Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Comparativa de algoritmos de navegación de interiores mediante el uso de un robot Pioneer 3-DX

ESCUELA POLITECNICA SUPERIOR

Autor: Roberto Cuesta Lucas

Tutor: Ángel Llamazares Llamazares

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Comparativa de algoritmos de navegación de interiores mediante el uso de un robot Pioneer 3-DX

Autor: Roberto Cuesta Lucas

Tutor: Ángel Llamazares Llamazares

Tribunal:

Presidente: IGNACIO BRAVO MUÑOZ

Vocal 1º: SIRONA VALDUEZA FELIP

Vocal 2º: ÁNGEL LLAMAZARES LLAMAZARES

Fecha de depósito: 15 de septiembre de 2022

Quisiera agradecerle a mis padres, así como a mi hermano, todo el apoyo incondicional que me han brindado durante todo este tiempo, por ayudarme durante tantas horas dedicadas a editar y corregir este proyecto.

Darle las gracias también a mis amigos, los cuales me han acompañado y soportado durante tantos años.

Gracias a todos vosotros he podido seguir adelante y en estos momentos puedo seguir disfrutando de la informática.

Muchas gracias a todos :)

Resumen

En la actualidad, existe una creciente implantación de los robots móviles en espacios interiores, tanto en el entorno doméstico como en el industrial. Debido a esta creciente demanda, es necesario comparar los algoritmos de navegación existentes con el fin de decidir el más adecuado en cada situación. Por ello, en este proyecto se pretenden comparar distintos métodos de navegación por entornos interiores. Estas comparativas se realizarán tanto en simulación como en entornos reales. Para ello se va a utilizar un robot Pioneer 3-DX, al cual se le incorporará el hardware y software necesario para poder realizar las pruebas. Para ello, se dotará al robot de un sensor LiDar Hokuyo URG-04LX y se instalará un PC de bajo coste, como es Raspberry Pi. Desde un dispositivo externo se podrán obtener los datos necesarios para calcular tiempos de ejecución, seguridad, etc.

Para la realización del proyecto se ha decidido utilizar una placa Raspberry Pi 4 debido a que este modelo posee mayor memoria RAM que la versión anterior, así como mejor velocidad de procesador. A su vez, se ha decidido utilizar Ubuntu Mate como sistema operativo para dicho PC, debido a su menor consumo de recursos que la versión Ubuntu estándar. Por último, se utilizará ROS como sistema de desarrollo, debido a lo ampliamente utilizado y estandarizado que se encuentra para desarrollos de software robótico y por ser código abierto (BSD) y libre.

Palabras clave: Planificadores de rutas, Algoritmos de búsqueda.

Abstract

Nowadays, there is a growing implementation of mobile robots in indoor spaces, both in domestic and industrial environments. Due to this growing demand, it is necessary to compare the existing navigation algorithms in order to decide the most appropriate in each situation. Therefore, this project aims to compare different indoor navigation methods. These comparisons will be carried out both in simulation and in real environments. For this purpose, a Pioneer 3-DX robot will be used, which will be equipped with the necessary hardware and software to be able to accomplish the tests. To do this, the robot will be equipped with a LiDar Hokuyo URG-04LX sensor and a low cost PC, such as Raspberry Pi, will be installed. From an external device it will be possible to obtain the necessary data to calculate execution times, safety, etc.

For the realization of the project it has been decided to use a Raspberry Pi 4 board because this model has more RAM than the previous version, as well as better processor speed. At the same time, it has been decided to use Ubuntu Mate as the operating system for this PC, due to its lower resource consumption than the standard Ubuntu version. Finally, ROS will be used as the development system, due to its wide use and standardization for robotic software development and because it is open source (BSD) and free.

Keywords: Path Planning, Search Algorithms.

Índice general

Resumen vii

Abstract ix

Índice general xi

Índice de figuras xiii

Índice de tablas xv

1 Introducción 1

1.1 Presentación . . . 1

1.2 Algoritmos a comparar . . . 1

2 Estudio teórico 3

2.1 Introducción . . . 3

2.2 Estado del Arte . . . 3

2.2.1 Métodos determinísticos . . . 4

2.2.1.1 Dijkstra . . . 4

2.2.1.2 A* . . . 5

2.2.2 Métodos probabilísticos . . . 6

2.2.2.1 RRT . . . 6

2.2.3 Otros algoritmos . . . 7

2.2.3.1 Best First . . . 7

2.2.3.2 Breadth-First . . . 8

2.2.3.3 Bidirectional A* . . . 8

2.2.3.4 Anytime Repairing A* . . . 9

2.3 Comparativa teórica . . . 11

2.4 Conclusiones . . . 12

3 Desarrollo **13**

3.1 Introducción 13

3.2 Desarrollo del sistema de experimentación 13

3.2.1 Preparación del entorno 13

3.2.2 Realización de pruebas 19

4 Resultados **21**

4.1 Introducción 21

4.2 Eficiencia temporal 28

4.3 Consumo de memoria 30

4.4 Distancia recorrida y cambios de dirección 32

4.5 Aplicación de los algoritmos 35

5 Conclusiones y líneas futuras **39**

5.1 Conclusiones 39

5.2 Líneas futuras 40

6 Presupuesto **41**

6.1 Hardware 41

6.2 Software 41

6.3 Personal 41

6.4 Presupuesto total 41

Bibliografía **43**

Apéndice A Herramientas y recursos **45**

Apéndice B Funciones **47**

B.1 ImageToArray 47

B.2 RecommendedNodeSize 48

B.3 Resize 49

Índice de figuras

2.1 Grafo de ejemplo para *Dijkstra* 4

2.2 Resolución del algoritmo 4

2.3 Grafo de ejemplo para *A** 5

2.4 Expansión de *RRT* [1] 6

2.5 Generación de un camino subóptimo con *RRT* 7

2.6 Caso favorable y desfavorable para el uso de *Best First* 7

2.7 Funcionamiento de *BFS* 8

2.8 Caso desfavorable para *BiA** 9

2.9 Caso favorable para *BiA** 9

2.10 Comparativa *ARA** $\epsilon = 0.9$, 1.3, 1.7 y 2.1 10

3.1 Posibles errores al escoger un tamaño de celda inadecuado 14

3.2 Ejemplo de segmentación 15

3.3 Comparativa nodos necesarios para desplazamientos verticales 15

3.4 Comparativa nodos necesarios para desplazamientos diagonales 16

3.5 Relación anchura máxima transitable / tamaño de cada casilla 16

3.6 Obtención de tamaño de las celdas 17

3.7 Comparativa «resize» de Matlab vs función propia, respectivamente 17

3.8 Carga de límites 18

3.9 Carga de obstáculos 18

3.10 Error / Solución límites 18

3.11 Ejemplo de cálculo de longitud y giros 19

4.1 Recorridos realizados en el entorno 1 23

4.2 Recorridos realizados en el entorno 2 24

4.3 Recorridos realizados en el entorno 3 25

4.4 Recorridos realizados en el entorno 4 26

4.5 Recorridos realizados en el entorno 5 27

4.6 Porcentajes relación nodos visitados / totales 31

4.7 Recorrido 1 en simulación 35
4.8 Recorrido 3 con camino sinuoso 36
4.9 Reducción de puntos del path 36
4.10 Recorrido 1 (Pioneer 3-DX) 37
4.11 Recorrido 2 (Pioneer 3-DX) 37
4.12 Recorrido 3 (Pioneer 3-DX) 37
4.13 Recorridos Robot Pioneer 3-DX 38

Índice de tablas

4.1 Comparativa algoritmos - Eficiencia temporal 28
4.2 Comparativa algoritmos - Consumo de memoria 30
4.3 Comparativa algoritmos - Comparativa en distancia 32
4.4 Comparativa algoritmos - Cantidad de cambios de dirección 34

6.1 Recursos hardware utilizados 41
6.2 Recursos software utilizados 41
6.3 Personal requerido 41
6.4 Presupuesto total 41

Capítulo 1

Introducción

¿Sabes cuál es el problema?
Imaginarte el algoritmo y no programarlo.

Paul Huenca

1.1 Presentación

Este proyecto se realiza como Trabajo Final de Grado para la obtención del título de Grado en Ingeniería de Computadores.

Con el paso del tiempo ha habido un gran crecimiento del uso de los robots móviles. Este trabajo busca ofrecer una comparativa entre los mejores algoritmos para desplazar dichos robots por entornos interiores. Para ello utilizaremos un robot Pioneer 3-DX, así como un simulador para la realización de las pruebas.

En las comparativas buscaremos obtener los mejores resultados basándonos en los tiempos computacionales, las distancias recorridas o la memoria utilizada.

1.2 Algoritmos a comparar

Para comenzar el estudio debemos mencionar primero cuáles son los algoritmos más utilizados en navegación actualmente.

En primer lugar debemos mencionar la existencia de dos clasificaciones distintas según la forma de realizar las búsquedas, si bien añadiremos una tercera clasificación para aquellos algoritmos que no se adapten a ninguna de las dos definiciones propuestas. La distribución se realizará de la siguiente forma:

- Los métodos determinísticos o exactos
- Los algoritmos probabilísticos o aleatorios
- Otros algoritmos

Los métodos probabilísticos son algoritmos complejos computacionalmente, utilizados para localización y mapeo simultáneo, es decir, cuando debemos desplazarnos por un espacio, pero desconocemos el

estado inicial del sistema y por tanto debemos establecer en qué punto del espacio se encuentra y trazar trayectorias para alcanzar el destino. [2]

Los métodos determinísticos son menos complejos computacionalmente y obtienen resultados exactos y no sujetos a ciertas probabilidades de error, siempre y cuando el sistema tenga la información necesaria para poder implementarlos, y este camino sea posible. [2]

Los algoritmos que estudiaremos en este proyecto son:

- Algoritmo A*
- Algoritmo Dijkstra
- Algoritmo RRT
- Algoritmo Best-First
- Algoritmo BFS
- Algoritmo Bidirectional A*
- Algoritmo Anytime Repairing A*

Hemos escogido los algoritmos *A** y *Dijkstra* porque son los más utilizados en el ámbito del *pathfinding* debido a su corto tiempo de ejecución, además de que siempre alcanzarán su destino. Los demás algoritmos, a excepción del *RRT* (el cual ha sido escogido para estudiar algún algoritmo probabilístico en la comparativa y siendo este el más utilizado en *pathfinding*), son variantes de *A** y *Dijkstra*, las cuales mejoran en algunos aspectos a sus predecesores como veremos más adelante.

Capítulo 2

Estudio teórico

Regla número uno de todo programador:
Si el código funciona, no lo toques.

2.1 Introducción

Durante los últimos años se han realizado numerosas investigaciones con el objetivo de optimizar los algoritmos anteriormente mencionados aplicando diferentes heurísticas [3].

En este capítulo se hablará sobre las investigaciones llevadas a cabo con respecto a los algoritmos *Dijkstra*, *A** (junto a algunas de sus variantes) y *RRT*.

2.2 Estado del Arte

Debido al gran aumento del uso de robots móviles en la actualidad, es necesario estudiar cuáles son los mejores métodos para desplazar estos robots de un punto a otro. Es por ello que a día de hoy existen numerosas técnicas para calcular los mejores caminos. Al conjunto de todas estas técnicas se le conoce como *pathfinding*.

Al realizar un estudio acerca de cuáles son los mejores algoritmos generadores de caminos, no solo tenemos que tener en cuenta cuál es el camino más corto, sino también otros valores como qué tan seguro es ese camino (la distancia con respecto a los obstáculos), cómo de rápido encuentre ese camino, o las propias capacidades de desplazamiento del robot.

Dentro de los algoritmos de *pathfinding*, podemos encontrar dos métodos de generadores: los de tipo determinístico o exacto, que encontrarán siempre el el mejor camino en cuanto a distancia (si existe) 2.2.1, y los de tipo probabilístico o aleatorio, los cuales no tienen intención de encontrar el mejor camino, sino construir un árbol de exploración que cubra todo el espacio libre de obstáculos 2.2.2. A raíz de la cantidad de estudios realizados sobre el campo del *pathfinding*, se han desarrollado variaciones de los algoritmos más utilizados con la intención de mejorar los ya existentes. De estos algoritmos hablaremos en la sección 2.2.3

2.2.1 Métodos determinísticos

2.2.1.1 Dijkstra

Comenzaremos explicando el algoritmo **Dijkstra** (también llamado algoritmo de caminos mínimos), concebido en 1956 y publicado en 1959. Este algoritmo realiza una comparativa de forma recursiva del coste que se produce desde un nodo inicial hasta el resto de nodos, obteniendo siempre el camino más corto hasta cada uno de ellos [4].

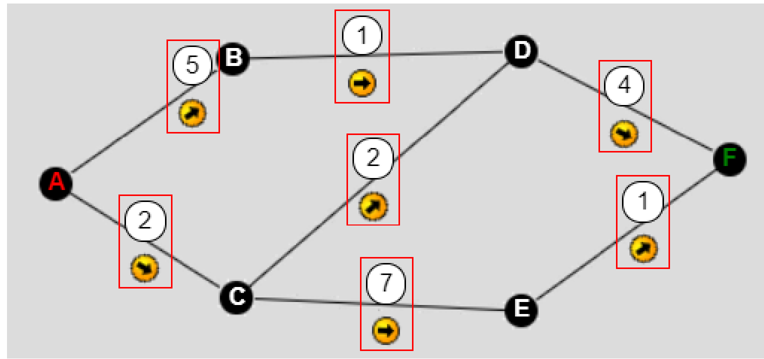


Figura 2.1: Grafo de ejemplo para *Dijkstra*

fin / ini	A	B	C	D	E	F
A	_	5	2	_	_	_
B	_	_	_	1	_	_
C	_	_	_	2	7	_
D	_	_	_	_	_	4
E	_	_	_	_	_	1
F	_	_	_	_	_	_

(valorada con pesos)
Matriz de Adyacencia

(a)

	Paso 1	Paso 2	Paso 3	Paso 4	Paso 5	Paso 6
A	(0,A)	*	*	*	*	*
B	(5,A)	(5,A)	(5,A)	(5,A)	*	*
C	(2,A)	(2,A)	*	*	*	*
D	∞	(4,C)	(4,C)	*	*	*
E	∞	(9,C)	(9,C)	(9,C)	(9,C)	(9,C)
F	∞	∞	(8,D)	(8,D)	(8,D)	

Matriz Analítica

(b)

Figura 2.2: Resolución del algoritmo

Debido a que *Dijsktra* calcula todos los posibles caminos hasta alcanzar el destino, añadir un nuevo nodo implicaría implementar una nueva fila y columna en las tablas de la figura [2.2], dando lugar a que la complejidad al usar este algoritmo sea exponencial.

En el caso concreto de la figura 2.1, el camino generado sería A-C-D-F, como podemos comprobar en la figura 2.2.

Dijkstra escanea todas las posibilidades para alcanzar el destino, lo que lo hace mucho más lento que otros algoritmos, pero posee una gran ventaja respecto a estos: en caso de querer alcanzar varios destinos, este método será más veloz ya que muchas zonas han sido escaneadas anteriormente.

