Proyecto Fin de Máster
Máster Universitario en Ingeniería Industrial

Planificación de caminos basada en modelo combinando algoritmos de búsqueda en grafo, derivados de RRT y RRT*.

Autor: Ángel Manuel Montes Romero
Tutores: José Antonio Cobano Suárez
Jesús Capitán Fernández

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

Sevilla, 2017

Proyecto Fin de Máster
Máster Universitario en Ingeniería Industrial

Planificación de caminos basada en modelo combinando algoritmos de búsqueda en grafo, derivados de RRT y RRT*.

Autor:
Ángel Manuel Montes Romero

Tutores:
Dr. D. José Antonio Cobano Suárez
Dr. D. Jesús Capitán Fernández

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

Proyecto Fin de Máster: Planificación de caminos basada en modelo combinando algoritmos de búsqueda en grafo, derivados de RRT y RRT*.

Autor: Ángel Manuel Montes Romero

Tutores: José Antonio Cobano Suárez

Jesús Capitán Fernández

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

A mi familia, que siempre han estado ahí en todo momento

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN A LA ROBÓTICA Y PLANIFICACIÓN DE CAMINOS

Como la propia definición de la RAE nos indica, un *robot* es una máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas solo a las personas. Cada vez nos acostumbramos más a verlos y a convivir con ellos, y es que estos robots han entrado en nuestras vidas para quedarse gracias a su capacidad de realizar funciones ya indispensables para nosotros.

Algunas de sus aplicaciones son:

- *Labores peligrosas para los seres humanos:* inspección en funcionamiento de chimeneas industriales, volcanes, actuación en zonas nucleares, fumigado químico, etc.
- *Labores en condiciones desfavorables:* inspección y limpieza de basureros, clarificadores, alcantarillado, etc.
- *Labores repetitivas, extenuantes o poco rentables:* por ejemplo labores del campo.
- *Labores imposibles para el ser humano:* estudio de fosas marinas, manipulación de grandes pesos, operaciones aéreas, etc.

La robótica está experimentando un crecimiento exponencial a medida que entran en juego más profesionales con capacitación técnica para desarrollarla y que la tecnología del silicio avanza. Este avance de tecnología ha propiciado la caída de precios de los sensores y la mejora de la electrónica de potencia, de señal y del hardware de las computadoras. La presencia creciente de profesionales de la robótica es la que posibilita el continuo crecimiento actual de esta técnica puramente multidisciplinar, y es que modelar el comportamiento humano no es nada trivial. Mucho menos llevarlo a la práctica. Entran en juego leyes físicas de la mecánica clásica, sistemas electrónicos embebidos, sistemas de potencia, técnicas de control, actuadores eléctricos, etc.

Muchas de las investigaciones actuales tratan de mejorar la autonomía de los robots, definiendo *autonomía* en robótica como la medida en que el robot es capaz de tomar decisiones e interactuar con el entorno por sí mismo. No confundir con la autonomía de las baterías. Actualmente se pueden encontrar robots con diversos niveles de autonomía, desde esclavos teleoperados hasta auténticos robots independientes (dentro de la función con la que fue programada).

El objeto de este Proyecto es el de estudiar un campo de la robótica cuyo fin es el de aumentar la autonomía de los robots móviles (aunque también es aplicable a manipuladores), la planificación de caminos. En la *robótica móvil* los robots se desplazan en su entorno, en su inteligencia artificial hay que incluir el que sean capaces de decidir qué camino tomar para llegar al destino, identificando obstáculos, zonas de paso y siendo conscientes con la mayor precisión posible de su propia ubicación respecto un sistema de referencia dado.

Como ya se ha comentado, este Proyecto tratará de la planificación de caminos de robots móviles, y aunque principalmente se hace uso de métodos probabilísticos, también se combinan con métodos tradicionales de búsqueda en grafo, aprovechando sinergias entre ambos métodos. Los métodos probabilísticos aquí usados son versiones modernas de los RRT (Rapidly-exploring Random Tree) [1]. Estos métodos probabilísticos están muy activos en investigación, tanto que algunos de los algoritmos aquí presentados tienen tan solo 2 años de antigüedad a día de hoy.

En este primer capítulo se hará una introducción a la robótica y planificación de caminos, pasando en los dos siguientes capítulos a hacer una breve introducción respectivamente de la planificación tradicional basada en modelo geométrico y de la planificación probabilística basada en modelo. En el cuarto capítulo se presenta la forma en la que se han programado los algoritmos en MATLAB [2]. A partir del quinto capítulo se explican por separado los distintos algoritmos de planificación evaluados en este Proyecto, ordenados según se apliquen en la fase de búsqueda de camino inicial no óptimo en estático, en la optimización en estático o en la optimización en tiempo real.

Cabe señalar que este Proyecto Fin de Máster surge como ampliación (extensa) de mi anterior Proyecto Fin de Grado, del cual se aprovechó el bagaje para partir de un cierto nivel en la materia y conseguir un mayor alcance. Esto no significa que sea necesaria la lectura del Proyecto anterior para la comprensión de este texto, ya que está planteado de la forma más autoexplicativa y clara posible.

1.1 La paradoja de Moravec.

El que un robot sea capaz de decidir por sí mismo una trayectoria en un entorno con infinitas posibles trayectorias válidas es un problema particular de la inteligencia artificial. Por este motivo, y antes de entrar en materia técnica, me voy a tomar la libertad de hablar de la paradoja de Moravec.

Y es que tras décadas de desarrollo en inteligencia artificial, se ha llegado a una conclusión paradójica: el pensamiento razonado humano (considerado generalmente como inteligencia) es recreable fácilmente, mientras que las habilidades sensoriales y motoras requieren grandes esfuerzos computacionales. [3]

En palabras del propio Moravec: "*Es fácil conseguir que las computadoras muestren capacidades similares a las de un humano adulto en tests de inteligencia, pero difícil o imposible lograr que posean las habilidades perceptivas y motrices de un bebé de un año*".

Esto significa que, hoy día, un simple teléfono móvil (con el software adecuado) es mejor que cualquier humano resolviendo trabajos de mayor necesidad de pensamiento estructurado, como pueden ser resolver cálculos matemáticos o jugar al ajedrez. Sin embargo tareas más triviales y cotidianas como reconocer objetos por visión o desplazarte a una estantería a coger algo son tareas mucho más desafiantes a la hora de replicar.

La explicación de esto, curiosamente, la tiene la evolución. Cuando el primer homo estaba empezando a desarrollar su inteligencia racional hace 2,5 millones de años, el reino Animalia llevaba ya 530 millones de años de evolución desarrollando habilidades para sobrevivir, interaccionando con un entorno cambiante e incierto. Estamos hablando de que el pensamiento perceptivo y motriz tiene una ventaja evolutiva sobre el pensamiento racional de dos órdenes de magnitud. Esto explica por qué la mayor parte del cerebro está dedicada al sentido y al movimiento, mientras que la parte dedicada a lógica o aritmética está menos desarrollada y es más fácil de emular. [3]

En este escenario entran en juego los profesionales de la robótica, visión y control, que mediante su continua y encomiable labor de investigación en los diversos campos hacen posible que, poco a poco, se vaya derribando dicha brecha evolutiva, haciendo posible y llevando a la práctica lo "difícil o imposible".

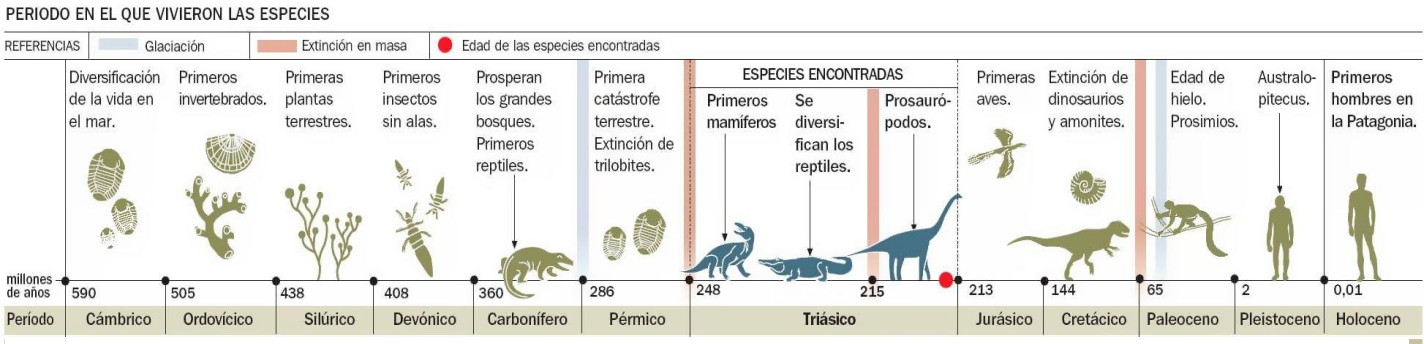

Figura 1. Explicación evolutiva de la paradoja de Moravec: hablando en términos de optimización genética, el pensamiento perceptivo y motriz ha tenido muchas más generaciones de optimización que el pensamiento racional. [51]

1.2 Fundamentos y conceptos principales.

1.2.1 Tipos de robots y definiciones básicas: planificación, navegación y entorno.

Tradicionalmente los robots se han clasificado como manipuladores o móviles. Un *robot manipulador* normalmente tiene como finalidad mover piezas, materiales o realizar operaciones con un actuador final sobre algún producto, permaneciendo fijo en su sitio. También se llaman a menudo *robots industriales* por su uso intensivo en la industria, normalmente en configuración de brazo articulado.

Figura 2. Ejemplo de robot manipulador: robot manipulador de pallets vacíos. [4]

Los *robots móviles* surgen ante la necesidad de ampliar el rango de alcance de los mismos, siendo capaces de moverse por su entorno de forma automática. Así, la planificación y navegación son las características principales de un robot móvil, siendo al mismo tiempo las más desafiantes. A continuación se definirán dichos conceptos esenciales en este Proyecto.

Planificación es encontrar el camino o trayectoria que une un punto inicial (la posición inicial del robot) con uno final (el destino) sin chocarse, mientras que *navegación* hace referencia a seguir dicha trayectoria sin salirse de la misma y sin colisionar con el entorno, el cual puede ser cambiante. A su vez, el *entorno* es el espacio físico en el que se halla el robot y los lugares que debe alcanzar.

Por ejemplo en un vehículo autónomo destinado al transporte de mercancías, su entorno será la nave industrial en la que realiza sus tareas. Esto también es aplicable para los robots articulados, en los que en el cálculo de trayectorias se incluirán las referidas a todas y cada una de las partes móviles que contengan. Por supuesto aunque los robots se clasifiquen en manipuladores y móviles, un robot puede ser manipulador y móvil al mismo tiempo.

Figura 3. Ejemplo de robot móvil: robot de limpieza doméstica Roomba. [5]

Los robots pueden clasificarse de otras muchas formas debido a la gran variedad de configuraciones que pueden presentar y su enorme variedad de aplicaciones. Otra de ellas muy importante está relacionada con su movilidad, denominándose como:

- *Holonómicos*: robots que tienen tantos grados de libertad efectivos como controlables.
- *No holonómicos:* robots que tienen menos grados de libertad controlables que efectivos.
- *Redundantes:* robots que tiene más grados de libertad controlables que efectivos.

Simplificando, podemos decir que un robot es holonómico si es capaz de modificar su dirección instantáneamente (masa nula) y sin necesidad de rotar o maniobrar previamente. Un vehículo con un sistema de dirección como el de un coche en configuración Ackerman, por ejemplo, no lo es, porque para poder desplazarse en el sentido lateral tiene que realizar varias maniobras previas. Del mismo modo, un robot con 2 ruedas (tanto en configuración motocicleta como diferencial) es no-holonómico ya que no puede moverse hacia la izquierda o la derecha de forma directa. [6]

En general, a mayor diferencia entre grados de libertad controlables y grados totales, más difícil será controlar el robot. Tanto si la diferencia es positiva (robot redundante) como negativa (no holonómico).

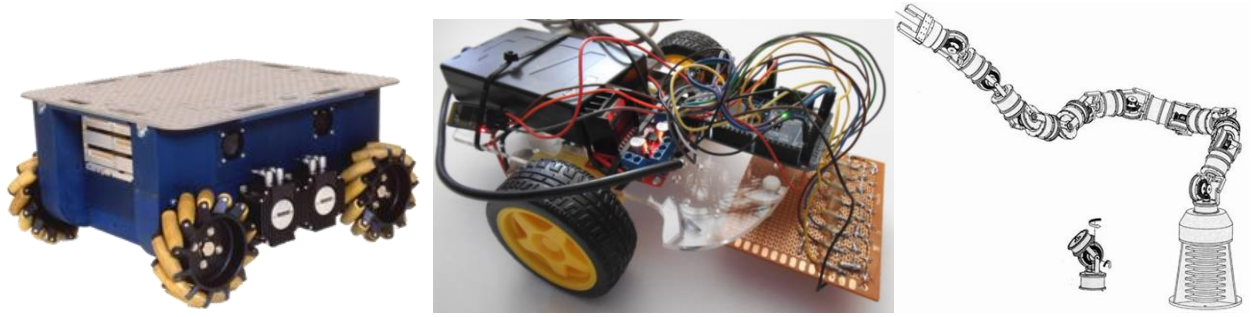

Figura 4. De izquierda a derecha: robot holonómico URANUS (gracias a sus ruedas omnidireccionales) [7], robot no holonómico (en configuración diferencial) [8], manipulador redundante de 20 DoFs (Degrees of Freedom) [9].

1.2.2 Métodos de planificación de caminos.

La planificación de caminos se puede abordar mediante varios métodos distintos. Los principales en robótica y los que se describirán en este Proyecto son los siguientes:

1.2.2.1 Métodos tradicionales basados en modelo geométrico.

En estos métodos primero se modela el espacio de trabajo como una red o grafo de posibles caminos, uniendo los puntos inicial y final a dicho grafo, para después en una segunda fase aplicar un algoritmo de búsqueda en grafo que escoja aquel camino que optimice el coste (minimizando distancia o tiempo) de entre todos los posibles caminos que unen el punto (nodo) inicial con el final dentro del grafo.

Estos métodos son válidos para problemas de planificación sencillos en los que no se pide mucha complejidad al entorno o los robots no tienen muchos grados de libertad. Sin embargo, en robots con muchos grados de libertad y en entornos muy complejos a los que se le pida un nivel de precisión razonable, el número de nodos y arcos del modelado del entorno se dispara exponencialmente (problema NP-completo [10]), haciendo inviable computacionalmente este tipo de algoritmos en estos entornos. Además, la mayoría de estos algoritmos son off-line que no sirven en tiempo real. No obstante, hay versiones que evitan probar de forma exhaustiva todos los arcos del grafo introduciendo una heurística (A*), o incluso también los hay con modificaciones en tiempo real (D*), aunque son excepciones que igualmente no resultan adecuadas en problemas complejos.

Pero no todo son desventajas con estos métodos. Usando un grafo inicial con pocos nodos posicionados

estratégicamente, el tiempo de búsqueda del primer camino viable se puede reducir en varios órdenes de magnitud respecto esa misma búsqueda usando un método probabilístico 100%. En otras palabras, existen sinergias entre los algoritmos de búsqueda en grafo tradicionales y los probabilísticos que merece la pena explotar. Es por eso que uno de ellos se ha usado en este Proyecto, el algoritmo Dijkstra. Se explicará todo esto en detalle en capítulos posteriores.

1.2.2.2 Métodos reactivos.

Pueden generar directamente órdenes al controlador de movimientos a partir de las medidas obtenidas con sensores y/o el modelo del entorno, usando típicamente sensores de rango. Como indica su nombre, el objetivo es reaccionar, puede ser sin planificar, en aquellos casos donde se debe resolver un conflicto en muy poco tiempo. Los algoritmos más conocidos para implementar esta característica son los basados en campos potenciales y fuerzas virtuales.

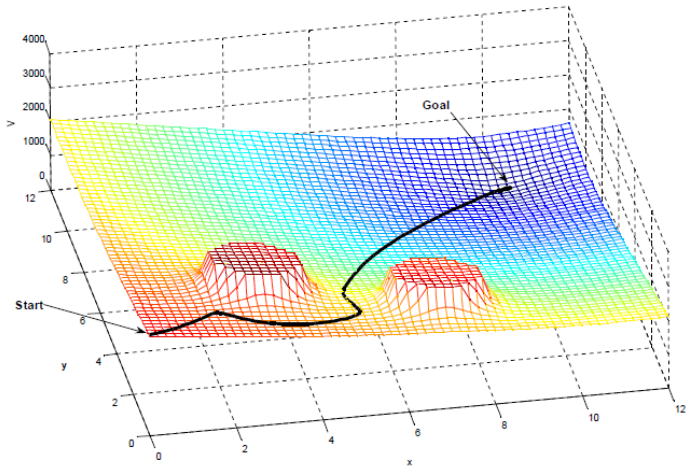

Figura 5. Ejemplo de planificación de camino basado en campos potenciales, el destino atrae al robot mientras que los obstáculos lo repelen. [11]

Estos algoritmos presentan dos grandes inconvenientes:

- La mayoría de ellos son presa fácil de los *mínimos locales*.
- Necesitan computar todos los límites de los obstáculos. Para entornos con pocos obstáculos geométricos simples (rectángulos, circunferencias, etc.) esto no es problema, pero en entornos complejos con obstáculos de geometría arbitraria resultan ineficientes.

En este Proyecto se ha implementado un novedoso algoritmo que, siendo probabilístico, también consigue tener la propiedad de ser de tiempo real. Este algoritmo es el llamado RT-RRT* (Real Time RRT*) [12] y solventa los inconvenientes antes citados gracias a las propiedades inherentes de los árboles de exploración rápida aleatorios. Se explicará e implementará en detalle en capítulos posteriores.

1.2.2.3 Métodos probabilísticos o basados en muestreo.

Estos algoritmos son considerados actualmente como el estado del arte de la planificación de caminos en problemas muy complejos, empleándose en entornos de decenas o cientos de dimensiones tales como robots manipuladores, doblado de moléculas biológicas, movimiento de personajes digitales, piernas robóticas, etc. En este Proyecto serán los que más aparezcan, concretamente los algoritmos derivados del RRT [13]. Su diferencia radica en que el grafo de consulta se construye no mediante cálculos geométricos sobre los obstáculos, sino muestreando puntos al azar en el espacio de trabajo e introduciéndolos como nodos en el grafo.

Su principal ventaja es que son mucho menos sensibles a la complejidad del entorno ya que no necesitan hacer

cálculos con los obstáculos. Hablando ahora particularmente de los RRT, el que no tenga que operar con los obstáculos se traduce en una rápida y eficiente expansión del árbol de búsqueda, buen comportamiento ante mínimos locales, y en completitud probabilística (la probabilidad de encontrar una solución subóptima tiende a 1 a medida que el árbol se expande). Los puntos negativos de los métodos probabilísticos es su incapacidad de reconocer la no existencia de camino libre de obstáculos, aunque como ya se ha comentado, normalmente si no encuentra camino en mucho tiempo es que no existe.

Los RRT a secas tienen naturaleza subóptima (cuando encuentran un camino cualquiera al azar dejan de iterar y devuelven dicho camino) y al no tratarse de un método reactivo no pueden usarse en tiempo real en un entorno cambiante. Existen variaciones dignas de mención, como el RRT-Connect [14] que se comporta mucho mejor ante mínimos locales expandiendo 2 árboles de búsqueda e intentando unirlos con heurística voraz la mitad del tiempo, o el RRT multi-query, que intenta unir un punto inicial con más de un punto final (en vez de solo uno).

Los RRT* [15] sin embargo son asintóticamente óptimos, esto significa que devuelven la trayectoria óptima cuando el tiempo de búsqueda tiende a infinito. Para acelerar el proceso se aplican heurísticas que reducen el tiempo de convergencia en la solución óptima, estos algoritmos heurísticos son el RRT*-Smart [16] e Informed RRT* [17]. Además para saturar el crecimiento excesivo del tamaño del árbol (que podría acabar agotando la memoria de cálculo) se emplea el algoritmo RRT*FN [18]. Por último, como ya se ha comentado anteriormente, existe el RT-RRT* [12] que permite aplicar estos algoritmos en tiempo real.

Los métodos probabilísticos se explicarán más en detalle en el capítulo tercero, explicándose cada algoritmo de forma más minuciosa en capítulos posteriores.

1.2.3 Control posicional de los robots conforme a su relación con el entorno.

Se distinguen:

a) Sin percepción del entorno: configuración de control en bucle abierto típica en robots manipuladores. Para que sea viable es estrictamente necesario:

- Elevada precisión y repetibilidad. Por ejemplo en el caso de un manipulador soldador en una línea de fabricación "transfer" de coches, ejecutando siempre el mismo movimiento de soldadura por puntos en bucle abierto, es vital que los encoders y servomotores de cada articulación sean de la máxima precisión para garantizar que no se produzcan errores de integración y que la repetibilidad sea máxima. La *repetibilidad* es una medida de capacidad del robot para ubicarse en un punto definido en el volumen de trabajo tras realizar el mismo movimiento muchas veces. Cada vez que el robot intenta regresar al punto programado, este regresará a una posición ligeramente diferente.
- Ubicación de objetos perfectamente conocida. En el ejemplo de antes del robot soldador, por mucho que haga siempre perfectos los movimientos, si el coche no está posicionado en su sitio la soldadura saldrá movida.
- Calibración automática y periódica de cada una de todos los robots y sistemas involucrados.

b) *Con percepción del entorno:* en la robótica móvil, normalmente no asociada a tareas repetitivas, es fundamental la interacción con el entorno. Siempre que no puedas garantizar que el entorno sea 100% conocido y estático vas a necesitar controlar si se producen modificaciones en el entorno, ya que estas podrían producir colisiones, siendo necesario detectarlas y evitarlas. De aquí la importancia de los sensores y de la detección de colisiones.

1.2.4 Sensores.

Los s*ensores* son los dispositivos que reciben la información del entorno y la traducen en una magnitud comprensible para el sistema de control (eléctrica normalmente). No solo son elementos fundamentales en la obtención de la autonomía buscada, sino que la información que suministren tendrá gran influencia en los métodos que se empleen para la evitación de colisiones en la planificación y navegación, así como en su estructura de datos.

Por ejemplo, mientras que para detectar la presencia de obstáculos en direcciones de movimiento en tiempo real, se necesita relativamente poca información, para la construcción de modelos geométricos online durante el movimiento en tres dimensiones, se necesita un volumen importante y una gran cantidad de procesamiento. Por ello las estructuras de datos para representar la información geométrica del entorno tendrán notable influencia en el rendimiento de los algoritmos de planificación.

Es frecuente que en los sistemas robóticos móviles, el controlador a bordo carezca del modelo completo del entorno (o de capacidad suficiente de procesamiento para unificar los datos de los sensores con el modelo y calcular la trayectoria), y en consecuencia, necesite un sistema *telerrobótico* que realice las tareas de planificación y envíe al sistema robótico móvil la trayectoria deseada.

Los sensores pueden ser*:*

- *Pasivos*: detectan el entorno absorbiendo energía directa del entorno. Por ejemplo, cálculo de profundidad mediante dos cámaras en estéreo.
- *Activos*: para su detección emiten energía del algún tipo al entorno. Por ejemplo, cálculo de distancia por emisión de láser o ultrasonidos.

Algunos tipos de sensores utilizados en robótica:

- *Infrarrojos:* mediante diodos LED emisores de luz infrarroja se proyecta un haz de luz infrarroja sobre una superficie que refleja dicho haz sobre unos fototransistores. La lectura de los fototransistores da información sobre la distancia (o ausencia) de la superficie reflectora.
- *Ultrasonidos*: emplean el principio del tiempo de vuelo o de eco, la distancia viene dada por el tiempo transcurrido entre la emisión de un pulso y su recepción después de ser reflejado por una superficie. Tienen la limitación importante del alcance de la transmisión del sonido por el aire, por ello son utilizados en vehículos robóticos en interiores y en vehículos submarinos debido a las mejores propiedades de transmisión de los ultrasonidos en el agua.
- *Láser*: funcionan gracias a una electrónica muy rápida que cronometra el tiempo de vuelo y mide el desfase entre la onda emitida y reflejada. Son de alcance mucho mayor que los ultrasonidos, con limitaciones en la detección de obstáculos próximos debido a la gran velocidad de la luz. También usan el principio de la triangulación (de alcance reducido) en efectores finales de manipuladores.
- *De contacto*: para detección de colisiones, en reconocimiento de objetos, etc.
- *De esfuerzos*: tienen la ventaja de permitir el control de esfuerzos en movimientos que implican contacto, como coger objetos sin aplastarlos
- *De procesamiento de imágenes*: mediante una cámara, cámaras en estéreo u otra configuración.

Decir que muchas veces se combinan distintos tipos de sensores que aparentemente valen para lo mismo, como por ejemplo infrarrojos con ultrasonidos, o ultrasonidos con escáneres láser. Esto se debe a que cada tipo de sensor tiene un rango de medición limitado y se necesitan varias tecnologías para alcanzar el nivel de rango deseado.

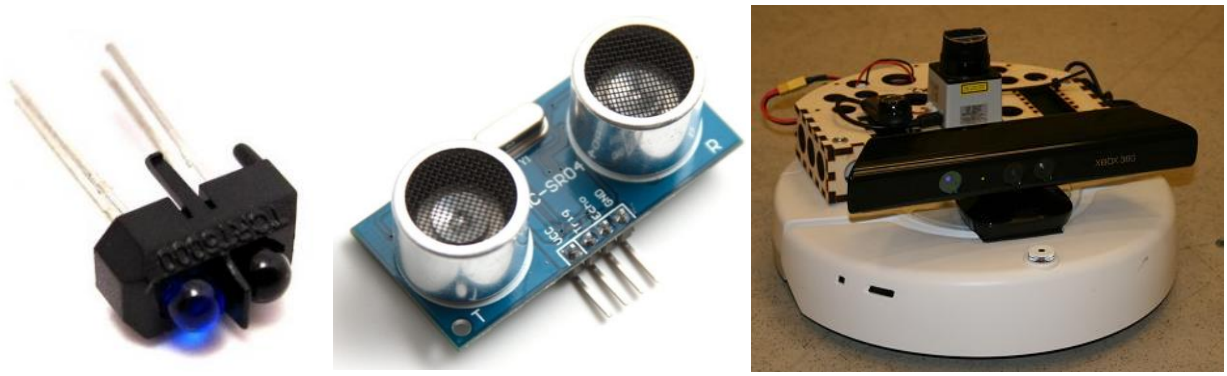

Figura 6. De izquierda a derecha: sensor infrarrojo (en oscuro el fototransistor para mayor absorción) [19], sensor ultrasonido [20], Kinect montada sobre un robot Roomba (la Kinect es muy práctica en robótica por su capacidad de hacer un barrido láser tridimensional, recreando un modelo del entorno mediante mapa de puntos) [21].

1.2.5 Detección de colisiones.

Si se pretende dar al robot más autonomía y que no haya que suministrarle la trayectoria, este debe ser capaz con su sistema informático de detectar automáticamente caminos libres de colisiones.

El robot debe trazar una trayectoria hacia su destino, y después seguirla lo más fielmente posible, a veces, no dispone de mapa y ha de maniobrar de acuerdo a lo que detecta a través de sus sensores, este tipo de técnicas reciben el nombre de *navegación reactiva* y una de las formas de implementarla es generando trayectorias locales destinadas a evitar los obstáculos que el robot vaya reconociendo.

La detección de colisiones es esencial por seguridad y fiabilidad, para su prevención es fundamental la obtención de información sobre la posición del robot, su posible carga y de los objetos (y personas) en su entorno. Su complejidad dependerá de:

- La complejidad e incertidumbre del entorno y medidas.
- El número de articulaciones.
- La movilidad de obstáculos.
- Restricciones cinemáticas y dinámicas.

Acciones necesarias a generar ante una posible colisión.

- *Disminución inmediata de velocidad y realización de parada*, esta acción debe ser de tiempo real, implementada en el software del controlador de a bordo.
- *Reacción autónoma,* lo cual consiste en movimientos seguros calculados con antelación por el planificador para evitar la colisión (navegación reactiva).

Tipos de colisiones.

a) Entre distintos enlaces del manipulador. Se evita utilizando el modelo cinemático del manipulador.

b) Entre enlaces y efecto final-carga. Se evita disponiendo de un modelo geométrico de la carga.

c) Entre enlaces y objetos del entorno.

d) Entre final-carga y objetos del entorno.

Será necesario también disponer de técnicas para detectar colisiones debidas a errores o imprecisiones en la ejecución de los caminos planificados, así como también, para evitar posibles falsas alarmas debidas a problemas relacionados con los sensores o con modelos demasiado conservadores, manteniendo la seguridad, todo ello en tiempo real.

Caminos libres de colisión.

El *espacio de configuraciones libres de colisión* será, el conjunto de configuraciones que puede adoptar el robot sin tocar a ninguno de los obstáculos existentes.

La configuración, según el tipo de robot, estará ligada a conocer:

- En robot manipuladores, la posición y orientación de los elementos del robot, el brazo y el efector final.
- En robot móviles, además de los elementos del robot, su posición y orientación en el plano o espacio.

La *planificación* de movimientos en robots móviles, se puede definir como el establecimiento de una trayectoria correcta que permita llevar el robot desde una configuración inicial hasta otra final dentro del espacio libre de colisiones. Por correcta se entiende que la trayectoria no colisione con el entorno y que cumpla con las restricciones cinemáticas (y dinámicas idealmente) de los vehículos rodados.

En general, la detección de colisiones mediante intersección de sólidos es compleja computacionalmente, haciéndose necesario emplear estructuras eficientes y métodos para acelerar la búsqueda.

Suponiendo que el algoritmo de planificación ha generado un camino, se trata de comprobar muestreando el

movimiento planificado cada cierto de tiempo, si está libre de colisión.

Diferentes sistemas de detección de colisiones.

Los distintos sistemas de detección de colisiones entre los que se puede escoger dependerán del modelado del problema y del método de planificación. En este Proyecto se ha modelado el entorno como una matriz que contiene información de si hay obstáculo o no en un punto dado. Así, la comprobación de colisión entre dos puntos consistiría en recorrer el segmento que une ambos puntos y comprobar en la matriz del mapa si hay obstáculos en algún punto del segmento. Otros ejemplos de sistema de detección de colisiones válidos para otros modelos distintos de problema son:

- Técnica de la construcción a priori de una estructura *octree- región* para los objetos estáticos. [22]
- Sistema para manipuladores con intersecciones entre *primitivas* de sólidos (*cilindro-esferas*).
- Algoritmos de cálculo de mínima distancia entre objetos.
- Utilización de las características del movimiento, como la velocidad de aproximación de dos sólidos en movimiento y la mínima distancia entre dos puntos de sus superficies.

1.3 Objeto del Proyecto Fin de Máster.

1.3.1 Objetivos del Proyecto.

Estudiar y comparar en el entorno MATLAB los algoritmos de planificación basados en modelo más populares actualmente, principalmente los derivados de búsqueda en grafo, RRT y RRT*. Estos se combinarán en algoritmos propuestos para conseguir un mejor desempeño y se aplicarán a entornos desafiantes, como por ejemplo: los planos de la ETSI, laberintos, entornos masificados aleatoriamente, etc.

La planificación de caminos se abordará desde tres perspectivas: búsqueda del primer camino subóptimo en estático, optimización del camino en estático, y optimización del camino en tiempo real. Cada una engloba la anterior, siendo la optimización del camino en tiempo real la más completa.

Para ello se programarán funciones de los algoritmos estudiados cuya llamada en código devuelva la trayectoria deseada, haciendo especial énfasis en la reducción del tiempo de ejecución y en la minimización del coste. Los parámetros de los algoritmos planificadores serán configurables para el ajuste a distintos problemas, pudiéndose visualizar gráficamente los resultados de así desearlo.

1.3.2 Planteamiento del Proyecto.

El planteamiento general es muy complejo, por lo que se han adoptado hipótesis simplificativas:

- Entorno conocido mediante su modelo.
- Robot holonómico modelado como un punto.

No se ha considerado ni cinemática ni dinámica en los algoritmos de este Proyecto, ya que es más eficiente la aplicación de un postprocesado que construir el grafo con condiciones cinemáticas y dinámicas [23], y dicho postprocesado es totalmente independiente de los métodos de planificación. En cualquier caso, si se quisiera construir el grafo con las condiciones cinemáticas y dinámicas, las modificaciones en los algoritmos de este Proyecto no serían muy significativas, y dado que los algoritmos aquí presentados son muy eficientes en tiempo de ejecución, podrían todavía responder rápido ante la carga computacional extra de tener en cuenta dichas condiciones. Pese a lo interesante que hubiera sido incluirlas, no se pudo hacer porque en algún momento había que limitar el alcance de este Proyecto.

El *modelo* del entorno se ha considerado como una matriz con mallado (estructura eficiente) y un grafo de nodos estratégicamente esparcidos. Para el caso plano la matriz del modelo puede ser extraída de una imagen (como un plano, por ejemplo, siempre que se edite y binarice correctamente) en la que cada elemento de la matriz (o píxel de la imagen) vale 0 si no hay obstáculo y 255 si hay obstáculo. Las dimensiones del robot se tendrán en cuenta dilatando la matriz de obstáculos. El que la matriz esté mallada significa que los nodos se registrarán por regiones, lo cual aligerará de forma crítica cálculos exhaustivos que suponen cuellos de botella, como se explicará más adelante. El grafo de nodos estratégicamente esparcidos sirve para aplicar una búsqueda en grafo con el algoritmo de Dijkstra, reduciendo en varios órdenes de magnitud la búsqueda del camino inicial subóptimo.

A continuación se explicará conceptualmente cómo está organizada la combinación de algoritmos más completa, la de tiempo real, y el porqué se ha hecho así. Realmente se entrará en detalle más adelante, pero tener en mente las ideas clave del algoritmo más completo desde el principio ayudará a tener las cosas más claras.

Se busca la **planificación tiempo real**, conseguida mediante planteamiento **Anytime Motion Planning**, el cual consiste en dos fases:

1º) **Resuelve la planificación de forma óptima en estático**, esto es sin obstáculos móviles y con los puntos inicial y final fijos. Se deja optimizando el algoritmo hasta que sea necesario moverse o hasta un cierto tamaño de árbol de búsqueda.

   1º) Búsqueda del primer camino inicial subóptimo en estático: El objetivo es conseguir lo más rápido posible un primer camino que una los puntos inicial y final, dando una calidad de solución buena. Se verá más adelante que el algoritmo que mejor resuelve esta fase es el RRT multi-query + Dijkstra. Por último se aplica un postprocesado por desigualdad triangular.

   2º) Optimización del camino en estático: Sobre el camino inicial anterior se comienza a construir el árbol de optimización con los algoritmos Informed RRT* y RRT*-Smart, que mediante sus heurísticas consiguen acelerar la convergencia al óptimo. La existencia de camino inicial es vital para el funcionamiento de las heurísticas, si no hubiera camino inicial ambos algoritmos convergerían a velocidad RRT*, lo cual es mucho más lento. Es por eso que primero se calcula un camino inicial por otros medios más rápidos.

2º) **Optimización en tiempo real:** Cuando llega la hora de mover el agente, continúa optimizando en tiempo real la trayectoria a seguir. Para esto se aplica RT-RRT* de forma continua hasta que el agente haya cumplido su objetivo. Este algoritmo RT-TRRT* permite mantener y adaptar el árbol a cambios de posición del agente, movimiento del objetivo (modo persecución), y al movimiento de obstáculos móviles. RT-RRT* incluye ya Informed RRT*, aunque se ha modificado para que también incluya RRT*-Smart y RRT*FN. Este último algoritmo lo que consigue es permitir seguir iterando y optimizando el árbol incluso tras su llenado, limitando su tamaño y ahorrando memoria.

2 PLANIFICACIÓN TRADICIONAL BASADA EN MODELO GEOMÉTRICO

Como ha quedado suficientemente indicado anteriormente, el cálculo de trayectorias en un entorno dado es un aspecto de gran relevancia en el ámbito de la robótica. Las características del modelo geométrico del robot y su entorno tendrán una notable influencia en la eficiencia de los métodos de detección de colisiones y planificación de caminos.

Como recordatorio, el problema de la planificación de movimientos en *robots móviles* consistirá en determinar trayectorias admisibles o conjunto de movimientos que permitan llevarlos desde una configuración *inicial* con su carga, a otra *final* cumpliendo un *objetivo*, dentro del espacio de configuraciones libres de colisión, se entiende por "admisible" que la trayectoria cumpla con la restricción cinemática del vehículo.

A continuación se abordan algunos métodos geométricos tradicionales basados en modelo para solucionar el problema. También se llaman métodos *"roadmap" (RM)*, caracterizados por construir primero una descripción del espacio libre con la forma de una red o grafo. Lo segundo que se hace es unir los puntos inicial y final al grafo, para por último escoger aquél camino dentro del "roadmap" que minimice el coste (distancia o tiempo).

Estos métodos son válidos para problemas de planificación sencillos, off-line, en los que no se pide mucha complejidad al entorno o los robots no tienen muchos grados de libertad. Sin embargo, en robots con muchos grados de libertad y en entornos muy complejos a los que se le pida un nivel de precisión razonable, el número de nodos y arcos del modelado del entorno se dispara exponencialmente (problema NP-completo [10]) y con ellos el número de consultas necesarias, haciendo inviable computacionalmente este tipo de algoritmos en muchos casos. De todo esto se hablará en este capítulo y el siguiente.

Pese a ello, estos métodos también presentan ventajas. Existen sinergias entre los algoritmos tradicionales de búsqueda en grafo y los probabilísticos. Usando un grafo de pocos nodos posicionados estratégicamente se puede reducir el nivel de dificultad del problema en la primera fase de búsqueda de un primer camino inicial. Esto se traduce en una reducción de tiempo en dicha fase, incluso en situaciones de mínimo local difícil de resolver mediante cualquier otro método. No solo eso, sino que se puede demostrar que la solución es de mayor calidad que la proporcionada por un método probabilístico.

Este capítulo servirá de introducción de estos métodos, explicando aquellos con interés histórico y también explicando aquel usado en este Proyecto, el algoritmo Dijkstra.

2.1 Construcción del grafo o "roadmap".

Los *modelos geométricos,* abordan la planificación empleando modelos que introducen métodos de búsqueda en el espacio cartesiano y en el de configuraciones. Existen distintas formas de modelar el espacio por el que se puede mover el robot libre de colisiones:

2.1.1 Grafos de visibilidad

Los grafos de visibilidad proporcionan un enfoque geométrico para solventar el problema de la planificación. Este método opera con modelos poligonales de entorno, se encuentra muy extendido y existen algoritmos que construyen esta clase de grafos con un coste computacional relativamente bajo.

Está limitado a modelos de entornos definidos como polígonos, y puede trabajar tanto en el plano como en el espacio. En el plano garantiza generar grafos que contienen el camino óptimo, aunque esto no se da en el espacio. Tienen la contrapartida de ajustarse demasiado cerca de los obstáculos, necesitando expansionarlos por seguridad.

En un conjunto de distintos obstáculos poligonales situados en un plano, el grafo de visibilidad estará formado por las uniones de los pares de vértices que se ven mutuamente, sin que se produzca intersección alguna con los obstáculos.

Si pensamos que las uniones entre los puntos es una banda elástica cuyos extremos son los puntos origen y final del recorrido, tiramos de ella ajustándola hasta tocar las aristas o vértices de los obstáculos, el resultado de líneas rectas a través del espacio libre entre ellos que se forma, será el camino mínimo. [24] [25]

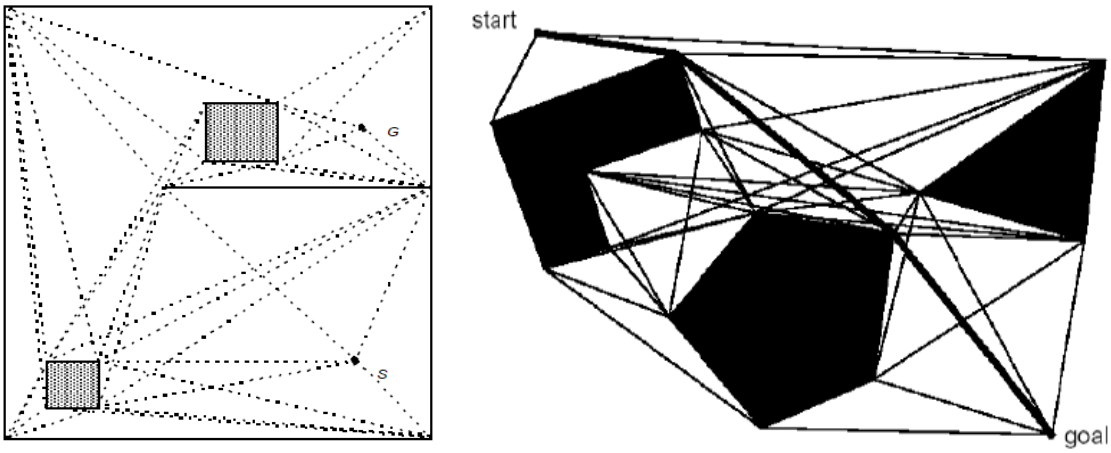

Figura 7. Ejemplos de modelado por grafos de visibilidad. [11]

2.1.2 Diagramas de Voronoi

Se basa en una descomposición de un espacio métrico en regiones asociada a la presencia de obstáculo, asignándole en dicha descomposición a cada objeto una región del espacio métrico, a su vez formada por los puntos más cercanos a él que a ninguno de los otros objetos. [26]

En conclusión, dividir el espacio en tantas regiones como puntos u objetos tengamos, asignando cada punto a la región formada por todo lo que está más cerca de él que de nadie.

Ejemplo, en un conjunto de *ocho puntos* en el *plano euclídeo*, su *Diagrama de Voronoi*, será la partición del plano de este estilo en ocho zonas, estando en cada zona uno de los puntos. Ante la presencia de un nuevo punto extraño en una de las zonas, se reconocerá de inmediato cuál de los puntos originales está más cercano a él, el punto generador de la región en la que ha caído. Esta propiedad es aplicable directamente a clasificación (tal como encontrar en una determinada ciudad, el restaurante más cercano a tu ubicación).

En planificación, un diagrama de Voronoi representaría los caminos más conservadores y alejados a obstáculos (modelados como puntos) que puede seguir nuestro agente.

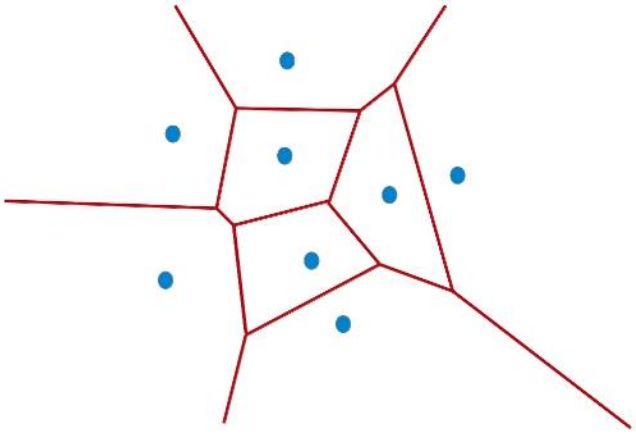

Figura 8. El diagrama de Voronoi es aquel cuyas líneas son equidistantes a los obstáculos modelados como puntos. [26]

Como curiosidad, los diagramas de Voronoi son de esas propiedades matemáticas que se pueden observar en la naturaleza:

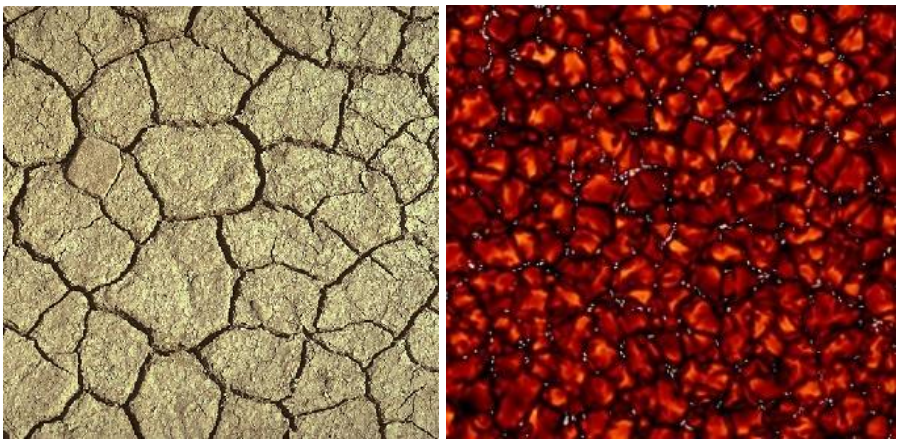

Figura 9. Diagramas de Voronoi construidos de forma natural a partir de los últimos puntos húmedos en la tierra (izquierda) y a partir de los puntos más calientes de la superficie del Sol (derecha). [26]

2.1.3 Descomposición en celdas

Se trata de la división del entorno en celdas, cada una con sus propios atributos, grado de ocupación, probabilidad, información del terreno, etc. A través de este modelo puede modelarse cualquier entorno, siendo fácil de implementar y de hacer planificación sobre él.

2.1.3.1 Modelado mediante ocupación de celdas

El volumen de trabajo se divide en celdas o rejilla de una determinarla dimensión, marcándose cada celda como ocupada o libre. La actualización del modelo es relativamente simple, pero su consulta puede ser muy ineficiente si la resolución es alta, lo cual puede ser necesario en las proximidades de obstáculos y posición final.

Se han propuesto también modelos que tienen en cuenta la incertidumbre de las medidas asociando a cada rejilla una probabilidad de estar ocupada, y actualizando la probabilidad cuando se toman nuevas medidas [27]

Para hacer más eficiente el modelo puede acudirse a técnicas de tamaño de celda variable. Esto es lo que se ha implementado en este Proyecto para acelerar la consulta del problema de nodo más cercano.

2.1.3.2 Empleo de estructuras jerárquicas.

Se trata de emplear estructuras de datos eficientes que proporcionen una resolución elevada, sólo en las regiones del espacio de trabajo en que sea necesaria.

Los *quadtrees* y *octrees* [28] [29] son estructuras de datos de amplia utilización en informática gráfica.

El *quadtree,* estructura jerarquizada subdivida recursivamente en volúmenes cúbicos más pequeños.

El *octree,* estructura jerárquica que subdivide recursivamente un volumen cúbico en ocho volúmenes más pequeños, denominados octantes, hasta que se satisface un cierto criterio al que se denomina regla de la descomposición, dependiendo de la regla de subdivisión, existen diversas variedades de *octrees.*

- *Octree-región,* estructura en la cual se comienza con un cubo que incluye todo el espacio de trabajo. El proceso de subdivisión se realiza hasta que cada octante esté completamente dentro de un objeto o completamente vacío. La aplicación del *octree-región* en algoritmos de evitación de colisiones y planificación de caminos ha sido estudiada piar numerosos investigadores [30] [31].
- *Octrees N-objetos,* también subdividen el espacio, pero en este caso se almacena una lista de los objetos asociados a cada nodo del árbol. El proceso de subdivisión acaba cuando en cada hoja del árbol no existen más de *N-objetos*. Estos *octrees* han sido también empleados para la evitación de obstáculos en

tiempo real [32]. Como propiedad importante cabe señalar, que sólo los objetos que comparten un nodo pueden colisionar de forma inminente.

Existen autores que emplean *octrees*, y en particular *octrees-región*, con cuatro dimensiones, añadiendo el tiempo como una dimensión, para planificación de movimientos [33].

2.1.3.3 Modelos del entorno basado en primitivas 3D de sólidos.

Se trata de modelar el robot y su entorno empleando objetos geométricos simples, que en términos informáticos corresponden a *primitivas, esferas, cilindros, paralelepípedos, conos y planos*.

Se emplean también otras estructuras que resultan de la combinación de las anteriores, destacando los *cilindro-esferas* o *"cylspheres"*, consistentes en un cilindro con una esfera en cada uno de sus extremos, mediante las cuales se modela el enlace de un manipulador con una articulación en cada extremo.

La elección de las *primitivas* es básica para conseguir una representación suficientemente precisa y fiable, a su vez, facilitar las operaciones de intersección de primitivas en las cuales se basan los algoritmos de detección de colisiones. Compruébese que las esferas son objetos que se describen por un solo parámetro, el radio, y la descripción de su movimiento se reduce al del movimiento de su centro. Existen también algoritmos suficientes para intersección de esferas [34]. Sin embargo, el número de esferas necesarias para conseguir una aproximación aceptable puede ser muy grande.

Por otra parte, la representación mediante *"primitivas"* puede combinarse también con las *estructuras de datos jerárquicas,* ejemplo, el entorno supuesto estático representado mediante *octree-región* y el robot en movimiento mediante *primitivas* cuya posición es fácilmente actualizable.

En Shaffer y Herb [32] se modela el manipulador mediarte *cilindro-esferas*, empleándose un *octree N-objetos* para minimizar el número de intersecciones, de primitivas sólo si comparten un nodo de *octree N-objetos*.

2.1.4 Expansión de obstáculos.

Existen numerosos algoritmos que permiten determinar el camino más corto y libre de obstáculos entre dos puntos. Para aplicarse cuando lo que se mueve es un objeto de determinadas dimensiones, es necesario adoptar modelos apropiados, el más simple es un círculo de determinado radio. De esta forma, si todos los posibles obstáculos se expanden según arcos circulares de radio "*r"*, pueden aplicarse los mismos algoritmos que suponen que el objeto que se mueve es un punto.

Esta técnica presenta problemas cuando el robot es de contorno rectangular, la expansión de dichos obstáculos mediante arcos circulares de radio "*r"* podría ser demasiado conservadora, pudiendo llegar al caso extremo de que no existiera ningún camino que permitiera el paso del robot entre los obstáculos, si se adopta un *contorno poligonal* como modelo del robot, se puede determinar una trayectoria para pasar entre los obstáculos, en este caso no basta con conocer las coordenadas del centro de guía del robot, se necesita también la orientación, el ángulo.

En un plano, 2D se necesitarán tres parámetros para especificar la configuración del robot las coordenadas "*x"* e "*y"* y su *"orientación"*. En un espacio, 3D, se emplearían poliedros en vez de contornos poligonales para representar al robot, su posición y orientación, se necesitan seis parámetros, tres coordenadas y tres ángulos de orientación.

En este Proyecto se ha considerado la expansión de obstáculos para facilitar el modelado del problema.

2.2 Métodos de búsqueda en grafo.

Los métodos *"roadmap"* construyen un *grafo* que representa en sus *nodos* algunas configuraciones factibles del robot, y en sus *arcos* la posibilidad constatada de conexión entre dos nodos. Una vez construido este grafo, halla un camino entre el origen y el destino conecta dichos puntos y hallar una secuencia dentro de él.

Se pueden definir los *algoritmos* como un conjunto ordenado de operaciones sistemáticas que permiten hacer un cálculo y hallar la solución de un tipo de problemas. En problemas de búsqueda en grafo para planificación de caminos, los algoritmos de búsqueda en grafo son los encargados de encontrar la sucesión de arcos del grafo que, conectados entre sí, encuentran un camino viable que une los nodos inicio y fin. Además deben ser capaces de minimizar el coste de desplazamiento, normalmente distancia o tiempo. Para este Proyecto, se entenderá el coste como la distancia. A continuación se comentarán algunos de los más relevantes:

2.2.1 Algoritmo Dijkstra.

El algoritmo de Dijkstra, también llamado algoritmo de ruta mínima, es un algoritmo para la determinación del camino más corto desde un vértice o nodo origen al resto de los nodos de un grafo con costes en cada arco. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959. [35]

La idea está en explorar todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices. Cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, el algoritmo se detiene. En los siguientes dos párrafos se explicará en detalle el algoritmo.

Para ello primero se prepara el problema, inicializando los costes de todos los nodos distintos del origen con valor infinito (el origen tendrá coste cero siempre) y clasificando todos los nodos como "abiertos". Un nodo "abierto" es aquel que no se ha explorado, y "cerrado" aquel explorado totalmente.

Después se empieza el proceso iterativo. Partiendo desde el nodo origen, se le asigna a todos los nodos accesibles desde él el mínimo valor entre el costo actual del nodo y el que tendría si se accede a él desde el origen (coste del nodo origen más el coste del arco). Como inicialmente los costos son infinitos y el coste del origen es nulo, se asignaran los costos de los arcos que los unen con el origen. Una vez explorado totalmente el nodo origen se clasifica como "cerrado". Este proceso se repetiría nodo a nodo, cerrando nodos en orden de menor a mayor coste actual hasta poner como "cerrado" el nodo destino, en cuyo caso se dejaría de iterar en el algoritmo.

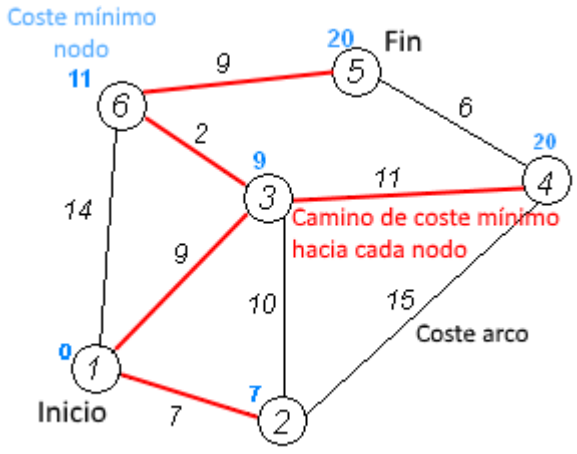

Figura 10. Resultado de aplicar el algoritmo Dijkstra en este grafo de ejemplo.

La complejidad computacional del algoritmo de Dijkstra no es alta ya que cada una de sus iteraciones consisten en apenas unas comparaciones, sumas y asignaciones de memoria, aunque el número de iteraciones necesarias es de N-1, siendo N el número de nodos total del grafo. Así, se puede decir que su complejidad es clase P (tiempo polinómico).

Esto convierte el algoritmo de Dijkstra en uno *exhaustivo*, esto quiere decir que siempre necesitará de explorar la totalidad del grafo para poder garantizar optimalidad, al fin y al cabo calcula los caminos de coste mínimo

desde el origen hacia todos y cada uno de los nodos. Al ser exhaustivo resulta vital mantener pequeño el tamaño grafo. Si se pretende una solución de precisión resulta computacionalmente inviable.

En este Proyecto se ha usado el algoritmo de Dijkstra con un grafo predefinido, el cual se diseñó con la cantidad mínima de nodos posicionados estratégicamente. De esta forma se consigue reducir el nivel de complejidad y eliminar totalmente la influencia de mínimos locales en entornos tan complejos como, por ejemplo, la Entreplanta 2 de la ETSI (Escuela Técnica Superior de Ingeniería) de Sevilla.

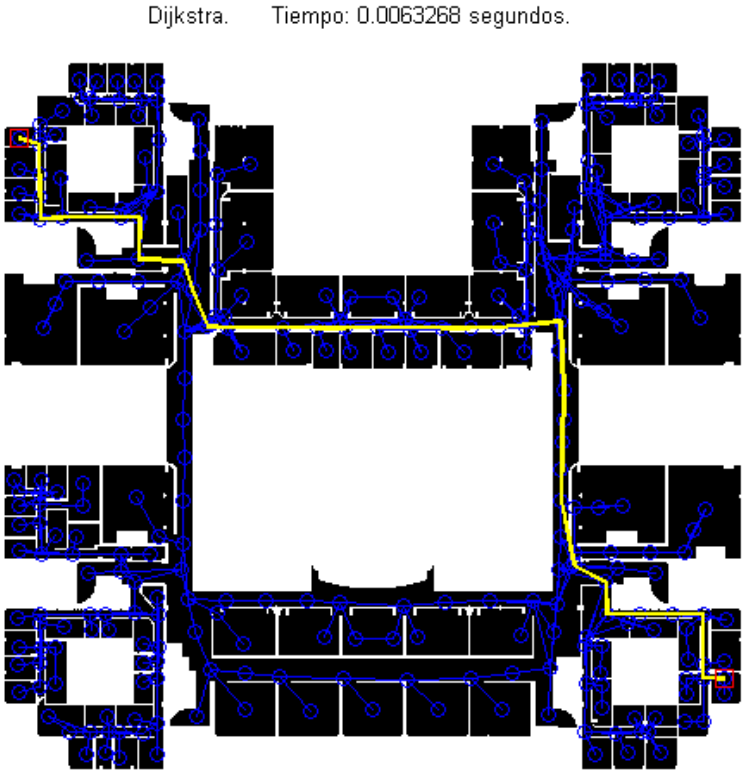

Figura 11. Algoritmo de Dijkstra aplicado al grafo predefinido de la Entreplanta 2 de la ETSI de Sevilla.

