorithm.
    """

    # System's date and time
    now = datetime.now()
    dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
    #print("Fecha y hora actuales =", dt_string)

    # Date formatting
    today = date.today()
    today_format = today.strftime("%d/%m/%Y")

    # Time formatting
    moment = now.strftime("%H:%M:%S")
    used_time = datetime.strptime(moment, "%H:%M:%S").time()

    # User inputs
    # Date and time
    source_date = input(
        "Enter the travel's date, in DD/MM/YYYY 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()

```
