Universidad de Cantabria

Facultad
de
Ciencias

Generador de rutas de montaña
(Mountain route generator)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Rodrigo Fernández Silió

Director: Esteban Stafford Fernández

Septiembre - 2023

Resumen

En este Trabajo de Fin de Grado, se ha desarrollado un sistema de generación de rutas que consta de dos componentes principales: el primero genera un grafo a partir de datos geográficos, y el segundo calcula rutas basadas en ese grafo y una lista de puntos proporcionados por el usuario.

El sistema desarrollado resuelve uno de los problemas más comunes encontrados en aplicaciones de mapas convencionales, como Google Maps, donde la falta de información detallada sobre senderos y caminos rurales suele ser un impedimento para diseñar rutas a través de estos lugares.

Este sistema ofrece una solución eficaz para abordar este problema. La clave está en permitir a los usuarios utilizar pistas que han sido registradas previamente, ya sea por ellos mismos o por otros usuarios, que recorren estás ubicaciones menos conocidas, y que posibilitan crear rutas que atraviesan estos lugares.

El primer componente es responsable de la creación del grafo a partir de las pistas contenidas en un conjunto de archivos GPX. Este proceso implica la identificación de puntos de intersección entre las pistas, donde diferentes rutas se cruzan o se conectan. Estos puntos de intersección se incorporan al grafo, permitiendo que durante el cálculo de la ruta, se pueda transitar de una pista a otra de manera fluida. Esto posibilita la creación de rutas que combinan segmentos de varias pistas, lo que resulta en itinerarios tremendamente variados.

El segundo se encarga de calcular rutas a partir del grafo y de una serie de puntos proporcionados por el usuario. Los usuarios pueden introducir solo dos puntos, marcando el inicio y el fin de la ruta o añadir más de dos puntos, siendo los intermedios puntos de paso de la ruta. Además, para satisfacer las preferencias individuales de los usuarios, estos pueden optar por una ruta de mínima distancia o de mínimo desnivel acumulado. Esta flexibilidad en el cálculo de rutas garantiza que el sistema se adapte a las necesidades específicas de cada usuario, lo que contribuye significativamente a su satisfacción.

Durante el desarrollo de este trabajo se ha prestado gran atención a la complejidad de los algoritmos utilizados con el objetivo de garantizar un rendimiento eficiente y una respuesta rápida del sistema.

En resumen, este Trabajo Fin de Grado presenta un sistema de generación de rutas que supera las limitaciones de las aplicaciones de mapas convencionales, permitiendo a los usuarios crear itinerarios personalizados a través, sobre todo, de senderos y caminos rurales mediante la utilización de pistas ya grabadas.

Palabras clave: Generación de grafos, Cálculo de rutas, GPX, Python.

Abstract

In this Bachelor's Thesis, a route generation system has been developed, consisting of two main components: the first one generates a graph from geographical data, and the second one calculates routes based on that graph and a list of points provided by the user.

The developed system solves one of the most common problems found in conventional mapping applications like Google Maps, where the lack of detailed information on trails and rural roads often an impediment to designing routes through these places.

This system offers an effective solution to address this problem. The key is to allow users to use tracks that have been previously recorded, either by themselves or by other users, that travel through these less known locations, and that make it possible to create routes that cross these places.

The first component is responsible for creating the graph from the tracks contained in a set of GPX files. This process involves identifying intersection points between tracks, where different routes intersect or connect. These intersection points are incorporated into the graph, allowing to smoothly transition from one track to another during route calculation. This enables the creation of routes that combine segments from various tracks, resulting in highly diverse itineraries.

The second component is responsible for calculating routes based on the graph and a series of points provided by the user. Users can input just two points, marking the start and end of the route, or add more than two points, with the intermediates serving as waypoints along the route. Furthermore, to satisfy the individual preferences of users, they can opt for a route of minimum length or minimum accumulated gradient. This flexibility in route calculation ensures that the system adapts to the specific needs of each user, which contributes significantly to user satisfaction.

Throughout the development of this work, great attention has been paid to the complexity of the algorithms used with the aim of ensuring efficient performance and a fast system response.

In summary, this Bachelor's Thesis presents a route generation system that surpasses the limitations of conventional mapping applications, allowing users to create customized itineraries, primarily through trails and rural roads, using pre-recorded tracks.

Keywords: Graph generation, Route calculation, GPX, Python.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Motivación

Hoy en día no se concibe viajar sin el uso de algún dispositivo de navegación. Herramientas como Google Maps, Waze y otras aplicaciones de navegación han revolucionado la forma en que las personas se desplazan y exploran el mundo. Estas se basan en dos aspectos fundamentales: la capacidad de conocer la posición del usuario en cualquier momento y la capacidad de calcular rutas que guíen al usuario hacia su destino.

El primero se ha conseguido apoyándose en el Sistema de Posicionamiento Global (GPS), que proporciona estimaciones precisas de posición, velocidad y tiempo de un dispositivo en cualquier parte del mundo. Utiliza conjuntamente una red de ordenadores y una constelación de 24 satélites para determinar por triangulación, la altitud, longitud y latitud del dispositivo [8]. En cambio, el segundo aspecto se ha conseguido mediante el uso de bases de datos de mapas que contienen información sobre carreteras y algoritmos de cálculo de rutas que permiten encontrar la ruta más rápida y eficiente para llegar a un destino. Antes, planificar un viaje o una ruta requería la consulta de mapas físicos y guías de carreteras, lo que podía llevar mucho tiempo y esfuerzo. Ahora, con las herramientas de cálculo de rutas, este proceso es prácticamente instantáneo.

En este trabajo se aborda una mejora de este segundo aspecto, el cálculo de rutas. Si bien los servicios citados anteriormente dan información fundamentalmente sobre carreteras, en este trabajo se aborda el cálculo de rutas sobre caminos y senderos. Existe una gran cantidad de plataformas como Wikiloc, AllTrails, Strava, entre otras, para crear, compartir y descubrir rutas. Estas plataformas permiten a los usuarios grabar sus actividades y rutas, y compartirlas con otros usuarios. Sin embargo, estas plataformas no permiten a los usuarios crear rutas personalizadas a partir de rutas ya grabadas por otros usuarios. Simplemente pueden seleccionar una ruta existente y seguirla con la ayuda de su dispositivo GPS. Afortunadamente, estas plataformas suelen permitir descargar las rutas, y este es un aspecto clave para la concepción de este trabajo.

Un usuario puede entonces descargar una cantidad considerable de rutas. Si estas cubren una zona geográfica determinada de manera que representan todos los senderos de esta zona, se podría utilizar esta información para crear rutas nuevas a partir de estas. Aunque el usuario podría, de manera rudimentaria e ineficaz, componer visualmente la nueva ruta a partir de trozos de otras utilizando herramientas de representación de rutas, el objetivo principal de este trabajo es automatizar esta tarea, de manera que sea posible utilizar el conjunto de rutas descargadas como base de datos para crear rutas de manera similar a un generador de rutas de carreteras, como Google Maps. Así, un usuario podría indicar dos o más puntos y se calcularía un itinerario que los conecte a base de trozos de las rutas descargadas.

1.2. Objetivos

- Construir un grafo a partir de una serie de rutas en formato GPX:

  Los nodos del grafo estarán compuestos por el punto inicial y final de cada

pista, así como por los puntos de intersección entre las pistas. Estos puntos, esenciales para construir el grafo, serán referidos como *puntos relevantes*.

Las aristas del grafo representarán las conexiones entre los nodos. Para cada arista se registrará tanto el camino que une los extremos como la distancia y el desnivel acumulado a lo largo de dicho camino.

- Poder crear una ruta utilizando el grafo y una lista de puntos introducida por el usuario:

  Dado una lista de puntos se podrá utilizar el algoritmo de Dijkstra o el algoritmo A* para calcular la ruta de mínima distancia o mínimo desnivel acumulado.

1.3. Estructura de la memoria

Aquí se presenta la estructura general de la memoria, proporcionando una visión general de cómo se organizarán y presentarán los contenidos a lo largo del documento.

En la sección de "Introducción", se brinda una visión general de la motivación detrás del TFG, los objetivos del proyecto y una breve descripción de la estructura general de la memoria.

En la sección de "Tecnologías y Conceptos", se detallan las tecnologías y conceptos esenciales utilizados en el proyecto.

La sección de "Diseño" aborda los requisitos funcionales y no funcionales identificados, junto con la arquitectura general del sistema.

En la sección de "Implementación", se detalla la implementación del proyecto. Además, aquí se analiza la complejidad temporal de los algoritmos utilizados.

La sección de "Evaluación", se enfoca en las pruebas y evaluación del rendimiento del sistema.

Y en la sección de "Conclusiones y Trabajos Futuros", se ofrecen conclusiones generales, resumiendo los logros y desafíos. También se discute sobre posibles áreas de trabajo futuro relacionadas con el proyecto.

2. Tecnologías y conceptos

2.1. Grafos

Un grafo simple G consiste en un conjunto finito no vacío $V(G)$ de elementos llamados vértices o nodos, y un conjunto finito $E(G)$ de pares distintos no ordenados de elementos distintos de $V(G)$ llamados aristas [11].

Un camino en un grafo es una lista de nodos de forma que dos nodos consecutivos son adyacentes. Es decir, es una sucesión de nodos del grafo en la que cada nodo está conectado al siguiente mediante una arista.

Un ciclo es un camino donde el vértice de inicio y el final son iguales. Un grafo es cíclico si contiene al menos un ciclo en su estructura.

Dado un grafo a cuyas aristas se han asociado una serie de pesos, se define el camino de coste mínimo de un vértice u a otro v, como el camino donde la suma de los pesos de las aristas que lo forman es la más baja entre las de todos los caminos posibles de u a v.

El algoritmo de Dijkstra, diseñado por el holandés Edsger Wybe Dijkstra en 1959, es un algoritmo que sirve para encontrar el camino de coste mínimo desde un nodo origen a todos los demás nodos del grafo. [9]

Es un algoritmo eficiente, su complejidad temporal es $O(V^2)$ donde V es el número de vértices. Si el algoritmo se implementa utilizando una cola de prioridad su complejidad temporal es $O((E+V) \cdot log(V))$ siendo V el número de vértices y E el número de aristas. [1]

Vale la pena mencionar que el algoritmo de Dijkstra no funciona correctamente si el grafo contiene aristas con pesos negativos, ya que puede quedar atrapado en bucles infinitos. En este proyecto, esta limitación no será un problema porque como pesos se usa la distancia o el desnivel acumulado, que son siempre positivos, nunca negativos.

Aunque el algoritmo es eficiente, su tiempo de ejecución puede aumentar mucho en grafos grandes o densos.

En la búsqueda de un algoritmo más eficiente que el de Dijkstra aparece el algoritmo A*. Este algoritmo fue presentado en 1968 por Peter Hart, Nils Nilsson y Bertram Raphael. A* combina la búsqueda informada con la propiedad de "completitud" de la búsqueda en anchura.

Este algoritmo evalúa los nodos combinando $g(n)$, el costo para llegar al nodo n, y $h(n)$, el costo para llegar desde el nodo n hasta el nodo objetivo:

$$f(n) = g(n) + h(n)$$

Dado que $g(n)$ da el costo del camino desde el nodo inicial al nodo n, y $h(n)$ es el costo estimado del camino mínimo desde n hasta el nodo objetivo, tenemos

$$f(n) = \text{costo estimado del camino mínimo a través de n.}$$

Por lo tanto, si estamos tratando de encontrar el camino de coste mínimo, una opción razonable para probar primero es el nodo con el valor más bajo de $g(n) + h(n)$. Resulta que esta estrategia es más que razonable, siempre que la función heurística $h(n)$ satisfaga ciertas condiciones, la búsqueda A* es completa y óptima [7].

Una de las condiciones que se debe cumplir para que la búsqueda sea completa y óptima es que la heurística sea admisible, es decir, que nunca sobrestime el costo para alcanzar la meta.

Por ejemplo, para una aplicación de cálculo de rutas, como la de este proyecto, $h(x)$ podría representar la distancia en línea recta al objetivo, ya que es físicamente la distancia más pequeña posible entre dos puntos [5].

Aplicaciones como Google Maps utilizan tanto el algoritmo de Dijkstra como el algoritmo A* en su motor de enrutamiento para proporcionar direcciones y rutas a los usuarios.

2.2. Sistema de coordenadas geográficas

Las coordenadas geográficas son un sistema de referencia formado por un conjunto de líneas imaginarias que permiten ubicar con exactitud un lugar en la superficie de la Tierra. Estas coordenadas son representadas mediante dos valores principales: la latitud y la longitud.

La latitud de un punto en la superficie de la Tierra es el ángulo entre el plano ecuatorial y la línea que pasa por este punto y el centro de la Tierra.

La longitud de un punto en la superficie de la Tierra es el ángulo entre el meridiano de referencia, que es el meridiano de Greenwich, y el meridiano que pasa por este punto.

También existe una tercera referencia, la elevación, que es la distancia vertical que existe entre cualquier punto de la Tierra en relación con el nivel del mar. La indicación de la elevación exige la elección de un modelo esferoide que represente la Tierra.

El cálculo de distancias a partir de coordenadas geográficas se basa en la geometría esférica, ya que la Tierra es un objeto tridimensional y curvo. Existen varias fórmulas y métodos para calcular distancias geográficas.

Debido a la forma elipsoidal de la Tierra, la distancia en metros correspondiente a una diferencia de longitud varía en función de la latitud del lugar.

Esto ocurre porque los meridianos terrestres no son líneas rectas, sino que se curvan progresivamente desde el ecuador hasta coincidir en los polos. Como resultado, una diferencia de longitud en grados representará una distancia más larga en metros cerca del ecuador en comparación con las zonas cercanas a los polos.

Existen diversas fórmulas, como la fórmula de Haversine, también conocida como fórmula del semiverseno, que asume que la Tierra es una esfera perfecta y calcula la distancia ortodrómica, que es el camino más corto entre dos puntos de la superficie terrestre. Estas fórmulas, que utilizan un modelo esférico de la Tierra, presentan resultados algo inexactos.

Ya que la Tierra no es una esfera perfecta, sino que tiene una forma elipsoidal debido a su rotación y distribución de la masa, es mejor utilizar fórmulas que tengan en consideración esta característica. Algunas de ellas son la fórmula de Vincenty, la fórmula de Gauss-Legendre o la fórmula de Charles F. F. Karney presentada en su artículo *Algorithms for geodesics*

Para evitar cualquier mínimo error y conseguir la máxima precisión, en este trabajo se buscará utilizar una fórmula que tome en consideración esta propiedad de la Tierra.

2.3. Formato de archivo GPX

GPX es un formato de intercambio GPS, es un esquema XML diseñado como formato de datos GPS común para aplicaciones de software [4].

Un archivo GPX contiene metadatos, *waypoints* (puntos de referencia), *routes* (rutas) y *tracks* (pistas)

La sección de metadatos contiene información general sobre el archivo GPX, como el nombre del creador, la descripción, la fecha, etc.