El coste estimado del mejor camino se calcula aplicando la siguiente fórmula en cada iteración del bucle:

$$f(n) = g(n) \tag{2.1}$$

donde g(n) equivale al coste mínimo entre el nodo inicial y el nodo que queremos calcular en cada instante (n). Estos valores equivaldrían a los obtenidos en la figura 2.2.b

Para la generación del *path*, *Dijkstra* utiliza una cola de prioridad, la cual tiene una complejidad de tiempo O(log n) para las operaciones push y pop. La complejidad total del tiempo sería:

$$O(V + E(logV)) \tag{2.2}$$

donde V equivale al número de nodos totales y E, al número de aristas o relaciones totales en el grafo. [5]

2.2.1.2 A*

Otro de los algoritmos exactos que estudiaremos será el **A*** (conocido como *A Star*), publicado en 1968 [6]. Al igual que ocurría con *Dijkstra*, este algoritmo también nos devolverá el mejor camino si existe.

*A** es una versión ampliada de *Dijkstra* en la que se utiliza una función heurística [7] para la generación de los caminos. La adición de esta nueva variable da lugar a que la expansión de los posibles caminos para alcanzar el destino no se haga en todas direcciones como ocurría con *Dijkstra*, sino que se dará mayor prioridad a aquellos nodos que matemáticamente tengan menor coste.

El coste estimado aplicando el algoritmo *A** se calcula con la siguiente fórmula:

$$f(n) = g(n) + h(n) \tag{2.3}$$

donde g(n) equivale al coste mínimo entre el nodo inicial y el nodo que queremos calcular en cada instante (n), y h(n) es el coste mínimo entre el nodo seleccionado y el nodo final.

Figura 2.3: Grafo de ejemplo para *A**

En la figura 2.3 podemos ver el funcionamiento de *A**. El valor azul de la parte superior izquierda de cada celda equivale a g(n), mientras que el valor rojo de la parte superior derecha equivale a h(n). El valor negro central se corresponde por tanto al valor de f(n).

El algoritmo visitará de menor a mayor peso todos los nodos hasta encontrar el final. Los nodos anaranjados, verdes y azules son aquellos que el algoritmo ha visitado. Los verdes indican uno de los

posibles caminos que puede tomar el algoritmo, pudiendo seleccionar otro camino entre los azulados en función de la implementación del usuario. En ningún caso el algoritmo pasará por las casillas anaranjadas.

La complejidad de este algoritmo puede ser calculada aplicando la fórmula:

$$O(|V| + |E|) \tag{2.4}$$

siendo en este caso V los nodos **visitados** y no los totales, y E las aristas **utilizadas** y no las totales.

Otra forma de poder calcular la complejidad de este algoritmo sería aplicando la fórmula:

$$O(b^d) \tag{2.5}$$

El valor de b se corresponde con la ramificación del árbol, mientras que d se corresponde con la profundidad en la que se encuentra el nodo objetivo.

2.2.2 Métodos probabilísticos

2.2.2.1 RRT

En esta sección hablaremos acerca del algoritmo **RRT** (*Rapidly exploring random tree*), desarrollado por Steven M. LaValle y James J. Kuffner Jr. [8]. Este algoritmo genera un árbol de manera incremental y aleatoria. Para ello, escoge un nodo al azar e intenta relacionarlo con el nodo más cercano a este. Si es posible, se añade un nuevo estado. La probabilidad de expandir un estado es proporcional al tamaño de su región de Voronoi [9]. De esta manera el algoritmo se expandirá hacia áreas menos concurridas.

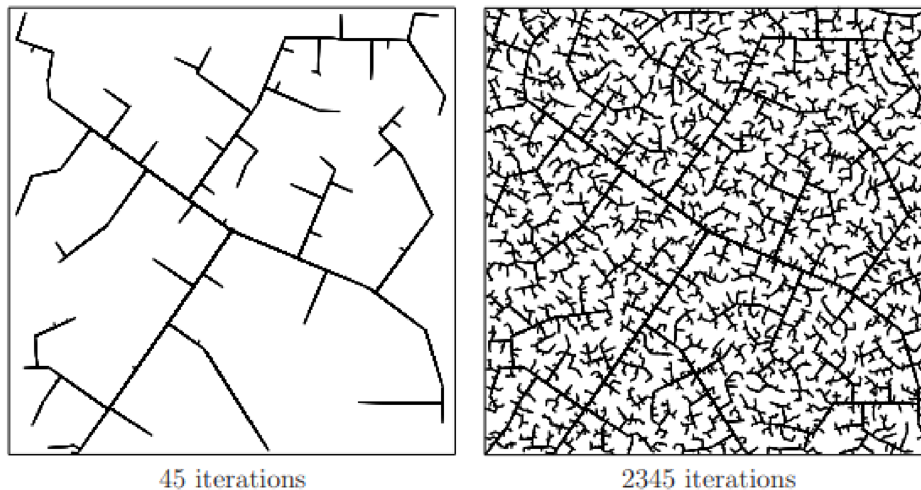


Figura 2.4: Expansión de *RRT* [1]

RRT tiene la ventaja de ser eficaz y ligero computacionalmente, pero su clara desventaja es que, al generar caminos aleatorios, estos nunca serán óptimos debido a la gran cantidad de giros, rotaciones sobre sí mismo y oscilaciones en el camino [fig. 2.5].

Para corregir el problema de la optimización se desarrolló *RRT** [10], el cual implementa la heurística. Con este nuevo algoritmo aseguramos el camino óptimo (en espacio), pero se vuelve asintóticamente óptimo temporalmente, es decir, el camino será el óptimo si el tiempo tendiese a infinito.

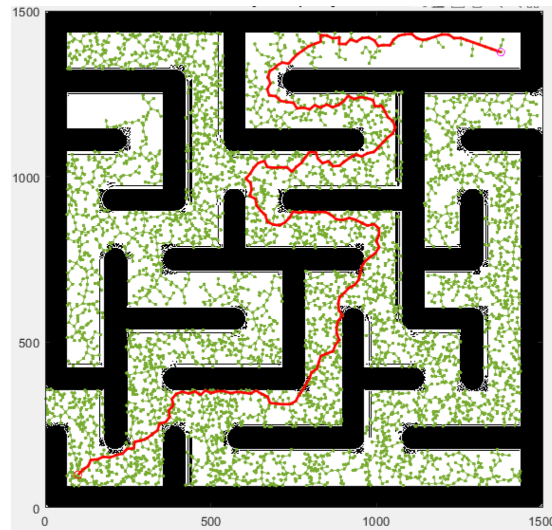


Figura 2.5: Generación de un camino subóptimo con *RRT*

2.2.3 Otros algoritmos

2.2.3.1 Best First

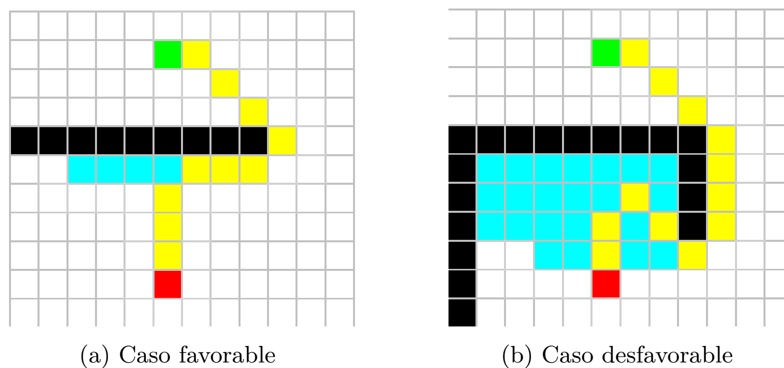
Dentro de esta sección empezaremos mencionando el algoritmo **Best First** [11]. Este algoritmo selecciona como siguiente nodo aquel que se encuentre más cerca del destino, ignorando todos los demás.

El coste estimado aplicando este algoritmo se calcula con la siguiente fórmula:

$$f(n) = h(n) \tag{2.6}$$

donde h(n) es el coste mínimo entre el nodo seleccionado y el nodo final.

Debido a que solo tiene en cuenta la heurística para calcular los costes, en los casos en los que la salida y el destino no tengan demasiados obstáculos, y/o estos sean fácilmente rodeables, este método sería el más eficaz. Sin embargo, esto no siempre es posible. En otros casos el algoritmo puede acceder a caminos sin salida, en cuyo caso empleará mucho tiempo y recursos en lograr salir.



(a) Caso favorable (b) Caso desfavorable

Figura 2.6: Caso favorable y desfavorable para el uso de *Best First*

Por todo esto, es preferible el uso de *A** en lugar de *Best First* ya que este primero, además de aplicar la heurística, también tiene en cuenta el camino que le precede, tal como vimos con la fórmula 2.3.

Por otra parte, la complejidad de este algoritmo se puede calcular mediante la siguiente fórmula:

$$O(V * log(V)) \tag{2.7}$$

siendo V el número de nodos totales.

2.2.3.2 Breadth-First

Otro algortimo que debemos mencionar es el **Breadth-First** (conocido como *BFS*), publicado por primera vez por Edward F. Moore en 1959 [12]. Este algoritmo recorrerá todo el grafo según su cercanía al nodo inicial, es decir, comprobará si los nodos a 1 de distancia se corresponden con el final. Si lo es, termina, pero si no lo es, comprobará todos los nodos a 2 de distancia y así sucesivamente.

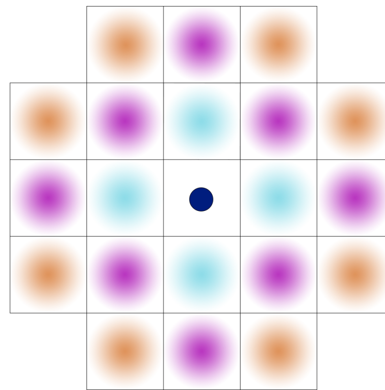


Figura 2.7: Funcionamiento de *BFS*

En el caso de la figura 2.7 el algoritmo comprobará si la salida se encuentra en el nivel 1 (sombra azul); si no se encuentra, comprobará el nivel 2 (sombra morada); continuará con el nivel 3 (sombra anaranjada) y así sucesivamente.

Se podría considerar similar a *Dijkstra* en el sentido de que ambos ignoran la heurística, pero en este caso, además, también ignora los pesos. De hecho, en un escenario en el cual todos los pesos sean iguales, *Dijkstra* y *BFS* serían el mismo algoritmo.

La complejidad temporal de este algoritmo puede ser calculada aplicando la fórmula:

$$O(V + E) \tag{2.8}$$

donde V equivale al número de nodos totales y E, al número de aristas o relaciones totales en el grafo. [5]

2.2.3.3 Bidirectional A*

Otro algoritmo que vamos a incluir en esta selección va a ser el **Bidirectional A*** (conocido como *BiA**). Este algoritmo es una variante directa de *A** siendo su diferencia que en este caso la búsqueda del camino se realizará en ambos sentidos (origen ↔ destino). [13]

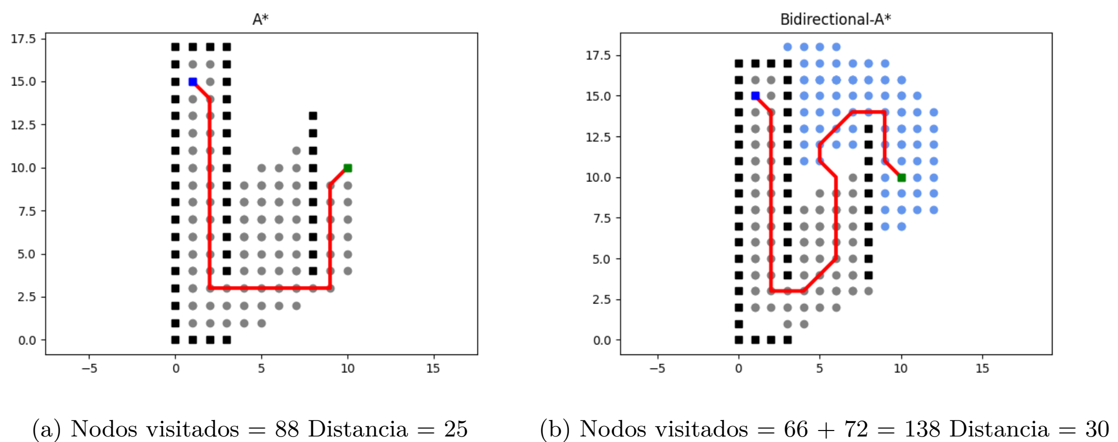
La complejidad de este algoritmo estaría repartida en dos partes equivalentes:

$$O(b^{d/2}) \tag{2.9}$$

siendo que la suma de ambos tiempos es mucho menor al de *A** [fig. 2.5].

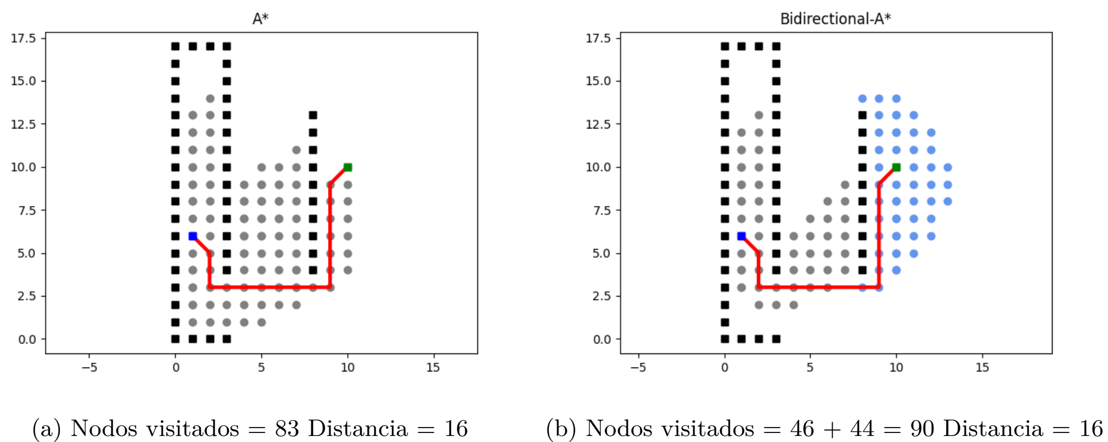
Si bien la complejidad es mucho menor en *BiA** que en *A**, el camino propuesto no es siempre el mejor. Este camino generado dependerá del escenario en el que se encuentre: si existe un buen punto medio, el resultado será optimo, pero si ambos caminos se ramifican en direcciones opuestas, el camino será más costoso que el encontrado por *A**.

Un ejemplo visual para comparar los casos favorables y desfavorables propuestos lo podemos observar en las figuras 2.8 y 2.9. Aunque el caso favorable sigue visitando más nodos que el *A** original, el tiempo empleado para obtener el camino ha sido la mitad.



