PRÁCTICA 2

Grupo TSI1 (Miércoles 17:30-19:30)

Resumen

Partiendo de la implementación proporcionada se ha realizado un A* básico y a partir de éste se han realizado dos mejoras con el fin de reducir el tiempo necesario para trazar el plan. Modificar las estructuras de datos (listas) por otras tales como colas de prioridad y árboles y, principalmente, la implementación de la técnica ***Jump Point Search***, que reduce significativamente el número de vecinos generados por cada uno de los nodos.

David Criado Ramón

D.N.I.:

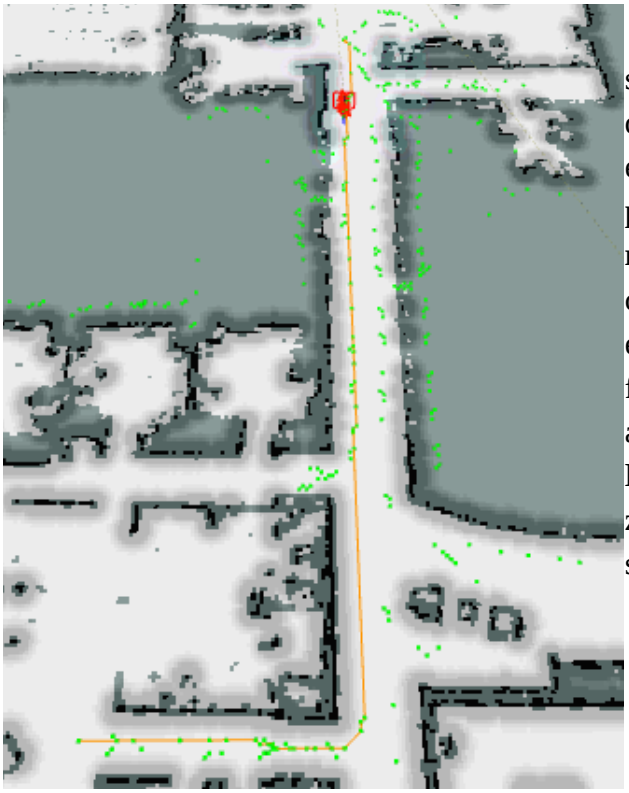
1. Implementación del algoritmo A* básico.

Partiendo del código proporcionado en el paquete my_astar_planner la implementación del algoritmo A* para heurísticas monótonas sin ninguna mejora es bastante simple. Sacamos de la lista de abiertos el nodo con menor f y lo metemos en cerrados y en el método que se encarga de añadir un nuevo vecino defino como g el coste del padre más la distancia euclídea del padre al hijo y como h la distancia euclídea entre el nodo actual y el nodo objetivo. Haciendo estos pequeños cambios disponemos de un A* que ya es mejor que la búsqueda en anchura inicial.

Para asegurarnos de que el camino sea el más seguro he añadido un nuevo parámetro que indicará el máximo valor que puede tener una casilla en el costmap para no ser considerada un obstáculo. Este valor lo inicializo a 10 y si el algoritmo no es capaz de trazar un camino con esta restricción en el próximo intento de crearlo aumentará este umbral en 10 más así hasta llegar a 127 empezará a subir de 1 en 1 con el fin de intentar salir del lugar en el que probablemente ya ha colisionado. Esta técnica hace que caminos muy arriesgados serán explorados como última opción (haciendo también que el tiempo que tarde para estos casos sea bastante más elevado).

Otra mejora que se ha hecho para evitar la colisión con objetos es aumentar ligeramente el parámetro *inflation_radius* de *costmap_common_params.yaml* de 0.55 a 0.70 aumentando así ligeramente la distancia en la que realiza la inflación de los obstáculos y reducir el parámetro *const_scaling_factor* de 10.0 a 9.0 para que haya mayores valores en la zona de inflación y facilitar que esta estrategia funcione bien y evitar considerablemente que el planificador local a veces colisione rotando con la esquina de algún obstáculo.

Observemos cómo se comporta con respecto a los obstáculos (téngase en cuenta que en esta experimentación ya están implementadas las mejoras realizadas).