Un waypoint representa un punto, un punto de interés o un elemento con nombre en un mapa. Contiene la latitud y longitud en grados, la elevación en metros y más información descriptiva. Esta información descriptiva puede ser un nombre si se trata de un punto de interés o una marca de tiempo que se registró durante la grabación.

Una ruta es una lista ordenada de waypoints que representan una serie de puntos de giro que conducen a un destino.

Una pista es una lista ordenada de puntos que describen un camino.

Una ruta y una pista pueden parecer lo mismo, pero no son iguales. La principal diferencia entre una ruta y una pista es que una ruta está compuesta por waypoints y segmentos para navegar por ella, pero no necesariamente tiene que coincidir con un recorrido real, mientras que una pista es una representación precisa de la ruta seguida por un usuario con todos los detalles y puntos capturados por un dispositivo GPS en tiempo real.

Para este proyecto, la información más relevante y fundamental son las pistas que se encuentran contenidas en cada archivo GPX, así como los puntos individuales que componen estas pistas.

2.4. gpx.studio

Interpretar directamente un archivo GPX en formato crudo es una tarea complicada. Por esta razón, es beneficioso utilizar una herramienta de visualización de archivos GPX que pueda transformar estos datos en información más comprensible y visualmente atractiva.

gpx.studio es un visor y editor GPX en línea gratuito. Como cualquier otro visor y editor GPX permite abrir, ver y editar archivos en formato GPX.

gpx.studio se destaca por su sencillez y, al mismo tiempo, por ofrecer una amplia gama de funciones. Algunas de estas características, que influyeron en la elección de gpx.studio, incluyen:

- La visualización de múltiples pistas en diferentes colores.
- La visualización de la longitud y el desnivel acumulado de las pistas.

2.5. Python

En este proyecto, se ha empleado Python como el lenguaje de programación. Python es un lenguaje de programación interpretado, orientado a objetos, de alto nivel y con semántica dinámica.

Python se destaca sobre otros lenguajes de programación en diversos aspectos, algunos de los cuales motivaron su elección para este proyecto, incluyendo:

- Sintaxis clara y legible: Python tiene una sintaxis clara y legible que se asemeja al lenguaje humano. Esto facilita la escritura y lectura del código, lo que conduce a una menor propensión a errores y a una mayor productividad.
- Amplia comunidad activa: Como resultado de su popularidad, Python tiene una abundante documentación y una comunidad en línea activa. Esto facilita encontrar ayuda y soluciones a problemas comunes.
- Multiplataforma: Python es compatible con una amplia variedad de plataformas, incluyendo Windows, macOS y diversas distribuciones de Linux. Esto facilita el desarrollo de aplicaciones que pueden ejecutarse en diferentes sistemas operativos sin problemas.
- Librerías y frameworks: Python cuenta con una amplia gama de bibliotecas y frameworks, que permiten acelerar el desarrollo de aplicaciones y proyectos.
- Código abierto: Python es un lenguaje de código abierto, lo que significa que es gratuito.

Las tareas principales de este proyecto son el procesamiento de datos geoespaciales, la construcción de grafos y el cálculo de rutas. Esto refuerza la elección de Python, ya que cuenta con una gran cantidad de librerías con métodos y funciones útiles para llevar a cabo estas tareas.

2.6. Librerías de Python relevantes

El uso de librerías es muy beneficioso para el desarrollo del proyecto. A continuación voy a exponer algunas de las ventajas que supone la utilización de librerías.

- Se evitan errores en el código porque estas están desarrolladas por profesionales y sus funcionalidades están sumamente probadas.
- Se mejora el rendimiento del programa, ya que el código está escrito por expertos y optimizado para un rendimiento óptimo.
- Se consigue un código más limpio y legible.
- Se ahorra tiempo y esfuerzo pues se reutiliza el código.

Algunas de las librerías que voy a utilizar no están incluidas en la instalación por defecto de Python, sino que se debe descargarlas e instalarlas manualmente.

Para facilitar esta tarea se ha elaborado un archivo "requirements.txt". En este archivo se escribe una lista de los paquetes y su versión que el proyecto requiere. Y con el comando `pip install -r requirements.txt` cualquier otro usuario podrá descargar e instalar todas las dependencias requeridas por el programa sin necesidad de buscar y descargar cada una individualmente.

A continuación, se listarán y explicarán las librerías más relevantes utilizadas en el proyecto.

2.6.1. Shapely

Shapely es una librería de Python para el análisis y manipulación de figuras geométricas planas mediante operaciones basadas en la teoría de conjuntos. Emplea funciones

de la conocida y ampliamente utilizada librería GEOS (Geometry Engine - Open Source), escrita en C++, de código abierto, y que proporciona funcionalidades avanzadas para el análisis geométrico y espacial [2].

Con respecto a las figuras geométricas planas. Estas son entidades que se representan en un plano 2D. Existen múltiples figuras geométricas planas, las más relevantes para este proyecto son el punto, la línea y el polígono.

Entre las funcionalidades que aporta la librería, las más esenciales para el proyecto, y por la que se decidió usar esta librería, son las operaciones espaciales de búfer e intersección.

La operación de búfer consiste en, dado una distancia o tamaño, crear un área o región alrededor de una geometría dada. Si se aplica a una línea el resultado es un polígono que rodea la línea original.

La intersección entre dos figuras geométricas planas es una operación espacial que determina la región común o superpuesta entre las dos figuras. En este proyecto se realiza mayoritariamente la intersección entre un polígono y una línea dando como resultado uno o más polígonos dependiendo de si existe una o múltiples regiones comunes.

2.6.2. GeoPy

GeoPy es una librería que proporciona herramientas para realizar operaciones relacionadas con la geolocalización y los cálculos geodésicos.

Dentro de la librería GeoPy se encuentra el módulo distance. Este es uno de los módulos más utilizados dentro de esta librería, y permite calcular la distancia entre dos puntos computando la distancia geodésica o la distancia ortodrómica.

Como se explicó en la sección 2.2, para calcular la distancia entre dos puntos y conseguir la máxima precisión se tiene que considerar la forma elipsoidal de la Tierra. El módulo distance proporciona la función geodesic, que calcula la distancia entre dos puntos utilizando la fórmula presentada por Charles F. F. Karney en su artículo *Algorithms for geodesics*. Tal como se indicó con anterioridad, esta fórmula tiene en cuenta la forma elipsoidal de la Tierra y proporciona resultados sumamente precisos.

2.6.3. NetworkX

También se ha utilizado la librería NetworkX, que permite la creación, manipulación y estudio de la estructura, dinámica y funciones de grafos complejos [3].

Ofrece una amplia variedad de opciones y funcionalidades para trabajar con grafos. Permite crear grafos, agregar y eliminar nodos y aristas, asignar información adicional a los nodos y aristas, etc. Además proporciona algoritmos de grafos como algoritmos de búsqueda del camino más corto, algoritmos de flujo máximo, algoritmos de conectividad, etc.

2.6.4. gpxpy

Para analizar y manipular archivos GPX se utiliza la librería gpxpy, que facilita la lectura y escritura de archivos GPX y proporciona una interfaz para acceder a los

datos contenidos en estos archivos.

gpxpy es la mejor elección para este proyecto, ya que es sencilla, fácil de usar y tiene todas las funcionalidades que este proyecto requiere. Además es una librería bastante conocida con una comunidad activa de desarrolladores y cuenta con una extensa documentación y una gran cantidad de ejemplos de uso.

2.6.5. json

Para trabajar con archivos con formato JSON (JavaScript Object Notation) se utiliza la librería json. Esta ya está integrada en Python y está disponible sin tener que descargar e instalar ninguna librería externa.

Esta librería ofrece dos funciones principales:

- dumps: Esta función se utiliza para serializar (convertir) objetos de Python en una cadena de texto en formato JSON. En este proyecto se utiliza para convertir un grafo en una cadena de texto en formato JSON.
- loads: Esta función se utiliza para deserializar (reconvertir) una cadena de texto en formato JSON en objetos de Python. En este proyecto se utiliza para reconvertir una cadena de texto en formato JSON a un grafo.

2.6.6. os

Para la lectura y escritura de archivos, incluyendo los archivos GPX y JSON se utiliza la librería os. Esta librería nos permite acceder a funcionalidades dependientes del sistema operativo. En este proyecto se utiliza para navegar por el sistema de archivos, así como para leer y escribir directorios y archivos.

2.6.7. argparse

Puesto que el programa recibe argumentos por línea de comandos, se utiliza también la librería argparse. Esta librería facilita la definición y el análisis de los argumentos de entrada. Además, proporciona una ayuda automática y es altamente personalizable. Encima ya está integrada en Python, lo que facilita su uso y despliegue.

2.7. Spyder

El desarrollo de código se realiza a través de Spyder, un entorno de desarrollo integrado (IDE). En esencia, un IDE es una aplicación de software que proporciona un conjunto completo de herramientas y funcionalidades para simplificar y mejorar el proceso de desarrollo de software.

Se ha optado por utilizar Spyder por varias razones: En primer lugar, Spyder proporciona un editor de código, una consola interactiva y herramientas de depuración en un solo lugar, lo que simplifica enormemente el flujo de trabajo. Otro aspecto crucial es que Spyder es una herramienta de código abierto, lo que significa que es gratuita. Además, cuenta con una función de autocompletado, que agiliza la escritura de código y ayuda a prevenir errores. Por último, mi familiaridad con Spyder es un factor determinante, al estar más acostumbrado a su interfaz y flujo de trabajo, me siento cómodo y productivo al utilizarlo.

2.8. Git

Durante la implementación se han aplicado diversas prácticas de ingeniería de software para garantizar la calidad del proyecto. Entre ellas destaca la gestión de versiones.

La gestión de versiones es un proceso esencial en el desarrollo de software que permite controlar y rastrear los cambios en el código fuente y otros archivos a medida que evolucionan con el tiempo. Esto facilita la colaboración en equipos de desarrollo, la restitución de versiones anteriores si es necesario y la documentación de cambios. La gestión de versiones se realiza comúnmente utilizando sistemas de control de versiones como Git, y este proyecto no es una excepción.

Git es un sistema de control de versiones ampliamente utilizado que permite rastrear y gestionar cambios en el código fuente y otros archivos de un proyecto de software. Fue desarrollado por Linus Torvalds en 2005.

Se ha elegido Git para este proyecto por algunos de los aspectos claves en los que destaca:

- Velocidad y eficiencia: Git es conocido por su velocidad y eficiencia en la gestión de proyectos de cualquier tamaño. Sus operaciones suelen ser muy rápidas, lo que facilita el flujo de trabajo diario.
- Historial detallado: Git almacena un historial completo y detallado de cada cambio realizado en el código, incluyendo información sobre cuándo se realizó y qué se modificó. Esto facilita la revisión de la historia del proyecto.
- Repositorios remotos: Git es compatible con repositorios remotos en servicios en línea como GitHub, GitLab y Bitbucket. Esto permite mantener fácilmente una copia de seguridad del proyecto.
- Amplia comunidad activa: Git cuenta con una comunidad de usuarios y desarrolladores muy activa y una amplia base de conocimientos en línea. Esto facilita la obtención de ayuda y la resolución de problemas.
- Código abierto: Git es de código abierto y completamente gratuito. Esto significa que cualquier persona puede descargar y utilizar Git sin costos asociados.

2.8.1. GitHub

Como se ha mencionado anteriormente, Git se integra perfectamente con repositorios remotos en línea, y el servicio más destacado en este sentido es GitHub. Este proyecto no es una excepción y se beneficia de las ventajas que ofrece GitHub.

GitHub proporciona una serie de herramientas y características diseñadas para simplificar la gestión de proyectos. GitHub permite almacenar el repositorio de código de manera segura en la nube, lo que actúa como un respaldo automático. Esto garantiza la protección del código contra la pérdida de datos en la máquina local

En resumen, utilizar GitHub en conjunto con Git amplía las capacidades de Git. GitHub agrega una capa de funcionalidades en línea que mejora significativamente la eficiencia y la calidad en el desarrollo del software.

A continuación, se encuentra el enlace al repositorio de GitHub, que proporciona acceso directo al código fuente y a los recursos relacionados utilizados en este TFG. Es una herramienta esencial para explorar la implementación y los resultados presentados en este trabajo.

Enlace al repositorio de GitHub

3. Diseño

3.1. Requisitos

La especificación de requisitos de software consiste en una descripción exhaustiva del funcionamiento previsto para el sistema que se va a desarrollar.

Los requisitos de un proyecto de software engloban las funciones, características y restricciones que debe cumplir el producto final. En esencia, los requisitos definen qué debe hacer el software, su apariencia visual y las condiciones que deben cumplirse para que se considere exitoso.

3.1.1. Requisitos funcionales

Los requisitos funcionales son declaraciones detalladas que describen las acciones específicas que un sistema o software debe llevar a cabo. Estos requisitos definen las funcionalidades y las operaciones que el sistema debe realizar para cumplir con los objetivos del proyecto y satisfacer las necesidades de los usuarios.

En la Tabla 1 se presentan los requisitos funcionales del sistema.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales describen las restricciones o requisitos impuestos al sistema. Los requisitos no funcionales abordan aspectos como la escalabilidad, la facilidad de mantenimiento, el rendimiento, la portabilidad, la seguridad, la confiabilidad y otros aspectos similares.

A diferencia de los requisitos funcionales, los requisitos no funcionales no se centran en las acciones específicas que debe realizar el sistema, sino en las características y cualidades que el sistema debe poseer para cumplir con estándares de calidad, rendimiento y experiencia del usuario.

En la Tabla 2 se presentan los requisitos no funcionales del sistema.

3.2. Arquitectura

La arquitectura, referida al software, es una estrategia de planificación que se basa en modelos, patrones y abstracciones teóricas. Se aplica antes de la implementación y es fundamental al desarrollar software complejo. Implica diseñar la estructura y organización del software, estableciendo un plan teórico que guía cómo se construirá la aplicación.

El proyecto tiene dos objetivos principales:

Referencia	Descripción
RF-01	El programa debe ser capaz de comprender y procesar la información dada por el usuario al momento de ser ejecutado.
RF-02	El programa debe ser capaz de leer y analizar un conjunto de archivos GPX.
RF-03	El programa debe ser capaz de chequear la circularidad de las pistas.
RF-04	El programa debe ser capaz de encontrar los puntos de intersección entre las pistas.
RF-05	El programa debe ser capaz de generar un grafo utilizando los puntos *relevantes* como vértices.
RF-06	El programa debe ser capaz de serializar el grafo y guardar esta representación en un archivo.
RF-07	El programa debe ser capaz de leer un archivo y deserializar su contenido, y luego utilizar esa representación para reconstruir el grafo original.
RF-08	El programa debe ser capaz de encontrar los puntos en el grafo más cercanos a los puntos proporcionados por el usuario.
RF-09	El programa debe ser capaz de buscar el camino de mínima distancia o mínimo desnivel acumulado a partir de los puntos anteriormente encontrados.
RF-10	El programa debe ser capaz de crear una ruta en formato GPX con el(los) camino(s) encontrado(s).
RF-11	El programa debe ser capaz de guardar la ruta en un archivo.

