

# Técnicas de los Sistemas Inteligentes

## Práctica 1 - Entrega 3: Navegación global

José Pimentel Mesones      Lothar Soto Palma      Francisco David Charte Luque  
José Carlos Entrena Jiménez

### Resumen

En computación, el algoritmo A\* es un proceso de búsqueda entre nodos de un grafo, que usa una heurística de estimación del coste hasta la solución, una función usualmente llamada  $h(n)$ , y una función de coste,  $g(n)$ , cuya suma nos da una estimación del coste de llegar de un nodo a una solución del problema. En esta práctica se ha completado una implementación del algoritmo A\* con el uso de una cola con prioridad para la gestión de la lista de nodos abiertos, y lo hemos probado en distintos mundos para comprobar su efectividad.

## Índice

<b>1. Extensión del algoritmo A*</b>	<b>2</b>
1.a. Modificación de estructuras de datos . . . . .	2
1.b. Implementación . . . . .	2
<b>2. Mejora del algoritmo A*</b>	<b>2</b>
2.a. Modificación del número de iteraciones . . . . .	2
2.b. Adición de pesos . . . . .	2
2.c. Gestión de la resolución . . . . .	2
2.d. Cálculo del coste del <i>footprint</i> . . . . .	3
<b>3. Experimentación en Stage</b>	<b>3</b>
3.a. willow_garage . . . . .	3
3.b. autolab . . . . .	4
3.c. simple_rooms . . . . .	4

## 1. Extensión del algoritmo A\*

### 1.a. Modificación de estructuras de datos

Para conseguir un funcionamiento más intuitivo del algoritmo A\*, hemos hecho algunos cambios en las estructuras de datos ya presentes en el código. Hemos pasado de gestionar la lista de ABIERTOS como una lista de la STL a una cola con prioridad, que nos permite insertar ordenadamente los nodos en ABIERTOS según el valor de la función  $f(n)$ , lo que hace que tomar el mejor nodo sea rápido y no implique un proceso de búsqueda en toda la lista.

Dicha cola con prioridad va soportada sobre un vector de la STL, y utiliza un operador de comparación para el orden. Este operador se ha implementado para los elementos `CoupleOfCells`, que ha pasado de ser un *struct* a una clase para definir el método `operator>`, que simula el comportamiento que había implementado en la función `compareFCost`.

### 1.b. Implementación

El resto de la implementación consiste en seguir los pasos del algoritmo A\*. Comenzamos creando el *goal* y el *start*, e insertamos el punto de inicio en la cola de ABIERTOS. Una vez tenemos los datos iniciales, empezamos el proceso del algoritmo: tomamos el primer elemento de la cola de ABIERTOS y lo insertamos en la lista de CERRADOS. Si el elemento no es el *goal*, lo quitamos de la cola de ABIERTOS y lo expandimos, obteniendo sus nodos vecinos mediante el uso de la función `findFreeNeighborCell`. Obtenidos los vecinos, consideramos únicamente aquellos que no se encuentran en la lista de CERRADOS, y comprobamos si están en la cola de ABIERTOS, actualizando su valor si fuera necesario. Una vez aquí, repetimos el proceso.

Hemos mantenido las funciones  $g(n)$  y  $h(n)$  que venían en el código, que usan la distancia euclídea.

## 2. Mejora del algoritmo A\*

En la experimentación hemos detectado que al pasarle un objetivo lejano al robot, el número de iteraciones no era suficiente para que el algoritmo A\* encontrase un camino hasta dicho objetivo, lo que hacía que tuviera un comportamiento errático no deseado. Para solucionar esta situación, hemos hecho diversas modificaciones.

### 2.a. Modificación del número de iteraciones

Aumentar el número de iteraciones máximas que el algoritmo puede realizar, lo que nos permite llegar a objetivos más lejanos, en el caso en el que la exploración en A\* sea muy costosa.

### 2.b. Adición de pesos

Para añadir un peso a la función  $h(n)$ , usamos una nueva función  $w(n)$  que actúa como peso dinámico de la función  $h(n)$ . Para calcular dicho peso, dividimos el valor de la función  $h$  calculada en el nodo actual entre el valor en el nodo *start*. De esta forma, conseguimos que la función  $h$  pierda relevancia conforme nos vamos acercando al objetivo, tomando más importancia la distancia real hasta el punto (la función  $g$ ) que la estimación que calculamos.

