FIB

Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Proyecto final robótica móvil

Introducció a la Robòtica

Autores:

Pablo González Monfort, **Adrià Cantarero Carreras**, **Violeta Bonet Vila**

15 de abril de 2023

Índice

1. Introducción

Este documento resume el trabajo realizado en el proyecto final de robótica móvil. Puesto que el proyecto se ha desarrollado en un ámbito de aprendizaje, se intentará reflejar no solo el resultado final, sino también el proceso: los retos, las soluciones y el conocimiento adquirido durante todo el proceso, usando un punto de vista más subjetivo.

2. Desarrollo

2.1. Planteamiento

En una primera reunión de grupo, planteamos hacer el proyecto propuesto por el profesorado de hacer que el robot salga de un laberinto. Sin embargo, nos dimos cuenta de que para cumplir esa tarea basta con seguir una pared (izquierda o derecha) durante todo el camino hasta eventualmente encontrar la salida, por lo que lo rechazamos en busca de un problema más complejo.

Entonces surgió la idea de hacer que el robot llegara de un punto A a un punto B esquivando los obstáculos que encontrara a su paso. Parecía una idea simple, y nos vinieron a la mente ideas sencillas como que al detectar un obstáculo lo rodeara para seguir su camino. Sin embargo, tras una reflexión más profunda, entendimos que el problema era bastante más complicado, ya que no todos los obstáculos se pueden rodear. De hecho, un laberinto se podría considerar como un obstáculo (o un conjunto de ellos), por lo que nuestra solución debería ser capaz de solucionar también el laberinto del que se había hablado anteriormente. Por otra parte, teniendo el laberinto en mente, el hecho de no poder usar el "truco" de seguir una pared (ya que nuestra solución pretendía ser más general) sugería que habría que dotar al robot de algún tipo de memoria para poder recordar por dónde podía pasar y por dónde no en caso de tener que rehacer el camino al encontrar una ruta sin salida.

Finalmente determinamos que los pasos que debería seguir nuestro programa serían los siguientes:

1. Creación de un mapa a partir de los datos recibidos del láser por el topic /scan.
2. Uso de un algoritmo de *pathfinding* para encontrar el camino hasta el objetivo, como puede ser el de A*
3. Mover el robot por el camino

Estos pasos los debería realizar en bucle, actualizando el mapa a medida que se iba moviendo hasta llegar al objetivo.

NOTA: *Tras este planteamiento inicial, nos dimos cuenta de que la fecha de entrega del proyecto publicada en la web era la fecha límite de entrega, y que los proyectos entregados en esa fecha eran evaluados sobre 7 y no sobre 10. Dada la magnitud que empezábamos a intuir que tendría nuestro proyecto, decidimos hacerlo lo más rápido posible y acabarlo antes de la fecha marcada para la evaluación sobre 10, por lo que todo el desarrollo se ha hecho en un periodo de cinco días.*

2.2. SLAM

La parte más compleja del proyecto iba a ser sin duda la primera. Lograr crear un mapa a partir de un vector de distancias en los 360 grados parecía una tarea prácticamente imposible. Había que usar las direcciones junto con las distancias para estimar las coordenadas de cada punto detectado, y luego almacenar los datos en un array bidimensional de booleanos que representara nuestro mapa.

Aunque lográsemos programar correctamente el algoritmo, probablemente el mapa resultante tendría una gran imprecisión y sería muy susceptible a errores de lectura. Eso, unido al poco tiempo que teníamos, hizo que nos decantásemos por usar slam_gmapping [4], un paquete de ros que se encarga precisamente de mapear el terreno (probablemente con mucho más acierto de lo que podríamos hacer nosotros) y visualizar el mapa resultante en rviz, un programa que permite leer y representar de una forma visual información emitida por el robot a través de topics.

2.3. El problema de buscar el camino más corto

Mientras tomábamos esta decisión, otro problema vino a nuestras mentes: el algoritmo que pretendíamos usar para encontrar el camino hasta el destino, probablemente buscase una ruta lo más corta posible, y eso implicaba que a la hora de girar en una esquina, trazaría el camino pegándose a la pared para acortar distancia. Esto suponía un problema porque el robot no es simplemente un punto en el espacio, sino que tiene una forma aproximadamente cilíndrica con un radio de unos 10 a 15 centímetros (medida estimada) y al girar una esquina de forma tan apurada se chocaría con la pared.