Tabla 1: Requisitos funcionales

- Generar un grafo a partir de un conjunto de archivos GPX
- Calcular rutas entre una serie de puntos usando el grafo previo.

El proyecto se dividirá en dos módulos principales. Cada módulo principal se encargará de completar uno de los objetivos principales. Además, se desarrollarán módulos auxiliares que contendrán funciones y utilidades auxiliares que serán utilizadas por los módulos principales.

Los módulos principales no interactúan entre sí y los módulos auxiliares tampoco interactúan entre sí. La interacción es solo entre módulo principal y módulo auxiliar, siguiendo un patrón de dependencia.

Como cualquier diseño, esta arquitectura tiene sus propias ventajas y desventajas. A continuación se detallan algunas de ellas.

Ventajas:

- **Modularidad:** La división en módulos independientes facilita el desarrollo y el mantenimiento. Cada módulo se puede desarrollar y probar por separado, lo que mejora la escalabilidad y la capacidad de actualización.
- **Reutilización:** Los módulos auxiliares contienen funciones y utilidades que pueden ser reutilizadas en diferentes partes del proyecto o incluso en futuros

Referencia	Descripción
RNF-01	El programa debe procesar y generar el grafo de manera eficiente para asegurar tiempos de respuesta adecuados, incluso cuando se manejen conjuntos de datos grandes.
RNF-02	El programa debe calcular las rutas de manera eficiente, minimizando el tiempo de procesamiento y utilizando algoritmos optimizados para la búsqueda de caminos en el grafo.
RNF-03	El código debe estar correctamente documentado con comentarios explicativos.
RNF-04	El código debe seguir convenciones de nomenclatura.
RNF-05	La interfaz de línea de comandos debe ser intuitiva y fácil de entender para los usuarios, con comandos y opciones claramente nombrados y explicados.
RNF-06	El programa debe ser capaz de manejar múltiples pistas, manteniendo un rendimiento aceptable a medida que aumenta la cantidad de datos.
RNF-07	El programa debe ser capaz de ejecutarse en diferentes sistemas operativos y entornos sin problemas, sin depender de plataformas específicas.
RNF-08	El programa debe ser capaz de manejar el caso en el que no exista una ruta entre dos puntos dados, proporcionando una retroalimentación clara y adecuada al usuario.
RNF-09	Se debe proporcionar una documentación detallada que explique como instalar el intérprete del programa y las librerías necesarias para su funcionamiento.
RNF-10	Se debe proporcionar un procedimiento rápido y sencillo que permita instalar las librerías necesarias de forma automática.
RNF-11	Se debe proporcionar ejemplos claros de diferentes casos de uso, demostrando cómo ingresar los datos requeridos y cómo interpretar los resultados.

Tabla 2: Requisitos no funcionales

proyectos.

- **Flexibilidad:** Ni los módulos principales ni los módulos auxiliares interactúan entre sí directamente. Esto permite cambiar o mejorar uno de ellos sin afectar al resto, lo que facilita la adaptación a cambios futuros.
- **Claridad de responsabilidades:** Cada módulo tiene una tarea específica, lo que facilita la comprensión de su propósito y función en el proyecto.

Desventajas:

- **Complejidad de comunicación:** La necesidad de utilizar un archivo de texto para intercambiar datos entre el módulo de generación del grafo y el módulo de cálculo de la ruta podría agregar complejidad a la comunicación.
- **Acoplamiento implícito:** Aunque se evita el acoplamiento directo entre los módulos principales, la dependencia del archivo introduce un tipo de acopla-

miento indirecto entre ellos a través del archivo compartido.

4. Implementación

Como se ha mencionado anteriormente, el proyecto se divide en dos módulos principales. Ambos módulos reciben argumentos de entrada, y para ambos se utiliza la librería argparse para leer, analizar y manejar los argumentos de entrada.

La utilización de argparse permite una definición clara y documentada de los argumentos que se esperan. Además, mantener la misma metodología en ambos módulos principales garantiza una experiencia uniforme para los usuarios, lo que facilita la comprensión y la interacción con el programa.

4.1. Generación del grafo

La generación del grafo se realiza en el módulo **genera_grafo.py**

La complejidad temporal de esta parte del proyecto está dominada por la tarea de "Determinar intersecciones entre las pistas", descrita en el apartado 4.1.5. En el peor caso, esta tarea tiene una complejidad temporal de $O(n^2 \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos de cada pista. Sin embargo, debido a la utilización de técnicas de reducción del espacio de búsqueda, en la mayoría de escenarios su complejidad temporal real es mucho menor, tendiendo a ser cuadrática.

4.1.1. Leer archivos GPX

El primer paso es leer el conjunto de archivos GPX, para ello se utiliza el método `leer_directorio_gpx` del módulo gpx_auxiliar_methods.py

El método `leer_directorio_gpx` tiene como objetivo leer e interpretar los archivos GPX que se encuentran en un directorio específico. Recibe como entrada la ruta del directorio y busca todos los archivos con extensión .gpx en ese directorio. Luego, itera sobre cada uno de ellos, los abre en modo de lectura, y utiliza la librería gpxpy para analizar su contenido y crear objetos GPX que representan el contenido de cada archivo. Estos objetos GPX se agregan a una lista que se devuelve como resultado.

La complejidad temporal de este método se estima en $O(n \cdot m)$, donde n representa el número de archivos GPX presentes en el directorio y m denota el tamaño promedio de cada archivo GPX. Esta complejidad se debe a que el método itera sobre los archivos GPX en el directorio, y por cada archivo GPX, realiza un análisis de su contenido.

4.1.2. Reducir el número de puntos de las pistas

Antes de comenzar a analizar las pistas, es interesante reducir el número de puntos de estas por varias razones.

- Reducir el tamaño de los datos: Dependiendo de la frecuencia de muestreo del dispositivo que grabó la ruta, es posible que algunas pistas contengan una densidad excesiva de puntos. Un ejemplo ilustrativo de este exceso de puntos

podría ser la grabación de diez puntos en un tramo recto, cuando únicamente dos puntos son necesarios para identificar los extremos de la recta.

Al reducir la cantidad de puntos, se reduce el tiempo empleado en generar el grafo, y también se obtiene un grafo cuyas artistas contienen menos información no relevante, lo que permite también reducir el tiempo empleado en el cálculo de la ruta.

- Reducir el "vagabundeo": El término "vagabundeo" se utiliza para describir los errores o desviaciones en la grabación del dispositivo GPS debido a la imprecisión en las mediciones. Es especialmente aplicable a situaciones en las que el dispositivo de grabación permanece estático, pero debido a la inexactitud en las mediciones, el dispositivo registra puntos que pueden estar ligeramente desplazados con respecto a su ubicación real.

  Empleando ciertos algoritmos de reducción de puntos, es posible eliminar o mitigar el "vagabundeo". Estos algoritmos seleccionan los puntos más significativos en una pista, eliminando aquellos que no contribuyen sustancialmente a la representación precisa del recorrido. Al reducir la cantidad de puntos, se puede suavizar el efecto del "vagabundeo" y obtener una ruta más clara y coherente.

- Igualar la longitud y el desnivel acumulado: Existe la posibilidad de que pistas que recorren el mismo camino tengan distinta longitud y desnivel acumulado. Esto se debe en gran medida a la diferencia entre la densidad de puntos de las pistas y la imprecisión inherente en las mediciones de posición.

  Al reducir el número de puntos de las pistas con mayor concentración de puntos conseguimos equiparar la densidad de puntos entre las distintas pistas. Esto, a su vez, tiende a igualar la longitud y el desnivel acumulado entre ellas.

Después de haber comprendido las ventajas de reducir la cantidad de puntos en las pistas, el siguiente paso es seleccionar el algoritmo adecuado para lograr esta reducción.

La clase GPXTrackSegment de la librería gpxpy proporciona dos métodos para reducir la cantidad de puntos de un segmento de una pista GPX: `reduce_points` y texttttsimplify. Aunque ambos métodos tienen el propósito de simplificar el segmento, hay una diferencia importante entre ellos en cómo logran esta simplificación.

`reduce_points(self, min_distance: float)`: Este método examina cada punto de una pista, en orden, y descarta cada punto que no esté al menos a una distancia mínima (min_distance) del último punto no descartado. En otras palabras, este método busca suavizar y simplificar la pista eliminando puntos cercanos que no aportan información relevante.

`simplify(self, max_distance: Optional[float]=None)`: Este método implementa el algoritmo de Ramer–Douglas–Peucker, este algoritmo selecciona los puntos extremos y luego encuentra el punto más alejado de la línea. Si la distancia entre la línea y el punto más alejado está por debajo de un umbral de tolerancia, se descartan los puntos intermedios. Por otro lado, si la distancia excede el umbral, se divide la línea en dos segmentos y se repite el proceso en cada uno de ellos.

En términos de eficiencia y rendimiento, el método `reduce_points` tiende a ser más rápido que el método `simplify`. La razón principal es que el algoritmo utilizado en `reduce_points` es más simple y directo, ya que se basa en eliminar puntos consecutivos que están más cerca entre sí que una distancia específica. En cambio, el algoritmo de Douglas-Peucker utilizado en `simplify` implica cálculos más complejos, porque debe determinar la distancia perpendicular entre cada punto y la línea que conecta los puntos inicial y final del segmento.

Por otro lado, el algoritmo de Douglas-Peucker utilizado en `simplify` es conocido por su capacidad para preservar mejor la forma general de la línea y mantener detalles importantes. A diferencia de `reduce_points`, que simplemente elimina puntos consecutivos basándose en una distancia fija.

Para este proyecto es más importante el resultado final y la calidad de la simplificación, por tanto es preferible utilizar el método `simplify`.

Una vez seleccionado el método a utilizar, se procede a realizar un análisis exhaustivo de su complejidad temporal. Posteriormente, se llevará a cabo el estudio de la complejidad temporal del bloque en su totalidad.

Como se ha mencionado previamente, el método `simplify` implementa el algoritmo de Ramer–Douglas–Peucker. La complejidad temporal de este algoritmo depende de varios factores [10].

- En el peor caso, la complejidad temporal del algoritmo es $O(n^2)$, donde n es el número de puntos de la polilínea original. Esto ocurre cuando, en cada invocación recursiva, el valor de i es igual a 1 o igual a $n-2$.

  La variable i en el contexto de este algoritmo representa el índice del punto más lejano con respecto a la línea que conecta el primer y el último punto de la polilínea en cada invocación recursiva.

  Cuando el valor de i es igual a 1 o igual a $n-2$ significa que en cada paso de la recursión se está seleccionando el primer punto o el penúltimo punto de la polilínea como el punto más lejano.

  Esto conduce al peor caso en términos de complejidad temporal porque, en cada paso recursivo, se está dividiendo la curva en dos partes desiguales: una parte que contiene solo un punto y otra parte que contiene todos los demás. Esta división desigual causa que el algoritmo deba realizar muchas más llamadas recursivas para procesar todos los puntos, lo que aumenta significativamente su complejidad temporal.

- En el mejor caso, la complejidad temporal es $O(n \cdot log(n))$, cuando en cada invocación recursiva el valor de i es aproximadamente igual a $\frac{n}{2}$ o $\frac{n\pm1}{2}$.

  Cuando el valor de i es igual a $\frac{n}{2}$ o $\frac{n\pm1}{2}$ significa que se está seleccionando aproximadamente el punto medio de la polilínea como el punto más lejano.

  Esto divide la curva en dos partes casi iguales, lo que resulta en una división equitativa y eficiente. Este equilibrio es lo que conduce a una complejidad temporal de $O(n \cdot log(n))$.

- Con el uso de estructuras de datos de envolvente convexa (total o semi) dinámicas, la simplificación realizada por el algoritmo se puede lograr siempre en $O(n \cdot log(n))$ tiempo.
- En ciertas circunstancias, la complejidad puede reducirse a $O(n)$ mediante un enfoque iterativo.

La implementación de este algoritmo en el método `simplify` no utiliza estructuras de datos de envolvente convexa (total o semi) dinámicas ni un enfoque iterativo. Por tanto, en el peor caso, la complejidad temporal es $O(n^2)$ y, en el mejor caso, la complejidad temporal es $O(n \cdot log(n))$.

Como este proceso se repite para cada pista, la complejidad temporal total de este paso es, en el peor caso, $O(n \cdot m^2)$ y, en el mejor caso, $O(n \cdot m \cdot log(m))$, donde n es el número de pistas y m es el número promedio de puntos de cada pista.

4.1.3. Convertir objetos GPXTrack en objetos LineString

Por cada objeto GPX se va a crear una instancia de la clase LineString a partir de la pista contenida en el objeto GPX. El proceso de creación de un objeto LineString es el siguiente: A partir de la pista, representada por una instancia de la clase GPXTrack que incluye un objeto GPXSegment con una lista de puntos, se toma latitud, longitud y elevación de cada punto. A continuación, se crea una instancia de la clase LineString a partir de la la latitud, longitud y elevación esos puntos.

La conversión de pistas en instancias de la clase LineString es muy útil porque esta clase ofrece una serie de funcionalidades significativas que simplifican en gran medida el análisis y la manipulación de datos geográficos.

La utilidad principal radica en la capacidad de realizar operaciones geométricas en líneas de una manera excepcionalmente sencilla. LineString proporciona una serie de métodos y funciones que facilitan operaciones como intersecciones, uniones y otras.

La complejidad temporal del bloque de código, que se encarga de completar esta tarea, es $O(n \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos en cada pista. El bucle externo itera sobre las n pistas, lo que implica una complejidad $O(n)$. Las dos instrucciones dentro de este bucle tienen una complejidad lineal dependiente de m, que es el número promedio de puntos en cada pista. Dado que estas dos operaciones se realizan para cada una de las n pistas, la complejidad temporal total del bloque es $O(n \cdot m)$.

4.1.4. Chequear la circularidad de las pistas

En una pista circular donde el punto inicial y final están muy próximos o coinciden, es posible ir desde el punto inicial al final y viceversa sin ningún problema. Esto se plasmará en el grafo creando una arista entre el punto final e inicial.

Por tanto, es necesario determinar cuándo una pista es circular o no. Parece sencillo, simplemente consiste en comprobar si el punto inicial y final son cercanos, la dificultad aparece a la hora de determinar cuándo dos puntos son cercanos.

La manera más sencilla para determinar si dos puntos están próximos entre sí es calcular la distancia entre ellos, si la distancia es menor que un umbral establecido,

se considera que los puntos son cercanos.

Elegir este umbral de distancia no es una tarea fácil. En el contexto que nos influye, una pista puede ser considerada circular sin estar su punto inicial y final relativamente cerca. Por ejemplo, una pista que empieza y termina en el mismo pueblo, desde una perspectiva lógica es claramente una pista circular, pero quizá el punto de inicial se encuentra en un extremo del pueblo y el punto final en otro extremo, separados a una distancia considerable.