Recordar que el algoritmo de Dijkstra es muy rápido siempre que el grafo tenga un tamaño reducido y que necesita explorar casi la totalidad de los nodos del grafo, garantizando el camino mínimo dentro del mismo. Observando la imagen superior puede parecer que el tamaño del grafo es grande, al fin y al cabo son N=306 nodos, algo desorbitado para un cálculo manual, pero para una computadora actual N-1=305 iteraciones no es nada. En el momento de la captura se tardó 6 milisegundos en completar el algoritmo, lo cual es una cantidad de tiempo despreciable respecto al tiempo de navegación de un robot.

Evidentemente, que la solución dada sea la óptima dentro del grafo predefinido no significa que sea la *óptima global*, ya que el grafo es de poca resolución. Habría que montar un grafo de visibilidad para garantizar el óptimo, lo cual es una locura en entornos tales como este. Ahí es donde entran en juego los algoritmos RRT* que optimizan dicha trayectoria ajustándola mejor a los obstáculos.

2.2.2 Algoritmos A* y D*.

Este apartado no pretende explicar los algoritmos A* y D* ya que no son de interés en este Proyecto, pero qué menos que mencionarlos brevemente y comentarlos dada su relevancia.

A* surge como adaptación con heurística del algoritmo Dijkstra, pretendiendo reducir la cantidad de consultas mediante una heurística de predicción del camino que queda por recorrer, garantizando la optimalidad. Esto lo consigue prescindiendo de calcular el camino mínimo desde el inicio hasta cada uno de los nodos, por lo que no sirve para múltiple consulta como Dijkstra. Esto es ventajoso con grafos demasiado grandes, pero con grafos pequeños como los que manejaremos en este Proyecto no parece un algoritmo adecuado, por eso se escogió el algoritmo Dijkstra.

D* a su vez surge como adaptación del A* para entornos dinámicos, comportándose más eficientemente que un replanificador A* en entornos grandes y complejos. Garantiza optimalidad y es completo

3 PLANIFICACIÓN PROBABILÍSTICA BASADA EN MODELO

Los *métodos probabilísticos de planificación*, a diferencia de los clásicos por geometría, se basan en la construcción de un grafo mediante la creación de *puntos aleatorios* en el espacio de trabajo (fase de muestreo o aprendizaje), intentando la unión con el grafo de los puntos iniciales y finales y su posterior conexión dentro del mismo minimizando el coste (fase de búsqueda). También algunos continúan después optimizando dicho grafo (fase de optimización), no teniendo por qué ir separadas secuencialmente dichas fases.

El nombre de método probabilístico se debe a que se muestrea a ciegas el espacio de trabajo, confiando en que cuantos más puntos se muestreen más probable será encontrar un camino que una el punto inicial con el final.

En la *fase de muestreo* cada punto que se selecciona, aleatoriamente, debe cumplir *ser libre de obstáculos y poderse unir con el grafo mediante un camino sin colisiones*.

En la *fase de búsqueda* se busca dentro del grafo obtenido en la fase de muestreo aquel que minimice el coste, definiendo como coste la distancia o el tiempo que tarda en alcanzar el punto final desde el punto de inicio.

No todos los métodos de planificación probabilística incluyen *fase de optimización* debido a que aumentan el tiempo de ejecución del programa. Aunque siempre es deseable reducir el coste con alguna estrategia barata computacionalmente como por ejemplo, el *postprocesado* de desigualdad triangular.

En este capítulo se entrará más en detalle.

3.1 Justificación de los algoritmos probabilísticos.

Estos algoritmos son considerados actualmente como el estado del arte de la planificación de caminos en problemas muy complejos, empleándose en entornos de decenas o cientos de dimensiones tales como robots manipuladores, doblado de moléculas biológicas, movimiento de personajes digitales, piernas robóticas, etc. En este Proyecto serán los que más aparezcan, concretamente los algoritmos derivados del RRT.

En estos casos complejos, que involucran altos números de grados de libertad, entorno demasiado completo o gran número de *robots* compartiendo e interaccionando en un mismo espacio de trabajo, los métodos tradicionales de planificación geométrica no responden tan rápido como se exige. Necesitamos métodos flexibles adaptables a la complejidad creciente de la robótica.

La principal ventaja de los métodos probabilísticos es que son mucho menos sensibles a la complejidad del entorno, ya que no necesitan hacer cálculos con los obstáculos, los cuales pueden tener una geometría arbitraria y compleja. El no tener que operar con los obstáculos se traduce en una rápida y eficiente expansión del grafo de búsqueda para todo tipo de problemas, buen comportamiento ante mínimos locales, y en *completitud probabilística* (la probabilidad de encontrar una solución subóptima tiende a 1 a medida que el árbol se expande). Los puntos negativos de los métodos probabilísticos es su incapacidad de reconocer la no existencia de camino libre de obstáculos, aunque como ya se ha comentado, normalmente si no encuentra camino en un tiempo es que no existe.

Esta planificación probabilística puede parecer poco intuitiva para un humano, ya que por ejemplo, si quisiéramos coger un objeto visible para nosotros de una mesa de forma probabilística, no lo haríamos hasta haberlo buscado y encontrado aleatoriamente en otros múltiples sitios (sillas, mochilas, suelo, otras mesas, etc.). La clave está en que para la computadora ese objeto no es "visible" de la misma forma evidente que para un humano. Un mejor símil humano para la planificación probabilística podría ser buscar algo en una mesa con los ojos cerrados. Para ello tendría que buscarse por la mesa palpando puntos al azar.

3.2 Mapas probabilísticos o PRM (Probabilistic Roadmap Method).

Se comentará brevemente la base de los algoritmos de planificación por mapas probabilísticos o PRMs (Probabilistic Roadmap Method). Al ser un *método probabilístico* una de sus principales virtudes, como ya se ha comentado, es su eficacia en el cálculo de trayectorias con *robots* de muchos grados de libertad. Puede ser tanto de un *grafo* (consulta única) como de varios (múltiple consulta). En cualquier caso la metodología secuencial (cada fase es independiente) es la siguiente:

- *Fase de muestreo o de aprendizaje:* Un grafo llamado mapa probabilístico se construye a partir de puntos buscados aleatoriamente en el espacio de trabajo libre, puntos que deben cumplir que se pueden conectar al grafo mediante una trayectoria sin colisión. El objetivo es alcanzar una densidad de nodos determinada.
- *Fase de búsqueda:* Trata de unir el *grafo aleatorio* existente con los puntos *inicial* y *final*, puntos que están *fuera del grafo* (normalmente). Después mediante algún algoritmo de búsqueda en grafo (Dijkstra usualmente) se calcula el mínimo camino que une los puntos inicial y final dentro del grafo.
- *Fase de optimización:* Mediante un algoritmo de búsqueda en *grafo* se puede tratar de optimizar el camino devuelto por la fase de búsqueda (esta fase es opcional). Esto se hace en vez de parando el algoritmo al encontrar el camino mínimo, continuando muestreando e introduciendo nodos nuevos.

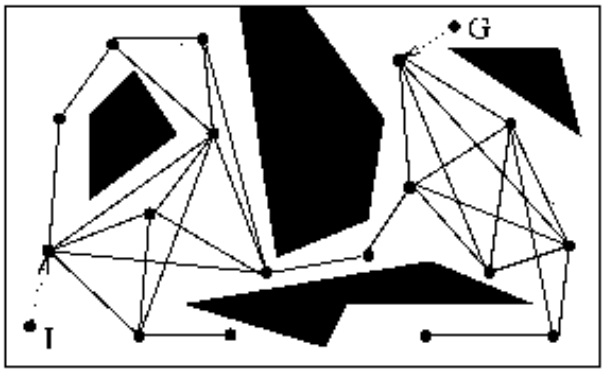

Figura 12. Grafo construido por PRM de consulta única. [36]

Las ventajas de los PRM son las inherentes en los métodos probabilísticos. No es necesario calcular los límites de los obstáculos ni de la región libre, tarea nada fácil cuando el número de dimensiones del problema es elevado o los obstáculos tienen formas totalmente arbitrarias. Basta con utilizar un algoritmo que compruebe si existe o no colisión, lo cual en la mayoría de los casos resulta bastante más práctico. Otra de las propiedades que hacen atractivo este método es su *completitud probabilística,* que como se dijo en la página anterior, consiste en que la probabilidad de encontrar una solución subóptima tiende a 1 a medida que el árbol se expande. [37]

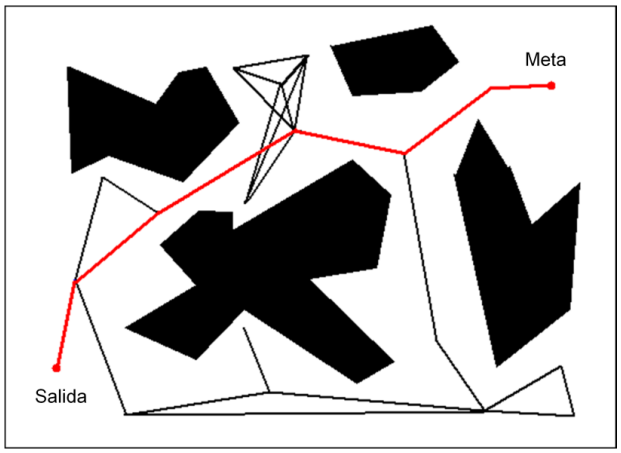

Figura 13. Grafo generado por un PRM de múltiple consulta. [23]

3.3 Exploración rápida de árboles aleatorios o RRT (Rapidly Exploring Random Tree).

Este algoritmo y sus derivados serán usados intensivamente en este Proyecto. En este apartado se presentará brevemente ya que se irá profundizando poco a poco en otros capítulos de la memoria. A grandes rasgos, es un algoritmo que tiene las mismas ventajas que un PRM (completitud probabilística, no necesidad de calcular límites de obstáculos, etc.) pero con mayor versatilidad.

Es un algoritmo bastante popular, fue introducido en la literatura científica por Lavalle, S.M. en 1998 en su artículo "Rapidly-exploring random trees: A new tool for path planning". Desde entonces debido a su versatilidad y eficacia ha sufrido un constante creciendo en número de aplicaciones y en algoritmos alternativos que de alguna forma lo mejoran. [38]

Consiste en un método de planificación probabilístico basado en modelo parecido al antes descrito pero con algunas diferencias. La principal diferencia que destaca a la vista es que el grafo se expande en forma de *árbol* desde el punto inicial, el cual es la *raíz*. Esto se debe a su peculiar fase de muestreo.

La fase de muestreo busca un punto aleatorio cualquiera del espacio libre de obstáculos, con la particularidad de que no incorpora dicho punto al grafo si la trayectoria entre ambos está libre de colisión, sino que antes de hacerlo acerca el punto aleatorio una distancia determinada (parámetro configurable del planificador) al punto más cercano del grafo. Si la trayectoria entre el punto nuevo y el punto del grafo (nodo) está libre de colisión se incorpora el punto nuevo. Esta fase de muestreo da una forma muy particular al grafo, asemejándose a un árbol cuya longitud de rama es constante, por ello a partir de ahora se le denominará *árbol.*

Como el *árbol* parte desde el punto inicial, la fase de búsqueda como tal consiste en intentar unir el último nodo introducido al árbol con el *punto final*. De esta forma te evitas tener una fase de búsqueda independiente y posterior al muestreo, puedes hacer ambas fases a la vez, resultando en un algoritmo más eficiente que los PRM (que además necesitan de unir el grafo a los puntos inicial y final). El algoritmo se detendrá cuando incorporemos un nodo al árbol dentro de un radio del punto final si tiene visibilidad con él, incluyéndolo.

El grafo en forma de árbol garantiza que para llegar desde el final al origen solo tienes que moverte de nodo hijo a nodo padre, comenzando desde el nodo fin. Este camino es el óptimo dentro del grafo sin necesidad de aplicar Dijkstra, aunque por supuesto fuera del grafo no es una trayectoria óptima, sino *subóptima*. Las trayectorias son subóptimas ya que no se considera ninguna función de coste ni tiene fase de optimización.

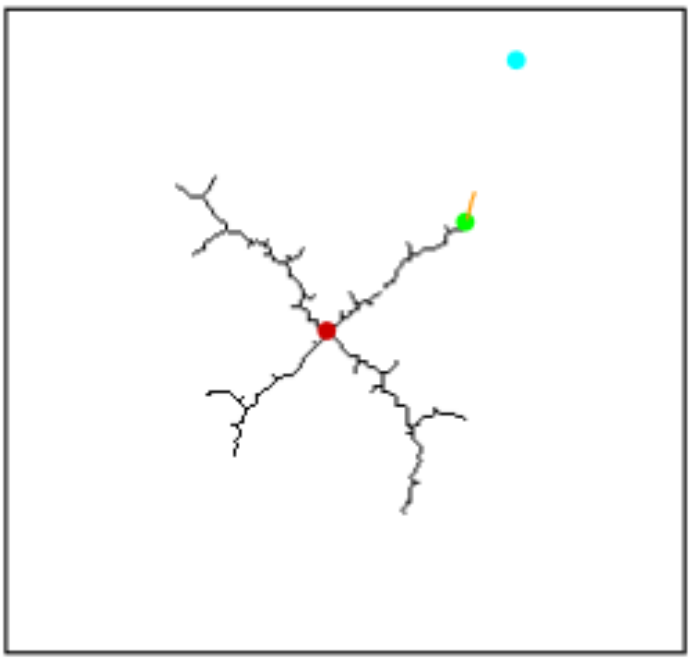

Figura 14. Árbol RRT expandiéndose una distancia constante hacia el punto aleatorio a partir del nodo más cercano. [1]

Como ya se ha comentado los caminos generados no serán óptimos, de hecho resultan tortuosos y oscilantes, con muchos giros e incluso rotaciones sobre sí mismos. Esto se debe a la naturaleza puramente aleatoria de la construcción del *árbol*. Para mejorar esto de forma barata computacionalmente se suele recurrir al *postprocesado de trayectoria por desigualdad triangular*, esto es un *postprocesado* que se aplica al camino una vez se ha encontrado que consiste en unir cada nodo trayectoria con el más lejano visible. Sigue sin ser óptimo, pero al menos se corrigen las oscilaciones y giros sobre sí mismo sin sentido alguno.

Al ser un proceso tan eficaz y barato computacionalmente este postprocesado se aplicará no solo a este algoritmo, sino a todos los que no sean en tiempo real estudiados en este Proyecto.

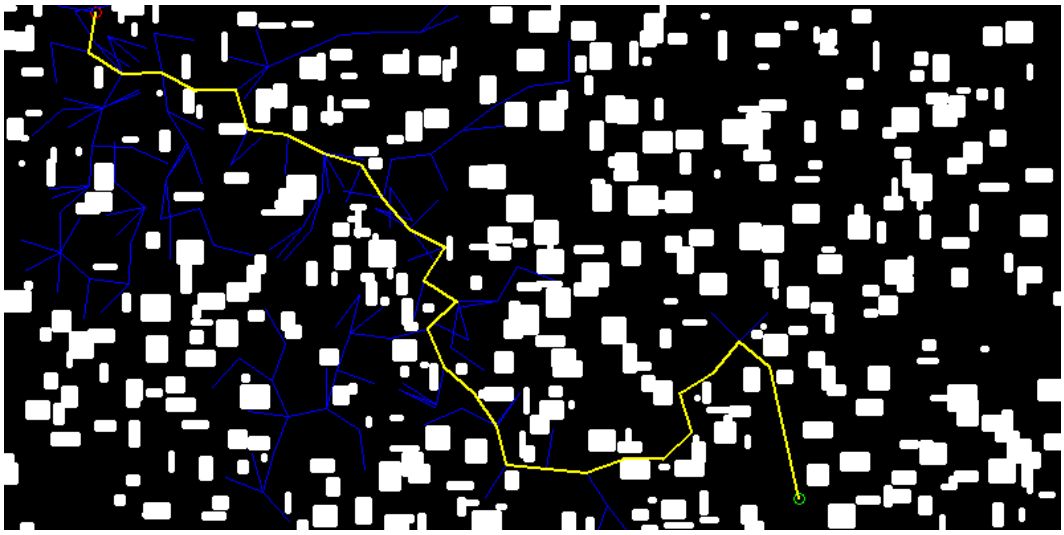

Figura 15. RRT resolviendo un mapa aleatorio de obstáculos expandidos para tener en cuenta la dimensión del robot.

Para corregir la falta de optimización surgieron múltiples alternativas, pero no fue hasta la aparición del algoritmo *RRT** cuando se produjo la auténtica revolución de la optimización de caminos *RRT*. *RRT** garantiza la optimalidad pero es *asintóticamente óptimo*, lo cual significa que el óptimo se garantiza cuando el tiempo tiende a infinito.

Los algoritmos *Informed RRT** y *RRT*-Smart* implementan heurísticas para acelerar la convergencia hacia el óptimo, esto lo consiguen focalizando la fase de muestreo en las zonas donde meter nodos nuevos tienen más probabilidad de mejora de trayectoria. *RRT*FN* se creó para que, limitando el número de nodos del árbol, se pudiera seguir optimizando sin límite.

Un problema típico en planificación es la presencia de *mínimos locales* en el modelo, que limitan la expansión del árbol, resultando en una exploración deficiente del entorno. Los métodos probabilísticos en general se comportan bien ante mínimos locales y con el tiempo salen de ellos, aunque sí son ralentizados, lo cual no es deseable. Para solucionarlo surgieron otros tantos algoritmos. En este Proyecto se estudiaron dos alternativas, la primera fue el RRT-Connect, el cual crea dos árboles con una heurística voraz o "greedy" que hace que se persigan mutuamente entre ellos hasta encontrarse. La segunda combinar un RRT multi-query con Dijkstra. Como se verá en capítulos posteriores, resultó mucho más rápida esta segunda opción.

Cabe decir que tanto el algoritmo RRT como sus derivados son fácilmente extrapolables del plano al espacio, resolviendo trayectorias eficientemente ya no solo para UGV (Unmanned Ground Vehicle) limitados al caso plano, sino para UAV (Unmanned Aerial Vehicle) e incluso ROUV (Remotely Operated Underwater Vehicle) en espacio tridimensional.

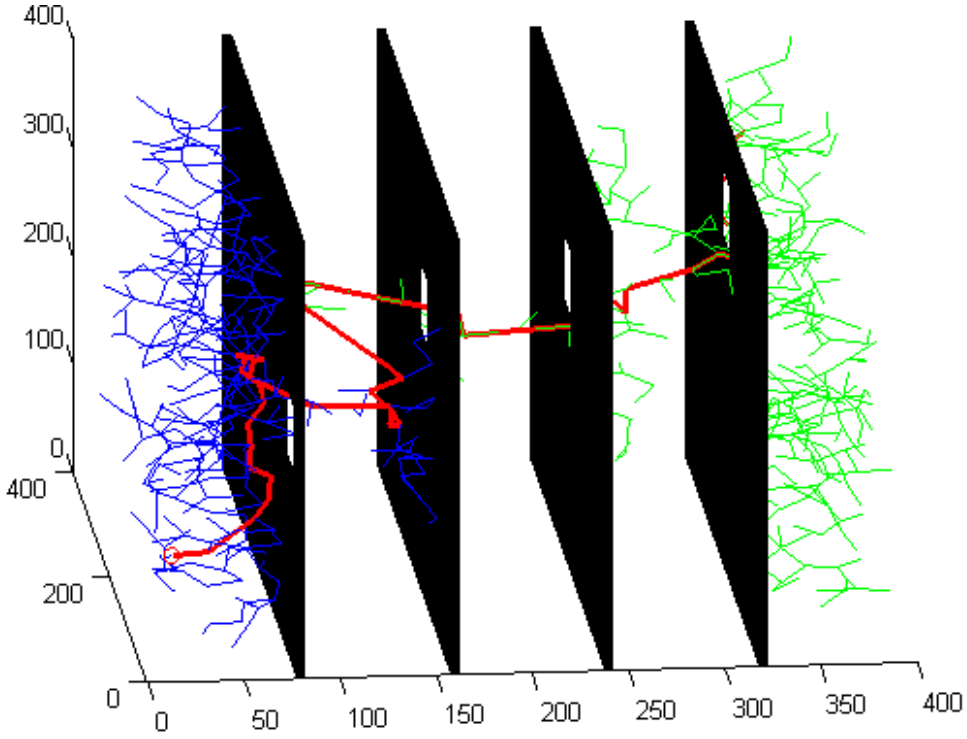

Figura 16. Ejemplo de un algoritmo RRT-Connect 3D sin postprocesado.

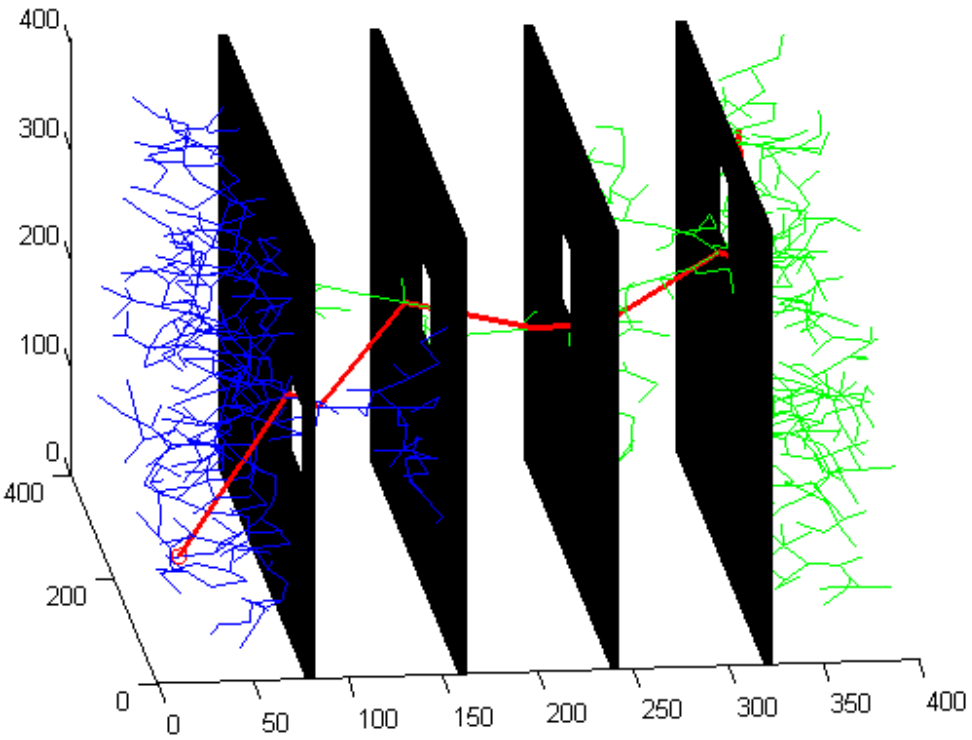

Figura 17. Ejemplos de un algoritmo RRT-Connect 3D con postprocesado.

Por último añadir que todos los algoritmos estudiados en esta memoria son para *robots holonómicos*. En la práctica muchos de los robots móviles corresponden a no holonómicos, de aquí la limitación de estos algoritmos en casos prácticos, ya que las trayectorias que devuelven no tienen en consideración las restricciones cinemáticas (por ejemplo ángulos máximos de giro y velocidad) ni dinámicas (aceleraciones, inercias, etc.). Esto se traduce en que devuelven trayectorias con giros demasiado bruscos para el robot.

Los RRT con restricciones cinemáticas devuelven trayectorias suavizadas realizables para el robot. Esto significa que los parámetros del planificador deben contemplar modificaciones en función de la geometría del robot para alcanzar la optimización máxima. De igual forma pasa con los dinámicos, que tienen en cuenta más parámetros del sistema robótico y devuelven trayectorias incluso más suavizadas.

La justificación de que en este Proyecto no se considerara ni cinemática ni dinámica es que, según la tesis doctoral de Diego Antonio López García "Nuevas aportaciones en algoritmos de planificación para la ejecución de maniobras en robots autónomos no holónomos" [23], la inclusión de restricciones cinemáticas y dinámicas durante la construcción del árbol resulta lenta e ineficiente. Es mucho más rápido aplicar algoritmos independientes de *postprocesado,* tales como los que él mismo propone en su tesis. Dichos algoritmos de postprocesado reciben como datos de entrada el mapa de obstáculos y la trayectoria óptima sin cinemática ni dinámica, dando como resultado una trayectoria perfectamente viable según las restricciones del robot.

Dicho postprocesado es totalmente independiente de los métodos de planificación. En cualquier caso, si se quisiera construir el grafo con las condiciones cinemáticas y dinámicas, las modificaciones en los algoritmos de este Proyecto no serían muy significativas, y dado que los algoritmos aquí presentados son muy eficientes en tiempo de ejecución, podrían todavía responder rápido ante la carga computacional extra de tener en cuenta dichas condiciones. Pese a lo interesante que hubiera sido incluirlas, no se pudo hacer porque en algún momento había que limitar el alcance de este Proyecto.

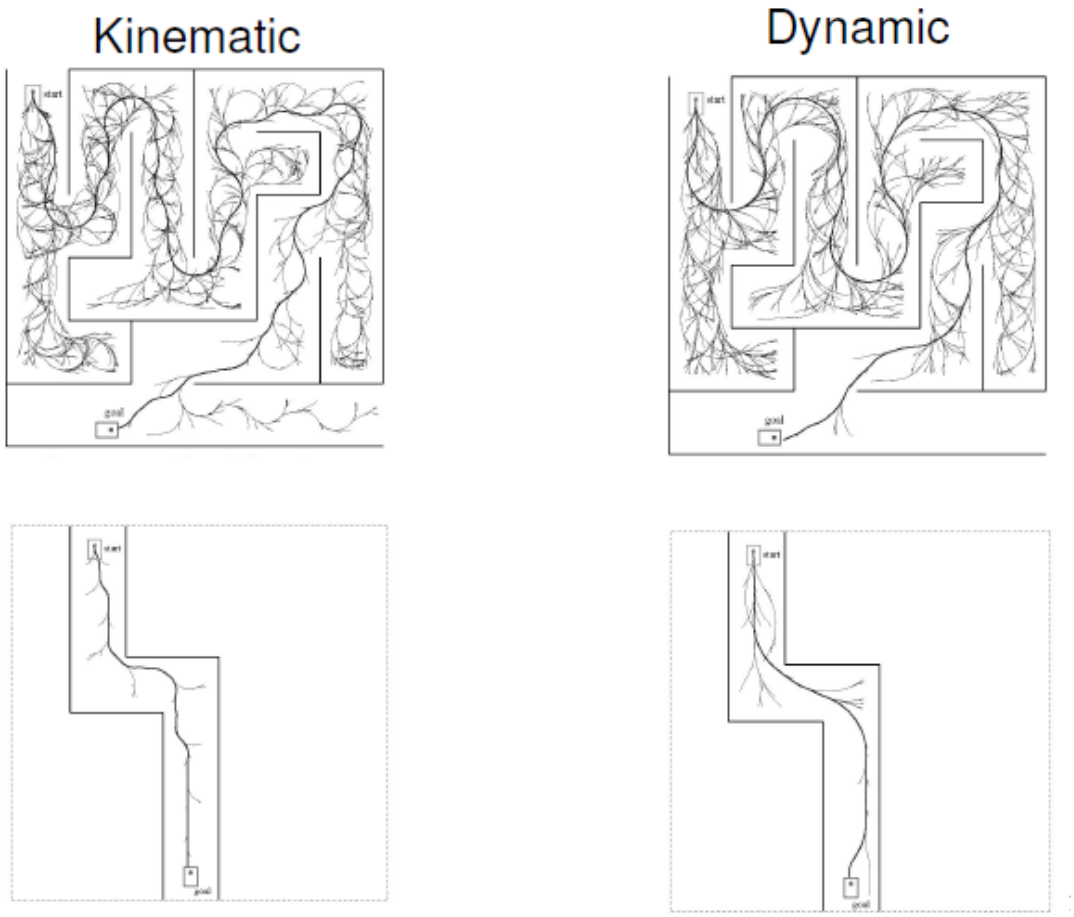

Figura 18. Comparación entre árboles RRT construidos con restricciones cinemáticas y dinámicas. [39]

4 MODELADO DEL ENTORNO EN MATLAB

Como ya se explicó en el objeto del Proyecto, para poder brindar resultados cuantitativos de cada uno de los algoritmos y poder reutilizar el código de este Proyecto para otros fines distintos, se optó por agrupar los *algoritmos* en funciones MATLAB [2]. Se decidió el entorno MATLAB (MATrix LABoratory) por su lenguaje intuitivo, flexible, de gran optimización en el trabajo de matrices y su gran número de funciones ya implementadas. Esto se traduce en productividad, con menos esfuerzo se puede programar más.

Si bien eso es cierto, MATLAB tiene también sus puntos negativos. Su lenguaje de programación es interpretado, resultando mucho más lento que un lenguaje compilado como puede ser C++. Además, se trata de un entorno privativo caro (1.000-3.000 €), pesado (4-6 GB) y sin comunicación directa con robots. De esta forma, de querer llevarse los algoritmos aquí estudiados a la práctica con robots reales, sería más conveniente pasarlos de MATLAB a C++ en forma de nodo en ROS (Robot Operating System) [40].

La potencia de la agrupación en funciones radica en que el usuario que las utiliza se abstrae totalmente de lo que contiene, solo necesita saber sus entradas y sus salidas. La función se llama y devuelve la información necesaria sin alterar las variables de su WorkSpace. Dichas funciones que contienen los *algoritmos* a su vez llaman a otras tantas funciones más específicas que se irán explicando poco a poco a medida que se vayan usando más adelante.

Las soluciones planteadas solo son válidas para *robots holónomos*, un UAV real por ejemplo no sería capaz de coger las curvas igual que estas trayectorias porque tienen restricciones cinemáticas (por ejemplo ángulos máximos de giro) y dinámicas (masa, inercia, etc.). Las trayectorias necesitarían un *postprocesado* cinemático o dinámico, lo cual se puede aplicar posteriormente de forma independiente (fuera del alcance de este Proyecto).

4.1 Representación del entorno.

Debido a la gran potencia de MATLAB tratando con matrices, se ha decidido representar el mapa de obstáculos o modelo del entorno como una matriz. Dicha matriz, para el caso plano será de dos dimensiones y para el caso tridimensional de tres.

Se considerará que una *celdilla está libre de obstáculos si tiene valor 0, y se considerará que tiene un obstáculo si vale 255*. La justificación de estos valores está, en que para el caso 2D se tratará como una imagen, facilitando muchísimo su manejo (la lectura de un modelo introducido por ejemplo es con *imread*, representación con *imshow*, expansión de obstáculos con una *acreción con imdilate*, etc.) y en imágenes el 0 representa el negro y el 255 el blanco (8 bits de color monocromo).

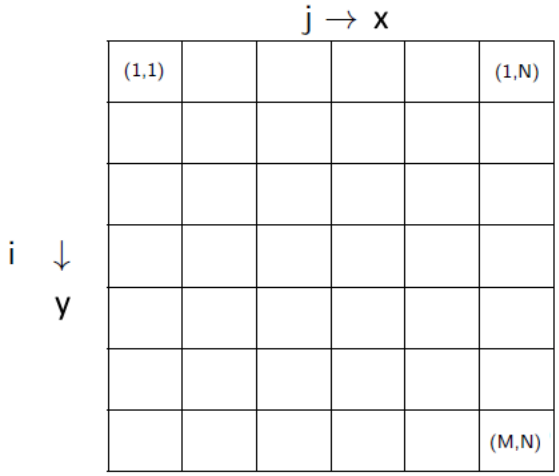

Figura 19. Convención de coordenadas en MATLAB. [41]

La convención de coordenadas dentro de la matriz es la misma que la que toma MATLAB para las imágenes. La ordenada "y" se corresponde con las filas de la matriz comenzando en el 1 desde arriba, dirección creciente hacia abajo. La coordenada "x" se corresponde con las columnas, comenzando en la izquierda, dirección creciente hacia la derecha.

En el caso 3D, aunque los *algoritmos* son fácilmente extrapolables, fue complicado de implementar, debido a la deficiente visualización de sólidos tridimensionales a partir de matrices con MATLAB. La función *imshow* que en el caso plano resulta excelente para visualizar matrices como imágenes, no existe como tal para matrices tridimensionales. Existe la función *imshow3Dfull* que muestra los sólidos por secciones o capas, lo cual está lejos de la visión 3D en perspectiva deseada. Así, la solución escogida fue trocear los sólidos 3D en capas y representar capa a capa los sólidos con la función *surf*. Esta solución, sin ser perfecta, permite visualizar todo tipo de mapa de obstáculos y desde cualquier perspectiva.

Para todos los puntos consideraremos siempre que el primer elemento del vector es la "x", el segundo la "y", y de existir, el tercero (caso tridimensional) sería la "z". La convención del origen de coordenadas de los puntos será el mismo que el del modelo.

Otro aspecto muy importante necesario de explicar es la clasificación del árbol dentro del entorno según el *mallado del entorno*. Como ya se verá, los algoritmos de planificación probabilística necesitan hacer algunos cálculos exhaustivos dentro del árbol, cálculos vitales en importancia, tales como el cálculo del nodo más cercano a un determinado punto de muestreo. El mallado del entorno clasifica los puntos del árbol según su ubicación y lo registra en la matriz "mallado", por lo que cuando se requiera hacer dichos cálculos exhaustivos el número de consultas se convierte en una fracción del total.

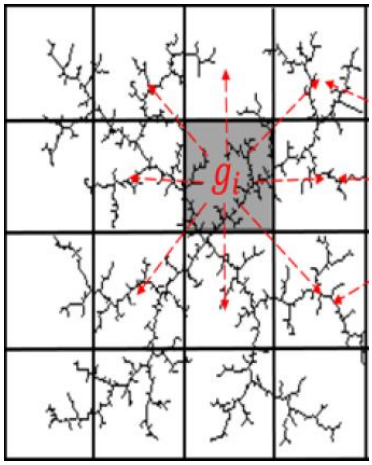

Figura 20. Mallado en el entorno que registra cada nodo del árbol en una región. [12]

Este uso de mallado reduce el tiempo de ejecución de los algoritmos probabilísticos de forma muy considerable, si el tamaño del mallado está bien ajustado al problema. Para poder cuantificar la mejora de tiempo con mallado respecto a sin mallado se probó muchas veces a resolver el laberinto de la siguiente página con el algoritmo combinado Informed RRT*-Smart Connect (RRT-Connect en la fase de búsqueda del camino inicial, Informed RRT* y RRT*-Smart en la de optimización). Después se calculó la media de dichos tiempos, arrojando estos resultados:

Iteraciones en fase de optimización	Tiempo sin mallado	Tiempo con mallado
1000	6	1
2000	24	4,25
3000	51	8,4
4000	91	15

Tabla 1. Para igualdad de iteraciones (y optimización de distancia) el uso de entorno con mallado bien ajustado al problema reduce el tiempo de ejecución alrededor de 6 veces (de media), al menos en este problema.

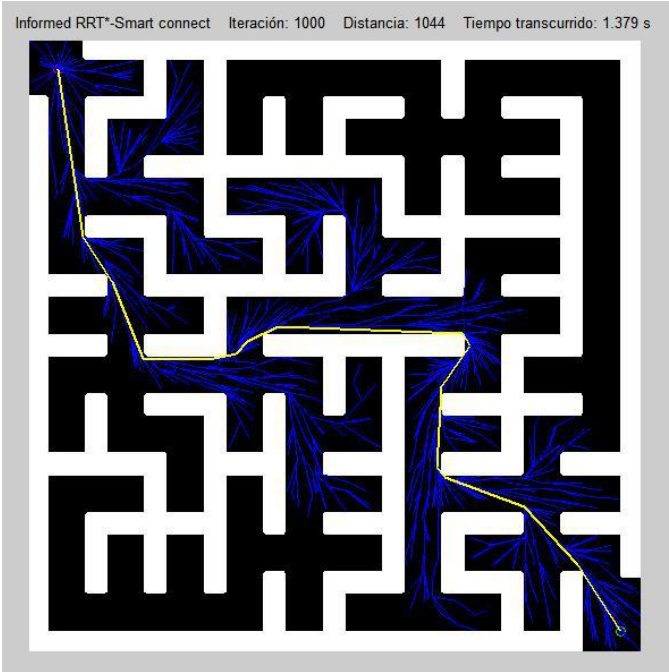

Figura 21. Laberinto usado en la comparación entre tiempos con mallado y sin mallado. La captura se corresponde con la situación con mallado.

4.2 Entornos utilizados.

Para su validación, se han empleado los algoritmos de planificación aquí estudiados en distintos tipos de entornos. A continuación se presentarán algunos de los más empleados para el caso 2D, explicando después el caso 3D.

Algunos entornos son generados *aleatoriamente*. Para ello se fija aleatoriamente el tamaño y número de rectángulos, que representarán los obstáculos, distribuyéndolos al azar por el mapa y expansionándolos posteriormente para tener en cuenta las dimensiones del robot. Tienen la ventaja de que ponen a prueba el planificador ante entornos muy dispares y masificados. Evidentemente, para este tipo de mapas inciertos, el algoritmo RRT multi-query + Dijkstra no vale (no existe grafo preexistente), así que el más adecuado sería un RRT-Connect. En las siguientes imágenes se muestra este tipo de entornos resueltos con RRT a secas.

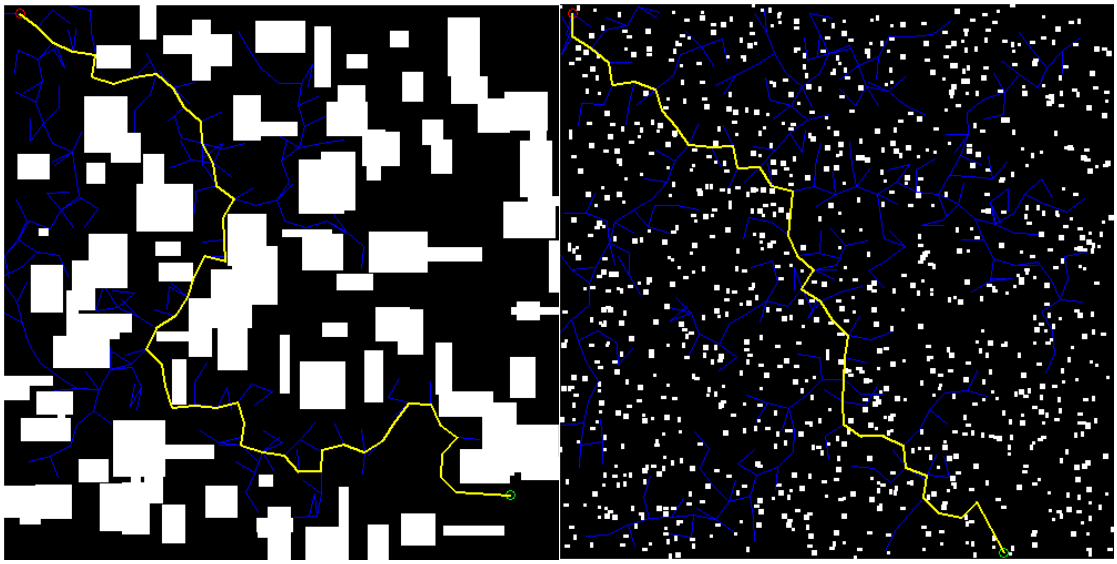

Figura 22. Ejemplos de entornos generados aleatoriamente.

Otros entornos probados son *laberintos* de complejidad variable. Los laberintos son entornos ricos en mínimos locales, por lo que un algoritmo que los resuelva demostrará ser robusto ante la influencia de mínimos locales:

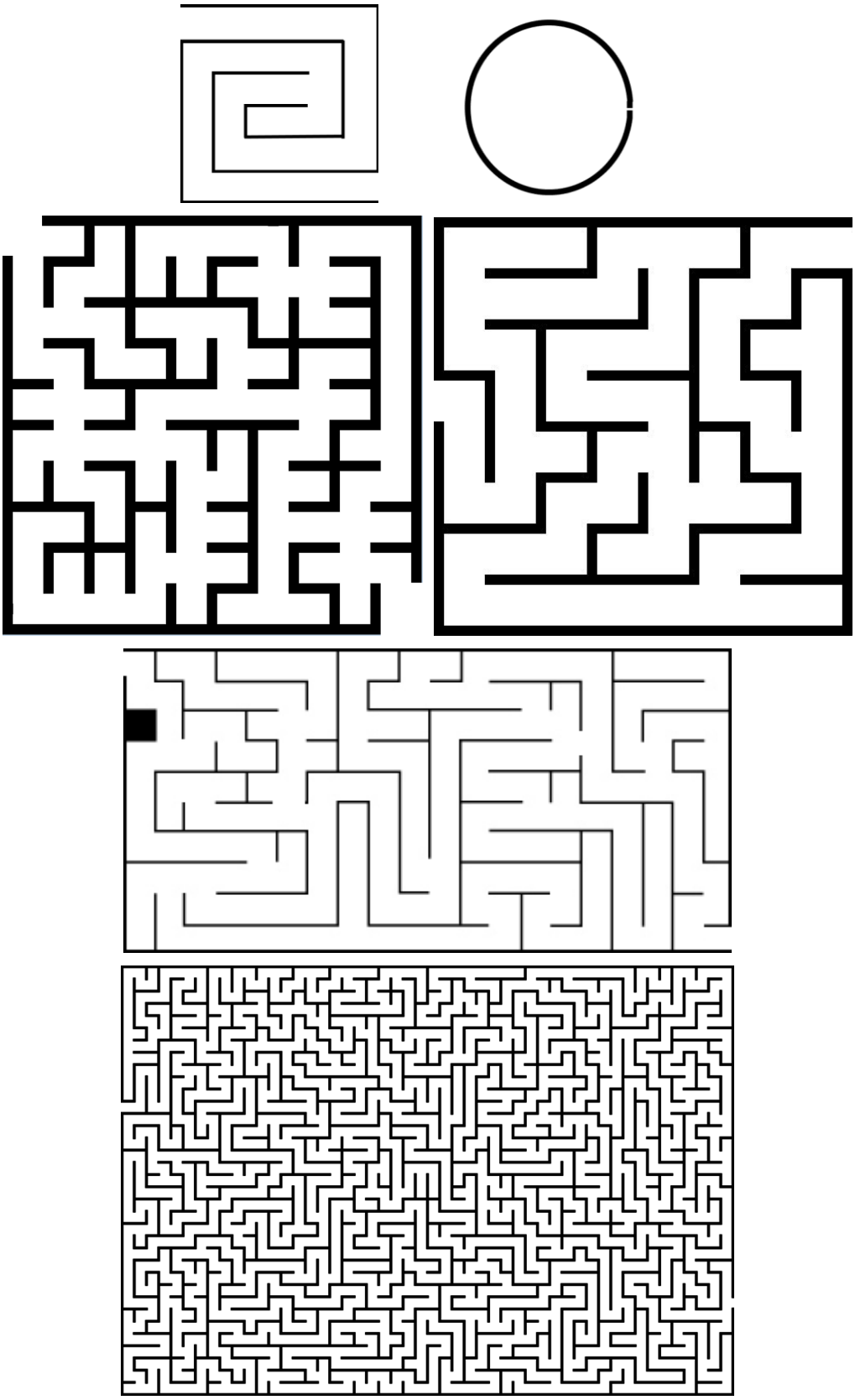

Figura 23. Laberintos utilizados, de menor a mayor dificultad.

Los dos primeros de ellos buscan la representación gráfica evidente de los mínimos locales, es por eso que el primero se usará en el siguiente capítulo para explicar la influencia de los mínimos locales. El último laberinto, el laberinto imposible, solo es resoluble mediante el uso de un grafo preexistente y aplicando un algoritmo de búsqueda en grafo, tal como el algoritmo de Dijkstra.

Aunque los anteriores entornos sirven perfectamente para la validación de los algoritmos de planificación, parecen bastante irreales, en el sentido de que en la vida real las personas raramente nos topamos con entornos laberínticos o tan masificados. Quizás en la Feria de Abril de Sevilla, Semana Santa, o en el Laberinto de Alicia de Disneyland Paris.

Con el fin de que el lector de esta memoria se sienta familiarizado con los mapas resueltos por los algoritmos, en este Proyecto se han usado también los *planos de la ETSI* (Escuela Técnica Superior de Ingeniería) de Sevilla.

Para ello la tarea no fue fácil, ya que hubo que encontrar los planos adecuados de cada planta, y tratarlos con un editor de imágenes hasta llegar a tener los mapas de obstáculos binarizados aquí presentados. En el proceso se descartaron muchas zonas porque podrían resultar inaccesibles para un robot móvil (tales como escaleras), otras se descartaron porque están cerradas al público, y otras tantas simplemente por simplificar.

Es en este tipo de mapas realistas en los que los métodos de planificación probabilísticos destacan frente a otros, ya que su geometría es compleja y arbitraria en muchos casos, con mínimos locales en todas partes.

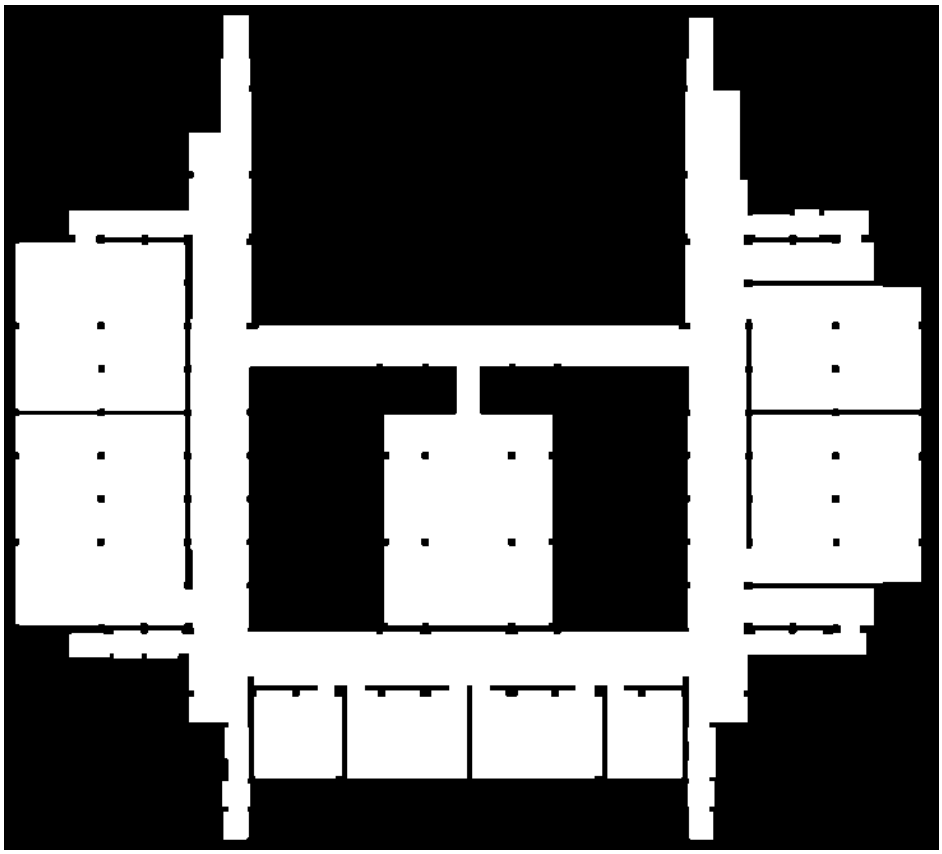

Figura 24. Planta Sótano de la ETSI de Sevilla.

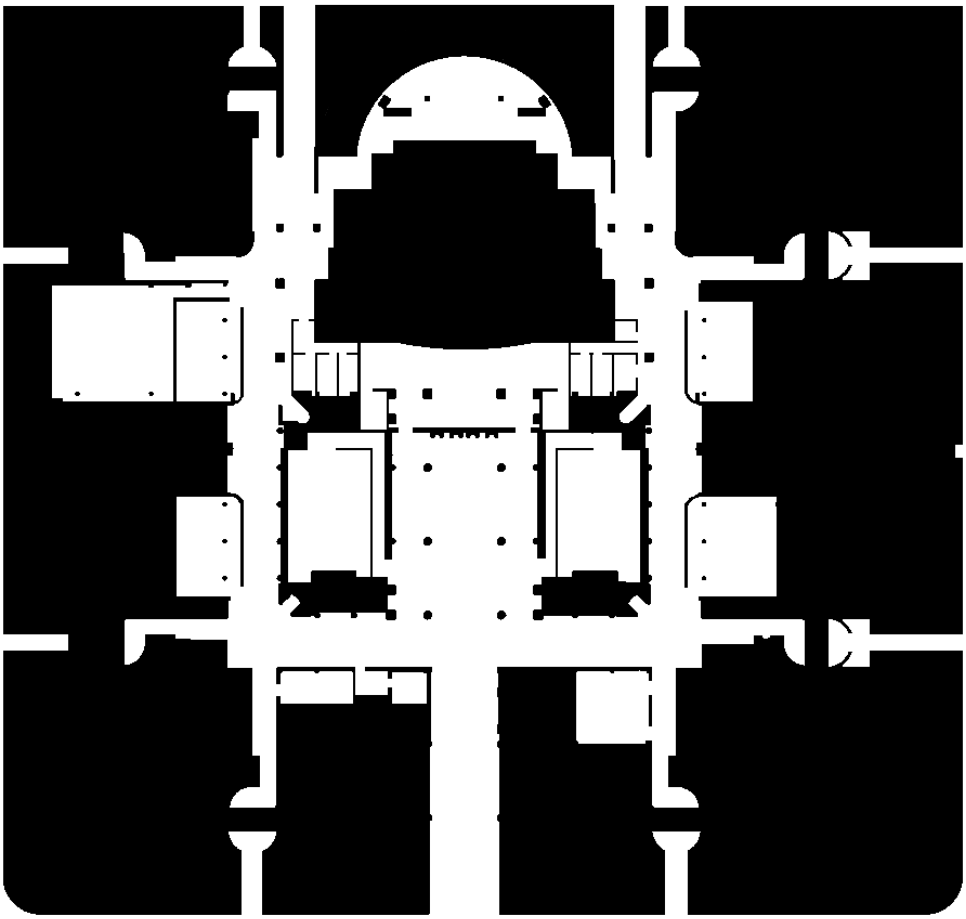

Figura 25. Planta Baja de la ETSI de Sevilla.

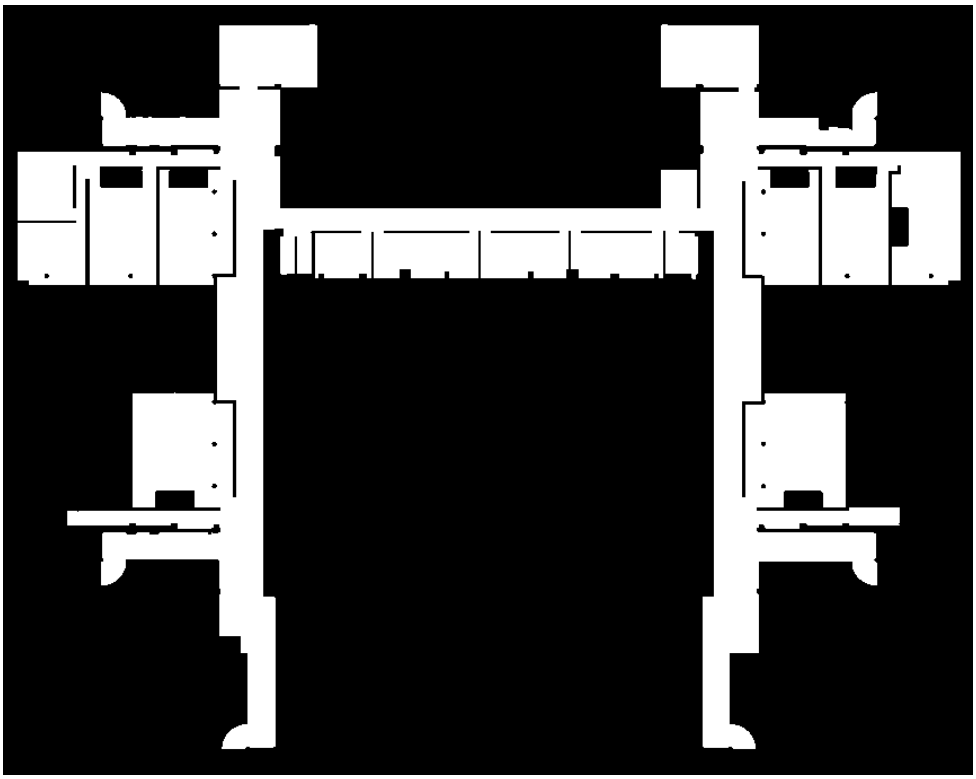

Figura 26. Entreplanta 1 de la ETSI de Sevilla.

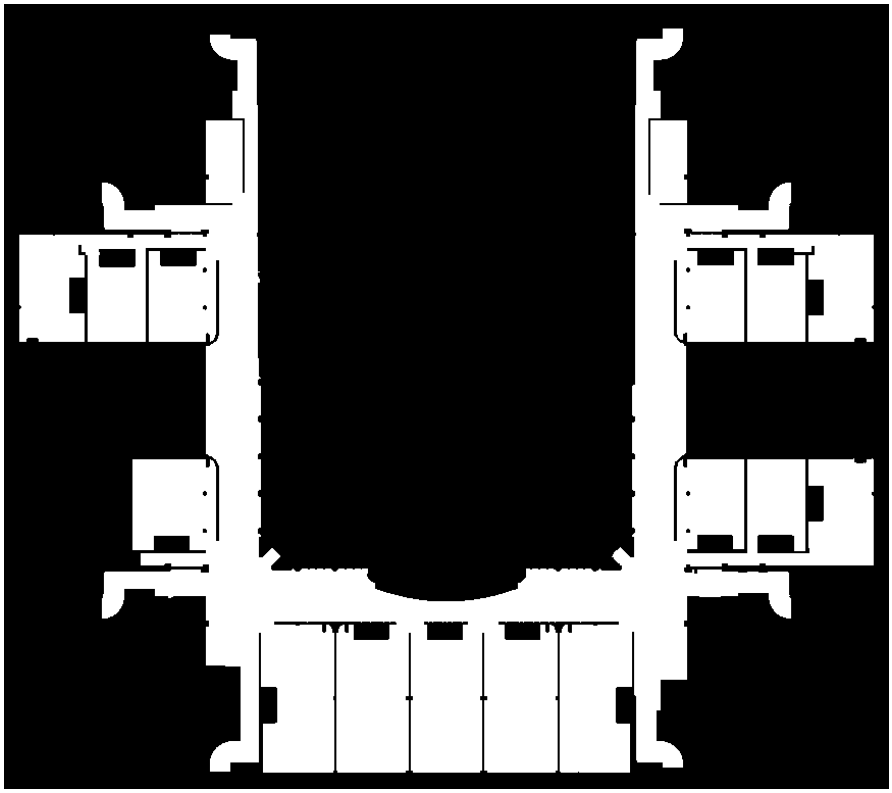

Figura 27. Primera Planta de la ETSI de Sevilla.

