Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Implementación y análisis de algoritmos de planificación de caminos. Aplicación a sistemas aéreos no tripulados.

Autor: Nikita Tretiyakov
Tutor: Anibal Ollero Baturone

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Ingeniería Aeroespacial

Implementación y análisis de algoritmos de planificación de caminos. Aplicación a sistemas aéreos no tripulados.

Autor:
Nikita Tretiyakov

Tutor:
Anibal Ollero Baturone
Catedrático

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Implementación y análisis de algoritmos de planificación de caminos. Aplicación a sistemas aéreos no tripulados.

Autor: Nikita Tretiyakov

Tutor: Anibal Ollero Baturone

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

Agradecimientos

La realización de trabajo ha sido posible gracias a Anibal Ollero Baturone, mi tutor y profesor, que me ha transmitido los conocimientos necesarios y a Álvaro Caballero Gómez que me ha ayudado y me ha guiado durante la realización de este proyecto.

Resumen

El objetivo de este trabajo es el análisis y la comparación de los algoritmos de *búsqueda de camino* más usados hoy en día para los Vehículos Aéreos No Tripulados (UAVs). La *búsqueda de camino* es un problema fundamental dentro del campo de la robótica y su resolución permite que los vehículos sean totalmente autónomos y puedan ir desde un punto inicial o otro final sin ayuda del ser humano. Dentro de las metodologías existentes, se va a centrar en los algoritmos de búsqueda basados en descomposición en celdas y planificadores probabilísticos. Para el estudio de estas metodologías,se han seleccionado 5 algoritmos: A*, θ^*, RRT, RRT* y RRT-Connect, y se han programado en MATLAB siguiendo las pautas de su funcionamiento teórico. Una vez programados, se han estudiado sus posibles modificaciones y mejoras y, se ha hecho un análisis comparativo entre ellos

Índice general

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

1.1. Estado del arte

Los Vehículos Aéreos No Tripulados, UAVs (Unmanned Aereal Vehicles), o también denominados Drones, han existido desde la Primera guerra mundial (1917). Durante la mayor parte de su existencia, no gozaban de mucha popularidad debido a su poca fiabilidad y autonomía limitada en esa época. A partir de los años 90, se ha visto un aumento en producción de UAVs militares que cumplían misiones de vigilancia de alto riesgo. Sin embargo, en la última década, gracias a los últimos avances tecnológicos en los campos de robótica y electrónica, se ha producido un auge de su popularidad y sus aplicaciones se han multiplicado rápidamente en campos muy diversos, tanto militares como civiles [17].

(a) General Atomics MQ-9 Reaper. UAV esta-
dounidense militar diseñado para misiones de vi-
gilancia a grandes altitudes.

(b) UAV de transporte desarrollado por Con-
nect Robotics.

Figura 1.1: Ejemplos de UAVs.

Figura 1.2: Kettering Bug desarrollado durante la Primera Guerra Mundial.

Figura 1.3: UAV militar AAI RQ-2 Pioneer. Estuvo en servicio en los años 1986-2007.

Hoy en día existen numerosas aplicaciones para estas máquinas, que vienen en diferentes formas y tamaños. Cabe destacar que el empleo civil se desarrolla muy rápido ya que, las posibilidades son innumerables. Se han creado muchas investigaciones, la mayoría de las cuales se plantearon hace tiempo pero estaban en desuso.

A continuación se van a enumerar algunas de las aplicaciones más importantes (ver Figura 1.4 como ejemplo) que se usan por todo el mundo:

- **Búsqueda y rescate**: los drones pueden usarse para tener contacto visual en todo momento con las personas que se encuentran en una situación peligrosa e, incluso, traer objetos necesarios para mantener a alguien con vida con una velocidad muy elevada. Por ejemplo: ayuda al rescate de personas en el mar o que están dentro de un edificio incendiado.
- **Seguridad y vigilancia**: los UAVs resultan ser ideales para supervisión de zonas de difícil acceso y seguimiento de objetivos móviles, como, por ejemplo los drones de vigilancia del parque natural Doñana.
- **Inspecciones**: la mayoría de las infraestructuras de difícil acceso pueden ser inspeccionadas desde aire y a una distancia mínima gracias a los drones. El mantenimiento de las turbinas eólicas y las líneas eléctricas es un buen ejemplo de la mejora de la seguridad gracias a los drones.
- **Ciencia e investigación**: el uso de los drones ha abierto numerosas posibilidades de observar la naturaleza sin perturbarla o alcanzar sitios de difícil acceso.
- **Cartografía**: los UAVs también se utilizan para crear mapas de alta resolución de paisajes o edificios. Sobre todo destaca su uso para crear mapas 3D.
- **Grabación de vídeos**: vista aérea crea nuevas perspectivas en cinematografía y grabaciones a distancia.
- **Transporte**: como cualquier otro vehículo, los drones pueden transportar objetos y, se cree que en un futuro, serán lo suficientemente seguros, como para transportar personas.

En la actualidad, debido a su gran popularidad, los drones están siendo el objetivo de numerosas investigaciones de desarrollo y siguen apareciendo innumerables aplicaciones al ver que estos pueden ofrecer un servicio muy versátil, viable y, sobre todo, por un precio razonable. El objetivo principal de esta tecnología es automatizar los procesos, minimizar el trabajo humano y, sobre todo, aumentar la seguridad, ya que, la mayoría de los fallos o de accidentes se deben al error humano. Al automatizar los procesos también se mejora la rapidez y la precisión de los aparatos y se reducen los costes operativos.

(a) Dron de rescate RTS.

(b) DJI MAVIC PRO. Realiza fotografías y grabaciones profesionales.

(c) UAV realizando operaciones de inspección y mantenimiento en las turbinas eólicas.

Figura 1.4: Algunos ejemplos de usos variados de UAVs.

1.1.1. Tipos de algoritmos existentes

A fecha de hoy, existen numerosos algoritmos, que resuelven el problema de encontrar un camino hacia el objetivo, teniendo en cuenta diferentes enfoques y la tecnología disponible. A continuación, se van a describir algunos de los algoritmos mas usados:

1.1.1.1. Bug algorithm

Estos algoritmos fueron unos de los primeros en implementarse y son de los más sencillos que existen (ver Figura 1.5). Bug algorithm [15] está creado para los los entornos 2D en los que el vehículo tiene un sensor de contacto o de un rango finito. El comportamiento básico de estos algoritmos se puede resumirse en 2 actividades:

- Movimiento hacia el objetivo.
- Seguimiento de las fronteras de los obstáculos.

Como ya se ha dicho antes, destaca por su sencillez pero, al mismo tiempo, esto conlleva a que las soluciones no sean las óptimas.

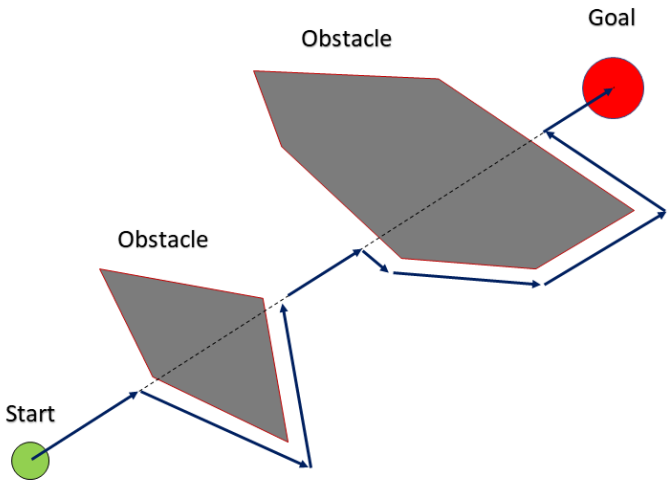


Figura 1.5: Ejemplo de camino seguido por Bug.

1.1.1.2. Campos potenciales

Los algoritmos basados en los campos potenciales [10], utilizan las funciones diferenciables, reales, cuyo valor es la energía potencial artificial, y el gradiente de dicha función es la fuerza que afectaría al vehículo.

La función potencial se define normalmente como la suma de un potencial de atracción hacia el objetivo y un potencial de repulsión de los obstáculos, como se indica en la Figura 1.6.

Estos algoritmos se usan en aplicaciones donde se requiere bajo coste computacional. Sin embargo, su mayor desventaja consiste en la existencia de los mínimos locales donde el programa puede quedarse atrapado.

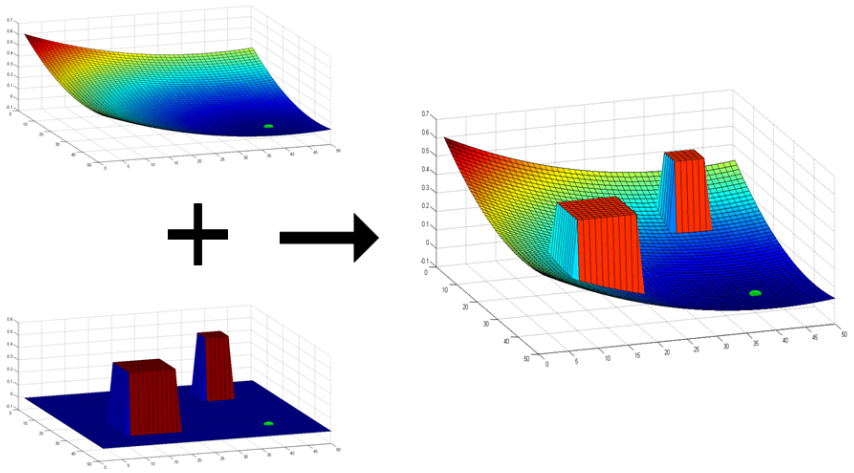


Figura 1.6: Ejemplo de campo potencial con obstáculos.

1.1.1.3. Algoritmos basados en grafos/ Roadmaps

Los algoritmos basados en grafos, o también llamados Roadmaps, construyen una red de rutas conectados entre sí.

Antes de empezar describiendo los algoritmos, se va a explicar brevemente lo que es un grafo: un grafo es un conjunto de nodos unidos por los enlaces llamados arcos, que posibilitan representar relaciones entre los dichos nodos. Un ejemplo claro de esto está representado en la Figura 1.7

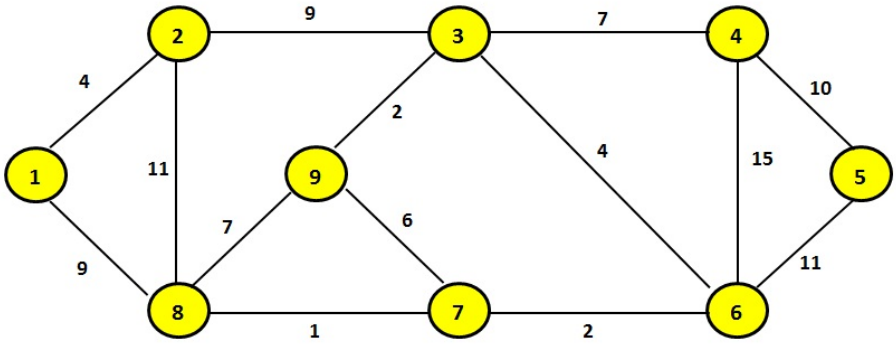


Figura 1.7: Ejemplo de un grafo básico con costes indicados.

Al ser bastante reciente el estudio de búsqueda de caminos de forma autónoma, no existe una clara diferenciación entre distintos tipos de algoritmos, por lo que en este trabajo se propone a diferenciar dos grupos: grafos «fijos» o «constantes», y grafos probabilísticos.

Se va a denominar como grafos fijos los que obtienen toda la información necesaria para su construcción exclusivamente del mapa dado (En este trabajo solo se van a contemplar los espacios estáticos). Es decir, son independientes del camino que elija el algoritmo y se mantienen igual. Algunas de las técnicas de obtención más conocidas de este tipo de grafos son:

- **Grafos de visibilidad**: se construyen conectando los vértices de los obstáculos presentes en el mapa. Eso genera rutas directas y cortas, pero suelen tener un coste computacional muy elevado [18]. Ver Figura 1.8.
- **Diagramas de Voronoi**: es una alternativa que pretende que el camino este lo más alejado posible de los obstáculos. Se consigue dividiendo el espacio con lineas equidistantes de los obstáculos. Este método es muy efectivo si los obstáculos son puntuales. Sin embargo, no existen modificaciones elementales para detectar obstáculos complejos, por lo que su eficiencia baja notablemente si estos son polígonos complicados e irregulares [1]. Ver Figura 1.9.
- **Descomposición en celdas**: se basa en un simple principio: dividir un espacio complejo en fragmentos sencillos y pequeños. Habitualmente, la zona de trabajo se divide en regiones cuadradas, trapezoidales o triangulares, de tal forma que estos fragmentos se puedan definir con un sistema binario, dependiendo de si son accesibles o no. Esta descomposición permite tratar el mapa como una matriz, que es muy apropiado para el uso computacional. Figura 1.10.

Una vez construido el grafo, se utilizan los algoritmos de búsqueda de ruta para encontrar la solución deseada. Los algoritmos más usados para los espa-

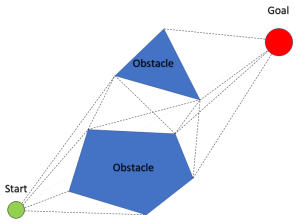


Figura 1.8: Grafos de visibilidad.

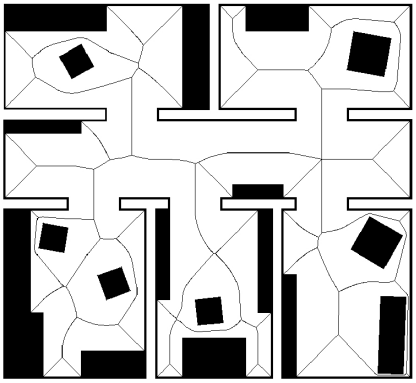


Figura 1.9: Diagramas de Voronoi.

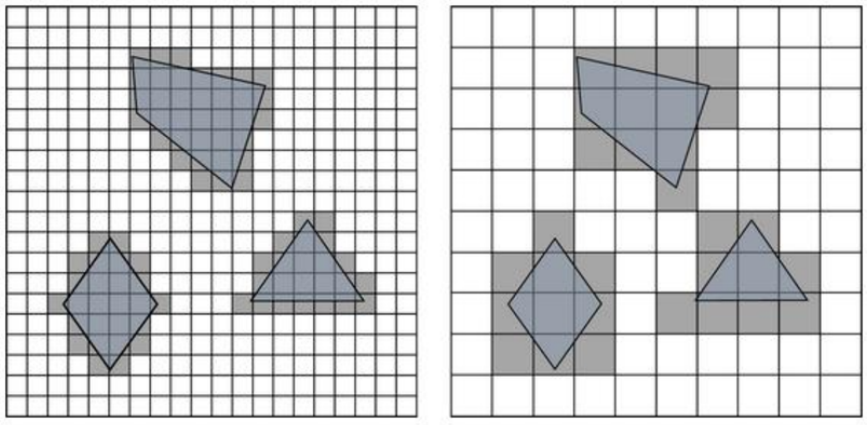


Figura 1.10: Descomposición en celdas.

cios estáticos son Dijkstra [5], A* [7] y sus modificaciones como θ^* [4], y otros como ANYA [6]. Algunos de ellos, en concreto, A* y θ^* se van a explicar más detenidamente en el siguiente capítulo.

Los planeadores probabilísticos, también llamados SBP (Sampling-Based Planning) [14], a diferencia de los anteriores, se basan en generación aleatoria de los estados para construir el camino. Es decir, su función es la de aproximar la conectividad del espacio mediante una estructura de grafos, muestreando las zonas de este espacio.

La aleatoriedad de este método permite resolver problemas muy complejos en relativamente poco tiempo, ya que el tiempo empleado en encontrar la solución no tiene relación directa con la complejidad del mapa. La desventaja del enfoque probabilístico es que las soluciones obtenidas no son óptimas en los casos reales. Además, el muestreo aleatorio no garantiza encontrar la solución, si es que existe. Aunque, teóricamente, si el programa se deja funcionando durante un tiempo infinito, sí obtiene una solución y la optimiza. Por lo tanto, este tipo de planificación no es completamente exhaustivo, donde por exhaustivo se entiende que tiene que resolver el problema y hacerlo en un tiempo finito. Por esta razón, para este tipo de planificadores, se admite una noción de exhaustividad más débil, denominada exhaustividad probabilística, lo que significa que la probabilidad de encontrar una solución converge a uno a medida que pasa más tiempo. La velocidad de esta convergencia es lo que indica la eficacia del planificador.

Los planeadores probabilísticos comparten entre sí los siguientes elementos:

- **Muestreo**: el procedimiento que selecciona un punto nuevo al azar y lo agrega al grafo o arbol ya existente. El punto puede estar en el espacio inaccesible, lo que permite explorar más opciones que otros métodos. Dependiendo del tipo del planificador este punto puede ser válido o no. El generador de estados tiene que tener una cierta uniformidad para poder explorar todo el espacio, aunque se le puede introducir una cierta parcialidad.