La solución que pensamos a este problema fue crear un mapa modificado, haciendo que cada punto del mapa original se convirtiera en un círculo del mismo radio que el robot en el nuevo. De esta forma, suponiendo que el punto de referencia del robot se sitúa en el centro del mismo, cuando ese punto de referencia estuviera pegado a la nueva pared engrosada, el borde del robot estaría prácticamente tocando la pared real.

Tras estos cambios, los pasos a seguir para hacer el proyecto se convertían en los siguientes:

1. Lectura del mapa creado por slam_gmapping
2. Engrosamiento de las paredes del mapa y almacenamiento del nuevo mapa
3. Uso de un algoritmo de *pathfinding* en el mapa de paredes engrosadas para encontrar el camino hasta el objetivo, como puede ser el de A*
4. Mover el robot por el camino

2.4. Lectura del mapa creado por slam_gmapping

Tras una breve investigación, descubrimos que slam_gmapping publica los resultados de su mapeo al topic /map, que luego es leído por rviz. El mensaje publicado es de tipo nav_msgs/OccupancyGrid [2], que contiene un header, un MapMetaData llamado info y un array de tipo int8 llamado data.

De toda esta información, los datos útiles para el proyecto son:

- int8[] data: El mapa de ocupación en sí, representado como un vector de enteros que van del 0 al 100 según sea la probabilidad de ocupación (se hablará de este vector más adelante).
- uint32 info.height: La altura del mapa de ocupación, medida en celdas.
- uint32 info.width: La anchura del mapa de ocupación, medida en celdas.
- float32 info.resolution: La medida del lado de cada celda (las celdas son cuadradas), medida en metros.
- geometry_msgs/Pose info.origin: Un objeto que almacena la posición en metros de la celda (0, 0) del mapa.

Para leer el mapa, se creó un subscriber que escucha en el topic /map y llama a una función llamada mapCallBack. Esta función hace tres cosas:

1. Vacía el array grid, que es un array de boolenaos en el que se guardará el mapa de ocupación modificado. Para esto hace uso de la función memset, que parece ser la más eficiente para esta tarea.
2. Llama a la función enlargeWalls (encargada de engrosar las paredes), pasando como parámetro el mapa.
3. Llama a la función mapPublication, que se encargará de la publicación del mapa para que pueda ser leído por rviz.

2.5. Engrosamiento de paredes

El primer paso para crear el algoritmo de engrosamiento de paredes fue interpretar el mapa original. El mapa que genera slam_gmapping es un array unidimensional, pero usando los valores de altura y anchura proporcionados como metadata en info, se puede interpretar como una matriz bidimensional. Tras investigar en internet, descubrimos que el mapa se guarda por filas, si consideramos que una fila es un valor fijo de y con la x variando. Esto significa que data[1] será la coordenada en celdas (1, 0) y que data[width] será (0, 1).

El mapa que genera slam_gmapping inicialmente, es un mapa cuadrado de 384 x 384 celdas, con una resolución de 0.05, o lo que es lo mismo, cada celda tiene 5 centímetros de lado. Esto significa que el mapa generado en un inicio es de aproximadamente 20 metros por lado. Luego, si el robot

se mueve y se sale de este mapa, el mapa se amplía. Para no complicar más el proyecto, decidimos trabajar en este mapa inicial, es decir con la restricción de que no podría salir de este marco.

Ya sabiendo cómo interpretar el mapa, nos pusimos a programar la función que engrosaría las paredes. La forma de cumplir esta tarea es la siguiente. Se recorre todo el array unidimensional en el que está almacenado el mapa original, y si la probabilidad de ocupación de la celda es mayor al 50 %, se considera como ocupada y se llama a la función updateCircleInGrid. Esta función, como su nombre indica, actualiza la variable grid (recordemos que es el array de booleanos en el que se guarda el mapa modificado) para poner como true todas las celdas que estén dentro del círculo que tiene como centro las coordenadas de esa celda. Para ello, define las variables top, bottom, left y right que serán los topes del cuadrado en el cual está inscrito el círculo. Luego, para todas las celdas dentro de ese cuadrado, usando la función insideCircle, se calcula la distancia entre ella y el centro, y si es menor al radio del círculo, se actualiza a true esa celda en el array grid. Puesto que la función insideCircle va a ser llamada muchas veces durante el programa, se ha implementado enteramente usando números enteros para que las operaciones sean más rápidas.

De esta forma, queda almacenado el mapa nuevo de una forma más clara y comprimida que el original (ya que en vez de usar enteros representando probabilidades se usan booleanos, que ocupan menos).