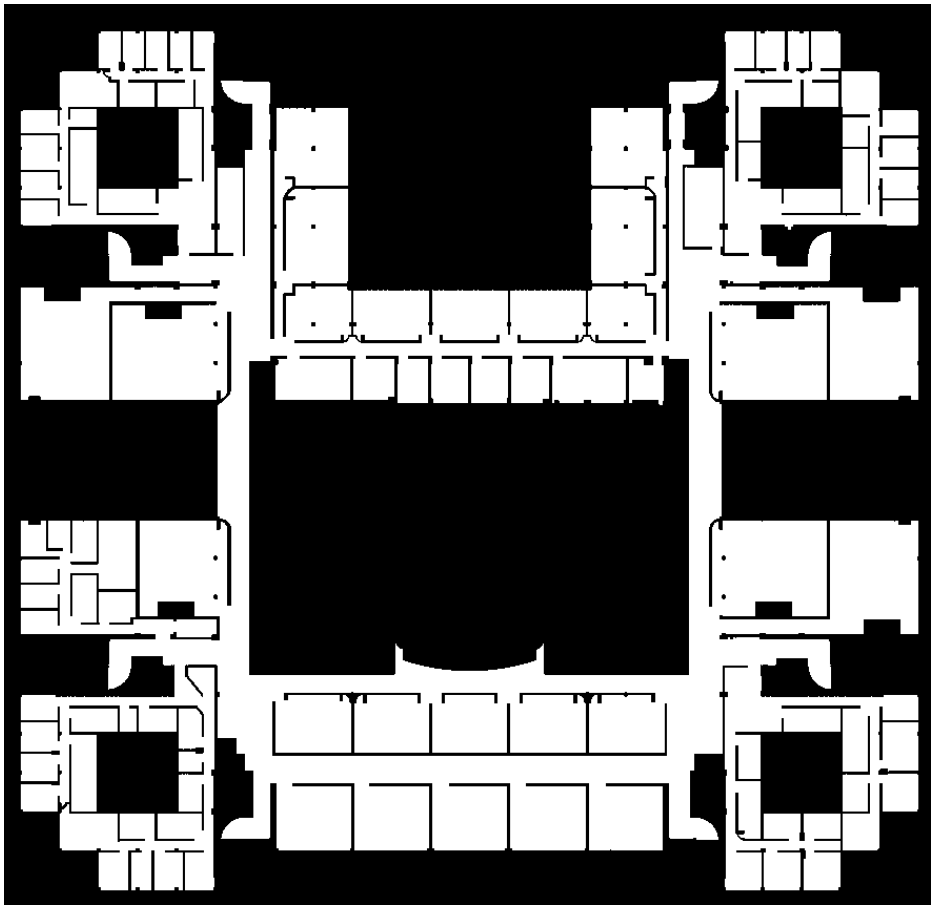

Figura 28. Entreplanta 2 de la ETSI de Sevilla. Este plano fue al que más detalle se le dio, al albergar el Departamento de Ingeniería de Sistemas y Automática.

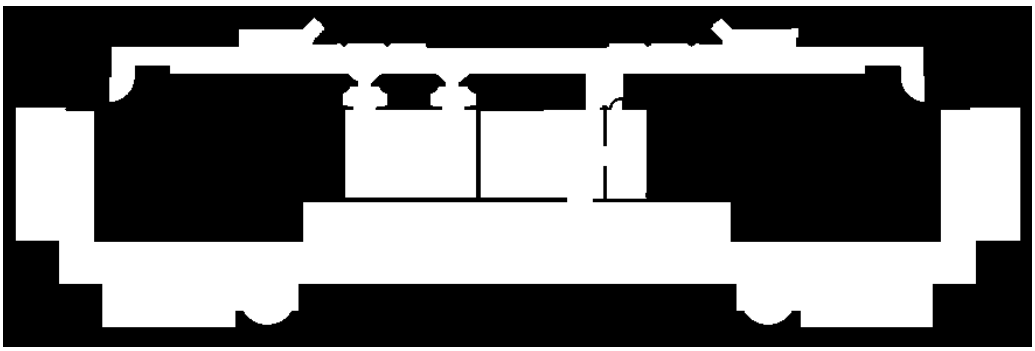

Figura 29. Planta Ático.

Como ya se ha comentado anteriormente, resolver la fase de búsqueda de primer camino inicial combinando búsqueda probabilística con búsqueda en un grafo preexistente resulta mucho más rápido y eficiente. Es por eso que, si los obstáculos son estáticos (como las paredes y columnas de los edificios), merece la pena el esfuerzo de construirse un grafo preexistente. Los nodos de este grafo deben estar estratégicamente distribuidos, de forma que con el mínimo número de nodos se facilite la resolución de todos los mínimos locales.

A continuación explicaré la construcción de dichos grafos preexistentes. Primero, con un editor de imágenes, se marcaban de un determinado color (verde, por ejemplo) aquellos píxeles donde se deseaba un nodo (recordar que su ubicación debe ser estratégica). Después en MATLAB se recorrían todos los píxeles, guardando los puntos verdes como nodos del grafo. Con dichos nodos se construía el roadmap por visibilidad cercana, anotándose los costes de los arcos. Por último, dichos grafos preexistentes se guardan junto al mapa de obstáculos para tenerlos disponibles siempre, sin tener que recalcularlos continuamente.

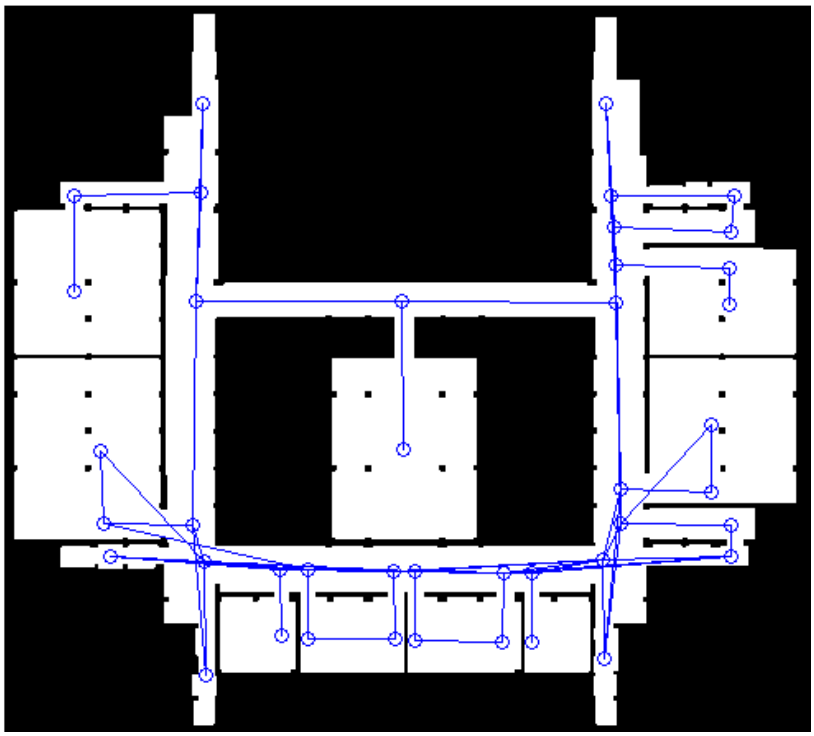

Figura 30. Grafo preexistente en la Planta Sótano.

Finalmente, se presentarán también los mapas de obstáculos usados para la validación de algoritmos 3D. En 3D el mapa de obstáculos se compone de una matriz tridimensional, en la que de igual forma, un elemento de la matriz con valor de 255 significa presencia de obstáculo y un valor de 0 significa su ausencia. Se han usado dos tipos de mapas:

- *Paredes con perforaciones aleatorias*.

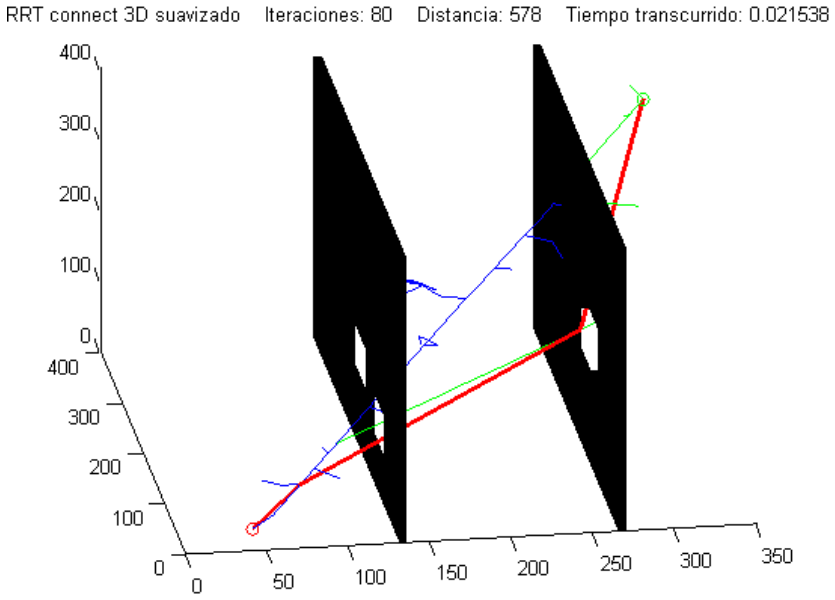

Figura 31. Entorno de 2 paredes con 2 agujeros dispuestos aleatoriamente (resuelto por RRT-Connect 3D).

- *Asteroides prismáticos aleatorios.*

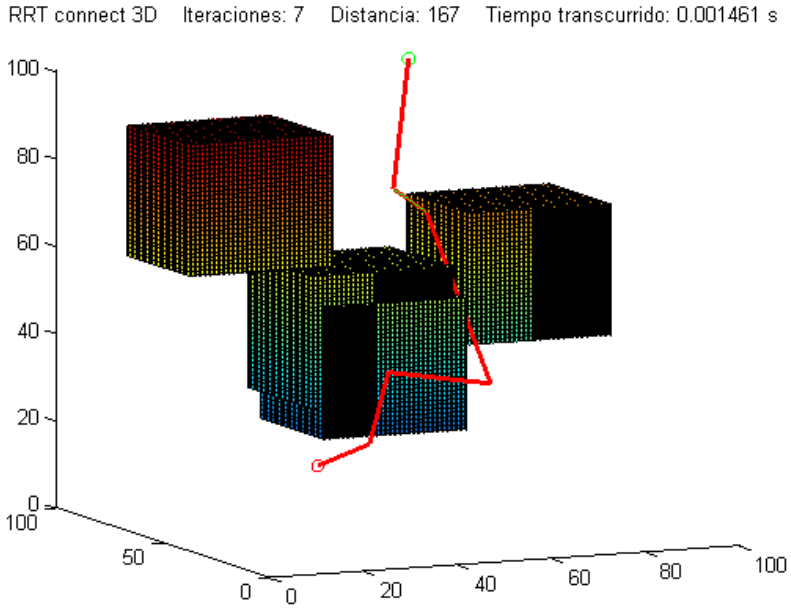

Figura 32. Entorno de varios asteroides pequeños a bordear (resuelto por RRT-Connect 3D).

5 ALGORITMOS DE BÚSQUEDA DEL CAMINO INICIAL NO ÓPTIMO EN ESTÁTICO

5.1 Algoritmo RRT

5.1.1 Introducción.

Fue presentado por primera vez en 1998 por Steven M. LaValle en su paper "Rapidly-Exploring Random Trees: A New Tool for Path Planning" [38].

El *algoritmo RRT* es un *algoritmo de planificación probabilístico subóptimo*, basado en modelo estático que construye un único *grafo unidireccional* en forma de *árbol*, este parte desde el punto inicial y se expande por todo el entorno de trabajo mediante un proceso de muestreo en el que busca puntos aleatorios hasta llegar al punto final, momento en el que se detiene.

A continuación se explica más detalladamente dicho procedimiento gracias al *pseudocódigo* que se puede encontrar en la página oficial de dicho algoritmo, creada y mantenida por el propio Steven M. LaValle, creador del RRT. [1]

Algorithm BuildRRT

Input: Initial configuration q_{init}, q_{end}, number of vertices in RRT K, incremental distance Δq

Output: RRT graph G

G.init(q_{init})

for $k = 1$ **to** K

$q_{rand} \leftarrow$ RAND_FREE_CONF()

$q_{near} \leftarrow$ NEAREST_VERTEX(q_{rand}, G)

$q_{new} \leftarrow$ NEW_CONF(q_{near}, q_{rand}, Δq)

G.add_vertex(q_{new})

G.add_*edge*(q_{near}, q_{new})

if near(q_{new}, q_{end})

G.add_node(q_{end})

G.add_parent(q_{new}, q_{end})

break

return G

El *algoritmo* recibe como entradas los parámetros del planificador: configuraciones (puntos) inicial y final, número de nodos máximos (iteraciones máximas del bucle *for*) y el salto entre nodos (distancia en píxeles que medirán las ramas del árbol). Como salida devolverá el árbol en sí.

Inicialmente introduce en el árbol el punto inicial, que será el primer nodo o *raíz*. Después iterará como máximo K veces si no se encuentra resultado. En cada iteración busca un punto cualquiera en el espacio libre de obstáculos y su nodo más cercano del árbol. Con esos dos puntos, en la misma recta que une ambos calcula un tercer punto q_{new} a una distancia Δq de q_{near} en dirección a q_{rand}. Si entre q_{near} y q_{new} no hay colisiones con obstáculos, q_{new} se introduce al árbol asociándolo a q_{near} y teniendo a este como padre.

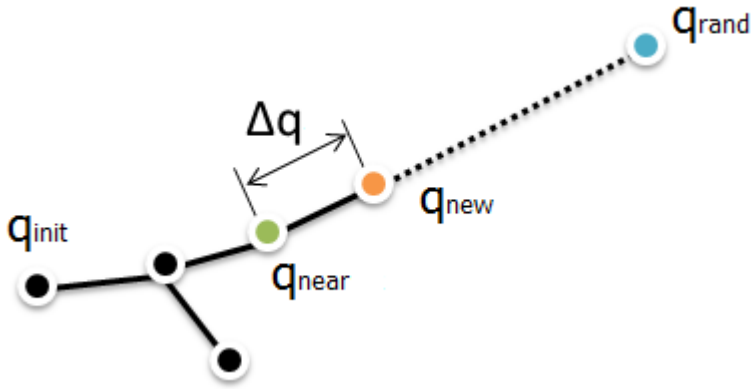

Figura 33. Fase de muestreo del algoritmo RRT [42].

El concepto de padre es importante comprenderlo, ya que es de principal importancia en la construcción de un árbol. El padre de un nodo es el nodo al que va unido en dirección hacia q_{ini}. Por definición, q_{ini} es el único nodo que no tiene padre. Un nodo solo puede tener un padre, mientras que puede tener varios hijos. Así, cada nodo se podría definir como un vector de 3 componentes (4 para el caso 3D), en el que las primeras componentes son las coordenadas y la última es el nodo padre.

Dicho esto, el algoritmo seguiría iterando hasta que se cumpla que q_{new} está cerca de q_{end}. Dicha "cercanía" se puede definir con un valor fijo o como un parámetro de entrada del planificador, incluso se puede prescindir de dicha condición de cercanía y poner una condición de visibilidad (más costosa computacionalmente). Si estuviera cerca y no hay colisión, añadimos q_{end} al árbol y acabamos el algoritmo.

5.1.2 Función implementada en 2D.

Se explicará brevemente la función implementada para el caso plano, la cual tiene esta forma:

```
trayectoria = RRT(mapa, P_ini, P_fin, muestra_animaciones, muestra_resultado,
Nmax, salto);
```

Como entradas tiene el mapa de obstáculos, siendo como ya se ha comentado una matriz bidimensional representativa del modelo estático del entorno. P_ini y P_fin los puntos iniciales y finales, muestra_ animaciones y muestra_resultado son variables booleanas para decidir si se desea o no representar gráficamente las animaciones y resultados respectivamente, Nmax el número máximo de iteraciones y salto la distancia de las ramas en píxeles.

Devuelve una matriz trayectoria cuyas filas son los nodos de la trayectoria suavizada encontrada (comenzando desde P_ini y acabando en P_fin) y las columnas las coordenadas de dichos nodos.

Este algoritmo resulta demasiado básico en la práctica, no habiendo ningún motivo para escoger este frente a otros tales como el RRT-Connect. También es por eso por lo que no se ha implementado mallado de entorno para acelerar su rendimiento cuando el número de nodos es alto.

La estructura interna de esta función servirá de guía para los siguientes algoritmos. No se entra a explicarla porque a grandes rasgos es lo mismo que el pseudocódigo en código MATLAB y no tiene gran sentido repetirlo en la memoria. Además el código viene perfectamente comentado, si se desea profundizar en él se recomienda verlo directamente desde él mismo con sus comentarios.

Para algunas operaciones, tales como el muestreo en el espacio libre, la búsqueda de nodos cerca, la comprobación de colisiones y el dibujar los resultados se han creado funciones a su vez para facilitar su reusabilidad de código entre algoritmos similares.

5.1.3 Postprocesado de desigualdad triangular.

Como ya se ha comentado con anterioridad, los caminos generados con este algoritmo son subóptimos, siendo tortuosos y oscilantes, con muchos giros e incluso rotaciones sobre sí mismos. Esto se debe a la naturaleza puramente aleatoria de la construcción del árbol.

Para mejorar esto de forma barata computacionalmente se suele recurrir al postprocesado de trayectoria por desigualdad triangular, esto es un *postprocesado* que se aplica al camino una vez se ha encontrado. Consiste en unir cada nodo trayectoria con el más lejano visible. Sigue sin ser óptimo, pero al menos se corrigen las oscilaciones y giros sobre sí mismo sin sentido alguno, minimizando bastante la distancia final. Al ser un proceso eficaz y rápido, este *postprocesado* se aplicará no solo a la trayectoria solución de este *algoritmo*, sino a la trayectoria arrojada por el resto de algoritmos estudiados en este Proyecto (que no sean de tiempo real).

El tiempo que se tarda en aplicar un postprocesado de desigualdad triangular es muy variable, aunque suele rondar el orden de varios milisegundos. La reducción de distancia depende en gran medida del algoritmo de planificación empleado.

En el apartado 5.1.5 de "Resultados" se pueden ver algunos ejemplos de trayectorias antes y después del postprocesado.

5.1.4 Problema de mínimos locales.

Los planificadores probabilísticos en general, se comportan bien ante mínimos locales, siendo el RRT de los mejores en este aspecto. Muchas veces se los prefiere antes que a los planificadores de campos de potencial por esto mismo, de hecho el RRT ha sido usado comúnmente como *algoritmo secundario* para salir de los mínimos locales en los campos de potencial.

Esto se debe a su *completitud probabilística* (la probabilidad de encontrar una solución subóptima tiende a 1 a medida que el árbol se expande). El porqué de esta propiedad es lo que se explicará a continuación. Se puede comprobar haciendo un diagrama de Voronoi de los nodos del *árbol* en construcción, como se observa en las siguientes imágenes, que las regiones más grandes en área del Voronoi son las zonas donde es más probable que caiga un punto muestreado al azar. Dichas regiones marcan la dirección más probable de expansión del árbol, forzando al árbol a explorar lo más lejano del espacio. Con el tiempo, dichas regiones más grandes del Voronoi desaparecen, regulando la expansión del *árbol* y haciendo que aumente su densidad de ramas. [13]

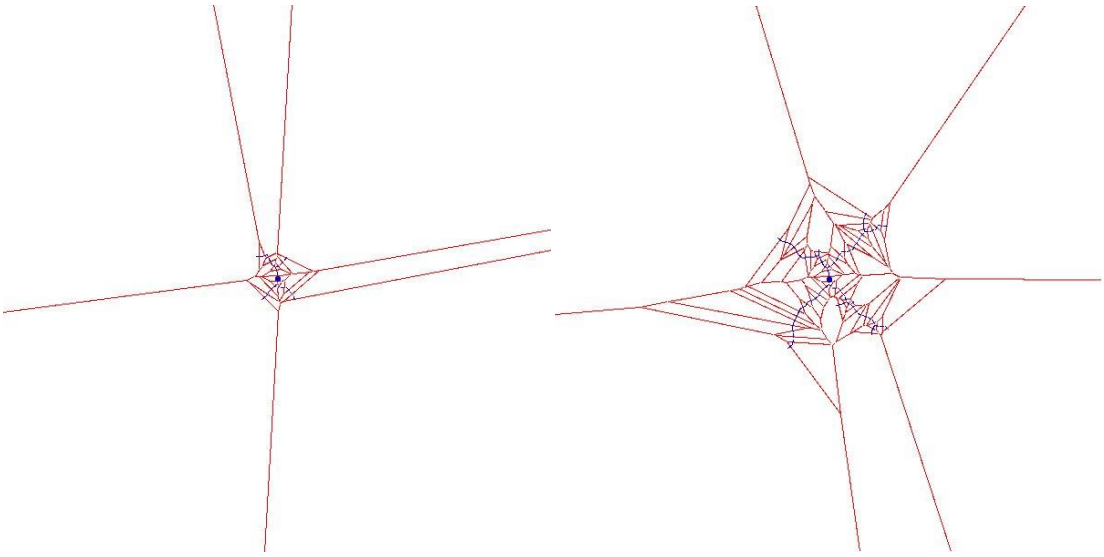

Figura 34. Diagrama de Voronoi de los nodos de un árbol RRT en expansión (1º y 2º). [13]

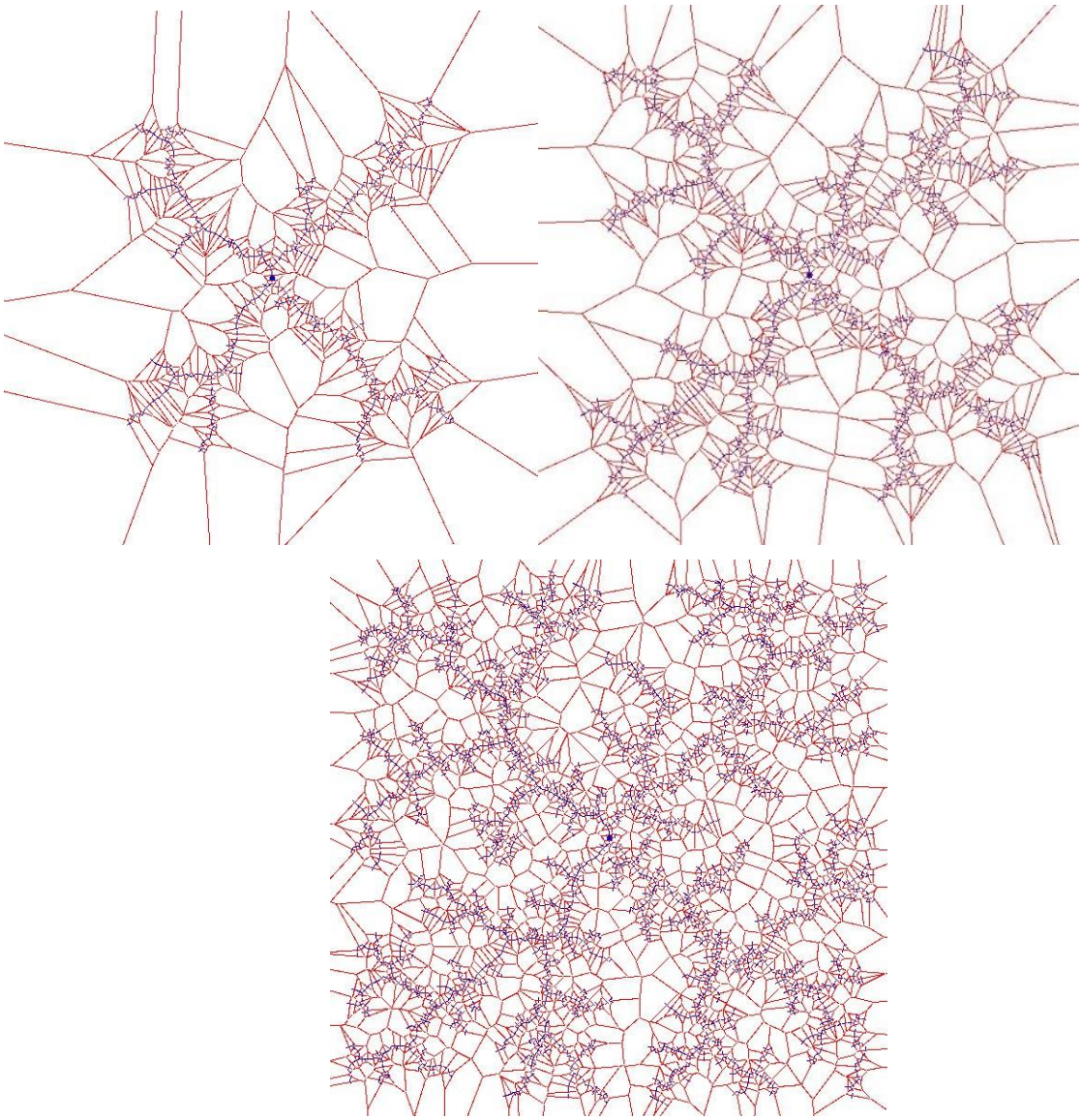

Figura 35. Diagrama de Voronoi de los nodos de un árbol RRT en expansión (3º, 4º y 5º). [13]

Pese a las ventajas de los RRTs al lidiar con mínimos locales, estos siguen existiendo en el entorno, y pueden ralentizar significativamente el rendimiento del planificador. Por la naturaleza de los RRT, normalmente acaban escapando de la trampa, pero con una penalización en tiempo de ejecución y recursos del sistema que interesa evitar.

No todos los RRTs se comportan igual ante mínimos locales. El *algoritmo orientado a fin* o *RRT Goal-Bias* consiste en que cada cierto número de iteraciones, fijado por el programador, no se hace un muestreo con un punto aleatorio, sino que se muestrea con el punto final. Este algoritmo es más propenso a caer en mínimos locales si la frecuencia con la que se muestrea el punto final es demasiado elevada. El *RRT-Connect* sin embargo se comporta mucho mejor que el RRT tradicional ante mínimos locales.

Si bien hay algoritmos más propensos que otros a caer en mínimos locales, estos forman parte de los mapas del entorno, como el que se verá en las siguientes figuras. Para avanzar el árbol debe pasar por un camino relativamente estrecho de uno de los bordes, estando el resto de la imagen muy ramificada. Esto produce que en la fase de muestreo, al escoger un punto aleatorio, el que su nodo más cercano sea el que haría avanzar el árbol hacia el punto final es probabilísticamente muy improbable.

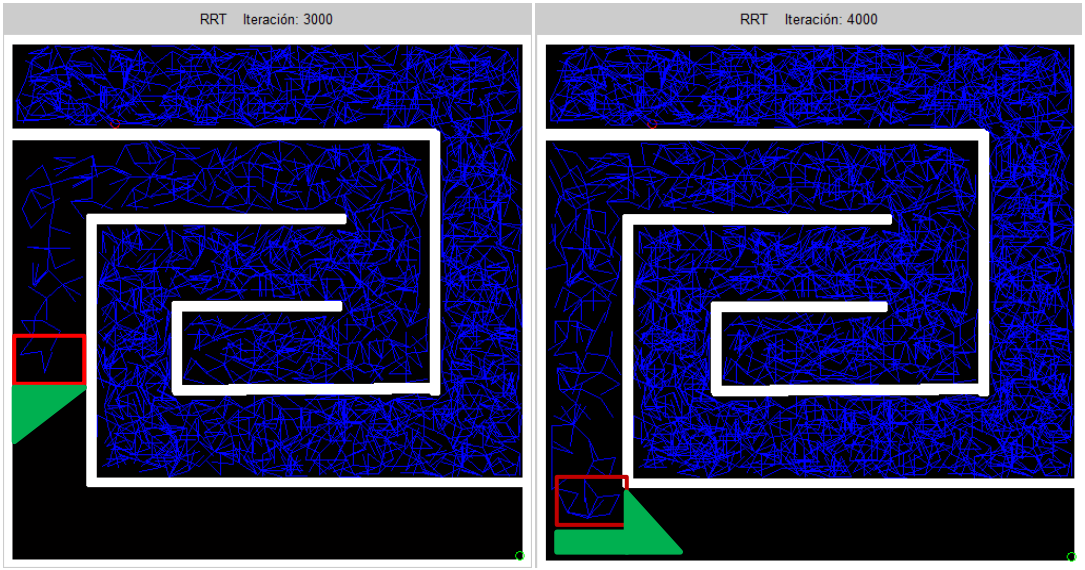

Figura 36. Mínimo local. La "superficie de avance" (en verde) en la que tiene que caer un punto muestreado para que el árbol avance es reducida, siendo improbable que caigan puntos ahí. Esto genera una expansión lenta del árbol.

Para que el árbol avance los puntos muestreados aleatoriamente deben caer demasiado cerca (área verde) de los nodos de avance del árbol (recuadro rojo). De caer el punto aleatorio en cualquier otro punto del mapa, este se uniría a zonas excesivamente ramificadas que no aportan nada a la resolución del problema.

Se puede ver a simple vista la poca probabilidad de que los puntos muestreados caigan en una zona tan chica de todo el mapa, probabilidad que sería incluso menor si fuera más chico el conducto. Es por eso que el árbol avanza muy lentamente en estas condiciones.

Con el tiempo los RRT suelen salir de estos mínimos locales, debido a que por naturaleza caen puntos aleatorios en la fase de muestreo por todo el espacio de trabajo, pero les resta muchísima eficacia y ralentiza mucho más de lo deseado. En el ejemplo anterior veíamos como para 4000 iteraciones el algoritmo estaba lejos de llegar al punto final, cuando los RRT suelen hacerlo por debajo de las 1000 iteraciones. Para solucionar esto surgen *algoritmos alternativos* principalmente basados en el uso de dos árboles, que abarcan el problema de forma doble.

El primer algoritmo que se plantea es el RRT bidireccional [43], que crecen dos árboles (uno desde el principio y otro desde el final) y van abarcando el espacio desconocido y buscándose entre ellos. Este algoritmo da buenos resultados, pero el escogido fue otro aún mejor basado en el bidireccional pero incorporando una cierta heurística voraz o *"greedy"*. Dicho algoritmo es el *RRT-Connect* [14] y será objeto del siguiente subcapítulo.

5.1.5 Resultados.

A continuación se muestran varios resultados obtenidos con este planificador, servirán para dar luz a posibles conclusiones y comparaciones entre planificadores. En todos ellos el salto de rama es de 20 píxeles y la dimensión de la imagen de 500x500, excepto la última imagen del laberinto que es 188x375. En cada gráfica el título muestra el algoritmo empleado y el número de iteraciones con el que ha acabado, la distancia de la trayectoria en píxeles y el tiempo de ejecución del programa. Recordar que si se hubiera implementado un mallado de entorno los resultados se arrojarían considerablemente más rápido.

Para tener referencia del hardware usado, todos los planificadores de algoritmos ejecutados en este Proyecto se han hecho en un portátil Asus con Windows 10 de 64 bits, CPU Intel Core i7 4510U, 6GB de RAM, gráfica NVIDIA GeForce 820M y versión de MATLAB R2013b.

Algoritmo	Iteración	Distancia (píxeles)	Tiempo (segundos)
RRT (izq.)	*3657*	2659	39,7341
RRT postprocesado (der.)	*3657*	2238	39,7474

Tabla 2. RRT vs RRT postprocesado.

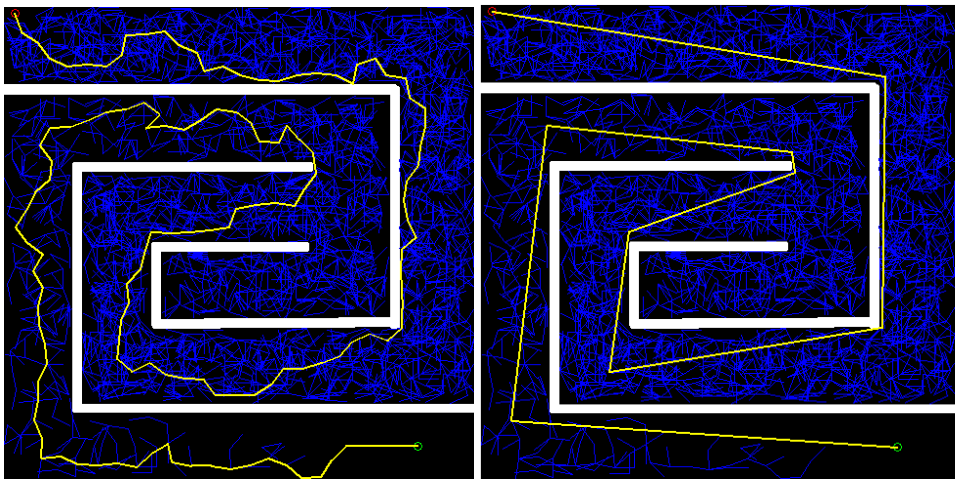

Figura 37. RRT normal vs postprocesado.

En esta imagen podemos ver cómo el algoritmo RRT ha llegado a resolver el escenario de mínimo local planteado en el apartado anterior. No olvidemos que, por la naturaleza del RRT, resulta un algoritmo azaroso, y es que lo que en el apartado anterior estaba lejos de llegar al final en 4000 iteraciones, en este ejemplo han bastado solo en 3657. El tiempo de ejecución del programa resulta inconcebiblemente lento para un método RRT no óptimo (39.7 s), los RRT subóptimos suelen converger en menos de un segundo, unos pocos a lo sumo. Si tarda más es indicativo de que el problema está dominado por mínimos locales y es conveniente usar otro algoritmo. A continuación se muestran más ejemplos de planificación resueltos con el algoritmo RRT:

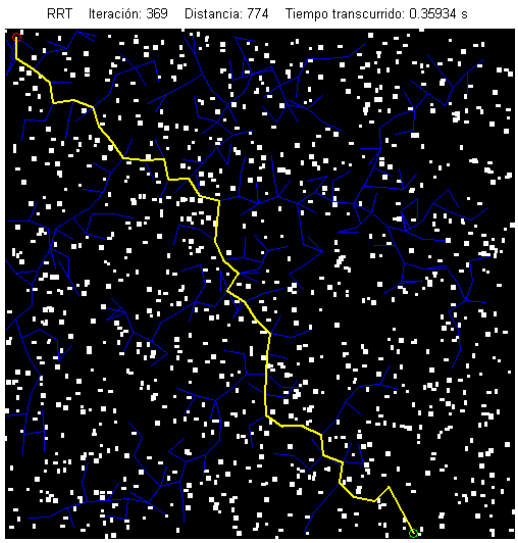

Figura 38. RRT sin suavizar en entorno aleatorio con muchos obstáculos diminutos.

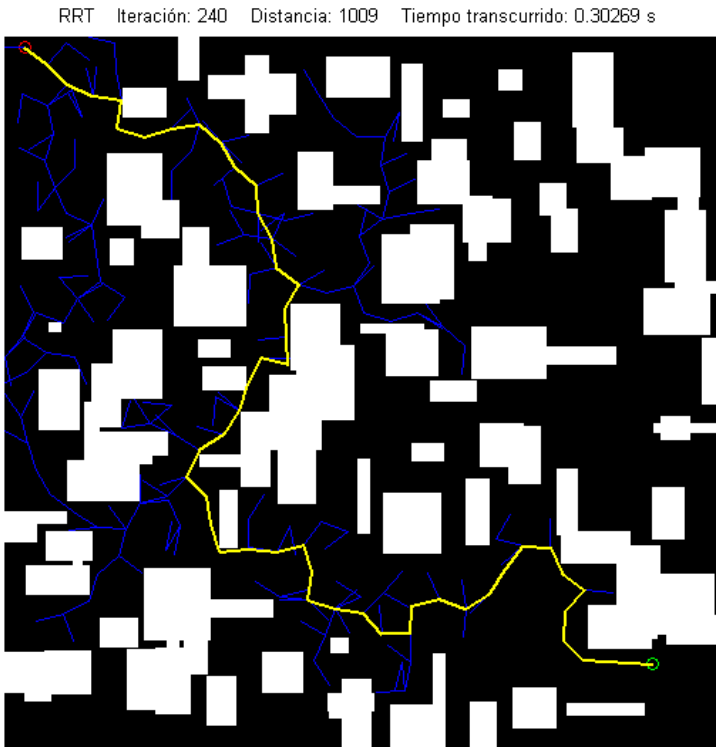

Figura 39. RRT sin suavizar en entorno aleatorio con obstáculos gruesos.

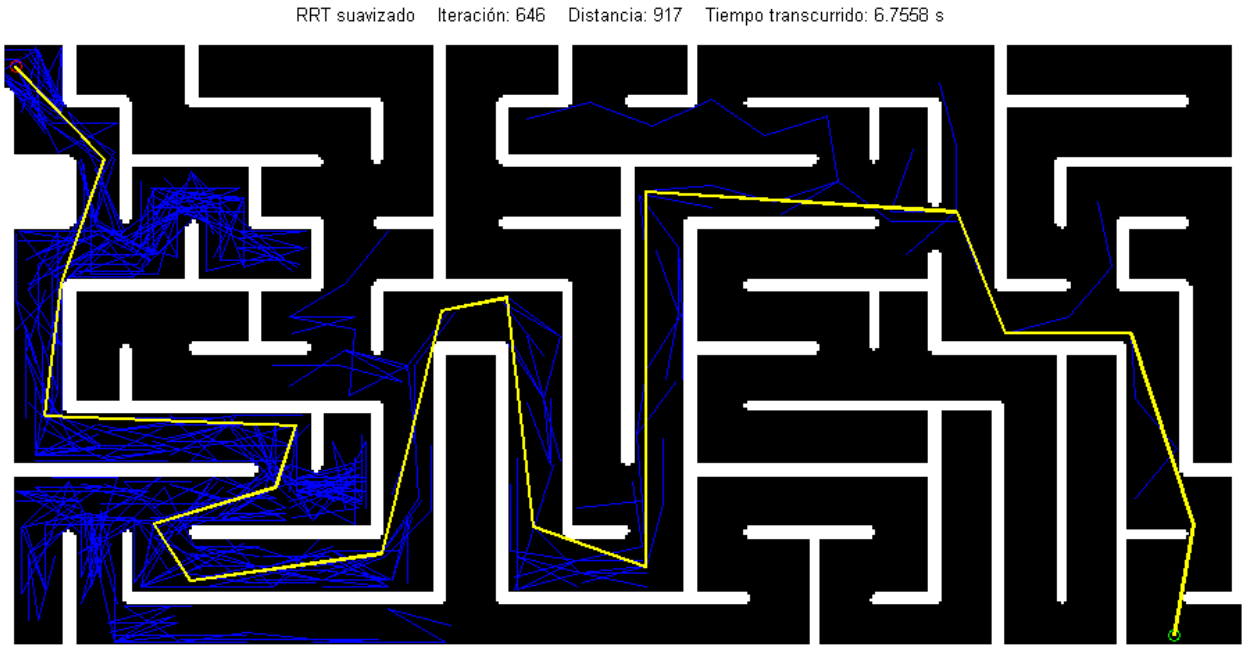

Figura 40. Algoritmo RRT tras postprocesado en un entorno laberíntico.

5.2 Algoritmo RRT-Connect

5.2.1 Introducción.

Como se adelantó en el subcapítulo anterior, este algoritmo surge por la necesidad de mejorar el RRT tradicional ante problemas de mínimos locales, siendo más rápido en prácticamente cualquier entorno.

Este algoritmo fue presentado por James J. Kuffner y Steven M. LaValle en "RRT-Connect: An Efficient Approach to Single-Query Path Planning" en el 2000 [14], surgiendo como un *algoritmo simple y eficaz* para resolver problemas con gran complejidad en la configuración de espacio. El método funciona creciendo incrementalmente dos árboles de exploración rápida, uno desde el inicio y otro desde el final. Los árboles no solo se expanden hacia el espacio que les rodea, sino que también se buscan entre ellos implementando una heurística voraz o *"greedy"*. Aunque originalmente se diseñó para controlar brazos robóticos antropomórficos (7-DOF), *el algoritmo* se ha conseguido aplicar con éxito en numerosos problemas de planificación de caminos.

A continuación puede verse el *pseudocódigo de dicho algoritmo*, del cual se comentan los conceptos principales.

```
CONNECT(T, q)
 1  repeat
 2      S ← EXTEND(T, q);
 3  until not (S = Advanced)
 4  Return S;
```

```
RRT_CONNECT_PLANNER(q_init, q_goal)
 1  T_a.init(q_init); T_b.init(q_goal);
 2  for k = 1 to K do
 3      q_rand ← RANDOM_CONFIG();
 4      if not (EXTEND(T_a, q_rand) = Trapped) then
 5          if (CONNECT(T_b, q_new) = Reached) then
 6              Return PATH(T_a, T_b);
 7      SWAP(T_a, T_b);
 8  Return Failure
```

Figura 41. Pseudocódigo del algoritmo RRT-Connect. [14]

Antes que nada se explica la función CONNECT, alternativa voraz del tradicional EXTEND del RRT. En vez de incorporar un solo punto nuevo al árbol, a una distancia dada del nodo más cercano del árbol al punto muestreado, CONNECT irá repitiendo la función EXTEND tradicional del RRT hasta que choca con algo o hasta que alcanza el punto muestreado. Como resultado de esta función heurística voraz o *"greedy"* se explora todo el espacio, mientras se van formando ramas largas que tratan de unir ambos árboles.

Explicado la función CONNECT, pasamos al planificador RRT_CONNECT_PLANNER, el cual, comienza inicializando los árboles inicial y meta, y durante K iteraciones busca un punto aleatorio como en un RRT normal. Si dicho punto muestreado se puede incorporar al árbol, desde ese nodo se busca el otro árbol hasta que lo encuentra o se choca con algo. Si se choca se cambia de árbol y continúa iterando. Si se alcanza el otro árbol acabamos.

5.2.2 Funciones implementada en 2D y 3D.

Debido a lo bien que responde este algoritmo ante diversos entornos, especialmente los dificultosos (como se verá en el siguiente apartado), ha sido escogido como uno de los extrapolados al caso tridimensional. La extrapolación al espacio resulta sencilla, solo la detección de colisiones y la visualización de sólidos en perspectiva tridimensional en MATLAB aporta realmente dificultad.

Se explicarán brevemente las funciones implementadas, las cuales tienen la siguiente forma:

```
[trayectoria, trayectoria_postprocesada] = RRT_connect(mapa, P_ini, P_fin,
muestra_animaciones, muestra_resultado, Nmax, salto, subdivisiones_M,
subdivisiones_N);

[trayectoria, trayectoria_ postprocesada] = RRT_connect_3D(mapa, P_ini,
P_fin, muestra_animaciones, muestra_resultado, Nmax, salto, subdivisiones_M,
subdivisiones_N, subdivisiones_Z);
```

Como entradas tiene el mapa de obstáculos, siendo como ya se ha comentado una matriz (bidimensional para el caso plano, tridimensional para 3D) representativa del modelo estático del entorno. P_ini y P_fin los puntos iniciales y finales, muestra_animaciones y muestra_resultado, son variables *booleanas* para decidir si se desea o no representar gráficamente las animaciones y resultados respectivamente, Nmax es el número máximo de iteraciones y salto la distancia de las ramas (en píxeles). Subdivisiones_M, N y Z son el número de subdivisiones del mallado de entorno para acelerar las búsquedas de nodo más cercano si hay demasiados nodos.

Devuelve dos matrices trayectoria, una primera sin suavizar y otra segunda suavizada. Esta más adelante nos será útil en algoritmos de planificación óptimos, en los que antes de comenzar a optimizar buscaremos el camino inicial con RRT-Connect. Como ya se sabe, para las matrices trayectoria sus filas son los waypoints de la trayectoria (comenzando por P_ini y acabando en P_fin) y las columnas las coordenadas de dichos nodos.

Al contrario que con RRT, en esta función RRT-Connect sí se ha implementado un mallado de entorno. Dicho mallado acelera la búsqueda cuando el número de nodos del árbol es elevado. *Para situaciones en las que se requiera expandir demasiado el árbol en la fase de búsqueda habría que considerar mejor el algoritmo RRT multi-query + Dijkstra, el cual se estudiará mejor en el siguiente subcapítulo.*

Como en los casos anteriores, no se entra en explicar dicha función, a grandes rasgos es lo mismo que el pseudocódigo en código MATLAB, y no tiene sentido repetirlo en la memoria. Además el código viene perfectamente comentado, si se desea profundizar en él se recomienda verlo directamente desde él mismo con sus comentarios.

5.2.3 Solución a mínimos locales.

Para demostrar que el algoritmo RRT-Connect se comporta mejor ante mínimos locales que el RRT tradicional haremos pruebas sobre entornos conflictivos y compararemos el rendimiento de ambos. Para comparaciones entre algoritmos más justas se ha desactivado el mallado de entorno del RRT-Connect. En todas las capturas el árbol azul se corresponde con el inicial, y el verde con el final. Comenzaremos con el que sirvió de ejemplo para demostrar el problema de los mínimos locales en el subcapítulo anterior.

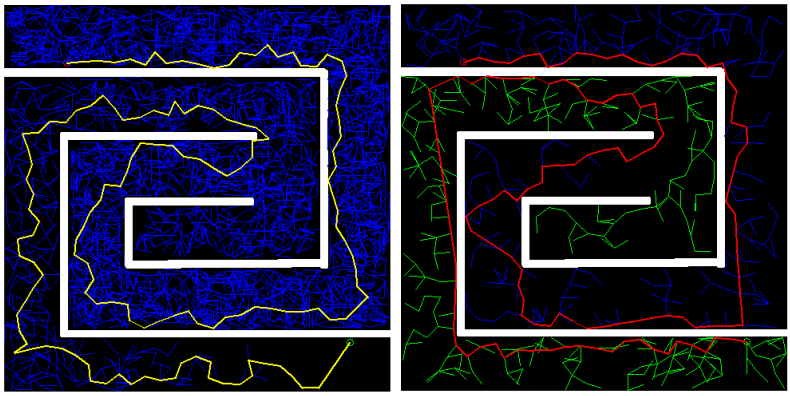

Figura 42. RRT vs RRT-Connect en entorno de mínimo local.

Algoritmo	Iteración	Distancia (píxeles)	Tiempo (segundos)
RRT (izq.)	*3956*	2867	40,088
RRT-Connect (der.)	*675*	2600	1,266

Tabla 3. RRT vs RRT-Connect en entorno de mínimo local.

A la vista de estos resultados la mejora es abrumadora con el RRT-Connect:

- Número de iteraciones 6 veces menor. Se exploran muchos menos puntos y el árbol se ramifica menos para llegar al mismo resultado.
- Distancia más o menos igual.
- Tiempo de ejecución 32 veces menor. Mucho más difícil caer en problemas de mínimos locales que ralenticen la convergencia.

Por supuesto, los números de mejora son dependientes del problema, no pudiendo afirmarse nunca que el tiempo de ejecución será siempre x veces menor. Habrá mapas en los que se comporten más o menos igual el RRT y RRT-Connect, aunque por lo general el RRT-Connect se comporta mejor en la mayoría de situaciones [14], que es lo que nos interesa a la hora de decidir entre uno y otro. Si hacemos más pruebas podemos ver mejoras en tiempo de ejecución en mayor o menor grado según la existencia o no de mínimos locales:

Figura 43. RRT postprocesado resolviendo un laberinto.

Figura 44. RRT-Connect postprocesado resolviendo un laberinto.

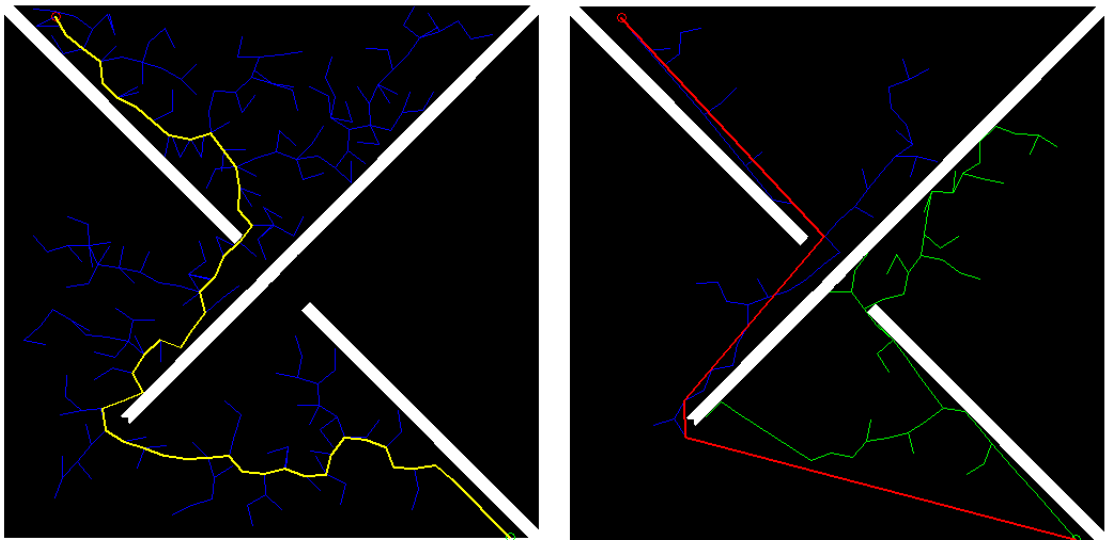

Figura 45. RRT vs RRT-Connect en entorno sencillo sin mínimo local.

Algoritmo	Iteración	Distancia (píxeles)	Tiempo (segundos)
RRT (izq.)	*269*	1042	0,2767
RRT-Connect (der.)	*125*	893	0,085327

Tabla 4. RRT vs RRT-Connect en entorno sencillo sin mínimo local.

5.2.4 Resultados.

A continuación probaremos el *algoritmo RRT-Connect en 3D* gracias a la función programada. Aunque se pueden resolver y representar gráficamente todo tipo de entornos, solo se han programado dos tipos de obstáculos aleatorios: entorno con paredes perforadas y con asteroides.

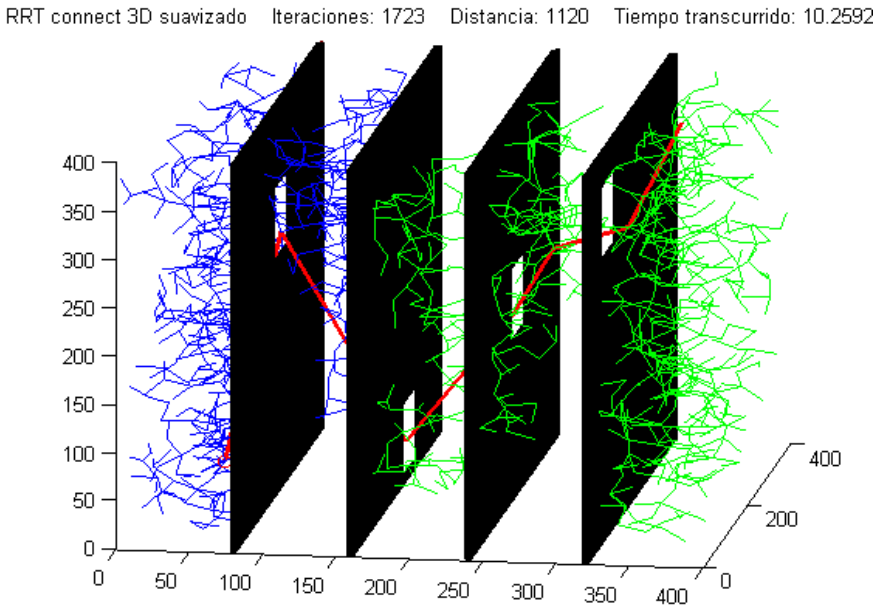

Figura 46. RRT-Connect (con postprocesado) 3D en un entorno de 4 paredes con 1 agujero dispuesto aleatoriamente.

Puede apreciarse cómo el planificador resuelve el problema en un tiempo bastante aceptable para la complejidad del problema. A continuación se muestran unas capturas del planificador resolviendo entornos con asteroides prismáticos, primero un solo asteroide muy grande que hay que bordear y después con varios asteroides más chicos. Estos entornos resultan más sencillos de resolver, consiguiéndolo en milisegundos.

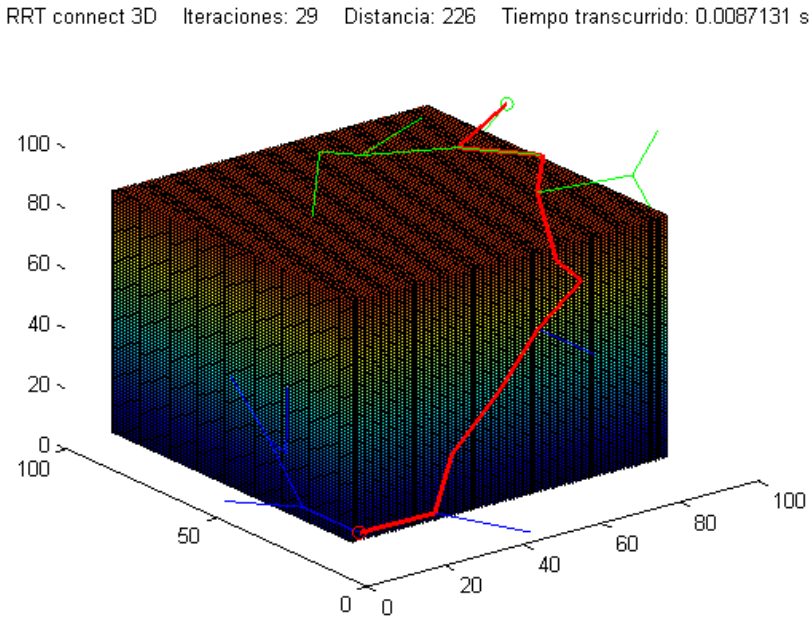

Figura 47. RRT-Connect (sin postprocesado) 3D resolviendo un solo asteroide muy grande a bordear.

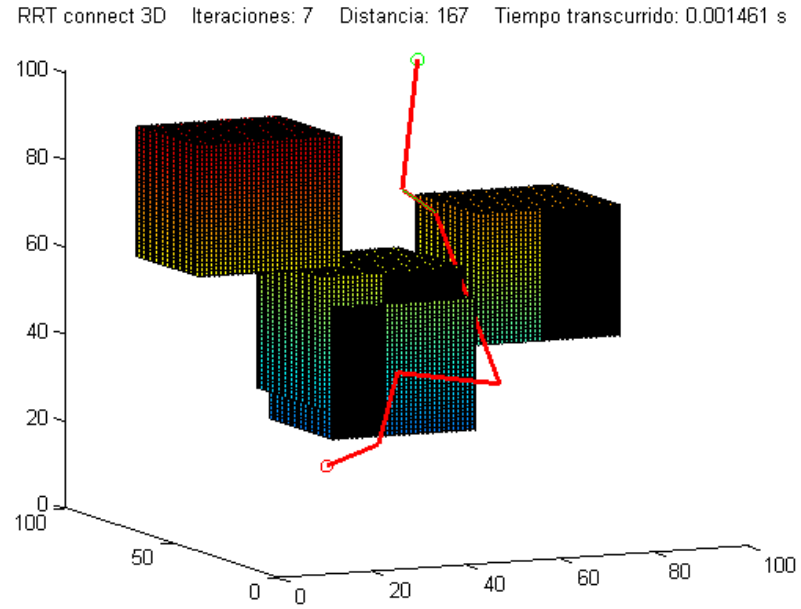

Figura 48. RRT-Connect (sin postprocesado) 3D resolviendo varios asteroides pequeños a bordear.

5.3 Algoritmo RRT multi-query + Dijkstra

5.3.1 Introducción.

El algoritmo propuesto para búsqueda de camino inicial no óptimo en estático busca resolver el problema lo más rápido posible, e idealmente, cuanto más cerca del óptimo mejor. A este algoritmo se le ha llamado RRT multi-query + Dijkstra. El lector encontrará que en este Proyecto no se ha sido muy imaginativo poniendo nombres a los algoritmos propuestos, ya que todos son llamados en base a los algoritmos combinados. De esta forma se reconoce de un vistazo los algoritmos originales empleados, y se les da el crédito que merecen.

Este algoritmo acelera la búsqueda de camino inicial mediante la búsqueda en un grafo preexistente, de esta forma se reduce en gran medida el espacio de configuraciones y la complejidad del problema. Los nodos del grafo se introducen por el usuario en el diseño del modelo del entorno, y se colocan la mínima cantidad de nodos que abarquen de forma estratégica la mayor cantidad de mapa posible (colocados en puertas, centro de habitaciones, intersección de pasillos, etc.). Mantener el tamaño del grafo pequeño es clave para que así también lo sea la complejidad del problema.

Hay tres fases en este algoritmo, en las cuales se considerarán exclusivamente los obstáculos fijos y que los puntos inicial y final están quietos. Las tres fases son las siguientes:

1) ***RRT multi-query:*** Los puntos inicial y final de la trayectoria pueden caer en cualquier punto del espacio libre de colisiones, y el grafo preexistente tiene pocos nodos, así que la probabilidad de que los puntos inicial y final caigan en un nodo del árbol tiende a cero. Lo primero a hacer es evidente, hay que unir dichos puntos al grafo preexistente.

   Se escogió el algoritmo RRT multi-query para dicha tarea, el cual se trata básicamente del algoritmo RRT normal pero adaptándolo para que en vez de considerar 1 solo punto final, considere varios a la vez (todos los nodos del grafo en nuestro caso). De esta forma, corriendo el RRT multi-query dos veces, una empezando desde P_{ini} y otra desde P_{fin}, buscando todos los nodos del grafo, se pueden conectar eficientemente dichos puntos al grafo preexistente. Ahora ya estamos listos para entrar en la segunda fase.

2) ***Búsqueda en grafo con el algoritmo de Dijkstra:*** Sabiendo de la fase anterior los nodos del grafo preexistente donde se conectan P_{ini} y P_{fin}, resta conectar dichos nodos dentro del grafo. Dado que el grafo no es muy grande no se consideró necesario un algoritmo con heurística tipo A*, empleando Dijkstra con buenos resultados.