- **Métrica**: define la distancia o el esfuerzo que hay entre dos estados q_a y q_b. Es importante tenerlo en cuenta para que la solución sea la de menor coste posible.

- **Vecino más cercano**: busca y selecciona el nodo más cercano al punto propuesto por el muestreo.

- **Selección de predecesor**: esta operación selecciona uno o varios nodos existentes para conectarse con el el nodo recién generado. El algoritmo RRT, por ejemplo, selecciona el nodo más cercano, mientras que PRM conecta varios vecinos a la vez.

- **Planificación local**: dados dos nodos q_a y q_b, esta operación intenta conectarlos de forma adecuada. Aquí pueden intervenir las restricciones cinemáticas o dinámicas del modelo, cambiando el resultado.

- **Comprobación de colisiones**: es una operación que determina si es posible la conexión entre dos nodos. Suele ser una función booleana con una respuesta simple de éxito o fracaso, aunque su coste computacional llega a ser considerablemente mayor de las operaciones mencionadas anteriormente, si hay una cantidad elevada de obstáculos o, si la geometría de estos (o del vehículo) es muy compleja.

A continuación se enumeran los algoritmos principales de los planeadores probabilísticos:

- **Probabilistic Roadmap Method (PRM)** [9], es un planificador que construye un mapa de carretera generando nodos de forma aleatoria y uniéndolos entre si. Una vez construida la carretera empieza la búsqueda del camino más óptimo usando los planificadores ya conocidos como A* o Dijkstra.
- **Rapidly-Exploring Random Tree (RRT)** [13], es un planificador que toma un enfoque diferente a la hora de construir el grafo. Como el nombre lo indica, construye un árbol (Tree), que se puede definir como un grafo sin ciclos. Su principal propiedad es que cualesquiera dos vértices del grafo solo están conectados por un único camino. [2] Este planificador es muy versátil ya que su algoritmo básico permite fácil implementación de modificaciones.
- **Expansive Space Trees (EST)** [9], este planificador con estructura de árbol busca las zonas menos exploradas para expandir las ramas. El algoritmo construye 2 árboles desde el inicio y la meta, hasta que los dos puedan ser conectados. Esto permite aumentar el rendimiento en espacios extremadamente complejos.

Cabe mencionar que hay otros métodos notables, similares a estos que introducen nociones cinemáticas y dinámicas para obtener resultados más realistas, por ejemplo: Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE) [16] y Path-Directed Subdivision Trees (PDST) [12].

1.2. Objetivos y planteamiento

En este trabajo, se va a centrar en la autonomía de UAVs, es decir, que sean capaces de tomar decisiones sin ayuda de las personas. En concreto, se estudiarán los algoritmos más usados de búsqueda de camino, que es un problema fundamental y de los más estudiados hoy en día. Los problemas de planificación de camino no solo se aplican a los drones, es un área bastante extenso, que también se estudia en la robótica, automóviles de ultima generación e, incluso, en los videojuegos.

Específicamente, se va a centrar en los espacios 2D, ya que un número importante de las operaciones básicas que realizan los UAVs suelen estar contenidas en un plano. Se estudiará el uso óptimo de cada algoritmo basándose en los diferentes escenarios y condiciones de trabajo dadas. En las Figuras 1.11, 1.12 y 1.13 pueden verse algunos ejemplos.

Figura 1.11: Roomba iRobot realizando limpieza de un piso, siguiendo una ruta.

Figura 1.12: Automóvil Tesla: coche eléctrico capaz de conducir de forma totalmente autónoma.

Figura 1.13: Ejemplo de planificador de camino A* en el videojuego Banished.

1.3. Estructura del documento

En esta introducción se ha pretendido explicar brevemente la importancia de los algoritmos de planificación del movimiento y estado del arte actual. Dicho esto, se comentarán las propiedades de los escenarios elegidos, asimismo, se analizarán sus aplicaciones en la realidad. El capítulo 2 se centra en la descripción de los entornos elegidos para la investigación. En el capítulo 3 se van a explicar con más detalle los planificadores estudiados, para ver las características básicas de cada uno de ellos. En el capítulo 4 se comentarán los resultados y las gráficas obtenidos. Finalmente, el capítulo 5 expone las conclusiones del trabajo y las posibles lineas futuras.

Capítulo 2

Escenarios de aplicación

Para poder observar el comportamiento de diferentes algoritmos se han creado varios escenarios que permiten explorar diferentes facetas de los dichos algoritmos.

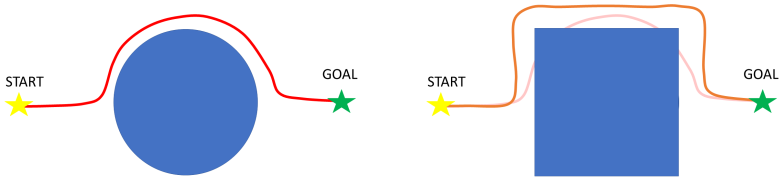
Estos escenarios han de cumplir varias condiciones para que sean apropiados en el contexto dado.

Lo primero que hay que mencionar, es que el entorno de trabajo solo tiene 2 dimensiones, lo que significa que todos los espacios del trabajo estudiados aquí van a estar contenidos en un plano. Esto se debe a varias razones:

- En este trabajo, se va a hacer una prueba de concepto para estudiar los algoritmos y, por ello, para simplificar el problema, se ha decidido limitarlo a solo dos dimensiones.
- La gran parte de aplicaciones que existen hoy en día pueden cumplir su tarea de forma autónoma usando un plano de trabajo o varios por separado.
- El coste computacional de los algoritmos al pasar de ser bidimensionales a tridimensionales crece considerablemente.

Otra de estas condiciones es que el mapa cumpla con las expectativas de la realidad, es decir, que pueda ser usado para modelar en un entorno real. En principio, cualquier escenario con obstáculos sería válido ya que en las innumerables aplicaciones de los UAVs y la robótica se puede encontrar una enorme diversidad de variables. En este caso se va a centrar en dos tipos de entorno: espacio cerrado y espacio abierto.

Por espacio cerrado se entiende que tiene unas dimensiones reducidas, como una habitación o un edificio, donde es importante mantener la precisión de los movimientos. Este tipo de entornos suele tener pocos obstáculos de notable extensión, por lo que, habitualmente, la ruta es bastante enrevesada. Por tanto habría que considerar las dimensiones del UAV, que no serían despreciables en este caso.



(a) Obstáculo circular. (b) Obstáculo simple.

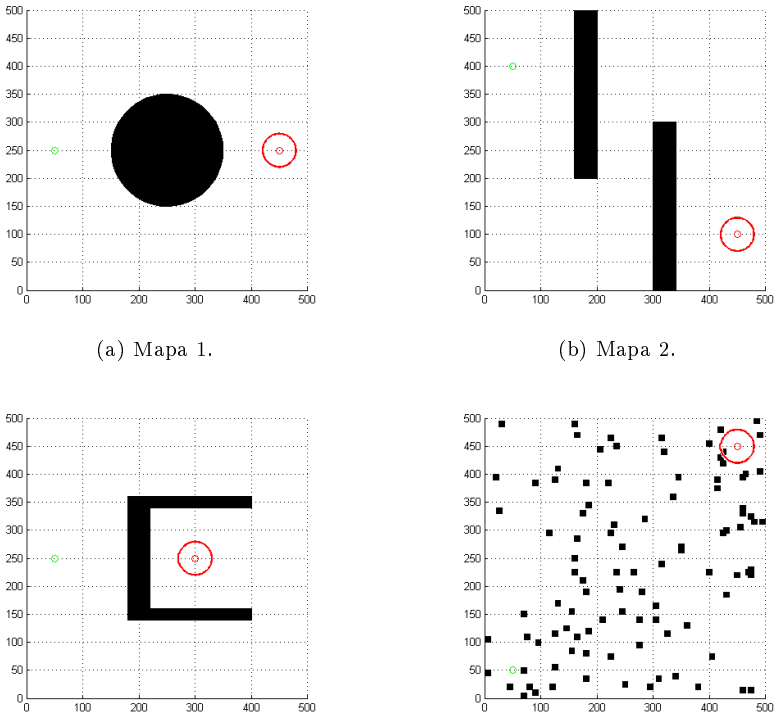
Figura 2.1: Ejemplos de obstáculos.

El espacio abierto, sin embargo, es un claro ejemplo de una ciudad o de un bosque, donde el tamaño del UAV sería despreciable. Este tipo de entornos suele tener una cantidad abundante de obstáculos de pequeñas dimensiones, lo que ralentiza la computación.

No obstante, al tener tantas posibilidades de modelar un entorno, es interesante tener en cuenta también la forma de estos. La mayoría de los algoritmos son capaces de abarcar unicamente obstáculos que tengan formas muy sencillas como: cuadrados, rectángulos u obstáculos puntuales. En cambio, tuberías, árboles y otros elementos del entorno pueden presentar una forma muy irregular, lo que afectaría considerablemente la precisión del algoritmo. En este trabajo, los mapas están creados de tal forma que cualquier obstáculo pueda ser programado como un polígono. Esto puede elevar los costes de computación, pero mejora notablemente la precisión y los costes de las sendas encontradas. En la Figura 2.1 se muestran las diferencias de un polígono de muchas caras, que imita un obstáculo circular y uno simple, que sustituye el círculo por un cuadrado.

En total hay 4 escenarios programados, estos son:

- **Mapa 1**: un escenario, contenido en un plano vertical, de dimensiones reducidas y con un obstáculo circular en medio. Simula una habitación o un entorno cerrado por el que pasa una tubería. Ver Figura 2.2a.
- **Mapa 2:** este mapa también simula un entorno pequeño con paredes verticales. Una colocación específica de los puntos inicial y final, pueden crear «mínimos locales» o «trampas» que pueden confundir algoritmos basados en campos potenciales o similares. Ver Figura 2.2b.
- **Mapa 3:** este escenario recrea un un espacio cerrado con un obstáculo en forma de «C». Puede presentar dificultades para algunos algoritmos debido ala formación de mínimos locales. También está contenido en un plano vertical. Ver Figura 2.2c.
- **Mapa 4:** este mapa, situado en el plano horizontal, simula un espacio abierto, de grandes dimensiones, con una importante cantidad de obstáculos simples. Este entorno puede representar un bosque de árboles o



(a) Mapa 1.

(b) Mapa 2.

(c) Mapa 3.

(d) Mapa 4.

Figura 2.2: Entornos propuestos para este trabajo. El círculo verde indica el estado inicial y, el rojo, el objetivo final.

una ciudad con edificios. En este caso, las dimensiones del UAV serían despreciables. Ver Figura 2.2d.

Los planos en los que están contenidos los entornos se han elegido para concordar con las situaciones reales. Su posición no es relevante para los algoritmos, pero se va a mantener así en el resto del trabajo.

Por último, cabe decir que los entornos creados tienen que estar adaptados para la funcionalidad de todos los algoritmos estudiados: los planificadores probabilísticos y los algoritmos de búsqueda tienen diferentes enfoques a la hora de procesar la información, por ello a los algoritmos A* y θ^* se les ha introducido una parte de preprocesamiento del mapa. El preprocesamiento no se tiene en cuenta a la hora de comparar los tiempos de los programas, ya que, se pretende comparar únicamente los tiempos de ejecución de los algoritmos con un mapa ya dado. Asimismo, se crean condiciones iguales para todos los algoritmos. En este caso, los obstáculos están definidos como polígonos, por lo que es una parte necesaria para su funcionamiento.

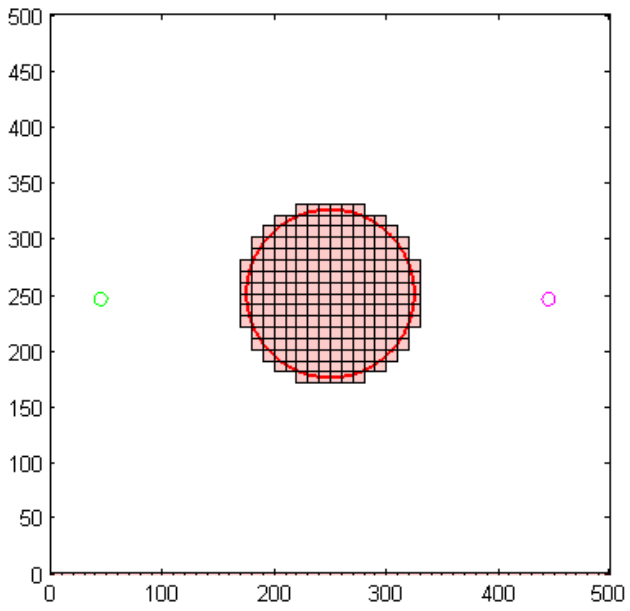


Figura 2.3: Descomposición de un obstáculo en celdas para A* y θ^*.

En la Figura 2.3 se ve como se descompone un obstáculo circular y lo marca con celdas que indican una zona inaccesible.

Capítulo 3

Algoritmos de planificación de caminos estudiados

Como ya se ha visto en la introducción, existen numerosos métodos para planificar el movimiento de una máquina autónoma. Aquí se van a comentar con más profundidad los algoritmos más usados para los vehículos aéreos. En concreto, se van a explicar algunos de los algoritmos de búsqueda de grafos y varios algoritmos probabilísticos que se expondrán continuación.

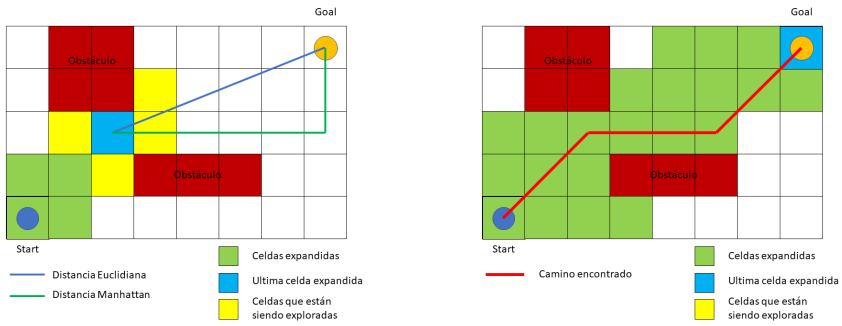
Siendo el objetivo de este trabajo, comparar el comportamiento de diferentes algoritmos, lo primero que hay que hacer es definir los criterios de la evaluación. Se van a tener en cuenta los siguientes parámetros:

- El tiempo de resolución: es una de las características más importantes de los algoritmos, ya que, muchas veces determina su viabilidad para diferentes aplicaciones. Esto también engloba el coste computacional y el uso de la memoria, que están estrechamente relacionados con el tiempo dedicado.
- La calidad del camino encontrado: esta característica indica si la ruta encontrada se parece a una real, que elegiría una persona o no. También se tiene en cuenta el coste del camino encontrado y cuánto de óptimo es. Por coste se entiende una función que calcula el «esfuerzo» que tiene que hacer el UAV para llegar a un punto determinado, y puede ser cualquier parámetro: coste energético, consumo de combustible, etc. En este caso, por simplicidad, el coste va a ser equivalente a la distancia que recorre el dron. Por tanto va a ser una función lineal.

3.1. Algoritmos de búsqueda (Search algorithms)

3.1.1. Algoritmo A*

Este algoritmo de búsqueda de camino es uno de los más usados debido a su alto rendimiento, fácil implementación y robustez. A* [7] es una variación



(a) Funcionamiento de A* y dos heurísticas diferentes, aplicables.

(b) Ejemplo del resultado final

Figura 3.1: Ejemplo de A*

del Dijkstra [5] que incluye una tendencia hacia la meta, lo que ayuda a reducir los tiempos de computación y ahorra la exploración innecesaria. Básicamente, el algoritmo explora todos los posibles recorridos desde el inicio hasta la meta y encuentra el menos costoso de todos.

Se aplica, sobre todo, en mapas de carreteras ya construidos o espacios divididos en celdas. En este caso, al tener obstáculos definidos como polígonos, el algoritmo necesita una función de preprocesamiento del mapa, para poder crear la cuadrícula sobre la que va a trabajar. Una vez creada la cuadrícula, esta actúa como un grafo, donde el algoritmo tiene que encontrar la ruta desde el inicio hasta la meta.

En cada iteración del bucle principal, A* determina qué nodo quiere expandir para seguir construyendo la senda. Para hacerlo, necesita estimar el coste (peso) del nodo actual. En concreto, el algoritmo utiliza la siguiente función

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

donde s es el último nodo en el camino $g(s)$ es el coste del recorrido desde el inicio hasta el nodo s, y $h(s)$ es la heurística que estima el coste del camino mas corto desde el nodo n hasta el final. La idea es encontrar, entre todos las sendas posibles, la que minimiza esta función, habiendo llegado a este nodo. Es decir, estando en el nodo s, el programa estima si este podría ser uno de los nodos del camino más corto. Para poder hacerlo, la heurística ha de cumplir el criterio de admisibilidad. Por admisibilidad se entiende que el valor $h(s)$ nunca puede sobrestimar el coste real de de llegar al objetivo $h_r(s)$, asimismo asegurándose de que el trayecto encontrado es siempre el menos costoso. Hay varias formas de estimar el valor de $h(s)$, pero en este caso se ha decidido elegir la distancia directa entre el nodo y la meta, también llamada distancia Euclidiana.

