Algoritmos de planificación de Robots

José Manuel Bueno Carmona
Department of Systems Engineering and Automation
University Carlos III of Madrid
Madrid, Spain
jobuenoc@ing.uc3m.es

Carlos Manuel Gómez Jiménez
Department of Social Robotics
University Carlos III of Madrid
Madrid , Country
cagomezj@ing.uc3m.es

***Abstract*—En este artículo se pretende mostrar la realización de varios algoritmos de path planning así como explicar las diferencias entre ellos y los puntos fuertes y débiles de cada uno.**

***Index Terms*—Path Planning, A star, Dijkstra, Visibility graph, Turtlebot, Gazebo, Robot Operating System.**

I. Introduction

En las últimas décadas gracias al desarrollo de la Robótica y especialmente en el sector de Robots móviles, surge la necesidad de poder ir desde un punto inicial a otro final de cualquier entorno en el menor tiempo posible y protegiendo la integridad física del dispositivo, esto es, esquivando obstáculos y con un camino ajustado lo máximo posible a la mínima distancia a recorrer. Hoy en día existen numerosos tipos de algoritmos para realizar la planificación, sin embargo, este documento se centra en la utilización de los métodos A* (A estrella) y Dijkstra.

A. Preparación del entorno.

Para poder realizar la búsqueda de un camino viable entre dos puntos es necesario disponer de un gráfico de la planta del entorno o *mapa*. Existen numerosas técnicas de escaneado pero no entran dentro del ámbito de estudio de este artículo. En este caso, al mapa se le ha aplicado una *descomposición en celdillas* lo que facilita su transformación a un archivo que trabaje con celdas como es el *.csv* y que es la base de trabajo en este documento. Utilizar esta metodología permite leer el mapa en forma de matriz y asignar valores binarios a los elementos en función del estado de ese espacio (obstáculo, no obstáculo).

B. Herramientas utilizadas.

El Sistema Operativo utilizado ha sido una distribución de Linux con la plataforma de comunicación ROS (Robot Operating System) la cual permite realizar una programación basada en nodos conectados entre sí encargados de funciones específicas. Esto es muy útil ya que es posible reutilizar los mismos nodos de una manera sencilla para distintos algoritmos. El simulador 3D utilizado es *Gazebo* con el modelo del Robot *turtlebot*. Dicho modelo incluye sus propiedades físicas y responde a las entradas de manera muy precisa a como lo haría el modelo real.

Así mismo, se ha utilizado la herramienta de ROS *Rviz* para poder visualizar si es necesario datos de mensajes en un entorno de tres dimensiones.

II. Diagrama de funcionamiento.

El flujo de mensajes se ha determinado de manera global como se muestra en la figura 1 en el que el papel principal lo juega el planificador, encargado de proporcionar un camino óptimo en cualquiera de los **mapas** que llame, teniendo como ayuda operaciones como la **detección de esquinas**. Una vez obtenido el camino, la parte encargada del movimiento del robot publicará mensajes indicando la velocidad lineal y angular objetivo del turtlebot, el cual estará recibiendo dichos datos con cada publicación.

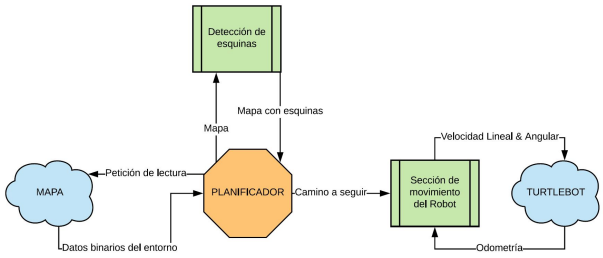


Fig. 1. Diagrama de funcionamiento general de planificación.

A. Path Planning

Como ya se ha comentado anteriormente, los algoritmos de búsqueda se centran en encontrar un camino desde un origen a un destino con la menor distancia posible. Para ello es necesario proporcionar cada paso o cada tramo que se debe realizar para seguir la ruta.

