Generador de Mapas con SLAM y RTT

Luis Menéndez García
MSc Robótica y Automatización
ragnemul@gmail.com

***Abstract*—En este documento se describe el método empleado para implementar un simulador sobre MATLAB que realiza la exploración de mapas y la navegación hasta el punto inicial una vez completada la exploración de la superficie disponible.**

***Index Terms*—SLAM, RTT, Robot, map**

I. Introducción

El objetivo general del trabajo consiste en elaborar un mapa basándose exclusivamente en la navegación autónoma de un robot, sin ningún tipo de ayuda ni intervención externa, el cual estará dotado de sensores LIDAR.

Una vez elaborado el mapa, el robot deberá ser capaz de volver al punto inicial usando el mapa generado en la etapa anterior.

Se ha querido implementar un algoritmo de navegación autónoma usando SLAM, para lo cual se ha partido de uno de los ejemplos propuestos en la asignatura, denominado SLAM_Zaragoza. Este ejemplo sólo implementa la detección de una serie de obstáculos prefijados junto con el algoritmo SLAM para la localización del robot y de los obstáculos. En el presente trabajo se ha completado este ejemplo con cerca de 1.400 líneas de código adicionales, implementando numerosos algoritmos y heurísticas, para soportar navegación autónoma para la exploración y realizar el recorrido del mapa elaborado.

Con el objetivo de simplificar los cálculos, se ha decidido adaptar el mapa original reduciendo el número de landmarks, manteniendo los mismos límites del mapa original, tal y como puede observarse en la figura 3

II. Descripción General del algoritmo

En la figura 1 puede observarse la descripción del algoritmo principal desarrollado en el presente trabajo. Inicialmente se lee el mapa del fichero csv, se generan las estructuras de datos, y se insertan los movimientos iniciales mediante la función *InsertarMovimientosIniciales*. Estos movimientos iniciales se insertan en la estructura de datos *ground.motion* para que el robot gire sobre sí mismo 360º, con el objetivo de que reconozca los osbtáculos que tiene alrededor, y así pueda ir planificando sus movimiento a más largo alcance con seguridad.

Después se entra en el bucle principal –líneas 2 a 18–, que se ejecutará mientras queden movimientos planificados a ejecutar, o mientras el perímetro reconocido o la superficie recorrida estén por debajo de cierto valor.

En la línea 4, mediante la función *Planifica()* se procede a insertar nuevos movimientos planificados dentro de la cola de ejecución. Se procederá a describir esta función más adelante.

En las líneas 6 a 8, ejecutadas después de llamar a la función *Planifica()*, si no hay movimientos en la cola de ejecución, significa que el planificador no encontró lugares libres para explorar desde la ubicación actual del robot, y se llama a la función *RouteToEnd(*P_ini*)* para crear un grafo sobre la superficie explorada que nos permita volver a un punto donde pueda continuar la exploración. Se describirá esta función más adelante.

En la línea 10 se verifica si el movimiento actual que se ha llevado a cabo era programado de una planificación anterior, y si es así, no se planifican nuevos movimientos desde este punto.

La función *VeHaciaLaLuz()* analiza el perímetro descubierto hasta la fecha y calcula una apertura hacia el mismo, y localiza dentro del espacio recorrido del robot un punto con conectividad hacia esa región, con el objetivo de planificar movimientos hacia ella y así poder cerrar el perímetro. Se describirá esta función más adelante.

En las líneas 14 a 16 se verifica, mediante la llamada a la función *superficieRecorrida()*, la superficie recorrida y el perímetro descubierto, y se da por terminada la elaboración del mapa cuando el perímetro descubierto es superior al 97% y la superficie recorrida es superior al 90%. Se describirá está función más adelante.

Se sale del bucle principal sólo cuando se cumple el criterio de superficie recorrida y perímetro descubierto, y así en la línea 20, se llama a la función *Route()* para buscar una ruta desde la posición actual del robot hasta la posición de salida según el mapa que se ha elaborado.