Aunque matemáticamente la distancia entre los puntos puede no cumplir con la definición estricta de una pista circular, que es que el punto inicial y final están muy próximos o coincidan, en términos prácticos y racionales, el hecho de que el recorrido forme un bucle cerrado y regrese cerca del punto de partida sugiere que se trata de una pista circular en la mayoría de los contextos.

Después de este análisis y diversas pruebas se ha tomado la siguiente decisión. Si la distancia entre el punto inicial y final es inferior a 100 metros, se considerarán cercanos y, en consecuencia, se clasificará la pista como circular.

El tiempo de ejecución de chequear la circularidad de una pista es constante $O(1)$. Esta eficiencia se debe a que las operaciones realizadas en el proceso, como el acceso al punto inicial y final de la línea, el cálculo de la distancia geodésica y la comparación de esta distancia con el valor umbral son constantes en términos temporales.

Como este proceso se repite para cada pista, la complejidad temporal total de este paso es $O(n)$, donde n es el número de pistas.

4.1.5. Determinar intersecciones entre las pistas

Existen dos tipos de intersección entre pistas.

- *Intersección de Cruce:* En este tipo de intersección, dos pistas se cruzan en un punto específico. Las pistas se encuentran y forman un cruce.
- *Intersección con Tramo Compartido:* En este tipo de intersección, dos pistas comparten un tramo de camino durante un tiempo. En lugar de cruzarse en un solo punto, las pistas comparten una sección del camino antes de separarse nuevamente.

Para este proyecto, con el fin de simplificar el código, ambos tipos de intersección se tratan de la misma forma. A partir de este momento se emplea el término *tramo común* para hacer referencia a ambas.

Identificar intersecciones entre pistas puede resultar complicado por una razón fundamental, a pesar de que las pistas se crucen o recorran el mismo camino, la probabilidad de que el punto o puntos coincidan exactamente es mínima.

El objetivo es encontrar los tramos comunes entre todas las combinaciones posibles de dos pistas.

Para poder iterar sobre cada par de pistas se necesitan dos bucles anidados. Es importante comprender que al buscar tramos en común entre cualquier par de pistas, como por ejemplo, entre la pista A y la pista B, se obtiene el mismo resultado que

al buscar entre la pista B y la pista A. Por lo tanto, es fundamental evitar repetir combinaciones.

El bucle exterior debe iterar sobre todas las pistas. Por otro lado, el bucle interior debe iterar solo sobre las pistas que aún no se han comparado con la pista actual en el bucle exterior, es decir, debe iterar desde la pista actual en el bucle exterior hasta la última pista.

Mediante la implementación de esta técnica, que evita repetir combinaciones, conseguimos optimizar la complejidad temporal de este bucle doble, reduciendo el número de iteraciones de n^2 a $(n-1)^2$, siendo n la cantidad de pistas presentes. Nos quedamos con $O(n^2)$ porque las constantes no son un factor relevante, ya que ambas tienen la misma forma cuadrática, y la diferencia entre $(n-1)^2$ y n^2 es marginal a medida que n crece.

Aunque la complejidad es menor, es importante destacar que asintóticamente es equivalente a $O(n^2)$, por lo que nos quedamos con esta notación más simple.

Una vez establecida la estructura para analizar los pares de pistas evitando repeticiones, se podría asumir que es el momento de buscar intersecciones. Sin embargo, buscar intersecciones entre pistas conlleva un costo significativo. Por esta razón, es interesante dirigir la búsqueda de intersecciones únicamente hacia aquellos casos donde exista la posibilidad de que las pistas compartan algún tramo.

Para comprobar, de una forma rápida, si existe la posibilidad de que un par de pistas compartan un tramo se calcula el cuadro delimitador mínimo o minimun bounding box para cada pista. Un cuadro delimitador es un sistema de coordenadas rectangulares utilizado para definir un área geográfica en los mapas. Un cuadro delimitador mínimo es el cuadro con la medida más pequeña dentro del cual se encuentran todos los puntos de la geometría a delimitar. En este proyecto, se utiliza el atributo `bounds`, de la clase LineString, para obtener los límites y generar a partir de estos el cuadro delimitador mínimo.

Solo si los cuadros delimitadores de ambas pistas se superponen, existe la posibilidad de que haya un tramo común entre ellas.

Gracias a esta técnica se limita la búsqueda de intersecciones solo a aquellos casos en los que existe la posibilidad real de que dos pistas tengan algún tramo en común. De esta manera se logra un enfoque más eficiente y efectivo, lo que propicia un ahorro considerable de recursos y tiempo en comparación con la búsqueda exhaustiva de intersecciones sin ningún criterio de preselección.

Ahora que hemos establecido si hay una posibilidad real de que el par de pistas tenga algún tramo en común, estamos en posición de avanzar hacia una búsqueda minuciosa de este.

Para encontrar el/los tramo/s común/es entre dos pistas, representadas como instancias de la clase LineString, se ha desarrollado el método `encontrar_interseccion` en el módulo `lineString_auxiliar_methods.py` que realiza el siguiente proceso:

1. Se crea un búfer alrededor de una línea: Se utiliza el método `bufer` de la clase LineString para crear un búfer alrededor de esa línea. Un búfer es una zona o área alrededor de una geometría que se crea a cierta distancia de la geometría

original. Aplicado a una instancia de la clase LineString, el búfer resultante es un polígono que sigue la forma de la línea original pero expandida.

2. Se utiliza el método `intersection` para encontrar los puntos de intersección entre la línea expandida y la otra línea.

3. En una situación donde dos pistas, en parte de su recorrido, pasan por el mismo camino, lo ideal sería identificar esta porción como una única área de intersección. Sin embargo, debido a posibles inexactitudes en la lectura del GPS, es factible que un punto de la pista esté desviado de su verdadera ubicación. Como resultado, en lugar de reconocer un solo espacio de intersección, se pueden identificar múltiples áreas de intersección fragmentadas.

   El segundo buffer, aplicado al resultado de la intersección, actúa como un puente que conecta estas áreas de intersección que pueden haber sido detectadas de forma separada. Esto garantiza que cualquier tramo compartido por ambas pistas se trate como uno único.

4. La intersección bufferizada es una instancia de la clase Polygon si solo existe una zona de intersección, es decir, si solo existe un tramo común, o una instancia de la clase MultiPolygon si existen múltiples zonas de intersección.

La complejidad temporal de este método es $O(n+m)$, donde n es el número de puntos de una línea y m el número de puntos de la otra. Esto se debe a que la operación más costosa es encontrar los puntos de intersección entre la línea buferizada y la otra línea, y su complejidad es $O(n + m)$.

Sin embargo, en este contexto, la complejidad temporal puede considerarse como $O(n)$. Dado que el número promedio de puntos en ambas líneas es similar, la complejidad se simplifica a $O(n)$, ya que las constantes multiplicativas se omiten en la notación de complejidad asintótica. En otras palabras, el método recibe dos objetos LineString que representan pistas, y dado que el número promedio de puntos en cada pista es igual, la notación $O(n+m)$ se convierte en $O(n+n)$, que se simplifica a $O(2n)$ y, finalmente, se reduce a $O(n)$.

Con el fin de conseguir una estructura de datos uniforme, tanto las instancias de la clase Polygon como las instancias de la clase MultiPolygon se almacenan como elementos individuales en una lista de objetos Polygon.

Si se tiene un objeto Polygon, se agrega directamente a la lista. Si es un MultiPolygon, se extraen los objetos Polygon individuales que lo componen mediante el método `geoms` y se añaden a la lista.

Para almacenar el/los tramo/s común/es entre cada par de pistas se utiliza un diccionario. Cada clave en el diccionario es una tupla de dos números, que representan los identificadores de las dos pistas que forman el par en cuestión. El valor asociado a cada clave es una lista de objetos de la clase Polygon. Cada objeto Polygon en la lista representa un tramo común entre las dos pistas correspondientes al par de identificadores en la clave.

La complejidad temporal del bloque de código que se encarga de realizar esta tarea está dominada por el bucle doble y la llamada a `encontrar_interseccion` que se encuentra dentro de él. La complejidad del bucle doble, gracias a evitar repe-

tir combinaciones, es $O(n^2)$, donde n es el número de pistas y la complejidad de `encontrar_interseccion` es $O(m)$, siendo m el número promedio de puntos de cada pista. Por consiguiente, la complejidad temporal del bloque se puede expresar como $O(n^2 \cdot m)$.

Esta complejidad temporal puede parecer alta, pero es la complejidad en el peor caso, donde los cuadros delimitadores mínimos de todas las pistas se intersecan entre sí. Sin embargo, en la mayoría de escenarios, gracias a la técnica de reducción del espacio de búsqueda, la complejidad real tiende a ser cuadrática o un ligeramente mayor. Esta es claramente menor que la complejidad temporal en el peor caso, que es cúbica.

4.1.6. Crear grafo

Se utiliza el método `Graph` de la librería NetworkX para crear un grafo vacío sin nodos ni aristas. En este grafo se incluirán los puntos iniciales y finales y los puntos de intersección como nodos y los caminos que conectan estos puntos, junto con la información asociada, como aristas.

La complejidad temporal de este método es $O(1)$, ya que la creación de un grafo vacío no depende del tamaño de ningún conjunto de datos o estructura existente, sino que simplemente inicializa una estructura de grafo vacía en la memoria.

4.1.7. Agregar puntos iniciales y finales al grafo

Por cada pista se añade el punto inicial y final como nodos del grafo. Es necesario añadir estos puntos para garantizar que aunque una pista no interseque con ninguna otra, esta esté recogida en el grafo y pueda ser tenida en cuenta en la parte del cálculo de rutas.

La complejidad temporal de este fragmento de código es lineal, es decir, $O(n)$, donde n es el número de pistas, porque en cada iteración del bucle, se realizan dos operaciones, añadir el punto inicial y final), ambas con una complejidad constante de $O(1)$.

4.1.8. Hallar puntos de intersección entre tramo común y par de pistas

A continuación, se explicará como encontrar, para cada tramo común, el punto inicial y final de la intersección entre el tramo común y el par de pistas asociadas a este tramo común.

Previamente, se explicó cómo obtener una lista de objetos de la clase Polygon que representaban los tramos en común entre un par de pistas.

También se citaron y expusieron los dos tipos de intersección entre pistas, y se declaró que ambos tipos se iban a tratar de la misma forma con el fin de producir un código más sencillo.

Se podría decir que tanto la *intersección de cruce* como la *intersección de tramo* se van a manejar como una *intersección de tramo*. En cierta manera, la *intersección de cruce* es una *intersección de tramo* solo que de una longitud extremadamente pequeña.

El tramo común entre dos pistas tiene un inicio y un fin, véase en la Figura 1. La idea es obtener por cada pista, el primer y último punto dentro del tramo común, véase en la Figura 2. Se obtendrán entonces cuatro puntos por cada tramo común.

Estos cuatro puntos mantienen una correspondencia específica: el punto de inicio de la intersección entre la primera pista y el tramo común está vinculado con el punto de inicio de la intersección entre la segunda pista y el mismo tramo común. Del mismo modo, el punto final de la intersección entre la primera pista y el tramo común tiene una relación con el punto final de la intersección entre la segunda pista y el mismo tramo común. En la Figura 2, se evidencia la relación entre A1 y B1 y entre A2 a B2. Se podría clasificar tanto a A1 y B1 como a A2 y B2 como *puntos de intersección vinculados*.

Para encontrar estos cuatro puntos, primero se encuentran los dos de una pista y luego los dos de la otra. Si se toma como ejemplo la Figura 2, primero se encuentran A1 y A2, y posteriormente, B1 y B2. El método de búsqueda para los primeros dos difiere del utilizado para los otros dos.

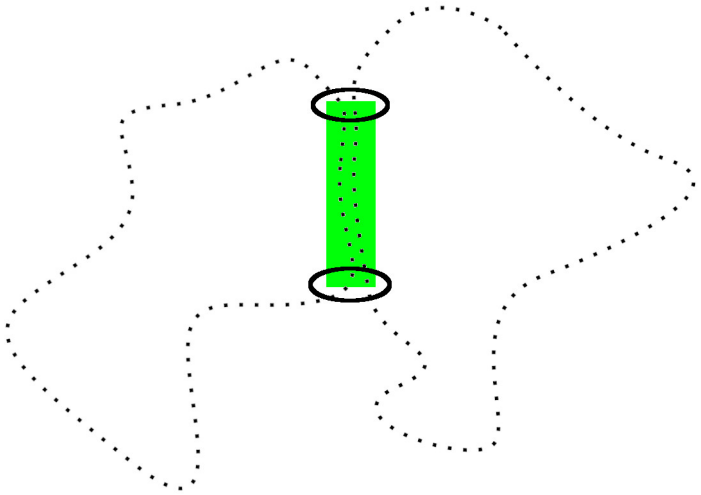

Figura 1: Inicio y fin de un tramo común

Primero, se utiliza la función `encontrar_puntos_interseccion` del módulo lineString_auxiliar_methods.py para encontrar el primer y último punto de la intersección entre la primera pista y el tramo común.

Este método es sencillo en su base, simplemente debe encontrar los puntos de inicio y fin de la intersección entre la pista, que es una instancia de la clase LineString, y el tramo común, que es una instancia de la clase Polygon.

La dificultad para implementar este método reside en tratar el caso en el que ya el primer punto de la pista interseca con el tramo común. Si el primer punto de la pista está dentro del tramo común, significa que la intersección comenzó antes del primer punto de la pista. Para solventar este caso especial se debe recorrer la pista hacia delante en busca del último punto de intersección, y luego se debe recorrer hacia atrás, utilizando la funcionalidad `reversed`, para encontrar el primer punto.

La complejidad temporal de este método es $O(n)$, donde n representa el número de puntos de la línea.

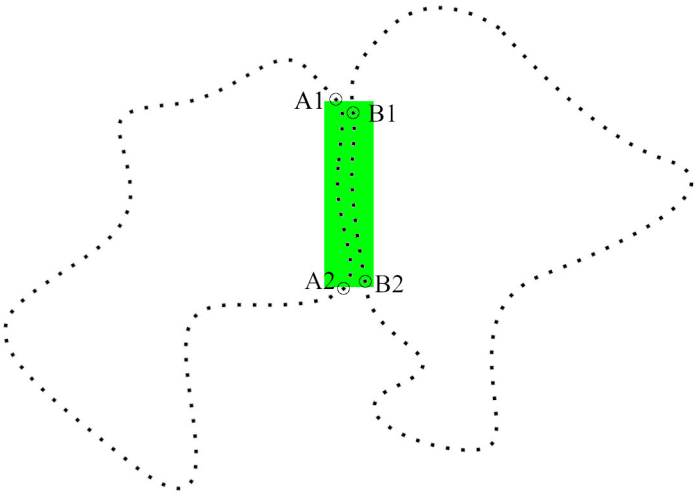

Figura 2: Primer y último punto de las pistas dentro del tramo común

Una vez encontrados estos dos puntos, correspondientes al primer y último punto de la intersección entre la primera pista y el tramo común, podemos buscar los otros dos puntos, los de la intersección entre la segunda pista y el mismo tramo común.