(a) Nodos visitados = 88 Distancia = 25 (b) Nodos visitados = 66 + 72 = 138 Distancia = 30

Figura 2.8: Caso desfavorable para *BiA**



(a) Nodos visitados = 83 Distancia = 16 (b) Nodos visitados = 46 + 44 = 90 Distancia = 16

Figura 2.9: Caso favorable para *BiA**

2.2.3.4 Anytime Repairing A*

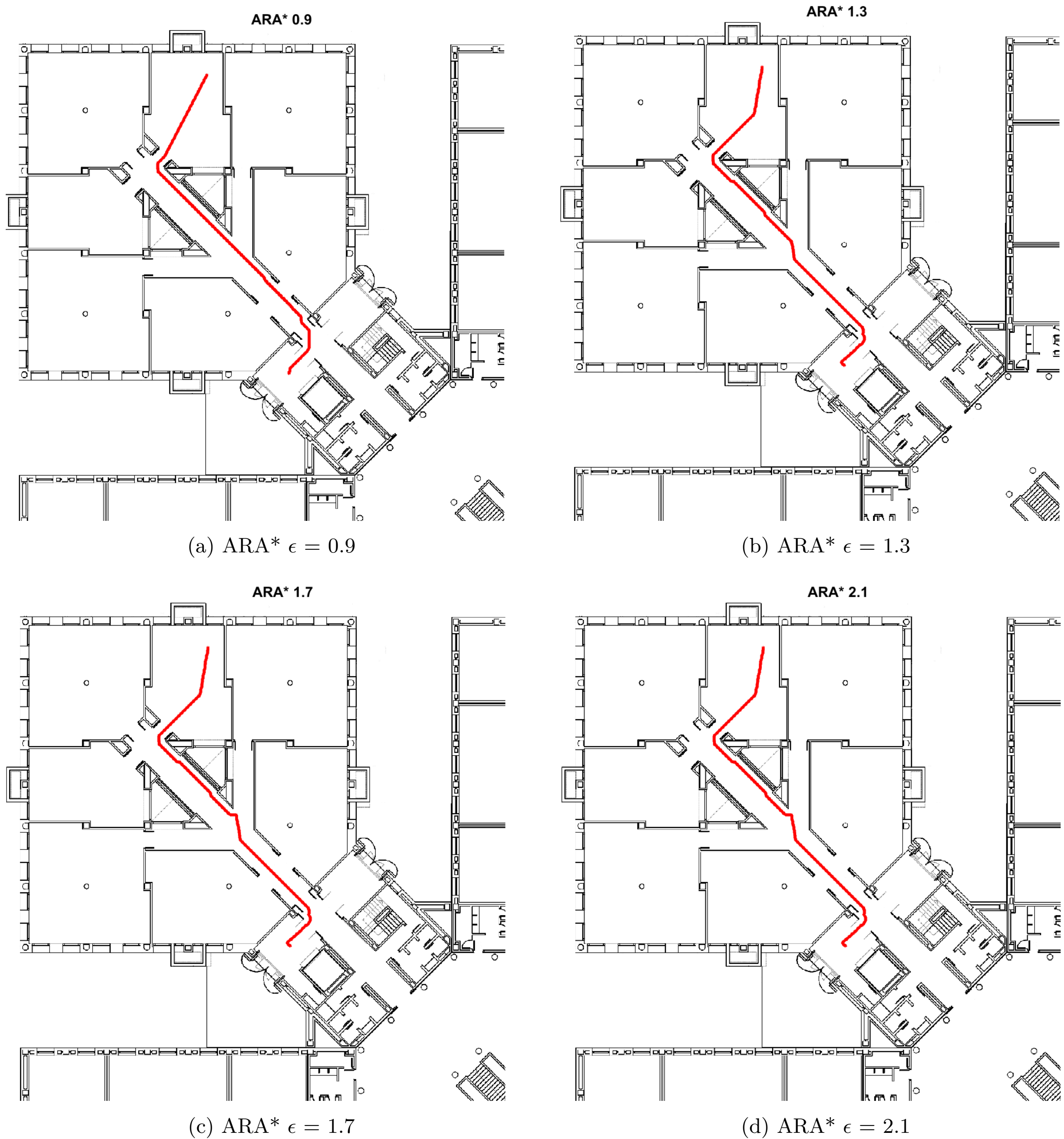
El último algoritmo con el que vamos a trabajar es una variante de *A** conocida como **ARA*** (*Anytime Repairing A**) [14]. Este algoritmo aplica la misma lógica que su antecesor, el A*. La diferencia con este radica en que la heurística está inflada, es decir, está multiplicada por un valor superior a 1. Con esta

multiplicación se obtienen varias rutas, las cuales pueden ser subóptimas en espacio pero resultan ser más rápidas en búsqueda. Para hallar cuál es el multiplicador ideal se deben realizar varias pruebas e ir aproximando el resultado. Un ejemplo de esto podemos observarlo en la figura 2.10.

La nueva fórmula para estimar los costes sería:

$$f(n) = g(n) + \epsilon * h(n) \tag{2.10}$$

donde g(n) equivale al coste mínimo entre el nodo inicial y el nodo que queremos calcular en cada instante (n); h(n) es el coste mínimo entre el nodo seleccionado y el nodo final y ϵ es un valor arbitrario que multiplica a la heurística.



(a) ARA* $\epsilon = 0.9$

(b) ARA* $\epsilon = 1.3$

(c) ARA* $\epsilon = 1.7$

(d) ARA* $\epsilon = 2.1$

Figura 2.10: Comparativa $ARA*$ $\epsilon = 0.9$, 1.3, 1.7 y 2.1

2.3 Comparativa teórica

	Clasificación	Ventajas	Desventajas
Dijkstra	Determinístico	Siempre alcanzará el camino óptimo si es posible ——— Tiene una complejidad lineal en el tiempo	Sigue un enfoque ciego avanzando en muchas ocasiones en la dirección opuesta al objetivo
*A**	Determinístico	Siempre alcanzará el camino óptimo si es posible ——— Implementa la heurística optimizando los tiempos de búsqueda	Tiempo de cómputo mejorable
RRT[1]	Probabilístico	Debido a que en cada paso solo es necesario trabajar con un nodo, es relativamente más rápido que otros algoritmos	Utilizar diagramas de Voronoi implica una alta complejidad ——— Debido a la aleatoriedad de la generación, la trayectoria podría ser en zigzag
Best-First	Otros	En casos con pocos obstáculos o fácilmente rodeables es el algoritmo más rápido	Al no tener en cuenta el nodo de origen no siempre generará la ruta óptima
BFS	Otros	Siempre encuentra la solución si existe	Muy complejo tanto en distancia como en tiempo
*BiA**	Otros	Más rápido temporalmente que A*	Estadísticamente es poco probable que genere el camino óptimo
*ARA**	Otros	Puede ser más rápido que A*	La solución obtenida variará en función del valor de épsilon inicial escogido

[1] Debido a la aleatoriedad de este algoritmo nos es imposible realizar una comparativa justa con respecto al resto de algoritmos, por lo que hemos descartado su participación en posteriores comparativas.

2.4 Conclusiones

En este capítulo se ha realizado una comparativa teórica acerca de algunos de los métodos de *pathfinding* más utilizados actualmente. En esta comparativa tan solo se ha tenido en cuenta cuál sería el mejor camino para alcanzar un punto desde otro, por lo que, si vamos a aplicar estos conocimientos a vehículos autónomos, deberíamos tener en cuenta otros factores de seguridad.

A su vez, debido a la gran cantidad de algoritmos de *pathfinding* existentes a día de hoy, nos es imposible realizar una comparativa que sea equitativa en su dificultad para generar el camino para todos ellos. Si en un grafo "X" se reunen las condiciones necesarias para que un algoritmo sea más eficiente, otro algoritmo generará un peor camino o tardará más tiempo en encontrarlo, siendo que en otro grafo "Y" este segundo algoritmo puede ser más eficiente que el primero.

Es por ello que en el siguiente capítulo pondremos a prueba la teoría mencionada en este, pudiendo comprobar en qué escenarios se desenvuelve mejor un algoritmo en comparación a los demás, además de comprobar otros valores importantes para la navegación de robots, debido a que no siempre el camino más rápido es el más útil.

Capítulo 3

Desarrollo

Primero resuelve el problema, después escribes el código.

John Johnson

3.1 Introducción

En el capítulo anterior se han mencionado los algoritmos de *pathfinding* más utilizados en la actualidad así como algunas variantes que mejoran dichos algoritmos en casos más concretos.

En este capítulo se pretende aplicar la teoría mencionada a código programable para poder comprobar la veracidad de la eficiencia de cada uno de los algoritmos, además de poder visualizar los resultados y comprobar otros valores independientes a la eficacia del mismo, tales como la seguridad del camino.

Para la comprobación del funcionamiento de cada uno de los algoritmos en mapas simulados aplicaremos distintos programas desarrollados en Python. A su vez utilizaremos Matlab para poder convertir mapas reales en grafos en los que realizar pruebas, así como visualizar los caminos generados. Por último, simularemos los recorridos obtenidos mediante el uso de ROS y, más adelante, realizaremos estos mismos recorridos con un robot Pioneer 3-DX en entornos reales.

3.2 Desarrollo del sistema de experimentación

Para llevar a cabo nuestra comparativa, se han buscado y probado diferentes proyectos dedicados al *pathfinding* con el objetivo de poder comparar las características de cada uno de estos algoritmos. Tras realizar pruebas con cada uno de los proyectos, decidimos utilizar el desarrollado por "zhm-real"[15] debido a que contiene todos los algoritmos estudiados en este trabajo además de otras variaciones de los mismos.

3.2.1 Preparación del entorno

Para poder realizar la comparativa hemos decidido descomponer el mapa en celdas fijas. Estas celdas se corresponden con los nodos que los diferentes algoritmos deben visitar para encontrar el camino. Para esta discretización se ha creado un script en Matlab [B.1], el cual nos permite convertir una imagen de un plano en una matriz que utilizaremos para la generación de nuestra zona de trabajo. En este script se ha decidido implementar algunas opciones necesarias para la correcta generación del entorno así como para el futuro manejo del robot Pioneer en este terreno.

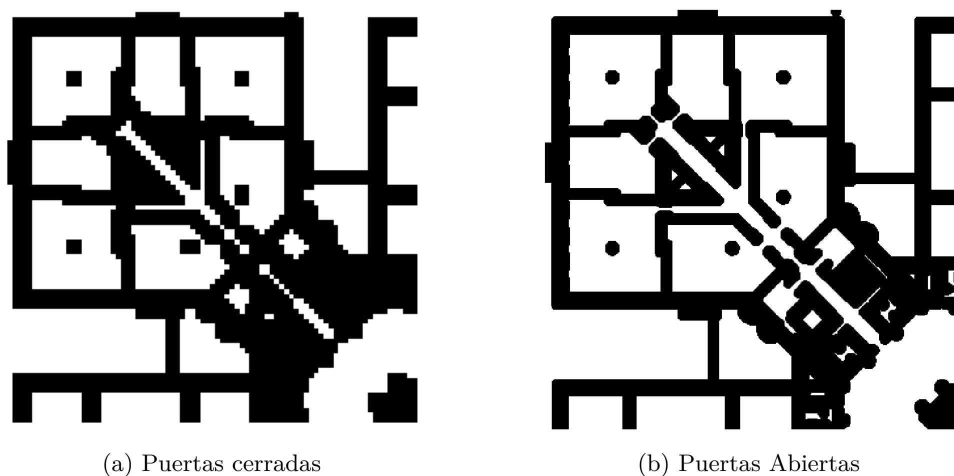
Para la generación del mapa se ha tenido en cuenta en primer lugar el tamaño del robot Pioneer. Algunos algoritmos tienden a ir muy cercanos a las paredes y, si bien en una imagen el recorrido es válido, en la realidad el robot iría chocando con la pared al no medir lo mismo que una celda.

También se ha tenido en cuenta la distancia de seguridad, que podremos variar en función del espacio en el que vayamos a desplazarnos.

Estos dos primeros valores nos permitirán adaptar el entorno de tal manera que el algoritmo (y futuro robot) interprete que hay casillas ocupadas donde realmente no las hay y guarde esta distancia con respecto a los obstáculos reales.

Si bien el procedimiento anterior soluciona este primer problema, tenemos también que tener en cuenta que todos los pasillos o puertas de un tamaño inferior al diámetro del robot, sumado a la distancia de seguridad para ambas paredes, quedarán cerrados como podemos observar en el ejemplo 3.1.

Para solucionar esto debemos reducir la distancia de seguridad. Sin embargo, si no fuésemos a atravesar este tipo de puertas o pasillos, la distancia de seguridad mencionada podría ser válida.



(a) Puertas cerradas (b) Puertas Abiertas

Figura 3.1: Posibles errores al escoger un tamaño de celda inadecuado

Otro valor imprescindible en la generación del entorno es o bien el tamaño real del mapa (introducido en metros) o bien la resolución del mismo. Este valor, junto a un tamaño de celda que le introduciremos también al programa, generará un entorno con un menor número de casillas totales lo que daría lugar a una mayor eficiencia computacional y temporal de los algoritmos.

Cuando segmentamos el entorno debemos tener en cuenta que no siempre es posible realizar una división exacta de casillas del tamaño especificado. Para estos casos se ha decidido añadir tantas filas y/o columnas libres como fuesen necesarias para realizar la división de forma correcta. Al añadir casillas libres, el entorno no se ve afectado por estas nuevas casillas. Si bien con esto estamos añadiendo información extra al entorno, creemos más importante añadir esta información extra frente a perder varias casillas con valores importantes.

Un ejemplo de esto podemos observarlo en la figura 3.2, en la cual tenemos un entorno de 25x20 metros y queremos segmentarlo en celdas de 15x15 metros. Para no perder la información de los 10 metros derechos o de los 5 inferiores, hemos decidido agregar las filas y columnas extras necesarias para que la división sea exacta. Estas casillas serán consideradas libres para que durante el discretización del entorno (el cual explicaremos más adelante) no se altere su forma.

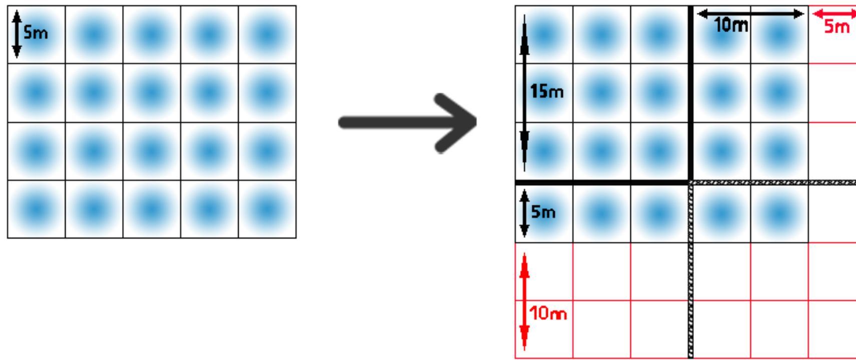


Figura 3.2: Ejemplo de segmentación

Para seleccionar un tamaño de celda que no cierre los caminos por los que queramos que se puedan desarrollar los algoritmos se ha creado una función en Matlab [B.2] en la cual, introduciéndole únicamente el tamaño del pasillo o puerta más pequeño (en centímetros) por el que queremos que el algoritmo pueda buscar, nos devuelva un rango de tamaños de celda válidos. El uso de esta función es opcional, pero hemos decidido utilizarla para reducir el número de nodos totales y aumentar de esta manera la eficiencia de todos los algoritmos.