Las líneas 21 a 22 muestran la superficie recorrida y el perímetro descubierto, y la línea 23 muestra el mapa en formato binario que se ha elaborado.

Como se verá más adelante, se han desarrollado algoritmos de exploración basados en sampling aleatorio, mientras que para la planificación entre zonas no exploradas –con un reconocimiento parcial del mapa– y la planificación final –con exploración total o prácticamente total– se ha implementado un algoritmo basado en Rapidly Exploring Random Trees (RTT), modificado convenientemente para agilizar la resolución del problema, al que hemos denominado RTT+, y que difiere del original en que logra la conectividad con el punto inicial más rápidamente que el anterior.

Se han implementado diferentes heurísticas en un marco de competición para hacer recorridos en el corto plazo y planificación de recorridos a larga distancia haciendo navegación sobre el mapa que se va generando.

```
Algoritmo Principal
1.  InsertarMovimientosIniciales;
2.  while !empty(motion)
3.
4.    Planifica();
5.
6.    if (!motion)
7.      RouteToEnd(P_ini)
8.      break
9.
10.   if movimientoProgramado
11.     continue
12.
13.   VeHaciaLaLuz();
14.   [perimetro,superficie] = superficieRecorrida ()
15.   if perimetro ≥ 0.97 or superficie ≥ 0.90
16.     break
17.
18. end
19.
20. Route(P_ini)
21. mostrarSuperficieRecorrida
22. mostrarPerimetroRecorrido
23. mostrarMapaObstaculos();
```

Fig. 1: Algoritmo principal

El algoritmo comienza generando una vuelta completa sobre el sitio con el objetivo de comprobar qué obstáculos rodean al robot y así poder fijar una primera estrategia de búsqueda de espacios libres para poder moverse dentro del rango del LIDAR. Dicha búsqueda puede observarse en la figura 4.

Cuando no es posible obtener pasos libres cercanos a la posición del robot, entra en escena la heurística lejana que busca lugares abiertos en el perímetro del mapa generado, y realiza una planificación de movimientos sobre el espacio conocido del robot, posicionándolo en el punto libre más cercano al hueco observado en el perímetro, tal y como se obseva en la figura 5

Cada ciertos ciclos de ejecución se comprueba el grado de completitud de la exploración, atendiendo a dos parámetros: la uniformidad del perímetro descubierto y el porcentaje de terreno explorado, para lo cual se divide el espacio de configuraciones en celdas y se realiza un conteo y estimación de los trazados, tal y como puede verse en la figura 6

A continuación se detallan las funciones anteriormente descritas durante la explicación del algoritmo general.

A. Insertar Movimientos Iniciales

Con el objetivo de que el robot pueda hacer planificaciones con el mayor alcance posible, se ha optado por insertar estos movimientos iniciales que hacen girar al robot 360º sobre la posición inicial, de manera que tenga un mapa cercano y así se pueda planificar mediante sampling aleatorio los movimientos

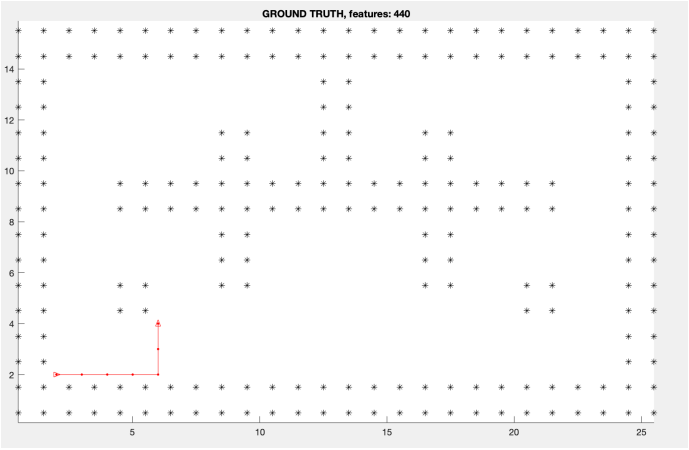

Fig. 2: Simplificación de Landmarks para facilitar el cómputo

dentro del rango del LIDAR sin riesgo de colisionar con algún obstáculo.