En teoría, se podría emplear el procedimiento utilizado previamente. Sin embargo, surge un desafío, dependiendo del sentido de la pista, se podría obtener los puntos desordenados. Reorganizar estos puntos en el orden correcto podría resultar complicado. Si se toma como ejemplo la Figura 2 podría darse el caso favorable de encontrar A1, A2 y B1, B2 o A2, A1 y B2, B1 o el caso desfavorable de encontrar A1, A2 y B2, B1 o A2, A1 y B1, B2.

Por tanto, es mejor utilizar otro método que evite tener que reorganizar los puntos. El método desarrollado para este propósito es `encontrar_punto_mas_cercano`. Este método recibe la segunda pista, que es una instancia de la clase LineString y un punto. Su función consiste en localizar el punto más cercano en la polilínea en relación con el punto de referencia proporcionado.

Tras encontrar A1 y A2 con el método `encontrar_puntos_interseccion`, se utiliza el método `encontrar_punto_mas_cercano` pasándole A1 para encontrar B1 y pasándole A2 para encontrar B2. Este método simplemente busca el punto de línea que tenga la menor distancia geodésica respecto al punto de referencia.

Es un método muy sencillo que encima nos permite tener la correspondencia entre los puntos de inicio y fin de la intersección del tramo común con cada pista.

La complejidad temporal de este método es también $O(n)$.

Con el fin de construir un grafo con el menor número de nodos y aristas, pero sin perder información ni precisión, se almacenan los puntos de intersección y sus relaciones siguiendo un enfoque específico.

En la variable `listas_puntos_interseccion`, que es una lista de conjuntos, cada elemento contiene los puntos de intersección de una pista con respecto a las demás. La longitud de esta lista es igual al número total de pistas, y cada conjunto dentro de la lista contiene un número variable de puntos. La cantidad de puntos en cada

conjunto depende del número de veces que la pista en cuestión se cruza con las demás.

La variable `lista_parejas_puntos_interseccion` se emplea para guardar la relación entre los puntos de intersección entre el tramo común y el par de pistas asociado. Esta variable es una lista de tuplas donde cada tupla contiene dos listas, ambas con el mismo número de puntos. En cada tupla, el elemento i de la primera lista se relaciona con el elemento i de la segunda lista. Por ejemplo, siguiendo el escenario presentado en la Figura 2, el elemento i de la primera lista sería A1 y el elemento i de la segunda lista sería B1, y el elemento j de la primera lista sería A2 y el elemento j de la segunda lista sería B2.

A continuación, se va a detallar la complejidad temporal del bloque de código encargado de realizar todas las tareas descritas anteriormente.

Para iterar sobre todos los *tramos comunes* se itera sobre cada par de pistas que tienen algún *tramo común* y luego se itera sobre cada *tramo común* del par de pistas. Se podría plasmar la complejidad temporal de este bucle doble como $O(n \cdot m)$, donde n es el número de pares de pistas que tienen algún *tramo común* y m es el número promedio de *tramos comunes* por cada par de pistas. Sin embargo, si se multiplica n por m da el número total de *tramos comunes*, llámese p, entonces la complejidad sería $O(p)$, ya que p representa el número total de iteraciones que se realizan en el bucle doble. En este caso, $O(p)$ es una medida más representativa de la complejidad del algoritmo, puesto que refleja directamente la cantidad total de trabajo realizado.

Por otro lado, para buscar los *puntos de intersección vinculados* a los puntos de intersección ya encontrados hay que iterar sobre los puntos de intersección de cada pista. Nuevamente, se utiliza un bucle doble, de complejidad $O(n \cdot m)$, donde n es el número de pares de pistas que tienen algún *tramo común* y m es el número promedio de puntos de intersección de una pista del par de pistas. Se puede decir que su complejidad es $O(p)$ siendo p la mitad del número total de puntos de intersección. La mitad se debe a que estamos iterando solo sobre los puntos de intersección de una de las pistas del par. Como ya se conoce, por cada *tramo común* hay cuatro puntos de intersección, y como se está iterando sobre la mitad, se tiene dos puntos de intersección por cada *tramo común*. Entonces, se puede decir que la complejidad de este bucle doble, ahora estimada en $O(p)$ siendo p la mitad del número total de puntos de intersección, es $O(2q)$ siendo q el número total de *tramos comunes*, y que se puede recudir a $O(q)$, ya que las constantes multiplicativas se omiten en la notación de complejidad asintótica.

Dentro del primer bucle doble está la llamada a `encontrar_puntos_interseccion` y dentro del segundo está la llamada a `encontrar_punto_mas_cercano`. Ambos métodos tiene una complejidad temporal lineal que depende del número de puntos de la línea recibida, que es en promedio el número promedio de puntos de cada pista.

Entonces, para ambos bucles y su respectivo contenido se puede estimar una complejidad lineal de $O(n \cdot m)$, siendo n el número de *tramos comunes* y m el número promedio de puntos por cada pista.

Como la complejidad se rige por el bucle con la mayor complejidad temporal, y en este caso, ambos bucles tienen la misma, la complejidad total del bloque es $O(n \cdot m)$.

4.1.9. Añadir vértices y aristas a partir de los puntos de intersección

Seguidamente, se describirá cómo añadir los puntos de intersección obtenidos previamente y sus conexiones de manera altamente eficiente, minimizando la inclusión tanto de nodos como sobre todo de aristas.

Cada punto de intersección está conectado con dos *puntos relevantes* de la misma pista y con el *punto de intersección vinculado*.

En la Figura 3 se pueden observar los puntos de intersección de la pista A con la pista B. En la Figura 4 se pueden observar los puntos de intersección de la pista A con el resto de las pistas. En la Figura 5 se puede observar el grafo tras añadir los puntos de intersección como nodos y las conexiones entre ellos como aristas.

El método auxiliar `genera_arsitas` se encarga de crear y añadir al grafo las aristas que conectan los puntos de intersección de una misma pista. El método crea una arista entre el punto inicial y el primer punto de intersección, entre cada par de puntos de intersección consecutivos y entre el último punto de intersección y el punto final. Este proceso implica iterar a través de los puntos de la pista y establecer conexiones entre los puntos de intersección a medida que se encuentran durante la iteración.

La complejidad temporal de este método es $O(n)$, donde n es el número promedio de puntos de una pista. La razón detrás de esta complejidad es que el método recorre todos los puntos de la pista una sola vez para crear las aristas.

Y como este proceso se realiza por cada pista, la complejidad temporal total es $O(n \cdot m)$, donde n es el número de pistas y m el número promedio de puntos de una pista.

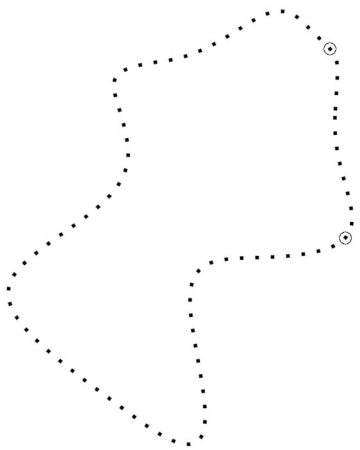

Figura 3: Puntos de intersección de la pista A con la pista B

Tras establecer las conexiones entre los puntos de intersección de una misma pista, el último paso sería agregar las conexiones entre los *puntos de intersección vinculados*

En la Figura 6, se pueden observar los puntos de intersección de la pista B con las demás pistas, en este caso, la pista B solo se cruza con la pista A. En la Figura 7, se muestra el grafo resultante después de añadir las conexiones entre los *puntos de intersección vinculados* de las pistas A y B. Posteriormente, se continuaría añadiendo las conexiones entre los puntos de intersección de la pista A con el resto de las pistas.

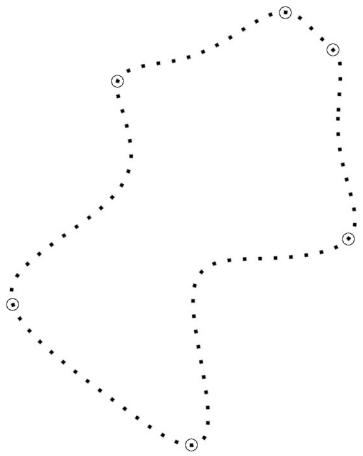

Figura 4: Puntos de intersección de la pista A con el resto de las pistas

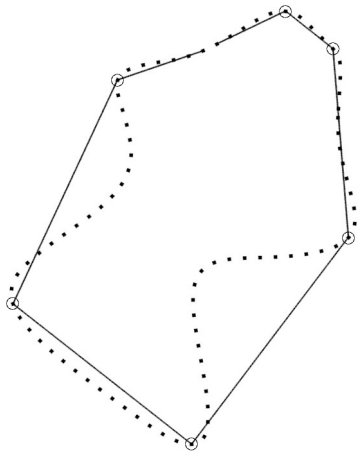

Figura 5: Conexiones entre los puntos de intersección de la pista A

Añadir la arista entre los *puntos de intersección vinculados* tiene complejidad $O(n)$, donde n es el número de puntos de intersección, ya que simplemente hay que iterar sobre los puntos de intersección y crear una arista para la pareja.

4.1.10. Añadir arista si la pista es circular

Agregar una arista que conecte el punto inicial con el punto final en una pista circular no solo se justifica desde una perspectiva lógica, sino que también aporta numerosos beneficios en cuanto a la creación de rutas.

La adición de esta arista añade múltiples posibilidades de rutas. Se puede comenzar en cualquier punto, llegar al punto de inicio y pasar al punto final o viceversa, y finalmente terminar en otro punto distinto. Esta versatilidad aumenta las posibilidades al crear rutas.

La inclusión de la arista debe realizarse únicamente si la pista es circular. Para verificar si la pista cumple esta condición, se comprueba en el diccionario donde se registró la información acerca de la circularidad de las pistas.

Los atributos de la arista tendrán los siguientes valores. El atributo “camino” representa la ruta que conecta los dos extremos de la arista, en este caso son el punto

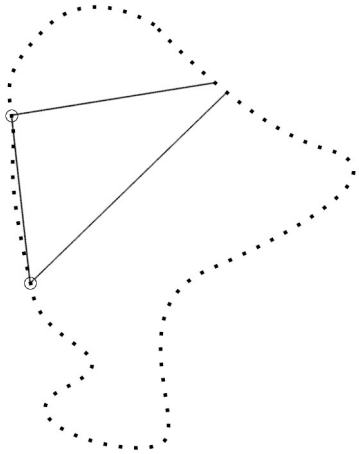

Figura 6: Conexiones entre los puntos de intersección de la pista B

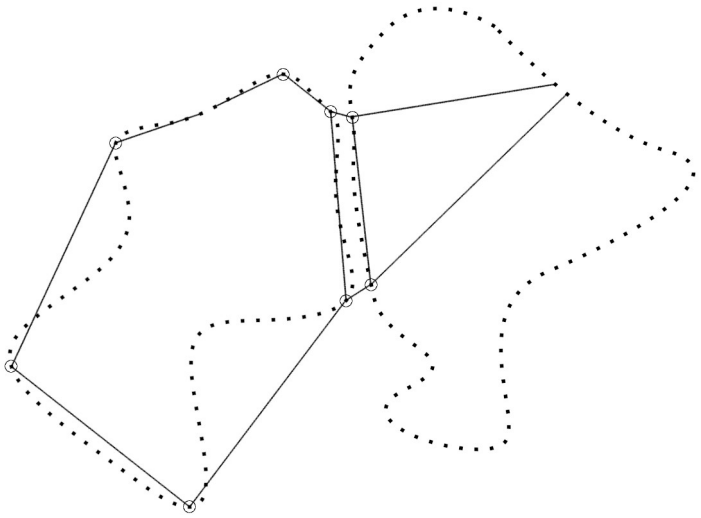

Figura 7: Conexiones entre los puntos de intersección de las pistas A y B

inicial y el punto final. Por lo tanto, el camino estará compuesto por ambos puntos. La "distancia" será la distancia geodésica entre estos dos puntos. Y finalmente, el atributo "desnivel acumulado" será el valor absoluto de la diferencia de altura entre los dos puntos.

La complejidad temporal de realizar esta tarea es $O(n)$, donde n es el número de pistas, ya que solamente hay que recorrer las pistas y añadir la arista entre el punto inicial y final si la pista es circular.

4.1.11. Añadir cuadro delimitador mínimo por cada arista

En este apartado se describirá el proceso de cálculo y adición del cuadro delimitador mínimo como atributo de arista, sin entrar en detalles sobre su utilidad.

El cuadro delimitador mínimo se refiere al rectángulo de menor tamaño que encierra todo el camino de una arista y se define por sus coordenadas mínimas y máximas en latitud y longitud.

Para determinar el cuadro delimitador mínimo, se obtienen las coordenadas mínimas

y máximas de latitud y longitud de todos los puntos que componen el camino de una arista. Estas coordenadas extremas establecen las esquinas del cuadro delimitador mínimo.

La complejidad temporal de realizar esta tarea es $O(n \cdot m)$, donde n es el número de aristas y m es el número promedio de puntos del camino de cada arista.

Quizá es difícil entender la magnitud de estos valores, pero se puede encontrar una equivalencia, que además, facilita la comparación de la complejidad del temporal de esta tarea con la del resto.

Gracias a la creación eficiente del grafo, el número de puntos del camino de todas las aristas es igual al número de puntos de todas las pistas. Entonces, la complejidad temporal de este método se puede expresar como $O(n \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos de cada pista.

4.1.12. Serializar y exportar el grafo

El propósito primordial de serializar y exportar el grafo es preservar la información de manera que se pueda recuperar y utilizar posteriormente en el cálculo de rutas.

Además de esta función principal, la serialización y exportación del grafo presentan otras ventajas significativas. Estas incluyen la capacidad de transferir datos entre sistemas a través de la red, así como la oportunidad de llevar a cabo análisis externos del grafo. Por ejemplo, al exportar el grafo en el formato GPX, es posible emplear herramientas de visualización de archivos GPX para observar una representación del grafo en un mapa. Asimismo, la serialización y exportación permiten la creación de copias de seguridad, lo que garantiza la disponibilidad de la información en caso de pérdida de datos u otros problemas técnicos. Además, posibilita trabajar con el mismo grafo en diferentes sesiones, sin necesidad de regenerarlo continuamente, lo que se traduce en un ahorro de tiempo y recursos.

Según el modo elegido por el usuario a través de la línea de comandos, se serializa el grafo en formato JSON o en formato GPX.

Para serializar el grafo en formato JSON se usa el método `serializa_graph_a_json` del módulo json_auxiliar_methods.py. Este método convierte el grafo en un diccionario compatible con JSON mediante la función de NetworkX `node_link_data` y luego utiliza la función `dump` de la librería json para escribir este diccionario en el archivo especificado.

Serializar el grafo en formato GPX tiene como propósito poder ser visualizado. Para ello, los vértices del grafo se convierten en waypoints y el atributo "camino" de las aristas se representa como pistas.