De forma resumida, hemos aplicado la función:

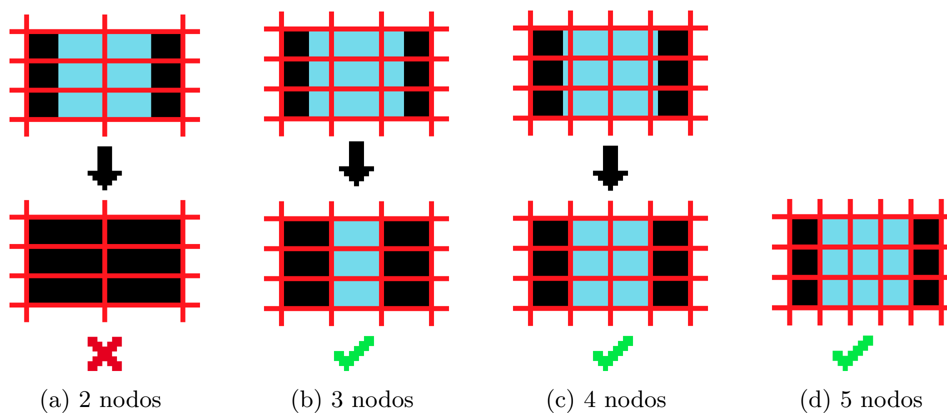
$$((((distSeguridad * 2 + radioRobot * 2) * 100)/tamCelda) + nodosExtra) * tamCelda \tag{3.1}$$

donde:

distSeguridad → Distancia de seguridad, en metros
radioRobot → Radio del robot, en metros
tamCelda → Tamaño de la celda, en centímetros
nodosExtra → Número de nodos extra necesarios para no cerrar caminos

En la fórmula anterior (3.1) hemos decidido que la variable *nodosExtra* sea 3 debido a que, en el peor de los casos, este es el valor mínimo de celdas adicionales necesarios para que el robot pueda atravesar cualquier pasillo o puerta, independientemente de su orientación en el entorno.

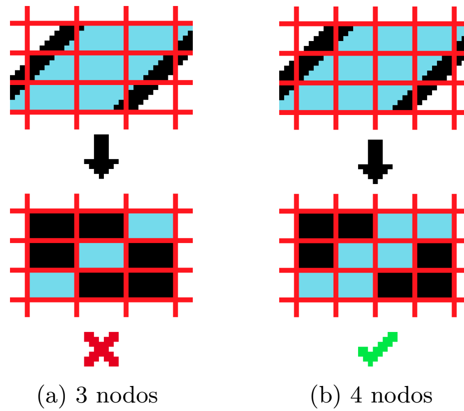
Para casos horizontales y verticales podemos comprobarlo de forma visual con el siguiente ejemplo:



(a) 2 nodos (b) 3 nodos (c) 4 nodos (d) 5 nodos

Figura 3.3: Comparativa nodos necesarios para desplazamientos verticales

Si bien podemos comprobar que para entornos verticales (mismo ejemplo para entornos horizontales) el mínimo de nodos es 3 (siendo uno de ellos el camino que debe seguir el robot y los otros dos extras, celdas ocupadas), para los casos en que los desplazamientos deban realizarse en diagonal sucede que los algoritmos no pueden desplazarse de esa manera directamente, sino que deben analizar el terreno de forma escalonada como podemos ver en el siguiente ejemplo:



(a) 3 nodos (b) 4 nodos

Figura 3.4: Comparativa nodos necesarios para desplazamientos diagonales

Para crear una función que englobe todos los casos posibles de desplazamiento se ha decidido que la variable *nodosExtra* sea 3 debido a que este es el valor mínimo de nodos extra válido para ambos casos.

Aplicando la función 3.1 obtenemos la siguiente gráfica (3.5), la cual nos relaciona el tamaño máximo por el que puede transitar el robot con el tamaño de las celdas que vamos a utilizar.

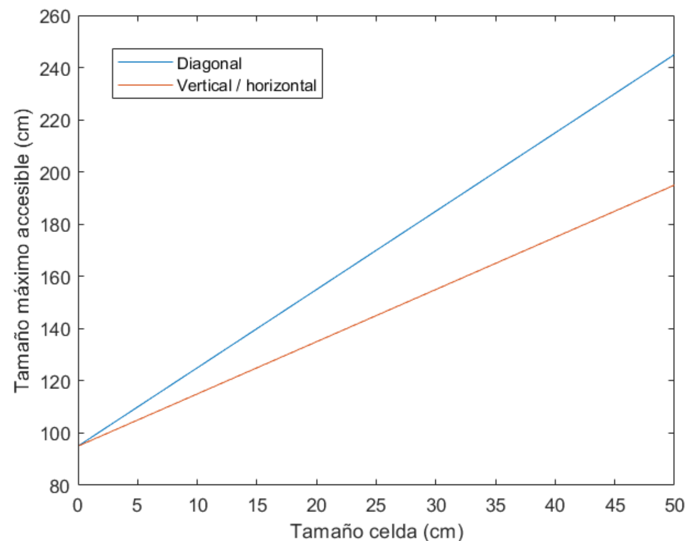


Figura 3.5: Relación anchura máxima transitable / tamaño de cada casilla

Como se ha mencionado anteriormente, para que el proyecto englobe todas las orientaciones para los caminos utilizaremos la función diagonal debido a que esta contiene a su vez la función vertical y horizontal. Como ejemplo, en el caso de querer comprobar qué tamaño de casilla debemos introducir para atravesar una puerta de 1.45m con nuestro robot, obtendríamos la siguiente salida por pantalla:

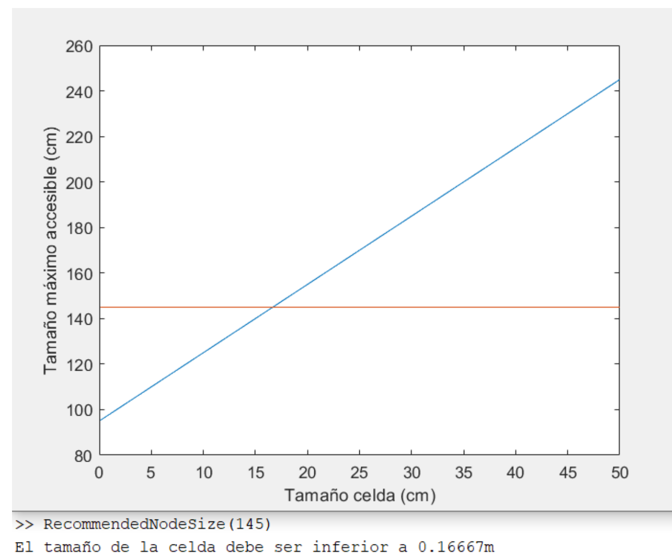


Figura 3.6: Obtención de tamaño de las celdas

Todos los valores por debajo de la línea horizontal son válidos y, cuanto más grande sea el valor de cada celda, más eficientemente funcionarán los algoritmos ya que el entorno contendría un menor número de casillas totales.

Por último, para la generación del entorno hemos visto necesario tratar la imagen del entorno de varias formas: en primer lugar convertimos la imagen a blanco y negro; a continuación llevamos a cabo la "ampliación" de las paredes; y por último, llevaremos a cabo el reescalado de la imagen de tal forma que el tamaño de las celdas coincida con el solicitado al comienzo. Si bien podríamos utilizar la función «resize» que nos ofrece Matlab, tras investigar acerca de su funcionamiento esta función podría hacer desaparecer paredes al hacer una media de los valores de las casillas vecinas y aproximar al más cercano. Es por ello que hemos creado una función propia [B.3], la cual comprueba las casillas vecinas y, si alguna de ellas es considerada "ocupada", la casilla resultante será ocupada independientemente de la cantidad de estas.

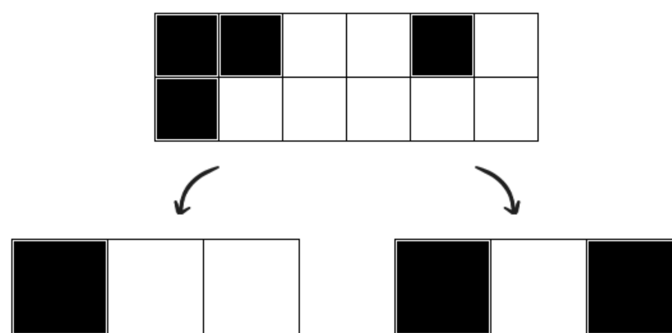


Figura 3.7: Comparativa «resize» de Matlab vs función propia, respectivamente

Tras introducir todos los datos solicitados generaremos un archivo que contendrá los datos necesarios para crear el entorno. Es importante que nos fijemos en que el tamaño de la celda solicitado y el devuelto no siempre coincidirán al completo. En caso de que la segmentación del entorno en celdas del tamaño solicitado no sea exacto, el valor del tamaño de las celdas será sustituido por otro tamaño que sí coincida, mostrando por pantalla el resultante así como las dimensiones del nuevo entorno.

Dentro de los archivos de Python, comenzaremos editando el llamado env.py, el cual es el encargado de generar un entorno útil para todos los algoritmos que vamos a utilizar. A su vez, será en este archivo

en el que podremos limitar los movimientos del robot, permitiéndole desplazarse en las cuatro direcciones cardinales o añadiéndole la posibilidad de ir en diagonal en los resultados (en la búsqueda, solo en los ejes cardinales).

Inicialmente deberemos introducirle los límites del grafo a través de las variables «x_range» e «y_range».

A continuación deberemos introducirle las coordenadas en las que encontraremos obstáculos, mediante el uso de «obs.add(CoordenadaX, CoordenadaY)».

Para la realización de este proyecto se ha decidido automatizar el proceso de configuración de los parámetros anteriormente mencionados mediante la carga de un archivo externo llamado «variables.txt»y «MapaEnTexto.txt» (fig. 3.8 y fig. 3.9 respectivamente). Estos valores, a su vez, serán editados mediante un archivo propio del que se hablará en el apartado 3.2.2.

```
### CARGA DE DATOS ###
lineX = linecache.getline(os.getcwd() + r'\Search_2D\variables.txt', 4)
self.x_range = int(lineX)
print("Tamano del eje X del mapa, Cargado")

lineY = linecache.getline(os.getcwd() + r'\Search_2D\variables.txt', 5)
self.y_range = int(lineY)
print("Tamano del eje Y del mapa, Cargado")
######################
```

Figura 3.8: Carga de límites

```
## Generación del grafo ##

with open(os.getcwd() + "\Search_2D\MapaEnTexto.txt") as f:
    for linea in f:
        coordenadas = linea.split(',')
        coordenadas = [i.replace('\n','') for i in coordenadas]
        x = coordenadas[0];
        y = coordenadas[1];
        obs.add((int(x), int(y)))

############
```

Figura 3.9: Carga de obstáculos

Otro de los problemas que debemos afrontar al trabajar con mapas es que, para aquellos que no tienen los límites establecidos, debemos generarlos nosotros aprovechando los valores de los rangos obtenidos de la carga inicial. Para ello, hemos delimitado el entorno de manera que los algoritmos no puedan extenderse más allá de la zona permitida. De esta forma evitaremos errores de búsqueda similares a los de la figura 3.10.

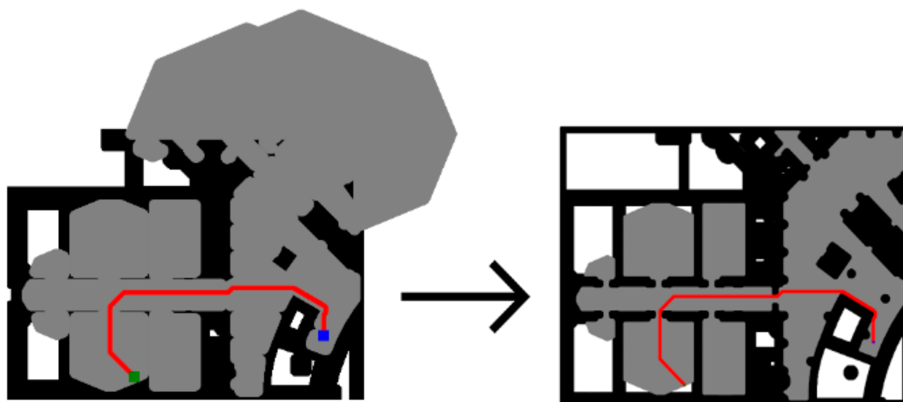


Figura 3.10: Error / Solución límites

3.2.2 Realización de pruebas

Para probar los distintos algoritmos aprovechando el desarrollo del proyecto llevado a cabo por "zhm-real" [15], deberemos comenzar introduciendo manualmente los valores de los nodos inicial y final. Con esto se nos mostraría por pantalla una animación en la que veríamos el recorrido que habríamos obtenido al aplicar un algoritmo u otro. Sin embargo, para la comparativa que estamos llevando a cabo, hemos visto necesario ampliar este proyecto con aportaciones propias que permitan comparar otros atributos tales como el tiempo de generación o el número de nodos visitados, ya que una imagen por sí sola no nos aporta demasiada información.

Por todo esto, hemos decidido modificar cada uno de los algoritmos. En todos ellos hemos insertado un temporizador al comienzo de la búsqueda de su camino, el cual finalizará cuando encuentre la ruta. A su vez hemos implementado una función que lleva la cuenta del número de nodos visitados, así como cuáles son estos nodos. Esto último es debido a que algunos algoritmos como *ARA** realizan múltiples búsquedas y en varias ocasiones estos nodos visitados son repetidos.

Para automatizar la realización de pruebas hemos generado un archivo llamado «Generador_de_pruebas», el cual cargará los datos necesarios para poder ejecutar todos los algoritmos de manera sucesiva (no en paralelo ya que afectaría a los tiempos) y permitirá que, en caso de que algún algoritmo no encontrase el resultado, se mostrase un mensaje de "Path not found" y continuase con el siguiente. Estos datos se corresponden con los nodos inicial y final, el nombre con el que queramos guardar la prueba realizada y los valores de altura y anchura obtenidos a su vez al generar el entorno con Matlab. Estos datos se almacenan en un archivo de texto llamado «variables.txt» que será leído por cada uno de los algoritmos.

Para finalizar la generación de los *path* de cada algoritmo hemos añadido un sistema de guardado de datos que toma los valores calculados (número de nodos del *path*, nodos visitados, tiempos...) y los almacena para una posterior comparación.

Para la conversión del número de nodos del *path* a medidas útiles (en nuestro caso, a metros) se ha creado una última función, la cual tiene en cuenta la posición entre cada nodo y su posterior (vertical, horizontal o diagonal) para indicarnos la longitud correcta del camino así como el número de cambios de dirección que realiza cada uno.

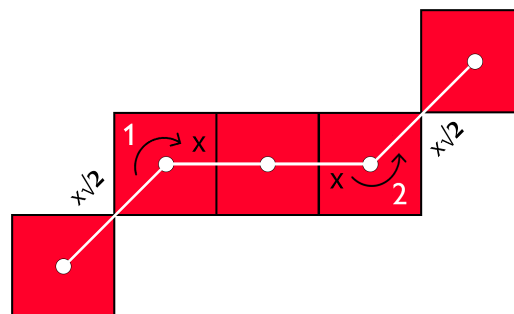


Figura 3.11: Ejemplo de cálculo de longitud y giros

Si utilizamos como ejemplo la figura 3.11 y siendo 'x' el tamaño de la celda con el que se generó el entorno, la longitud (l) recorrida por el algoritmo será:

$$l = x\sqrt{2} + x + x + x\sqrt{2} \text{ metros}$$
$$\textit{Número de giros realizados} = 2$$

Capítulo 4

Resultados

<p> Si se puede imaginar, se puede programar </p>

4.1 Introducción

En este capítulo se detallarán los resultados obtenidos al ejecutar todas las pruebas realizadas sobre los distintos planos simulados.