B. Planificación

La planificación de los movimientos se realiza coordinando la heurística cercada con la lejana, simulando un proceso BackTracking sin recursividad, para lo que se ha empleado una estructura tipo cola. Esto ha permitido meter movimientos planificados que se ejecutarán en ciclos posteriores cuando se quiere que el robot planifique movimientos sobre el mapa a zonas distantes de la actual. Cuando la heurística cercana no produce ningún lugar al que ir, entra en escena la heurística lejana, que analizará el mapa generado hasta la fecha, y buscará nuevos lugares de exploración.

```
Planifica
1.  % hacemos un sampling de los puntos sobre la zona cercana
2.  sampling_points = Sampling
3.  % se obtienen los puntos libres de colisiones de la po. Act.
4.  candidates = FreeWay(sampling_points)
5.  if (candidates)
6.    % se descartan los candidatos que estén cerca de lugares visitados
7.    Candidates = discardVisited(candidates)
8.
9.    Target = selectCandidate(candidates,'euclidean');
10.   Giro = calcTurn();
11.
12.   % se inserta un giro hacia el candidato
13.   InsertarGiro;
14.   % se inserta un movimiento de cierta distancia hacia el candidato
15.   InsertarMovimiento (getStepDistance());
16. end
```

Fig. 3: Algoritmo de planificación

En la linea 2 se realiza un sampling aleatorio de radio $2 * \pi$ sobre la posición actual del robot, con radio desde 1 hasta el alcance máximo del LIDAR.

En la línea 4, mediante la función *FreeWay* se comprueba si los puntos del sampling tienen conectividad directa desde la posición actual del robot, es decir, si no atraviesan ningún obstáculo del mapa elaborado hasta ese momento, y si no pasan muy cerca de alguno de ellos. Se deja la lista en candiadtes.

Si hay algún candidato, en la 7 se descartan aquellos que queden cerca de zonas ya exploradas, y de esta manera se

facilita que el robot priorice los pasos largos sobre zonas inexploradas sobre las zonas ya exploradas.

En la línea 9 se selecciona el candidado más lejano respecto a la posición actual del robot empleando distancia euclídea.

En la línea 10 se calcula el giro que ha de planificarse para que el robot se oriente hacia ese target, y en la línea 13 se planifica el movimiento indicando que fue planificado y que por lo tanto no requiere planificación sobre ese punto cuando se ejecute esta instrucción

En la línea 15 se inserta el movimiento en la cola de ejecución con el paso calculado y se indica que cuando se ejecute este movimiento se deberá planificar otra vez sobre esta posición.

De manera esquemática, las funciones arriba comentadas, se enumeran con el nombre equivalente de fichero MatLab, con una pequeña descripción.

- samplimg.m: realiza un muestreo aleatorio sobre la zona cercana del robot, dentro del rango del LIDAR, entre los ángulos 0 y 2π.
- FreeWay.m: obtiene los puntos del Cspace libres desde la posición actual a cada uno de los puntos cercanos muestreados.
- discardVisited.m: Se descartan los candidatos que están cerca de obstáculos y de zonas ya visitadas
- getStepDistance.m: obtiene la distancia entre la posición actual y el punto candidato.
- selectCandidate.m: prioriza el uso de candidato lejanos frente a los cercanos
- directView.m: indica si hay camino directo libre de osbtáculos entre el punto cercano y un candidado.

1) Heurística cercana: La heurística cercana implementada se basa en la disposición alatoria de puntos de muestreo dentro del rango del Lidar, priorizando aquellos puntos que estén alejados de otros puntos ya visitados o cerca de los obstáculos. La implementación de esta medida logró que el robot avanzara más en línea recta hacia zonas libres aún no exploradas.

