Escuela Politécnica Superior

Planificación de trayectorias para un robot de exteriores.

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Autor:
Alberto Joaquín López Sellers

Tutor/es:
Francisco Andrés Candelas Herias

Universitat d'Alacant
Universidad de Alicante

Enero 2021

Índice

Índice de figuras

Índice de tablas

1. Introducción

A lo largo de los siglos el ser humano ha tendido al uso de herramientas con la intención de ayudar y facilitar la labor del trabajo a realizar. Esas herramientas, permitían una mejora de tiempo y esfuerzo en las tareas beneficiándose de una mayor productividad. En la evolución de las herramientas, apareció un proceso de mecanización que permitiera el uso de máquinas en la tarea, mejorando así de forma considerable la productividad en muchas actividades obteniendo una mejoría considerable de tiempo y esfuerzo.

Actualmente se siguen elaborando herramientas y máquinas que ayuden en el desarrollo de distintas actividades, pero a eso se ha sumado un concepto más, la automatización. La automatización permite que mediante un conjunto de procesos informáticos, mecánicos y electromecánicos se operen distintas actividades con una mínima o nula intervención del ser humano. Es por eso que muchos de los procesos a nivel industrial que se realizan hoy en día, tienen una gran parte del proceso automatizada ya que según se ha demostrado mejora la productividad y los beneficios.

Aunque en nuestro entorno sea muy común ver procesos automatizados, aún queda mucho por realizar. Hay ciertas tareas en la automatización que para el ser humano pueden resultar sencillas, pero trasladarlo a una máquina puede ser complicado. Como veremos en este documento, se mostrará un ejemplo de automatización de una tarea tan sencilla a nivel humano, como es ir desde una posición dada en un mapa a otra posición del mismo por el camino más corto, y como a nivel de máquina puede ser complicado, además de todas las implicaciones que tiene automatizar dicha tarea.

1.1. Objetivos

El principal objetivo de este proyecto es la creación de un sistema que sea capaz de hacer una planificación de los caminos que debe seguir un robot, a partir del cálculo de caminos entre dos posiciones. Para conseguir esto se hace uso de información externa ofrecida por el proyecto de OpenStreetMap.[1]

Este proyecto pretende ubicarse dentro del proyecto BLUE, que básicamente consiste en un robot móvil para exteriores capaz de moverse en entornos no estructurados. [2]

Figura 1.1: Robot BLUE

Para llevar a cabo el objetivo general será necesario abordar las siguientes tareas que también pueden verse como objetivos concretos:

- ❏ Estudio de los sistemas y entornos que faciliten el cálculo de caminos.
- ❏ Estudio de los algoritmos necesarios para el cálculo más eficiente y preciso.
- ❏ Uso de OpenStreetMap como fuente de mapas e información etiquetada en los mismos.

- Adecuación del entorno para facilitar su uso en aplicaciones reales de robótica, mediante la correcta definición de APIs, y el desarrollo de módulos cuya ejecución sea transparente al sistema que los usa.
- Implementación de la planificación de caminos como un servicio que pueda ser utilizado desde clientes remotos, que pueden ser procesos ejecutados a bordo de robots móviles.
- Realización de pruebas que cumplan los objetivos previstos.
- Optimización del sistema para cumplir tiempos de procesamiento adecuados para aplicaciones de robótica en tiempo real.

2. Marco teórico

Antes de comenzar en el desarrollo del proyecto, es necesario un estudio previo de los aspectos importantes que van a conformar el proyecto. Para ello, se ha hecho un estudio de distintos sistemas y algoritmos posibles para la realización del proyecto y su implantación.

2.1. Estudio del proyecto donde se aplicará el desarrollo.

El proyecto BLUE es un proyecto con un desarrollo a varios niveles [3]. Este proyecto tiene como finalidad desarrollar un sistema capaz de hacer que el robot móvil se mueva de forma autónoma en entornos no estructurados. En la figura 2.1 se muestra un diagrama de los distintos procesos que tiene el sistema de este proyecto.

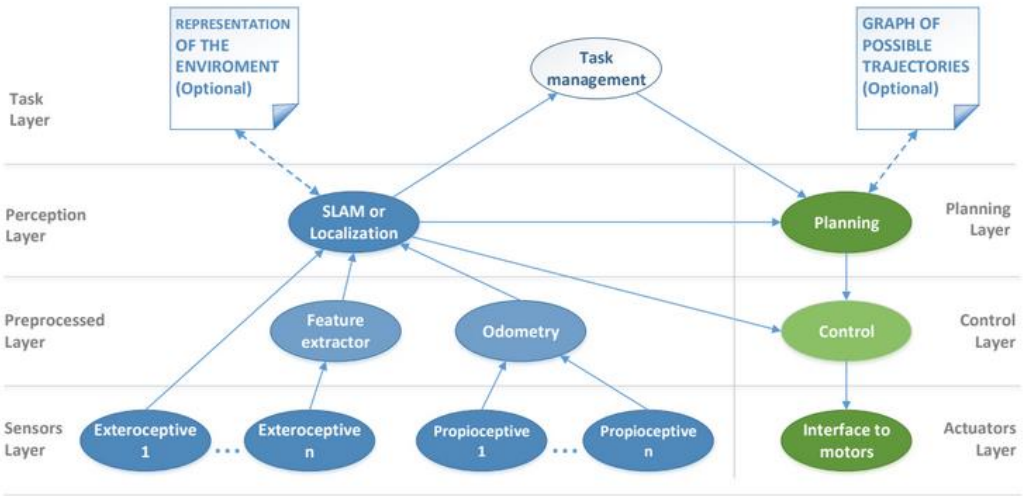

Figura 2.1.: Diagrama BLUE

Entre estos procesos, podemos ver algunos cuya finalidad es centrarse en obtener una localización precisa y en tiempo real del robot, otro tiene la finalidad de generar un control que determine cómo debe moverse el robot, o incluso la detección de objetos del entorno. Durante este documento se va a abordar el desarrollo de uno de estos procesos, más concretamente el de *planning*.

Este proceso de *planning* será el encargado de calcular los caminos que debe recorrer el robot. Para hacer una integración más fácil, se desarrollará una biblioteca que pueda ser utilizada por el proceso de *planning*, la cual determinará cuál es la siguiente posición a escoger para poder minimizar la distancia al punto final.

Una vez todo integrado, se hacen simulaciones para comprobar que todo funciona como debería. En la figura 2.2 se muestra un ejemplo de simulación.

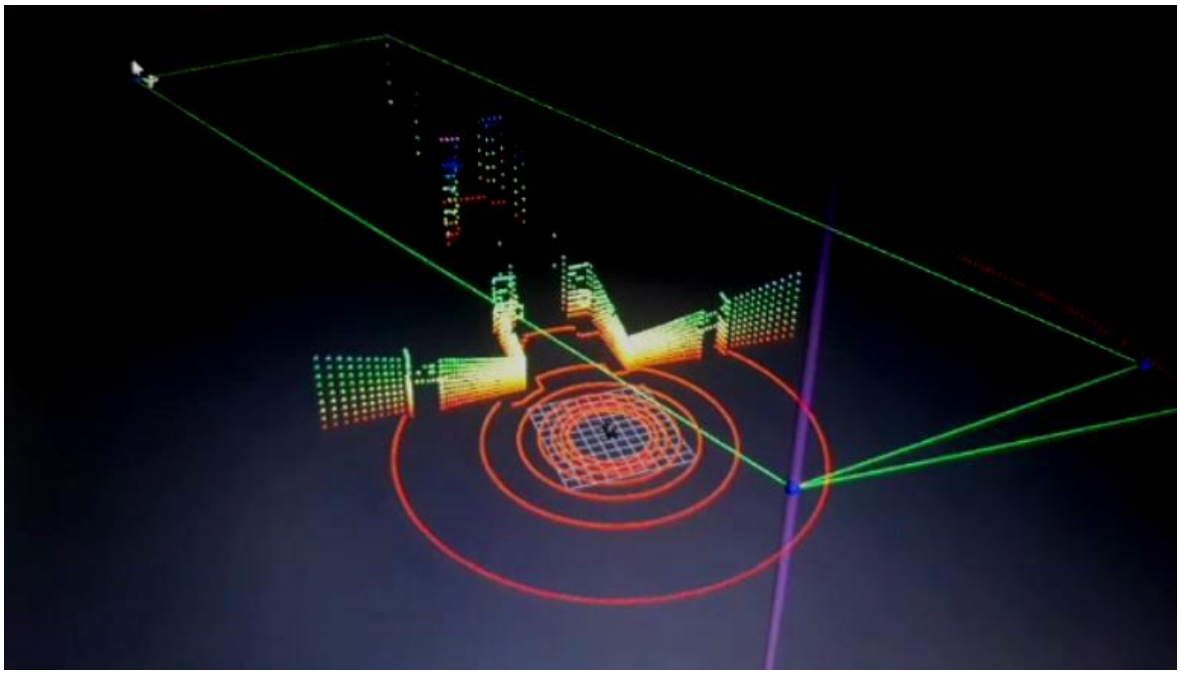

Figura 2.2.: Ejecución de la simulación

Tras comprobar que todo es correcto en la simulación, se comienza a probar en entornos reales cargando el sistema en el robot.