   Se podría ver esta fase de búsqueda dentro del grafo preexistente como una heurística personalizada por el diseñador del mapa de entorno, heurística en la que solo se estudian aquellos caminos que el diseñador del mapa consideró estratégicamente convenientes.

3) ***Postprocesado:*** Por último se aplica un postprocesado de desigualdad triangular para quitar picos aleatorios en la trayectoria sin sentido, típicos de la planificación probabilística no óptima RRT. Además, los segmentos de la trayectoria resultado son recortados en segmentos más chicos unidos entre sí, esto se hace para facilitar la optimización en los siguientes capítulos (RRT* y sus derivados no funcionan bien con ramas del árbol más largas que la longitud predefinida en el planificador).

En los dos siguientes capítulos el camino inicial calculado con este algoritmo será el que se optimizará con algoritmos derivados de RRT*. La ventaja de aplicar ahora un algoritmo de planificación probabilístico es que es capaz de buscar eficientemente en el espacio de configuraciones continuo, considerando toda su complejidad, y conociendo la trayectoria inicial se pueden aplicar rápidamente heurísticas que aceleran la convergencia al **óptimo**.

Este concepto de acelerar la búsqueda inicial del camino con búsqueda en grafo y combinarlo con RRT* fue presentado por primera vez por Michael Brunner, Bernd Bruggemann y Dirk Schulz en su paper “Hierarchical Rough Terrain Motion Planning using an Optimal Sampling-Based Method” [44] en 2013, donde combinaron A* en un grafo de alta resolución con RRT*. El algoritmo aquí propuesto se inspira en el presentado por ellos. En el siguiente apartado se trata más detenidamente de las diferencias entre ambos.

Para mejor comprensión del el funcionamiento del algoritmo propuesto, en las dos páginas siguientes se ha representado esquemáticamente un entorno resuelto mediante el algoritmo propuesto. El robot debe ir desde el

punto inicial P_ini (en verde) hacia el punto final P_fin (en rojo), saliendo de un obstáculo en forma de herradura (en negro). Los obstáculos en forma de herradura representan un influyente mínimo local. Los nodos del grafo preexistente son los puntos celestes y las líneas celestes discontinuas los arcos de dicho grafo.

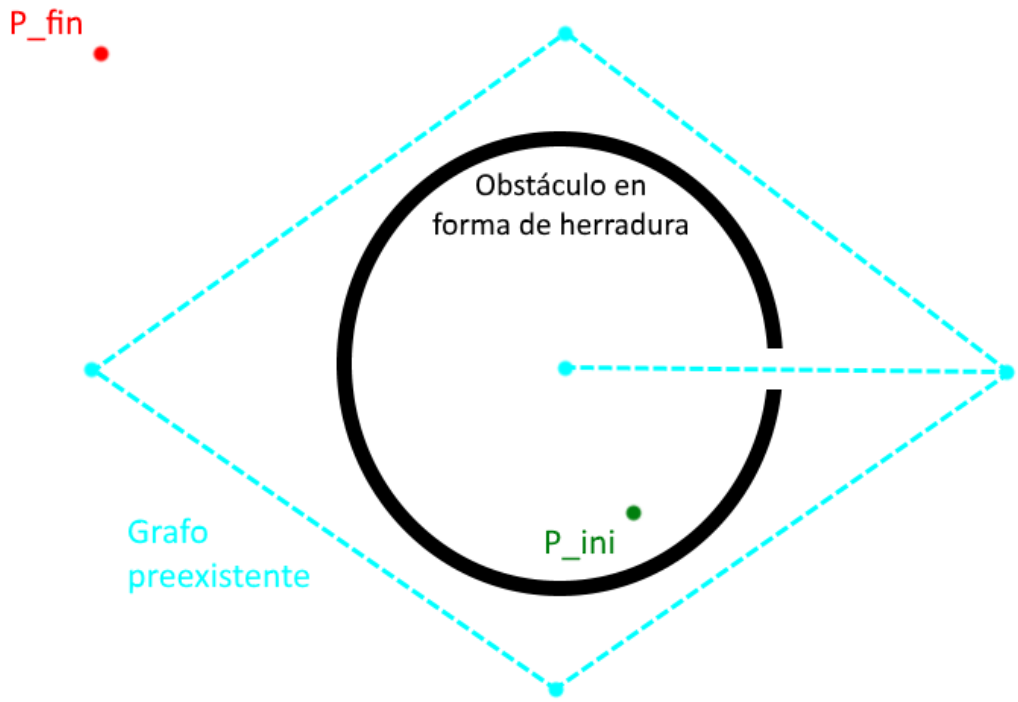

Figura 49. Entorno de ejemplo de herradura (con mínimo local) para explicar el algoritmo propuesto.

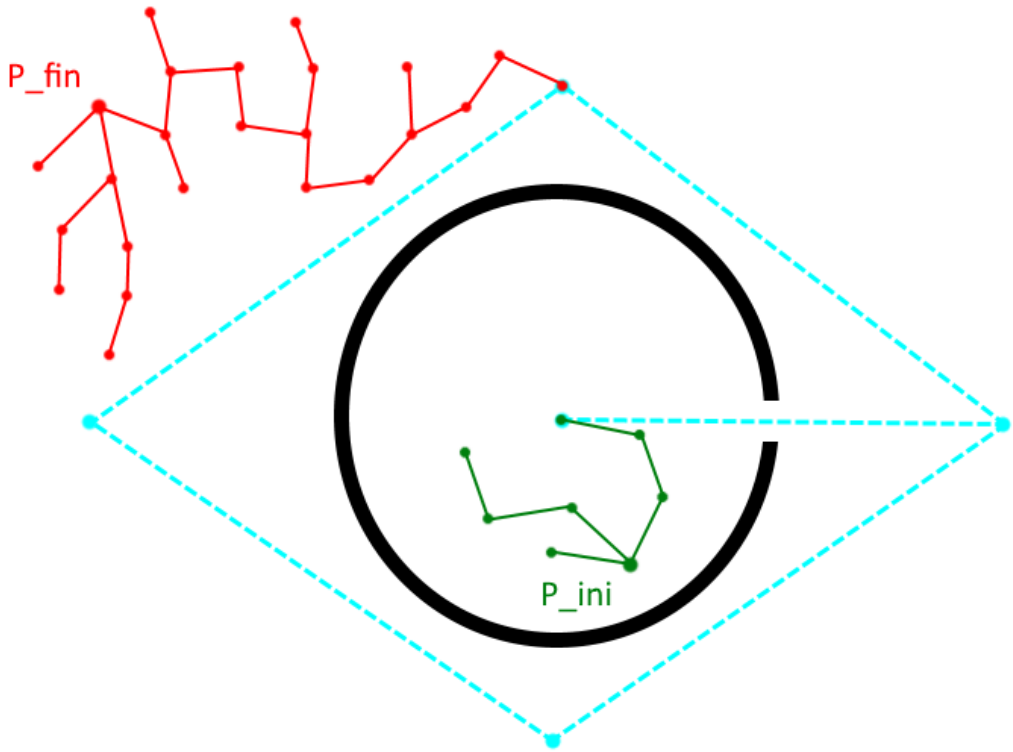

Figura 50. Fase 1 completada: mediante la construcción de dos árboles RRT multi-query se unen los puntos inicial y final con cualquier nodo del grafo preexistente.

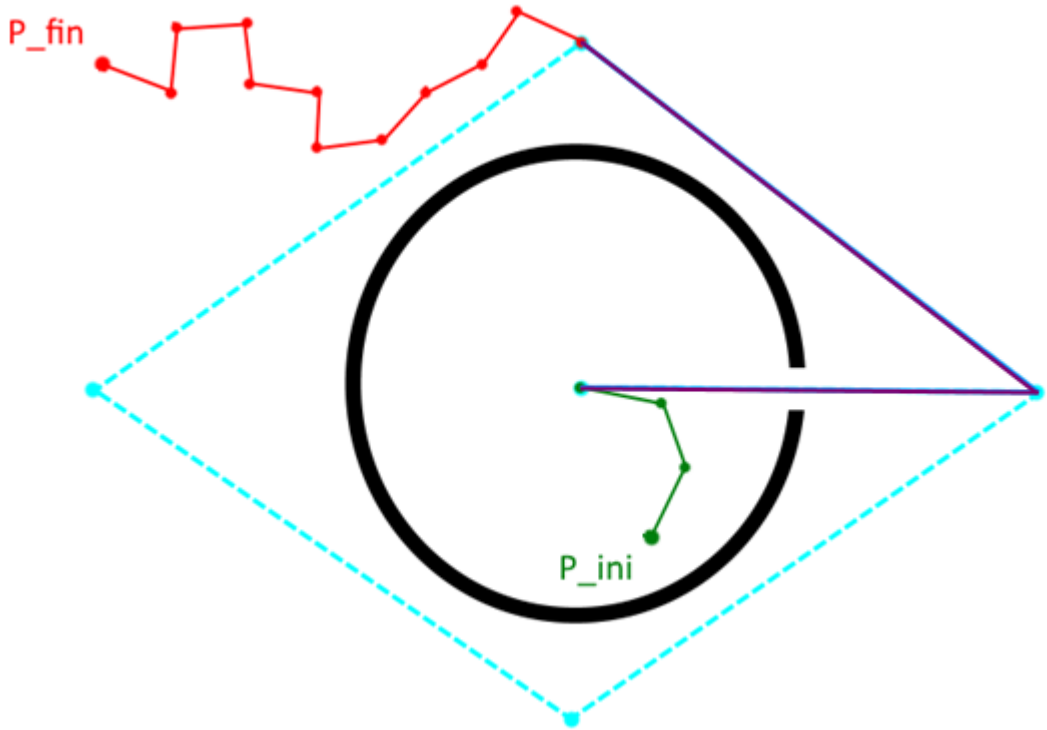

Figura 51. Fase 2 completada: descartando los nodos inútiles de los árboles verde y rojo, se unen por dentro del grafo por su camino mínimo mediante el algoritmo de Dijkstra (línea continua morada). Dentro del grafo el efecto del mínimo local es nulo.

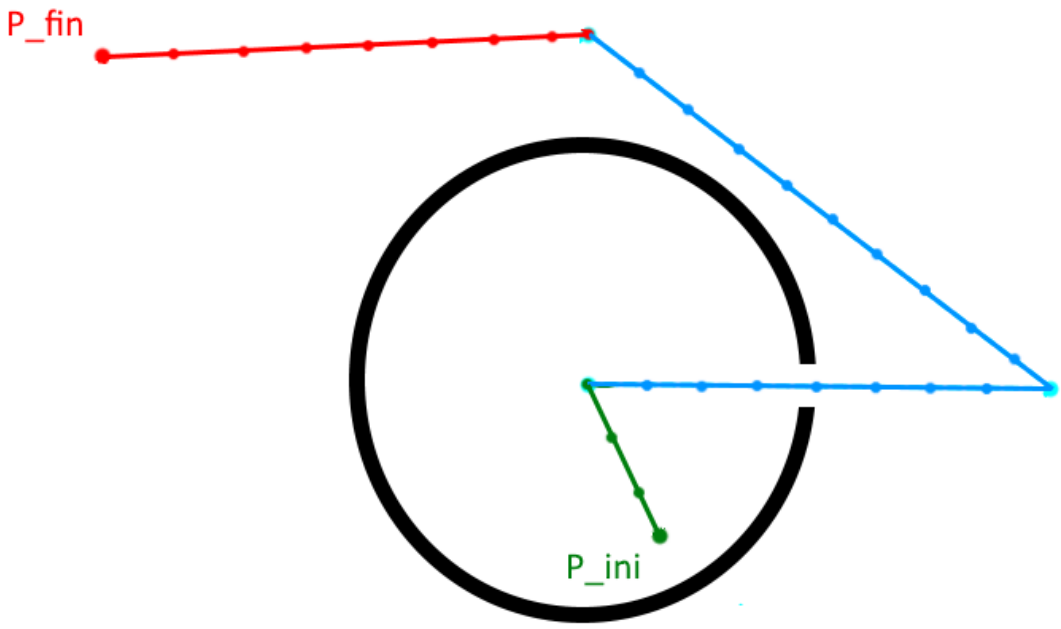

Figura 52. Fase 3 completada: descartando los nodos inútiles del grafo preexistente, se aplica primero un postprocesado por desigualdad triangular a la trayectoria y después se trocea en segmentos más pequeños (para que sea fácilmente optimizable por RRT* en los próximos capítulos). Nótese la no optimalidad de la solución.

5.3.2 Comparación con A*-RRT* y resolución del grafo.

Como se comentó anteriormente, este algoritmo propuesto está inspirado en el algoritmo A*-RRT* presentado en "Hierarchical Rough Terrain Motion Planning using an Optimal Sampling-Based Method" [44]. La principal diferencia entre ambos (sin tener en cuenta la fase de optimización del segundo) es que la resolución del grafo predefinido en A*-RRT* es muy alta, no así en el grafo predefinido del algoritmo propuesto. Recordar que la resolución del grafo preexistente de RRT multi-query + Dijkstra es la mínima que garantiza abarcar la mayor cantidad de mapa, con nodos ubicados en lugares estratégicos tales como puertas, intersección de pasillos, centro de habitaciones, etc.

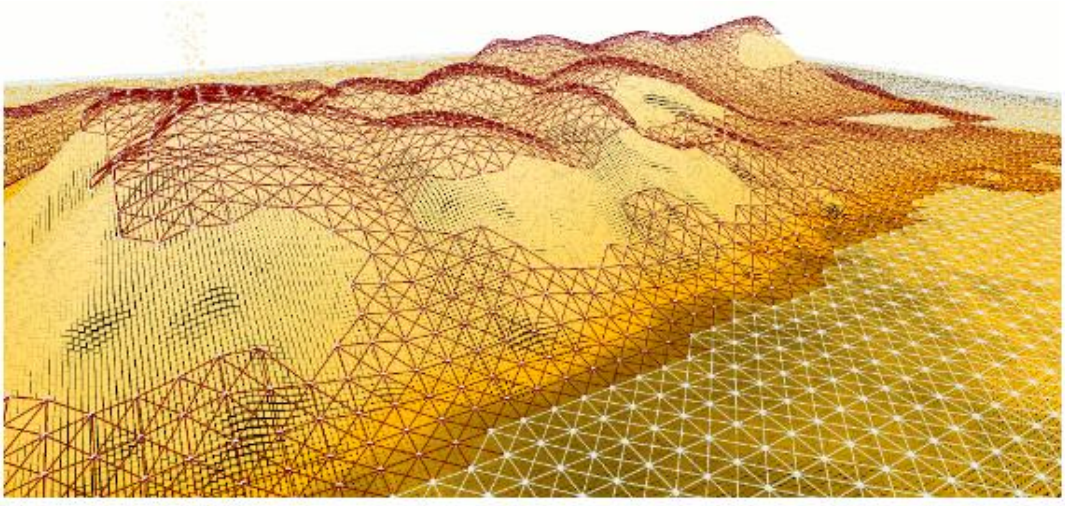

Figura 53. Ejemplo de grafo de alta resolución usado en A*-RRT* [44] para la búsqueda del camino inicial.

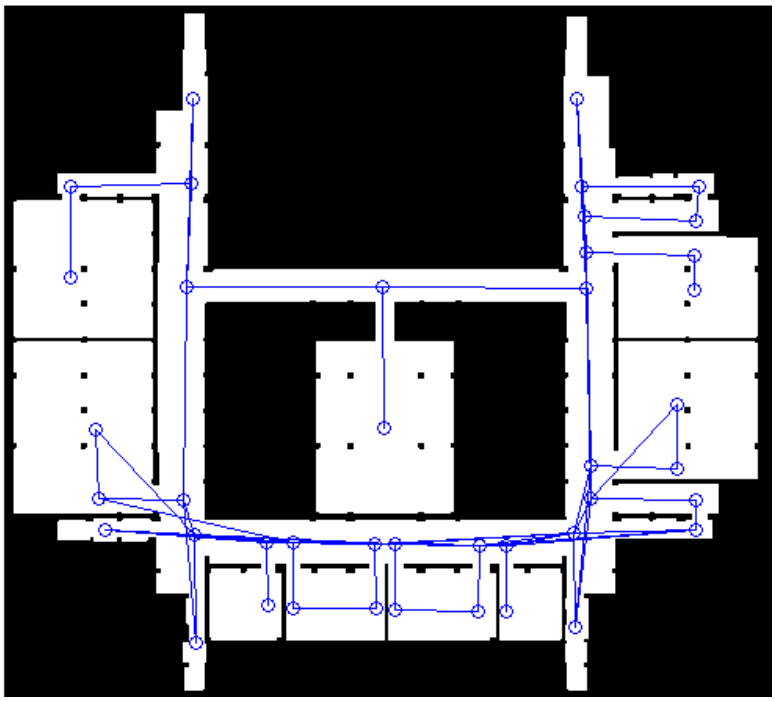

Figura 54. Ejemplo de grafo de resolución mínima usado en el algoritmo propuesto RRT multi-query+Dijkstra para la búsqueda del camino inicial.

Si bien en este Proyecto no se ha comparado el desempeño de A* (con su grafo de alta resolución) con el RRT multi-query + Dijkstra (con su grafo de resolución mínima), algunas conclusiones parecen evidentes:

- Aunque a igualdad de resolución de grafo el algoritmo A* necesita comprobar menos nodos que Dijkstra, resulta justo decir que usar Dijkstra con un grafo de resolución varios órdenes de magnitud menor que A* necesitará un número menor o igual de comprobaciones de nodos.
- El cálculo de la comprobación de un nodo con el algoritmo de Dijkstra es más ligero computacionalmente que con A*, ya que el primero solo contiene unas pocas sumas y comparaciones if-else y el segundo aparte de eso contiene el cálculo de la estimación heurística de la distancia hasta el final.
- Aunque a Dijkstra habría que sumarle el tiempo de ejecución del RRT multi-query para el acercamiento al grafo, A*-RRT* no tiene la suficiente resolución como para no implementar un algoritmo de acercamiento al grafo, por lo que también perderá tiempo en este aspecto.
- El empleo de un grafo de alta resolución puede dar caminos iniciales más cercanos al óptimo.

Combinando las tres primeras conclusiones se deduce que Dijkstra en baja resolución es más rápido que A* en alta resolución. Por el contrario, la cuarta conclusión indica que A* podría dar una solución más cercana al óptimo, quedando en futuras fases de optimización menos trabajo por hacer.

Esto dejaría ambos algoritmos en un equilibrio tiempo-coste, inclinándose A*-RRT* por una primera solución de menor coste y más lenta, y el algoritmo de búsqueda inicial aquí propuesto por una primera solución más rápida pero menos cercana al óptimo.

Por supuesto lo anterior son deducciones lógicas, no afirmaciones empíricas. Por motivos de alcance del Proyecto, no se ha podido llegar a comparar ambos algoritmos y demostrar dichas conclusiones y deducciones, por muy interesante que ello fuera.

Cabría plantearse ir variando el nivel de resolución del grafo y ver cómo ello afecta a los tiempos de ejecución y distancias. También cabría estudiar el caso de usar un A* con el grafo de baja resolución, en vez del algoritmo de Dijkstra. El resultado esperado sería que aumentando la resolución del grafo aumentara la calidad de la solución, pero el tiempo de ejecución aumentara en mayor medida, y que empleando A* en grafos de baja resolución no mejorara de forma significativa por el peso del cálculo de la estimación heurística. Todo esto también por motivos de alcance ha quedado fuera del Proyecto.

Por último, añadir que los tiempos de ejecución arrojados por el RRT multi-query + Dijkstra son totalmente despreciables (del orden de unidades o como mucho decenas de milisegundos) respecto al tiempo que tarda el robot en desplazarse. Esto justifica por qué el resto de esfuerzo del autor del Proyecto fuera encaminado a optimizar otras fases cuello de botella, como pueden ser la optimización y la navegación en tiempo real.

5.3.3 Funciones implementadas en 2D y 3D.

Aunque realmente este algoritmo se haya implementado tanto en 2D como 3D, aquí se presentará solo el caso 2D. En el siguiente apartado se usará para presentar resultados y comparar con el algoritmo RRT-Connect, el cual pese a ser más lento (como ya se verá), tiene la ventaja de poderse usar sin grafo preexistente.

La función MATLAB del algoritmo RRT multi-query + Dijkstra es la siguiente:

```
trayectoria = Planificador_RRT_multi_query_Dijkstra(mapa, P_ini, P_fin,
muestra_animaciones, muestra_resultado, Nmax_RRT_multi_query, salto, grafo,
dist_grafo);
```

Como entradas tiene el mapa de obstáculos, siendo como ya se ha comentado una matriz representativa del modelo estático del entorno. P_ini y P_fin los puntos inicial y final, muestra_animaciones y muestra_resultado, son variables *booleanas* para decidir si se desea o no representar gráficamente las animaciones y resultados respectivamente, Nmax_RRT_multi_query es el número máximo de iteraciones de búsqueda de cada RRT multi-query, salto es la distancia de las ramas (en píxeles) del RRT. Por último grafo y dist_grafo son dos matrices que representan respectivamente el grafo de nodos (con sus posiciones en el espacio e interconexiones entre nodos) y el coste entre interconexiones (distancia de los arcos).

Devuelve una única trayectoria tras aplicar un postprocesado por desigualdad triangular y tras trocear, como se comentó anteriormente.

Para este algoritmo no se ha implementado mallado de entorno ya que normalmente el mallado acelera la búsqueda cuando el número de nodos del árbol es elevado, y este algoritmo suele encontrar el camino en una cantidad muy reducida de iteraciones, no saliendo a cuenta en este caso.

Como en los casos anteriores, no se entra en explicar dicha función al detalle, ya que a grandes rasgos ya se ha explicado. Como el código MATLAB viene debidamente comentado, si se desea profundizar más en el algoritmo se recomienda verlo directamente desde ahí.

5.3.4 Resultados y comparación con análisis Montecarlo.

A continuación se comparará en todo tipo de mapas el desempeño de los tres algoritmos vistos en este capítulo: RRT, RRT-Connect y RRT multi-query + Dijkstra. El RRT-Connect esta vez tendrá mallado de entorno para su máximo rendimiento, aunque para este algoritmo concreto no suele suponer mucha mejora (el mallado se nota sobre todo en algoritmos que necesiten mucha cantidad de nodos). Los tres algoritmos incorporan al final un postprocesado por desigualdad triangular.

Para tener referencia del hardware usado, todos los planificadores de algoritmos ejecutados en este Proyecto se han hecho en un portátil Asus con Windows 10 de 64 bits, CPU Intel Core i7 4510U, 6GB de RAM, gráfica NVIDIA GeForce 820M y versión de MATLAB R2013b.

5.3.4.1 Problema de la herradura.

El primer problema será el de la **herradura** ya visto para ilustrar el algoritmo RRT multi-query + Dijkstra, pero esta vez con un agujero más pequeño para que sea más desafiante. Lo primero que hay que hacer es configurar los parámetros de cada planificador por separado, de forma que todos puedan funcionar a su máxima potencia. Al ser planificadores similares todos se han comportado mejor con la misma longitud de rama, 100 píxeles. En el caso del RRT-Connect el mallado que aparentemente le ha ido mejor han sido 4 subdivisiones horizontales y 6 verticales.

Una vez configurados los planificadores y para que la comparación sea de calidad, se procede a realizar un *análisis Montecarlo* de los tiempos de ejecución y distancias solución, esto se hace corriendo unas 100 veces cada planificador en dicho problema de la herradura. Una vez se tienen los datos en bruto se analizan con un software estadístico. En este Proyecto se ha usado *Oracle Crystal Ball* [45], el cual te reconoce automáticamente según los datos de entrada la *distribución estadística de mejor bondad de ajuste*, así como sus parámetros representativos. Los resultados obtenidos fueron los siguientes:

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (segundos)	Mediana (segundos)	Desviación estándar (segundos)	Mínimo (segundos)	Máximo (segundos)
RRT	2%	Logarítmica normal	6,0184	1,0854	13,0412	0,0049	71,9278
RRT-Connect	0%	Logarítmica normal	0,1539 - 0,1499	0,0930 - 0,1079	0,2028 - 0,1614	0,0129	0,8533
RRT multi-query + Dijkstra	0%	Logarítmica normal	0,0023	0,0022	0,0005	0,0016	0,0044

Tabla 5. Análisis Montecarlo (100 cálculos de trayectoria) del **tiempo de ejecución** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El mapa resuelto es el de la **herradura**.

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (píxeles)	Mediana (píxeles)	Desviación estándar (píxeles)	Mínimo (píxeles)	Máximo (píxeles)
RRT	2%	Binomial negativa	900,85	894,00 - 874,00	130,86	697	1242
RRT-Connect	0%	Binomial negativa	998,69	985,00 - 993,00	201,39 - 195,25	698	1549
RRT multi-query + Dijkstra	0%	Beta	1145	994 - 1145	177,12 - 178,01	994,32	1353,19

Tabla 6. Análisis Montecarlo (100 cálculos de trayectoria) de la **distancia** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El mapa resuelto es el de la **herradura**.

Sobra decir que el análisis Montecarlo es una herramienta poderosísima a la hora de comparar con criterio el desempeño de varios algoritmos. En la estimación de tiempo de ejecución este análisis nos demuestra cómo el RRT estándar es un orden de magnitud más lento (de media y mediana) que el RRT-Connect. Y lo que es peor, su desviación estándar es dos órdenes de magnitud mayor, pudiendo tardar tiempos tan dispares como mínimo 5 ms y máximo 1 minuto y 12 s. Esto es una barbaridad, tener tiempos tan poco fiables no es nada deseable, es por eso que RRT no tiene ninguna aplicación frente al RRT-Connect.

Aunque los resultados del tiempo de ejecución del RRT-Connect son muy positivos, palidecen comparados con los resultados del RRT multi-query + Dijkstra. La media, mediana y máximo del segundo son 2 órdenes de magnitud menores que las del Connect, y la desviación estándar 3 órdenes de magnitud menores. Dicha desviación estándar de 0,5 ms garantiza que el tiempo de ejecución del RRT multi-query + Dijkstra se puede considerar constante a efectos prácticos (al menos en este problema), lo cual es tremendamente deseable, sobre todo con un rango de valores tan rápido como 1,6 - 4,4 ms.

Respecto al análisis Montecarlo de distancia, los resultados no son tan rompedores, siendo similares en los tres casos. Irónicamente el RRT normal parece tener ligeramente mejor comportamiento en cuanto a distancia, mientras que en el RRT multi-query + Dijkstra se da un curioso fenómeno, y es que solo había determinados valores posibles de distancias (2 en este caso), es decir, *el empleo de un grafo predefinido discretiza los valores de distancia*. Es por eso que la distribución estadística de mejor ajuste es distinta.

A continuación se mostrarán algunas imágenes de los algoritmos resolviendo el problema de la herradura:

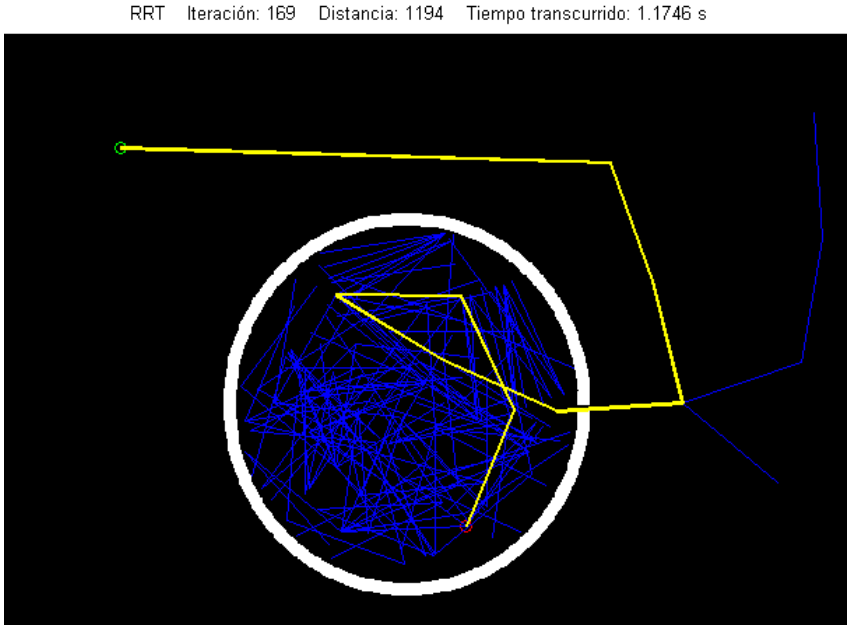

Figura 55. RRT saliendo con dificultad de la herradura.

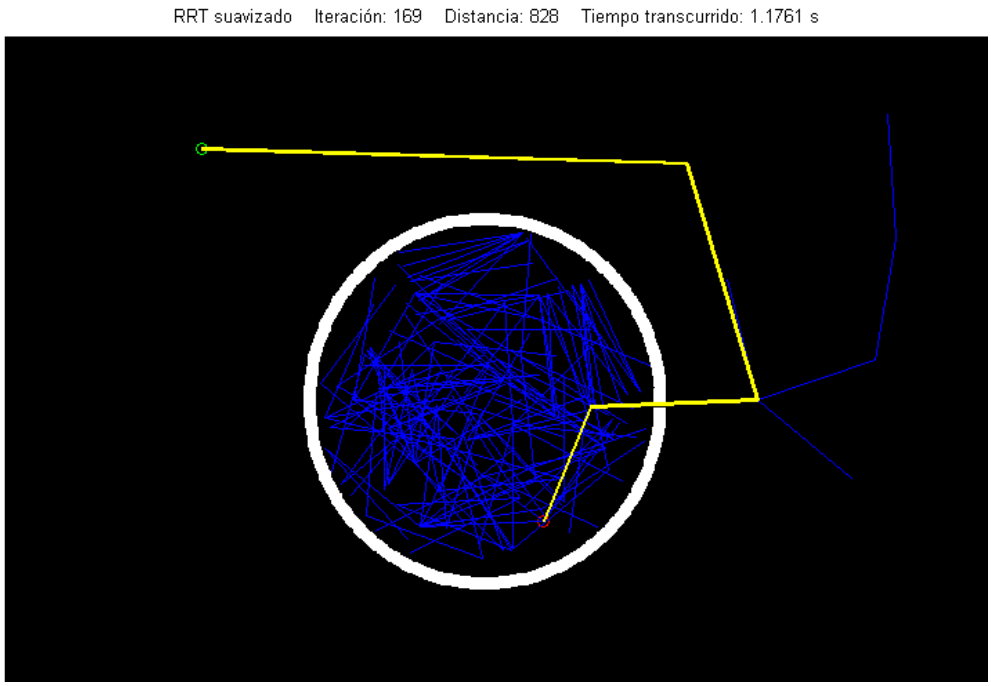

Figura 56. RRT (con postprocesado) saliendo con dificultad de la herradura.

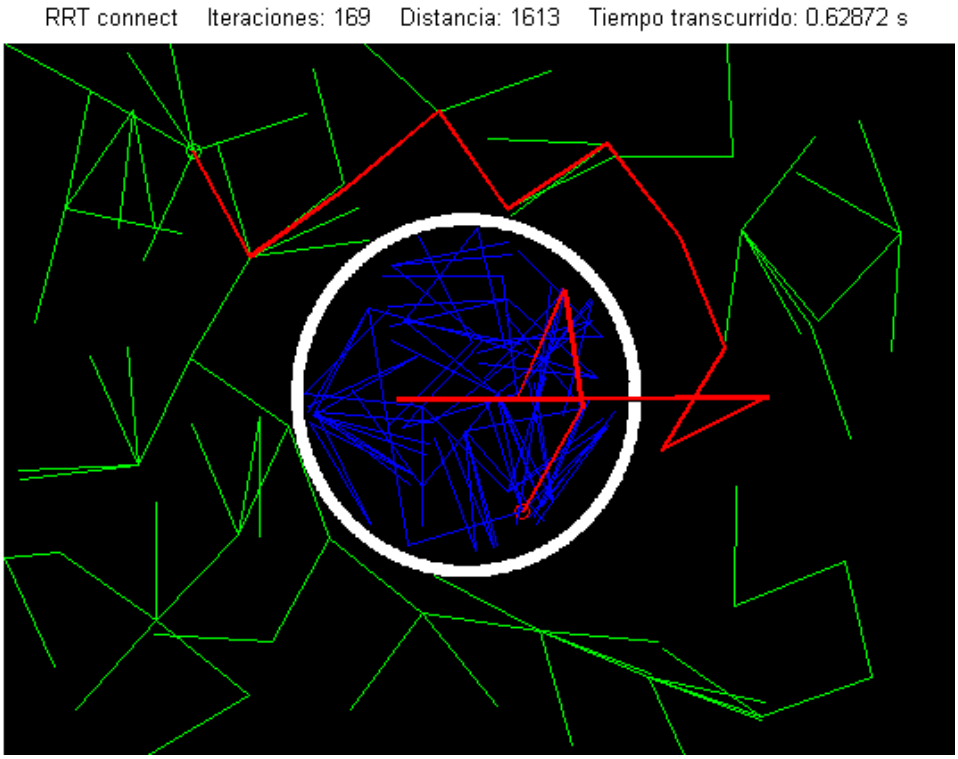

Figura 57. RRT-Connect también con cierta dificultad para salir de la herradura.

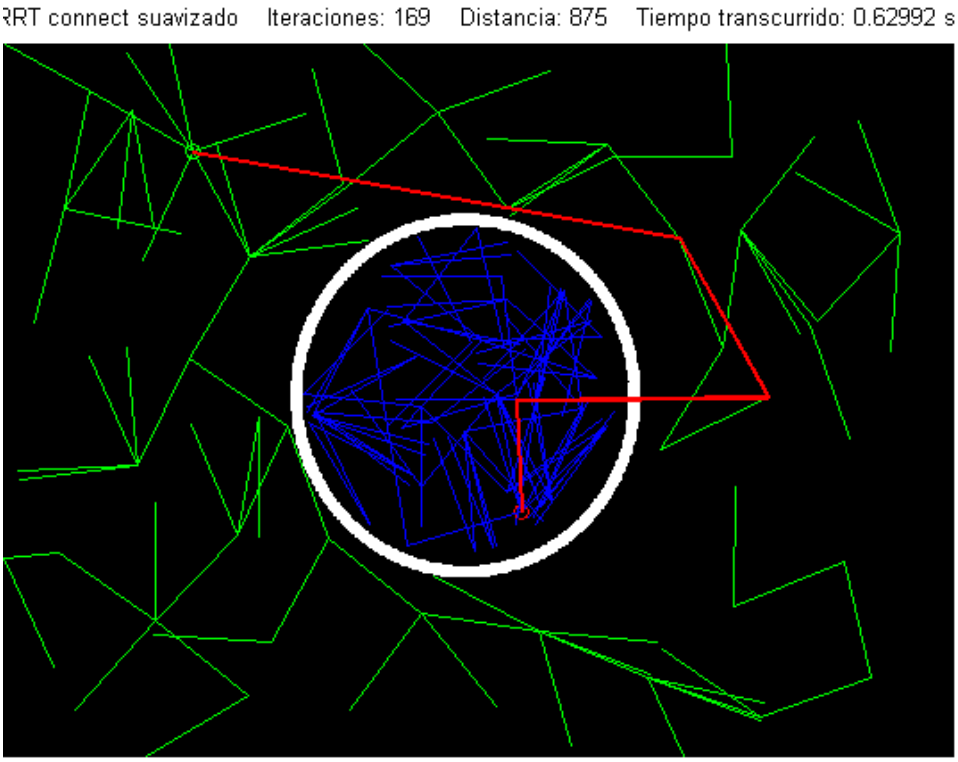

Figura 58. RRT-Connect (con postprocesado) también con cierta dificultad para salir de la herradura.

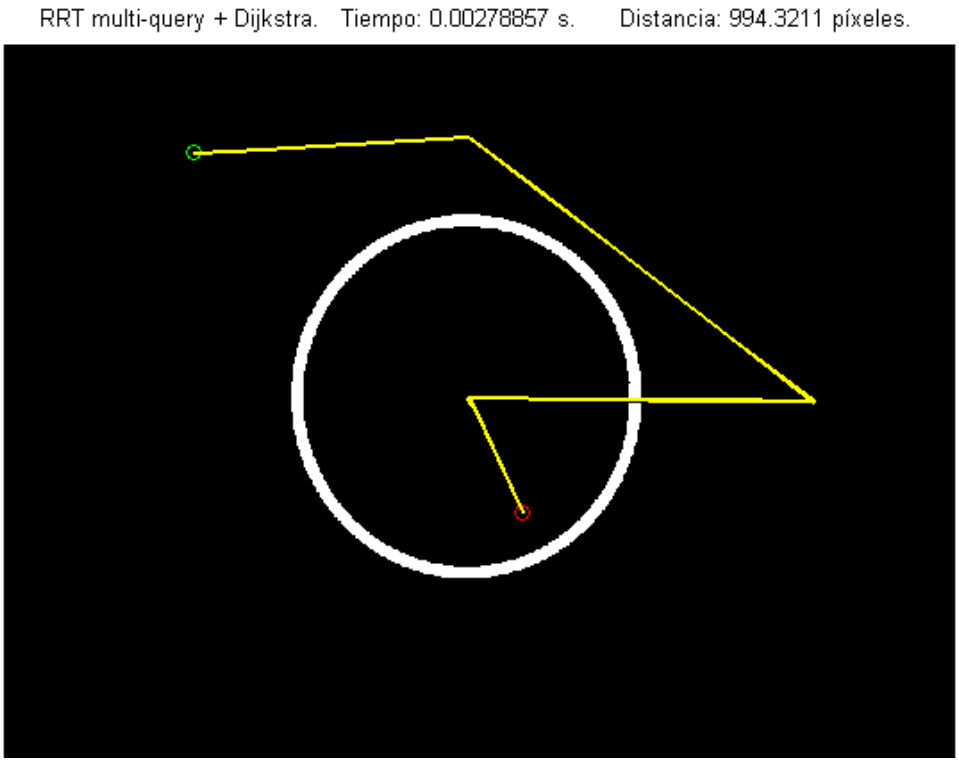

Figura 59. RRT multi-query + Dijkstra saliendo instantáneamente de la herradura.

5.3.4.2 Problema del laberinto imposible.

El segundo problema será el del **laberinto imposible.** Este laberinto ya se adelantó en el apartado 4.2 que resultaba irresoluble si no se usaba búsqueda en grafo, y ahora es cuando se va a demostrar. Lo primero que hay que hacer, igual que antes, es configurar los parámetros de cada planificador por separado, de forma que todos puedan funcionar a su máxima potencia. Los algoritmos RRT y RRT-Connect por mucho que se intenten configurar son incapaces de resolver el laberinto, ni si quiera con 5000 nodos en el árbol están cerca de hacerlo. En el caso del RRT multi-query + Dijkstra la longitud de rama de árbol que mejores resultados ha dado ha sido 15 píxeles.

Una vez configurado el planificador y para que la comparación sea de calidad, igual que antes se procede a realizar un *análisis Montecarlo*, corriendo unas 100 veces el planificador en dicho problema del laberinto imposible. Una vez se tienen los datos en bruto de tiempos de ejecución y distancias, estos se analizan con un software estadístico. En este Proyecto se ha usado *Oracle Crystal Ball* [45], el cual te reconoce automáticamente según los datos de entrada la *distribución estadística de mejor bondad de ajuste*, así como sus parámetros representativos. Los resultados obtenidos fueron los siguientes:

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (segundos)	Mediana (segundos)	Desviación estándar (segundos)	Mínimo (segundos)	Máximo (segundos)
RRT	100%	-	-	-	-	-	-
RRT-Connect	100%	-	-	-	-	-	-
RRT multi-query + Dijkstra	0%	Logarítmica normal	0,9654	0,9638 - 0,9634	0,0071 - 0,0072	0,9552	0,9976

Tabla 7. Análisis Montecarlo (100 cálculos de trayectoria) del **tiempo de ejecución** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El mapa resuelto es el del **laberinto imposible**.

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (píxeles)	Mediana (píxeles)	Desviación estándar (píxeles)	Mínimo (píxeles)	Máximo (píxeles)
RRT	100%	-	-	-	-	-	-
RRT-Connect	100%	-	-	-	-	-	-
RRT multi-query + Dijkstra	0%	Constante	10549,85	10549,85	0	10549,85	10549,85

Tabla 8. Análisis Montecarlo (100 cálculos de trayectoria) de la **distancia** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El mapa resuelto es el del **laberinto imposible**.

Este análisis de Montecarlo refleja nuevamente la bajísima desviación estándar del algoritmo RRT multi-query + Dijkstra, así como su rápida búsqueda del camino inicial incluso en situaciones en las que RRT-Connect falla totalmente. También nos reafirma la discretización del conjunto de distancias solución, como en este caso, que solo existe un posible valor para la distancia del camino inicial.

En la siguiente página se muestran dos capturas del laberinto imposible siendo resuelto, o al menos intentándose:

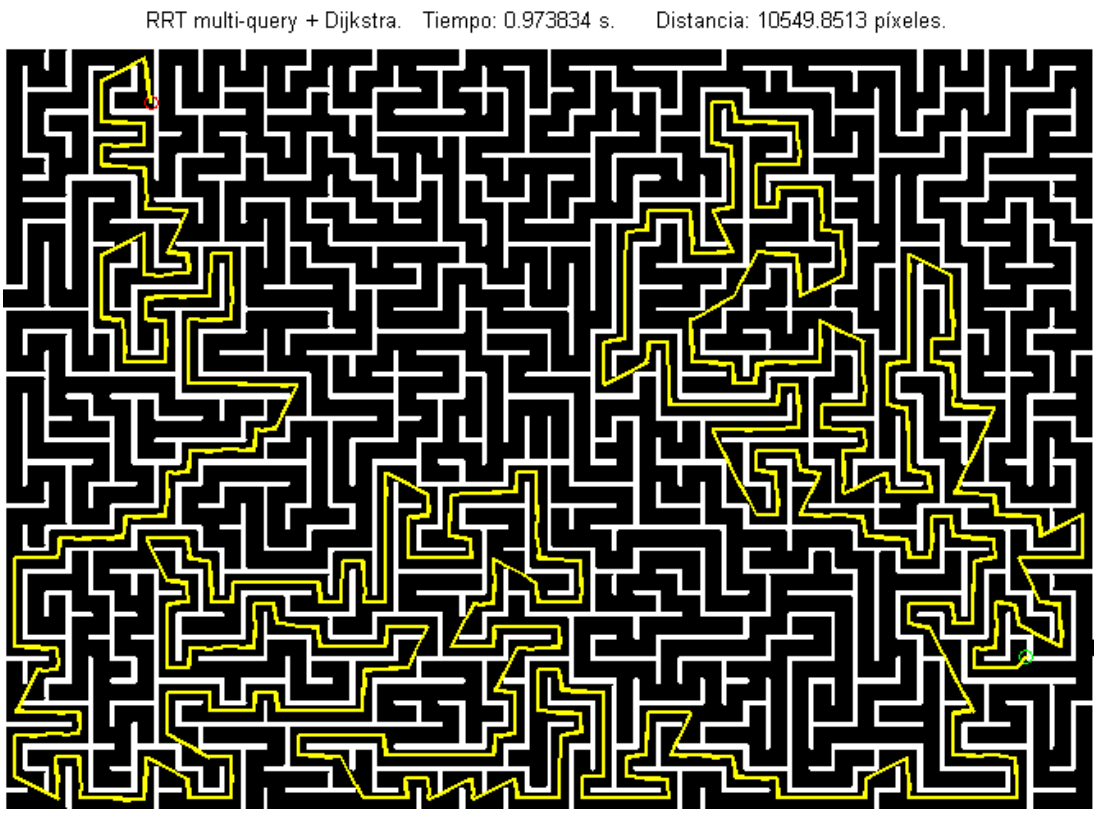

Figura 60. RRT multi-query + Dijkstra resolviendo con éxito el laberinto imposible.

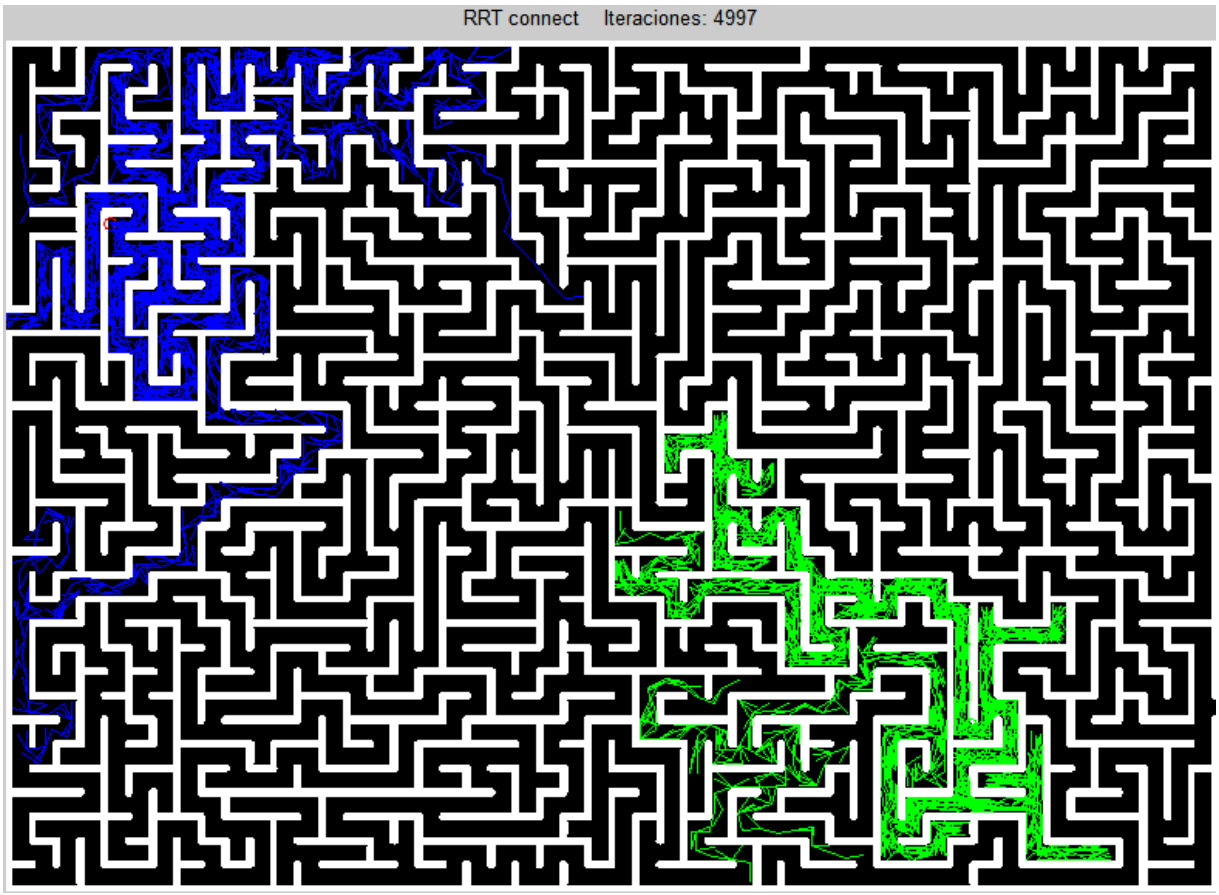

Figura 61. RRT-Connect fracasando en resolver el laberinto imposible en 5000 iteraciones. Demasiados mínimos locales incluso para un método probabilístico.

5.3.4.3 Problema de salir de la ETSI.

El tercer problema será el de **salir de la ETSI desde las piscinas hasta la parada de autobús.** Como ya se ha demostrado la superioridad de RRT-Connect frente al RRT clásico en todos sus aspectos, a partir de ahora solo se compararán los algoritmos RRT multi-query + Dijkstra con el RRT-Connect. Este último pese a ser más lento, se puede usar en situaciones en las que no se disponga de grafo predefinido del mapa.

En la creación del mapa se ha supuesto que el robot sea un robot móvil terrestre que no pueda subir ni bajar escaleras, esto pondrá las cosas difíciles al planificador ya que el robot solo podrá entrar y salir del edificio por rampas. Zonas no accesibles cerradas al público se descartaron a la hora de crear el mapa. Lo primero que hay que hacer, igual que antes, es configurar los parámetros de cada planificador por separado, de forma que todos puedan funcionar a su máxima eficiencia. En el caso del RRT multi-query + Dijkstra la longitud de rama de árbol que mejores resultados ha dado ha sido 50 píxeles, y en RRT-Connect de 15 píxeles. En el caso de RRT-Connect hubo que aumentar el mallado de entorno para que los tiempos no se disparasen, dejándolo en 20 subdivisiones tanto horizontales como verticales.

Una vez configurado el planificador y para que la comparación sea de calidad, igual que antes se procede a realizar un análisis Montecarlo, corriendo muchas veces el planificador en dicho problema de salir de la ETSI. Se ejecutó el planificador RRT multi-query + Dijkstra 100 veces y el *RRT-Connect, dado su lentitud, solo 30 veces.* Una vez se tienen los datos en bruto se analizan con un software estadístico. En este Proyecto se ha usado Oracle Crystal Ball [45], el cual te reconoce automáticamente según los datos de entrada la distribución estadística de mejor bondad de ajuste, así como sus parámetros representativos. Los resultados obtenidos fueron los siguientes:

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (segundos)	Mediana (segundos)	Desviación estándar (segundos)	Mínimo (segundos)	Máximo (segundos)
RRT-Connect	16,67%	Logarítmica normal	34,6249 - 33,8141	27,3485 - 24,1615	26,6890 - 21,8849	7,3561	88,4162
RRT multi-query + Dijkstra	0%	Logarítmica normal	0,0070	0,0069	0,0016	0,0046	0,0128

Tabla 9. Análisis Montecarlo del **tiempo de ejecución** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El problema resuelto es el de **salir de la ETSI**.

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (píxeles)	Mediana (píxeles)	Desviación estándar (píxeles)	Mínimo (píxeles)	Máximo (píxeles)
RRT-Connect	16,67%	Binomial negativa	1580 - 1580	1572 - 1729	195,052 - 193,007	1333	1823
RRT multi-query + Dijkstra	0%	Constante	1513,67	1513,67	0	1513,67	1513,67

Tabla 10. Análisis Montecarlo de la **distancia** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. El problema resuelto es el del **salir de la ETSI**.

El análisis Montecarlo actual refuerza las conclusiones anteriores. La distribución de tiempos de ejecución de todos los algoritmos de planificación se pueden modelar como una logarítmica normal, presentando el algoritmo propuesto en este Proyecto una desviación estándar de milisegundos sea cual sea la complejidad del problema. RRT-Connect no presenta desviación estándar constante ante distinta complejidad de problema, subiendo a ser en este problema 4 órdenes de magnitud mayor que la del algoritmo propuesto.

Sobra decir que el problema actual ha resultado ser más complicado de lo que podía parecer a simple vista, ya que la mediana de la duración del RRT-Connect ha sido de 24-27 segundos. Esto se debe a la influencia de dos mínimos locales: la piscina y la entrada al edificio desde la esquina por la acera tan angosta que tiene. Sin embargo, RRT multi-query + Dijkstra no ha sufrido retraso alguno, tardando de mediana 7 ms.

Respecto a distancia de la solución, los resultados son similares a los anteriores también, presentando el algoritmo propuesto una discretización del conjunto de distancias solución. En este caso, en el 100% de los casos la distancia ha sido la misma. Todos los algoritmos RRT parecen presentar una distribución de distancias de binomial negativa, teniendo en este caso RRT-Connect una mediana de longitud ligeramente superior al RRT multi-query + Dijkstra, significando que la mitad de veces mejorará al algoritmo propuesto y la otra mitad empeorará dicha distancia.

A continuación se muestran capturas de este problema, primero las distribuciones de tiempos de ejecución y después el mapa siendo resuelto por cada algoritmo.

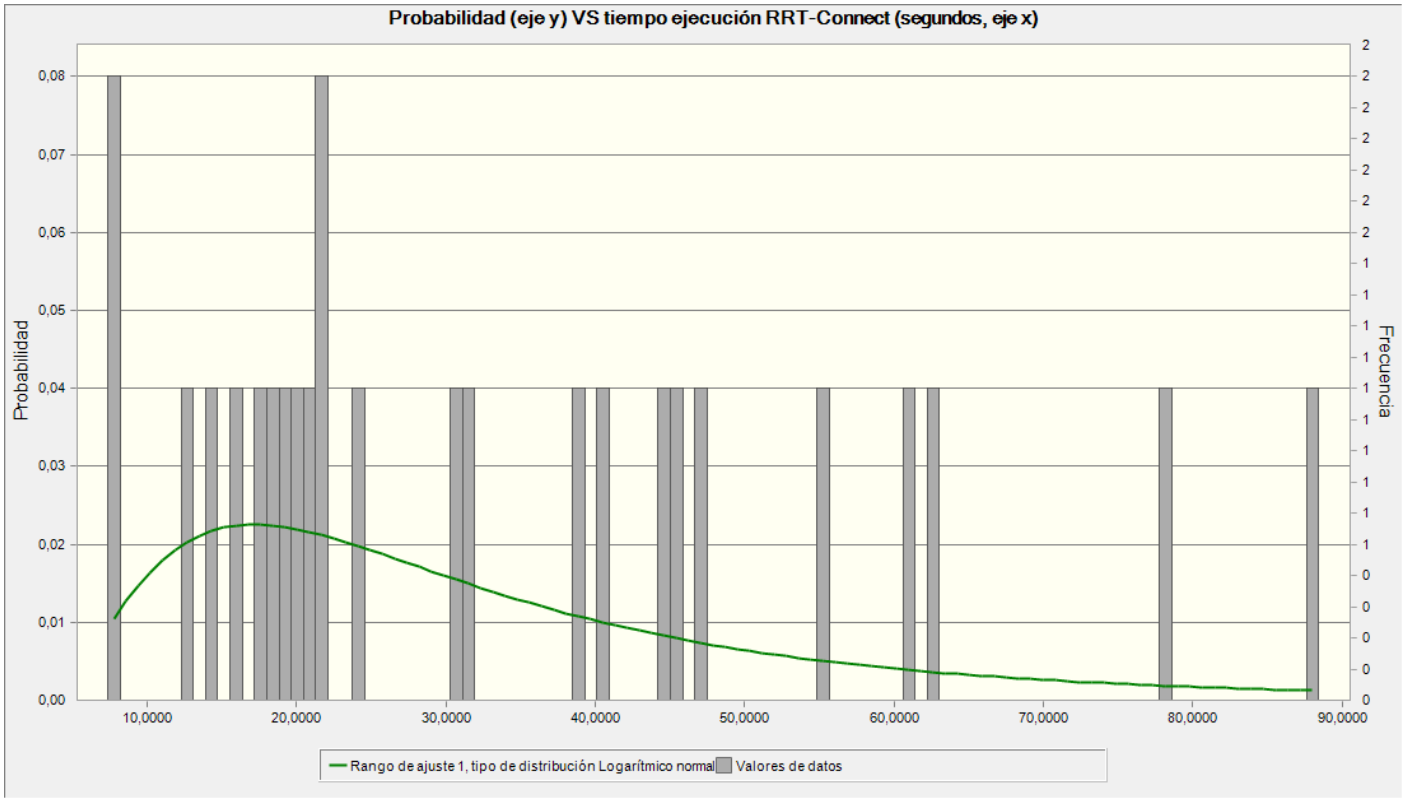

Figura 62. Distribución logarítmica normal, la de mejor bondad de ajuste sugerida por Oracle Crystal Ball [45] en base a los resultados del análisis Montecarlo del tiempo de ejecución del algoritmo RRT-Connect.