1) Celdas contiguas: El mapa se encuentra dividido en varias celdas que se pueden identificar como vacías, ya que el robot puede moverse libremente por ellas, y celdas ocupadas, en las cuales se encontrará un muro que impida el paso. Los algoritmos de celdas contiguas son aquellos que evalúen todas las conexiones de cada celda vacía con las celdas que se encuetran colindantes. Los movimientos entre celdas colindantes pueden ser de dos tipos. Por un lado están los desplazamientos en diagonal y por otro los desplazamientos en vertical u horizontal. Éstos se diferencian simplemente por la distancia en línea recta que tienen que recorrer.

a) Algoritmo Dijkstra: El algoritmo Dijkstra [1] se ocupa de optimizar al máximo la variable de coste del camino. El coste del camino es la distancia que se recorrería desde una

celda a la celda objetivo siguiendo el camino creado por el planificador. El cálculo del coste comienza en la casilla objetivo y va aumentando a medida que se aleja de la misma. Cuando se obtenga un valor en la casilla inicial, se puede empezar a elegir los puntos del recorrido más corto. Esto se realiza buscando cuál es la casilla con el resultado de coste menor entre las adyacentes a la actual. De esta manera se creará un camino hasta llegar al valor 0, que pertenece al valor de coste en la meta. Este algoritmo de búsqueda del camino más corto funciona especialmente bien cuando la zona a recorrer tiene características uniformes.

b) Algoritmo A:* El método A* [1] es un de los más utilizados de pathfinding. Se diferencia con el algoritmo anterior en que se utiliza más de una característica para definir el mejor camino posible. Se vale de la fórmula

$$f(n) = g(n) + h(n) \tag{1}$$

en nuestro caso, las funciones usadas son:

- *g(n)* :funcion de coste.
- *h(n)* :funcion de distancia real.

La función de distancia real nos proporciona un dato de longitud existente entre una casilla y el objetivo en línea recta, es decir, sin tener en cuenta los obstáculos que pueden existir por el camino. Con está variable se consigue un camino que tiene la intención de acercarse siempre lo máximo posible a su objetivo. Es necesario dar más prioridad a una de las funciones, puesto que en caso de que ambas tengan la misma importancia es mucho más probable que el planificador otorgue movimientos innecesarios. Cuando se han calculado ambas funciones, simplemente hay que sumarlas teniendo en cuenta la prioridad de ambas. El valor total, que de nuevo se incrementará en cuanto más se aleje del objetivo, es el valor que se requiere utilizar. Nuevamente, empezando por la celda inicial se elije el valor con menor total de las celdas adyacentes a la actual hasta llegar al punto requerido.

2) Punto a punto: Los procedimientos explicados anteriormente deben calcular las posiblidades en muchos puntos, o nodos, por los que puede pasar el robot. En esta sección los algoritmos van a calcular únicamente una serie de nodos de interés. Este método tiene dos beneficios:

1) Reducir el espacio recorrido por el robot.
2) Reducir el número de cálculos.

El método utilizado se denomina **Grafo de visibilidad** (Fig. 5) [1] y consiste en tener en cuenta como nodos únicamente las esquinas salientes de los obstáculos del mapa. Como resultado, el camino se creará siguiendo las conexiones entre todas las esquinas y los puntos final e inicial. Aquí es donde entra en juego la importancia de conocer la conectividad entre los nodos, es decir, saber si dos de los puntos elegidos tienen obstáculos entre ellos o no (ver II-A2a).

a) Detección de obstáculos.: La detección de obstáculos es imprescindible para saber si un tramo comprendido entre dos puntos como son `punto_actual` y `punto_siguiente` tienen conectividad directa sin una pared de por medio. Se ha implementado como un servicio

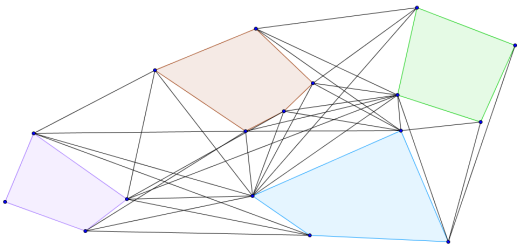


Fig. 2. Diagrama grafo de visibilidad.

de ROS, es decir, el nuevo diagrama de estructuración es el de la figura , donde el planificador proporciona el mapa que está procesando en ese momento y dos puntos del mismo. Dicho servicio contestará de manera afirmativa si se encontró un obstáculo y negativa en el caso contrario.