2.2. Estudio de la idea del proyecto

El proyecto tiene como objetivo la implantación de un sistema que calcule el camino más corto. Para ello, es necesario tener en cuenta que este sistema se va a ubicar dentro del proyecto BLUE y con ello es necesario desde un comienzo facilitar la integración.

Antes de iniciar con el proyecto se clarificaron varios aspectos a tener en cuenta con el sistema para permitir usarlos en el sistema BLUE. Por ello el sistema debía adecuarse a las siguientes posibilidades de uso:

- Uso en entornos de alto nivel donde el mapa a recorrer podría ser de aproximadamente un kilómetro cuadrado.
- Uso en entornos de más bajo nivel donde el mapa a recorrer sea de alrededor de unos veinte metros cuadrados.
- Uso de sistema UTM para el cálculo de posiciones en el sistema.
- Posición de inicio y final del robot desconocidas. Debido a que el sistema a usar es un modelo de grafo, puede darse la posibilidad que las posiciones de inicio y fin estén fuera de la parte del grafo por lo que habría que adecuarlo para permitir su recorrido en entornos desconocidos para el grafo.

Una vez establecidos los aspectos que el sistema tendrá que realizar, se procede a un estudio de las tecnologías o entornos a usar.

2.3. Estudio del entorno de programación

Debido a que pretende ubicarse el sistema dentro de un entorno con un desarrollo ya empezado, las opciones de elección se limitan a las que permitan una fácil integración. Los requisitos de entorno principales son:

- ❏ Uso de la herramienta de Github como controlador de versiones.
- ❏ Uso del lenguaje de C++ para programar.
- ❏ Uso de Eclipse como entorno de desarrollo.
- ❏ Uso del inglés como idioma en el desarrollo.

Una vez establecidos estos requisitos, se valoró distintas opciones para la integración en el sistema. Como el *framework* de desarrollo usado es ROS [4], se valoró la posibilidad de implementar un paquete de ROS que incluyera las funcionalidades del sistema de cálculo de caminos, aunque finalmente se decidió realizar una biblioteca fuera de ROS. Esta elección permite independizar aún más todo el código incluido en la biblioteca para permitir su uso desde distintos paquetes de ROS.

Para todo el desarrollo, tal y como establecía en uno de los requisitos, se ha almacenado y organizado el código en un repositorio de github accesible públicamente [12]. En el mismo repositorio se explica como instalar y hacer funcionar la biblioteca, que básicamente se compone de tres carpetas principales:

- **includes:** incluye los archivos '.h'.
- **src:** incluye los archivos '.cpp**'.**
- **tests:** incluye los archivos necesarios para hacer los *tests*.

Además de esas carpetas también se incluye un archivo llamado **CMakeLists.txt** que ayuda a compilar y a instalar la biblioteca tal y como se indica en la descripción situada en el mismo repositorio.

Una vez se ha definido como llevar a cabo el desarrollo de una biblioteca para el sistema de caminos más cortos, se procede al estudio de los distintos algoritmos a incluir en el sistema.

2.4. Estudio de la herramienta OpenStreetMap

OpenStreetMap es un proyecto colaborativo, el cual tiene como intención crear mapas con gran cantidad de información del entorno. Este proyecto ofrece información de carácter libre, y que los usuarios, de forma colaborativa, pueden ampliar ofreciendo y modificando información. Para obtener esa información, OpenStreetMap ofrece una herramienta que se utiliza mediante la web, y que permite seleccionar el área de la que se quiere obtener información (Figura 2.3).

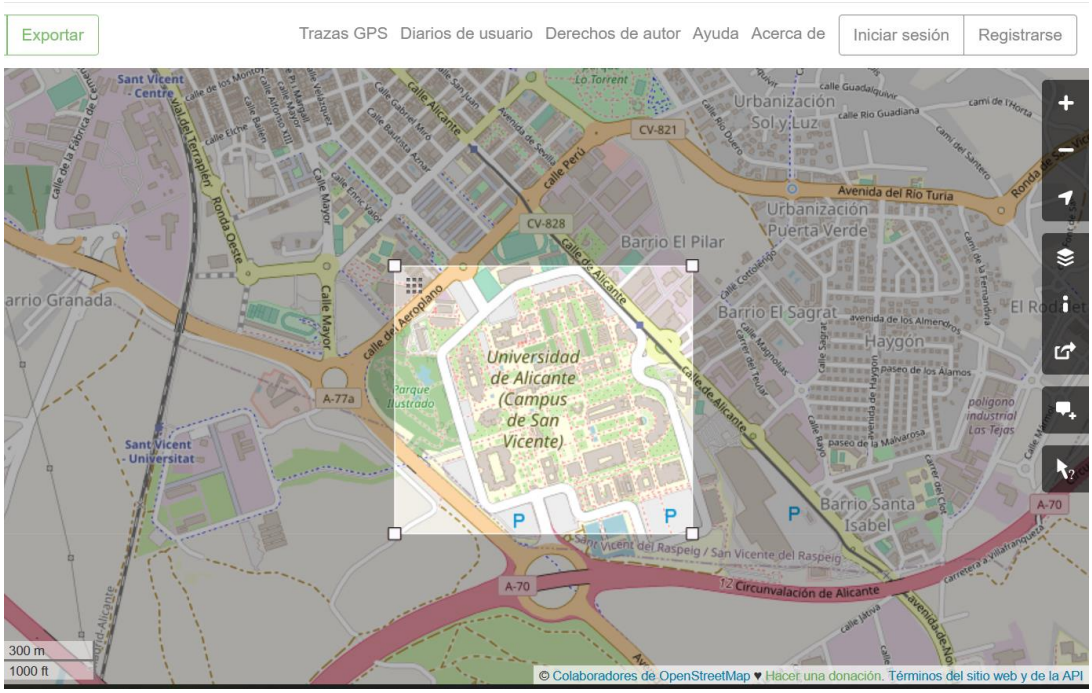

Figura 2.3: Selección de la información de openStreetMap

Esta herramienta ofrece información como elementos en el mapa (arboles, edificios, esculturas, intersecciones…), su latitud y longitud o conexiones entre ellos. Toda esta información se ofrece en forma de un fichero XML que la contiene de forma estructurada (Figura 2.3). Esa información se estructura en forma de grafo, donde cada elemento es un nodo y se identifica con un *tag* en el caso de ser un elemento del entorno y no una intersección.

Figura 2.4: Ejemplo de fichero XML

Por último, el fichero XML también ofrece una relación entre los elementos del mapa. Para ello, OpenstreetMap define un sistema de caminos que relaciona los distintos elementos del mapa, e indica cuales son los nodos a recorrer para completar un camino. Esto ofrece la posibilidad de adaptar toda esta información al grafo que se quiere desarrollar en la biblioteca, y así poder aplicar los distintos algoritmos que permitirán obtener el camino más corto entre dos posiciones.

2.5. Estudio de los algoritmos a utilizar

El uso del algoritmo más apropiado es uno de los aspectos más importantes para el buen funcionamiento del sistema. Para la elección del algoritmo a utilizar había que tener en cuenta que se ajustara a los requisitos ya establecidos, y una vez habiendo confirmado que se iba a usar una estructura de grafos, no fue difícil establecer el algoritmo.

El algoritmo de Dijkstra [5] (o algoritmo del camino mínimo) cumplía a la perfección lo que se estaba buscando, por lo que desde un principio se tuvo en cuenta para su implantación. Dicho algoritmo permite la obtención del camino mínimo entre dos posiciones del grafo una vez se hayan asignado los pesos en las aristas.

En un principio, se predijo también la posibilidad de que los tiempos de ejecución del algoritmo de Dijkstra pudieran ser un poco grandes para la aplicación. Para solventar esto, se estudió la posibilidad del uso del algoritmo A*, aunque la implementación del mismo en el sistema se aplazó hasta después de obtener los primeros resultados de aplicar el algoritmo de Dijkstra que permitieron evaluar sus tiempos.

El algoritmo A* [6], a diferencia de Dijkstra, es un algoritmo que no necesita recorrer todos los caminos para obtener el camino mínimo. Este algoritmo hace una estimación de cuál va a ser el camino mínimo mediante una función heurística como:

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

En esta fórmula hay que tener en cuenta que:

- n: Es el nodo a ser evaluado.
- $g(n)$: Representará el coste del camino recorrido desde la posición inicial, hasta el nodo que se está evaluando. Esta distancia es la suma de todas las

distancias entre los nodos del camino, hasta llegar al nodo que se está evaluando.

- $h'(n)$: Representará el valor obtenido al aplicar la heurística. Para obtener el valor se realizará el cálculo de la distancia desde el nodo a evaluar hasta el nodo final sin tener en cuenta los nodos intermedios.
- $f(n)$: Es la suma de los valores obtenidos de $g(n)$ y $h'(n)$. Cuanto menor sea este valor, más favorable es el nodo evaluado para ser el camino más corto al destino.

2.6. Estudio del sistema de pruebas

En todo desarrollo es necesario que siempre se pruebe el código programado para asegurar su correcto funcionamiento. Además, esta biblioteca se ha ido realizando de forma evolutiva, lo que ocasionaba que fuese necesario ir retocando elementos ya programados para mejorar y aumentar las funcionalidades de las que dispone la biblioteca. Todo esto, hace que sea necesario el uso de pruebas unitarias que verifiquen en todo momento el correcto funcionamiento de todos los métodos programados.