En esta experimentación partimos del puntos superior hacia el punto inferior a la izquierda. Puesto que la heurística es la distancia euclídea vemos que el algoritmo A* nos lleva lo más a la izquierda posible en el camino para llegar de la forma más rápida. Además, para evitar colisiones, sólo son casillas libres aquellas cuyo valor del costmap sea estrictamente menor que 10. Por tanto, para llegar de forma segura, vemos cómo no sigue una recta hacia abajo durante el pasillo en el camino sino que hace ligeros movimientos en zig-zag (recomendable hacer zoom para apreciarlo) para garantizar un camino seguro.

2. Mejoras realizadas sobre el algoritmo inicial

Vamos a subdividir las mejoras realizadas en dos partes:

2.1 Mejoras basadas en la modificación de estructuras de datos

La primera idea para mejorar la eficiencia del algoritmo que realicé fue cambiar las estructuras de datos. Mientras que es cierto que producen una mejora considerable del tiempo que se tarda en realizar el algoritmo estas mejoras no fueron suficientemente buenas para resolver los casos más complejos.

La lista de abiertos que en principio erá una única lista pasó a ser dos cosas. Una cola con prioridad para garantizar sacar rápido (orden costante) el nodo con un mínimo de f y una inserción en orden logarítmico. Para comprobar la pertenencia a abiertos utilizo un conjunto (set) en el que la clave es el índice de la celda permitiéndonos también reducir el tiempo a un orden logaritmo. En la lista, si hubiese sido bien realizado, todas estás operaciones excepto insertar que hubiera sido constante, habrían sido de orden lineal con lo que conseguimos una cierta mejora. La mejora en tiempo en abiertos viene a costa de gastar cierta memoria extra para almacenar de nuevos los índices en una nueva estructura.

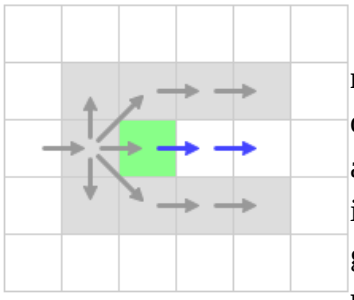
La lista de cerrados viene ahora dada por un map donde la clave es el índice de la celda y la descripción es el struct que define a esa celda. Así pues volvemos a hacer que, aunque la inserción sea más costosa (pasa de orden constante a orden logarítmico), encontrar una celda concreta pase de orden lineal a orden constante).

Una mejora que se notaría mucho, pero que no he realizado puesto que en la versión de C++ que funciona correctamente con la distribución Indigo no está disponible, son las versiones en tabla hash del set y el map propuestos para comprobar la pertenencia. *Nota: A partir de la distribución Kinetic, si que podrían ser usados activado C++11 en el CMakeList y aumentarían considerablemente la velocidad en el caso de que haya muchos nodos.*

2.2 Jump Point Search (JPS)

Uno de los principales problemas del algoritmo A* en mapas grandes, es la gran cantidad de nodos a explorar. La idea de esta técnica relativamente moderna (2011) es reducir el número de nodos a explorar modificando la forma de crear vecinos. Mientras que la primera vez que abrimos un nodo generamos todos sus adyacentes. A partir de la siguiente tenemos en cuenta el nodo actual y su padre y seguimos explorando en la dirección determinada por el vector que va desde el nodo padre hasta el hijo.

Diferenciamos dos tipos de reglas, las aplicadas en movimientos de línea recta (arriba, abajo, izquierda, derecha) y las realizadas en diagonal. Para facilitar la explicación vamos a acompañarnos con imágenes.



En este caso nuestro padre era el nodo gris a la izquierda del verde y el nodo a explorar es el verde. En este caso estamos explorando hacia la derecha, aunque lo veamos hacia la derecha, este mismo caso se puede aplicar para el resto de direcciones en línea recta (abajo, arriba, izquierda). En la imagen el nodo verde es el nodo a explorar, y los nodos grises han sido podados por lo que sólo debemos seguir aplicando las reglas recursivamente por los blancos que viene marcados con una flecha azul. Primero vamos a ver la razón por la que hemos podido eliminar los nodos.

Recordatorio: Estamos viendo lo que ocurre para ir a la derecha, estas reglas (adaptando las direcciones a sus equivalentes para el resto de casos) son las mismas.

1. Si venimos de nuestro padre, volver al padre vuelve al mismo resultado que teníamos antes pero aumentando el coste en uno, por lo que ese movimiento es innecesario.