El proceso de serialización en formato GPX involucra la utilización del método `construir_gpx_con_waypoints` del módulo gpx_auxiliar_methods.py. Primero, se construye la parte correspondiente a las pistas mediante la función `construir_gpx` y luego, por cada punto, se crea un objeto GPXWaypoint y se incorpora al archivo GPX previamente construido. Finalmente, se llama a la función `escribir_gpx` para escribir y guardar el archivo GPX resultante en el sistema de archivos.

La complejidad temporal de serializar y exportar el grafo, tanto en formato JSON

como en formato GPX, depende del número de vértices y aristas. Como se ha mencionado anteriormente, la creación eficiente del grafo implica no añadir el mismo punto más de una vez en el grafo. Por tanto, el número de puntos del camino de todas las aristas es igual al número de puntos de todas las pistas. Entonces, se puede considerar la complejidad temporal como $O(n \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos de cada pista.

4.2. Cálculo de la ruta

La generación del grafo se realiza en el módulo **calcula_ruta.py**

La complejidad temporal de esta parte del proyecto está dominada por la tarea de "Buscar los puntos más cercanos en el grafo" y por la tarea 'Buscar camino mínimo' descritas en los apartados 4.2.2 y 4.2.6 respectivamente.

En el peor caso, la complejidad temporal de la primera tarea es $O(n \cdot m \cdot p)$, donde n es el número de aristas, m es el número promedio de puntos del camino de cada arista y p el número de puntos introducidos por el usuario, y la complejidad temporal de la segunda es $O((E + V) \cdot log(V)) \cdot p$, donde V es el número de vértices, E el número de aristas y p el número de puntos introducidos por el usuario.

En la mayoría de escenarios, la complejidad temporal de la primera tarea se reduce a aproximadamente $O(n \cdot p)$ y la de la segunda tarea a una estimación sustancialmente mejor que $O((E + V) \cdot log(V) \cdot p)$.

4.2.1. Importar y deserializar el grafo

Importar y deserializar el grafo solo se puede hacer a partir del archivo en formato JSON. La serialización y exportación del grafo en formato GPX se diseñó exclusivamente con el propósito de posibilitar la visualización del grafo, sin la intención de permitir su recuperación en un momento posterior.

Para importar y deserializar el grafo desde el archivo JSON se utiliza el método `serializa_graph_a_json` del módulo json_auxiliar_methods.py. Este método utiliza la función `load`, proporcionada por la librería json, para cargar los datos del grafo en formato JSON desde un archivo. Luego, utiliza el método `node_link_graph` de la librería NetworkX para reconstruir el grafo original a partir de los datos cargados desde el archivo JSON.

Al serializar y exportar el grafo, los atributos de las aristas pasaban de ser tuplas a ser listas. Cuando se serializa un grafo usando la función `dump`, los datos son convertidos a un formato que pueda ser almacenado en un archivo JSON. En este proceso, las estructuras de datos específicas, como las tuplas, a menudo se convierten en una estructura más genérica, como las listas.

Por tanto, tras importar y deserializar el grafo, se debe realizar una transformación inversa, convirtiendo las listas nuevamente en tuplas para que el grafo se recupere en su forma original.

La complejidad temporal de importar y deserializar el grafo depende del número de vértices, aristas y la información contenida en estas aristas. Esto se relaciona directamente con el número de pistas y el número promedio de puntos de cada

pista. Entonces, se puede estimar la complejidad temporal como $O(n \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos de cada pista.

4.2.2. Buscar los puntos más cercanos en el grafo

El usuario, por línea de comandos, ha introducido los puntos de la ruta. Desafortunadamente, es probable que ninguno de los puntos introducidos por el usuario coincida exactamente con ningún punto de los caminos representados por las aristas.

Por lo tanto, es necesario iterar sobre todos los puntos del camino de todas las aristas en busca de los puntos más cercanos a los puntos introducidos por el usuario. Este proceso garantiza que se encuentren los puntos en el camino de las aristas que tienen la menor distancia geodésica a los puntos introducidos por el usuario.

Los puntos que se encuentren como resultado de esta búsqueda serán los puntos que se utilizarán para calcular la ruta.

El algoritmo base, por cada punto introducido por el usuario, recorre todos los puntos del camino de todas las aristas y obtiene los puntos más cercanos. Aunque este algoritmo es funcional, su eficiencia es significativamente baja. Su complejidad temporal sería $O(n \cdot m)$, donde n es el número de aristas y m es el número promedio de puntos del camino de cada arista. Dado que el algoritmo debe ejecutarse para cada punto introducido por el usuario, la complejidad total sería $O(n \cdot m \cdot p)$, donde n y m son los valores mencionados anteriormente y p es el número de puntos introducidos por el usuario.

Para intentar mejorar la eficiencia del algoritmo, se va a implementar una estrategia de preselección de aristas. Esta estrategia tiene como objetivo reducir el espacio de búsqueda al descartar las aristas cuyo camino es imposible que contenga el punto más cercano al introducido por el usuario.

Antes de explicar el criterio de preselección, resulta fundamental aclarar que cuando se utiliza el término *esquina*, se hace referencia a una esquina del cuadro delimitador mínimo del camino representado por una arista.

Para cada arista en el grafo se calcula la distancia entre el punto introducido por el usuario y la *esquina* más lejana al punto, como se ilustra en la Figura 8. Entre todas las distancias calculadas, se selecciona la menor, que será la "distancia umbral", tal como se muestra en la Figura 9. Luego, para cada arista, se calcula la distancia entre el punto introducido y la *esquina* más cercana al punto, representado en la Figura 10. Esta distancia se compara con la "distancia umbral" calculada previamente. Si la distancia es menor que la "distancia umbral", se preselecciona la arista; si es mayor, se descarta, como se ejemplifica en la Figura 11.

En consecuencia, las aristas que superen esta fase de preselección serán aquellas en las que se realizará una búsqueda exhaustiva con el propósito de localizar el punto más cercano al introducido por el usuario.

Gracias a esta reducción del espacio de búsqueda, llevamos a cabo la búsqueda exhaustiva solo en una pequeña fracción del conjunto total de aristas, lo que reduce de manera significativa el tiempo empleado en este proceso a nivel global.

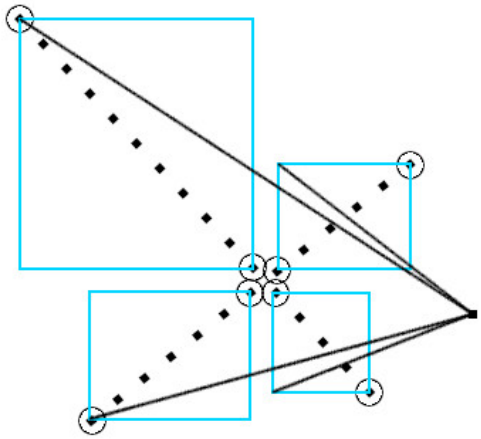

Figura 8: Distancias entre el punto y las *esquinas* más lejanas al punto

Entonces, la complejidad temporal, en la mayoría de escenarios, sería $O(n \cdot p)$, donde n es el número de aristas y p es el número de puntos introducidos por el usuario.

4.2.3. Crear vértices temporales

Anteriormente, dado una serie de puntos introducidos por el usuario, se ha buscado, en el camino de cada arista, los puntos más cercanos. Se necesita que estos puntos, que solo son elementos de un atributo de las aristas, sean vértices del grafo, para poder utilizarlos en los métodos que implementan los algoritmos de Dijkstra y A*.

El caso fácil es que el punto se corresponda con un extremo del camino, lo que significa que ya es un vértice y, por consiguiente, no hace falta hacer nada más.

Sin embargo, si el punto es un punto intermedio del camino, entonces se debe crear un vértice. Para ello, Habrá que dividir la arista en dos, utilizando este punto como nuevo vértice, y habrá que recalcular los atributos de las dos nuevas aristas generadas.

El método `genera_arista` es el encargado de recalcular los atributos del camino de las dos nuevas aristas, que son la longitud y el desnivel acumulado, y añadirlas al grafo. La complejidad temporal de este método es $O(n)$, siendo n el número de puntos del camino de la arista, ya que tiene que recorrer cada punto del camino para calcular la distancia y diferencia de elevación entre los puntos consecutivos, y así obtener la longitud y desnivel acumulado.

En el peor de los casos, donde ningún punto se corresponde con un vértice existente en el grafo, este método se ejecuta para cada punto más cercano en el grafo a los puntos introducidos por el usuario. Por tanto, la complejidad temporal total es $O(n \cdot p)$, donde n el número promedio de puntos del camino de las aristas y p el número de puntos introducidos por el usuario.

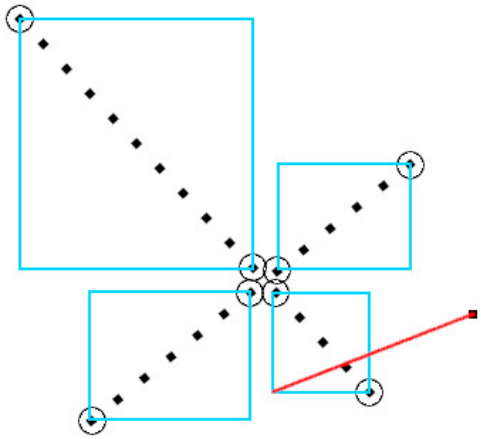

Figura 9: Distancia umbral

4.2.4. Definir lambdas para obtener distancia y desnivel de la arista

Las aristas del grafo tienen como peso una tupla. El primer elemento contiene la distancia y el segundo elemento contiene el desnivel acumulado.

Tanto el método `shortest_path`, que implementa el algoritmo de Dijkstra, como `astar_path`, que implementa el de A*, requieren que se les pase una función para seleccionar uno de los dos valores como peso de la arista.

Para definir estas funciones se va a utilizar las funciones lambda. Estas son similares a las funciones definidas por el usuario pero sin nombre. Suelen denominarse funciones anónimas. Las funciones lambda son eficaces siempre que se quiera crear una función que solo contenga expresiones sencillas, es decir, expresiones que suelen ser una sola línea de una sentencia [6].

Entonces, se definen dos funciones lambda: una para extraer el primer elemento de la tupla de peso, que indica la distancia, y otra para obtener el segundo elemento de la tupla de peso, que refleja el desnivel acumulado.

La complejidad temporal de definir una función lambda en Python es constante, es decir, $O(1)$. Cuando se define una función lambda, Python realiza una serie de tareas de procesamiento interno para comprender y almacenar la expresión. Estas tareas incluyen la creación de un objeto de función lambda y la asignación de la expresión a ese objeto. Sin embargo, estas operaciones son extremadamente eficientes y su tiempo de ejecución se mantiene constante.

4.2.5. Definir e implementar heurísticas

El algoritmo A* requiere una heurística para un funcionamiento eficiente y efectivo.

Es fundamental que esta sea admisible, es decir, que nunca sobreestime el coste de alcanzar el objetivo. En caso contrario, el camino encontrado por el algoritmo podría no ser el camino de mínimo coste.

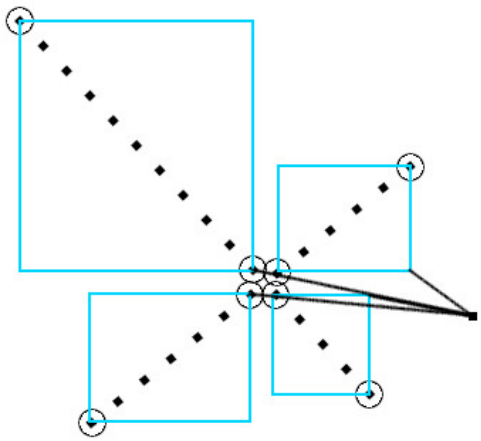

Figura 10: Distancias entre el punto y las *esquinas* más cercanas al punto

Si el peso de la arista es la distancia de recorrer el camino, la heurística utilizada devuelve la distancia en línea recta. Si el peso es el desnivel acumulado, devuelve el desnivel.

La heurística que devuelve la distancia en línea recta (también conocida como distancia euclidiana) entre dos puntos como estimación es siempre admisible debido a la relación entre la distancia en línea recta y la distancia recorriendo un camino. La propiedad de que la distancia en línea recta es siempre menor o igual a la distancia recorriendo un camino garantiza la admisibilidad de esta heurística.

La desigualdad triangular establece que en cualquier triángulo, la suma de las longitudes de dos lados es siempre mayor que la longitud del tercer lado. Aplicado a este contexto, es posible considerar el camino como los dos lados del triángulo y la distancia en línea recta como el tercer lado. La desigualdad triangular afirma que el tercer lado (la distancia en línea recta) no puede superar a la suma de los otros dos lados (la distancia recorrida a lo largo del camino).

La heurística que devuelve el desnivel entre dos puntos como estimación es siempre admisible debido a la relación entre el desnivel y el desnivel acumulado a lo largo de un camino. El hecho de que el desnivel sea siempre menor o igual al desnivel acumulado asegura la propiedad de admisibilidad.

El desnivel es la diferencia de altura entre dos puntos, mientras que el desnivel acumulado es la suma total de todos los cambios de altura a lo largo de un camino. Dado que cada cambio de altura contribuye al desnivel acumulado, el desnivel entre los puntos es inherentemente menor o igual al desnivel acumulado a lo largo de cualquier camino posible.

Por lo tanto, la heurística basada en desnivel nunca sobrestimará el costo real, cumpliendo así con la propiedad de admisibilidad.

En ambos casos, estas heurísticas cumplen con la propiedad de admisibilidad al proporcionar estimaciones que siempre son menores o iguales al costo real.

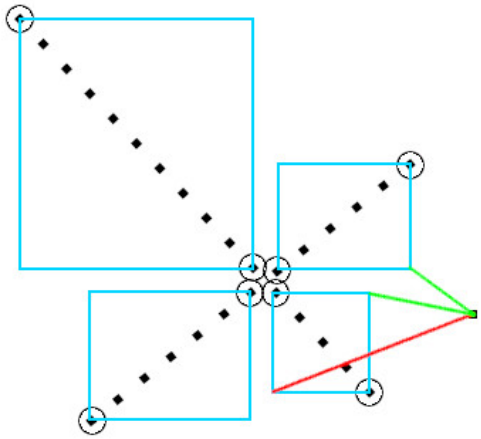

Figura 11: En rojo: Distancia umbral. En verde: Distancias entre el punto y las *esquinas* más cercanas al punto menores que la distancia umbral

La función lambda diseñada para calcular la distancia directa entre dos puntos emplea el método `geodesic`, que proporciona una estimación precisa de la distancia geodésica en línea recta entre los puntos.

La función lambda destinada a obtener el desnivel entre dos puntos calcula el valor absoluto de la diferencia entre sus elevación. Utilizar el valor absoluto garantiza que el resultado sea siempre positivo y refleje la magnitud del cambio en la elevación entre los puntos.

Como se ha explicado en el apartado anterior, la complejidad temporal de definir una función lambda en Python es constante.

4.2.6. Buscar camino mínimo

En este apartado se explica como encontrar la ruta de mínima distancia, mínimo desnivel o ambas, utilizando el algoritmo de Dijkstra o el algoritmo A*.

