

PRÁCTICA 1

PLANIFICACIÓN DE CAMINOS EN ROBÓTICA

Óscar David López Arcos, 75571640-B
odlarcos@correo.ugr.es
Grupo A2: Miércoles 17:30-19:30

Resumen	3
1 ^a Tarea: Construcción del algoritmo A* y Seguridad	3
Algoritmo A*	3
Seguridad	4
2 ^a Tarea: Mejora del Algoritmo	4
Pesos en la función f(x)	4
A* Bidireccional	5
A* generación de vecinos cada n casillas	5
3 ^a Tarea: Experimentación	6
Experimento 1:	6
Experimento 2:	6
Experimento 3:	7

Resumen

La gran mejora que he planteado en esta práctica ha sido construir un A* bidireccional. Además, si una vez explorado un número máximo de nodos éste no ha encontrado solución, se lanzará un A* especial que genere vecinos cada n casillas (como explicaré a continuación), siendo mucho más rápido aunque poco eficiente. También he incluido una leve mejora basada en asociar pesos en heurística y coste.

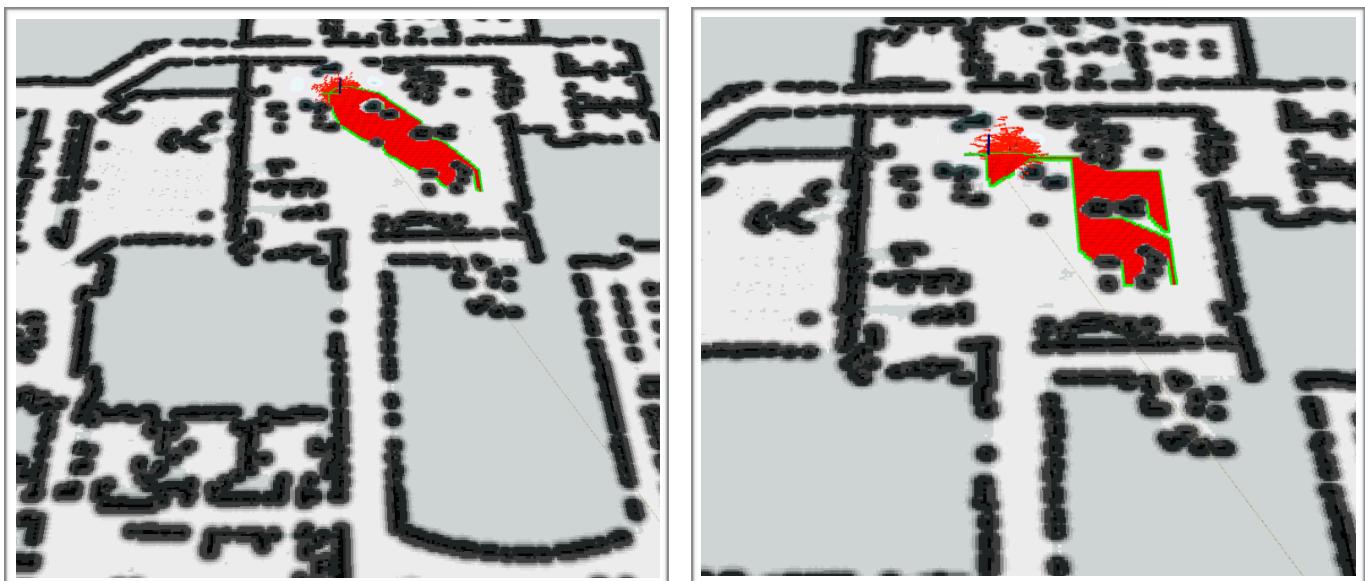
Antes de eso, tuve que modificar myastarPlanner.cpp para conseguir pasar de una búsqueda en anchura a un algoritmo A*, incluyendo una heurística para llegar al objetivo y manteniendo el coste de la forma “coste_padre + distancia(padre,nodo)”. Por otro lado, he añadido al camino cierta seguridad, no permitiendo que acepte como nodos vecinos aquellos que comprometen la seguridad del footprint del robot (footprintCost).

1ª Tarea: Construcción del algoritmo A* y Seguridad

Algoritmo A*

Para construir el algoritmo A*, el primer paso fue modificar la función “addNeighborCellsToOpenList” para incluir en cada nodo una heurística que nos indique un coste aproximado hasta alcanzar la posición goal.

En cuanto a la heurística, probé tanto la distancia Manhattan (derecha) como la Euclídea (izquierda), generando cada una soluciones diferentes.



Por ser más directa la Euclídea, finalmente decidí trabajar con ella.