2.6. Algoritmo A*

Para definir el camino que debería seguir el robot, decidimos usar A*. Para ello creamos la clase llamada AStarCell que representa las celdas del mapa que son usadas por A*. Los objetos de esta clase tienen:

- Las coordenadas de la celda en cuestión.
- La celda a partir de la cual se ha explorado o celda padre.
- El coste hasta el origen o g_cost, que es la suma del g_cost de la celda padre más el coste del movimiento (se explica más adelante).
- El coste heurístico hasta el destino o h_cost (se explica más adelante)
- Y el f_cost, que es la suma de g_cost y h_cost.

Puesto que en una clase en C++ no se puede crear un atributo de la propia clase, se guardó parent como un puntero a la celda padre.

El algoritmo A* funciona como una frontera de terreno explorado que se va expandiendo intentando llegar al objetivo. Para hacer esto, crea dos listas, una llamada open, que almacena las celdas de la frontera, es decir aquellas que se han explorado pero no expandido, y otra lista llamada closed, que almacena las celdas interiores del área explorada, aquellas que ya se han explorado y expandido. Inicialmente, el nodo de origen se sitúa en la lista de open y se inicia un bucle que se repetirá hasta llegar al destino o haber explorado todo. En este bucle, se escoge la celda de la frontera (la lista open)

con menor f_cost, se cambia de open a closed y se expande. Expandir una celda hace lo siguiente: Para todos los vecinos de esa celda que no estén ya en closed, si no han sido explorados, se añaden a open, y si ya están en open, pero tienen un g_cost mayor que el que podría tener con la celda que se está expandiendo como parent, se actualiza el parent y el g_cost de esa celda vecina, actualizándose también el f_cost.

Cada vez que se añade una celda a la lista de open, se calculan sus costes. Para ello, primero se define el coste de mover el robot de una celda a otra. Como nuestro programa permite el movimiento diagonal, establecimos que el movimiento ortogonal tendría un coste de 10 y el diagonal de $\sqrt{2} \cdot 10$ que es aproximadamente 14. Usando esos valores, para calcular el g_cost se suma al g_cost del parent 14 o 10 dependiendo de si la celda está en diagonal o en horizontal o vertical con el parent. El cálculo del h_cost es más comprometido, ya que se pueden aplicar muchas heurísticas diferentes, y de su elección dependerá en gran medida la capacidad del algoritmo [1]. La heurística que escogimos fue la de la distancia diagonal u octal, que calcula la distancia al objetivo con movimientos diagonales y ortogonales.

Relacionado con el algoritmo de A*, cabe mencionar que puede darse el caso de que el robot se mueva de manera imprecisa y se "meta" dentro de la pared, en cuyo caso, el algoritmo no encontraría el camino. Para solucionar este problema, creamos una función que encuentra la celda libre más cercana buscando en ocho direcciones.

El algoritmo, al acabar, vacía el vector path, que es donde se almacena el camino, y lo llena con el nuevo camino. Esta operación no es completamente trivial, ya que en un inicio el camino solo está definido por la celda destino y su parent, que a su vez contiene otro parent, y así hasta el origen. Para almacenar el camino en el orden correcto (aunque hay muchas otras maneras) se usa una función recursiva que se llama a sí misma hasta llegar a la celda origen y luego añade las celdas al path en orden.

2.7. Punteros y direcciones de memoria

El programa no funcionaba. Después de resolver todos los errores de compilación, se ejecutaba, pero no parecía hacer nada.

Estuvimos haciendo pruebas, añadiendo comandos de ROS_INFO para intentar detectar qué era lo que pasaba, y descubrimos que el programa se quedaba estancado en la función recursiva de setPath, la encargada de actualizar la variable path tras el algoritmo. Revisamos largo y tendido esa función, pero es una función bastante simple y no veíamos dónde podía estar el error. Finalmente decidimos imprimir toda la información posible por pantalla, y fue entonces cuando descubrimos que los punteros de las celdas que hacían referencia al parent, a partir de una cierta celda, eran todos iguales.

Tras horas de pruebas, entendimos cuál era el problema. A la hora de expandir una celda de la lista open, lo que hacía el programa era obtener la dirección de memoria de esa celda y guardarla en una variable para usarla después a la hora de establecer los atributos parent de las celdas vecinas. Sin embargo, luego el objeto cambiaba de la lista open a la closed, por lo que la dirección de memoria

ahora contenía otro objeto diferente. Si daba la casualidad de que ese objeto tenía como parent esa misma dirección de memoria, entraba en un bucle infinito.