Algunas de las funciones implementadas para la gestión de la heurística cercana fueron:

- discardVisited.m
- select Candidate.m
- select calcTurn.m, que calcula el giro que hay que darle al robot en función de la posición actual y el punto de destino

C. Heurística lejana

La heurística lejana busca zonas del mapa no visitadas analizando el perímetro del mapa explorado hasta el momento y los casillas que no han sido recorridas por el robot y que están libres del eco del LIDAR. Como puede observarse en la figura 5, cuando el análisis del perímetro muestra que no es cerrado, se realiza una comprobación para verificar qué puntos del recorrido del robot tienen visibilidad directa, y se asigna el punto más cercano con visibilidad directa como el punto al que el robot debe dirigirse para iniciar una aproximación hacia la zona, que será completada usando la heurística cercana.

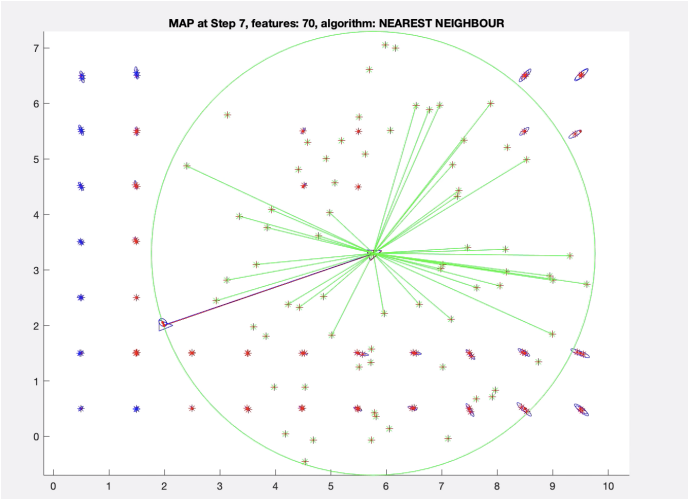

Fig. 4: Heurística para búsqueda de espacio libre cerca del robot

Este método necesita que se realice una planificación basada en RTT, implementada también para este trabajo, sobre los puntos recorridos por el robot, para que sea un movimeiento rápido, ya que este recorrido está libre de obstáculos.

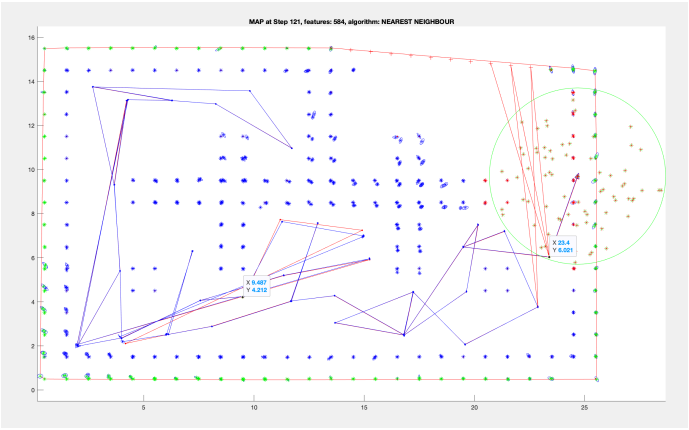

Fig. 5: Heurística para búsqueda de espacio libre no explorado

D. Detección de exploración completa

Cada ciertos ciclos de movimiento, o bien cuando no se encuentre un lugar libre al que ir según los heurísticos disponibles se activará la función ⌈superficieRecorrida, que realizará la estimación del perímetro explorado y de las celdas no exploradas o exploradas muy parcialmente, generando una ruta de exploración hacia el lugar.

En caso de que el perímetro se haya explorado por encima del 97% y de que la superficie mapeada alcance al menos el 90% se dará por terminada el proceso , mostrando los porcentajes de exploración.

