Universitat de les Illes Balears

TRABAJO DE FIN DE MÁSTER

ESTUDIO E IMPLEMENTACIÓN DEL ALGORITMO PROBABILÍSTICO DE PLANIFICACIÓN DE CAMINOS SFF, Y SU APLICACIÓN A LA RESOLUCIÓN DEL PROBLEMA DEL VIAJANTE DE COMERCIO

Juan Isaac Cifre Izquierdo

Máster Universitario en Ingeniería Industrial

Centro de Estudios de Postgrado

Año Académico 2021-22

ESTUDIO E IMPLEMENTACIÓN DEL ALGORITMO PROBABILÍSTICO DE PLANIFICACIÓN DE CAMINOS SFF, Y SU APLICACIÓN A LA RESOLUCIÓN DEL PROBLEMA DEL VIAJANTE DE COMERCIO

Juan Isaac Cifre Izquierdo

Trabajo de Fin de Máster

Centro de Estudios de Postgrado

Universidad de las Illes Balears

Año Académico 2021-22

Palabras clave del trabajo:

Algoritmos de planificación de caminos.
RRT: Rapidly-exploring Random Trees.
SFF: Space-Filling Forest.
SBPL: Search-Based Planning Library.
TSP: Travelling Salesman Problem.

Nombre Tutor/Tutora del Trabajo: Javier Antich Tobaruela

Universitat de les Illes Balears

Trabajo de Fin de Máster

MÁSTER EN INGENIERÍA INDUSTRIAL

Estudio e implementación del algoritmo probabilístico de planificación de caminos SFF, y su aplicación a la resolución del problema del viajante de comercio

JUAN ISAAC CIFRE IZQUIERDO

Tutor
Javier Antich Tobaruela

Centro de Estudios de Postgrado
Universidad de las Islas Baleares
Palma, 16 de septiembre de 2022

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

ACRÓNIMOS

ROS *Robot Operating System*

SBPL *Search-Based Planning Lab*

TSP *Travelling Salesman Problem*

MATLAB *MATrix LABoratory*

PRM *Probabilistic Roadmaps*

RRT *Rapidly-exploring Random Tree*

RRT-MO RRT Multi-Objetivo

SFF *Space-filling forest for multi-goal path planning*

NN *Nearest neighbor search*

GRASP *Greedy Randomized Adaptive Search Procedures*

SLAM *Simultaneous Localization And Mapping*

RESUMEN

Muchas aplicaciones dentro del campo de la robótica móvil requieren de algoritmos que calculen el camino que un robot debe recorrer para poder visitar un conjunto de puntos. A dicha tarea se la conoce como planificación multi-objetivo. Estos algoritmos son uno de los pilares de las arquitecturas de navegación de tipo deliberativo e híbrido, ya que son capaces de conseguir que un robot lleve a cabo de forma autónoma misiones complejas como la exploración y/o la vigilancia de entornos, el traslado y/o la manipulación de objetos, etc.

Por ello, se propone el estudio e implementación de un algoritmo probabilístico de planificación de caminos llamado *SFF* (*Space-Filling Forest*), con el cual se pretenden abordar situaciones en las que los puntos objetivo se encuentren en zonas de difícil acceso y por ello, el uso de algoritmos probabilísticos clásicos no sea del todo recomendable. Además, posterior a la ejecución de *SFF* y por lo tanto una vez obtenido un "*roadmap*", es decir, una red de caminos libres de obstáculos que conectan los puntos objetivos, se pretende calcular cual es la ruta más corta que permite visitar cada uno de ellos. Para resolver este problema, conocido como el problema del viajante de comercio (*TSP* o *Travelling Salesman Problem*), se incorpora un algoritmo adaptado a la situación y al tipo de *roadmap* obtenido una vez aplicado *SFF*.

Posteriormente, una vez implementado el algoritmo *SFF* e incorporado el *TSP*, se realizan una serie de pruebas para comprobar su correcto funcionamiento. Además, con el objetivo de poder comparar los resultados obtenidos, se implementa una versión del conocido algoritmo *RRT* (*Rapidly-exploring Random Trees*) adaptada a la resolución del problema del viajante de comercio.

CAPÍTULO 1

INTRODUCCIÓN

En este primer capítulo se presenta la motivación por la cual se ha llevado a cabo la realización del proyecto, además de los objetivos principales que se desean conseguir a lo largo de éste. Se explican de forma breve los distintos algoritmos de muestreo probabilístico que conforman el estado del arte actual, haciendo especial énfasis en el algoritmo que se ha decidido analizar. Finalmente, se expone la estructura del presente documento para así tener una mejor visión de cómo se ha enfocado el trabajo realizado.

1.1 Motivación

La necesidad de conocer de forma previa el camino que debe recorrer un robot entre dos o más puntos es una tarea primordial y uno de los campos de estudio más amplios dentro de la robótica móvil. Gracias al conocimiento de la ruta que se debe recorrer y la posible detección de obstáculos gracias a los dispositivos que lleva incorporado el robot, se pueden diseñar arquitecturas de navegación que permitan que se pueda desplazar a través del recorrido calculado en la mayor brevedad posible y con la mayor garantía respecto a la evasión de obstáculos.

En la figura 1.1, se puede observar el entorno de un hospital, el cual tiene una determinada cantidad de obstáculos a evitar. El robot debe viajar por cada uno de los puntos señalados para, por ejemplo, poder dejar la medicación a cada uno de los pacientes.

La mayor motivación de este proyecto es adentrarse en el campo de los algoritmos de planificación, es decir, aquellos que de forma previa nos ayudan a calcular el recorrido que el robot debe realizar para poder navegar a través de dos o más puntos del espacio.

Por ello y de forma introductoria, se puede mencionar que existen dos clases de algoritmos de planificación: los basados en grafos y en muestreo probabilístico.

- *Basados en grafos*. Los conocidos algoritmos de *Dijkstra* [9] [10], A* [11] [12] y Bellman–Ford [13] forman parte de este primer grupo. Este tipo de algoritmos utilizan un entorno conocido para formar una red de nodos conectados entre ellos.

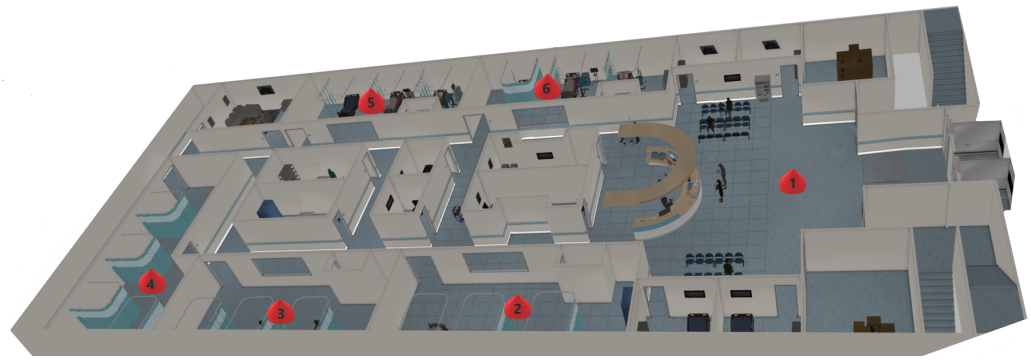


Figura 1.1: Ejemplo de entorno que representa un hospital. Se pueden ver, señalados en rojo, los puntos que el robot debe visitar para poder completar la misión. Para ello es de gran utilidad que de forma previa se haya calculado una ruta gracias a un algoritmo de planificación.

Cada par de nodos tiene un coste asociado representando el valor del desplazamiento que le supone al robot.

Por lo tanto, el objetivo de los algoritmos basados en grafos es encontrar el camino con un coste inferior, de esta forma se halla la ruta que permite al robot dirigirse hasta el punto objetivo de forma óptima. Por contrapartida, hay que tener en cuenta que encontrar el camino óptimo puede conllevar un elevado coste computacional.

- *Basados en muestreo probabilístico*. Tales como los algoritmos *PRM* [14] [15] y *RRT* [16] [17], se basan en la creación de puntos aleatorios alrededor del espacio de configuración[18] [19] libre de obstáculos. El espacio de configuración (C) define cada uno de los estados en los cuales el robot se puede encontrar. Estos puntos se generan en espacios libres de obstáculos, es decir, en el espacio de configuración libre de ellos (C_{free}), y a medida que se van creando se interconectan, formando así entramados de distinto tipo. Finalmente y si es posible, se consigue encontrar una ruta hasta el punto objetivo. A diferencia de los algoritmos basados en grafos, con el muestreo probabilístico se consigue explorar el entorno de forma más amplia y rápida. En contraposición, debido a que la ruta hasta el punto objetivo surge de una exploración aleatoria, es lógico razonar que la solución que se encuentre no sea la que cumpla con un coste mínimo.

En las figuras 1.2 y 1.3, extraídas de [20], se puede observar un ejemplo simple en el cual se compara un algoritmo basado en grafos (A*) y distintos algoritmos basados en muestreo probabilístico (*PRM*, *RRT* y RRT-Bidireccional).

En 1.2 se presenta el terreno explorado por cada algoritmo y el recorrido calculado. En 1.3 se puede observar como el tiempo de ejecución de los algoritmo *PRM*, *RRT* y RRT-Bidireccional es considerablemente inferior al de A*. Además, tal y como se ha comentado, el coste del camino es más elevado en el caso de los algoritmos probabilísticos, ya que A* es capaz de encontrar el camino óptimo.

Centrándonos en los *RRT*, se puede decir que a día de hoy, son una de las herramientas más utilizadas y eficientes de planificación de caminos cuando el número de dimensiones

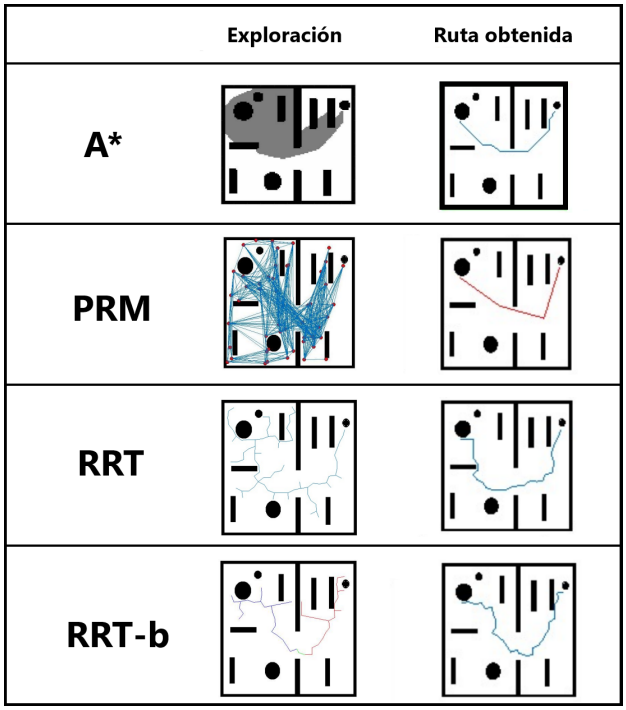


Figura 1.2: Comparación entre un algoritmo basado en grafos (A*) y distintos algoritmos basados en muestreo probabilístico (*PRM*, *RRT* y RRT-Bidireccional). Se puede observar el terreno explorado y el recorrido calculado por cada algoritmo.

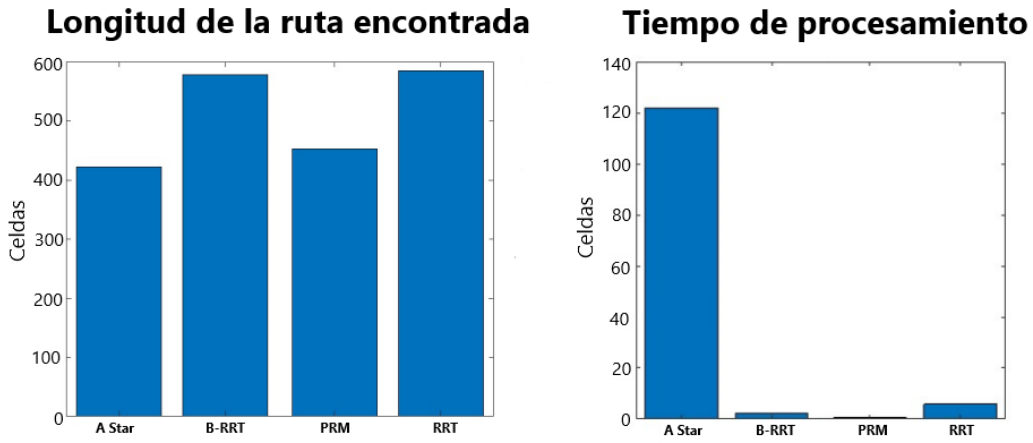


Figura 1.3: Comparación entre un algoritmo basado en grafos (A*) y distintos algoritmos basados en muestreo probabilístico (*PRM*, *RRT* y RRT-Bidireccional). A la izquierda se observa la longitud media de las rutas encontradas y a la derecha el tamaño de las celdas que se han explorado, lo que corresponde proporcionalmente al tiempo de ejecución del algoritmo.

del espacio de configuración es elevado. Por ello probablemente sean los que tienen una mayor popularidad dentro de los algoritmos de muestreo probabilístico. Esto es debido a que la versión original del algoritmo *RRT* [16] tiene dos características muy importantes

que lo diferencian del resto: es sencillo de comprender y de implementar.

Por ello, constantemente se investiga entorno a él y es habitual encontrar publicaciones científicas respecto a nuevas variantes o modificaciones. De esta forma, se puede mejorar para cubrir algún tipo de necesidad específica o simplemente con el objetivo de potenciar alguna de sus características.

En el siguiente apartado se muestran diversos algoritmos de planificación que se basan en *RRT*, estos se pueden clasificar como los más significativos e interesantes ya que son los que definen el estado del arte actual.

1.1.1 Algoritmos de muestreo probabilístico basados en *RRT*

Para poder elegir el algoritmo objeto de estudio, se ha realizado una búsqueda clasificándolos en dos grandes grupos:

- **Grupo 1**. Aquellos que no ofrecen ninguna garantía de optimalidad sobre el camino encontrado. A este grupo pertenecen, entre otros, el algoritmo RRT original y el algoritmo RRT-Bidireccional (de doble árbol).

  - Algoritmo RRT original [16]. Algoritmo en el que en cada iteración se pretende añadir una nueva rama a un árbol cuya raíz se encuentra en el punto inicial desde el cual el robot desea empezar a planificar. El árbol crece de forma probabilística sobre C_{free} hasta que una de sus ramas se encuentra lo suficientemente cerca del punto objetivo para decidir parar la búsqueda.
  - Algoritmo RRT-Bidireccional, más conocido como *RRT-Connect*. De forma similar al anterior algoritmo, en este caso se pretende crear un árbol en el nodo de partida y otro en el nodo objetivo. De esta forma, añadiendo comunicación entre los dos árboles y la capacidad de poder conectarlos, se puede conseguir que el algoritmo original sea más veloz ya que se explora el espacio de configuración con un árbol más.

- **Grupo 2**. Aquellos que ofrecen la garantía de que, en un tiempo infinito, el camino encontrado convergerá a la solución óptima. A este grupo pertenece el algoritmo RRT* original y, también, el algoritmo RRT* Smart (que acelera la convergencia al camino óptimo), entre muchos otros.

  - Algoritmo RRT* original [21]. Variante del algoritmo RRT original que pretende converger hacia una solución óptima. Cada vez que se añade un nodo al árbol, éste se acopla a uno de los nodos más cercanos el cual garantice que para llegar desde la raíz hasta el nodo que se desea añadir, se recorra la mínima distancia posible. De esta forma, cada vez que se añaden nuevos nodos a un árbol, se consigue que la distancia hasta la raíz de éste sea la óptima.
  - Algoritmo RRT* *Smart* [22]. Mejora del algoritmo RRT* el cual pretende solucionar las limitaciones de éste. Aunque RRT* tienda a converger hacia la solución óptima, lo hace en un tiempo infinito y con un bajo ratio de convergencia. Por ello, el principal objetivo de RRT* *Smart* es acelerar este ratio de convergencia para encontrar la solución óptima mucho más rápido y así reducir considerablemente el tiempo de ejecución.

***Space-filling forest for multi-goal path planning* (*SFF*)**

Dentro de los algoritmos del primer grupo se encuentra éste. Se ha separado del resto debido a que se le desea hacer una mención especial. Se trata de un algoritmo publicado en 2018 por Vojtěch Vonásek y Robert Pěnička [23] el cual propone una nueva forma de explorar el espacio para poder encontrar una ruta entre una serie de puntos objetivo. El robot debe pasar por cada uno de ellos para poder cumplir la misión, normalmente se trata de misiones de vigilancia o recolección de información.

Esta tarea se puede llevar a cabo con *RRT* o con *PRM*, pero cada uno de ellos tiene una serie de inconvenientes, los cuales *SFF* trata de suplir.

Para poder cumplir con una misión multi-objetivo, *RRT* necesita generar un árbol desde cada uno de los puntos de forma secuencial, hasta explorar el último punto, lo que puede ser costoso computacionalmente.

PRM genera un grafo en el espacio de configuración, en el cual se pueden encontrar caminos entre los puntos objetivo en un solo intento. *PRM* también intenta conectar todos los nodos con sus vecinos más cercanos (independientemente de si son relevantes o no) para encontrar caminos entre los puntos objetivo, por lo que el tiempo de ejecución también puede verse perjudicado.

Un problema común de estos dos algoritmos que SSF desea menguar, es el hecho de que para los algoritmos de planificación, de forma genérica, suele ser costoso explorar secciones del entorno en las cuales se encuentran pasillos estrechos o zonas de difícil acceso. Sobre todo, si a un entorno que tenga grandes cantidades de espacio libre se le introducen regiones de este tipo, a *RRT* y *PRM* les puede costar encontrar rutas entre los objetivos. Por ello, de nuevo, el tiempo de planificación tiende a aumentar.

SFF se basa en *RRT*, la base de su funcionamiento es la de crear, desde cada punto objetivo que se quiera explorar, un árbol de forma independiente a los demás.

Cada uno de los árboles explora una región aleatoria de C_{free} y éstos detienen su crecimiento cuando cumplen una serie de condiciones. De esta forma se consigue explorar el espacio de forma uniforme, con una cantidad de nodos inferior a *PRM* y a diferencia de *RRT*, se consiguen encontrar los caminos entre los puntos objetivo en un solo intento.

Por ello, al mejorar los principales inconvenientes de *RRT* y *PRM*, se consigue un algoritmo que es posible que mejore el tiempo de planificación.

En las figuras 1.4, 1.5 y 1.6 se pueden observar las exploraciones que realizan *RRT*, *PRM* y *SFF*. Los puntos azules representan, en el caso de *RRT* y *PRM*, los nodos de inicio y fin. En el caso de *SFF*, representan todos los puntos que se desean visitar.

1.2 Objetivos

Tal y como se ha comentado previamente, en este proyecto se desea conocer en profundidad el funcionamiento de los algoritmos de planificación basados en muestreo probabilístico, y más concretamente, los que se basan en *RRT*. La sencillez para comprenderlos y su simpleza a la hora de implementarlos hacen que los *RRT* sean un tipo de algoritmo atractivo para adentrarse en su investigación.

Por ello, gracias a la búsqueda realizada en el apartado 1.1.1, se ha decidido estudiar, implementar y evaluar el algoritmo *SFF*.

Con su utilización, se pretende explorar un espacio definido para poder encontrar caminos entre una serie de puntos objetivo.

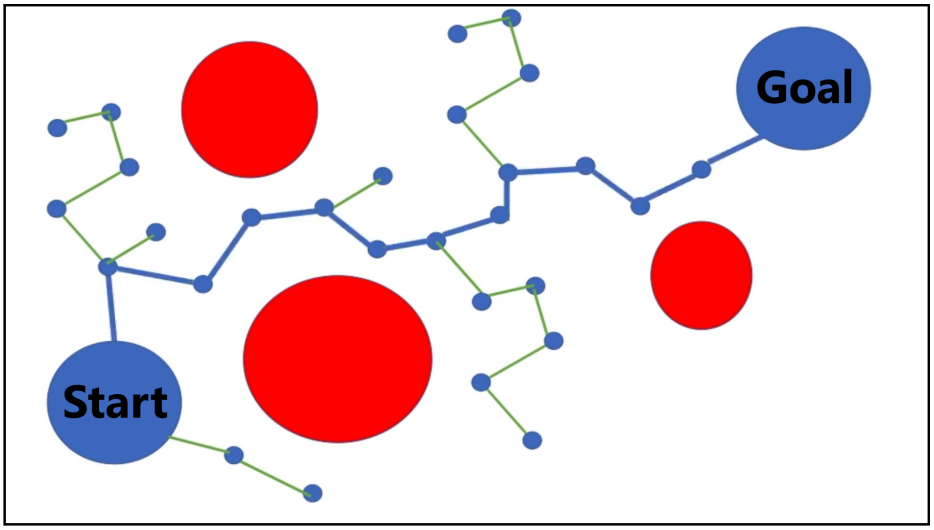


Figura 1.4: Ejemplo de exploración del algoritmo *RRT*. Podemos ver de color azul los nodos de inicio y fin, de color verde el árbol que se ha creado y resaltado en azul el camino que se debe recorrer para llegar al nodo objetivo.

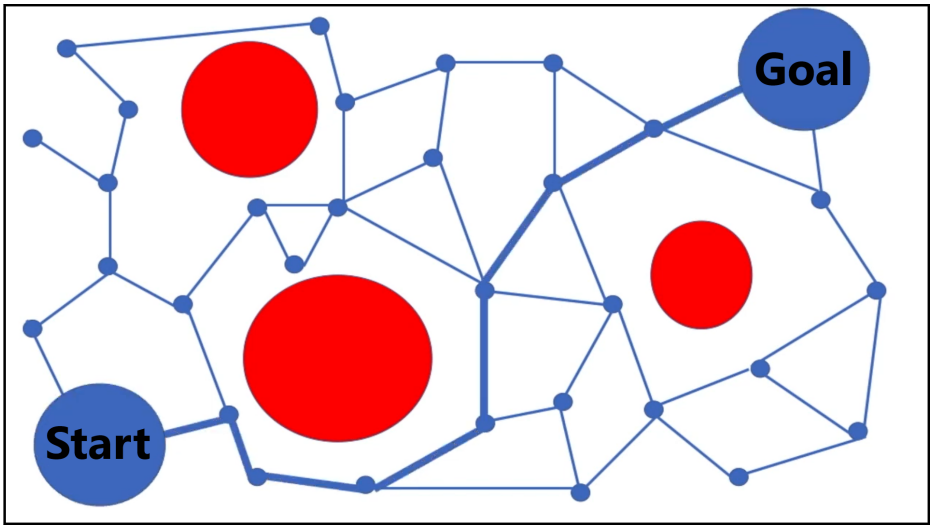


Figura 1.5: Ejemplo de exploración del algoritmo *PRM*. Podemos ver de color azul los nodos de inicio y fin, posteriormente se puede observar el grafo que *PRM* ha generado y a partir del cual, podrá realizar cualquier recálculo para poder encontrar el camino desde un punto inicial a uno objetivo. De color azul se observa el camino que se debe seguir para llegar al punto seleccionado.

Posteriormente, tal y como menciona su publicación, se desea utilizar como base para poder resolver el problema del viajante o *TSP* (*Travelling Salesman Problem*) [24] y así, poder encontrar el camino más corto posible que pasa por cada uno de los puntos objetivo y que finaliza en el punto inicial.

En la figura 5.41 se observa el resultado de una ejecución de *SFF* juntamente con *TSP*. Se puede apreciar un ejemplo sencillo de creación de árboles y del cálculo que realiza el

Figura 1.6: Ejemplo de exploración del algoritmo *SFF*. Se observa cómo de cada uno de los puntos surge un árbol y, de color negro, se pueden observar los caminos que se han generado entre ellos.

algoritmo TSP sobre los caminos (*roadmap*) calculados por SFF.

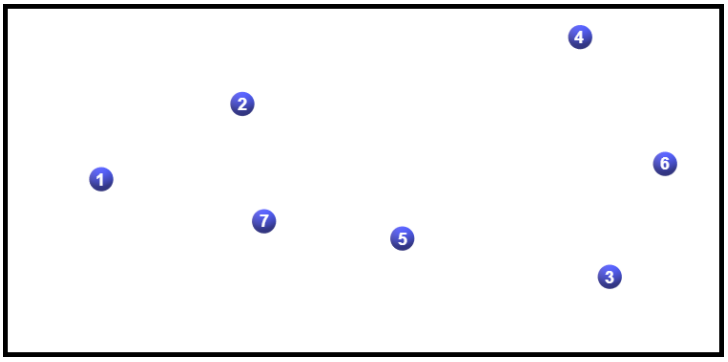
Finalmente, se llevarán a cabo una serie de pruebas para poder analizar las ventajas y desventajas del algoritmo. De esta forma y centrándonos en su comparación con *RRT*, se puede llegar a la conclusión de, para el problema multi-objetivo enunciado, cuál de los algoritmos presenta mejores resultados.

1.3 Estructura del documento

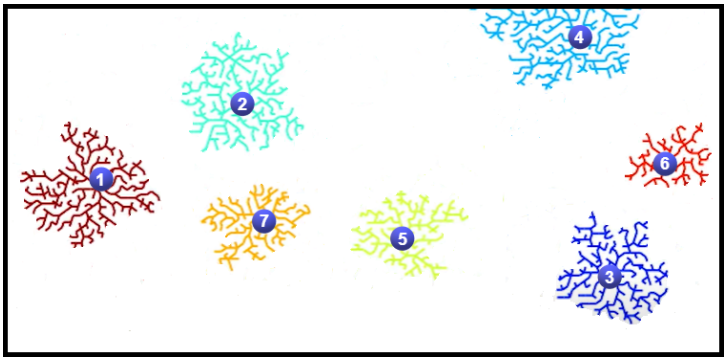
La memoria está organizada en los siguientes capítulos:

- **Capítulo 2**. Herramienta utilizada: en este capítulo se explica la principal herramienta que se ha utilizado durante la realización del proyecto.
- **Capítulo 3**. Descripción de los algoritmos *SFF* y *RRT* además del problema *TSP*: en este capítulo se explica en detalle el funcionamiento de los algoritmos *SFF* y *RRT*. Además, también se explica el problema *TSP* en detalle.
- **Capítulo 4**. Implementación de los algoritmos *SFF*, *RRT* e incorporación de la librería *TSP*: en este capítulo se detalla cada uno de los pasos que se han seguido para implementar los algoritmos *SFF* y *RRT* juntamente con todas las variables, funciones y herramientas que se han utilizado para poder llevar a cabo su implementación. Por otro lado, también se detalla como se incorpora el algoritmo *TSP* tras la utilización de *SFF* para poder resolver el problema propuesto.
- **Capítulo 5**. Resultados obtenidos: en este capítulo se llevan a cabo una serie de pruebas para comprobar que los algoritmos se han implementado correctamente y para observar cómo se comportan en función de como se modifiquen los parámetros principales que rigen el comportamiento de cada uno. Además, también se realizan una serie de pruebas para determinar qué algoritmo *TSP* se utiliza. Finalmente, se realiza una comparación entre *SFF* y *RRT* en una serie de entornos definidos para así determinar cuáles son las fortalezas y debilidades de cada uno.

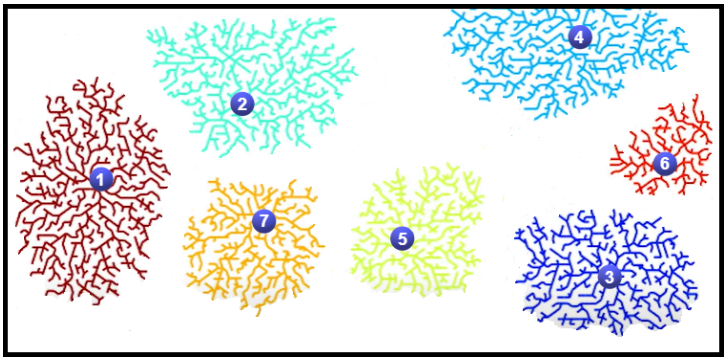
- **Capítulo 6**. Conclusiones y futuras mejoras: en este capítulo se valoran los resultados obtenidos y se proponen diversas mejoras que se podrían realizar al trabajo realizado en función de las debilidades más importantes que se han encontrado.



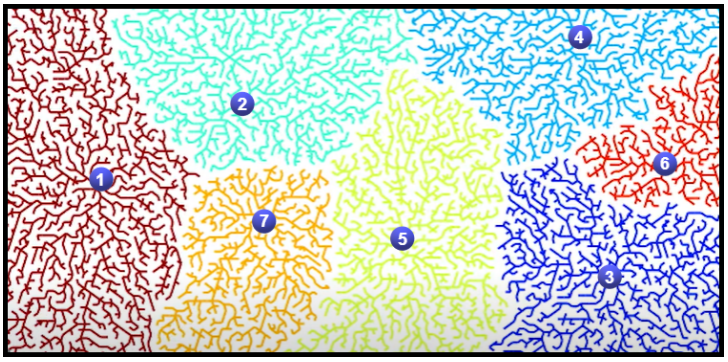
(a) Conjunto de 7 nodos en un entorno vacío.



(b) De cada nodo empieza a crecer un árbol.



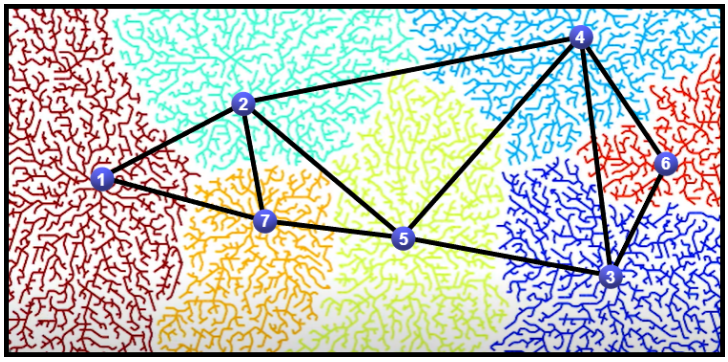
(c) Cada árbol prosigue su crecimiento.



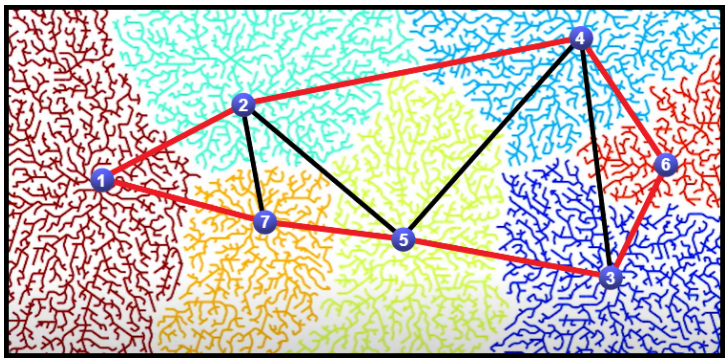
(d) Finalmente, cada árbol se acaba de expandir.

Figura 1.7: Resultado de la ejecución del algoritmo *SFF* y posterior utilización del algoritmo *TSP*.

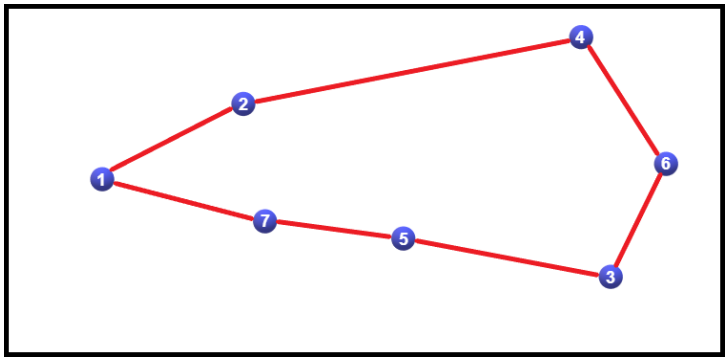
(e) Se forma el *roadmap* entre cada uno de los puntos objetivo



(f) Se suaviza el *roadmap* y en este caso, se obtienen líneas rectas que unen los puntos objetivo.



(g) Se aplica el algoritmo *TSP* para obtener la ruta que permite viajar por cada uno de los puntos objetivo.



(h) Finalmente, únicamente se muestra la ruta que se debe seguir.

Figura 1.7: Resultado de la ejecución del algoritmo *SFF* y posterior utilización del algoritmo *TSP*.

CAPÍTULO 2

HERRAMIENTA UTILIZADA

A continuación, se explica la principal herramienta que se ha utilizado durante la realización del proyecto.

2.1 *Search-Based Planning Library* (*SBPL*)

Librería desarrollada por Maxim Likhachev en la Universidad de Pennsylvania en colaboración con Willow Garage [25]. Contiene una serie de algoritmos de planificación basados en grafos que resuelven el problema de encontrar un camino desde un punto inicial a un punto objetivo. A estos algoritmos comúnmente se les denomina heurísticos [26], ya que tienen implementada algún tipo de función heurística que permite encontrar la solución óptima al problema.

Los algoritmos que contiene realizan las búsquedas a través de grafos y los entornos se definen como un conjunto de celdas. En las figuras 2.1, 2.2 y 2.3 se observan tres ejemplos visuales del desarrollo de las búsquedas de un punto inicial a uno objetivo utilizando los algoritmos *Dijkstra*, A* y *weighted* A* [27] respectivamente.

SBPL es una librería independiente (*standalone*) [28] desarrollada en C++ la cual se puede utilizar bajo Linux o Windows. Cabe decir que dos de los usos más comunes que se le pueden dar a la librería son:

- Fines didácticos. Se puede aprender cómo funcionan los algoritmos de planificación heurísticos ya que se ve como están implementados. Además, gracias a los ejemplos que se pueden llevar a cabo con los entornos que lleva incorporada la librería, se puede entender mucho mejor su funcionamiento.

- Utilización junto a *ROS* [29] [30]. *ROS* es una plataforma que permite el desarrollo de *software* orientado al campo de la robótica. Por ello, su uso como un paquete de *ROS* posiblemente sea la forma más común de utilizarla. También es frecuente combinar su uso con el del paquete GMapping [31], el cual es un paquete que permite crear mapas mediante el algoritmo *SLAM* (*Simultaneous Localization And*

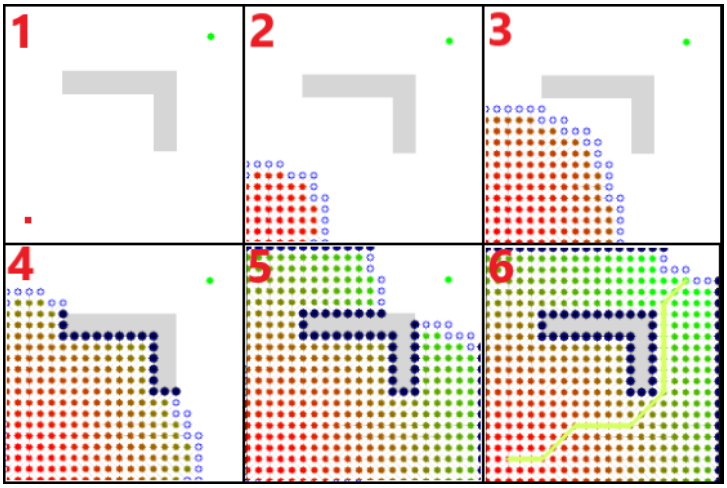


Figura 2.1: Ejemplo de exploración del algoritmo *Dijkstra*. El punto rojo representa el inicio y el punto verde el objetivo. Se observa como *Dijkstra* realiza una exploración de todos los nodos que se encuentran alrededor mientras no se encuentre el punto objetivo.

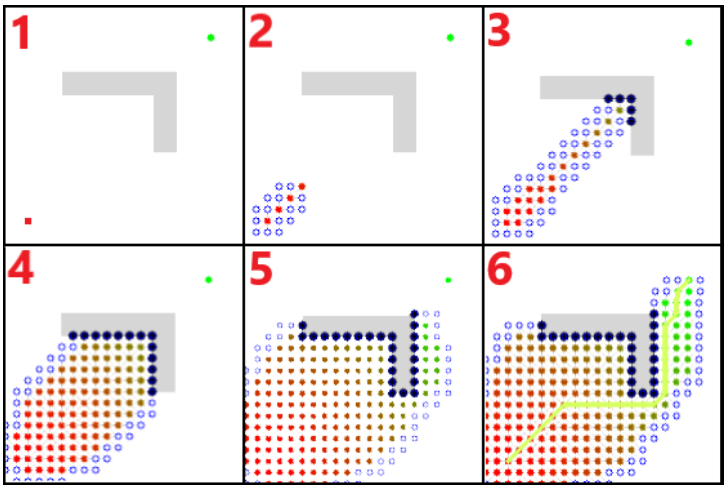


Figura 2.2: Ejemplo de exploración del algoritmo A*. El punto rojo representa el inicio y el punto verde el objetivo. Se observa como, al encontrarse con un obstáculo, el algoritmo decide explorar los dos lados de éste de forma homogénea.

Mapping) [32]. De esta forma, se puede generar un mapa con GMapping gracias a los sensores que tenga incorporado el robot para posteriormente, gracias a *SBPL*, poder planificar el camino que el robot tiene que seguir. Así, a medida que el robot avanza sobre un entorno desconocido, se puede ir generando el mapa con GMapping y realizar la planificación con *SBPL*.

En la figura 2.4 se puede ver de forma genérica cómo funciona la librería y cómo se utiliza:

- En una situación donde el entorno es parcialmente conocido y por ello se tenga que realizar una navegación local, el mapa que se genera no contempla todo el entorno disponible y por eso es necesario actualizarlo a medida que se planifica.

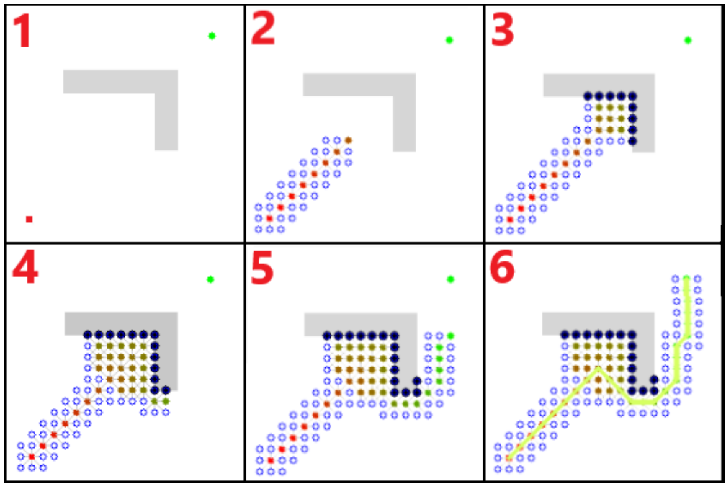


Figura 2.3: Ejemplo de exploración del algoritmo *weighted* A*. El punto rojo representa el inicio y el punto verde el objetivo. A diferencia de A*, al encontrarse con el obstáculo, el algoritmo decide seguir explorando por el lado que se encuentra más próximo del punto objetivo.

- En una situación donde se dispone del entorno en su totalidad desde el inicio, es suficiente con realizar una planificación para conocer la ruta que se debe seguir hasta llegar al punto objetivo.

Para poder entender mejor cómo está estructurada, la librería se puede dividir en tres grandes grupos:

1. Entornos (*environments*). Nos ayudan a representar el problema como un grafo.
2. Algoritmos de planificación (*Planners*). Resuelven el problema definido con métodos de búsqueda de grafos, es decir, con los algoritmos de planificación heurísticos.
3. Otros aspectos, sobretodo de configuración, importantes para el uso correcto de la librería. Entre ellos se incluyen los ficheros de configuración del entorno y la representación de la trayectoria calculada.

2.1.1 Entornos (*Environments*)

El objetivo de cada uno de los entornos definidos en la librería es el de definir el espacio de configuración del robot como un grafo. De esta forma, el espacio se puede definir en una serie de estados de configuración en los cuales se puede encontrar el robot. Dependiendo del problema que se desea plantear, se utiliza un tipo distinto de entorno.

Por ejemplo, si se quiere definir un entorno para un vehículo de cuatro ruedas, se deben utilizar las coordenadas x, y y la orientación θ. En cambio, para un dron, se debe utilizar x, y, z, *roll*, *pitch*, and *yaw*.

Cada uno de los posibles estados de configuración se representa como un nodo en el grafo, el cual tiene un identificador único. De esta forma, para el ejemplo del vehículo de 4 ruedas, si definimos un entorno que sea de 500·500 celdas y suponemos que el robot solo se puede encontrar en 1 de 4 orientaciones posibles, se tendría un espacio de configuración

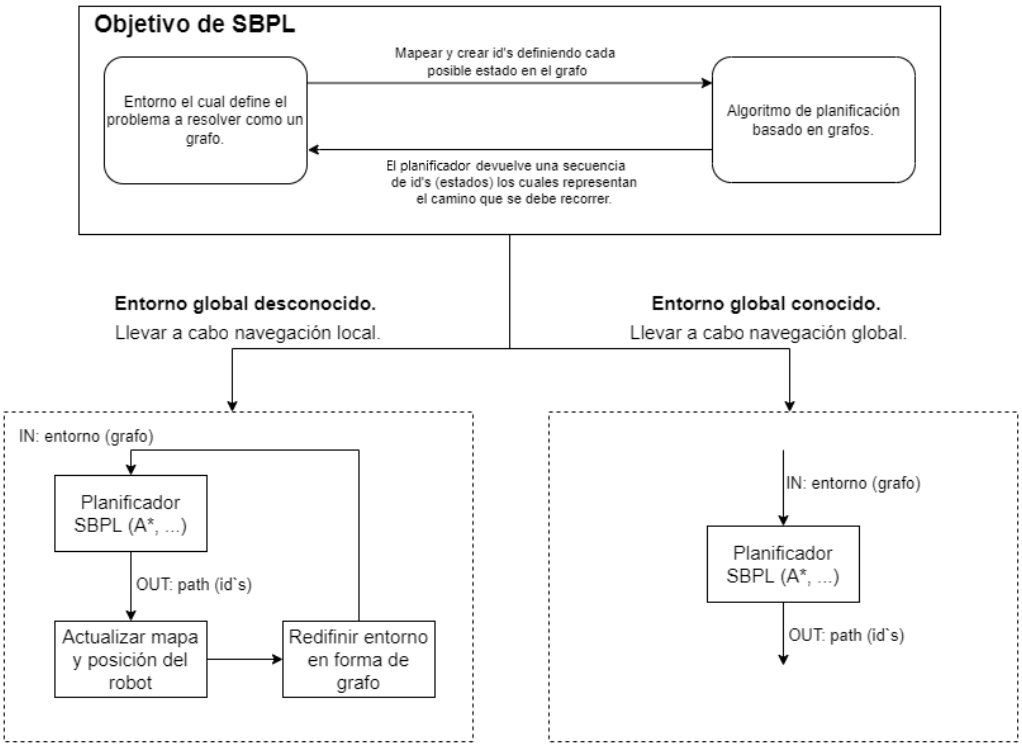


Figura 2.4: Visión global del funcionamiento de *SBPL*. Se puede observar cómo dependiendo de como sea el entorno a explorar, se puede utilizar de distinta forma.

con 500·500·4=1000000 de identificadores distintos. En este entorno ideal, cualquier par de identificadores adyacente nos permitiría el movimiento del robot.

Una de las grandes ventajas es que, independientemente de las dimensiones con las que se quiera trabajar, siempre se podrá definir el problema como un grafo bidimensional con identificadores únicos.

Los principales espacios de configuración de *SBPL* se definen a continuación. En la figura 2.5 se puede ver un esquema de ellos. En el centro podemos observar la clase que agrupa todos los entornos: *DiscreteSpaceInformation*. Se encarga de establecer todas las funciones necesarias para que los planificadores se puedan comunicar correctamente con cada tipo de entorno.

- *Environment2D*. Para la navegación de robots en coordenadas x,y.
- *Environment2DUU*. Ampliación del entorno Environment2D diseñado para insertar incertidumbre.
- *EnvironmentXYTHETALAT*. Para la navegación de robots en coordenadas x, y, θ.
- *EnvironmentXYTHETALEVLAT*. Para la navegación de robots en coordenadas x,y, θ. Además, permite la detección de obstáculos.
- *EnvironmentROBARM*. Para la planificación de brazos robóticos 7D. Es decir, con 7 articulaciones y por lo tanto 7 grados de libertad.

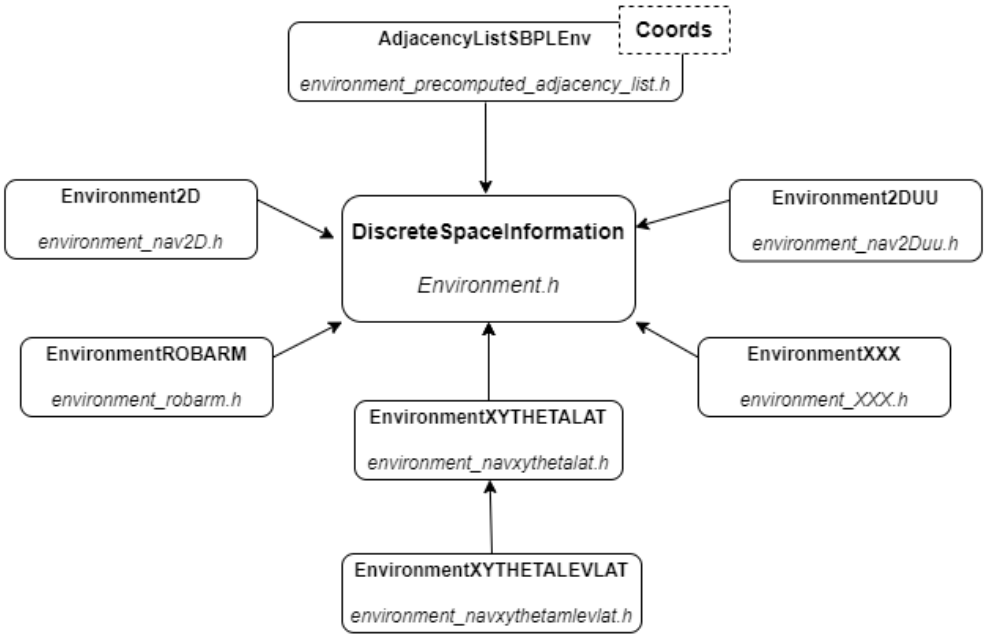


Figura 2.5: Esquema de la organización interna de los entornos de *SBPL*. Se puede ver como la clase *DiscreteSpaceInformation* es el centro de cada uno de los entornos.

- *AdjacencyListSBPLEnv*. Entorno representado como un grafo de tipo lista de adyacencia [33].

- *EnvironmentXXX*. Se utiliza como plantilla para que el usuario que utilice la librería, en función del robot que utilice o dependiendo de sus necesidades, pueda definir y crear entornos personalizados.

2.1.2 Planificadores (*Planners*)

En este punto se recopilan los distintos algoritmos que incorpora *SBPL* para poder solucionar el problema que se plantee y así poder encontrar un conjunto de nodos (identificadores) el cual nos indique como se tiene que mover el robot para poder llegar hasta el punto objetivo.

Este conjunto de nodos no tiene que ser el óptimo, ya que puede ser que consuma excesivo tiempo y memoria. Se pueden utilizar diversos parámetros para modificar los planificadores y así encontrar un equilibrio entre la calidad de la trayectoria y tiempo que se tarda en calcular. Todo esto se consigue gracias a las funciones heurísticas que se pueden introducir a cada uno de los algoritmos de planificación.

Sin entrar en excesivo detalle ni en aspectos de implementación, a continuación se muestra una breve descripción de cada uno de los algoritmos que contiene la librería *SBPL*.

Además, en la figura 2.6 se observa un esquema de como la librería los organiza.

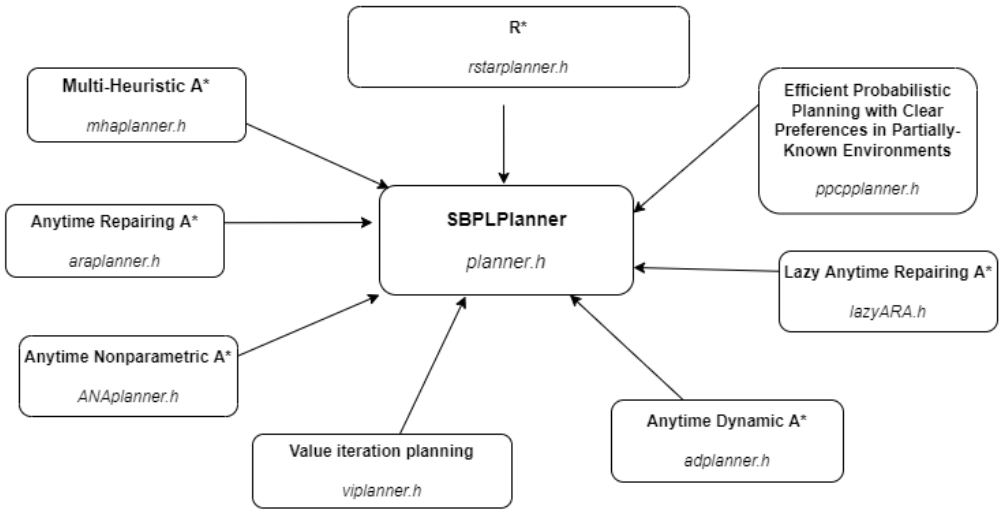


Figura 2.6: Esquema de la organización interna de los algoritmos de planificación de *SBPL*. Se puede observar como la clase *SBPLPlanner* es el centro de cada uno de los planificadores.

Planificador *Anytime Repairing A** **(ARA*) [34]**

Variante de A* la cual es capaz de modificar su rendimiento de búsqueda en función del tiempo disponible. Gracias a una variable (ε) que se define al principio de la ejecución, se encuentra una solución no óptima de forma rápida.

Posteriormente, en función del tiempo de búsqueda programado, esta variable se ajusta para poder mejorar la solución inicial y así obtener un resultado de mayor calidad. Si se configura el tiempo de búsqueda adecuado, es posible que encuentre el camino óptimo. Para poder mejorar el camino se utiliza el tiempo de reparación (*repair time*), de esta forma gran parte de los cálculos realizados en búsquedas anteriores se reutilizan para encontrar un camino mejor, lo que hace que se convierta en un algoritmo realmente eficiente.

Planificador *Anytime Nonparametric A** **(ANA*) [35]**

Utiliza un parámetro ε, el cual en cada iteración ajusta su valor para así poder expandir el nodo que se considera más prometedor. De esta forma se realiza una búsqueda muy optimizada que ayuda a conseguir, en comparación con ARA*, un camino de mayor calidad. En su publicación muestran, con los resultados obtenidos, que ANA* es tan eficiente como ARA* y que, en la mayoría de los casos:

- Se encuentra una solución inicial más rápido.
- ANA* tarda menos tiempo a encontrar mejoras de la solución inicial.
- En definitiva, acaba encontrando la solución óptima más rápidamente.

Planificador *Anytime Dynamic A** **(AD*) [36]**

Este algoritmo se enfoca en el objetivo de poder encontrar una solución que unifique los dos principales beneficios que tienen los algoritmos de replanificación:

1. Debido a que los entornos que se utilizan son dinámicos, constantemente se debe corregir el camino previamente calculado para poder obtener uno nuevo.

2. Cálculo *anytime* [37], es decir, debido a que la solución del problema es compleja, se pretende ofrecer una lo suficientemente óptima ajustándose al tiempo de búsqueda del que se disponga.

Este algoritmo pretende unir estas dos características de los algoritmos de planificación. Para poder conseguirlo, continuamente se mejora la solución inicial calculada en función del tiempo restante de búsqueda y además, también se corrige continuamente a medida que va recibiendo información del entorno. Cabe decir que funciona especialmente bien en entornos con muchas dimensiones, parcialmente conocidos y dinámicos, los cuales cambian constantemente.

Planificador *Lazy Anytime Repairing A** (LAZYARA*) [38]

Esta es la version *lazy* [39] [40] del algoritmo ARA*. Las versiones *lazy* suelen ser las que sacrifican alguna de las características del algoritmo para así hacer que sea computacionalmente más ágil.

A diferencia de ARA*, no tiene en cuenta el tiempo de búsqueda restante y únicamente se calcula una posible mejora al finalizar la ejecución del algoritmo.

Planificador *Efficient Probabilistic Planning with Clear Preferences in Partially-Known Environments* (PPCP) [41]

La mayoría de los algoritmos de planificación trabajan con entornos parcialmente conocidos. Este algoritmo se centra en darle importancia a los valores de la información desconocida que podrían dar como resultado una mejor planificación. Es decir, existe una clara preferencia por los valores de la información faltante.

Parte de su implementación se basa en A*, de esta forma se pueden realizar búsquedas deterministas para construir e ir mejorando el resultado en cualquier momento. Sobretodo se utiliza para la programación de brazos robóticos.

Planificador R* [42]

A diferencia de A*, este algoritmo se enfoca en problemas de planificación más grandes y complejos. En estas situaciones, es posible que A* guíe la búsqueda hacia un gran mínimo local, esto es debido a la dependencia que se tiene de la función heurística a la hora de guiar al robot hacía el punto objetivo.

En cambio, R* propone que se dependa mucho menos de dicha función. De esta forma, se realizan búsquedas de corto alcance que sean sencillas de resolver, cada una de ellas guiada por la función heurística hacia un nodo elegido al azar.

Planificador *Value iteration planning* (VI) [43]

Este planificador ofrece las funcionalidades de lo que se conoce como *value iteration planning*. A diferencia de los demás algoritmos, los cuales son deterministas, este es un algoritmo estocástico [44]. Es decir, está sometido al azar y es objeto de análisis estadístico. A grandes rasgos, a diferencia de los otros planificadores deterministas:

Environment2D	
Parámetro	**Descripción**
discretization(cells)	Este es el número total de celdas del cual dispone el espacio de configuración.
obsthresh	Coste a partir del cual una celda se considera como obstáculo.
start(cells), *end(cells)*	Celdas correspondientes a la posición inicial del robot y al punto objetivo el cual se quiere alcanzar.
environment	Este es el entorno, el cual se define como una matriz a partir de *discretization(cells)*.

Tabla 2.1: Parámetros de configuración del entorno *Environment2D*.

- Este planificador calcula una *policy* para todo el espacio conocido y alcanzable desde el estado inicial del robot.
- Para poder planificar, en vez de necesitar un punto inicial y un punto objetivo, este planificador recibe como entrada una función de recompensa y una función de terminación.

Planificador *Multi-Heuristic A (MHA*) [45]**

El rendimiento de los algoritmos heurísticos (como A*) se basa principalmente en la correcta elección de la función heurística dependiendo de la búsqueda que se desee realizar.

Normalmente se suele encontrar la función adecuada para adaptarse al tipo de problema, haciendo que los algoritmos de este tipo funcionen de forma rápida, generen soluciones de calidad e incluso tengan buen rendimiento en problemas con muchas dimensiones.

De todas formas, existen ocasiones en las que es complicado encontrar una función heurística que se adapte o que tenga en cuenta todas las complejidades que pueda tener el problema a resolver.

Por este motivo surgió el planificador MHA*, el cual utiliza un conjunto de funciones heurísticas arbitrarias juntamente con una función heurística consistente, para así utilizar todas ellas de forma simultánea y encontrar una solución subóptima completa y acotada.

De esta forma, se vuelve más sencillo crear estas funciones heurísticas que no tener que invertir más tiempo y esfuerzo en encontrar una que se adapte perfectamente al problema.

2.1.3 Archivos de configuración de entornos

Respecto a los entornos mencionados en 2.1.1, en nuestro caso nos centramos en el que se utiliza a lo largo del proyecto: *Environment2D*. Hay que emplear un archivo de configuración para poder utilizar los planificadores ya que estos contienes los parámetros necesarios del problema a resolver.

En la figura 2.7 se puede observar la forma que tiene este archivo.

Además, en la tabla 2.1 se observan los parámetros de configuración del entorno.

2.1.4 *MATLAB*

MATLAB (*MATrix LABoratory*) [46] [47] es una herramienta de software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M). *MATLAB* dispone de multitud de paquetes o *Toolbox* [48] para poder expandir sus prestaciones. Las características que destacan a *MATLAB* del resto de plataformas son la posibilidad de trabajar de forma muy eficiente con matrices, incluyendo su representación en memoria.

```
discretization(cells): 15 15
obsthresh: 1
start(cells): 0 0
end(cells): 8 0
environment:
0 0 0 0 0 0 1 1 0 0 0 0 1 1 1
0 0 0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 1 1 0 0 1 1 0 0 0 0 0 0 1
0 0 1 1 0 0 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 1 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 0
1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0 0 0 0 1 1 0 0 0 0 0 1 0 0 1
0 0 0 0 1 1 0 0 0 0 0 1 1 1 1
0 0 0 0 1 1 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
```

Figura 2.7: Ejemplo de archivo de configuración del entorno *Environment2D*.

En la librería *SBPL* vienen incluidos dos *scripts* con los cuales se puede realizar lo siguiente:

- Creación de entornos con el *script env_2Dcreator.m*. Se ha utilizado para poder crear los archivos de configuración para utilizar el entorno *Environment2D*. Con él se puede introducir una imagen previamente diseñada con algún programa de edición de imágenes, por ejemplo el que se ha utilizado en este caso ha sido GIMP [49] [50]. De esta forma, se puede convertir en un fichero con extensión *.cfg* y con la estructura mencionada en 2.1.3.

  En el pseudocódigo 1 se puede observar cómo funciona.

- Representación de trayectorias con el *script plot_3Dpath.m*. Con él se pueden realizar representaciones sencillas del resultado que se obtiene después de la utilización de alguno de sus planificadores.

  En la figura 2.8 se puede observar un ejemplo.

Algoritmo 1: Pseudocódigo *env_2Dcreator.m*

Input: Imagen con extensión *.jpg* o *.png* con *sizeX* y *sizeY* en píxels.
Output: Entorno con extensión *cfg* para poder introducirselo a un planificador de *SBPL*

```
1  function crearEntorno()
2      Close all, clear
3      env_nombre ← Definir nombre del entorno
4      start_x, start_y, end_x, end_y ← Definir posiciones inicial y final del problema.
5      inflar_obstáculos, radio_robot ← Definir si se desea inflar los obstáculos y
         definir el radio que tendrá el robot en tal caso.
6      Threshold ← Definir valor a partir del cual una celda se considera obstáculo.
7      img = escala_grises(img) //Convertir imagen a escala de grises.
8      if inflar_obstáculos == true then
9          Proceder a inflar los obstáculos un radio radio_robot.
10     env_sizeX, env_sizeY ← obtener tamaño del entorno a partir de img
11     Escribir cabecera del entorno //Depende del entorno que se esté tratando
         (2.1.3)
12     for i = 1 to filas do
13         for j = 1 to columnas do
14             if leer_Pixel(img, i, j) == blanco then
15                 Entorno(i, j) = 0;
16             else
17                 Entorno(i, j) = Threshold;
18     return Entorno //Se devuelve el archivo cfg creado
```

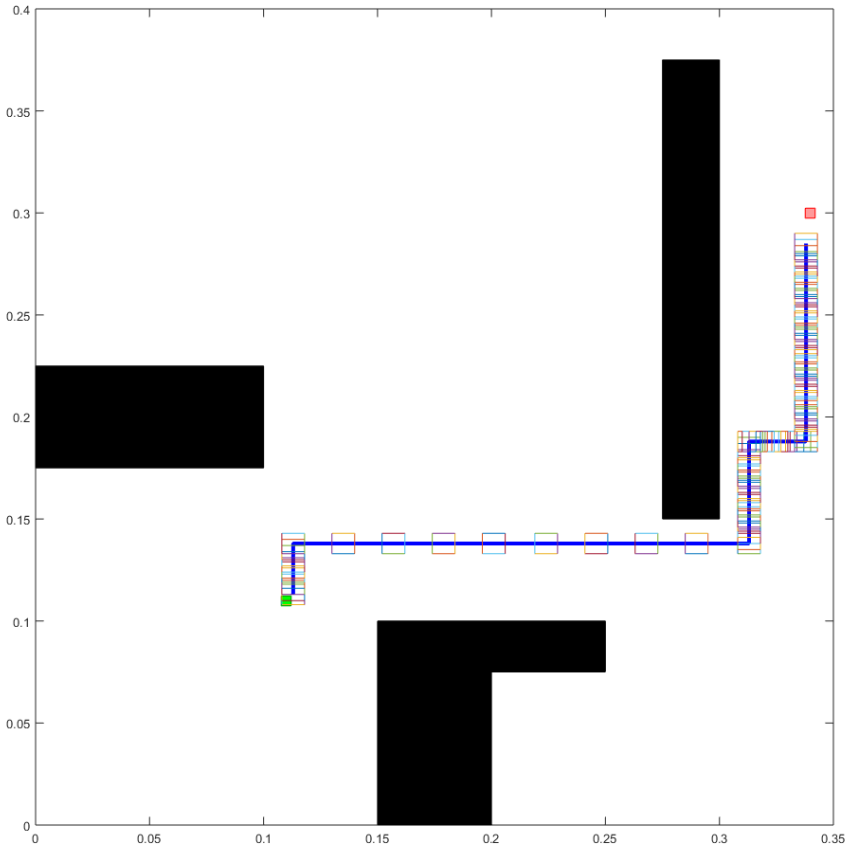


Figura 2.8: Ejemplo de representación de una trayectoria con el *script* incorporado en *MATLAB*.

CAPÍTULO 3

DESCRIPCIÓN ALGORITMOS *RRT*, *SFF* Y *TSP*.

A lo largo de este capítulo se describen los algoritmos que se han implementado:

- *RRT*. Necesario e imprescindible para comprender los algoritmos de planificación probabilísticos.
- *SFF*. Tal y como se ha comentado en 1.2, este es el algoritmo que se ha decidido estudiar e implementar.
- *TSP*. Después de aplicar *SFF* es necesario encontrar la ruta definitiva que el robot debe seguir. Tal y como se ha mencionado en 1.2, en la publicación de *SFF* se utiliza y por ello también se ha decidido implementar.

3.1 Algoritmo *RRT*

El algoritmo *RRT* (*Rapidly-Exploring Random Trees*) se introdujo en [16] y fue creado por Steven M. LaValle. Probablemente se trate del algoritmo de planificación probabilístico más conocido y del cual se han realizado más investigaciones.

Originalmente fue diseñado para hacer frente a problemas relacionados con robots con restricciones holónomicas y no holonómicas [51] [52] [53], para problemas de *kinodynamics planning* y para cubrir problemas con robots que tuviesen muchos grados de libertad.

Se ha demostrado que *RRT* es probabilísticamente completo, es decir, si existe un camino entre el punto inicial y el punto objetivo, el algoritmo lo acaba encontrando.

3.1.1 Construcción del árbol

El objetivo es crear un árbol desde un punto inicial (q_{start}) hasta un punto objetivo (q_{goal}). El árbol crece en cada iteración de forma aleatoria hasta que se alcance q_{goal}.

Inicialmente la raíz del árbol se sitúa en q_{start} y, en cada iteración, se genera un punto aleatorio (q_{rand}) en el espacio de configuración libre de obstáculos (Q_{free}). Éste punto q_{rand} se genera con cierta probabilidad de que coincida con el punto q_{goal}, de esta forma se puede modificar el hecho de que el árbol converja hacia q_{goal} más rápidamente.

Entonces, se busca el punto del árbol más cercano a este q_{rand}, el cual se denota como q_{near}. Posteriormente, se intenta avanzar sobre la línea que une q_{near} con q_{rand}, normalmente este desplazamiento lineal se realiza teniendo en cuenta una variable llamada *stepsize*.

Del desplazamiento que se haya podido realizar surge q_{new}, el cuál, si se puede unir a q_{near} sin que haya ningún obstáculo entre ellos, se añade al árbol formando así una nueva rama entre q_{near} y q_{new}.

En los algoritmos 2 y 3 se puede ver como se crea el árbol y como se extiende en cada iteración.

En la figura 3.1 se puede observar de forma visual cada una de las variables que se han mencionado y como se realiza una expansión del árbol.

Algoritmo 2: Creación árbol *RRT*

Input:

q_0	Configuración en la cual se sitúa la raíz del árbol
n	Número de intentos total para expandir el árbol

Output:

T	Árbol T = (Nodos, Ramas), cuya raíz se encuentra en q_0 y tiene ≤ n configuraciones

```
1 function crearRRT(q_0, n)
2     Nodos ← q_0
3     Ramas ← ∅
4     for i = 1 to n do
5         q_rand ← Se selecciona una configuración del espacio aleatoriamente
6         extenderRRT (T, q_rand);
7     return T
```

3.2 Algoritmo *SFF*

El algoritmo *SFF* (*Space Filling Forest Algorithm*) se publicó en el articulo [23] y fue creado por Vojtěch Vonásek y Robert Pěnička. En este capítulo se realiza una explicación teórica detallada del funcionamiento del algoritmo. Primeramente, se expone la motivación por la cual se vio la necesidad de crear un algoritmo de estas características y posteriormente, se presenta el objetivo que tiene el algoritmo y se explica en detalle su funcionamiento.

3.2.1 Motivación de la implementación del algoritmo

Se desea abordar el problema de la navegación multi-objetivo (*multi-goal*), es decir, el hecho de poder encontrar una ruta entre un conjunto de puntos objetivo.

Algoritmo 3: Extensión árbol RRT

Input:

T	Árbol RRT T = (Nodos, Ramas)
q	Configuración hacia la que debe crecer el árbol T

Output:

True	En caso de que se pueda haber extendido el árbol hacia q
False	En caso contrario

```
1 function extenderRRT(T, q)
2     q_near ← Vecino más cercano de q en T
3     q_new ← q que se obtiene al progresar sobre la línea recta que une q_near con
        q_rand en Q
4     if existeCaminoLibre(q_near, q_new) then
5         Nodos ← Nodos ∪ q_new
6         Ramas ← Ramas ∪ (q_near, q_new)
7         return True
8     return False
```

Creando las rutas entre los distintos puntos objetivo, este algoritmo quiere servir como preparación para poder encontrar el camino definitivo que se debe seguir.

El que típicamente más se utiliza y probablemente sea el más conocido es el algoritmo del Problema del Viajante (*Travelling Salesman Problem* [*TSP*]), el cual, se explica en detalle en el apartado 3.3.1. Se basa en el problema de encontrar el camino más corto posible que pasa por cada uno de los puntos objetivo y que finaliza en el punto inicial.

Tal y como se ha comentado en 1.1.1, *SFF* se quiere centrar en problemas con una gran cantidad de obstáculos y en los cuales, sobretodo, existan pasillos estrechos y de difícil acceso. En las figuras 3.2 y 3.3 se observan dos ejemplos de entornos en los cuales *SFF* desea mostrar su potencial.

Por ello, se desea crear un algoritmo el cual pueda resolver este problema y así servir de antesala a la utilización del algoritmo *TSP*, encontrando rutas entre los puntos que se desean visitar. Además, otro de sus objetivos es hacerlo de forma más eficiente y rápida que por ejemplo otros algoritmos de muestreo probabilístico tales como *RRT* y *PRM*.

3.2.2 Definición del problema y objetivo: encontrar la secuencia de puntos con *TSP*

Primeramente, es importante mencionar que nos centramos en un espacio de configuración de dos dimensiones (x e y).

Si denotamos como C al espacio de configuración del robot, $C_{free} \subseteq C$ es la zona libre de obstáculos dentro del espacio de configuración, es decir, donde el robot puede navegar.

Lo que se desea es poder encontrar la secuencia de puntos objetivo $C_i \in C_{free}, i = 1, \ldots, n$, consiguiendo realizar el mínimo recorrido posible.

Para poder encontrar esta secuencia de puntos, primeramente se debe conocer cada una de las distancias entre los puntos objetivo $\varrho(c_i, c_j)$. Posteriormente, se debe encontrar la correcta permutación entre los valores descritos por el vector que contiene los índices de los puntos objetivo $w = (\sigma_1, \ldots, \sigma_n)$.

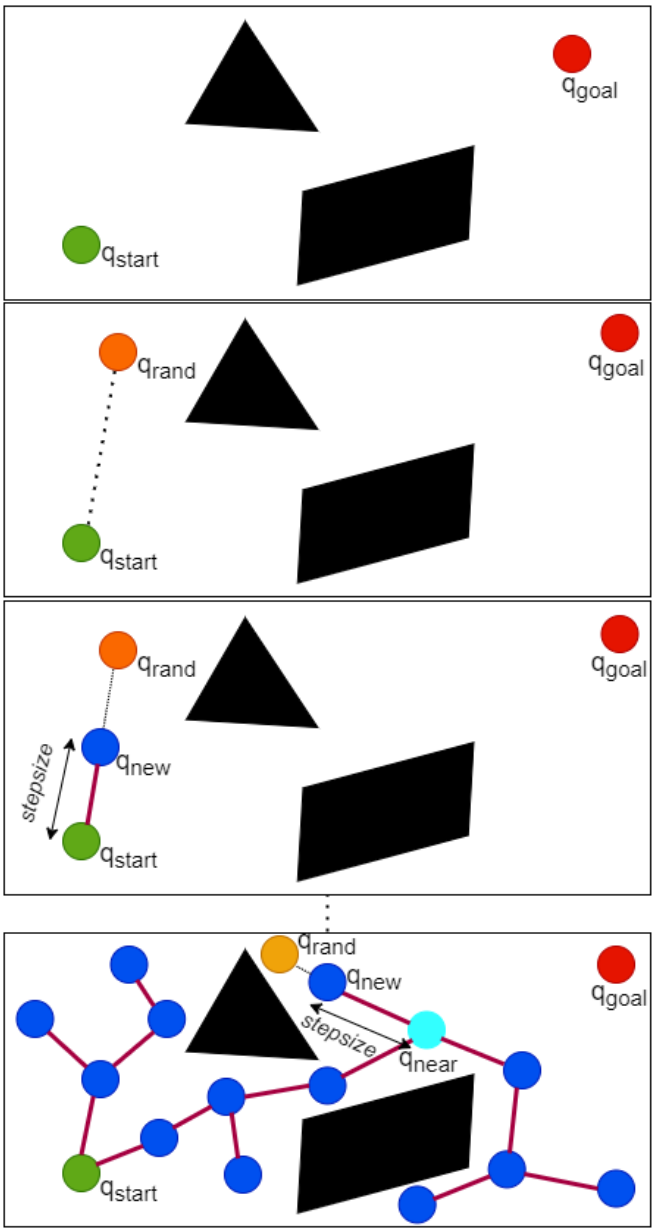


Figura 3.1: Variables *RRT* y expansión del árbol. Se puede observar de forma visual como *RRT* utiliza los puntos q_{rand} y q_{near}, además de la variable *stepsize*.