Hemos decidido que, para una mejor comparativa de los algoritmos, los entornos utilizados serán muy diversos en sus características. Algunos de ellos tendrán pasillos estrechos, mientras que otros dispondrán de gran espacio por el que moverse; otros estará mas estructurados mientras que otros serán más laberínticos. A su vez, se ha decidido realizar cada una de las simulaciones sobre los entornos con el mayor tamaño de celda válido para no cerrar caminos, ya que los tiempos de búsqueda y generación de los distintos *path*, así como la memoria consumida en el proceso, son inferiores en estos casos.

Todas estas pruebas se han ejecutado para cada uno de los algoritmos estudiados en el proyecto. Con los datos obtenidos, hemos decidido llevar a cabo tres comparativas al respecto:

- Eficiencia temporal
- Consumo de memoria
- Distancia recorrida y cambios de dirección

Para este proyecto hemos realizado 20 recorridos distintos repartidos en 5 entornos que veremos a continuación.

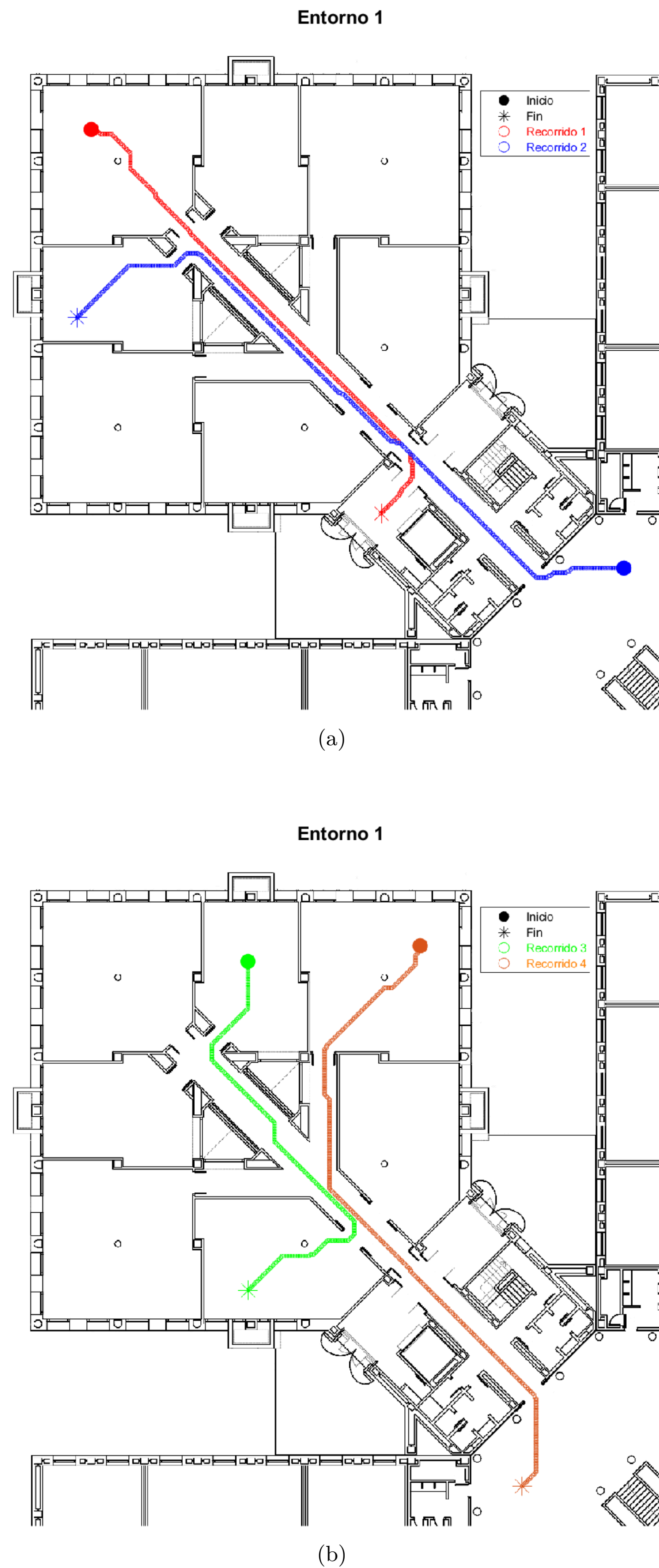
Los entornos 1 y 2 se corresponden con estancias de la zona oeste del edificio de la Escuela Politécnica Superior de la Universidad de Alcalá de Henares. La imagen del primer entorno que hemos tomado tiene unas medidas de 1669x1669 píxeles que se corresponden con 41,72x41,72 metros. La imagen utilizada para el segundo entorno tiene unas medidas de 1780x1339 píxeles correspondiéndose con 44,50x33,47 metros. Al tratarse de distintas partes de la misma zona, la escala para ambos es de 1:2,5 (px:cm). Hemos decidido escoger estos entornos debido a que se tratan de zonas con pasillos por los que podríamos probar los recorridos obtenidos con el robot real.

El entorno 4 por su parte se puede asimilar a una zona de oficinas con una mayor cantidad de caminos disponibles y con pasillos más estrechos que los entornos anteriores. La imagen de este entorno tiene

unas medidas de 1086x443 píxeles que se corresponden con 54,30x22,15 metros. Obtenemos por tanto una escala de 1:5 (px:cm)

Por último, los entornos 3 y 5 se corresponden con dos laberintos. Hemos decidido escoger estos entornos debido a la gran diversidad de pasillos que presentan, así como por los distintos tamaño de los mismos. El primero de ellos tiene unas medidas de 837x498 píxeles que se corresponden con 36x21,42 metros. Esto nos da una escala de 1:4,3 (px:cm), con pasillos de 3m de ancho. Por su parte, el entorno 5 tiene unas medidas de 403x403 píxeles que equivalen a 75x75 metros, dándonos una escala de 1:18,6 (px:cm), con pasillos de 1,47m de ancho.

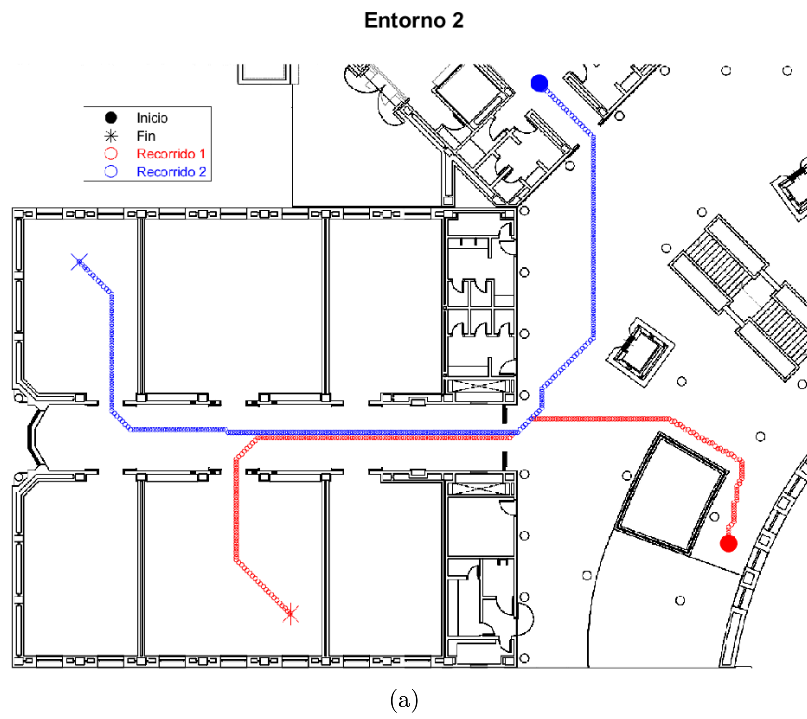
Los entornos utilizados, así como las pruebas realizadas en cada uno de ellos, son los siguientes:



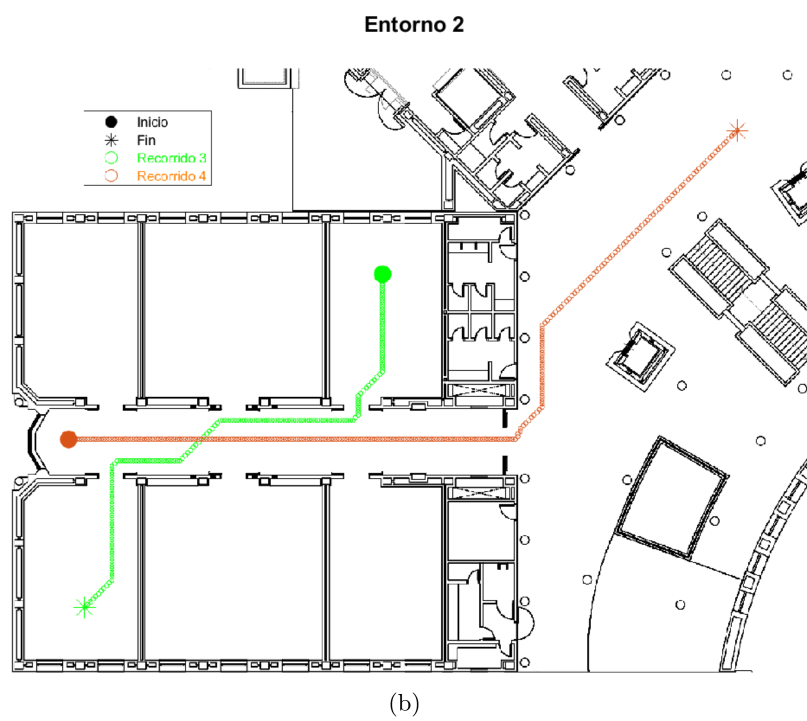
(a)

(b)

Figura 4.1: Recorridos realizados en el entorno 1

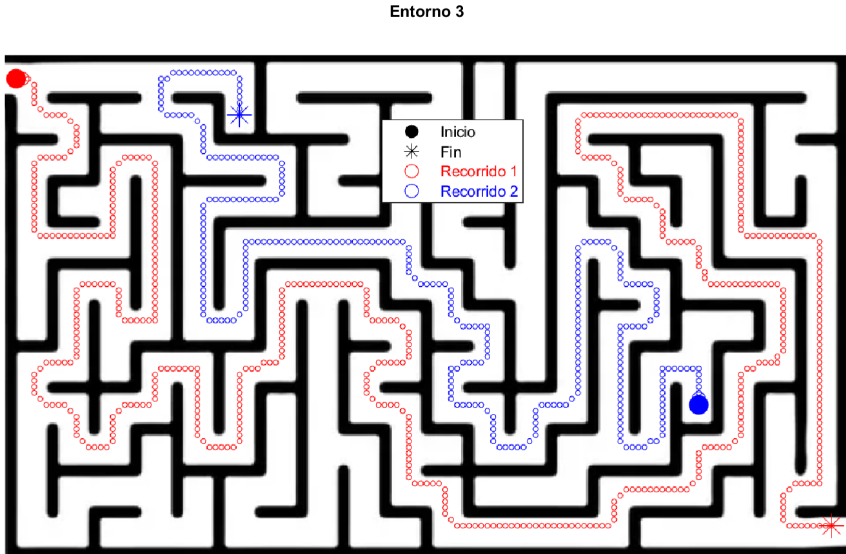


(a)

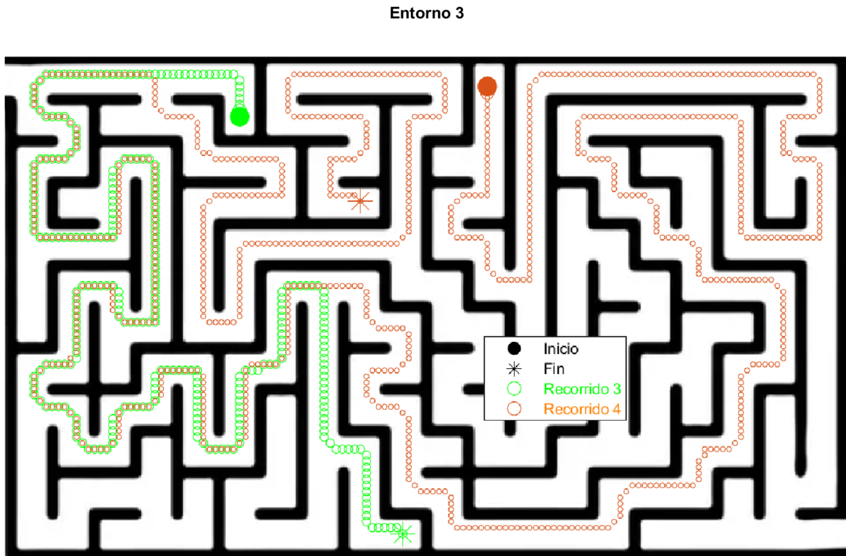


(b)

Figura 4.2: Recorridos realizados en el entorno 2

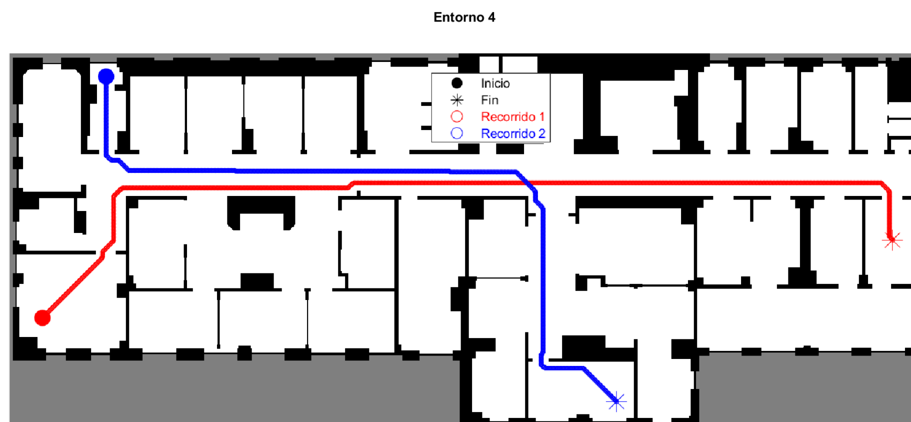


(a)

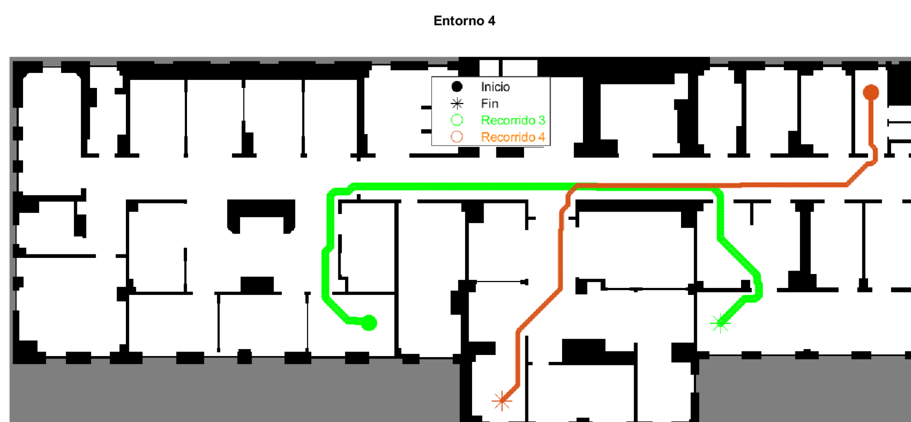


(b)

Figura 4.3: Recorridos realizados en el entorno 3

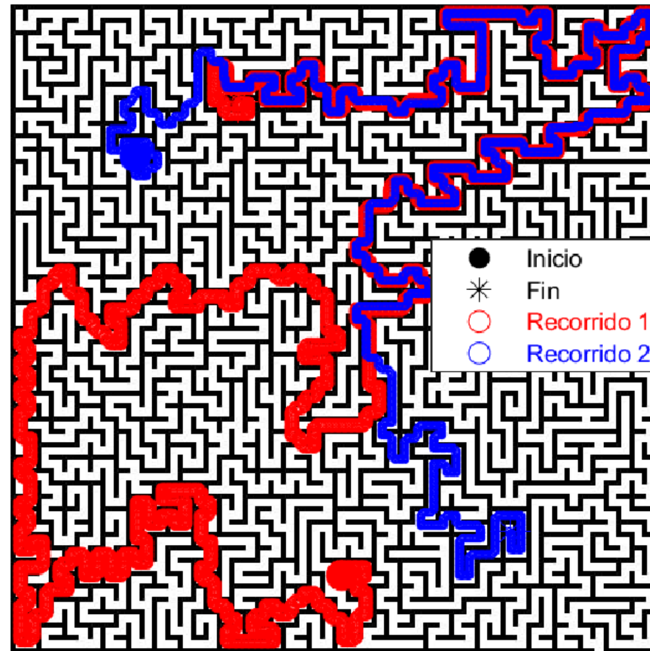


(a)

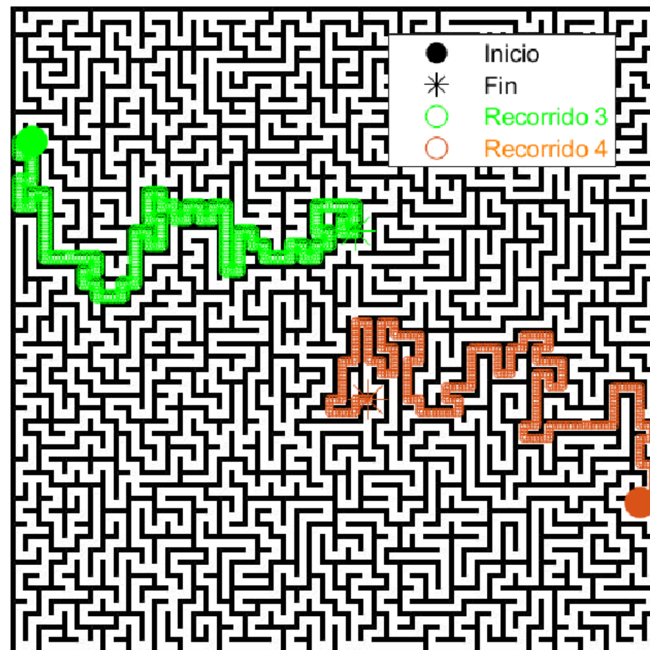


(b)

Figura 4.4: Recorridos realizados en el entorno 4



(a)



(b)

Figura 4.5: Recorridos realizados en el entorno 5

4.2 Eficiencia temporal

Los tiempos de generación del *path* que hemos obtenido son los siguientes:

	Entorno 1				Entorno 2				Entorno 3				Entorno 4				Entorno 5			
	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4
A*	0,419 ns	0,681 ns	0,544 ns	0,473 ns	0,256 ns	0,283 ns	0,218 ns	0,310 ns	0,063 ns	0,036 ns	0,044 ns	0,048 ns	2,432 ns	3,456 ns	2,975 ns	2,301 ns	1,902 ns	0,925 ns	0,091 ns	0,093 ns
ARA*	0,731 ns	1,760 ns	0,646 ns	0,725 ns	0,660 ns	0,790 ns	0,397 ns	0,795 ns	0,092 ns	0,028 ns	0,038 ns	0,045 ns	27,138 ns	45,096 ns	39,388 ns	12,184 ns	12,523 ns	5,173 ns	0,002 ns	0,002 ns
Best First	0,218 ns	0,215 ns	0,357 ns	0,374 ns	0,416 ns	0,539 ns	0,477 ns	1,281 ns	0,046 ns	0,033 ns	0,042 ns	0,072 ns	12,858 ns	16,640 ns	27,079 ns	5,749 ns	4,508 ns	1,223 ns	0,088 ns	0,101 ns
BFS	0,767 ns	1,110 ns	0,807 ns	1,295 ns	0,574 ns	0,632 ns	0,443 ns	0,718 ns	0,054 ns	0,046 ns	0,045 ns	0,050 ns	5,643 ns	5,452 ns	5,392 ns	4,632 ns	1,175 ns	1,070 ns	0,093 ns	0,118 ns
BiA*	0,286 ns	0,360 ns	0,460 ns	0,464 ns	0,200 ns	0,265 ns	0,174 ns	0,422 ns	0,053 ns	0,040 ns	0,047 ns	0,053 ns	1,446 ns	2,222 ns	1,534 ns	2,570 ns	1,802 ns	0,697 ns	0,093 ns	0,142 ns
Dijkstra	0,645 ns	0,870 ns	0,776 ns	0,935 ns	0,382 ns	0,434 ns	0,383 ns	0,440 ns	0,056 ns	0,040 ns	0,047 ns	0,056 ns	3,799 ns	3,380 ns	3,551 ns	2,617 ns	0,923 ns	0,780 ns	0,095 ns	0,100 ns

Tabla 4.1: Comparativa algoritmos - Eficiencia temporal

Si observamos los tiempos en los entornos estructurados, tanto de zonas de paso amplio (entornos 1 y 2), como de zonas de paso estrecho (entorno 4), el algoritmo *BiA** es el que nos ha ofrecido mejores tiempos seguido del *Best First* y el *A**. Este algoritmo es una variante directa de *A** que, para entornos cerrados no muy extensos, mejora a su predecesor. Por otra parte, si bien el algoritmo *Best First* es el más rápido en el entorno 1, estos resultados dependen en gran parte del número de "caminos prometedores" descartados entre el punto inicial y el final ya que el algoritmo visitará todos aquellos que se encuentren en línea recta entre la posición del robot y el nodo final en cada instante, por lo que no es un algoritmo fiable obteniendo muy malos resultados en los demás recorridos en comparación al resto.

Por el lado contrario, en estos entornos podemos observar que el algoritmo *ARA** ha llegado a tardar más de un 2000 % (en el peor de los casos) respecto al más rápido (*BiA**). Vemos también que el algoritmo *BFS* también es bastante lento en comparación a los demás (tardando más del 300 % respecto a *BiA**).

Los algoritmos que no se han mencionado (*BFS* y *Dijkstra*) son más lentos que *BiA**, llegando a tardar entre dos y tres veces más en la mayor parte de los casos.

Para los casos de los laberintos (entornos 3 y 5) podemos observar que los algoritmos *ARA** y *Best First* son muy dependientes del entorno en el que se ejecuten, siendo los más rápidos en algunos entornos y los más lentos en los demás. De nuevo, esto da lugar a que estos algoritmos no sean recomendables.

Si queremos asegurarnos un tiempo reducido en los mapas laberínticos es muy recomendable utilizar el algoritmo *A**.

Podemos concluir por tanto que, en base a las pruebas realizadas, si conocemos a priori las características del entorno por el que queramos desplazarnos, podemos escoger un algoritmo u otro. Por ello, para entornos estructurados (tanto con zonas de paso amplias como estrechas) recomendamos el uso del algoritmo *BiA**. Por el contrario, para entornos laberínticos recomendamos el uso de A*. Los algoritmos *ARA** y *Best First*, por su parte, pueden ofrecernos mejores tiempos que *A**, pero puede suceder que nos den tiempos mucho mayores que cualquier otro algoritmo y, por tanto, no los recomendamos.

4.3 Consumo de memoria

Los porcentajes obtenidos al comparar el número de nodos que ha visitado cada algoritmo con respecto al número de nodos visitables de cada entorno son los siguientes:

	Entorno 1				Entorno 2				Entorno 3				Entorno 4				Entorno 5			
	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4
A*	10,96 %	21,50 %	11,64 %	11,60 %	27,61 %	34,79 %	16,10 %	26,58 %	85,32 %	31,30 %	57,52 %	87,25 %	42,85 %	51,54 %	48,46 %	27,65 %	89,08 %	58,09 %	0,95 %	1,70 %
ARA*	13,86 %	28,57 %	13,72 %	14,73 %	31,39 %	43,50 %	20,51 %	41,53 %	86,10 %	31,73 %	58,78 %	87,64 %	62,71 %	61,88 %	55,36 %	33,11 %	89,21 %	61,76 %	0,96 %	1,74 %
Best First	0,28 %	0,44 %	3,71 %	1,53 %	8,97 %	13,13 %	7,62 %	12,79 %	37,46 %	19,70 %	48,82 %	66,08 %	17,36 %	14,91 %	23,85 %	6,06 %	61,51 %	21,75 %	0,56 %	1,11 %
BFS	30,92 %	49,34 %	31,42 %	51,82 %	60,48 %	73,42 %	57,28 %	70,03 %	89,83 %	32,59 %	62,32 %	90,69 %	98,17 %	81,26 %	90,16 %	52,15 %	90,29 %	72,03 %	1,04 %	2,15 %
BiA*	4,49 %	12,79 %	15,18 %	18,02 %	20,75 %	42,74 %	20,51 %	52,04 %	94,45 %	36,68 %	50,39 %	89,65 %	22,57 %	42,38 %	27,96 %	49,63 %	75,05 %	62,89 %	1,03 %	1,73 %
Dijkstra	35,83 %	51,83 %	35,90 %	51,60 %	63,46 %	73,37 %	59,60 %	71,99 %	89,83 %	32,34 %	63,79 %	90,69 %	97,07 %	82,95 %	90,98 %	55,01 %	90,18 %	70,86 %	1,04 %	2,41 %

Tabla 4.2: Comparativa algoritmos - Consumo de memoria

Con todos los porcentajes anteriores se ha decidido dividir la comparativa en entornos estructurados (entornos 1, 2 y 4) y entornos laberínticos (entornos 3 y 5), generándose la siguiente gráfica:

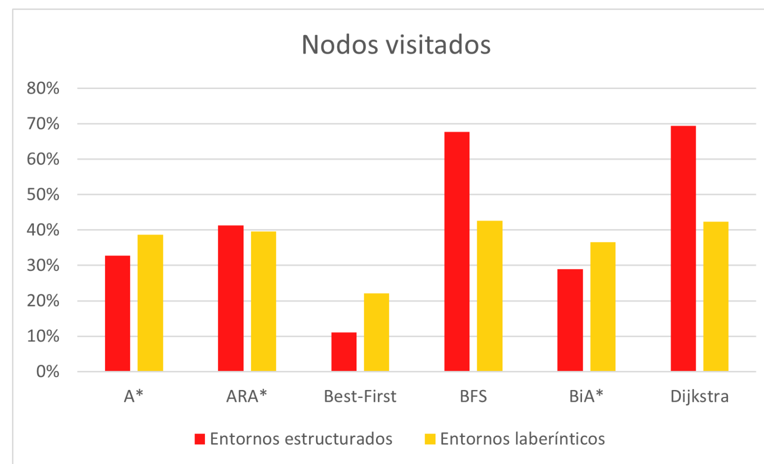


Figura 4.6: Porcentajes relación nodos visitados / totales

En primer lugar podemos observar que el algoritmo *Best First* es el método que menos nodos visita. Los demás algoritmos visitan en todos los casos, como mínimo, más del doble de nodos que este. Otras opciones a tener en cuenta son *A** y *BiA** para los entornos estructurados, mientras que, para entornos laberínticos, todos los algoritmos (en comparación con *Best First*) requieren de una mayor memoria de almacenaje de nodos. De nuevo, en caso de no utilizar el algoritmo *Best First*, es recomendable el uso de *A**.

Si bien todos los algoritmos (a excepción del *Best First*) superan el 35 % en entornos estructurados y el 65 % en los laberínticos, no siendo ninguno de ellos recomendable si requerimos de un menor uso de memoria, el algoritmo *Dijkstra* tiene una ventaja con respecto al resto, y es que almacena todas las distancias entre cada nodo y el resto de nodos visitados, lo cual significa que si quisiéramos realizar más búsquedas posteriormente, *Dijkstra* ya tendría analizados todos los nodos de las búsquedas anteriores.

Como conclusión a esta comparativa podemos deducir que, si disponemos de memoria limitada, deberemos escoger el algoritmo *Best First*. Si por el contrario, disponemos de una mayor memoria y vamos a realizar varias consultas, es recomendable el uso de *Dijkstra*.

4.4 Distancia recorrida y cambios de dirección

Las longitudes de los *paths* en metros desde el nodo inicial hasta el final obtenidas en cada prueba son las siguientes:

	Entorno 1				Entorno 2				Entorno 3				Entorno 4				Entorno 5			
	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4
A*	35,25 m	46,54 m	31,16 m	44,35 m	45,68 m	55,67 m	32,39 m	48,11 m	244,69 m	134,96 m	140,63 m	390,28 m	58,88 m	47,23 m	43,12 m	37,29 m	1109,43 m	731,41 m	255,76 m	307,52 m
ARA*	35,25 m	46,54 m	31,16 m	44,35 m	45,68 m	55,67 m	32,39 m	48,11 m	244,69 m	134,96 m	140,63 m	390,28 m	58,88 m	47,23 m	43,12 m	37,29 m	1109,43 m	731,41 m	255,76 m	307,52 m
Best First	36,13 m	49,13 m	32,54 m	49,37 m	50,24 m	56,74 m	33,41 m	50,40 m	246,02 m	135,68 m	140,99 m	390,28 m	75,51 m	59,16 m	45,46 m	55,57 m	1118,71 m	731,51 m	255,76 m	307,52 m
BFS	35,25 m	46,54 m	31,16 m	44,35 m	45,68 m	55,67 m	32,39 m	48,11 m	244,69 m	134,96 m	140,63 m	390,28 m	58,88 m	47,23 m	43,12 m	37,29 m	1109,43 m	731,41 m	255,76 m	307,52 m
BiA*	35,25 m	46,54 m	31,16 m	44,35 m	45,68 m	55,67 m	32,39 m	48,11 m	244,69 m	134,96 m	140,63 m	390,28 m	58,88 m	47,23 m	43,12 m	37,29 m	1109,43 m	732,03 m	255,76 m	307,52 m
Dijkstra	35,25 m	46,54 m	31,16 m	44,35 m	45,68 m	55,67 m	32,52 m	48,11 m	244,69 m	134,96 m	140,63 m	390,28 m	58,88 m	47,27 m	43,12 m	37,29 m	1109,75 m	731,41 m	255,76 m	307,52 m

Tabla 4.3: Comparativa algoritmos - Comparativa en distancia

De la tabla anterior podemos comprobar que todos los algoritmos recorren la misma longitud de camino a excepción del *Best First*, el cual ha llegado a recorrer hasta un 25 % más de distancia en algunos casos ya que es el único de los estudiados que genera un camino diferente a los demás por su lógica de búsqueda.

Además de comparar las distancias recorridas en metros para cada algoritmo, nos ha parecido importante contabilizar el número de veces que cambia de dirección cada uno de ellos para poder determinar cuál de todos realiza un recorrido mas suave, es decir, con menos giros. Hemos obtenido los resultados de la tabla 4.4.

De estos nuevos datos podemos observar que en los entornos estructurados (entornos 1, 2 y 4) el algoritmo que menos cambios de dirección realiza y, por tanto, genera caminos más suaves es el *Dijkstra*. En los entornos laberínticos (entornos 3 y 5) el algoritmo que parece funcionar mejor es el *BFS*, pero no podemos concluir que ninguno de los algoritmos sea mejor que el resto en este aspecto ya que este tipo de entornos genera un gran número de giros.

	Entorno 1				Entorno 2				Entorno 3				Entorno 4				Entorno 5			
	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 1	Rec. 2	Rec. 3	Rec. 4
A*	20	75	61	47	80	49	30	27	126	71	69	177	41	110	73	95	483	297	59	61
ARA*	21	86	68	48	78	70	29	32	124	71	68	177	41	135	84	62	490	296	59	61
Best First	19	29	23	37	61	42	22	41	127	72	66	174	111	94	71	70	497	304	59	61
BFS	28	35	44	52	50	50	40	26	118	65	68	173	42	75	76	103	608	391	59	61
BiA*	16	33	18	33	45	28	13	12	119	72	67	178	27	43	55	26	484	302	59	61
Dijkstra	20	26	14	18	40	11	10	8	124	68	67	176	19	28	38	28	500	314	59	61

Tabla 4.4: Comparativa algoritmos - Cantidad de cambios de dirección

4.5 Aplicación de los algoritmos

Para concluir con los resultados hemos decidido probar 3 recorridos independientes a los anteriores utilizando un robot Pioneer 3-DX. Las pruebas realizadas se han realizado en la zona oeste del edificio de la Escuela Politécnica de la Universidad de Alcalá de Henares. A su vez, hemos decidido que daremos prioridad a un recorrido más suave por lo que, aplicando las comparativas realizadas anteriormente, hemos escogido el algoritmo *Dijkstra*.

Los recorridos escogidos para las pruebas son:

- Recorrido 1 - Laboratorio OL3 a OL6
- Recorrido 2 - Laboratorio OL6 a OL3
- Recorrido 3 - Laboratorio OL6 a vestíbulo

En primer lugar y antes de realizar las pruebas con el robot, se llevaron a cabo estas mismas pruebas en el simulador STDR mediante el uso de ROS.

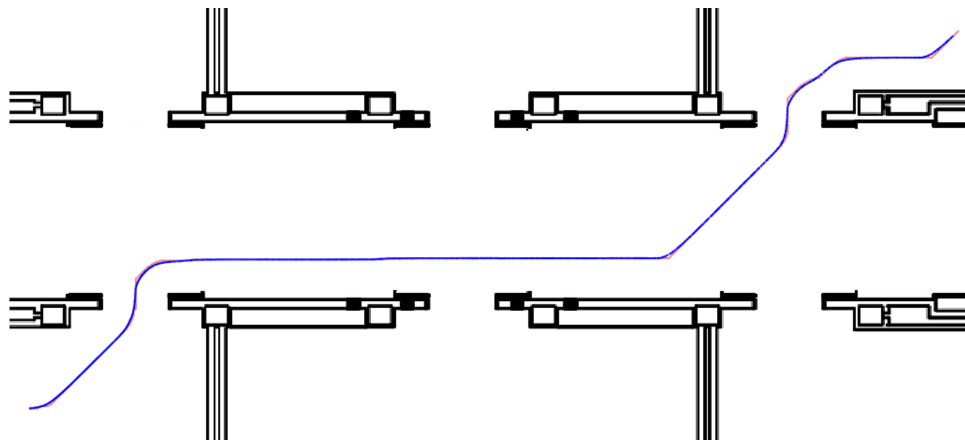


Figura 4.7: Recorrido 1 en simulación

En la figura 4.7 podemos observar que la línea azul correspondiente con el camino realizado por el robot simulado se superpone casi al completo a la linea anaranjada, correspondiente con el path generado al utilizar el algoritmo *Dijkstra*.

Si bien los recorridos realizados en el simulador son aceptables, al utilizar el robot real debemos tener en cuenta otros factores que no hemos aplicado en el simulador, tales como que los planos no coinciden con la realidad en todos los casos, que los ejes de coordenadas del robot, así como la posición inicial de este deben coincidir con el generado al aplicar el algoritmo, etc.

Tras realizar todas las modificaciones necesarias para que el path generado por *Dijkstra* se adapte al robot real, hemos podido observar que el recorrido realizado generaba un camino sinuoso como podemos observar en la figura 4.8.

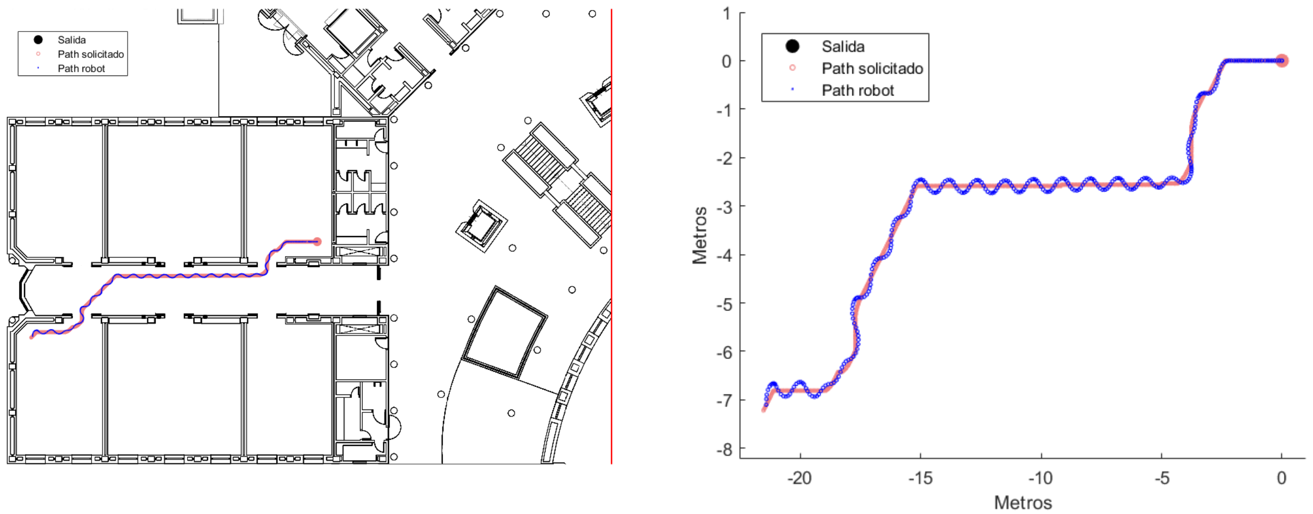


Figura 4.8: Recorrido 3 con camino sinuoso

Para solventar este problema, se ha decidido reducir la velocidad angular del robot, así como su velocidad lineal. A su vez, hemos decidido segmentar el path en zonas estrechas (en las cuales el robot deba seguir más fielmente el camino debido a posibles obstáculos) y zonas amplias (aquellas en las que el robot pueda moverse con mayor flexibilidad debido a la falta de obstáculos) para reducir el número de posiciones en el path y evitar posibles errores en el caso de que se saltase algún punto y debiese retroceder. En las zonas estrechas los puntos son muy cercanos entre sí para que siga el camino de forma más fiel. Por el contrario en las zonas amplias los puntos se encontrarán más distantes. Podemos observar este cambio en la figura 4.9.

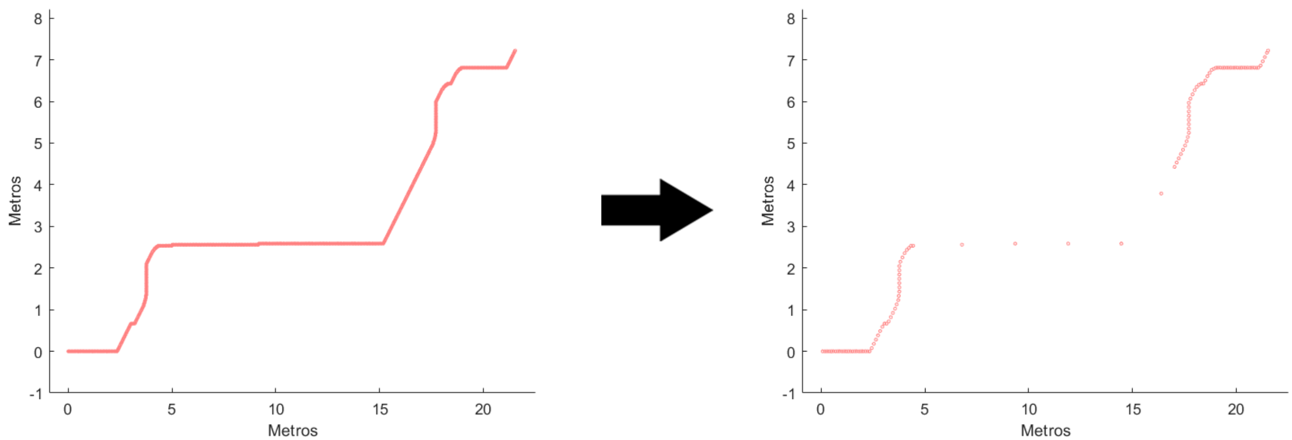


Figura 4.9: Reducción de puntos del path

Tras realizar todos los cambios mencionados anteriormente, hemos realizado los 3 recorridos propuestos, obteniendo los siguientes resultados:

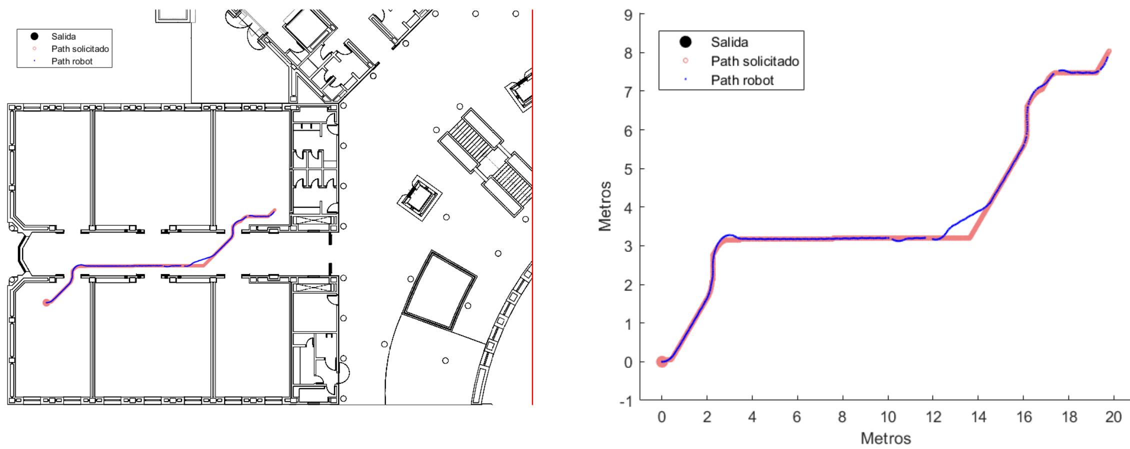


Figura 4.10: Recorrido 1 (Pioneer 3-DX)

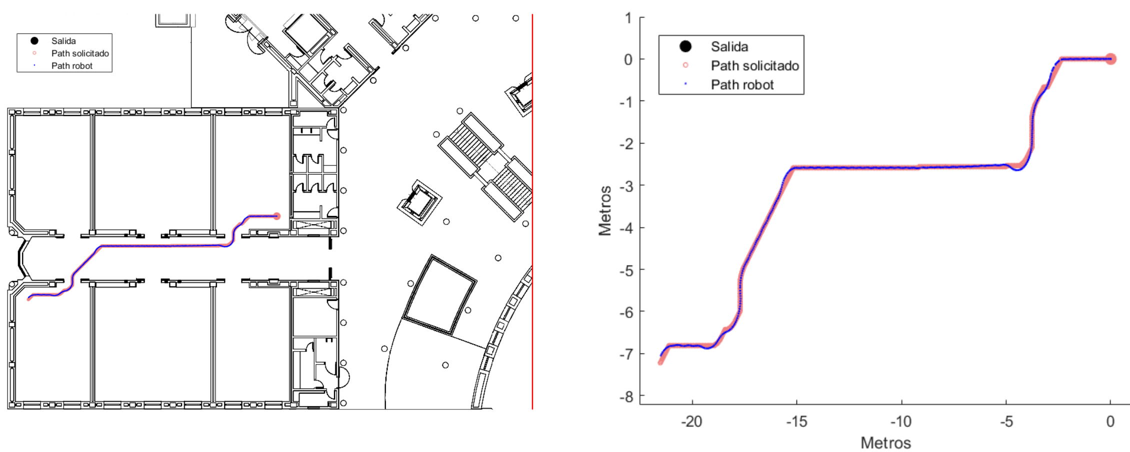


Figura 4.11: Recorrido 2 (Pioneer 3-DX)

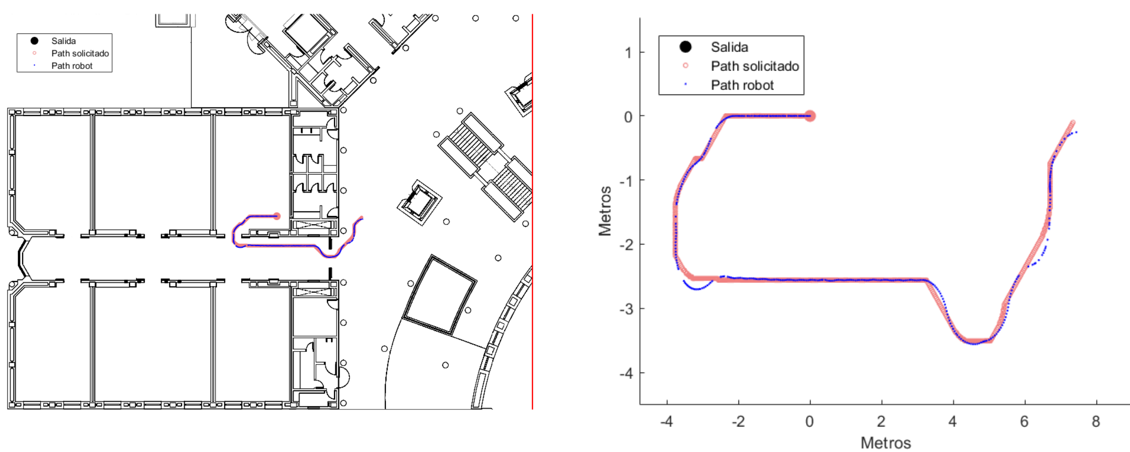


Figura 4.12: Recorrido 3 (Pioneer 3-DX)

A continuación se muestran algunas fotografías del robot utilizado durante los recorridos:

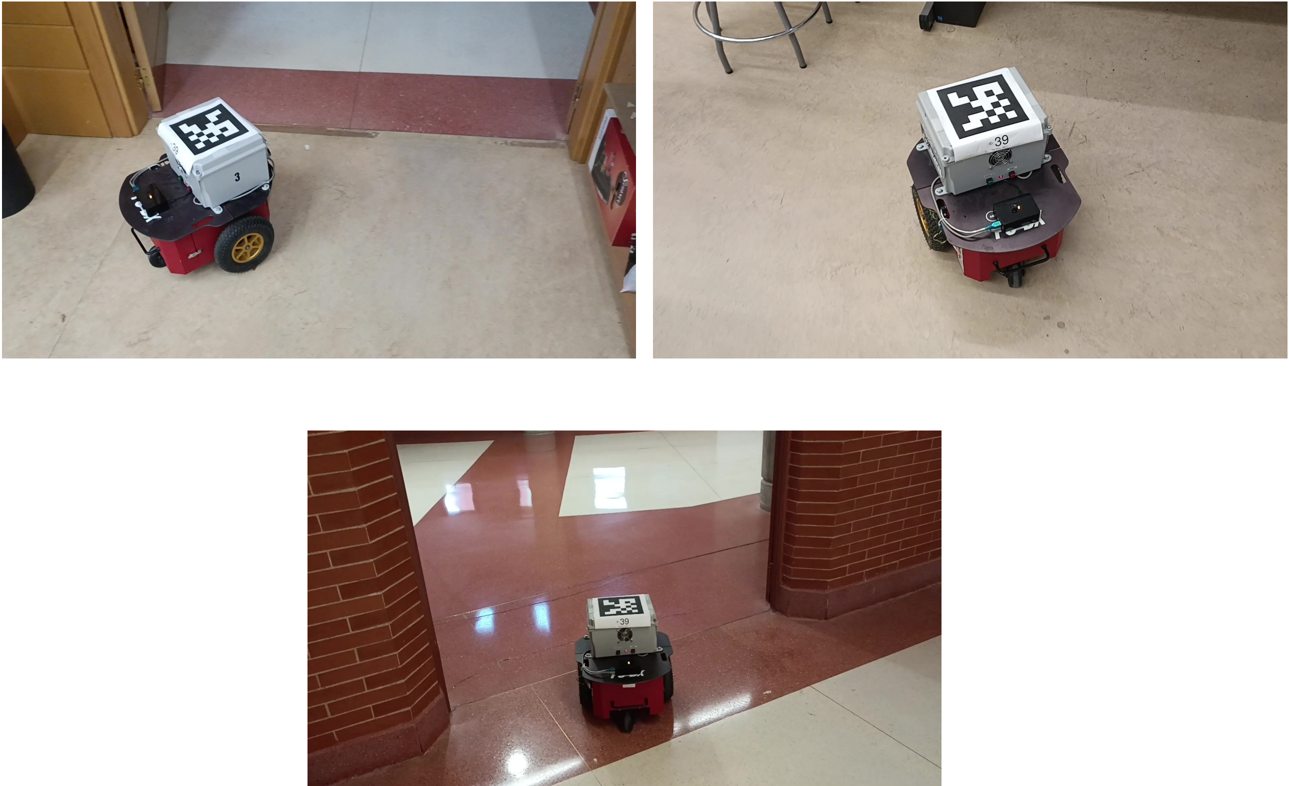


Figura 4.13: Recorridos Robot Pioneer 3-DX

Con estos recorridos con el robot real se ha comprobado que el path obtenido con el algoritmo de *Dijkstra* es factible tanto en simulación como con el robot real aunque al aplicarlo en este, con todo lo que ello conlleva, son necesarias algunas adaptaciones como se han comentado anteriormente.

Capítulo 5

Conclusiones y líneas futuras

En este capítulo se resumen las conclusiones obtenidas y se proponen futuras líneas de investigación que se deriven del trabajo.

5.1 Conclusiones

En base a las comparativas realizadas en el capítulo anterior podemos obtener las siguientes conclusiones:

- Si buscamos generar un *path* válido en el menor tiempo posible, debemos tener en cuenta el tipo de entorno con el que vamos a trabajar. Podemos suponer que a priori, en entornos estructurados el algoritmo *BiA** es el más recomendable. Sin embargo para entornos laberínticos recomendamos usar *A**
- Si comparamos el uso de memoria debemos tener en cuenta cuántas consultas queremos realizar. Para una única búsqueda el algoritmo que menos memoria necesita para sus cálculos, independientemente del entorno con el que se trabaje, es *Best First*. Si por el contrario queremos realizar varias búsquedas, es aconsejable el uso de *Dijkstra* debido a que no realiza la misma búsqueda en más de una ocasión.
- Si analizamos las distancias recorridas, cualquiera de los algoritmos estudiados en este proyecto, a excepción del *Best First*, genera recorridos muy similares en distancia, no destacando especialmente ninguno de ellos.
- Si queremos que el *path* sea lo más suave posible (sin muchos cambios de dirección), el algoritmo más aconsejable para ello es *Dijkstra* para entornos no laberínticos, siendo que para entornos laberínticos dependeremos del escenario en el que nos desplacemos y, por tanto, no hay preferencia por ninguno de los estudiados.

5.2 Líneas futuras

En este proyecto se han trabajado con 6 de los algoritmos más utilizados en *pathfinding*. Sin embargo existen muchas variaciones de estos así como otros nuevos que pueden ser estudiados de esta misma forma. Por tanto, para proyectos futuros, proponemos el estudio de otros algoritmos así como la realización de estas comparativas en entornos exteriores.

A su vez, se podría crear un sistema automático, el cual se encargue de procesar cada uno de los entornos que le introduzcamos. Este sistema podría detectar el tipo de entorno con el que se va a trabajar y en función de este, escoger un algoritmo u otro para la generación del path en base a la comparativa realzada en este proyecto y las necesidades del usuario.

Capítulo 6

Presupuesto

6.1 Hardware

Concepto	Precio	
Ordenador	900,00 €	
Robot Pioneer 3-DX	3900,00 €	
Placa Raspberry	170,00 €	
Telémetro Hokuyo URG-04LX-UG01	110,00 €	
	5080,00 €	Total

Tabla 6.1: Recursos hardware utilizados

6.2 Software

Concepto	Precio	
Matlab (1 año)	840,00 €	
	840,00 €	Total

Tabla 6.2: Recursos software utilizados

6.3 Personal

Concepto	Precio por hora	Cantidad de horas	Subtotal	
Ingeniero informático	30,00€/h	200 h.	6000,00 €	
			6000,00 €	Total

Tabla 6.3: Personal requerido

6.4 Presupuesto total

Concepto	Precio	
Hardware	5080,00 €	
Software	840,00 €	
Personal	6000,00 €	
	11920,00 €	Total

Tabla 6.4: Presupuesto total

Bibliografía

[1] S. LaValle, "Planning algorithms," *Motion Planning*, vol. 2, pp. 231–239, 2016.

[2] M. D. Armesto and L. E. Borgnino, "Sistema de control de navegación aplicado a vehículo autónomo," Ph.D. dissertation, Universidad Nacional de Córdoba (República Argentina), Septiembre 2018.

[3] A. V. Goldberg and R. F. Werneck, *Computing Point-to-Point Shortest Paths from External Memory*, 2005.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[5] "Difference between bfs and dijkstra's algorithms," https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[7] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving." [Online]. Available: https://www.osti.gov/biblio/5127296

[8] S. M. LaValle, "Rapidly-exploring random trees : a new tool for path planning," *The annual research report*, 1998.

[9] Z. Sen, *Spatial Modeling Principles in Earth Sciences.* Springer, 2016.

[10] 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. [Online]. Available: https://doi.org/10.1177/0278364911406761

[11] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving," *Addison-Wesley*, p. 48, 1984.

[12] H. U. C. Laboratory, H. Aiken, and I. S. on the Theory of Switching, *Proceedings of an International Symposium on the Theory of Switching: 2-5 April 1957*, ser. Annals of the computation laboratory. Harvard University Press, 1959.

[13] *Bidirectional A* Search with Additive Approximation Bounds*, https://www.studocu.com/en-us/document/georgia-institute-of-technology/artificial-intelligence/bi-directional-a-star-with-additive-approx-bounds/29207655.

[14] M. Likhachev, G. J. Gordon, and S. Thrun, "Ara*: Anytime a* with provable bounds on sub-optimality," in *NIPS*, 2003.

[15] H. Zhou, Y. Qi, Jason233Wang, and H. Dai, “Path planning,” https://github.com/zhm-real/PathPlanning, 2020.

[16] MATLAB, *version 9.12.0 (R2022a)*. Natick, Massachusetts: The MathWorks Inc., 2022.

[17] Python, *version 3.10*. Guido van Rossum, 2021.

[18] *Información acerca del robot Pioneer 3dx*, https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf.

[19] *Iniciación a Raspberry*, https://www.xataka.com/makers/cero-maker-todo-necesario-para-empezar-raspberry-pi.

[20] *Información sobre Telémetro de Escaneo Láser Hokuyo URG-04LX-UG01*, https://www.robotshop.com/es/es/telemetro-escaneo-laser-hokuyo-urg-04lx-ug01-ue.html.

[21] *Sitio web oficial de Ubuntu MATE*, https://ubuntu-mate.org/.

[22] Stanford Artificial Intelligence Laboratory et al., “Robotic operating system.” [Online]. Available: https://www.ros.org

[23] *Información acerca del simulador STDR*, http://wiki.ros.org/stdr_simulator.

[24] *Información acerca de LaTeX*, https://es.wikipedia.org/wiki/LaTeX.

Apéndice A

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC con Matlab [16]
- Compilador Python [17]
- Robot Pioneer 3-DX [18]
- Placa Raspberry [19]
- Telémetro de Escaneo Láser Hokuyo URG-04LX-UG01 [20]
- Entorno de desarrollo Ubuntu Mate [21]
- ROS [22]
- Stdr-simulador [23]
- LaTeX [24]

Apéndice B

Funciones

B.1 ImageToArray

```
%####### EDITAR #######

% Cargo la imagen
image = imread('Imagenes\MAPAS\Oeste-P1-Lab1-6.png');

% distancia de seguridad en metros
distanciaSeguridad = 0.25;

% Tamaño de cada nodo en metros
tamano = 0.2;

% metros reales equivalentes a la altura de la imagen
metros = 22.15;

radioRobot = 0.225; % radio en metros

%####### FIN EDITAR #######


% resolución del mapa original (metros reales/pixeles de la imagen)
resolucionOriginal = metros/size(image,1);

% resolución del mapa (metros reales/pixeles de la imagen)
resolucion = metros/size(image,1);

dSeguridadPixeles = distanciaSeguridad/resolucion; % distancia de seguridad
optimo = radioRobot/resolucion;

inflado = dSeguridadPixeles + optimo;

% Modificamos el tamaño de la imagen
bwimage = imresize(image,[size(image, 1), size(image, 2)]);

bwimage = rgb2gray(bwimage);
bwimage = bwimage>150;  % Ajustar valor rgb a blanco y negro según la necesidad

imageMaskResized = imcomplement(bwimage);

grid = binaryOccupancyMap(imageMaskResized);
inflate(grid, inflado);       % Ampliamos el número de celdas ocupadas
bwimage = occupancyMatrix(grid);
```

```
% Redimensionado

resolucion = size(image,1) / ((size(image,1) * resolucionOriginal) / tamano);
[image, image2, tamanoReal] = Segmentacion2(bwimage, resolucion, metros, tamano);

image = imcomplement(image);

figure
imshow(image);
impixelinfo;


% Si el blanco y el negro estuviesen invertidos, aplicamos esta función
%bwimage = imcomplement(bwimage);

% Almacenamiento de 0 de la imagen (paredes)
salida = double(image);

[y,x] = find(salida==0);

final = [x, size(image, 1)-y];

finalSinDecimales =arrayfun(@num2str,final,'un',0);

writecell(finalSinDecimales, 'MapaEnTexto.txt');
```

B.2 RecommendedNodeSize

```
function RecommendedNodeSize(tamanoPared)
% Este script permite, introduciéndole un tamaño en cm para el pasillo o
% puerta más pequeño, obtener rangos de valores para posibles tamaños de
% celda válidos para la exploración del mapa

seguridadYdiametro = (0.25 * 2 + 0.225 * 2) * 100;
x = linspace(0.1,50,500);
y = ((seguridadYdiametro./x)+3).* x; % Cambiar a +2 si no hay puertas diagonales

% el +3 porque: 2 posibles fallos en paredes exteriores %%% + 1 camino correcto      _|
% del nodo (no sumar)%%% + 1 camino extra para el nodo debido a que no puede ir    _|
% directamente en diagonal, por lo que el mínimo de caminos correctos sería      _|
% de 2, dando lugar a que el algoritmo buscase en escalera                       _|
|
l1 = [x;y];
y2 = ones(1,500);
y2(1,:) = tamanoPared;

l2 = [x;y2];

coincidencias = InterX((l1), (l2));

plot(l1(1,:), l1(2,:)); % Diagonal
hold on;
plot(l2(1,:), l2(2,:)); % Linea horizontal
ylabel('Tamaño máximo accesible (cm)')
xlabel('Tamaño celda (cm)')

disp("El tamaño de la celda debe ser inferior a " + coincidencias(1,1)/100 + "m")

end
```

B.3 Resize

```
function [imagenEditada, imagenFinal,tamanoReal] = Segmentacion2(imagenOriginal, resolucion, metros, tamano)

    pixelesAlturaFinal = ceil(metros/tamano);
    anchoOriginal = (metros * size(imagenOriginal,2)) / size(imagenOriginal,1);
    pixelesAnchuraFinal = ceil(anchoOriginal / tamano);

    %METODO 1 AUTOMATICO, PERO ELIMINA ALGUNAS PAREDES
    imagenFinal = imresize(imagenOriginal, [pixelesAlturaFinal, pixelesAnchuraFinal], 'nearest');

    %METODO 2, DIBUJA MEJOR EL MAPA PERO SU PRECISION DE TAMAÑO RESPECTO A
    %LA SOLICITADA VARÍA LIGERAMENTE
    pxNecesarios = ceil(metros/tamano);

    %-----------------------------------------------------------------

    divisionesAnchura = size(imagenOriginal,1) / resolucion;

    tamanoCelda = ceil(size(imagenOriginal,1)/divisionesAnchura);

    columnasVacias = zeros(size(imagenOriginal,1), (pxNecesarios * tamanoCelda) - size(imagenOriginal,2));

    imagenOriginal = [imagenOriginal, columnasVacias];

    %-----------------------------------------------------------------

     for h = 1: size(imagenOriginal,1)/tamanoCelda
         for k = 1: size(imagenOriginal,2)/tamanoCelda

             for j = (h-1) * tamanoCelda + 1 : h*tamanoCelda
                 for i = (k-1) * tamanoCelda + 1 : k*tamanoCelda
                     try
                         valorCeldaNueva = valorCeldaNueva + imagenOriginal(j,i);
                     catch
                         %disp('Me salí del rango')
                     end
                 end
             end

             if floor(valorCeldaNueva) >= 1
                 imagenEditada(h,k) = 1;
             else
                 imagenEditada(h,k) = 0;
     for k = 1: ceil(size(filasVacias,1)/tamanoCelda)
         imagenEditada(size(imagenEditada,1),:) = [];
     end



     disp("Altura = " + size(imagenEditada,1));
     disp("Anchura = " + size(imagenEditada,2));

     tamanoReal = metros / size(imagenEditada,1);
     disp("El tamaño de la celda es de: " + tamanoReal + " m/nodo");

 end
```

Universidad de Alcalá
Escuela Politécnica Superior

Universidad
de Alcalá