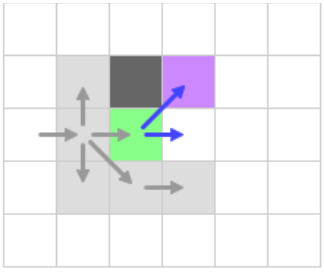
2. Volver a las casillas encima y debajo del padre tampoco es óptimo ya que desde el padre estamos a un paso normal y yendo por el hijo estamos a un paso normal y uno en diagonal que siempre va a ser peor.

3. Si quería ir a las casillas que están encima y debajo de la casilla actual (verde) ir a través de mí costaría un paso del padre hacia mí y otro paso desde mí hacia el objetivo pero había un camino más corto yendo en diagonal desde el padre.

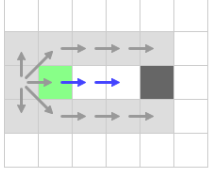
4. Para ir hacia los que están diagonal hacia la derecha del nodo actual tengo dos opciones o desde el padre tome la diagonal y sigo recto o desde el padre sigo recto hasta el hijo y tomo la diagonal hacia el objetivo. Ambas son de igual coste y puesto que el algoritmo siempre realiza movimientos en diagonal en un nodo anterior si fuese posible también podemos podar esos nodos.

Esto nos permite avanzar en una dirección aplicando estas reglas recursivamente durante un rato largo hasta que una de las condiciones de parada ocurre que será la que dará lugar a un vecino.

Veamos cuál es la condición de parada:

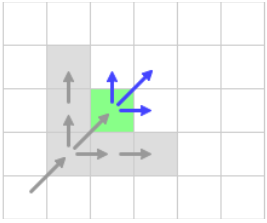


Este es el caso conocido como **vecino forzado**. En este caso estamos explorando un nodo que en perpendicular a la dirección de movimiento (en este caso arriba o abajo puesto que vamos hacia la derecha) tiene un obstáculo (casilla oscura). En este caso la suposición 4 deja de ser cierta puesto que no puede tomar un camino que desde el padre primero tome la diagonal así que, a parte de seguir hacia la derecha, el punto verde tiene cómo vecino al punto morado, lo que nos permitirá seguir explorando la parte superior salvando el obstáculo. En un mundo real, es muy importante una buena inflación de los obstáculos para evitar que el agente colisione con un obstáculo de este estilo al moverse en diagonal. Aquí añadimos el punto morado como hijo de verde en abiertos.



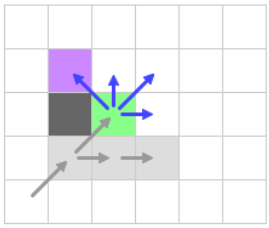
Si nos encontráramos un obstáculo en la dirección que estamos explorando sabemos entonces que por otro de los caminos explorados había un camino mejor que el que estamos explorando por lo que descartamos el salto y no añadimos nada más a abiertos.

Las condiciones para los movimientos en diagonal son bastante similares pero con ligeras variaciones. En este caso consideramos la diagonal superior derecha, las reglas explicadas se aplican a todas las diagonales modificando relativamente las posiciones usadas en las reglas.



1. Se vuelven a cumplir las restricciones de que no podemos volver al padre, y tampoco a ninguno de los caminos que van hacia las direcciones en línea recta de la diagonal (arriba y derecha) en este caso puesto que todos esos movimientos costarían el movimiento que costaría desde el padre más el movimiento en diagonal de ir al padre al hijo.

El resto de movimientos: direcciones en línea recta de la diagonal (arriba y derecha) desde el hijo y continuar el camino en diagonal **sí que han de ser explorados ahora.**



Como en el caso en línea recta tenemos también **vecinos forzados.** Si las direcciones en linea recta opuestas a la dirección de la diagonal (abajo o izquierda) contienen un obstáculo hemos añadir como vecino de la casilla verde la casilla morada (que estará en diagonal superior izquierda o diagonal inferior derecha dependiendo del obstáculo) en la lista de abiertos para ser explorado posteriormente.

Aplicando estas reglas, partiendo desde el punto inicial añadimos nodos nuevos en abiertos sólo en vecinos forzados y la otra condición de parada es evidentemente alcanzar el objetivo.

Esta técnica hace que el cuello de botella de nuestro algoritmo pase a ser la exploración de vecinos en vez de la enorme cantidad de nodos en abiertos que podrían haber aparecido. Aún así, esta técnica, es mucho más rápida que el A* con generación de vecinos convencional y existen mejoras sobre esta misma técnica que no he podido llegar a implementar y probar por falta de tiempo (http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7914/8020) (Año 2014).