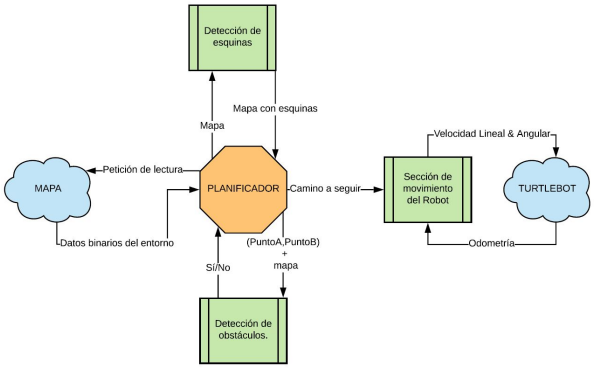


Fig. 3. Diagrama grafo de visibilidad.

El algoritmo desarrollado para encontrar obstáculos sigue los siguientes pasos (Figura 4):

- Convertir el mapa a una matriz para facilitar el trabajo con las celdas.
- Calcular el ángulo entre los dos puntos dados con sus coordenadas.
$$\theta = \tan\left(\frac{Filafinal - FilaInicial}{ColumnaFinal - ColumnaInicial}\right) = \frac{\triangle y}{\triangle x}$$
- Incrementar en intervalos pequeños la distancia en x desde el `nodo_actual` y hallar con θ la altura del triángulo rectángulo, es decir, hallar la fila donde se halla la línea que une ambos puntos para ese valor de columna.
- Evaluar el elemento de la matriz que se acaba de hallar para comprobar si es 0 (vacío) o 1 (pared). Al utilizar incrementos es necesario el uso de variables tipo *float*, por lo que lo más directo para hallar el elemento de la matriz es convertirlo a *int* con lo que se eliminaría la parte decimal y se tendrían en cuenta solo las partes enteras.
- Realizar esta comprobación a una altura superior y a otra inferior equidistantes. Esta es una de las propiedades más importantes del detector de obstáculos, ya que el robot

tiene un volumen por su forma física y existen casos donde chocaría al no haberlo tenido en cuenta por lo que dicha distancia entre la altura normal y sus paralelas será igual o mayor que el radio del robot.

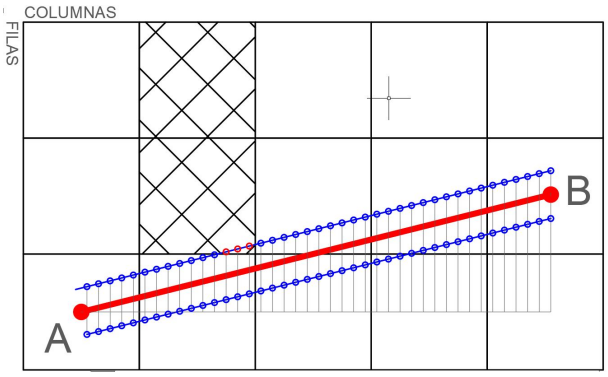


Fig. 4. Esquema detección de obstáculos.

En el ejemplo mostrado se comprueba que solamente con la línea que une los puntos **A** y **B** no daría un resultado adecuado ya que el robot colisionaría con el elemento del mapa (2,2).

b) Algoritmo Dijkstra: El algoritmo Dijkstra funciona igual sin importar el número de nodos. No obstante, en el caso anterior los nodos tenían siempre la misma relación de distancia entre ellos y solo era necesario calcular si había que moverse diagonalmente o vertical u horizontalmente. Ahora es muy importante saber el coste de movimiento de un vértice a otro cuando se consideran adyacentes entre ellos. Una vez se calculan todos los costes para llegar desde cualquier nodo al objetivo, se mueve el robot buscando el nodo visible con menos coste.

c) Algoritmo A:* El algoritmo A* cambia considerablemente cuando no se consideran todas las celdas como nodos. Anteriormente se calculaban los costes totales desde el objetivo en cada casilla para prevenir errores cuando se tienen varias casillas con resultados de total iguales. Pero en este caso no se necesitan hacer esos cálculos previos, pues se tendrá en cuenta solamente los costes que hay entre nodos adyacentes. Empezando el nodo inicial, se calcula qué nodo tiene un total menor, siendo el total la suma entre la distancia real y el coste con el nodo actual. Cuando se ha elegido el siguiente punto, se vuelve a hacer este método recursivamente hasta llegar a la meta. Debido a que no se calcula el coste total, es posible que sea necesario dar marcha atrás si el camino llega a un punto muerto o a un bucle.