Por lo tanto, *TSP* debe calcular el path S, el cual se puede encontrar resolviendo el problema de optimización que consigue minimizar el vector w: $S = \sum_{i=2}^{n} \varrho(c_{\sigma_{i-1}}, c_{\sigma_i})$, donde el vector w se optimiza para minimizar el coste del camino entre los puntos objetivo.

3.2.3 Creación de los árboles

Antes de poder utilizar el algoritmo *TSP* para encontrar el camino que se debe seguir, se debe encontrar un conjunto de rutas que unan cada uno de los puntos objetivo. Sobre esas rutas es donde se puede aplicar el algoritmo *TSP*.

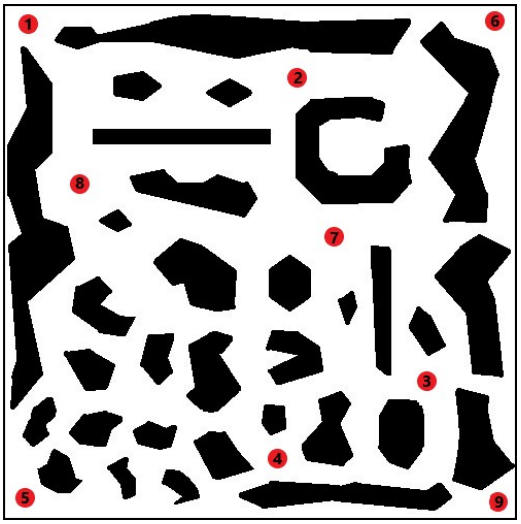


Figura 3.2: Ejemplo de entorno *SFF* con gran cantidad de obstáculos. En este ejemplo con 9 puntos a visitar se puede observar como hay una gran cantidad de obstáculos a evitar.

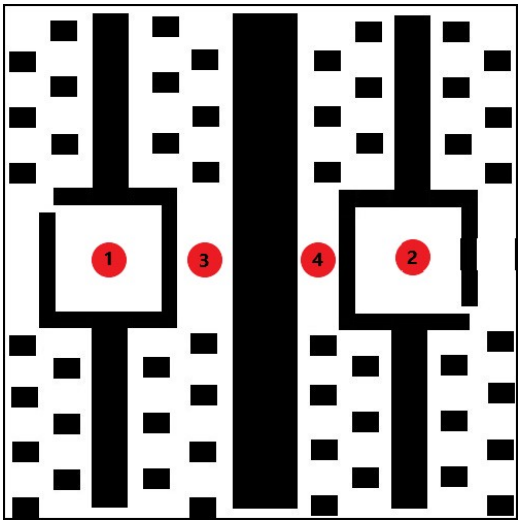


Figura 3.3: En este ejemplo con 4 puntos a visitar se puede observar como hay una gran cantidad de obstáculos y además, existen pasillos estrechos que dificultan el acceso a los puntos 1 y 2.

Para poder generarlas, *SFF* se centra en la principal idea de generar árboles arbitrarios $T_i, i = 1, ..., n$ desde cada uno de los puntos objetivo c_i. Estos árboles se expanden hasta que se encuentren con otros árboles, con los obstáculos que se encuentren en el entorno o con los límites de éste. Para poder expandirse, se crea un vector llamado *Open List* O, en el cual se van almacenando los nodos de los árboles que están disponibles para poder expandirse. Inicialmente, este vector se rellena con cada uno de los puntos objetivo.

Se puede observar el pseudocódigo en el algoritmo 4. En cada iteración, se intenta expandir uno de los árboles. Un nodo q se selecciona del vector *Open List* para intentar expandir el árbol T_i al que pertenece. El objetivo de la expansión es encontrar un nuevo punto q_c que se pueda añadir al árbol T_i. Por ello, $q_c \in C$ se genera de forma aleatoria alrededor de q una distancia d $(0 < d \leq R)$, donde R se define como la longitud de la expansión. El punto q_c se añade finalmente árbol T_i (y también al vector *Open List*, ya que será un nuevo punto desde el cual el árbol podrá expandirse) si se cumplen las siguientes condiciones:

1. El punto q_c no se encuentra lo suficientemente cerca de un nodo del mismo árbol: $\varrho(q_c, q_c') > \varrho(q, q_c)$, dónde q_c' es el nodo de T más cercano a q_c. De esta forma, se evita que el árbol se enquiste y le crezcan ramas hacia su interior. Así se garantiza que el árbol se expanda hacia zonas aún inexploradas.

2. La distancia de q_c hasta el resto de los árboles es mayor que un umbral d_{tree}. De esta forma se asegura que dos árboles no exploren una misma región de C_{free}.

3. El nodo q_c no colisiona con los obstáculos o con los límites del entorno.

Cada una de las expansiones que se realizan en cada iteración del algoritmo se intentan k veces, es decir, el nodo q seleccionado del vector *Open List* tiene k intentos

para expandirse. Si en esas k veces no se ha podido generar un punto q_c, el nodo q seleccionado se elimina del vector *Open List*.

Algoritmo 4: Algoritmo *SFF*

Input:

Vector de puntos c	Puntos objetivo $c_1, ..., c_n \in C_{free}$
d_{tree}	Distancia mínima entre los árboles
R	Tamaño de expansión de las ramas
k	Número de intentos para cada expansión

Output:

P_{ij}	Caminos entre los puntos objetivo

```
1  function algoritmoSFF(vector_c, d_tree, R, k)
2      T_i.añadirNodo(c_i), i = 1,...,n; // Se colocan los puntos objetivo en la raíz de
         los árboles
3      O.indexar((c_i, i)), i = 1,...,n; // Indexar Open List con el nodo y el índice de
         cada árbol
4      P_ij = ∅, ∀_i, j = 1,...,n; // Inicializar P_ij
5      while O no esté vacío do
6          q, i = seleccionar ítem aleatorio de O;
7          succ = false;
8          for trial = 1 : k do
9              q_c = configuración aleatoria alrededor de q la cual cumple
                 ϱ(q, q_c) ≤ R;
10             d_i, q' = T_i.vecinoMasCercano(q_c); // Mismo árbol
11             d_j, q_j = argmin_{j≠i} T_j.vecinoMasCercano(q_c); // Otros árboles
12             if d_j > d_tree then
13                 T_i.añadirNodo(q_c);
14                 T_i.añadirRama(q, q_c);
15                 O.añadirNodo((q_c, i));
16                 succ = true;
17                 break;
18             else
19                 if P_ij = ∅ and esPosibleConectar(q, q_j) then
20                     P_ij = T_i.ruta(c_i, q) ∪ T_j.ruta(q_j, c_j);
21         if not succ then
22             O.eliminarNodo((q, i));
23     return P_ij
```

Cuando dos nodos de dos árboles T_i y T_j se encuentran a una distancia menor que d_{tree} y entre ellos no hay ningún obstáculo, estos dos nodos se pueden conectar creando así una ruta entre ellos. Posteriormente a la unión de estos dos nodos, se reconstruyen los caminos hasta la raíz de cada uno de los árboles, generando así una nueva ruta para P_{ij}, la matriz que contiene los caminos entre los puntos objetivo. Este camino tiende a tener zigzags ya que se ha conseguido generar gracias a una construcción aleatorizada.

Por ello, será necesario utilizar algún tipo de algoritmo de suavizado para conseguir que los caminos que se generen no sean tan abruptos.

Finalmente, como resultado de la aplicación de *SFF*, se obtiene la mencionada matriz P_{ij} donde cada nodo es un punto objetivo y las conexiones entre cada uno de los nodos viene definida por las rutas encontradas.

En las figuras 3.4, 3.5, 3.6 y 3.7 se puede observar:

- Figura 3.4: selección de un nodo de forma aleatoria del vector *Open List*.
- Figura 3.5: distancia d_j, q_j hasta el nodo más cercano del resto de árboles.
- Figura 3.6: expansión que se realiza desde el nodo q, i. Se puede ver la distancia d_i, q' hasta el nodo más cercano del mismo árbol que se está expandiendo, además del radio R y el punto q_c generado de forma aleatoria dentro del radio.
- Figura 3.7: distancia d_{tree} a la que se quedan los árboles unos de otros.

cada uno de los parámetros mencionados del algoritmo *SFF*, además de como se realiza una expansión.

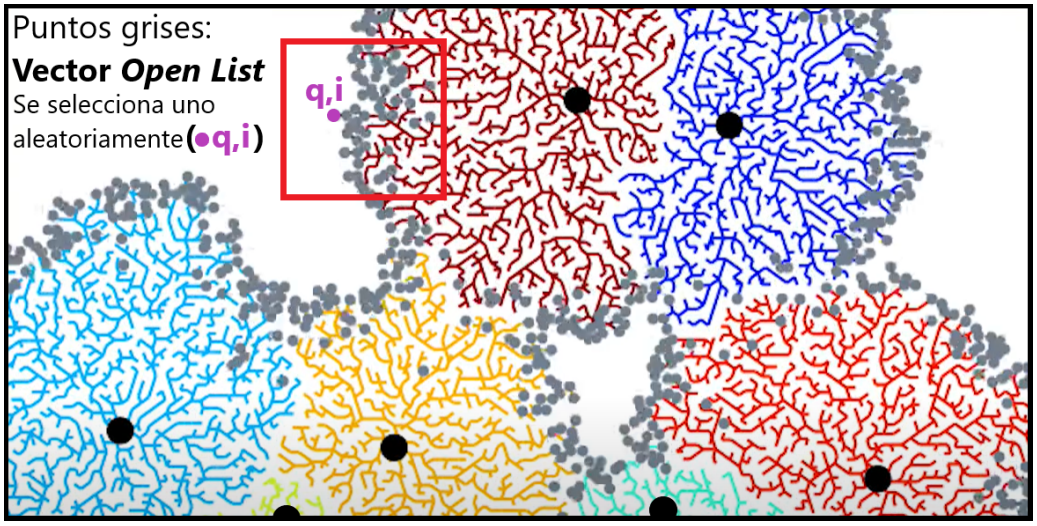


Figura 3.4: Vector *Open List* representado con los puntos grises. El punto magenta q, i representa el nodo escogido al azar del vector *Open List*.

3.3 Problema *TSP*

A continuación se describe el problema *TSP* en detalle, se menciona de forma breve cada uno de los tipos que pueden existir y finalmente se explica que librería se ha escogido para poder utilizar en el proyecto.

3.3.1 Descripción del problema

Karl Menger lo definió de forma matemática por primera vez en 1930. Es conocido por ser uno de los problemas de optimización más estudiados.

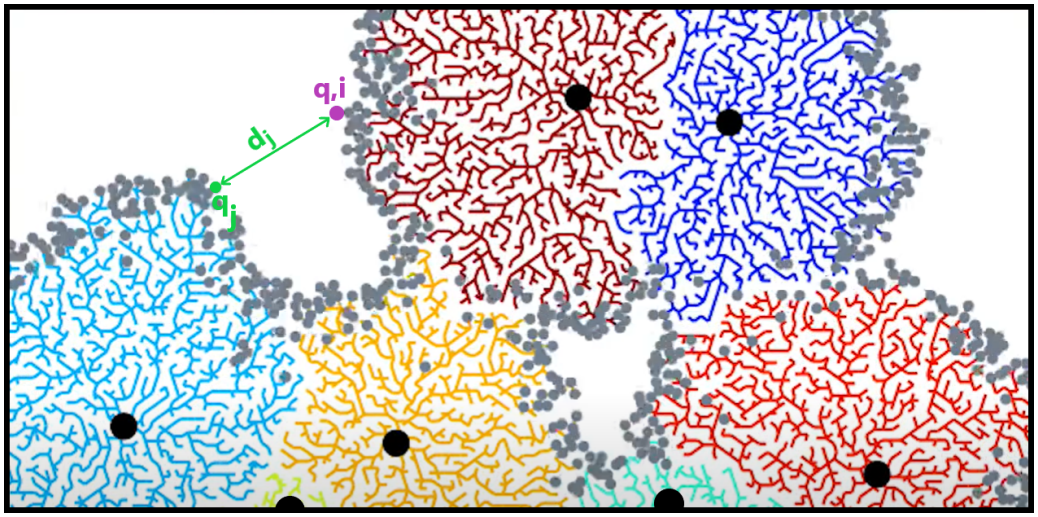


Figura 3.5: Distancia d_j que hay entre el punto q, i y q_j.

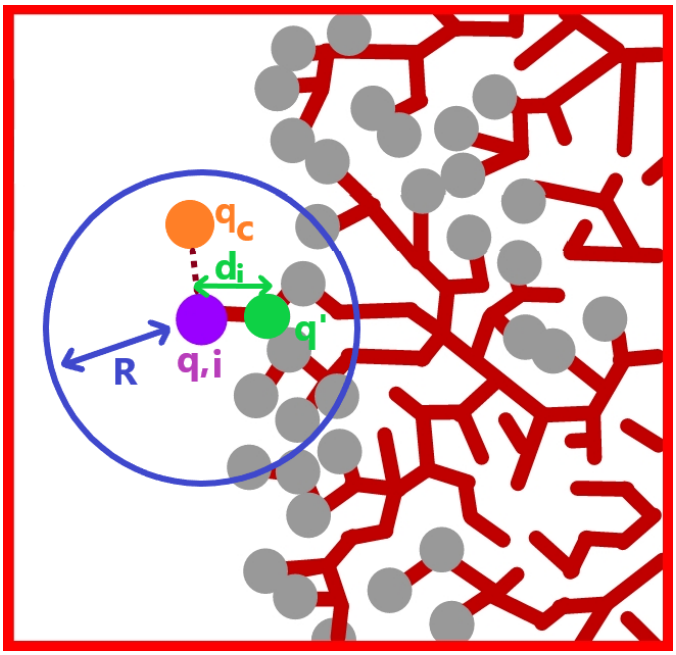


Figura 3.6: Expansión realizada desde el nodo q, i. Se puede ver representado el radio $0 < d \leq R$, el punto q_c generado aleatoriamente y la distancia d_i que existe hasta el nodo más cercano del mismo árbol.

Dados una serie de puntos finitos en el espacio, de los cuales se conocen las distancias entre cada par, el algoritmo *TSP* (*Travelling Salesman Problem*) [24] trata de resolver el problema de encontrar la ruta óptima que pase por cada uno de los puntos y que finalice en el punto inicial.

Por lo tanto, el problema se puede expresar tal que existan $\frac{(N-1)!}{2}$ rutas posibles. El factor "-1"se debe a que el punto de partida no es relevante y el factor "2" a que es indiferente en qué dirección nos desplacemos para cumplir el recorrido.

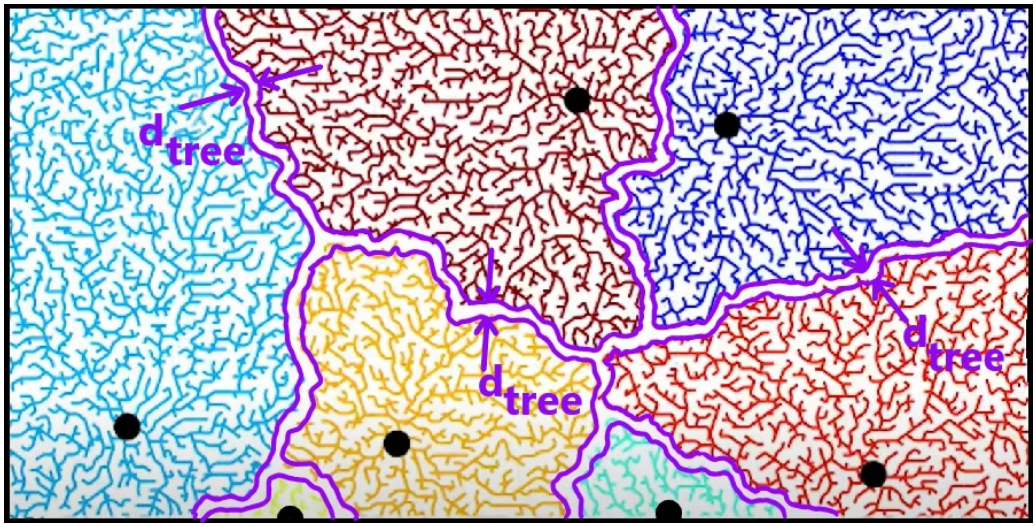


Figura 3.7: Distancia d_{tree} a la que se quedan los árboles unos de otros.

Puntos	Rutas	Tiempo
5	$\frac{(5-1)!}{2} = 12$	$12\mu s$
10	$\frac{(10-1)!}{2} = 181440$	$181ms$
50	$\frac{(50-1)!}{2} = 3{,}0414093201713 \cdot 10^{62}$	$\approx 9{,}64 \cdot 10^{48}$ años

Tabla 3.1: Tiempo de resolución algoritmo *TSP* en función del número de puntos. Se resuelve con una capacidad de cálculo de un millón de rutas por segundo

Se puede observar que a medida que se aumente el número de puntos, la cantidad de rutas posibles crece exponencialmente. Por ejemplo:

- Para 5 puntos: $\frac{(5-1)!}{2} = 12$ rutas diferentes.
- Para 10 puntos: $\frac{(10-1)!}{2} = 181440$ rutas diferentes.
- Para 50 puntos: $\frac{(50-1)!}{2} = 3{,}0414093201713 \cdot 10^{62}$ rutas diferentes.

Por lo tanto, si se tuviera un computador con una capacidad de procesamiento de un millón de rutas por segundo, el tiempo que se tardaría en resolver cada uno de los casos sería el que se observa en la tabla 3.1.

Vemos entonces, como es crucial intentar encontrar mecanismos de búsqueda que hagan que sea posible realizar el cálculo para números de nodos elevados. A este tipo de problemas se les llama NP-Completos [54], los cuales son muy importantes para los campos de búsqueda y ciencias de la computación.

La investigación que se ha realizado a lo largo del tiempo entorno a este algoritmo es muy importante, debido a que nos lo podemos encontrar en diversos campos de estudio y como hemos observado, su resolución en un tiempo aceptable es uno de los factores más importantes a tener en cuenta.

3.3.2 Resolución como un problema de grafos

Para poder resolver el problema *TSP*, éste se puede plantear como un problema de grafos y más concretamente, en un grafo ponderado no dirigido. Es decir:

- Los nodos del grafo son los puntos que se deben visitar.
- Los caminos que existen entre cada una de las ciudades son las aristas.
- La distancia de cada uno de los caminos representa el coste de viajar de un punto a otro.

De forma ideal y para la mayoría de problemas *TSP* se trabaja con un grafo completo, esto conlleva que cada par de puntos está conectado por una arista y por lo tanto, todos los puntos están conectados entre sí y se conoce el coste asociado a cada par de nodos.

En la figura 3.8 se observa un grafo de 7 nodos completo y simétrico con su correspondiente matriz. A continuación se explica el concepto de simetría.

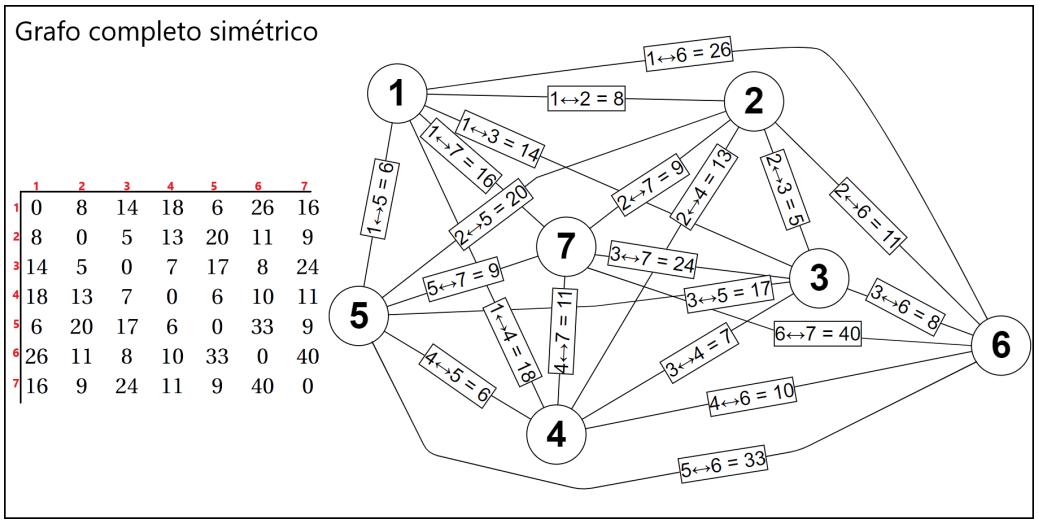


Figura 3.8: Representación de un grafo de 7 nodos completo y simétrico, junto a su matriz. Se puede observar como existe una conexión entre cada uno de los nodos.

Más adelante se observa que, debido a cómo se obtienen las aristas con el algoritmo *SFF*, se trabaja con grafos que no son completos. Por lo tanto, se puede dar el caso en el que existan nodos que se encuentren aislados o que solo estén conectados con un nodo.

Otra de las características más importante a tener en cuenta de los grafos es la simetría y la asimetría.

- *TSP* simétrico (grafos dirigidos). Se forma un grafo simétrico cuando la distancia entre dos puntos es la misma tanto si el camino se recorre en un sentido o en otro. Esto conlleva a que la matriz resultante sea simétrica y por lo tanto se pueda trabajar solo con la mitad, reduciendo así también a la mitad el número de soluciones posibles.

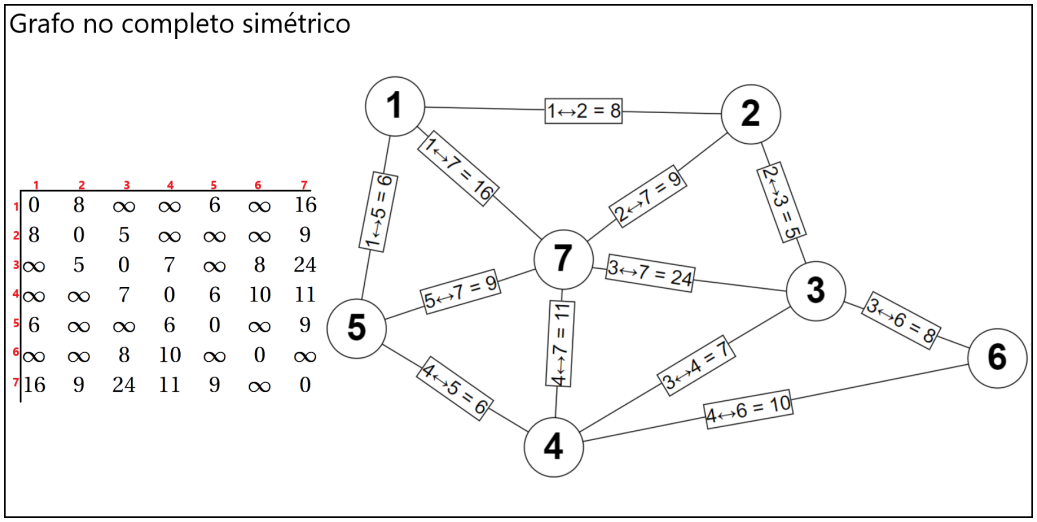
  Un ejemplo de este tipo se puede encontrar en la fabricación de circuitos impresos, los huecos de la PCB son los puntos objetivos que se deben perforar y el coste

entre cada uno de ellos es el tiempo que se tarda en reequipar al robot y en llegar al siguiente hueco. De esta forma, se puede encontrar la secuencia óptima para poder desarrollar estar tarea y así ahorrar tiempo y costes a la hora de fabricarlas.

- *TSP* asimétrico (grafos no dirigidos). Se forma un grafo asimétrico cuando la distancia entre dos puntos no es la misma dependiendo del sentido que en el que se realice el desplazamiento.

  Un ejemplo de este tipo se puede encontrar en la planificación que realiza una *app* de navegación para poder, por ejemplo, visitar una serie de lugares en una ciudad. Si por ejemplo el recorrido se desea realizar en coche, las calles que tienen dos sentidos puede que sufran una retención en uno de los dos carriles, por lo tanto el coste para ir de un punto a otro cambia en función del sentido que se escoja.

En las figuras 3.9 y 3.10 se pueden observar un ejemplo de estos dos tipos de grafos.



	1	2	3	4	5	6	7
1	0	8	∞	∞	6	∞	16
2	8	0	5	∞	∞	∞	9
3	∞	5	0	7	∞	8	24
4	∞	∞	7	0	6	10	11
5	6	∞	∞	6	0	∞	9
6	∞	∞	8	10	∞	0	∞
7	16	9	24	11	9	∞	0

Figura 3.9: Representación de un grafo de 7 nodos no completo y simétrico, junto a su matriz. Se puede observar como no tiene porqué existir una conexión entre cada uno de los nodos. Los nodos que no están conectados se representan como ∞.

3.3.3 Tipología de los algoritmos

Para poder resolver el problema *TSP* se suelen presentar tres posibles alternativas: los algoritmos exactos, los heurísticos y los basados en casos particulares de *TSP*.

1. Algoritmos exactos

Son aquellos que desean encontrar la solución exacta al problema *TSP*. Tal y como se ha mencionado en el apartado 3.3.1 con el ejemplo de la tabla 3.1, se entiende que el tiempo de computación para poder encontrar el resultado exacto es costoso, por ello este tipo de algoritmos funcionan correctamente para un número reducido de nodos.

Mencionándolos de forma breve, algunos de los más relevantes son:

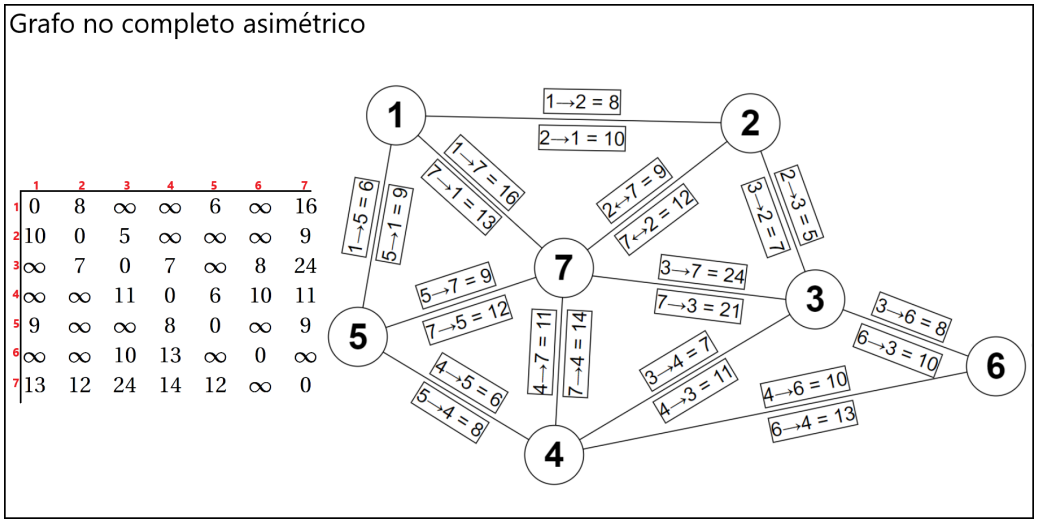


Figura 3.10: Representación de un grafo de 7 nodos no completo y asimétrico, junto a su matriz. Se puede observar como no existe una conexión entre cada uno de los nodos y, dependiendo del sentido en el que se recorran los nodos, el coste cambia.

- Fuerza bruta (*Brute force*) [55]. Algoritmo de orden $O(n!)$, el cual consiste en probar todas las posibles permutaciones [56] posibles.
- Programación dinámica (*Dynamic programming*) [57]. Algoritmos que tratan de resolver el problema dividiéndolo en fragmentos más sencillos y resolviéndolos recursivamente. Uno de los más importantes es el algoritmo de *Bellman–Held–Karp* [58], el cual es de orden $O(n^2 2^n)$.
- Ramificación y poda (*Branch and bound*) [59]. Se forma un árbol con distintos subconjuntos de posibles soluciones al problema, posteriormente se van explorando las ramas para encontrar la solución óptima.
- Programación lineal (*Linear programming*) [60]. Algoritmos que utilizan técnicas de programación lineal para poder resolver el problema.
- Ramificación y corte (*Branch and cut*) [61] [62]. Caso particular de Branch and Bound, se utilizan los denominados planos de corte para encontrar una solución óptima a través de la simplificación de la búsqueda.

2. Algoritmos heurísticos y aproximados

En este punto es donde se encuentra la mayor cantidad de algoritmos, ya que ofrecen soluciones muy buenas y aproximadas a la exacta en un tiempo de computación mucho menor a los algoritmos exactos. Este tipo de algoritmos funcionan mejor para un mayor número de puntos. Constan de dos fases diferenciadas: una primera de construcción de la ruta y una de mejora de ésta. Por ello, se pueden dividir en dos fases, una dónde se realizan las búsquedas heurísticas constructivas y una segunda donde se pretende mejorar el camino encontrado.

Algunos de los algoritmos más relevantes son:

- Heurísticas Constructivas (*Constructive heuristics*). **Construir la ruta inicial**.
  - Algoritmo del vecino más cercano (*Nearest neighbour (NN) algorithm*) [63]. Se trata de un algoritmo del tipo *greedy* [64] Pretende encontrar una ruta visitando cada vez el nodo más cercano al actual, consiguiendo así una solución inicial de forma rápida.
  - Algoritmo de Christofides [65] [66]. Este algoritmo combina las búsquedas del *Minimum Spanning Tree [MST]* [67] de un grafo G juntamente con la búsqueda del *minimum cost perfect matching* [68] del subgrafo c extraído de G.
  - *(Match Twice and Stitch*) [69]. Se basa en realizar emparejamientos entre los nodos de forma cíclica. Al final de los emparejamientos, estos se unen para formar un recorrido.
- *Tour improvement*. **Mejorar la ruta inicialmente construida**.
  - *K-opt* o *Lin-Kernighan* [70]. Se cambia de forma iterativa un cierto número de aristas de la solución dada para encontrar una solución con un coste inferior. 2-opt [71] y *3-opt* [72] son dos de los casos específicos más conocidos de *K-opt*.
  - Búsqueda Tabú [73]. Busca la mejor solución posible en la vecindad de la solución ya existente. Mantiene una lista tabú en la cual se marcan diversas soluciones como incorrectas para así no volver a tenerlas en cuenta.
  - Algoritmos genéticos [74]. Se basan en el proceso natural de selección y utilizan la mutación, el cruce y la selección como herramientas para poder encontrar una ruta e ir mejorándola a medida que el algoritmo progresa.
  - *Simulated annealing* [75]. Pretende buscar una aproximación al valor óptimo de una función dada en un espacio de búsqueda relativamente grande.
  - Colonia de hormigas [76]. Inspirado en una colonia de hormigas real, este algoritmo utiliza grafos y principalmente se basa en las feromonas que las hormigas depositan a medida que exploran el entorno.

3. Algoritmos basados en subproblemas de *TSP*

En este grupo encontramos los que tratan de encontrar casos especiales del algoritmo *TSP*, para así poder aprovechar ciertas características de TSP y de esta forma crear algoritmos adaptados a éstas.

- Desigualdad triangular (Δ-TSP) [77]. Son los que se basan en la desigualdad triangular $d_{AB} \leq d_{AC} + d_{CB}$. Algunos ejemplos de métricas que utilizan esta desigualdad son:
  - *TSP* euclidiano. La distancia entre dos ciudades es la distancia euclidiana entre los puntos correspondientes.
  - *TSP* rectilíneo. Utiliza la distancia de Manhattan, para la cual la distancia entre dos ciudades es la suma de los valores absolutos de las diferencias de sus coordenadas x e y.

- *TSP* asimétrico. En este caso se intenta tratar la asimetría explicada en 3.3.2 para así poder transformar el problema a simétrico y que sea más sencillo de resolver.

3.3.4 Librería utilizada

Una vez se consigue la matriz con cada uno de los caminos descubiertos por *SFF*, es momento de resolver el problema *TSP* aplicando alguna de las técnicas vistas en el apartado anterior.

Para ello, se ha tenido que realizar una búsqueda para encontrar una implementación que funcione correctamente para un número de nodos relativamente pequeño. Es interesante que esta librería tenga la posibilidad de ofrecer una solución exacta y otra que se le acerque bastante, y además, que esta segunda la consiga en un tiempo razonable.

El factor del tiempo de ejecución es importante ya que en nuestro caso, al implementar un algoritmo de navegación y posteriormente aplicar *TSP*, es posible que se quiera utilizar el trabajo realizado como parte de algún otro proyecto. Por ejemplo, para algún algoritmo de navegación que se base en una arquitectura híbrida.

Entonces, tal y como se ha comentado, el tiempo de ejecución influye ya que si se aplica sobre un robot en movimiento, seguramente sea necesario que se cumplan unos requisitos respecto a dicho tiempo.

Se ha encontrado un proyecto [78] de la escuela de ingenieros (ISEN-Nantes) [79] el objetivo del cual era estudiar e implementar ciertos algoritmos para poder resolver el problema de *TSP*.

En particular, se deseaba implementar un algoritmo exacto, un *constructive heuristic*, un *local search heuristic* y un algoritmo tipo *greedy*, en particular un algoritmo llamado *GRASP* (*Greedy Randomized Adaptive Search Procedures*).

Backtracking (Algoritmo exacto)

Este algoritmo exacto, el cual, tal y como se ha comentado en 3.3.3, es de orden $O(n!)$. Tiene una modificación para que no se trate directamente de un algoritmo de fuerza bruta.

Utiliza recursividad para así poder descartar las peores soluciones antes de calcularlas. De esta forma, el peor escenario posible es en el que la última solución que se calcule sea la exacta. Es poco probable que se reproduzca pero por ese motivo la complejidad es del orden $O(n!)$.

Constructive heuristic

Para este tipo algoritmo se decidió implementar un algoritmo *NN*. El punto de partida es un vértice aleatorio del grafo, a partir del cual, en un primer bucle del algoritmo, se van descubriendo todos los vértices que lo componen.

Dentro de este bucle se realizan movimientos hacia los vértices más cercanos que aún no se han descubierto. Este algoritmo, tal y como se ha mencionado en 3.3.3, es de orden $O(n^2)$.

En líneas generales tiende a tener un bajo coste computacional ya que se ejecuta de forma veloz y en contraposición, de media, la calidad del camino que se encuentra es un 25 % peor que el camino óptimo. El mejor caso se encuentra cuando, empezando desde el primer vértice elegido al azar, todos los vértices que se encuentran posteriormente tienen el coste más pequeño. Por ello, el peor caso se haya cuando se van encontrado vértices

con costes pequeños pero el último vértice a analizar tiene un coste muy elevado, lo que hace que la calidad de la solución decrezca considerablemente.

Local search heuristic

En este caso se decidió implementar el algoritmo 2-opt[ref], uno de los más sencillos de los algoritmos *local search*.

El objetivo principal del algoritmo es el de encontrar rutas las cuales tengan algún cruzamiento, para así, posteriormente, reordenarlo y así conseguir un camino más corto.

En la figura 3.11 se puede observar un ejemplo de cruzamiento con su correspondiente ordenación.

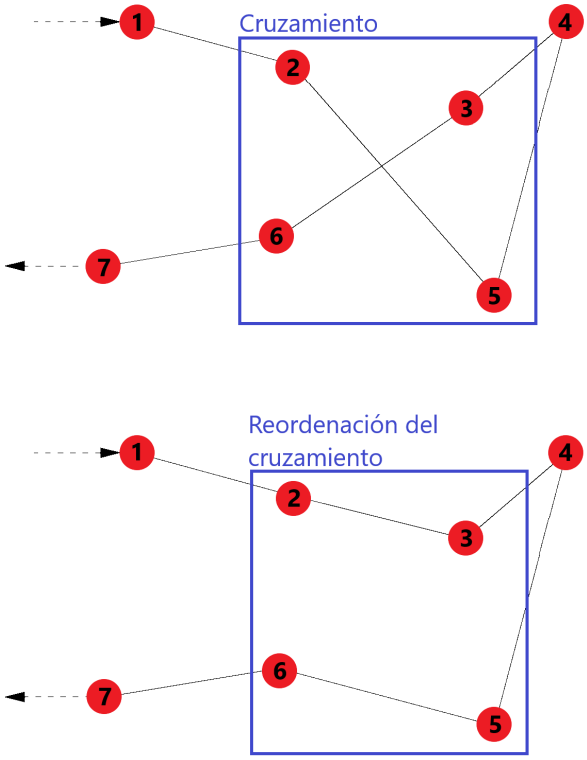


Figura 3.11: Cruzamiento de un camino y su correspondiente ordenación gracias al algoritmo 2-opt.

La complejidad es O(n^2) debido a que se tienen dos factores: uno O(n^2) por considerar todos los pares de ejes y otro O(n) por invertir el camino. Por lo tanto, se tiene una complejidad $O(n^2 + n) = O(n^2)$. En contraposición con el algoritmo *NN constructive heuristic*, hay que tener en cuentra que 2-opt es un algoritmo *local search*, lo que puede conllevar a que el camino que se devuelva sea un mínimo local. El mejor caso posible es en el que el mínimo local es el mínimo global, esto conlleva a que el tiempo de ejecución sea bastante variable y ligeramente superior a *NN*.

De todas formas, utilizando este algoritmo se obtienen mejores caminos que si se utiliza *NN*.

Greedy Randomized Adaptive Search Procedures (*GRASP*)

GRASP (*Greedy Randomized Adaptive Search Procedures*) es un algoritmo metaheurístico definido por Feo y Resende el cual realiza búsquedas adaptativas aleatorizadas.

Cada iteración del algoritmo es una búsqueda y en cada una de ellas se obtiene una solución al problema *TSP*. Cada iteración se compone de dos partes bien diferenciadas.

Primeramente, se genera una solución inicial gracias a una función adaptativa randomizada y después, en la segunda fase, se realiza una búsqueda local para poder mejorar la solución inicialmente encontrada.

- En la primera fase, en cada una de las iteraciones que se realizan para poder construir la solución inicial, se van añadiendo elementos gracias a una función randomizada la cual elige una solución (no necesariamente la mejor) aleatoria de una lista llamada RCL (*Restricted Candidate List*).

  Esta lista RCL está ordenada según el criterio que se decida seguir, se puede decir que se ordena en función del beneficio estimado que puede producir el añadir cada uno de sus elementos a la solución. Este beneficio o forma de ordenar la lista es el hecho por el cual el algoritmo es adaptativo, ya que en cada iteración de esta primera fase se actualizan los valores de los candidatos.

  Por otro lado, es aleatorio ya que como se ha mencionado, no necesariamente se elige el mejor candidato en cada iteración, si no que se elige de forma aleatoria dependiendo de la función que se utilice.

- En la segunda fase se utiliza un algoritmo de búsqueda local para poder mejorar las soluciones obtenidas en la primera fase. Estos algoritmos trabajan de forma iterativa y suelen reemplazar la solución que tengan en ese momento por una mejor hasta que se obtenga un mínimo local lo suficientemente bueno como para alcanzar el criterio de parada.

En relación al tiempo de ejecución, comparándolo con los otros dos algoritmos (*constructive heuristic* y *local-search heuristic*), debido al componente aleatorio que éste presenta, es complicado establecer el orden al cual responde, ya que los tiempos de ejecución pueden ser variables.

Las figuras 3.12 y 3.13, extraídas de la solución que se obtuvo en [78], representan de forma sencilla, en función de las pruebas que se realizaron, el tiempo de ejecución y la calidad de la solución obtenida de cada uno de los algoritmos respecto al número de nodos del problema *TSP*.

De cada uno de los gráficos se ha eliminado el algoritmo exacto, debido a que el tiempo de ejecución en comparación con los demás algoritmos es mucho mayor y la solución obtenida es siempre la mejor, por lo que no tiene sentido compararlo con los demás.

Para el caso del tiempo de ejecución, se observa como *GRASP* es el peor. Se toma como referencia para realizar el gráfico y así ver como *constructive heuristic* y *local search heuristic* son prácticamente idénticos. A grandes rasgos, se puede observar como son un 90 % más rápidos que *GRASP*.

Por otro lado, es importante tener en cuenta la calidad del camino. Se puede observar como *GRASP* es el que mejores soluciones obtiene respecto a los otros dos algoritmos, por una ligera diferencia respecto a *local search heuristic*.

Por lo tanto, aunque los algoritmos tengan el mismo orden de complejidad O(n^2), tanto el tiempo de ejecución como la calidad de la solución encontrada varían en función de cual se utilice.

Cabe mencionar que estos resultados pueden variar considerablemente en función de cómo se implemente cada uno de estos algoritmos. En nuestro caso, se ha utilizado esta librería ya que es sencilla de comprender y de implementar, por ello se considera como una herramienta correcta para poder utilizar *TSP* durante el proyecto.

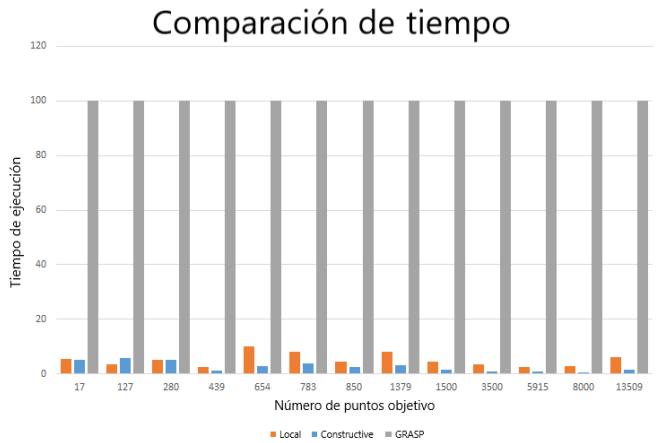


Figura 3.12: Comparación del tiempo de ejecución entre los algoritmos que solucionan *TSP*.

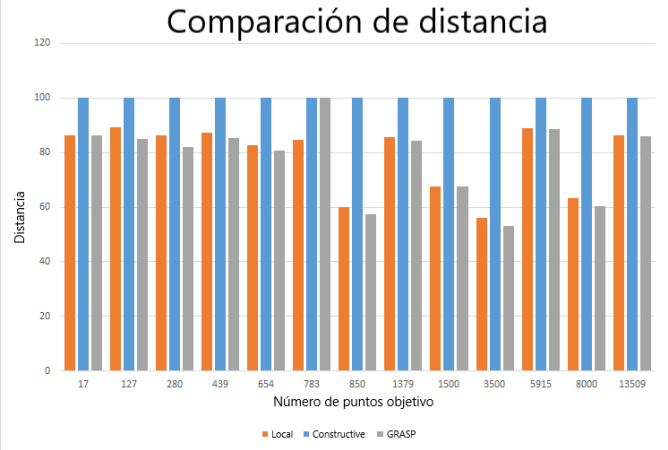


Figura 3.13: Comparación de la calidad del camino obtenido por cada uno de los algoritmos que solucionan *TSP*.

CAPÍTULO 4

ESTRUCTURA DEL SISTEMA, IMPLEMENTACIÓN DE *SFF*, *RRT-MO* E INCORPORACIÓN DE *TSP*

En este capítulo se explica la estructura general del sistema, cómo se organiza la librería *SBPL* y los diversos componentes que se han añadido para el correcto funcionamiento del conjunto. También se aborda la implementación de *SFF* con la correspondiente integración de *TSP*.

Además, también se explica cómo se ha llevado a cabo la adaptación del algoritmo *RRT* original al problema del viajante para así crear una versión multi-objetivo. Es decir, una variante del algoritmo *RRT* que crea árboles de forma secuencial visitando cada uno de los nodos que componen el problema a resolver. Se ha decidido denominar a este algoritmo como *RRT-MO*.

Finalmente, cabe decir que a medida que se avance, se comentan las herramientas complementarias que se han utilizado así como diversos problemas obtenidos durante esta fase del proyecto.

4.1 Estructura general del sistema

Para llevar a cabo la explicación de cómo se ha implementado *SFF*, *RRT-MO* y de cómo se ha incorporado *TSP*, se ha optado por primero dar una visión global de cada uno de los componentes del sistema, para así poder avanzar cómo se ha organizado el programa implementado. En la figura 4.1 se puede observar la estructura general con cada uno de sus componentes. A continuación, durante los siguientes subapartados, se comentan los aspectos más destacables de cada uno de los distintos bloques.

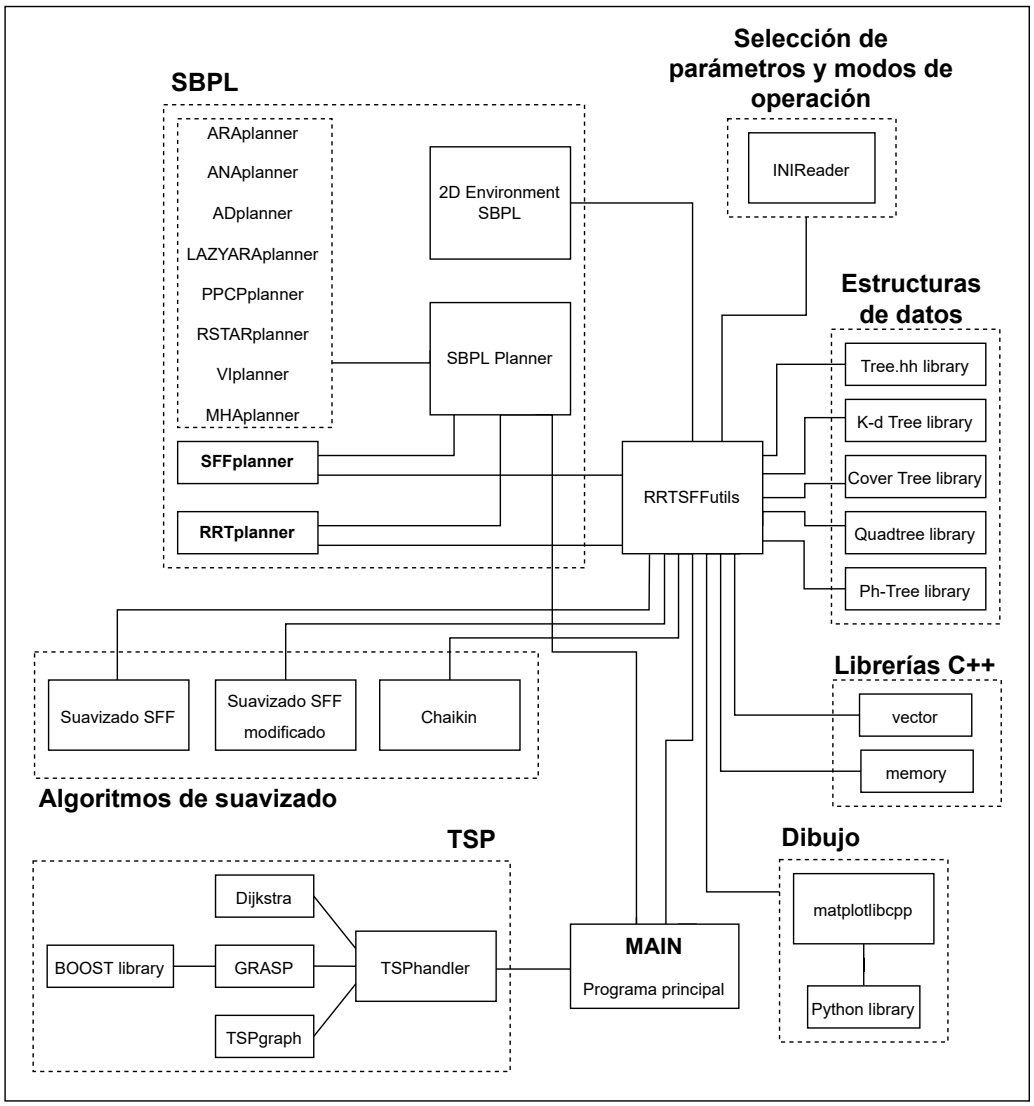


Figura 4.1: Diagrama de la estructura general del sistema con cada uno de sus componentes.

4.1.1 Planificadores *SFF* y *RRT-MO* incluidos en la librería *SBPL*

Los dos algoritmos que se desean implementar, explicados previamente en 3.1 y 3.2, se han incorporado a los planificadores ya existentes de *SBPL*, de esta forma se pueden utilizar sencillamente tal y como se haría con uno de los planificadores originales de la librería.

Para poder incluirlos, se han modificado los archivos *headers.h* y *CMakeLists.txt*. En el archivo general que contiene las cabeceras (*headers.h*) de todos los planificadores, se han incluido los archivos *SFFplanner.h* y *RRTplanner.h* para que se puedan utilizar en *SBPL*. Respecto al archivo *CMakeLists.txt*, es necesario añadir los archivos que contienen el código fuente de los planificadores, por ello, se han incorporado los archivos *SFFplanner.cpp* y *RRTplanner.cpp* para que a la hora de compilar la librería, ésta pueda tener en cuenta los nuevos planificadores.

En los apartados 4.3.1, 4.3.3, 4.5.1 y 4.5.3 se explica la estructura de los dos planificadores así como cada una de las variables y funciones más relevantes.

4.1.2 Librería común y auxiliar para los planificadores *SFF* y *RRT-MO*

Para poder gestionar todo el volumen de información necesario a la hora de implementar los algoritmos, se ha decidido crear una librería llamada *RRTSFFutils.h* para así dar soporte a la hora de realizar diversos cálculos y permitir el uso de distintas características tales como:

- Utilización de estructuras de datos en forma de árbol.
- Utilización de la librería de gestión de archivos con extensión *.ini* [80].
- Utilización de librería de dibujo *matplotlibcpp* [81], basada en *Python*.
- Puente para la utilización de *TSP*.

De esta forma, debido a que gran parte de los cálculos que se deben realizar en cada uno de los planificadores son comunes, queda definido de una forma más compacta y sencilla de entender. En el apartado 4.6 se explica en detalle cada una de las variables y funciones que la componen.

4.1.3 Estructuras de datos en forma de árbol

Durante la realización del proyecto se ha necesitado del uso de algún tipo de estructura de datos que permitiera trabajar con árboles. Por ello, tal y como se explica en el apartado 4.2, se ha utilizado una librería genérica llamada *tree.hh* y diversas librerías para poder trabajar con árboles binarios [82].

4.1.4 Gestión de parámetros y selección de modos de uso

Para poder gestionar de forma sencilla los parámetros de cada uno de los algoritmos así como diversas opciones de configuración y selecciones de modo de uso que se han creado, se ha optado por utilizar un archivo de tipo *.ini* el cual se ha llamado *params_RRTSFF.ini*. De esta forma, modificando dicho archivo, asignando los valores deseados a cada uno de los parámetros, se puede llevar a cabo un test o una ejecución determinada.

Este bloque se comunica con la librería *RRTSFFutils.h*, que a su vez gestiona los valores para que los algoritmos *SFF* y *RRT-MO* los puedan utilizar.

En el apéndice B se explica cada una de las partes que componen el archivo *params_RRTSFF.ini*.

4.1.5 Bloque *TSP*

En este apartado se explican los distintos subbloques que gestionan el funcionamiento de *TSP*. Se ha creado una clase llamada *TSPhandler*, la cual permite utilizar *TSP* desde los algoritmos *SFF*, *RRT-MO* y desde el programa principal o *main.cpp*.

TSPhandler se encuentra conectada con los siguientes bloques:

- ***Dijkstra***. Bloque contenedor del algoritmo *Dijkstra* debido al problema que se explica en el apartado 4.4.2, por el cual es necesario crear más conexiones entre nodos para que así *TSP* pueda encontrar una solución adecuada.
- ***TSPgraph***. Librería para poder crear y gestionar los grafos que utiliza el algoritmo *TSP*. Se encuentra conectada con *TSPhandler* y con *GRASP*.
- ***GRASP***. El algoritmo mencionado en 3.3.4, el cual se encarga de encontrar una solución al problema *TSP*. El motivo por el cual se utiliza este algoritmo se encuentra expuesto en el apartado 5.2, en él se realiza una comparación entre los algoritmos *Backtracking*, *Constructive heuristic*, *Local search heuristic* y *GRASP*.
- Librerías ***Boost***. Conjunto de librerías desarrolladas en C++ las cuales dan soporte para realizar diversas operaciones tales como:
  - Cálculo de álgebra lineal.
  - Generación de números pseudoaleatorios.
  - Trabajos multitarea o *multithreading*.
  - Procesamiento de imágenes.
  - Expresiones regulares.

  entre otras muchas aplicaciones. Para el correcto funcionamiento de *GRASP*, es necesario tenerla instalada en la máquina encargada de ejecutar los algoritmos *SFF* y *RRT-MO*.

En los apartados 4.4.3 y 4.4.4 se explican en detalle las variables y funciones que componen la librería *TSPhandler*. Se presenta un diagrama de flujo para que se pueda comprender la problemática encontrada acerca del algoritmo *SFF* y cómo se intenta solventar gracias a la utilización del algoritmo *Dijkstra* y *TSP*.

4.1.6 Bloque de dibujo

La herramienta de *MATLAB* que *SBPL* incluye para poder realizar representaciones solo contempla el dibujo que se puede realizar entre un punto inicial y un punto objetivo. La representación que se necesita en este caso es más compleja ya que contiene diversas características a tratar tales como:

- Necesidad de dibujar diversos puntos objetivo.
- Dibujar el crecimiento de cada uno de los árboles que avanzan desde los diversos puntos objetivo.
- Dibujar los caminos que *SFF* o *RRT-MO* han encontrado entre los diversos puntos objetivo.
- Dibujar el camino que finalmente se debe seguir para poder recorrer todos los puntos objetivos gracias al cálculo realizado por *TSP*.

Por ello, se ha realizado una búsqueda entre diversas librerías de dibujo desarolladas en C++ y se ha optado por utilizar *matplotlibcpp* [81]. Se trata de una interfaz de la librería original *matplotlib* [83] desarrollada en *Python*. Su principal característica es la sencillez y la gran documentación que existe, lo que la convierte en una librería simple de utilizar.

En el apartado 4.8 se mencionan las funciones más importantes que se han utilizado.

4.1.7 Librerías de C++ destacadas: *Vector* y *memory*

Dos de las librerías de C++ más importantes a lo largo de la realización del proyecto han sido *vector* [84] y *memory* [85]. Con estas dos librerías se ha pretendido simplificar ciertos procedimientos del código para que éste sea más sencillo de implementar.

- Librería ***vector***. Gracias a ella se ha podido trabajar continuamente con vectores y así simplificar de forma notable la creación de árboles. Gracias a diversas funciones tales como *vector::push_back*, *vector::insert*, *vector::emplace*, *vector::clear*, *vector::size*, *vector::begin* o *vector::end* el manejo de árboles y por lo tanto el uso de la librería *tree.hh* ha sido posible.
- Librería ***memory***. Gracias a ella se han podido evitar problemas de memoria, más conocido como *memory leak* [86]. Debido al uso continuo de punteros, utilizar esta librería ha sido de gran utilidad ya que de forma automática se realizan las operaciones necesarias para que dichos punteros dejen libres las posiciones de memoria que ocupan para que así el sistema operativo pueda asignarlas a nuevos recursos. Si no se hubiera utilizado, se tendría que haber prestado atención a cada puntero y liberarlo cada vez que se acaba de utilizar.

En el apartado 4.9 se mencionan las funciones más relevantes que se han utilizado a lo largo de la implementación.

4.1.8 Bloque *main*: programa principal

El programa principal o *main* es desde donde se declaran cada uno de los planificadores para así poder utilizar sus respectivas funciones y obtener un resultado al problema que se plantee. En este caso, éste archivo prácticamente no se ha modificado y simplemente se han añadido las líneas de código necesarias para llevar a cabo un ejemplo sencillo de como se utiliza el planificador *SFF* junto al algoritmo *TSP* y como se utiliza el planificador *RRT-MO*.

De esta forma, si en un futuro se desea implementar algún tipo de programa o aplicación el cual utilice estos planificadores, se puede acceder a un ejemplo de uso.

4.2 Estructuras de datos utilizadas

A continuación se lleva a cabo una breve explicación de cada uno de los árboles binarios contemplados para poder resolver el problema del tiempo invertido en las búsquedas del vecino más cercano, *NN* (*Nearest neighbor search*) de ahora en adelante. De esta forma se puede comprender cada uno para así posteriormente valorar cuál es una mejor opción a utilizar.

4.2.1 Librería *tree.hh* y problema de rendimiento sobre la búsqueda *NN*

Antes de abordar la explicación de los planificadores *SFF*, *RRT-MO* y de la incorporación del algoritmo *TSP*, se ha creído conveniente explicar la problemática que se tuvo en la primera fase de implementación del algoritmo *SFF* y el motivo por el cual se decidió acudir al uso de los árboles binarios.

Primeramente, cabe decir que, tal y como se ha mencionado anteriormente sobre las estructuras de datos utilizadas, para poder gestionar la creación de árboles se ha decidido utilizar una librería llamada *tree.hh*.

Se trata de una librería de tipo STL (*Standard Template Library*) [87] desarrollada en C++, la cual contiene todas las herramientas necesarias para poder crear y mantener árboles poblados por nodos con un número desconocido de hijos. Los nodos pueden contener información de diversos tipos y en este caso, contienen información acerca de las coordenadas de dónde se encuentra cada uno.

En la figura 4.2 se observa un árbol representando una estructura que se puede crear con la librería.

Durante la realización del proyecto, se observó que no era suficiente la utilización de la librería *tree.hh* para poder implementar de forma correcta el algoritmo *SFF*. Resulta que no posee un mecanismo de búsqueda *NN* eficiente, únicamente provee los mecanismos necesarios para crear y añadir nodos a la par que recorrer los árboles de diversas formas tales como:

- *pre_order_iterator*. Se recorre el árbol de forma completa visitando cada uno de sus nodos de forma secuencial. La secuencia correspondiente a la figura 4.2 es 0 →1 →4 →5 →2 →3 →6 →7 →9 →8.

- *post_order_iterator*. Primero se recorre ordenadamente los hijos de cada uno de los nodos más profundos, para posteriormente ir avanzando hasta la raíz del árbol. La secuencia correspondiente a la figura 4.2 es 4 →5 →1 →2 →6 →9 →7 →8 →3 →0.

- *breadth_first_iterator*. Se recorren los nodos padre para continuar con los hijos. La secuencia correspondiente a la figura 4.2 es 0 →1 →2 →3 →4 →5 →6 →7 →8 →9.

- *sibling_iterator*. Únicamente se recorren los hermanos de un nodo. Respecto a la figura 4.2, la secuencia de los *siblings* del nodo 1 es 4 →5, mientras que del nodo 3 es 6 →7 →8.

- *fixed_depth_iterator*. Únicamente se recorren los nodos debajo del nivel del nodo seleccionado. Respecto a la figura 4.2, si se selecciona el nodo 1, la secuencia es 4 →5 →6 →7 →8.

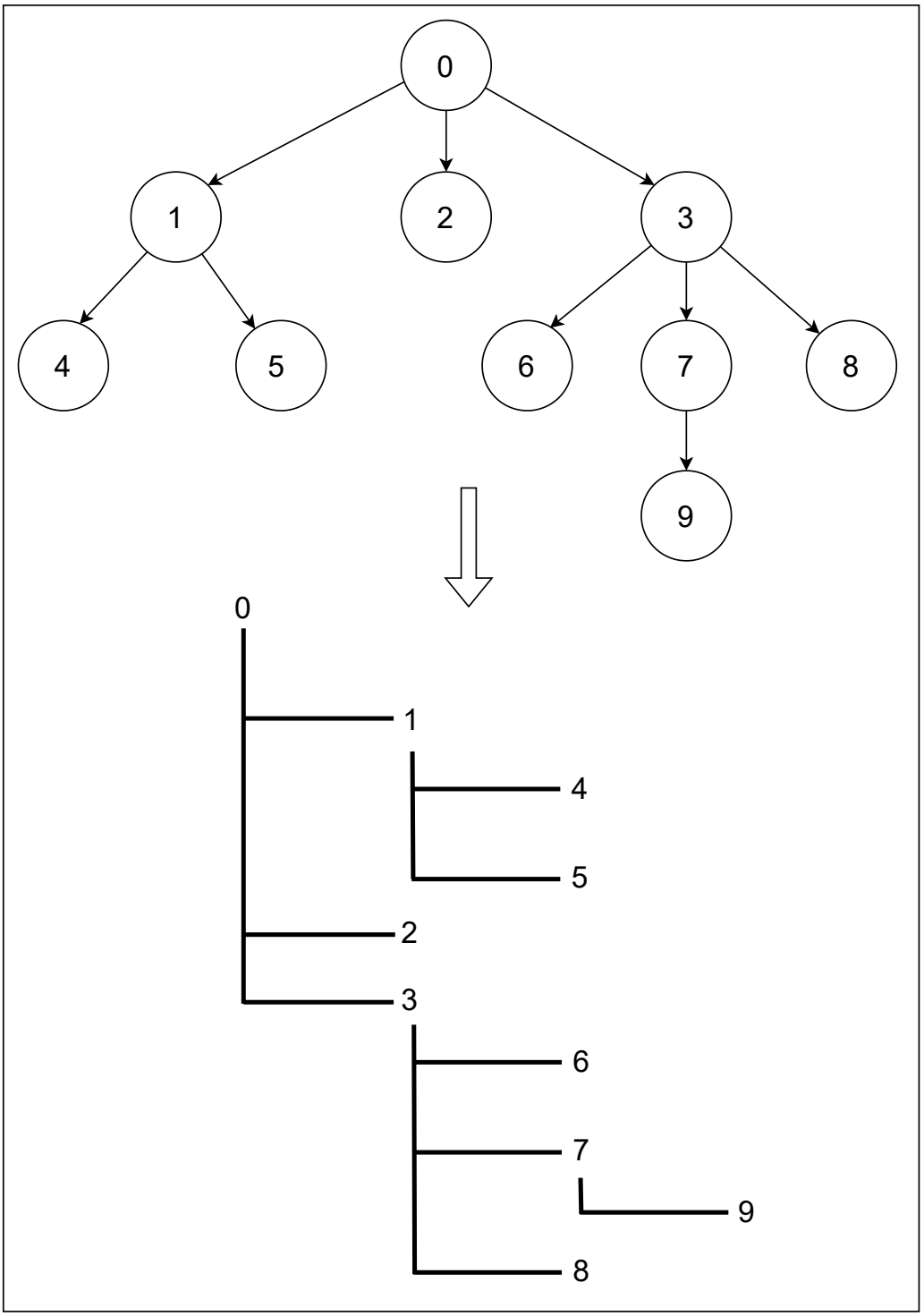


Figura 4.2: Ejemplo de árbol creado con la librería *tree.hh*. Cada uno de los nodos puede tener un número indefinido de hijos.

- *leaf_iterator*. Tan solo se recorren las hojas del nivel más alejado de la raíz del árbol. La secuencia correspondiente a la figura 4.2 es 4 →5 →2 →6 →9 →8.

Entonces, a la hora de realizar las siguientes búsquedas que *SFF* lleva a cabo en cada iteración (observar líneas 10 y 11 del algoritmo 4):

- $d_i, q' = T_i$.vecinoMasCercano(q_c). Búsqueda del nodo más próximo del árbol actual respecto del nodo de dicho árbol que se desea expandir.
- $d_j, q_j = argmin_{j \neq i} T_j$.vecinoMasCercano($q_c$). Búsqueda del nodo más próximo de todos los árboles exceptuando el árbol propio del nodo que se desea expandir.

Se tiene un problema, ya que hay que tener en cuenta que si se realiza la búsqueda con *tree.hh* recorriendo cada uno de los árboles con alguno de los métodos mencionados anteriormente, el coste computacional que puede conllevar es muy elevado, por lo que el tiempo de ejecución también lo es.

Por ello, se han decidido utilizar árboles binarios. De esta forma estas búsquedas se pueden realizar de forma eficaz, lo que conlleva un descenso considerable en el tiempo de computación.

Los árboles binarios se caracterizan por el hecho de que cada uno de sus nodos solo puede tener dos hijos. Son un tipo de estructura de datos muy usada en computación la cual se centra en permitir que las búsquedas *NN* se hagan de una forma rápida y eficiente.

Los árboles binarios que se han utilizado son:

- ***K-d Tree*** (*k-dimensional tree*) [1][2]
- ***Cover Tree*** [3][4]
- ***Quadtree*** [5][6]
- ***PH-tree*** [7][8]

Se ha leído la documentación pertinente acerca de ellos y posteriormente se ha realizado una búsqueda para encontrar cada una de las librerías que nos permitan utilizarlos. Cada uno de ellos se encuentra explicado a continuación.

4.2.2 Árboles *K-d Tree* [1] [2]

Árbol de N dimensiones cuyo procedimiento consiste en dividir el espacio en planos cada vez que se añade un nodo al árbol. En este caso, los árboles utilizados son de 2 dimensiones (x e y). En la figura 4.3 se puede observar un ejemplo sencillo de este tipo de árboles. Cada hijo de un nodo se puede añadir a la izquierda o a la derecha de éste, dividiendo el espacio siguiendo las coordenadas x e y alternativamente.

A la hora de realizar la búsqueda del vecino más cercano, primeramente se encuentra un nodo de referencia respecto al punto objetivo, descendiendo desde el nodo raíz hasta un nodo hoja del árbol. Posteriormente, se recorre el árbol en sentido ascendente buscando planos cercanos al punto objetivo para así encontrar el nodo más cercano. Finalmente, cuando se llega al nodo raíz de nuevo, se detiene la búsqueda.

La librería que se ha utilizado se encuentra disponible en [88].

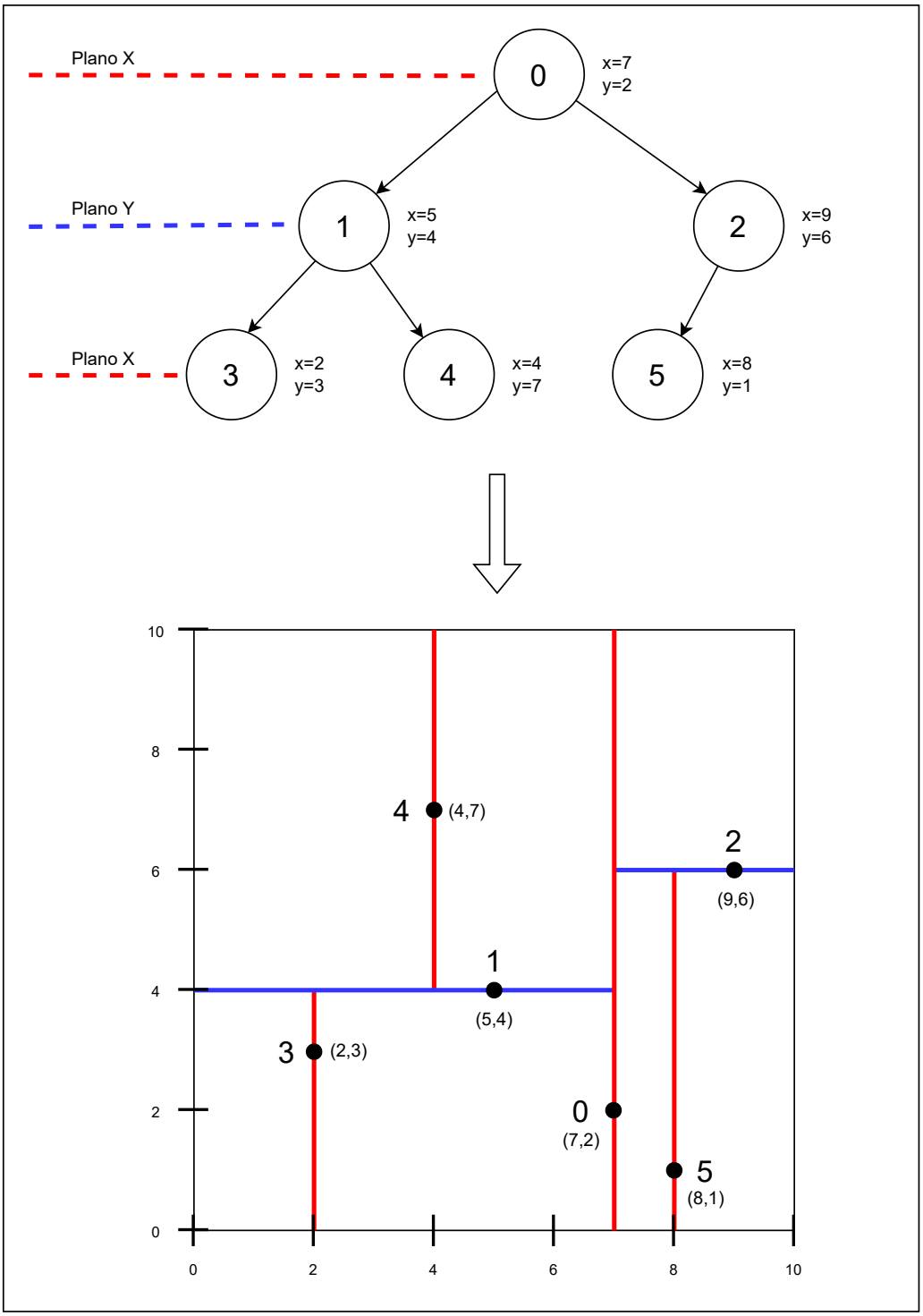


Figura 4.3: Ejemplo de creación de un árbol *K-d Tree*. Cada vez que se añade un nodo, se divide el espacio en planos dependiendo de las dimensiones con las que se trabaja.

4.2.3 Árboles *Cover Tree* [3] [4]

Los árboles *Cover Tree* han sido específicamente diseñados para realizar búsquedas del vecino más cercano de forma más eficaz y, por lo tanto, más rápida de las que por ejemplo ofrece el método de fuerza bruta.

Se trata de un árbol basado en una cantidad de puntos S determinada, que se divide en niveles en el que cada uno de ellos es una cobertura para el nivel inferior. Cada nivel se indexa con un valor entero i el cual decrece cada vez que el árbol va descendiendo. Cada nodo del árbol está asociado a uno de los puntos de S. Cada punto de S puede estar asociado a múltiples nodos del árbol y además, se requiere que cualquiera de los puntos que componen S aparezca como máximo una vez en cada nivel.

Respecto a los valores enteros i, se deben cumplir las siguientes características:

- Anidamiento: $C_i \subset C_{i-1}$. Cuando un punto $p \in S$ aparece en C_i, entonces cada nivel inferior del árbol tiene un nodo asociado en p.
- Propiedad de *Covering*. Para cada $p \in C_{i-1}$, existe un punto $q \in C_i$ tal que $d(p,q) < 2^i$ y el nodo en el nivel i asociado con q es padre del nodo en el nivel $i-1$ asociado con p.
- Separación . Para cada distinto $p, q \in C_i, d(p,q) > 2^i$

Cabe decir que en su publicación [3] se menciona cómo se deben realizar las inserciones, búsquedas y eliminaciones de los nodos que componen el árbol. Se cree que habiendo observado las características principales del árbol es suficiente para entender cuál es su propósito, por ello no se entrará en más detalle respecto al funcionamiento de este tipo de árboles.

La librería que se ha utilizado se encuentra disponible en [89].