El usuario, a través de la línea de comandos, elige si buscar la ruta de mínima distancia, de mínimo desnivel o ambas. Por defecto, si el usuario no indica nada, se buscan ambas rutas. En relación al algoritmo utilizado, este también queda a elección del usuario, si bien el algoritmo A* es el elegido por defecto, dado que es el más eficaz entre los dos.

Para determinar el camino de mínimo coste, que abarca todos los puntos ingresados por el usuario, se procede a calcular los caminos mínimos que conectan cada punto y, posteriormente, se combinan.

Para buscar el camino mínimo utilizando el algoritmo de Dijkstra, se emplea el método `shortest_path` de la librería NetworkX. Por otro lado, para hacer uso del algoritmo A*, se utiliza el método `astar_path` de la misma librería.

El método `shortest_path` recibe un grafo, un nodo origen, un nodo objetivo y una función que indica cómo obtener el peso de las aristas en el grafo. Por otro lado,

el método `astar_path` recibe los mismos parámetros que `shortest_path` más una función heurística que estima la distancia entre un nodo y el nodo de destino.

Ambos métodos devuelven el camino como una lista de vértices. El programa debe recorrer las aristas que conectan estos vértices, extraer los caminos que unen estos puntos y combinarlos, para así obtener punto a punto el camino.

Si se buscaba el camino de mínima distancia y el camino de mínimo desnivel acumulado, se comparan, y si son idénticos, se descarta uno.

Como se expuso en la sección 2.1, la complejidad temporal del algoritmo de Dijkstra es $O((E+V) \cdot log(V))$ siendo V el número de vértices y E el número de aristas, y la del algoritmo A*, en el peor caso, es similar a la de Dijkstra, y en el caso promedio, es bastante mejor que la de Dijkstra.

Y, al igual que en las anteriores tareas, hay realizar esta operación por cada punto introducido por el usuario. Por tanto, la complejidad temporal total es la complejidad de Dijkstra o A* multiplicada por el número de puntos introducidos por el usuario, es decir, $O((E+V) \cdot log(V) \cdot p)$.

4.2.7. Construir archivo GPX y exportarlo

A partir del camino mínimo encontrado, que es una lista de puntos, y la lista de puntos introducida por el usuario se construye el objeto GPX y luego se escribe en un archivo.

La idea es plasmar el camino mínimo o caminos mínimos si se han buscado ambos como una pista o como dos pistas respectivamente. Y, además, añadir los puntos introducidos por el usuario como waypoints, de tal manera, que al visualizar el archivo GPX en algún visualizador, el usuario vea fácilmente los puntos que ha introducido y comprenda más fácilmente el recorrido de la pista.

Se utiliza el método `construir_gpx_con_waypoints_sin_ele`, implementado en el módulo gpx_auxiliar_methods, para construir el objeto GPX. Este método requiere una lista que contiene listas de puntos, es decir, una lista de pistas y una lista de puntos sin elevación, que servirá para crear los waypoints. Su funcionamiento para construir la pista es el siguiente: Primero se crea un objeto GPX. Luego, por cada pista recibida, se agrega un nuevo objeto GPXTrack al objeto GPX, se añade un nuevo objeto GPXTrackSegment al objeto GPXTrac y, finalmente, se agregan los puntos de la pista al objeto GPXTrackSegment. En cuanto a la adición de waypoints, simplemente itera sobre la lista recibida y los integra al objeto GPX como instancias de la clase GPXWaypoint.

Para escribir el objeto GPX en un archivo se usa el método `escribir_gpx` del mismo módulo. Este método recibe dos parámetros: El objeto GPX que se va a escribir en el archivo y el nombre del archivo GPX que se creará. El método opera de la siguiente manera: Primero, abre el archivo especificado en modo de escritura, luego, obtiene el contenido XML del objeto GPX, después, escribe este contenido XML en el archivo y, finalmente, cierra el archivo.

La operación que más tiempo consume es el recorrido de los puntos de la pista o pistas y su adición al objeto GPX. Esta operación tiene una complejidad de $O(n)$,

donde n representa el número de puntos de la pista. Si el usuario decide calcular la ruta de mínima distancia y mínimo desnivel acumulado, la complejidad será de $O(n + m)$, donde m es el número de puntos de la otra pista.

5. Evaluación

5.1. Pruebas

A medida que el proyecto ha avanzado en su implementación, se ha probado meticulosamente cada uno de los módulos y funcionalidades que lo componen. Estas pruebas se han llevado a cabo tanto de manera individual como en conjunto, garantizando que cada funcionalidad estuviera probada antes de proceder con la siguiente. Ha sido un proceso cuidadoso y meticuloso que ha tenido como resultado una notable reducción en la ocurrencia de errores y en la necesidad de reestructurar excesivamente el código.

5.1.1. Pruebas unitarias

Las pruebas unitarias son un tipo fundamental de pruebas de software que se centran en evaluar unidades individuales de código, como funciones, métodos o clases, de forma aislada. Las pruebas unitarias verifican que estas unidades funcionen correctamente en términos de lógica y comportamiento esperado.

En busca de garantizar la integridad y el rendimiento del software, se llevaron a cabo pruebas unitarias a lo largo de las diferentes etapas del proyecto. Aunque estas pruebas no siguieron un proceso estandarizado, desempeñaron un papel esencial al examinar las unidades de código más pequeñas y críticas del sistema.

Cada unidad de código fue sometida a escenarios de prueba variados y casos límite para evaluar su robustez. En particular, se puso énfasis en evaluar exhaustivamente los métodos de los módulos auxiliares.

A continuación, se muestran algunas de las pruebas unitarias para cada uno de los métodos del módulo gpx_auxiliar_methods.py. Estas son solo algunos ejemplos, pero hay muchas más pruebas unitarias para el resto de módulos del proyecto.

- `leer_gpx`:
  - Verificar que el objeto GPX devuelto contiene el nombre correcto.
  - Verificar que el objeto GPX devuelto contiene el número de puntos correcto.
  - Verificar el valor de algún punto específico.
- `leer_directorio_gpx`:
  - Comprobar que el número de objetos GPX en la lista coincide con la cantidad de archivos .gpx.
- `escribir_gpx`:

    - Verificar que el archivo se ha creado adecuadamente en el sistema de archivos. item Evaluar si el contenido del archivo coincide con el del objeto GPX.

- `construir_gpx`:
    - Verificar que el objeto GPX creado contiene en el segmento correspondiente la lista de puntos de la pista recibida
- `construir_gpx_con_waypoints`:
    - Verificar que el objeto GPX contiene tanto la pista como los waypoints.
    - Verificar algún waypoint específico, comprobando su latitud, longitud y elevación.
- `es_pista_circular`:
    - Verificar que la función devuelve verdadero para una pista circular.
    - Verificar que la función devuelve falso para una pista no circular.
    - Verificar que la función opera correctamente cuando recibe una pista de un solo punto.

Las pruebas unitarias permitieron identificar y abordar errores en etapas iniciales. Esta identificación temprana resultó en un ahorro significativo de tiempo y recursos, ya que los problemas se resolvieron antes de propagarse a otras partes del software.

5.1.2. Pruebas de integración

Las pruebas de integración son un tipo de pruebas de software que se enfocan en verificar la interacción y la cooperación entre diferentes componentes, módulos o sistemas en un sistema más grande. El objetivo principal de estas pruebas es asegurarse de que las partes individuales de un sistema funcionen correctamente cuando se combinan y se utilizan juntas. Las pruebas de integración son esenciales para identificar y resolver problemas que puedan surgir debido a la interacción entre componentes.

Aunque las pruebas formales de integración son ideales para garantizar que los componentes trabajen juntos de manera efectiva, las pruebas ad hoc y las pruebas a medida que se desarrolla el proyecto también pueden ser valiosas para identificar problemas tempranos y asegurar de que los componentes básicos funcionen correctamente.

Un ejemplo evidente de interacción y cooperación se da en el proceso de serialización y exportación del grafo por una parte del sistema y la importación y deserialización de dicho grafo por la otra parte. Sin embargo, es importante destacar que se detectó un fallo durante este proceso, ciertos atributos de las aristas, representados originalmente como tuplas, pasaban a ser listas durante el proceso de serialización y esta transformación provocaba que el grafo recuperado fuera distinto del original.

Afortunadamente, se pudo identificar la raíz de este problema de forma temprana gracias a las pruebas realizadas durante el proceso de desarrollo. Esta observación permitió tomar medidas inmediatas para solucionarlo y evitar perder tiempo valioso en etapas posteriores del desarrollo.

Además, es importante destacar que la capacidad de serializar el grafo en formato GPX y exportarlo no solo proporciona una utilidad valiosa para el usuario al permitir una visualización sencilla del grafo generado, sino que también beneficia al desarrollador. Esta funcionalidad permite verificar de manera eficaz si el grafo ha sido generado de manera correcta y acorde a los archivos fuente.

5.1.3. Pruebas de aceptación

Las pruebas de aceptación son un tipo de pruebas de software que se realizan para verificar si una aplicación cumple con los requisitos y expectativas del cliente o usuario final. El objetivo principal de estas pruebas es asegurarse de que el software funcione correctamente en escenarios del mundo real y que cumpla con los criterios de aceptación previamente establecidos.

Para comprobar de manera efectiva el correcto funcionamiento en un escenario real se ha tomado como fuente para generar el grafo una carpeta con 157 archivos GPX, cada uno de los cuales presenta, en promedio, una pista compuesta por 1300 puntos geográficos. Incluso ante la presencia de un conjunto de datos tan extenso, la generación del grafo se lleva a cabo con rapidez. Además, gracias al diseño del grafo, pensado para un enrutamiento óptimo, el cálculo de rutas se realiza también con gran rapidez.

No obstante, después de llevar a cabo pruebas de aceptación adicionales, se observó que el requisito no funcional "La interfaz de línea de comandos debe ser intuitiva y fácil de entender para los usuarios, con comandos y opciones claramente nombrados y explicados"no se estaba cumpliendo completamente. Aunque la interfaz en sí es intuitiva y fácil de entender, se notó que al utilizarla el usuario por primera vez, podría tener dificultades para comprender la estructura correcta de los comandos.

Para resolver este problema, se decidió agregar una sección llamada "Uso" en el archivo README del proyecto. En esta sección, se proporcionan ejemplos claros y explicaciones detalladas de casos de uso básicos, como generar un grafo, calcular una ruta entre dos puntos o calcular una ruta entre más de dos puntos. Esto mejora significativamente la comprensión y el uso correcto de la interfaz de línea de comandos para los usuarios que están utilizando el sistema por primera vez.

5.2. Rendimiento

Como se ha explicado a lo largo del documento, en este sistema la generación del grafo es una operación que se realiza con poca frecuencia, y una vez generado el grafo, se calculan múltiples rutas a partir de él sin necesidad de regenerarlo a menos que cambien las pistas fuentes.

Dado que la generación del grafo se realiza con poca frecuencia y su objetivo principal es proporcionar un grafo óptimo para el cálculo de rutas, la prioridad principal radica en maximizar la calidad del grafo, sin embargo, también es esencial analizar su eficiencia en términos de tiempo de ejecución.

Como se ha explicado anteriormente, en el peor caso, la generación del grafo tiene una complejidad temporal de $O(n^2 \cdot m)$, donde n es el número de pistas y m es el número promedio de puntos de cada pista. Sin embargo, gracias a la implementación

de técnicas de reducción del espacio de búsqueda, que en esta parte del proyecto consistía en buscar solo intersecciones entre los puntos de las pistas si el cuadro delimitador mínimo de estas se intersecaban, la complejidad real es mucho menor.

En la Figura 12 se puede observar el tiempo empleado en generar el grafo dependiendo del número de pistas o de archivos GPX. Es importante recordar que el número de pistas y el número de archivos GPX son equivalentes, ya que cada archivo GPX contiene una pista.

De esta prueba de rendimiento, se puede extraer una clara conclusión. El incremento en el tiempo empleado para generar el grafo no es lineal ni cuadrático. Esto es coherente con la complejidad temporal teórica del proceso, que tiende a ser cuadrática, según el número de pistas, en el peor de los casos, pero que, en la mayoría de los escenarios, es menor, oscilando entre lineal y cuadrática.

Por lo tanto, hemos confirmado que la técnica de reducción del espacio de búsqueda es efectiva y ahorra un tiempo valioso.

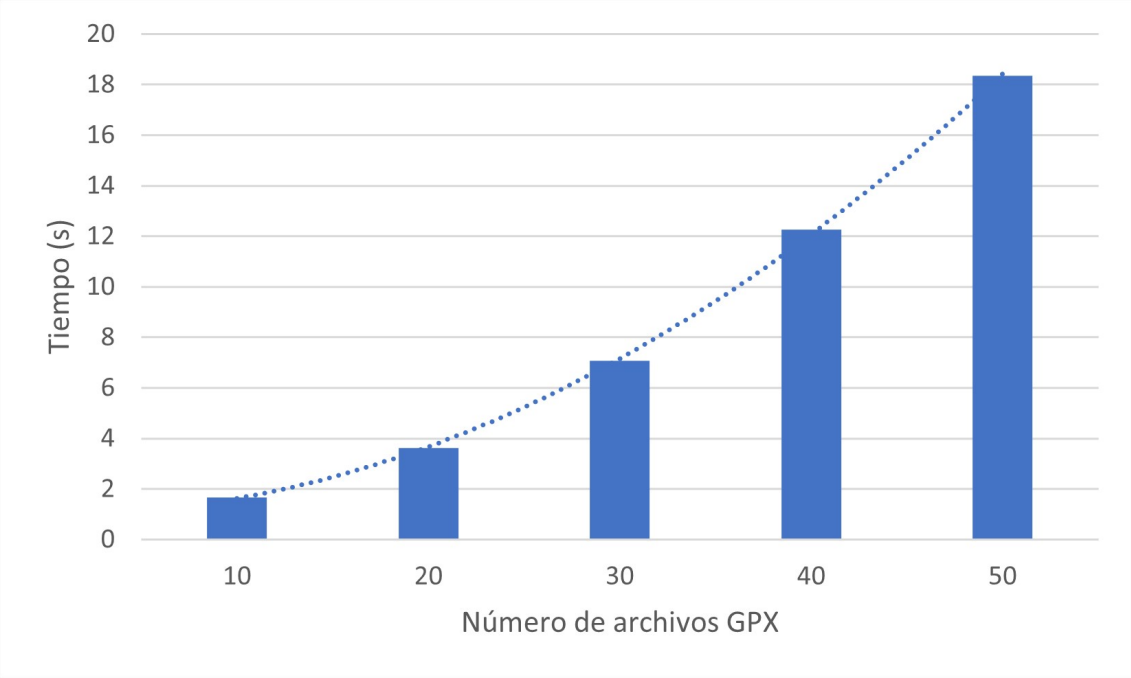

Figura 12: Tiempo empleado en generar el grafo dependiendo del número de archivos GPX

En la Figura 13 se puede observar el tiempo empleado en generar el grafo dependiendo del número promedio de puntos de cada pista.

Teóricamente, en el peor caso, el tiempo de ejecución aumenta de manera proporcional al número de puntos de las pistas. Sin embargo, gracias a la reducción del número de puntos utilizando el método `simplify` y a la búsqueda de intersecciones entre los puntos de las pistas solo si el cuadro delimitador mínimo de estas se intersecaban, se consigue una complejidad temporal menor.

El resultado de la prueba es coherente con la estimación teórica, ya que se puede observar que el tiempo no aumenta de manera lineal con el incremento lineal del número de puntos.

Esto respalda la utilidad de la reducción del número de puntos como una estrategia para mejorar el rendimiento de tu programa y vuelve a confirmar que la técnica de reducción del espacio de búsqueda es efectiva y ahorra tiempo.

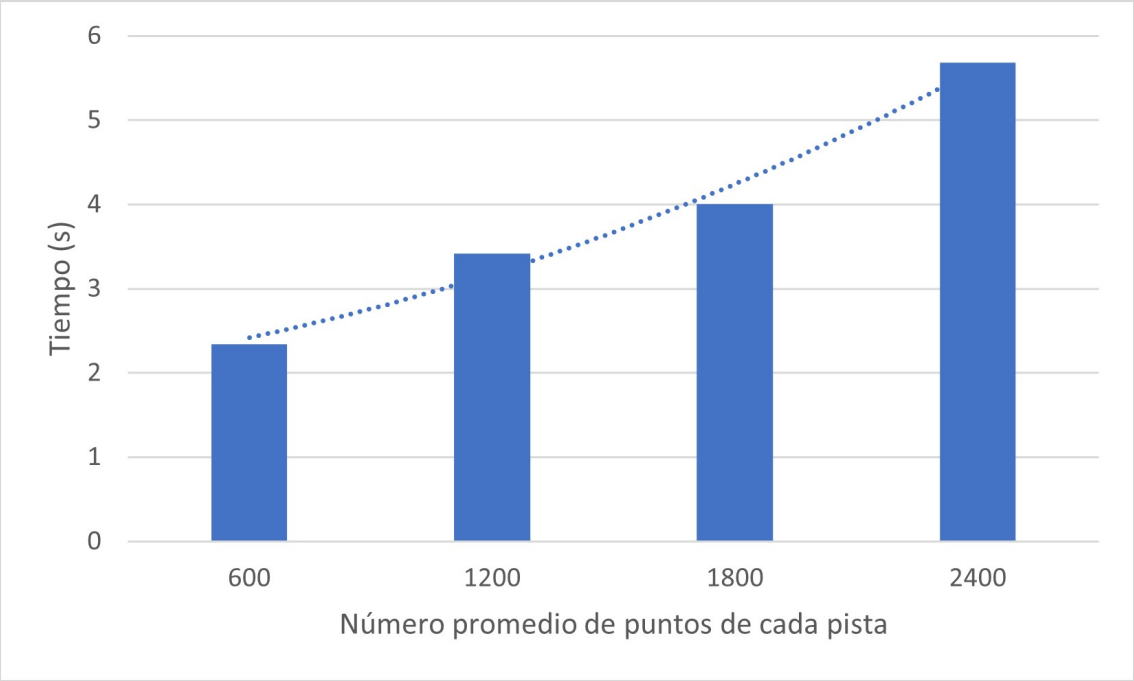

Figura 13: Tiempo empleado en generar el grafo dependiendo del número promedio de puntos por cada pista

Ahora es el momento de verificar si la estimación teórica de la complejidad temporal en la parte del cálculo de rutas se ajusta a la realidad.

La primera comprobación y la más sencilla es verificar si el tiempo de ejecución aumenta proporcionalmente al número de puntos introducidos por el usuario.

En la Figura 14 se muestra el tiempo empleado por el programa en calcular rutas dependiendo del número de puntos introducido por el usuario. A modo de recordatorio, si el usuario ingresa dos puntos, el primero es el punto de inicio y el último es el punto de fin. Cuando se ingresan más de dos, el primero y el último son idénticos, mientras que los demás son puntos de paso de la ruta.

El resultado de la prueba confirma la estimación, ya que se observa claramente que el tiempo de ejecución aumenta de manera proporcional al número de puntos introducidos por el usuario.

Y aunque el objetivo principal de esta prueba no era comparar los algoritmos Dijkstra y A*, se puede observar que el tiempo de ejecución de ambos es similar, a pesar de que teóricamente A* debería ser más eficiente.

Esto se debe a que el grafo es simple y no tiene muchas bifurcaciones o caminos alternativos, por tanto, el beneficio de utilizar un algoritmo de búsqueda informada como A* puede verse contrarrestado por el tiempo adicional requerido para calcular la estimación heurística en cada paso del algoritmo. En conclusión, en escenarios con grafos simples y rutas directas, el cálculo de la estimación heurística puede

no proporcionar una ventaja significativa en términos de tiempo de ejecución en comparación con Dijkstra.

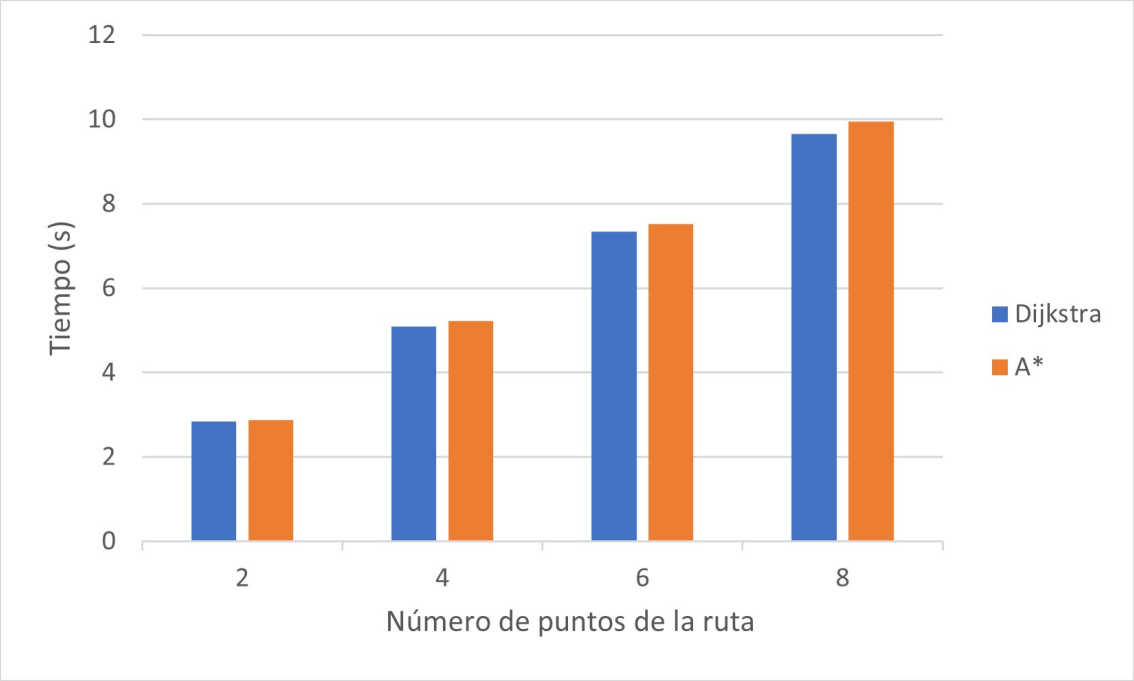

Figura 14: Tiempo empleado en calcular la ruta dependiendo del número de puntos de la ruta

La segunda comparación debe centrarse en observar el tiempo de ejecución en función de la complejidad del grafo, es decir, el número de vértices y aristas.

El problema radica en que no podemos elegir la complejidad del grafo de antemano, ya que esta está determinada por el número de pistas utilizadas para generar el grafo y la cantidad de intersecciones entre estas pistas.

Así que para evaluar el rendimiento del programa encargado del cálculo de rutas se va a utilizar grafos generados a partir de diferentes números de pistas.

En la figura 15 se puede observar el tiempo empleado en calcular la ruta dependiendo de la complejidad del grafo.

Se observa fácilmente que el tiempo de ejecución aumenta proporcionalmente a la complejidad del grafo aproximadamente.

Como recordatorio, la complejidad temporal está principalmente influenciada por dos tareas. En la mayoría de escenarios, la complejidad temporal de la primera es aproximadamente $O(n \cdot p)$, donde n es el número de aristas y p el número de puntos introducido por el usuario, y la de la segunda tarea es sustancialmente mejor que $O(((E+V) \cdot log(V)) \cdot p)$, donde E es el número de aristas y V el número de vértices.

Los resultados que hemos obtenido en esta prueba confirman las estimaciones teóricas mencionadas anteriormente.

En conclusión, a pesar de la complejidad temporal inherente a este problema, se han aplicado las técnicas apropiadas para desarrollar un programa altamente eficiente.

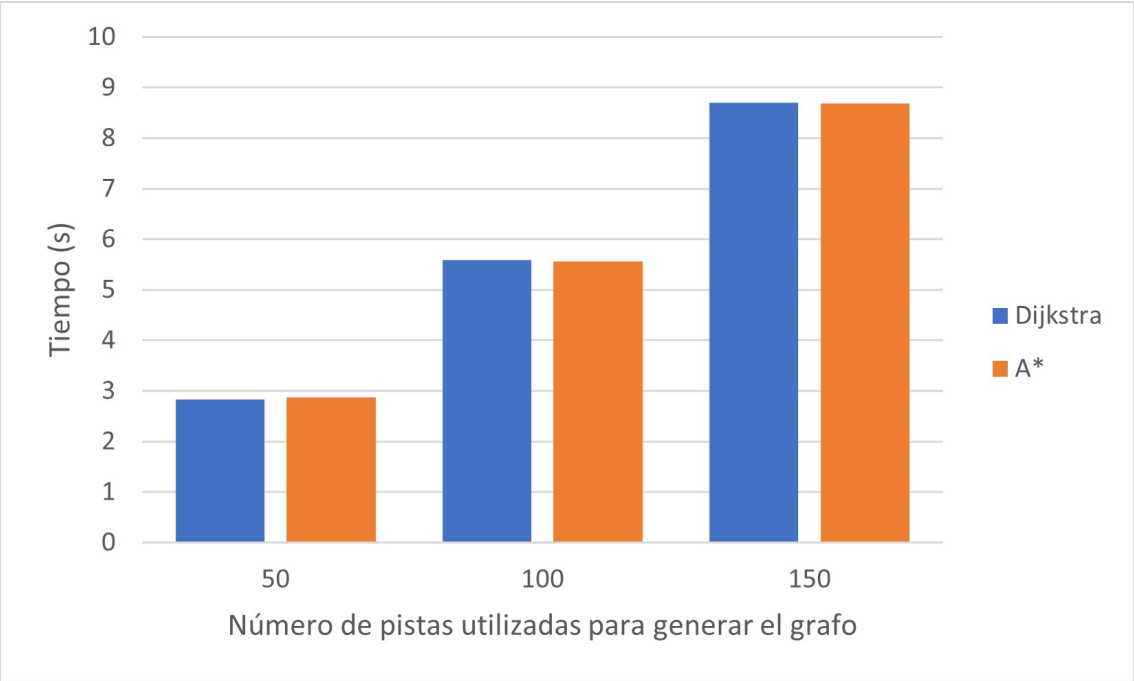

Figura 15: Tiempo empleado en calcular la ruta dependiendo de la complejidad del grafo

Esto se refleja de manera sólida en las pruebas de rendimiento realizadas, donde los resultados respaldan el esfuerzo dedicado a la optimización del software.

6. Conclusiones y trabajos futuros

6.1. Conclusiones

Los dos objetivos que se establecieron inicialmente eran: poder construir un grafo a partir de una serie de rutas en formato GPX y poder crear una ruta utilizando el grafo y una lista de puntos introducida por el usuario. Ambos objetivos se han alcanzado con éxito, lo que demuestra la plena realización de este proyecto.

Además, la aplicación de técnicas apropiadas ha resultado en un sistema que no solo es funcional, sino también es rápido y ágil. Esta eficiencia agrega un valor adicional a nuestra solución, beneficiando a los usuarios al ahorrar tiempo y recursos.

A nivel personal, este proyecto ha sido realmente gratificante. No solo ha contribuido al aprendizaje adquirido durante mis estudios de grado en la universidad, sino que también ha supuesto un gran desafío. Estoy contento de haber realizado este proyecto y de haber tenido la oportunidad de poner en práctica mis conocimientos en un entorno real.

6.2. Trabajos futuros

A pesar de haber alcanzado exitosamente los objetivos planteados en este proyecto, existen aspectos que se podrían incluir y que aportarían un valor adicional al proyecto.

En particular, una de las principales áreas de desarrollo futuro sería la implementación de una interfaz gráfica de usuario. Esta adición sería altamente beneficiosa, ya que simplificaría la interacción del usuario con el sistema. En lugar de ingresar los puntos de la ruta por medio de la consola de comandos, el usuario podría seleccionarlos de manera intuitiva haciendo clic en un mapa interactivo. Además, esta interfaz gráfica permitiría la visualización de la ruta generada sobre el propio mapa.

Otro aspecto clave que merece atención futura es la optimización en el proceso de generación del grafo. Por ejemplo, en situaciones donde dos segmentos de pistas comparten el mismo tramo, podría ser altamente beneficioso almacenar solo la información del camino medio, en lugar de almacenar ambos. Esta mejora simplificaría el grafo resultante, lo que podría llevar a cálculos de rutas más rápidos.

Referencias

[1] M. Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE Transactions on Computers*, 47(2):263–, 1998.

[2] Sean Gillies. *The Shapely User Manual*, 2023.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[4] Bailey A. Hanson and Christopher J. Seeger. Apps for collecting gps tacks: mytracks - the gps - logger. *Iowa State University Digital Repository*, 2015. Extension and Outreach Publications, 179.

[5] Masoud S. Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the * (star) search algorithms: Characteristics, methods and approaches - ti journals. *World Applied Programming*, 2012.

[6] Ibrahim Abayomi Ogunbiyi. How the python lambda function works - explained with examples. https://www.freecodecamp.org/news/python-lambda-function-explained. Accedido el 17 de agosto de 2023.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 3rd edition, 2009.

[8] Ana Pozo Ruz, Adela Ribeiro, Maria C. García-Alegre, Leonardo Rejon Garcia, Demetrio Guinea, and Francisco Sandoval Hernández. Sistema gps: descripción, análisis de errores, aplicaciones y futuro. *Instituto de Automática Industrial, Consejo Superior de Investigaciones Científicas*, 2000.

[9] M. Gloria Sánchez Torrubia and Víctor M. Lozano Terrazas. Algoritmo de dijkstra. un tutorial interactivo, 2001.

[10] Wikipedia contributors. Ramer–douglas–peucker algorithm. https://en.wikipedia.org/w/index.php?title=Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm&oldid=1167822259. Accedido el 21 de agosto de 2023.

[11] Robin J Wilson. *Introduction to Graph Theory*. Prentice Hall, fourth edition, 1996.