Técnicas de los Sistemas Inteligentes (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

PRÁCTICA 1:
Planificación de caminos en Robótica

Mª del Mar Alguacil Camarero

Resumen

En esta práctica se ha implementado en ROS técnicas de navegación global para un robot móvil mediante la implementación del algoritmo A*, a partir del código proporcionado por el profesor, como un plugin de ROS para ser ejecutado como planificador global. Para ello, en una primera fase, se ha modificado el código dado para implementar el algoritmo A*, teniendo en cuenta las distintas posiciones (simplificadas) del robot para una mayor seguridad. En una segunda fase, se ha procedido a mejorar el tiempo de planificación del robot mediante técnicas como *Jump Point Search* y pesos proporcionales, suponiendo que el robot es cuadrado, teniendo en cuenta una matriz que nos indica si una celda ha sido expandida y explorada, y utilizando una lista ordenada para la de abiertos. Por último, se ha realizado una visualización, tanto de los nodos expandidos y explorados como del camino hallado, y una comparativa de ambas tareas.

A continuación se explica detalladamente todos los pasos seguidos para las distintas tareas que se han nombrado en el resumen anterior.

1. Algoritmo A* con seguridad

En esta primera fase, se ha modificado el código proporcionado para conseguir el algoritmo A*. En este se pretende explorar el nodo más prometedor, por ello, se ha ordenado la lista de abiertos según el `fCost` de cada nodo expandido. Pero también debemos tener en cuenta que podemos encontrar un camino mejor, y por tanto, debemos actualizar la información (`parent`, `gCost` y `hCost`) de los nodos. Como sabemos, encontramos un camino mejor si `el costo desde el nodo inicial al nuevo padre + el costo del ir del padre al nodo < el costo anterior desde el inicio al nodo`; en dicho caso, cambiamos el padre por el nuevo, actualizamos el `gCost` de la celda por este nuevo y en consecuencia el `fCost`, que es la suma de `hCost` y `gCost`.

Como deseamos también que el robot sea lo más seguro posible debemos comprobar la probabilidad de colisión que se produce si dicho robot se posiciona en un lugar determinado. Como sabemos que cada celda tiene un valor comprendido entre 0 y 255, indicando de 0 a 254 la probabilidad de choque (siendo 254 obstáculo y, por tanto, colisión asegurada) y 255 las celdas desconocidas, la seguridad se ha conseguido teniendo en cuenta que la posición central debe tener un valor menor de 127y comprobando si en alguna de las posiciones simplificadas del robot de 0º, 90º, 180º y 270º se encontraba algún obstáculo o casilla desconocida mediante la función `collision` que comprueba dichas posiciones ayudándose de *footprintCost*, la cual se ha modificado para que devuelva un valor booleano indicando si se produce o no choque o se encuentra dicha celda fuera del mapa; se descartaba dicho movimiento del robot hacia esa posición. Pero a pesar de determinar un camino seguro, según el mapa considerado, en ocasiones, el planificador local no realiza los movimientos adecuados o el mapa no se corresponde fielmente a la realidad y, por tanto, el robot sufre colisiones o debe volver a planificar de nuevo dicho camino.

2. Mejora algoritmo A*

Inicialmente, para que la comprobación más rápida de celdas expandidas o exploradas o ni expandidas ni exploradas (las cuales llamaremos libres) se ha procedido a crear una matriz llamada `mnode`, de igual tamaño que el mapa, que almacena dicho valores. Permitiéndonos averiguar de forma más eficiente si un celda está en la lista de abiertos o cerrados o en ninguna de estas, sin tener que recorrer cada vez cada una de ellas. Por tanto, se ha modificado el `isConstains`, al cual solo debemos pasarle como parámetro la identificador de la celda y la lista a la que queremos

comprobar si pertenece, y además cada vez que se inserta un elemento en una de estas listas se debe modificar dicha matriz.

Para evitar tener que pasar a coordenadas del mundo y luego comprobar las distintas posiciones simplificadas del robot, se ha supuesto que el robot era cuadrado, hallando inicialmente el punto más alejado del robot mediante distancia euclídea y calculando el número de celdas equivalentes. Seguidamente se ha procedido de forma análoga a la comprobación de seguridad: el punto central debe ser menor que 150 y en las celdas de alrededor que conforman el cuadrado no debe existir ningún obstáculo o desconocimiento.