B. Movimiento del robot.

La parte que se encuentra a más bajo nivel es la encargada de hacer que el robot se desplaze y gire en el entorno. Por defecto, el turtlebot utilizado tiene integración con ROS por medio de *topics* (mensajes entre nodos) específicos que se listan a continuación:

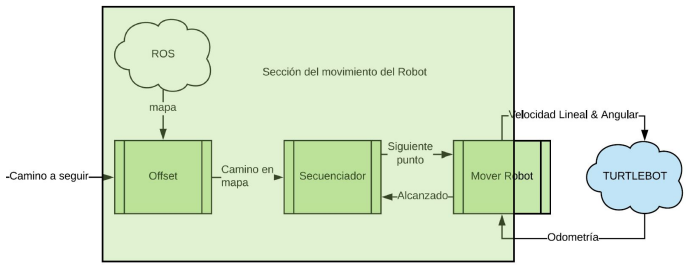


Fig. 5. Diagrama grafo de visibilidad.

- `geometry_msgs::Twist`. Tiene dos componentes : *linear* y *angular* que son los encargados de llevar la información relativa a ambas velocidades.
- `geometry_msgs::Point`. Se compone de x, y y z y se utilizan las dos primeras para designar puntos, ya que se considera un mapa bidimensional (X,Y).

En este artículo, en vez de utilizar un servicio de ROS que estuviera mandando la siguiente posición y esperando hasta que ésta sea alcanzada; se ha utilizado un método que incurre en menos problemas a la hora de defectos con las propiedades físicas del robot, a la vez que ayudará a optimizar el camino.

Dicho método consiste en la comprobación contínua de la odometría del robot, es decir, datos recibidos por los sensores del robot como los encoders de las ruedas, láser, etc con respecto a una posición final deseada. Para posicionarlo en una ubicación diferente a la actual basta con publicar unas coordenadas objetivo (topic : `goal_pose`) que se asignarán directamente a la variable a comprobar dentro del ciclo del nodo. Más concretamente, se tiene en cuenta un intervalo angular dentro del cual se puede estar sin corregir el ángulo actual ($\pm 2^o$), en caso de no tener que realizar ninguna correción se pasa a la fase de identificación de posición en la cual se distinguen tres posibles casos:

1) En medio del camino sin acercarse al siguiente punto.
2) En medio del camino dentro de un radio determinado desde el siguiente punto.
3) Al final del camino con distancia casi nula al nodo final.

Con respecto a mejorar los problemas físicos comentados anteriormente, se quiere indicar que en el caso de ordenar al modelo un movimiento totalmente recto de un punto a otro, el robot empezará a desviarse de la ruta causando inconvenientes en la navegación. Gracias a esta forma de navegar, en caso de que el ángulo se encuentre fuera del intervalo simplemente se habrá de girar en uno u otro sentido para estar de nuevo orientado correctamente hacia el siguiente punto.