E. Planificación con Rapidly Exploring Random Trees

Se ha de dar solución a un problema de planificación sobre plano para un robot móvil que puede moverse sobre el espacio

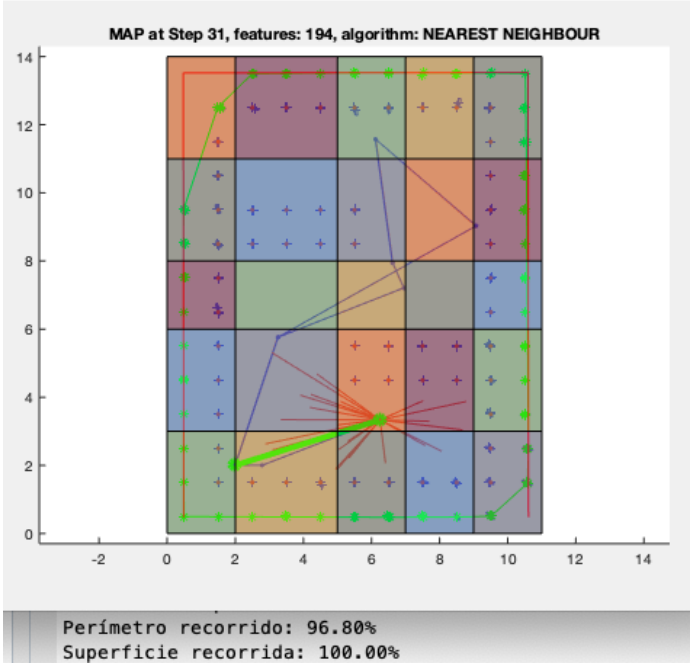

Fig. 6: Detección de expliración completa

de configuraciones $R^2 x S_1 = SE(2)$ para que desplace entre dos puntos cualesquiera del mapa, con la premisa de que se conoce el plano de antemano, y éste es estático. Para conseguir estos objetivos se ha desarrollado una variante del algoritmo RTT descrita en [1], denominada RTT+, que permite realizar un camino de conexión libre de obstáculos entre el punto inicial q_i y el final q_f, sin necesidad de completar el muestreo aleatorio de todos los puntos del espacio de configuraciones libre C_free, dando como resultado el camino de conexión más corto entre q_iy q_f, según el grafo de conectividad G que se haya ido formando, o bien mostrando que no hay camino posible debido a una insuficiencia del muestreo de puntos aleatorios sobre el espacio C_free.

Se ha implementado una modificación del algoritmo RTT para facilitar tanto la navegación global según la heurística lejana, como la navegación al punto inicial una vez recorrido todo el entorno. Para implementar el algoritmo se ha usando funciones de grafos disponibles en MatLab. La implementación de RTT se ha realizado sin usar algoritmos de MatLab. De hecho se ha modificado el algoritmo original para hacer que converja más rápidamente, denominándolo RTT+.

El algoritmo necesita como entrada un conjunto de muestreo aleatorio sobre C_free, distribuido uniformemente, y que se denominara C_sam, sobre el cual se construirá el grafo G de conectividad empleando una métrica L^2, así como los puntos de inicio y final qi y qf respectivamente, y el mapa binario que representa tanto los espacios pertenecientes a Cfree. En la figura 9 puede observarse el algoritmo general implementado.

El uso de este algoritmo permite realizar planificaciones rápidas, en función de la búsqueda sobre el grafo generado, y añadir nuevos subgrafos al existente, encontrando caminos

RRT+ $(q_i, q_f, C_{sam}, C_{obs}, \mathcal{G})$
1. $\mathcal{G}.init(q_i)$
2. **while** $!\,empty(C_{sam})$
3. $p = GetFirst(C_{sam})$
4. n = $Nearest(\mathcal{G}, p, C_{obs})$
5. **if** $(!\,n)$
6. $C_{sam} = InsertAtEnd(C_{sam}, p)$
7. **continue**
8. **if** $FreePath(\mathcal{G}, C_{obs}, q_f)$
9. **break**
10. $\mathcal{G}.add\_edge(n, p)$
11. $\mathcal{G}.add\_edge(n, q_f)$

Fig. 7: Algoritmo RTT modificado

con el menor coste posible y más directos.

Como puede observarse el algoritmo comienza introduciendo el punto de salida q_i, como un vértice sobre el grafo G (línea 1), para continuar explorando el conjunto de muestreo aleatorio C_sam, que en esta variación sobre el algoritmo original se trata como una pila. Se extrae el último punto p de C_sam (línea 3) y se obtiene el vértice n del grafo conectado G más cercano al punto p que puede formar un camino libre de obstáculos (línea 4). Si no se puede obtener un path libre entre los puntos n y p, se vuelve a encolar el punto p al final de C_sam y se fuerza a explorar una nueva iteración (líneas 5 – 7). Después se comprueba si se puede conectar el punto final q_f con el grafo conectado G, y si es así se interrumpe la ejecución del bucle (líneas 8 y 9). Finalmente se conecta el punto q_f al grafo conectado G (línea 1). Finalmente se hallará el camino más corto entre los puntos q_i y q_f del grafo conectado G, empleando técnicas de recorridos sobre grafos. En función de la distribución de los puntos de muestreo aleatorio C_sam, esta técnica es sensiblemente más rápida que la mostrada por el algoritmo RTT estándar, al no ser preciso construir completamente el grafo G usando todos los puntos de C_sam.

En las figuras 8 se puede observar cómo se configura el grafo mediante el algoritmo RTT+ implementado, sobre diferentes tipos de mapas. El recorrido que conecta los dos puntos se puede apreciar en color verde.

A continuación se especifican los subalgoritmos empleados en el desarrollo del algoritmo de búsqueda del camino una vez completado un mapa, completamente o parcialmente.

1) Función Nearest: La función NEAREST devuelve el punto $n \in G$ más cercano al punto explorado $p \in C_sam$, siempre que el segmento que une n y p no colisione con ningún obstáculo.

Para la elección del punto $n \in G$ más cercano se ha empleado la distancia euclídea.

2) Detección de obstáculos: Para la detección de los obstáculos se ha optado por localizar sobre el bitmap los