3. Experimentación

Comprobemos que, efectivamente, el algoritmo mejorado nos da caminos rápidos y óptimos dentro de la seguridad establecida por las restricciones en el *costmap*. Vamos a comparar los resultados obtenidos por mi A* mejorado con JPS, con el planificador global de ROS.

Nota: hay que tener en cuenta que por errores del mapa proporcionado, hasta que el robot lleve cierto tiempo sobre el terreno habrá lugares en los que cambiará su ruta radicalmente al descubrir que hay un objeto que obstaculiza una ruta con la seguridad propuesta y creer (que puede ser cierto o no) que hay una ruta más seguro según la información que tiene.

Primero probamos a bajar por el pasillo central hacia la parte inferior del mapa en ambos casos.

Segundos plan: Tiempo que se tarda en crear el plan desde el punto más lejano

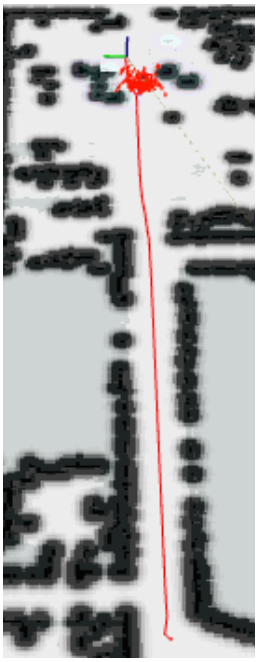
Segundos llegar: Tiempo que se tarda en llegar desde que se manda el goal

JPS

15 nodos explorados.

0 segundos plan.

47.5 segundos llegar.



Como podemos observar el camino dado por JPS es ligeramente más recto que el dado por el planificador local de ROS. Ambos se obtienen de forma instantánea y tardan aproximadamente lo mismo.

Ahora probamos a ir a la habitación a la izquierda de la habitación donde aparece el robot al iniciar.

El planificador de ROS ahora trata una ruta insegura de manera inmediata. Al final, consumiendo más tiempo (50 segundo más) del que consume mi algoritmo es capaz de llegar al punto propuesto.

JPS

1021 nodos explorados.

0.2 segundos bucle, 89.1 segundos llegar.

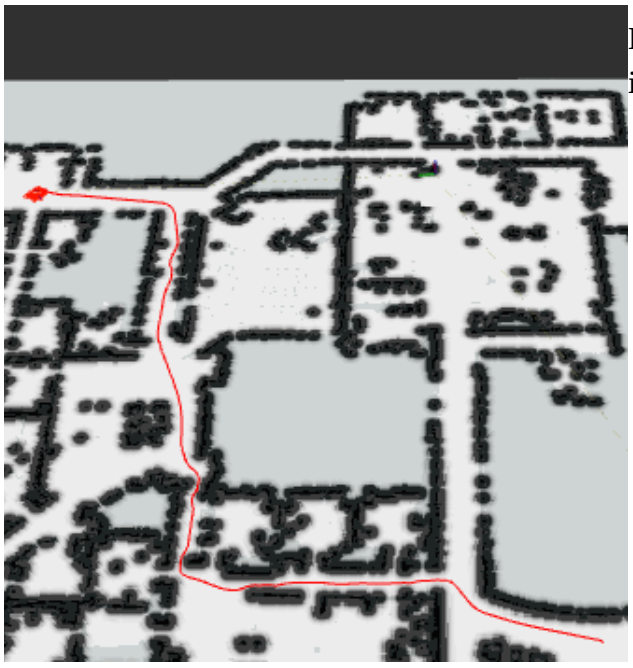
En el último caso a revisar vamos intentar recorrer un gran trecho.

JPS

1287 nodos

0.2 segundos bucle

95.3 segundos llegar



El planificador de ROS traza de nuevo una ruta insegura pero de manera inmediata.

En principio, mientras que es evidente que JPS es una gran mejora con respecto a A* debido al gran cantidad de nodos que no ha habido que explorar (reduciendo drásticamente el tiempo dedicado a la planificación) no resulta evidente quién es mejor en tiempo con respecto a JPS y ROS puesto que ROS no utiliza el concepto de seguridad mencionado en la primera parte tarda más puesto que tiene que explorar caminos en los que potencialmente aparecerá un obstáculo inflado.