En principio, todos los nodos disponibles para la exploración están marcados como *abiertos*. A medida que va avanzando, el programa marca los nodos ya

explorados o no alcanzables como *cerrados.* Teniendo los datos de los nodos abiertos/cerrados se consigue que el algoritmo no explore el mismo más de una vez, lo que aumenta su eficiencia.

El algoritmo está representado en la Figura 3.1a y funciona de la siguiente forma:

1. El programa busca un nodo sucesor, abierto y adyacente a los ya explorados, y que tenga el mínimo valor de $f(s)$.
2. Una vez encontrado el nodo, lo marca como cerrado y calcula los costes estimados $f(s')$ de los nodos cercanos. En este caso, al tratarse de un mapa dividido en celdas, un nodo puede tener como máximo 8 vecinos que lo rodean.
3. Si el nodo sucesor es el objetivo, la exploración se para. Si no lo es, vuelve al paso 1

Si hay una solución posible, el programa siempre la encontrará, de lo contrario, se parará cuando no queden más nodos abiertos para la exploración. Este planificador se describe con más detalle en Algoritmo 1.

Dentro del algoritmo aparecen las siguientes funciones principales:

- *Main*: el bucle principal que explora los nodos abiertos, sigue funcionando hasta que no encuentre la solución o hasta que todos los nodos no estén explorados.
- *UpdateVertex:* función que se encuentra dentro del bucle principal y se encarga de dar el paso al siguiente nodo de coste mínimo

3.1.2. Algoritmo θ^*

El algoritmo A* es muy eficiente, sin embargo, al estar unicamente construidos sobre los bordes de la cuadrícula, los caminos encontrados suelen tener un aspecto poco realista y no ser óptimos del todo.

De la necesidad de mejorar dicho aspecto y diminuir el coste, nace la idea del algoritmo θ^*[4].

Esencialmente se trata de quitar la limitación de solo poder construir la ruta siguiendo tan solo 8 direcciones, características del algoritmo A*. Esto se consigue combinando el algoritmo A* con los grafos de visibilidad, lo que permite crear caminos en cualquier dirección. La principal diferencia entre θ^* y A* es que el nodo padre del nodo sucesor tiene que estar a su lado usando A*, mientras que con θ^* el nodo padre puede ser cualquiera.

El algoritmo θ^* básico es casi idéntico al A*, con la excepción de que cuando este actualiza el valor de $g(s)$ y el padre del nodo inexplorado s', vecino cercano del nodo s, considera dos caminos diferentes:

- Camino 1: Este camino es el mismo que toma el algoritmo A*, es decir, al camino ya existente del inicio hasta el nodo s, le añade la linea que une s con s'.

Algoritmo 1: Pseudocódigo de A*.

```
1  begin Main()
2      g(s_start) := 0;
3      parent(s_start) := (s_start);
4      open := ∅;
5      open.Insert(s_start, g(s_start) + h(s_start));
6      closed := ∅;
7      while open ≠ ∅ do
8          s := open.Pop() ;
9          if s := s_goal then
10             return "path found" ;
11         end
12         closed := closed ∪ {s};
13         [UpdateBounds(s)] Solo se usa en θ* ;
14         for s' ∈ succ(s) do
15             if s' ∉ closed then
16                 if s' ∉ open then
17                     g(s') := ∞ ;
18                     parent(s') := NULL  ;
19                 end
20                 UpdateVertex(s, s') ;
21             end
22         end
23     end
24 end

25 begin UpdateVertex(s, s')
26     if g(s) + c(s, s') < g(s') then
27         g(s') := g(s) + c(s, s') ;
28         parent(s') := s;
29         if s' ∈ open then
30             open.Remove(s') ;
31         end
32         open.Insert(s', g(s') + h(s')) ;
33     end
34 end
```

- Camino 2: θ^* considera también otro camino que va desde el padre del vértice s, directamente hasta s' en línea recta. Si el camino esta libre de obstáculos, esta opción siempre será menos costosa que la anterior por la desigualdad de triángulos.

El programa comprueba si es posible avanzar por el Camino 2, si no, se toma el Camino 1, lo que asegura que siempre el algoritmo siempre encuentre la solución, si es que la hay. En la Figura 3.2 se puede ver la diferencia entre los caminos encontrados de θ^* y A*.

Es importante señalar que la comprobación de si el camino es libre de obstáculos no es tan trivial. La solución que utiliza θ^* básico es bastante rápida, ya que utiliza el algoritmo de Bresenham [3]. Este algoritmo consiste en examinar las celdas por donde pasaría el camino y ver, una por una, si alguna de esas celdas esta obstaculizada. Si no se encuentra con ningún obstáculo, el camino se considerará como válido.

En este trabajo en concreto, se ha programado AP-θ^* [4], que es otra modificación del algoritmo que usa nociones de visibilidad. La visibilidad indica si «se ve» el nodo deseado s' desde el nodo predecesor de s. Si hay un obstáculo en medio, este impide que haya una linea de visión entre ellos, lo que significaría que no hay camino directo. En la Figura 3.3 se muestra como un obstáculo bloquea el campo de visión del nodo padre. En otras palabras, los nodos dentro del cono no serán capaces de conectarse directamente al nodo seleccionado.

Teniendo esto en mente, se ha programado el algoritmo *line-of-sight* que utiliza ángulos θ_1 y θ_2, que son los ángulos que forma la recta entre s_{parent} y s, con los vértices de los obstáculos más cercanos, para para definir el campo de visión. El algoritmo implementado no sigue exactamente el procedimiento teórico. En vez de usar unicamente los ángulos, también incluye comprobación de si algún punto del obstáculo se encuentra dentro del triángulo formado por s_{parent}, s y s'. El procedimiento viene explicado con detalle en la Figura 3.4.

Dicho algoritmo se describe con más detalle en el pseudocódigo teórico que se muestra en los Algoritmos 2 y 3.

El algoritmo principal sigue siendo igual a A* pero las funciones dentro de este cambian:

- *UpdateVertex*: ahora considera dos caminos posibles: el que tomaría A* y el que actualiza s_{parent}de s', para acortar el camino.

- *UpdateBounds:* calcula el cono de visión del nodo predecesor. Lo consigue buscando los ángulos límite entre los que el camino estaría libre de obstáculos. Si los nodos s'están dentro de estos límites, significa que están en su campo de visión.

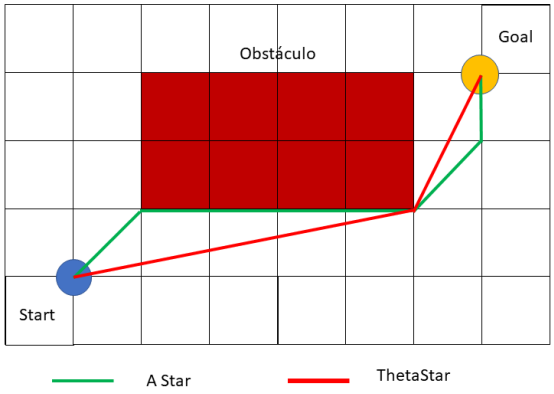


Figura 3.2: θ^* comparado con A*.

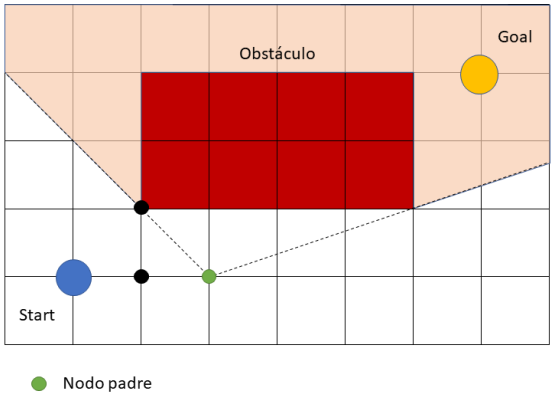
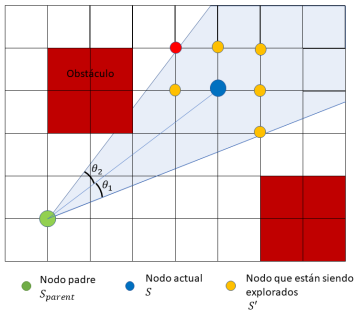
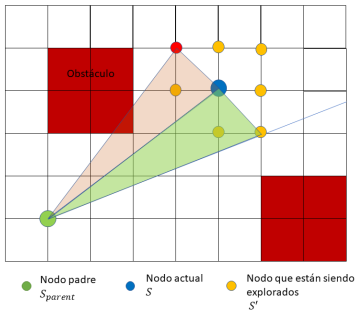


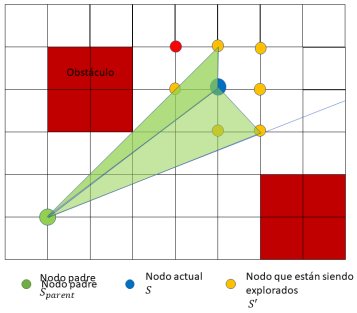
Figura 3.3: Visibilidad de un nodo. El área roja se encuentra fuera de su campo de visión.



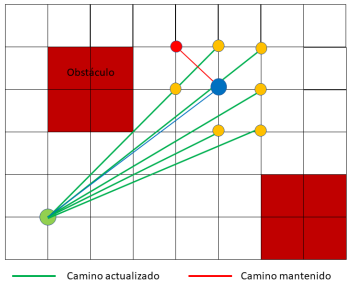
(a) El algoritmo comprueba los ángulos del cono de visión necesario para conectar todos los nodos s' al nodo predecesor.



(b) Se forman dos triángulos, con la máxima apertura posible, que comprueban si hay algún punto del obstáculo dentro del camino. Si resulta que el triángulo se corta con un obstáculo, s' que lo forma no tiene visibilidad del padre.



(c) Se coge el siguiente triángulo que forma el mayor ángulo de θ posible y se vuelve a hacer la comprobación. Si no colisiona, todos los s'que se encuentren dentro de θ tienen visibilidad sobre s_{parent}.



(d) Aquí se muestran los caminos finales.

Figura 3.4: Metodología de detección del campo de visión, utilizado por θ^* implementado.

Algoritmo 2: Algoritmo AP-θ^*. Primera parte.

```
1  begin UpdateVertex(s, s')
2      if s ≠ s_start AND lb(s) ≤ Θ(s, parent(s), s') ≤ ub(s) then
3          /* Path 2 */ ;
4          if g(parent(s)) + c(parent(s), s') < g(s') then
5              g(s') := g(s) + c(s, s') ;
6              parent(s') := parent(s);
7              if s' ∈ open then
8                  open.Remove(s') ;
9              end
10             open.Insert(s', g(s') + h(s')) ;
11         end
12     else
13         /* Path 1 */ ;
14         if g(s) + c(s, s') < g(s') then
15             g(s') := g(s) + c(s, s') ;
16             parent(s') := s;
17             if s' ∈ open then
18                 open.Remove(s') ;
19             end
20             open.Insert(s', g(s') + h(s')) ;
21         end
22     end
23 end
```

Algoritmo 3: Algoritmo AP-θ^*. Segunda parte.

```
1  begin UpdateBounds(s)
2      lb(s) := −∞; ub(s) := ∞ ;
3      if s ≠ s_start then
4          foreach blocked cell b adjacent to s do
5              if ∀s' ∈ corners(b): parent(s) = s' OR Θ(s, parent(s), s') < 0
                OR (Θ(s, parent(s), s')) = 0 AND
                c(parent(s), s') ≤ c(parent(s), s)) then
6                  lb(s)=0 ;
7              end
8              if ∀s' ∈ corners(b): parent(s) = s' OR Θ(s, parent(s), s') > 0
                OR (Θ(s, parent(s), s') = 0 AND
                c(parent(s), s') ≤ c(parent(s), s)) then
9                  ub(s)=0 ;
10             end
11         end
12         foreach s' ∈ nghbrs_vis(s) do
13             if s' ∈ closed AND parent(s) = parent(s') AND s' ≠ s_start
                then
14                 if lb(s') + Θ(s, parent(s), s') ≤ 0 then
15                     lb(s) := max(lb(s), lb(s') + Θ(s, parent(s), s')) ;
16                 end
17                 if ub(s') + Θ(s, parent(s), s') ≥ 0 then
18                     ub(s) := min(ub(s), ub(s') + Θ(s, parent(s), s')) ;
19                 end
20             end
21             if c(parent(s), s') < c(parent(s), s)) AND parent(s) ≠ s' AND
                (s' ∉ closed OR parent(s) ≠ parent(s')) then
22                 if Θ(s, parent(s), s') < 0 then
23                     lb(s) := max(lb(s), Θ(s, parent(s), s')) ;
24                 end
25                 if Θ(s, parent(s), s') > 0 then
26                     ub(s) := max(ub(s), Θ(s, parent(s), s')) ;
27                 end
28             end
29         end
30     end
31 end
```

3.2. Algoritmos basados en el muestreo (Sampling-Based Algorithms)

3.2.1. Algoritmo RRT

A continuación se explica el funcionamiento del algoritmo RRT básico [13] y su implementación.

Como ya se ha explicado anteriormente, este algoritmo organiza los nodos en una estructura de árbol, que se caracteriza por que cada nodo tiene únicamente un padre.

La idea principal es construir un árbol que explore uniformemente el mapa de obstáculos. La uniformidad se consigue gracias a la elección aleatoria de puntos.

Teniendo un punto inicial p_s , una configuración final p_e y la distribución de los obstáculos en el espacio, el algoritmo sigue la siguiente metodología:

1. Se selecciona un punto aleatorio del entorno, p y se une, mediante una linea, con el punto más cercano a este $x_{nearest}$. Si es el primer paso se une con el inicial p_s.

2. En la dirección del segmento creado, se coloca un nodo nuevo v^+, a una distancia máxima D, midiendo desde $x_{nearest}$.

3. Se comprueba si el camino encontrado esta libre de los obstáculos. En el caso contrario se descarta el nodo y el punto elegidos y se empieza de nuevo

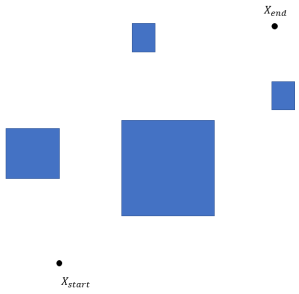
4. Se comprueba si es posible alcanzar la configuración final p_e desde el nodo v^+, si no es posible se pasa a la siguiente iteración y se vuelve al paso 1.

El nodo final p_e puede sustituirse por *Goal*, lo que significa que el UAV tiene que llegar a una zona determinada en vez de un punto final o que, al llegar a esta zona, se considere que ha alcanzado p_e. Este cambio necesario para los planificadores probabilísticos, ya que, es muy poco probable que encuentren un punto concreto en el espacio.

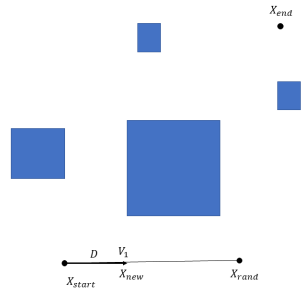
En la Figura 3.5 se puede ver el funcionamiento del dicho algoritmo. El pseudocódigo utilizado para la programación esta descrito con exactitud en el Algoritmo 4.

Las principales funciones de este algoritmo son:

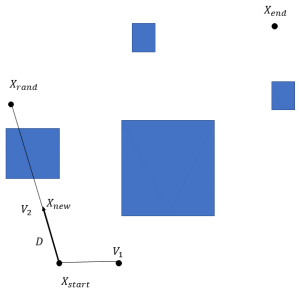
- *SampleFree*: escoge un punto aleatorio x_{rand}.
- *Nearest*: busca el nodo más cercano al punto aleatorio $x_{nearest}$.
- *Steer*: situá un nodo x_{new} en la linea entre x_{rand} y $x_{nearest}$ a una distancia concreta del segundo.
- *ObstacleFree:* esta función detecta si el camino está libre de obstáculos. Está explicada con más detalle a continuación.



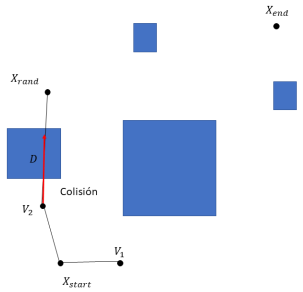
(a) Configuración inicial con obstáculos.