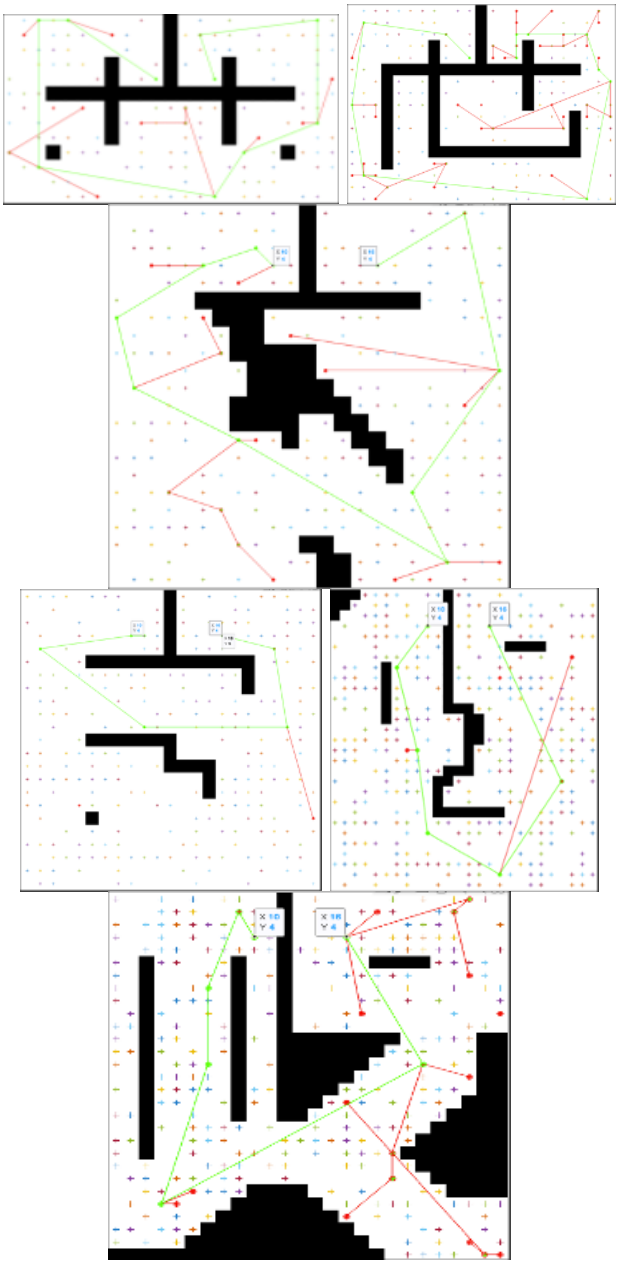

Fig. 8: Algoritmo RTT modificado

puntos contiguos con conectividad 8. Esta será la estructura base que se consultará cuando se necesite comprobar si el segmento que une $p \in C_sam$ con el punto más cercano $n \in G$, empleando una métrica L^2.

3) Detección de path libre: Se han diseñado e implementado dos algoritmos para la detección del path libre entre dos puntos, que se detallan a continuación. Primera aproximación para la detección de path libre El primer algoritmo –ArcoLibre– se basa en la obtención del perímetro de cada pixel que compone el obstáculo detectado $Pixel_obs$, entiendo como pixel la unidad de construcción básica de dicho obstáculo.

En el proceso de construcción del grafo G, para cada segmento candidato a ser incorporado se examina si se produce la intersección con cada uno de los perímetros del conjunto $Pixel_obs$. Si no se produce la intersección del segmento éste pasa a considerarse un camino libre y se incorpora a G. En caso contrario el segmento en cuestión se descarta. Este primer algoritmo nos permitirá abordar una evolución del presente trabajo para poder incorporar caminos semi libres, al obtener, si el segmento no es libre, el punto de intersección con C_obs.

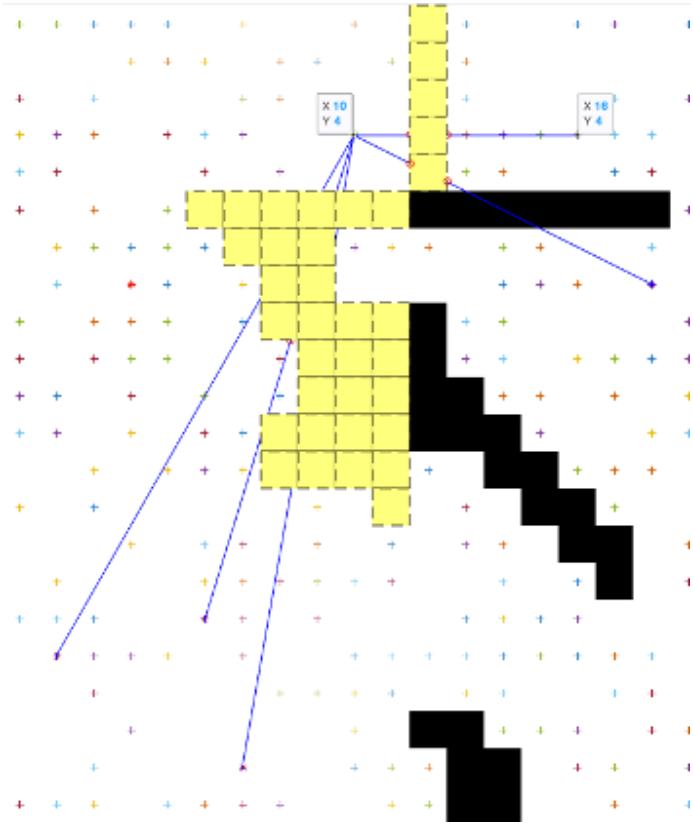

Fig. 9: Verificación path libre en función ArcoLibre

4) Obtención del punto más cercano:

F. Generación del mapa

Una vez que el terreno se ha explorado según los criterios de finalización, se llama a la función ImprimMapaObstaculos para mostrar por pantalla los obtáculos detectados y para generar el fichero csv. En la imagen 10 pueden obserarse los mapas de referencia y el obtenido. En la figura 11 se puede observar el contenido del fichero .csv correspondiente.