4.2.4 Árboles *Quad Tree* [5] [6]

Árbol de 2 dimensiones creado a partir de la división del espacio de configuraciones, dónde cada subdivisión representa un nodo del árbol. En la figura 4.4 se puede observar un ejemplo del proceso que sigue este tipo de árbol. Inicialmente tenemos el nodo raíz, que es todo el espacio (nodo A). A continuación, se divide el nodo A en 4 cuadrantes iguales, por lo que tenemos que las secciones contenidas en A (nodo raíz) serán los hijos B, C, D y E. El proceso de inserción de puntos consiste en buscar el subcuadrante al que pertenece éste sin que esté ya ocupado por otro, si se da el caso, dicho cuadrante se divide de nuevo. Si por ejemplo se inserta un punto en *x=1*, *y=1,5* se debe subdividir el nodo I, debido a que ya contiene un punto. Este proceso permite crear un árbol dónde cada nodo representa un espacio lógico de 2 dimensiones.

La librería que se ha utilizado se encuentra disponible en [90].

4.2.5 Árboles *PH-tree* [7] [8]

PATRICIA-hypercube-tree (PH-tree) es una estructura de almacenamiento multidimensional basada en *Quad Trees*. Por defecto se almacenan puntos k-dimensionales que consisten en k valores enteros de 64 bits. Esta destinado a realizar búsquedas *k-NN*, es decir, de un

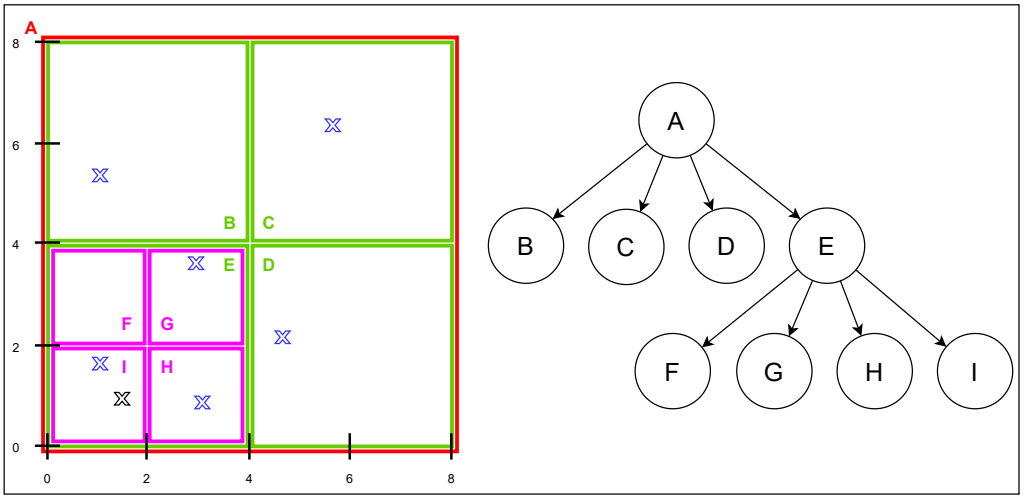


Figura 4.4: Ejemplo de creación de un árbol *quadtree*.

cierto número de vecinos más cercanos, sin necesariamente tener que encontrar el más cercano posible.

De forma similar a lo comentado en el caso de los *Cover Tree*, se trata de una estructura de datos sofisticada, en cuyo documento [7] se encuentra toda la información acerca de los datos que se pueden utilizar, los experimentos que se han realizado acerca de las búsquedas del vecino más cercano, inserciones y más información acerca de su funcionamiento.

La librería que se ha utilizado se encuentra disponible en [91].

Simplemente se utilizan las inserciones dinámicas y las búsquedas *k-NN* forzando a que se encuentra el vecino más cercano gracias a las funciones que incorpora la librería.

4.3 Bloque *SFF*: implementación algoritmo *SFF*

Tal y como se ha explicado en el apartado 3.2 y como se observa en el algoritmo 4, el objetivo principal de *SFF* es obtener una matriz P_{ij} la cual contenga las rutas entre los puntos objetivo que *SFF* ha logrado conectar gracias al crecimiento de los árboles. Se observa una representación de esta matriz en la figura 4.5, la cual está compuesta por tres dimensiones. Se aprecia como en la tercera dimensión se almacenan las rutas entre los diversos nodos. A continuación se presenta cómo se ha organizado la clase *SFFplanner* para así poder utilizar el algoritmo *SFF* y posteriormente se muestran los diagramas de flujo correspondientes a sus dos principales funciones. Finalmente, se explican las variables y funciones más relevantes para comprender cada uno de sus elementos.

4.3.1 Organización del planificador *SFF*

Se ha realizado una copia de la clase *ARAplanner* ya que todos los planificadores de *SBPL* tienen la misma estructura. Las clases *AbstractSearchState* y *SBPLPlanner* se heredan de la cabecera *planner.h* para así poder utilizar las variables y funciones que provee.

Posteriormente, se introducen todas las variables y funciones necesarias en la sección privada de la clase para así dejar paso a la función pública *replan*. La función *replan* es la

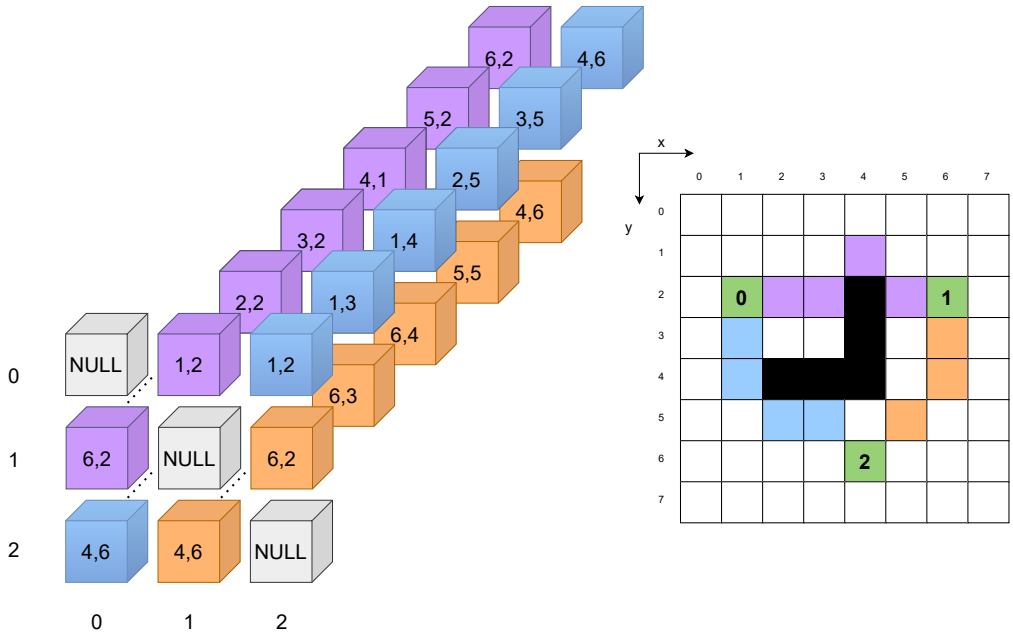


Figura 4.5: Matriz P_{ij} contenedora de las rutas entre los diversos puntos objetivo.

destinada a ser usada desde un programa externo para poder hacer uso del planificador. De esta forma se replanifica cada vez que sea necesario.

En la figura 4.6 se observa cómo está organizada la clase.

4.3.2 Diagramas de flujo del planificador *SFF*

A continuación, en las figuras 4.7 y 4.8, se presentan los diagramas de flujo de las dos funciones principales del planificador. Por un lado, se puede observar la función *replan_SFF*, la cual permite que el planificador sea utilizado desde un programa externo, en este caso el programa de test *main.cpp*. La función *replan* se encarga de utilizar la función *SFF_algorithm* para ejecutar el algoritmo *SFF* y así poder calcular la matriz P_{ij} y la matriz de costes que el programa principal desea obtener.

Debido a cómo se ha construido, la función *replan* es algo más que una simple función destinada a replanificar, en su interior se ha construido una estructura diseñada para poder ejecutar el algoritmo una sola vez (tal y como debe ser desde el punto de vista de un programa externo) o para ejecutarla varias veces en función del modo de uso que se haya seleccionado en el archivo *params_RRTSFF.ini*.

De esta forma, gracias a un solo bucle, se tiene la posibilidad de ejecutar el test que se desee. Por ello, también se utiliza el algoritmo *TSP* en el interior de la función *replan*, de esta forma se calcula el coste total del algoritmo y se almacena para posteriormente analizar el comportamiento del coste calculado por *TSP* en función de la prueba realizada.

De cara al usuario que utilice la función *replan*, no se observa ningún impacto respecto a tiempo de computación, ya que cada uno de los bloques innecesarios para una simple ejecución (crear *targets* en cada iteración aplicar algoritmo *TSP*, guardar resultados en archivo excel, limpieza de variables o dibujar camino *TSP*) se desactivan par así no tener influencia en el tiempo de computación.

class SFFplanner : public SBPLplanner

public:
-
protected:
-
private:
struct algorithm_params params; // *struct algorithm_params (utilsRRTSFF.h)*

std::shared_ptr<utilsRRTSFF> SFFutils; // *Utils*

// *Punto OListPoint, compuesto por un árbol y un iterador.*
struct OListPoint{
std::shared_ptr<tree<RRTSFFNode>> tr;
tree<RRTSFFNode>::iterator it_pre;
};

vector<struct OListPoint> Open_List; // *Creación del vector Open_List. Nodos disponibles para expandir los árboles.*

struct OListPoint random_item_from_OL; // *q (Paper SSF)*
RRTSFFNode q_rand; // *q_c (Paper SSF)*
float dist_q_near; // *d_i (Paper SSF)*
float dist_q_item; // *d(q, q_c) (Paper SSF)*
float dist_q_near_tree; // *d_j (Paper SSF)*
bool succ; // *(Paper SSF) Para asegurar que si en k iteraciones el árbol no ha crecido, ese nodo se elimina.*
RRTSFFNode q_point_found; // *Punto que sirve para guardar las coordenadas del nearest neighbour de otro árbol.*

Variables

Funciones

public:
virtual int replan_SFF(std::vector<std::vector<std::vector<int>>> &Pij, std::vector<std::vector<int>> &costs);
protected:
-
private:
// *Algoritmo SFF*
void SFF_Algorithm(std::vector<std::vector<std::vector<int>>> &Pij, std::vector<std::vector<int>> &costs);

// *Funciones del algoritmo*
RRTSFFNode create_qrand(struct OListPoint);
float find_qnear(tree<RRTSFFNode> &tr, RRTSFFNode q_rand, RRTSFFNode &q_near, int mode);

// *Funciones auxiliares del algoritmo*
void initBinaryTree();
void rootBinaryTree();
void NNSearch_1();
void NNSearch_2();
void addNode();

Figura 4.6: Organización de la clase *SFFplanner*. Se pueden observar las variables y funciones implementadas.

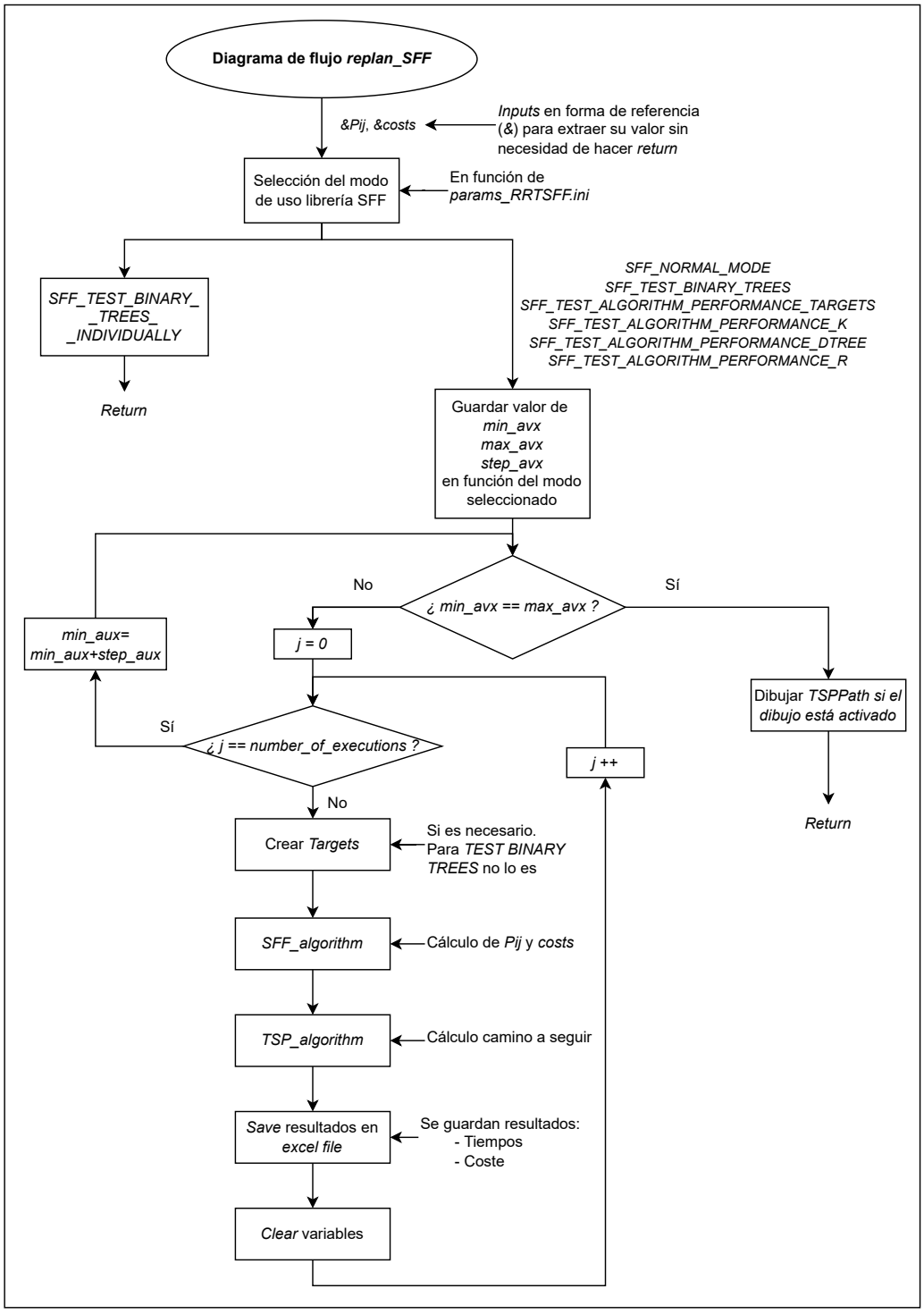


Figura 4.7: Diagrama de flujo de la función *replan_SFF*.

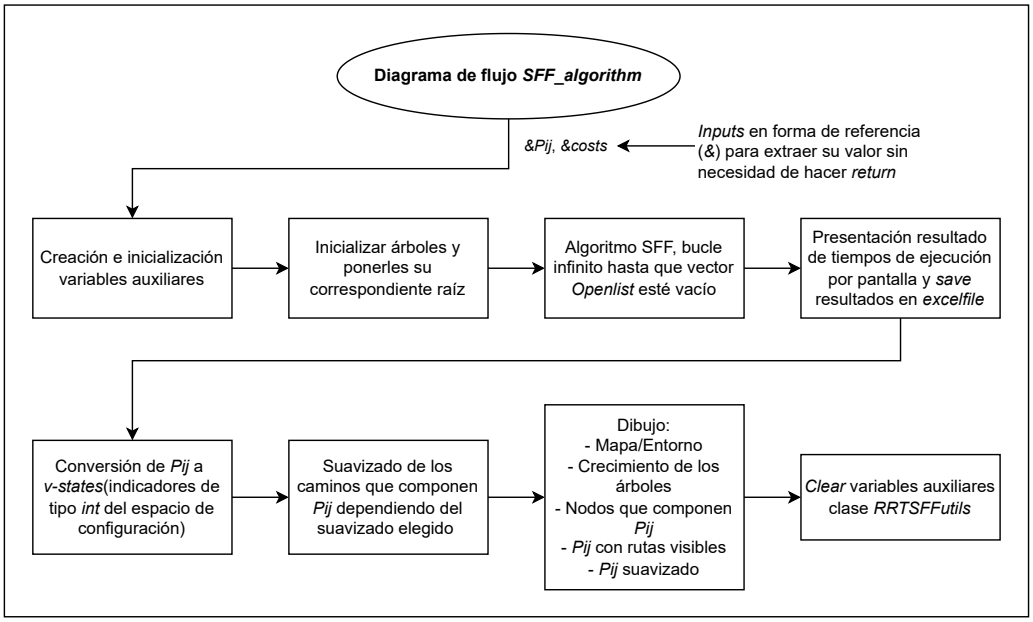


Figura 4.8: Diagrama de flujo de la función *SFF_algorithm*.

Respecto a la función *SFF_algorithm*, se puede observar que a parte de implementar el algoritmo descrito en 3.2.3 y presentado en el pseudocódigo 4, se realizan una serie de acciones tales como: presentar los tiempos de cada una de las secciones del algoritmo *SFF*(muy útil para el apartado 5 a la hora de calcular tiempos de ejecución), representar en figuras todo el proceso que *SFF* realiza, o limpieza de variables las cuales, tal y como pasa con la función *replan_SFF*, se desactivan a la hora de utilizar el algoritmo de forma normal desde un programa externo.

4.3.3 Definición de las variables utilizadas y contenido de las funciones del planificador

Refiriéndonos a la figura 4.6, a continuación se hace una descripción breve respecto a las variables y funciones declaradas para el correcto funcionamiento del planificador.

- Respecto a las variables utilizadas, se puede observar que son las mencionadas en 3.2.3, necesarias para poder implementar el algoritmo *SFF*:
  - ***params***. Incluye el vector de puntos objetivo, k, d_{tree} y R, entre otras variables importantes para inicializar el algoritmo.
  - ***Open_List***. Vector de puntos de tipo *OListPoint*, el cual sirve para guardar todos los nodos disponibles que en cada iteración pueden ser expandidos. *OListPoint* guarda tanto el árbol como la posición de cada nodo dentro del árbol.
  - ***random_item_from_OL***. Corresponde al nodo q, i que se selecciona de forma aleatoria del vector *Open_List* para intentar ser expandido.
  - ***q_rand***. Correspondiente a la variable q_c del algoritmo *SFF*. Configuración aleatoria alrededor de q la cual cumple $\varrho(q_c, q_c) < R$

  - ***dist_q_near***. Correspondiente a la variable d_i, q' del algoritmo *SFF*. Distancia hasta el vecino más cercano del árbol al que pertenece el nodo que se desea expandir.
  - ***dist_q_item***. Correspondiente a la distancia $\varrho(q, q_c)$ del algoritmo *SFF*. Distancia entre los nodos q y q_c.
  - ***dist_q_near_tree***. Correspondiente a la variable d_j, q_j del algoritmo *SFF*. Distancia hasta el vecino más cercano de todos los árboles restantes.
  - ***q_point_found***. Nodo encontrado como vecino más cercano de todos los árboles restantes. Corresponde al nodo q_j.
  - ***succ***. Sirve como comprobante para asegurar que si en k iteraciones no se ha podido expandir el árbol, el nodo seleccionado *random_item_from_OL* se elimina del vector *Open_List*.

- ***replan_SFF***. Método principal el cual se debe llamar desde el programa principal para poder utilizar el algoritmo *SFF*. Como parámetros se le introducen las matrices P_{ij} y *costs*, con las cuales se obtienen:
  - P_{ij}: identificadores del espacio de configuración entre cada uno de los nodos que se hayan conectado gracias a los árboles que se han generado con *SFF*.
  - ***Costs***: costes correspondientes a cada uno de los caminos generados entre los puntos objetivo.

  La ejecución de esta función se ha explicado en 4.3.2.

- ***SFF_algorithm***. Esta es la función encargada de ejecutar el algoritmo *SFF*, tal y como se puede observar en el pseudocódigo 4. En primer lugar, se inicializan todas las variables necesarias y en particular, se coloca la raíz a cada árbol en función de los puntos objetivo que existan. Posteriormente, se lleva a cabo el algoritmo creando así las matrices P_{ij} y *costs*. El comportamiento se ha detallado en 4.3.2.

- ***create_qrand***. Función necesaria para poder calcular la variable *q_rand*. Generar el punto aleatorio utilizando un bucle infinito el cual comprueba que el punto generado cumple dos condiciones: se encuentre dentro del radio R que define el algoritmo *SFF* y que no forma parte de ningún obstáculo, es decir, que se encuentra dentro del espacio de configuración libre de éstos.

- ***find_qnear***. Función utilizada para calcular el vecino más cercano cuando se utiliza la librería *tree.hh*. Se recorre con fuerza bruta el árbol que se le pase por parámetro hasta encontrar el vecino más cercano.

  Cabe mencionar que también se realizó una prueba recorriendo solo las hojas del árbol gracias a *breadth_first_iterator* pero no se obtuvieron buenos resultados debido a que no se tenían en cuenta todos los nodos del árbol y por ello, en ocasiones los árboles colisionaban entre ellos. Esta implementación no se ha añadido a la versión final del planificador.

- ***initBinaryTree***. Función para inicializar el árbol binario correspondiente.

- ***rootBinaryTree***. Función para inicializar todas las raíces de los árboles en función de los puntos objetivo que se estén utilizando.
- ***NNSearch_1***. Búsqueda del vecino más cercano correspondiente al mismo árbol del nodo que se desea expandir. En función del árbol binario que se esté utilizando, se utilizan las funciones de cada una de las librerías mencionadas en 4.2.2, 4.2.3, 4.2.4 y 4.2.5 y, posteriormente, se calcula la distancia hasta dicho nodo.
- ***NNSearch_2***. Búsqueda del vecino más cercano de todos los árboles exceptuando el propio al que se realiza la expansión. Se recorren todos los árboles calculando el vecino más cercano y éste se actualiza en función de si se encuentra uno nuevo. Se utilizan las funciones correspondientes dependiendo del árbol binario utilizado y finalmente se calcula la distancia hasta dicho nodo.
- ***addNode***. Función que añade el nodo q_c al árbol binario correspondiente, nuevamente se utilizan las funciones pertinentes de cada una de las librerías que se han mencionado en la función *NNSearch_1*.

4.4 Bloque *TSP*: incorporación de *TSP*

El principal objetivo de este bloque es dar el soporte necesario al planificador *SFF* o al programa principal para que sea posible utilizar *TSP* una vez se haya aplicado *SFF*. A partir de la matriz de costes calculada por *SFF*, *TSP* debe ejecutarse y calcular el camino que se debe recorrer juntamente con su coste asociado.

A continuación se muestra cual de los algoritmos *TSP* de la librería seleccionada en el apartado 3.3.4 se utiliza.

Posteriormente, en el apartado 4.4.1, se enseñan las modificaciones realizadas para poder adaptar dicho algoritmo a *SBPL*, juntamente con la organización y diagrama de flujo que sigue la clase *TSP* creada.

Finalmente, se explican todas las variables y funciones creadas para el correcto funcionamiento de *TSP*.

4.4.1 Selección del algoritmo *TSP* y adaptación a *SBPL*

En esta sección se menciona la adaptación que se ha realizado del algoritmo elegido para resolver el problema de *TSP*. Realmente, para poder seleccionar un algoritmo entre los que se presentan en el trabajo [78] (*Exact, Constructive, Local Search y GRASP*), se han tenido que realizar diversas pruebas para comprobar cuál es el algoritmo que resuelve el problema de *TSP* de forma correcta y en el menor tiempo posible.

En el apartado 5.2 se detallan las pruebas realizadas y se adelanta que el algoritmo elegido ha sido *GRASP*. A partir de este punto, en este apartado se detallan los archivos añadidos a la librería y la modificación que se ha realizado a uno de ellos para adaptarse al tipo de utilizados en *SBPL*.

Archivos añadidos a *SBPL* para el correcto funcionamiento de *TSP*

De la librería original [78] se han extraído ciertos archivos para poder utilizar el algoritmo *GRASP*. En concreto, se han añadido las siguientes cabeceras y archivos de código fuente:

- *UndirectedCompleteGraph.h* y *UndirectedCompleteGraph.cpp*: la clase *UndirectedCompleteGraph* se encarga de gestionar los grafos que se deben crear antes de utilizar alguno de los algoritmos *TSP*. Se ha añadido un nuevo constructor llamado *UndirectedCompleteGraph(float *G, unsigned int size)* para poder crear un grafo de forma sencilla gracias a una matriz de *floats* y su tamaño

  Se utilizan las funciones *UndirectedCompleteGraph(float *G, unsigned int size)*, *getPath()* y *getDistance()* para crear el grafo necesario para aplicar *TSP*, para obtener el camino y la distancia una vez el algoritmo *GRASP* finaliza.

- *Grasp.h* y *Grasp.cpp*: Clase que se encarga de ejecutar el algoritmo *GRASP*. Se utiliza la función *grasp(UndirectedCompleteGraph& graph, unsigned int alpha, unsigned int improvedIteration, unsigned int graspMaxIteration, unsigned int localSearchMaxIteration)* con la cual se obtiene nuevamente el grafo introducido, para posteriormente obtener la distancia del camino que ha calculado *GRASP* gracias a la función *getDistance()* de la clase *UndirectedCompleteGraph*.

- *ConstructiveHeuristic.h*, *ConstructiveHeuristic.cpp*, *LocalSearchHeuristic.h* y *LocalSearchHeuristic.cpp*: clases respectivas a los algoritmos *Constructive* y *Local Search* que se encuentran incluidos dentro de la clase *Grasp*, por lo tanto, también es necesario incluirlos.

4.4.2 Problema de obtención de nodos aislados tras aplicar *SFF* y repercusión sobre *TSP*

Quizás el problema más grave encontrado durante el transcurso del proyecto haya sido el que provoca que tras haber ejecutado el algoritmo *SFF*, exista algún nodo que se encuentre aislado o que tenga una sola conexión con algún otro árbol.

Durante las primeras pruebas realizadas se encontró este problema, la causa es debida a la aleatoriedad del crecimiento de los árboles *SFF*. Aunque en la mayoría de casos el crecimiento sea uniforme y los árboles acaben teniendo tamaños similares, puede ser que alguno de ellos crezca en menor proporción y por ello, solo consiga conectarse con un árbol vecino o incluso con ninguno.

En las figuras 4.9 y 4.10 se puede observar un ejemplo de cada uno de estos casos. Se han representado cada uno de los árboles con la matriz P_{ij} y el *roadmap* obtenido. Por lo tanto, se tienen dos situaciones:

- En el caso de la figura 4.9, donde el árbol número 0, situado a la izquierda del entorno, no tiene ninguna conexión. Por ello, es imposible el uso de *TSP*, debido a que la premisa inicial no puede ser cumplida. No se puede encontrar un camino que recorra todos los puntos objetivo, ya que para uno de los nodos no se ha encontrado ninguna ruta.

- En segundo lugar, respecto a la figura 4.10, se puede observar como el árbol número 1 solo está conectado con el árbol número 7, lo que provoca que tampoco se pueda cumplir la condición anterior. Se puede observar como el punto objetivo número 7 tendrá que ser visitado dos veces si se quieren visitar por lo menos una vez el punto número 1.

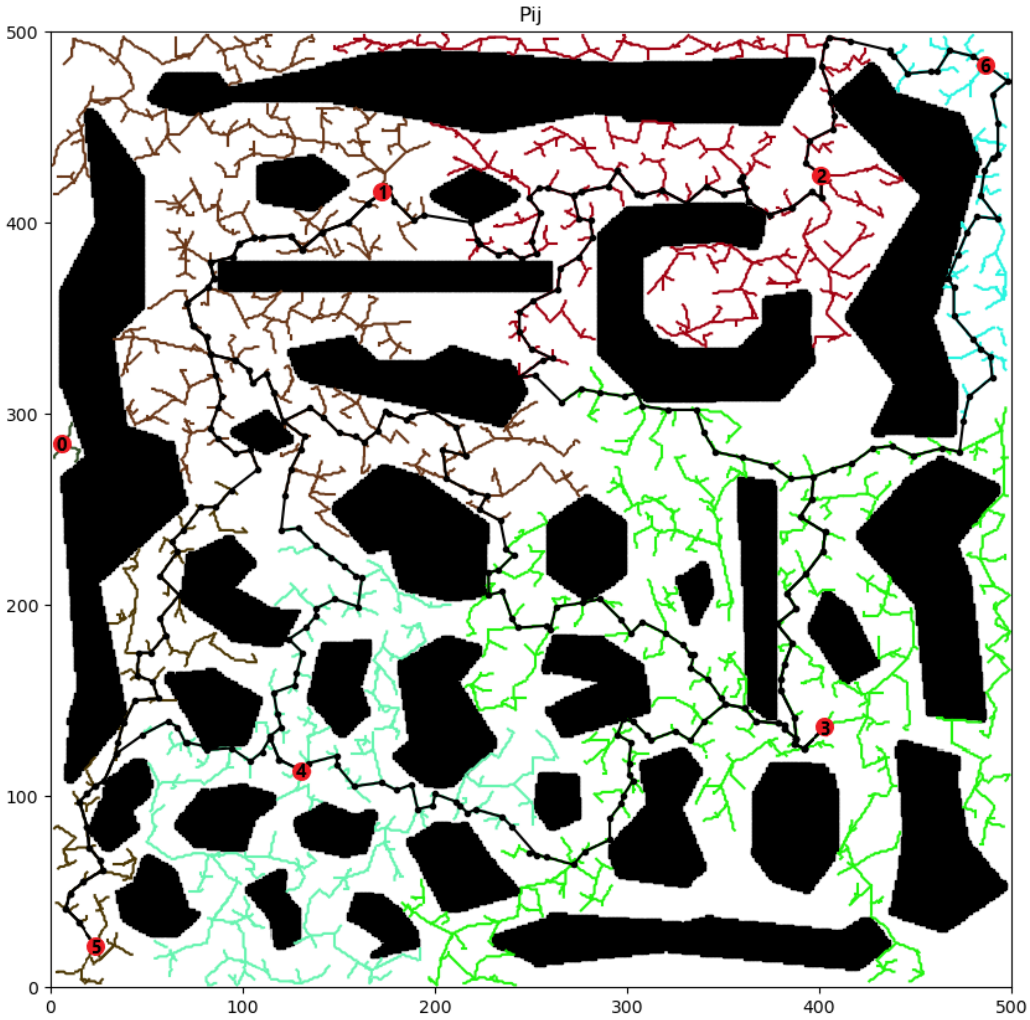


Figura 4.9: Caso en el que existe un árbol que no ha podido encontrar conexión con ninguno de sus vecinos más cercanos.

Por lo tanto, para cada una de las situaciones, se propone una solución distinta, las cuales se exponen a continuación.

Caso 1. Solución al caso de los nodos aislados: eliminación de éstos y aplicación de *TSP* sobre los nodos restantes

Para el primer problema, hay que tener en cuenta que se trata de una situación poco común e incluso excepcional, que no es sencilla de reproducir y que básicamente puede pasar por una de las siguientes razones:

- Mala selección del parámetro k del algoritmo *SFF*. Debido a una mala selección de este parámetro, es posible que alguno de los árboles no pueda desarrollarse lo suficiente y por ello se quede aislado. En la figura 4.11 se puede observar un caso simple en el que los árboles número 1 y 3 no pueden desarrollarse y por ello quedan asilados.

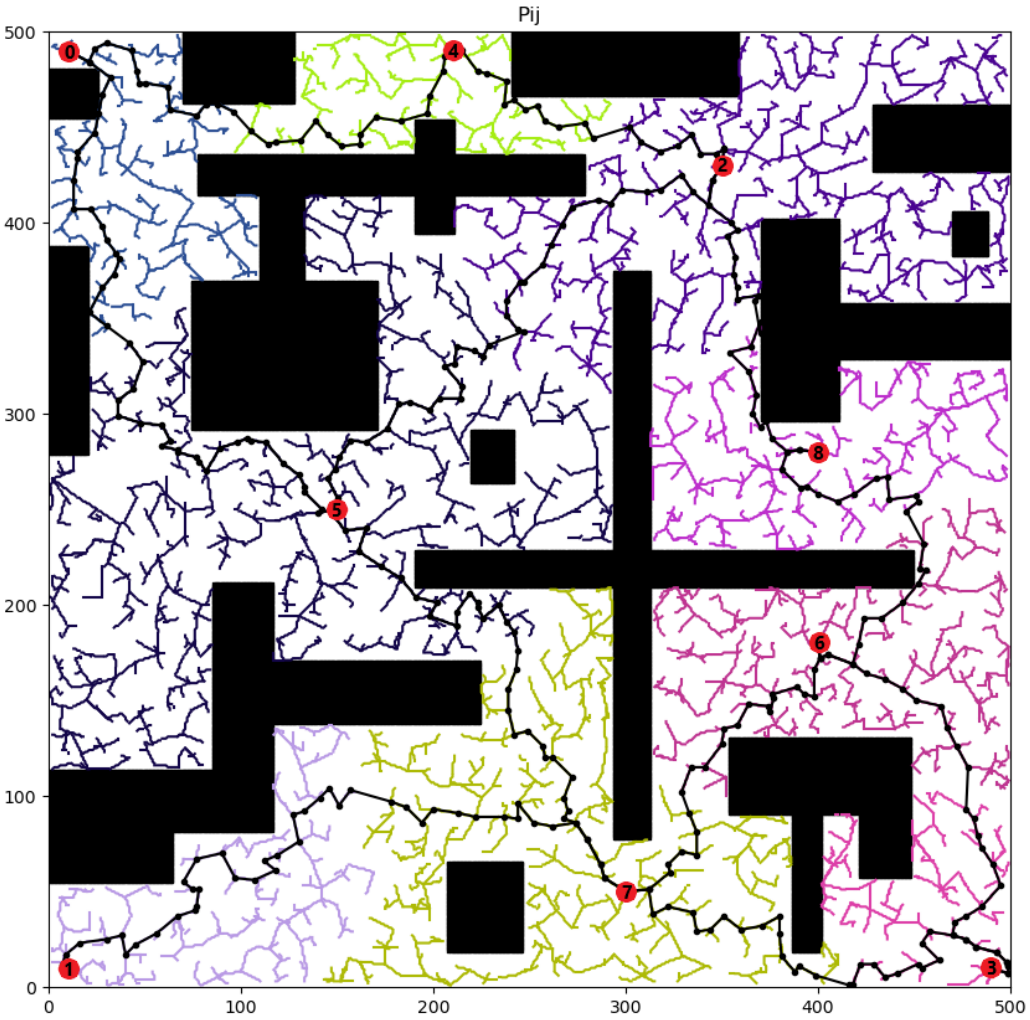


Figura 4.10: Caso en el que existe un árbol que solo se ha podido conectar a su vecino más cercano.

Por lo tanto, seleccionar para k un valor más bajo del que debería, hace que cada uno de los nodos tenga menos intentos para poder expandirse, por lo que probabilísticamente se reduce la posibilidad de que el árbol crezca haciendo así que quede aislado.

- Aunque no sea objetivo de este proyecto y se sobrepase el alcance de éste, hay que tener en cuenta que puede ser que se quiera utilizar el planificador *SFF* en algún tipo de entorno dinámico con objetos o partes del entorno cambiantes. Por lo tanto, es posible que en algún momento existan puntos objetivo que se encuentren aislados físicamente, por lo que sería imposible conectarlos con alguno de los árboles vecinos y por ello quedarían aislados.

Para el primer caso, la solución pasaría por identificar el valor erróneo de k y corregirlo para que en futuras ejecuciones no volviese a ocurrir.

Para el segundo caso, se ha decidido que una buena solución sería, debido a que es físicamente imposible alcanzar los nodos aislados, retirarlos de la lista de puntos objetivo y por lo tanto de la matriz de costes con la que trabaja *TSP* y simplemente utilizar el algoritmo con los puntos objetivo restantes. En la figura 4.12 se ve un ejemplo en el que los árboles 0 y 1 se encuentran aislados completamente, por ello se tendrian que retirar de la matriz de costes y aplicar *TSP* sobre los nodos restantes.

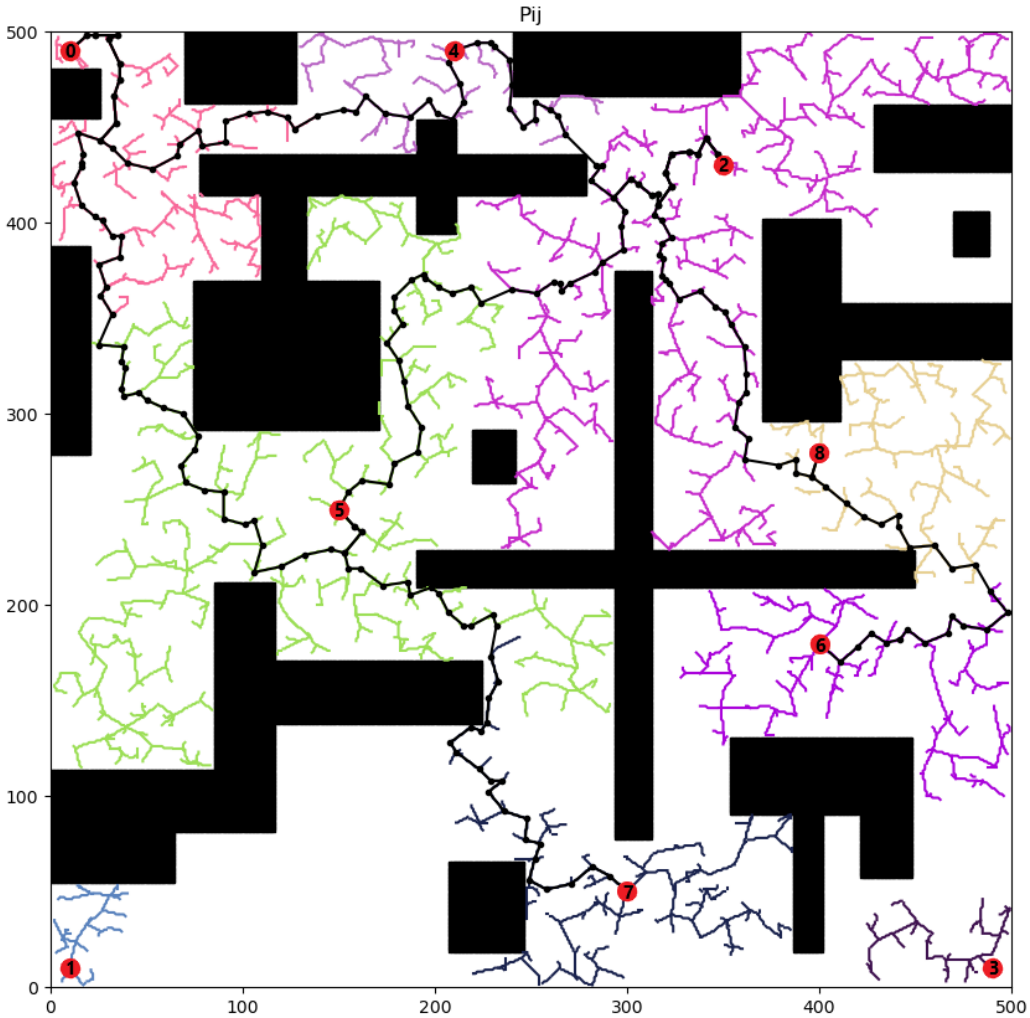


Figura 4.11: Caso en el que existen árboles aislados debido a una mala selección del parámetro k.

Caso 2. Solución al problema de los nodos con una sola conexión: aplicación del algoritmo *Dijkstra*

Para el segundo caso, en el que hay uno o más puntos los cuales solo tienen una conexión con otro de los puntos objetivo, se ha detectado que principalmente ocurre cuando nos encontramos en entornos en los cuales los puntos objetivo afectados se encuentran en zonas estrechas y de difícil acceso. Añadido a esto, de forma similar a como pasa en el

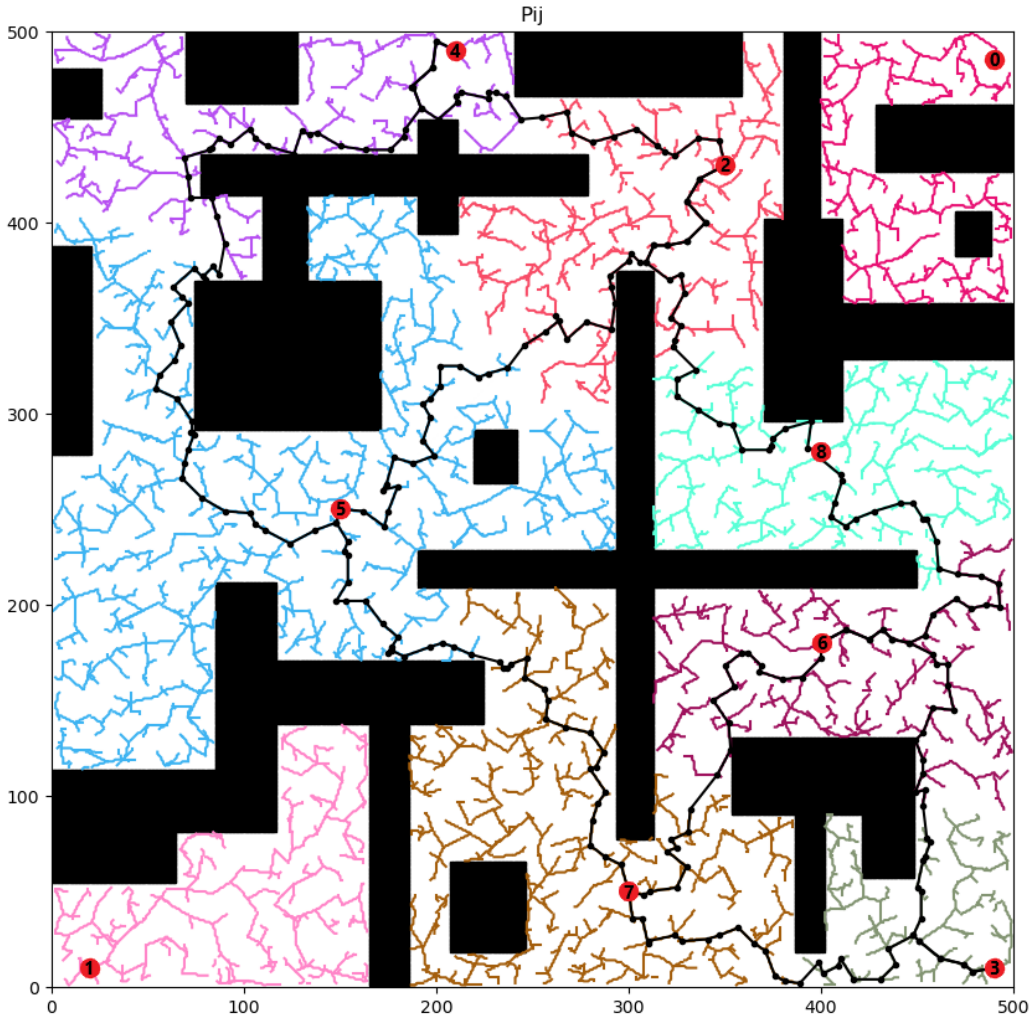


Figura 4.12: Caso en el que existen dos árboles aislados debido a que se han quedado físicamente aislados.

caso anterior, una mala selección de los parámetros k, d_{tree} y R causa que el efecto sea aun más notorio y que pueda ocurrir con más frecuencia. De todas formas, también puede ocurrir por el simple hecho de cómo se distribuyen los puntos objetivo.

En la figura 4.13 se observa un ejemplo en el que los nodos 0 y 5 solo tienen una conexión debido a que:

- El nodo 0 se encuentra en una zona de difícil acceso y está posicionado muy cerca del nodo 1.
- El nodo 5 está posicionado en la esquina inferior izquierda y muy alejado de los demás puntos.

Entonces, la repercusión de este problema respecto a *TSP*, es que éste no sea capaz de encontrar un camino que pase por cada uno de los puntos objetivo una sola vez y vuelva al punto inicial, ya que a la fuerza se tiene que pasar por lo puntos 1 y 6 dos veces para poder cumplir la misión.

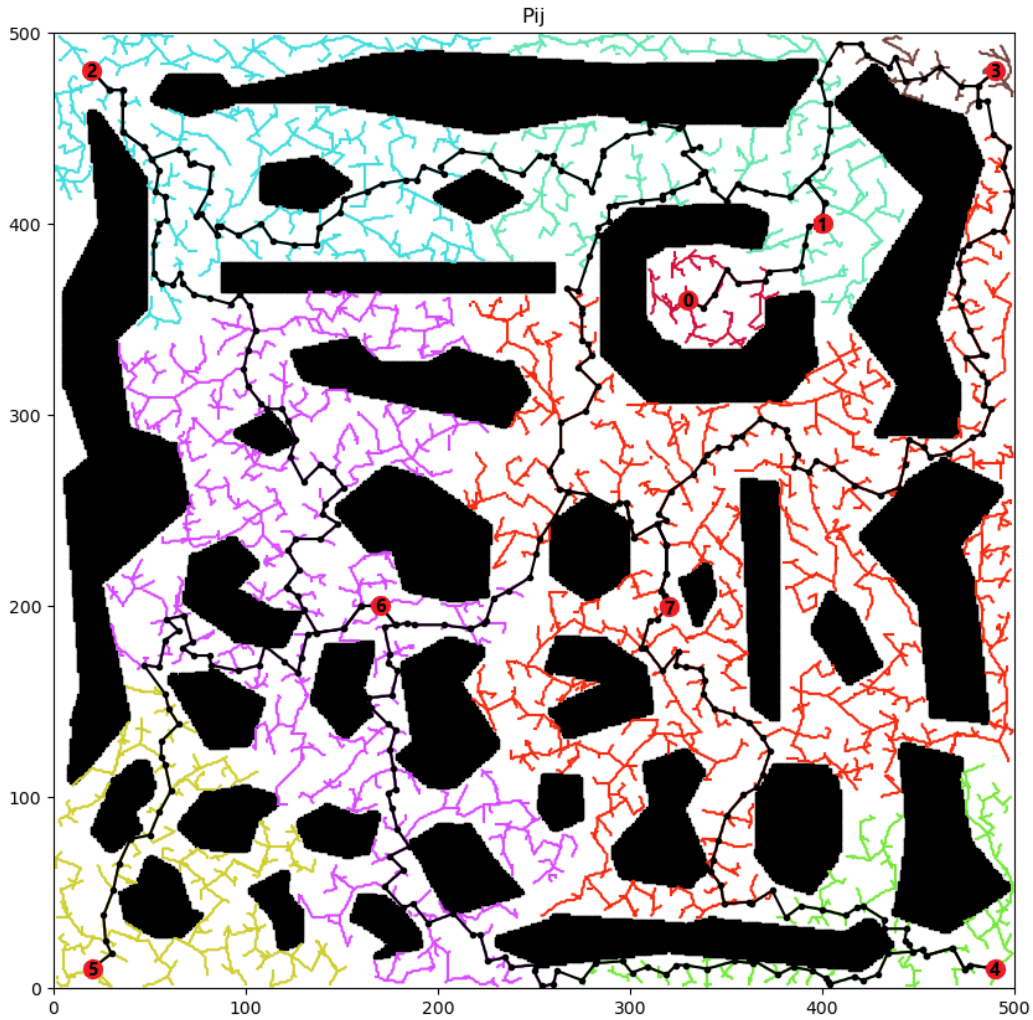


Figura 4.13: Caso en el que existen dos nodos con una sola conexión debido a su incorrecto posicionamiento.

Para poder solventar este problema y permitir que el algoritmo *GRASP* pueda devolver un resultado correcto, se ha decidido utilizar el algoritmo *Dijkstra*. Con él se pretende que, para aquellos nodos que solo tengan una conexión, se pueda actuar sobre ellos y así aplicar *Dijkstra* para aumentar su número de conexiones.

El algoritmo *Dijkstra* se presenta en el pseudocódigo 5, acompañado junto a la figura 4.15 para una mejor comprensión del algoritmo.

El objetivo del algoritmo es, dependiendo del nodo introducido por parámetro, encontrar las rutas óptimas a cada uno de los nodos que componen el grafo. Para ello se crean los siguientes vectores:

- ***Q***. Vector en el que se almacenan los nodos a visitar. A medida que el algoritmo avanza y se van alcanzando los nodos, este vector se va vaciando.

- ***dist***. En él se almacenan las distancias que se calcularán.

- ***prev***. En él se almacenan los nodos previos que se deben visitar para alcanzar cada uno de los nodos.

En cada iteración se selecciona el nodo que posea un menor valor de *dist*. Posteriormente, se calcula la distancia hasta cada uno de los nodos aún no visitados con los que se encuentra conectado y posteriormente, si el valor calculado es inferior al que ya posee ese nodo, se actualizan los valores de *dist* y *prev*. El resultado de este bucle se observa en la figura 4.15.

En la figura 4.14 se observa un ejemplo en el que el nodo 8 solo tiene una conexión. Por ello, se aplica *Dijkstra* sobre él para calcular el coste de los caminos hasta el resto de nodos. Además, en la figura 4.14 se puede observar el coste de cada una de las rutas y en las dos siguientes matrices presentadas en 4.1 y 4.2, se observa la modificación que la matriz P_{ij} original ha sufrido correspondiente a la aplicación de *Dijkstra* sobre el nodo número 8, hecho que ha conseguido calcular todas las conexiones sobre dicho nodo.

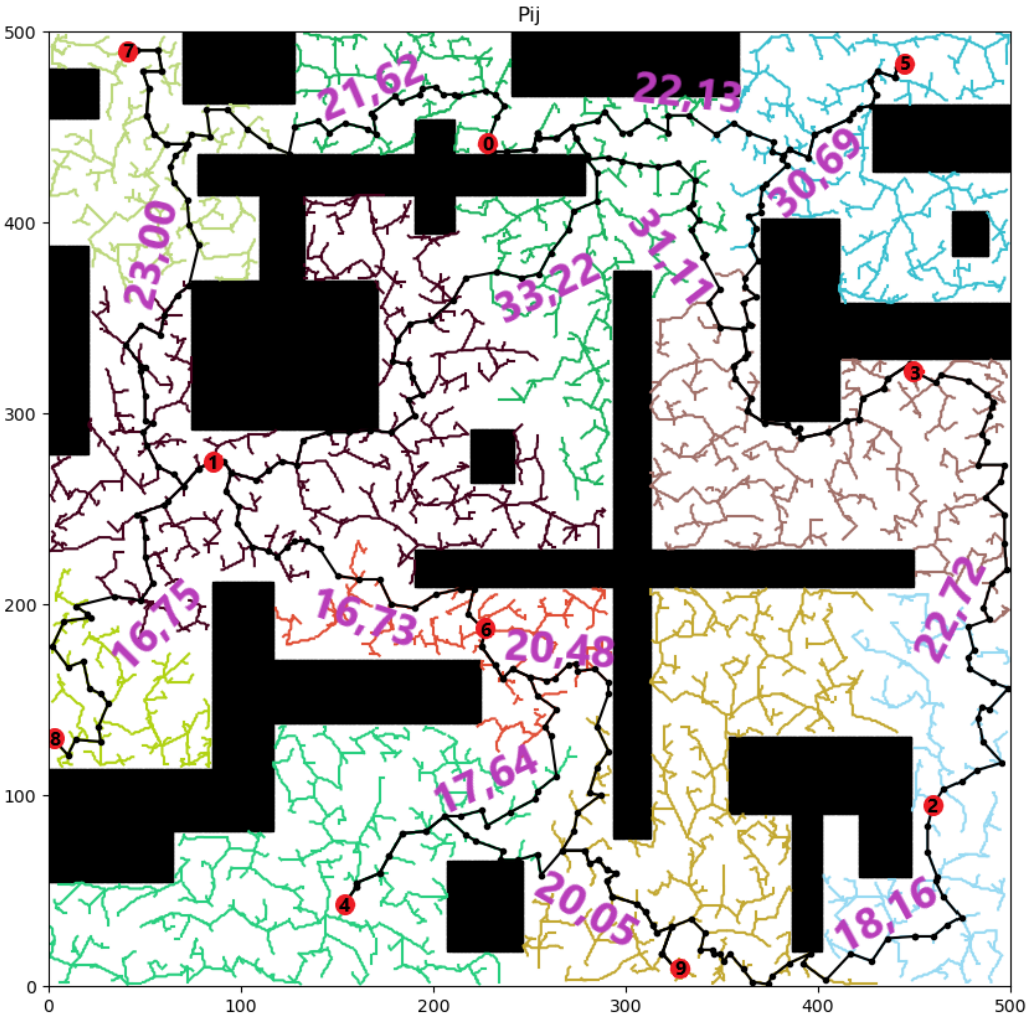


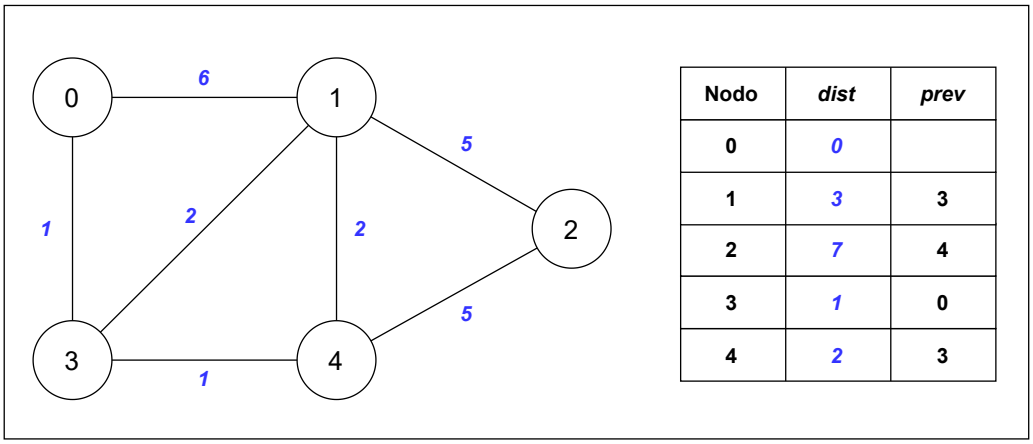
Figura 4.14: Ejemplo en el que tras aplicar el algoritmo *SFF*, uno de los nodos únicamente se queda con una conexión y por ello hay que aplicar el algoritmo de *Dijkstra* sobre él.

MATRIZ ORIGINAL OBTENIDA POR EL ALGORITMO SFF

$$\begin{bmatrix} 0 & 33{,}22 & \infty & 31{,}11 & \infty & 22{,}13 & \infty & 21{,}62 & \infty & \infty \\ 33{,}22 & 0 & \infty & \infty & \infty & \infty & 16{,}73 & 23{,}00 & 16{,}75 & \infty \\ \infty & \infty & 0 & 22{,}72 & \infty & \infty & \infty & \infty & \infty & 18{,}16 \\ 31{,}11 & \infty & 22{,}72 & 0 & \infty & 30{,}69 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 17{,}64 & \infty & \infty & 20{,}05 \\ 22{,}13 & \infty & \infty & 30{,}69 & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & 16{,}73 & \infty & \infty & 17{,}64 & \infty & 0 & \infty & \infty & 20{,}48 \\ 21{,}62 & 23{,}00 & \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & 16{,}75 & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 18{,}16 & \infty & 20{,}05 & \infty & 20{,}48 & \infty & \infty & 0 \end{bmatrix} \quad (4.1)$$

MATRIZ MODIFICADA TRAS APLICAR *DIJKSTRA* SOBRE EL NODO 8

$$\begin{bmatrix} 0 & 33{,}22 & \infty & 31{,}11 & \infty & 22{,}13 & \infty & 21{,}62 & 49 & \infty \\ 33{,}22 & 0 & \infty & \infty & \infty & \infty & 16{,}73 & 23{,}00 & 16 & \infty \\ \infty & \infty & 0 & 22{,}72 & \infty & \infty & \infty & \infty & 70 & 18{,}16 \\ 31{,}11 & \infty & 22{,}72 & 0 & \infty & 30{,}69 & \infty & \infty & 80 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 17{,}64 & \infty & 49 & 20{,}05 \\ 22{,}13 & \infty & \infty & 30{,}69 & \infty & 0 & \infty & \infty & 71 & \infty \\ \infty & 16{,}73 & \infty & \infty & 17{,}64 & \infty & 0 & \infty & 32 & 20{,}48 \\ 21{,}62 & 23{,}00 & \infty & \infty & \infty & \infty & \infty & 0 & 39 & \infty \\ 49 & 16 & 70 & 80 & 49 & 71 & 32 & 39 & 0 & 52 \\ \infty & \infty & 18{,}16 & \infty & 20{,}05 & \infty & 20{,}48 & \infty & 52 & 0 \end{bmatrix} \quad (4.2)$$



Nodo	*dist*	*prev*
0	*0*	
1	*3*	3
2	*7*	4
3	*1*	0
4	*2*	3

Figura 4.15: Ejemplo sencillo de implementación del algoritmo *Dijkstra*. Se observa como se ha aplicado sobre el nodo 0 para conocer el menor coste para llegar a cada uno de los nodos que componen el grafo.

El procedimiento genérico que se ha implementado cuando ocurre esta casuística es el siguiente:

1. Se ordenan los nodos de menor a mayor número de conexiones.

Algoritmo 5: Algoritmo *Dijkstra*

Input:
Graph Grafo que contiene nodos y vértices
Nodo Nodo del grafo sobre el cual se quiere aplicar el algoritmo
Output:
dist[] Distancia desde el nodo que se aplica *Dijkstra* hasta los nodos restantes
prev[] Nodos predecedores justo antes de llegar al resto de nodos

```
1  function Dijkstra (Graph, Nodo)
2      foreach (v de Graph) do
3          dist[v] ← ∞;
4          prev[v] ← INDEFINIDO;
5          dist[nodo] ← 0;
6      Añadir v a Q;
7      while (Q.size()> 0) do
8          u = nodo en Q con la menor dist[];
9          Eliminar u de Q;
10         foreach (vecino v de u que aún pertenezca a Q) do
11             alt ← dist[u] + Graph.edges(u, v);
12             if (alt < dist[v]) then
13                 dist[v] ← alt;
14                 prev[v] ← u;
15     return (dist[], prev[]
```

2. Se aplica el algoritmo *TSP* y se observa si el resultado que se obtiene, es decir, el coste del camino encontrado, es menor a un valor definido previamente.

3. En caso de que el coste sea mayor que infinito, se debe aplicar el algoritmo *Dijkstra* sobre el primer nodo de los que se han ordenado previamente para así aumentar el número de sus conexiones y así poder volver a aplicar *TSP*.

4. El proceso del punto anterior se repite hasta que el algoritmo *TSP* devuelva un coste menor al definido, lo que significa que se ha obtenido una ruta que no tiene caminos imposibles de realizar y que tiene el menor coste posible.

Aunque no se consiga visitar todos los puntos una sola vez y volver al punto de origen, se ha creído necesario realizar este proceso ya que es posible que en situaciones reales en las que no se haya contemplado exactamente todas las variabilidades que tenga el entorno, aparezca este problema. Entonces, es necesario contemplar algún mecanismo para calcular una ruta que nos permita visitar todos los puntos, aunque sea más de una vez, y volver al punto objetivo inicial.

4.4.3 Diagrama de flujo del algoritmo *TSP*

En este apartado se desea presentar el diagrama de flujo, brevemente enunciado en el apartado anterior 4.4.2, en el cual se observa el proceso que se sigue para poder resolver el problema de los nodos que se quedan completamente aislados o con una sola conexión.

En las figuras 4.16 y 4.17 se pueden observar los diagramas de flujos. Tal y como se ha introducido anteriormente, tras crear el grafo y aplicar el algoritmo *TSP* por primera vez, se entra en un bucle del cual solo se puede salir cuando *TSP* encuentre una ruta con un coste menor al requerido. Esto significa que todas las subrutas que se han encontrado son viables, ya sean pertenecientes a la matriz original obtenida por *SFF* o a una de las rutas que se han obtenido después de aplicar el algoritmo *Dijkstra*.

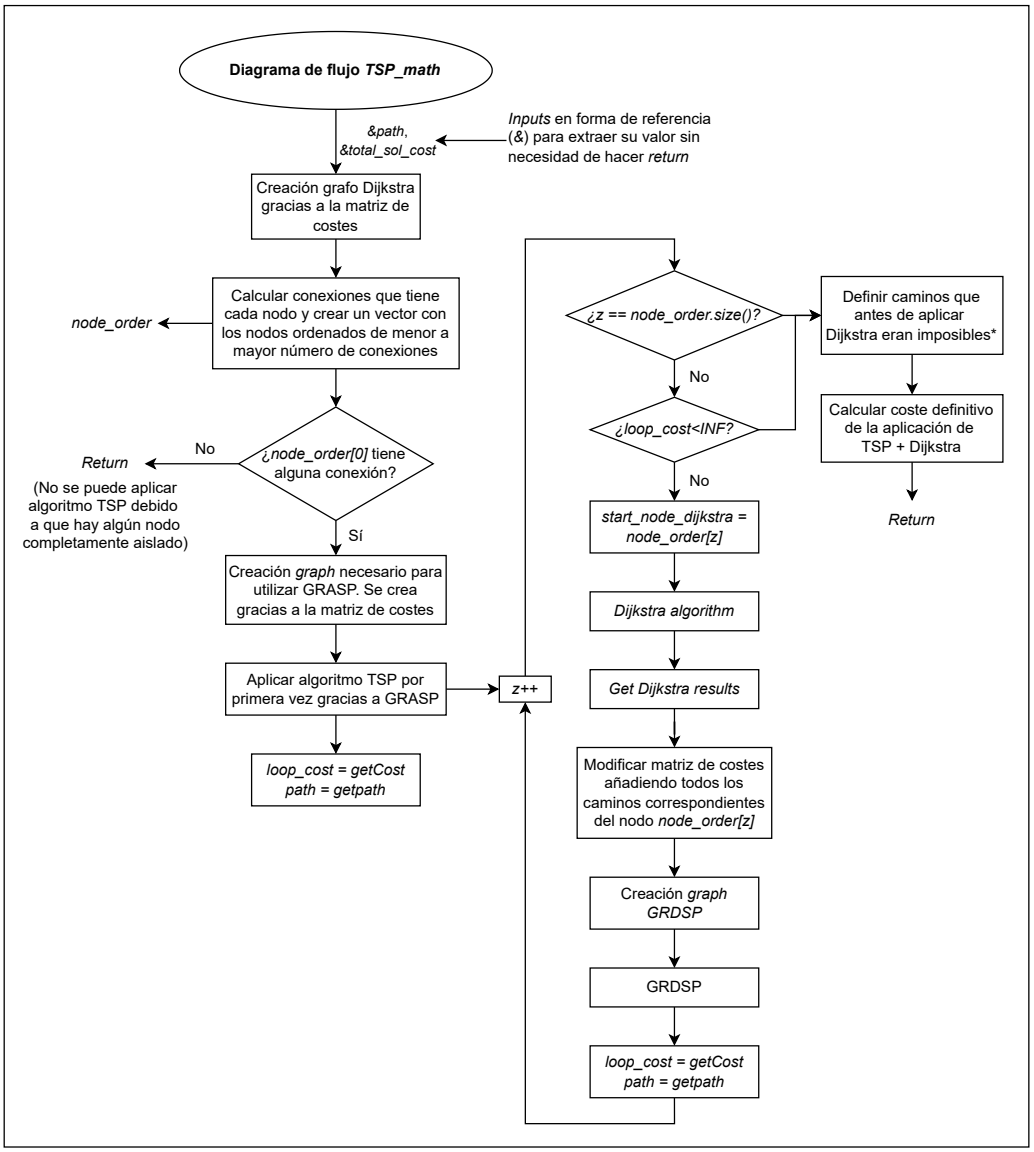


Figura 4.16: Diagrama de flujo de la función *TSP_math*.

El paso final es de vital importancia, ya que se tiene que repasar la ruta obtenida para

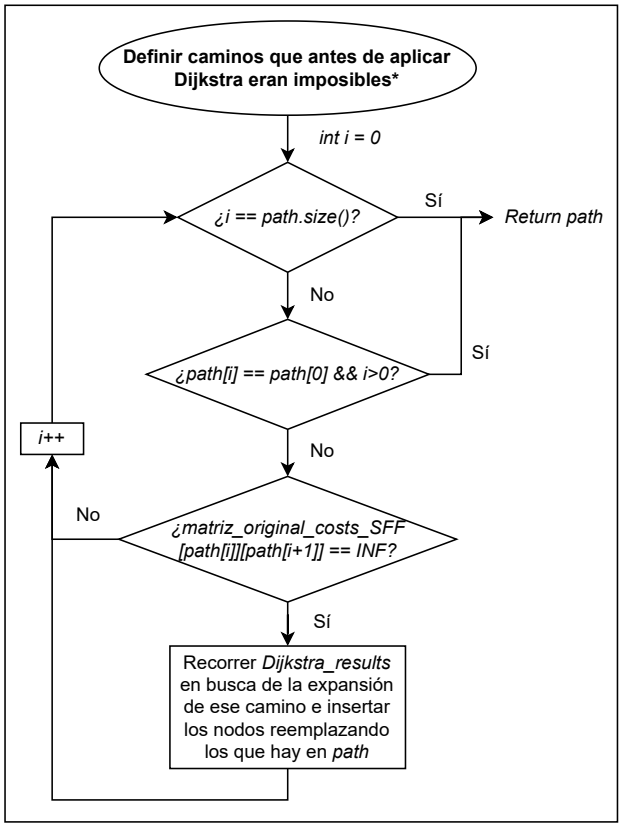


Figura 4.17: Continuación del diagrama de flujo de la función *TSP_math*.

comprobar si cada par de nodos adyacentes formaba parte de una de las rutas originales o si anteriormente esa ruta era imposible de realizar y se ha obtenido gracias a *Dijkstra*. Si ese es el caso, se debe consultar la ruta que *Dijkstra* ha creado y se debe añadir a la ruta original.

4.4.4 Definición de las variables utilizadas y contenido de las funciones del algoritmo *TSP*

A continuación se muestran las variables y funciones que componen la clase *TSPhandler*. Solo hay que realizar el proceso comentado anteriormente, por lo que solo se utilizan dos funciones: el constructor de la clase y la función *TSP_math*.

En la figura 4.18 se puede ver la composición de la librería y a continuación se explica cada elemento:

- Respecto a las variables utilizadas:
  - ***define INF 10000***. Se ha elegido que el valor de referencia (∞) sea 10000. Para todos los valores superiores se considera que el coste supera el límite permitido y por lo tanto que *TSP* ha encontrado una ruta que es imposible de realizar.

  - ***define RCL_QUALITY 10***. Calidad del criterio de la selección de la RCL. Es un porcentaje con el cual *GRASP* puede seleccionar elementos que no se encuentran más lejos que un % del mejor elemento.
  - ***define IMPROVED_ITERATION 1000***. Se utiliza para especificar el número de iteraciones permitidas sin realizar mejoras. Cuando *GRASP* encuentra una solución mejor que la actual, se resetea el contador de éste parámetro a 0. Para cada solución encontrada se aumenta el contador y si se llega al límite sin encontrar una, el algoritmo se detiene.
  - ***define GRASP_MAX_ITERATION 1000***. Se utiliza para seleccionar el número máximo de iteraciones permitido. Si se alcanza éste número de iteraciones, el algoritmo se detiene y devuelve la mejor solución hasta el momento.
  - ***define LOCAL_SEARCH_MAX_ITERATION 100***. Máximo de iteraciones permitidas para el algoritmo *Local Search*.
  - ***number_of_targets***. Número de puntos objetivo.
  - ***loop_cost***. Coste del camino que calcula *TSP*, el cual si es necesario cambia su valor debido a las disintas itraciones que se realizan en la función *TSP_math*.
  - ***cost_reduction***. Para poder trabajar con costes menores y para que el algoritmo *GRASP* tenga un valor de referencia *INF* más sencillo, se ha elegido reducir el coste.
  - ***matriz_costes***. Esta es la matriz con los costes que *SFF* ha calculado.

- Respecto a las funciones utilizadas:

  - ***TSPhandler***. Este es el constructor de la clase, recibe como parámetros la matriz de costes que *SFF* ha calculado. De esta matriz de costes se realiza una copia sobre la variable *matriz_costes* y se extrae el número de puntos objetivo.
  - ***TSP_math***. Función que engloba todo el funcionamiento del algoritmo *TSP* con el flujo que se ha comentado previamente en 4.4.3. Recibe como parámetro las referencias de *path* y *total_sol_cost* para que una vez calculadas se puedan utilizar desde fuera de la clase.

4.5 Bloque *RRT-MO*: implementación del algoritmo *RRT-MO*

El objetivo de la implementación del algoritmo *RRT-MO* es el de poder tener un planificador probabilístico con el que comparar *SFF*. Por ello, es necesario implementar una lógica que consiga que se pueda utilizar el algoritmo *RRT* original en situaciones multi-objetivo, es decir, tal y como pasa con *SFF*, visitar más de un punto objetivo para cumplir la misión.

Como en este caso no se obtiene un entramado de árboles y por lo tanto no se obtiene una matriz con los costes asociados a cada una de las rutas entre los árboles que se han conectado, no se puede utilizar *TSP* después del algoritmo *RRT-MO*. Por ello, simplemente se construye el *roadmap* a medida que se avanza y se van visitando nuevos puntos objetivo.

Para que el algoritmo *RRT-MO* avance entre los diversos puntos objetivo, se han decidido utilizar dos técnicas:

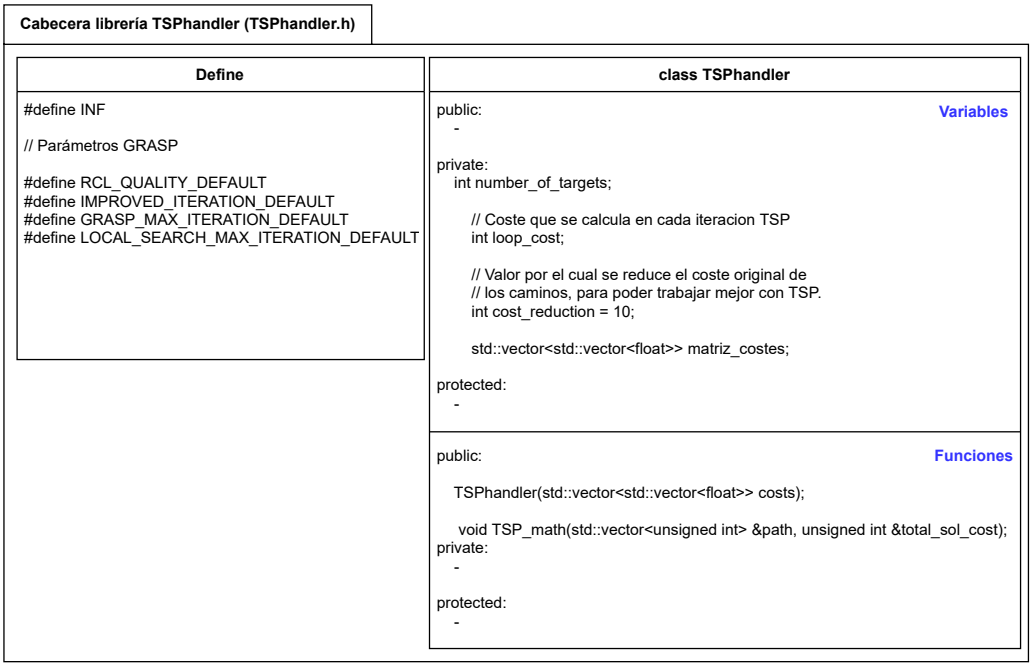


Figura 4.18: Organización de la librería *TSPhandler*. Se pueden observar las variables y funciones implementadas.