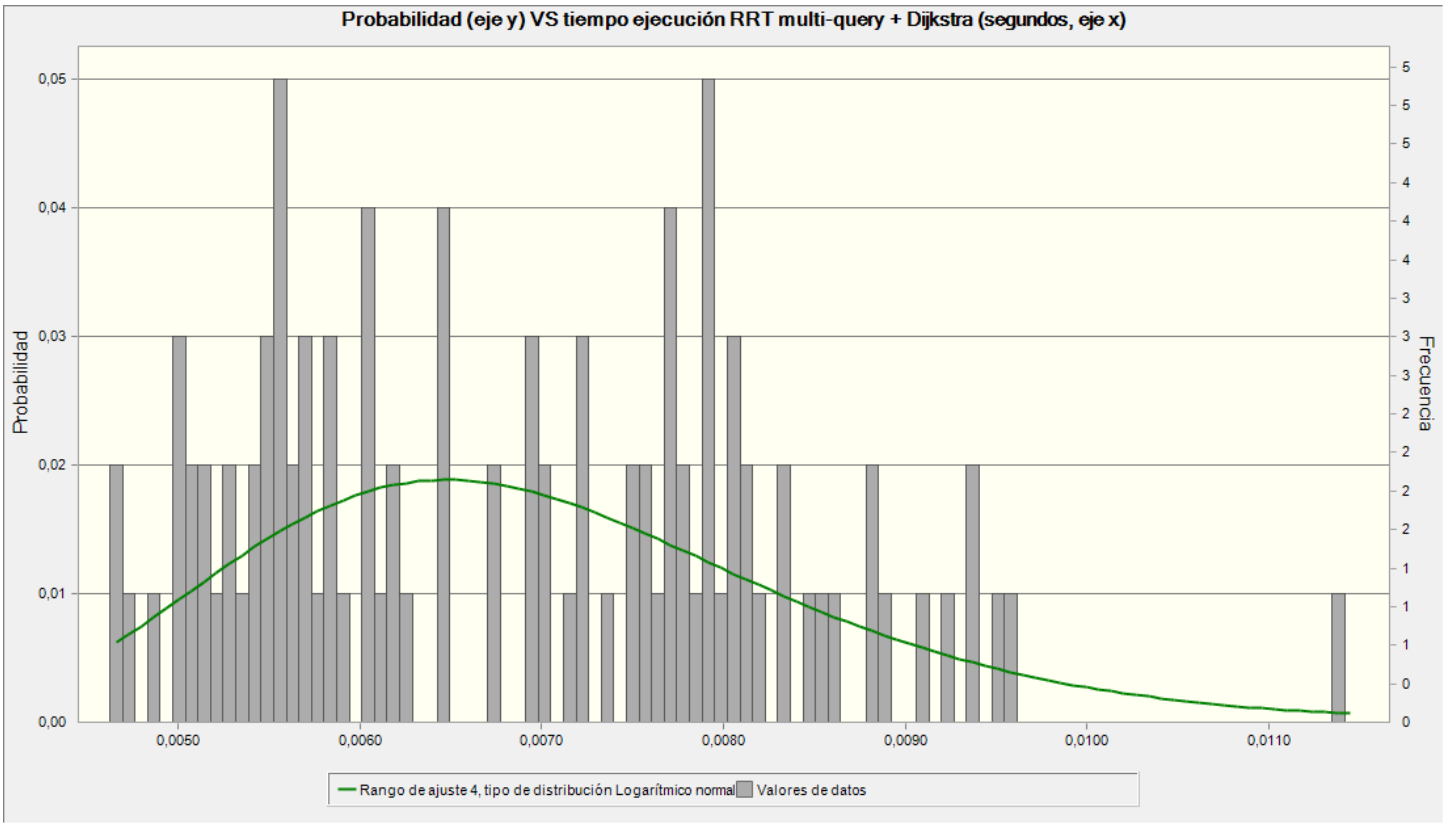

Figura 63. Distribución logarítmica normal, la de mejor bondad de ajuste sugerida por Oracle Crystal Ball [45] en base a los resultados del análisis Montecarlo del tiempo de ejecución del algoritmo RRT multi-query + Dijkstra.

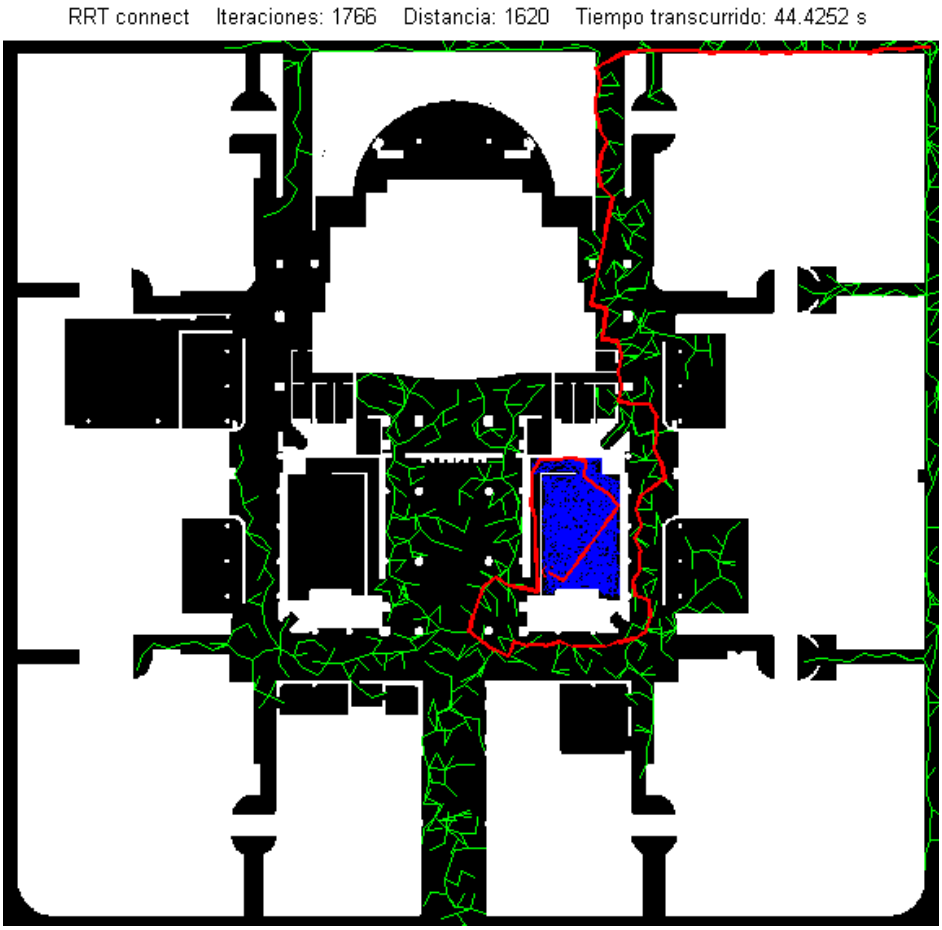

Figura 64. RRT-Connect resolviendo con mucho esfuerzo el mapa.

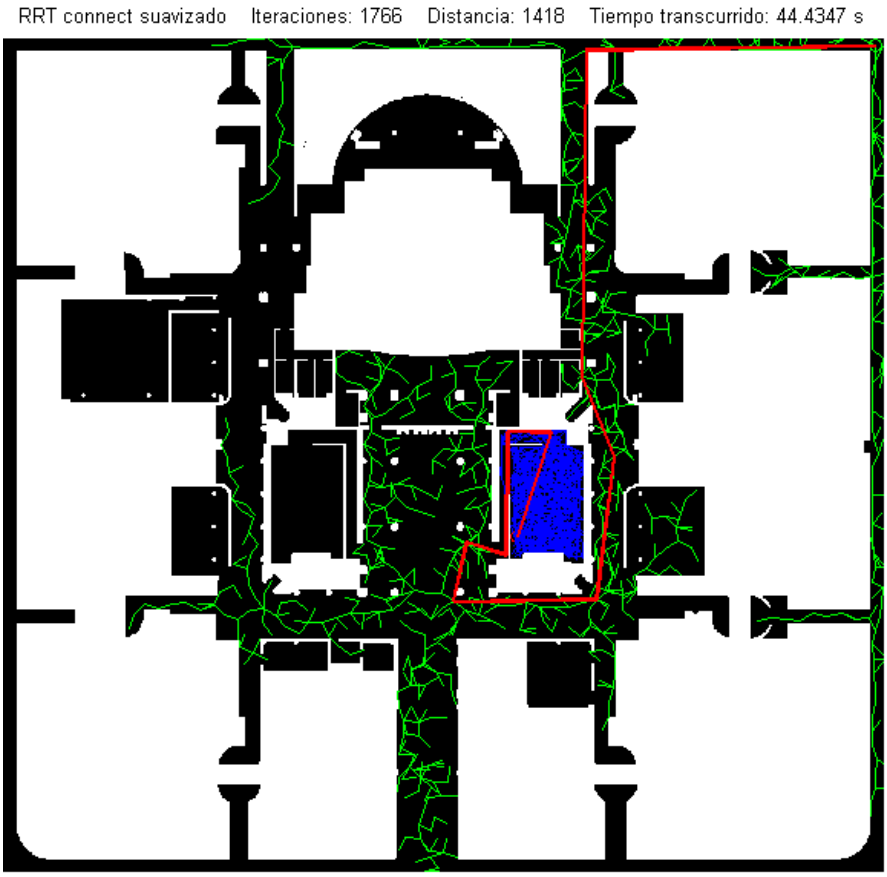

Figura 65. RRT-Connect (con postprocesado) resolviendo con mucho esfuerzo el mapa.

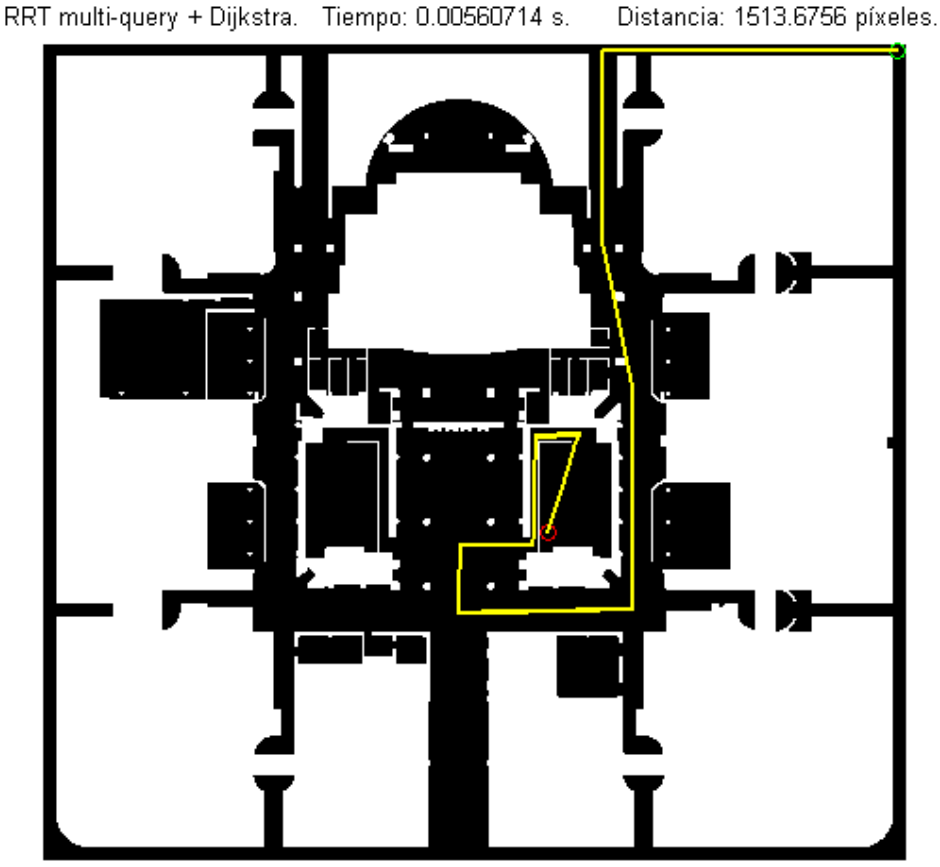

Figura 66. RRT multi-query + Dijkstra resolviendo instantáneamente el problema.

5.3.4.4 Problema sencillo de orientarse en los sótanos de la ETSI.

Por último, se presentará un problema más sencillo, para que no todo sean laberintos imposibles y mínimos locales. El problema de ahora será uno de orientación en el sótano de la ETSI, teniendo que ir de un punto a otro por sus pasillos. Nuevamente, se descartará el uso del algoritmo RRT clásico, ya que RRT-Connect ha demostrado comportarse mejor en todos los casos. Lo primero, igual que siempre, es configurar los parámetros de cada planificador por separado, de forma que todos puedan funcionar a su máxima eficiencia. En este problema ambos planificadores han funcionado mejor con 50 píxeles de longitud de rama de árbol. En el caso de RRT-Connect con un mallado de 4 subdivisiones tanto horizontales como verticales fue suficiente.

Una vez configurado el planificador y para que la comparación sea de calidad, igual que antes se procede a realizar un análisis Montecarlo, corriendo 100 veces el planificador. Una vez se tienen los datos en bruto se analizan con un software estadístico. En este Proyecto se ha usado Oracle Crystal Ball [45], el cual te reconoce automáticamente según los datos de entrada la distribución estadística de mejor bondad de ajuste, así como sus parámetros representativos. Los resultados obtenidos fueron los siguientes:

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (segundos)	Mediana (segundos)	Desviación estándar (segundos)	Mínimo (segundos)	Máximo (segundos)
RRT-Connect	0%	Logarítmica normal	0,0281 - 0,0292	0,0241 - 0,0230	0,0148 - 0,0261	0,0097	0,2517
RRT multi-query + Dijkstra	0%	Logarítmica normal	0,0028	0,0027	0,0008	0,0018	0,0060

Tabla 11. Análisis Montecarlo (100 cálculos de trayectoria) del **tiempo de ejecución** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. Problema: de **orientarse en el sótano de la ETSI**.

Algoritmo	Trayectorias no encontradas	Distribución estadística mejor ajuste	Media (píxeles)	Mediana (píxeles)	Desviación estándar (píxeles)	Mínimo (píxeles)	Máximo (píxeles)
RRT-Connect	0%	Binomial negativa	801,37	801,00 - 796,50	22,51 - 22,56	767	902
RRT multi-query + Dijkstra	0%	Constante	781,91	781,91	0	781,91	781,91

Tabla 12. Análisis Montecarlo (100 cálculos de trayectoria) de la **distancia** de distintos algoritmos para la búsqueda de camino inicial no óptimo en estático. Problema: de **orientarse en el sótano de la ETSI**.

En vista de los resultados en este problema sencillo, queda claro que RRT multi-query + Dijkstra se comporta más rápido que RRT-Connect no solo en los problemas complejos, sino también en los sencillos. Aparte de ser más rápido de mediana y de media, también su desviación estándar es notablemente menor. Respecto a distancia la diferencia entre ambos no es muy significativa, siendo ligeramente menor la del algoritmo propuesto.

Se puede concluir que si es posible construir un grafo de resolución mínima en la fase de diseño del mapa, este resultaría en tiempos de ejecución mucho menores en la primera fase de búsqueda de camino inicial no óptimo en estático. La distancia no se vería muy afectada, pero el algoritmo de optimización tendría mucho más tiempo para reducir la distancia a igualdad de condiciones. Si no se pudiera construir dicho grafo por lo que fuera, RRT-Connect también da buenos resultados si la complejidad del problema no es demasiada.

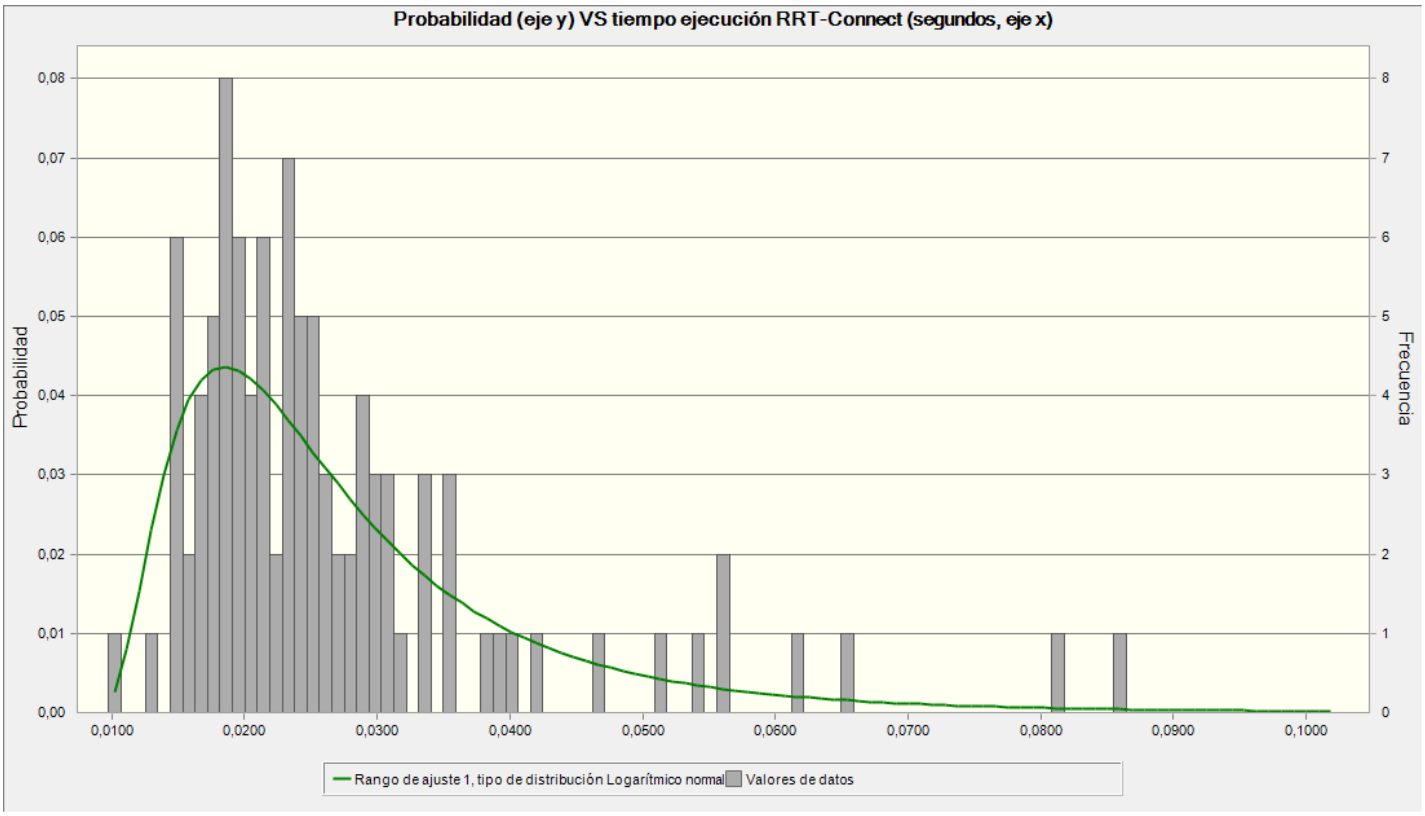

Figura 67. Distribución logarítmica normal, la de mejor bondad de ajuste sugerida por Oracle Crystal Ball [45] en base a los resultados del análisis Montecarlo del tiempo de ejecución del algoritmo RRT-Connect

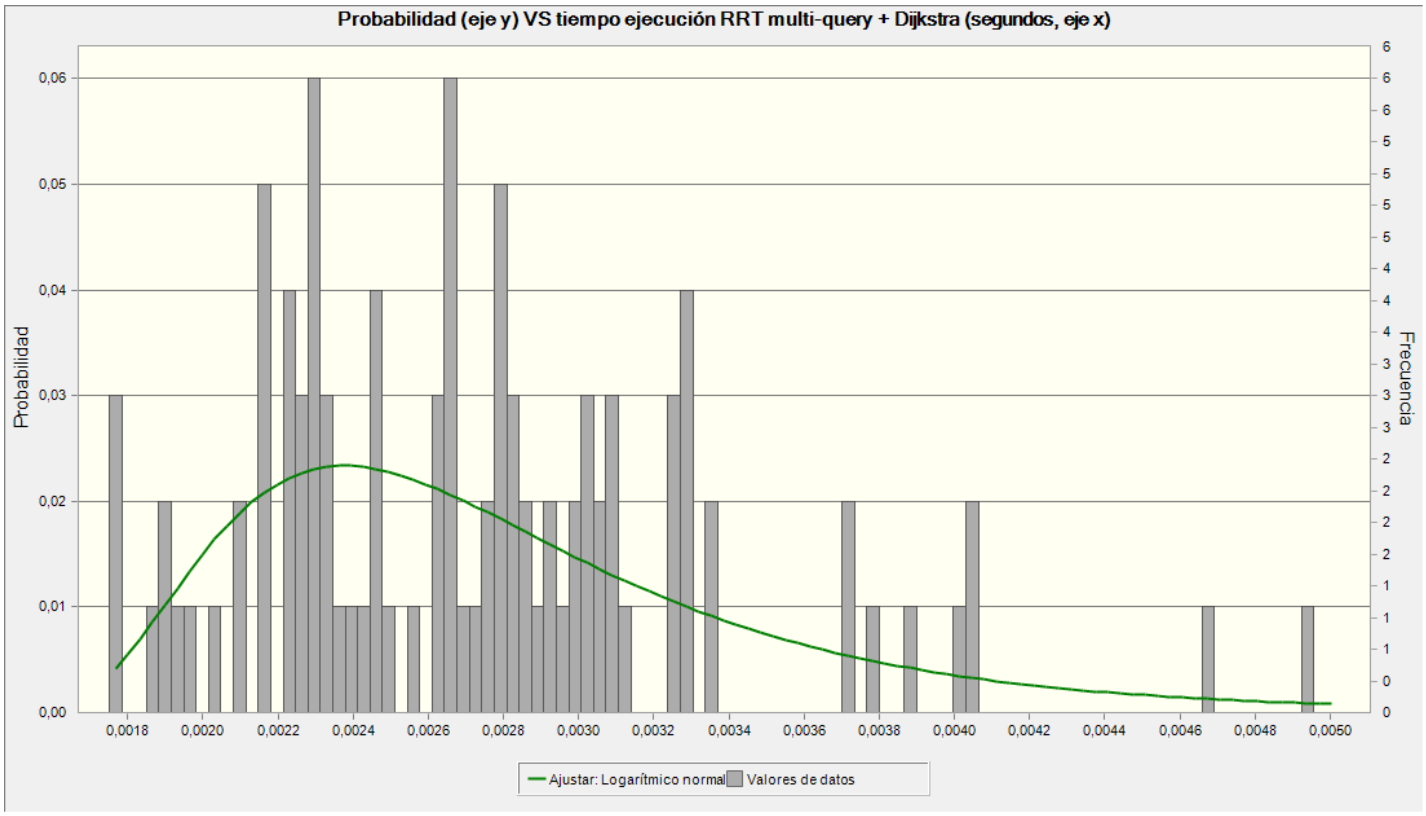

Figura 68. Distribución logarítmica normal, la de mejor bondad de ajuste sugerida por Oracle Crystal Ball [45] en base a los resultados del análisis Montecarlo del tiempo de ejecución del algoritmo RRT multi-query + Dijkstra.

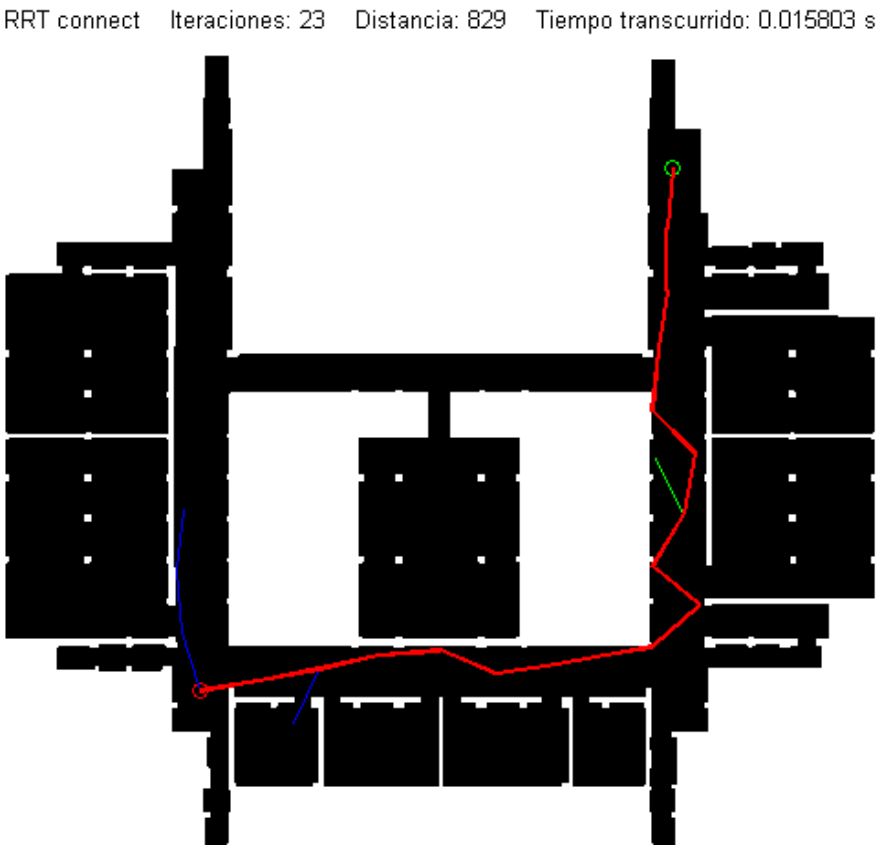

Figura 69. RRT-Connect resolviendo rápidamente este problema sencillo.

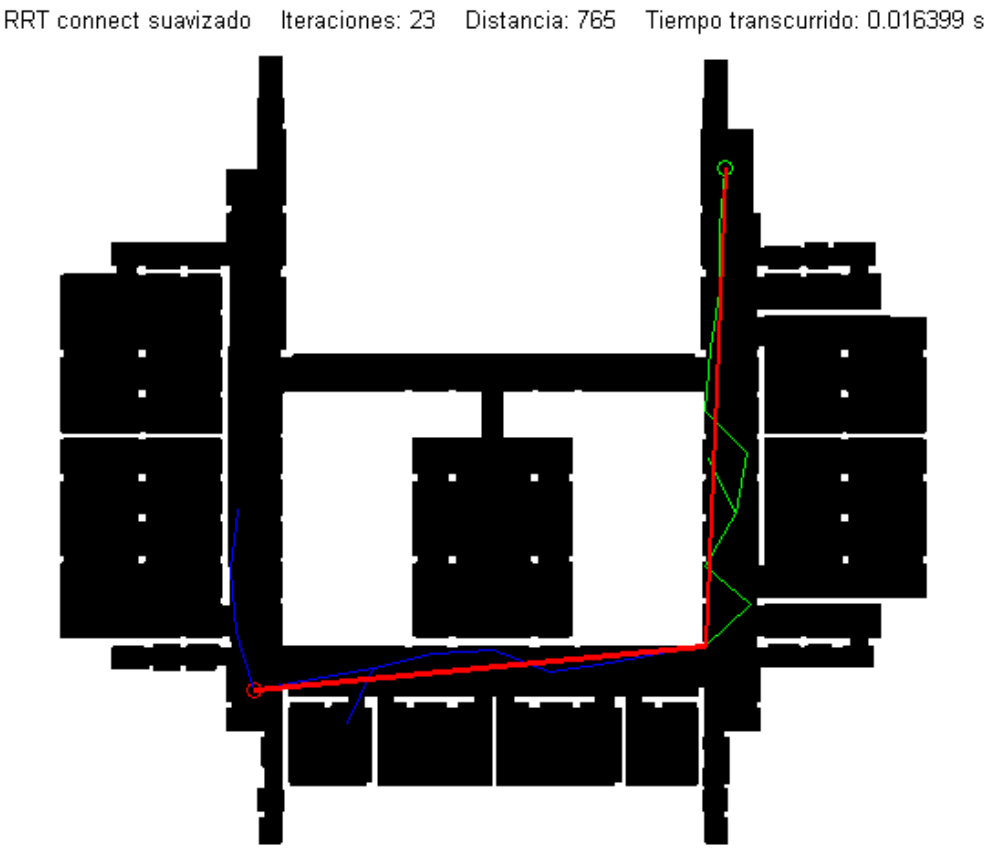

Figura 70. RRT-Connect (con postprocesado) resolviendo rápidamente este problema sencillo.

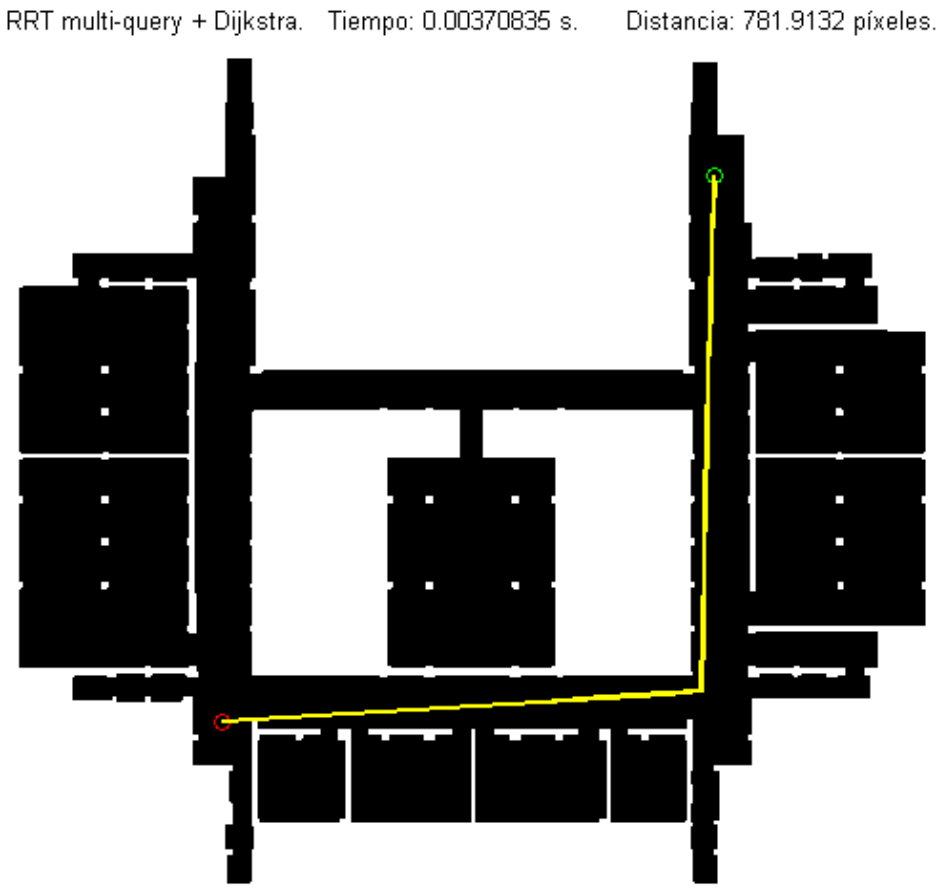

Figura 71. El algoritmo propuesto, RRT multi-query + Dijkstra, resolviendo la trayectoria aún más rápido que RRT-Connect, demostrando ser mejor no solo en problemas complejos, sino también en los sencillos.

6 ALGORITMOS DE OPTIMIZACIÓN DEL CAMINO EN ESTÁTICO

6.1 Algoritmo RRT*

En este capítulo nos adentramos en los *algoritmos de planificación probabilísticos* ***óptimos***. Por planificación óptima se entiende aquella que tiene en cuenta el coste de la trayectoria y pretende minimizarlo, pudiendo definirse *el coste de la trayectoria como la distancia o el tiempo que tarda el robot en recorrerla*. En este Proyecto, el coste será la distancia.

Normalmente esta optimización es computacionalmente cara, ya que una vez se encuentra el camino inicial se debe seguir muestreando hasta que el camino converja al óptimo con el tiempo. Dicho tiempo para este algoritmo RRT* tiende a infinito, es por eso que surgen algoritmos heurísticos que están enfocados en una convergencia más rápida.

Esto significa que en estos algoritmos de optimización, de no tiempo real, el usuario debe acotar la duración del programa porque si no estaría ejecutándose por tiempo indefinido. Esta acotación de la duración del programa puede hacerse bien limitando el número de iteraciones del planificador, poniendo una condición de distancia, leyendo una señal de parada o directamente limitando el tiempo. De momento en estos capítulos del Proyecto que estudian el problema estático, la acotación temporal será limitando el *número de iteraciones*.

La justificación de la planificación óptima es que el tiempo que pierde el programa optimizando la trayectoria, si el planificador está bien configurado, es menor al tiempo que perdería el robot no pasando por el camino más corto.

Durante mucho tiempo, se estuvo intentando mejorar el algoritmo RRT para que optimizara eficazmente el problema de planificación. Surgieron muchos algoritmos que lo intentaban, pero la revolución no llegó hasta la primera aparición en el 2010 del *RRT*, en "Sampling-based Algorithms for Optimal Motion Planning"*, de Sertac Karaman y Emilio Frazzoli [15]. La principal ventaja del RRT* es que al igual que el RRT ofrece completitud probabilística, pero garantizando la convergencia del camino hacia el óptimo.

La desventaja es que para dicha convergencia el tiempo tiende a infinito (*asintóticamente óptimo*) y el árbol tiende a crecer demasiado (y con él la cantidad de memoria necesaria). Los siguientes subcapítulos tratarán de aportar soluciones a este problema con algoritmos heurísticos basados en el RRT*.

RRT* con un único árbol encuentra el primer camino y lentamente va convergiendo al óptimo. El árbol además de contener las coordenadas de los nodos y el índice del nodo padre, contiene el coste (distancia) entre padre e hijo y el coste total entre dicho nodo y el nodo raíz u origen. Es muy importante decir que al contener el árbol el histórico de costes totales entre cada nodo y el origen, al optimizarse un nodo intermedio del árbol, todos los costes de los nodos posteriores en el árbol a dicho nodo deben ser actualizados.

En resumen, las diferencias en el muestreo entre RRT y RRT* son las siguientes:

- Al insertar un nodo nuevo no se pone como padre al más cercano, sino que se busca dentro de un determinado radio el *padre óptimo*, este es el que le dé menos coste total pasando por él desde el punto nuevo hasta el punto inicial.
- Una vez el nodo nuevo está unido a su padre óptimo, se comprueba si a otros nodos del árbol ya existentes cerca del punto nuevo (por cerca nos referimos a dentro de un radio que es parámetro del planificador) les interesa tenerlo a él como padre para minimizar su coste hasta el punto inicial. De ser así el punto nuevo pasa a ser su padre. A esto se le llama "*Rewire*" o *recableado*.

Este es el pseudocódigo de todo lo explicado con anterioridad:

```
1 T ← InitializeTree();                                  // Inicializa el árbol
2 T ← InsertNode(Ø, zinit, T);                           // Inserta el punto inicial como padre raíz
3 for i=0 to N                          // Itera un número acotado de veces para limitar el tiempo
4   zrand ← Sample(i);                                   // Igual que RRT
5   znearest ← Nearest(T, zrand);                        // Igual que RRT
6   (xnew, unew, Tnew) ← Steer (znearest, zrand);        // Igual que RRT
7   if Obstaclefree(xnew) then                           // Igual que RRT
8     Znear ← Near(T, znew, |V|);                            // Busca nodos dentro de un radio
9     zmin ← Chooseparent (Znear, znearest, znew, xnew);     // Escoge padre óptimo
10    T ← InsertNode(zmin, znew, T);                         // Inserta el nodo con padre óptimo
11    T ← Rewire (T, Znear, zmin, znew);                     // Revisa si algún a nodo le conviene
                                                             // el punto nuevo como padre
12 return T                                                  // Devuelve el árbol
```

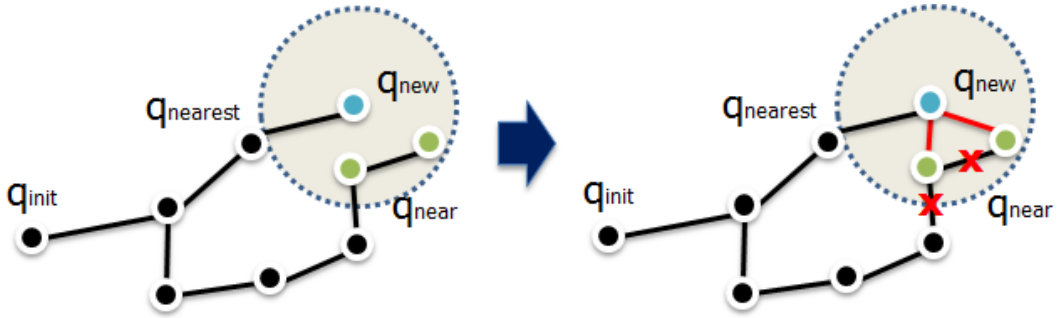

Figura 72. A la izquierda la función "Near" del pseudocódigo y a la derecha la función "Rewire". [42]

Este proceso de optimización mediante sucesivos recableados genera árboles cuyas ramas se expanden radialmente, minimizando el coste.

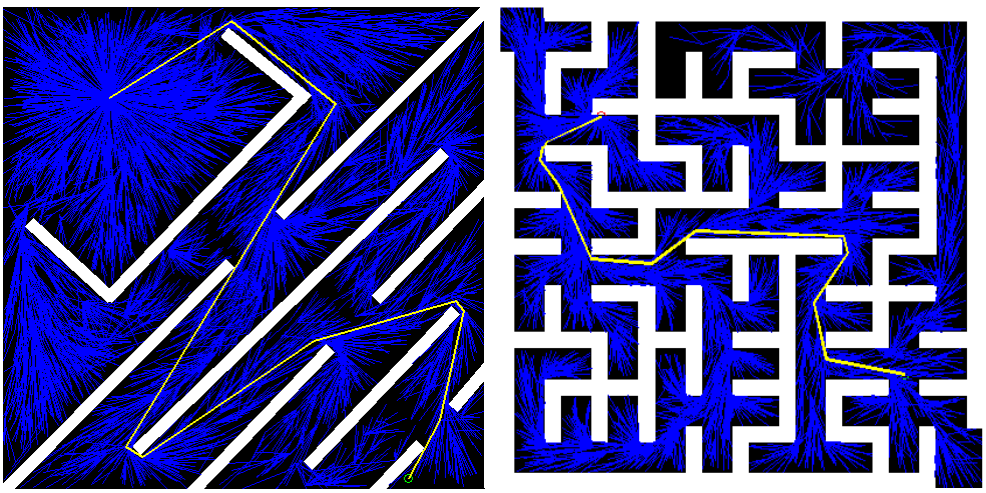

Figura 73. Ejemplos de RRT*.

6.2 Algoritmo RRT*-Smart

Este algoritmo fue presentado en *"RRT*-SMART: A Rapid Convergence Implementation of RRT*"* en el 2013 por Jauwairia Nasir, Fahad Islam y otros [16]. El propósito de este algoritmo, como ya se adelantó anteriormente, es acelerar la convergencia asintótica del RRT* mediante una heurística, evitando que el tiempo de ejecución del programa tienda a infinito.

Para alcanzar dicha optimización de forma más rápida se hace un Muestreo Inteligente. El *Muestreo Inteligente* consiste en poner *balizas* (a partir de ahora *beacons*) en los puntos de la trayectoria de ángulo considerable, y tras cada cierto número de iteraciones (parámetro del planificador) hacer un muestreo cerca de un radio de los *beacons*. Dicho muestreo forzado cerca de los *beacons* se denomina Muestreo Inteligente.

Este método garantiza el muestreo cerca de las esquinas más pronunciadas de la trayectoria, dejando tiempo también para la exploración del resto de espacio libre. Adelantar que este método da mejores resultados pero hay que escoger sabiamente el porcentaje de tiempo que queremos optimizar la trayectoria existente y explorar espacio libre.

Este es el pseudocódigo de lo arriba explicado [16]:

```
1 T ← InitializeTree();                                  // Inicializa el árbol
2 T ← InsertNode(Ø, zinit, T);                           // Inserta el punto inicial como padre raíz
3 for i=0 to i=N do                                      // Itera un número acotado de veces para limitar el tiempo
4   if i=n+b, n+2b, n+3b…. then                          // Si iteración actual es múltiplo de n + parámetro b
5     zrand ← Sample(i, zbeacons);                       // Hace un muestreo inteligente
6   else                                                 // Si no
7     zrand ← Sample(i);                                 // Igual que RRT*
8   znearest ← Nearest(T, zrand);                        // Igual que RRT*
9   (xnew, unew, Tnew) ← Steer (znearest, zrand);        // Igual que RRT*
10 if Obstaclefree(xnew) then                            // Igual que RRT*
11   Znear ← Near(T, znew, |V|);                         // Igual que RRT*
12   zmin ← Chooseparent (Znear, znearest, znew, xnew);  // Igual que RRT*
13   T ← InsertNode(zmin, znew, T);                      // Igual que RRT*
14   T ← Rewire (T, Znear, zmin, znew);                  // Igual que RRT*
15   if InitialPathFound then                            // Guarda n, iteración en la que se encuentra el camino
16     n ← i;
17   (T, directcost) ← PathOptimization(T, zinit, zgoal); // Comprueba el coste actual
18   if (directcostnew < directcostold)                  // Si es mejorable por desigualdad triangular…
19     zbeacons ← PathOptimization(T, zinit, zgoal);     // … actualiza los beacons
20   return T
```

Algorithm 2. T = *(V,E)* ← RRT*Smart(zinit)

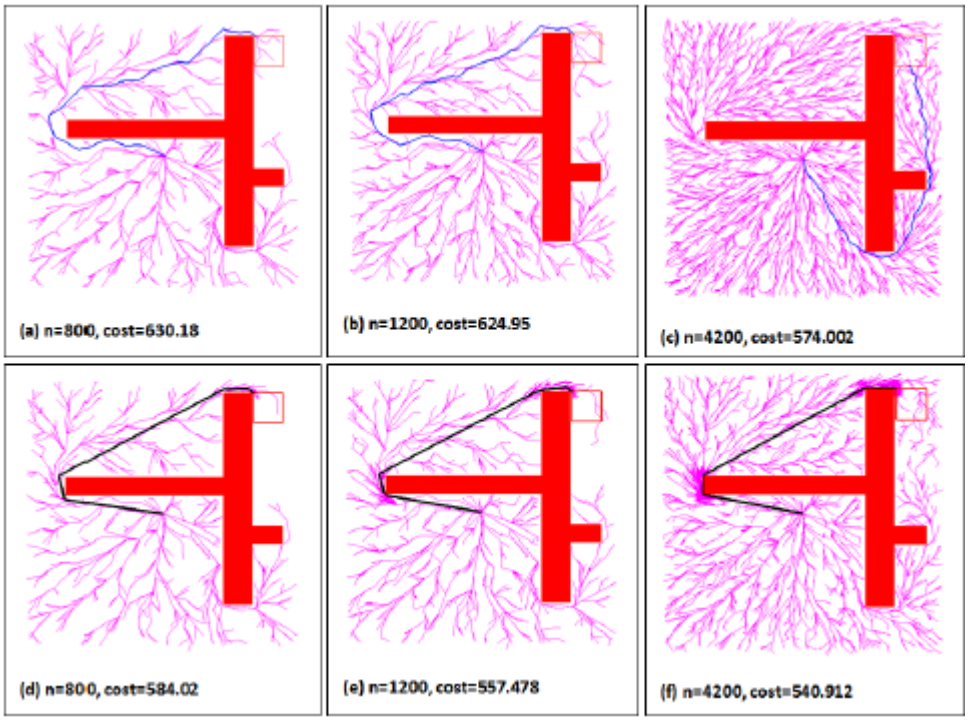

Figura 74. RRT* (arriba) vs RRT*-Smart (abajo) para 800, 1200 y 4200 iteraciones. [16]

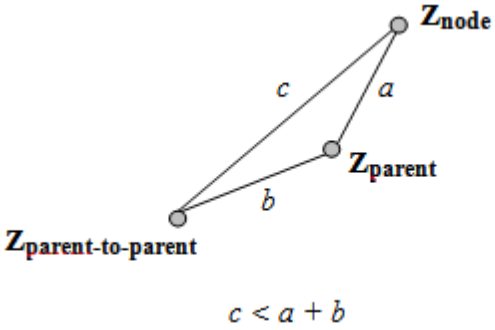

Figura 75. Optimización por desigualdad triangular, c siempre será menor que a+b. [16]

Es destacable el hecho de que los nodos de la trayectoria más angulados no se reconocen mediante cálculos de arcotangente. Dicho cálculo es demasiado pesado. Para calcular los ángulos "sin calcularlos", se recurre a comparar diferencia de vectores en valor absoluto. Esto se hace para recalcular los beacons solo cuando el recableado reduce la distancia de la trayectoria.

6.3 Algoritmo Informed RRT*

Este algoritmo fue presentado por Jonathan D. Gammell, Siddhartha S. Srinivasa, y Timothy D. Barfoot por primera vez en 2014 en *"Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic"* [17], y surge como algoritmo basado en RRT* con una heurística para minimizar su tiempo de convergencia. La idea básica es buscar el camino inicial como un RRT* normal y una vez lo encuentra calcula la elipse equivalente de dicho camino, solo muestreando puntos dentro de la misma.

Recordar que una elipse es el lugar geométrico de todos los puntos de un plano tal que la suma de todos sus puntos a dos puntos fijos (focos) es constante. Sabiendo esto, se puede asegurar que solo aquellos puntos dentro de la elipse equivalente son capaces de reducir la distancia total de la trayectoria, dando lugar a la heurística Informed.

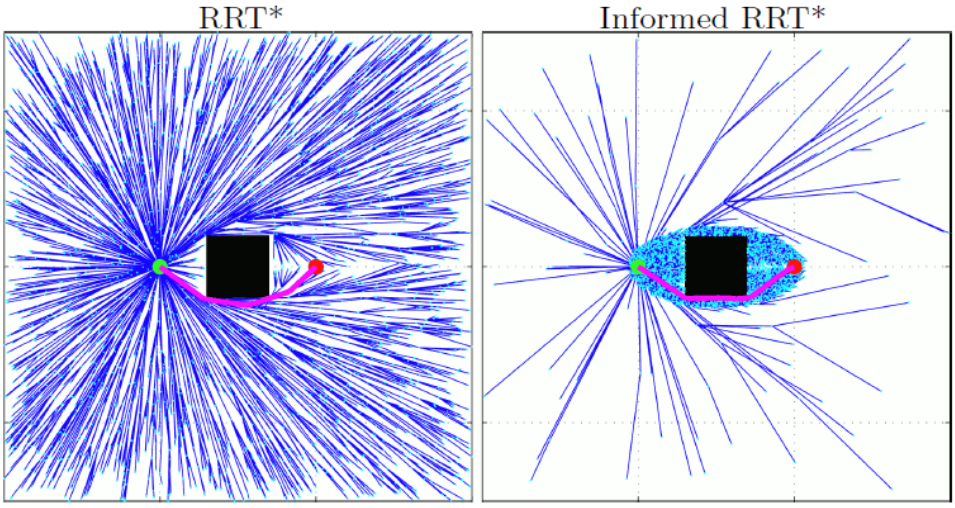

Figura 76. Comparativa entre RRT* e Informed RRT*. [17]

Se llamará *elipse equivalente de la trayectoria* a aquella elipse que contenga a la trayectoria ajustándose al máximo a la misma. El cálculo de la elipse equivalente de forma ligera computacionalmente es vital, ya que habrá que recalcularla muchas veces durante el algoritmo, y si sus cálculos resultan demasiado pesados se conseguirá más mal que bien con esta heurística.

Algunos de los parámetros de la elipse equivalente serán constantes ya que solo dependen de los puntos inicial y final, tales como su centroide, su c_{min} (distancia entre punto inicial y final), la orientación de la elipse respecto a los ejes y su matriz de rotación (servirá para muestrear dentro de la elipse si está girada). Otros parámetros serán necesarios de recalcular cada vez que se minimice el coste total de la trayectoria (tras cada recableado exitoso sobre la trayectoria), estos parámetros son c_{best} (suma de la distancia desde ambos focos hasta el punto de la trayectoria que maximiza dicha distancia) y c_{sqrt} (raíz cuadrada de c_{best} al cuadrado menos c_{min} al cuadrado).

Una vez calculada la elipse equivalente se muestrearán puntos solo dentro de la misma. El muestreo es tal que así: primero, se calcula un punto al azar en polares centrado en (0,0), de forma que su radio esté en el rango [0,1] y su ángulo entre $[0, 2\cdot\pi]$. Después dicho punto en polares se deforma según las dimensiones de la elipse equivalente. Por último, dicho punto se gira multiplicándolo por la matriz de rotación y se desplaza hacia el centroide. Tras esto ya se tiene un punto muestreado al azar dentro de la elipse equivalente, restaría comprobar que dicho punto ha caído fuera de obstáculos, si no habría que empezar de nuevo.

Con este algoritmo la elipse equivalente se va reduciendo a medida que la trayectoria se va minimizando. En una situación trivial sin obstáculos, la elipse degenerará en una recta, si esta situación ocurre debe reconocerse automáticamente para dejar de iterar inútilmente.

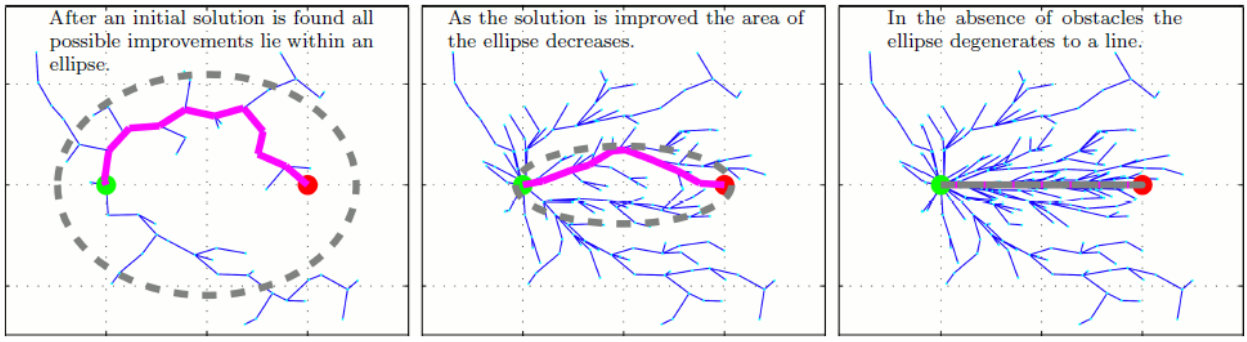

Figura 77. Ejemplo de elipse equivalente degenerando en una recta [17].

La principal diferencia y distinción del algoritmo Informed RRT* respecto al RRT* es la explicada en la página anterior. No obstante, se presentará también a continuación el pseudocódigo que describe todo ello:

Algorithm 1: Informed RRT*$(\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}})$

```
1  V ← {x_start};
2  E ← ∅;
3  X_soln ← ∅;
4  T = (V, E);
5  for iteration = 1 ... N do
6      c_best ← min_{x_soln ∈ X_soln} {Cost(x_soln)};
7      x_rand ← Sample(x_start, x_goal, c_best);
8      x_nearest ← Nearest(T, x_rand);
9      x_new ← Steer(x_nearest, x_rand);
10     if CollisionFree(x_nearest, x_new) then
11         V ← ∪ {x_new};
12         X_near ← Near(T, x_new, r_RRT*);
13         x_min ← x_nearest;
14         c_min ← Cost(x_min) + c · Line(x_nearest, x_new);
15         for ∀x_near ∈ X_near do
16             c_new ← Cost(x_near) + c · Line(x_near, x_new);
17             if c_new < c_min then
18                 if CollisionFree(x_near, x_new) then
19                     x_min ← x_near;
20                     c_min ← c_new;
21         E ← E ∪ {(x_min, x_new)};
22         for ∀x_near ∈ X_near do
23             c_near ← Cost(x_near);
24             c_new ← Cost(x_new) + c · Line(x_new, x_near);
25             if c_new < c_near then
26                 if CollisionFree(x_new, x_near) then
27                     x_parent ← Parent(x_near);
28                     E ← E \ {(x_parent, x_near)};
29                     E ← E ∪ {(x_new, x_near)};
30         if InGoalRegion(x_new) then
31             X_soln ← X_soln ∪ {x_new};
32 return T;
```

Figura 78. Figura 79. Pseudocódigo del algoritmo Informed RRT* (1) [17].

Algorithm 2: `Sample` $(\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}, c_{\max})$

1 **if** $c_{\max} < \infty$ **then**
2 $\quad c_{\min} \leftarrow \left|\left|\mathbf{x}_{\text{goal}} - \mathbf{x}_{\text{start}}\right|\right|_2$;
3 $\quad \mathbf{x}_{\text{centre}} \leftarrow \left(\mathbf{x}_{\text{start}} + \mathbf{x}_{\text{goal}}\right)/2$;
4 $\quad \mathbf{C} \leftarrow$ `RotationToWorldFrame` $\left(\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}\right)$;
5 $\quad r_1 \leftarrow c_{\max}/2$;
6 $\quad \{r_i\}_{i=2,\ldots,n} \leftarrow \left(\sqrt{c_{\max}^2 - c_{\min}^2}\right)/2$;
7 $\quad \mathbf{L} \leftarrow \text{diag}\{r_1, r_2, \ldots, r_n\}$;
8 $\quad \mathbf{x}_{\text{ball}} \leftarrow$ `SampleUnitNBall`;
9 $\quad \mathbf{x}_{\text{rand}} \leftarrow (\mathbf{C}\mathbf{L}\mathbf{x}_{\text{ball}} + \mathbf{x}_{\text{centre}}) \cap X$;
10 **else**
11 $\quad \mathbf{x}_{\text{rand}} \sim \mathcal{U}(X)$;
12 **return** $\mathbf{x}_{\text{rand}}$;

Figura 80. Pseudocódigo del algoritmo Informed RRT* (2) [17].

6.4 Algoritmo Informed RRT*-Smart multi-query + Dijkstra (2D y 3D)

6.4.1 Introducción.

Este algoritmo propuesto tiene dos fases diferenciadas:

- **Primera fase de búsqueda de camino inicial no óptimo en estático:** usando para esta fase el *algoritmo RRT multi-query + Dijkstra* se consigue en un tiempo muy reducido encontrar un primer camino que una los puntos inicial y final. Dicho camino, aunque no es óptimo, sí que garantiza una calidad mayor que la que arrojaría otro tipo de algoritmo 100% probabilístico (ver apartado 5.3.4).
- **Segunda fase de optimización del camino en estático:** trabajando sobre el camino inicial aportado por la primera fase, se *optimiza con RRT* aplicando las heurísticas Informed y Smart*. Como la cantidad de nodos del árbol se dispara en esta fase, es vital el uso de estructuras de datos eficientes, proporcionando aquí el mallado de entorno resultados casi un orden de magnitud más rápidos. El mallado de entorno aligera cálculos exhaustivos con el árbol que son cuello de botella, como pueden ser el cálculo del punto vecino más cercano, o el "Near" que proporciona los nodos cércanos a un radio dado.

La correcta sintonización de los parámetros del planificador es clave en este algoritmo, especialmente la del número de subdivisiones del mallado, longitud de rama de árbol y porcentaje de muestreos Smart.

Aunque se puede buscar el camino inicial y optimizar a la vez directamente desde el inicio del algoritmo (lo que hacen los algoritmo de optimización anteriormente presentados), los resultados son mucho más lentos, ya que un algoritmo especializado en búsqueda inicial es mucho más rápido, y un algoritmo de optimización con heurística es mucho más rápido a su vez conociendo el camino inicial. La especialización diferenciando entre fases hace a este algoritmo uno muy flexible y potente, como se demostrará en el subcapítulo final de este capítulo, dedicado a comparaciones. Si bien está limitado al problema en estático (sin cambios en el entorno ni movimiento en los puntos inicial y final), en el último capítulo se ampliará dándole capacidad de tiempo real.

6.4.2 Funciones implementadas en 2D y 3D.

Las funciones MATLAB implementadas para el caso plano (2D) y tridimensional en el espacio (3D) del algoritmo Informed RRT*-Smart multi-query + Dijkstra son las siguientes:

```
trayectoria = Planificador_Informed_RRTstar_Smart_Dijkstra(mapa, P_ini,
P_fin, muestra_animaciones, muestra_resultado, Nmax_RRT_multi_query,
Nmax_optimizacion, salto, radio, frecuencia, subdivisiones_M_optimizacion,
subdivisiones_N_optimizacion, grafo, dist_grafo);
```

```
trayectoria = Planificador_Informed_RRTstar_Smart_Dijkstra_3D(mapa, P_ini,
P_fin, muestra_animaciones, muestra_resultado, Nmax_RRT_multi_query,
Nmax_optimizacion, salto, radio, frecuencia, subdivisiones_M_optimizacion,
subdivisiones_N_optimizacion, subdivisiones_Z_optimizacion,grafo,dist_grafo);
```

Reciben como entrada el mapa de obstáculos (matriz de 2 dimensiones para el caso plano y de 3 dimensiones para el espacio), los puntos inicial y final de la trayectoria, las variables booleanas muestra_animaciones y muestra_resultado (que valdrán 1 si se desea ver la evolución de la optimización y ver el resultado respectivamente, y 0 si no se desea), el número máximo de iteraciones del RRT multi-query (a partir del cual se considerará que no existe camino posible), el número máximo de iteraciones o nodos de optimización, el salto o longitud de rama del árbol, el radio de optimización del "rewire" del RRT*, la frecuencia de sampleado de punto Smart, las subdivisiones del mallado de entorno, y por último el grafo para el Dijkstra (dist_grafo contiene los costes de los arcos).