Es importante destacar que la inserción de nodos en la openList está implementada para que resulten ordenados según la función $f(x) = g(x)+h(x)$, de menor a mayor (usando operador de inserción de stl con eficiencia logarítmica), donde g representa el coste (distancia real recorrida para llegar a ese nodo) y h la heurística. De esta forma en cada iteración el nodo situado al comienzo de la lista es el más prometedor y, por tanto, con el que trabajaremos ($CofCells = openList.front()$).

También he implementado un algoritmo que revise aquellos hijos de $CofCells$ (nodo actual) que ya estaban en la $openList$ para actualizar su coste en el caso de que $CofCells$ sea mejor parente que el que tenían antes. Esta función la he llamado “ $updateNeighborsInOpenList$ ”.

Los nodos en closedList no es necesario actualizarlos, ya que al ser un coste monótono este no variaría.

Seguridad

Para resolver el problema de la seguridad, he experimentado con distintas posibilidades que comentaré a continuación.

Para evitar que el robot colisione con las paredes, he modificado el método “footprintCost”, de forma que nos permita conocer cuándo uno de los vértices de éste estaría sobre un LETHAL_OBSTACLE si el robot ocupase la posición definida por parámetro. En el método findFreeNeighborCell compruebo entonces que el nodo vecino no devuelva el valor 99999 (correspondiente a LETHAL OBSTACLE) para añadirlo a la lista de vecinos a explorar. También compruebo que el valor de esa casilla en el costmap sea menor de 128, donde siendo 0 libre y 255 ocupado, supone una proximidad a obstáculos media.

También implementé dentro de la función “footprintCost” que devolviese un valor en función de lo cerca que esté el robot quedarse de un obstáculo. Este valor se sumaría a la heurística de la casilla en cuestión. Sin embargo, tras realizar unas cuantas pruebas, pude comprobar que esta última mejora no era del todo necesaria llegando a entorpecer en ciertos casos, explorando bastantes nodos por querer encontrar un camino demasiado seguro.

Finalmente, he considerado una variación de esta última mejora siguiendo la misma filosofía de la primera. Si en un rango cercano a un vértice del FootPrint se encuentra sobre un LETHAL OBSTACLE, también devolverá 99999.

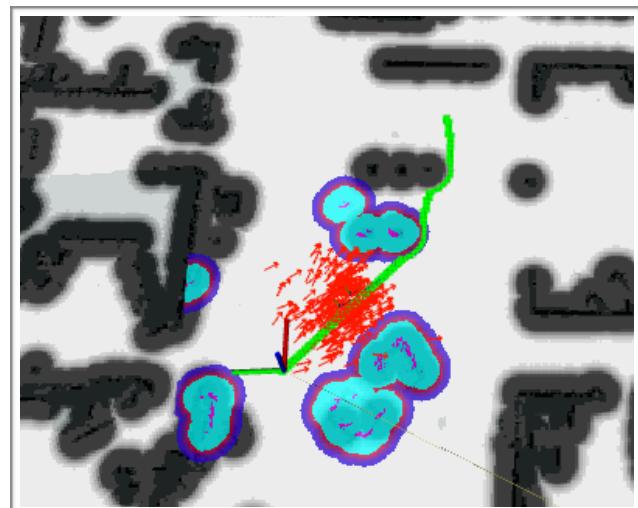
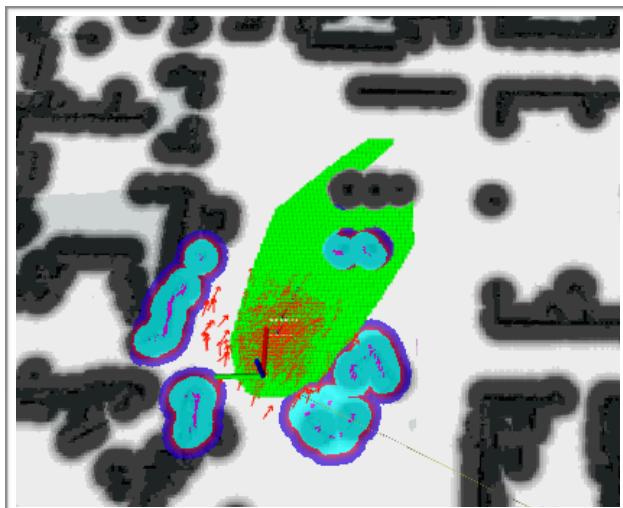
En el caso de querer implementar mi algoritmo a un robot real, sí haría uso de la segunda medida, ajustando los parámetros para un funcionamiento más eficiente.