- Una en la que los puntos objetivo se vayan visitando de forma aleatoria, es decir, que el siguiente punto objetivo a visitar sea elegido al azar (modo *RANDOM*).
- Otra en la que los puntos objetivo se visiten por proximidad, el vecino más cercano en ese momento será al que se le realice la próxima visita (modo *PROXIMITY*).

Por lo tanto, se puede observar como en el segundo caso, cuando se visite el último nodo y por el hecho de que se tiene que volver al punto inicial para cumplir las premisas del algoritmo *TSP*, es muy probable que se tenga que recorrer una gran cantidad de espacio. Aunque el recorrido hasta el último punto por descubrir se haya hecho de forma eficiente ya que se han ido visitando cada vez los nodos por proximidad, seguramente el último esté alejado del nodo inicial y por ello el coste del camino sea bastante elevado.

Sin embargo, para el primer caso, debido a que los nodos se visitan aleatoriamente, puede ser que en algunas de las combinaciones que se realicen se obtenga un coste inferior al que se obtendría con el segundo caso y en algunos casos puede ser que hasta incluso se aproxime al coste que encuentra *TSP* después de aplicar *SFF*.

4.5.1 Organización del planificador *RRT-MO*

Respecto a la organización del planificador, se ha seguido la misma estructura que en el caso de *SFF*, tal y como se ha comentado en 4.3.1. La clase vuelve a ser una copia del planificador ARA, ubicado en *ARAplanner.h*.

Las variables y funciones necesarias se introducen en la sección privada de la clase para dejar paso a la función *replan*, la única que se utiliza desde fuera del planificador, ubicada en la sección pública.

En la figura 4.19 se observa la organización tal y como se acaba de comentar.

class RRTplanner : public SBPLplanner

Variables

```
public:
  -
protected:
  -
private:
  struct algorithm_params params;  // struct algorithm_params (utilsRRTSFF.h)

  std::shared_ptr<utilsRRTSFF> RRTutils; // Utils

  // Variables auxiliares
  int index; // Punto objetivo seleccionado al azar en cada iteración

  int index_goal; // Para cambiar el índice entre vectores de puntos objetivo a medida que se van descubriendo
  int index_goal_aux; // Auxiliar antes de asignar el valor definitivo a index_goal

  vector<RRTSFFNode> list_of_targets_aux;
```

Funciones

```
public:
  virtual int replan_RRT(std::vector<std::vector<std::vector<int>>> &Pij, std::vector<std::vector<int>> &costs,
  std::vector<int> &path_IDs, unsigned int &total_sol_cost);
protected:
  -
private:
  // Algoritmo RRT
  void RRT_Algorithm(std::vector<std::vector<std::vector<int>>> &Pij, std::vector<std::vector<int>> &costs);

  // Funciones del algoritmo
  RRTSFFNode create_qrand(RRTSFFNode q_goal);
  RRTSFFNode find_qnear(tree<RRTSFFNode> &tr, RRTSFFNode q_rand);
  RRTSFFNode find_qnew(RRTSFFNode q_near, RRTSFFNode q_rand);

  // Imprimir modo de búsqueda (por proximidad o aleatoria)
  std::string SearchModeToStr(search_type_RRT search_mode);
```

Figura 4.19: Organización de la clase *RRTplanner*. Se pueden observar las variables y funciones implementadas.

4.5.2 Diagramas de flujo del planificador *RRT-MO*

En las figuras 4.20 y 4.21, se presentan los diagramas de flujo de las dos funciones principales del planificador, *replan_RRT* y *RRT_algorithm*. De la misma forma en la que se ha hecho con *SFF*, dentro de la función *replan* se puede encontrar toda la estructura que alberga tanto la posibilidad de ejecutar el programa una sola vez como utilizarlo desde un programa externo. Por otro lado, también se encuentra la posibilidad de usar los modos de test para poder realizar pruebas y así poder compararlo con *SFF*.

Respecto a la función *RRT_algorithm*, se puede observar que a parte de implementar el algoritmo descrito por los autores y presentado en los pseudocódigos 2 y 3, se realizan una serie de acciones tales como:

- Presentar los tiempos de cada una de las secciones del algoritmo, acción muy útil para el apartado 5 de resultados a la hora de calcular tiempos de ejecución.
- Representar en figuras todo el proceso realizado.
- Limpiar variables, las cuales, tal y como pasa con la función *replan_SFF*, se desactivan a la hora de utilizar el algoritmo de forma habitual desde un programa

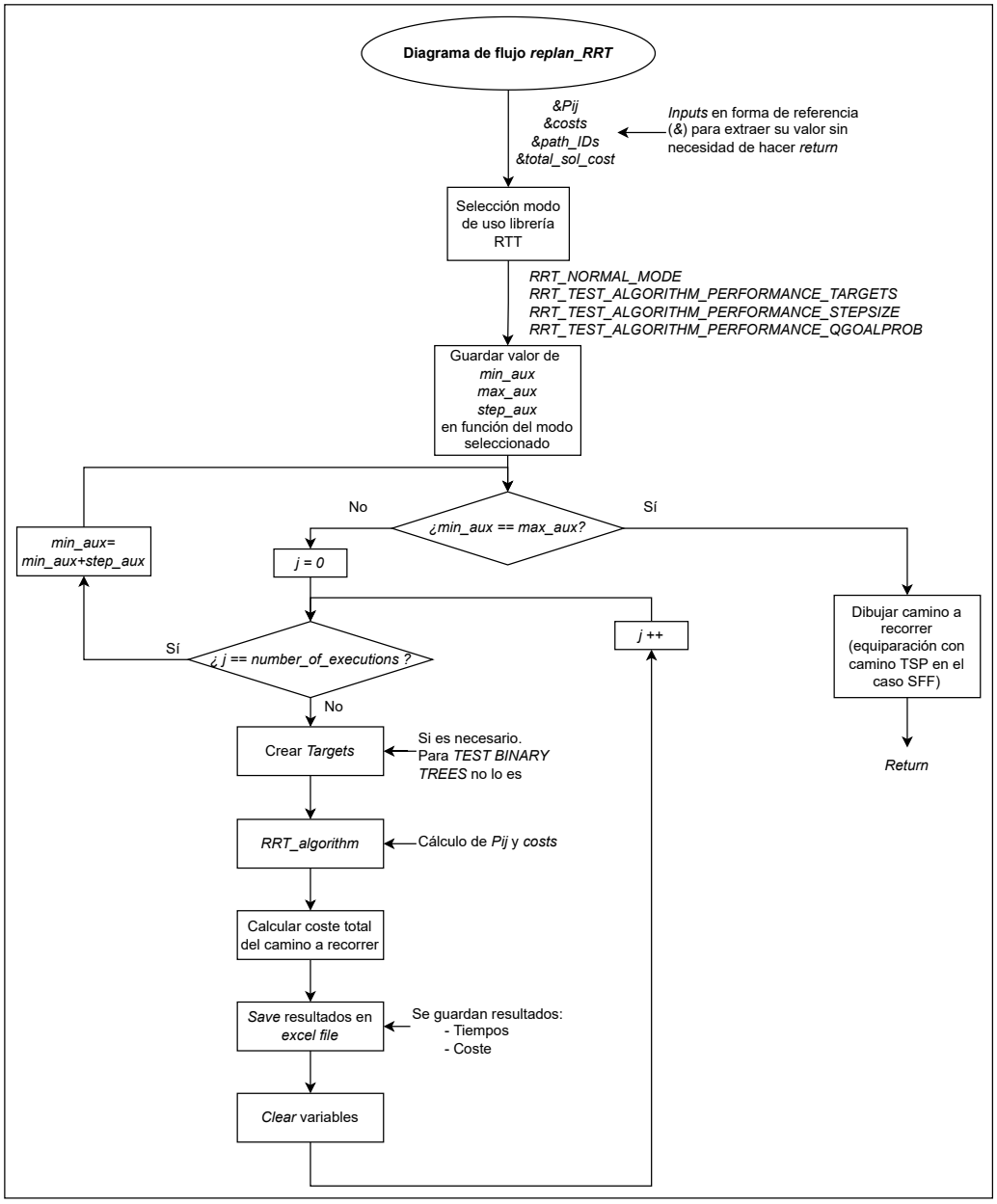


Figura 4.20: Diagrama de flujo de la función *replan_RRT*.

externo.

4.5.3 Definición de las variables utilizadas y contenido de las funciones del planificador

Refiriéndonos a la figura 4.19, a continuación se hace una descripción breve respecto a las variables y funciones declaradas en la clase *RRTplanner*.

- Respecto a las variables utilizadas:

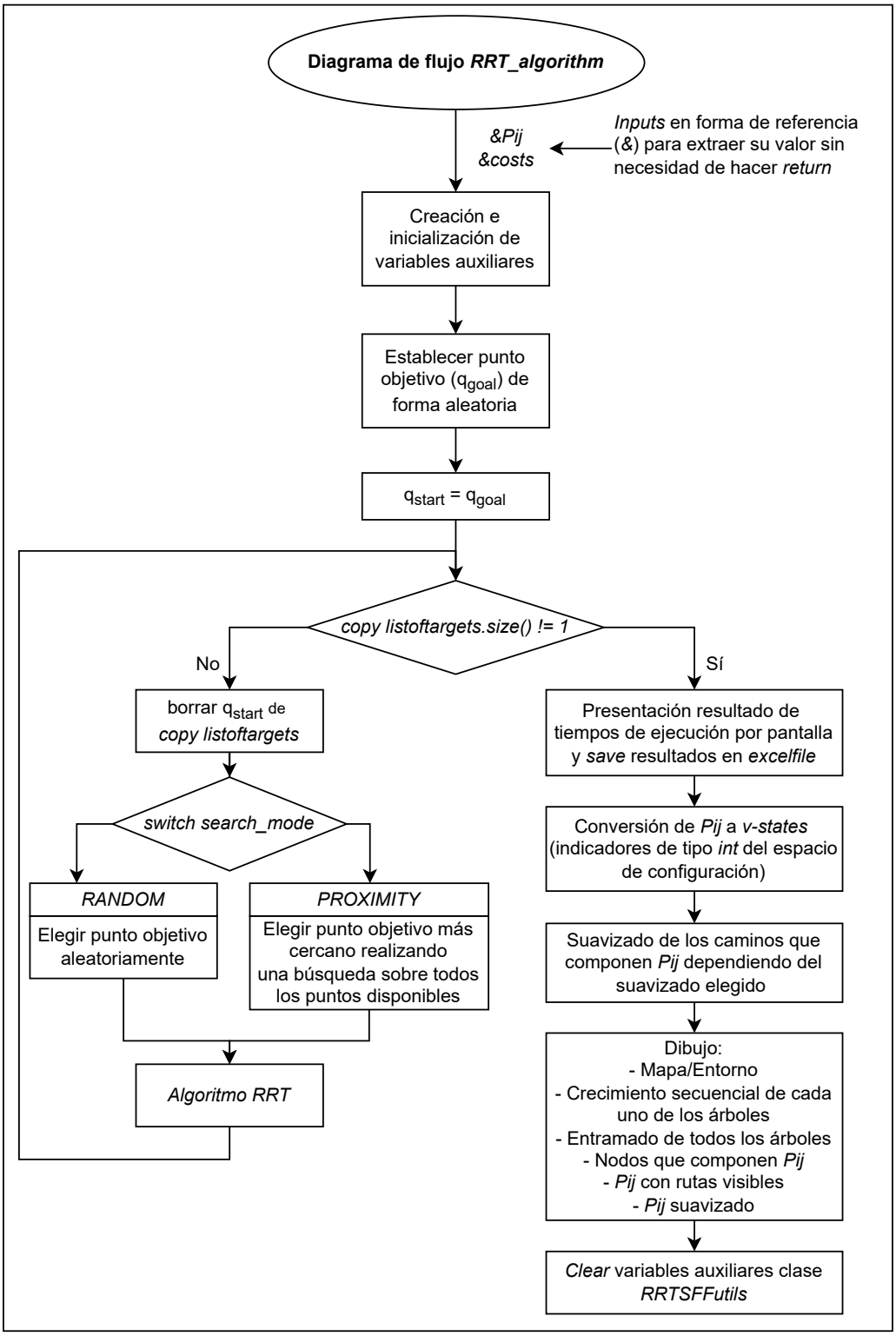


Figura 4.21: Ejemplo de diagrama de flujo.

  - ***params***. Contiene los parámetros principales del algoritmo *RRT-MO*: *stepsize*, *qogalprob* y *searchmode*, entre otras variables importantes para inicializar el algoritmo.
  - ***RRTutils***. Objeto de la clase *utilsRRTSFF* (explicada en el apartado 4.6) que nos permite utilizar todas las funciones auxiliares creadas.
  - ***index***. Este es el punto objetivo que, en cada iteración, en el caso de utilizar el modo de búsqueda aleatorio, se selecciona al azar.
  - ***index_goal***, ***index_goal_aux***. Se utilizan para trabajar entre los índices de los nodos restantes que aún quedan por visitar respecto a todos los nodos originales.
  - ***list_of_targets_aux***. Este es el vector de puntos objetivo auxiliar que se utiliza para saber cuántos puntos objetivo aún quedan por visitar. Cuando este vector sea de tamaño 1 significa que ya se han visitado todos los nodos y que por lo tanto, la búsqueda ha finalizado.

- Respecto a las funciones utilizadas:

  - ***replan_RRT***. Función principal que se utiliza para replanificar y por lo tanto, para hacer uso del algoritmo *RRT-MO*. El detalle de las operaciones que se llevan a cabo en su interior se ha detallado en el apartado 4.5.2.
  - ***RRT_Algorithm***. Algoritmo *RRT* modificado para trabajar con una serie de puntos objetivo en dos modos de funcionamiento distintos: por proximidad y por aleatoriedad. El funcionamiento detallado de esta función se ha explicado en el apartado 4.5.
  - ***create_qrand***. Correspondiente a la línea 5 del pseudocódigo 2. Esta función es la encargada de generar un punto aleatorio dentro del espacio de configuración libre de obstáculos. Además, tal y como menciona el algoritmo *RRT*, con una probabilidad *qgoalprob*, se establece que el punto generado sea el punto objetivo. De esta forma, con cierta probabilidad se ayuda al algoritmo a avanzar más rápido hacia el punto objetivo.
  - ***find_qnear***. Función correspondiente a la línea 2 del pseudocódigo 3. Se encarga de realizar la búsqueda del nodo más cercano sobre el árbol respecto al nodo *q_rand* generado en la función anterior. Para ello, se recorre el árbol con fuerza bruta gracias a la librería *tree.hh* y al modo de recorrido *pre_order_iterator*.

    Cabe mencionar que en este punto se podrían haber tenido en cuenta los árboles binarios utilizados para el algoritmo *SFF* e incluirlos en esta función para que la búsqueda fuese más eficiente. Para simplificar y debido a que se encuentra que la búsqueda a realizar en cada iteración no es lo suficientemente costosa, se ha decidido no incluir los árboles binarios y dejar la búsqueda por fuerza bruta desarrollada gracias a la librería *tree.hh*.
  - ***find_qnew***. Función que se encarga de encontrar el nuevo punto que se añadirá al árbol. Tal y como se explica en el algoritmo *RRT*, se avanza sobre la línea recta que une q_{near} con q_{rand} una distancia *stepsizepixel*. Si esta distancia se puede recorrer sin encontrarse con ningún obstáculo, se puede guardar el punto q_{new} para posteriormente añadirlo al árbol.

4.6 Bloque *RRTSFFutils*: librería auxiliar de los planificadores *RRT-MO* y *SFF*

En la librería *utilsRRTSFF* se agrupan las funciones comunes entre los algoritmos *SFF* y *RRT-MO*. De esta forma se ahorra duplicar contenido y de una forma simple y clara se pueden acceder a funciones que abarcan desde cálculo de distancias hasta asistencia a la hora de realizar las representaciones una vez concluye la ejecución de los algoritmos *SFF* y *RRT-MO*.

4.6.1 Organización de la librería

Respecto a la organización de esta librería auxiliar, ésta se puede observar en la figura 4.22.

Se han creado diversos enumerados para definir distintos tipos de variables propias que se han utilizado durante la implementación del algoritmo, distintas estructuras las cuales contienen variables agrupadas y las clases *RRTSFFNode* y *utilsRRTSFF* las cuales contienen respectivamente toda la información acerca de la mínima unidad de información, un nodo. Además, también contiene toda la información necesaria respecto a las variables y funciones comunes entre los algoritmos *SFF* y *RRT-MO*.

En la figura 4.22 se observa la organización de la librería.

4.6.2 Definición de variables utilizadas y contenido de las funciones

- **Enumerados**. Los enumerados se utilizan para crear tipos de variables personalizadas y así poder trabajar mejor con ellas. A continuación se mencionan las que se han utilizado en la librería *utilsRRTSFF*.
  - ***mode_used***. Utilizado para recopilar los distintos modos de uso que se han creado a la hora de utilizar los algoritmos *SFF* y *RRT*. Cada uno de ellos se corresponde con una sección del archivo *params_RRTSFF.ini*. El valor del modo seleccionado se almacena en la variable *mode* que se encuentra dentro de la *algorithm_params*. De esta forma, siempre y cuando sea necesario, se puede utilizar para realizar distintas operaciones a lo largo del programa.
  - ***search_type_RRT***. Con este enumerado se define el tipo de búsqueda que se puede realizar con el algoritmo *RRT*. Puede ser por proximidad (*PROXIMITY_SEARCH*) o realizada de forma que los puntos objetivo se escojan aleatoriamente (*RANDOM_SEARCH*).
  - ***algorithm_type***. Enumerado para diferenciar el algoritmo que se está utilizando, *RRT* o *SFF*. Los valores del enumerado son *RRT_ALGORITHM* y *SFF_ALGORITHM*.
  - ***binary_tree_type***. Enumerado para saber si se está utilizando la librería *tree.hh* o los árboles binarios. Los valores del enumerado son *TREEHH*, *KD_TREE*, *COVER_TREE*, *QUAD_TREE* y *PH_TREE*
  - ***smooth_type***. Enumerado para saber que tipo de algoritmo de suavizado se aplica una vez finalicen y se tengan las rutas entre los caminos. Los valores del enumerado son *NOTSMOOTHED_PATH_TYPE*, *CHAIKIN_SMOOTH*, *SFF_SMOOTH*, *SFF_SMOOTH_ISAAC* y *SFF_SMOOTH_ISAAC_CHAIKIN*.

- **Estructuras**. Con las *structs* se han organizado variables que guardan relación entre ellas, de esta forma es más simple gestionar su uso desde los planificadores *RRT* y *SFF* o desde la misma librería. A continuación se muestran las que se han creado:

  - ***params_to_draw***. Parámetros comunes de *RRT* y *SFF* para el dibujo. Estructura que almacena las variables necesarias para posteriormente dibujar cada una de las figuras necesarias a la hora de representar el crecimiento de los árboles, la creación de los caminos y el recorrido a seguir el cual calcula *TSP*. Las variables son las siguientes:
    - ***Tree_Vector***: es una copia de los árboles que se crean durante la ejecución del algoritmo.
    - ***Trees_growth***: en este vector se almacenan todos los nodos que se han ido añadiendo a los árboles, independientemente de cuál sea al que pertenecen.
    - ***Trees_order***: se guarda el correspondiente índice de cada árbol para saber a cuál corresponde cada uno de los nodos almacenados en *Trees_growth*.
  - ***initRRT***. Estructura que sirve para almacenar todos los parámetros de inicialización necesarios para que el algoritmo *RRT* funcione correctamente. Su contenido es:
    - ***Stepsize***, ***qgoalprob***, ***searchmode***: *inputs* principales del algoritmo.
    - ***Min_targets***, ***max_targets***, ***step_targets***: valores mínimo, máximo y de paso para modificar el número de puntos objetivo a utilizar cuando se está utilizando el modo de test que lo requiera.
    - ***Min_stepsize***, ***max_stepsize***, ***step_stepsize***: valores mínimo, máximo y de paso para modificar el valor de la variable *stepsize* del algoritmo *RRT* a utilizar cuando se está utilizando el modo de test que lo requiera.
    - ***Min_qgoal***, ***max_qgoal***, ***step_qgoal***: valores mínimo, máximo y de paso para modificar el valor de la probabilidad de que el siguiente punto q_{rand} se seleccione como el punto objetivo cuando se está utilizando el modo de test que lo requiera.
    - ***Number_of_executions***: número de ejecuciones que se deben realizar en cada iteración para posteriormente, poder realizar la media y así sacar un valor que represente a cada iteración con sus respectivos parámetros.
  - ***struct initSFF***. Estructura que sirve para almacenar todos los parámetros de inicialización necesarios para que el algoritmo *SFF* funcione correctamente.
    - ***k***, ***d_tree***, ***R***: *inputs* principales del algoritmo.
    - ***Min_targets***, ***max_targets***, ***step_targets***: realizan la misma función que en el caso de la *struct init_RRT*.
    - ***Min_R***, ***max_R***, ***step_R***: valores mínimo, máximo y de paso para modificar el valor de la variable *R* del algoritmo *SFF* a utilizar cuando se está utilizando el modo de test que lo requiera.
    - ***Min_k***, ***max_k***, ***step_k***: valores mínimo, máximo y de paso para modificar el valor de la variable *k* del algoritmo *SFF* a utilizar cuando se está utilizando el modo de test que lo requiera.

    - ***Number_of_executions***: mismo parámetro que en el caso de *init_RRT*.

  - ***algorithm_params***. Con esta estructura se desean recopilar todos los parámetros comunes y necesarios a la hora de utilizar los algoritmos *RRT* y *SFF*, de esta forma con una misma estructura se pueden utilizar los dos algoritmos. Las variables que contiene son las siguientes:

    - ***mode***: modo utilizado entre todas las opciones presentadas por el enumerado *mode_used*.
    - ***Seed***: semilla que se utiliza a la hora de ejecutar el algoritmo o el test que se desee realizar. Así, si se quiere repetir alguna prueba, se puede introducir la misma semilla.
    - ***Number_of_targets***: número de puntos objetivos que se deben generar.
    - ***Min_dist_between_targets***: distancia mínima que debe existir entre los puntos objetivo a la hora de generarlos aleatoriamente, así se asegura que no se creen puntos muy cercanos.
    - ***List_of_targets***: vector dónde se almacenan las coordenadas de cada uno de los puntos objetivo.
    - ***initRRT RRT***: estructura *RRT* que almacena todos los parámetros de inicialización del algoritmo *RRT*.
    - ***initSFF SFF***: estructura *SFF* que almacena todos los parámetros de inicialización del algoritmo *SFF*.
    - ***path***: vector dónde se almacenan los índices de los puntos objetivo que se consiguen gracias a la ejecución del algoritmo *TSP*.
    - ***total_sol_cost***: coste total del recorrido que se debe realizar una vez lo ha calculado el algoritmo *TSP*.

- ***Class RRTSFFNode***. Clase dónde se define la estructura que tiene un nodo, es decir, sus coordenadas *X* e *Y*. Con el constructor se define el valor de las coordenadas, de esta forma se pueden crear vectores los cuales apunten a esta unidad de información para posteriormente ir añadiendo nodos a dicho vector.

- ***Class utilsRRTSFF***. Clase que pretende recopilar todas las funciones comunes y necesarias para que los algoritmos *RRT* y *SFF* funcionen correctamente. A continuación se menciona el uso que se le da a cada una de sus funciones.

  - ***utilsRRTSFF***. Constructor en el que se inicializan las variables esenciales tales como los parámetros del entorno y de dibujo.
  - ***Read_ini_file***. Función encargada de leer el archivo de configuración *params_RRTSFF.ini* y, en función del algoritmo y modo de uso que se esté utilizando, guardar las variables necesarias para que el algoritmo pueda ejecutarse.
  - ***Line_bresenham_stepsizepixel*** (destinado a usarse con RRT) y ***Line_bresenham*** (destinado a usarse con *SFF*). Las funciones *Line_bresenham* son las encargadas de utilizar el algoritmo *Bresenham* [92]. Cada función está orientada a ser usada por un algoritmo u otro.

En el caso de *RRT*, la función tiene como parámetro principal a *step_size_pixel*, de esta forma, a medida que se avanza sobre al línea que une los dos puntos introducidos, se tiene en cuenta la distancia *step_size_pixel*. Esta distancia se usa para detenerse si no ha encontrado ningún obstáculo, de esta forma se asegura de cumplir las condiciones de expansión de *RRT*.

En el caso de *SFF*, se utiliza para conocer si existe camino libre entre los dos puntos introducidos por parámetro. Además, se puede seleccionar si se desea guardar o no el camino recorrido en el vector que introduce en forma de referencia. Notar que esta función también se utiliza en el algoritmo *RRT* para conocer si existe camino libre entre dos puntos.

- ***Dist euclidean***. Distancia euclidiana [93] entre dos puntos.
  - Fórmula: $d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
  - Código: *dist = sqrt((abs(x2-x1)*abs(x2-x1))+(abs(y2-y1)*abs(y2-y1)));*
- ***Dist without sqrt***. Distancia euclidiana sin utilizar las raíces cuadradas.
  - Fórmula: $d(P_1, P_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2$
  - Código:
    ```
    x_dif = x2 - x1;
    y_dif = y2 - y1;
    dist = (x_dif*x_dif + y_dif*y_dif);
    ```
- ***Dist manhattan***. Distancia de manhattan [94] entre dos puntos.
  - Fórmula: $d(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|$
  - Código: dist = (abs(x2-x1)+abs(y2-y1));
- ***Create random targets***. Con esta función se pretende generar un número aleatorio de puntos objetivo que se encuentren separados una distancia mínima. De esta forma, se pueden generar casos de prueba con suma facilidad ya que se pueden generar puntos objetivo automáticamente.

  Para poder generar los puntos se utiliza el árbol binario *Cover Tree*, de esta forma para cada nuevo punto que se desea añadir a la lista de puntos objetivo, se comprueba que el vecino más cercano se encuentre como mínimo a la distancia requerida.
- ***Save nodes to the root***. Función utilizada a la hora de generar los caminos hasta las raíces de los árboles una vez que dos de ellos se encuentran. Se recorre el árbol gracias a la librería *tree.hh* para ir encontrando los padres de cada uno de los nodos, hasta llegar al nodo raíz.
- ***Is inside***. Función utilizada para generar un nuevo punto aleatorio. Se comprueba si el punto generado se encuentra dentro del radio R que designa el algoritmo *SFF*. Simplemente se realiza una comparación de distancias entre el punto introducido a la función y el radio, de esta forma se sabe si el punto se encuentra en el interior del círculo o no.
- ***Convert vpaths to vstates***. Para poder crear el *output* del algoritmo *SFF* se utiliza esta función. Convierte la matriz P_{ij} de nodos de la clase *RRTSFFNode* a estados de tipo *int* del espacio de configuración.

- ***Smooth handler***. Destinada a gestionar las funciones encargadas de realizar los suavizados de los caminos. En función del tipo de suavizado seleccionado en el fichero de configuración *params_RRTSFF.ini*, se utiliza un tipo de suavizado u otro, y en consecuencia se seleccionan las funciones adecuadas.
- ***Set save figures flag***, ***Set figure title manual***, ***Set draw tsp steps flag manual***. Funciones auxiliares que se encargan de establecer si se debe guardar la siguiente figura o no, de guardar el nombre de la siguiente figura y de establecer si se debe guardar una imagen por cada paso que realice el algoritmo *TSP*, respectivamente.
- ***Set things to draw***. Una vez finaliza la ejecución de los algoritmos, se utiliza esta función para guardar la información necesaria de las variables correspondientes para posteriormente, poder guardar las imágenes que se referencian a la ejecución que se acaba de realizar.
- ***Draw targets***, ***Draw map***. Funciones con las cuales se dibuja el entorno y los puntos objetivo situados sobre él.
- ***Draw trees growth SFF***. Función encargada de dibujar el crecimiento de los árboles. Se recorre el vector *Trees_growth* y en función del árbol al que corresponda, el cuál se comprueba con el vector *tree order*, se dibuja el nodo con el color del árbol correspondiente, conectándolo con el padre para así formar una rama. A medida que se avanza, se van guardando imágenes para posteriormente observar la evolución de los árboles.
- ***Draw trees growth RRT***. Funciona de forma similar al caso de *SFF*, pero en este caso es más sencillo debido a que se dibuja el crecimiento de cada árbol de forma secuencial.
- ***Draw pij nodes***. Función destinada a dibujar todos los nodos de la matriz P_{ij}, de esta forma se ven claramente los nodos correspondientes a esta matriz. Posteriormente, sobre estos nodos, se aplica el suavizado.
- ***Draw pij x***. Función encargada de recorrer P_{ij} para dibujar los nodos que contiene. Se utiliza para representar P_{ij} antes y después de suavizar las rutas entre los puntos objetivo.
- ***Draw tsp path***. Función que dibuja la ruta que *TSP* ha calculado. Se representa de color rojo, pero si hay caminos que se han recorrido más de una vez debido al problema respecto a los nodos aislados explicado en el apartado 4.4.2, se representa de color azul. De esta forma, se puede observar visualmente el recorrido que se debe realizar y queda representado cuando se recorre una misma ruta más de una vez.
- ***Test binary trees***. Función para ejecutar el modo de test utilizado para probar los árboles binarios de forma individual. Para cada uno de los árboles, se realizan una serie de acciones:
  - Crear los árboles dinámicamente, añadiendo los nodos uno a uno en función de los nodos seleccionados.
  - Realizar una serie de búsquedas del vecino más cercano.

  Con esta información se puede analizar cada uno de los árboles para descubrir cuál tiene mejores prestaciones desde un punto de vista individual.

- ***Clear figure manual***. Función utilizada para limpiar el lienzo o *figure* que se está utilizando y así poder proseguir con otras representaciones.
- ***Create path from nodes***. Cada vez que se desea construir un camino dados una serie de nodos, se utiliza esta función. Así se construye la matriz P_{ij} que se utiliza para representarla de forma contínua. En su interior, se recorre el vector insertado por parámetro para así aplicar el algoritmo *bresenham* entre cada uno de ellos.
- ***Chaikin algorithm step***. Función que lleva a cabo cada paso del algoritmo *Chaikin*, es decir, cada creación de los puntos intermedios Q_x, Q_y, R_x y R_y.
- ***Chaikin algorithm***. Función que se encarga de llamar a *chaikin_algorithm_step* de forma recursiva el número de veces (*chaikin_iterations*) que se quiera ejecutar el algoritmo *Chaikin*.
- ***Path smoothing sff***. Suavizado que se presenta en la publicación del algoritmo *SFF*. Se ha implementado tal y como se presenta en el pseudocódigo 6.
- ***Path smoothing sff Isaac version***. Dado el simple algoritmo de suavizado que se presenta en la publicación de *SFF*, se ha decidido realizar una modificación para convertirlo en un algoritmo de suavizado que se encargue de estirar el recorrido lo máximo posible. De esta forma se optimizan las rutas que se deben realizar, aunque éstas sigan quedando con zonas abruptas y poco suavizadas a simple vista.

  Cabe decir que utiliza recursividad, y que cada vez que se eliminan nodos, se empieza de nuevo analizando la ruta desde su nodo inicial.

  Este algoritmo modificado se puede observar en el pseudocódigo 7.
- ***Recursive path smoothing sff isaac version***. Función utilizada para inicializar la recursividad de la función *Path smoothing sff isaac version*.
- ***Create trees colors***. En función de los árboles que se deban dibujar, esta función se encarga de asignar un color aleatorio a cada uno de los árboles para posteriormente poder dibujarlos correctamente.
- ***Inflate point***. Función encargada de inflar los puntos objetivo una serie de píxeles para que a simple vista se vean más grandes y fáciles de reconocer.
- ***Draw targets specific color***. De forma predeterminada, los puntos objetivo se dibujan de color rojo. Gracias a esta función se pueden dibujar del color que se desee. Sobre todo se utiliza para intercambiar el color de los nodos dependiendo si son *start* o *goal* en la función *Draw trees growth RRT*.

4.7 Algoritmos de suavizado

En este apartado se lleva a cabo la explicación detallada de cada uno de los algoritmos de suavizado que se han añadido a la librería *RRTSFFutils* para que se pueda llevar a cabo un suavizado del *roadmap* P_{ij} que se obtiene tras aplicar los algoritmos *SFF* o *RRT-MO*.

4.7.1 Suavizado implementado en la publicación de *SFF*

Este suavizado se presenta en la publicación del algoritmo *SFF*, su funcionamiento se puede observar en el pseudocódigo 6. Funciona gracias a una ventana de actuación

llamada w. Empezando desde la posición $w+1$ del vector introducido, en cada iteración se comprueba si es posible conectar las posiciones $q_i - q$ y $q_i + w$ con una línea recta. Si es así, se elimina la posición q_i y se avanza una distancia w sobre el vector de puntos. Si no es posible conectarlos, se sigue avanzando sobre el vector.

Algoritmo 6: Suavizado presentado en el documento del planificador *SFF*.

Input:

P	$P = (q_1, ..., q_m), q_i \in C_{free}$ de m configuraciones
w	Ventana de actuación o *window size*

Output:

P	Camino suavizado

```
1 function smooth_SFF_original (P)
2     for i = w + 1 : m − w do
3         if esPosibleConectar(q_{i−w}, q_{i+w}) then
4             eliminar q_i de P;
5             i = i + w;
6     return P
```

4.7.2 Modificación del algoritmo para estirar los caminos

Respecto al suavizado anterior, se ha decidido realizarle una modificación para conseguir que los caminos se estiren lo máximo posible. En vez de eliminar solamente el punto q_i cuando se consigue conectar $q_i - q$ con $q_i + w$, se ha decidido eliminar todos los puntos comprendidos entre $q_i - q + 1$ y $q_i + w - 1$, de esta forma se consiguen rutas lo más optimizadas posibles. El pseudocódigo de la implementación realizada se puede observar en 7.

De todas formas, cabe comentar que en ninguno de los dos algoritmos de suavizado comentados se consigue que las rutas sean suaves, es decir, siguen existiendo tramos abruptos.

Por ello, se ha decidido utilizar el algoritmo *Chaikin* (explicado en 4.7.3) y fusionarlo con el suavizado que se ha implementado.

4.7.3 Algoritmo de Chaikin

El algoritmo de Chaikin [95] se encarga de generar nuevos puntos entre la recta que une dos nodos para así, a medida que se itera sobre un conjunto de puntos y por lo tanto sobre un conjunto de segmentos, la ruta final intercambia las zonas abruptas por curvadas, quedando así suavizada.

Dado un conjunto de n puntos $\{P_0, P_1, ..., P_{n-1}\}$ se definen Q_i y R_i como los puntos a $\frac{1}{4}$ y a $\frac{3}{4}$ de distancia entre los dos puntos que forman el segmento $P_i\bar{P}_{i+1}$. En la ecuación 4.3 se pueden observar las expresiones que proporcionan los puntos Q_i y R_i. Por otro lado, en la figura 4.23, se observa un ejemplo visual de como actúa el algoritmo *Chaikin* tras realizar tres iteraciones, es decir, tras generar tres veces los puntos Q_i y R_i sobre todos los segmentos que componen una ruta.

Algoritmo 7: Suavizado presentado en el documento del planificador *SFF* modificado para estirar los caminos lo máximo posible.

Input:
- P — $P = (q_1, ..., q_m), q_i \in C_{free}$ de m configuraciones
- w — Ventana de actuación o *window size*
- *initial_index* — Índice inicial para saber por donde empezar a recorrer el camino P

Output:
- P — Camino suavizado

```
1  function smooth_SFF_Isaac_version (P, initial_index)
2      i = w+initial_index;
3      while i ≤ P.size() - w do
4          if (i + w >P.size() − 1) then
5              break;
6          if esPosibleConectar(q_{i−w}, q_{i+w}) then
7              eliminar desde (q_{i−w}) hasta (q_{i+w}) de P;
8              P =smooth_SFF_Isaac_version(P,initial_index);
9              break;
10         i = i + w;
11         initial_index = i − w;
12     return P
```

$$Q_i = \frac{3}{4}P_i + \frac{1}{4}Pi + 1, \qquad R_i = \frac{1}{4}P_i + \frac{3}{4}P_{i+1} \tag{4.3}$$

En los pseudocódigos 8 y 9 se puede observar la implementación realizada del algoritmo *Chaikin*, correspondientes a las funciones *Chaikin algorithm* y *chaikin_algorithm_step* mencionadas en el apartado 4.6.2. La función *Chaikin algorithm* utiliza recursividad para llamar a la función *chaikin_algorithm_step* el número de iteraciones (*chaikiniter*) que se desee aplicar el algoritmo Chaikin.

Algoritmo 8: Función *chaikin_algorithm*, encargada de ejecutar el algoritmo chaikin un número determinado de iteraciones (*chaikiniter*).

Input:
- *pi* — Camino que se tiene que suavizar
- *chaikiniter* — Iteraciones del algoritmo *Chaikin*

Output:
- *pi* — Camino suavizado

```
1  function chaikin_algorithm (vector pi, initial_index)
2      if chaikiniter == 0 then
3          return pi
4      return chaikin_algorithm(chaikin_algorithm_step(pi), chaikiniter-1)
```

Algoritmo 9: Función *chaikin_algorithm_step*, encargada de ejecutar cada iteración del algoritmo *Chaikin*.

Input:
pi Camino que se tiene que suavizar
Output:
V_{smooth} Camino suavizado

```
1  function chaikin_algorithm_step (pi)
2      for (i = 0  to  pi.size()−1) do
3          Q_x = pi_x[i] * 0,75 + pi_x[i + 1] * 0,25;
4          Q_y = pi_y[i] * 0,75 + pi_y[i + 1] * 0,25;
5          R_x = pi_x[i] * 0,25 + pi_x[i + 1] * 0,75;
6          R_y = pi_y[i] * 0,25 + pi_y[i + 1] * 0,75;
7          Q_vector.push_back(Q_x, Q_y);
8          R_vector.push_back(R_x, R_y);
9          i++;
10     // Añadir puntos a vector resultante comprobando
11     // que no se ha atravesado ningún obstáculo
12     V_smooth.push_back(pi[0]); // Añadir punto inicial
13     for (i = 0 to Q_vector.size()−1) do
14         if (esPosibleConectar(R_vector[i], Q_vector[i + 1])) then
15             V_smooth.push_back(R_vector[i]);
16             V_smooth.push_back(Q_vector[i + 1]);
17     V_smooth.push_back(pi.[size() − 1]); // Añadir punto final
18     return V_smooth
```

4.7.4 Algoritmo de Chaikin junto a la modificación realizada

Se ha querido utilizar el algoritmo de suavizado que mejor se adaptase a la situación que se tiene tras aplicar *SFF* o *RRT*. Debido a la gran cantidad de nodos que se generan y a la mayoría de los casos en los que entre ellos no existe una distancia lo suficientemente grande, se ha decidido primero utilizar el algoritmo mencionado en el apartado 4.7.2, el cual se encarga de estirar al máximo cada una de las rutas y posteriormente utilizar el algoritmo de Chaikin para que las suavice.

4.7.5 Comparación de los algoritmos de suavizado

Se ha querido poner un ejemplo sencillo en el que se observe el resultado del camino resultante una vez se aplican los algoritmos de suavizado implementados. En la figura 4.24 se observa la comparativa con los distintos suavizados implementados:

- En la figura 4.24a, se observa el camino que se debe suavizar con sus respectivos nodos.
- En la figura 4.24b, se observa el suavizado presentado en el documento de *SFF*, explicado en el pseudocódigo 6. En este caso la variable *window size = 1*, por lo que el algoritmo intentará recortar 1 nodo en la medida de lo posible.

- En la figura 4.24c, se observa el algoritmo de estirado implementado. Al contrario de lo que ocurre con el anterior, este suavizado intenta estirar al máximo posible el camino.

- En la figura 4.24d se observa el resultado de aplicar el algoritmo Chaikin con la variable *chaikiniter* = 5

- Finalmente, en la figura 4.24d, se observa el resultado de aplicar el algoritmo implementado (figura 4.24c) juntamente con el algoritmo *Chaikin* (figura 4.24d). Este es el suavizado que consigue un mejor resultado sobre el camino resultante, ya que aprovecha al máximo el espacio y su trazada es suave y sencilla de recorrer. Se ha decidido que éste será el algoritmo que se utilizará durante la realización de las pruebas.

4.8 Bloque de dibujo

Para poder realizar las representaciones que se presentan en el capítulo 5 de resultados, se ha decidido utilizar una librería llamada *matplotlibcpp*, código fuente de la cual se ha extraído de [96]. Se trata de una librería bastante ligera desarrollada en C++, destinada a la creación de imágenes sencillas utilizando comandos básicos de la librería *Matplotlib* de *Python*. No se trata de una traslación completa de la librería original y no se pueden utilizar todas las funciones originales, pero gracias a la simpleza que brinda a la hora de utilizarla, la convierte en una buena librería para llevar a cabo la creación de figuras sencillas.

Con una única cabecera llamada *matplotlibcpp.h*, añadiendo las siguientes líneas de código:

- *#include "matplotlibcpp.h"*

- *namespace plt = matplotlibcpp;*

se pueden utilizar todas la funciones, así que no es necesario compilar ningún archivo adicional.

A continuación se muestran las funciones más relevantes que se han utilizado durante el proyecto:

- ***backend***. Existen diversos tipos de *backend* con los que *matplotlibcpp* puede trabajar, estos se eligen en función del *render* que se quiera utilizar. También depende de las necesidades del usuario en cada tipo de aplicación. En nuestro caso se ha utilizado el *backend Agg* (*Anti-Grain Geometry engine*) debido a que trabaja con figuras estáticas de extensión *.png*, es decir, figuras que solo se pueden visualizar una vez y que no pueden modificarse dinámicamente mientras se encuentran abiertas. En la tabla 4.1, extraída de la documentación de *matplotlibcpp* se observan los diferentes *renders* estáticos que existen.

- ***figure_size***. Se establece el tamaño *X* e *Y* que tiene la figura a realizar. El tamaño se define en píxeles.

backends matplotlibcpp		
Renderer	**Extensión**	**Descripción**
AGG	*png*	raster graphics – high quality images using the Anti-Grain Geometry engine
PDF	*pdf*	*vector graphics – Portable Document Format*
PS	*ps, eps*	*vector graphics – Postscript output*
SVG	*svg*	*vector graphics – Scalable Vector Graphics*
PGF	*pgf, pdf*	*vector graphics – using the pgf package*
Cairo	*png, ps, pdf, svg*	*raster or vector graphics – using the Cairo library*

Tabla 4.1: Diferentes *backends* estáticos que ofrece la librería *matplotlibcpp*.

- ***xlim***, ***ylim***. Límites del valor de las coordenadas *X* e *Y* representadas en el dibujo.
- ***scatter***. Función utilizada para dibujar al mismo tiempo un conjunto de puntos. Introduciéndole dos vectores *x_coords* e *y_coords*, es posible dibujar cada uno de los puntos en una sola sentencia.
- ***title***. Se utiliza para establecer el título de la próxima figura que se guarde.
- ***plot***. Función utilizada para dibujar líneas. Introduciéndole dos vectores *x_coords* e *y_coords* es posible concatenar dichos puntos con líneas.
- ***save***. Una vez se acaban de utilizar las funciones encargadas de representar puntos y líneas sobre la figura en curso, se utiliza esta función para guardar la figura tal y como se encuentre en ese momento.

4.9 Funciones utilizadas de las librerías *vector* y *memory*

Tal y como se ha mencionado en el apartado 4.1.7, durante la realización del proyecto las librerías de C++ *vector* [84] y *memory* [85] han sido de vital importancia. Por ello, se ha querido mencionar cuál es el objetivo de cada una de las librerías y cuáles han sido las funciones más utilizadas.

4.9.1 *Vector*

Un vector es una secuencia de posiciones de memoria que puede cambiar su tamaño. Es muy similar a los *arrays*, normalmente más utilizados que los vectores, con la diferencia que se ha comentado: pueden cambiar su tamaño a lo largo del tiempo. Por ello, los convierte en una herramienta muy útil a la hora de almacenar algún tipo de información que se sabe que a lo largo de la ejecución puede aumentar o disminuir su tamaño.

Cuando se trabaja con vectores es muy común utilizar *iterators*. Un *iterator* es un objeto parecido a un puntero que apunta a una posición dentro de un contenedor de posiciones, en nuestro caso, un vector.

Tanto los árboles como gran cantidad de información que se ha utilizado a lo largo del proyecto, se almacena en vectores.

Las funciones más utilizadas han sido las siguientes:

- ***vector::at***. Devuelve la referencia del elemento de la posición *n* del vector, es decir, el contenido al que haga referencia la posición del vector que se introduzca.

- ***vector::begin***, ***vector::end***. Devuelven el *iterator* de la primera y última posición del vector, respectivamente.
- ***vector::clear***. Elimina todos los elementos del vector, los cuales son destruidos, dejando así las posiciones de memoria que ocupaba libres para que se puedan utilizar en el futuro.
- ***vector::emplace***. Se utiliza para colocar un nuevo elemento en el vector en la posición que se desee.
- ***vector::emplace_back***, ***vector::push_back***. Con estas dos funciones se puede introducir un nuevo elemento justo al final del vector. En el caso de *emplace_back* se puede usar el parámetro *args* para introducir los diversos argumentos que se deseen, en función de las características del vector.

  Por otro lado, en el caso de *push_back*, simplemente se copia el valor del parámetro al nuevo elemento que se añada, sin la posibilidad de utilizar argumentos.
- ***vector::erase***. Se elimina del vector un elemento de una posición en concreto o una serie de ellos dependiendo del rango que se le introduzca.
- ***vector::insert***. Se utiliza para introducir una serie de nuevos elementos tras la posición del vector que se indique.
- ***vector::size***. Devuelve el número de elementos que contenga el vector.

4.9.2 *Memory*

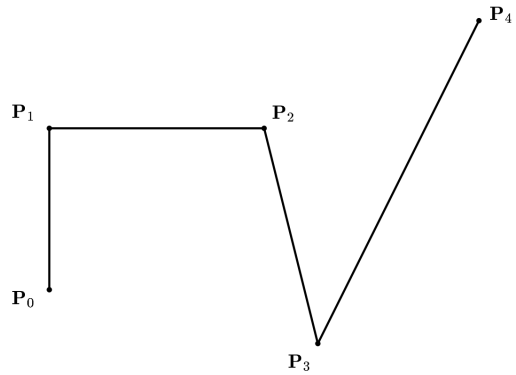
Debido al problema mencionado en 4.1.7 relacionado con el problema de *memory leak*, se ha decidido utilizar esta librería, la cuál permite gestionar de forma dinámica la memoria que se utilice sin tener que preocuparse por liberarla cada vez que ya no se necesite.

La librería *memory* es muy extensa y proporciona las características generales para poder gestionar la memoria de forma dinámica.

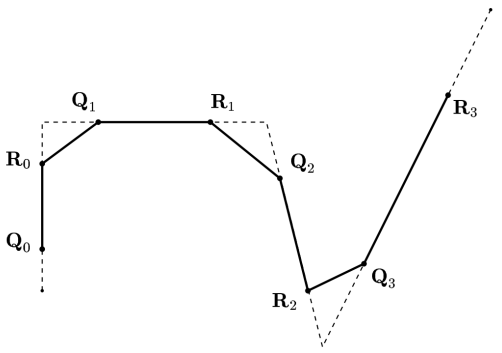
Las dos funciones principales que se han utilizado han sido:

- ***std::shared_ptr***. Sirve para crear el puntero a la dirección de memoria que se desee. Sustituye el uso de "*" a la hora de declarar un puntero.
- ***std::make_shared***. Sirve para asignar y construir un nuevo objeto del tipo *shared_ptr*, el cuál guardará el puntero del objeto recién creado. Sustituye al uso de la sentencia *new* a la hora de crear un nuevo objeto.

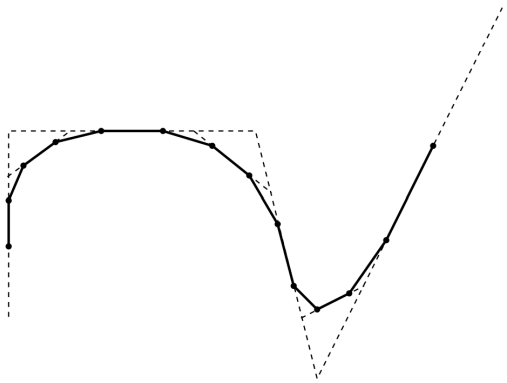
Figura 4.22: Organización de la clase *utilsRRTSFF*. Se pueden observar los enumerados, estructuras, variables y funciones de las clases *RRTSFFNode* y *utilsRRTSFF*.



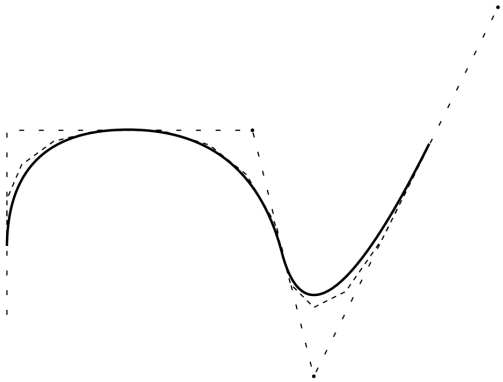
(a) Ruta con 5 puntos.



(b) Primera iteración del algoritmo Chaikin.

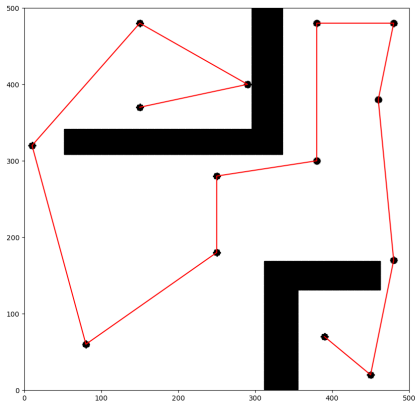


(c) Segunda iteración del algoritmo Chaikin.

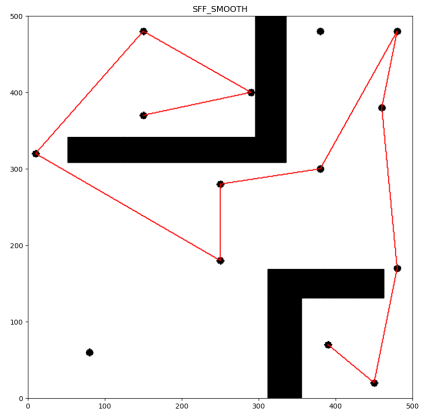


(d) Tercera iteración del algoritmo Chaikin.

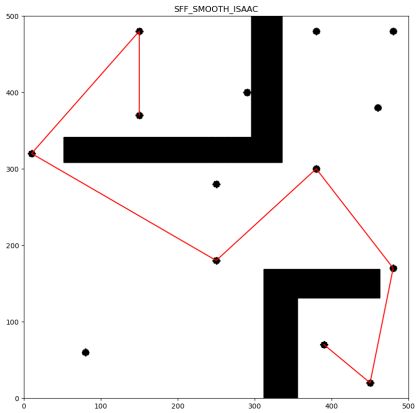
Figura 4.23: Proceso de suavizado del algoritmo Chaikin. Ejemplo de 3 iteraciones sobre una ruta de 5 puntos.



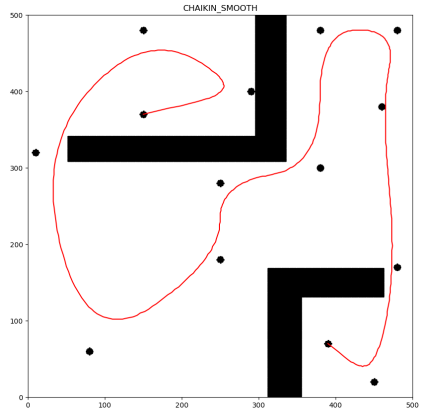
(a) Ruta que se debe suavizar.



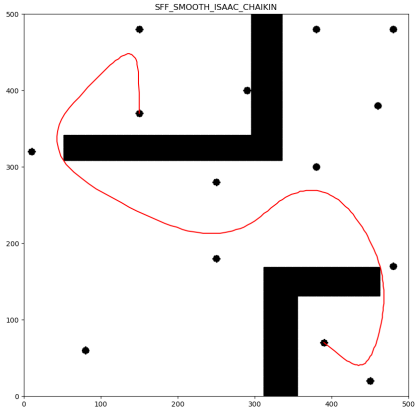
(b) Suavizado *SFF*.



(c) Suavizado *SFF* modificado para estirar el camino.



(d) Suavizado con algoritmo Chaikin.



(e) Combinación del suavizado *SFF* modificado con el algoritmo Chaikin.

Figura 4.24: Comparativa de los algoritmos de suavizado. Se puede observar cada uno de los algoritmos implementados durante la realización del proyecto.

CAPÍTULO 5

INFORME Y ANÁLISIS DE RESULTADOS

En este capítulo se llevan a cabo una serie de pruebas con el fin de:

1. Determinar qué árbol binario se comporta mejor.
2. Conocer qué algoritmo destinado a resolver el problema *TSP* presenta mejores características y, por lo tanto, se elige para resolver el problema.
3. Como parte central de éste apartado de resultados, se muestran los que se han obtenido respecto al análisis realizado a los algoritmos *SFF* y *RRT-MO*.
4. Comparación entre ellos para comprobar cuáles son las fortalezas y debilidades de cada uno.

Es importante mencionar que todas las pruebas se han realizado bajo el mismo computador INTEL® Core™ *i7-4710MQ CPU@2,50GHz x 8/ 8GB RAM*, utilizando únicamente un núcleo.

A modo de interés, en el apéndice A se explica cada uno de los pasos que se deben seguir para instalar el programa implementado y poder realizar las pruebas que se explican durante este apartado.

5.1 Selección del árbol binario a utilizar

Para elegir el árbol binario a utilizar se ha realizado una comparación individual entre ellos. Se mide el tiempo de creación de cada árbol y el tiempo que se tarda en realizar las búsquedas *NN*.

5.1.1 Comparación individual entre árboles binarios

Para poder realizar esta prueba se ha utilizado el modo de uso *SFF_TEST_BINARY_TREES _INDIVIDUALLY* (ver archivo *params_RRTSFF.ini*), de esta forma se utiliza la función

SFF_TEST_BINARY_TREES_INDIVIDUALLY	
Parámetros	**Valor**
min_nodes	10
max_nodes	10000
step	10
NN_searches	100

Tabla 5.1: Parámetros utilizados para realizar la prueba individual de cada árbol binario (*SFF_TEST_BINARY_TREES_INDIVIDUALLY*).

específica *Test binary trees* mencionada en el apartado 4.6.2. Los parámetros utilizados se encuentran en la tabla 5.1.

La prueba consiste en, para cada uno de los árboles binarios (*K-d tree*, *Cover Tree*, *Quad Tree*, *PH-tree*):

1. Crear el árbol dinámicamente, es decir, añadiendo los nodos uno a uno. Se ha escogido crear árboles desde 10 nodos hasta 10000 (*min_nodes* = 10 y *max_nodes* = 10000) avanzando 10 nodos en cada iteración (*step* = 10).

2. Después de cada creación, se realizan *nn_searches* = 100 búsquedas *NN* y se calcula la media para obtener un resultado concluyente.

Notar que para que la prueba sea precisa, se crean exactamente los mismos árboles y se realizan las búsquedas *NN* del vecino más cercano sobre el mismo nodo, para así poder comparar de forma correcta el tiempo que tarda cada árbol en realizar cada operación.

Después de recopilar la información, se realizan los gráficos que se observan en la figura 5.1. Por un lado, se observa el tiempo de creación de los árboles en función del número de nodos, y por otro, el tiempo invertido en las búsquedas *NN* en función de los nodos.

A primera vista, se observa que el tiempo de creación de *Cover Tree* es muy superior al resto de árboles, y por otro lado, el tiempo de búsqueda de los *Quad Tree* también es muy superior al resto de árboles.

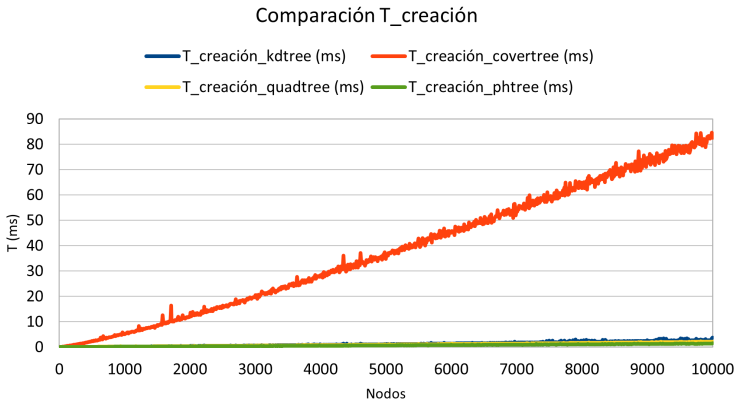
Por ello, si eliminamos estos dos árboles de sus respectivos gráficos, observamos el resultado en la figura 5.2. Ahora se observa como el *PH-tree* es el árbol que se crea más rápido en función del número de nodos. Por otro lado, se observa como el *K-d tree* es el que realiza las búsquedas *NN* más rápidamente.

De todas formas, no se puede concluir cual de los árboles será el elegido para utilizarlo junto a *SFF*. A simple vista parece ser que los que tienen un mejor comportamiento son *K-d Tree* y *PH-tree* pero para poder asegurarlo, durante el apartado 5.3 en el que se prueba el algoritmo *SFF*, se utiliza cada uno de los árboles para verificar si *K-d Tree* y *PH-tree* siguen siendo los que mejor se comportan.

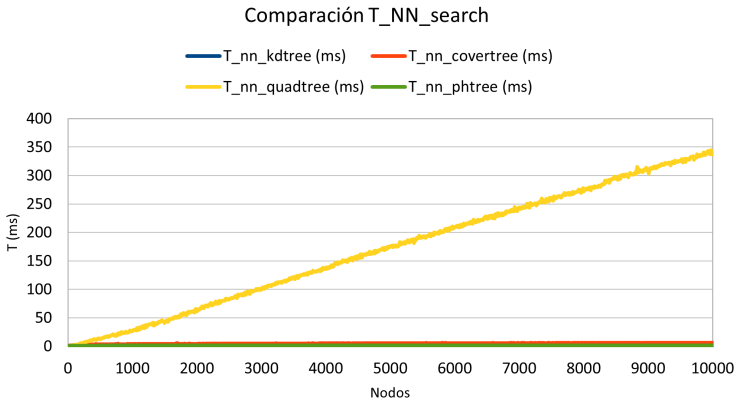
5.2 Selección del algoritmo *TSP* a utilizar

Tal y como se han comentado en 4.1.5, en este apartado se aborda la explicación acerca del estudio realizado entre los distintos algoritmos que resuelven el problema de *TSP*, los cuales se han presentado en 3.3.4.

Para poder llevar a cabo tal elección se ha decidido comparar cada uno de ellos y observar como resuelven diversos problemas con distintos números de puntos objetivo. Se



(a)



(b)

Figura 5.1: Comparación unitaria entre árboles binarios. Se compara el tiempo de creación dinámico de cada árbol y el tiempo en realizar las búsquedas del vecino más cercano.

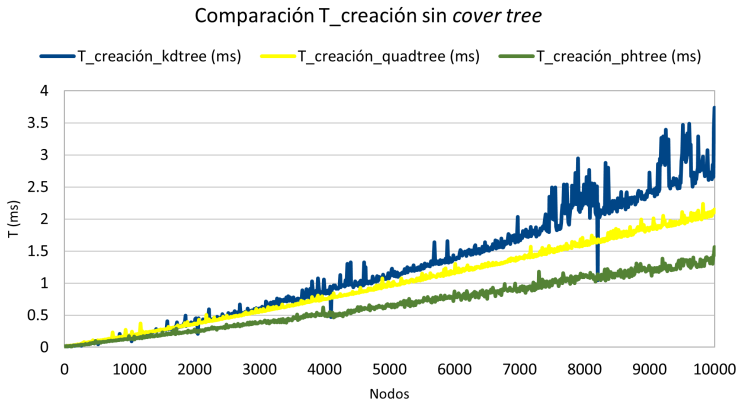
analiza el tiempo de ejecución de cada uno de ellos y el coste del camino encontrado para ver cuál se adapta mejor a las circunstancias que se tienen en este proyecto.

Recordar que de la aplicación de *SFF* se obtiene un grafo no completo y simétrico. La librería seleccionada no contempla la resolución de este tipo de grafos, pero debido a la sencillez que ofrece a la hora de utilizarla, se ha decidido realizar una modificación para simular este tipo de matrices. Para ello, se asigna un valor elevado a las posiciones de la matriz cuyos nodos no están conectados entre sí. De esta forma, se representan dichas conexiones como imposibles y en principio, los algoritmos deben encontrar un camino alternativo que no pase por dichas conexiones.

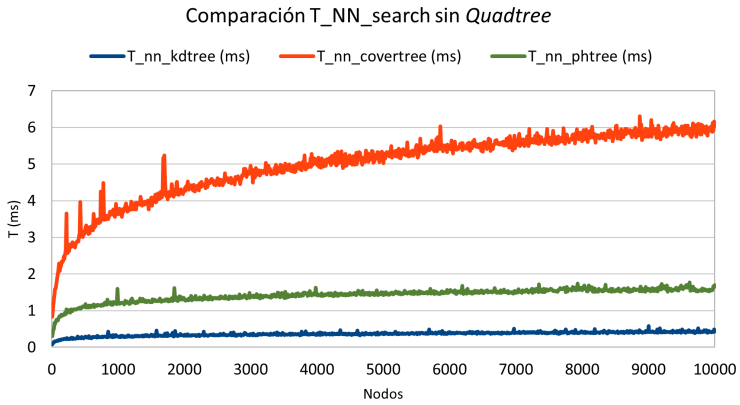
Teniendo en cuenta que durante todas las pruebas realizadas se han utilizado entornos de 300x300 y 500x500 celdas, se ha decidido que las rutas obtenidas entre los distintos puntos objetivo se dividan por un factor 10 y que el valor que represente a ∞ sea 10000. De esta forma, a las celdas que no tengan nodos interconectados se les asigna este valor.

El principal objetivo es que los algoritmos *TSP* nos devuelvan un coste inferior a 10000, de esta forma, sabemos que se ha encontrado un camino posible de realizar.

En el caso en el que se desee trabajar con entornos más grandes o sin reducir el coste



(a)



(b)

Figura 5.2: Comparación *SFF* entre árboles binarios eliminando los árboles *Cover Tree* y *Quad Tree*.

de cada una de las rutas, es necesario replantear y cambiar el valor que representa ∞ para que los algoritmos funcionen correctamente.

5.2.1 Selección del entorno

Respecto al entorno utilizado para realizar esta prueba, se ha decidido utilizar uno completamente libre de obstáculos. El motivo es el hecho de intentar evitar el problema mencionado en 4.4.2, es decir, que alguno de los nodos solo tenga conexión con el vecino más cercano y por lo tanto, para poder resolver el problema de *TSP*, se tenga que aplicar *Dijkstra*. Como se trata de realizar una prueba para cada uno de los algoritmos de forma independiente y ver cómo funciona la librería escogida, se ha decidido obtener una matriz que se pueda resolver y no obtenga el problema mencionado.

5.2.2 Ejecución de *SFF* y obtención de matrices

Se ha decidido realizar la prueba con 6, 8, 12, 17, 30, 40 y 50 nodos. En las figuras 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 y 5.9 se observa el resultado de aplicar el algoritmo *SFF* sobre un entorno completamente vacío de 500x500 celdas. Respecto a cada una de las subfiguras, se observa lo siguiente:

(a) Puntos objetivo repartidos sobre el entorno.

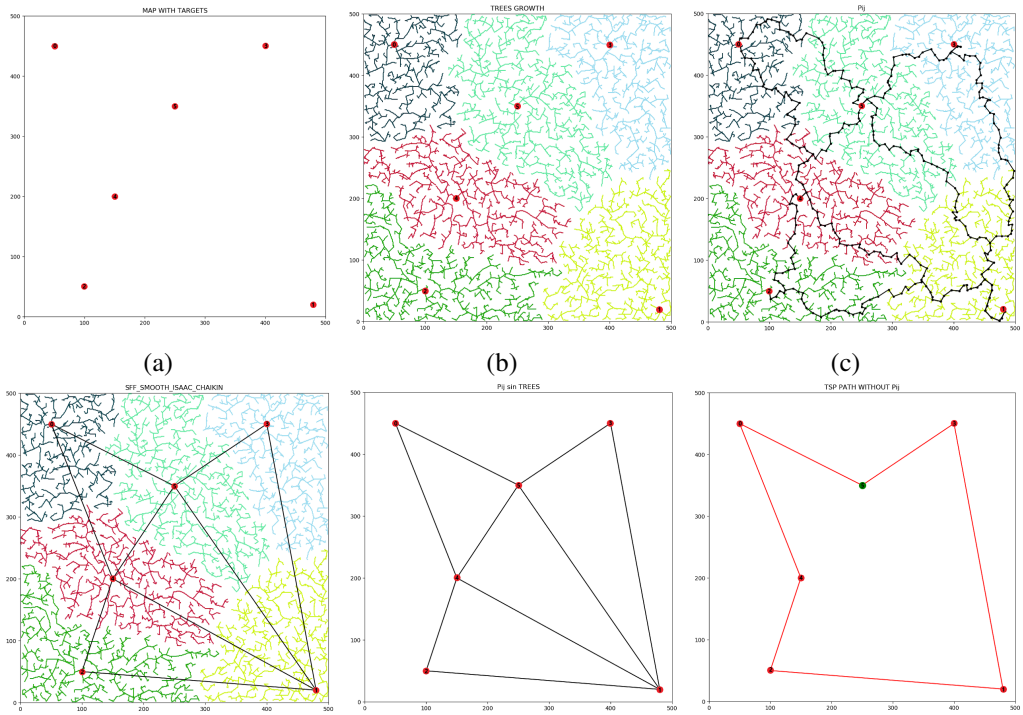
(b) Los árboles obtenidos después de la ejecución de *SFF*.

(c) Matriz P_{ij} con el *roadmap* obtenido.

(d) Matriz P_{ij} una vez se ha aplicado el suavizado *SFF* + *Chaikin* juntamente con los árboles.

(e) Matriz P_{ij} suavizada sin los árboles.

(f) Resultado obtenido, en este caso por *GRASP*, respecto al problema *TSP*. El nodo marcado de color verde representa el punto de partida seleccionado al azar.



(a) (b) (c)

(d) (e) (f)

Figura 5.3: Proceso de creación para el caso de prueba de 6 puntos objetivo.

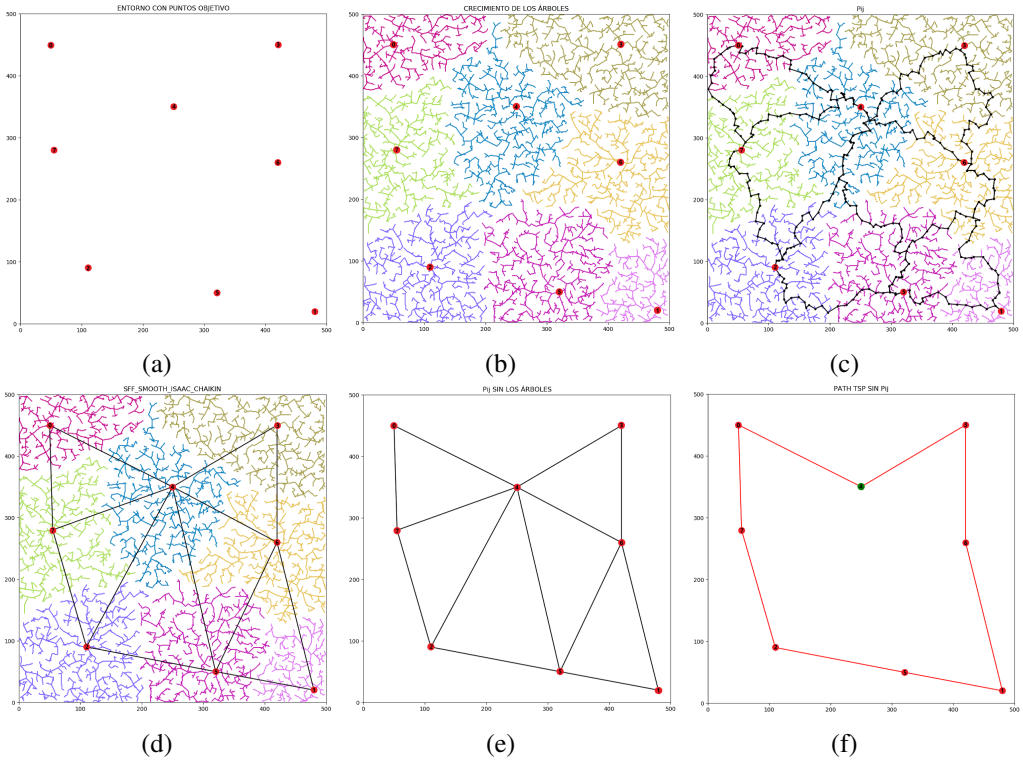


Figura 5.4: Proceso de creación para el caso de prueba de 8 puntos objetivo.

5.2.3 Resultados obtenidos

Para cada una de las pruebas realizadas, se ha ejecutado cada uno de los algoritmos (*Exact, Constructive, Local Search y GRASP*) 10 veces. Posteriormente, se realiza la media del tiempo de ejecución y del coste obtenido y se presenta en los diagramas de barras 5.10 y 5.11.

Respecto al diagrama del tiempo de ejecución, éste ha sido limitado a 0.05 ms, debido a que se establece un límite de tiempo superior debido a la indiferencia de los resultados que sobrepasan éste valor. Por otro lado, el diagrama del coste ha sido limitado a 11000, debido a que distancias superiores a 10000 no son soluciones válidas.

Cabe decir que el algoritmo *Exact* deja de poder ser utilizado a partir de los 17 nodos debido a que el tiempo de ejecución es muy elevado.

En el diagrama de costes, se puede observar que para 6 nodos, el algoritmo *Constructive* no es capaz de encontrar una solución válida ya que devuelve un coste superior a 10000.

De forma similar, el algoritmo *Local Search* deja de devolver resultados correctos a partir de 12 nodos.