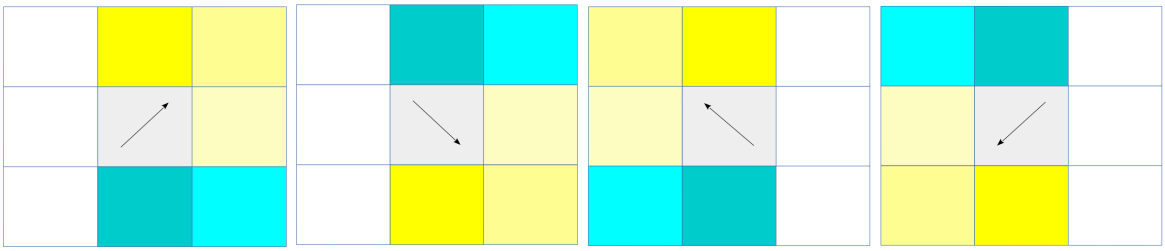
Para no tener que ordenar la lista de abiertos cada vez que introducimos nuevos elementos se ha creado la clase `OPlist` que simula una lista de celdas enlazadas ordenada, en la que cada vez que introducimos un elemento en una instancia de dicha lista, se coloca en la posición en la cual su $fCost$ es mayor que el del elemento anterior pero menor que el del siguiente. Para dicha clase se ha implementado sólo los métodos necesarios para esta práctica como son `push_back`, `clear`, `tope`...

Una mejora básica en el algoritmo es la técnica de los pesos proporcionales, en la cual se ha decidido que tenga una pequeña mayor consideranción el valor de `hCost` según el 70 % de la distancia al nodo objetivo. Esto se ha conseguido modificando el `fCost` de la siguiente manera: `fCost = gCost + w*hCost` siendo `w = 1.0 + 0.7*hCost/hCostStart`, con `hCostStart` la longitud del camino estimado del nodo inicial al objetivo. En `w` se ha sumado 1 para que siempre podamos considerar el `hCost`, es decir, impedir que `fCost` $\Rightarrow$ `gCost` cuando `w`$\Rightarrow$`0`.

Por último se ha implementado la técnica de *Jump Point Search* que a continuación detallaremos. En esta técnica se van recorriendo las distintas celdas adyacentes pero sólo almacenamos en la lista de abiertos los sucesores que muestran algún cambio de dirección hacia un pasillo, una entrada o simplemente para evitar un obstáculos, los cuales llamaremos vecinos forzados.

Consideramos tres casos de vecinos forzados generales, que dependerán de la dirección del movimiento:

1. **Diagonal:** La celda justo debajo de la que estamos comprobando o la adyacente a esta en la dirección del movimiento en el eje X tiene alta probabilidad de colisión (estás se muestran en la figura 2.1 de azul oscuro a claro, respectivamente). Y la adyacente a la actual en la dirección X o la adyacente considerando ambas direcciones está libre o sólo considerando la dirección Y (correspondientes a las celdas del amarillo más claro al más oscuro en la figura 2.1, respectivamente).

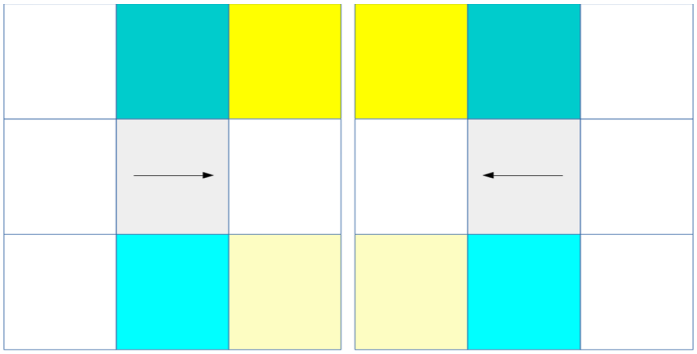


(a) 1 (b) 2 (c) 3 (d) 4

Figura 2.1: Posibles vecinos forzados diagonales.

2. **Horizontal:** Alguna de las celdas de arriba o abajo (las cuales se muestran en la figura 3.3 de azul oscuro y claro respectivamente) tiene alta probabilidad de colisión. Mientras que la

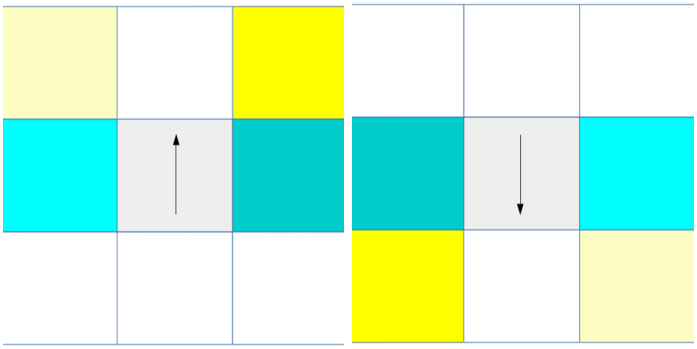
adyacente a la obstaculizada debe estar libre (correspondientes a las celdas amarrillas).