Para solucionar este problema, cambiamos el vector closed por un array global, para asegurarnos de que no cambiara su dirección de memoria (ya que por cómo funciona std::vector, no estaba asegurado que tuviera la misma dirección de memoria en un vector) y nos ocupamos de guardar la dirección de memoria de la celda después de añadirla a closed.

2.8. Visualización y cambio en el peso de la heurística

Por fin el programa para construir el camino funcionaba correctamente. Para poder visualizar que el camino era el correcto, creamos un publisher que publica al topic /robot_path el camino como objeto nav_msgs/Path [3]. Buscamos cómo visualizar objetos path en rviz y comprobamos que el camino se construía correctamente.

Sin embargo, el algoritmo tardaba mucho en encontrar el camino, por lo que creamos una función diseñada únicamente para pruebas, que cambiaba a ocupadas todas las celdas exploradas por el algoritmo A*, es decir todas las celdas en closed. De esta forma, podíamos visualizar en tiempo real cómo se expandía el algoritmo de A*. La frontera de celdas exploradas se expandía de una forma muy homogénea en todas las direcciones, lo que hacía que tardara más en encontrar el camino al destino. Buscamos cómo mejorar el rendimiento del algoritmo, cómo hacer que se centrara más en el objetivo, y descubrimos el concepto de peso de la heurística.

El peso de la heurística es algo así como la atracción que genera la celda destino en la decisión de celdas a explorar por el algoritmo. Simplemente es un valor que se multiplica por el h_cost a la hora de calcular el f_cost, dándole así más valor a la heurística. Al incrementar este valor, se reduce el tiempo de cómputo del algoritmo, pero aumenta la longitud de éste.

Al incluir este valor en nuestro cálculo, mejoró mucho el tiempo que tardaba en computar el camino.

2.9. odomCallback, AsyncSpinner y mutex

Trazado ya el camino a seguir, solo quedaba mover el robot. Para ello, tendríamos que usar en múltiples ocasiones la posición y orientación del robot, por lo que se creó un subscriber que leyera la odometría y almacenase la posición en una variable current_pose. La función odomCallback es la que recibe el mensaje de la odometría y está copiada del código cedido en el enunciado de la práctica 7 de robótica móvil.

Dado que nuestro código contenía varios subscribers y algunos realizaban operaciones pesadas, pensamos que quizá eso podría generar algún cuello de botella ralentizando el programa. En efecto, tras una breve investigación, nos enteramos de que el spin de ros que habíamos usado hasta ese momento procesa los callbacks de uno en uno, y eso supone que si un callback tarda mucho en acabar, el siguiente llegará tarde. La forma de solucionar esto fue usando un spin que procesara los callbacks en paralelo, en múltiples hilos de procesamiento. El spinner escogido, a pesar de que hay

más spinners multihilo, fue AsyncSpinner [5].

Ahora, parte de nuestro programa funcionaba en multihilo, y eso implicaba nuevos retos. Si dos hilos de procesamiento intentan acceder de forma asíncrona a una misma variable, se pueden originar fallos. La solución a esto es implementar un sistema de "semáforos" que detenga los hilos entrantes mientras un hilo está accediendo a una variable. Por suerte, este sistema viene ya implementado en C++ y se puede trabajar con él a través de mutex.

Cada vez que se quiere acceder a una variable compartida, se bloquea el mutex, haciendo que ningún hilo pueda entrar en una región de código protegida por ese mismo mutex. Cuando se ha acabado de acceder a la variable, se desbloquea y entra el siguiente hilo. En nuestro código, creamos tres mutex, uno para el path, otro para la current_pose y otro para grid, y añadimos bloqueos y desbloqueos en todas las regiones de código en las que se accedía a esas variables.

2.10. Movimiento del robot

Ahora sí, nos pusimos finalmente a crear el sistema de movimiento del robot. El proceso era simple: se recorrería el camino y, para cada punto, el robot tendría que rotar hasta encarar el punto y avanzar hasta situarse sobre él.

Para implementar esto, se definió un margen de precisión en la rotación, que es la distancia en radianes a partir de la cual se considera que el robot está en la posición adecuada. Para la parte de avanzar, tuvimos en cuenta que si la rotación había sido por algún motivo imprecisa, no avanzara hasta llegar al punto de destino (aunque tuviera también una distancia de margen) porque podía pasar al lado y no alcanzarlo nunca. Por ello, el robot se detiene cuando la distancia hasta el destino comienza a aumentar en vez de disminuir, y por tanto está en el punto más cercano.