En cambio, se observa como el algoritmo *GRASP* siempre devuelve un coste correcto y para los casos en los que aún es comparable al algoritmo *Exact*, se consigue obtener el mismo resultado.

De todas formas, es importante observar que, respecto al tiempo de ejecución de *GRASP*, cuando se trabaja con 6 y 8 nodos, consigue peores resultados que los algoritmos *Constructive* y *Local Search*.

Cabe decir que para que el algoritmo *GRASP* ofrezca estos buenos resultados, se han

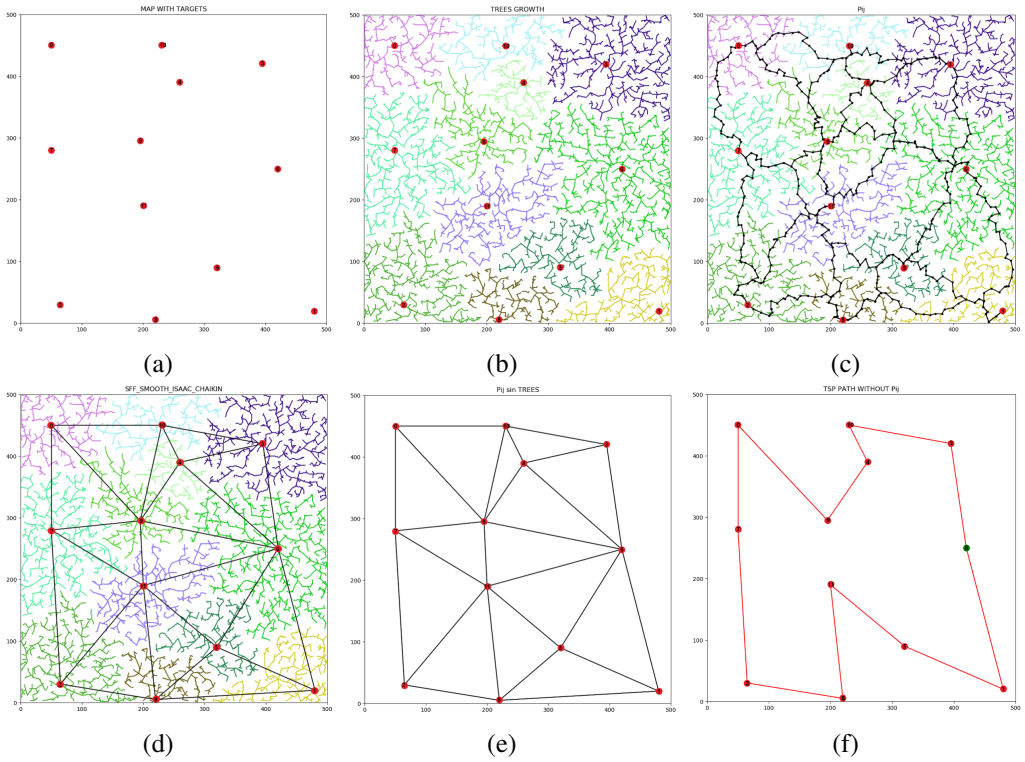


Figura 5.5: Proceso de creación para el caso de prueba de 12 puntos objetivo.

utilizado los siguientes valores para cada una de sus variables:

- Para los casos de 6 a 40 nodos:

  RCL_QUALITY $(r) = 10$

  IMPROVED_ITERATION $(i) = 100$

  GRASP_MAX_ITERATION $(g) = 100$

- Para los casos entre 41 y 50 nodos:

  RCL_QUALITY $(r) = 10$

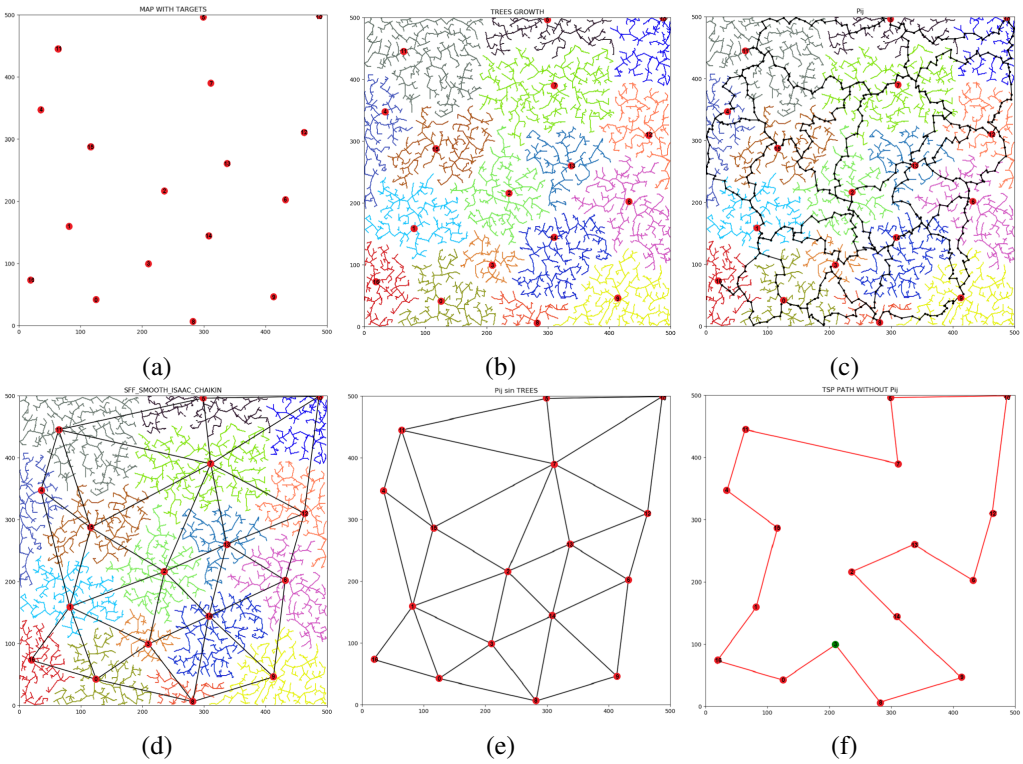
  IMPROVED_ITERATION $(i) = 1000$

  GRASP_MAX_ITERATION $(g) = 1000$

Por otro lado cabe mencionar que, para el caso del algoritmo *Local Search*, independientemente de los valores que se le asignen a sus variables, no se ha encontrado una mejora en el algoritmo. A partir de 12 nodos, no encuentra solución válida.

Por ello, debido a los resultados obtenidos, aunque se sacrifique el tiempo de ejecución y éste pueda ser superior a los 0.05 ms y en algunos casos, para 50 nodos, y con los parámetros $r = 10$, $i = 1000$, $g = 1000$ se pueda llegar hasta los 2s, utilizar GRASP garantiza encontrar siempre una solución correcta con un coste inferior a 10000.

Por ello, se ha implementado de tal forma que para problemas con más de 40 nodos se utilicen los parámetros $r = 10$, $i = 1000$, $g = 1000$ y para problemas con nodos inferiores se utilicen los parámetros $r = 10$, $i = 100$, $g = 100$, por lo que se obtiene un muy buen



(a) (b) (c)

(d) (e) (f)

Figura 5.6: Proceso de creación para el caso de prueba de 17 puntos objetivo.

tiempo de ejecución para problemas con menos de 40 nodos y un tiempo de ejecución que puede llegar a los 2s en problemas con más de 40 nodos. Así, por lo menos se garantiza que se encuentre una solución válida al problema de *TSP*.

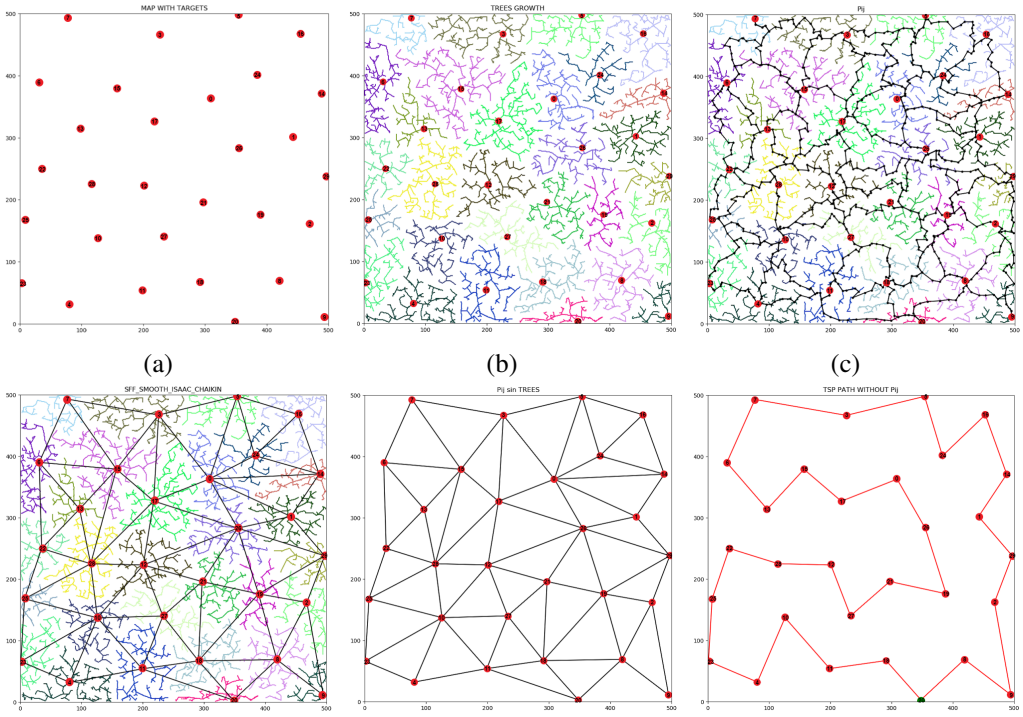
5.3 Resultados sobre el comportamiento del algoritmo *SFF*.

Para verificar que se ha comprendido e implementado el algoritmo *SFF* correctamente, en este apartado se llevan a cabo una serie de pruebas para determinar cual es el comportamiento del algoritmo en función de las diversas modificaciones que se le pueden realizar a sus parámetros. Las pruebas que se llevan a cabo son:

1. Desglose del tiempo que tarda cada una de las secciones del algoritmo *SFF*. Así se observa con que parte de la implementación se debe tener cuidado ya que si se realiza de forma incorrecta puede repercutir negativamente sobre el tiempo de ejecución.

2. Ejecución completa del algoritmo juntamente con *TSP*, para así visualizar cada una de las partes que componen una ejecución completa. De esta forma, se observa el proceso de visualización que se ha implementado.

3. Comportamiento del algoritmo debido al cambio del número de puntos objetivo. En función del número de puntos objetivo con los que trabaje el algoritmo, se estudia el comportamiento de éste.



(a) (b) (c)

(d) (e) (f)

Figura 5.7: Proceso de creación para el caso de prueba de 30 puntos objetivo.

4. Comportamiento del algoritmo debido al cambio del parámetro k. En función de la variación de este parámetro se puede modificar el tiempo que se invierte en las expansiones de los nodos, modificando así la forma que tendrán los árboles y el tiempo que se tarda en crearlos.

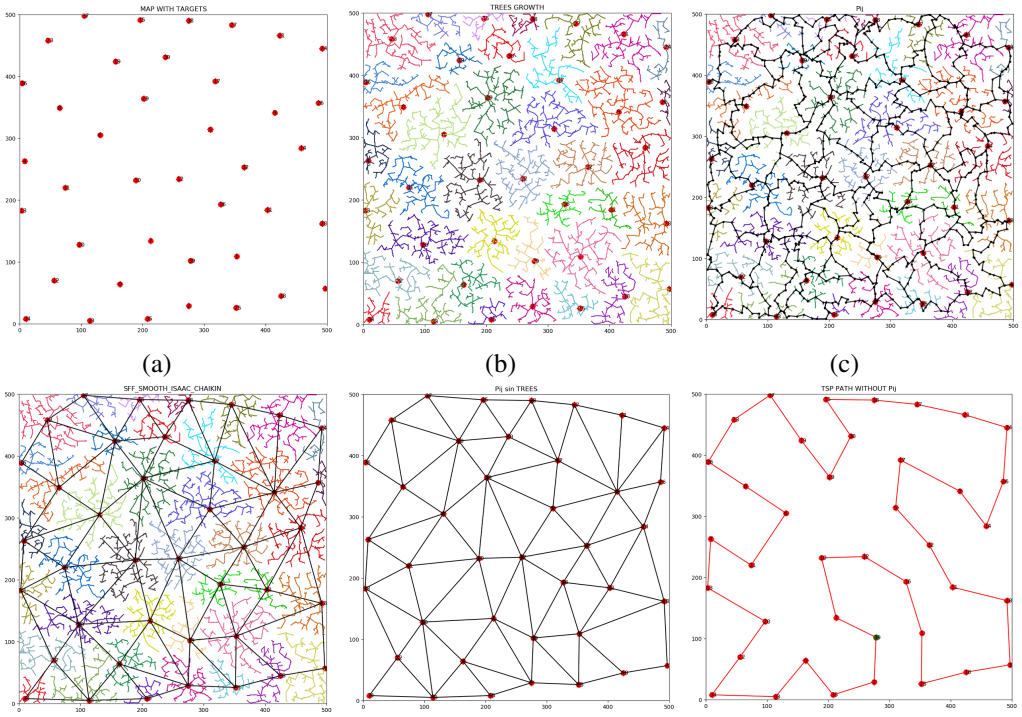
5. Comportamiento del algoritmo debido al cambio del parámetro d_{tree}. En función de lo lejos que se queden los árboles unos de otros, el *roadmap* obtenido es diferente, por eso se intenta analizar el impacto que tiene la modificación de este parámetro.

6. Comportamiento del algoritmo debido al cambio del parámetro R. De forma parecida a los dos anteriores, modificar este parámetro tiene repercusión en el tamaño que pueden tener cada una de las ramas, haciendo así que el resultado final del algoritmo, y por lo tanto el *roadmap* obtenido, cambie.

Es importante notar que durante las pruebas de los puntos 3, 4, 5 y 6 se tiene en cuenta el árbol binario a utilizar, para observar si en función del parámetro que se modifique pero tambien en función del árbol binario que se utilice, se obtiene un resultado u otro.

5.3.1 Ejemplo de ejecución del algoritmo *SFF*

Primeramente se desea mostrar un ejemplo completo de ejecución del algoritmo, debido a que posteriormente no se utilizará toda la representación que se muestra a continuación para mostrar cada uno de los casos comentados anteriormente.



(a) (b) (c)

(d) (e) (f)

Figura 5.8: Proceso de creación para el caso de prueba de 40 puntos objetivo.

De esta forma, en apartados posteriores, es más sencillo de entender lo que se está leyendo debido a que primeramente se ha observado un ejemplo completo de la ejecución del algoritmo.

Una ejecución completa consta de tres fases:

- Creación de los árboles. Se muestra el crecimiento de los árboles hasta que se detienen por uno de los motivos comentados en el apartado 3.2.3.
- Creación de la matriz P_{ij} o *roadmap* y suavizado de las rutas. Una vez finalizada la fase de construcción de los árboles, se obtiene la matriz P_{ij} conteniendo así los caminos entre cada uno de los puntos objetivo.
- Aplicación del algoritmo *TSP* y obtención de la ruta que se debe seguir para visitar cada uno de los puntos objetivo y volver al inicial.

El entorno que se ha decidido utilizar para realizar la mayoría de las pruebas de comprensión del algoritmo ha sido el que se ha mostrado previamente en la figura 4.14.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.2.

El crecimiento de los árboles se puede observar en la figura 5.12. Se han situado 12 puntos a una distancia mínima de 135 celdas entre ellos.

Posteriormente, en la figura 5.13, se observa la creación de la matriz P_{ij} y el suavizado de cada una de las rutas que la componen.

El camino que *TSP* ha calculado sobre la matriz P_{ij} se muestra en la figura 5.14.

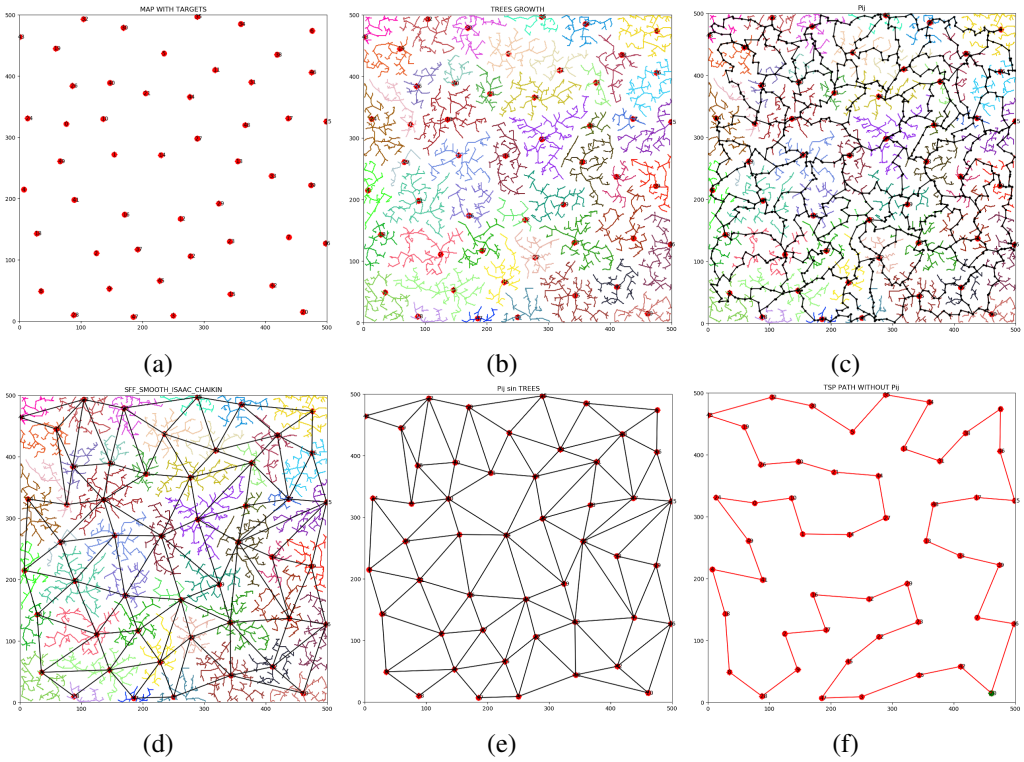


Figura 5.9: Proceso de creación para el caso de prueba de 50 puntos objetivo.

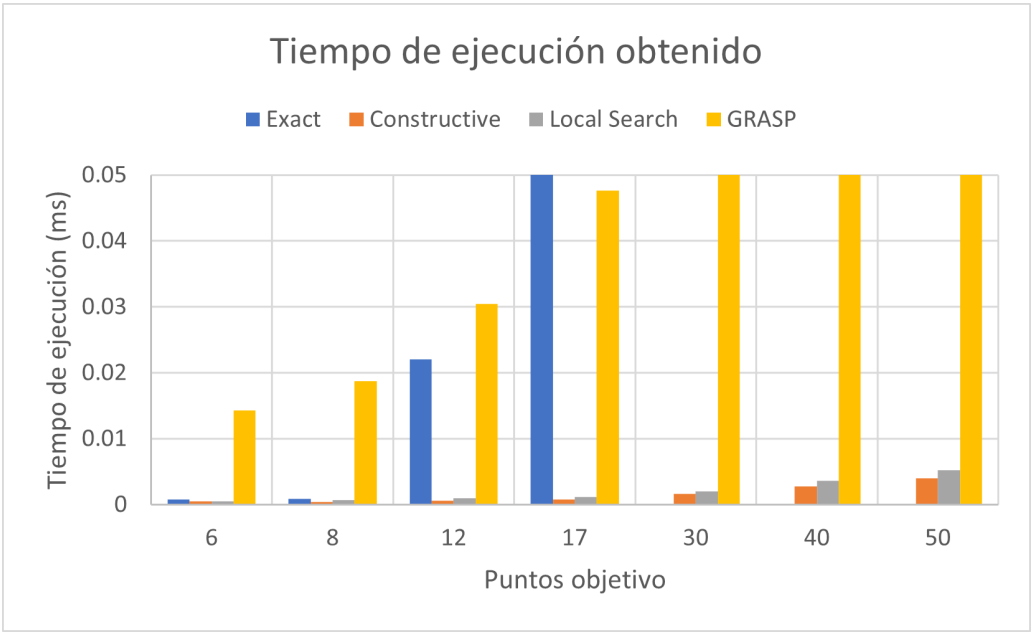


Figura 5.10: Diagrama de barras del tiempo de ejecución en función de los puntos objetivo para los algoritmos (*Exact, Constructive, Local Search y GRASP*).

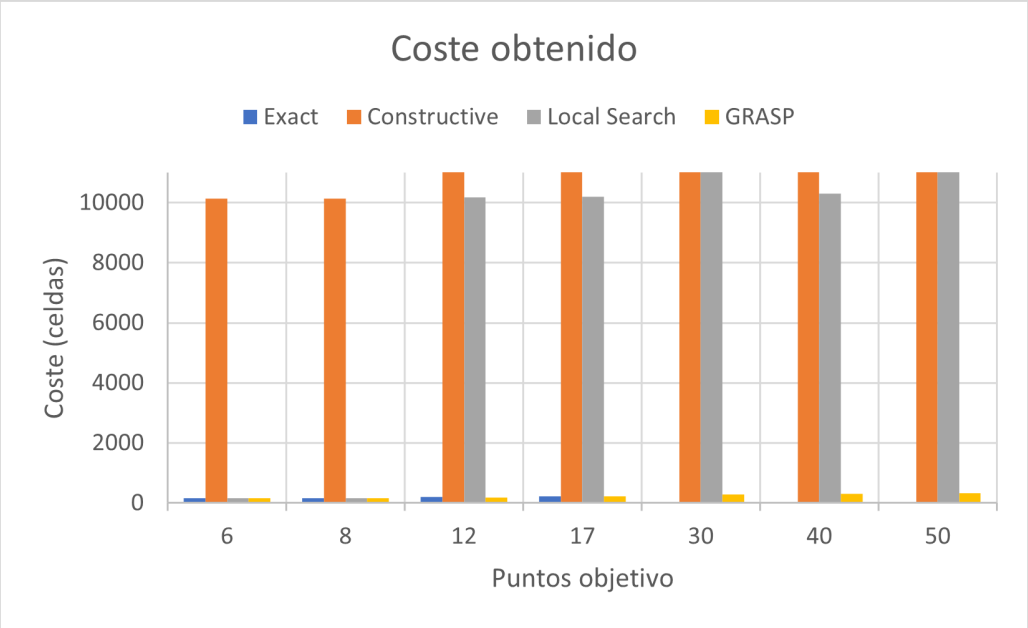


Figura 5.11: Diagrama de barras del coste obtenido en función de los puntos objetivo para los algoritmos (*Exact, Constructive, Local Search y GRASP*).

EJEMPLO DE EJECUCIÓN DEL ALGORITMO SFF	
Parámetros	**Valor**
number_of_targets	12
min_dist_between_targets	135
k	5
d_tree	10
R	15
binary_tree	1 = KDTREE

Tabla 5.2: Parámetros utilizados para realizar el ejemplo de ejecución del algoritmo *SFF*.

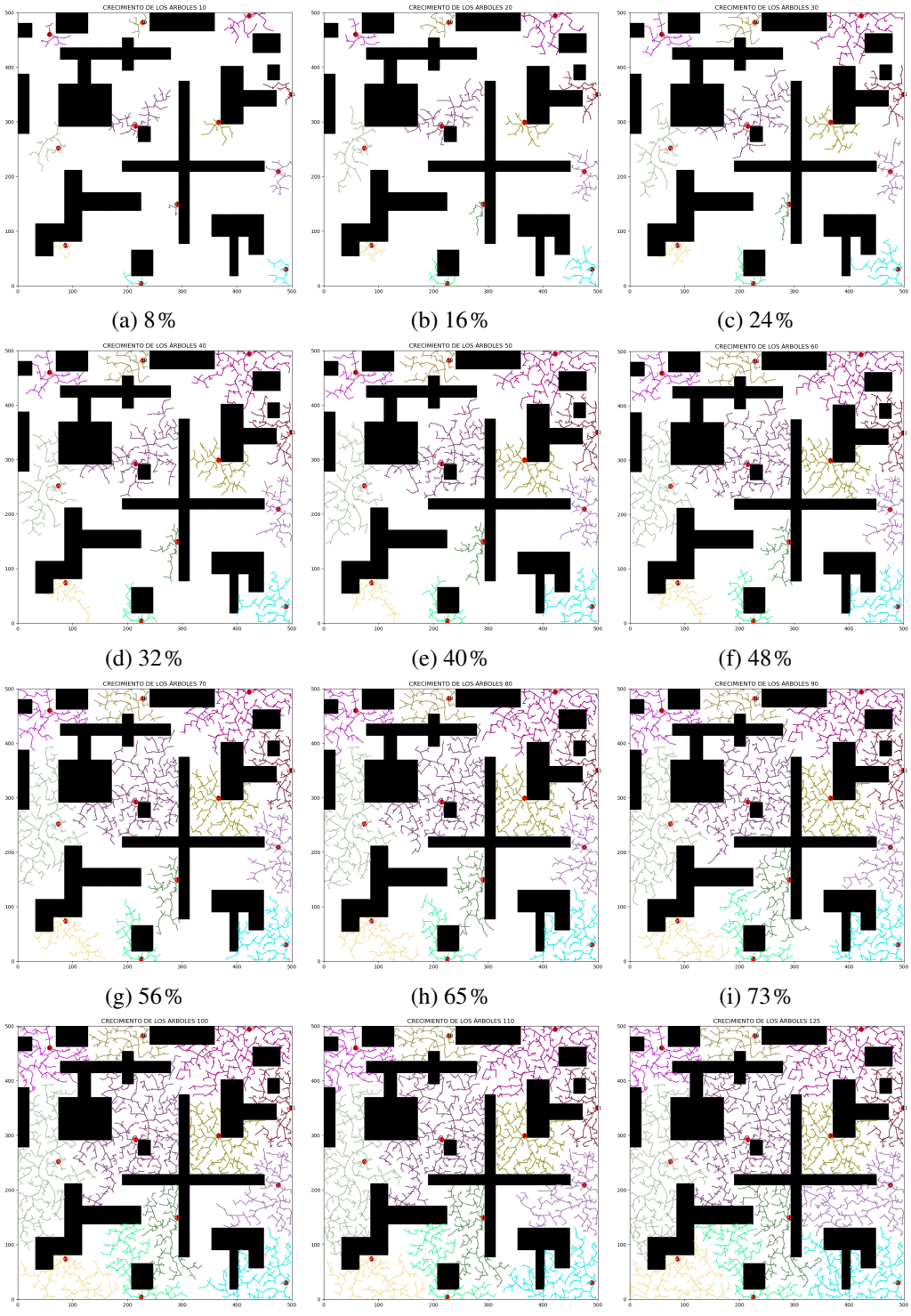
5.3.2 Desglose del tiempo invertido en cada sección del algoritmo *SFF*

Tal y como se ha comentado anteriormente y debido a que en las primeras fases de implementación del algoritmo se tuvieron muchos problemas respecto a los grandes tiempo de ejecución obtenidos, se ha decidido realizar un desglose de cada uno de los tiempos en los cuales se puede dividir el algoritmo *SFF*. Pese a que se muestren los tiempos una vez ya se solucionó el problema (gracias a los árboles binarios), se ha querido mostrar de todas formas ya que así se ve la gran cantidad de tiempo que consumen las búsquedas *NN*.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.3.

Los tiempos que se han querido calcular son, por orden de aparición en el algoritmo *SFF*:

- *T_preamble_time*. Tiempo invertido antes de empezar el bucle principal del algoritmo.



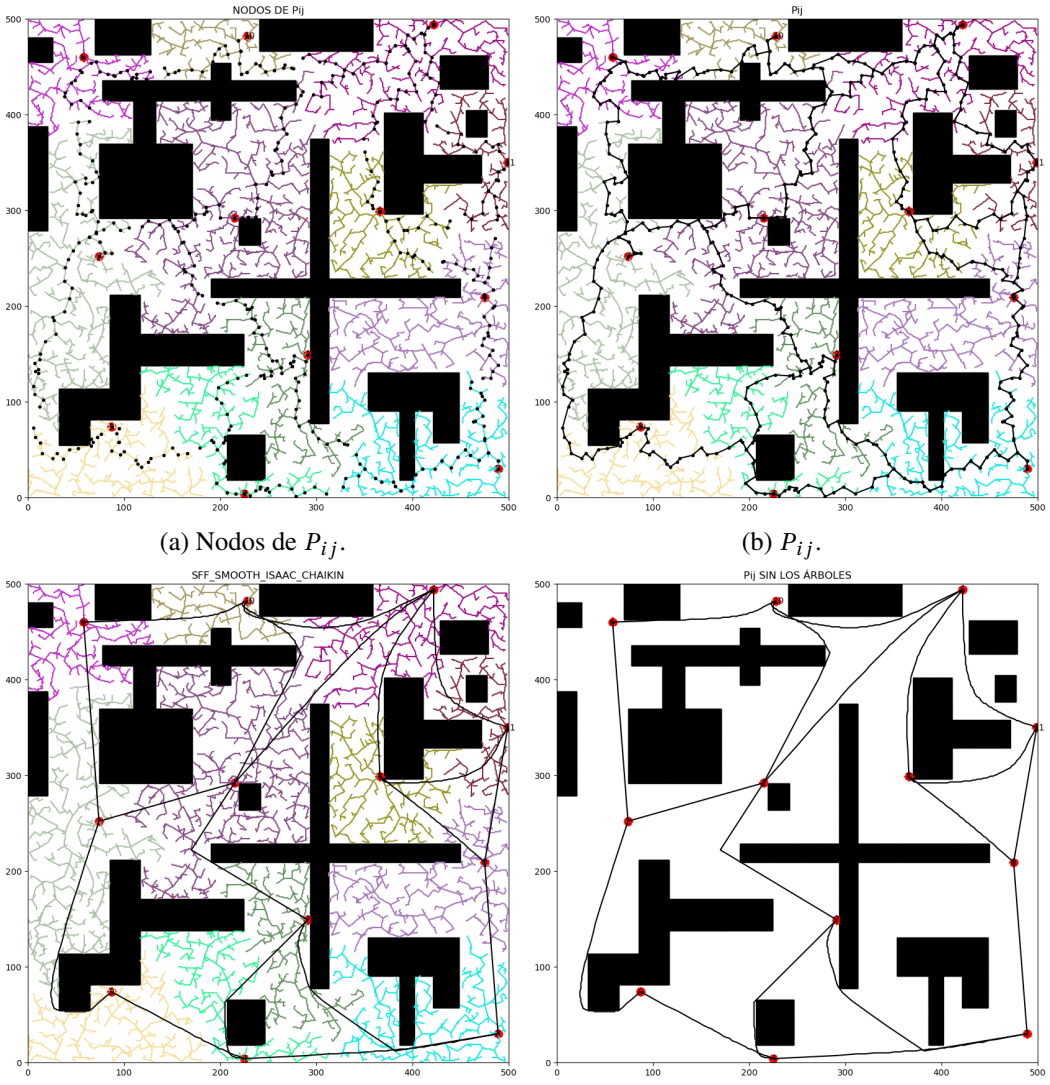
(a) 8 % (b) 16 % (c) 24 %

(d) 32 % (e) 40 % (f) 48 %

(g) 56 % (h) 65 % (i) 73 %

(j) 81 % (k) 89 % (l) 100 %

Figura 5.12: Ejemplo de crecimiento de los árboles durante la ejecución del algoritmo *SFF*.



(a) Nodos de P_{ij}.

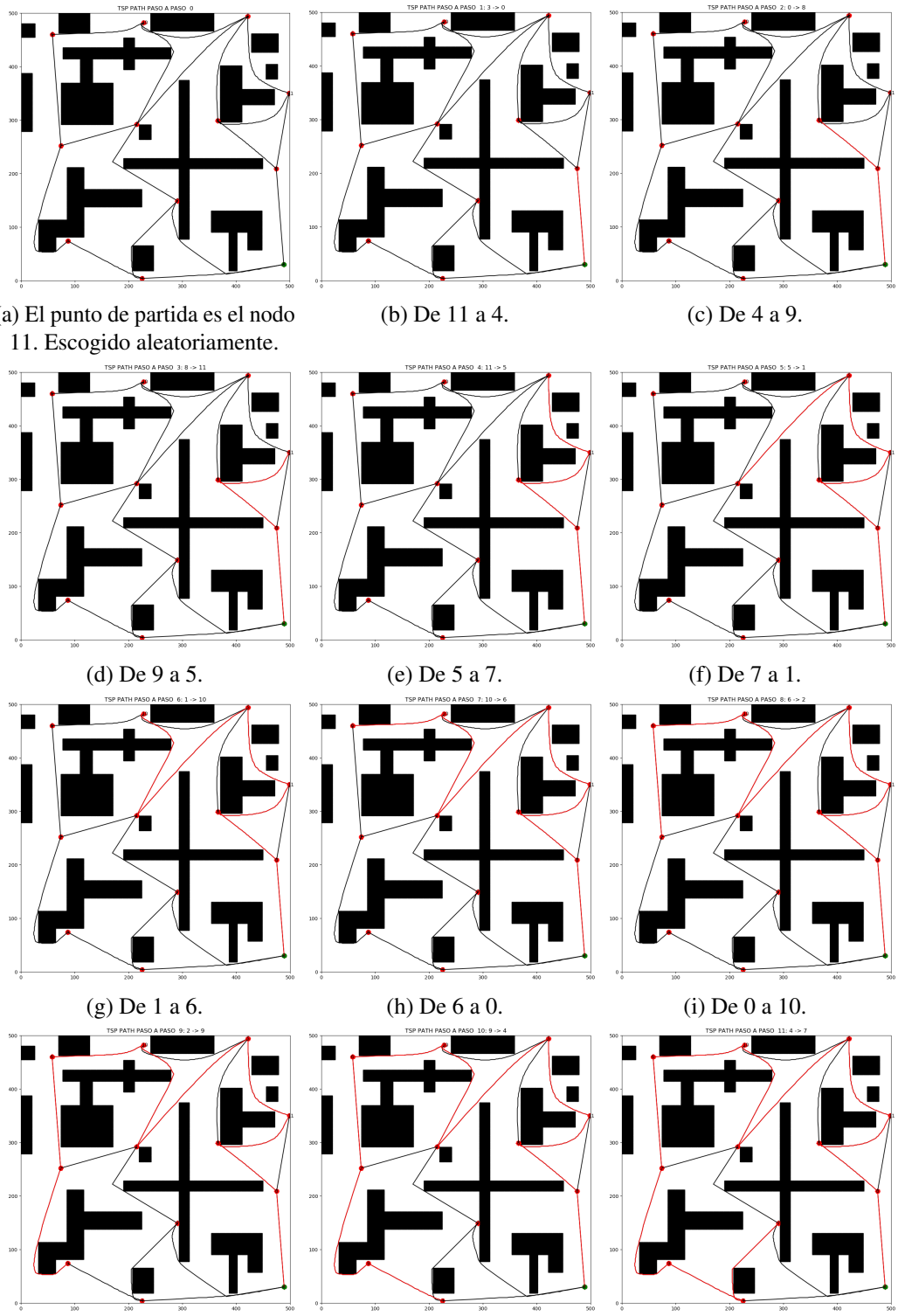
(b) P_{ij}.

(c) P_{ij} suavizado.

(d) P_{ij} suavizado sin los árboles.

Figura 5.13: Ejemplo de creación de la matriz P_{ij} a partir de los nodos de los árboles y posterior suavizado del *roadmap* resultante.

- *T_1_time*. Tiempo invertido en seleccionar el nodo aleatorio del vector *Open_List* y en establecer *succ* = *false*.
- *T_2_time*. Tiempo invertido en seleccionar aleatoriamente el nodo q_c.
- *T_D_i*. Tiempo invertido en realizar las búsquedas *NN* sobre el mismo árbol que se desea expandir.
- *T_ D_j*. Tiempo invertido en realizar las búsquedas *NN* sobre el resto de árboles.
- *T_addnode_time*. Tiempo que se tarda en añadir un nuevo nodo a un árbol.
- *T_add_openlist_time*. Tiempo que se tarda en añadir un nodo al vector *Open_List*.



(a) El punto de partida es el nodo 11. Escogido aleatoriamente.

(b) De 11 a 4.

(c) De 4 a 9.

(d) De 9 a 5.

(e) De 5 a 7.

(f) De 7 a 1.

(g) De 1 a 6.

(h) De 6 a 0.

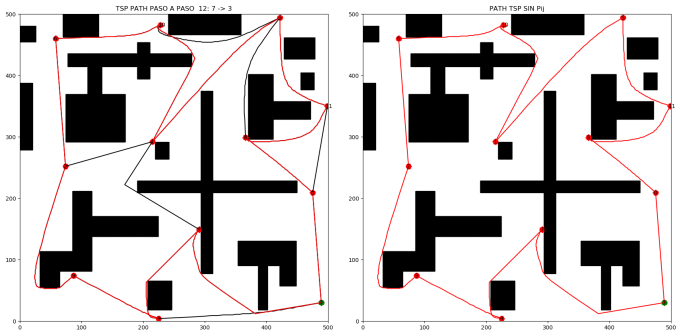
(i) De 0 a 10.

(j) De 10 a 8.

(k) De 8 a 2.

(l) De 2 a 3.

Figura 5.14: Seguimiento punto a punto del camino que ha calculado el algoritmo *TSP*.



(m) De 3 a 11. (n) Camino calculado por *TSP* sobre P_{ij}

Figura 5.14: Seguimiento punto a punto del camino que ha calculado el algoritmo *TSP*.

CÁLCULO DE TIEMPOS DEL ALGORITMO SFF	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
k	5
d_tree	10
R	15
number_of_executions	1000
binary_tree	1 = KDTREE

Tabla 5.3: Parámetros utilizados para realizar el desglose del tiempo que se invierte en cada sección del algoritmo *SFF*.

- *T_connect_other_tree_time*. Tiempo que se tarda en conectar un árbol con otro y añadir los nodos a la matriz P_{ij}.

- *T_remove_item_openlist_time*. Tiempo que se tarda en eliminar un nodo del vector *Open_List*.

En el pseudocódigo 10, se observa de color azul en que punto se calcula cada uno de los tiempos presentados anteriormente.

Se observa que después de ejecutar el algoritmo 1000 veces con los parámetros de la tabla 5.3, gracias nuevamente al modo *SFF_TEST_BINARY_TREES*, los porcentajes de cada uno de los tiempos son los que se observan en el diagrama 5.15. Se han ordenado de mayor a menor y podemos ver como los más influyentes son D_j y D_i, es decir, el tiempo que se tarda en realizar las búsquedas *NN* entre todos los árboles vecinos y del mismo árbol del nodo que se desea expandir en cada iteración, respectivamente.

Por este motivo, el trabajo realizado en el apartado 4.2, dónde se buscan e incorporan árboles binarios para realizar este tipo de búsquedas ha sido tan importante. De esta forma, se consigue reducir los tiempos D_j y D_i considerablemente, permitiendo que el tiempo de computación total también se reduzca, creando así un algoritmo más eficiente y rápido a la hora de ejecutarse.

Algoritmo 10: Algoritmo *SFF* con las anotaciones de las medidas de tiempo tomadas

Input:

Vector de puntos c	Puntos objetivo $c_1, ..., c_n \in C_{free}$
d_{tree}	Distancia mínima entre los árboles
R	Tamaño de expansión de las ramas
k	Número de intentos para cada expansión

Output:

P_{ij}	Caminos entre los puntos objetivo

```
1  function algoritmoSFF(vector_c, d_tree, R, k)
2      T_i.añadirNodo(c_i), i = 1,...,n; // T_preamble_time
3      O.indexar((c_i, i)), i = 1,...,n; // T_preamble_time
4      P_ij = ∅, ∀_i, j = 1,...,n; // T_preamble_time
5      while O no esté vacío do
6          q, i = seleccionar ítem aleatorio de O; // T_1_time
7          succ = false; // T_1_time
8          for trial = 1 : k do
9              q_c = configuración aleatoria alrededor de q la cual cumple
                ϱ(q, q_c) ≤ R; // T_2_time
10             d_i, q' = T_i.vecinoMasCercano(q_c); // T_Di
11             d_j, q_j = argmin_{j≠i} T_j.vecinoMasCercano(q_c); // T_Dj
12             if d_j > d_tree then
13                 T_i.añadirNodo(q_c); // T_addnode_time
14                 T_i.añadirRama(q, q_c); // T_addnode_time
15                 O.añadirNodo((q_c, i)); // T_add_openlist_node
16                 succ = true;
17                 break;
18             else
19                 if P_ij = ∅ and esPosibleConectar(q, q_j) then
20                     P_ij = T_i.ruta(c_i, q) ∪ T_j.ruta(q_j, c_j);//T_connect_other_tree_time
21         if not succ then
22             O.eliminarNodo((q, i)); // T_remove_item_openlist_time
23     return P_ij
```

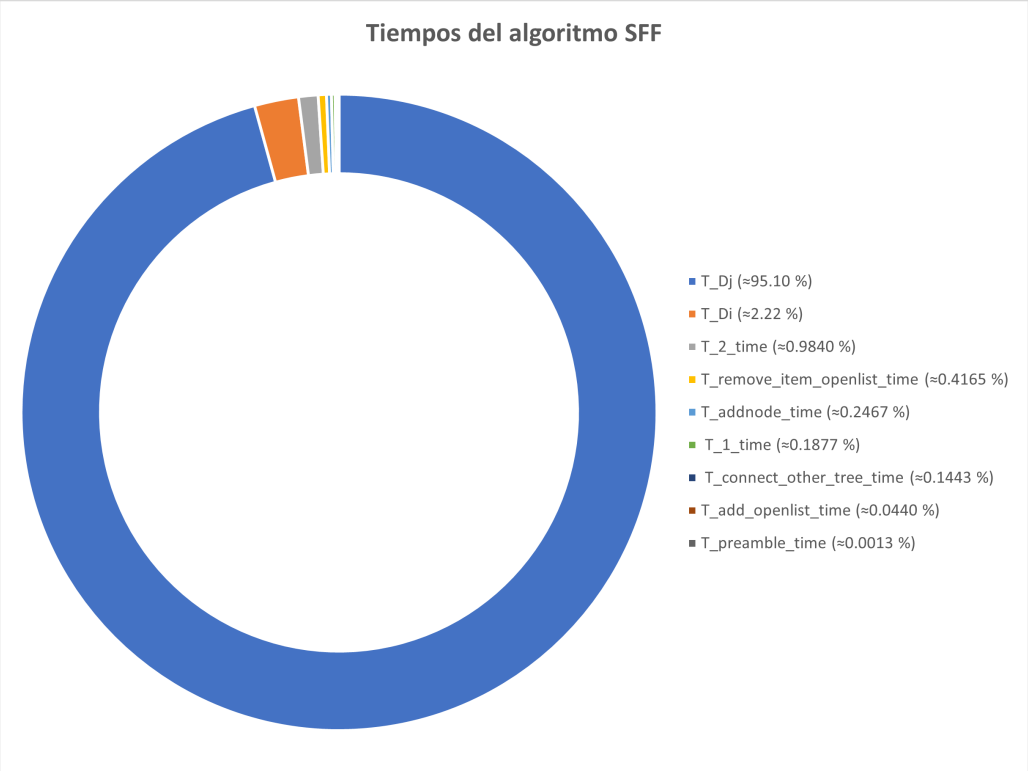


Figura 5.15: Desglose de tiempos de cada sección del algoritmo *SFF*.

5.3.3 Comportamiento de *SFF* debido al cambio del número de puntos objetivo

En esta prueba se verifica el comportamiento del algoritmo en función del número de puntos objetivo sobre los que se ejecuta.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.4. Se establece una distancia mínima de 30 celdas entre los puntos objetivo, se fijan también los valores de los parámetros *k*, *d_tree* y *R*. Además, gracias al modo de test *SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS* seleccionado desde el archivo *params_RRTSFF.ini* se puede iterar entre un número de puntos objetivo delimitado por los parámetros *min_targets* y *max_targets* avanzando *step_targets* en cada iteración.

Posteriormente, se ejecuta la prueba 4 veces en función del árbol binario que se utilice. En los gráficos de la figura 5.16 se observan los resultados obtenidos:

- Respecto al tiempo de ejecución (5.16a). Se observa como el tiempo (en milisegundos) varía en función del número de puntos objetivo establecido. Se observa como a medida que el número de puntos objetivo aumenta, el tiempo de ejecución también lo hace. Esto es debido a que como existe una mayor cantidad de árboles, en cada iteración se tarda más en realizar las búsquedas *NN*. También se observa como el árbol que en este caso ofrece mejores prestaciones es *K-d Tree*.

- Respecto al coste del camino calculado por *TSP* (5.16b). El coste del camino que calcula *TSP* también aumenta debido a que a mayor cantidad de nodos, más caminos

SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS	
Parámetros	**Valor**
min_dist_between_targets	30
k	5
d_tree	10
R	15
min_targets	2
max_targets	50
step_targets	1
number_of_executions	10
binary_tree	1 = KDTREE 2 = COVER_TREE 3 = QUAD_TREE 4 = PH_TREE

Tabla 5.4: Parámetros utilizados para realizar la prueba de incremento del número de puntos objetivo *SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS*

SFF_TEST_ALGORITHM_PERFORMANCE_K	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
d_tree	10
R	15
min_k	2
max_k	15
step_k	1
number_of_executions	10
binary_tree	1 = KDTREE 2 = COVER_TREE 3 = QUAD_TREE 4 = PH_TREE

Tabla 5.5: Parámetros utilizados para realizar la prueba de incremento del valor del parámetro *k SFF_TEST_ALGORITHM_PERFORMANCE_K*

existirán, por lo que se tendrá que recorrer una mayor cantidad de espacio para poder visitarlos todos.

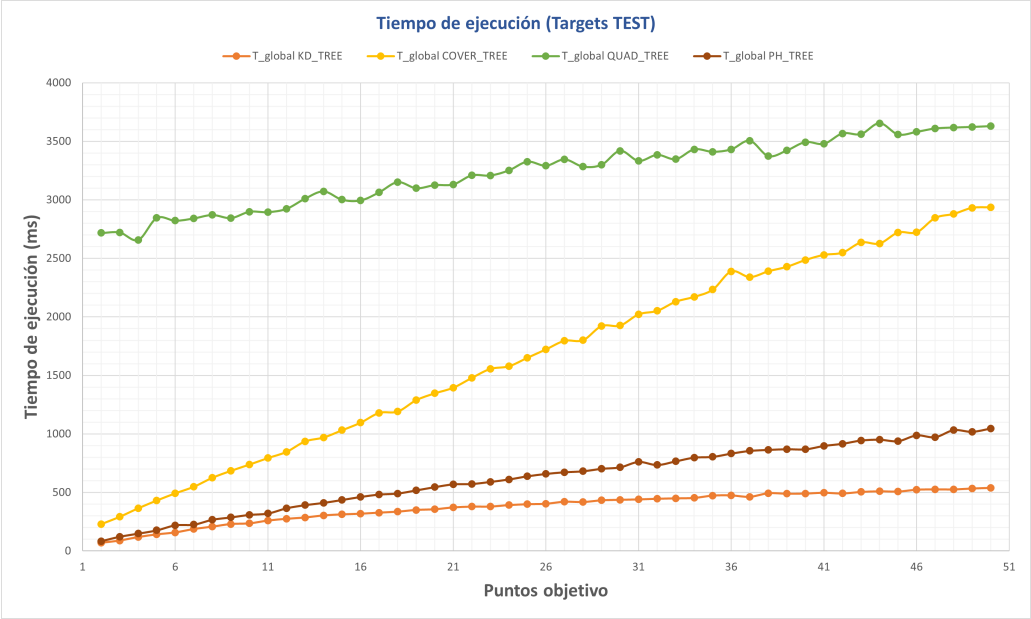
En la figura 5.17 se ha querido mostrar un pequeño ejemplo de ejecución una vez los árboles han acabado de crecer para observar así como se comporta el algoritmo cuando se le incrementa el número de puntos objetivo para los que tiene que encontrar un *roadmap*.

5.3.4 Comportamiento de *SFF* debido al cambio del parámetro k

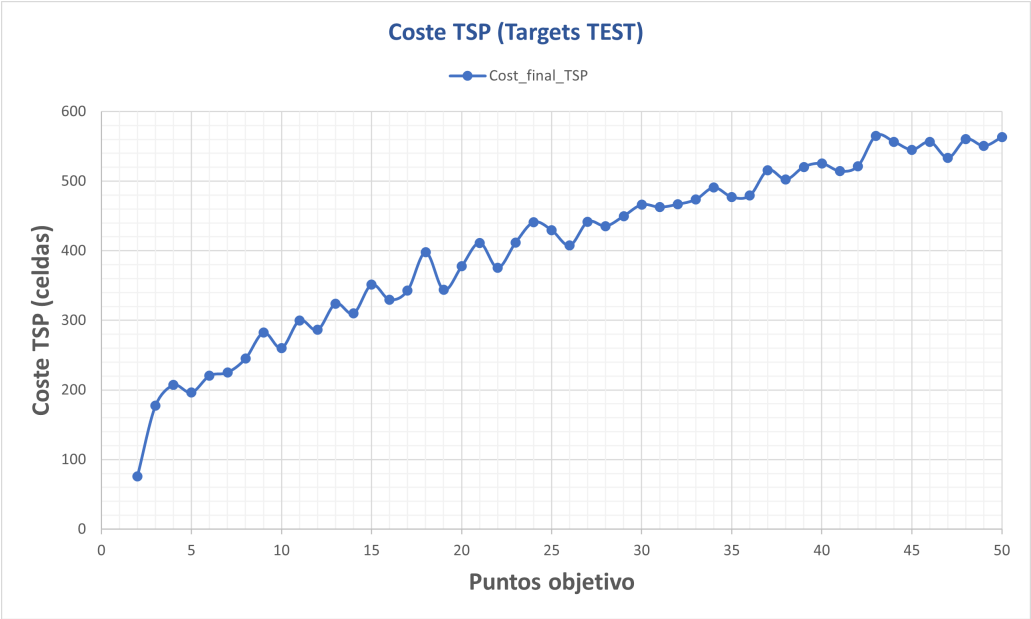
En esta prueba se verifica el comportamiento del algoritmo en función del valor que se le asigne al parámetro k.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.5. En este caso, se ha fijado el número de puntos objetivo a 25 con una distancia mínima entre ellos de 80 celdas, además de los parámetros *d_tree* y *R*. En este caso se itera con el parámetro k (min_k, $step\_k$, max_k) gracias al modo de test *SFF_TEST_ALGORITHM_PERFORMANCE_K*.

En los gráficos de la figura 5.18 se observan los resultados obtenidos:



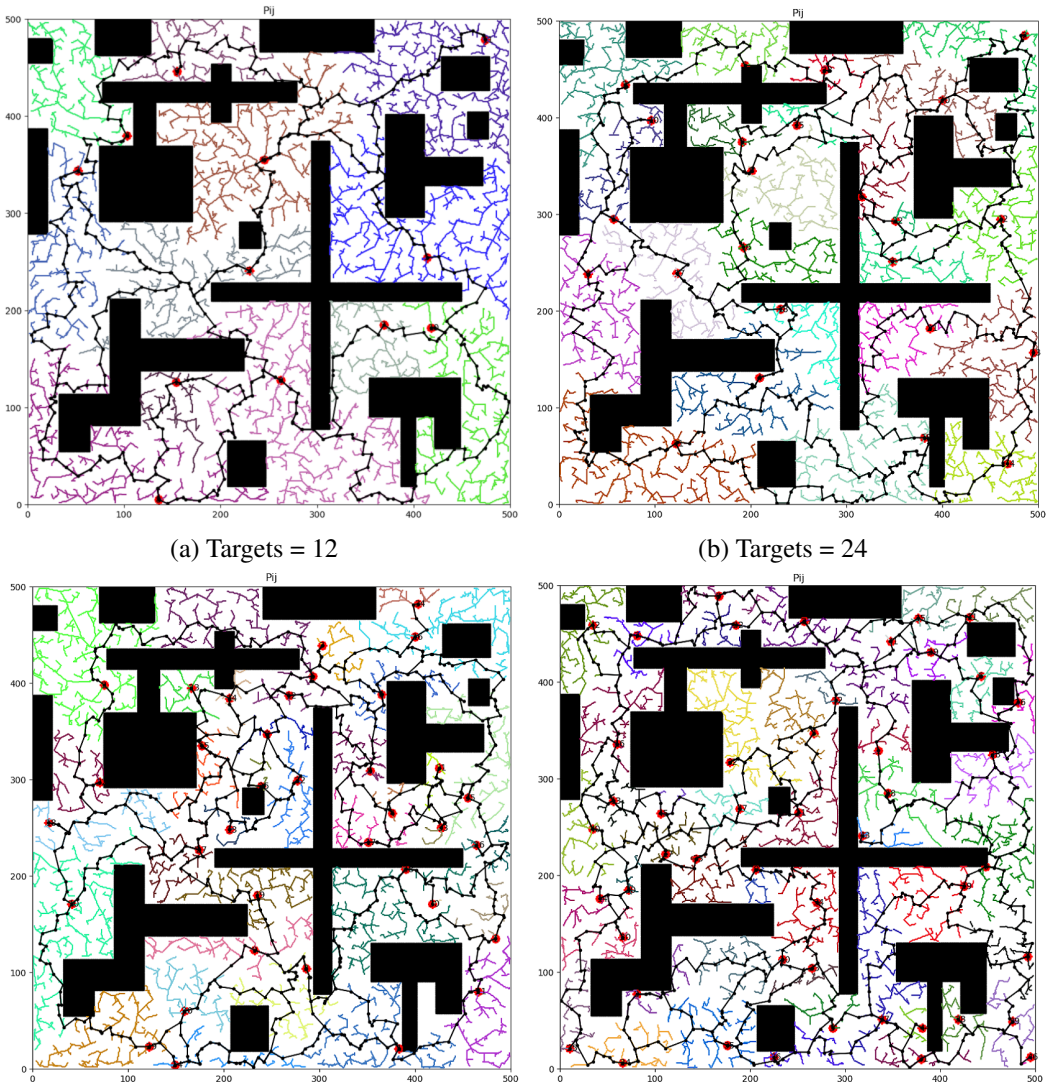
(a)



(b)

Figura 5.16: Resultados de la prueba del incremento del número de puntos objetivo aplicado al algoritmo *SFF*. Se observa el tiempo de ejecución y el coste de *TSP* en función del incremento del número de puntos objetivo.

- Respecto al tiempo de ejecución (5.18a). Se observa como, a medida que el número de puntos objetivo aumenta, el tiempo de ejecución también lo hace. A diferencia del caso anterior en el que aumentaba el número de puntos objetivo, se observa como, el orden del tiempo de ejecución es muy superior al anterior. El motivo por el que aumenta tanto es que, a medida que k aumenta, se le dan muchas más



(a) Targets = 12 (b) Targets = 24

(c) Targets = 36 (d) Targets = 48

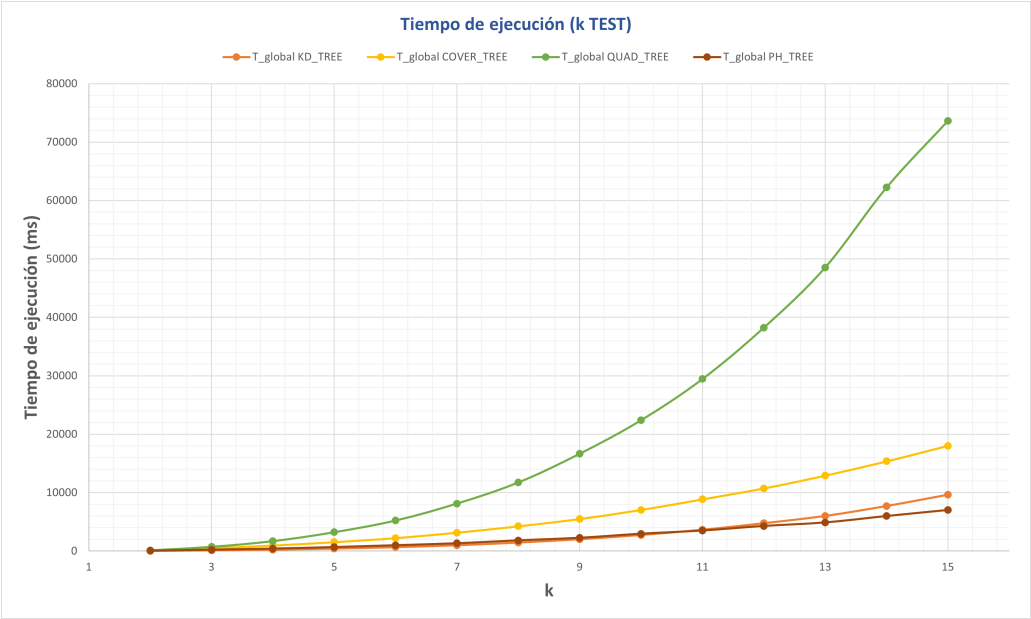
Figura 5.17: Diversos ejemplos de ejecución de la prueba del incremento del número de puntos objetivo aplicado al algoritmo *SFF*.

oportunidades a cada nodo para poder expandirse, haciendo así que las expansiones sean más lentas y que al final se tengan árboles con muchos más nodos. En este caso el árbol *PH-tree* se comporta mejor que *K-d Tree*.

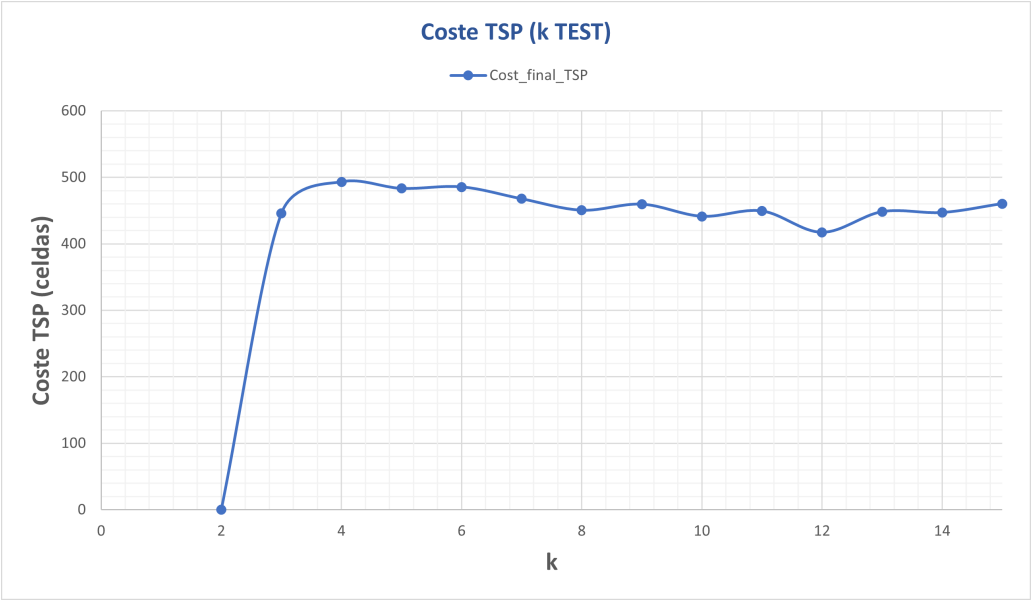
- Respecto al coste del camino calculado por *TSP* (5.18b). El coste del camino que calcula *TSP* también aumenta debido a que a mayor cantidad de nodos, pero a diferencia del caso anterior, parece ser que se estabiliza y no crece a partir de k=5.

Notar que para valores pequeños de k el tiempo de ejecución y el coste del camino se anulan debido a que aparece el problema mencionado en 4.4.2. Los árboles no son capaces de crecer debido a que no se ha escogido un valor lo suficientemente elevado y

por lo tanto se detienen antes de poder conectarse con los demás, imposibilitando así la creación de un *roadmap*.



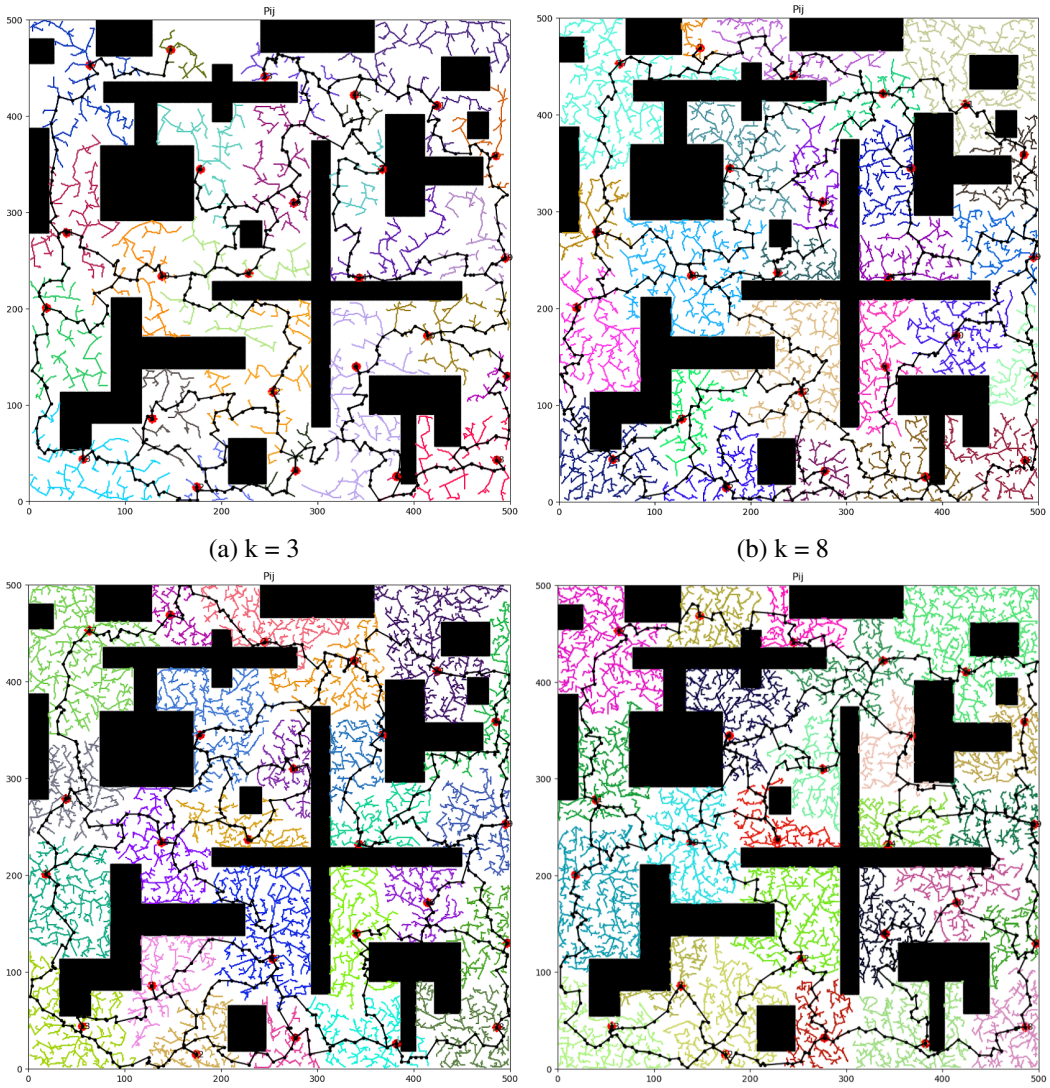
(a)



(b)

Figura 5.18: Resultados de la prueba del incremento del valor del parámetro k. Se observa el tiempo de ejecución y el coste de *TSP* en función del parámetro k.

En la figura 5.19 se muestra ejemplo de ejecución una vez los árboles han acabado de crecer para observar así como se comporta el algoritmo cuando se modifica el parámetro k. Los árboles creados con una k mayor están mucho más poblados, lo que justifica el mayor tiempo de ejecución.



(a) k = 3 (b) k = 8

(c) k = 12 (d) k = 15

Figura 5.19: Diversos ejemplos de ejecución de la prueba del incremento del valor del parámetro k.

5.3.5 Comportamiento de *SFF* debido al cambio del parámetro d_{tree}

En esta prueba se verifica el comportamiento del algoritmo en función del valor que se le asigne al parámetro d_{tree}.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.6. Nuevamente, se ha fijado el número de puntos objetivo a 25 con una distancia mínima entre ellos de 80 celdas. Los parámetros k y R también se han fijado. En este caso se itera con el parámetro d_{tree} (min_d_{tree}, $step\_d_{tree}$, max_d_{tree}) gracias al modo de test *SFF_TEST_ALGORITHM_PERFORMANCE_DTREE*.

En los gráficos de la figura 5.20 se observan los resultados obtenidos:

- Respecto al tiempo de ejecución (5.20a). A medida que el número de puntos

SFF_TEST_ALGORITHM_PERFORMANCE_DTREE	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
k	5
R	15
min_d_tree	5
max_d_tree	30
step_d_tree	1
number_of_executions	10
binary_tree	1 = KDTREE 2 = COVER_TREE 3 = QUAD_TREE 4 = PH_TREE

Tabla 5.6: Parámetros utilizados para realizar la prueba de incremento del valor del parámetro d_{tree} *SFF_TEST_ALGORITHM_PERFORMANCE_DTREE*

objetivo aumenta, el tiempo de ejecución disminuye. A medida que d_{tree} aumenta, la distancia a la que los árboles se quedan unos de otros es mayor, por lo que el tiempo que los árboles tardan en expandirse y conectarse es menor, haciendo así que el tiempo de ejecución disminuya. *K-d Tree*, igual que en la primera prueba cuando se han modificado el número de nodos, es el árbol que hace que el algoritmo *SFF* se ejecute más rápidamente.

- Respecto al coste del camino calculado por *TSP* (5.20b). Por el resultado obtenido, parece ser que aumentar el parámetro d_{tree} no tiene un impacto relevante sobre la calidad del camino que se obtiene, es decir, sobre el coste que supone visitar todos los nodos. Se observa como se mantiene estable aproximadamente entre 400 y 500 celdas.

El ejemplo de la ejecución de la prueba se observa en la figura 5.21 como, efectivamente, al aumentar d_{tree} los árboles se quedan aún más alejados.

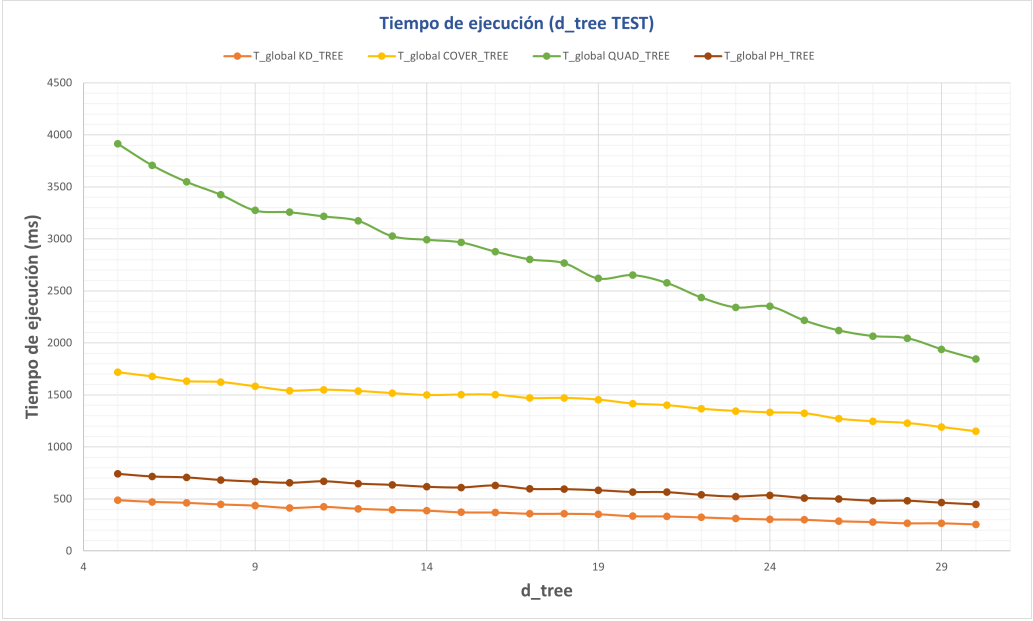
5.3.6 Comportamiento de *SFF* debido al cambio del parámetro R

En esta prueba se analiza el comportamiento del algoritmo en función del valor que se le asigne al parámetro R.

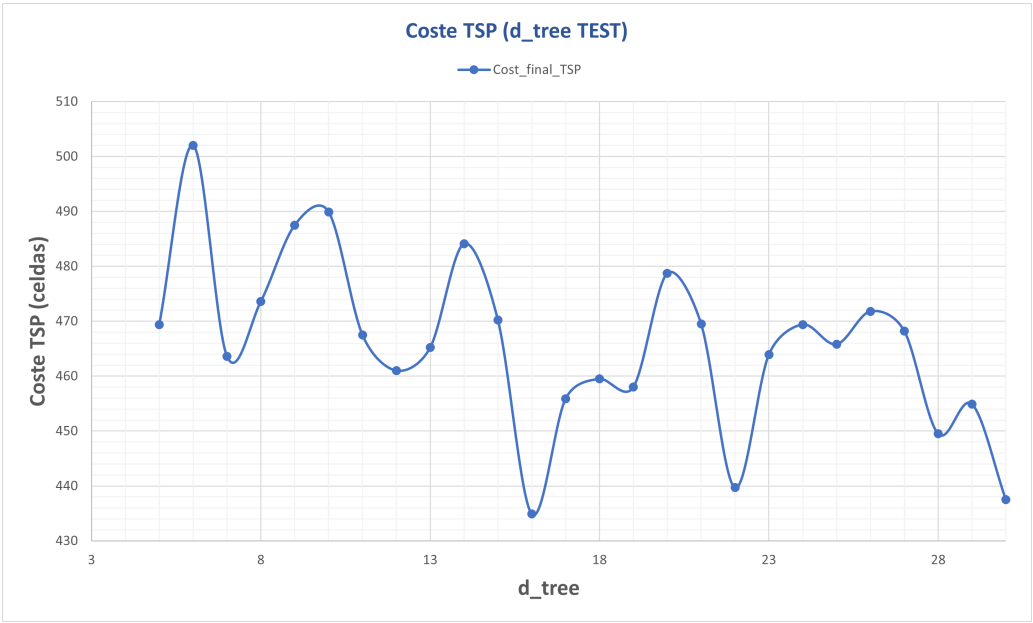
Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.7. Se ha fijado nuevamente el número de puntos objetivo a 25 con una distancia mínima entre ellos de 80 celdas y también se han fijado los parámetros k y d_{tree}. En este caso se itera con el parámetro R (min_R, $step\_R$, max_R) gracias al modo de test *SFF_TEST_ALGORITHM_PERFORMANCE_R*.