### 2.c. Gestión de la resolución

Se ha reducido la resolución del mapa, para tener un menor número de nodos que explorar. Al tener una resolución demasiado alta, el número de casillas exploradas puede ser excesivo en comparación con la distancia real recorrida, lo que nos lleva a tener unos tiempos de computación altos.

## 2.d. Cálculo del coste del footprint

Para evitar que el robot pase demasiado cerca de obstáculos e incluso se choque con ellos, se ha completado la implementación de la función `footprintCost`, que permite averiguar si la posición de la silueta del robot sobre una celda es legal (es decir, si no se choca con un obstáculo).

## 3. Experimentación en Stage

### 3.a. willow\_garage

El mapa `willow_garage` es complejo y está lleno de obstáculos, por lo que se ha elegido un camino corto para ilustrar el comportamiento del algoritmo A\* en nuestro robot. En una de las “habitaciones” inferiores, se ha colocado al robot en un extremo y se le ha ubicado el objetivo en el extremo opuesto de la habitación, como se observa en la Figura 1, que representa el camino (sobre RViz) que encuentra el robot.

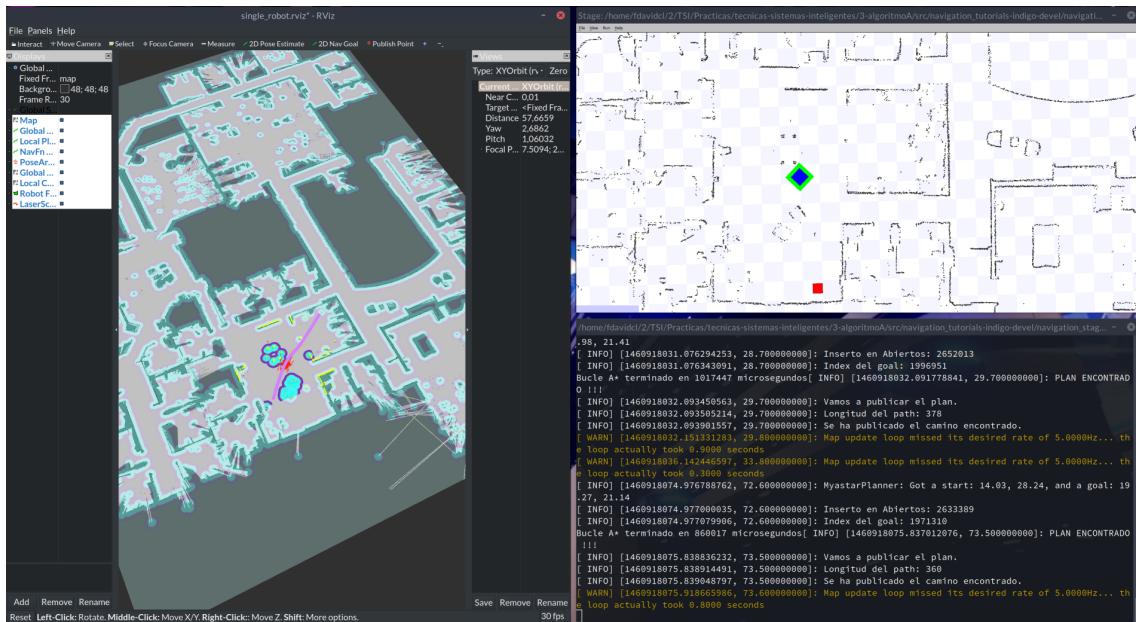


Figura 1: Camino seguido por el robot en `willow_garage`

Observamos en la Figura 2 cómo las mejoras de la sección 2 permiten que el cálculo se haya realizado de forma rápida (aproximadamente 0,8 segundos) y el robot deje algo de espacio con los obstáculos.

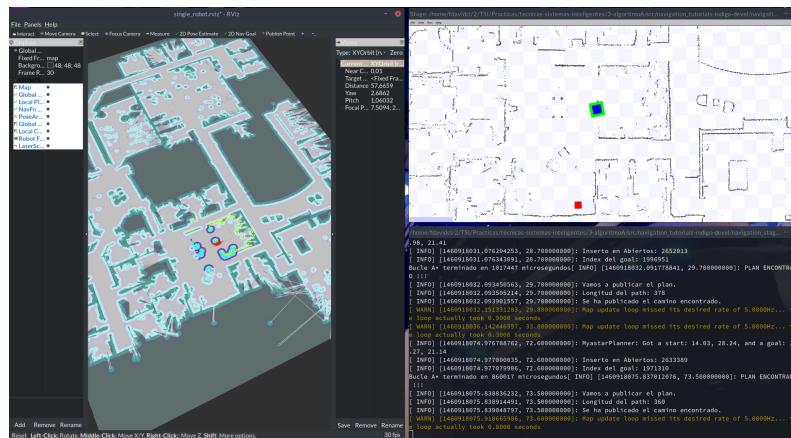


Figura 2: El robot sorteó los obstáculos del camino

Por último, se muestra en la Figura 3 que el robot alcanza su objetivo con éxito.