(a) 1 (b) 2

Figura 2.2: Posibles vecinos forzados horizontales.

3. **Vertical:** Análogas a la anterior pero considerando las colisiones a derecha e izquierda como se puede observar en la figura 2.3.



(a) 1 (b) 2

Figura 2.3: Posibles vecinos forzados verticales.

Nótese que la celda actual se representa en figura 2.1, 3.3 y 2.3 de color grisáceo.

Para identificar los sucesores prometedores que nos muestran algún cambio en el mapa (vecino forzado) o que permiten saltar a algún vecino forzado, se utiliza la función `identifySuccessors` que inicialmente busca los vecinos correspondientes al nodo pasado como parámetro, y para cada vecino halla la dirección del nodo actual al vecino y comprueba si en esa dirección hay algún vecino forzado o nodo de salto a alguno de estos (lo cual se averigua con la función `jump` que a continuación explicaremos). En caso de que nos interese dicho nodo, es decir, exista alguna celda que cumpla lo anterior, lo guardamos como en la lista de sucesores que será devuelta.

Para averiguar las celdas relevantes nos ayudamos de la función `jump` que comprueba si el nodo no ha sido explorado anteriormente se puedan diferentes casos:

- Si hay alta probabilidad de colisión o se encuentra fuera de los límites del mapa, se devuelve -1.

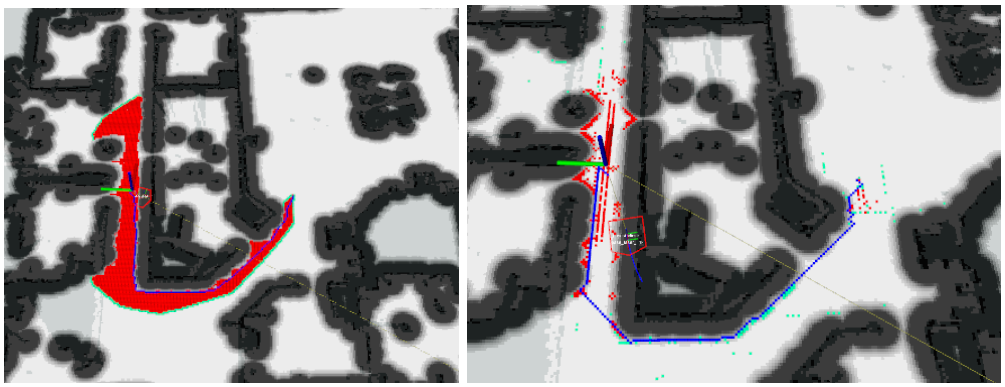
- Si es el nodo objetivo, devolvemos el índice de este.
- Si dicha celda es un vecino forzado diagonal, horizontal o vertical, devolvemos el identificador de dicho vecino.
- Si la dirección del nodo padre a él es en diagonal y no es un vecino forzado, se comprueba si existe algún vecino forzado en dirección vertical y horizontal.
- Si no se produjo ninguno de los casos anteriores, nos desplazamos a la siguiente celda en la misma dirección.

3. Comparativa

En esta sección se muestra la resolución de distinto experimentos cambiando el punto inicial y final para determinar el menor camino mediante las tareas explicadas anteriormente. Para ello, se visualizará dichos caminos de color azul (para el cual se ha añadido un botón denominado `Camino` modificando el fichero `single_robot_OpenClosed.rviz` y añadiendo las correspondientes líneas en `myAstarPlanner.h/.cpp`), y se mostrará los nodos expandidos en color verde y los explorados en rojo. Además se realiza una tabla sobre comparando los resultados obtenidos fijando dos puntos (inicial y final), en dicha tabla se podrá observar el tiempo que ha tardado en averiguar dicho camino, el número de nodos que ha tenido que expandir y la longitud del camino encontrado, tanto en metros como en cantidad de nodos. Nótese que dicho caminos se han hallado en el mapa de 5 cm, ya que se corresponde un poco mejor con la realidad.

A continuación se muestra la tabla con los distintos resultados de los diferentes experimentos realizados.

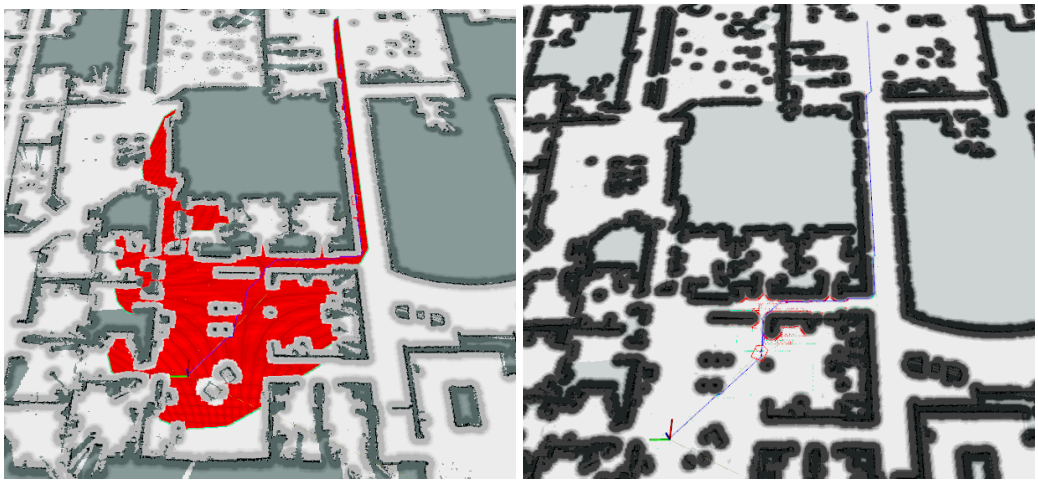
	Camino		Tiempo	Nodos	Longitud camino	
	Pto inicial	Pto final	búsqueda	expandidos	Metros	Nodos
Algoritmo A*	(26.123,	(25.989,	1.40	3374	10.301218	182
Algoritmo mejorado	41.580)	36.430)	1.00	433	11.093591	243



(a) Algoritmo A* (b) Algoritmo mejorado

Figura 3.1: Camino de (26,123, 41,580) a (25,989, 36,430).

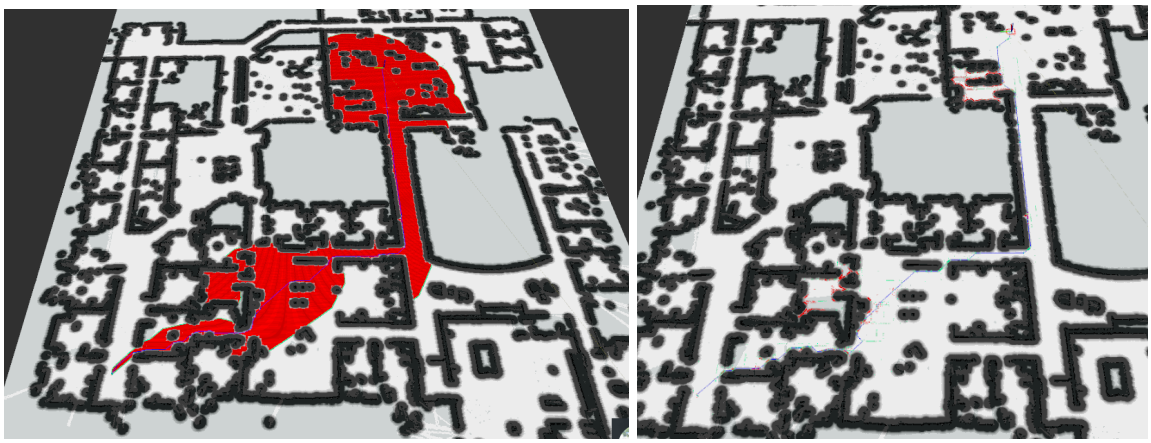
	Camino		Tiempo	Nodos	Longitud camino	
	Pto inicial	Pto final	búsqueda	expandidos	Metros	Nodos
Algoritmo A*	(13.562,	(43.820,	191.40	39552	37.831396	709
Algoritmo mejorado	28.610)	17.690)	1.30	541	38.814111	751



(a) Algoritmo A* (b) Algoritmo mejorado

Figura 3.2: Camino de (13,562, 28,610) a (43,820, 17,690).

	Camino		Tiempo	Nodos	Longitud camino	
	Pto inicial	Pto final	búsqueda	expandidos	Metros	Nodos
Algoritmo A*	(43.740,	(10.767,	1530.20	72846	50.038218	920
Algoritmo mejorado	17.666)	40.025)	3.80	1281	53.010	1081



(a) Algoritmo A* (b) Algoritmo mejorado

Figura 3.3: Camino de (43,740, 17,666) a (10,767, 40,025).

Es claro que para caminos cortos no hay mucha diferencia en cuanto a tiempo y longitud, aunque en nodos expandidos si es más significativa. Pero realmente las diferencias en tiempo de búsqueda

y en memoria requerida se va notando conforme aumentamos la distancia de separación del punto inicial y final, y los obstáculos que se encuentran en el camino. Podemos observar que a pesar de que el camino obtenido con la mejora no es tan bueno como con el algorimoA* normal, se obtiene una diferencia poco significativa en comparación con el tiempo que tarda en realizar la búsqueda, pudiéndose conseguir en tiempo real en puntos con bastante separación y obstáculos.