En los gráficos de la figura 5.22 se observan los resultados obtenidos:

- Respecto al tiempo de ejecución (5.22a). El tiempo de ejecución disminuye drásticamente a medida que se aumenta el valor de R, debido a que los árboles tienen las ramas más largas y, por lo tanto, se tarda menos tiempo en explorar el entorno. En este caso *PH-tree* es con diferencia el árbol que se comporta mejor ya que para valores de R pequeños, consigue hacer que el algoritmo *SFF* se ejecute entre 15 y



(a)

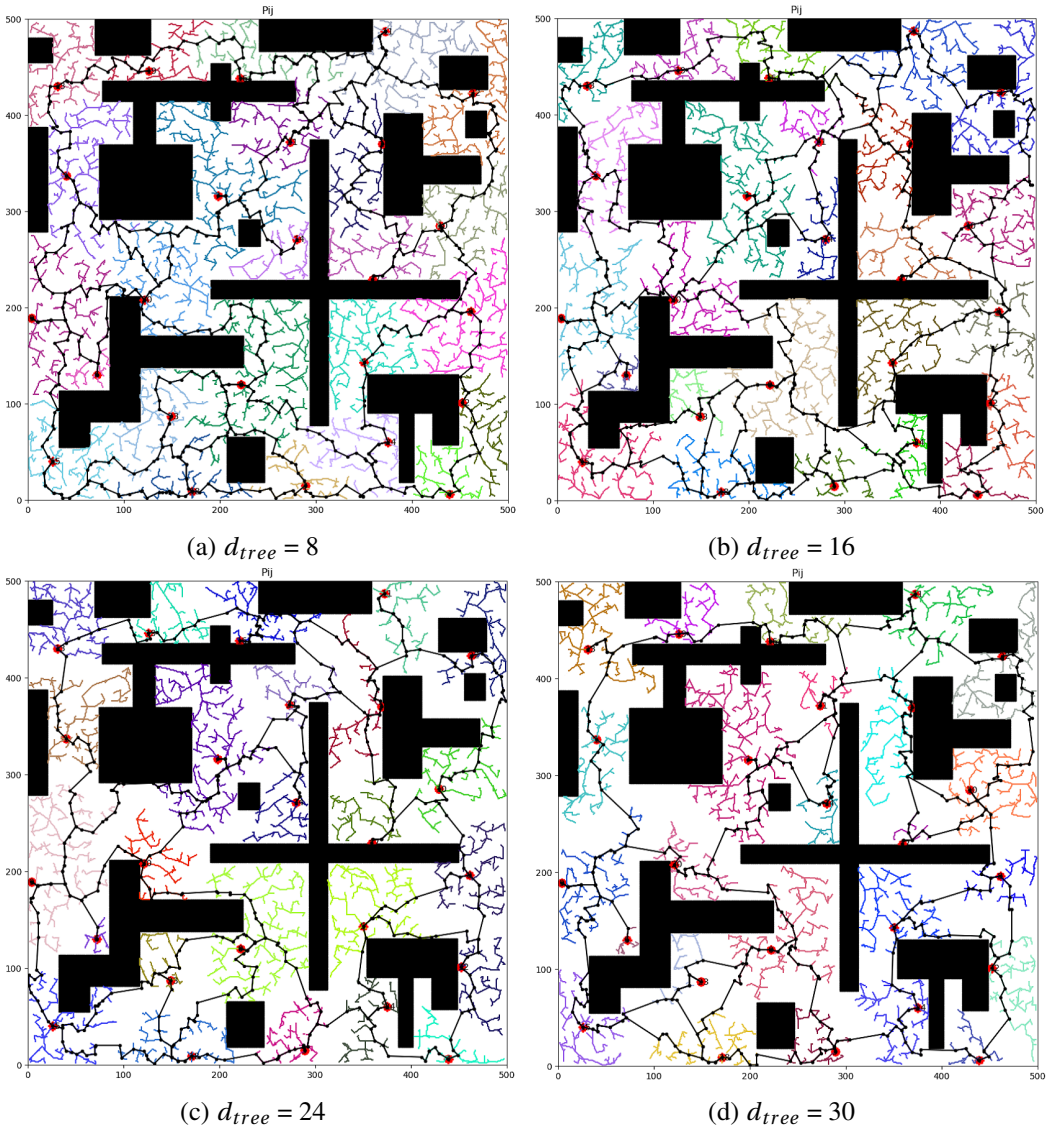


(b)

Figura 5.20: Resultados de la prueba del incremento del valor del parámetro d_{tree}. Se observa el tiempo de ejecución y el coste de *TSP* en función del incremento del parámetro d_{tree}.

20 segundos más rápido que utilizando el siguiente árbol que se comporta mejor, *K-d Tree*.

- Respecto al coste del camino calculado por *TSP* (5.22b). Por el resultado obtenido, parece ser que aumentar el parámetro R, tal y como ocurre con d_{tree}, no tiene un



(a) $d_{tree} = 8$ (b) $d_{tree} = 16$

(c) $d_{tree} = 24$ (d) $d_{tree} = 30$

Figura 5.21: Diversos ejemplos de ejecución de la prueba del incremento del valor del parámetro d_{tree}.

impacto relevante sobre la calidad del camino que se obtiene. Se observa como nuevamente éste varía entre 400 y 500 celdas.

Se ha detectado que puede surgir un problema debido a una mala elección del parámetro R respecto al valor de d_{tree}. Es decir, si por ejemplo se elige $R = 20$ y $d_{tree} = 5$, se obtiene el resultado que se muestra en la figura 5.23. Los árboles se entrelazan debido a que no son capaces de detectar la distancia a la que se deben detener a tiempo debido a la mala elección de los parámetros. Es un problema sencillo de resolver si se tiene cuidado a la hora de escoger los valores. Lo más sencillo es establecer que $R < d_{tree}$, pero se ha observado que el algoritmo se comporta bien si el valor de R no supera en más de 5 celdas al valor de d_{tree}.

SFF_TEST_ALGORITHM_PERFORMANCE_R	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
k	5
d_tree	15
min_R	5
max_R	20
step_R	1
number_of_executions	10
binary_tree	1 = KDTREE 2 = COVER_TREE 3 = QUAD_TREE 4 = PH_TREE

Tabla 5.7: Parámetros utilizados para realizar la prueba de incremento del valor del parámetro *R SFF_TEST_ALGORITHM_PERFORMANCE_R*

El ejemplo de la ejecución de la prueba se observa en la figura 5.24, dónde el tamaño de las ramas de los árboles aumenta a medida que se incrementa *R*.

5.3.7 Comportamiento de *SFF* en función del árbol binario utilizado

Como se ha podido observar durante el transcurso del apartado anterior, mientras se ha comprobado la correcta implementación y el correcto comportamiento del algoritmo *SFF*, también se ha aprovechado para utilizar cada uno de los árboles binarios para comprobar si *SFF* sufre variaciones respecto al tiempo que tarda en ejecutarse.

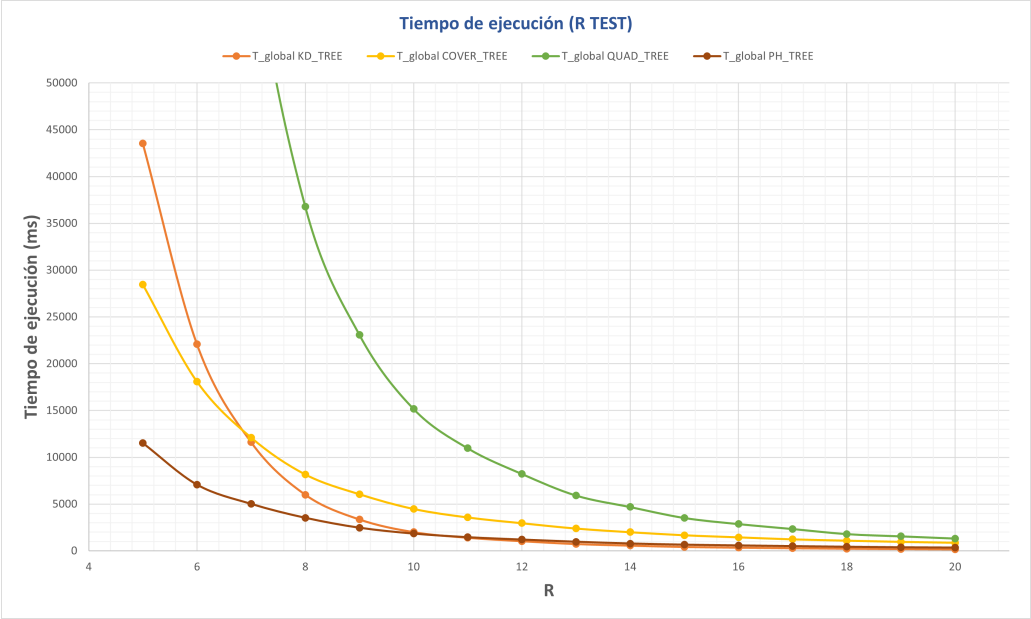
Los resultados han sido claros y, para cada uno de los casos, los árboles *Cover Tree* y *Quad Tree* no han resultado de gran utilidad ya que han sido los que hacen qué el tiempo de ejecución se eleve más.

En cambio, respecto a los árboles *K-d Tree* y *PH-tree*, parece ser que hacen que el algoritmo *SFF* se comporte mejor en función de la prueba que se esté realizando. Por ejemplo:

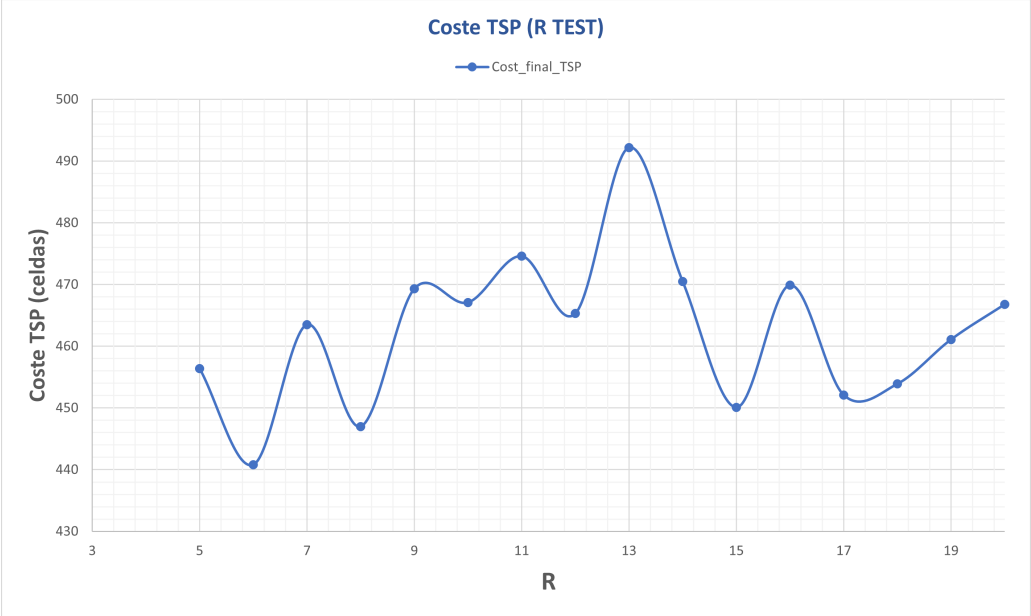
- Para la prueba de *R* y *k* el árbol *PH-tree* es el que ha ofrecido mejores resultados, posiblemente debido a que para árboles con gran cantidad de nodos (como es el caso de cuando se utiliza una *R* pequeña o una *k* grande) *PH-tree* se comporta mejor que *K-d Tree*.

- En cambio, para las pruebas de variación en el número de puntos objetivo y el parámetro d_{tree}, se han obtenido mejores resultados con *K-d Tree*.

En conclusión, excepto para el caso en el que se tengan *R* pequeñas (debido a la gran diferencia de tiempo obtenida), los árboles *K-d Tree* y *PH-tree* han ofrecido buenos resultados y su uso es recomendable juntamente con el algoritmo *SFF*.



(a)



(b)

Figura 5.22: Resultados de la prueba del incremento del valor del parámetro R. Se observa el tiempo de ejecución y el coste de *TSP* en función del parámetro R.

5.4 Resultados sobre el comportamiento del algoritmo *RRT-MO*

Tal y como se ha realizado para *SFF*, se desea verificar que se ha comprendido e implementado el algoritmo *RRT-MO* correctamente. Se vuelven a realizar una serie de pruebas similares a las realizadas con *SFF* para determinar cual es el comportamiento del algo-

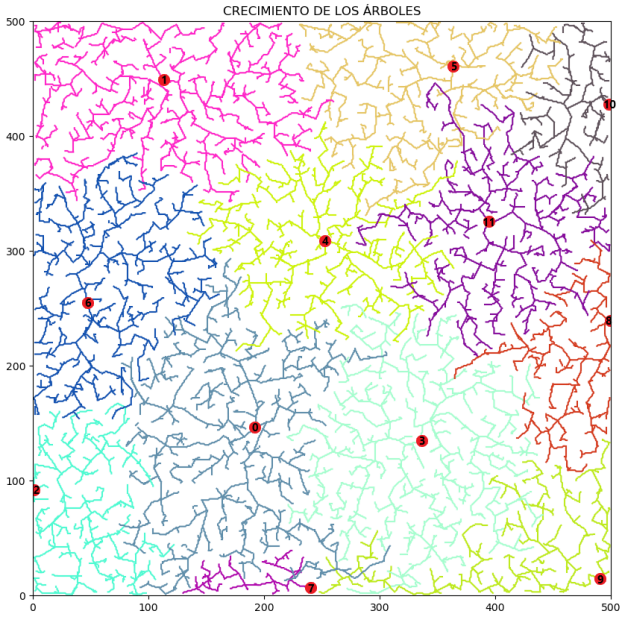


Figura 5.23: Ejemplo del problema que se obtiene cuando el parámetro R es mayor que d_{tree}. Los árboles se entrelazan entre ellos.

ritmo *RRT-MO* en función de las diversas modificaciones que se le pueden realizar a sus parámetros. Los pruebas llevadas a cabo son:

1. Ejecución completa del algoritmo, para así visualizar cada una de las partes que componen una ejecución.

2. Comportamiento del algoritmo debido al cambio del número de puntos objetivo.

3. Comportamiento del algoritmo debido al cambio del parámetro *step_size*. De forma similar al parámetro R del algoritmo *SFF*, modificar este parámetro tiene repercusión en el tamaño que tienen cada una de las ramas.

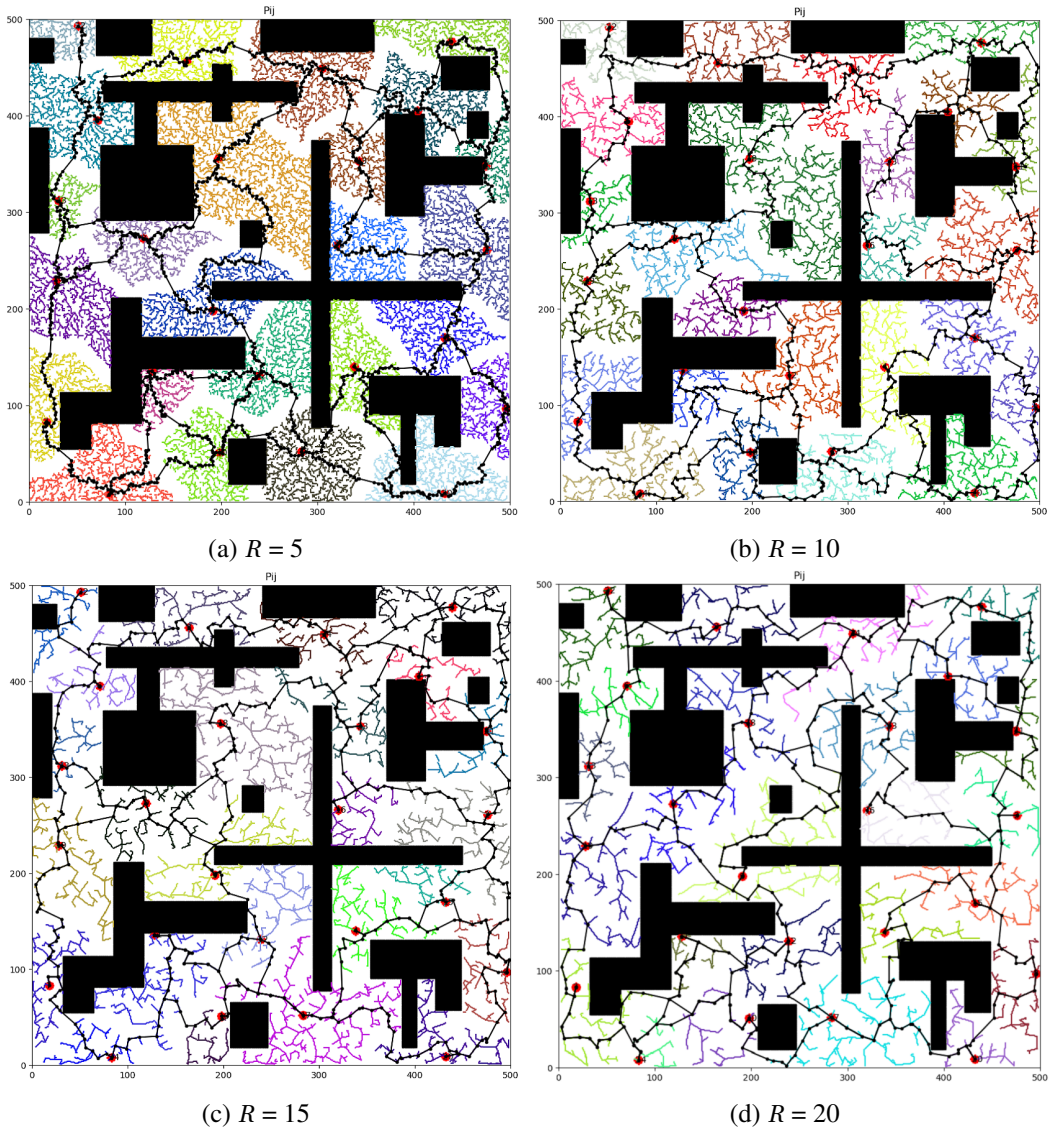
4. Comportamiento del algoritmo debido al cambio del parámetro *qgoal_probability*. En función de la probabilidad que se asigne, los árboles tenderán o no a crecer directamente hacia el punto objetivo, haciendo así que se explore más o menos el entorno.

5.4.1 Ejemplo de ejecución del algoritmo *RRT-MO*

En este subapartado se vuelve a mostrar un primer ejemplo de ejecución para aclarar como funciona el algoritmo *RRT-MO*.

Una ejecución de *RRT-MO*, al igual que *SFF*, consta de tres fases:

- Creación de los árboles. Se muestra el crecimiento de los árboles de forma secuencial (en modo *RANDOM* O *PROXIMITY*) hasta que se alcanza el último nodo posible. Una vez alcanzado, se genera un árbol desde él hasta el nodo inicial, para cerrar el bucle y conseguir así equiparar el algoritmo *RRT-MO* con el camino que calcula *TSP* sobre el *roadmap* obtenido una vez ejecutado *SFF*.



(a) $R = 5$ (b) $R = 10$

(c) $R = 15$ (d) $R = 20$

Figura 5.24: Diversos ejemplos de ejecución de la prueba del incremento del valor del parámetro R.

- Creación de la matriz P_{ij} o *roadmap* y suavizado de las rutas. De forma similar a *SFF*, se una vez finalizada la fase de construcción de los árboles, se obtiene la matriz P_{ij} conteniendo así los caminos entre cada uno de los puntos objetivo.

- Construcción de la ruta a seguir, se concatenan los caminos obtenidos anteriormente para así formar una ruta que pasa por todos los puntos y vuelve al punto inicial.

El entorno que se ha decidido utilizar es el mismo que se ha utilizado para *SFF*, éste se encuentra en la figura 4.14.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.8

EJEMPLO DE EJECUCIÓN DEL ALGORITMO RRT	
Parámetros	**Valor**
number_of_targets	12
min_dist_between_targets	135
mode	PROXIMITY
step_size	15
qgoal_probability	4%
tree	0 = TREE.HH

Tabla 5.8: Parámetros utilizados para realizar el ejemplo de ejecución del algoritmo *RRT-MO*.

RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS	
Parámetros	**Valor**
min_dist_between_targets	30
mode	0 = RANDOM 1 = PROXIMITY
step_size	15
qgoal_probability	4
min_targets	2
max_targets	50
step_targets	1
number_of_executions	10
tree	0 = TREE.HH

Tabla 5.9: Parámetros utilizados para realizar la prueba de incremento del número de puntos objetivo *RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS*

El crecimiento de los árboles se puede observar en la figura 5.25. Notar que es exactamente el mismo caso que el presentado para *SFF*, el cual se puede observar en la figura 5.12. Los árboles crecen secuencialmente desde un punto a otro, alcanzándolos por proximidad. En la figura 5.26 se muestra el entramado de árboles resultante y la matriz P_{ij} con el consecuente *roadmap* formado (ya suavizado).

Finalmente, en la figura 5.27 se muestra la ruta a seguir.

5.4.2 Comportamiento de *RRT-MO* debido al cambio del número de puntos objetivo

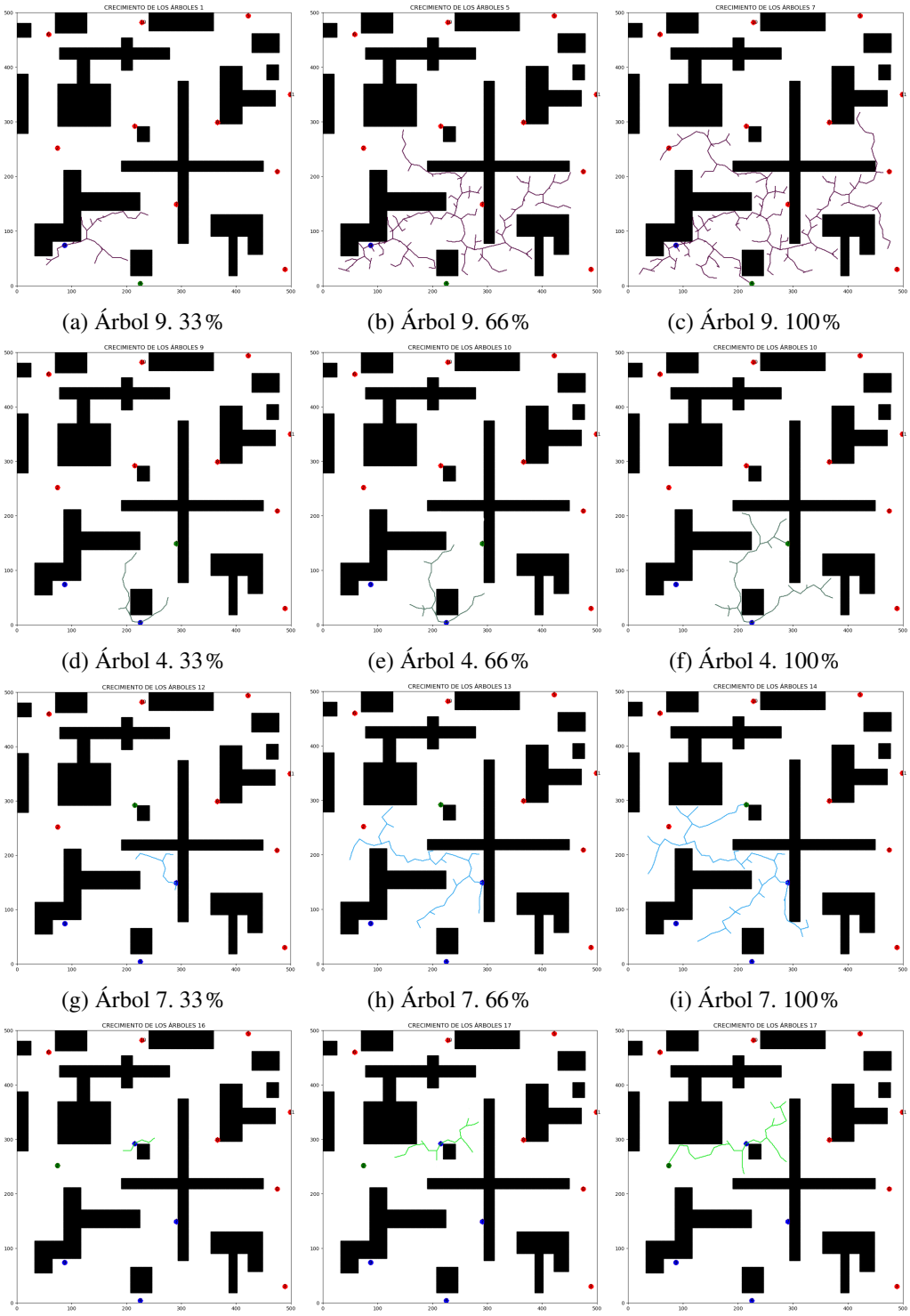
En esta prueba se verifica el comportamiento del algoritmo en función del número de puntos objetivo sobre los que se ejecuta.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.9. Se establece una distancia mínima de 30 celdas entre los puntos objetivo, se fijan también los valores de los parámetros *step_size* y *qgoal_probability*. Se itera entre un número de puntos objetivo delimitado por los parámetros *min_targets* y *max_targets* avanzando *step_targets*. En cada iteración, estos parámetros se definen gracias al modo de test *RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS* seleccionado en el archivo *params_RRTSFF.ini*.

Además, mencionar que durante esta prueba y las restantes del algoritmo *RRT-MO*, se realiza cada ejecución 2 veces, dependiendo de si se utiliza el modo *RANDOM* o el modo *PROXIMITY*.

En los gráficos de la figura 5.28 se observan los resultados obtenidos:

- Respecto al tiempo de ejecución (5.28a). Se observa como, igual que en *SFF*, a



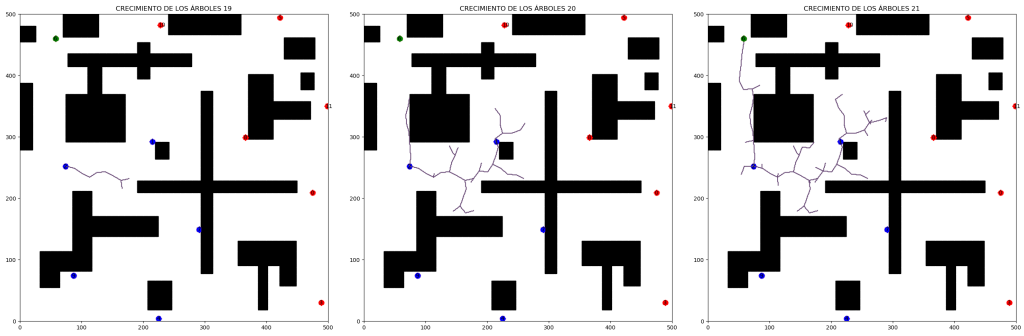
(a) Árbol 9. 33 % (b) Árbol 9. 66 % (c) Árbol 9. 100 %

(d) Árbol 4. 33 % (e) Árbol 4. 66 % (f) Árbol 4. 100 %

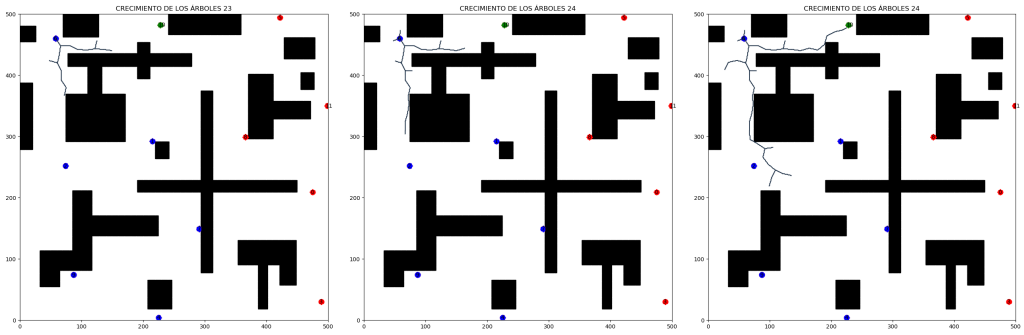
(g) Árbol 7. 33 % (h) Árbol 7. 66 % (i) Árbol 7. 100 %

(j) Árbol 1. 33 % (k) Árbol 1. 66 % (l) Árbol 1. 100 %

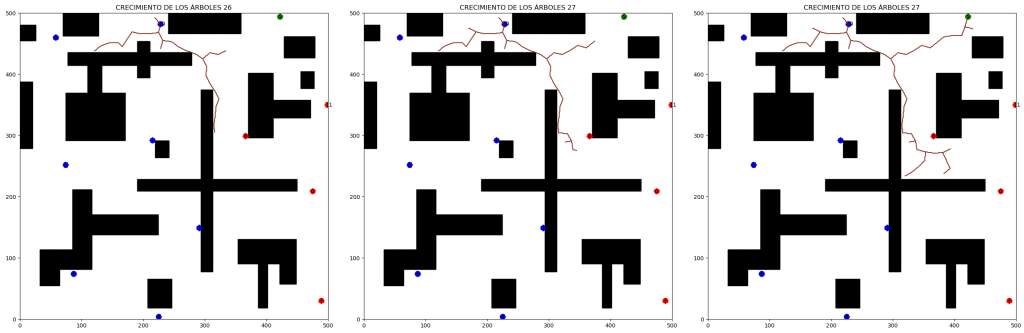
Figura 5.25: Ejemplo de crecimiento de los árboles durante la ejecución del algoritmo *RRT-MO*.



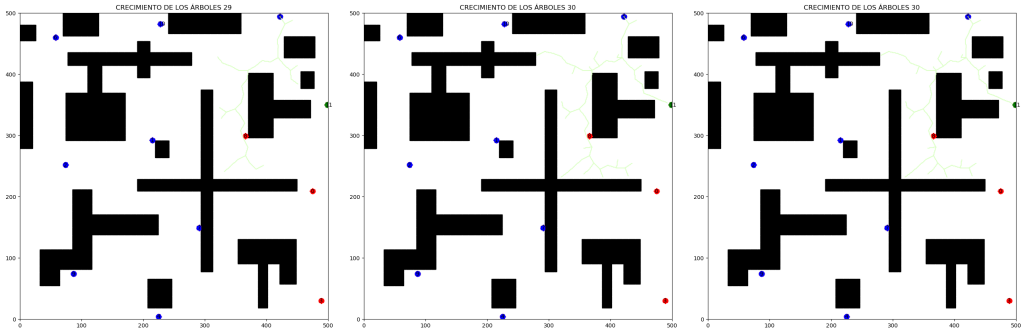
(m) Árbol 2. 33 % (n) Árbol 2. 66 % (o) Árbol 2. 100 %



(p) Árbol 6. 33 % (q) Árbol 6. 66 % (r) Árbol 6. 100 %

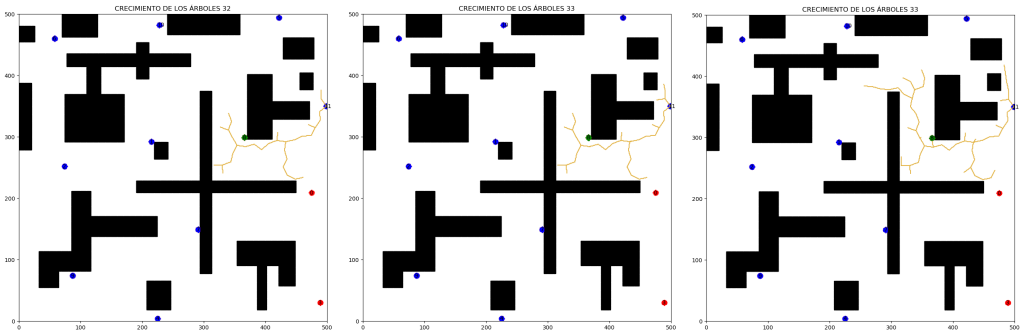


(s) Árbol 10. 33 % (t) Árbol 10. 66 % (u) Árbol 10. 100 %

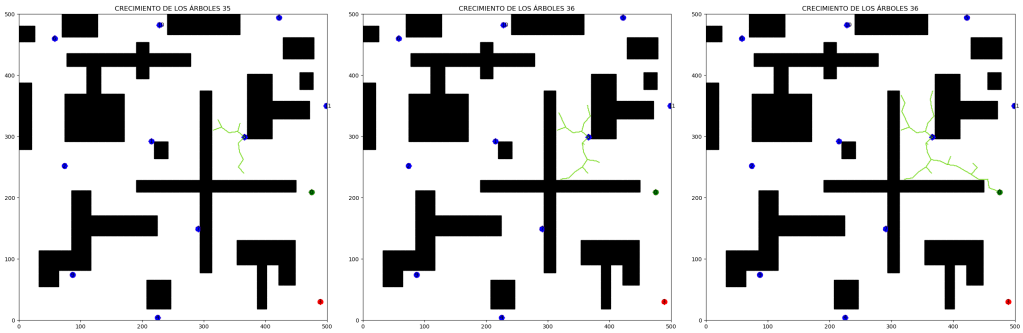


(v) Árbol 5. 33 % (w) Árbol 5. 66 % (x) Árbol 5. 100 %

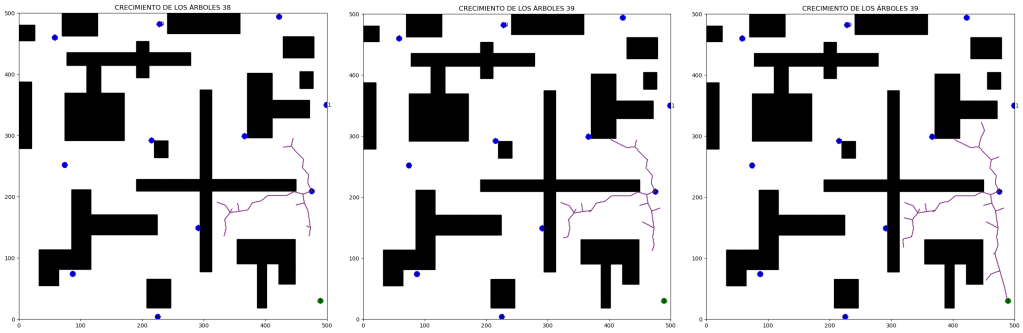
Figura 5.25: Ejemplo de crecimiento de los árboles durante la ejecución del algoritmo *RRT-MO*.



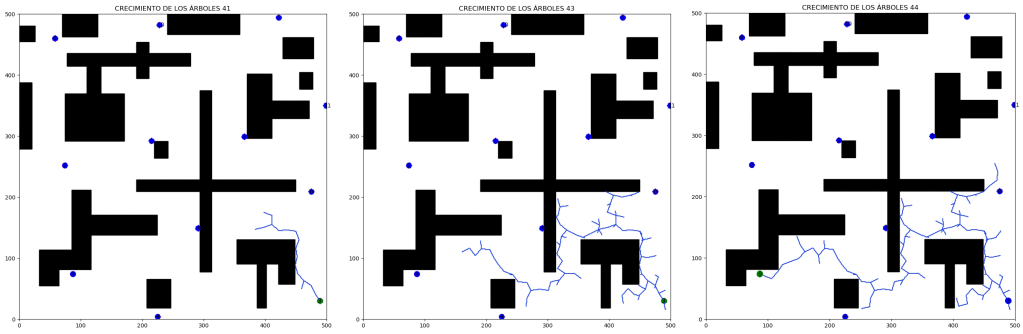
(y) Árbol 11. 33 % (z) Árbol 11. 66 % (aa) Árbol 11. 100 %



(ab) Árbol 8. 33 % (ac) Árbol 8. 66 % (ad) Árbol 8. 100 %



(ae) Árbol 0. 33 % (af) Árbol 0. 66 % (ag) Árbol 0. 100 %



(ah) Árbol 3. 33 % (ai) Árbol 3. 66 % (aj) Árbol 3. 100 %

Figura 5.25: Ejemplo de crecimiento de los árboles durante la ejecución del algoritmo *RRT*.

RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
mode	0 = RANDOM 1 = PROXIMITY
qgoal_probability	4
min_step_size	5
max_step_size	20
step_step_size	1
number_of_executions	10
tree	0 = TREE.HH

Tabla 5.10: Parámetros utilizados para realizar la prueba de incremento de la variable *step_size RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE*

medida que el número de puntos objetivo aumenta, el tiempo de ejecución también lo hace.

- Respecto al coste del camino calculado (5.28b). El coste del camino resultante también aumenta debido a que a mayor cantidad de nodos, más caminos existirán, por lo que se tiene que recorrer una mayor cantidad de espacio para poder visitarlos todos.

Hay que tener en cuenta que, dependiendo de si el algoritmo se ejecuta en modo *RANDOM* o en modo *PROXIMITY*, se observa como el resultado es distinto. Se obtiene un tiempo y coste más elevado cuando el algoritmo se ejecuta en modo *RANDOM*. Con esta prueba es suficiente para observar que se comporta de esta forma, pero aún así se ha querido dejar reflejado durante las pruebas restantes.

Se observa como el entramado de árboles resultante se vuelve cada vez más denso a medida que se aumenta el número de puntos objetivo. Los nodos cambian de color dependiendo de la asignación que tengan en cada momento:

- Rojo. Nodos inexplorados o que aún no han sido alcanzados.
- Azul. Nodo raíz o nodo *start* del árbol que está creciendo.
- Verde. Nodo objetivo o *goal*, el nodo que desea alcanzar el árbol para finalizar su búsqueda.

5.4.3 Comportamiento de *RRT-MO* debido al cambio del parámetro *step_size*

En esta prueba se analiza el comportamiento del algoritmo en función del valor que se le asigne al parámetro *step_size*.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.10. Se ha fijado nuevamente el número de puntos objetivo a 25 con una distancia mínima entre ellos de 80 celdas y también se ha fijado el valor del parámetro *qgoal_probability*. En este caso se itera con el parámetro *step_size* (*min_step_size*,*step_step_size*, *max_step_size*) gracias al modo de test *RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE*.

En los gráficos de la figura 5.30 se observan los resultados obtenidos:

RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB	
Parámetros	**Valor**
number_of_targets	25
min_dist_between_targets	80
mode	0 = RANDOM 1 = PROXIMITY
step_size	15
min_qgoal_probability	2
max_qgoal_probability	99
step_qgoal_probability	1
number_of_executions	10
tree	0 = TREE.HH

Tabla 5.11: Parámetros utilizados para realizar la prueba de incremento de la variable *step_size RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB*

- Respecto al tiempo de ejecución (5.30a). Éste disminuye drásticamente para posteriormente estabilizarse a medida que se aumenta el valor de *step_size*. Igual que ocurre con *SFF*, esto se debe a que las ramas son más largas y por lo tanto los árboles llegan al punto objetivo más rápidamente

- Respecto al coste del camino calculado (5.30b). Por el resultado obtenido, parece ser que aumentar el valor del parámetro *step_size* no repercute sobre el coste final del camino que se tendrá que recorrer, se mantiene estable a medida que el valor aumenta.

El ejemplo de ejecución se encuentra en la figura 5.31. Está realizado en modo *proximity* y en todas las figuras el nodo *start* y el nodo *goal* son los mismos, para así comparar mejor visualmente como aumenta el tamaño de las ramas.

5.4.4 Comportamiento de *RRT-MO* debido al cambio del parámetro *qgoal_probability*

En esta prueba se analiza el comportamiento del algoritmo en función del valor que se le asigne al parámetro *qgoal_probability*.

Los parámetros utilizados para realizar esta prueba son los que se muestran en la tabla 5.11. Se ha fijado nuevamente el número de puntos objetivo a 25 con una distancia mínima entre ellos de 80 celdas y también se ha fijado el valor del parámetro *step_size*. En este caso se itera con el parámetro *qgoal_probability* (*min_qgoal_probability*,*step_qgoal_probability*, *max_qgoal_probability*) gracias al modo de test
RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB.

En los gráficos de la figura 5.32 se observan los resultados obtenidos:

- Respecto al tiempo de ejecución (5.32a). La mayoría del tiempo se mantiene estable hasta que el valor de *qgoal_prob* se acerca al 100 %. Si se establece *qgoal_prob* a un valor muy elevado, se asignaría q_{rand}=q_{goal} prácticamente todo el tiempo, hecho que hace que no se pueda explorar correctamente el espacio y que si nos topamos con un obstáculo, el árbol no sea capaz de crecer ya que el nodo q_{new} no se podría crear debido a que no es posible avanzar la distancia *step_size*. Establecerlo al 100 % es un error y de todas formas no se recomienda superar valores cercanos al 5 % para evitar este tipo de problemas y así, conseguir que los árboles crezcan correctamente.

- Respecto al coste del camino calculado (5.32b). Éste no se altera a medida que se aumenta el valor del parámetro, parece ser que realmente para este tipo de entorno poblado de obstáculos el hecho de aumentarlo no aporta ningún beneficio.

El ejemplo de ejecución se observa en la figura 5.33. Aunque la prueba se haya realizado en el entorno de la figura 5.3, el cual está poblado por obstáculos, para mostrar el correcto funcionamiento de este parámetro se ha decidido hacerlo en un entorno muy sencillo el cual no tenga obstáculos y en el que solo coloquemos dos puntos objetivos. De esta forma, se puede comprobar para ver cómo a medida que se aumenta el valor del parámetro *qgoal_prob* se acaba obteniendo una línea recta que une los dos puntos.

5.5 Comparación de los algoritmos *SFF* y *RRT-MO*.

En este apartado se realiza la comparación entre los algoritmos *SFF* y *RRT-MO*. Los dos apartados anteriores han servido de antesala para observar como, en un entorno sencillo como es el de la figura 5.3, el algoritmo *RRT-MO* es más efectivo que *SFF*.

Recapitulando sobre lo visto en el apartado anterior y para realizar una primera pero efectiva comparación entre los dos algoritmos, se han realizado dos gráficos (ver figuras 5.34 y 5.35) en los que se compara el tiempo de ejecución y el coste en función de las modificaciones de puntos objetivo (apartado 5.3.3 para *SFF* y apartado 5.4.2 para *RRT-MO*) y los parámetros *R* y *step_size* respectivamente para los algoritmos *SFF* y *RRT-MO* (apartado 5.3.6 para *SFF* y apartado 5.4.3 para *RRT-MO*).

De esta forma, juntando la información obtenida en cada una de las pruebas se puede observar como se comportan estos dos algoritmos en el entorno 5.3.

Mencionar que para *RRT-MO* solo se ha escogido la versión que funciona por proximidad.

Respecto a la prueba de los puntos objetivo, en los gráfico de las figuras se observa que:

- Primeramente, respecto al tiempo de ejecución (ver figura 5.34a), se observa como el algoritmo *RRT-MO* es más rápido que *SFF*, . El tiempo de ejecución del algoritmo *SFF* tiende a aumentar a medida que se incrementa el número de puntos objetivo. En el mismo gráfico se observa el tiempo de cada uno de los extremos de las dos curvas, observando así el desfase existente entre cada algoritmo.

- En cambio, respecto al coste del camino que se debe recorrer (ver figura 5.34b), se observa como ocurre lo contrario y obviamente, gracias a la aplicación de *TSP*, se obtiene un coste muy inferior a medida que se aumenta el número de puntos objetivo.

Respecto a la prueba del incremento de los parámetros *R* (*SFF*) y *step_size* (*RRT-MO*):

- Respecto al tiempo de ejecución (ver figura 5.35a), se observa que en el caso de *SFF* disminuye mucho a medida que los parámetros aumentan su valor, pero de todas formas nunca llega a equipararse a los valores de *RRT-MO*.

- Por otro lado, de nuevo y aunque en este caso se mantenga estable, se observa como el coste (ver figura 5.35b) siempre es inferior en el caso de *SFF*.

Vista esta información, se observa como respecto al coste que se obtiene, si que es importante la utilización de *SFF* y *TSP* ya que se obtienen mejores caminos. En cambio, respecto al tiempo de ejecución, el algoritmo *RRT-MO* ofrece mejores prestaciones.

Por lo tanto, se puede llegar a la conclusión que en casos en los que el entorno contenga espacios relativamente amplios y zonas las cuales no contengan ningún tipo de complicación, el algoritmo *RRT-MO* ofrece mejores prestaciones respecto al tiempo de ejecución.

Lo que se desea realizar en los dos siguientes apartados es mostrar una serie de entornos favorables para cada uno de los algoritmos. De esta forma, se pretende hacer hincapié en las características principales que se mencionan en la publicación de *SFF*. Es decir, entornos en los que los puntos objetivo se encuentren en zonas de difícil acceso y que para llegar hasta ellas existan zonas estrechas. Este es el tipo de entorno ideal para el algoritmo *SFF* y en el cual se pretende obtener un tiempo de ejecución mejor gracias a su utilización.

De todas formas, primero se abarcan los entornos favorables a *RRT-MO* y posteriormente se pasa a los favorables para *SFF*.

5.5.1 Definición de los entornos favorables para el algoritmo *RRT-MO*

Respecto a los entornos favorables a *RRT-MO*, se han querido utilizar los entornos que se observan en la figura 5.36. Notar que son los entornos más utilizados hasta ahora en otras pruebas. Para cada uno de ellos (observar figuras 5.36a y 5.36b) se han establecido 6 puntos objetivo.

Se tratan de entornos con obstáculos entre los cuales existe suficientemente espacio como para no considerarlos pasillos estrechos. Este tipo de entornos en los que *RRT-MO* puede explorar y no es necesario que se introduzca por ningún espacio estrecho son perfectos para sus características.

Se ha procedido a ejecutar los algoritmos *RRT-MO* y *SFF* sobre estos entornos con los parámetros que se observan en la tabla 5.12. Se han querido utilizar entornos con diversos tamaños y por ello se han utilizado diversos valores para cada uno de los parámetros.

5.5.2 Definición de los entornos favorables para el algoritmo *SFF*

Respecto a los entornos favorables a *SFF*, se han querido utilizar los entornos que se observan en la figura 5.37. Poseen características similares:

- Existen obstáculos o paredes situadas de forma específica para así crear zonas aisladas.
- Para llegar hasta los puntos objetivo se tienen que pasar por pasillos realmente estrechos, en este caso se tratan de espacios entre 2 y 4 celdas de ancho.

Se ha procedido a ejecutar los algoritmos *RRT-MO* y *SFF* sobre estos entornos con los parámetros que se observan en la tabla 5.12. En este caso todos los entornos son de 300x300 celdas ya que debido a los pasillos estrechos, se deben modificar los parámetros *R* y *step_size* de cada algoritmo para que sea posible crear ramas lo suficientemente pequeñas como para que quepan por ellos. Si se eligen valores pequeños en entornos relativamente grandes hace que el tiempo de ejecución sea elevado, por ello se ha elegido utilizar entornos con estas características.

	Parámetros		***Tiempo (ms)***		***Coste (celdas)***	
Entornos	**SFF**	**RRT**	**SFF**	**RRT**	**SFF**	**RRT**
Entorno 1	$k = 5$ $d\_tree = 10$ $R = 15$	$step\_size = 15$ $q\_goal\_prob = 4\,\%$	189.71	11.45	291.54	318.14
Entorno 2	$k = 5$ $d\_tree = 10$ $R = 15$	$step\_size = 15$ $q\_goal\_prob = 4\,\%$	163.66	13.08	265.09	277.96
Entorno 3	$k = 5$ $d\_tree = 6$ $R = 5$	$step\_size = 5$ $q\_goal\_prob = 4\,\%$	3288.13	5944.81	619.64	514.34
Entorno 4	$k = 5$ $d\_tree = 5$ $R = 6$	$step\_size = 6$ $q\_goal\_prob = 4\,\%$	1471.98	1591.37	215.85	181.52
Entorno 5	$k = 5$ $d\_tree = 6$ $R = 4$	$step\_size = 6$ $q\_goal\_prob = 4\,\%$	1656.46	27309.76	362.28	328.64
Entorno 6	$k = 5$ $d\_tree = 5$ $R = 5$	$step\_size = 5$ $q\_goal\_prob = 4\,\%$	3243.42	31448.88	536.38	446.66

Tabla 5.12: Tabla comparativa entre los algoritmos *SFF* y *RRT-MO* para diversos tipos de entornos.

5.5.3 Resultados obtenidos de la comparación entre *SFF* y *RRT-MO*

Después de ejecutar cada algoritmo 100 veces con los parámetros mencionados, se obtienen los resultados de tiempo y coste que también se observan en la tabla 5.12.

Se observa como, al menos para el tiempo de ejecución, se cumple la expectativa que se ha comentado anteriormente. En los entornos 1 y 2 se comporta mejor el algoritmo *RRT-MO* y en los entornos 3, 4, 5 y 6 se comporta mejor el algoritmo *SFF*.

- Respecto al tiempo, se observa como se han cumplido las expectativas a la hora de realizar este tipo de prueba y es que, efectivamente, para entornos en donde los puntos objetivo se encuentran en zonas de difícil acceso y en las que existen pasillos realmente estrechos, es más beneficioso crear un árbol desde cada uno de los puntos objetivo, de esta forma es más sencillo salir por dichos pasillos y que los árboles se interconecten más adelante.

- En cambio, respecto al coste, se observa como no necesariamente al utilizar *TSP* se obtendrán mejores valores. Esto es debido a que en la mayoría de casos se obtienen los problemas mencionados en el apartado 4.4.2. En muchas ocasiones nos encontramos con nodos que se quedan aislados y por lo tanto hay que pasar por un mismo camino 2 veces, lo que supone un incremento en el coste del recorrido final que se debe realizar.

En el caso de los entornos 1 y 2 donde hay espacio suficiente como para que existan más conexiones entre los diversos puntos objetivo, *TSP* puede calcular rutas con un mejor coste que si que suelen superar a las que obtiene *RRT-MO* aunque se utilice el modo *PROXIMITY*.

Además, debido a que *RRT-MO* se ejecuta en dicho modo, se encuentran caminos con un coste inferior. Tal y como se ha visto en los resultados del algoritmo *RRT-MO* de los apartados 5.4.2, 5.4.3 y 5.4.4, si lo hubiésemos ejecutado en modo *RANDOM* hubiésemos obtenido resultados mucho peores y posiblemente el coste seria superior al de *SFF*.

Se muestra el resultado de una ejecución de cada caso para observar visualmente los árboles que se han creado y el *roadmap* final que se debe seguir para poder visitar todos los puntos objetivo:

- Respecto al primer entorno, se observan los resultados en la figuras 5.38 (*SFF*) y 5.39 (*RRT-MO*). Se trata de un entorno sencillo en el que el algoritmo *RRT-MO* es mucho más rápido que el algoritmo *SFF*. Por otro lado, *TSP* obtiene un mejor coste.

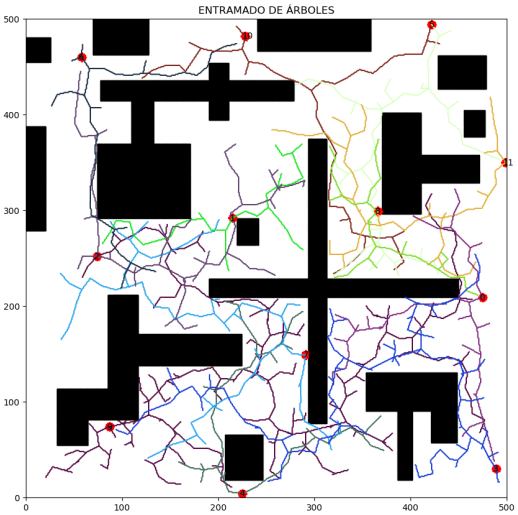
- Respecto al segundo entorno, se observan los resultados en la figuras 5.40 (*SFF*) y 5.41 (*RRT-MO*). Segundo entorno sencillo en el que ocurre lo mismo que en el caso anterior. El algoritmo *RRT-MO* es mucho mejor respecto al tiempo de computación y *TSP* consigue caminos con costes menores.

- Respecto al tercer entorno, se observan los resultados en la figuras 5.42 (*SFF*) y 5.43 (*RRT-MO*). Este es el primero de los entornos desfavorables para el algoritmo *RRT-MO*. Se observa como *SFF* es mejor respecto al tiempo de computación y por otro lado, debido al hecho de que existen nodos que solo tienen una conexión, se necesita rehacer según que caminos para poder visitar todos los nodos. Se observa como un camino se pinta de color azul cuando se rehace, siendo el primer ejemplo la figura 5.42h. El hecho de que se tenga que entrar a cada una de las habitaciones hace que el algoritmo *RRT-MO* sea más lento.

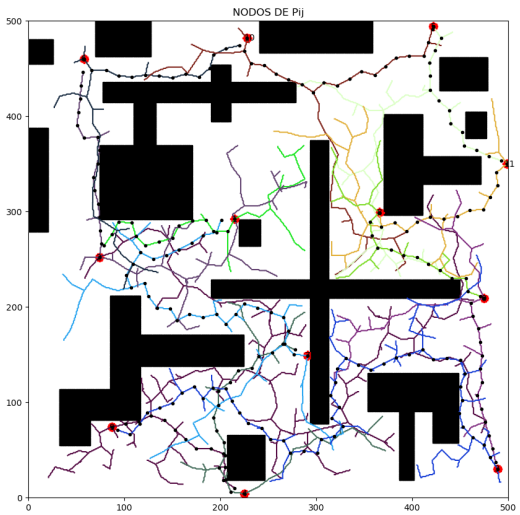
- Respecto al cuarto entorno, se observan los resultados en la figuras 5.44 (*SFF*) y 5.45 (*RRT-MO*). Posiblemente este sea el ejemplo más equiparado que se ha encontrado. Aunque exista un pasillo muy estrecho en el centro y sea la única forma de pasar de un lado a otro del entorno, el hecho de que exista mucho terreno libre hace que el algoritmo *RRT-MO* sea muy rápido. Posiblemente, si no se hubieran colocado 5 puntos objetivo y no se hubiera colocado uno en el centro del pasillo y se hubieran decidido colocar menos, el algoritmo *RRT-MO* sería superior a *SFF* respecto al tiempo de computación.

- Respecto al quinto entorno, se observan los resultados en la figuras 5.46 (*SFF*) y 5.47 (*RRT-MO*). Primer entorno en el que el el uso del algoritmo *SFF* es estrictamente necesario para poder conseguir un tiempo de computación razonable. Se observa como de media de las 100 ejecuciones se obtiene una diferencia de casi 26 segundos. El coste es ligeramente superior para *SFF* pero de todas formas el hecho de que exista tanta diferencia respecto al tiempo de ejecución hace inadmisible el uso de *RRT-MO* en este entorno.

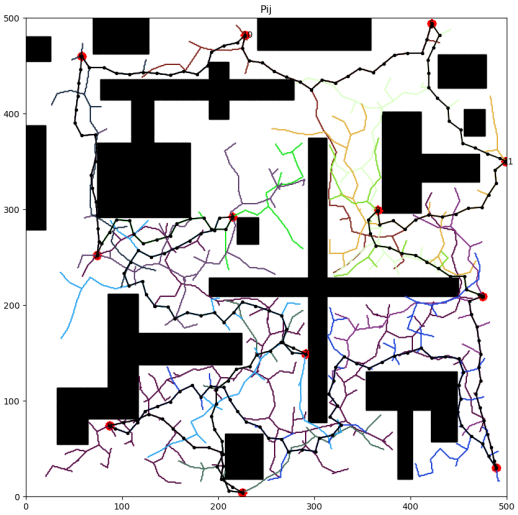
- Respecto al sexto entorno, se observan los resultados en la figuras 5.48 (*SFF*) y 5.49 (*RRT-MO*). Es igual que el caso anterior pero aún más agravado, respecto a la gran cantidad de zonas que se tienen que visitar, se observa como en este caso la diferencia entre los dos tiempos de ejecución es de 28 segundos.



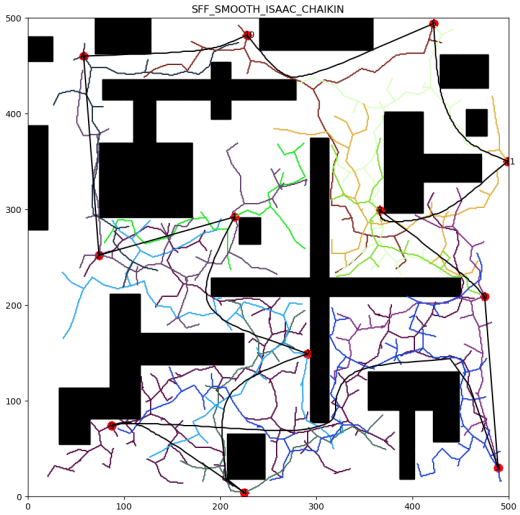
(a) Entramado de árboles.



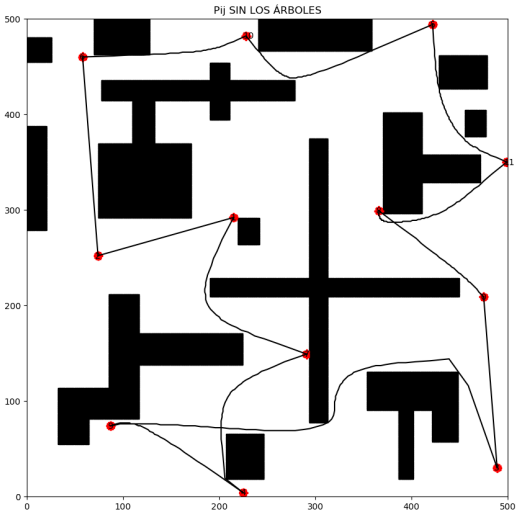
(b) Nodos de P_{ij}.



(c) P_{ij}.

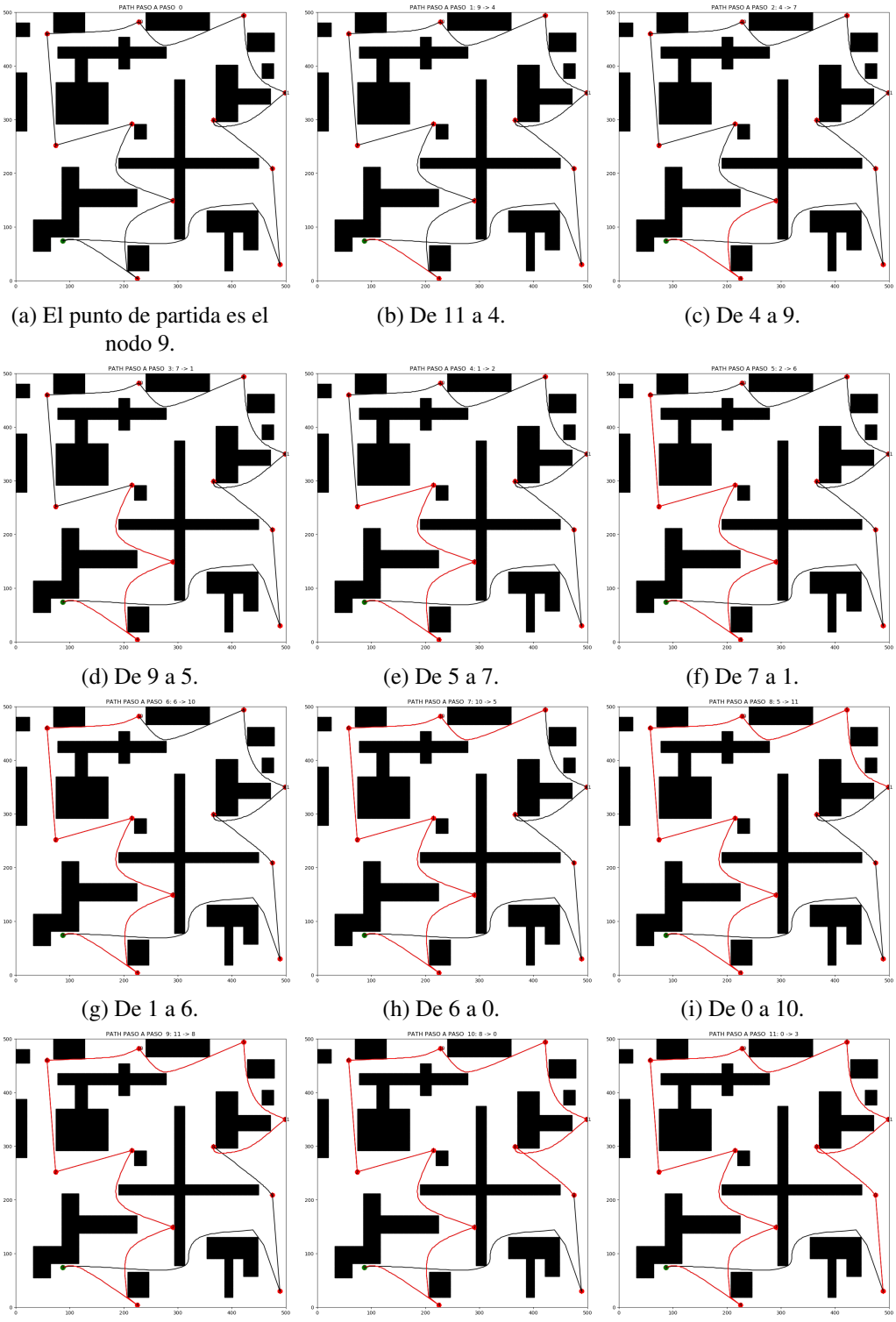


(d) P_{ij} suavizado.



(e) P_{ij} suavizado sin los árboles.

Figura 5.26: Ejemplo de creación de la matriz P_{ij} a partir de los nodos de los árboles y posterior suavizado del *roadmap* resultante.



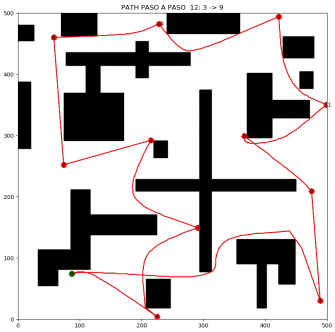
(a) El punto de partida es el nodo 9. (b) De 11 a 4. (c) De 4 a 9.

(d) De 9 a 5. (e) De 5 a 7. (f) De 7 a 1.

(g) De 1 a 6. (h) De 6 a 0. (i) De 0 a 10.

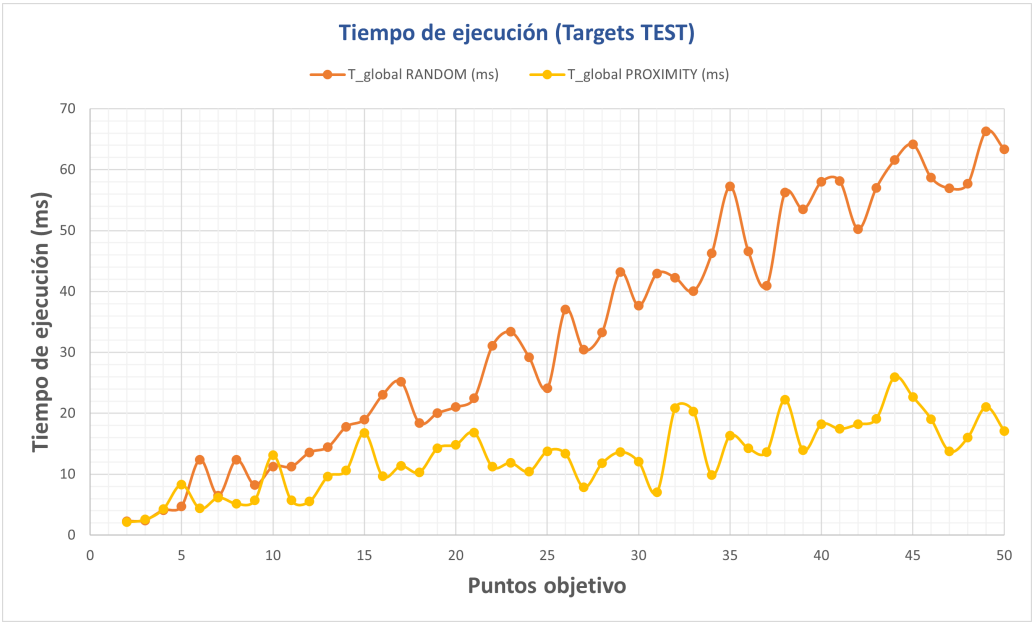
(j) De 10 a 8. (k) De 8 a 2. (l) De 2 a 3.

Figura 5.27: Seguimiento punto a punto del camino formado gracias a la aplicación del algoritmo *RRT-MO* de forma secuencial.

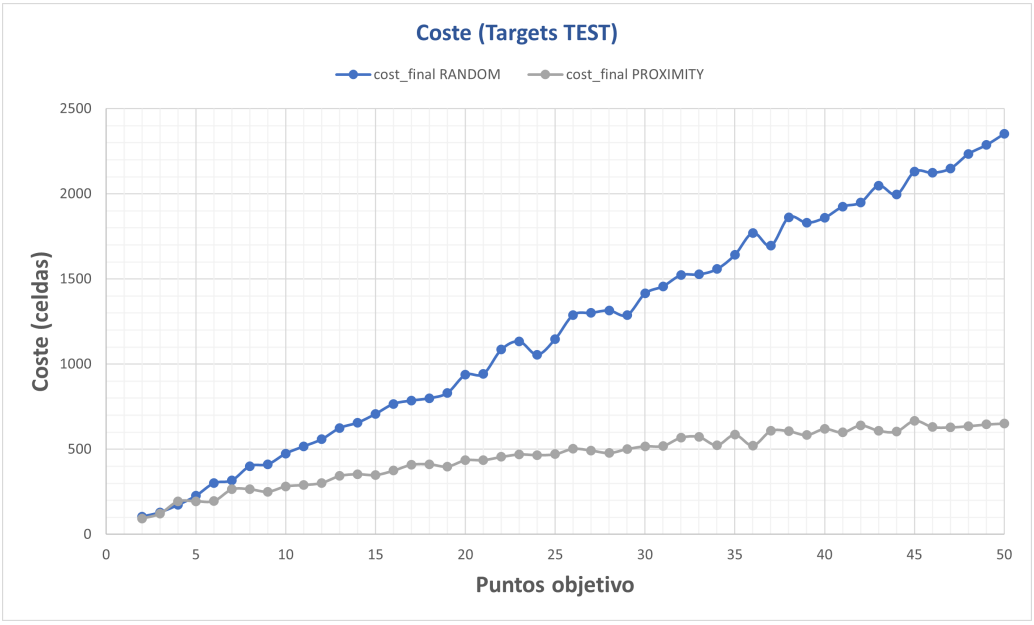


(m) De 3 a 11.

Figura 5.27: Seguimiento punto a punto del camino formado gracias a la aplicación del algoritmo *RRT-MO* de forma secuencial.

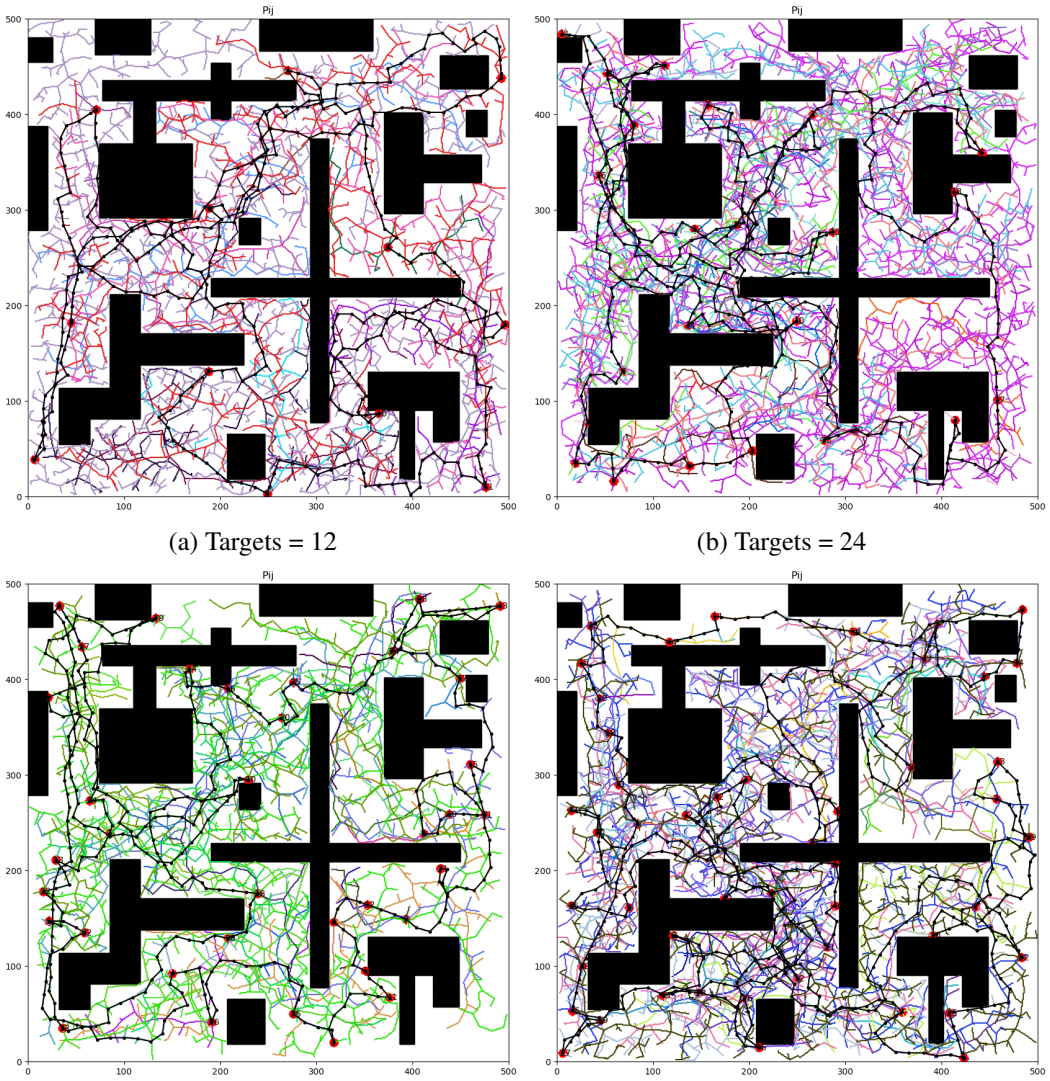


(a)



(b)

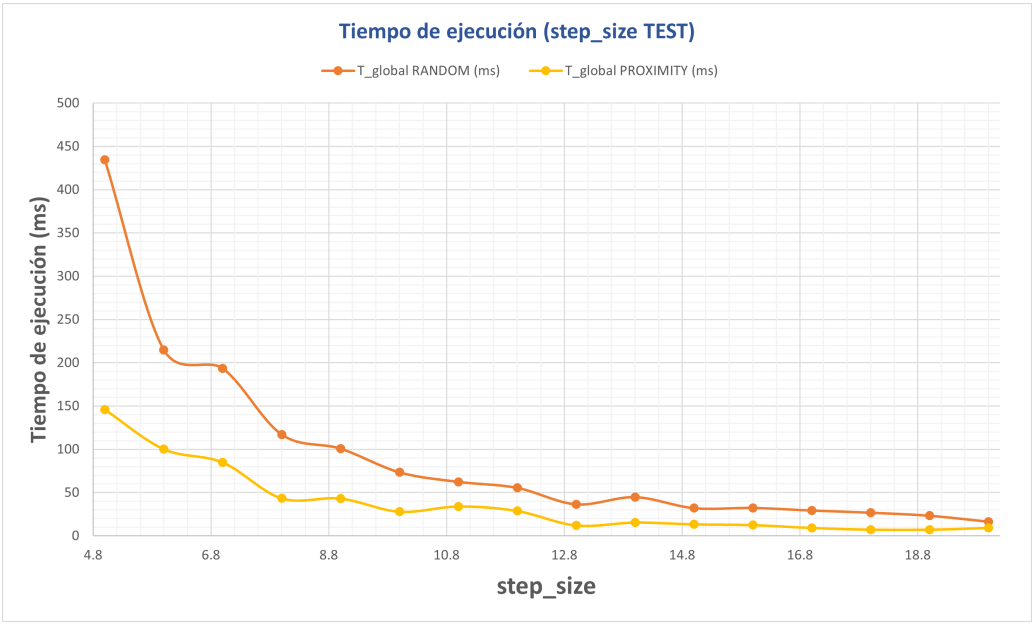
Figura 5.28: Resultados de la prueba del incremento del número de puntos objetivo aplicando el algoritmo *RRT-MO*. Se observa el tiempo de ejecución y el coste del camino en función del número de puntos objetivo.