Como salida proporciona los waypoints a seguir en una matriz llamada "trayectoria". Una vez optimizada la trayectoria, se le aplica un postprocesado de desigualdad triangular para un ajuste todavía más fino. Dicha trayectoria puede estar muy cercana al óptimo en poco tiempo si se configura correctamente el planificador.

6.4.3 Resultados 2D: moviéndonos de una planta a otra de la ETSI.

Para la presentación de resultados en 2D de los algoritmos propuestos se ha optado por algo original, resolver un problema de planificación de caminos real al que nos enfrentamos día a día en la ETSI (Escuela Técnica Superior de Ingeniería) de Sevilla: movernos de un lugar de una planta a otro sitio en una planta distinta, es decir, el punto inicial de la trayectoria está en una planta y el punto final en otra.

Para tal fin, se ha supuesto que el robot móvil no puede usar escaleras, pudiendo solo moverse entre plantas con ascensores, por supuesto teniendo en cuenta que no todos los ascensores paran en todas las plantas. Como la combinación de trayectorias entre plantas es infinita, se ha supuesto una heurística, que al fin y al cabo es la que usamos nosotros los humanos para movernos de una planta a otra, y es que el ascensor que se coge normalmente es el más cercano al punto inicial o al punto final. De esta forma, se tiene que en una de las plantas el robot buscará el ascensor más cercano, desplazándose poco, mientras que en otra planta se desplazará una distancia mucho mayor.

Pero, ¿en qué planta coger el ascensor más cercano? La respuesta más completa es calcular ambas situaciones y quedarte con la de menor distancia, mientras que también cabría plantearse una segunda heurística de preferencia entre plantas. Por ejemplo, la Planta Baja presenta más movilidad que la Entreplanta 1, ya que sus pasillos son mucho más amplios y hay más conexiones, por lo que el desplazamiento al ascensor más cercano se haría en la Entreplanta 1, mientras que la mayor cantidad de desplazamiento se haría en la Planta Baja.

En este Proyecto se han implementado ambas posibilidades, la de menor distancia calculando ambas situaciones y la de heurística de preferencia de circulación.

En las siguientes páginas se podrán ver los problemas de planificación entre distintas plantas resueltos por el algoritmo propuesto en este subcapítulo, pudiendo verse sus tiempos de ejecución (de cada una de las fases por separado así como el tiempo total) y costes de trayectoria (tras fase de búsqueda de camino inicial y tras optimización) en el título de las figuras presentadas.

Aunque en algunos pies de figura digan lo contrario, siempre la dirección de expansión del árbol de optimización ha sido desde los puntos inicial o final hacia los ascensores, al fin y al cabo las trayectorias (óptimas en este caso) sin condiciones cinemáticas ni dinámicas son reversibles. Esto se podría cambiar con pocas modificaciones. En dichas figuras se podrá apreciar la optimalidad de las trayectorias solución presentadas, así como sus cortísimos tiempos de ejecución.

Yendo del S1-Sur de la Planta Sótano a las vitrinas de la Planta Ático:

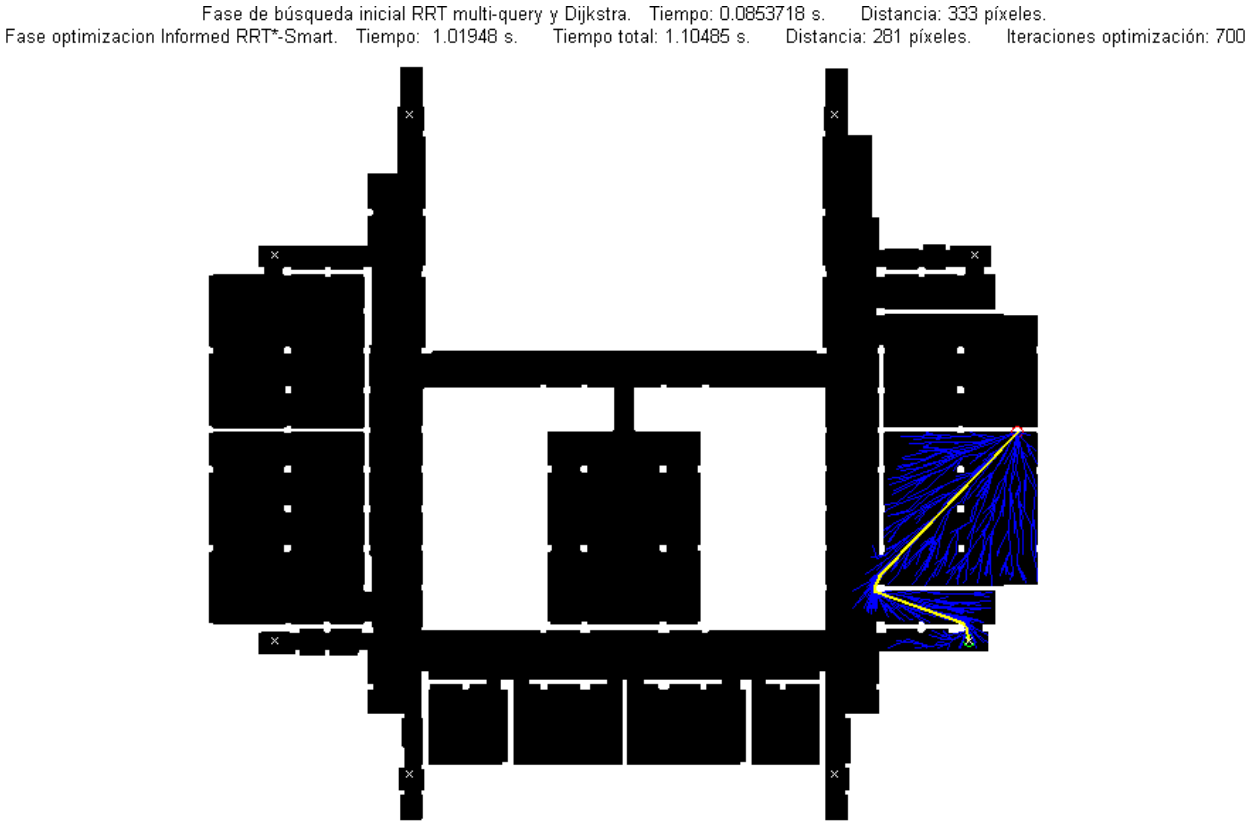

Figura 81. Planta Sótano. Saliendo de S1-Sur hasta el ascensor suresteeste.

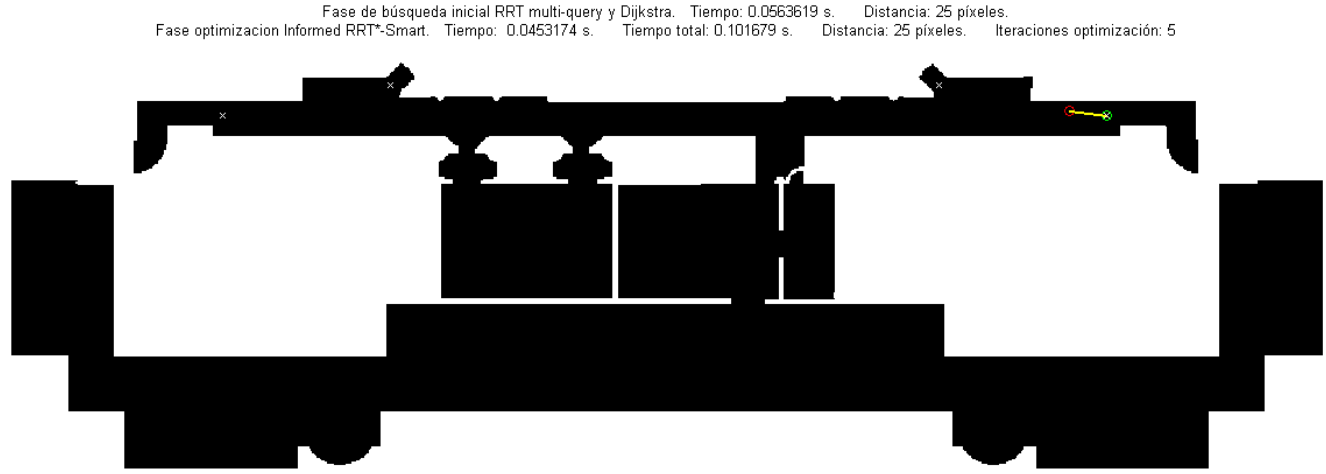

Figura 82. Planta Ático. Yendo del ascensor suresteeste a las vitrinas.

Se puede apreciar la forma elipsoidal del árbol de optimización en la primera figura, fruto del algoritmo heurístico Informed RRT*, y la mayor densidad de nodos en la esquina de mayor ángulo de la trayectoria, fruto de la heurística Smart. En la segunda figura la trayectoria era trivial, por lo que ha convergido al óptimo de forma casi instantánea, parándose automáticamente al degenerar la elipse equivalente en una recta.

En este ejemplo se han usado 700 nodos de optimización. Se puede confirmar la potencia del algoritmo cuando en tan poco tiempo ha dado el óptimo en mapas tan complejos, irregulares y reales como son las plantas de un edificio. Otros algoritmos que necesitasen explorar los límites de los obstáculos, con obstáculos tan complejos, hubieran tardado mucho más y hubieran caído víctimas de los mínimos locales.

Yendo de la escalera nornoroeste de la Primera Planta al aula 108 de la Entreplanta 1:

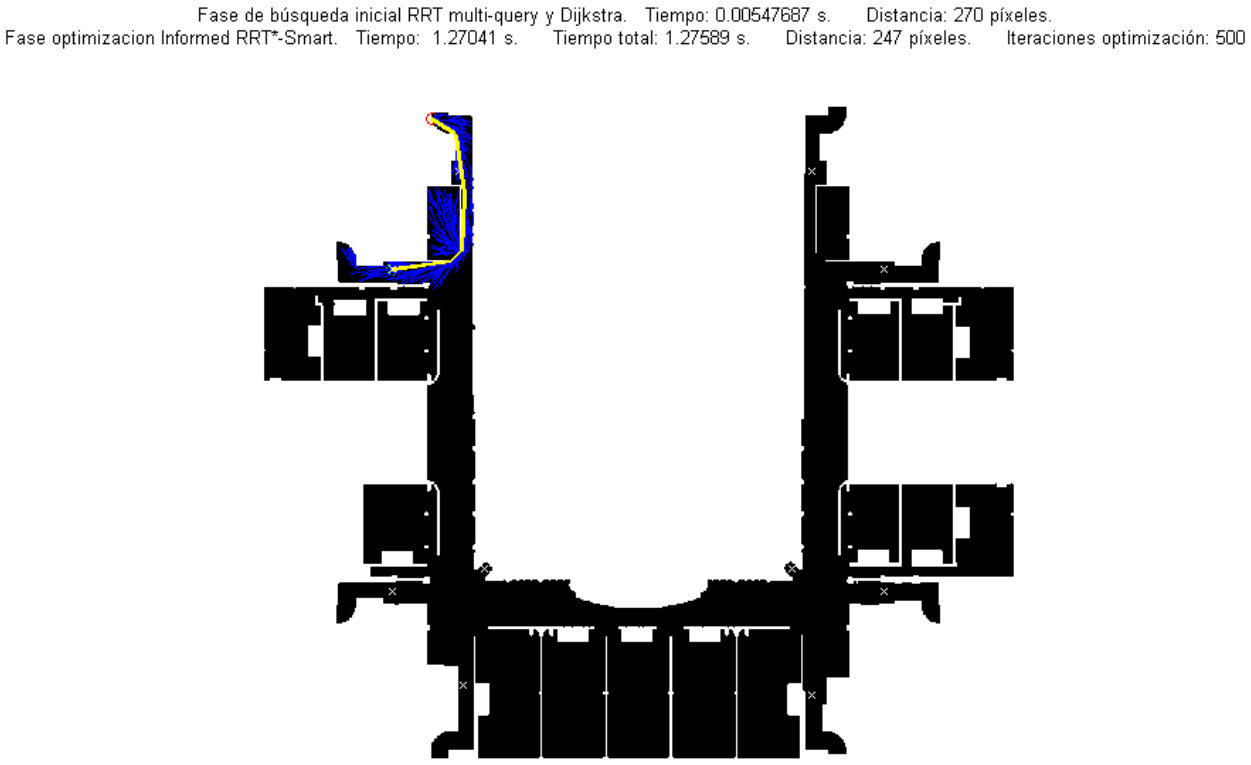

Figura 83. Primera Planta. Saliendo por el ascensor nornoroeste.

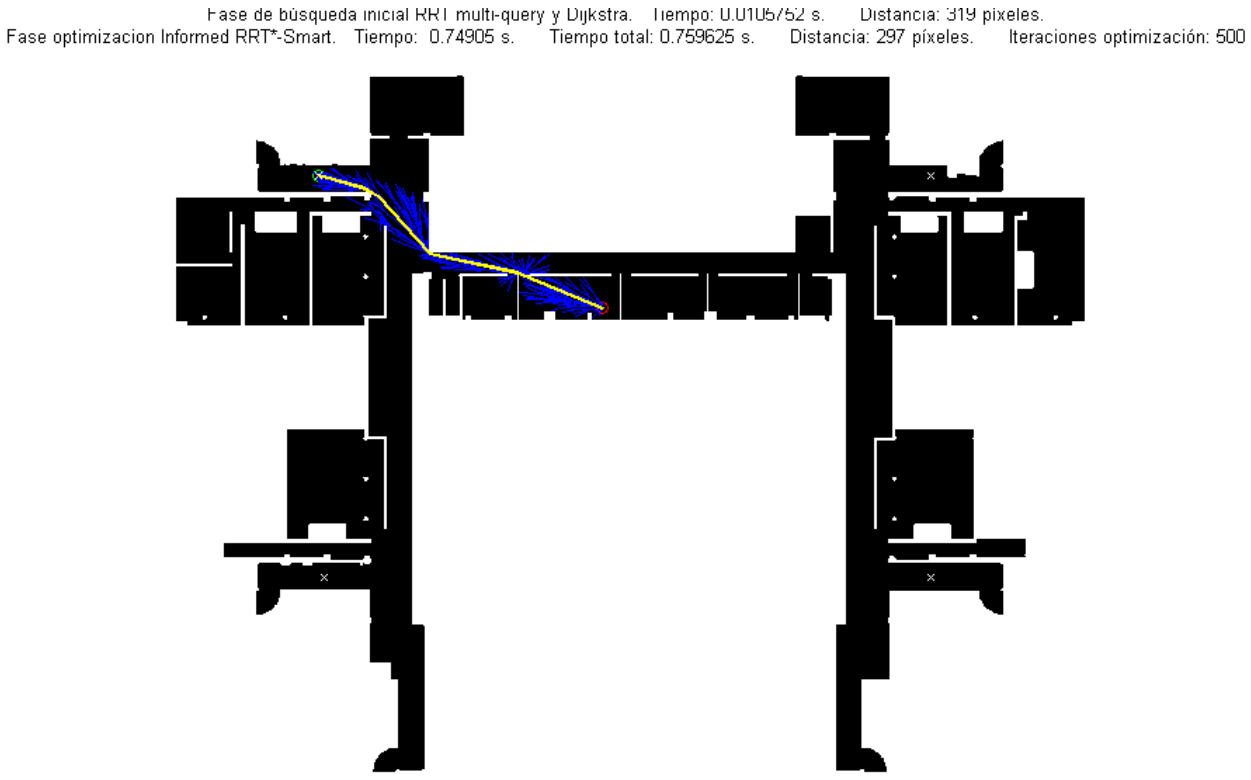

Figura 84. Entreplanta 1. Yendo del ascensor nornoroeste hacia el aula 108.

Igualmente, es fácilmente apreciable la forma elipsoidal del árbol de optimización en ambas imágenes, esta vez con menos iteraciones de optimización (500) e igualmente convergiendo al óptimo.

Yendo de copistería (Planta Baja) al aula 312 de la Entreplanta 2:

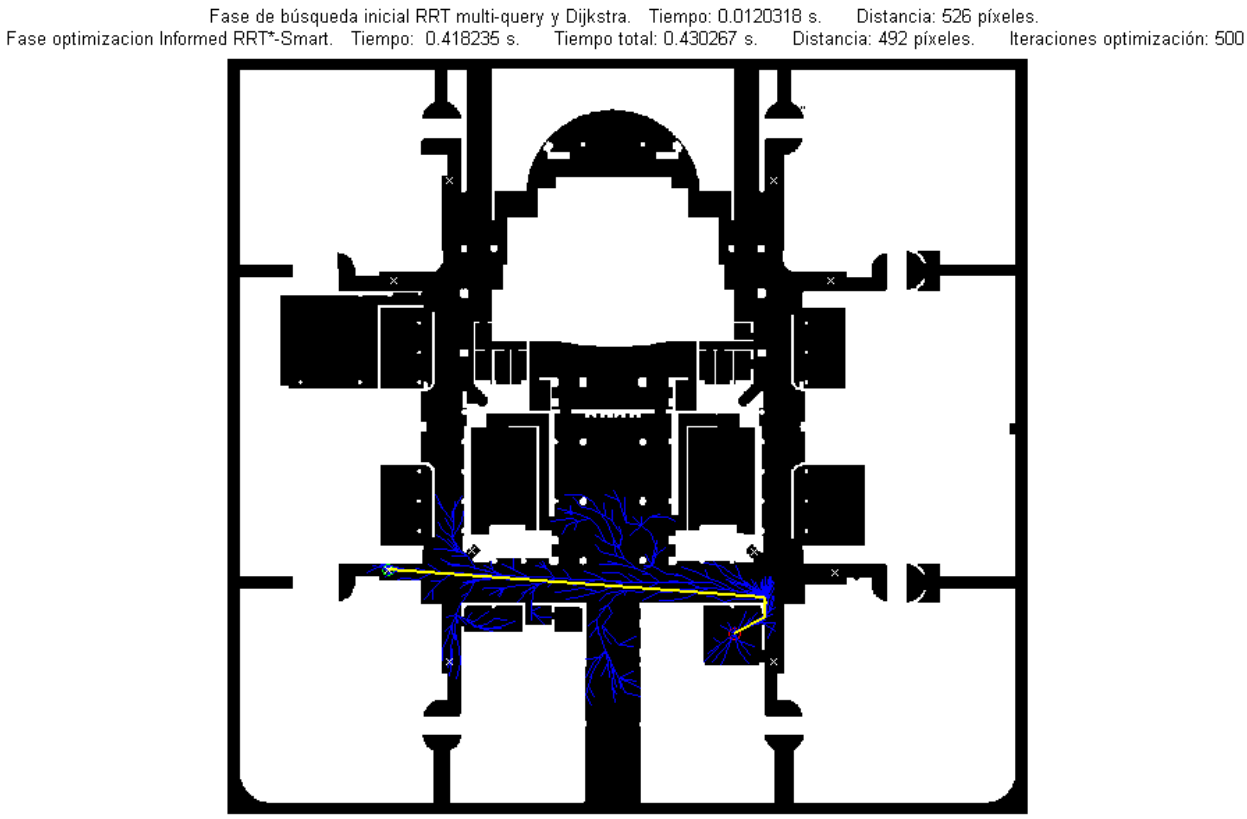

Figura 85. Planta Baja. Saliendo de copistería hacia el ascensor suroesteoeste.

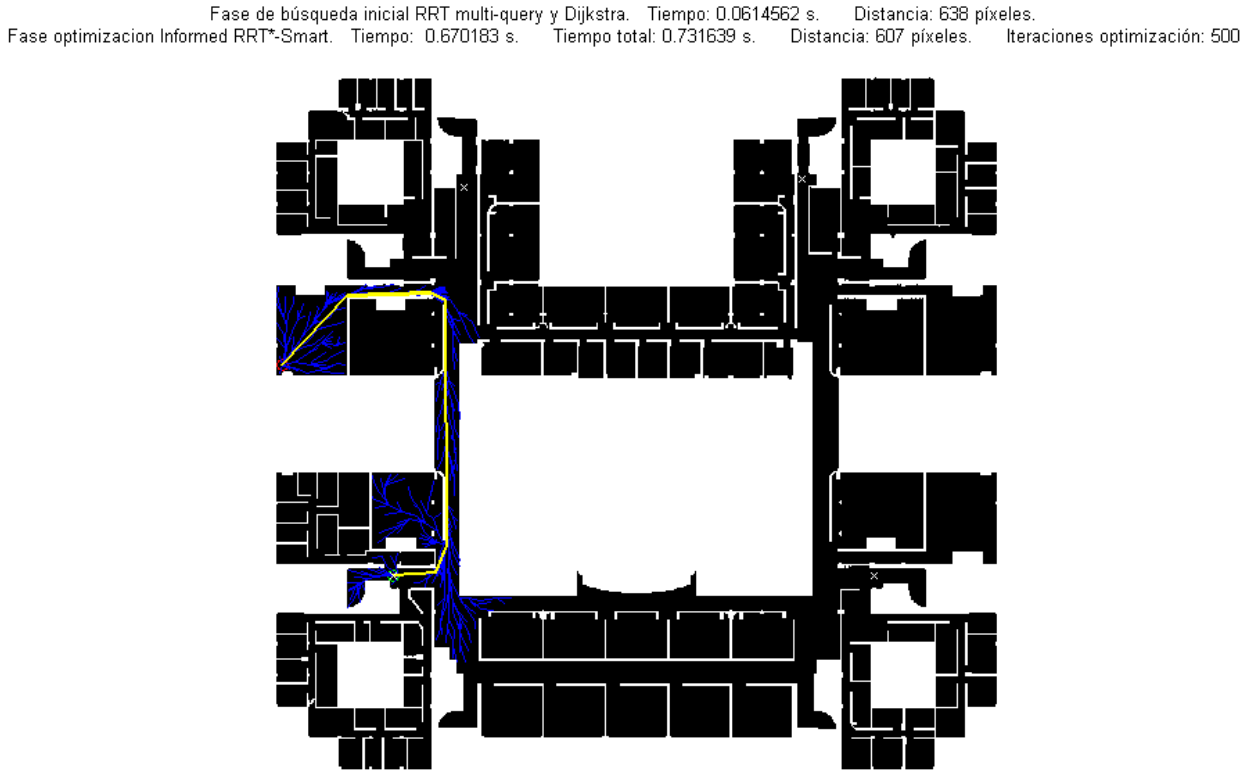

Figura 86. Entreplanta 2. Yendo del ascensor suroesteoeste al aula 312 de la Entreplanta 2.

Yendo desde el gimnasio (fuera del edificio) hasta la Sala de Estudio o S3-Norte de la Planta Sótano:

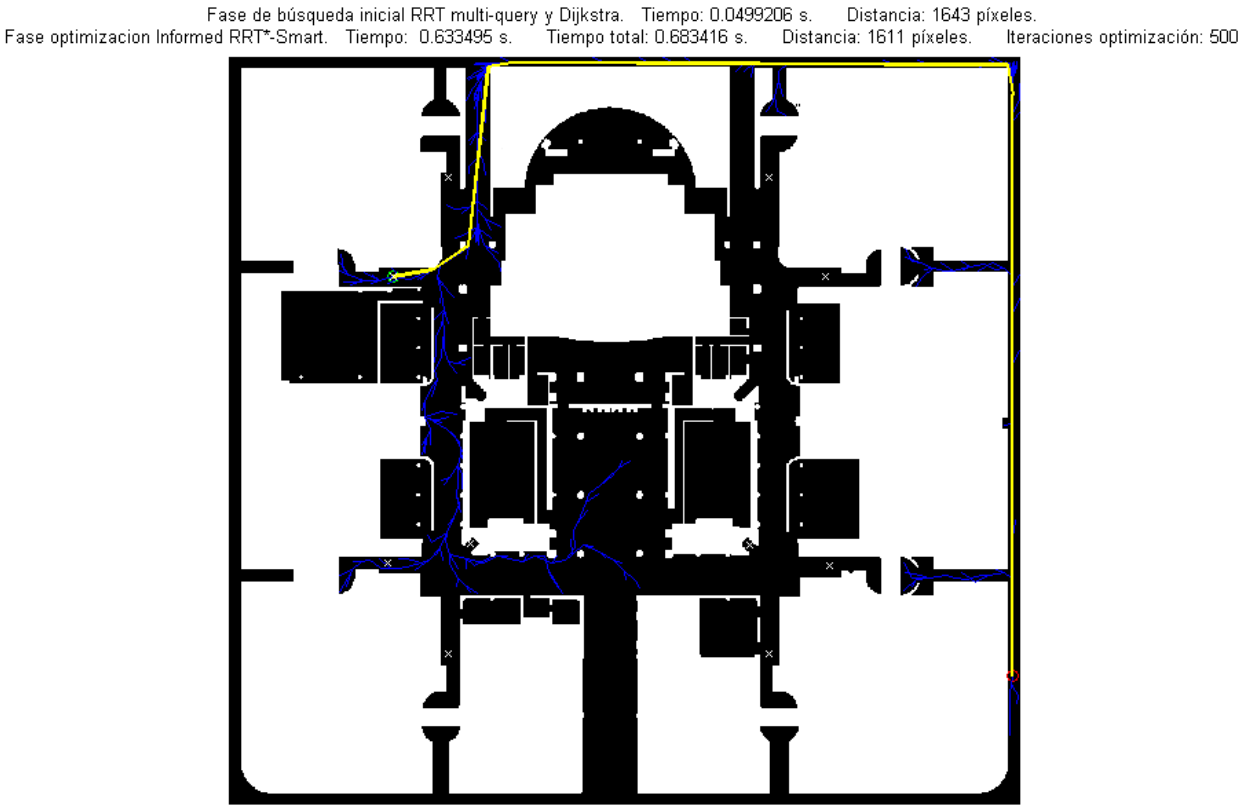

Figura 87. Planta Baja. Entrando a la ETSI bordeándola por fuera para entrar por la rampa noroeste y coger el ascensor noroesteoeste.

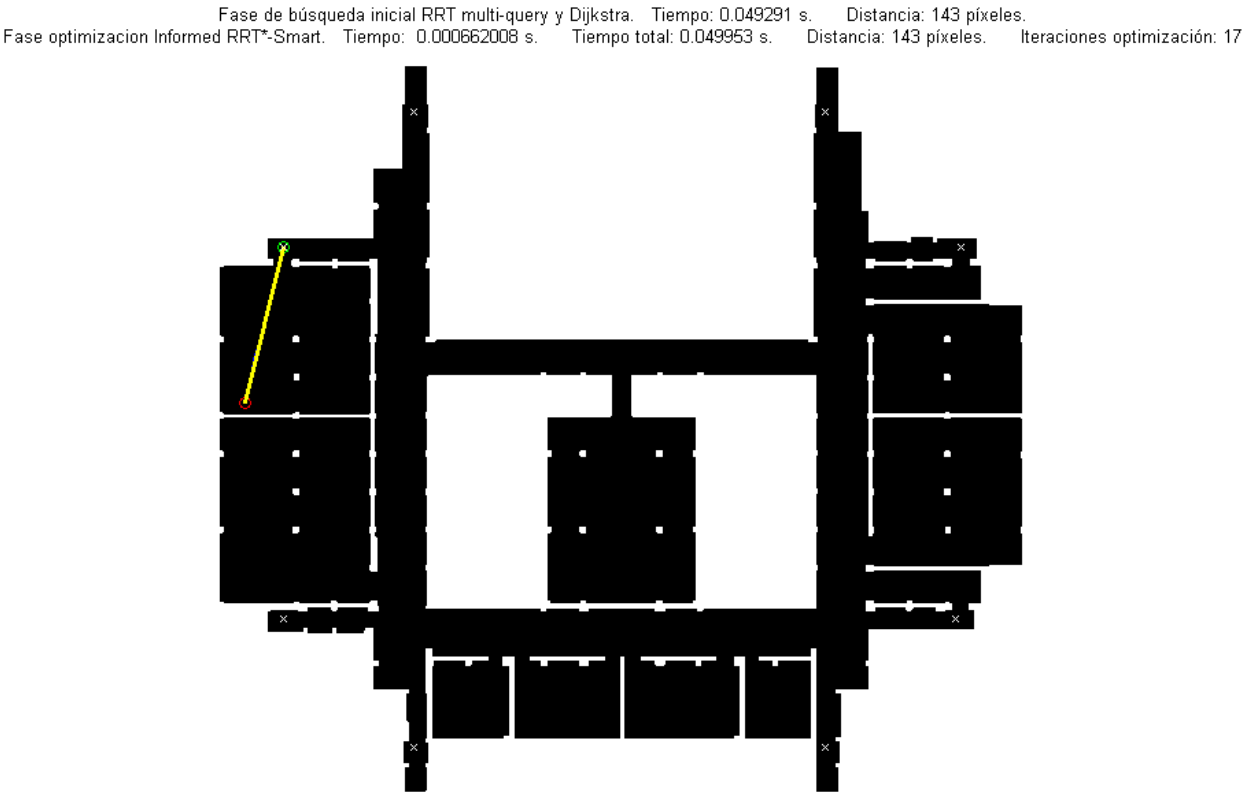

Figura 88. Planta Sótano. Entrando desde el ascensor noroesteoeste a la Sala de Estudio o S3-Norte.

6.4.4 Resultados 3D.

Para los resultados en 3D se ha usado el modelo de paredes con perforaciones aleatorias, con los nodos del grafo preexistente (para Dijkstra) a cada lado de la perforación. Los resultados son los siguientes para distintas iteraciones de optimización:

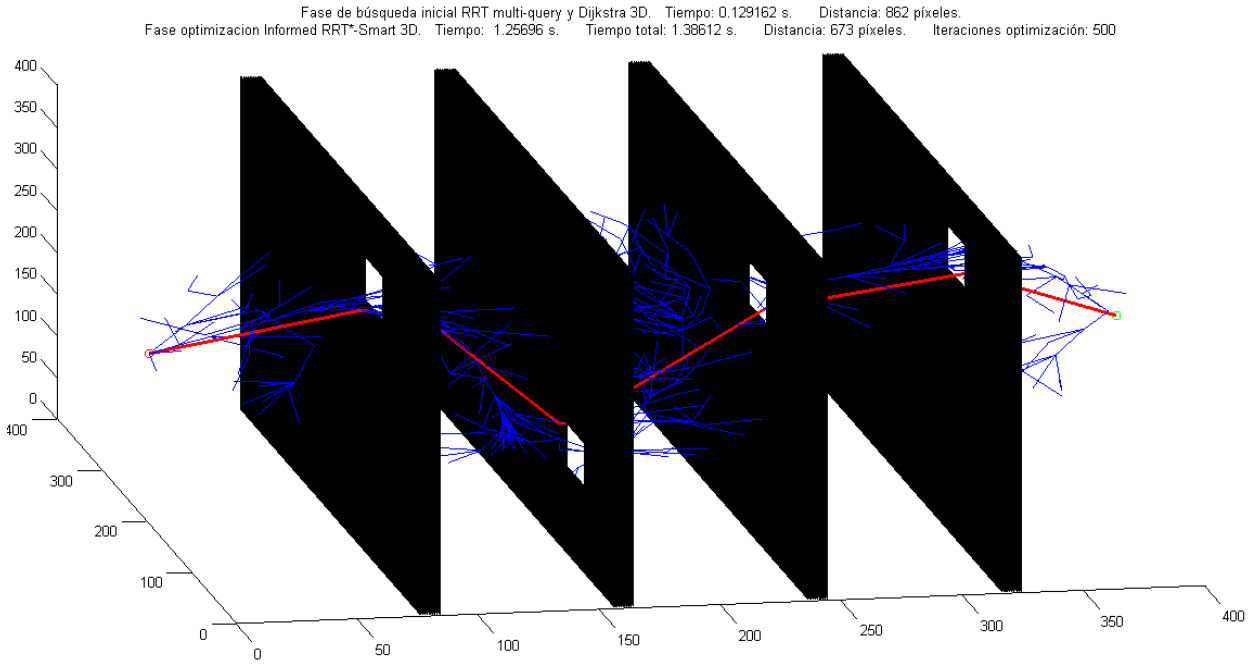

Figura 89. Informed RRT*-Smart multi-query + Dijkstra en 3D, 500 iteraciones (nodos) de optimización.

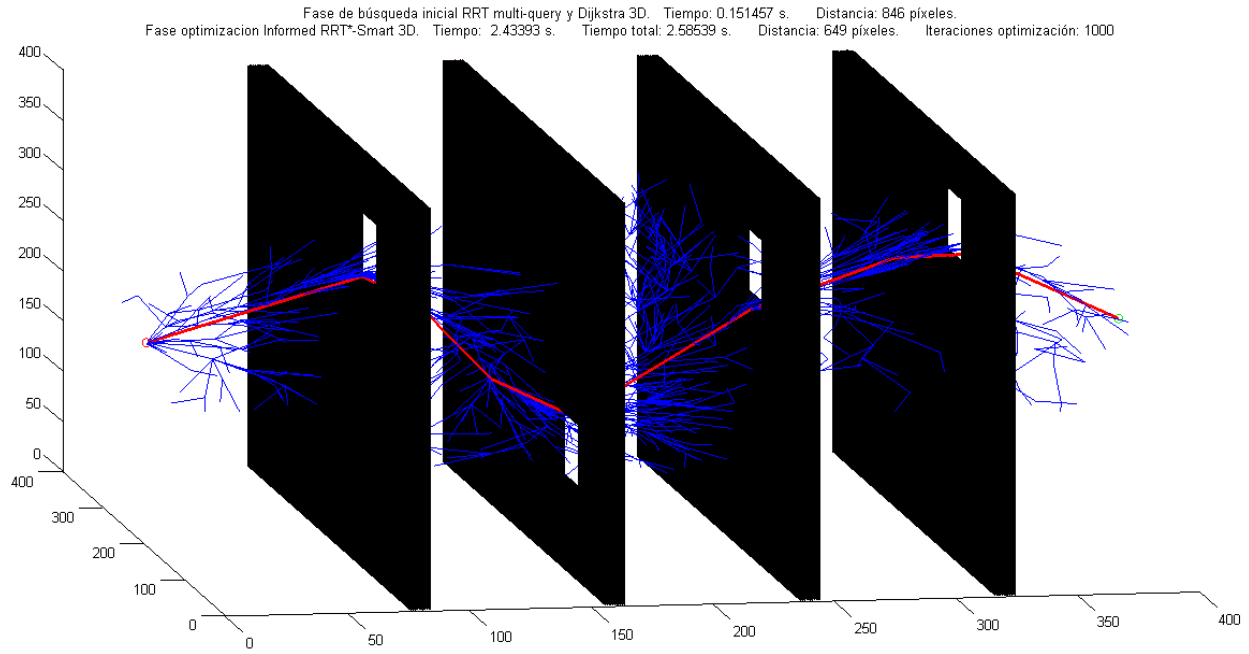

Figura 90. Informed RRT*-Smart multi-query + Dijkstra en 3D, 1000 iteraciones (nodos) de optimización.

6.5 Algoritmo Informed RRT*-Smart Connect (2D y 3D)

Este algoritmo propuesto es muy similar al anterior, diferenciándose en su fase de búsqueda del camino inicial:

- **Primera fase de búsqueda de camino inicial no óptimo en estático:** usando para esta fase el *algoritmo RRT-Connect* se consigue encontrar un primer camino que una los puntos inicial y final. Dicho algoritmo resulta menos rápido que el RRT multi-query + Dijkstra y se comporta peor ante mínimos locales, pero resulta muy útil en casos que no se pueda disponer de grafo predefinido del entorno. En esta fase también se empleará mallado de entorno, ya que es posible que al RRT-Connect le cueste más de la cuenta encontrar el primer camino.
- **Segunda fase de optimización del camino en estático:** exactamente igual que en el algoritmo anteriormente propuesto. Trabajando sobre el camino inicial aportado por la primera fase, se *optimiza con RRT* aplicando las heurísticas Informed y Smart*. Como la cantidad de nodos del árbol se dispara en esta fase, es vital el uso de estructuras de datos eficientes, proporcionando aquí el mallado de entorno resultados casi un orden de magnitud más rápidos. El mallado de entorno aligera cálculos exhaustivos con el árbol que son cuello de botella, como pueden ser el cálculo del punto vecino más cercano, o el "Near" que proporciona los nodos cércanos a un radio dado.

La correcta sintonización de los parámetros del planificador es clave en este algoritmo, especialmente la del número de subdivisiones del mallado, longitud de rama de árbol y porcentaje de muestreos Smart.

Aunque se puede buscar el camino inicial y optimizar a la vez directamente desde el inicio del algoritmo (lo que hacen los algoritmo de optimización RRT*, RRT*-Smart e Informed RRT*), los resultados son mucho más lentos, ya que un algoritmo especializado en búsqueda inicial es mucho más rápido, y un algoritmo de optimización con heurística es mucho más rápido a su vez conociendo el camino inicial. La especialización diferenciando entre fases hace a este algoritmo uno muy flexible y potente, como se demostrará en el subcapítulo final de este capítulo, dedicado a comparaciones.

6.5.1 Funciones implementadas en 2D y 3D.

Las funciones MATLAB implementadas para el caso plano (2D) y tridimensional en el espacio (3D) del algoritmo Informed RRT*-Smart Connect son las siguientes:

```
trayectoria = Planificador_Informed_RRTstar_Smart_connect(mapa, P_ini, P_fin,
muestra_animaciones, muestra_resultado, Nmax_connect, Nmax_optimizacion,
salto, radio, frecuencia, subdivisiones_M_connect, subdivisiones_N_connect,
subdivisiones_M_optimizacion, subdivisiones_N_optimizacion);
```

```
trayectoria = Planificador_Informed_RRTstar_Smart_connect_3D(mapa, P_ini,
P_fin, muestra_animaciones, muestra_resultado, Nmax_connect,
Nmax_optimizacion, salto, radio, frecuencia, subdivisiones_M_connect,
subdivisiones_N_connect, subdivisiones_Z_connect,
subdivisiones_M_optimizacion, subdivisiones_N_optimizacion,
subdivisiones_Z_optimizacion);
```

Reciben como entrada el mapa de obstáculos (matriz de 2 dimensiones para el caso plano y de 3 dimensiones para el espacio), los puntos inicial y final de la trayectoria, las variables booleanas muestra_animaciones y muestra_resultado (que valdrán 1 si se desea ver la evolución de la optimización y ver el resultado respectivamente, y 0 si no se desea), el número máximo de iteraciones del RRT Connect (a partir del cual se considerará que no existe camino posible), el número máximo de iteraciones o nodos de optimización, el salto o longitud de rama del árbol, el radio de optimización del "rewire" del RRT*, la frecuencia de muestreado Smart, las subdivisiones del mallado de entorno Connect y las subdivisiones de mallado de optimización.

Como salida proporciona los waypoints a seguir en una matriz llamada "trayectoria". Una vez optimizada la trayectoria, se le aplica un postprocesado de desigualdad triangular para un ajuste todavía más fino. Dicha trayectoria puede estar muy cercana al óptimo en poco tiempo si se configura correctamente el planificador.

6.5.2 Resultados 2D: moviéndonos de una planta a otra de la ETSI.

Yendo desde fuera del edificio hasta el aula 213 de la Primera Planta.

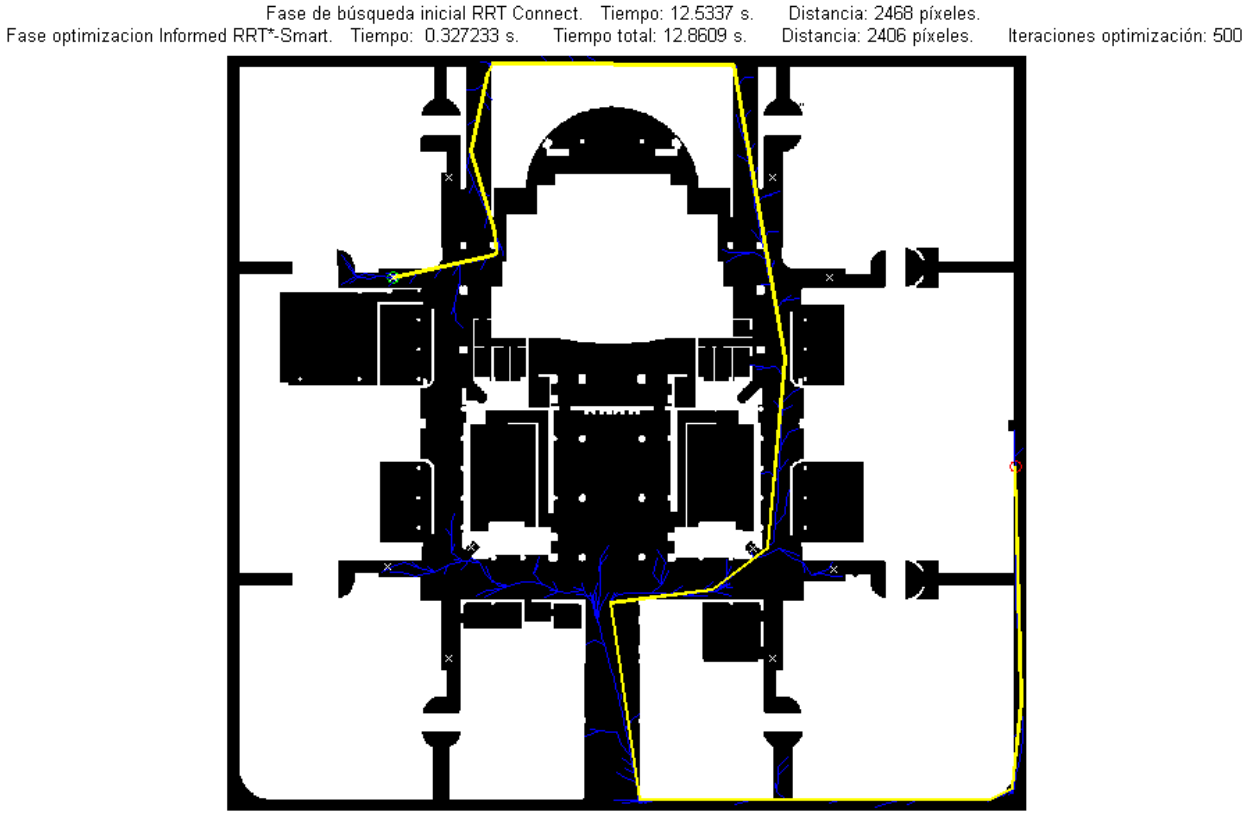

Figura 91. Planta Baja. La fase inicial RRT-Connect ha dado un camino muy ineficiente debido a su naturaleza aleatoria, entrando y saliendo varias veces sin sentido del edificio, tan ineficiente que la fase de optimización no ha podido corregirlo. Además, ha tardado 12 segundos.

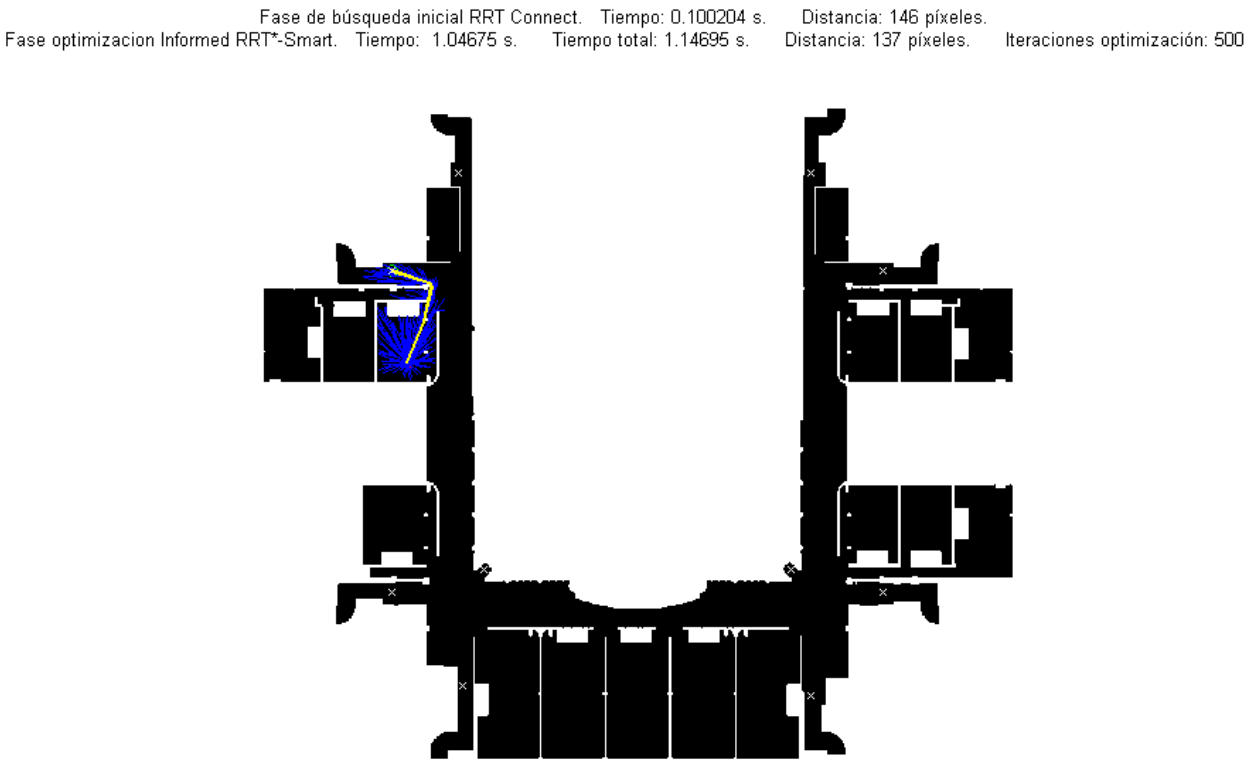

Figura 92. Primera Planta. Desde el ascensor noroesteoeste entra al aula 213.

Yendo desde el Departamento de Telemática (Entreplanta 2) hasta la terraza de la Planta Ático.

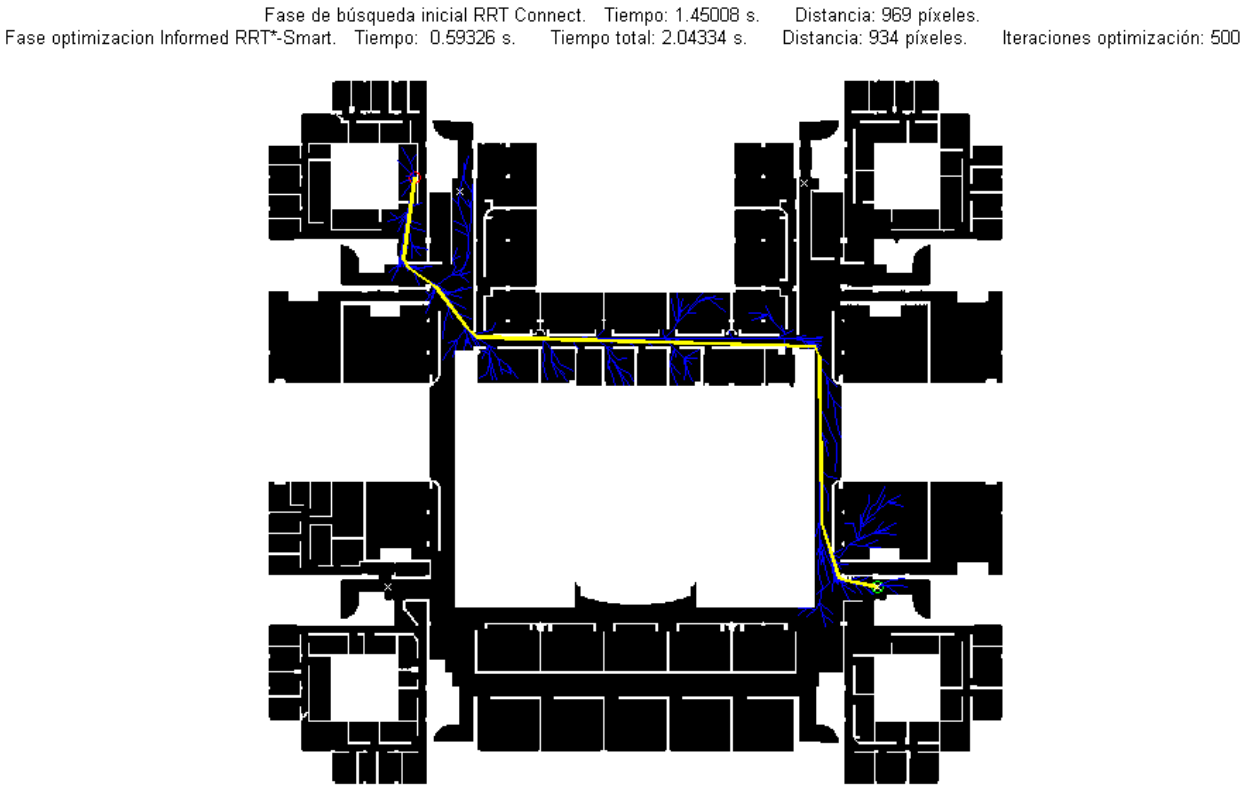

Figura 93. Entreplanta 2. Saliendo del Departamento de Telemática hacia el ascensor suresteeste.

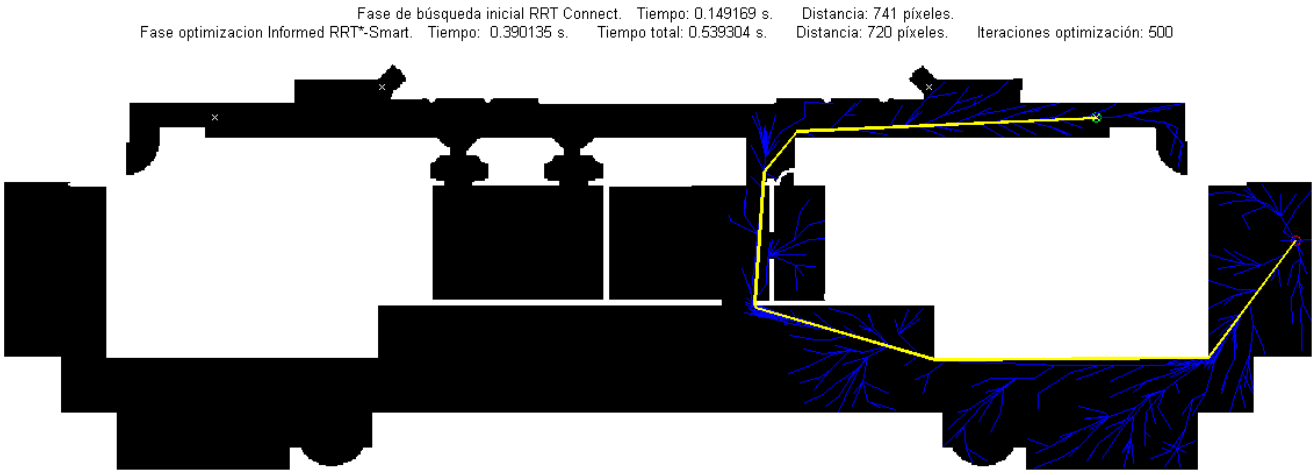

Figura 94. Planta Ático. Yendo desde el ascensor suresteeste hacia la terraza.

Yendo desde la Planta Baja a la cafetería de la Planta Ático con el ascensor panorámico.

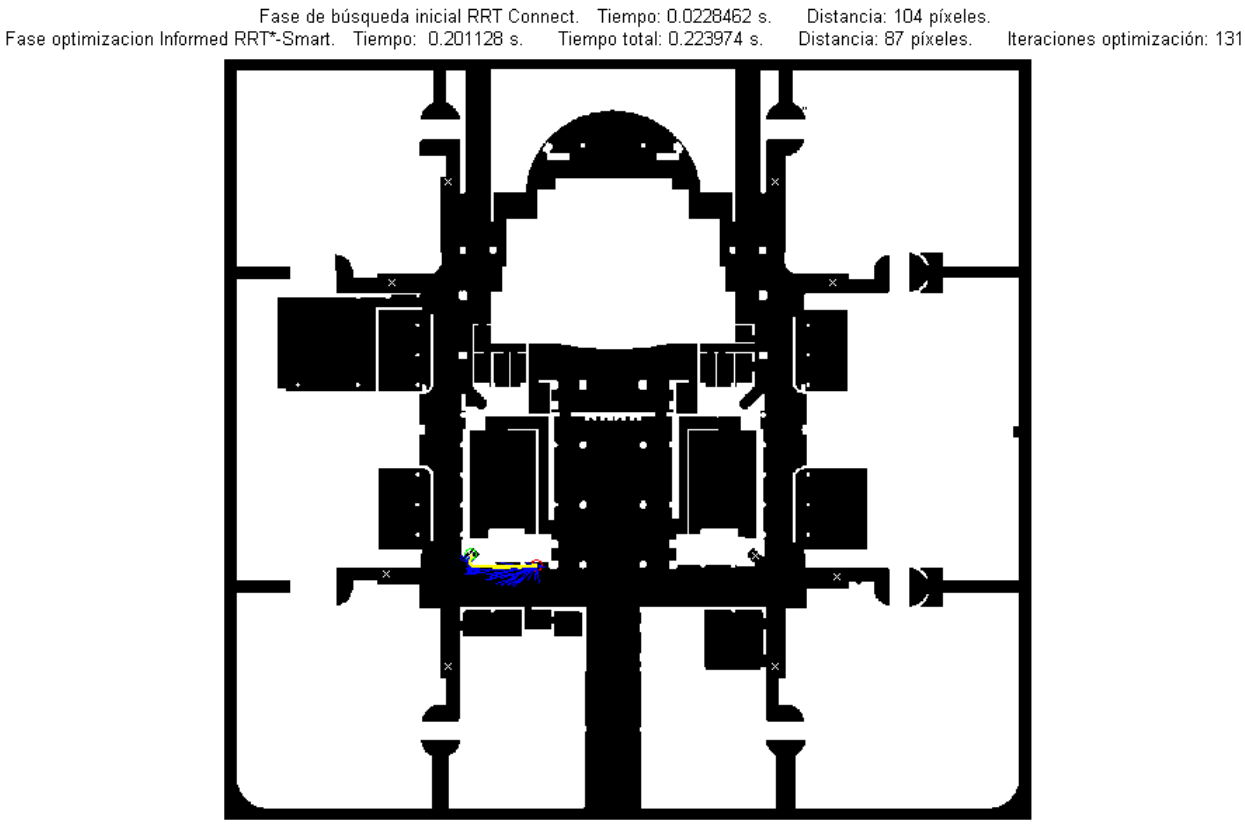

Figura 95. Planta Baja. Cogiendo el ascensor panorámico.

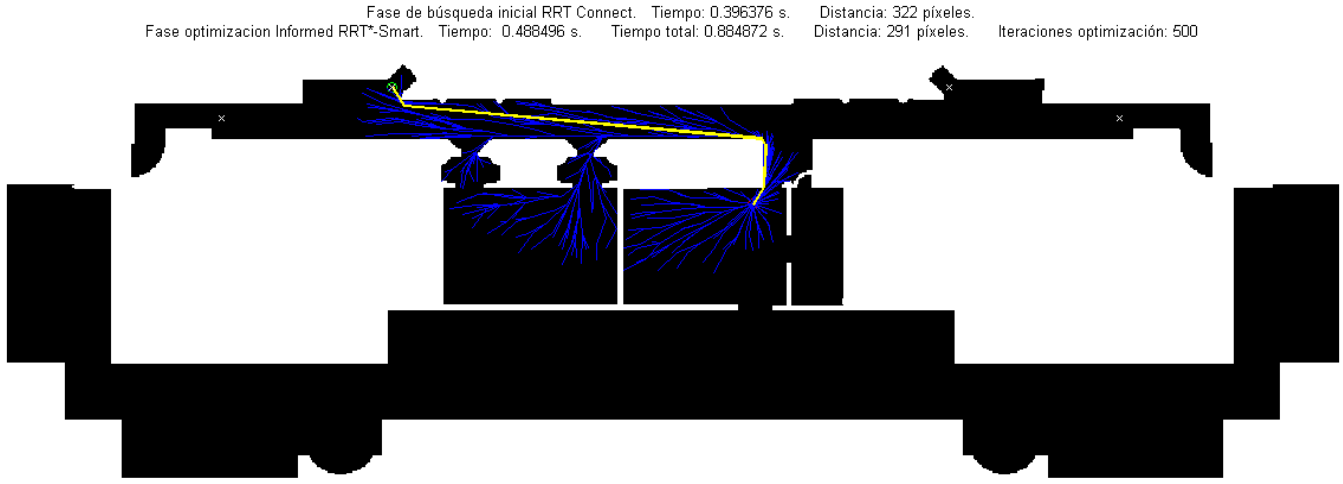

Figura 96. Planta Ático. Entrando en la cafetería desde el ascensor panorámico.