2^a Tarea: Mejora del Algoritmo

Pesos en la función f(x)

La primera mejora respecto al Algoritmo A* básico ha sido muy simple a la par que efectiva. Incluyendo unos pesos a la hora de evaluar la función f, de forma que $f(x) = \mu \cdot g(x) + (1-\mu) \cdot h(x)$ con un valor de μ entre 0 y 1, pueden conseguirse grandes mejoras.

Por ejemplo, para un $\mu=0'3$, damos mucha más importancia a la heurística que al coste, traduciéndose esto en una exploración de los nodos más directa hacia el objetivo. Esto provoca que menos nodos sean explorados y, por tanto, tarde menos tiempo.



A* Bidireccional

La segunda mejora, más compleja e interesante, ayuda a generar un camino explorando menos nodos en numerosas ocasiones. Se trata de construir un algoritmo A* bidireccional, es decir, que comience dos búsquedas simultáneas: una desde start a goal (original) y otra desde goal a start (inversa).

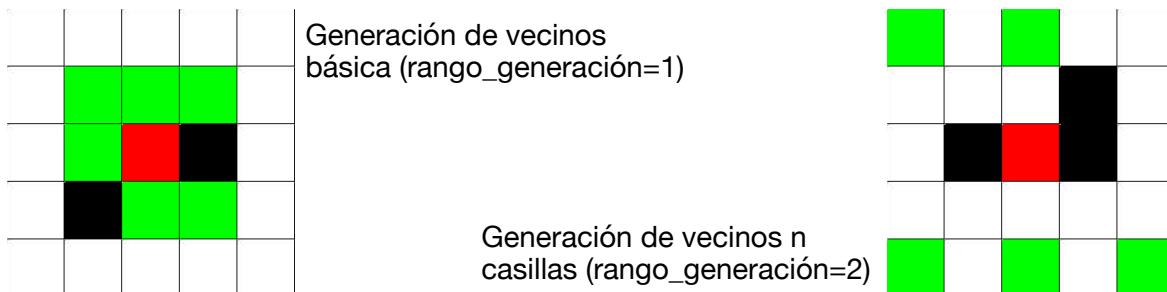
La mejora de este algoritmo respecto al anterior puede observarse mejor cuando uno de los caminos se atasca, permitiendo al otro llegar mucho más rápido. Sin embargo, no es perfecto, si ambos caminos quedan atascados la exploración será más costosa que el A* básico. Todos estos casos los veremos detallados con imágenes en la 3^a Tarea.

La implementación es bastante sencilla: trabajar con dos openList y closedList de forma simultánea, de manera que una trabaje en el camino original y otra en el inverso. Sin embargo, para mejorar aún más este algoritmo, si ambos caminos llegan a cruzarse en un punto x de la búsqueda, el robot construirá inmediatamente el camino start – x – goal.

A* generación de vecinos cada n casillas

Si el A* bidireccional no encuentra solución tras haber explorado un número máximo de casillas definido previamente (maximo_nodos), la búsqueda se reiniciará. Esta vez, comenzará un nuevo algoritmo de búsqueda que, a pesar de no devolver caminos muy eficientes, es capaz de sortear obstáculos con cierta soltura como veremos en la experimentación.

Para ello, modifica la distancia de generación de vecinos a una definida previamente (rango_generacion), comprobando siempre que entre padre-hijo no exista ningún obstáculo.



Este método permite avanzar con mucha más rapidez, ya que los hijos se generan n veces más rápido, alcanzando el objetivo más fácilmente. Una vez la distancia entre la casilla actual COfCells y la casilla objetivo cgoal sea menor que n, la variable rango_generación se volverá a igualar a 1, para así asegurarnos encontrar el camino.

3^a Tarea: Experimentación

Experimento 1:



En este experimento puede verse con claridad la mejora que ofrece en ciertas ocasiones el algoritmo A* bidireccional cuando uno de los caminos se atasca. En la imagen de la izquierda (bidireccional) se observa como el camino inverso alcanza el original, expandiendo un total de 520 nodos frente a 3527 del A* básico (derecha). El tiempo también se reduce considerablemente, de unos 10 segundos a 0,6.

Experimento 2:



Aquí, con 634 nodos frente a 2752, se demuestra que no es necesario que ambos caminos se crucen. Con que uno de ellos llegue al objetivo es suficiente para conseguir una mejora considerable.

Experimento 3:



Sin embargo, como ya aveciné antes, esta mejora no funciona siempre. En zonas donde ambos caminos quedan atascados, la exploración se duplica, alcanzando los 8500 nodos explorados y tiempos de 30 segundos.

Es entonces donde entra en escena la última mejora, donde con apenas 1042 nodos (teniendo en cuenta que comenzó una vez el A* bidireccional había explorado 4000) puede encontrar una solución.