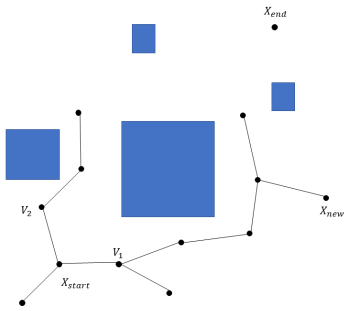
(b) Camino sin obstáculos.



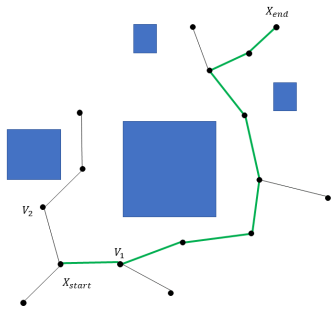
(c) Punto aleatorio detrás del obstáculo pero no hay colisión



(d) Colisión



(e) Crecimiento del árbol



(f) Camino encontrado

Figura 3.5: Procedimiento seguido por RRT para encontrar un camino.

Algoritmo 4: Pseudocódigo de RRT básico.

```
1  begin RRT
       Data: Terrain map M, start configuration p_s, end configuration p_e
2      V ← {x_init}; E ← ∅ ;
3      while The end node p_e, is not connected to the tree: p_e ∉ V do
4          x_rand ← SampleFree_i ;
5          x_nearest ← Nearest(G = (V, E), x_rand) ;
6          x_new ← Steer(x_nearest, x_rand) ;
7          if Obstaclefree(x_nearest, x_new) then
8              V ← V ∪ {x_new}; E ← E ∪ {x_nearest, x_new} ;
9          end
10     end
11     return G = (V, E) ;
12 end
```

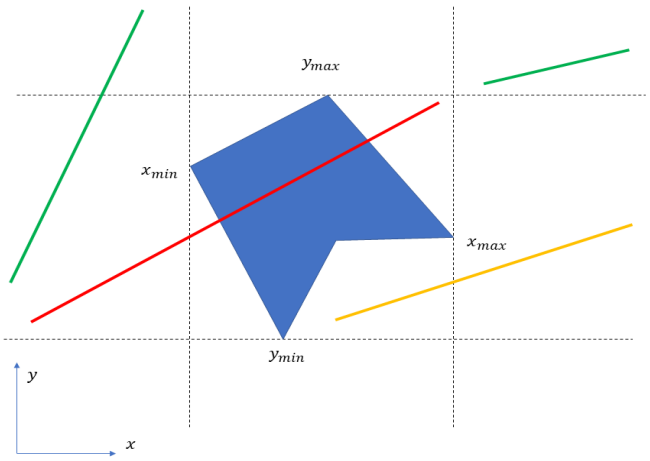


Figura 3.6: Doble comprobación de colisiones: Los segmentos verdes están lejos del espacio ocupado por el obstáculo y no hace falta la segunda comprobación. La linea naranja esta cerca del obstáculo pero no colisiona. El segmento rojo colisiona.

En cuanto a la detección de obstáculos, se ha implementado un algoritmo de doble comprobación: la primera muy rápida y sencilla que predice si hay riesgo de colisión, y la segunda, mas costosa, que comprueba definitivamente si la ruta está obstaculizada. Concretamente funciona de la siguiente forma:

1. Como se observa en la Figura 3.6, el algoritmo busca los puntos máximos y mínimos de las coordenadas de un obstáculo. Una vez hecho esto, mira si alguno de los puntos del último segmento trazado se encuentran dentro de este campo. Si no los hay, el camino no se cruza con el obstáculo.

2. Si se da el caso de que al menos un punto de la ruta está en la región delimitada por la primera parte del algoritmo, se hace la segunda comprobación donde se tiene en cuenta la forma real del obstáculo, pudiendo ser esta un polígono cualquiera. Esta comprobación divide el obstáculo en segmentos que lo delimitan y, con la función *polyxpoly* de MATLAB, verifica si algún segmento del obstáculo corta con el segmento seleccionado del camino.

3.2.2. Algoritmo RRT*

El algoritmo RRT* [8] se obtiene modificando RRT simple de tal forma que se evitan los ciclos, eliminando las esquinas «redundantes». Es decir, quitando las uniones entre los nodos que no forman parte del camino más corto desde la raíz del árbol.

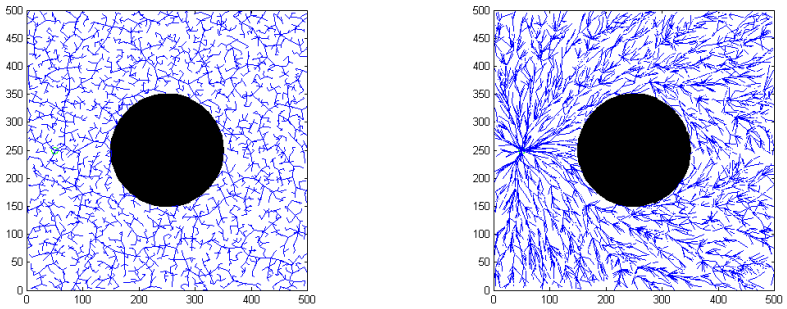
El funcionamiento básico es el mismo que el de RRT, aunque su funcionalidad es algo diferente. Este algoritmo está diseñado para optimizar el coste del camino después de encontrarlo. Esto se consigue dejando que el algoritmo siga funcionando durante más tiempo.

se le añaden dos modificaciones: «nodos cercanos» (*Near*) y «reconexión» (*Rewire*).

La función *Near* toma lugar después de que el algoritmo principal haya encontrado un nodo nuevo x_{new} y, por tanto, también es conocido su nodo padre $x_{nearest}$. Una vez hecho esto, la función busca otros vecinos cercanos en un radio determinado k, que por simplicidad, se ha programado como un valor constante, aunque no siempre lo es.

A continuación, la función *Rewire* determina si es posible reconectar el árbol de tal forma que los nodos cercanos X_{near} sean los sucesores del nodo actual x_{new}. Si las conexiones no están obstaculizadas, se calculan los nuevos *costes* de cada uno de los nodos cercanos X_{near} teniendo x_{new} como padre. Por coste se refiere el esfuerzo a la distancia que tiene que recorrer el UAV para llegar a este nodo. En este trabajo, los costes se consideran como distancias acumuladas desde el punto de inicio hasta dicho nodo, siguiendo la ruta encontrada. Si alguno de los nuevos costes resulta ser menor que el que tenía antes, la función elimina el enlace anterior y se queda con el camino reconectado.

Esto permite que el árbol sea menos «rizado» como se ve en la Figura 3.7, minimizando los costes y eliminando las conexiones innecesarias. El pseudocódi-



(a) RRT (b) RRT*

Figura 3.7: Comparación de árboles de RRT y RRT*.

go implementado viene dado por Algoritmo 5, que utiliza las mismas funciones principales que RRT, más las funciones *Near* y *Rewire* explicadas anteriormente.

3.2.3. Algoritmo RRT-Connect

RRT-Connect [11] es una modificación del planificador RRT que está diseñada especialmente para los problemas que no incluyen ligaduras diferenciales. En este caso, el algoritmo simplemente intenta moverse de la forma más rápida posible sin tener en cuenta las posibles restricciones del sistema. En otras palabras, se centra en sistemas holónomos, donde el estado de estas depende únicamente de los estados inicial y final.

El método esta basado en dos ideas principales: la heurística Connect que intenta moverse a distancias más largas, y el crecimiento de dos árboles desde el estado inicial y el final.

La heurística Connect es una alternativa a la función de RRT que elige donde poner el siguiente nodo. En RRT simple, el nodo nuevo se coloco a una distancia máxima D del padre. Sin embargo, en este caso, Connect intenta unir el nodo padre directamente con el punto aleatorio p o se para cuando el camino está obstaculizado. Esta heurística permite convergencia muy rápida hacia la solución, aunque su comportamiento aleatorio evita que se quede atascado en un «mínimo local» como pasa con los planificadores potenciales de rápida convergencia.

La otra función añadida consiste en que el algoritmo tenga dos raíces p_s y p_e (en vez de una única raíz p_s), de las que crecen dos árboles $\mathcal{T}_a$ y $\mathcal{T}_b$. En cada iteración del bucle principal, un árbol se expande y comprueba si es capaz de hacer conexión con un vértice del otro árbol. Después se cambian los roles cambiando de lugar los dos arboles.

Existen numerosas modificaciones para este algoritmo, dependiendo de si se quiere que este tenga una heurística más voraz o que explore el espacio en busca de soluciones más óptimas. En este caso, se ha elegido fomentar la rapidez del

Algoritmo 5: RRT*.

Data: Terrain map M, start configuration p_s, end configuration p_e

begin RRT*
  $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset$;
  while *The end node p_e, is not connected to the tree: $p_e \notin V$* **do**
    $x_{rand} \leftarrow \text{SampleFree}_i$;
    $x_{nearest} \leftarrow \text{Nearest}(G = (V, E), x_{rand})$;
    $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$;
    if *Obstaclefree($x_{nearest}, x_{new}$)* **then**
      $X_{near} \leftarrow \text{Near}(G = (V, E), x_{new}, D)$;
      $V \leftarrow V \cup \{x_{new}\}$;
      $x_{min} \leftarrow x_{nearesrt}$; $c_{min} \leftarrow$ $\text{Cost}(x_{nearest})+c(\text{Line}(x_{nearest}, x_{new}))$;
      foreach $x_{near} \in X_{near}$ **do**
        if *CollisionFree(x_{near}, x_{new})* $\wedge$ *Cost($x_{near} + c(Line(x_{near}, x_{new}))$<$c_{min}$* **then**
          $x_{min} \leftarrow x_{near}$; $c_{min} \leftarrow$ $\text{Cost}(x_{near} + c(\text{Line}(x_{near}), x_{new}))$
        end
      end
      $E \leftarrow E \cup \{x_{min}, x_{new}\}$;
      foreach $x_{near} \in X_{near}$ **do**
        if *CollisionFree(x_{new}, x_{near}* $\wedge$ *Cost($x_{new} + c(Line(x_{new}, x_{near}))$ <Cost(x_{near})* **then**
          $x_{parent} \leftarrow \text{Parent}(x_{near})$;
          $E \leftarrow (E \backslash \{(x_{parent}, x_{near})\}) \cup \{(x_{new}, x_{near})\}$
        end
      end
    end
  end
  return $G = (V, E)$;
end

algoritmo, haciendo que los dos arboles conecten al tener algunos de los nodos en su campo de visión. Esta alteración debería permitir bajar bastante el tiempo de computación a expensas del coste de camino, que podría ser lejos del óptimo.

En las Figuras 3.8 y 3.9 se puede observar el comportamiento de este método, mientras que el pseudocódigo viene descrito en el Algoritmo 6.

Las funciones nuevas dentro de este algoritmo son las siguientes:

- *Connect:* esta función conecta directamente $x_{nearest}$ con x_{rand}.
- *ExtendUntilObstacle:* si se da el caso que la linea obtenida gracias a *Connect* está obstaculizada, esta función hace que se pare justo delante del obstáculo sin tocarlo. Este nuevo punto se tomaría por x_{new}.

Algoritmo 6: RRT-Connect.

```
1  begin RRT-Connect
       Data: Terrain map M, start configuration p_s, end configuration p_e
2      V ← {x_init}; E ← ∅ ;
3      while The end node p_e, is not connected to the tree: p_e ∉ V do
4          x_rand ← SampleFree_i ;
5          x_nearest ← Nearest(G = (V, E), x_rand) ;
6          x_new ← Connect(x_nearest, x_rand) ;
7          if Obstaclefree(x_nearest, x_new) then
8              V ← V ∪ {x_new}; E ← E ∪ {x_nearest, x_new} ;
9          else
10             x_new ← extendUntilObstacle (x_nearest)
11         end
12     end
13     return G = (V, E) ;
14 end
```

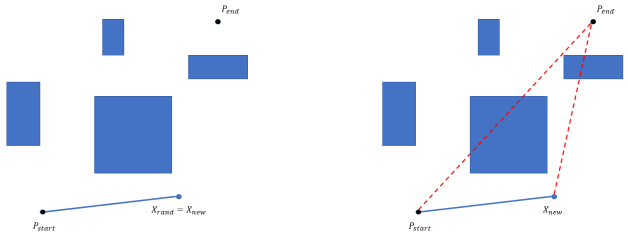
3.3. Modificaciones y mejoras

Los algoritmos descritos anteriormente están presentados en su forma más general, lo que significa, que están en su forma más básica para poder ser adaptados a las aplicaciones posteriores.

A continuación se van a explicar algunas modificaciones de lo algoritmos que pueden tener un gran interés para las aplicaciones reales.

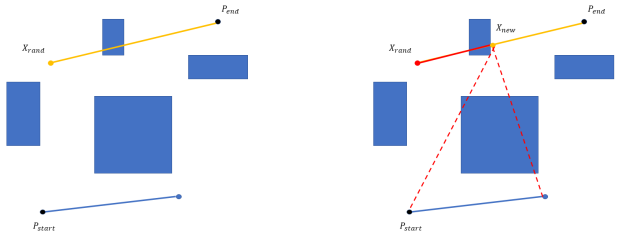
3.3.1. Introducción del Sesgo/Bias

El tiempo de resolución de los algoritmos basados en RRT depende totalmente del azar de los puntos generados, lo que hace imposible predecir la duración de la ejecución. Por ello, es interesante introducir un sesgo, o también llamado



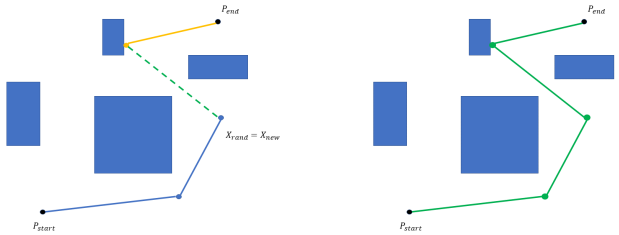
(a) El algoritmo empieza en una rama, escogiendo un punto al azar. Si estepunto no está obstaculizado se convierte en el nodo nuevo.

(b) Se comprueba si es visible algún punto de otro árbol desde el primero.



(c) Se intercambian los árboles, y el bucle empieza de nuevo. Se escoge unnodo punto al azar.

(d) Si el punto no esta accesible, el nuevo se pone en la misma línea justo al tocar el obstáculo. SE comprueba si tiene visibilidad sobre el otro árbol



(e) Se vuelven a intercambiar los árboles. El azul vuelve a poner un punto y comprueba la visibilidad.

(f) Si el otro árbol puede conectarse directamente con el nuevo punto, se crea la unión y se traza el camino encontrado.

Figura 3.8: Procedimiento de búsqueda de RRT-Connect.

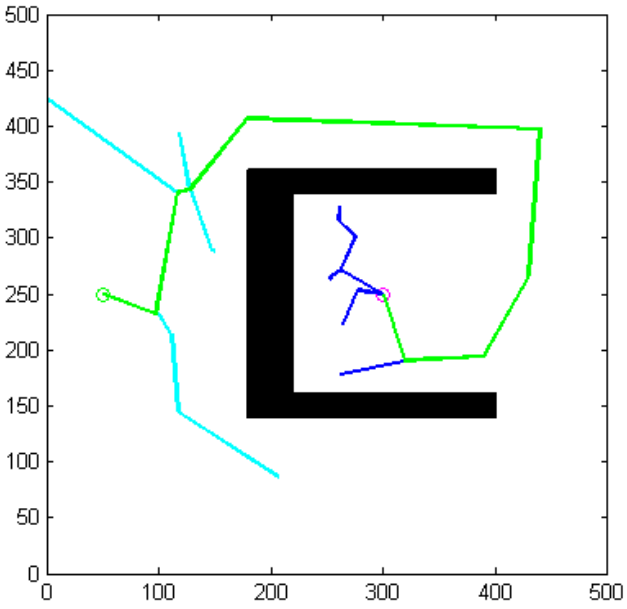


Figura 3.9: RRT-Connect. En la imagen se observan dos árboles: uno cían con la raíz en el punto verde y otro azul con la raíz en el punto rojo. Ambos están conectados por un camino final verde. Se puede intuir la rápida expansión de los árboles ya que apenas hay nodos explorados en el mapa. También se ve que el camino encontrado está lejos de ser óptimo.

Bias, dentro del algoritmo principal que marca una tendencia hacia la meta, lo que ayudaría a reducir el tiempo de búsqueda en muchos casos.

En este trabajo, para todos los algoritmos RRT, se ha introducido una tendencia en la que un cierto por ciento de los puntos de aleatorios se sustituye por x_{goal}. Esto hace posible una convergencia más rápida hacia el resultado, sobre todo si se tiene una vista directa del objetivo desde el árbol.

Al variar el porcentaje, varía la rapidez de la convergencia, aunque esto puede dar lugar a la aparición de los mínimos locales, igual que en los métodos de campos potenciales. Sobre todo, es peligroso aumentar el sesgo en entornos muy complicados como, por ejemplo, un laberinto. En un laberinto, el porcentaje de los puntos sesgados es desperdiciado, ya que, se encuentra con muchas barreras que impiden su funcionamiento.