III. Conclusiones

El uso de técnicas probabilísticas ha permitido recorrer el mapa de manera completa, llegando a explorar áreas recónditas, gracias a la implementación de diferentes heurísticas..

Algunos puntos de mejora podrían consistir en enriquecer el grafo con el recorrido final para hacer más suaves las trayectorias, y también aumentar el numero de heurísticas para hacer que el mapa se explore de manera más rápida y

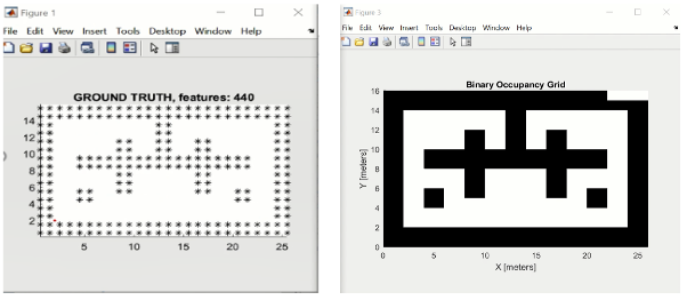

Fig. 10: Imágenes del mapa original a la izquierda, y del mapa elaborado a la derecha

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,1
1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,1
1,1,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,0,0,1,1
1,1,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,0,0,1,1
1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1
1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1
1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1
1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1
1,1,0,0,1,1,0,0,1,1,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1
1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1
1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1
1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Fig. 11: Ifichero csv con el mapa generado

directa. También hubiera tenido sentido introducir elementos de navegación semántica, como el reconocimiento de ciertos lugares, para poder hacer mención a ellos en el proceso de planificación de las rutas.

En la figura 12 puede apreciarse el ejemplo de ejecuación del mapa3, con el grafo del recorrido de vuelta una vez completado el mapa.

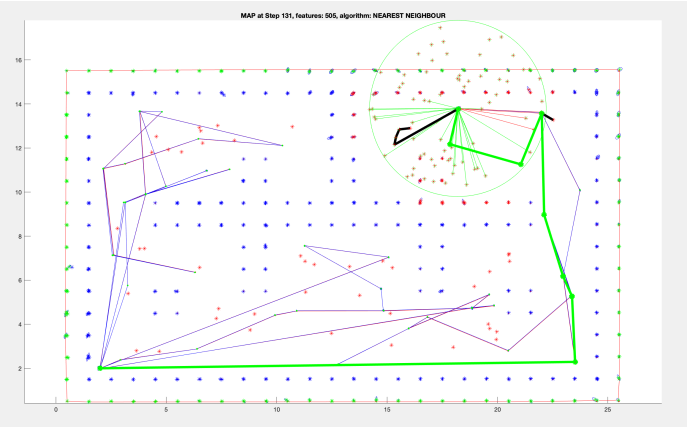

Fig. 12: IEjemplo de exploración completa

IV. Listado de funciones desarrolladas

Estas son las funciones desarroladas para lograr la navegación autónoma y la planificación sobre el mapa mediante RTT.

Listing 1: Código implementado

```
CircleIntersect.m
FreeWay.m
ImprimeMapaObstaculos.m
InsideCircle.m
PixelPerimeter.m
Route.m
addRoute.m
blob_rectangles.m
borderClosed.m
borderLine.m
calcTurn.m
closeEnough.m
closedToSegment.m
directView.m
discardVisited.m
fillLine.m
getObstacles.m
getStepDistance.m
getTrajectory.m
insert_movs_360.m
load_map.m
minBoundingBox.m
navSampling.m
nearestPoint.m
planifica.m
pol2cart_off.m
posIdx.m
routeToEnd.m
sampling.m
selectCandidate.m
superficieRecorrida.m
veHaciaLaLuz.m
vectorRadians.m
```

REFERENCES

[1] [1] LaValle, S. (2006). Planning Algorithms. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511546877