A la vista de todos estos ejemplos, se puede concluir lo que ya se anticipó con anterioridad, este algoritmo propuesto es mucho más lento que el propuesto en el subcapítulo anterior. De ser posible, merece la pena crear el grafo predefinido para aplicar RRT multi-query + Dijkstra. Si no se puede, este algoritmo ha demostrado ser viable en la práctica, pero con tiempos de ejecución más variables y mayores, así como con mayor distancia.

6.5.3 Resultados 3D.

Para los resultados en 3D se ha usado el modelo de paredes con perforaciones aleatorias. Los resultados son los siguientes para distintas iteraciones de optimización:

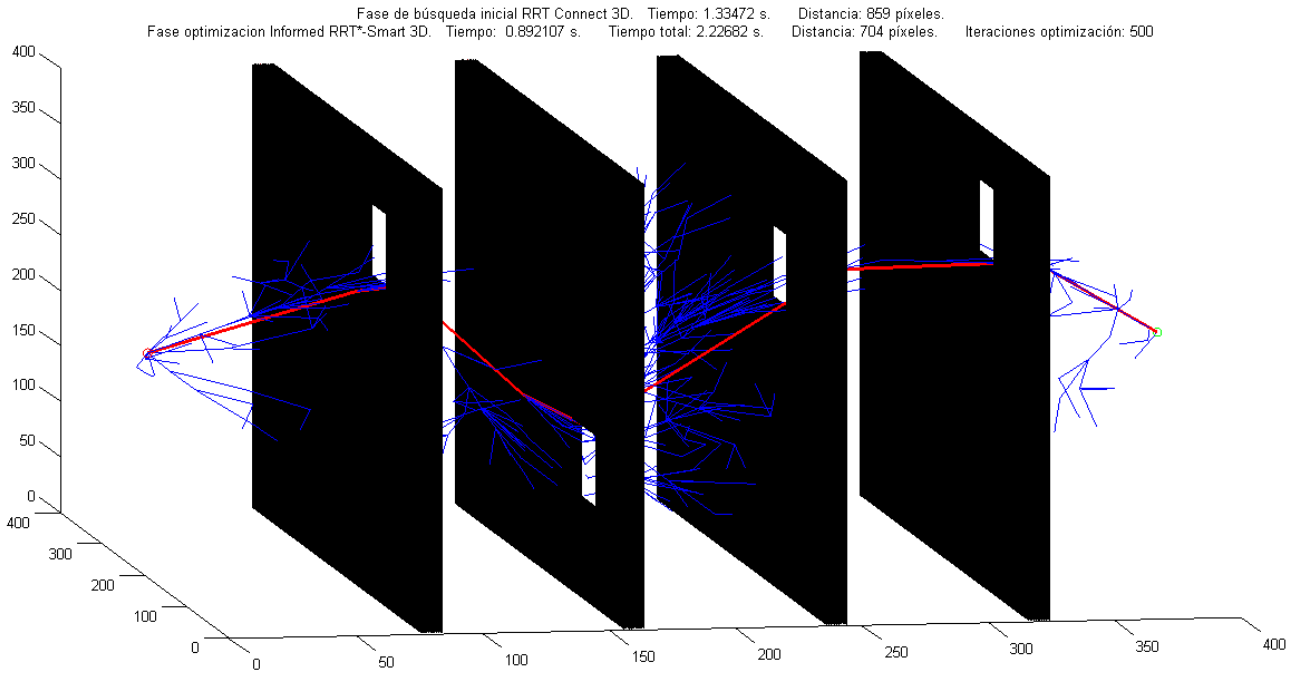

Figura 97. Informed RRT*-Smart Connect en 3D, 500 iteraciones (nodos) de optimización.

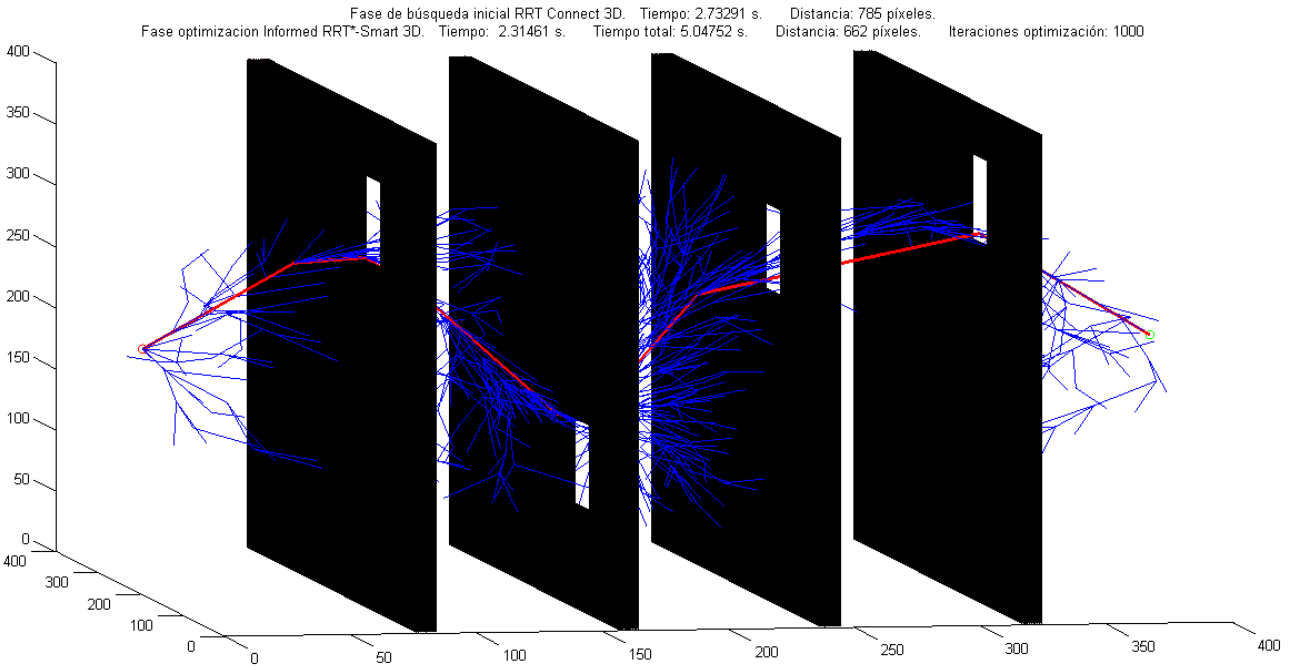

Figura 98. Informed RRT*-Smart Connect en 3D, 1000 iteraciones (nodos) de optimización.

Como se puede comprobar, con RRT-Connect también se dan en 3D tiempos de ejecución considerablemente más altos que con RRT multi-query + Dijkstra. Y no se dan costes menores.

6.6 Comparación con análisis Montecarlo.

La idea es comparar cada uno de los algoritmos de optimización en estático vistos en este capítulo. Para este fin se hará un análisis Montecarlo con ellos, tanto por separado como combinándolos entre sí, en dos problemas distintos:

- Uno de ellos complicado: planificación de caminos óptima desde una punta a otra de la Entreplanta 2.
- El otro sencillo: planificación de caminos óptima desde copistería hasta secretaría.

El análisis Montecarlo ejecutará 30 veces cada algoritmo en cada problema, recogiendo cada 100 iteraciones de optimización el tiempo de ejecución (segundos) y el coste (distancia en píxeles) del camino, si se ha encontrado. Esta información se exporta al programa de hojas de cálculo Microsoft Excel [46], donde se estima los valores medios de los pares tiempo-distancia. Dichos pares tiempo-distancia representan puntos graficables, que una vez representados servirán para comparar de forma consistente los algoritmos.

Problema sencillo: planificación de caminos óptima desde copistería hasta secretaría.

Se espera que todos los algoritmos converjan asintóticamente al óptimo, siendo el RRT* puro sin heurística el más lento, y convergiendo más rápido a más heurísticas se implementen.

Se espera también que los algoritmos que implementan fase de búsqueda inicial especializada (Connect o multi-query + Dijkstra) necesiten mucho menos tiempo en arrojar una primera solución, siendo multi-query + Dijkstra el más rápido. Los resultados obtenidos son los siguientes:

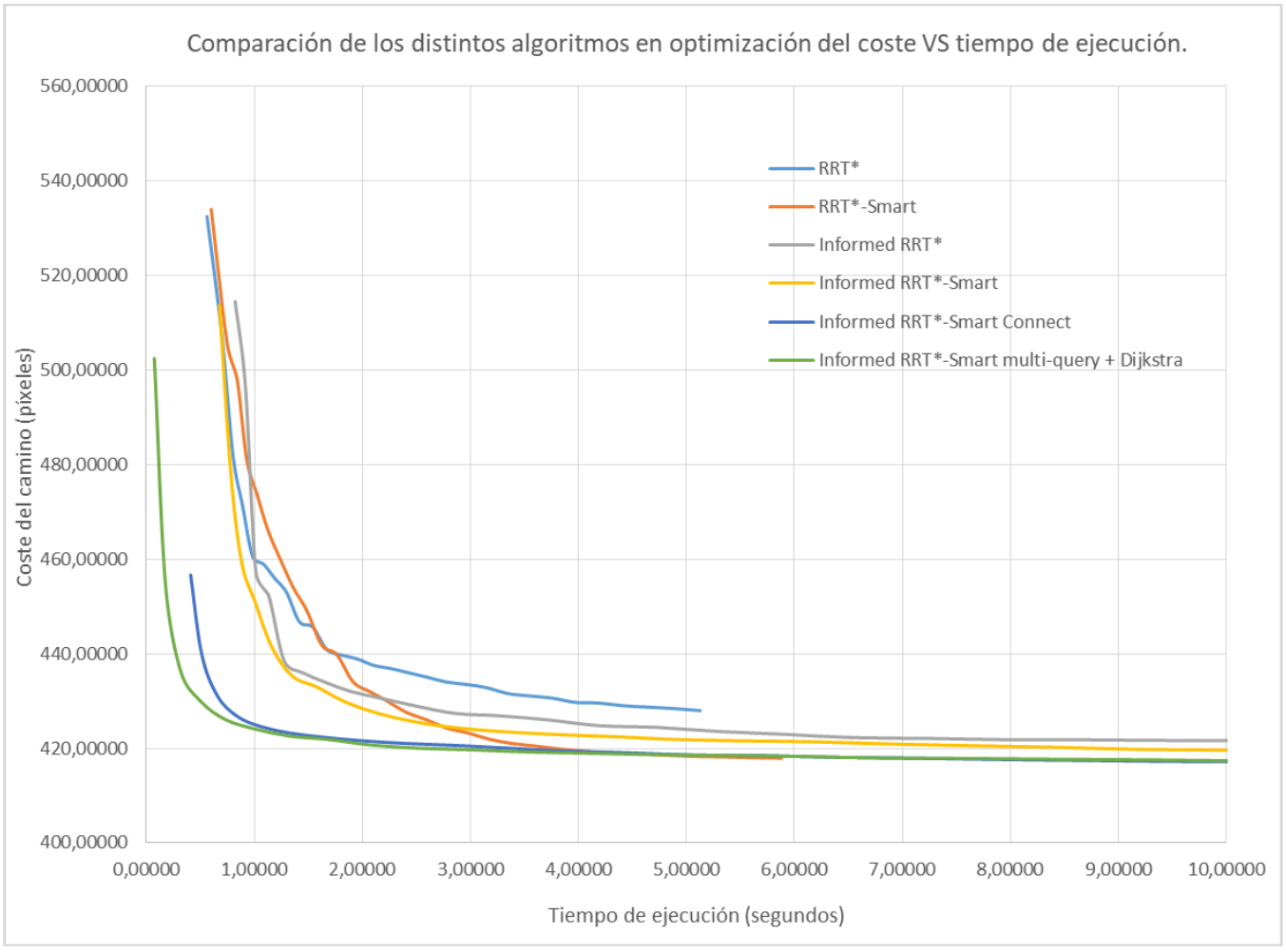

Figura 99. Comparación de algoritmos de optimización de caminos en estático (análisis de Montecarlo), problema sencillo.

Se puede apreciar en la gráfica resultado cómo se han cumplido las expectativas anteriores. Todos los algoritmos RRT* con heurística Informed y Smart a la vez han sido los que más rápidamente han convergido, de forma asintótica y a la misma velocidad. Eso sí, los que tenían fase de búsqueda de camino inicial especializada han llegado antes al óptimo.

Es importante añadir que los algoritmos que no implementan fase de búsqueda de camino inicial (los cuatro primeros en la leyenda) no presentan resultados consistentes en sus primeros instantes de aparición. Esto se debe a que no siempre se encontraban igual de rápido. A medida que el tiempo pasa ya siempre se encontraba solución.

A continuación se muestran algunos ejemplos de los algoritmos resolviendo el problema sencillo. Es curioso señalar cómo, a medida que el camino va convergiendo al óptimo, cada vez cuesta más tiempo de ejecución seguir iterando. Por eso no es fiable juzgar el algoritmo más rápido viendo las siguientes figuras, ya que están todas a igualdad de iteraciones, pudiendo haber llegado unos algoritmos al óptimo antes que otros.

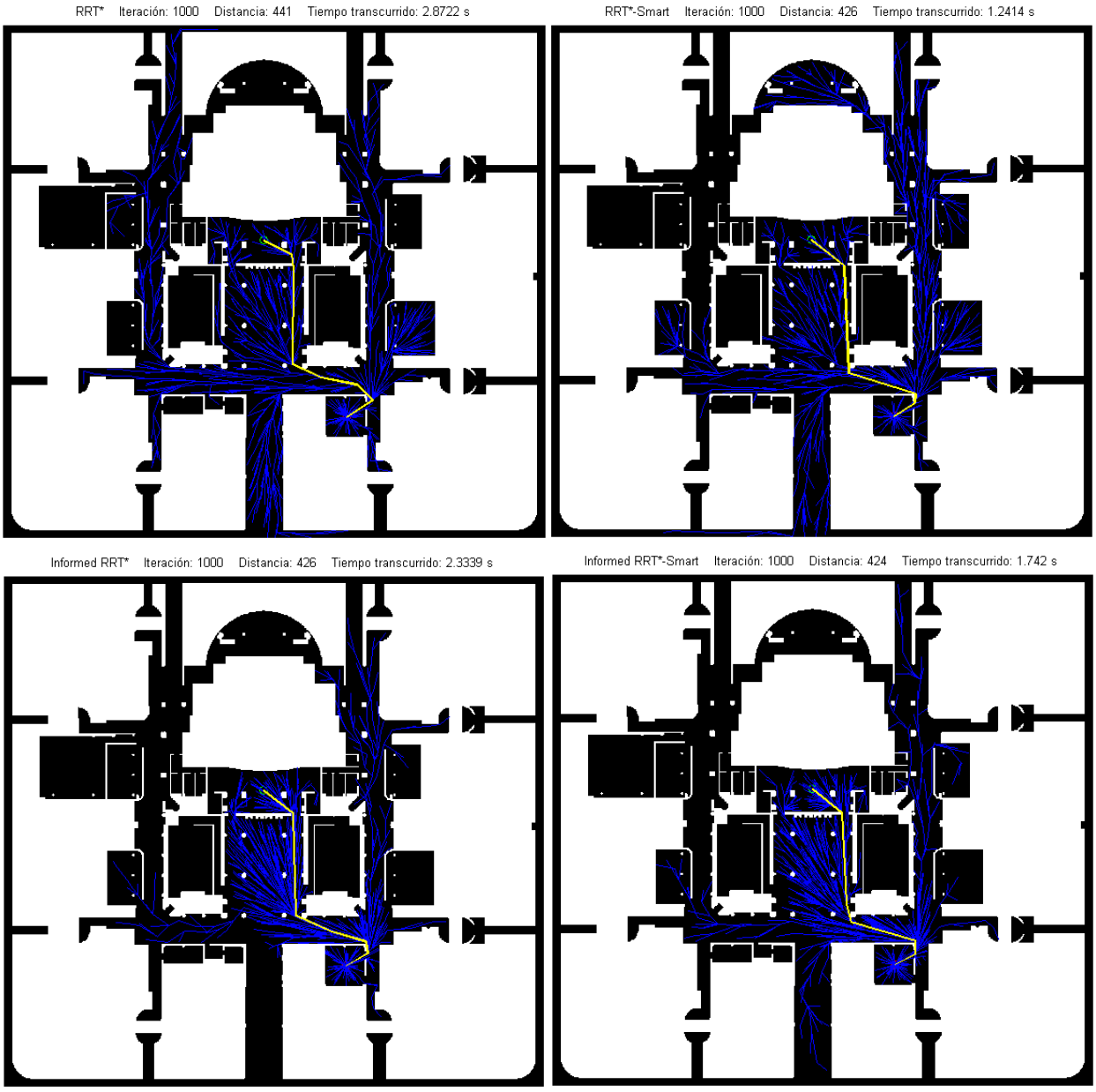

Figura 100. Problema sencillo siendo resuelto por los algoritmos sin fase de búsqueda de camino inicial.

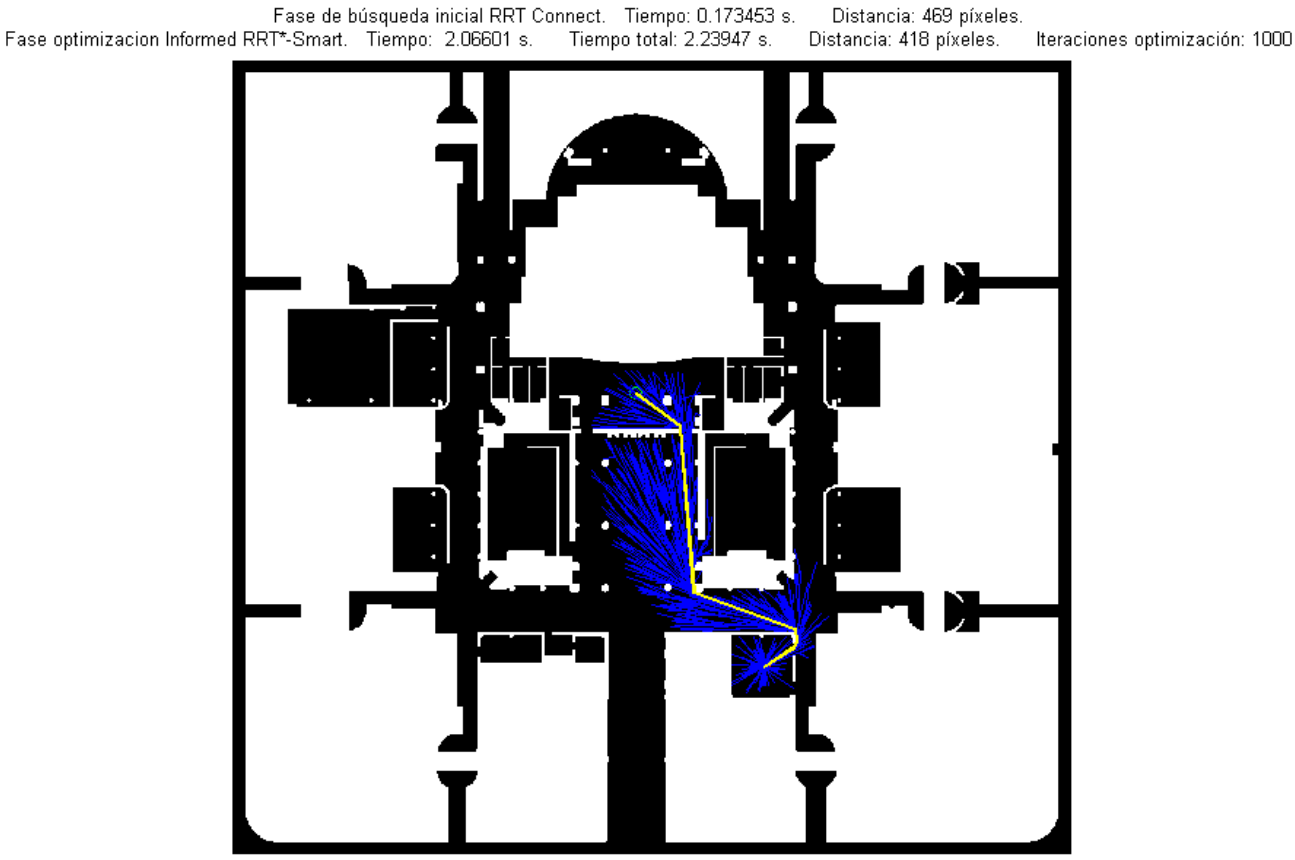

Figura 101. Problema sencillo siendo resuelto por Informed RRT*-Smart Connect.

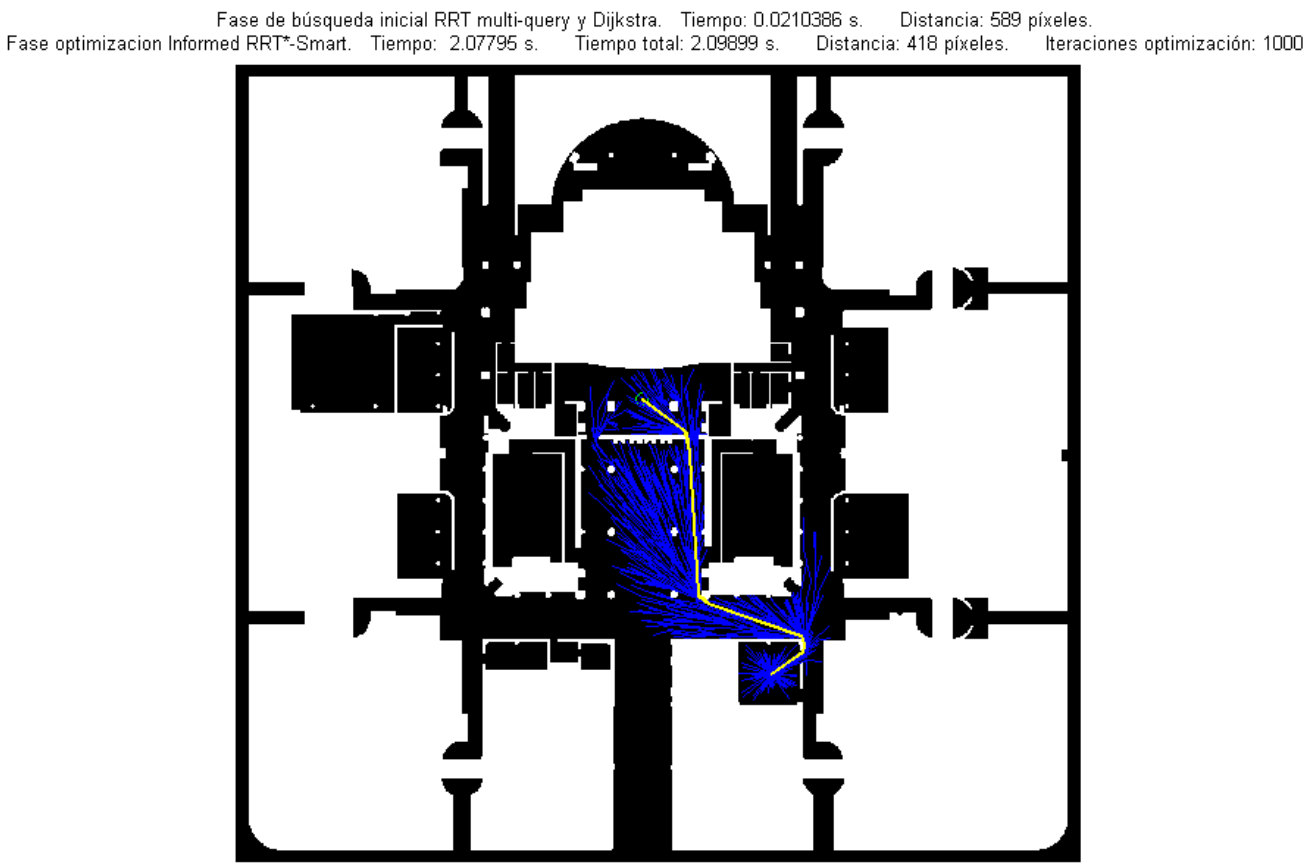

Figura 102. Problema sencillo siendo resuelto por Informed RRT*-Smart multi-query + Dijkstra.

Problema complicado: planificación de caminos óptima desde una punta de la Entreplanta 2 a la otra.

Este problema se concibió para demostrar la superioridad de los algoritmos con fase de búsqueda de camino inicial ante problemas complejos con abundantes mínimos locales, esperándose que los que no incluyen dicha fase fracasasen. Además, se esperaba que a igualdad de heurísticas la velocidad de convergencia fuera la misma, igual que antes. Los resultados lo confirmaron, indicando que multi-query + Dijkstra es más rápido y da mayor calidad de solución que Connect, lo cual era de esperar tras la comparación del apartado 5.3.4.

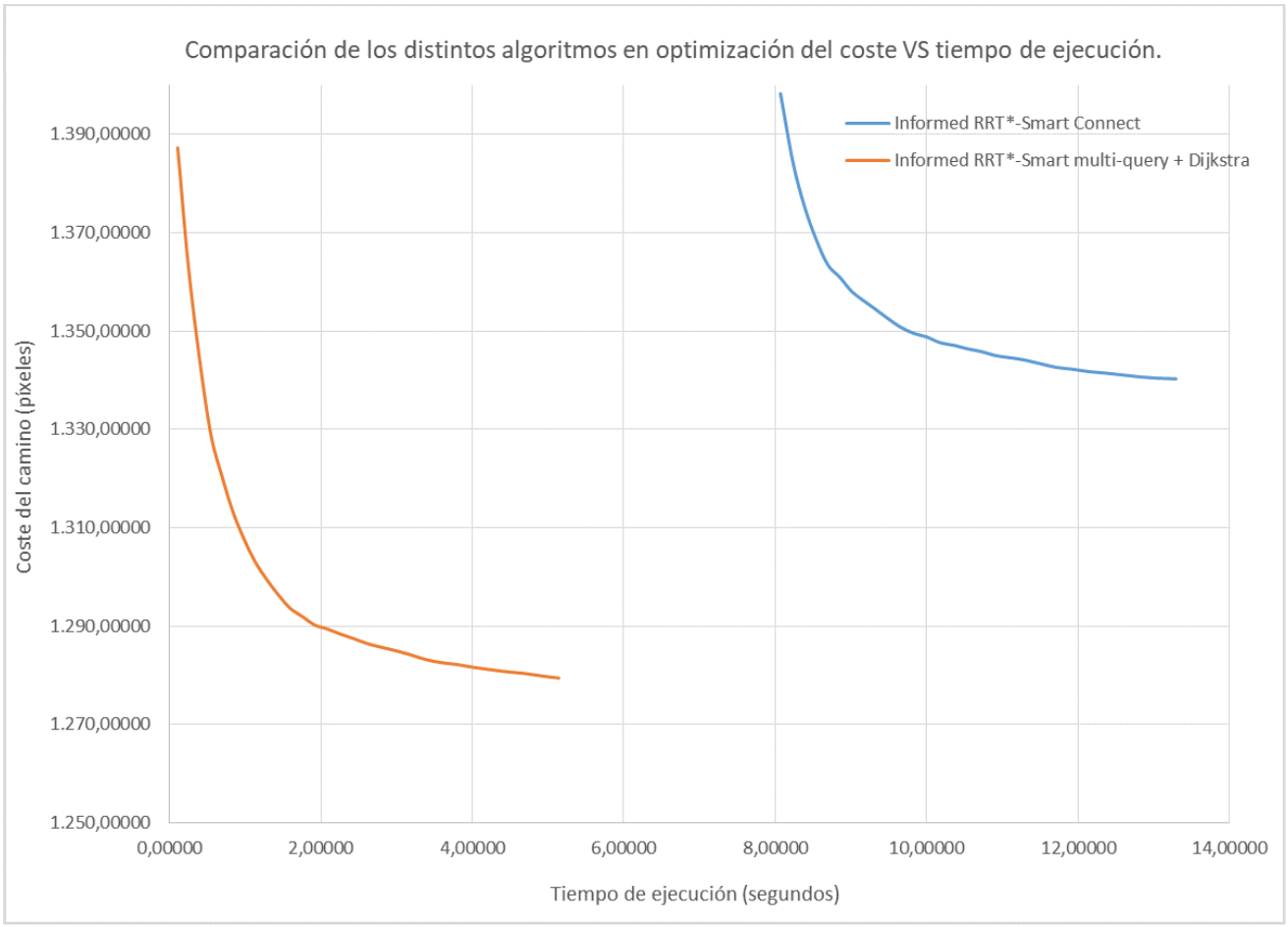

Figura 103. Comparación de algoritmos de optimización de caminos en estático (análisis de Montecarlo), problema complejo.

En la siguiente página se muestran algunos ejemplos de los algoritmos resolviendo el problema complicado.

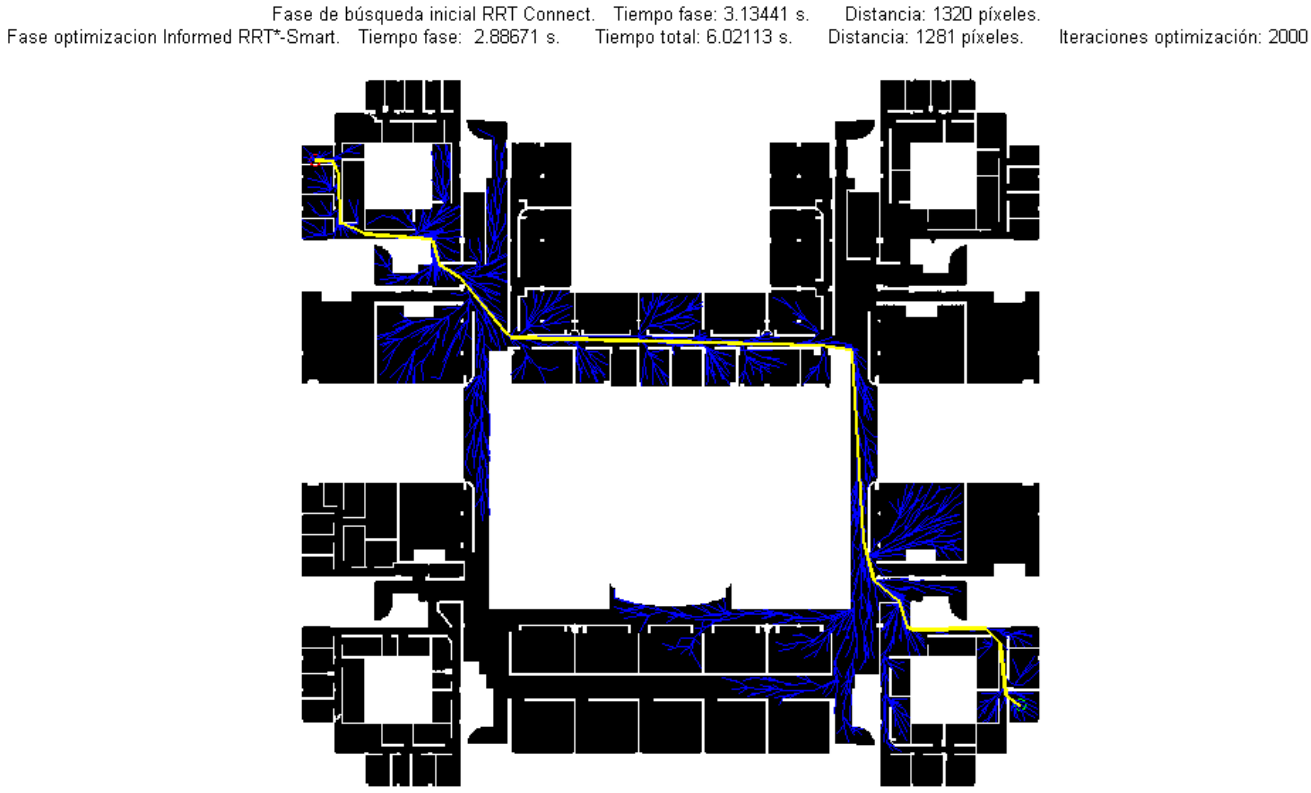

Figura 104. Problema complejo siendo resuelto por Informed RRT*-Smart Connect.

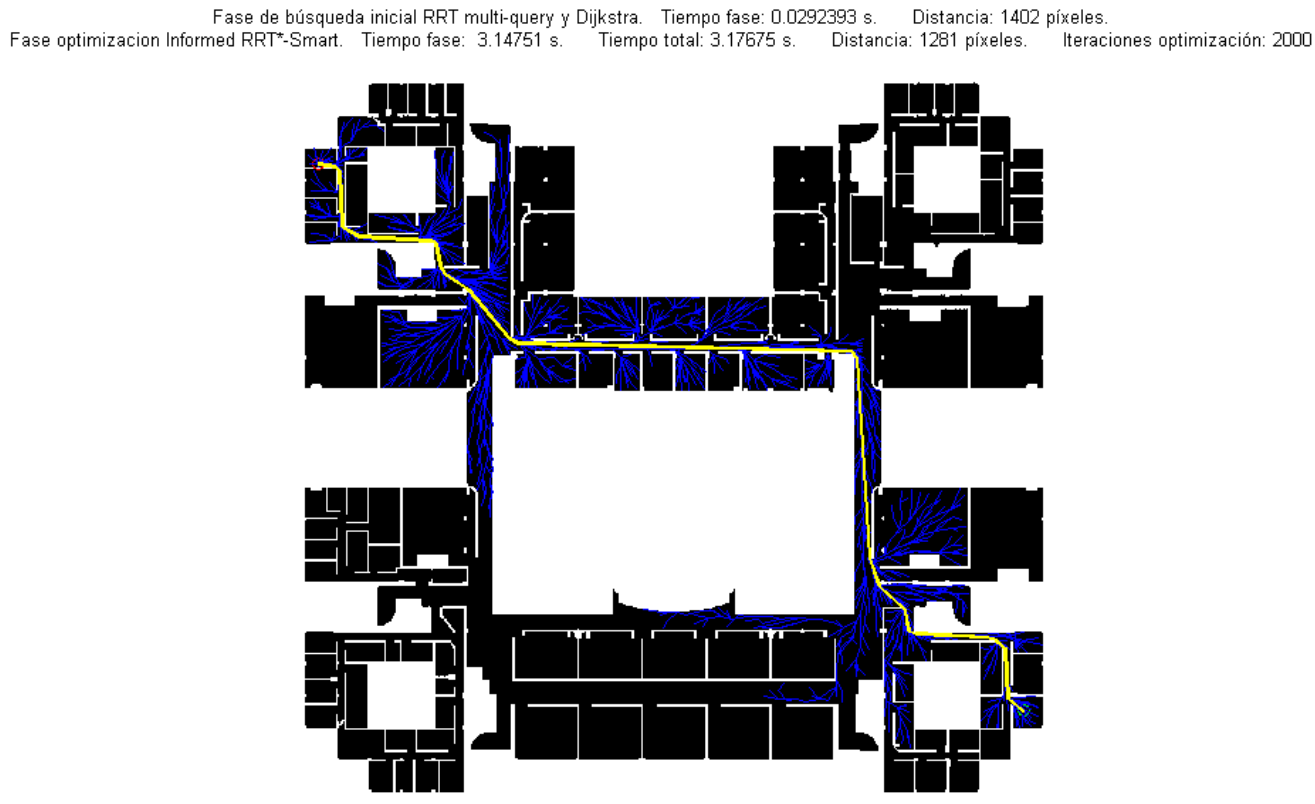

Figura 105. Problema complejo siendo resuelto por Informed RRT*-Smart multi-query + Dijkstra.

7 ALGORITMOS DE OPTIMIZACIÓN EN TIEMPO REAL

7.1 Anytime Path Planning

Los algoritmos "anytime" de tiempo real son muy interesantes en aplicaciones prácticas con robots. Estos algoritmos se caracterizan porque primero calculan rápidamente un camino inicial compatible y después usan el tiempo disponible durante la ejecución de la trayectoria para continuar optimizándola [47].

Hay distintas formas de implementar la filosofía anytime a los algoritmos. En este Proyecto se ha hecho tal que así:

1º) Cálculo rápido del primer camino no óptimo en estático mediante RRT multi-query + Dijkstra.

2º) Optimización durante un tiempo de la trayectoria en estático mediante RRT* con heurísticas Smart e Informed. Esta fase en la práctica representaría el tiempo que tarda en estar un robot listo para desplazarse, ya sea porque está preparando los motores, manipulando productos, esperando una señal, etc. La duración de esta fase de optimización está acotada por un temporizador (mínimo tiempo dedicado a optimizar), tamaño máximo del árbol o por una señal que indique que debe empezar el movimiento del robot.

3º) Optimización en tiempo real con el robot en movimiento mediante RT-RRT*, implementando también RRT*FN y las heurísticas Informed y Smart.

7.2 Algoritmo RRT*FN

Este algoritmo fue presentado en agosto de 2013 por Olzhas Adiyatov y Huseyin Atakan Varol en su paper "Rapidly-Exploring Random Tree Based Memory Efficient Motion Planning" [18], y consiste en una versión modificada del RRT* que limita la memoria requerida por el planificador para el almacenamiento del árbol.

Realmente este algoritmo no es de tiempo real, pero su uso será vital en este tipo de problemas ya que el planificador necesita ejecutarse de forma continua durante mucho tiempo y el tamaño del árbol podría hacerse totalmente intratable. Con RRT*FN (Fixed Node) eso no pasa.

La idea del RRT*FN es que cuando el número de nodos contenidos en el árbol lleguen a una determinada cantidad fijada como parámetro de entrada al planificador, el árbol deje de expandirse pero permitiendo seguir optimizándolo. Esto se consigue borrando un nodo "débil" antes de insertar otro nuevo.

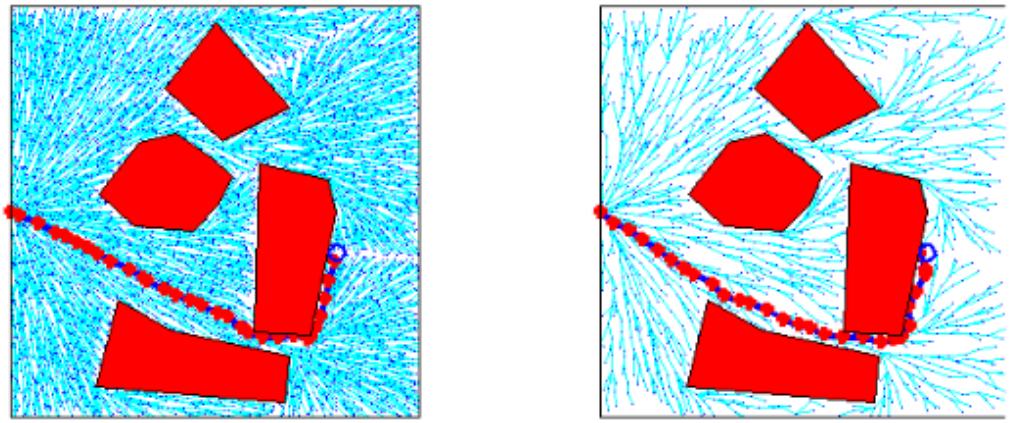

Figura 106. RRT* (izq.) vs RRT*FN (der.) [18]. Como se puede observar, con RRT*FN el resultado es bastante cercano al óptimo con un árbol mucho menos denso y pesado.

Se define como nodo *débil* aquel que no tiene ningún hijo. Estos nodos son abundantes en los bordes de los obstáculos, y la mayoría de ellos no tienen interés de cara a optimizar la trayectoria. Es por eso que se considera que el nuevo nodo a introducir en el árbol probabilísticamente vale más que un nodo sin hijos.

A continuación se presenta el pseudocódigo del RRT*FN, el cual no se ha implementado al pie de la letra:

Algorithm 5 $\tau = (V, E) \leftarrow$ RRT*FN(z_{init}, M)

```
1   τ ← InitializeTree();
2   τ ← InsertNode(Ø, z_init, τ);
3   for i = 1 to i = N do
4       if M < NodesAdded(τ) then
5         τ_old ← τ;
6       end if
7       z_rand ←Sample(i);
8       z_nearest ←Nearest(τ, z_rand);
9       (x_new, u_new, T_new) ← Steer(z_nearest, z_rand);
10      if ObstacleFree(x_new) then
11        Z_near ← Neighbors(τ, z_new, |V|);
12        z_min ← ChooseParent(Z_near, z_nearest, z_new, x_new);
13        τ ← InsertNode(z_min, z_new, τ );
14        τ ← ReWire(τ, Z_near, z_min, z_new);
15        τ ← ForcedRemoval(τ, z_goal);
16      end if
17      if NoRemovalPerformed() then
18            τ ← RestoreTree();
19      end if
20  end for
21  return τ
```

Algorithm 6 $z_{min} \leftarrow$ ChooseParent(Z_{near}, $z_{nearest}$, z_{new}, x_{new})

```
1   z_min ← z_nearest;
2   c_min ← Cost(z_nearest) + c(x_new);
3   for z_near ∈ Z_near do
4       (x', u', T') ← Steer(z_nearest, z_new);
5       if ObstacleFree(x') and x'(T') = z_new then
6         c' = Cost(z_nearest) + c(x_new);
7         if c' < Cost(z_new) and c' < c_min then
8               z_min ← z_near;
9               c_min ← c';
10        end if
11      end if
12  end for
13  return z_min
```

Algorithm 7 $\tau \leftarrow$ Rewire(τ , Z_{near}, z_{min}, z_{new})

```
1   for z_near ∈ Z_near \ z_min do
2       (x', u', T') ← Steer(z_nearest, z_new);
3       if ObstacleFree(x') and x'(T') = z_near and
          Cost(z_new) + c(x') < Cost(z_near) then
4         if onlyChild(Parent(z_near)) and
            M < NodesAdded(τ) then
5               RemoveNode(Parent(z_near));
6         end if
7         τ ← ReConnect(z_new, z_near, τ);
8       end if
9   end for
10  return τ
```

Figura 107. Pseudocódigo del algoritmo RRT*FN [18].

Como se puede apreciar, el pseudocódigo es exactamente igual que el del RRT* salvo por las dos funciones dedicadas a borrar nodos cuando el árbol está lleno:

1) "RemoveNode": es el primer intento de borrar nodos y se da durante el "rewire". Si al hacer "rewire" alrededor de un punto nuevo hay nodos que antes eran padres y dejan de serlo, dichos nodos se eliminarán. Esto no siempre se da, ya sea porque el recableado no es exitoso o porque el antiguo padre tenía varios hijos, lo cual nos lleva al siguiente tipo de borrado.

2) "ForcedRemoval": si durante el "rewire" no se ha conseguido borrar ninguno llega el momento de hacer una búsqueda global de nodos sin hijos, de los cuales se escogería uno al azar para su eliminación.

Por último, si ninguno de los 2 métodos anteriores tuviera éxito (realmente imposible, siempre hay nodos sin hijos potencialmente víctimas de "ForcedRemoval"), el pseudocódigo indica que se restaure el árbol anterior a la actual iteración con la función "RestoreTree". Básicamente lo que se pretende es dejar el árbol en su situación previa, sin el punto nuevo introducido en el árbol.

Como se dijo anteriormente, no todos los conceptos de RRT*FN se han aplicado al pie de la letra en el algoritmo propuesto al final de este capítulo. La principal diferencia radica en que no se ha implementado función "RemoveNode" en el recableado ni se ha contemplado la necesidad de un "RestoreTree". Esto quiere decir que todos los nodos borrados se harán con un "ForcedRemoval" sobre nodos sin hijos escogidos al azar de la totalidad del árbol.

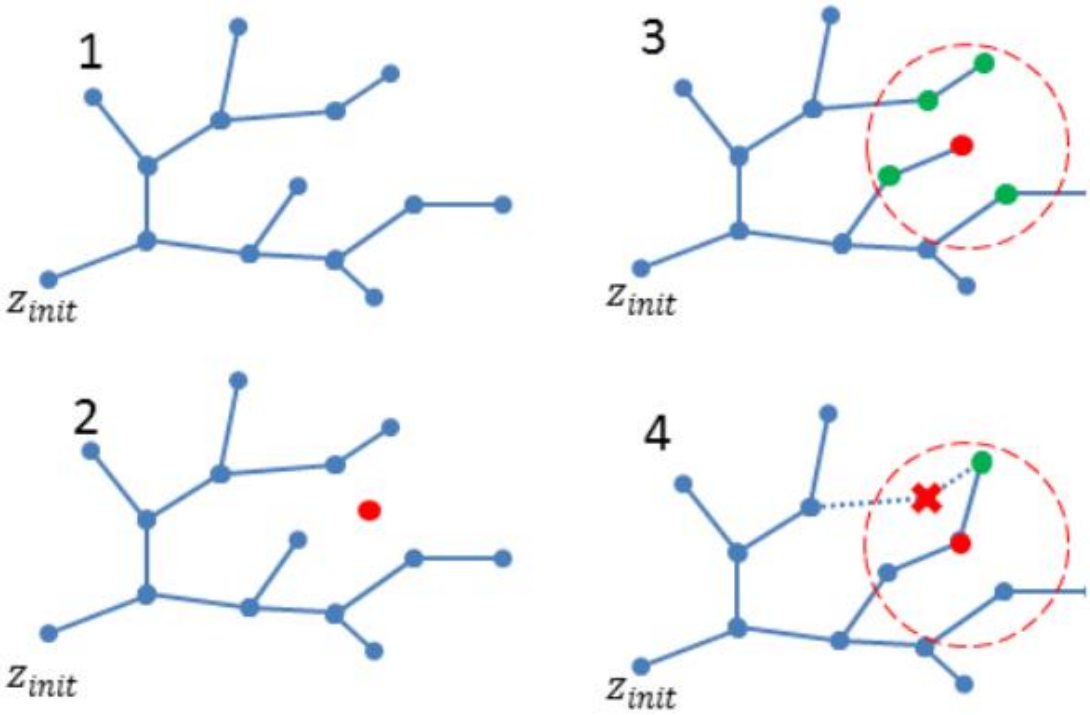

Figura 108. Ejemplo función "Rewire" borrando un nodo con "RemoveNode" [18].

7.3 Algoritmo RT-RRT*

Siendo presentado en noviembre de 2015 por Kourosh Naderi, Joose Rajamäkiy y Perttu Hämäläinenz en su paper "RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*" [12], se trata del algoritmo del estado del arte más novedoso de los aquí estudiados, a la par que el más ambicioso.

Su propósito es llevar al tiempo real el algoritmo RRT*, manteniendo el árbol y adaptándolo a los cambios que se produzcan en el problema. Recalcar el hecho de que no se reinicia el árbol para adaptarlo a los cambios como se hace en otros métodos. Dichos cambios serán:

- El robot o agente se irá moviendo para alcanzar el destino, de esta forma la raíz del árbol debe ir adaptándose para que las ramas salgan siempre de la posición actual del agente.
- El punto final o destino a alcanzar por el robot puede ser móvil. Si el destino se mueve y el árbol deja de contenerlo, RT-RRT* expandiría el árbol en modo persecución hacia el nuevo destino.
- A parte de los obstáculos fijos del mapa (como paredes, columnas, etc.), pueden existir obstáculos móviles, estos al moverse pueden llegar a bloquear ramas del árbol. Las ramas bloqueadas no se borran, ya que durante la fase de "Rewire" el árbol adaptará dichas ramas y las desbloqueará.

Como se puede apreciar, este algoritmo aborda el problema de planificación de caminos en tiempo real de forma completa. Es por eso que su implementación en este Proyecto resultó complicada técnicamente, aunque tras mucho trabajo y dedicación se llegó a conseguir.

Como el algoritmo RT-RRT* viene cargado de cosas nuevas respecto a los anteriores, se explicaran poco a poco por encima junto al pseudocódigo extraído del paper en el que se presenta, para más información leerse el paper directamente [12] o el código programado en MATLAB. Además, se irán detallando los puntos que se han programado en este Proyecto de forma distinta.

Algorithm 1 RT-RRT*: Our Real-Time Path Planning

```
1: Input: x_a, X_obs, x_goal
2: Initialize T with x_a, Q_r, Q_s
3: loop
4:     Update x_goal, x_a, X_free and X_obs
5:     while time is left for Expansion and Rewiring do
6:         Expand and Rewire T using Algorithm 2
7:     Plan (x_0, x_1, ..., x_k) to the goal using Algorithm 6
8:     if x_a is close to x_0 then
9:         x_0 ← x_1
10:    Move the agent toward x_0 for a limited time
11: end loop
```

Figura 109. Pseudocódigo de RT-RRT* (algoritmo 1) [12].

El algoritmo 1 representa el planificador en sí, es el algoritmo de más alto nivel de RT-RRT*. Este algoritmo irá llamando a otros que a su vez podrán llamar a otros más. El planificador lo primero que hace es llamar a la función Actualiza, la cual se dedica a leer los sensores de posición del robot, del destino, y comprobar si hay obstáculos móviles que se hayan movido. Mientras que en el pseudocódigo la raíz del árbol se actualiza lo penúltimo del planificador, en este Proyecto se implementó dentro de la función Actualiza. Esta función es vital en tiempo real ya que expresa de forma comprensible para el planificador los cambios del entorno.

Algorithm 6 Plan a Path for k Steps

1: **Input:** $\mathcal{T}$, x_{goal}
2: **if** Tree has reached x_{goal} **then**
3: | Update `path` from x_{goal} to x_0 if the `path` is rewired
4: | $(x_0, ..., x_k) \leftarrow (x_0, ..., x_{goal})$
5: **else**
6: | **for** $x_i \in (x_1, x_2, ..., x_k)$ **do**
7: | | x_i=child of x_{i-1} with minimum f_c=`cost`(x_c)+`H`(x_c)
8: | | **if** x_i is leaf or its children are blocked **then**
9: | | | $(x_0', ..., x_k') \leftarrow (x_0, ..., x_i)$
10: | | | Block x_i and Break;
11: | Update best `path` with $(x_0', ..., x_k')$ if necessary
12: | $(x_0, ..., x_k) \leftarrow$ choose to stay in x_0 or follow best `path`
13: **return** $(x_0, ..., x_k)$

Figura 110. Pseudocódigo de RT-RRT* (algoritmo 6) [12].

Lo siguiente que hace el algoritmo 1 planificador es llamar al algoritmo 2 de Expansión-y-Recableado, el cual se explicará en el siguiente párrafo. Después pasa a llamar a la función 6, cuyo pseudocódigo se puede encontrar sobre estas líneas. Este algoritmo 6 es el encargado de trazar la trayectoria a partir del árbol, lo cual es trivial si el punto final de destino está contenido en el árbol, aunque si no lo está (por ejemplo, porque se ha movido) es un problema. En el paper proponen solucionar esto aplicando un algoritmo A* que encuentre el camino hacia el nodo más cercano del punto final. En este Proyecto se ha hecho algo más simple, se ha supuesto que si el destino se ha movido no se puede haber ido muy lejos (tiempo de muestreo corto), así que la última trayectoria debería seguir siendo válida, simplemente hay que darle más tiempo al algoritmo para que introduzca el punto destino como nodo en el árbol. Así, el algoritmo 6 propuesto en estos casos devolvería la última trayectoria pasada con P_{goal} dentro del árbol, avisando de que no está contenido. Por último, el algoritmo 1 planificador exporta la trayectoria hacia el controlador del robot, el cual se encargará de su seguimiento.

El algoritmo 2 de Expansión-y-Recableado se ejecuta desde el planificador una cantidad de tiempo limitada por la constante de tiempo de muestreo, parámetro configurable del RT-RRT*. La correcta configuración del tiempo de muestreo es vital, ya que si se configura demasiado corto podría responder rápidamente ante cambios del entorno pero no le daría tiempo a que se optimizara lo suficiente, y de ser demasiado largo justo lo contrario. Dentro del algoritmo 1 del planificador, el algoritmo 2 de Expansión-y-Recableado debería ser el que dominara la duración de cada iteración, tardando más que la función Actualiza y que el algoritmo 6.

Algorithm 2 Tree Expansion-and-Rewiring

1: **Input:** $\mathcal{T}$, $\mathcal{Q}_r$, $\mathcal{Q}_s$, k_{max}, r_s
2: `Sample` x_{rand} using (1)
3: $x_{closest} = \arg\min_{x \in \mathcal{X}_{SI}}$ `dist`(x, x_{rand})
4: **if** `line`$(x_{closest}, x_{rand}) \subset \mathcal{X}_{free}$ **then**
5: | $\mathcal{X}_{near}$ = `FindNodesNear`$(x_{rand}, \mathcal{X}_{SI})$
6: | **if** $|\mathcal{X}_{near}| < k_{max}$ or $|x_{closest} - x_{rand}| > r_s$ **then**
7: | | `AddNodeToTree`$(\mathcal{T}, x_{rand}, x_{closest}, \mathcal{X}_{near})$
8: | | `Push` x_{rand} to the first of $\mathcal{Q}_r$
9: | **else**
10: | | `Push` $x_{closest}$ to the first of $\mathcal{Q}_r$
11: | `RewireRandomNode`$(\mathcal{Q}_r, \mathcal{T})$
12: `RewireFromRoot`$(\mathcal{Q}_s, \mathcal{T})$

Figura 111. Pseudocódigo de RT-RRT* (algoritmo 2) [12].

Lo primero que hace el algoritmo 2 es el sampleado de un punto al azar. RT-RRT* muestrea puntos de tres maneras:

- *Uniforme:* escoge un punto al azar entre todo el espacio libre de colisiones, garantizando que todo el espacio de trabajo se explora.

- *Informed:* RT-RRT* incluye de por sí muestreo con heurística Informed, escogiendo un punto al azar dentro de la elipse equivalente de la trayectoria. Hace énfasis en la optimización de la trayectoria si el punto destino está contenido en el árbol.

- *Alineado a fin:* escoge un punto al azar contenido en el segmento que une el punto final con el nodo del árbol más cercano al destino. Sirve para perseguir el punto final cuando se ha movido fuera del árbol.

Además, en el algoritmo propuesto en este Proyecto, se incluyó una 4º manera de muestreo con heurística *Smart*.

Se escoge el tipo de muestreo (Uniforme, Informed, Alineado a fin o Smart) al azar, de forma que cada uno tiene una probabilidad configurable de ser escogido, siendo bastante más probables los muestreos Uniforme e Informed que los Alineado a fin y Smart.

Con el punto ya muestreado se comprueba si entre él y su nodo del árbol más cercano no hay colisión, si la hay se pasa a recablear la cola de recableado desde la raíz del árbol (se explicará más adelante el qué significa eso) y la iteración actual termina. Si no hay colisión, se calculan sus puntos más cercanos con un "Near" (es vital el mallado de entorno en este punto para que este cálculo no sea excesivo en árboles grandes), si esa zona está poco ramificada se introduce dicho punto en el árbol y se encola en la cola de recableado aleatoria (se explicará también más adelante), si está demasiado ramificada no se añade y se encola en su lugar el nodo más cercano. Por último, se recablean las colas de recableado aleatoria y desde raíz del árbol.

Decir que el RT-RRT* en el paper propone un radio variable en el cálculo del "Near". Esto no se ha tenido en cuenta en este Proyecto por simplificar y para tener un árbol con longitud de ramas más o menos constantes.

El recableado en RT-RRT* se hace en dos partes y por medio de 2 colas de recableado, dichas colas contendrán la lista de nodos a los que hay que hacer "rewire". Estas colas no se borran entre distintas llamadas del algoritmo 2 Expande-y-Recablea, siendo las siguientes:

- *Cola de recableado aleatoria:* los nodos recién introducidos al árbol o aquellos escogidos al azar se introducen aquí para hacer "rewire" a su alrededor. Si los nodos de alrededor se optimizan con el "rewire", dichos nodos de alrededor se encolarán también ya que son propensos a mejorar otros nodos de su alrededor. Esta cola será la encargada de optimizar y adaptar el árbol a los obstáculos fijos y móviles del problema.

- *Cola de recableado desde raíz del árbol:* esta cola se inicializará conteniendo el nodo raíz del árbol cada vez que se mueva dicha raíz. Al hacer rewire se irán encolando los nodos optimizados como con la cola anterior. Esta cola es la encargada de adaptar el árbol desde su origen a los movimientos del agente.

Por último, comentar que las ramas del árbol bloqueadas por obstáculos móviles no se eliminan, sino que se ponen sus costes totales a infinito, de forma que si se hace "rewire" a su alrededor el árbol se combará para que dicha rama deje de tener coste infinito, bordeando el obstáculo móvil sin tocarlo.