3.3.2. Dimensiones del UAV

Es interesante también considerar el tamaño del UAV en los entornos de dimensiones reducidas, para poder simular las situaciones reales.

Una forma de hacerlo es modificando los obstáculos: ensanchando su tamaño o poniendo un margen de seguridad alrededor de estos. Este método es bastante rápido y fácil de utilizar, pero aumentando el tamaño de los obstáculos puede conllevar a la eliminación de posibles soluciones.

La otra forma consiste en simular el movimiento del UAV completo, teniendo en cuenta sus dimensiones reales y no solo siguiendo su centro de gravedad teórico. Al estar en un espacio $2D$, solo se necesitan dos dimensiones. En este trabajo en particular, los entornos pequeños están contenidos en los planos verticales, por lo que la forma del vehículo se ha asimilado a un rectángulo de dimensiones $90 \times 20cm$, correspondientes a la longitud y la altura respectivamente. Suponiendo que los trabajos realizados en espacios reducidos son de precisión, resulta razonable proponer la siguiente hipótesis: Los trabajos de precisión requieren movimientos lentos, lo que conlleva a ángulos de inclinación y aceleraciones pequeños, por tanto, se puede considerar que el UAV siempre se mantiene paralelo al suelo.

Teniendo en cuenta la hipótesis anterior, es posible simular el comportamiento de un dron, cuyas dimensiones no son despreciables, sin implicarse en la dinámica ni la cinemática del problema. Esto permite plantear un problema puramente geométrico, como se representa en la Figura 3.10.

La función utilizada para ello, *Hull,* intenta simplificar al máximo la geometría del vehículo durante en movimiento. Supongamos que tenemos un segmento cualquiera por el que se mueve el UAV. Para no tener que hacer múltiples comprobaciones a lo largo de todo el segmento que une dos nodos, se asemeja el paso entre ellos a una forma geométrica, en concreto, un polígono. Esto se puede ver en la Figura 3.11.

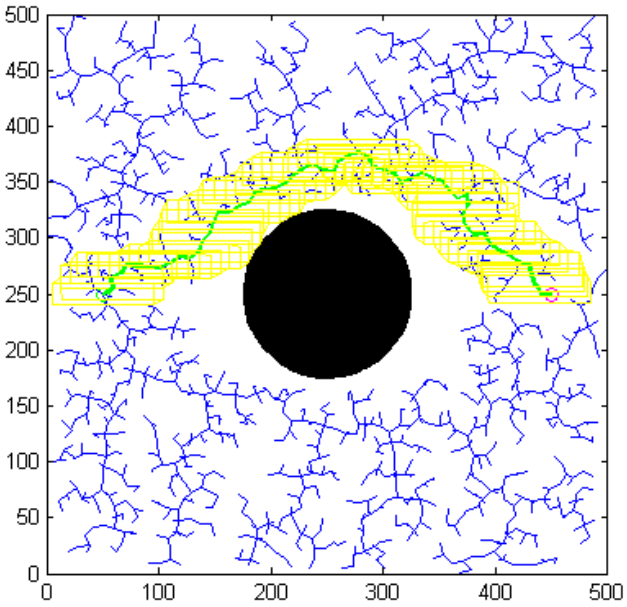
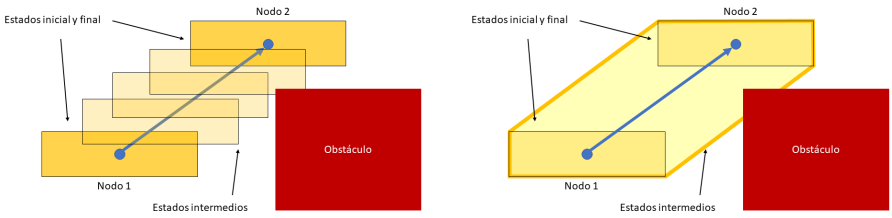


Figura 3.10: Trayectoria del UAV teniendo en cuenta sus dimensiones.



(a) Comprobación con estados intermedios. Co-(b) Comprobación con un único polígono crea-
mo se ve, la fiabilidad de este método dependedo por Hull. Este método no requiere puntos
de la cantidad de puntos intermedios que hay enpara su fiabilidad y es bastante preciso.
el segmento. Más puntos consumen más tiempo

Figura 3.11: Comparación de métodos de comprobación de colisiones.

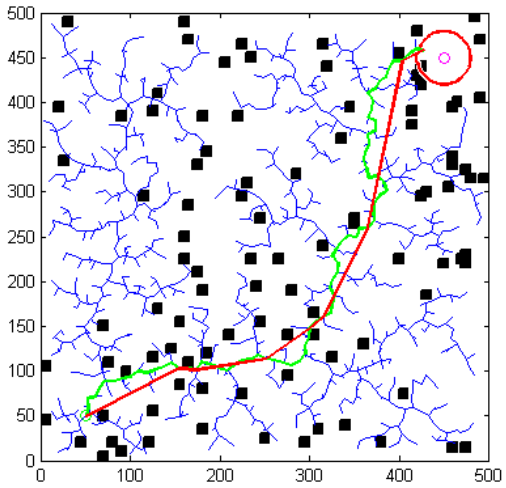


Figura 3.12: Ejempo RRT Smooth Path. El camino verde es el original y el rojo se consigue aplicando esta función.

3.3.3. RRT Smooth Path

Como una posible mejora para los algoritmos de tipo RRT, se ha incluido un suavizado de camino, para optimizar el coste de los resultados e intentar mejorar el aspecto de estas. Esta modificación posibilita acortar la ruta de un camino ya encontrado, saltando los nodos innecesarios.

Su funcionamiento es similar al algoritmo $\theta*$, donde el algoritmo, en vez unir de los nodos más cercanos, intenta unir los que están en su campo de visión. El pseudocódigo utilizado puede verse el ejemplo puesto en la práctica en la Figura 3.12

3.3.4. RRT*2

Como alternativa al algoritmo RRT*, en este apartado se propone otra modificación de este que se ha llamado RRT*2. En si, el programa cumple la misma función que RRT*: intenta construir el árbol quitando conexiones redundantes. La principal diferencia consiste en que a la hora de reconectar, se toma un enfoque diferente.

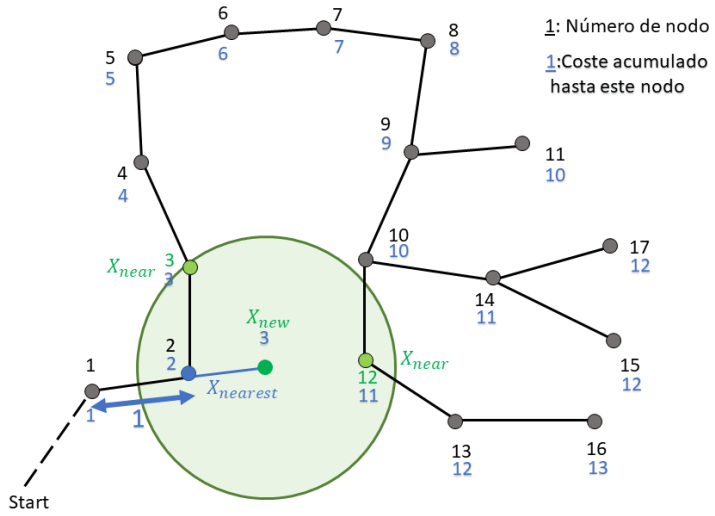
El procedimiento original consiste en reconectar los nodos vecinos al nuevo, de forma que el coste disminuya si se cambia el predecesor. En cambio, esta modificación comprueba si es posible reconectar toda la rama del árbol que conduce al nodo x_{near} para disminuir el coste de todos los nodos sucesores de la rama. Esto permite optimizar más los costes en los arboles muy «rizados».

En las Figuras 3.13 y 3.14 se puede ver la explicación gráfica y la comparación

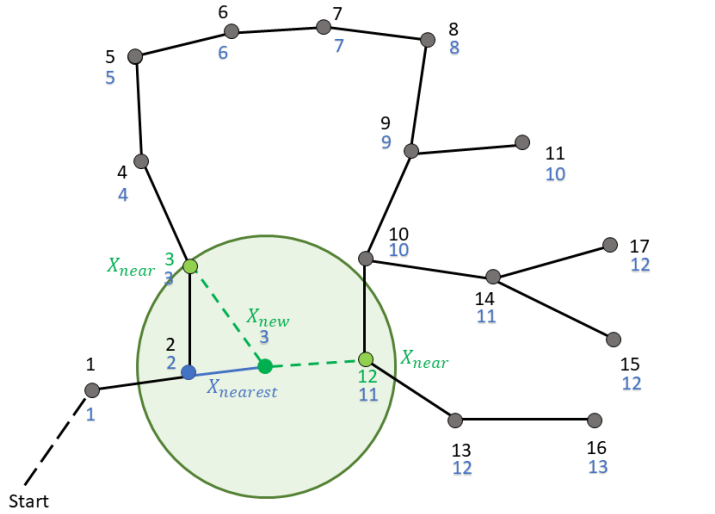
de los dos funcionamientos: se consigue una disminución de costes considerable aplicando este algoritmo.

Para comprender con mas detalle su funcionamiento, a continuación se va a explicar la nueva función que sustituye a *Rewire* de RRT*, llamada *Chain*. La función *Chain* reconecta los nodos de la misma forma, sin embargo la principal diferencia esta en la eliminación de enlaces.

Una vez reconectado el nodo, se crea un círculo cerrado de nodos: una «cadena», donde los enlaces son sus «eslabones» (Figura 3.15). La cadena empieza por el punto que tiene el menor coste acumulado X_c. Después se busca el eslabón más cargado, es decir, donde se alcanza el mayor coste relativo al nodo del comienzo de la cadena X_c, avanzando por los dos lados como se indica en la Figura 3.16. A continuación se rompe el enlace indicado y se reestructura la rama que empieza a partir de X_c. De esta forma, se consigue minimizar los costes en toda la rama cuyo predecesor es X_c, mientras que con RRT* solo se ven afectados los nodos x_{near} y sus sucesores.

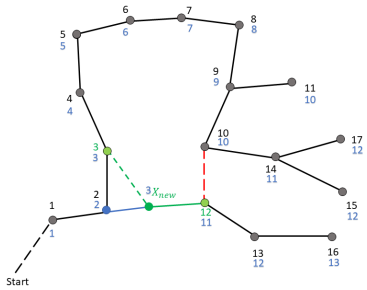


(a) Suponiendo que se tiene este árbol, donde la distancia entre los nodos es 1, y que el nuevo nodo x_{new} tiene varios vecinos cercanos.

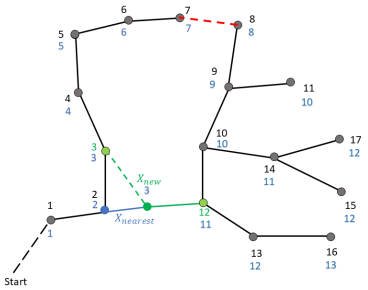


(b) Los dos algoritmos buscan si los vecinos x_{near} se pueden reconectar de tal forma que el nuevo camino sea menos costoso.

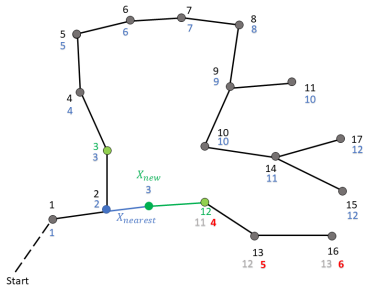
Figura 3.13: Comparación de funcionamiento de RRT* y RRT*2. Comparten el mismo comienzo.



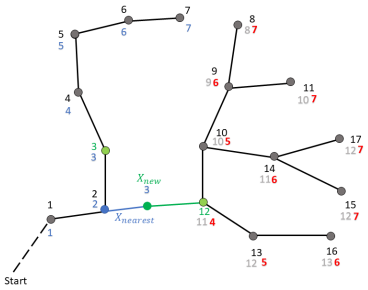
(a) RRT* reconecta el nodo 12, eliminando la conexión con su predecesor, si esto minimiza el coste.



(b) RRT*2 reestructura la rama y desconecta elotro arco, de manera que se minimiza los costes en toda la parte afectada.



(c) RRT* recalcula los costes de los nodos sucesores.



(d) RRT*2 reestructura toda la rama y recalcula los costes nuevos. Este proceso afecta a todos los nuevos sucesores de x_{new}.

Figura 3.14: Comparación de funcionamiento de RRT* y RRT*2. Diferencias.

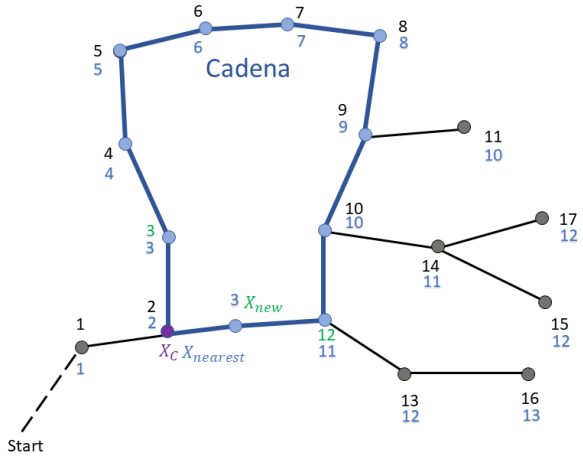


Figura 3.15: Cadena cerrada. El nodo de comienzo en este caso es el 2, que coincide $x_{nearest}$.

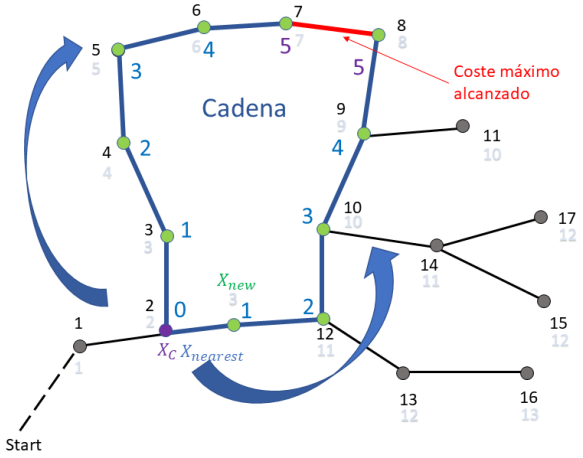


Figura 3.16: Cálculo de costes relativos siguiendo la cadena. Eslabón más costoso es eliminado, minimizando los costes en toda la rama.

Capítulo 4

Comparación de resultados

Teniendo los algoritmos programados y los entornos preparados para las pruebas, el siguiente paso es poner los ensayos en marcha.

En este capítulo se mostrarán las comprobaciones realizadas para poder determinar la eficacia y la eficiencia de los programas implementados. Se van a analizar los algoritmos en su estado básico y su respuesta a los diferentes entornos. Luego, se estudiaran las modificaciones explicadas en el capítulo anterior.

Para poder comparar los algoritmos, cuyas metodologías tienen enfoques tan diferentes, se ha propuesto que la distancia de un paso sea parecida en todos los casos (a excepción de RRT-Connect que elimina la limitación por paso) . En concreto, para RRT y RRT*, la distancia entre el nodo padre x_{parent} y el nodo nuevo x_{new} está delimitada por parámetro D, y en el caso de los algoritmos A* y θ^*, este parámetro controla el tamaño de los lineas de la cuadrícula sobre la que se encuentran los nodos. Este parámetro indica el tamaño relativo de un paso/celda en comparación con las dimensiones del mapa L. Se han usado valores diferentes en los programas: $D = 5$; $D = 10$; $D = 20$; $D = 50$. Teniendo en cuenta que los mapas siempre tienen la misma dimensión $L = 500$, se obtienen diferentes valores relativos de $L/D = 0{,}01$; $L/D = 0{,}02$; $L/D = 0{,}04$; $L/D = 0{,}1$. Hay que recordar, aunque las dimensiones del mapa iguales matemáticamente, estos mapas representan diferentes entornos: los primeros tres simulan una habitación de unas dimensiones $5m$, mientras que el último simula un espacio abierto de $50m$ o más. Las dimensiones absolutas no importan, las relativas son las que hay que tener en cuenta.

Otro parámetro importante es la influencia del Bias/Sesgo o heurística en la resolución del problema. Al introducir una cierta tendencia hacia la meta, un algoritmo puede encontrar la solución más rápido en situaciones de poca dificultad, al no estar «completamente ciego».

Cabe mencionar que, para los planificadores probabilísticos, con el fin de sacar resultados más fiables, se ha hecho una media de 50 pruebas para cada algoritmo. De esta forma, se reduce su aleatoriedad y su carácter impredecible, haciendo que las variables tomen un valor medio, más fidedigno.

En general, para la mayoría de las pruebas se va a usar la misma configura-

ción, al menos que se diga lo contrario: $D = 20$, $(D/L = 0{,}04)$, con $Bias = 5\,\%$ para los planificadores probabilísticos y heurística euclidiana simple aplicada a los planificadores de camino. El algoritmo se para cuando consigue encontrar un camino hacia la meta y no se tiene en cuenta la geometría del UAV.

Para los planeadores probabilísticos se ha definido un área-objetivo/Goal de radio $R = 30$ centrado en el punto final. Su valor es de libre elección y no hay ninguna documentación que especifique su tamaño. Se supone que si el algoritmo se acerca a una distancia R del punto final, significa que ha encontrado el camino. Hay que recordar que este parámetro es necesario, sobre todo, si el sesgo es nulo. Para mantener la igualdad entre los algoritmos de búsqueda y los planeadores probabilísticos, para los segundos, al llegar al Goal, se le añade un coste adicional proporcional a la distancia que queda hasta el punto final, sin completar el camino.

Como un dato adicional, las especificaciones del ordenador en el que se han realizado las pruebas son:

- Samsung NP-RC530
- Procesador Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
- RAM 4 GB
- Windows 10 Home

4.1. Comparativa general de los algoritmos: Tiempo y Coste

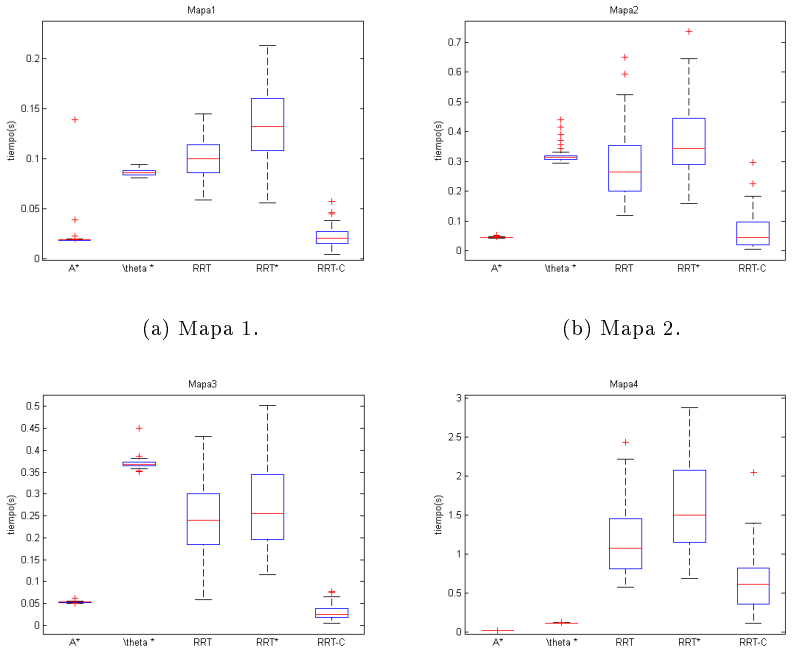
A continuación se van a estudiar los resultados obtenidos de los experimentos.

Una de las características más importantes para determinar si un algoritmo es eficiente es el tiempo, incluso, quizás sea la propiedad más crítica a la hora de determinar si este sirve para una aplicación particular. En la Figura 4.1 se representa el comportamiento general y los tiempos medios obtenidos en todos los mapas.

En la siguiente tabla se muestra el tiempo medio obtenido en segundos.

Algoritmo / Mapa	A*	θ^*	RRT	RRT*	RRT-Connect
1	0.022	0.084	0.100	0.135	0.025
2	0.044	0.295	0.292	0.366	0.058
3	0.051	0.354	0.252	0.274	0.034
4	0.016	0.113	1.052	2.145	0.664

En las gráficas y tabla se observa que A*, por lo general, obtiene los mejores resultados en el tiempo, al ser un algoritmo sencillo y bastante directo. En cambio, θ^* sacrifica el tiempo de resolución para mejorar los costes, llegando a



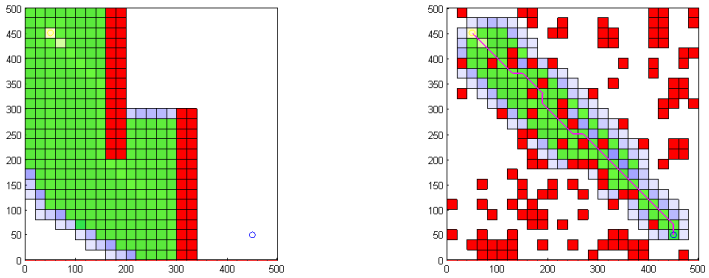
(a) Mapa 1.

(b) Mapa 2.

(c) Mapa 3.

(d) Mapa 4.

Figura 4.1: Tiempos obtenidos después de 50 pruebas.



(a) Mapa 2. A* está explorando la zona que podría considerarse como un mínimo local, lo que ralentiza el proceso. Las celdas exploradas se marcan de color verde y los obstáculos de rojo. El inicio está en la celda amarilla y el objetivo es el círculo azul.

(b) Mapa 4 modificado. La zona explorada es verde y los obstáculos son rojos. A* encuentra un camino bastante directo sin tener que explorar todo el espacio.

Figura 4.2: Ejemplos de exploración del algoritmo A*.

ser unas 5-8 veces más lento que su predecesor. Los tiempos elevados se deben a la comprobación exhaustiva de campo de visión explicado anteriormente. Como se ha explicado anteriormente, que la función line-of-sight está implementada de forma un tanto diferente la teórica: utiliza la combinación de ángulos y triángulos para comprobar el resultado, esto podría ralentizar el proceso.

Ambos planificadores son deterministas, por lo que exploran el área de manera idéntica. La diferencia principal consiste en la conexión entre los nodos explorados. Al tener el mismo procedimiento a la hora de reconocer los nodos nuevos, ambos algoritmos se comportan de la misma manera en entornos parecidos: si la ruta siempre sigue la dirección de la heurística, la encuentran muy rápido, pero si hay obstáculos «trampa» o los que obligan a cambiar de dirección, tardan más en encontrarla debido a los «mínimos locales». Este efecto se observa en la clara diferencia de tiempos en los mapas: el Mapa 1 y el 4 son más directos y, por ello, los tiempos son más bajos que en el 2 y en el 3, cuya geometría retrasa este tipo de planificadores.

En la Figura 4.2 se observa claramente el efecto del obstáculo «trampa» y su influencia en la búsqueda de camino y, en la Figura 4.3 se expone un ejemplo de θ^*, aplicado al Mapa 1.

En cuanto al algoritmo RRT, en media se han obtenido resultados más parecidos al θ^*, aunque la metodología utilizada es totalmente diferente. Sin embargo, el último Mapa lo ralentiza bastante más al tener que comprobar colisiones con cada obstáculo. Siendo un algoritmo probabilístico, existe una zona de incertidumbre donde se concentra la mayoría del rango de valores de los tiempos obtenidos. Esta zona puede ser bastante ancha, por lo que se hace más difícil predecir la rapidez del algoritmo en un caso concreto.

El algoritmo RRT* ha resultado ser el más lento en general. Sus tiempos son

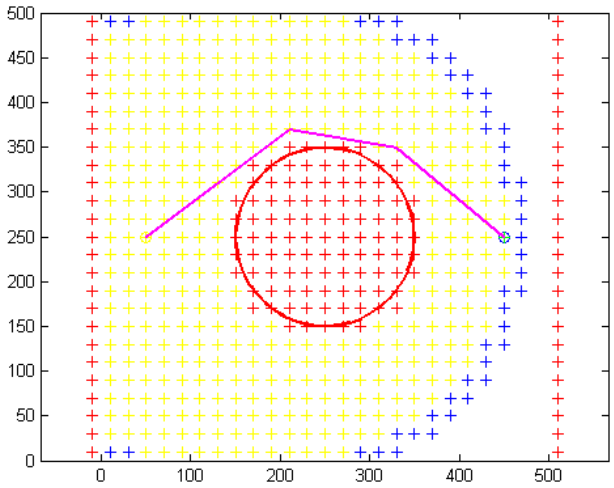


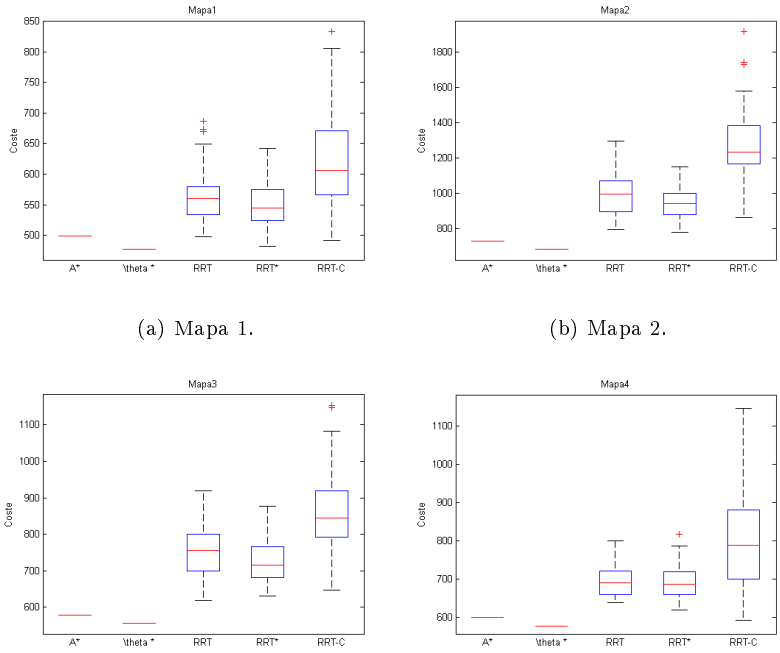
Figura 4.3: Ejemplo de funcionamiento de θ^* donde las celdas están sustituidas por los vértices de estas. Los nodos explorados son los amarillos y el camino final es morado. Se ve claramente la diferencia entre el aspecto del camino final de A* y θ^*.

del mismo orden que RRT, pero siguen siendo notablemente mayores, concretamente, con un 30 % de diferencia. La función *Reconectar* (*Rewire*) resulta tener un coste computacional bastante elevado, sobre todo, cuando el árbol esta más desarrollado, por lo que este algoritmo no se suele utilizar en aplicaciones que requieren rapidez.

El último algoritmo, RRT-Connect, ha resultado ser el más rápido de los planificadores probabilísticos, obteniendo tiempos parecidos a A*. Como ya se ha explicado anteriormente, esto se debe a que los nodos no tienen limitación de la distancia. Este algoritmo es interesante porque, siendo un programa aleatorio, la dispersión de los resultados es mucho menor que en los otros dos. Por ello, se puede deducir que este algoritmo sería el más apropiado, de los planificadores probabilísticos, para aplicaciones que requieren rapidez y su constancia temporal.

En conjunto, los tres algoritmos basados en RRT, aunque hayan obtenido resultados muy diferentes, siguen la misma tendencia general de comportamiento. Al ser totalmente aleatorios, los obstáculos «trampa» tienen menos efecto sobre sus resultados, sin embargo, la cantidad de obstáculos sí influye directamente en el tiempo de respuesta.

Otra propiedad muy importante que hay que tener en cuenta es el coste del camino encontrado. En la Figura 4.4 siguiente tabla se comparan los costes obtenidos de los algoritmos con los costes mínimos teóricos de cada mapa.



(a) Mapa 1.

(b) Mapa 2.

(c) Mapa 3.

(d) Mapa 4.

Figura 4.4: Costes obtenidos después de 50 pruebas.

Mapa	Mín. teórico	A*	θ^*	RRT	RRT*	RRT-Connect
1	451.12	499.41	477.86	559.2	555.74	616.99
2	677.92	729.12	684.76	998.18	948.77	1266.2
3	544.83	579.41	557.20	772.38	727.42	850.67
4	568.9	600.83	577.36	710.83	691.44	848.44

Como lo muestra la tabla, ningún algoritmo es totalmente óptimo en cuanto al coste. Los resultados más bajos han sido obtenidos por θ^*, cuyos costes son, como máximo, superan en 5 % los costes teóricos. Le sigue A*, con costes más altos debido a la limitación de poder expandirse en solo 8 direcciones diferentes.

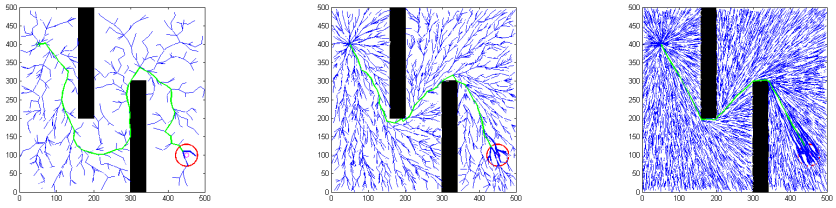
Los planeadores probabilísticos, como cabía de esperar, están lejos del camino óptimo, pero sigue habiendo una notable diferencia entre ellos. RRT* ha logrado los mejores costes entre ellos, siendo aproximadamente un 5 % más eficiente que RRT. Esta ventaja no es muy grande en comparación con la que ofrecen los planificadores deterministas, pero sigue siendo importante. Una propiedad importante de este algoritmo es que si se deja funcionando más tiempo, el coste obtenido se va acercando al óptimo, ya que el árbol se va reestructurando y acortando la ruta. En cambio, RRT-Connect ha obtenido los peores resultados, ya que sacrifica algunas de las características del RRT, que disminuyen el coste, por la rapidez. Además sus resultados son bastante inconsistentes, por lo que no se recomienda su uso en trabajos que requieren precisión.

Es interesante también comprobar cómo el algoritmo RRT* optimiza las trayectorias con el tiempo. En la Figura 4.5, se muestra como el algoritmo se va optimizando a medida que pasa el tiempo, hasta llegar casi al óptimo. Después del intervalo de 2 segundos, el árbol, teniendo pocos nodos, tiene un aspecto muy parecido a RRT, sin embargo, al pasar 20 segundos, va tomando una estructura característica y, en el último intervalo de 200 segundos, el árbol se extiende como si fuera un campo potencial. El camino encontrado, después de tanto tiempo, es tan solo 1 % mayor que el óptimo, e iría reduciendo la diferencia, hasta alcanzar el mínimo teórico en un tiempo infinito. En la siguiente tabla se exponen los costes obtenidos tras un tiempo de 100 segundos en comparación con θ^*, cuyas soluciones suelen estar más cerca del óptimo.

Mapa	Mín. teórico	θ^*	RRT*
1	451.12	477.86	455.35
2	677.92	684.76	689.77
3	544.83	557.20	558.80
4	568.9	577.36	594.83

Lo primero que destaca a la vista, es la diferencia que existe entre los dos algoritmos en el Mapa 1. Al ser un obstáculo circular, la división en celdas rectangulares pierde su eficacia. Por ello, el coste de θ^* es mucho mayor, al no ser capaz de adaptarse por completo a la geometría del problema. En los Mapas 2 y 3 los costes son muy parecidos y, en el 4, RRT* se ralentiza mucho por la gran cantidad de obstáculos, por lo que obtiene un resultado mucho peor.

Estos algoritmos están especializados en encontrar caminos cortos manteniendo tiempos razonablemente cortos. Después de analizar a los dos, se ha



(a) Tiempo: 2 segundos. Coste final: 1022 (b) Tiempo: 20 segundos. Coste final: 737 (c) Tiempo: 2 segundos. Coste final: 683

Figura 4.5: Muestra de caminos obtenidos por RRT* en el Mapa 2, durante intervalos diferentes del tiempo.

observado que RRT*, siendo un planificador probabilístico, está mejor adaptado a los entornos con obstáculos de geometría circular, irregular o cualquier otra, que resulte más difícil a la hora de descomponer en celdas. En cambio, θ^* obtiene ventaja en los entornos con muchos obstáculos.

4.2. Influencia del Sesgo/Bias

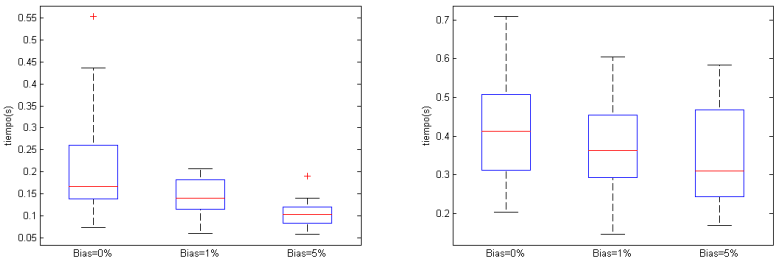
En este apartado se va a estudiar la influencia que tiene el sesgo sobre los resultados. Hay que tener en cuenta de que solo se aplica a RRT y RRT*, y el objetivo principal del Bias es mejorar el rendimiento de los programas.