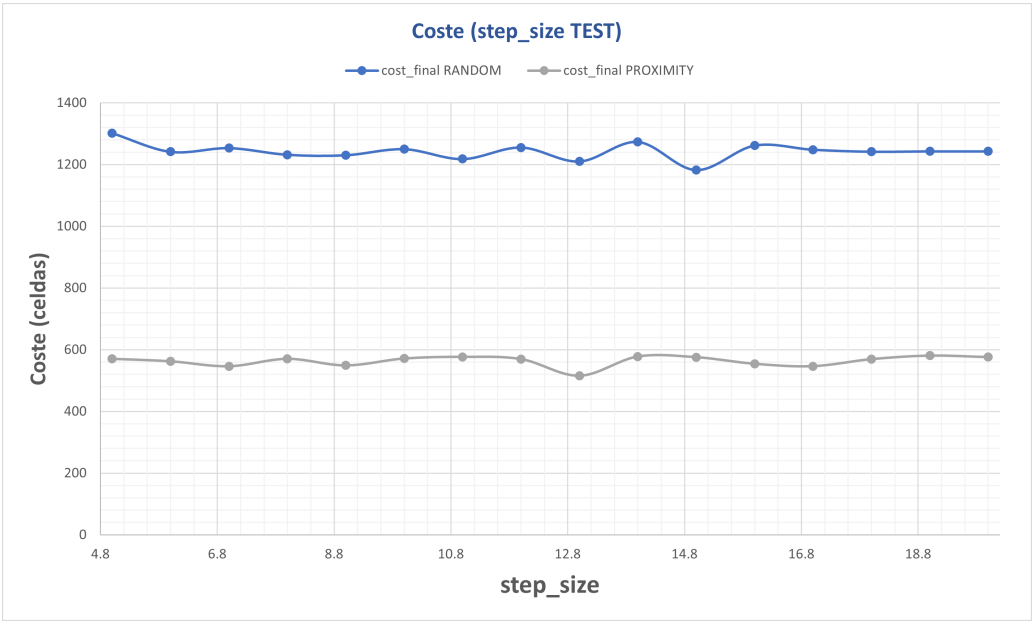
(a) Targets = 12 (b) Targets = 24

(c) Targets = 36 (d) Targets = 48

Figura 5.29: Diversos ejemplos de ejecución de la prueba del incremento del número de puntos objetivo aplicando el algoritmo *RRT-MO*.

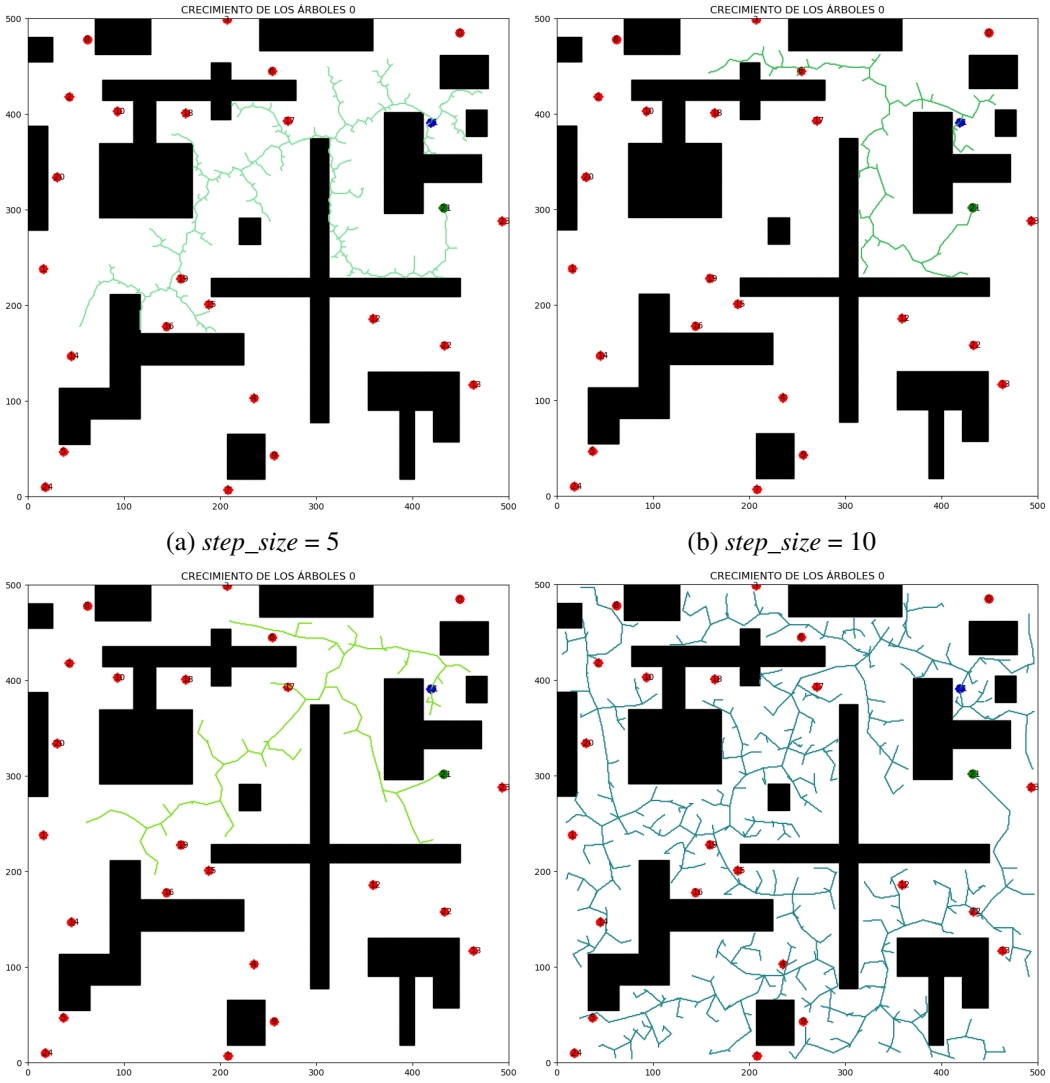


(a)



(b)

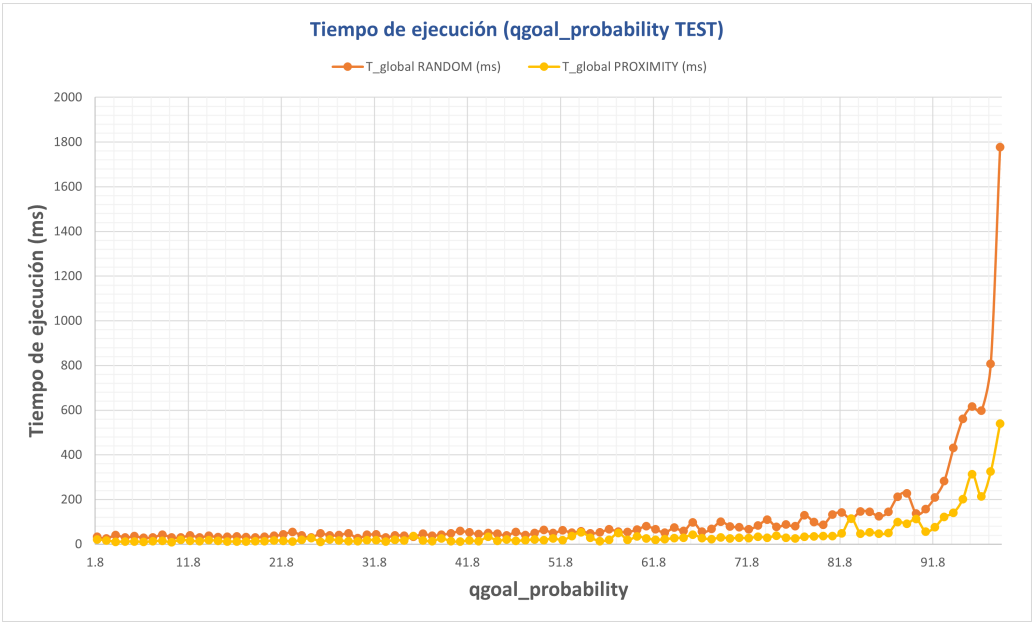
Figura 5.30: Resultados de la prueba del incremento de la variable *step_size*. Se observa el tiempo de ejecución y el coste del camino en función del número de puntos objetivo.



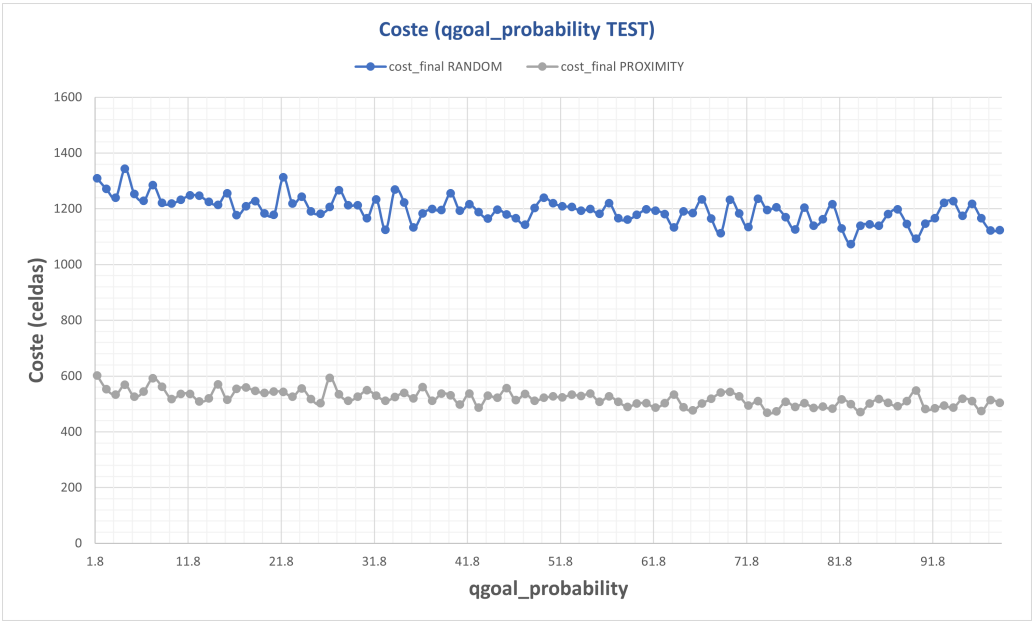
(a) *step_size* = 5 (b) *step_size* = 10

(c) *step_size* = 15 (d) *step_size* = 20

Figura 5.31: Diversos ejemplos de ejecución de la prueba del incremento de la variable *step_size*.

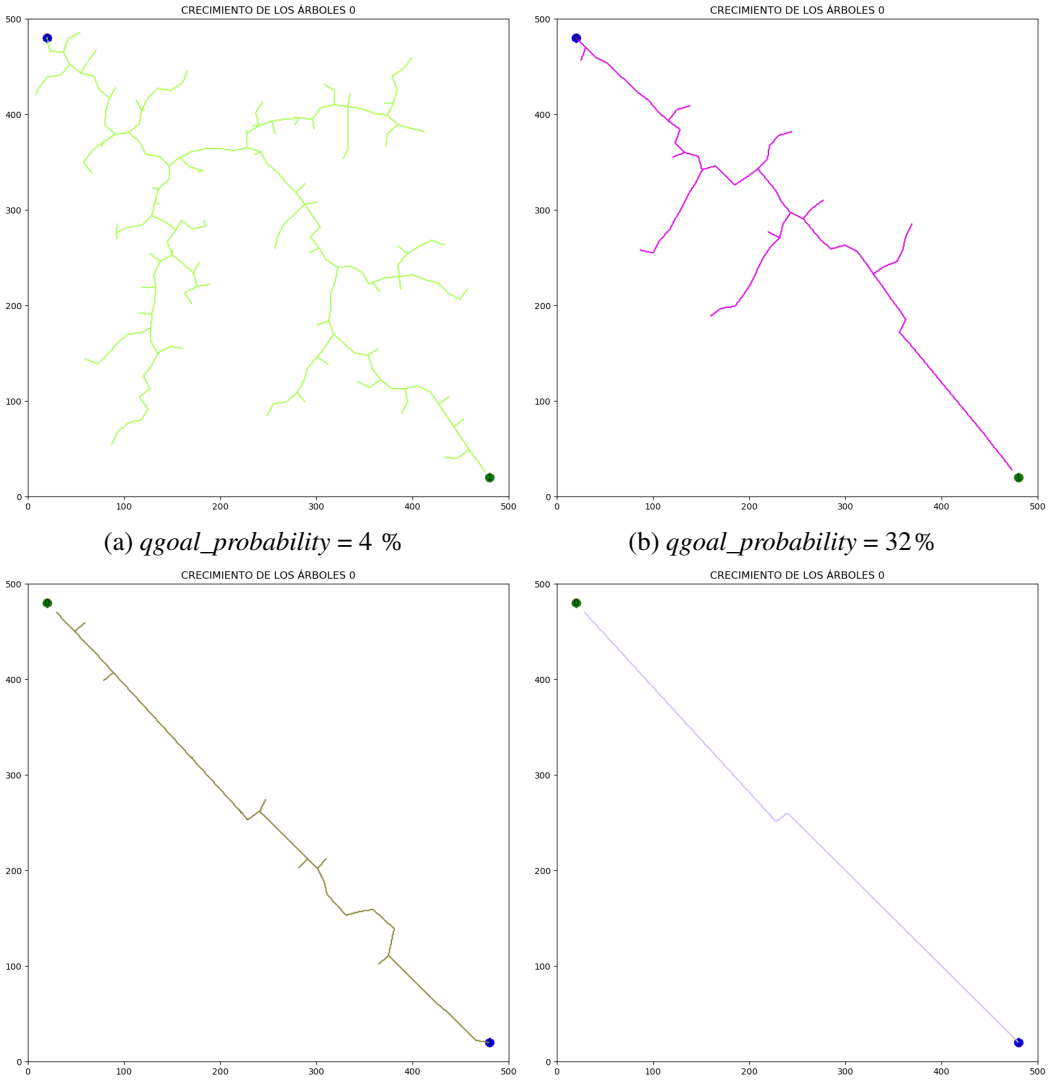


(a)



(b)

Figura 5.32: Resultados de la prueba del incremento de la variable *qgoal_probability*. Se observa el tiempo de ejecución y el coste del camino en función del número de puntos objetivo.



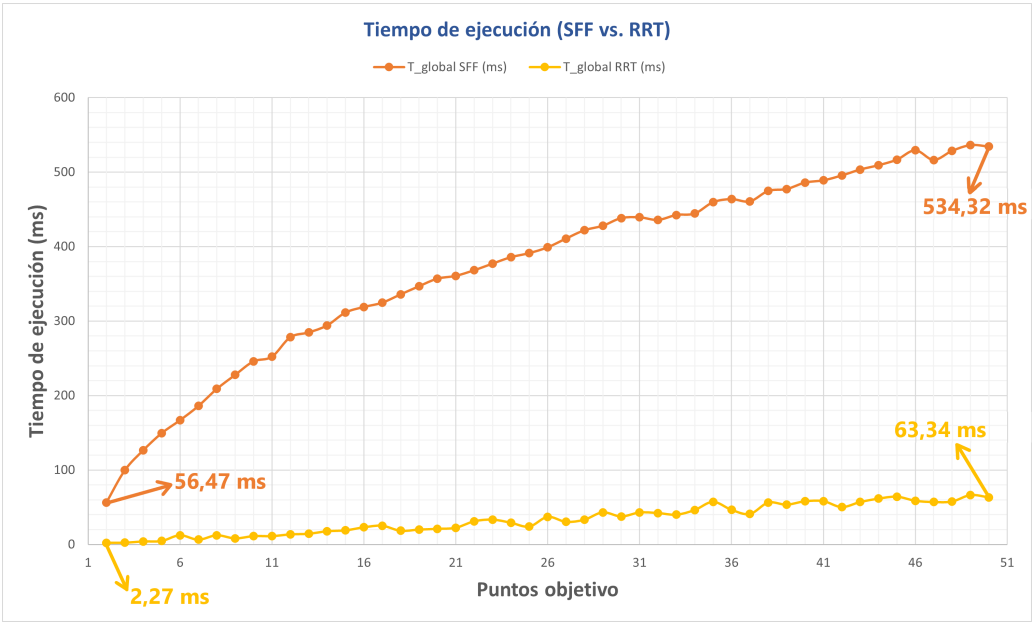
(a) *qgoal_probability* = 4 %

(b) *qgoal_probability* = 32 %

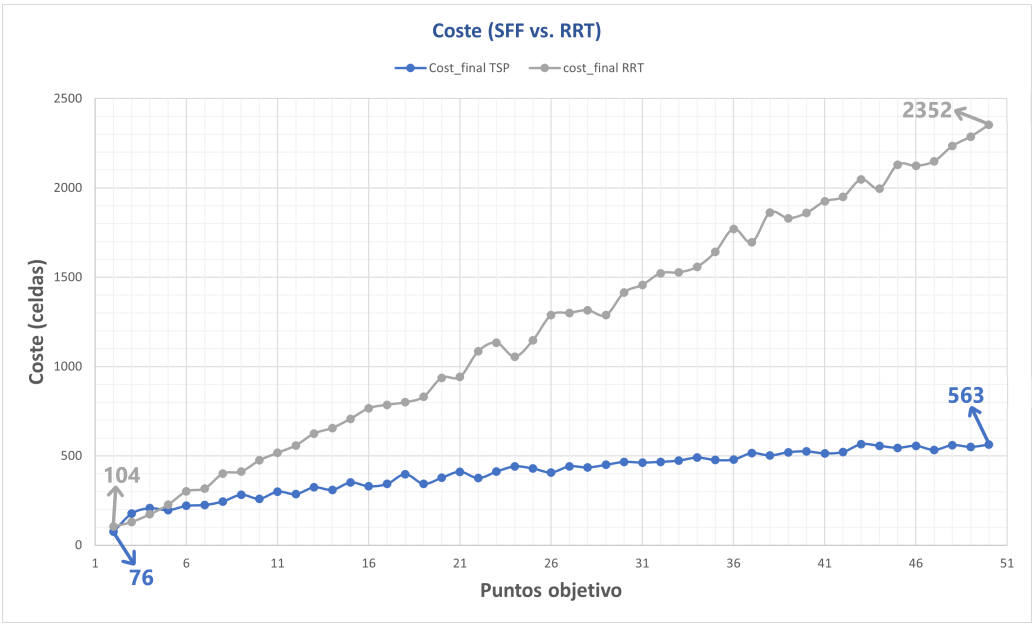
(c) *qgoal_probability* = 64 %

(d) *qgoal_probability* = 96 %

Figura 5.33: Diversos ejemplos de ejecución de la prueba del incremento de la variable *qgoal_probability*.

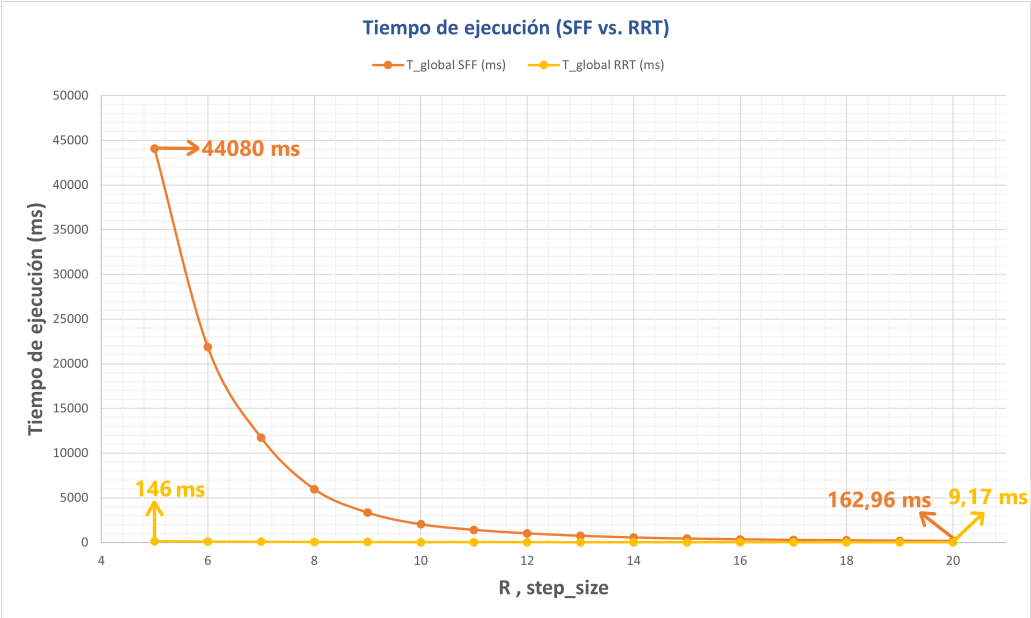


(a)

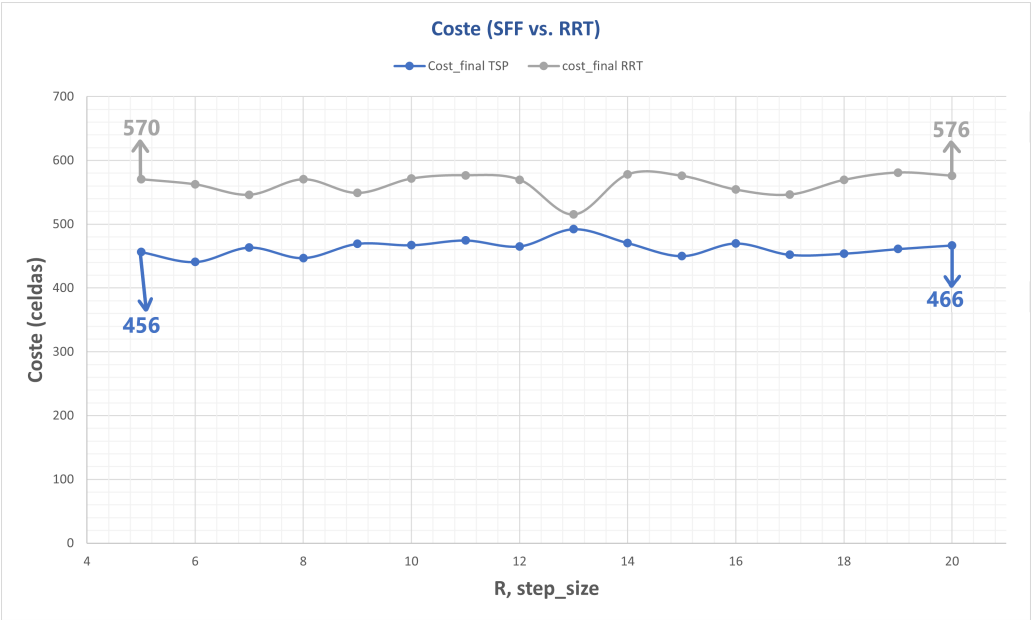


(b)

Figura 5.34: Comparación entre *SFF* y *RRT-MO* respecto al tiempo de ejecución y el coste del camino obtenido en función del incremento del número de puntos objetivo.

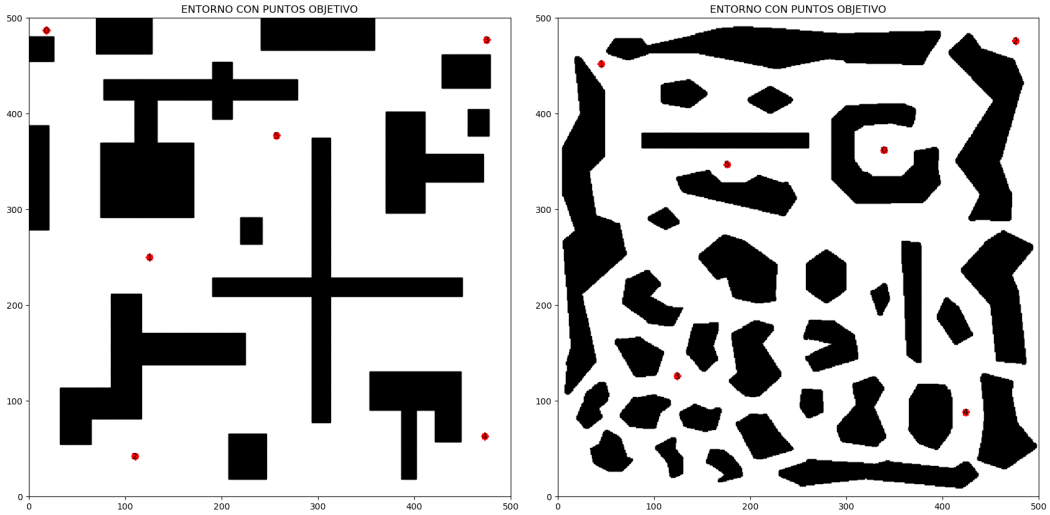


(a)



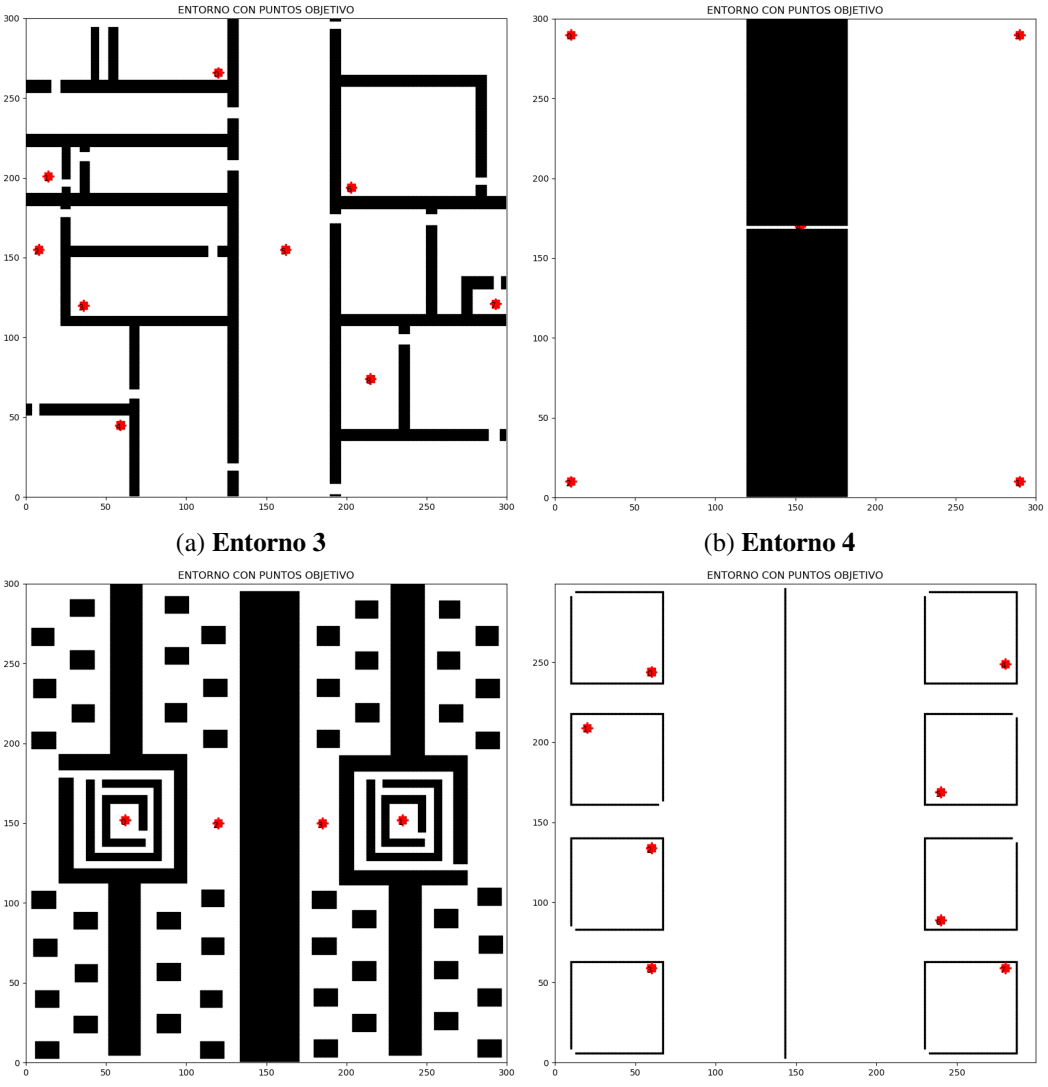
(b)

Figura 5.35: Comparación entre *SFF* y *RRT-MO* respecto al tiempo de ejecución y el coste del camino obtenido en función del incremento de los parámetros *R* y *step_size*.



(a) **Entorno 1** (b) **Entorno 2**

Figura 5.36: Entornos favorables para el algoritmo *RRT-MO*.



(a) **Entorno 3** (b) **Entorno 4**

(c) **Entorno 5** (d) **Entorno 6**

Figura 5.37: Entornos favorables para el algoritmo *SFF*.

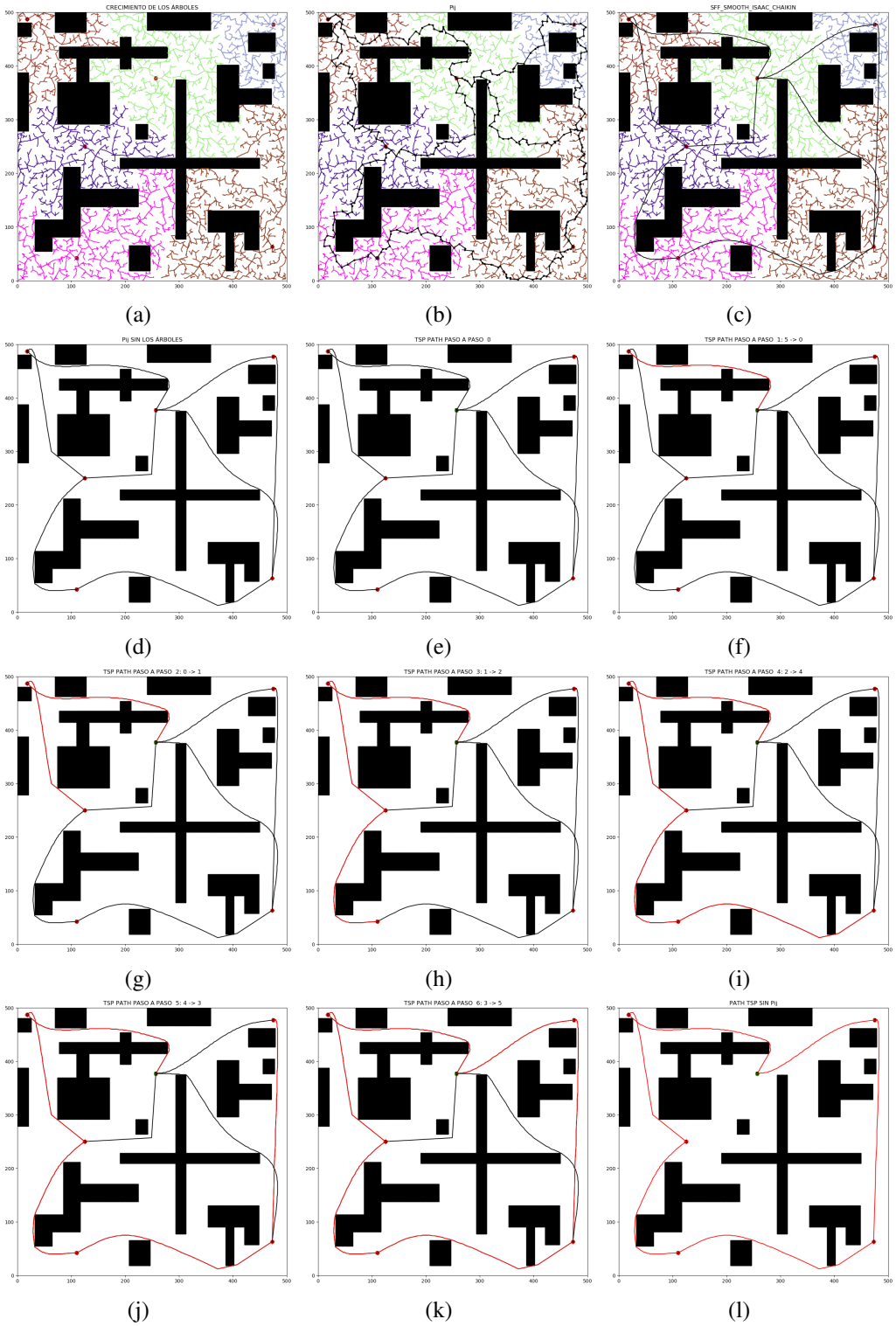


Figura 5.38: Resultado de la ejecución del algoritmo *SFF* en el Entorno 1.

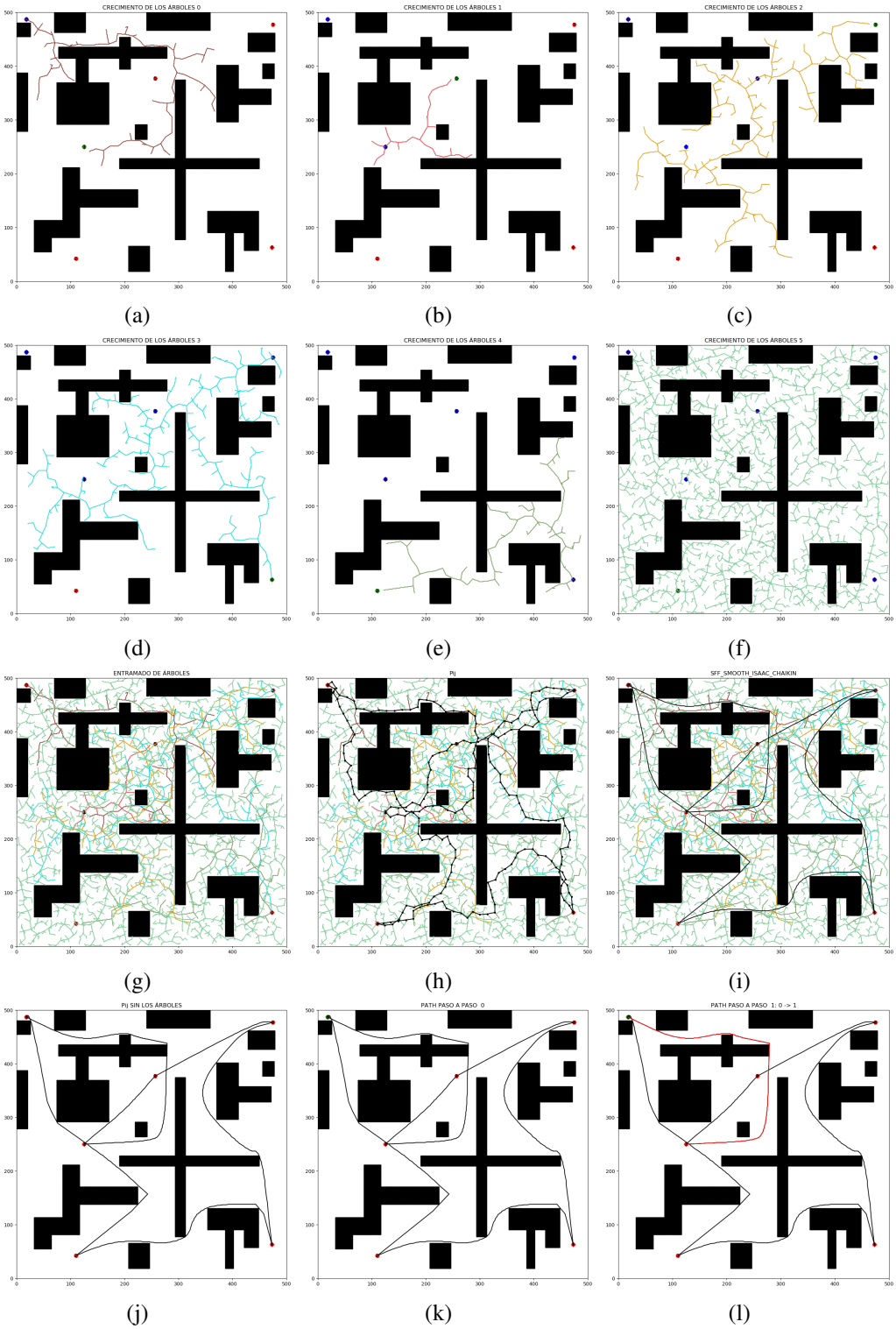
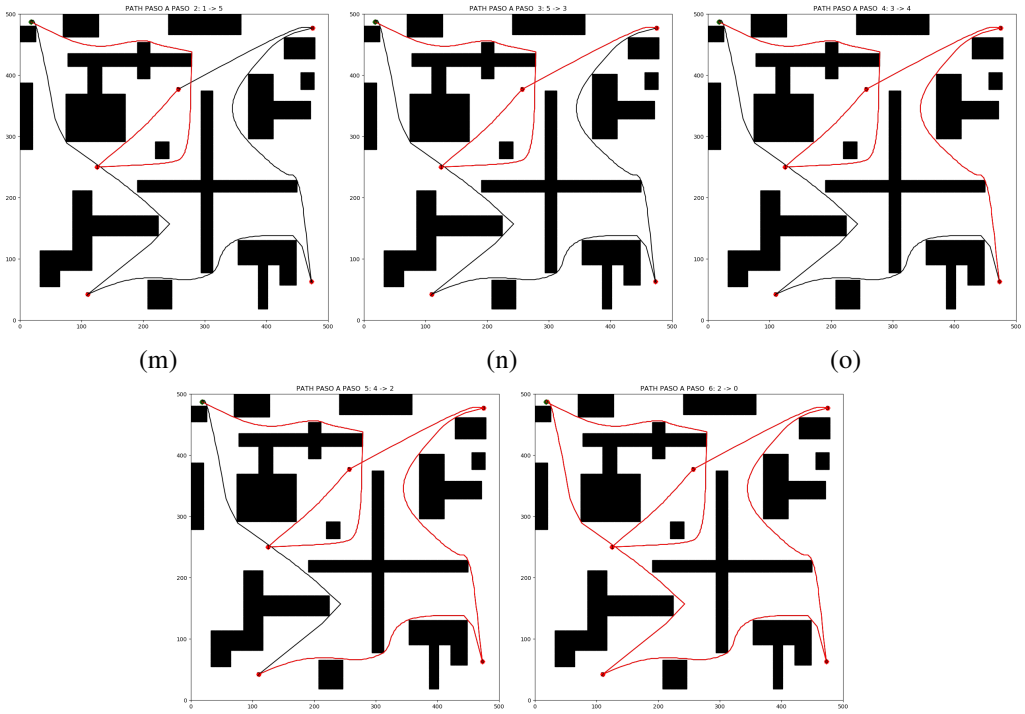


Figura 5.39: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 1.



(m) (n) (o)

(p) (q)

Figura 5.39: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 1.

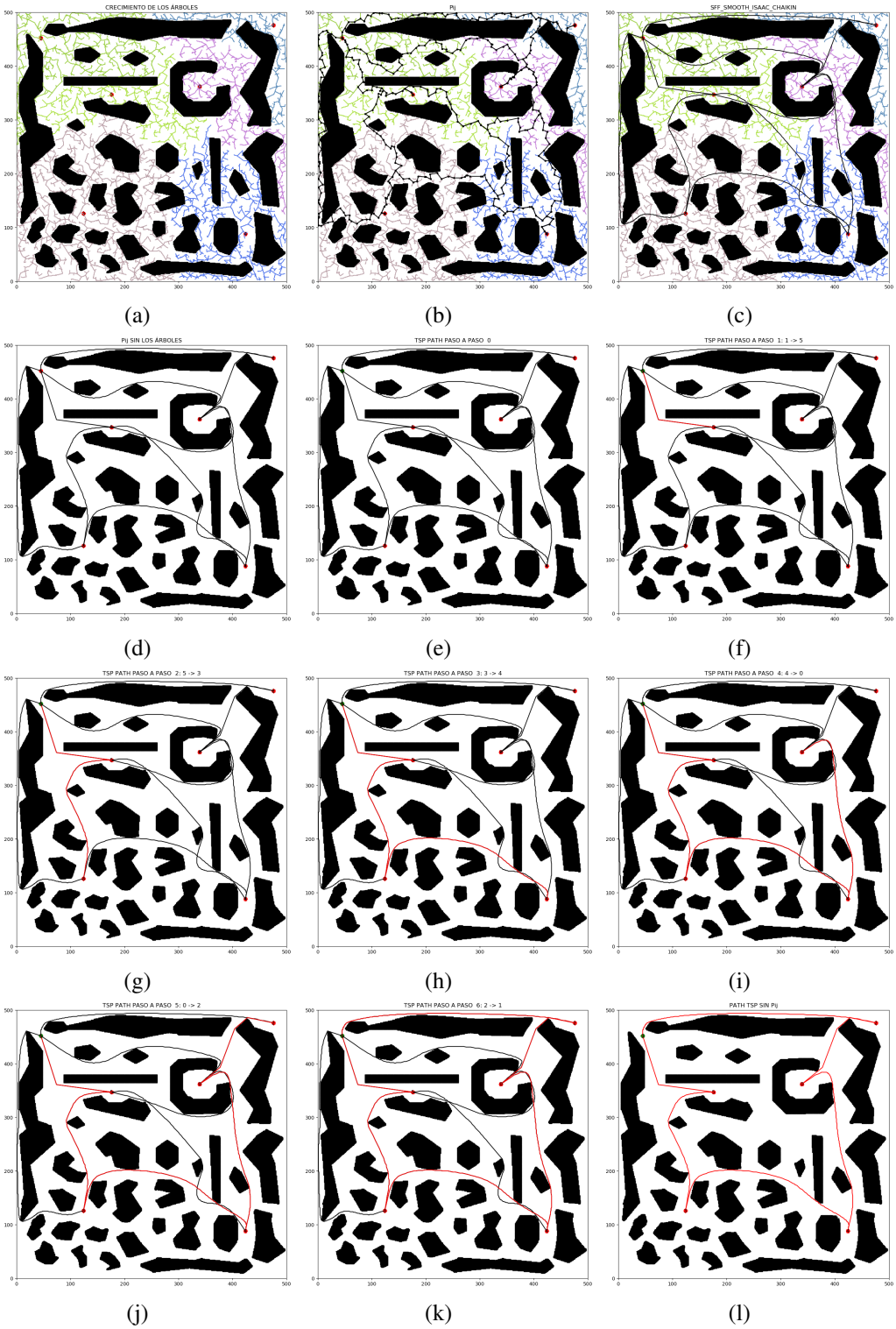


Figura 5.40: Resultado de la ejecución del algoritmo *SFF* en el Entorno 2.

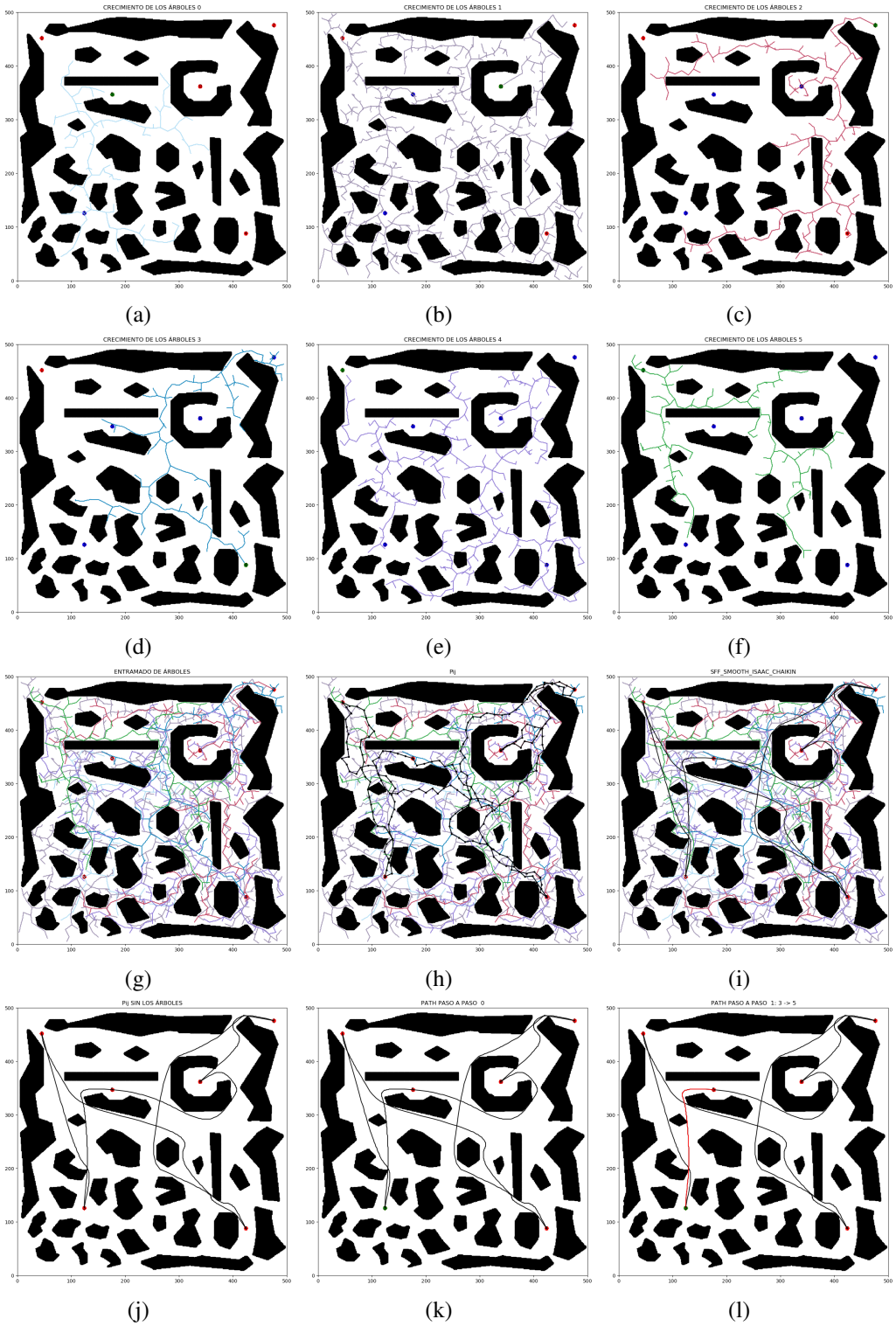
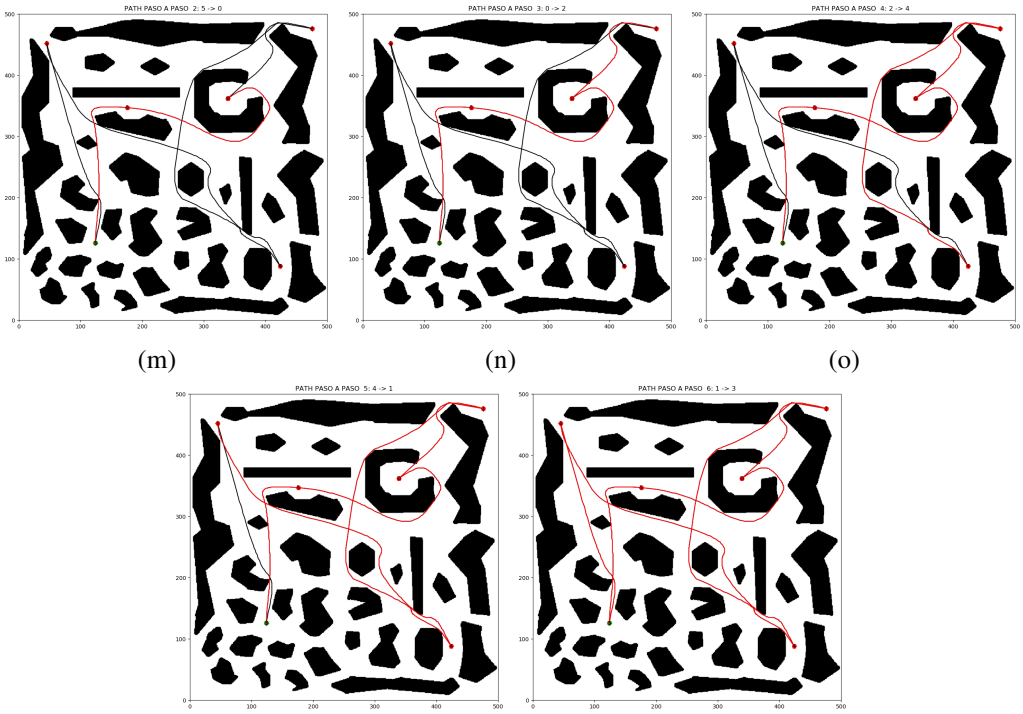


Figura 5.41: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 2.



(m) (n) (o)

(p) (q)

Figura 5.41: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 2.

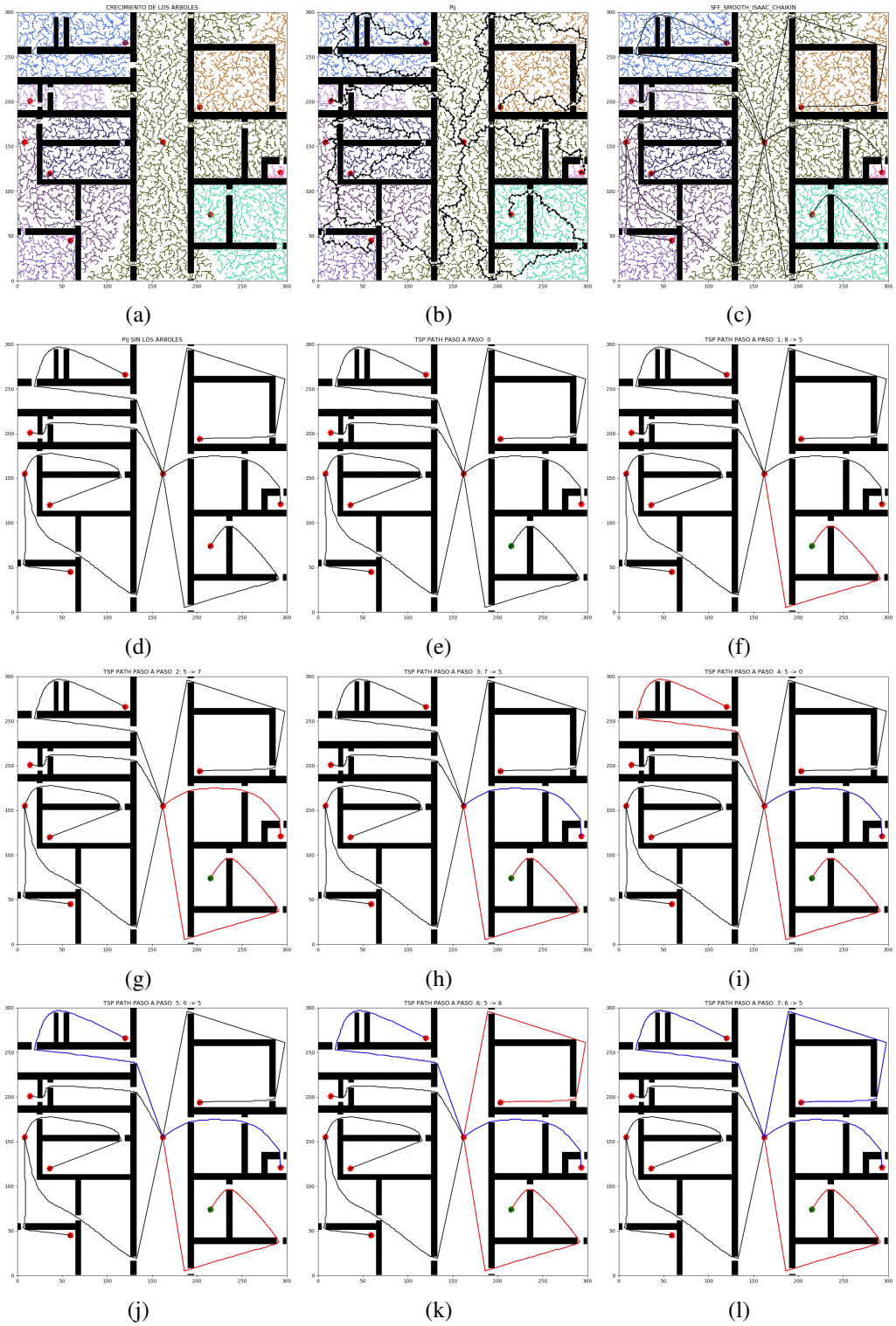


Figura 5.42: Resultado de la ejecución del algoritmo *SFF* en el Entorno 3.

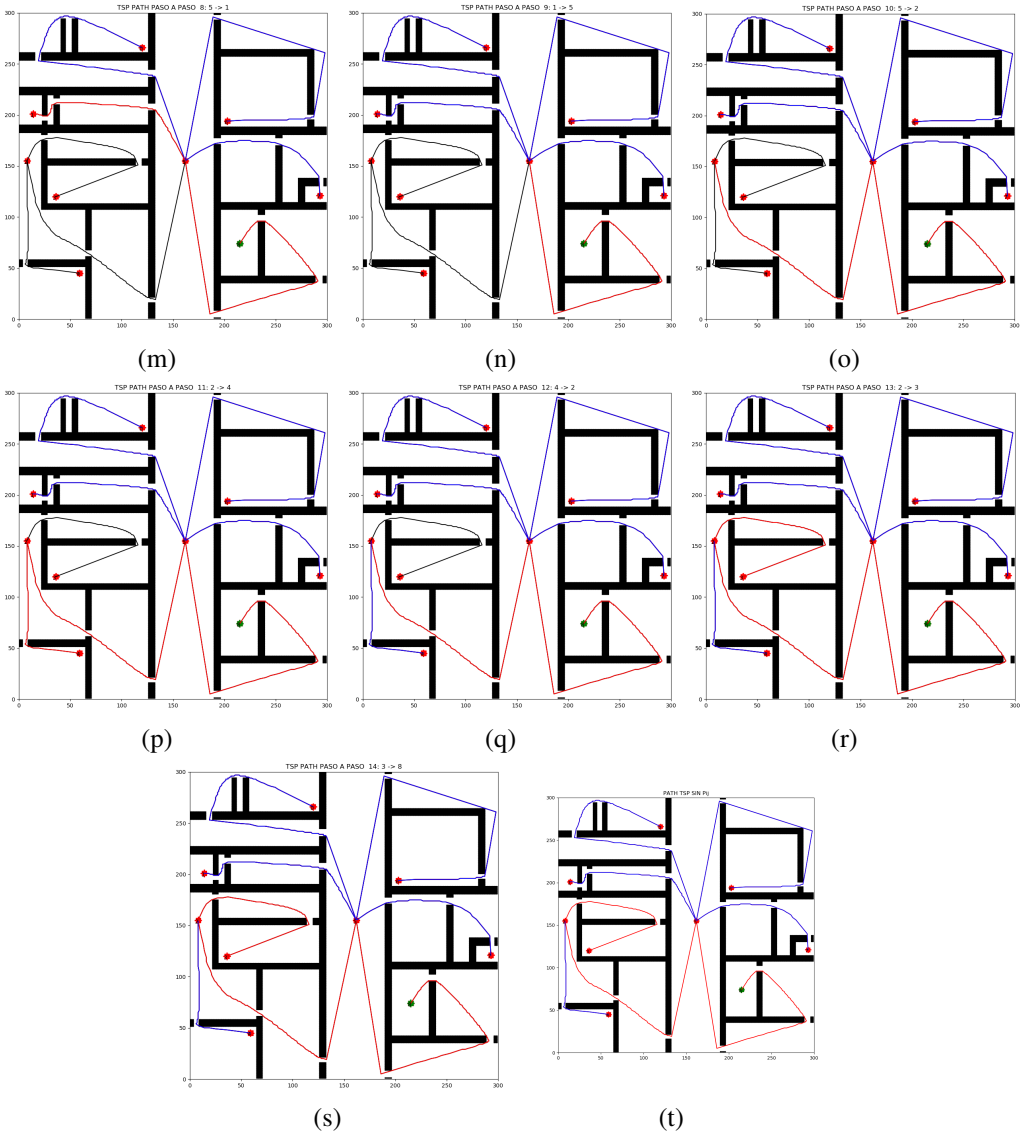


Figura 5.42: Resultado de la ejecución del algoritmo *SFF* en el Entorno 3.

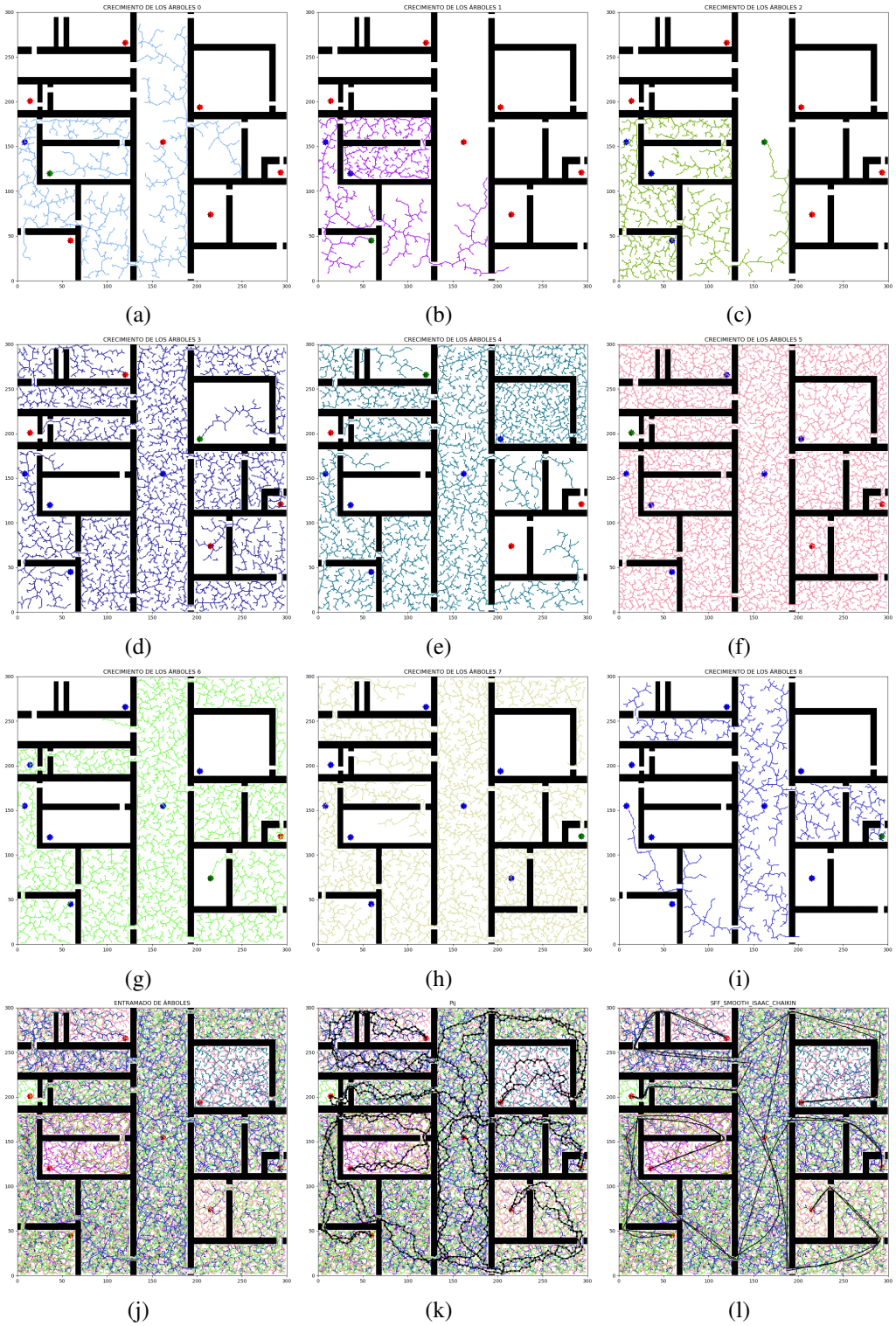


Figura 5.43: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 3.

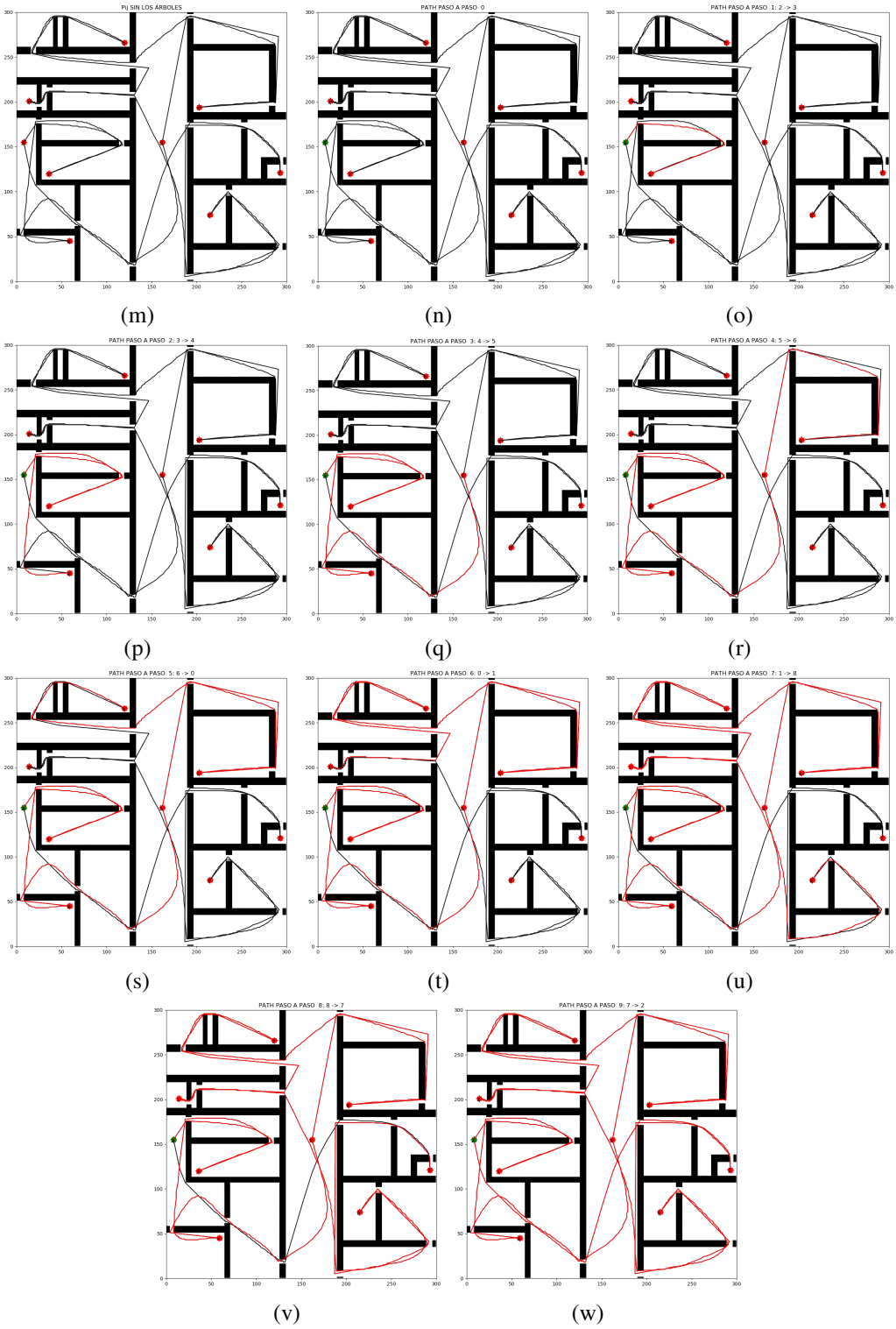


Figura 5.43: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 3.

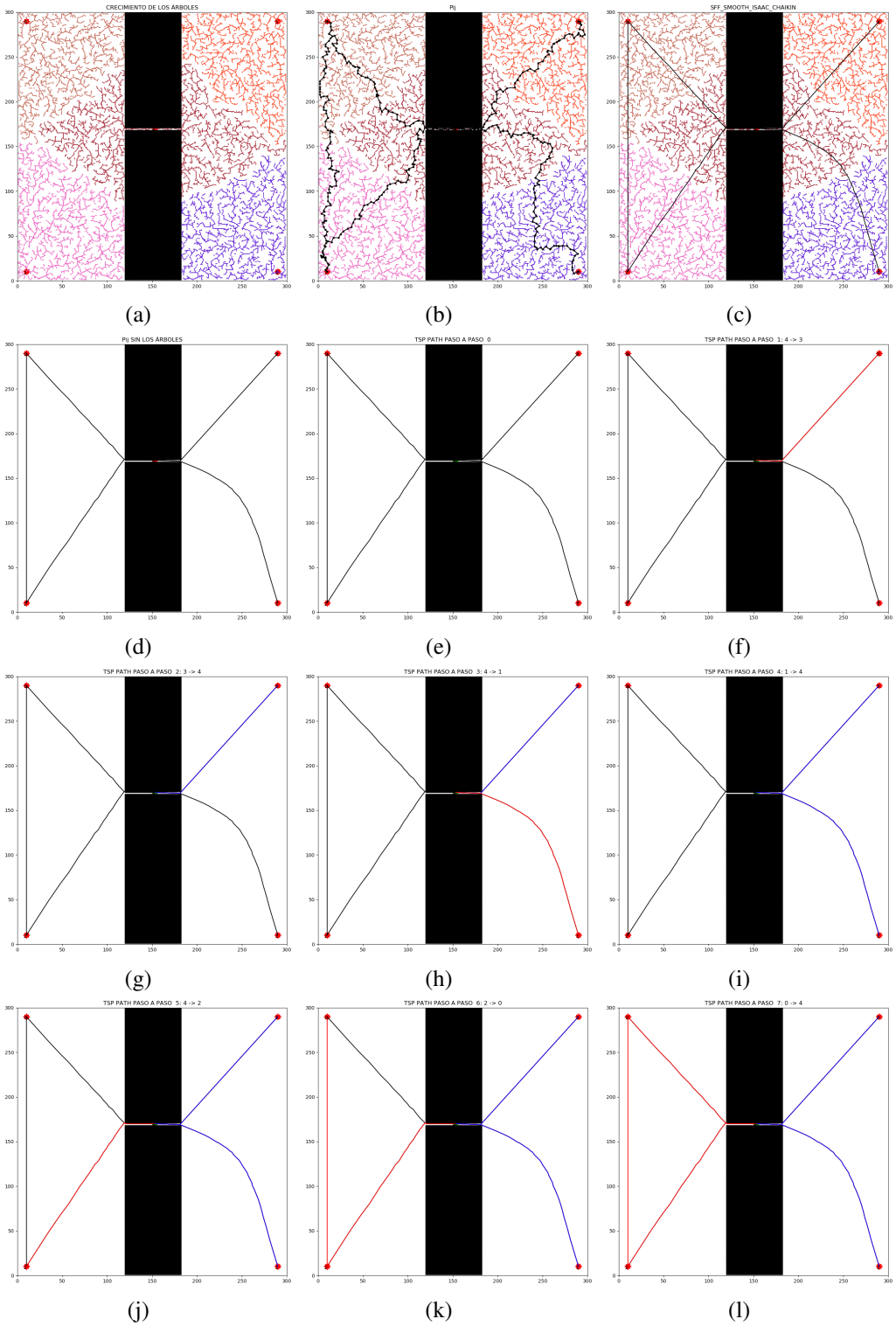


Figura 5.44: Resultado de la ejecución del algoritmo *SFF* en el Entorno 4.

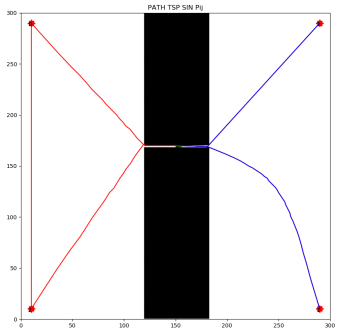


Figura 5.44: Resultado de la ejecución del algoritmo *SFF* en el Entorno 4.