En la siguiente página se muestran los pseudocódigos de las funciones dedicadas al recableado (en este Proyecto ambas son la misma función que hace rewire a cualquier cola de entrada).

Algorithm 3 Add Node To Tree

Input: $\mathcal{T}$, $\mathbf{x}_{new}$, $\mathbf{x}_{closest}$, $\mathcal{X}_{near}$
$\mathbf{x}_{min} = \mathbf{x}_{closest}$, $c_{min} = \texttt{cost}(\mathbf{x}_{closest}) + \texttt{dist}(\mathbf{x}_{closest}, \mathbf{x}_{new})$
for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
  $c_{new} = \texttt{cost}(\mathbf{x}_{near}) + \texttt{dist}(\mathbf{x}_{near}, \mathbf{x}_{new})$
  if $c_{new} < c_{min}$ and $\texttt{line}(\mathbf{x}_{near}, \mathbf{x}_{new}) \in \mathcal{X}_{free}$ **then**
    $c_{min} = c_{new}$, $\mathbf{x}_{min} = \mathbf{x}_{near}$
$V_{\mathcal{T}} \leftarrow V_{\mathcal{T}} \cup \{\mathbf{x}_{new}\}$, $E_{\mathcal{T}} \leftarrow E_{\mathcal{T}} \cup \{\mathbf{x}_{min}, \mathbf{x}_{new}\}$

Algorithm 4 Rewire Random Nodes

Input: $\mathcal{Q}_r$, $\mathcal{T}$
repeat
  $\mathbf{x}_r$=$\texttt{PopFirst}(\mathcal{Q}_r)$, $\mathcal{X}_{near}$=$\texttt{FindNodesNear}(\mathbf{x}_r, \mathcal{X}_{SI})$
  for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
    c_{old}=$\texttt{cost}(\mathbf{x}_{near})$, c_{new}=$\texttt{cost}(\mathbf{x}_r)$+$\texttt{dist}(\mathbf{x}_r, \mathbf{x}_{near})$
    if $c_{new} < c_{old}$ and $\texttt{line}(\mathbf{x}_r, \mathbf{x}_{near}) \in \mathcal{X}_{free}$ **then**
      $E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \setminus \{\texttt{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near}\}) \cup \{\mathbf{x}_r, \mathbf{x}_{near}\}$
      Push $\mathbf{x}_{near}$ to the end of $\mathcal{Q}_r$
until Time is up or $\mathcal{Q}_r$ is empty.

Algorithm 5 Rewire From the Tree Root

Input: $\mathcal{Q}_s$, $\mathcal{T}$
if $\mathcal{Q}_s$ is empty **then**
  Push $\mathbf{x}_0$ to $\mathcal{Q}_s$
repeat
  $\mathbf{x}_s$=$\texttt{PopFirst}(\mathcal{Q}_s)$, $\mathcal{X}_{near}$=$\texttt{FindNodesNear}(\mathbf{x}_s, \mathcal{X}_{SI})$
  for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
    c_{old}=$\texttt{cost}(\mathbf{x}_{near})$, c_{new}=$\texttt{cost}(\mathbf{x}_s)$+$\texttt{dist}(\mathbf{x}_s, \mathbf{x}_{near})$
    if $c_{new} < c_{old}$ and $\texttt{line}(\mathbf{x}_s, \mathbf{x}_{near}) \in \mathcal{X}_{free}$ **then**
      $E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \setminus \{\texttt{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near}\}) \cup \{\mathbf{x}_s, \mathbf{x}_{near}\}$
    if $\mathbf{x}_{near}$ is not pushed to $\mathcal{Q}_s$ after restarting $\mathcal{Q}_s$ **then**
      Push $\mathbf{x}_{near}$ to the end of $\mathcal{Q}_s$
until Time is up or $\mathcal{Q}_s$ is empty.

Figura 112. Pseudocódigo de RT-RRT* (algoritmos 3, 4 y 5) [12].

7.4 Algoritmo RT-Informed RRT*-Smart FN multi-query + Dijkstra (2D)

7.4.1 Introducción.

El algoritmo propuesto busca la **planificación en tiempo real**, conseguida mediante planteamiento **Anytime Motion Planning**, el cual consiste en dos fases:

1) **Planificación óptima en estático:** Esto se consigue corriendo el algoritmo **Informed RRT*-Smart multi-query + Dijkstra adaptado a Anytime Motion Planning**. Dicha adaptación consiste en que ya no solo se muestrean puntos Informed y Smart exclusivamente, sino también puntos uniformemente distribuidos. Además, se incluyen nuevas condiciones de finalización de la optimización: parada por entrada de señal externa, tiempo límite de optimización, y tamaño límite del árbol de optimización.
2) **Optimización en tiempo real:** Cuando llega la hora de mover el agente, se continúa optimizando en tiempo real la trayectoria teniendo en cuenta los obstáculos móviles. Para ello aplica **RT-RRT*** de forma continua hasta que el agente haya cumplido su objetivo. Este algoritmo RT-RRT* permite mantener y adaptar el árbol a cambios de posición del agente, movimiento del objetivo (modo persecución), y al movimiento de obstáculos móviles. RT-RRT* incluye ya Informed RRT*, aunque se ha modificado para que también incluya RRT*-Smart y **RRT*FN**. Este último algoritmo lo que consigue es permitir seguir iterando y optimizando el árbol incluso tras su llenado, limitando su tamaño y ahorrando memoria.

7.4.2 Función implementada en 2D.

La función MATLAB implementada para el caso plano (2D) del algoritmo RT-Informed RRT*-Smart FN multi-query + Dijkstra es la siguiente:

```
Planificador_RT_Informed_RRTstar_Smart_FN_Dijkstra(P_ini, P_fin, mapa_fijo,
radio_obstaculo, t_max_Anytime, tiempo_limite, muestra_animaciones, k_max,
Nmax_RRT_multi_query, Nmax_Anytime, Nmax_optimizacion, salto, radio,
subdivisiones_M, subdivisiones_N, grafo, dist_grafo);
```

Recibe como entrada los puntos inicial y final de la trayectoria, el mapa de obstáculos fijos (tales como paredes, columnas, etc.), el radio de modelado de los obstáculos leídos por los sensores, el tiempo máximo Anytime de optimización antes de empezar a moverse, el tiempo límite disponible para la Expansión-y-Recableado, el vector de variables muestra_animaciones (para configurar que se visualicen resultados o no), k_max (número máximo de nodos que pueden haber dentro del radio de optimización), el número máximo de iteraciones del RRT multi-query (a partir del cual se considerará que no existe camino posible), el número máximo de iteraciones de optimización Anytime antes de empezar a moverse, el tamaño máximo del árbol a partir del cual FN (Fixed Node) tiene que empezar a actuar, el salto o longitud de rama del árbol, el radio de optimización del "rewire" del RRT*, las subdivisiones del mallado de entorno, y por último el grafo para el Dijkstra (dist_grafo contiene los costes de los arcos).

Aparte, tiene entradas de sensores en tiempo real, llamadas desde dentro del planificador. Sus salidas también son desde dentro del planificador, exportándolas por el medio de comunicación preciso. Como salida proporciona los waypoints a seguir en una matriz llamada "trayectoria" y una variable booleana que vale 1 si realmente dicha trayectoria es capaz de llegar o no al punto final, y 0 si no. El sentido de esto es que hay veces que el destino se mueve y se pierde, pero realmente no puede haberse ido muy lejos, por lo que lo más sensato es decirle al agente que se siga moviendo en la misma dirección pero avisándole de que tenga cuidado. Lo mismo pasa cuando un obstáculo móvil lejano tapa la trayectoria, lo mejor es mantener la trayectoria ya que es posible que el obstáculo móvil se quite, pero indicando que tenga cuidado el controlador del robot ya que es posible que haya obstáculos.

7.4.3 Resultados.

Para mostrar la capacidad del algoritmo propuesto y sacar resultados, este se aplicará a dos problemas, ambos de planificación reactiva óptima en tiempo real. El primero de ellos será en un mapa complicado, partiendo desde una punta de la Entreplanta 2 (Departamento de Ingeniería de Sistemas y Automática) hasta la otra (Departamento de Telemática). El otro problema será en un mapa más sencillo, consistiendo en unos pocos obstáculos fijos en forma de paredes rectangulares.

Como en este capítulo de planificación óptima en tiempo real no se han programado más algoritmos, no se podrá comparar los resultados del algoritmo propuesto con otros distintos.

Para poder sacar conclusiones de cuán reactivo es el algoritmo RT-Informed RRT*-Smart FN multi-query + Dijkstra, en el entorno sencillo se ha hecho un análisis Montecarlo del tiempo de replanificación ante obstáculos móviles que se ponen en mitad del trayecto. El cálculo del tiempo de ejecución absoluto carece de mucho sentido en este caso, ya que el planificador está continuamente ejecutándose.

Respecto a la visualización gráfica, se tiene lo siguiente:

- La raíz del árbol se representa como una circunferencia roja.
- El destino se representa como una circunferencia verde.
- Los obstáculos móviles se representan con circunferencias rojas.
- Todos los obstáculos, tanto fijos como móviles, se representan como píxeles en blanco, mientras que la zona libre de obstáculos en negro. Los obstáculos móviles serán un círculo blanco expansionado un determinado radio alrededor ellos.
- Las ramas del árbol que representan trayectorias viables están en color azul, mientras que las ramas bloqueadas por algún obstáculo móvil están en rojo.
- La trayectoria que une la raíz del árbol con el destino, de existir y de estar encontrada, se representa como una línea gruesa amarilla.

Problema complicado de planificación óptima en tiempo real en la Entreplanta 2 de la ETSI.

Como este algoritmo es reactivo, para su simulación en MATLAB es necesario programar los estímulos del entorno. El planificador recibe estos estímulos llamando a la función "Input_sensores", el nombre se debe a que de usarse en un robot real, la función devolvería la información de los sensores. Pero como en este Proyecto se simulará todo por ordenador, la función "Input_sensores" devuelve información ficticia, más o menos ajustada a la realidad.

Para este problema, la función "Input_sensores" devuelve lo siguiente:

- *Posición del agente:* este estímulo se ha programado de forma que cada cierto número de iteraciones el robot se posicione en el siguiente waypoint de la trayectoria. En un robot real el desplazamiento sería gradual y continuo, no escalonado como aquí se simula.
- *Posición del destino:* el destino se puede mover y hay que perseguirlo, para ello es necesario saber dónde está. Aquí se ha programado para que se desplace dando saltos aleatorios, pudiendo incluso atravesar paredes. Este tipo de movimiento pondrá las cosas más difíciles al planificador de lo que son en la realidad.
- *Obstáculos móviles:* por simplificar, estos estímulos se han programado como puntos que pueden aparecer en cualquier punto del mapa al azar cada cierto número de iteraciones, expansionados un determinado radio. Esto realmente no tiene mucho sentido, ya que normalmente los obstáculos móviles reconocidos serían aquellos más cercanos al robot, y por supuesto no se teletransportarían. Pero al ser el escenario más desfavorable, si el algoritmo se comporta bien con estos estímulos, se comportará bien en la realidad.
- *Continúa:* variable booleana que vale 1 si se desea continuar el planificador de tiempo real, y 0 si se desea concluir, considerando el objetivo cumplido. En las simulaciones aquí presentadas se ha dejado siempre con valor de 1.

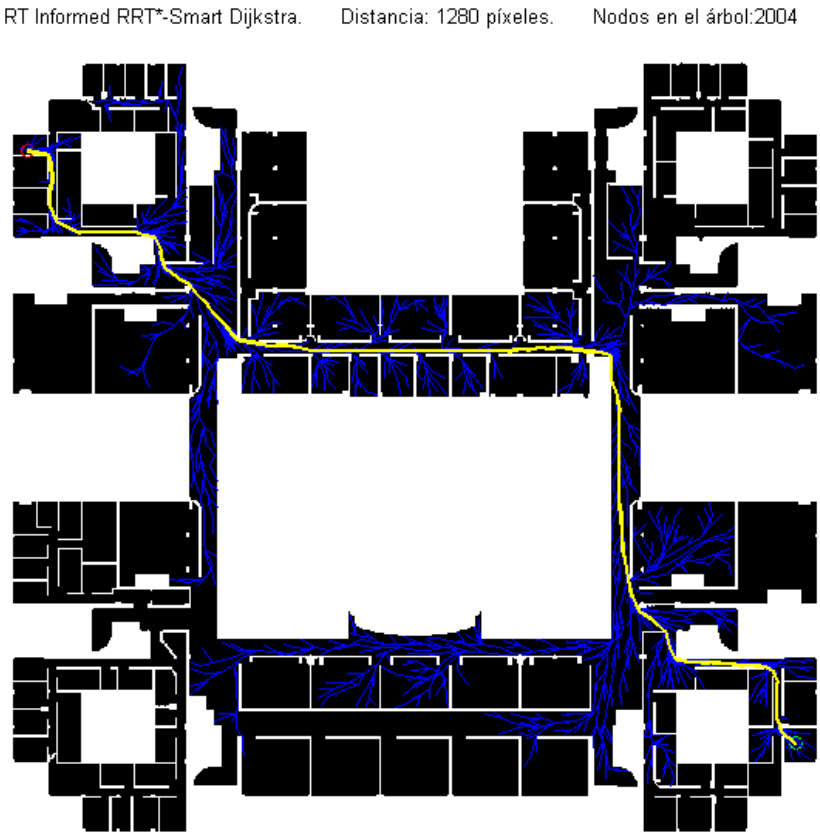

Figura 113. Situación inicial del problema, justo tras acabar la fase de optimización en estático del algoritmo propuesto. Todavía no se han movido los puntos inicial y final, ni han aparecido obstáculos móviles.

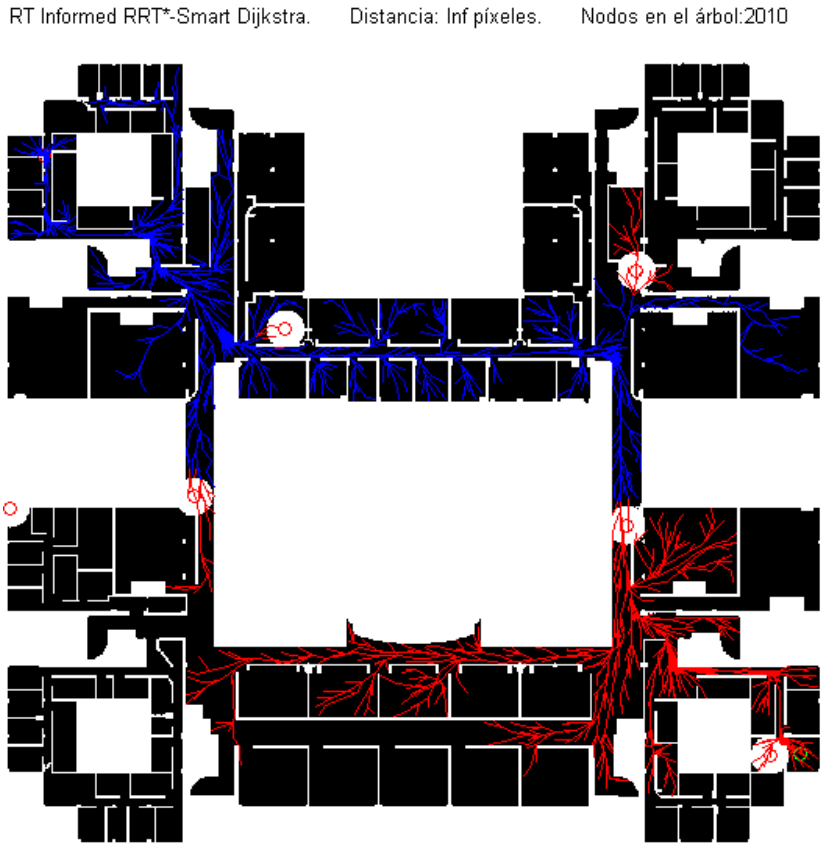

Figura 114. Han aparecido obstáculos móviles bloqueando los pasillos a cada lado de la biblioteca, haciendo imposible actualmente la existencia de solución.

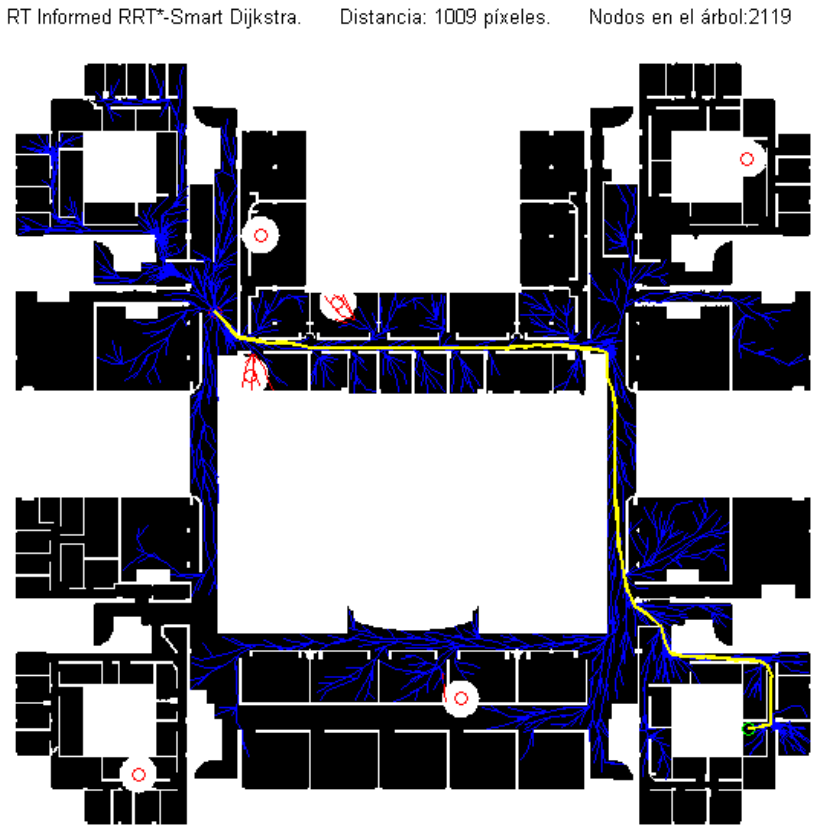

Figura 115. El tiempo ha pasado y los obstáculos móviles se han apartado, dejando paso libre. Nótese: (1) Pese a que el destino se ha movido al despacho de enfrente, se ha encontrado. (2) El robot en su avance está a punto de entrar en el Centro de Cálculo, expandiéndose el árbol de forma radial alrededor suya.

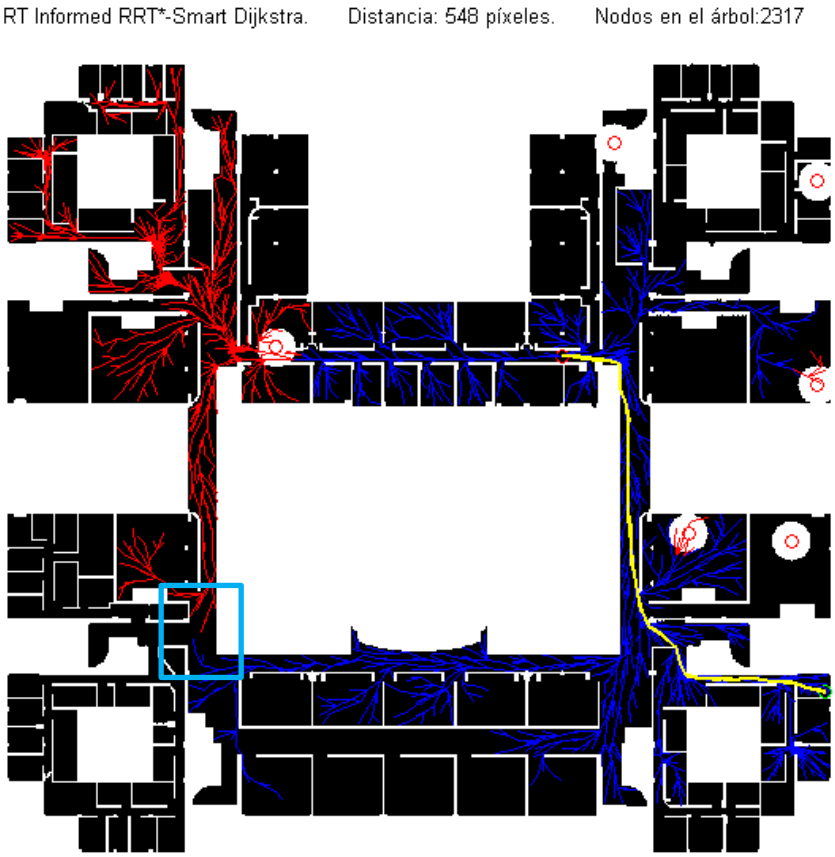

Figura 116. El agente está a punto de salir del Centro de Cálculo, encontrándose el destino en otro despacho distinto esta vez. Ver párrafo siguiente explicando el rectángulo celeste.

Es de especial interés comentar en esta última figura que, si el obstáculo del Centro de Cálculo permaneciera un tiempo ahí, sería muy probable que en la zona del rectángulo celeste las ramas azules del árbol entraran en contacto con las ramas rojas bloqueadas. De ser así, automáticamente dichas ramas rojas pasarían a proceder de la rama azul ("rewire"), rompiéndose a la altura del obstáculo móvil. Este fenómeno fruto de la optimización se verá más claramente en el siguiente problema.

Como se ha podido comprobar, este entorno resulta demasiado estrecho como para que el robot esquive los obstáculos móviles una vez caen en la trayectoria, por lo que aunque resulte impactante visualmente, no ofrece información sobre la capacidad reactiva del algoritmo propuesto. Para este fin, a continuación se tratará un entorno más amplio con un problema más simple.

Problema sencillo de planificación óptima en tiempo real.

Como ya se comentó anteriormente, el problema sencillo consiste en un mapa con pocos obstáculos fijos en forma de paredes rectangulares, en el que se estimará el tiempo de reacción ante obstáculos móviles que se ponen en mitad del trayecto.

Para ello, los estímulos de "Input_sensores" se han modificado respecto al problema anterior: tanto la posición del agente como la del destino se han dejado fijas, y los obstáculos móviles una vez aparecen (salidos de la nada) se quedan ahí permanentemente.

Para la estimación del tiempo de reacción se empieza a cronometrar cuando los obstáculos móviles se ponen en medio de la trayectoria, parando el cronómetro cuando se encuentra otra trayectoria nueva que bordee los obstáculos móviles. Este proceso se repite muchas veces (análisis Montecarlo), introduciéndose los resultados en Oracle Crystal Ball [45] para que calcule la distribución probabilística que mejor se ajusta a la distribución de tiempos de reacción ante obstáculo. Este análisis Montecarlo se hará en dos ejemplos distintos.

También se consideró estudiar el tiempo que se tarda en encontrar el destino cuando este se mueve de sitio, pero realmente en la práctica no es necesario, ya que el desplazamiento de un robot real es gradual y continuo en pequeños incrementos (respecto a la alta tasa de muestreo de la posición), y para este tipo de situaciones el código programado encuentra el destino automáticamente por visibilidad desde el último nodo fin. Esto solo no ocurriría en el preciso momento que el destino gira en una esquina, en cuyo caso se intentaría perseguir de forma normal con el "Alineado a fin".

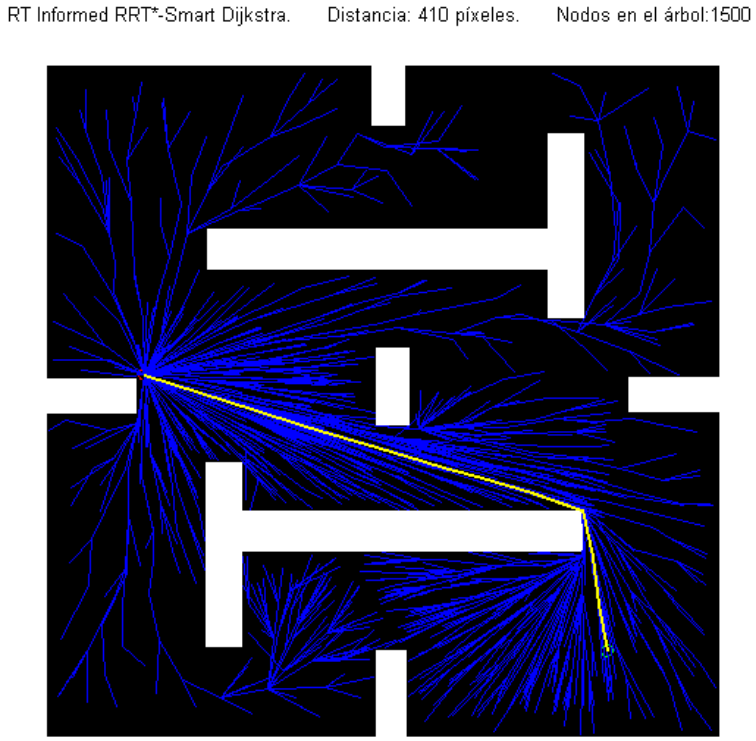

Figura 117. Ejemplo 1. Situación inicial óptima antes de que aparezcan los obstáculos móviles.

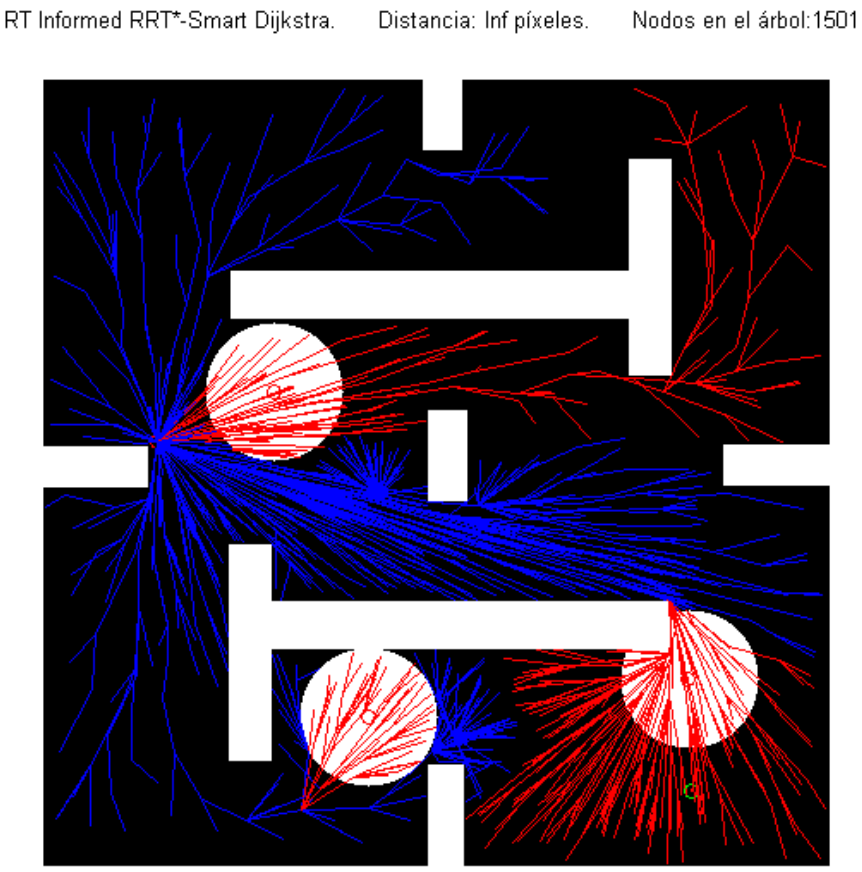

Figura 118. Ejemplo 1. Situación justo tras aparecer los obstáculos móviles. La trayectoria ha desaparecido.

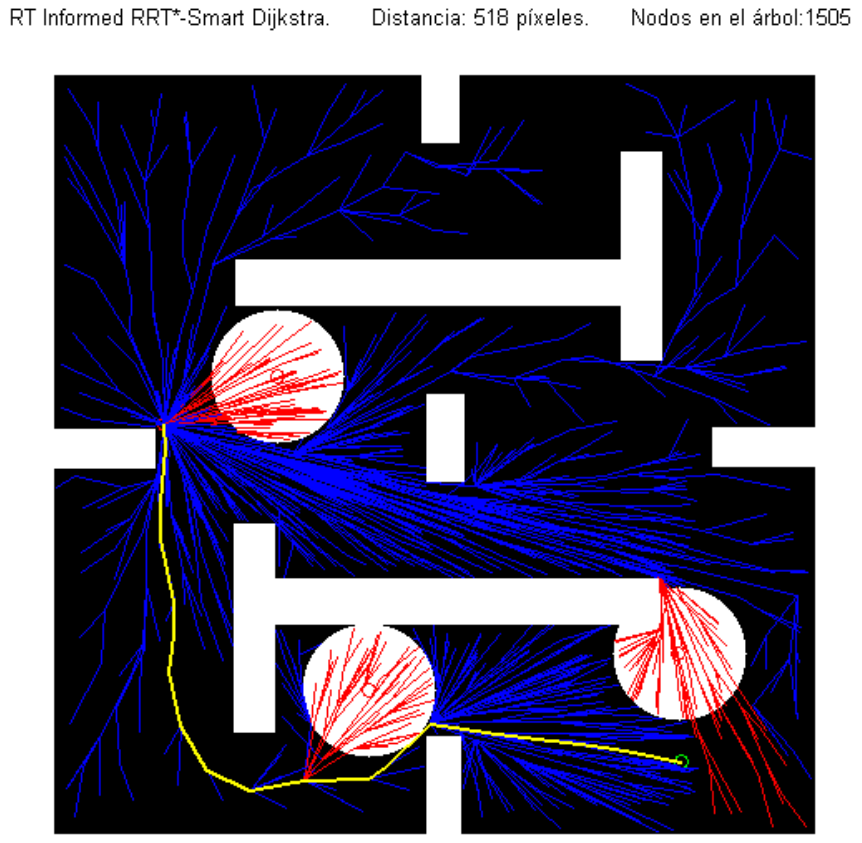

Figura 119. Ejemplo 1. Se ha conseguido encontrar una primera trayectoria que bordea los obstáculos móviles, aunque por la naturaleza aleatoria del algoritmo todavía dista de ser óptima.

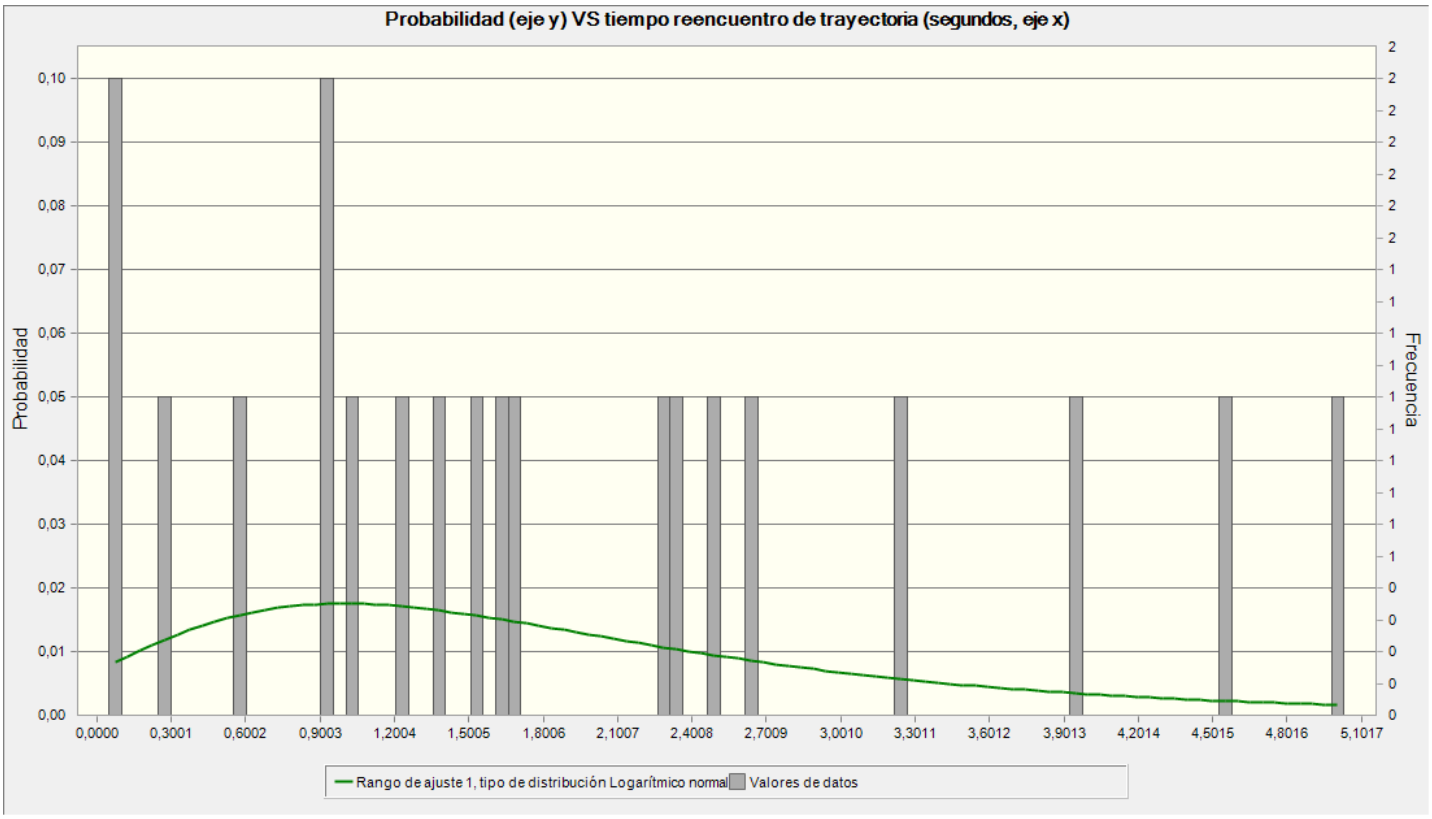

Figura 120. Ejemplo 1. Tiempo de reacción replanificando la trayectoria ante obstáculos móviles. Según el análisis de Montecarlo, la distribución logarítmica normal es la de mejor bondad de ajuste.

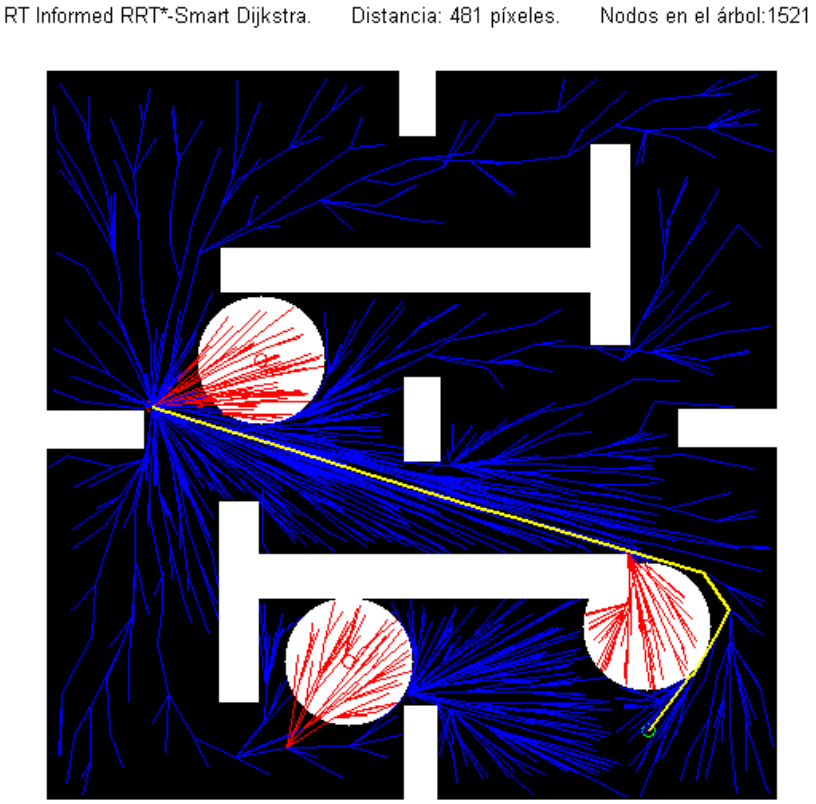

Figura 121. Ejemplo 1. Con el tiempo, el algoritmo propuesto vuelve e encontrar la trayectoria óptima.

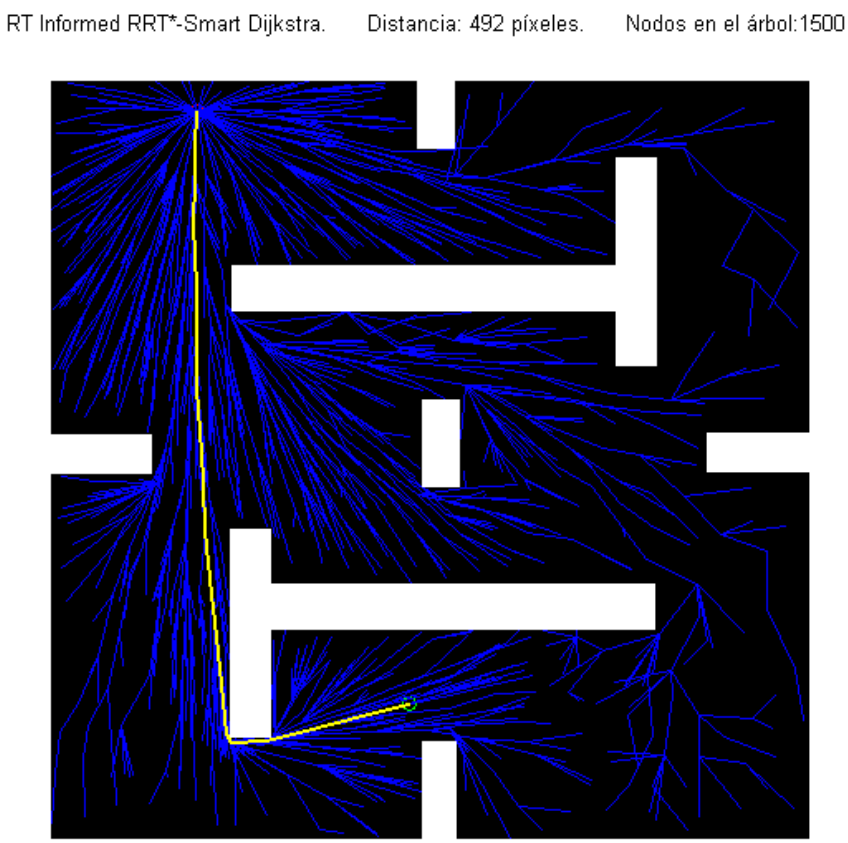

Figura 122. Ejemplo 2. Situación inicial óptima antes de que aparezcan los obstáculos móviles.

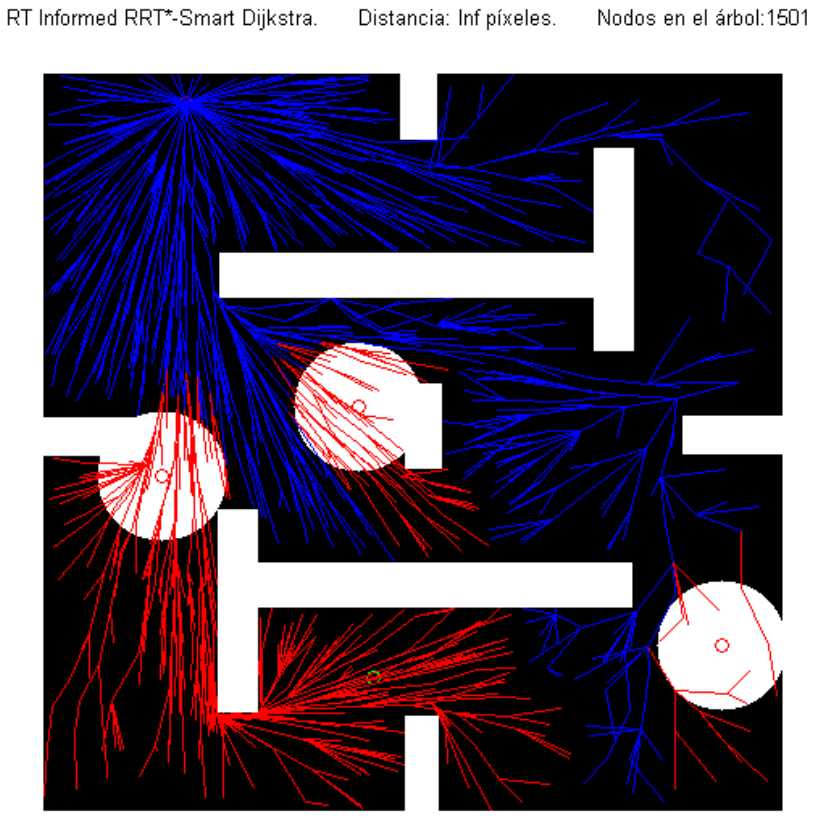

Figura 123. Ejemplo 2. Situación justo tras aparecer los obstáculos móviles. La trayectoria ha desaparecido.

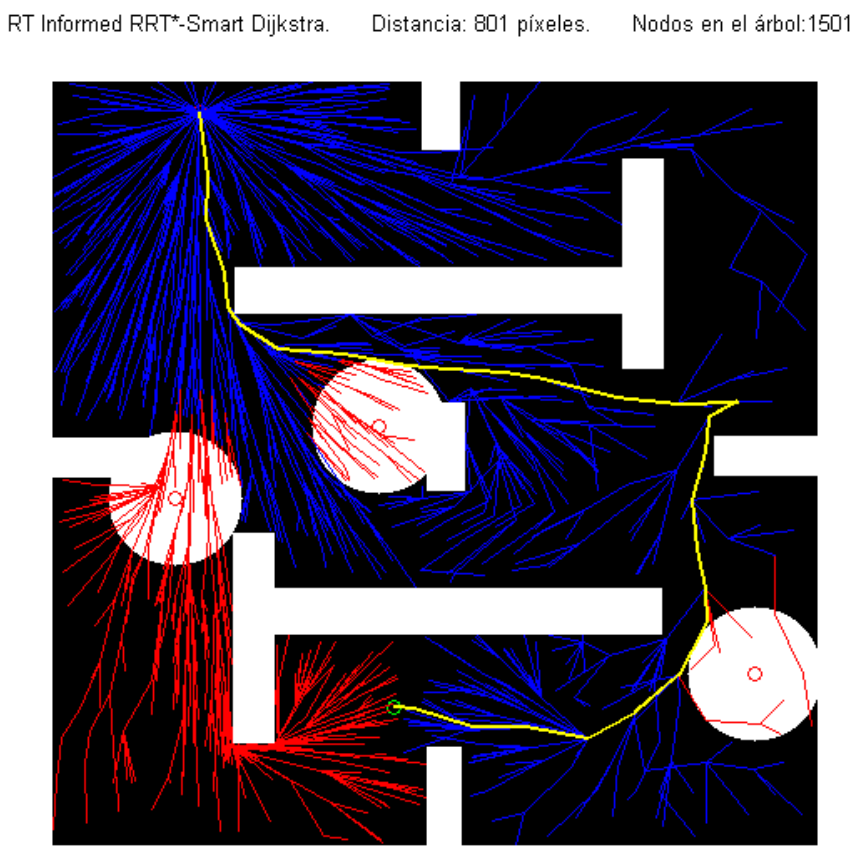

Figura 124. Ejemplo 2. Se ha conseguido encontrar una primera trayectoria que bordea los obstáculos móviles, aunque por la naturaleza aleatoria del algoritmo todavía no es óptima.

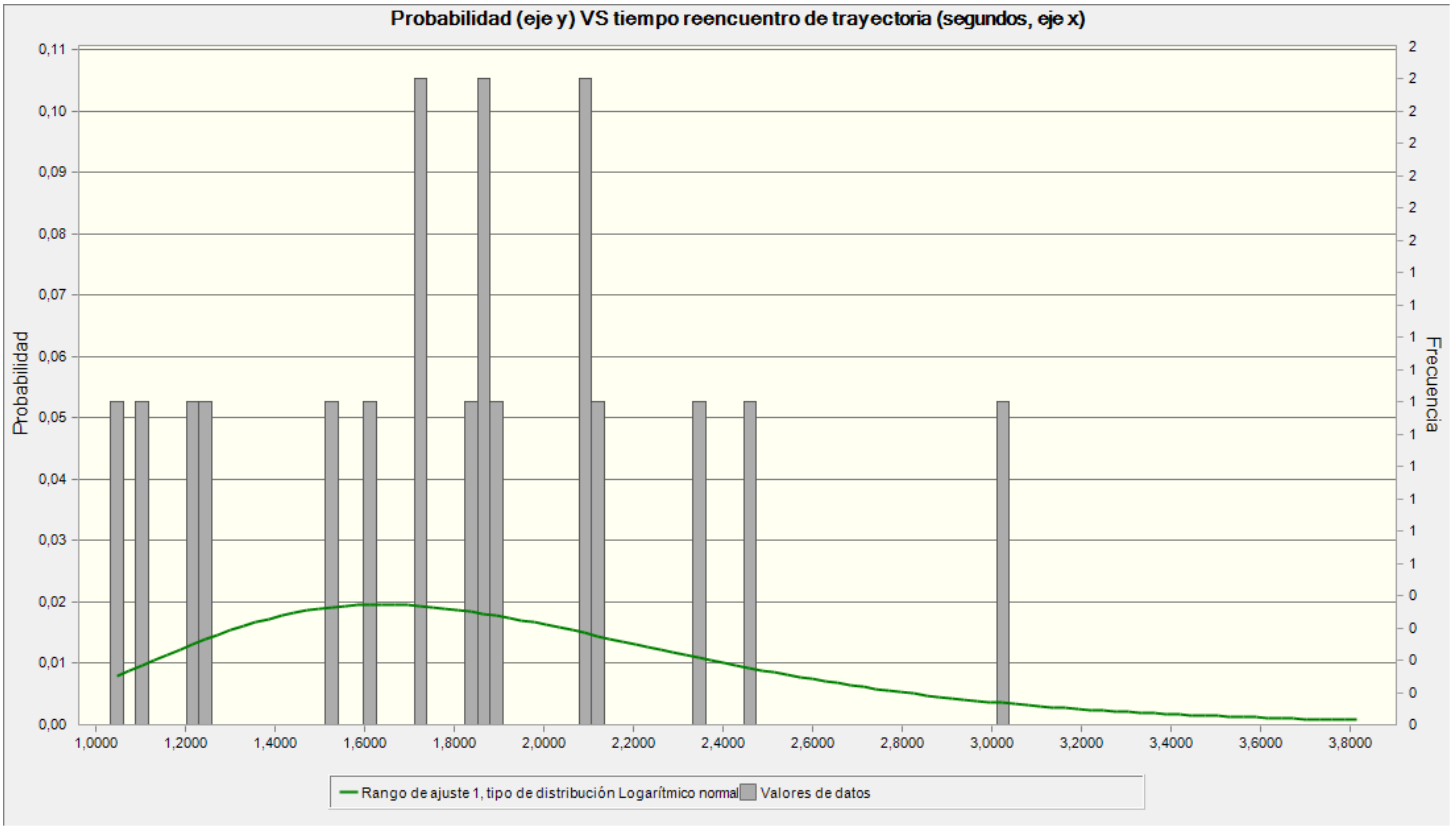

Figura 125. Ejemplo 2. Tiempo de reacción replanificando la trayectoria ante obstáculos móviles. Según el análisis de Montecarlo, la distribución logarítmica normal es la de mejor bondad de ajuste.

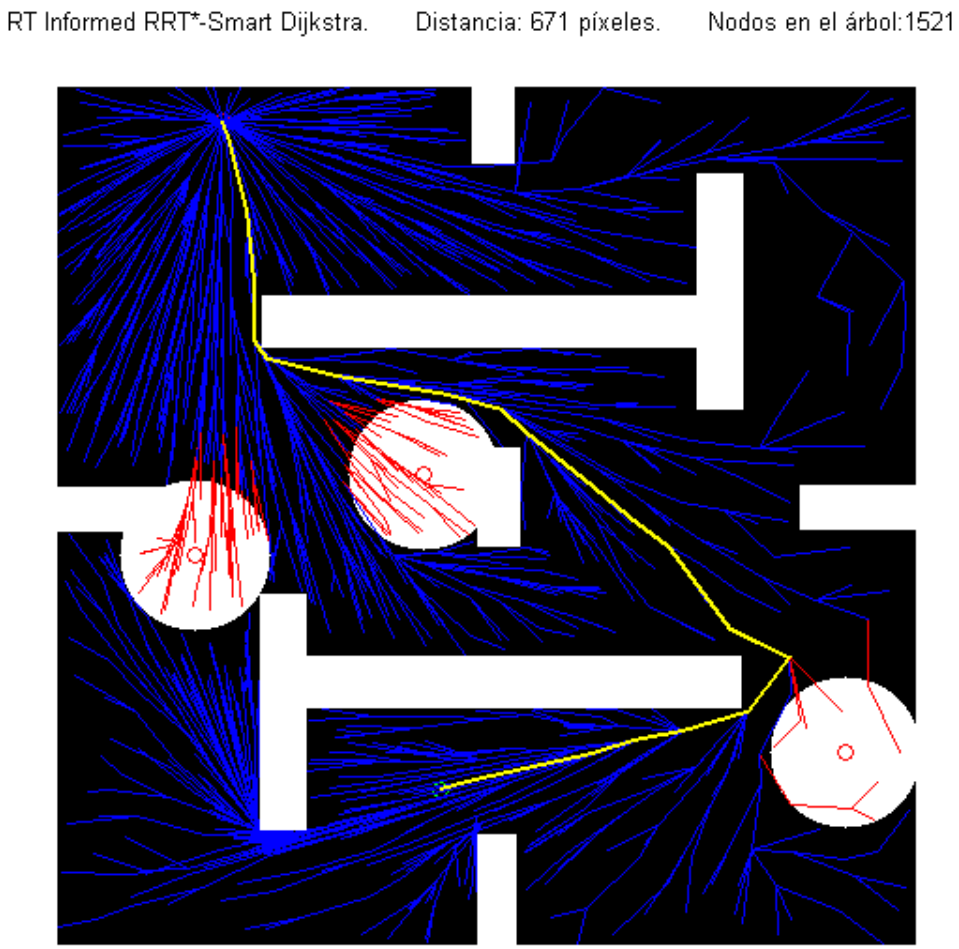

Figura 126. Ejemplo 2. Con el tiempo, el algoritmo propuesto se va acercando a la trayectoria óptima.

Decir que cuando aparece un obstáculo móvil, el planificador sabe desde el primer instante qué ramas del árbol han sido bloqueadas, trayectoria incluida. En otras palabras, el algoritmo es **reactivo**, desde el segundo cero puede dar la orden de frenar cuando detecta un obstáculo cerca. Sin embargo, la replanificación y optimización de la trayectoria sí que resulta más lenta, como se puede comprobar en los ejemplos anteriores.

En vista de los resultados, se puede decir que el algoritmo propuesto tiene una mediana del tiempo de reacción replanificando la trayectoria del orden de 1,2 – 1,8 segundos. Decir que este tiempo de reacción es muy dependiente del tipo de problema, como se puede apreciar en los ejemplos anteriores, donde el ejemplo dos (teniendo que pasar por un hueco más estrecho) tarda un 50% más de mediana que en el ejemplo uno.

Durante el análisis se observó que el planificador de optimización en tiempo real respondía mejor con mayor número de nodos calculados en la fase de optimización estática, esto tiene sentido ya que RT-RRT* está más focalizado en recablear nodos que en crearlos (como se puede ver en el pseudocódigo de Expande-y-Recablea).

REFERENCIAS

[1] «The RRT Page,» [En línea]. Available: http://msl.cs.uiuc.edu/rrt/about.html.

[2] «https://es.mathworks.com/,» [En línea].

[3] M. Vázquez, «Vida Revolucionaria,» [En línea]. Available: https://www.vidarevolucionaria.com/el-trabajo-en-la-era-de-los-robots-y-la-paradoja-de-moravec/.

[4] [En línea]. Available: https://www.logismarket.com.ar/hurtado-rivas/robot-manipulador/2132765000-1179608940-p.html.

[5] [En línea]. Available: http://www.robotreviews.com/wiki/comparison-of-roomba-500-series-robots.

[6] «electricBricks,» [En línea]. Available: http://blog.electricbricks.com/2010/07/holonomic-robot/ .

[7] «Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Holonomic_(robotics).

[8] G. E. G. Ángel Manuel Montes Romero, «SPD-Gonzales: robot sigue líneas, proyecto de asignatura "Laboratorio de Control",» Sevilla.

[9] «http://www.imdl.gatech.edu/haihong/Arm/Arm.html,» [En línea]. Available: http://www.imdl.gatech.edu/haihong/Arm/Arm.html.

[10] J. 1. T. C. o. R. M. P. M. P. Canny.

[11] R. A. E. S. d. I. U. d. S. G. Heredia y A. Ollero, «Apuntes de Robótica Avanzada. Planificación de caminos.».

[12] J. R. P. H. Kourosh Naderi, RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*.

[13] «The RRT Page,» [En línea]. Available: http://msl.cs.uiuc.edu/rrt/gallery_2drrt.html.

[14] J. J. Kuffner, "RRT-Connect: An Efficient Approach to Single-Query Path Planning", 200.

[15] S. K. E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning".

[16] F. Islam y J. Nasir, "RRT*-SMART: A Rapid Convergence Implementation of RRT*", 2013.

[17] J. D. Gammell y S. S. Srinivasa, "Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic", 2014.

[18] O. A. a. H. A. Varol, «Rapidly-Exploring Random Tree Based Memory Efficient».

[19] [En línea]. Available: https://www.ittgroup.ee/en/sensors-and-accessories/196-ir-reflective-sensor-tcrt5000l.html.

[20] [En línea]. Available: https://www.flipkart.com/robomart-ultrasonic-sensor-module-hc-sr-04/p/itme2zrvgz2hqznz.

[21] [En línea]. Available: https://www.wired.com/2010/11/gesture-controlled-3d-mapping-robot-just-add-kinect/.

[22] R. y. Boaz, 1987.

[23] D. A. López García, «Nuevas aportaciones en algoritmos de planificación para la ejecución de maniobras en robots autónomos no holónomos,» 2012.

[24] [En línea]. Available: http://webpersonal.uma.es/~VFMM/PDF/cap3.pdf.

[25] [En línea]. Available: http://www.dma.fi.upm.es/gregorio/JavaGC/Shortest_Path/html/Grafo_Visibilidad.htm#Grafo_Visibilidad_2 .

[26] C. Grima, «http://naukas.com/2011/12/23/cada-uno-en-su-region-y-voronoi-en-la-de-todos/,» [En línea].

[27] M. y. Elfes, 1985.

[28] S. y. Tamminen, 1985.

[29] Samet, 1989.

[30] Hayward, 1986.

[31] Hernian, 1986.

[32] S. y. Herb, 1992.

[33] F. y. Sammeti, 1988.

[34] H. y. otros, 1983.

[35] E. Dijkstra, «https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra,» [En línea].

[36] «Kavraki Lab,» [En línea]. Available: http://www.kavrakilab.org/robotics/prm.html.

[37] K. y. otros, 1996a.

[38] S. Lavalle, "Rapidly-exploring random trees: A new tool for path planning”, 1998.

[39] S. S. Ross Hatton, «"RRT Path Planning using a Dynamic Vehicle Model",» 2007.

[40] «http://www.ros.org/,» [En línea].

[41] M. Ruiz Arahal, «Bloque 1 de Sistemas de Percepción».

[42] «Joon's Lectures,» [En línea]. Available: http://joonlecture.blogspot.com.es/2011/02/improving-optimality-of-rrt-rrt.html.

[43] L. y. Kuffner, 2000.

[44] B. B. y. D. S. Michael Brunner, «Hierarchical Rough Terrain Motion Planning using an Optimal Sampling-Based Method».

[45] Oracle. [En línea]. Available: http://www.oracle.com/us/products/applications/crystalball/overview/index.html.

[46] [En línea]. Available: https://es.wikipedia.org/wiki/Microsoft_Excel.

[47] M. R. W. A. P. E. F. S. T. Sertac Karaman, «Anytime Motion Planning using the RRT*».

[48] Nilsson, 1969.

[49] Aurenhammer, 1991.

[50] L. P. y. Wesley, 1980.

[51] «https://losmisteriosdelatierra.es/homo-saurios-y-humanos-en-la-prehistoria-3211/,» [En línea].

[52] [En línea]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*.

[53] A. O. B. RÓBOTICA Manipuladores y robots móviles, Marcombo Boixareu Editores, 2007.