Por ello, se ha decidido añadir dos niveles de sesgo de 1 % y 5 %, para ver cuál obtiene mejores resultados. Cabe recordar que un porcentaje demasiado alto puede conllevar a la aparición del mismo efecto que crean los mínimos locales, así pues, por lo general, suelen estar por debajo del 10 %. También es importante tener en cuenta la geometría del mapa, que puede facilitar o dificultar este proceso, dependiendo de la colocación de obstáculos.

En las Figuras 4.6 y 4.7 se ven las comparaciones de tiempos y costes para todos los mapas, para los dos algoritmos

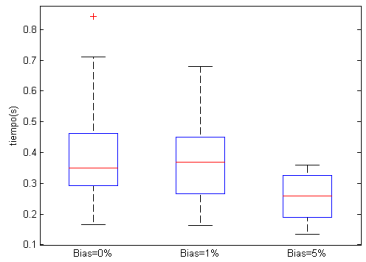
Se observa que el tiempo de resolución disminuye notablemente al aumentar el sesgo, incluso en los mapas donde es posible que se originen los «mínimos locales». La variación se nota, sobre todo, con el sesgo del 5 % , cuyo efecto puede reducir el tiempo total en unos 50 %. Esto es debido al carácter aleatorio del RRT, que explora todo el espacio disponible rápidamente pero algunas veces no consigue llegar a un punto determinado, hasta que el nuevo punto se elija justo en esa zona, que podría ser el objetivo. Teniendo una ayuda del sesgo, se elimina esa posibilidad, ya que si el árbol está cerca del final, unas iteraciones son suficientes para que lo encuentre.

También se ha incluido el coste del camino en el estudio del sesgo. Al observar el coste, se intuye una leve mejora ya que la tendencia introducida ayuda a dirigir el camino hacia la meta pero no hay una diferencia tan apreciable, por lo que se puede suponer que el coste es invariable.

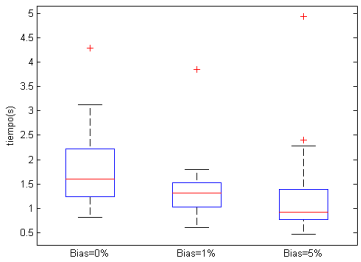


(a) Mapa 1

(b) Mapa 2

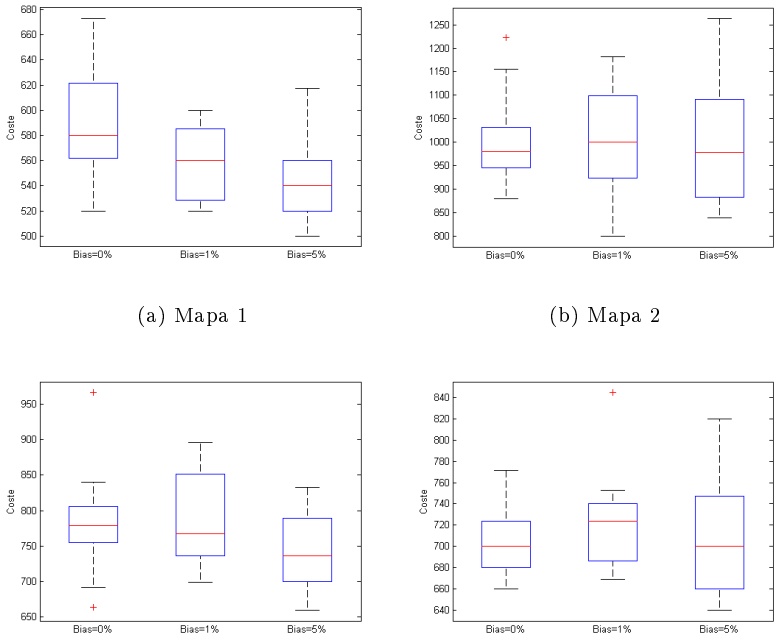


(c) Mapa 3



(d) Mapa 4

Figura 4.6: Influencia del Bias en el tiempo de resolución. RRT.



(a) Mapa 1

(b) Mapa 2

(c) Mapa 3

(d) Mapa 4

Figura 4.7: Influencia del Bias en el coste de resolución. RRT.

La influencia del Bias sobre los resultados del algoritmo RRT*es prácticamente idéntica a la del RRT, por lo que su estudio detallado no es de gran interés. Para el resto de los apartados se va a mantener el Bias de 5 %, el que ha obtenido los mejores tiempos.

4.3. Influencia de la dimensión de pasos D

Una vez estudiado el comportamiento general de los de los algoritmos, se va a ver la influencia del tamaño de pasos D en los resultados. Se han hecho pruebas del comportamiento en dos algoritmos: A* y RRT en diferentes mapas, pero en este apartado se muestran los resultados obtenidos del Mapa 2, que resume fielmente el comportamiento en el resto de Mapas. Los demás algoritmos, al ser basados en estos dos, obtendrían unos resultados muy parecidos. Al aumentar el parámetro D, la exploración se hace más rápida pero menos exhaustiva. En las Figuras 4.8 y 4.9 se observa la tendencia general de los tiempos y de los costes obtenidos:

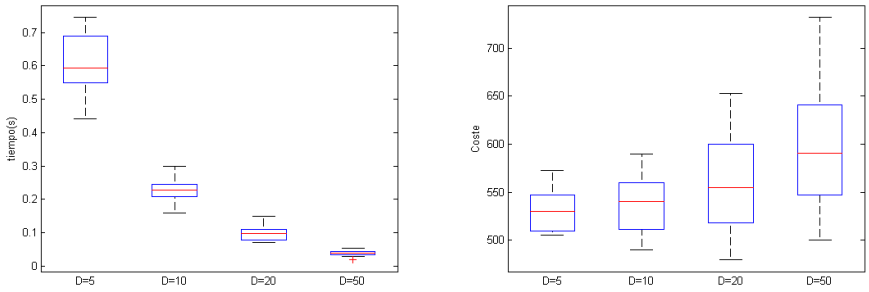
- El tiempo decrece de forma exponencial a medida que aumenta D. También disminuye la dispersión de los resultados, es decir, tienden a ser más uniformes. En el caso de A* la dispersión se debe a que el ordenador no tarda siempre el mismo tiempo en resolver un problema determinado, sin embargo la dispersión obtenida es despreciable.
- El coste, al contrario que el tiempo, crece, pero de forma más suave. Asimismo, en el caso de RRT, aumenta la dispersión de este, obteniendo un rango de valores más ancho. En el caso de A*, el coste depende más de la disposición de obstáculos que del tamaño de las celdas usadas debido a su limitación de 8 direcciones: diferentes caminos pueden tener el mismo coste.

4.4. RRT*2

En este apartado, se van a mostrar los resultados obtenidos de este algoritmo en comparación con RRT*, ya que son bastante semejantes. RRT* también sirve de una buena referencia por ser un algoritmo bastante conocido y bien estudiado en este proyecto. Las pruebas se han hecho con los mismos parámetros que para todos los demás algoritmos, con condición de parada de encontrar el camino.

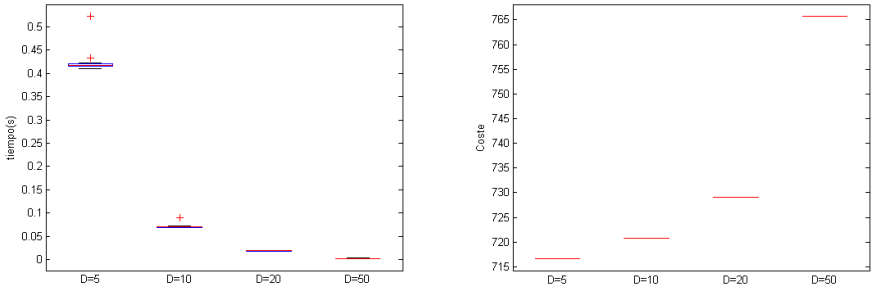
En las Figuras 4.10 y 4.11 se muestran los resultados para todos los mapas.

Como se observa en las imágenes, RRT*2 es algo más lento en la mayoría de situaciones, sin embargo en el Mapa 4, con muchos obstáculos y, teniendo un camino más directo, el tiempo es ligeramente inferior al algoritmo original. En los peores casos de los Mapas 2 y 3, la desigualdad llega a ser abismal: los tiempos de RRT*2 son casi el doble de RRT*. Sin embargo, la tendencia general de los valores de tiempo en el Mapa 1 es solamente ligeramente superior y, en el caso del Mapa 4, como ya se ha dicho, es menor. De aquí se puede concluir



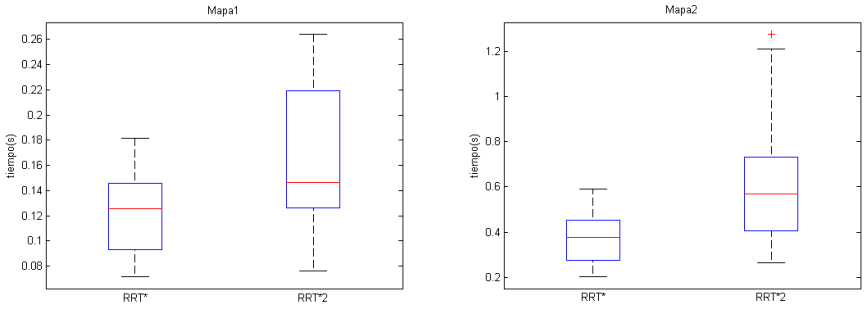
(a) Tiempos obtenidos en función de D. (b) Costes obtenidos en función de D.

Figura 4.8: Influencia de D sobre el comportamiento de RRT. Se han representado los resultados en el Mapa 2. En otros Mapas, los efectos son parecidos.



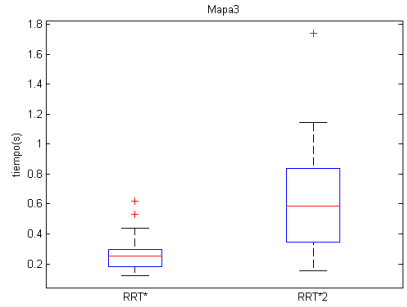
(a) Tiempos obtenidos en función de D. (b) Costes obtenidos en función de D.

Figura 4.9: Influencia de D sobre el comportamiento de A*. Se han representado los resultados en el Mapa 2. En otros Mapas, los efectos son parecidos.

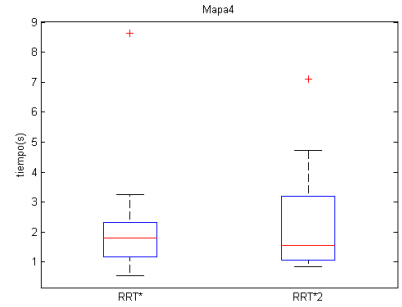


(a) Mapa 1.

(b) Mapa 2.

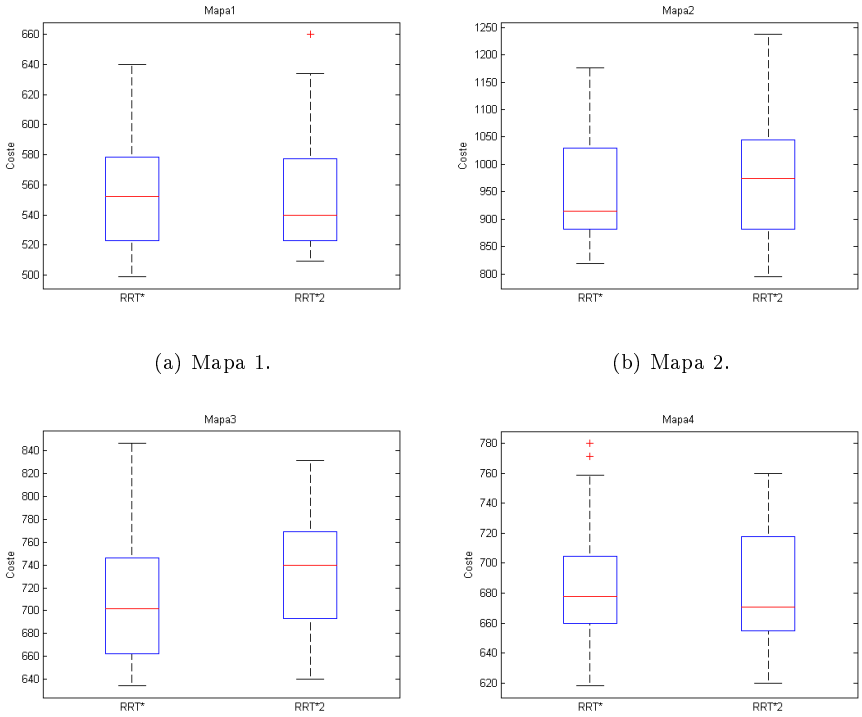


(c) Mapa 3.



(d) Mapa 4.

Figura 4.10: Comparación de tiempos entre RRT* y RRT*2.



(a) Mapa 1.

(b) Mapa 2.

(c) Mapa 3.

(d) Mapa 4.

Figura 4.11: Comparación de costes entre RRT* y RRT*2.

que RRT*2 Funciona más rápido en los entornos donde las rutas no se desvían demasiado y el camino, en general, es directo.

En cuanto a los costes, se puede decir que, igual que pasaba con el tiempo, resultan más favorecidos los mapas que no hacen desviar mucho la trayectoria. En este caso en los Mapas 1 y 4 se han obtenido mejores resultados y en los Mapas 2 y 3 han sido peores. La diferencia entre los resultados en el casos de mejora ha sido de un 3 % mientras que en los otros, supera los 5 %.

De los estudios posteriores a este, se ha averiguado que el parámetro D, también influye en los resultados. Parece ser, que RRT*2 funciona mejor con pasos más largos, disminuyendo las diferencias en los Mapas 2 y 3, y aumentando levemente la ventaja obtenida en el Mapa 4.

En conclusión, para este tipo de pruebas, se puede decir que, en general, el algoritmo original RRT* supera al RRT*2, pero RRT*2 le gana la ventaja cuando se trata de espacios abiertos, con muchos obstáculos de pequeñas dimensiones y que no haya desviaciones importantes.

Como ya se sabe, RRT* no destaca en la prueba de rapidez, pero si puede optimizar el camino, si se deja funcionando más tiempo. En la Figura 4.12 se muestran los resultados obtenidos en la prueba a tiempo donde el objetivo era llegar a los costes que superen los óptimos tan solo un 10 %. Como cabía de esperar, en los tres primeros entornos no tarda mucho, sin embargo el último Mapa crece significativamente. Mientras que los tiempos de RRT* están cerca del orden de 5 segundos, los tiempos de RRT*2, se disparan hasta 2 minutos, ya que, el algoritmo reconecta solo un nodo a la vez.

También se han hecho pruebas fijando el tiempo de trabajo. Como se ha visto en el apartado anterior, hay una inmensa diferencia entre los dos algoritmos si se dejan funcionando mucho tiempo. Por ello, en esta prueba se ha decidido que la duración máxima del funcionamiento sea de tan solo 2 segundos. Los resultados esta vez son los costes conseguidos, como se ve en la Figura 4.13. En esta prueba, RRT* también supera a RRT*2.

Con esto se puede concluir que RRT* es mejor que RRT*2 en la gran mayoría de los casos, pero, siendo ambos algoritmos aleatorios, su comportamiento sigue sin estar totalmente definido y, pueden darse casos contrarios.

4.5. RRT Smooth Path

El suavizado (o acortamiento) del camino trae muchas ventajas a la hora de usar los planificadores probabilísticos. Es una simple modificación del algoritmo que se aplica una vez que el camino haya sido encontrado. Este código modifica el camino final intentando saltar todos los nodos posibles para unir dos nodos alejados en linea recta como se ve en la Figura 4.14. De esta forma, se obtiene un camino considerablemente más corto aunque no se acerca demasiado al óptimo.

En la Figura 4.15 se observa la influencia de la modificación sobre los costes y los tiempos de ejecución. El coste promedio baja un 35 % mientras que el tiempo de ejecución apenas varía. Es un resultado sobresaliente para una modificación tan simple.

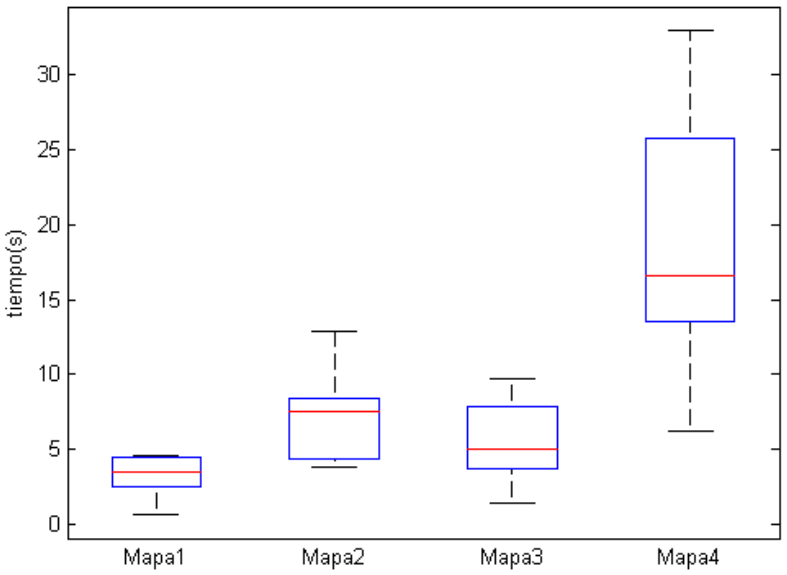
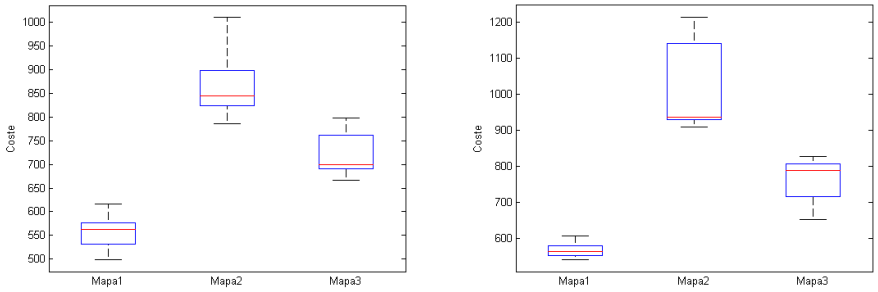
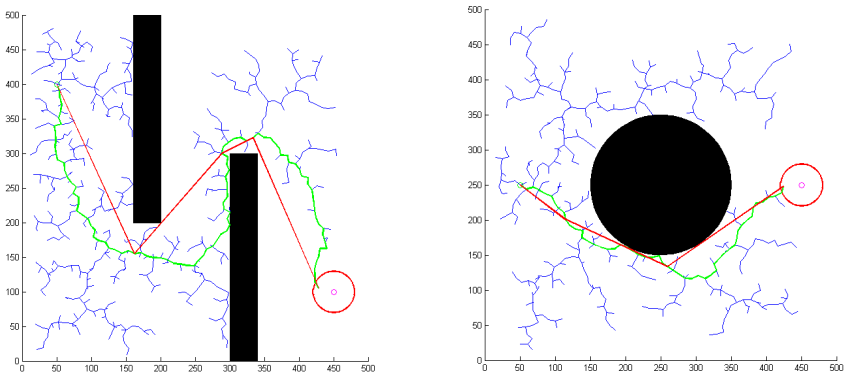


Figura 4.12: En esta figura se muestran los tiempos necesarios para conseguir los caminos que sean tan solo un 10 % más largos que los teóricos.



(a) RRT* (b) RRT*2

Figura 4.13: Comparación de costes para un tiempo de 2 segundos.



(a) (b)

Figura 4.14: Ejemplos del resultado de Smooth Path. La linea verde indica el camino original y la roja, el acortado.

Aplicar acortamiento de camino a RRT da resultados muy superiores al resto de los planificadores probabilísticos en cuanto al coste. En cuanto a tiempo, RRT no es el mejor, pero sigue siendo uno de los más rápidos y, teniendo en cuenta que la modificación apenas lo ralentiza, se puede aplicar en cualquier momento. La única desventaja que tiene este método es que esta diferencia de tiempo se vuelve muy notable si los obstáculos son numerosos como, por ejemplo, en el Mapa 4.

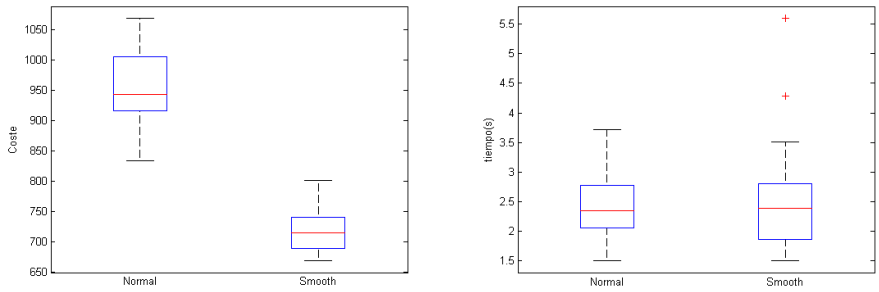
Para tener una mejor idea del ahorro de coste que supone esta modificación, a continuación se añade la siguiente tabla, donde se exponen los costes medios:

Mapa	Mínimo teórico	A*	θ^*	RRT	RRTSmooth
1	451.12	499.41	477.86	559.2	470.74
2	677.92	729.12	684.76	998.18	771.46
3	544.83	579.41	557.2	772.38	591.6
4	568.9	600.83	577.36	710.83	692.44

Hay que decir que, aunque los resultados medios son muy buenos, la modificación es algo inconsistente: su resultado depende, en la gran mayoría, del camino encontrado que es totalmente aleatorio. Por tanto, hay una cierta incertidumbre qui impide predecir con precisión cómo de corto va a ser el camino.

4.6. Considerando las dimensiones del UAV

Como otra modificación adicional, se han programado las dimensiones del UAV. La función *Hull* tiene en cuenta su tamaño a la hora de detectar los



(a) Costes sin el acortamiento y con el acortamiento.

(b) Tiempos de ejecución del algoritmo normal y con acortamiento.

Figura 4.15: Comparación de coste y de tiempo empleado sin y con Smooth Path.

obstáculos. Este programa sacrifica la rapidez de RRT y sus modificaciones, para calcular las rutas más seguras.

En este apartado se va a analizar su impacto en el coste computacional, es decir, el tiempo. En la Figura 4.16, se observan los tiempos conseguidos en los mapas de dimensiones reducidas. A primera vista, puede parecer que son demasiado altos, teniendo en cuenta que RRT normal no sale del rango de $0{,}25s$ pero estos resultados son muy lógicos teniendo en cuenta el estructura del mapa.

La Figura 4.17 muestra la razón principal de los tiempos tan altos. Cabe recordar que el UAV es $90cm$ de ancho, mientras que la distancia entre los obstáculos en el Mapa 2 es tan solo de $1m$, lo que deja solo $10cm$ de holgura para que este pueda pasar. Teniendo camino tan estrecho y tan largo, el algoritmo aleatorio RRT consigue pasar en un tiempo medio del orden de un segundo, que es un tiempo bastante razonable para muchas aplicaciones, pero el camino puede no ser el adecuado para las que necesitan mucha precisión. Para ello se ha implementado la función Hull en el algoritmo RRT*, obteniendo un camino más apropiado para los trabajos de precisión, como se ve en la Figura 4.18. También se han hallado los costes de las rutas conseguidas para otros mapas representados en la Figura 4.19:, con un tiempo de ejecución de $50s$:

- Mapa 1: 478,9.
- Mapa 2: 842.3.
- Mapa 3: 683.4.

Al dejar ejecutar el programa más tiempo, estos costes pueden obtener la ruta óptima con las dimensiones del UAV consideradas. Este método siempre encontrará un camino óptimo mientras que, ensanchando los obstáculos, esta posibilidad puede eliminarse.

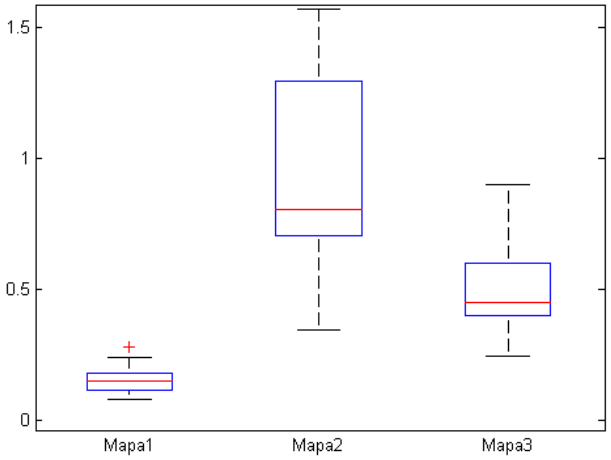


Figura 4.16: Tiempos obtenidos para RRT teniendo en cuenta las dimensiones.

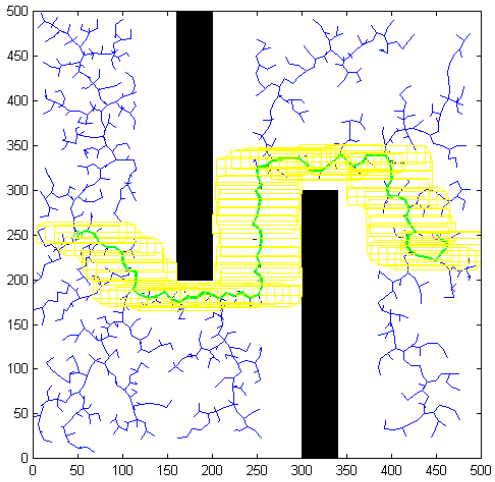


Figura 4.17: Mapa 2. RRT ejecutado teniendo en cuenta las dimensiones del UAV.

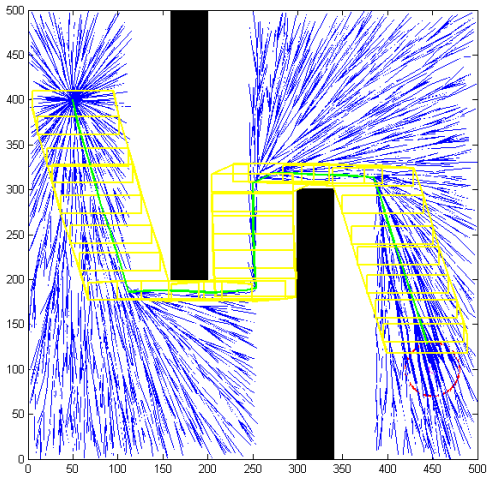
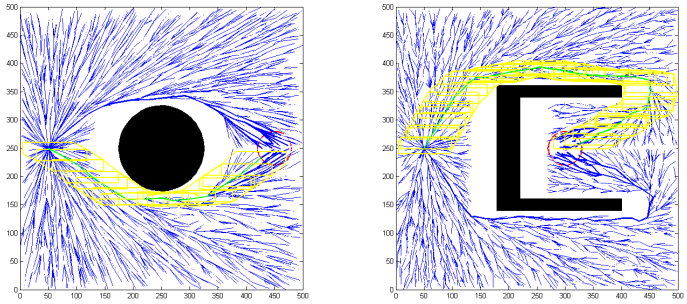


Figura 4.18: Mapa 2. RRT* ejecutado teniendo en cuenta las dimensiones del UAV.



(a) Mapa 1. (b) Mapa 3.

Figura 4.19: Representación gráfica de las rutas de RRT*, con las dimensiones del UAV implementadas. Tiempo empleado: $50s$.

Capítulo 5

Conclusiones y líneas futuras

5.1. Conlusiones

En este documento se han expuesto los algoritmos de planificación de rutas implementados en MATLAB y se han comparado sus características más principales. En concreto, se ha centrado en el estudio de 5 algoritmos más utilizados en el campo de la robótica, siendo dos de ellos planificadores deterministas y otros tres, planificadores probabilísticos. Estos algoritmos son: A*, θ^*, RRT, RRT* y RRT-Connect. Cada uno de ellos tiene propiedades únicas, adecuadas para diferentes aplicaciones. Comparándolos entre ellos y fijándose en sus características más cruciales: el coste de camino y el tiempo empleado; se ha conseguido estudiar sus comportamientos en diferentes entornos de distinta complejidad.

A continuación se resumen los resultados obtenidos de todas los algoritmos:

- **A*:** este planificador implementado ha resultado ser uno de los más rápidos, consiguiendo resolver entornos complejos con gran cantidad obstáculos en centésimas de segundo. Este algoritmo se aplica sobre un mapa preprocesado, cuyos elementos han sido descompuestos en celdas. En este trabajo se ha supuesto que el entorno ya está descompuesto para una aplicación inmediata del algoritmo, aunque este proceso también lleva su tiempo. La otra desventaja que tiene es que solo se puede extender en 8 direcciones. Al ser determinista y teniendo una heurística fija en el punto final, suele caer en los mencionados «mínimos locales» donde se mantiene un tiempo sin avanzar, investigando el área alrededor. Teniendo en cuenta todo esto, se puede concluir que se le puede dar mejor uso en trabajos con numerosos obstáculos simples como el Mapa 4, que simula una ciudad.
- **θ^*:** esta modificación de A* se comporta de forma parecida en diferentes situaciones pero, sacrifica un tiempo extra para acortar los caminos. Los caminos obtenidos no están limitados a 8 direcciones y son mas realistas, pero siguen sin ser los óptimos. El algoritmo puede tardar hasta 8 veces más en encontrar la solución aunque, siendo los tiempos de otro algoritmo muy bajos, se mantiene en el orden de unas décimas de segundo.

- RRT: este planificador probabilístico es más lento que A* en la mayoría de veces. Además el camino encontrado suele ser mucho más largo. Sin embargo, obtiene una gran ventaja en espacios con obstáculos «trampa» y su aleatoriedad le permite encontrar caminos en situaciones muy difíciles. Además de esto, no necesita un preprocesado o descomposición en celdas para su funcionamiento, lo que puede llegar a tener una duración de varios segundos dependiendo de la complejidad del mapa. En general, este algoritmo consigue mejores resultados en entornos con pocos obstáculos, ya que la comprobación de colisiones llega a ser bastante costosa en cuanto a la computación. En cuanto a las mejoras implementadas se tiene que:
  - Como ya se ha dicho, los costes de los caminos obtenidos están muy lejos de los óptimos. Para ello se ha creado RRT Smooth Path que los reduce significativamente en poco tiempo, llegando a obtener los costes similares a A*.
  - La consideración de las dimensiones del UAV, empeora los resultados del algoritmo, pero puede tener una inmensa utilidad práctica en muchas aplicaciones.
  - Ambas mejoras pueden ser implementadas en cualquier programa que deriva del RRT simple.
- RRT*: este algoritmo es una modificación del anterior que reestructura el árbol creado, minimizando los costes. La característica principal de este planificador es que, una vez encontrado el camino, si el programa se sigue ejecutando, el camino se sigue optimizando hasta llegar casi al óptimo. La modificación RRT*2 implementada no obtiene ninguna ventaja clara sobre este algoritmo.
- RRT-Connect: esta versión del RRT se libera de las limitaciones impuestas para su predecesor. En concreto, se elimina el tamaño máximo de los pasos, lo que hace que el árbol se expanda mucho más rápido. Aparte de esto, el crecimiento desde los estados inicial y final, mejora considerablemente las posibilidades de que estos se encuentren, haciéndolo más rápido. La principal desventaja es que la rápida expansión conlleva a cambios bruscos de dirección, por lo que se obtienen los peores costes de todos los algoritmos.

Por último, la implementación de las dimensiones del UAV en los algoritmos de búsqueda, añade más profundidad al problema, pero permite obtener resultados más realistas. Este método permite introducir geometría compleja del UAV y de los obstáculos sin eliminar posibles resultados y no incluye preprocesamiento del mapa. Además de esto, aplicando las hipótesis correctas, se puede usar para hallar soluciones aproximadas a problemas complejos que requieren implementación de la dinámica y de la cinemática del problema.

5.2. Líneas futuras

Los algoritmos de búsqueda de caminos implementados son de carácter muy general, para su posterior aplicación se proponen varias mejoras:

- Integración de la dinámica y de la cinemática del problema ya que, en este trabajo, se ha visto su aplicación puramente geométrica.
- Estudio de sus aplicaciones en 2.5D y 3D. En este trabajo se ha centrado en movimiento plano, el espacio con más dimensiones puede abrir más posibilidades y crea nuevos problemas que serían interesantes de estudiar.
- Estudio e implementación de geometría mas compleja: considerando las dimensiones más detalladas y posibles obstáculos reales.
- Implementación de los algoritmos en el lenguaje C, lo que mejoraría los tiempos de ejecución.

Bibliografía

[1] Randal W. Beard and Timothy W. McLain. *Small Unmanned Aircraft : Theory and Practice*. Princeton University Press, 2012.

[2] Norman Biggs, E Keith Lloyd, and Robin J Wilson. *Graph Theory, 1736-1936*. Oxford University Press, 1976.

[3] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[4] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *CoRR*, abs/1401.3843, 2014.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[6] Daniel Harabor and Alban Grastien. An optimal any-angle pathfinding algorithm. *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[8] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[9] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[10] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986.

[11] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[12] Andrew M Ladd and Lydia E Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems*, pages 233–240, 2005.

[13] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[14] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[15] Vladimir J Lumelsky and Alexander A Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1):403–430, 1987.

[16] Ioan A Şucan and Lydia E Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pages 449–464. Springer, 2009.

[17] Kimon P Valavanis. *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*, volume 33. Springer Science & Business Media, 2008.

[18] Paul Keng-Chieh Wang. *Visibility-based Optimal Path and Motion Planning*. Springer International Publishing, 2015.