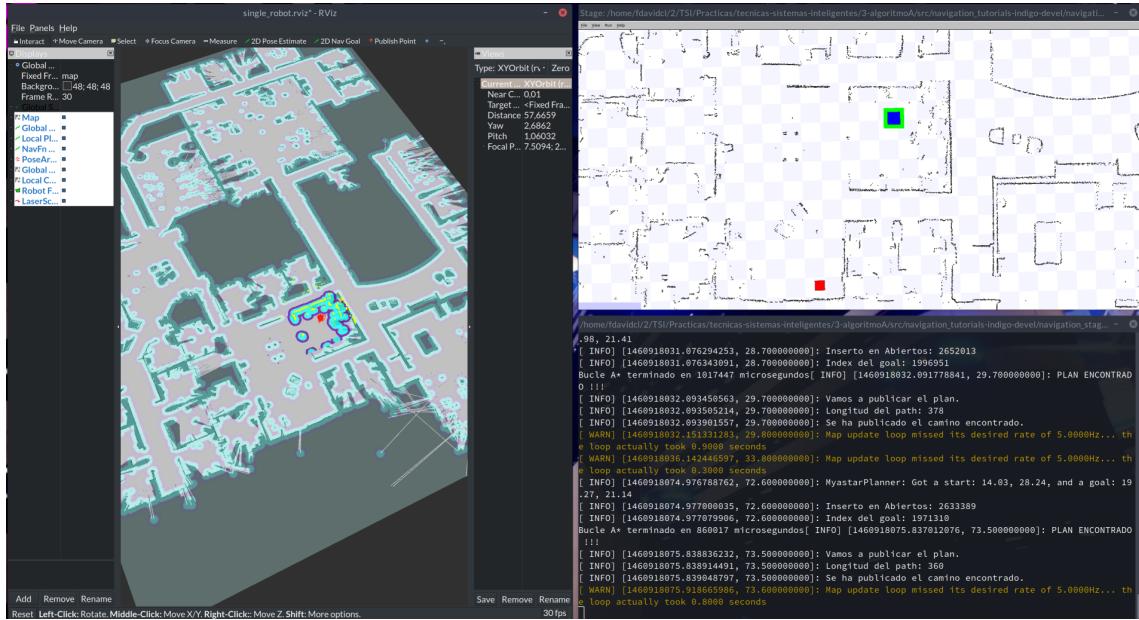


Figura 3: El robot alcanza el objetivo

El resto de la experimentación se han realizado en los mapas *autolab* y *simple\_rooms*. En estos casos, antes de pasar a la ejecución, hemos tenido que ajustar las coordenadas de los costmap de ambos mapas, pues no se correspondían con las del stage y tampoco con las coordenadas rotadas 270°, por lo que hemos tenido que calcularlas. Han sido añadidas al archivo **amcl\_node.xml** a modo de comentario.

### 3.b. autolab

El mapa *autolab* es relativamente sencillo al no tener apenas obstáculos, como observamos en la figura 4. El algoritmo es capaz de buscar caminos mínimos en el caso de que tenga una ruta sin muchos obstáculos, aunque debido a que la resolución del mapa es mayor a la anterior, distancias más cortas requieren de una búsqueda de mayor profundidad para obtener un camino mínimo, haciendo que la capacidad máxima de búsqueda del algoritmo pueda agotarse. Como se puede observar en la Figura 5, seguimos evadiendo celdas ilegales. Finalmente, en la Figura 6 vemos como el robot alcanza el objetivo.