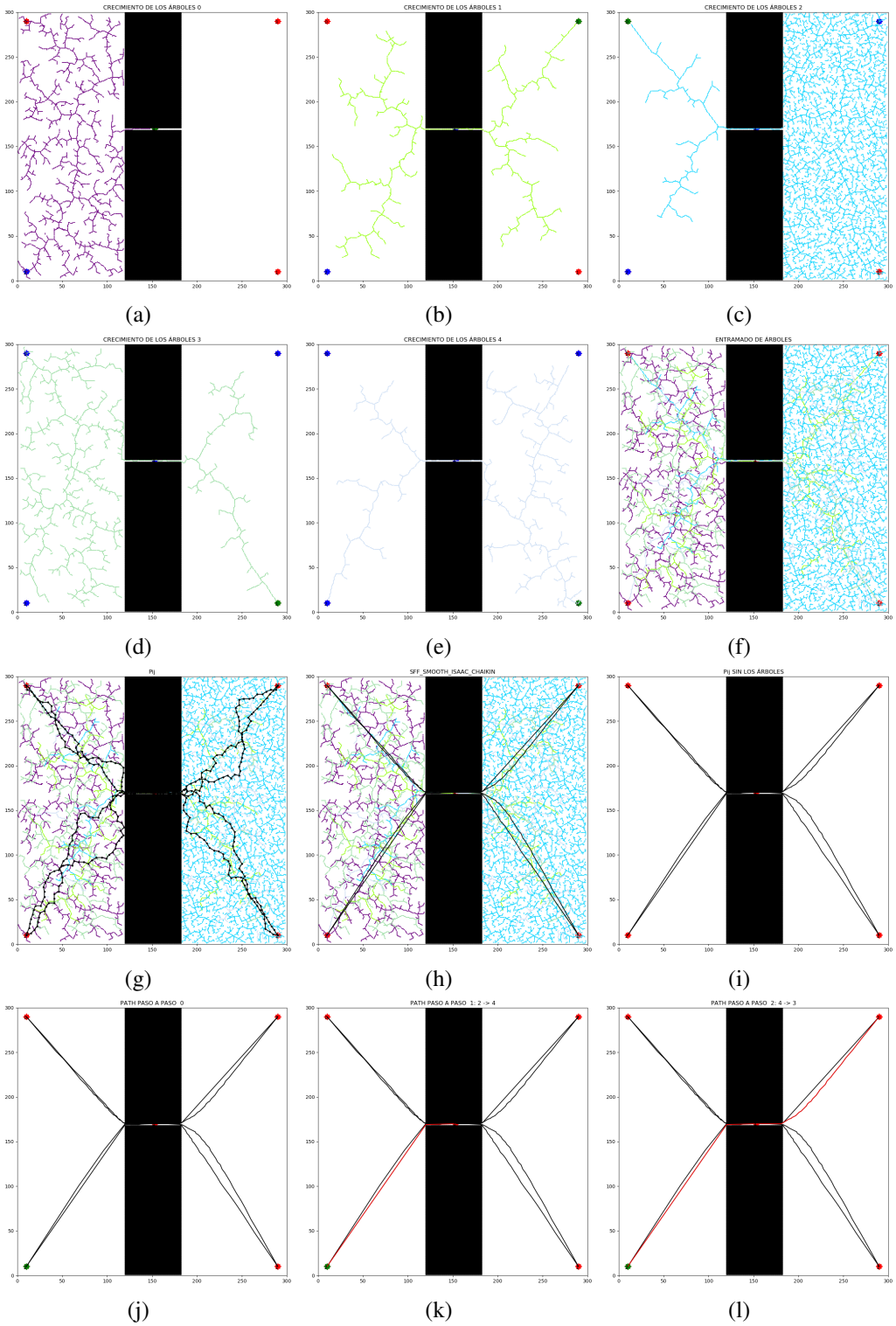
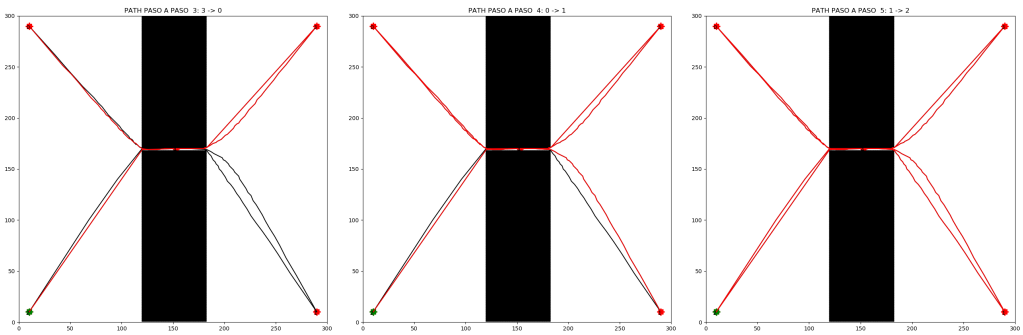


Figura 5.45: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 4.



(m) (n) (o)

Figura 5.45: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 4.

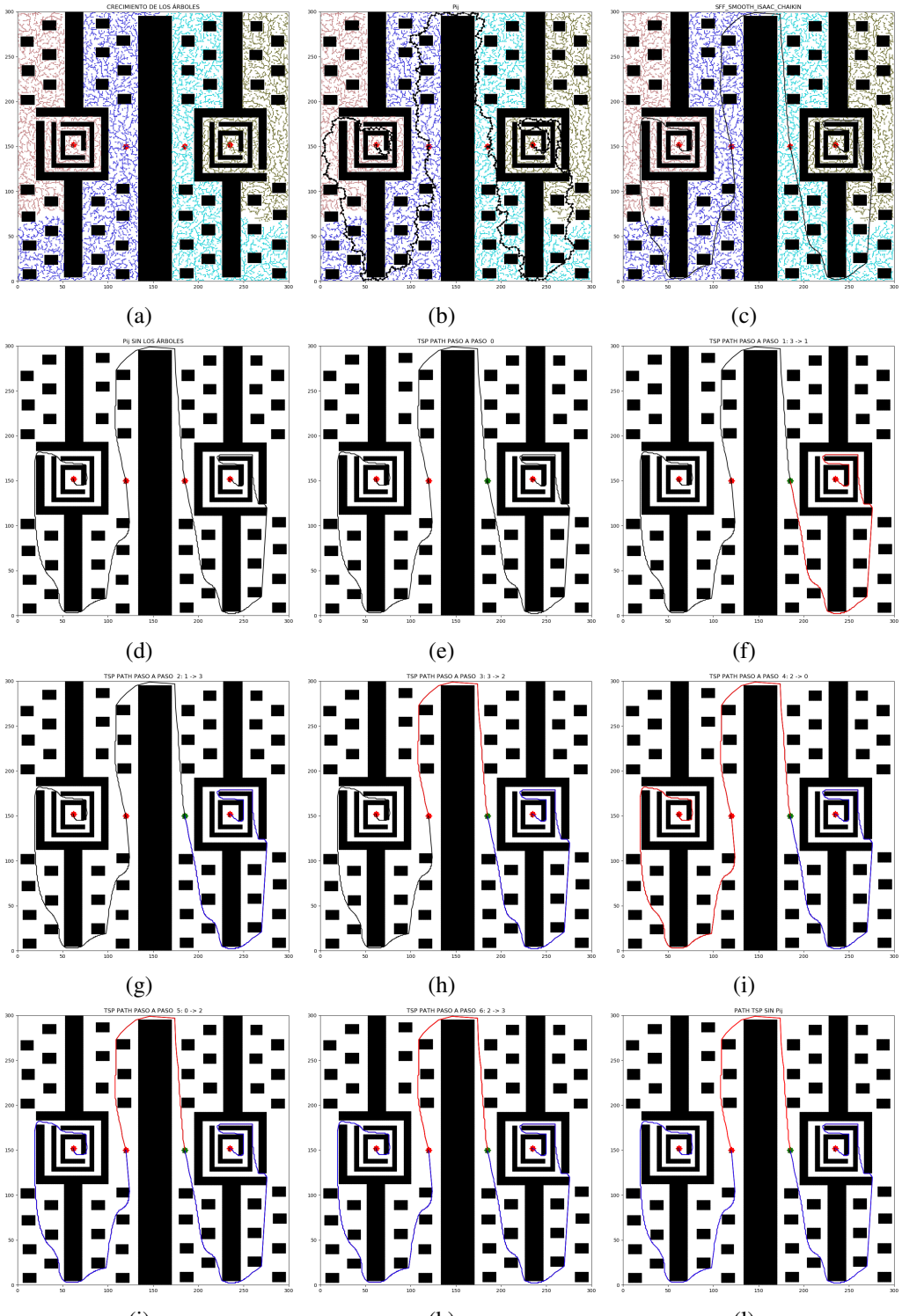


Figura 5.46: Resultado de la ejecución del algoritmo *SFF* en el Entorno 5.

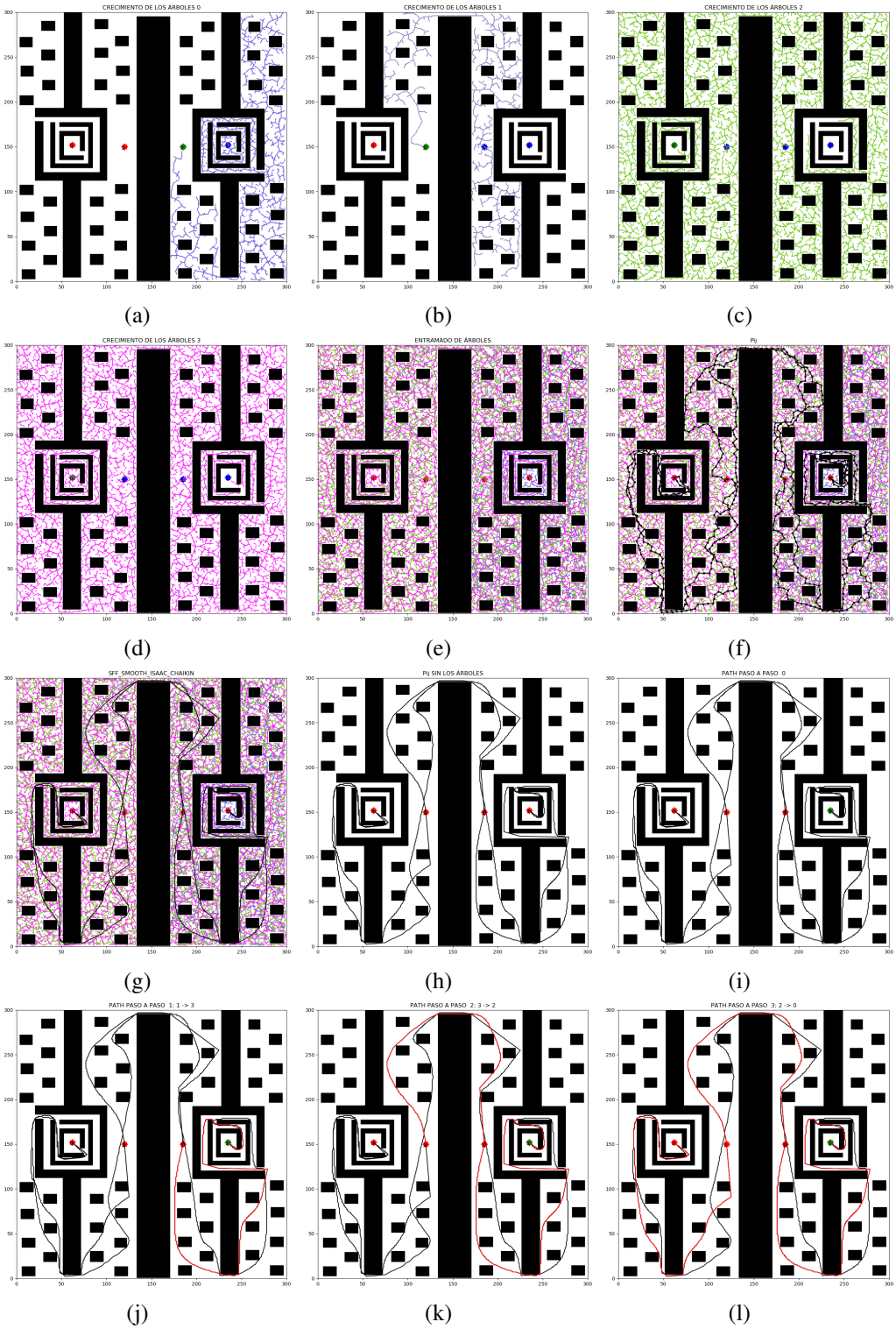


Figura 5.47: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 5.

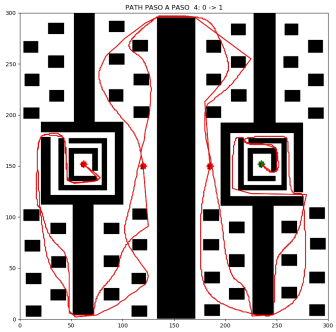


Figura 5.47: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 5.

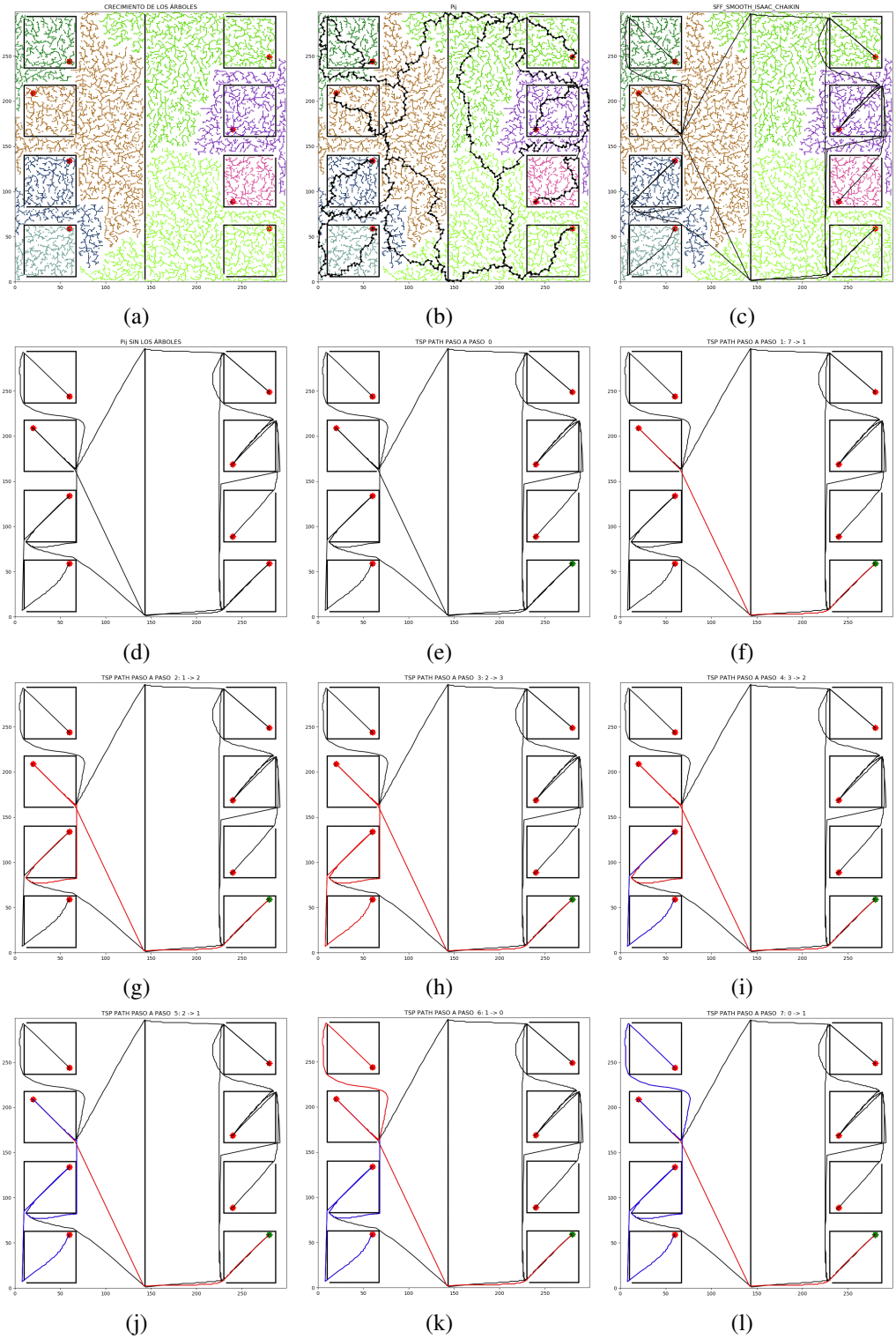
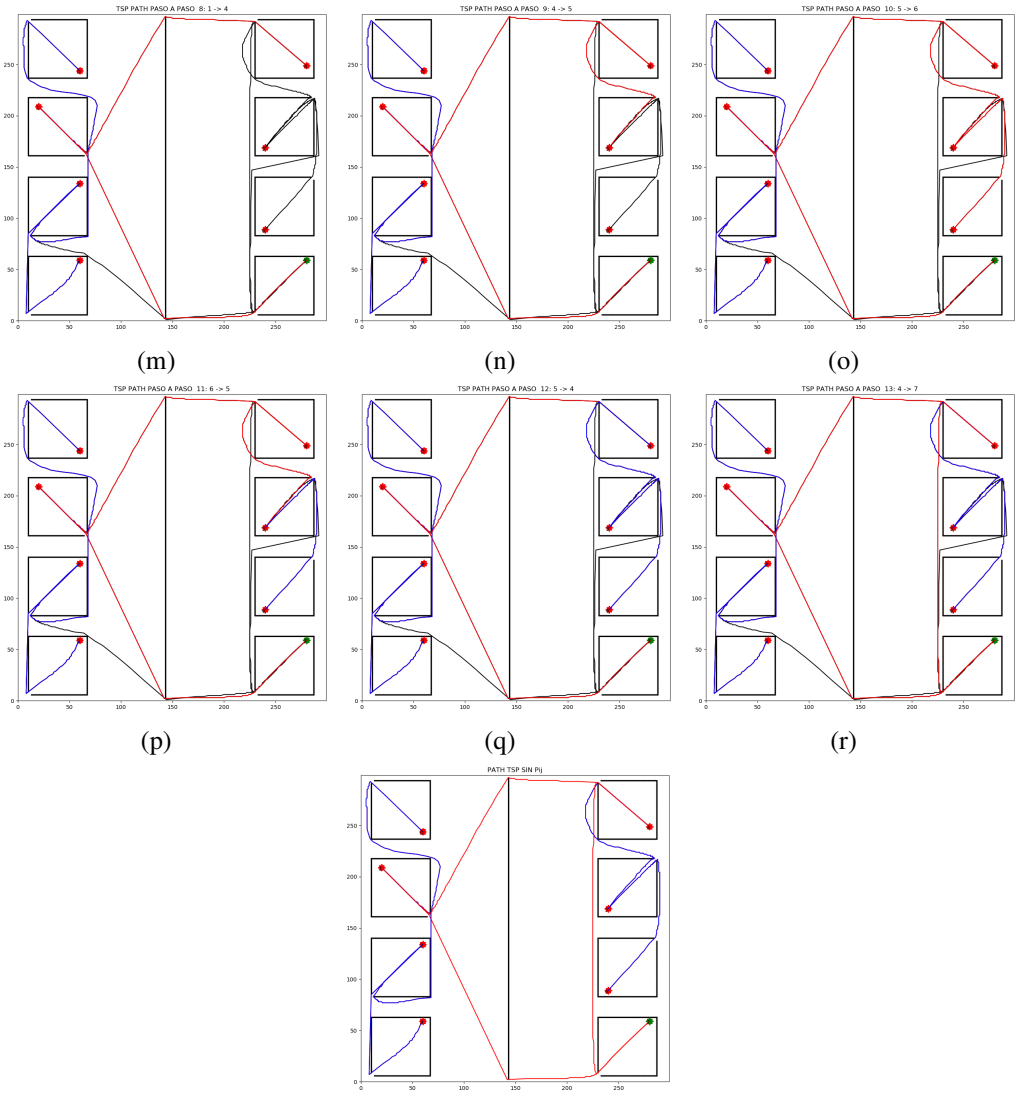


Figura 5.48: Resultado de la ejecución del algoritmo *SFF* en el Entorno 6.



(m) (n) (o)

(p) (q) (r)

(s)

Figura 5.48: Resultado de la ejecución del algoritmo *SFF* en el Entorno 6.

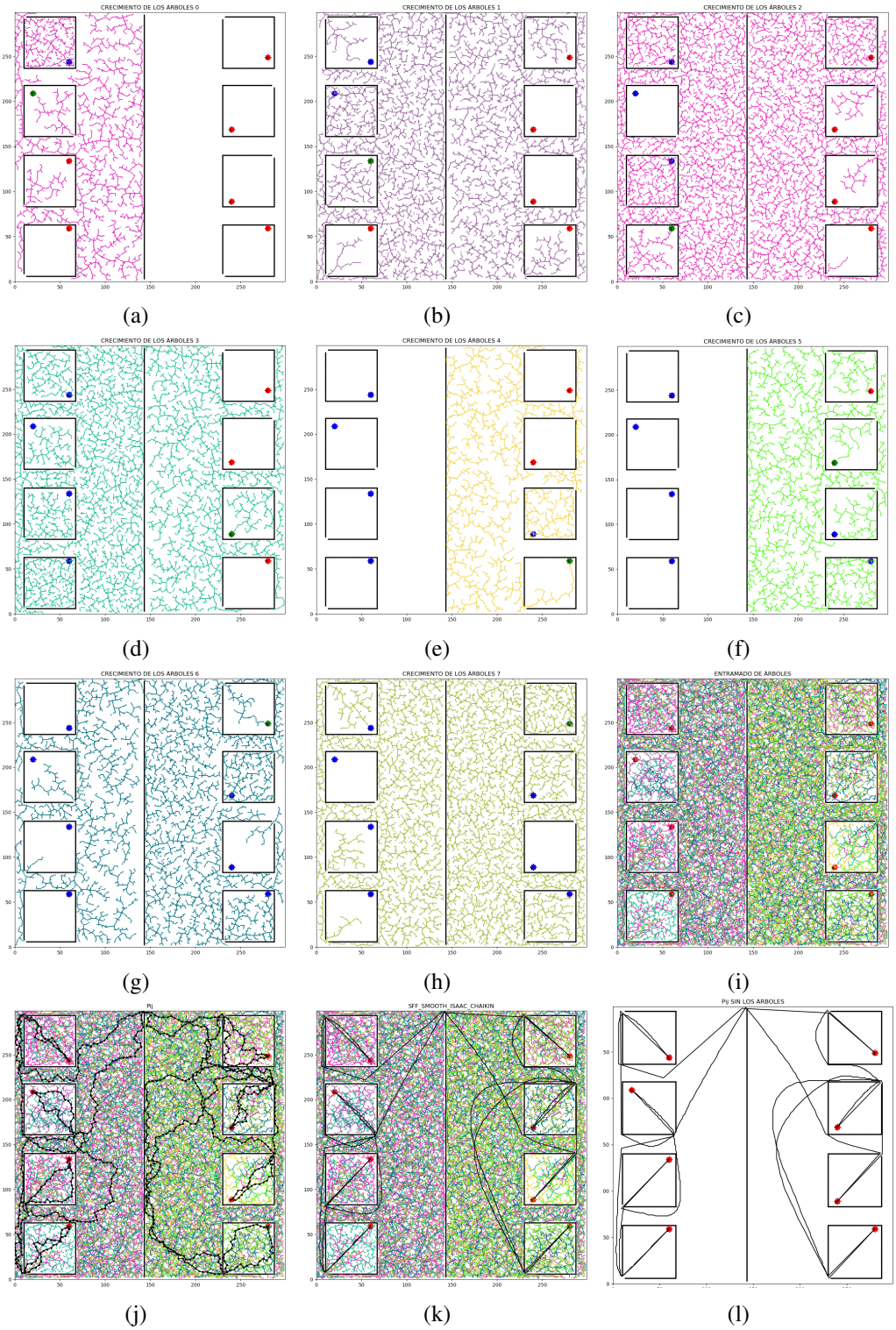
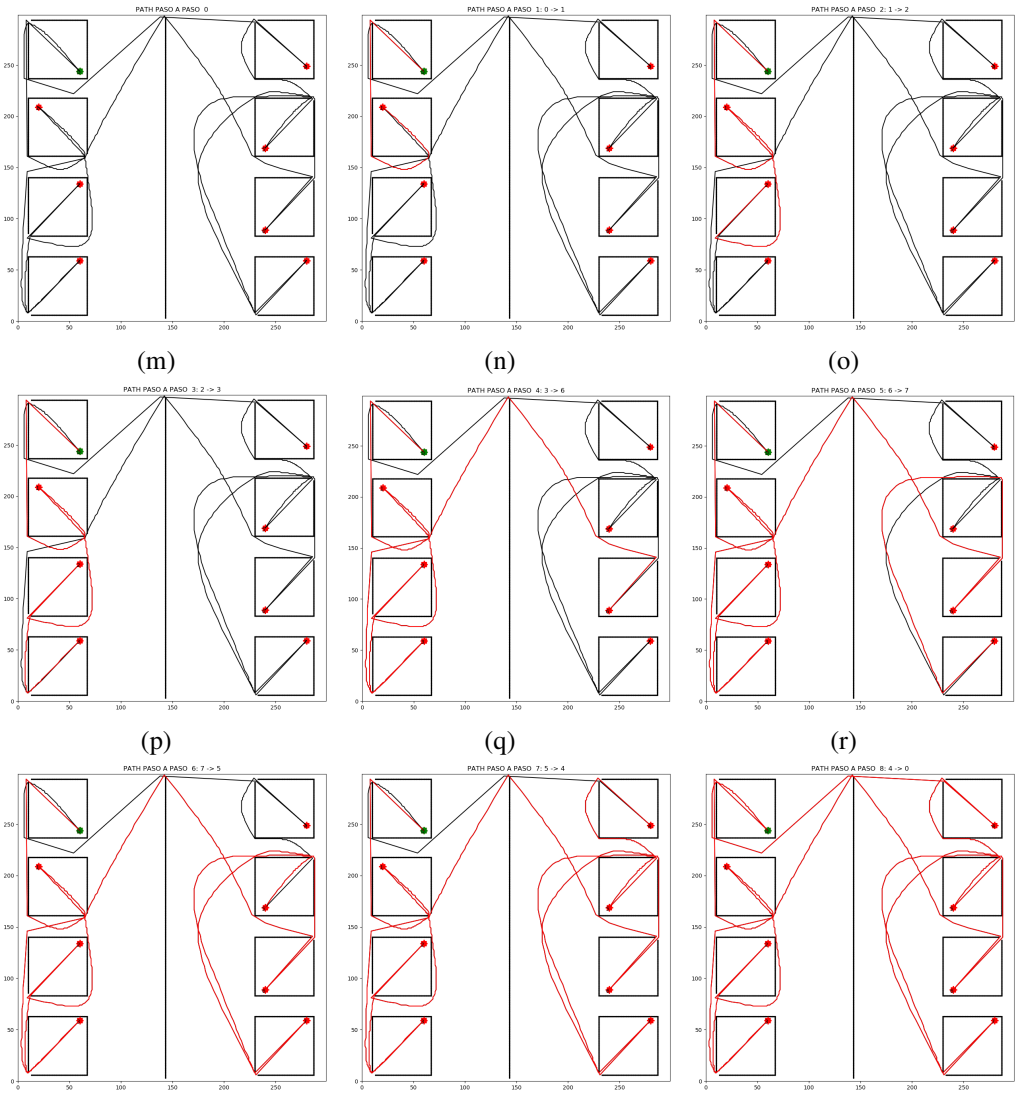


Figura 5.49: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 6.



(m) (n) (o)

(p) (q) (r)

(s) (t) (u)

Figura 5.49: Resultado de la ejecución del algoritmo *RRT-MO* en el Entorno 6.

CAPÍTULO 6

CONCLUSIONES

En este capítulo se analiza el desarrollo del proyecto revisando las fases que lo componen y, finalmente, se realiza una conclusión sobre el trabajo realizado exponiendo futuras ampliaciones que se pueden llevar a cabo.

6.1 Fase previa

Durante la primera fase del proyecto, ha sido necesario comprender conceptos tales como el espacio de configuración, adquirir conocimientos sobre los algoritmos de planificación basados en grafos y muestreo probabilístico, además de entender las principales diferencias que existen entre ellos.

Posteriormente, al decidir implementar el algoritmo basado en muestreo probabilístico *SFF* con la posterior incorporación del algoritmo *TSP*, ha sido necesario estudiar:

- El algoritmo *RRT*, quizás el algoritmo basado en muestreo probabilístico más básico y necesario para entender el resto de la familia de este tipo de algoritmos.
- El algoritmo *SFF*, objeto de estudio de este proyecto.
- El problema del viajante del comercio *TSP*, para así utilizarlo y obtener el camino que se debe seguir por cada uno de los puntos objetivo a los cuales se les aplica *SFF*.

También ha sido necesario familiarizarse con la librería que se ha utilizado, *SBPL*. Ha servido como base para poder implementar los algoritmos *SFF* y *RRT-MO*. Con las herramientas que posee ha sido más sencillo poder implementarlos, ya que buena parte de la base necesaria para poder programar un algoritmo de este tipo viene dada en la librería.

Después de haber analizado y comprendido cada uno de los conceptos mencionados, se ha pasado a la fase de desarrollo de los algoritmos y de integración de *TSP*.

6.2 Fase de desarrollo

La fase del desarrollo es la más importante del proyecto. Se divide en tres fases claramente diferenciadas:

- Implementación del algoritmo *SFF*.
- Integración de *TSP*.
- Implementación del algoritmo *RRT-MO*.

6.2.1 Implementación del algoritmo *SFF*

De la fase de desarrollo, la sección que más tiempo ha llevado implementar ha sido sin duda la del algoritmo *SFF*. Ha sido necesario utilizar conceptos de programación tales como vectores, punteros y diferentes conceptos sobre administración de memoria para poder gestionar el volumen de información preciso para crear y gestionar los árboles necesarios para que el algoritmo *SFF* funcione correctamente.

Se han definido una serie de variables y se han implementado una serie de funciones, comentadas en el apartado 4.3.3, para de esta forma poder crear, expandir y conectar los árboles. También ha sido necesario comprender el uso de la librería *matplotlibcpp* para poder realizar todas las representaciones, junto con diferentes algoritmos tales como *Bresenham*(creación de líneas discretizadas) y *Chaikin* (suavizado de caminos).

Además, durante la implementación del algoritmo, posiblemente el problema más grave que ha surgido ha sido el del tiempo que se tardaba en realizar las búsquedas *NN*. Por ello, fue necesario un estudio y posterior incorporación de diversas estructuras de datos en forma de árbol binario. De esta forma, se pueden realizar este tipo de búsquedas más eficazmente por lo que no se pierde tanto tiempo, consecuencia que ayuda a que el algoritmo sea más ágil computacionalmente hablando.

6.2.2 Incorporación de *TSP*

Posterior a la implementación del algoritmo *SFF*, ha sido necesario comprender cómo funciona el problema del viajante y las diferentes formas de solucionarlo que existen. De esta forma, se ha realizado una búsqueda de una librería ya implementada que fuera sencilla de entender y de incorporar a la librería *SBPL*. Se ha seleccionado una de unos alumnos de la escuela de ingenieros (ISEN-Nantes), la cual dispone de diversos algoritmos implementados para poder resolver el problema.

Adaptarlo a la librería no ha sido muy complicado debido a que solo ha requerido adaptar la librería *grasp.h* para poder utilizar *TSP* en el proyecto.

6.2.3 Implementación del algoritmo *RRT-MO*

Añadido a los apartados posteriores, ha sido necesario implementar el algoritmo *RRT-MO* creando así la librería *RRTplanner.h*. Gracias al trabajo realizado en la fase de implementación del algoritmo *SFF*, se ha podido reutilizar la mayoría de las funciones para así poder implementar una versión de *RRT* multi-objetivo, es decir, una versión de *RRT* que consiga alcanzar los puntos objetivo uno a uno para así posteriormente poder comparar este resultado con el que se obtiene tras la utilización de *SFF+TSP*.

6.2.4 Fase de obtención de resultados

Una vez finalizada la fase de desarrollo, ha sido necesario crear una estructura para poder realizar diversas pruebas sobre los algoritmos implementados. Ha sido muy importante la utilización del archivo *params_RRTSFF.ini* para organizar tanto la selección del modo de uso para poder realizar cada una de las pruebas, como la selección de parámetros para así agilizar el uso de los algoritmos.

La obtención de resultados se ha dividido en cinco etapas:

1. Primera fase en la que se muestra cuál de los árboles binarios incorporados a *SBPL* ofrece mejores resultados tanto si se utilizan únicamente las funciones de cada librería para poder medir el tiempo de creación y de búsquedas del vecino más cercano, como si se incorpora cada librería al algoritmo *SFF* para ver así cómo se comporta el algoritmo cuando se utiliza un árbol u otro.

2. Segunda fase en la que se prueba cada uno de los algoritmos *TSP* incluidos en la librería seleccionada para observar cuál de ellos ofrece mejores resultados.

3. Tercera fase en la que se prueba el algoritmo *SFF* analizando cada uno de sus parámetros para así observar si se ha implementado correctamente y ver como se comporta el algoritmo cuando estos se modifican. Se analiza tanto el tiempo como el coste obtenido en cada una de las pruebas realizadas.

4. Cuarta fase en la que, de forma similar a la tercera, se analiza el algoritmo *RRT-MO*.

5. Quinta y última fase en la que, en función de una serie de entornos supuestamente más beneficiosos para uno u otro algoritmo, se comparan para así comprobar cuál es mejor utilizar en función de las características que tenga el entorno que se utilice.

6.2.5 Problemas encontrados

Los problemas encontrados durante la fase de desarrollo han sido:

1. Problema del tiempo sobre las búsquedas NN realizadas por el algoritmo *SFF*. Comentado en el apartado 4.2.1 y tratado de solucionar con las estructuras de datos en forma de árboles binarios. Gracias a la utilización de este tipo de árboles se ha conseguido reducir considerablemente el tiempo ejecución del algoritmo *SFF*.

2. Crear una nueva forma de representar los casos de prueba en la librería *SBPL*. Debido al escaso mecanismo comentado en el apartado 2.1.4 que posee la librería *SBPL* para poder realizar representaciones, se ha creado una serie de funciones con las cuales se puede utilizar la librería *matplotlibcpp* para así poder realizar diversas representaciones de las ejecuciones de los algoritmos *SFF*, *RRT-MO* y de la aplicación de *TSP*.

3. Problemas de gestión de la memoria (*Memory Lea*k). Durante las primeras pruebas realizadas respecto a la repetición continuada de la ejecución de los algoritmo *SFF* y *RRT-MO*, resultó sumamente complicado liberar cada una de las posiciones de memoria reservada por lo punteros declarados a lo largo del algoritmo. Por ello, tal y como se comenta en el apartado 4.1.7 se ha hecho uso de la librería *memory* de C++ para poder gestionar la memoria de una forma mucho más sencilla.

4. Tras aplicar el algoritmo *SFF* es posible obtener puntos objetivo los cuales tienen una o ninguna conexión con alguno de sus vecinos. Debido a la posición que pueden tomar los puntos objetivo en un entorno repleto de obstáculos, es bastante probable que alguno de los puntos objetivo únicamente tenga una conexión con alguno de sus vecinos, por lo que imposibilita que se pueda aplicar el algoritmo *TSP* y éste encuentre una solución válida a la primera ya que a la fuerza se tiene que pasar por alguno de los nodos al menos dos veces. Por ello, en el apartado 4.4.2 se aborda la solución implementada respecto a este problema, la cual pasa por utilizar el algoritmo de *Dijkstra* para poder establecer más conexiones a los nodos que tengan menos para así tener más posibilidades de encontrar un camino válido.

5. Búsquedas de librerías de árboles binarios y *TSP*. Uno de los puntos fuertes también ha sido el de llevar a cabo búsquedas de librerías que se adaptasen correctamente a nuestras necesidades a la hora de utilizar árboles binarios y *TSP*. Es posible que no sean las más potentes ni las más sofisticadas, pero se considera que desde un punto de vista académico son lo suficientemente correctas para realizar la implementación que se ha llevado a cabo.

6. *RRT-MO* no tiene implementado un sistema de detención en caso que no se pueda alcanzar algún punto objetivo. Uno de los temas sobre el cual no se ha hecho énfasis es sobre la detección de parada del sistema *RRT-MO*. Si por algún motivo *RRT-MO* se ejecuta sobre un entorno en el cual no se puede obtener una solución, esté se ejecutará eternamente y habrá que matar el proceso manualmente. Esto se debe a que, a diferencia de *SFF* lo cual lo consigue gracias al parámetro *k*, *RRT-MO* no posee un mecanismo para detectar cuándo dejar de expandir los árboles.

6.2.6 Resultados obtenidos

Respecto a los resultados obtenidos, gracias a los casos de prueba realizados sobre cada uno de los algoritmos, se ha podido comprobar que su implementación es correcta y que cumplen su función dependiendo del valor de cada uno de sus parámetros.

Por otro lado, respecto a la comparación entre los algoritmos *SFF* y *RRT-MO*, se ha podido comprobar que el algoritmo *SFF* funciona realmente bien para entornos complejos en los que existen puntos objetivo que se encuentran en zonas de difícil acceso debido a la existencia de pasillos estrechos. El tiempo de ejecución en estos casos es mejor que si se utiliza el algoritmo *RRT-MO* debido a que le es realmente complicado encontrar dichos pasillos rápidamente.

De todas formas, aunque el algoritmo *RRT* no se concibió para tareas multi-objetivo como las de este proyecto, se ha visto que en entornos sencillos y con espacios relativamente anchos tiene un buen comportamiento respecto al tiempo de ejecución. Por ello, dependiendo del entorno, como por ejemplo el 1 y el 2 (ver figura 5.36), el algoritmo *RRT-MO* es perfectamente utilizable hasta el punto en el que se obtienen mejores resultados (respecto al tiempo de ejecución) que con el algoritmo *SFF*.

Eso sí, para este tipo de entornos, la aplicación de *TSP* hace que se obtengan caminos mucho mejores que simplemente utilizando *RRT-MO* y conectando los caminos obtenidos. Para el resto de casos (entornos 3, 4, 5 y 6), no necesariamente se obtienen mejores caminos con la aplicación de *TSP* ya que en la mayoría de ellos existen puntos objetivo situados en zonas de difícil acceso lo cual lleva a que los árboles solo se conecten con el

árbol vecino más próximo. Por ello, solo tendrán una conexión y habrá que realizar largos recorridos para poder visitarlos todos.

Por lo tanto, se puede decir que, tal y como se comenta en la publicación del algoritmo *SFF*, para casos en los que se tengan muchos obstáculos y puntos objetivo en zonas de difícil acceso y que para llegar hasta ellos se tenga que pasar por zonas estrechas, el uso de *SFF* es muy recomendable ya que el tiempo de ejecución es muy inferior debido a que se hace crecer un árbol desde cada nodo y eso hace que se puedan explorar los espacios estrechos muy fácilmente.

En cambio, si tenemos entornos más normales en los que existan zonas relativamente anchas y con suficiente espacio libre para recorrer, el algoritmo *RRT-MO* es muy rápido y ofrece mejores resultados que *SFF* respecto al tiempo de ejecución. De todas formas, si es más importante obtener mejores caminos y se decide sacrificar tiempo de ejecución, se recomienda utilizar *SFF+TSP* ya que se obtienen caminos más cortos.

6.3 Futuras ampliaciones

Es evidente que el mayor problema encontrado durante la realización del proyecto ha sido el tiempo que se debe invertir en las búsquedas del vecino más cercano y lo importante que es tener bien implementada la estructura de datos que se quiera utilizar para gestionar los árboles a la hora de realizar las búsquedas, en nuestro caso los árboles binarios.

Sería buena opción implementar o conseguir alguna librería realmente potente y que tuviera un gran rendimiento a la hora de realizar búsquedas sobre grandes árboles.

Otra forma de reducir considerablemente el tiempo de ejecución global del algoritmo *SFF* es sacar provecho de SU filosofía, la cual es generar diversos árboles desde cada uno de los puntos objetivo, por ello sería muy interesante realizar una implementación *multi-thread*. Así, cada *thread* (por ejemplo, uno por cada procesador que tenga el equipo que se utilice) podría encargarse del crecimiento de un cierto número de árboles, haciendo así que el tiempo de computación se redujera considerablemente.

Otra buena ampliación podría ser el hecho de realizar búsquedas k-NN en vez de *NN*, es decir, realizar búsquedas del vecino más cercano que no sean exactas, barajando así una cantidad determinada de resultados que para su obtención requieren de un menor tiempo de ejecución que en el caso de que se quiera encontrar de forma exacta el vecino más cercano.

APÉNDICE A

MANUAL DE INSTALACIÓN.

A continuación se muestran los pasos que se deben llevar a cabo para poder instalar todos los componentes necesarios para hacer uso de la librería *SBPL* y por lo tanto, de la implementación que se ha llevado a cabo.

Se describen cada una de las utilidades que se han tenido que instalar así como la configuración necesaria para dejar la librería correctamente preparada. Además, se muestra como se lleva a cabo la ejecución del programa principal con cada uno de los algoritmos.

1. En nuestro caso, la máquina utilizada tiene las siguientes características: INTEL® Core™ *i7-4710MQ CPU@2,50GHz x 8/ 8GB RAM* utilizando únicamente un núcleo.

2. Se ha utilizado el sistema operativo *Ubuntu 20.04.5 LTS* de 64 bits.

3. Primeramente, abrimos una consola (*Ctrl* + *Alt* + *T*) y ejecutamos:

   sudo apt-get update

   sudo apt -y upgrade

4. Instalamos *build-essential* y *cmake*:

   sudo apt-get install build-essential libssl-dev

   sudo apt install cmake

5. Instalamos *python3* juntamente con el paquete *numpy*:

   sudo apt-get install python3 python3-dev

   sudo apt-get install python3-numpy

6. Instalamos *pip*:

   sudo apt-get install python3-pip

7. Instalamos la librería *matplotlib*:

   pip install matplotlib

8. Instalamos la librería *boost*, necesaria para la utilización de *GRASP*:

   sudo apt-get install libboost-all-dev

9. Nos situamos sobre la librería *sbpl*, en este caso sobre el usuario *isaac*:

   cd /home/isaac/sbpl

10. Creamos la carpeta *build* y la compilamos:

    mkdir build

    cd build

    cmake ..

    make

11. Finalmente, se puede hacer uso de los algoritmos *SFF* y *RRT*. Hay que tener en cuenta el estado del archivo *params_RRTSFF.ini* para saber que modo de uso se está utilizando. Si por ejemplo ejecutamos cada uno de los algoritmos sobre el entorno *env_isaac_3.cfg*:

    Caso *SFF*: *./test_sbpl –env=2d –planner=sff ../env_examples/nav2d/env_isaac_3.cfg*

    Caso *RRT*: *./test_sbpl –env=2d –planner=rrt ../env_examples/nav2d/env_isaac_3.cfg*

APÉNDICE B

MANUAL DE USO DEL ARCHIVO *params_RRTSFF.ini*.

El archivo de configuración *params_RRTSFF.ini* se ha creado con la intención de poder gestionar de forma sencilla y rápida cada uno de los parámetros de los algoritmos *SFF* y *RRT*. Se puede elegir el modo de uso que se vaya a utilizar para así poder ejecutar los algoritmos en función de si se desea llevar a cabo una ejecución simple o de si se quiere realizar alguna de las pruebas específicas para observar como se comporta cada algoritmo.

El archivo está dividido en diferentes secciones, las cuales se explican a continuación:

- ***MODE***. Sección que encabeza el archivo, con ella se selecciona el modo que se va a utilizar. En función del algoritmo que se utilice desde el programa principal, se hará uso de una sección u otra. Cambiando el número asignado a ***SFF_MODE*** o a ***RRT_MODE*** se selecciona el modo de uso a utilizar y por lo tanto, a la sección que hay que referirse para fijarnos en las variables que deben ser modificadas.

  Los distintos modos que se han creado son:

  - ***SFF_MODE***
    - *0 = SFF_NORMAL_MODE*. Modo de uso normal del algoritmo *SFF*.
    - *1 = SFF_TEST_BINARY_TREES_INDIVIDUALLY*. Modo en el que se prueban los árboles binarios de forma individual.
    - *2 = SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS*. Modo en el que se prueba el algoritmo *SFF* cuando se le modifica el número de puntos objetivo.
    - *3 = SFF_TEST_ALGORITHM_PERFORMANCE_K*. Modo en el que se prueba el algoritmo *SFF* cuando se le modifica el parámetro k.
    - *4 = SFF_TEST_ALGORITHM_PERFORMANCE_DTREE*. Modo en el que se prueba el algoritmo *SFF* cuando se le modifica el parámetro d_{tree}.

        * *5 = SFF_TEST_ALGORITHM_PERFORMANCE_R*. Modo en el que se prueba el algoritmo *SFF* cuando se le modifica el parámetro *R*.

    - ***RRT_MODE***

        * *0 = RRT_NORMAL_MODE*. Modo de uso normal del algoritmo *RRT*.
        * *1 = RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS*. Modo en el que se prueba el algoritmo RRT cuando se le modifica el número de puntos objetivo.
        * *2 = RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE*. Modo en el que se prueba el algoritmo RRT cuando se le modifica el parámetro *step_size*.
        * *3 = RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB*. Modo en el que se prueba el algoritmo RRT cuando se le modifica el parámetro *qgoal_probability*.

- *SFF_NORMAL_MODE*. Sección que gestiona el modo de uso normal del algoritmo *SFF*.

- *SFF_TEST_BINARY_TREES_INDIVIDUALLY*. Sección que gestiona el modo de uso en el que se prueban los árboles binarios de forma individual.

- *SFF_TEST_ALGORITHM_PERFORMANCE*. Sección que gestiona cada uno de los modos de prueba que se pueden seleccionar para el algoritmo *SFF*.

- *RRT_NORMAL_MODE*. Sección análoga a *SFF_NORMAL_MODE*.

- *RRT_TEST_ALGORITHM_PERFORMANCE*. Sección análoga a *SFF_TEST_ALGORITHM_PERFORMANCE*.

- *DRAW*. Sección encarga de habilitar y deshabilitar la representación de cada uno de los pasos de los cuales se compone una ejecución. Además, se pueden modificar diversos parámetros para controlar la forma en la que se realizan dichas representaciones.

En la figura B.1 se puede observar la estructura del archivo con cada una de las secciones que se acaban de comentar.

```
;-------------------------------------------------------------------
[MODE]

SFF_MODE = 0 ; 0 = SFF_NORMAL_MODE
             ; 1 = SFF_TEST_BINARY_TREES_INDIVIDUALLY,
             ; 2 = SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS,
             ; 3 = SFF_TEST_ALGORITHM_PERFORMANCE_K,
             ; 4 = SFF_TEST_ALGORITHM_PERFORMANCE_DTREE,
             ; 5 = SFF_TEST_ALGORITHM_PERFORMANCE_R,

RRT_MODE = 0 ; 0 = RRT_NORMAL_MODE,
             ; 1 = RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS,
             ; 2 = RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE,
             ; 3 = RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB,
;-------------------------------------------------------------------

;-------------------------------------------------------------------
[SFF_NORMAL_MODE]

;Coords_X = 50 480 110 420 250 320 420 55
;Coords_Y = 50 480 410 50 150 450 240 220

Coords_X = 6 173 401 403 131 24 486
Coords_Y = 216 84 75 365 387 479 17

enable_random_targets = off ; Tener en cuenta tamaño size_X y size_Y
                            ; del archivo .cfg cuando se utilice esta
                            ; caracteristica.
number_of_targets = 10
min_dist_between_targets = 130
k = 4
d_tree = 10
R = 17

seed = 0 ; 0 = random seed

binary_tree = 1 ; 0 = Tree.hh(no binary tree)
                ; 1 = KDTREE
                ; 2 = COVER_TREE
                ; 3 = QUAD_TREE
                ; 4 = PH_TREE
;-------------------------------------------------------------------

;-------------------------------------------------------------------
[SFF_TEST_BINARY_TREES_INDIVIDUALLY]

min_nodes = 10
max_nodes = 10000
step = 10
nn_searches = 100 ; Búsquedas para cara árbol creado. Para cada step
                  ; se realizan nn_searches.

seed = 0 ; 0 = random seed
;-------------------------------------------------------------------

;-------------------------------------------------------------------
[SFF_TEST_ALGORITHM_PERFORMANCE]

number_of_targets = 25 ; Number_of_targets_default
min_dist_between_targets = 80

k = 5 ; k_default
d_tree = 10 ; DTree_default
R = 15 ; R_default

seed = 0 ; 0 = random seed

;Targets_test - SFF_TEST_ALGORITHM_PERFORMANCE_TARGETS
min_targets = 2
max_targets = 50
step_targets = 1

;k_test - SFF_TEST_ALGORITHM_PERFORMANCE_K
min_k = 2
max_k = 15
step_k = 1

;DTree_test - SFF_TEST_ALGORITHM_PERFORMANCE_DTREE
min_d_tree = 5
max_d_tree = 30
step_d_tree = 1

;R_test - SFF_TEST_ALGORITHM_PERFORMANCE_R
min_R = 5
max_R = 20
step_R = 1

number_of_executions = 10 ; Ejecuciones del algoritmo SFF.

binary_tree = 4  ; 0 = Tree.hh(no binary tree)
                 ; 1 = KDTREE
                 ; 2 = COVER_TREE
                 ; 3 = QUAD_TREE
                 ; 4 = PH_TREE
;-------------------------------------------------------------------
```

```
;-------------------------------------------------------------------
[RRT_NORMAL_MODE]

;Coords_X = 63 14 63 63 283 233 234 282
;Coords_Y = 58 85 163 240 58 133 211 241

enable_random_targets = off ; Tener en cuenta tamaño size_X y size_Y
                            ; del archivo .cfg cuando se utilice esta
                            ; caracteristica.
number_of_targets = 25
min_dist_between_targets = 30

seed = 0 ; 0 = random seed

mode = 1 ; 0 = random, 1 = proximity
step_size = 5
qgoal_probability = 4
;-------------------------------------------------------------------

;-------------------------------------------------------------------
[RRT_TEST_ALGORITHM_PERFORMANCE]

number_of_targets = 25 ; Number_of_targets_default
min_dist_between_targets = 80

mode = 1; 0 = random, 1 = proximity
step_size = 15
qgoal_probability = 4

seed = 0 ; 0 = random seed

;Targets_test - RRT_TEST_ALGORITHM_PERFORMANCE_TARGETS
min_targets = 2
max_targets = 50
step_targets = 1

;StepSize_test - RRT_TEST_ALGORITHM_PERFORMANCE_STEPSIZE
min_step_size = 5
max_step_size = 20
step_step_size = 1

;qgoalProb_test - RRT_TEST_ALGORITHM_PERFORMANCE_QGOALPROB
min_qgoal_probability = 2
max_qgoal_probability = 99
step_qgoal_probability = 1

number_of_executions = 10 ; Ejecuciones del algoritmo SFF.
;-------------------------------------------------------------------
```

```
;-------------------------------------------------------------------
[DRAW]

enabled = on ; on = enabled, off = Disabled
targets_pixels_inflated = 3
draw_tree_growth_enabled = off ; Si está deshabilidato, solo se
                               ; dibuja el árbol completo.
perc_save_figs = 3 ; Porcentaje que se desea guardar del total de figuras.
draw_TSP_steps_enabled = on ; Dibujar o no cada uno de los pasos de TSP.

;smooth
smooth_type_ = 4 ;0 = Without smooth
                 ;1 = Chaikin smooth
                 ;2 = Smooth SFF paper
                 ;3 = Smooth SFF paper modified Isaac
                 ;4 = Smooth 3 + 1 (Chaikin+Isaac)
;chaikin
chaikin_iterations = 5

;smooth SFF / smooth SFF Isaac
window_size = 1
;-------------------------------------------------------------------
```

Figura B.1: Figura del archivo *params_RRTSFF.ini*. Se puede observar cada una de las secciones que lo componen.

La lista de variables que se pueden modificar se muestra a continuación, siendo la mayor parte de ellas comunes entre cada una de las secciones:

- *Coords_X*, *Coords_Y*: sirven para seleccionar de forma manual las coordenadas de cada uno de los puntos objetivo. Hay que tener en cuenta las dimensiones del entorno y la ubicación de los obstáculos para colocar correctamente cada uno de los puntos objetivo.

- *enable_random_targets*, *number_of_targets*, *min_dist_between_targets*: Si esta opción está activada, se ignoran las coordenadas *Coords_X*, *Coords_Y* y se generan una serie de puntos objetivo en función del parámetro *number_of_targets*, separados una distancia mínima entre ellos correspondiente a la que se asigna al parámetro *min_dist_between_targets*.

- *k*, *d_tree*, *R*: se asigna un valor a cada uno de los parámetros del algoritmo *SFF*.

- *seed*: se escoge la semilla que se utiliza para la ejecución que se va a realizar. Si se le asigna el valor 0, se escoge una semilla al azar.

- *binary_tree*: con este parámetro, en función del valor que se le asigne, selecciona que tipo de árbol binario se utilizará durante la ejecución del algoritmo.

- *min, max, step*: todos los parámetros que empiecen con alguna de estas palabras sirven para delimitar los límites inferior y superior además del valor con el cual se avanza en cada iteración. De esta forma se puede probar el comportamiento que tiene cada algoritmo en función de la variabilidad de alguno de sus parámetros.

- *nn_searches*. Para el modo *SFF_TEST_BINARY_TREES_INDIVIDUALLY*, delimita el número de búsquedas NN que se realizan cada vez que se crea un árbol binario.

- *number_of_executions*. En los modos de test de los algoritmos, se establece el número de ejecuciones que se realizan en cada iteración del bucle principal *"min:step:max"*.

- *mode*. Establece el modo en el cual se ejecuta el algoritmo *RRT*, es decir, de forma aleatoria o por proximidad.

- *step_size*, *qgoal_probability*: se asigna un valor a cada uno de los parámetros del algoritmo *RRT*.

- *draw enabled*. Con este *flag* se activa y desactiva la representación.

- *targets_pixels_inflated*: con este parámetro se modifica el tamaño que tendrán los puntos objetivo a la hora de representarlos. De esta forma, para entornos grandes, se puede observar bien en que posición se encuentran.

- *draw_tree_growth_enabled*: con este *flag* se activa y se desactiva la opción de visualizar el crecimiento de los árboles. Si se encuentra desactivada, se dibuja el estado final de éstos.

- *perc_save_figs*: con este porcentaje se puede modificar la cantidad de figuras que se almacenan del crecimiento de los árboles.

- *draw_TSP_steps_enabled*: *flag* que se utiliza para dibujar o no el recorrido que ha calculado TSP o el algoritmo *RRT-MO* paso a paso. Al igual que en el caso de los árboles, si se desactiva se visualiza de forma directa el recorrido final.

- *smooth_type*: se escoje el tipo de suavizado que se utiliza.

- *chaikin_iterations*: número de iteraciones que realiza el algoritmo *Chaikin*.

- *window_size*: ventana de actuación con la que trabajan los algoritmos de suavizado *SFF* y el modificado para realizar el estirado de los caminos.

Finalmente, cabe mencionar la estructura creada para guardar la información que se genera una vez se utiliza algún modo de uso. El resultado de la ejecución o conjunto de ejecuciones se guarda en un archivo con extensión *.csv*.

Por otro lado, en función de si se tiene habilitado el dibujo, se generan las imágenes correspondientes a la ejecución simple que se haya realizado. Para las ejecuciones complejas no se guardan imágenes ya que sería muy costoso y ralentizaría considerablemente el tiempo que tarda en completarse todo el proceso.

En la figura B.2 se observa la estructura de carpetas creada bajo la librería *SBPL*.

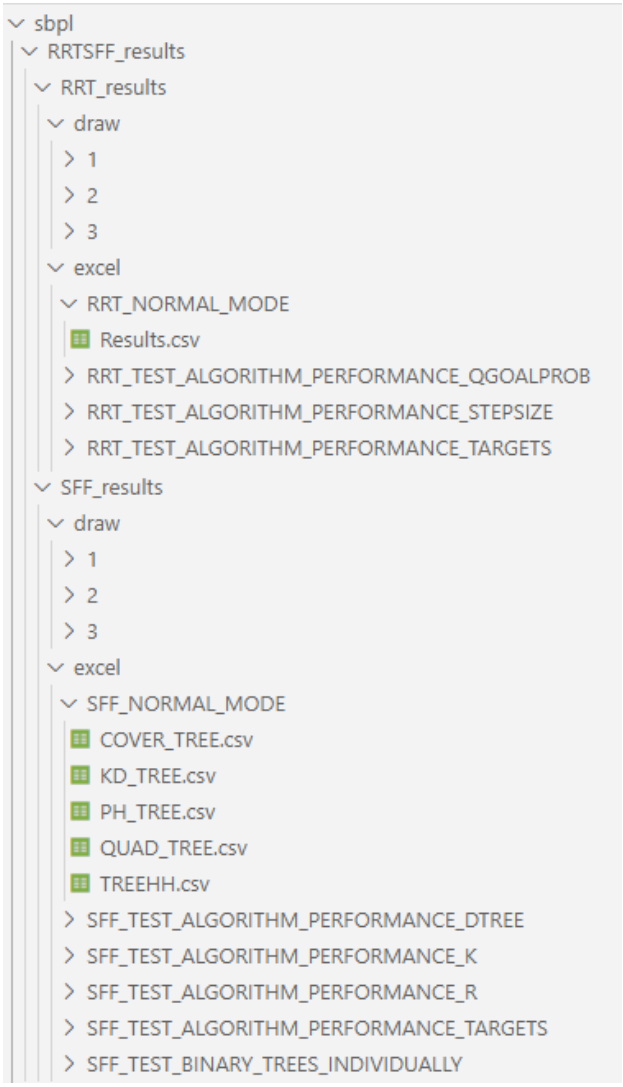


Figura B.2: Estructura de carpetas creada para almacenar los resultados obtenidos.

Se observa como de la raíz de *SBPL* cuelga una carpeta llamada *RRTSFF_results*. A su vez se han creado dos carpetas llamadas *RRT_results* y *SFF_results* en el interior de las cuales se hayan dos carpetas más: *draw* y *excel*.

En la carpeta *draw* se van creando subcarpetas con numeración ascendente en las que se guardan las figuras de la última ejecución realizada.

Por otro lado, en la carpeta *excel* se hayan diversas subcarpetas en función de la prueba que se haya realizado. Dentro de cada carpeta se crea un archivo *.csv* el cual contiene la información acerca de la última prueba realizada. En el caso de *SFF*, esta carpeta varía su nombre en función del árbol que se haya utilizado, en el caso de *RRT* estas carpetas siempre se llaman igual.

Bibliografía

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, 1975. (document), 4.2.1, 4.2.2

[2] "K-d tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=K-d_tree&oldid=1096839907, Accedido por última vez el 21-08-2022. (document), 4.2.1, 4.2.2

[3] A. Beygelzimer, S. M. Kakade, and J. Langford, "Cover trees for nearest neighbor," *Proceedings of the 23rd international conference on Machine learning*, 2006. (document), 4.2.1, 4.2.3

[4] "Cover tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=Cover_tree&oldid=1076477448, Accedido por última vez el 21-08-2022. (document), 4.2.1, 4.2.3

[5] R. Finkel and J. Bentley, "Quad trees: A data structure for retrieval on composite keys." *Acta Inf.*, vol. 4, pp. 1–9, 03 1974. (document), 4.2.1, 4.2.4

[6] "Quadtree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=Quadtree&oldid=1104066964, Accedido por última vez el 21-08-2022. (document), 4.2.1, 4.2.4

[7] T. Zäschke, C. Zimmerli, and M. Norrie, "The ph-tree - a space-efficient storage structure and multi-dimensional index," 06 2014, pp. 397–408. (document), 4.2.1, 4.2.5

[8] "Ph-tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=PH-tree&oldid=1097298390, Accedido por última vez el 21-08-2022. (document), 4.2.1, 4.2.5

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959. 1.1

[10] "Dijkstra's algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1094953874, Accedido por última vez el 21-08-2022. 1.1

[11] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://doi.org/10.1109/tssc.1968.300136 1.1

[12] "A* search algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1096126322, Accedido por última vez el 21-08-2022. 1.1

[13] "Bellman–ford algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Bellman-Ford_algorithm, Accedido por última vez el 21-08-2022. 1.1

[14] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996. 1.1

[15] "Probabilistic roadmap — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Probabilistic_roadmap, Accedido por última vez el 21-08-2022. 1.1

[16] S. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," *IEEE Transactions on Robotics and Automation*, 1998. 1.1, 1.1, 1.1.1, 3.1

[17] "Rapidly-exploring random tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree, Accedido por última vez el 21-08-2022. 1.1

[18] K. L. H. Choset, *Principles of Robot Motion*. MIT Press, Cambridge, Massachusetts, London, England, 2005, ch. 3. Configuration space. 1.1

[19] "Configuration space (physics) — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Configuration_space_(physics), Accedido por última vez el 21-08-2022. 1.1

[20] V. B. Sivaranjani Arthanari*, "Performance comparison of sampling-based algorithms for path planning of a mobile robot," *International Journal of Sciences and Applied Research*. 1.1

[21] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011. 1.1.1

[22] J. Nasir, F. Islam, U. A. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. Muhammad, "Rrt*-smart: A rapid convergence implementation of rrt*„" *International Journal of Advanced Robotic Systems*, vol. 10, p. 299, 07 2013. 1.1.1

[23] V. Vonásek and R. Pěnička, "Space-filling forest for multi-goal path planning," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 1587–1590. 1.1.1, 3.2

[24] "Travelling salesman problem — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Travelling_salesman_problem, Accedido por última vez el 21-08-2022. 1.2, 3.3.1

[25] "Willow garage — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Willow_Garage, Accedido por última vez el 21-08-2022. 2.1

[26] "Explanation greedy vs. heuristic algorithm," 2022, https://www.baeldung.com/cs/greedy-vs-heuristic-algorithm, Accedido por última vez el 21-08-2022. 2.1

[27] R. Ebendt and R. Drechsler, "Weighted a* search – unifying view and application," *Artificial Intelligence*, vol. 173, no. 14, pp. 1310–1342, 2009. 2.1

[28] "Standalone program — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/w/index.php?title=Standalone_program&oldid=1000487309, Accedido por última vez el 21-08-2022. 2.1

[29] ROS, "Robot operating system," https://www.ros.org/, Accedido por última vez el 21-08-2022. 2.1

[30] "Robot operating system — wikipedia, la enciclopedia libre," 2021, https://es.wikipedia.org/wiki/Robot_Operating_System, Accedido por última vez el 21-08-2022. 2.1

[31] B. Gerkey, "Gmapping package ros," http://wiki.ros.org/gmapping, Accedido por última vez el 21-08-2022. 2.1

[32] "Simultaneous localization and mapping — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping, Accedido por última vez el 21-08-2022. 2.1

[33] "Adjacency list — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Adjacency_list, Accedido por última vez el 21-08-2022. 2.1.1

[34] G. G. Maxim Likhachev and S. Thrun, "Ara*: Anytime a* with provable bounds on sub-optimality," https://papers.nips.cc/paper/2003/file/ee8fe9093fbbb687bef15a38facc44d2-Paper.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[35] A. H. Jur van den Berg1, Rajat Shah and K. Goldberg, "Ana*: Anytime nonparametric a*," https://goldberg.berkeley.edu/pubs/ana-aaai-2011-vdberg-shah-goldberg.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[36] G. G. A. S. Maxim Likhachev, Dave Ferguson and S. Thrun, "Anytime dynamic a*: An anytime, replanning algorithm," http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[37] "Anytime algorithm — Wikipedia, the free encyclopedia," 2020, https://en.wikipedia.org/wiki/Anytime_algorithm, Accedido por última vez el 21-08-2022. 2

[38] D. N. Y. Isaac, "Lazy ara* information," p. 39. 2.1.2

[39] "Lazy learning — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Lazy_learning, Accedido por última vez el 21-08-2022. 2.1.2

[40] "Lazy evaluation — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Lazy_evaluation, Accedido por última vez el 21-08-2022. 2.1.2

[41] A. S. Maxim Likhachev, "Ppcp: Efficient probabilistic planning with clear preferences in partially-known environments," https://www.aaai.org/Papers/AAAI/2006/AAAI06-136.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[42] ——, "R* search," https://www.cs.cmu.edu/~maxim/files/rstar_aaai08.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[43] BURLAP, "Basic planning and learning, section planning with value iteration," http://burlap.cs.brown.edu/tutorials_v1/bpl/p4.html, Accedido por última vez el 21-08-2022. 2.1.2

[44] "Stochastic programming — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Stochastic_programming, Accedido por última vez el 21-08-2022. 2.1.2

[45] V. N. V. H. M. L. Sandip Aine, Siddharth Swaminathan, "Multi-heuristic a*," http://www.roboticsproceedings.org/rss10/p56.pdf, Accedido por última vez el 21-08-2022. 2.1.2

[46] "Matlab," https://www.mathworks.com/products/matlab.html, Accedido por última vez el 21-08-2022. 2.1.4

[47] "Matlab — wikipedia, la enciclopedia libre," 2022, https://es.wikipedia.org/wiki/MATLAB, Accedido por última vez el 21-08-2022. 2.1.4

[48] "Matlab toolbox distribution," https://es.mathworks.com/help/matlab/creating-help.html?lang=en, Accedido por última vez el 21-08-2022. 2.1.4

[49] "Gimp — wikipedia, la enciclopedia libre," 2022, https://es.wikipedia.org/w/index.php?title=GIMP&oldid=144377918, Accedido por última vez el 21-08-2022. 2.1.4

[50] "Gimp," https://gimp.es/, Accedido por última vez el 21-08-2022. 2.1.4

[51] "Holonomic constraints — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Holonomic_constraints, Accedido por última vez el 21-08-2022. 3.1

[52] "Nonholonomic system — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Nonholonomic_system, Accedido por última vez el 21-08-2022. 3.1

[53] Mecharithm, "Explanation of holonomic and nonholonomic constraints," https://www.mecharithm.com/holonomic-nonholonomic-constraints-robots/, Accedido por última vez el 21-08-2022. 3.1

[54] "Np-completo — wikipedia, la enciclopedia libre," 2022, https://es.wikipedia.org/wiki/NP-completo, Accedido por última vez el 21-08-2022. 3.3.1

[55] "Brute-force search — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Brute-force_search, Accedido por última vez el 21-08-2022. 3.3.3

[56] "Permutation — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Permutation, Accedido por última vez el 21-08-2022. 3.3.3

[57] "Dynamic programming — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Dynamic_programming, Accedido por última vez el 21-08-2022. 3.3.3

[58] "Held–karp algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/HeldâĂŞKarp_algorithm, Accedido por última vez el 21-08-2022. 3.3.3

[59] "Branch and bound — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Branch_and_bound, Accedido por última vez el 21-08-2022. 3.3.3

[60] "Linear programming — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Linear_programming, Accedido por última vez el 21-08-2022. 3.3.3

[61] M. W. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM Rev.*, vol. 33, pp. 60–100, 1991. 3.3.3

[62] "Branch and cut — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Branch_and_cut, Accedido por última vez el 21-08-2022. 3.3.3

[63] "Nearest neighbour algorithm — Wikipedia, the free encyclopedia," 2021, https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm, Accedido por última vez el 21-08-2022. 3.3.3

[64] "Greedy algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Greedy_algorithm, Accedido por última vez el 21-08-2022. 3.3.3

[65] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," 1976. 3.3.3

[66] "Christofides algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Christofides_algorithm, Accedido por última vez el 21-08-2022. 3.3.3

[67] "Minimum spanning tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Minimum_spanning_tree, Accedido por última vez el 21-08-2022. 3.3.3

[68] "Matching (graph theory) — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Matching_(graph_theory), Accedido por última vez el 21-08-2022. 3.3.3

[69] "Match twice and stitch: a new tsp tour construction heuristic," *Operations Research Letters*, vol. 32, no. 6, pp. 499–509, 2004. 3.3.3

[70] "Lin–kernighan heuristic — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/LinâĂŞKernighan_heuristic, Accedido por última vez el 21-08-2022. 3.3.3

[71] "2-opt — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/2-opt, Accedido por última vez el 21-08-2022. 3.3.3

[72] "3-opt — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/3-opt, Accedido por última vez el 21-08-2022. 3.3.3

[73] "Tabu search — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Tabu_search, Accedido por última vez el 21-08-2022. 3.3.3

[74] "Genetic algorithm — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Genetic_algorithm, Accedido por última vez el 21-08-2022. 3.3.3

[75] "Simulated annealing — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Simulated_annealing, Accedido por última vez el 21-08-2022. 3.3.3

[76] "Ant colony optimization algorithms — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms, Accedido por última vez el 21-08-2022. 3.3.3

[77] "Triangle inequality — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/wiki/Triangle_inequality, Accedido por última vez el 21-08-2022. 3.3.3

[78] F. B. J. B. A. G. A. L. Saux, "Final project - travelling salesman problem," 2020, https://github.com/adrlsx/Travelling-Salesman-Problem, Accedido por última vez el 21-08-2022. 3.3.4, 3.3.4, 4.4.1, 4.4.1

[79] ROS, "Isen nantes," https://isen-nantes.fr/, Accedido por última vez el 21-08-2022. 3.3.4

[80] "Ini file — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=INI_file&oldid=1103400814, Accedido por última vez el 21-08-2022. 4.1.2

[81] "Matplotlibcpp," https://matplotlib-cpp.readthedocs.io/en/latest/, Accedido por última vez el 21-08-2022. 4.1.2, 4.1.6

[82] "Binary tree — Wikipedia, the free encyclopedia," 2022, https://en.wikipedia.org/w/index.php?title=Binary_tree&oldid=1092722102, Accedido por última vez el 21-08-2022. 4.1.3

[83] "Matplotlib," https://matplotlib.org/, Accedido por última vez el 21-08-2022. 4.1.6

[84] "Vector library c++," https://cplusplus.com/reference/vector/vector/, Accedido por última vez el 21-08-2022. 4.1.7, 4.9

[85] "Memory library c++," https://cplusplus.com/reference/memory/, Accedido por última vez el 21-08-2022. 4.1.7, 4.9

[86] “Memory leak — Wikipedia, the free encyclopedia,” 2022, https://en.wikipedia.org/w/index.php?title=Memory_leak&oldid=1099315217, Accedido por última vez el 21-08-2022. 4.1.7

[87] “Standard template library — Wikipedia, the free encyclopedia,” 2022, https://en.wikipedia.org/wiki/Standard_Template_Library, Accedido por última vez el 21-08-2022. 4.2.1

[88] “K-dimensional tree library,” https://rosettacode.org/wiki/K-d_tree, Accedido por última vez el 21-08-2022. 4.2.2

[89] “Covertree library,” https://github.com/DNCrane/Cover-Tree, Accedido por última vez el 21-08-2022. 4.2.3

[90] “Quadtree library,” https://github.com/andyzhshg/qtree, Accedido por última vez el 21-08-2022. 4.2.4

[91] “Ph-tree library,” https://github.com/improbable-eng/phtree-cpp, Accedido por última vez el 21-08-2022. 4.2.5

[92] “The bresenham line algorithm,” https://csustan.csustan.edu/~tom/Lecture-Notes/Graphics/Bresenham-Line/Bresenham-Line.pdf, Accedido por última vez el 21-08-2022. 4.6.2

[93] “Euclidean distance — Wikipedia, the free encyclopedia,” 2022, https://en.wikipedia.org/wiki/Euclidean_distance, Accedido por última vez el 21-08-2022. 4.6.2

[94] “Manhattan distance — Wikipedia, the free encyclopedia,” 2022, https://simple.wikipedia.org/wiki/Manhattan_distance, Accedido por última vez el 21-08-2022. 4.6.2

[95] “Chaikin’s algorithms for curves,” https://www.cs.unc.edu/~dm/UNC/COMP258/LECTURES/Chaikins-Algorithm.pdf, Accedido por última vez el 21-08-2022. 4.7.3

[96] “Matplotlibcpp library,” https://github.com/lava/matplotlib-cpp, Accedido por última vez el 21-08-2022. 4.8