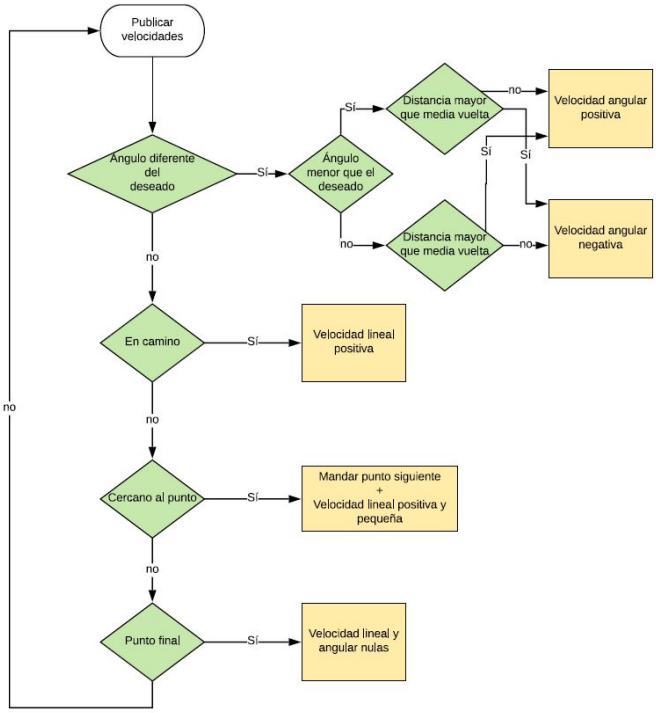
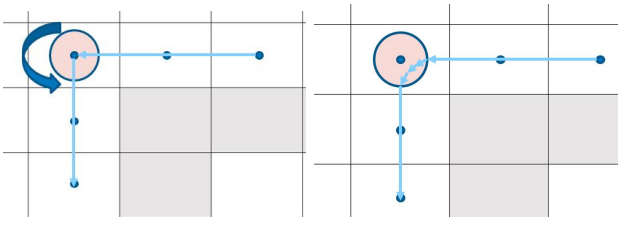


Fig. 6. Diagrama de flujo movimiento del Robot.

Como se puede observar en el diagrama de la figura 6, se tiene en cuenta un área alrededor de la posición del siguiente nodo hacia el que moverse. Al mandar la siguiente posición al Robot ayudándolo con una pequeña velocidad lineal, resultará en un movimiento de giro mucho más **suave** (smoothing) y sin necesidad de hacer **paradas** en cada transición de tramos del camino, ya sean con celdas contiguas o puntos separados. Es fácil pensar que ésto puede ser un elemento más que ayude a poder incrementar la velocidad media del robot encontrando la relación entre las velocidades linear y angular para que se produzca un giro lo más rápido posible teniendo en cuenta las inercias, así como las fuerzas centrípeta y centrífuga.



(a) (b)

Fig. 7. Comparación de giro sin suavizado y con suavizado.

III. Conclusiones

Después de implementar el código y comprobar su funcionamiento en diversos mapas, se puede detectar una gran diferencia entre el método punto por punto y el método de grafos de visibilidad. En el segundo método, el robot no debe recorrer tantas celdas vacías para llegar de un punto a otro y por lo tanto puede hacer ese mismo camino completamente en línea recta. La mejoría es muy destacable cuanto más grande es el mapa, pues ocurrirá en más ocasiones que el robot se dirija solamente al objetivo sin importarle cuantas celdas vacías recorre.

Por otro lado, cuando se comparan los algoritmos Dijkstra y A* también es visible que el primero realiza un recorrido un poco más corto. Esto se debe a que el algoritmo A* funciona mejor en mapas donde las condiciones no son uniformes y la segunda variable a tener en cuenta actúa para corregir esta uniformidad en el camino seleccionado. Por ejemplo, si existen diferencias de altura en distintos puntos y, por lo tanto, el robot debe variar su velocidad para que ésta sea adecuada según la inclinación en la trayectoria. Por lo tanto estaríamos usando, aparte de un valor de distancia, una variable de velocidad para elegir el camino. Para determinar un ejemplo más usual, si por ejemplo se quisiera aplicar este algoritmo para planificadores de trayectos de un aparato GPS para un automóvil, se podrían tener dos variables en cuenta: El camino más corto junto a la velocidada máxima de la vía, ó velocidad adecuanda de la vía si tenemos además información del clima y del tráfico en ese punto.

Hay que destacar la importancia en el movimiento del robot para que el recorrido se realice lo más rápido posibles. Evitar que el robot pare solamente para hacer un giro o hacer que el robot pueda moverse a velocidades más elevadas si no tiene que hacer una curva son medidas que son tan importantes como tener un buen planificador.

References

[1] Universidad Carlos III de Madrid. Planificación de tareas y movimientos de robots. Técnicas de planificación de trayectorias I: Classical approaches. 2018