### 3.c. simple\_rooms

El mapa *simple\_rooms* como el anterior carece de obstáculos, pero al ser habitaciones cerradas con una única salida y tener también una resolución mayor que el *willow* es posible que no encuentre un camino por falta de capacidad. En este caso se ha probado a avanzar con el robot dando objetivos cercanos sin muchos obstáculos o con obstáculos que pueden evitarse fácilmente, de esta manera el algoritmo encuentra soluciones con bastante rapidez como podemos ver en las figuras 8 y 9, sin embargo una vez nos hemos avanzado lo suficiente, probamos a darle un objetivo cercano pero obstaculizado por las paredes como en la figura 10, como podemos observar en la última figura 11. No es posible encontrar un plan debido a la resolución actual del mapa y su forma ya que excede la capacidad.

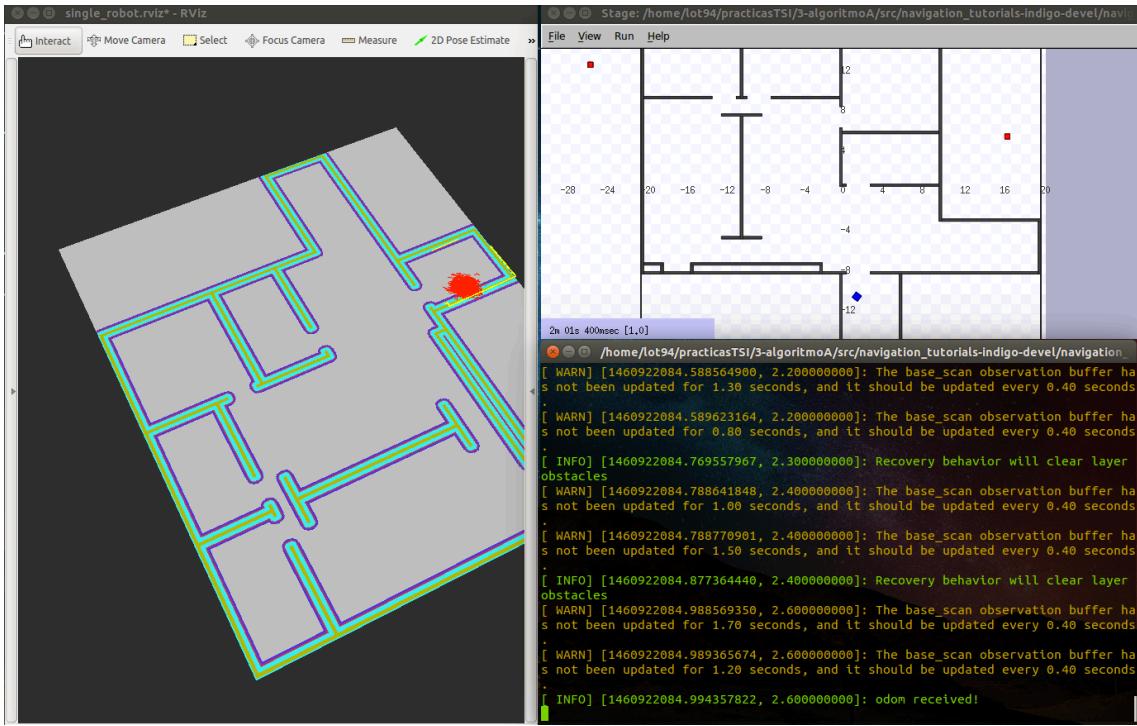


Figura 4: El robot en su pose inicial

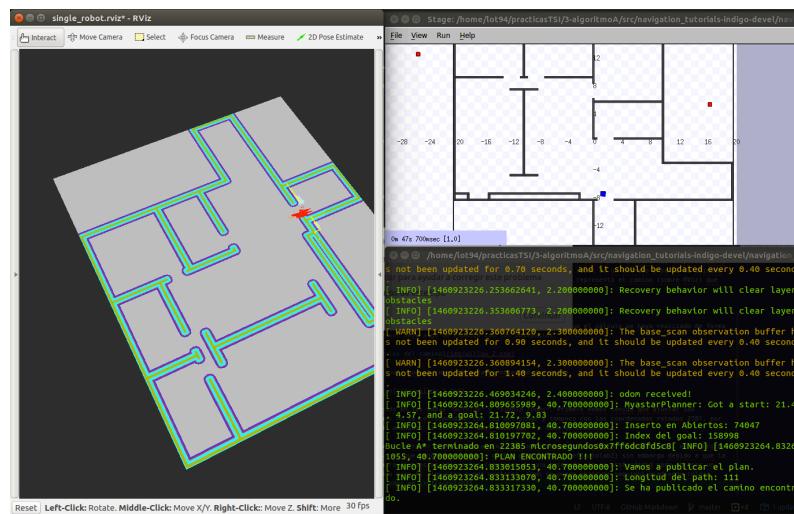


Figura 5: El robot se dirige el objetivo evadiendo celdas ilegales

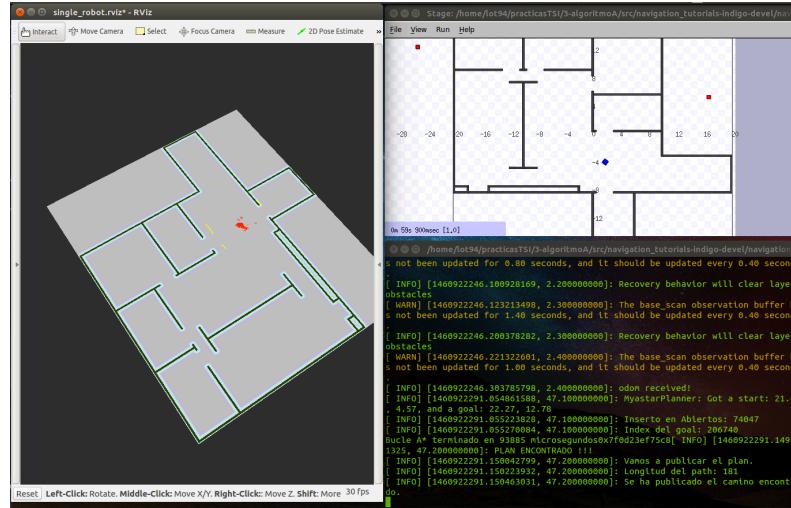


Figura 6: El robot alcanza el objetivo

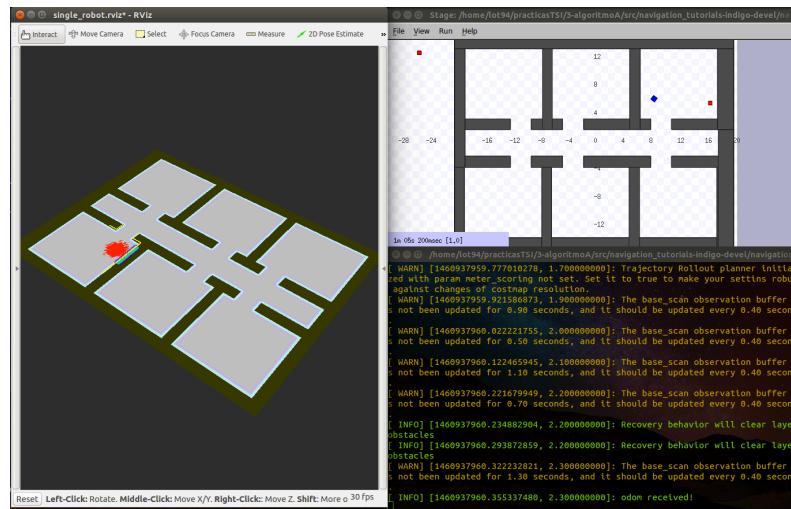


Figura 7: El robot en su pose inicial

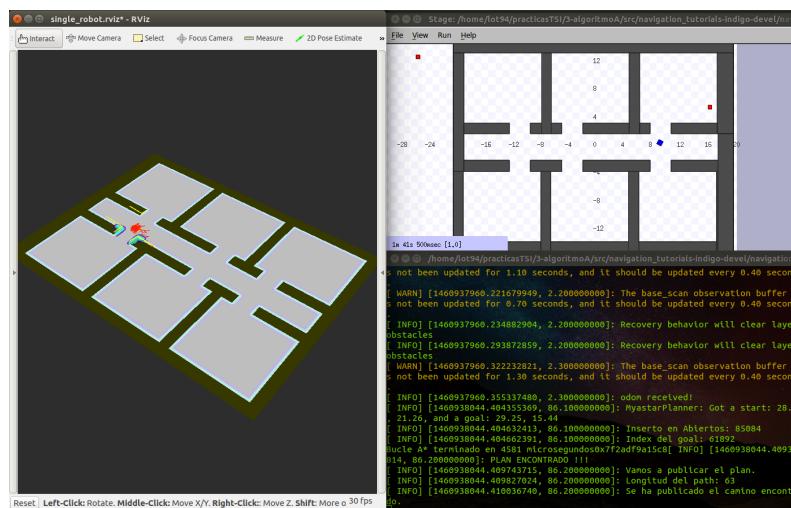


Figura 8: El robot alcanza un objetivo cercano

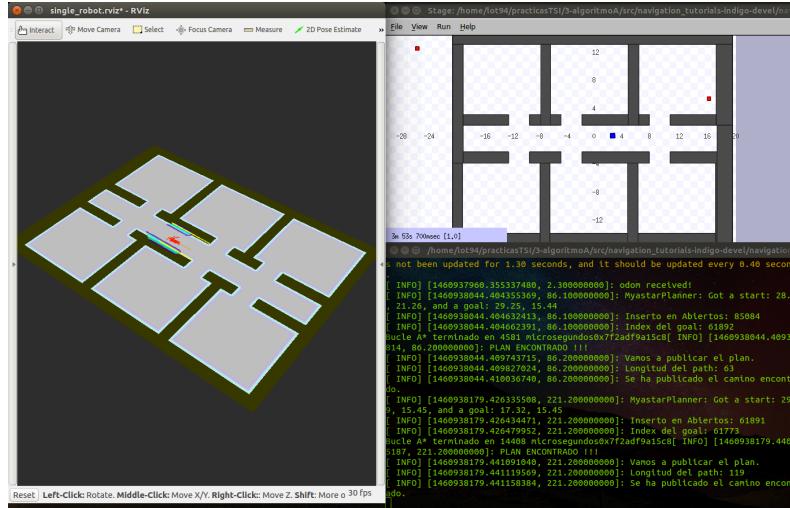


Figura 9: El robot alcanza el objetivo cercano

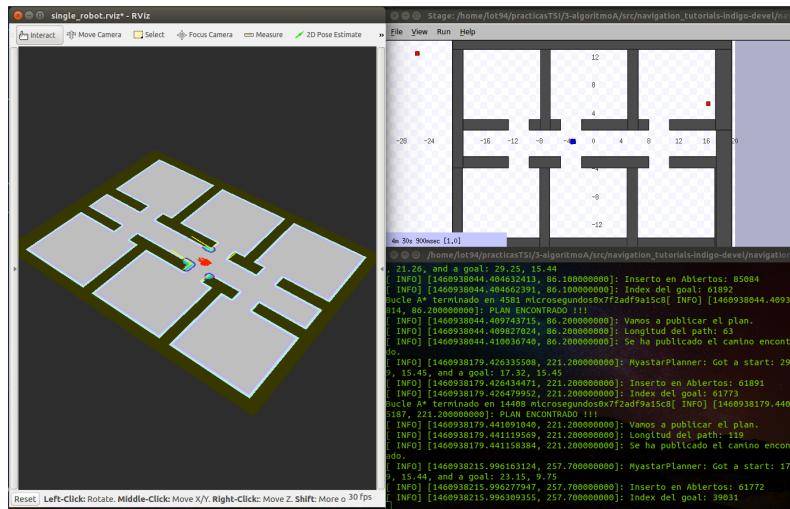


Figura 10: Se envia objetivo obstaculizado al robot

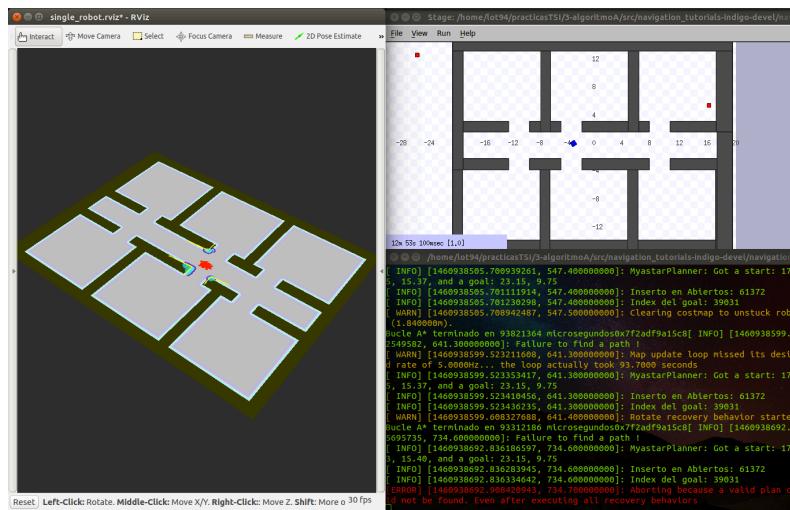


Figura 11: No se encuentra plan para el robot