Para la implementación de los *tests* unitarios se ha hecho uso de GoogleTest [11]. GoogleTest es una biblioteca de pruebas unitarias para el lenguaje de programación C++ que implementa gran cantidad de funcionalidades que facilitan el control del código programado.

2.7. Estudio del entorno para la API

En el trabajo también se considera crear un entorno que aloje la biblioteca de planificación desarrollada, al cual se pueda acceder de forma remota a través de una API. La intención de esta parte es la de permitir establecer en un entorno remoto toda la biblioteca desarrollada para planificar rutas, y así ofrecer la posibilidad de obtener la siguiente posición desde un entorno distinto con respecto al entorno donde se aloja la biblioteca. Las ventajas que puede ofrecer tener la biblioteca en remoto son:

- Mayor independencia con otros paquetes del desarrollo del BLUE.
- Facilita la compatibilidad con otros sistemas o robots.
- Facilita la implementación.
- Permite una ejecución más rápida y eficiente del planificador.

La ejecución de la biblioteca en un servidor remoto permite destinar los recursos del robot a otras tareas que tiene que realizar, como la detección del entorno o la evitación de obstáculos. El servicio remoto también evita la necesidad de incorporar hardware más potente, lo que también puede influir en un ahorro de la batería del robot.

Considerando las ventajas que se desean conseguir, se buscó la mejor manera de poder ofrecer este servicio en remoto analizando las características de distintos entornos disponibles para crear servicios web. Entre todos los entornos estudiados que permitieran ofrecer un servicio web de una forma acorde a los planteamientos del proyecto, destacan dos principalmente: NodeJS y Glove.

- NodeJS [9]

NodeJS es un entorno en tiempo de ejecución multiplataforma, de código abierto, que permite de forma sencilla la creación de servicios web para poder programar una API. Las ventajas que ofrece este entorno es la gran cantidad de documentación

de la que dispone, la facilidad que ofrece a la hora de configurar y generar el servicio web, y que usa una licencia MIT, la cual no tiene muchas limitaciones a la hora de la reutilización. NodeJS usa como lenguaje de programación JavaScript, además de disponer de un sistema de gestión de paquete (**npm**) muy eficiente y útil para poder hacer este tipo de trabajos en entornos web. Además de todo esto, NodeJS dispone de formas de cargar código C++.

Aunque en un principio las ventajas parecían muchas, fue necesario realizar la instalación para ver los inconvenientes que había a la hora de compatibilizar la biblioteca en C++ con NodeJS. Tras la instalación y puesta en funcionamiento de una API Rest que atendiera peticiones, fue en el momento de introducir la biblioteca cuando se produjeron los fallos al combinar ambos lenguajes. Debido a que la biblioteca usa tipos complejos de objetos (Nodos, Grafos, Links…) dificulta en gran medida la traducción entre lenguajes de estos objetos, por lo que al final resultaba muy tedioso realizar un simple paso de mensajes. Finalmente, y tras ver el esfuerzo que implicaba algunas de las tareas que deberían resultar sencillas como el paso de mensajes, se optó por buscar una alternativa para ofrecer la API.

- Glove [10]

Glove es una biblioteca de código abierto en C++, con una licencia de tipo MIT, y que permite la creación de una *webAPI* de forma sencilla. Aunque no ofrece tantas funcionalidades como puede ofrecer NodeJS, si ofrece las funcionalidades necesarias para la creación de una *webAPI* desde la cual se pueden tramitar solicitudes de forma remota. La principal ventaja que ofrece Glove, es que no se necesita una transformación de información entre lenguajes, lo cual facilita en gran medida su uso. Además, dispone de herramientas que ayudan a la traducción de los mensajes que se envían a través de la red, permitiendo que la compatibilidad sea mayor y que se pueda adaptar al sistema usado en la biblioteca en unas pocas líneas de código.

3. Metodología

Durante este apartado se expondrá la metodología usada para abordar el proyecto. Esta información explica cuál ha sido el camino escogido y cómo se plantearon los desarrollos una vez se hubo establecido todos los elementos estudiados en el apartado anterior.

3.1. Diseño interno (biblioteca)

Como estructura principal de los objetos de la biblioteca se escogió una tipología de grafo, la cual dispone de nodos (o *nodes*) y enlaces (o *links*). Esto fue diseñado de esta forma debido a dos aspectos clave:

1. El sistema previo ya tenía una estructura representada en forma de grafo para definir las posiciones por coordenadas.
2. La aplicación de OpenStreetMaps, además de otros ejemplos de diseños que usan representación de posiciones en el espacio, también usa un diseño a partir de grafos debido a su comodidad para la interacción entre los elementos.

Para este diseño en forma de grafos hay que tener en cuenta objetos clave que van a permitir la programación del algoritmo. Estos objetos definidos son:

- **Position:** Este objeto va a representar una posición dentro del espacio. Esta se va a representar mediante coordenadas **x,y,z**. Hay que tener en cuenta también que estas posiciones son transformaciones de las coordenadas geográficas (latitud, longitud y altitud) a coordenadas cartesianas (**x,y,z**). Además, hace uso de una variable llamada **yaw** para representar el ángulo de giro para llegar al siguiente objetivo del camino.
- **Node:** El grafo contendrá un conjunto de nodos representados por este tipo de objetos, los cuales serán los que contengan la mayor parte de información del grafo. Este elemento junto al objeto position representará la ubicación de una intersección en el mapa. Este elemento permite establecer posiciones de

"interés" en el mapa para establecer caminos de cambio de ruta por parte del robot.

- **Link:** Este objeto es el elemento que permite la relación entre nodos. Así, un objeto link permite conectar dos nodos del grafo entre sí, de forma que se pueda representar el tramo de la ruta que hay entre los dos nodos, y para ello un link mantiene dos nodos.
- **Planning_graph:** Este objeto representa la unificación de todos los elementos anteriores. Es un objeto más abstracto ya que tiene la funcionalidad de conectar los elementos entre sí y de realizar los cálculos de *planning* para la obtención del camino más corto. En este objeto se incluyen los algoritmos para el cálculo del camino más corto, los métodos para la creación e inclusión de los nodos y links en el grafo o los cálculos de distancias.
- **Graph:** Este objeto representa la interfaz con la que se va a comunicar el usuario. Este objeto incluye los métodos para poder importar el mapa con el trabajar, configuración del algoritmo (Dijkstra o A*) o tipos de distancias a usar (Mahalanobis o Euclídea) y para el cálculo del camino más corto.
- **Util**: Este objeto permite almacenar funcionalidades de un ámbito más general, para que pueden ser usadas por cualquiera de los objetos anteriores. Algunos métodos incluidos en este objeto son los que permiten generar de forma simple un vector para representar las coordenadas o una matriz de vectores para las matrices de covarianza

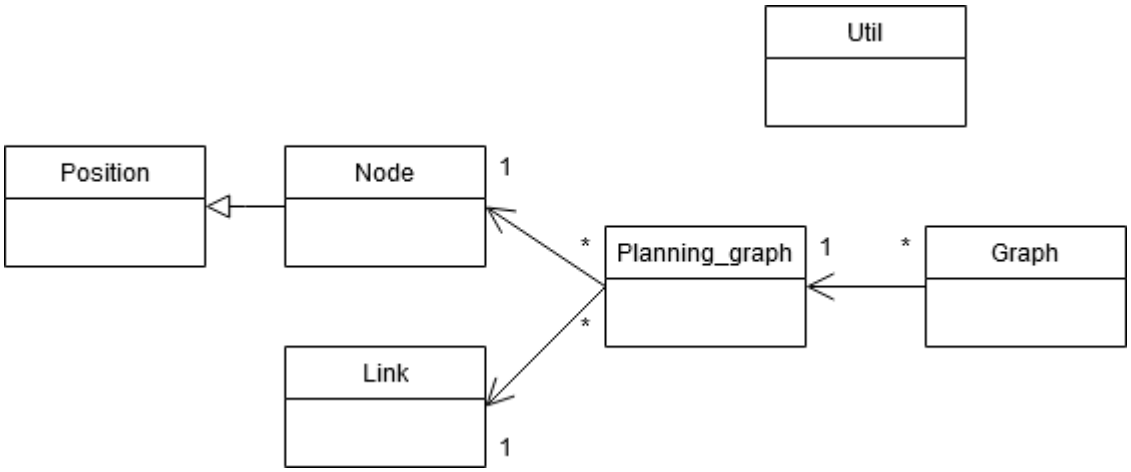

Figura 3.1.: Diagrama de clases de los objetos

Con todos estos objetos podremos obtener una representación como la mostrada en el ejemplo de la figura 3.2 de lo que sería el grafo. En el ejemplo se muestran nodos con sus coordenadas representados como marcas azules, mientras que los enlaces que conectan los nodos se representan con líneas rojas. La unión de todo esto se vería incluida en el **Planning_graph**.

Figura 3.2.: Representación del grafo en un mapa

Un enlace entre nodos representará una distancia que puede ser de uno de estos dos tipos:

- **Distancia Euclídea:** Es la distancia "ordinaria" la cual se calcula a partir del teorema de Pitágoras. [7]
- **Distancia Mahalanobis:** Este tipo de distancia es usada para determinar la longitud entre dos posiciones teniendo en cuenta el posible error que puede generarse al obtener las posiciones desde un receptor del GNSS. [8]

Para el cálculo del camino más corto se aplican dos tipos de algoritmos:

- **Dijkstra:** Este algoritmo, también llamado algoritmo de los caminos mínimos, es usado para calcular la distancia mínima entre dos nodos del grafo. Este algoritmo aunque siempre obtiene el mejor camino, puede ser un poco lento y por eso se combina con el siguiente algoritmo.

- **A*:** Para las ocasiones donde se necesita una mayor velocidad computacional, se ha hecho uso también del algoritmo A*, donde se evita pasar por todos los nodos aumentando la velocidad de computación del sistema.

Estos algoritmos se han combinado con una funcionalidad de inserción al grafo. Esta funcionalidad permite detectar en qué posición se encuentra actualmente el robot y busca si hay un nodo óptimo por el que introducirse al grafo. Así mismo, también se hace un cálculo para obtener el nodo óptimo para salida del grafo. Una vez se obtiene cual es el nodo de entrada y salida, se aplica el algoritmo para obtener el camino más corto. Esta funcionalidad también tiene detalles como tener en cuenta que la posición inicial y final deben ser posiciones representadas por algún nodo del grafo, o también si es conveniente entrar en un nodo previo al de salida, o es mejor ir directamente a la posición final.

3.2. Diseño de pruebas (tests)

Para el diseño de las pruebas hay que tener en cuenta que hay que comprobar casi todo el código del que dispone la biblioteca de *planning*. Para ello, se realizará una metodología de trabajo que implica probar el código una vez desarrollado. Debido a los elementos de los que se compone la biblioteca, se ha enfocado un sistema de pruebas que contendrá cuatro ficheros donde se ubicaran las pruebas para cada uno de los objetos de la biblioteca. Los ficheros se organizan de la siguiente forma:

1. **Node_test**: Probará el correcto funcionamiento del objeto **position** y **node**. También probará que las conexiones creadas por el objeto **link** son correctas.
2. **Planning_graph_test:** Probará todos los métodos de **planning_graph**.
3. **Graph_test:** Probará todos los métodos de **graph**.
4. **Util_test:** Probará los métodos del objeto **util** para comprobar que funcionan correctamente.

Como ya se ha mencionado, por cada método escrito será necesario hacer una prueba unitaria que verifique su correcto funcionamiento. Además, para facilitar la ejecución de los *tests*, se realizará un pequeño *script* que facilite esto. Este *script* se encargará de la compilación y ejecución del *test* seleccionado con una sola línea de comando.

3.3. Diseño externo (API)

La intención del diseño de la API es la de calcular de forma remota la siguiente posición del camino a seguir. La primera versión que se realizará durante este desarrollo no incorporará todas las funcionalidades que ofrece la biblioteca. Las funcionalidades que se buscan hacer que funcionen para la API son las que permiten obtener la siguiente o siguientes posiciones respecto a una posición inicial y un destino final. Además, se incorporarán las funcionalidades que permiten elegir el tipo de algoritmo y distancias a utilizar.

Con ese planteamiento, las funcionalidades que permitirá serán:

- Obtener la siguiente posición de la ruta.
- Obtener una lista de las siguientes posiciones de la ruta.
- Configuración del sistema para elegir el tipo de distancia a usar para los cálculos de distancia. Esto serán dos métodos para elegir las distancias:
    - Mahalanobis.
    - Euclídea.
- Configuración del algoritmo a utilizar. Los algoritmos a utilizar serán:
    - A*.
    - Dijkstra.

Estas funcionalidades se resumen a nivel de API en seis *end-points* para poder manejarlas. Las otras funcionalidades que tiene la biblioteca y que se omitirán en esta explicación por no aportar funcionalidades relevantes, serán las de cargar el grafo con un mapa. Actualmente para poder usar un mapa de OpenStreetMap es necesario descargarlo desde la página indicando la región de la que se quiere obtener la información, por lo que este mapa habrá que ubicarlo en el lugar de ejecución de la biblioteca. Esto hace que sea innecesario el uso de un método que transfiera los ficheros del mapa mediante esta API.

Por otro lado, en la biblioteca sí que existirán formas de cargar mapas autogenerados por el robot que podría ayudar a "recordar" al robot por donde ha pasado. De todas formas, esta funcionalidad a nivel de API no se ha contemplado en este trabajo, y será expuesta como trabajo a futuro descrito en las conclusiones.

La API se podría resumir con el diagrama de la figura 3.3.

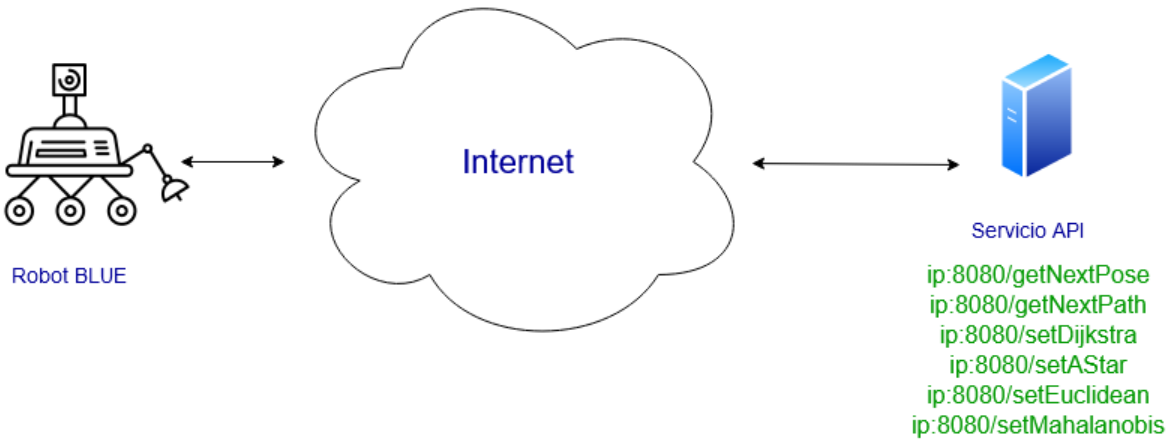

Figura 3.3.: Representación del API

Básicamente, la figura 3.3 representa como el robot BLUE hace peticiones a través de internet a la API a desarrollar. Como podemos ver es capaz de dar acceso a los seis *end-points* definidos anteriormente para ejecutar las funcionalidades de la biblioteca.

4. Cuerpo del trabajo

Con el diseño ya estudiado, se va a comenzar con el desarrollo del sistema. En este apartado se verá la forma para desarrollar toda la funcionalidad que tiene que haber detrás de la aplicación para que funcione correctamente y todo lo fundamental para la creación del sistema. Los desarrollos se pueden dividir en tres trabajos principales:

- Desarrollo de la biblioteca.
- Desarrollo de los *tests*.
- Desarrollo de la API.

Para cada uno de ellos se hará una explicación de cómo ha sido el desarrollo, explicando todos los factores principales envueltos a la hora de programarlo.

4.1. Desarrollo de la biblioteca

Esta biblioteca se ha desarrollado para planificar cuál va a ser la siguiente posición a devolver y así minimizar el camino. Para ello, la biblioteca deberá estar cargada con unas posiciones, y enlaces entre estas posiciones, permitiendo poder realizar un grafo para el análisis del camino más corto. Las clases usadas para esta biblioteca son las siguientes, que corresponden a los tipos de objetos considerados en el apartado 3.1:

- **Graph.**
- **Link.**
- **Node.**
- **Planning_Graph.**
- **Position.**
- **Util.**

Además, se hará uso de bibliotecas externas que han sido importadas al proyecto para facilitar el método de instalación:

- **RapidXml:** Incorpora métodos para la lectura y escritura de ficheros XML.
- **LatLongToUTM:** Facilita funcionalidades para cambiar el sistema de referencia de latitud y longitud a UTM y viceversa.

4.1.1. Las clases desarrolladas

En la biblioteca se desarrollaron las siguientes clases, de las cuales se explica su uso.

4.1.1.1. Position

Esta clase es usada para representar posiciones en un espacio de coordenadas. Se usan las coordenadas **x,y,z** para representar la posición en el espacio,y **yaw** para definir el ángulo de giro. Las variables principales de esta clase son:

- vector<double> coordinates_;

Vector de coordenadas con las 4 coordenadas **x**,**y**,**z**,**yaw**.

- vector<vector<double> > covarianceMatrix_;

Matriz 4x4 de covarianza de las 4 coordenadas del vector. Cada elemento de la matriz representa la variabilidad conjunta entre dos coordenadas, siendo la diagonal la variabilidad entre la misma coordenada. Esta matriz permite el cálculo de la distancia Mahalanobis explicada en el apartado 3.1.

4.1.1.2. Link

Esta clase es usada para conectar nodos entre sí. Como veremos con la siguiente clase, un nodo es la representación de un punto conocido en un grafo. Para conectar esos nodos y definir un "camino" entre ellos se hace uso de la clase **link**. La conexión entre dos nodos implica que dos posiciones son fácilmente accesibles entre sí, pudiendo llegar desde un nodo a otro de forma relativamente sencilla. El método principal de esta clase es:

- void addNodes(Node *n1, Node *n2);

Añade los nodos a conectar para este **link**. Cada instancia solo puede conectar dos nodos.

4.1.1.3. Node

Es la representación de una posición conocida en un grafo. Este objeto hereda del objeto **position** ya que busca conectar tanto la representación de una posición en el espacio tridimensional, como un punto conocido en el grafo. Cada nodo representa una intersección de caminos. Los métodos principales de esta clase son:

- double calculateEuclideanDistance(Node node);

Calcula la distancia euclídea entre dos nodos.

- double calculateMahalanobisDistance(Node node);

Calcula la distancia Mahalanobis entre dos nodos.

4.1.1.4. Planning_Graph

Esta clase será la encargada de realizar la búsqueda del camino mínimo. Contendrá todos los métodos necesarios para la búsqueda, además de todos los nodos y enlaces que conforman el grafo. Todos estos métodos para trabajar con el grafo abarcan desde los algoritmos de búsqueda hasta el cálculo entre posiciones. Los métodos y variables principales de esta clase son:

- vector<Node> nodes_;

Vector con los objetos **node** que van a pertenecer al grafo.

- vector<Link> links_;

Vector con los objetos **links** que van a pertenecer al grafo.

- Node getCloserNode(Position pos);

A partir de una posición, devuelve el nodo más cercano.

- double calculateSense(Node pos1, Node pos2, Node pos3);

A partir de 3 nodos, devuelve el ángulo de giro del nodo intermedio para poder pasar por los 3 nodos. Sería el valor **yaw** del nodo intermedio. Esto permite al robot saber cómo tiene que girar en la intersección.

- vector<Node> getCloserNodes(Position pos);

Devuelve los nodos cercanos.

    - En el caso de encontrarse lejos de cualquier nodo, devuelve los dos nodos más cercanos.
    - En el caso de haber alcanzado un nodo y estar dentro de su radio de proximidad, se devolverán todos los nodos conectados con el nodo en el que se encuentra.

- Node evaluateNextNode(Position initPos);

A partir de una posición inicial, se devuelve el siguiente nodo para alcanzar el goal final.

- vector<Node> getPathNodes(Position initPos);

Devuelve el mejor camino calculado con todos los nodos incluidos en él hasta la posición final.

- double calculateDijkstra(Node initNode, Node endNode);

Hace uso del algoritmo de Dijkstra para obtener la distancia mínima al nodo final.

- double calculateAStar(Node initNode, Node endNode);

Hace uso del algoritmo A* para obtener la distancia mínima al nodo final.

4.1.1.5. Graph

Esta clase permitirá llenar un grafo para poder calcular el camino mínimo entre dos puntos. Esta será la clase usada para llenar un grafo a partir de un archivo o contenido en formato XML y ofrecerá dos funciones principales. La primera es el constructor donde se llena el grafo a partir de un XML, y el otro será un método llamado **getNextPose** que permitirá devolver la siguiente posición del camino mínimo entre dos posiciones. Este objeto es el que hace de interfaz en el uso de la

biblioteca, además de abarcar las tareas de preprocesamiento antes de llamar a las funcionalidades principales. Los métodos principales de esta clase son:

- void xmlReadLatLong(string url, vector<vector<double> > matrix);

A partir de una url del XML indicada en el constructor y una matriz por defecto, carga el grafo. El XML debe estar en formato de latitud longitud como los que hay en OpenStreetMap.

- void xmlWrite(string fileName, int precision);

Este método crea un fichero XML a partir de la información del grafo. Esto permite guardar la información del grafo una vez finaliza el programa. La variable **precision** define con cuantos dígitos hay que guardar los valores numéricos.

- void xmlReadUTM(string url);

Este método lee un fichero XML que esté en formato UTM para llenar el grafo.

- void loadStructGraph(vector<StNodes> st_nodes);

Llena de datos el grafo a partir de un struct. Esto facilita la introducción de datos autogenerados por el robot y que ha ido obteniendo al moverse por el mapa.

4.1.1.6. Util

La clase **util** contiene métodos necesarios y usados por distintas clases. Se ha considerado que estos métodos no han tenido cabida en ninguna de las clases anteriores, ya que son usados por la mayoría de ellas y se han reunido aquí para reducir y facilitar el código. Además, esta clase contiene métodos que facilitan la inicialización de vectores de coordenadas, matrices de covarianza entre otros.

4.1.1.7. RapidXml

Esta biblioteca es usada para leer XMLs. Para la parte de la carga del grafo se usan archivos en formato OSM exportados de OpenStreetMap. El fichero OSM es un XML el cual nos aporta datos sobre posiciones y si hay conexión entre ellas, permitiendo cargar el grafo para luego poder hacer una búsqueda del camino más corto.

4.1.1.8. LatLongToUTM

Esta biblioteca tiene la funcionalidad de pasar las coordenadas en formato latitud y longitud a UTM. Debido a que OpenStreetMap ofrece los datos en latitud y longitud, es necesario la conversión de esos datos. Esta biblioteca hace uso de un archivo llamado 'constants.h' que contiene constantes necesarias para la transformación entre formatos.

4.1.2. Problemas encontrados

Durante el desarrollo han aparecido distintos problemas que han dificultado el desarrollo de la biblioteca. Debido a que el sistema tiene que funcionar "sin memoria" entre consultas del camino más corto, ha aumentado la dificultad del desarrollo. Esto se hace principalmente para evitar problemas de localización al obtener la posición mediante GNSS, ya que este sistema no es completamente preciso y tiene un margen de error. Al disponer de un sistema que calcule siempre el camino a recorrer, se irá ofreciendo siempre el siguiente objetivo a alcanzar, y se evitarán falsos positivos de nodos alcanzados o el seguimiento de una ruta desde una posición distinta a donde se encuentra realmente el robot.

Por lo que, en resumen, el sistema tiene que funcionar de forma que siempre que se pida la siguiente posición, se tenga que hacer uso del algoritmo de búsqueda y procesar toda la información. Esto, además de otras cosas, ha causado los siguientes problemas en el cálculo de la ruta y que se han ido solventando:

- Entrada al grafo

Esto determina cuál es el mejor nodo para la entrada al grafo que permite entrar en un entorno conocido. Aunque en el ejemplo la ruta más corta es entrando por el nodo C y luego ir a la estrella, la ruta hasta C puede no ser un camino. En este caso, el sistema evalúa los dos nodos más cercanos al robot y entra por el que mejor minimiza el camino. Por ejemplo, para el caso de la figura 4.1, se puede observar que los dos nodos más cercanos son A y B, y que el mejor de estos dos nodos es A porque es una entrada cercana al grafo y minimiza el camino hasta la posición final.

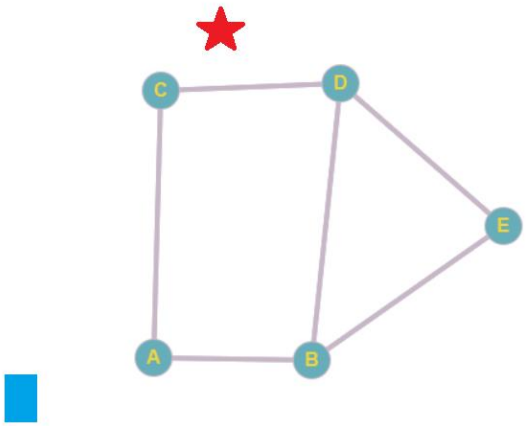

Figura 4.1.: Problema al entrar al grafo

Robot Blue

Posición final

- Desechar el grafo

Como se muestra en la figura 4.2, el robot se encuentra en una posición más cercana al destino final que los propios nodos del grafo, por lo que no es necesario entrar al grafo, siendo más conveniente ir directo al objetivo y no hacer uso de algoritmos para calcular el camino hasta la posición final.

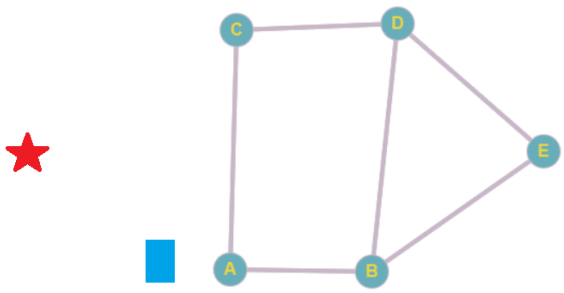

Figura 4.2.: Problema al "desechar" el grafo

- Evitar recorridos innecesarios

Aunque generalmente la prioridad es entrar en el primer nodo para ubicarse, es importante no rehacer caminos. Por ello, en el ejemplo de la figura 4.3, aunque el robot valoraría como entrada los dos nodos más cercanos (A y B), es necesario ir directamente al C ya que se encuentra posicionado en el camino.

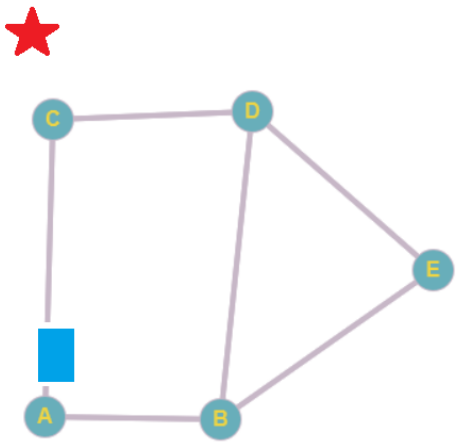

Figura 4.3.: Problema al evitar recorridos innecesarios

- No pasar por el último nodo

La mejor ruta no siempre tiene que pasar por el nodo más cercano a la posición final, sino que el robot puede ir directo a la posición final en el caso de encontrarse cerca del camino a recorrer. En el ejemplo de la figura 4.4., el robot llegaría a la posición final si tener que pasar por D.

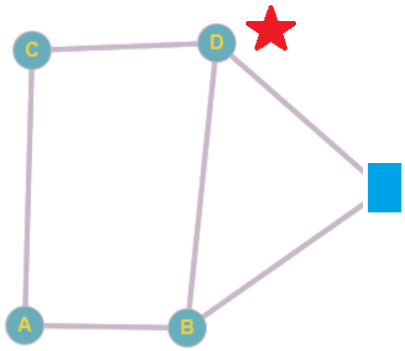

Figura 4.4.: Problema al no pasar por el último nodo

4.2. Desarrollo de las pruebas

Durante el desarrollo de la biblioteca, hay que garantizar que todo funcione correctamente. Por ello, se han estado realizando pruebas que verifiquen que todo está funcionando tal y como debería. Para verificar el funcionamiento, han habido dos puntos clave que han ayudado al correcto desarrollo, estos son los tests unitarios y las pruebas de carga.

4.2.1. Tests unitarios

Durante el desarrollo de los *tests* se han tenido bastantes variables a tener en cuenta. Aunque la intención es usar la biblioteca en entornos controlados y con información correcta, siempre puede ocurrir que se acceda elementos nulos o mal referenciados. Por ello, se han ido realizando pruebas teniendo en cuenta los siguientes factores:

- Correcta funcionalidad

Tal y como debería ser, se ha tenido en cuenta en todo momento que el sistema funcione como debería. Por ello se hacía un cálculo de cuáles son los valores que debería devolver cada uno de los métodos realizados y se hacía un conjunto de pruebas que validan que devuelven los valores correctos.

- Variables sin inicializar

Durante el uso de la biblioteca puede ocurrir que se pasen elementos nulos, o incluso hacer uso de variables sin inicializar. Estas pruebas permitieron realizar un sistema más seguro, ya que, al encontrar estos fallos, se programaron inicializadores para todas las variables.

- Elementos mal referenciados

En la biblioteca se hace uso de punteros. El tener que conectar todos los nodos con sus vecinos, mediante el uso de **links**, provoca que se genere una red de conexiones donde puedan aparecer problemas de referencias. Las pruebas que buscaban este tipo de errores permitieron encontrar fallos al referenciar, o elementos que deberían

estar referenciados y no lo estaban. Para solventar estos problemas se modificaron métodos para evitar que estos fallos pudieran ocurrir.

- Valores negativos o que no corresponden

También se tuvo en cuenta la realización de pruebas que evitaran este tipo de fallos donde puede ocurrir que se asignen valores que no corresponden al valor que debería tener. Gracias a este tipo de pruebas, se reforzaron variables para su uso mediante el tipo **enum**, y se añadieron las comprobaciones de la entrada de valores y evitar la asignación de los que no corresponden.

- Problemas en los ficheros de mapas

Entre el uso de los mapas de OpenStreetMap, y la generación propia de ficheros con los valores del grafo, se hicieron pruebas que controlarán que estos valores son correctos. Estas pruebas ayudaron a evitar fallos de escritura, además de evitar posibles fallos en la lectura de estos ficheros.

- Ejecución de los tests

Debido a la necesidad de ejecutar constantemente los *test*, se desarrolló un pequeño *script* que facilitara la ejecución de los mismos. El *script* es un pequeño programa de código que se encarga de toda la parte de compilación y ejecución del programa. Esto permite que, mediante la ejecución de un archivo '.sh', y añadiendo un valor por parámetro que indique el conjunto de pruebas a ejecutar, se automatice todo el proceso facilitando las pruebas.

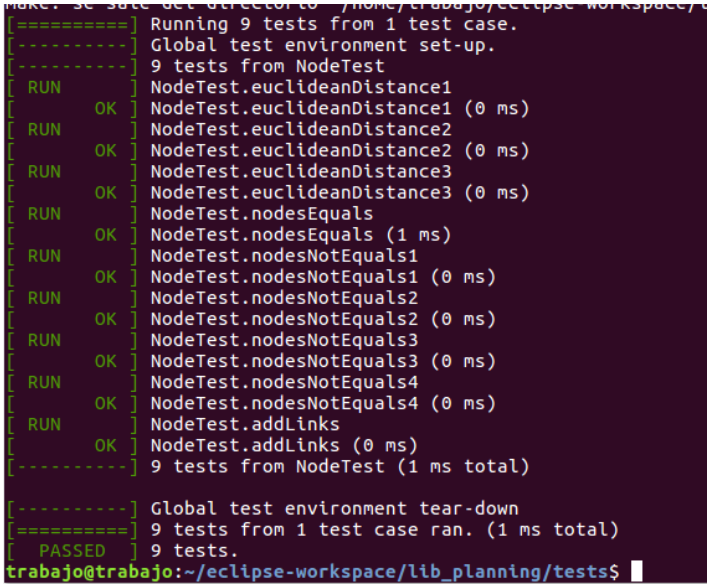

Figura 4.5.: Ejemplo de ejecución de los tests

4.2.2. Pruebas de carga

Una vez acabado el desarrollo, es necesario estudiar el correcto funcionamiento del algoritmo. Para comprobar esto, ya se ha explicado cómo se han usado pruebas unitarias que verifican que el sistema devuelve los resultados que debería. Esas pruebas unitarias no son las únicas que se han realizado para verificar el correcto funcionamiento, sino que también se ha hecho uso de pruebas de carga para ver cómo actuaría la biblioteca en un entorno real.

La navegación automática del robot va a estar constantemente verificando cuál es el camino que tiene que seguir para llegar a su destino. Todo esto lo tiene que hacer en un tiempo limitado, ya que todo el movimiento del robot se está calculando en tiempo real, y por ello es importante reducir los tiempos de cálculo del algoritmo.

Antes de comenzar con las pruebas de carga, se dedujo que en los tiempos de cálculo del algoritmo pueden influir tres aspectos importantes. Estos son:

- Cantidad de nodos.
- Cantidad de links.
- El algoritmo a usar.

Aunque esos son los principales aspectos que pueden influir en el tiempo, en la optimización también se tuvieron en cuenta otros aspectos más orientados a como estaba programada la biblioteca.

Con la identificación de los principales problemas que podrían causar un cálculo más lento, se comenzó a buscar una manera de optimizar la biblioteca. Antes de comenzar, se hicieron unas pruebas para ir viendo si los cambios iban a mejor. Las primeras pruebas de carga fueron con el algoritmo de Dijkstra y usando distancia Mahalanobis. Aunque también se hicieron pruebas con distancia Euclídea, estas no se tuvieron tan en cuenta ya que el robot BLUE, al que se desea aplicar el planificador, va a trabajar principalmente con distancia Mahalanobis. Con las pruebas se obtuvieron los tiempos de ejecución indicados en la siguiente tabla.

Tiempo de ejecución (segundos)					
		Número de nodos			
		1000	2000	3000	5000
Número de links por cada nodo	12	2,56873	5,45992	8,63852	16,26730
	10	2,58808	5,41785	8,38486	15,84720
	8	2,60058	5,30210	8,38486	15,70790
	6	2,46008	5,24208	8,31263	15,57670
	4	2,50336	5,23348	8,13742	15,24590

Tabla 4.1.: Prueba de carga inicial, con el algoritmo de Dijkstra y sin optimizar el código.

Cabe mencionar que para las pruebas se utilizó un PC modelo "HP ProDesk 490 G2 MT", con procesador Intel Core i5 3,6Hz (4 núcleos), con 8GB DDR3 de RAM, disco SSD, y tarjeta gráfica AMD radeon HD 8470. El sistema operativo ha sido Ubuntu 16.04.

Considerando la aplicación del planificador en el robot BLUE, cuya navegación en una aplicación típica de exteriores para la cual está diseñado puede requerir grafos de varios miles de nodos, y en la que una actualización de la navegación y los caminos se puede considerar en tiempo real si se realiza en tiempos inferiores al segundo, las pruebas arrojaron valores demasiado altos. Por ello se abordó la optimización de la biblioteca, en dos aspectos. Primero, la propia optimización del código para mejorar tiempos de ejecución en los puntos clave: nodos, enlaces y algoritmos. Y segundo, la mejora del algoritmo de búsqueda, implementando A* como segundo algoritmo.

Teniendo en cuenta los dos aspectos de optimización mencionados antes, la optimización del código y el añadido del algoritmo A*, se volvieron a realizar pruebas de carga, cuyos resultados están en las siguientes tablas. Así, primero se evaluó el algoritmo de Dijkstra con el código optimizado, obteniéndose una importante mejora como muestran los tiempos de la tabla 4.2, frente a los de la tabla 4.1. Después se evaluó la biblioteca con la implementación de A*, considerando también mejoras en el código, obteniéndose los tiempos mostrados en la tabla 4.3. También en este caso se obtuvo una disminución de los tiempos de ejecución, aunque menor, entre una décima y una centésima de segundo según el número de nodos. A modo de resumen, la tabla 4.3 muestra la mejora de tiempo lograda con las dos optimizaciones descritas.

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	**2000**	**4000**
Número de links por nodo	**4**	0,176132	0,394134	0,917828
	6	0,182673	0,392098	0,941233
	9	0,188840	0,403875	0,991109

Tabla 4.2.: Prueba de carga con optimización del código pero manteniendo el algoritmo de Dijkstra.

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	0,164559	0,356481	0,801405
	6	0,171616	0,353714	0,811522
	9	0,172934	0,361624	0,852614

Tabla 4.3.: Prueba de carga usando el algoritmo de búsqueda A, además de la optimización del código.*

Mejora de A* respecto a Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	107,03%	110,56%	114,53%
	6	106,44%	110,85%	115,98%
	9	109,20%	111,68%	116,24%

Tabla 4.4.: Mejora de A respecto a Dijkstra después de optimizar el código en ambos algoritmos.*

El trabajo no se concluyó con las dos optimizaciones descritas antes, sino que además se realizó una posterior optimización general del código. Tras esa optimización general del código, que se centró, a grandes rasgos, en una refactorización para evitar bucles innecesarios o repeticiones del uso de los algoritmos cuando no era necesario, se obtuvieron los resultados de tiempos de ejecución reflejados en las tablas 4.4 y 4.5. Estos tiempos sí que se consideran de tiempo real para la aplicación sobre el robot BLUE.

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	0,060025	0,163580	0,546865
	6	0,080008	0,213442	0,735899
	9	0,082154	0,217877	0,747052

Tabla 4.5.: Prueba de carga usando Dijkstra, con el código optimizado y refactorizado.

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	0,063420	0,129967	0,233220
	6	0,089122	0,165412	0,320598
	9	0,089907	0,185059	0,351731

Tabla 4.6.: Prueba de carga usando A, con el código optimizado y refactorizado.*

Mejora de A* respecto a Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	94,65%	125,86%	234,48%
	6	89,77%	129,04%	229,54%
	9	91,38%	117,73%	212,39%

Tabla 4.7.: Mejora de A respecto a Dijkstra después de refactorizar código en ambos.*

Con estas últimas pruebas, podemos ver que la mejora al refactorizar el código son las siguientes:

Mejora de Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	293,43%	240,94%	167,83%
	6	228,32%	183,70%	127,90%
	9	229,86%	185,37%	132,67%

Tabla 4.8.: Porcentaje de mejora de Dijkstra después de refactorizar código

Mejora de A*				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	259,47%	274,29%	343,63%
	6	192,56%	213,84%	253,13%
	9	192,35%	195,41%	242,41%

Tabla 4.9.: Porcentaje de mejora de A después de refactorizar código*

Además, tras esta última mejora, se hace notable como en la *tabla 4.5* generalmente A* funciona mejor que Dijkstra excepto en casos concretos. Estos casos son los que el grado no contiene gran cantidad de nodos. Esto ocurre porque comprobar todos los caminos como hace Dijkstra no supone aún un problema respecto a los cálculos extra que ocurre en el caso de A* para calcular la heurística. Estos cálculos extra de la heurística en distancia Mahalanobis suponen un coste computacional bastante alto, y con pocos nodos, no llega a notarse la mejora que ofrece A* al evitar una búsqueda completa del grafo como hace el algoritmo de Dijkstra.

Una vez refactorizado totalmente el algoritmo, también se hicieron pruebas usando distancia euclídea, cuyos resultados se muestran en las siguientes tablas.

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	2000	5000
Número de links por nodo	4	0,053958	0,149054	0,541624
	6	0,069561	0,200173	0,710075
	9	0,075295	0,207299	0,713196

Tabla 4.10.: Prueba de carga Dijkstra con distancia euclídea

Tiempo de ejecución (segundos)				
		Número nodos		
		1000	2000	5000
Número de links por nodo	4	0,049830	0,098099	0,186353
	6	0,066206	0,128935	0,247990
	9	0,069833	0,143254	0,275191

Tabla 4.11.: Prueba de carga A con distancia euclídea*

Mejora de A* respecto a Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	108,28%	151,94%	290,64%
	6	105,07%	155,25%	286,33%
	9	107,82%	144,71%	259,16%

Tabla 4.12: Mejora de A respecto a Dijkstra con distancia euclídea*

En la tabla 4.10 se puede comprobar cómo al no haber ahora ese coste computacional tan alto al calcular la distancia Mahalanobis en la heurística del algoritmo A*, este si que muestra una clara mejora aún con pocos nodos en el grafo. Con estas pruebas, también se puede calcular la mejora de usar la distancia euclídea respecto a la distancia Mahalanobis. Como se muestra en las tablas 4.11 y 4.12, la distancia euclídea tiene una mejora ya que el coste computacional de calcular la distancia es

menor, y este efecto se notará más en el algoritmo A* que hace más uso de este tipo de cálculos.

Mejora de A* respecto a Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	127,27%	132,49%	125,15%
	6	134,61%	128,29%	129,28%
	9	128,75%	129,18%	127,81%

Tabla 4.13.: Mejora de A cuando se usa distancia euclídea*

Mejora de A* respecto a Dijkstra				
		Número nodos		
		1000	2000	4000
Número de links por nodo	4	111,24%	109,75%	100,97%
	6	115,02%	106,63%	103,64%
	9	109,11%	105,10%	104,75%

Tabla 4.14.: Mejora de Dijkstra cuando se usa distancia euclídea

Aún con la notable mejora del tiempo que tiene el uso de las distancias euclídeas, en ciertos aspectos es mejor usar la distancia Mahalanobis. Esto es debido a que en entornos donde se está usando localizaciones obtenidas de un GNSS como es este caso, puede ocurrir que no se pueda determinar la ubicación de forma precisa. Debido a que pueda haber cierto margen de error a la hora de determinar la ubicación, es mejor usar la distancia Mahalanobis que permite determinar mediante el uso de la estadística la distancia a las distintas posiciones.

4.3. Desarrollo de la API

Con el desarrollo de la API se busca ubicar de forma remota todo el desarrollo explicado hasta ahora en el documento. La posibilidad de tener el sistema ubicado de forma remota facilita aún más su uso.

Como se ha descrito en apartados anteriores, al principio se hizo un primer desarrollo con NodeJS. En este desarrollo, se llegó a poner en funcionamiento una API capaz de hacer peticiones por GET y por POST que son las que se van a necesitar para esta biblioteca. Además de esto, se combinó con una interfaz en C++ capaz de comunicarse con la biblioteca.

La interfaz desarrollada en C++ permite hacer uso de la biblioteca, al mismo tiempo que se preparan los elementos para la comunicación entre la biblioteca y el código en JavaScript. Aunque la generación del servicio web es sencillo mediante este entorno, la comunicación posterior con una biblioteca desarrollada en C++ provocaba que se complicara en gran medida una tarea que debería ser relativamente sencilla. Los tipos sencillos como valores numéricos o cadenas de texto eran fáciles de pasar entre JavaScript y C++, pero los elementos más complejos como una posición que viene identificada por las coordenadas y la matriz de covarianza, era más complicado de pasar entre lenguajes, por lo que se optó por buscar otra solución.

Para solventar este problema, se buscó una alternativa en la que se pudiera poner en funcionamiento un servicio web desde el lenguaje C++. Aunque el uso de este lenguaje no se especializa en la realización de un servicio web, sí que hay documentación en internet que describe cómo realizarlo. Examinando esta información se encontró la biblioteca Glove [11], que ya resolvía este problema, por lo que se decidió usar la misma en vez de crear una biblioteca nueva.

Glove es una biblioteca en C++ que ofrece las funcionalidades necesarias para la creación de una *webAPI* desde la cual se pueden tramitar solicitudes de forma remota. Para hacerla funcionar se desarrolló una interfaz capaz de comunicar la biblioteca de *planning* con las funcionalidades de Glove para la creación del servicio web. Esta interfaz principalmente selecciona cuales van a ser los *end-points* que se van a ofrecer, y servir para "traducir" la información que llegará a la *webAPI* o será devuelta por ella. Los *end-points* definidos son los siguientes:

- **ip:8080/getNextPose:** Devuelve la siguiente posición del camino

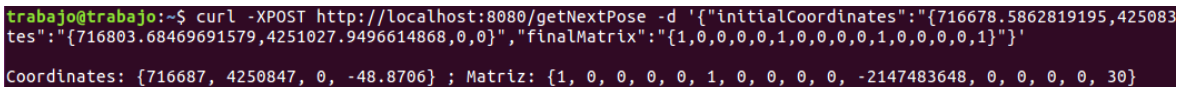

Figura 4.6.: Ejemplo de llamada a NextPose

Hay que tener en cuenta que, en los resultados que se pueden ver en los ejemplos, para facilitar su entendimiento, se muestra una respuesta mucho más detallada de lo que el *end-point* devuelve a un cliente en la ejecución final. En la implementación final el *end-point* devuelve un resultado más simplificado a los clientes.

- **ip:8080/getNextPath:** Devuelve el conjunto de posiciones del camino

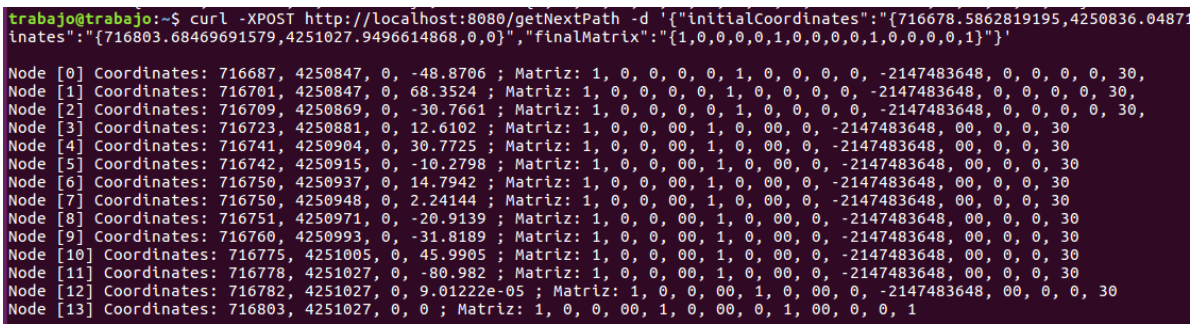

Figura 4.7.: Ejemplo de llamada a NextPath

- **ip:8080/setDijkstra:** Establece Dijkstra como algoritmo a usar

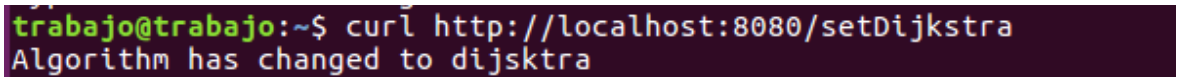

Figura 4.8.: Ejemplo de llamada a setDijkstra

- **ip:8080/setAStar:** Establece A* como algoritmo a usar

```
trabajo@trabajo:~$ curl http://localhost:8080/setAStar
Algorithm has changed to A*
```

Figura 4.9.: Ejemplo de llamada a setAStar

- **ip:8080/setEuclidean:** Establece la distancia euclídea como distancia a usar

```
trabajo@trabajo:~$ curl http://localhost:8080/setEuclidean
Type idstance has changed to euclidean
```

Figura 4.10.: Ejemplo de llamada a setEuclidean

- **ip:8080/setMahalanobis:** Establece la distancia Mahalanobis como distancia a usar

```
trabajo@trabajo:~$ curl http://localhost:8080/setMahalanobis
Type idstance has changed to mahalanobis
```

Figura 4.11.: Ejemplo de llamada a setMahalanobis

5. Conclusiones

El proyecto BLUE pretende acumular el desarrollo de varios años de investigación de distintos campos de la robótica, en los que varias personas han ido trabajando para poder ir formando un proyecto aún mayor. Esto obliga a que sea necesario un buen sistema organizativo del código, tal y como se ha descrito en el apartado 2.3, donde se indica que es necesario trabajar en repositorios para la correcta coordinación de todos los participantes en el desarrollo. Además de la coordinación, es importante una correcta integración de todos los sistemas desarrollados de forma independiente.

Aunque en un principio la biblioteca no permitía combinarse con los otros desarrollos debido a que tardaba mucho en ejecutarse para funcionar en tiempo real, se hizo una mejora considerable en los tiempos de cálculo de los algoritmos. Por este motivo, se ha hecho énfasis en la parte de optimización y pruebas del código y de los algoritmos, ya que han permitido generar un sistema más robusto y que se combine mejor con los otros desarrollos para permitir el funcionamiento del robot en tiempo real.

Una vez finalizada y probada la biblioteca, se pasa al siguiente nivel de integración. Esto consiste en combinar todas las partes que componen y permiten el funcionamiento del robot móvil para poder navegar en entornos no estructurados. En la parte de integración, se tiene que realizar otro conjunto de pruebas que verifican que el conjunto funciona correctamente. Estas pruebas de integración se realizan mediante simulaciones de un entorno al que se tiene que enfrentar cuando se pruebe en real. Una vez verificadas estas pruebas, ya se integra todo en el robot móvil y se hacen pruebas reales.

Por otro lado, la API no pasa por ese conjunto de pruebas ni se incluye en el proyecto de la investigación. Aunque la API fue desarrollada para que esta biblioteca sea usada de forma remota por cualquier robot, de momento, en el proyecto BLUE se

usa de forma local instalándola en el sistema. En cualquier caso, el desarrollo de la API remota supone una novedad en el software de planificación de caminos para robots, ya que con la aparición de la tecnología 4G y 5G, se facilita la adaptación de este tipo de desarrollos. La API permite gran cantidad de ventajas en la navegación de robots, ya que extrae el procesamiento del cálculo de caminos, para facilitar la integración de otros módulos como la detección del entorno o localización del robot. Además de estas ventajas, el ahorro de procesamiento supone consumir menos batería, lo que implica un mayor aguante de la batería del robot hasta la siguiente recarga.

Para el desarrollo de todo este proyecto del documento ha sido necesario aplicar conocimientos que hasta ahora yo nunca antes había puesto a prueba, y que he aprendido durante el máster, e incluso durante el grado de informática. Además, aunque los conocimientos adquiridos en el máster han sido una buena base para comenzar a abordar el proyecto, ha sido necesario aprender e investigar nuevos temas de robótica móvil y navegación, que hasta el comienzo del desarrollo eran totalmente desconocidos para mí. Todo este aprendizaje me ha permitido aplicarlo a un proyecto real, y que actualmente está siendo probado en el robot BLUE. A falta de las últimas comprobaciones con el equipamiento real, donde se está verificando diferentes algoritmos de percepción, mapeo y navegación en exteriores, actualmente se está redactando un manuscrito para enviar a una revista relevante, en el cual se incluye parte del desarrollo presentado en este documento, y en cuya autoría participo junto con los investigadores que trabajan con el robot BLUE.

5.1. Trabajo a futuro

Existen diversos desarrollos que se podrían hacer tanto a la biblioteca, como a la API remota, continuando con el trabajo desarrollado en el TFM. Estas nuevas implementaciones son:

Nuevos desarrollos de la biblioteca:

- Facilitar la interfaz de uso.
- Mejorar tiempos de carga del mapa.
- Facilitar la instalación de la biblioteca.
- Desarrollar métodos para combinar distintos grafos y así generar mapas más grandes que combinan varios grafos.
- Desarrollar métodos que permitan detectar puntos relevantes a incluir en el grafo.
    - Nuevas intersecciones. Esto se podría hacer calculando los ángulos de giro.
    - Puntos intermedios. Se podrían almacenar las posiciones intermedias entre nodos para tener aún más información sobre los caminos a recorrer.

Nuevos desarrollos de la API:

- Probar la API en un robot real, con conexión móvil a internet, y estudiar la mejora respecto a la descarga de carga de proceso en el robot móvil.
- Facilitar nuevas formas de cargar los datos.
- Añadir todos los métodos disponibles de la biblioteca.
- Permitir actualizar el grafo con la nueva información que vaya adquiriendo el robot en su recorrido.
- Aumentar la seguridad de la comunicación.
- Permitir varios clientes simultáneos.
- Buscar maneras de usar métodos más estandarizados para su uso en cualquier robot.

6. Bibliografía y Referencias

[1] OpenStreetMap, Fundación OpenStreetMap (OSMF). [En línea, consultado en diciembre de 2020]:
https://www.openstreetmap.org/#map=16/38.3836/-0.5159

[2] Iván del Pino, Miguel Á. Muñoz-Bañon, Saúl Cova-Rocamora, Miguel Á. Contreras, Francisco A. Candelas, Fernando Torres. "Deeper in BLUE: Development of a roBot for Localization in Unstructured Environments". Journal of Intelligent and Robotic Systems (Springer), Vol. 98, pp. 207–225, 2019. ISSN: 0921-0296.

[3] Miguel Á. Muñoz–Bañón, Iván del Pino, Francisco A. Candelas, Fernando Torres. "Framework for Fast Experimental Testing of Autonomous Navigation Algorithms". Applied Sciences (MDPI), Special Issue Mobile Robots Navigation, 2019, Vol. 9 (10), pp. 109-131. EISSN 2076-3417.

[4] ROS: Robot Operating System. Open Source Robotics Foundation (OSRF) & Open Robotics, 2021. [En línea, consultada en diciembre de 2020]:
https://www.ros.org/about-ros/

[5] Dijkstra's Algorithm for Adjacency List Representation, GeeksforGeeks, 2020 [En línea, consultada en diciembre de 2020]:
https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/

[6] Búsquedas Informadas, Fernando Sancho Caparrini, 2019 [En línea, consultada en diciembre de 2020]:
http://www.cs.us.es/~fsancho/?e=62

[7] Distancia euclidiana: concepto, fórmula, cálculo, ejemplo. Lifeder, 2021. [En línea, consultada en enero 2021]:
https://www.lifeder.com/distancia-euclidiana/

[8] Distancia Mahalanobis, Juan Martínez de Lejarza y otros. Proyecto CEACES, Universidad de Valencia, 2010 [En línea, consultada en enero de 2021]:
https://www.uv.es/ceaces/multivari/cluster/d_mahalanobis.htm

[9] NodeJS, OpenJS Foundation y Joyent Inc. [En línea, consultada en diciembre de 2020]:
https://nodejs.org/es/

[10] Glove: C++11 sockets TCP wrapper, Gaspar Fernández [En línea, consultada en diciembre 2020]:
https://github.com/gasparfm/glove

[11] GoogleTest: Google's C++ test framework, Abseil Team and CJ-Johnson, 2021. [En línea, consultada en enero 2021]:
https://github.com/google/googletest

[12] Planificación de trayectorias para un robot móvil de exteriores, Alberto Lopez Sellers, 2021. [En línea]:

- Biblioteca de *planning*:

https://github.com/AUROVA-LAB/lib_planning

- API:

https://github.com/Alvertoo/api_planning