A mayores, como los movimientos eran bastante imprecisos incluso en un entorno de simulación como gazebo con velocidades relativamente bajas, pensamos que podía deberse no a la velocidad, sino a la aceleración. Al decirle al robot parado que se ponga a una velocidad x, le estamos pidiendo a las ruedas que cambien instantáneamente a cierta velocidad, pero deben vencer a la inercia, y si no tienen un agarre perfecto al suelo pueden patinar. Por ello, se implementó en el movimiento del robot un sistema de aceleración y deceleración.

La aceleración la hace basándose en el tiempo, según una variable constante (acc_time) que indica en cuánto tiempo debe ponerse a velocidad máxima (linear_vel, angular_vel).

La deceleración, en cambio, la hace basándose en la distancia que le queda hasta llegar al objetivo. A partir de una distancia determinada, comienza a decelerar hasta parar por completo en la posición de destino.

2.11. Automatización del arranque

El proyecto, durante su desarrollo se lanzaba siguiendo los siguientes pasos:

1. Arrancar roscore
2. Lanzar gazebo
3. Lanzar slam_gmapping
4. Cambiar el topic del mapa que lee rviz de /map a /big_wall_map (el topic en el que se publica el mapa modificado para poder visualizarlo)
5. Añadir un path a la visualización de rviz y poner como topic /robot_path (el topic en el que se publica el camino trazado para poder visualizarlo)
6. Lanzar nuestro programa con las coordenadas de destino hard-codeadas en el código.

Para hacer todo el proceso de una forma más rápida, y poder pasar las coordenadas de destino en el roslaunch, vimos que era necesario cambiar el archivo de launch. Los cambios fueron:

- Definir dos argumentos para leer las coordenadas en el comando roslaunch (dest_x y dest_y).
- Definir dos parámetros iguales que los argumentos para poder recuperar los valores desde el programa (ya que no se pueden recuperar los argumentos)
- Incluir el archivo launch de slam_gmapping para que se lance automáticamente, cambiando un parámetro para que no abra rviz.
- Abrir rviz cargando una configuración personalizada para que visualice los topics deseados.

3. Resultado final

Para usar la versión final del proyecto, se debe, con roscore lanzado, o bien encender un robot y conectarlo o lanzar un entorno de simulación como gazebo. Hay que tener en cuenta que al encender el robot establece sus ejes de coordenadas, siendo el eje de las x positivas la dirección que está encarando y el eje de las y positivas la dirección que sale hacia el costado izquierdo. Hecho esto, debemos determinar a qué punto queremos que vaya el robot, y especificarlo según su eje de coordenadas en metros. Por ejemplo, si queremos que vaya a un punto que está situado a dos metros en frente del robot y uno hacia la derecha, las coordenadas serían (2, -1). Finalmente, debemos lanzar el proyecto usando el archivo atob.launch y especificando las coordenadas de destino en los argumentos dest_x y dest_y. Un ejemplo de comando sería el siguiente:

- roslaunch robofinal atob.launch dest_x:=2 dest_y:=-1

Tras lanzarlo, se abrirá rviz y podremos ver el mapa con las paredes engrosadas y el camino que pretende seguir el robot en rojo. El tamaño de las paredes no corresponde exactamente con el del robot, sino que es más grueso, para darle más margen.

El programa ejecuta un bucle hasta que se para el programa manualmente, llega al destino, o determina que no tiene forma de llegar. En este bucle, lo que hace es llamar al algoritmo A* usando la posición actual del robot, publicar el camino trazado y moverse a la posición del camino especificada por la variable pathStepIndex, fijada en 3. Se ha implementado que avance a pasos más grandes para que el movimiento sea más fluido.

Este proyecto comenzó con la intención de hacer algo un poco más complejo que los proyectos propuestos en el enunciado, pero la complejidad escaló rápidamente hasta desviarse completamente del nivel que pedía la práctica. Eso, sumado a la falta de tiempo, ha hecho que el desarrollo sea difícil y poco organizado, pero aun así ha supuesto un enorme aprendizaje en C++ y en ROS.

Referencias

[1] *A*'s Use of the Heuristic*. URL: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[2] *Nav_msgs/OccupancyGrid*. URL: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html.

[3] *Nav_msgs/Path*. URL: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Path.html.

[4] *Wiki*. URL: http://wiki.ros.org/slam_gmapping.

[5] *Wiki*. URL: http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning.