DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Técnicas de los Sistemas Inteligentes. Curso 2017-18. Práctica 1: Robótica y A*

Guía para el uso e implementación de costmaps

Objetivo

En este documento se encuentra una guía rápida para entender qué son los costmaps y para manejar costmaps en ROS. El documento tiene dos partes, una guía para implementar costmaps, donde se usan conceptos generales de costmaps, y una guía para la implementación de un planificador local con costmaps, basada en la arquitectura cliente servidor que estamos implementando en esta entrega.

Conceptos básicos de costm□ps.

1. ¿Qué es un costmap?

Un costmap es una representación icónica, generalmente una matriz bidimensional (aunque también puede ser tridimensional), de un mapa o de una porción de un mapa. Un costmap bidimensional es una matriz bidimensional de enteros positivos (tipo unsigned char, intervalo [0,255]) cuyas celdas contienen no solo información sobre posiciones libres y ocupadas, sino también información sobre "cómo de segura" es una posición considerando su proximidad a un obstáculo.

2. ¿Qué fuentes de información necesita un costmap?

Un costmap trabaja a partir de información proveniente de: **sensore**s, **matriz de ocupación (occupancy grid)**, **footprint del robot (la forma geométrica del robot en el plano)** e **información propia de configuración (proporcionada en un fichero de configuración)**. Cuando se crea un costmap (mediante la declaración de un objeto de tipo costmap_2d::costmap2DROS) se subscribe automáticamente (*esto no hay que programarlo, y es la ventaja de los costmaps*) a topics de sensores publicados en ROS (tienen que ser sensores de rango porque el costmap guarda información sobre distancias a obstáculos) y necesita conocer la información sobre la matriz de ocupación del mapa en el que está situado. Los **topics a los que se subscribe para capturar información sensorial** se definen en un fichero de configuración del costmap, **la matriz de ocupación** la obtiene de algún nodo que publique tal información (normalmente del paquete "mapserver" de ROS) y **la información del footprint** se representa en el propio fichero de configuración del costmap.

3. ¿Cómo funciona un costmap?

Un costmap accede a información sobre una matriz de ocupación, publicada por algún otro nodo en ROS (normalmente "mapserver") y, a partir de unos parámetros de configuración proporcionados por el usuario, lleva a cabo un procesamiento de la información sensorial del robot (por ejemplo, escaneo láser) que resulta en un poblamiento de las celdas de la matriz del costmap. La información adquirida permite que un robot pueda tomar decisiones sobre movimiento local o de navegación global.

El procesamiento de la información sensorial en un costmap se lleva a cabo automáticamente (**no hay que programarlo**) y en ciclos de actualización de la matriz a una velocidad determinada por un parámetro llamado "update_frequency" (ver más abajo cómo configurar los parámetros de un fichero de configuración de un costmap). En cada ciclo de actualización, el costmap recibe información de sensores y lleva a cabo operaciones de marcado o *marking* (insertar información de un obstáculo en la matriz) y limpieza o *clearing* (eliminar información de obstáculos del costmap).

1. Infl□ción (Infl□tion)

Una vez realizado el marcado y limpieza de acuerdo a la nueva información sensorial recibida en cada ciclo, se lleva a cabo un proceso de propagación (denominado *inflación*) de valores desde las celdas ocupadas que van decreciendo de acuerdo a la distancia del obstáculo. El proceso de inflación está bien documentado en http://wiki.ros.org/costmap_2d#Inflation.

4. ¿Qué necesitamos saber para manejar un costmap?

Para manejar un costmap en primer lugar tenemos que saber cómo se pueden manejar mapas en ROS y cómo publicar una matriz de ocupación (esto se hace mediante el paquete mapserver). Esto es así porque un costmap recibe y modifica (mediante el proceso de inflación) la información de la matriz de ocupación de un mapa previamente publicado por map_server.

Es necesario conocer qué topics publican información sensorial del robot. En el caso de un robot simulado por Stage nos interesa el topic "base_scan". También necesitamos saber cómo configurar un costmap, esto aparece más adelante en esta guía. Necesitamos también saber cómo visualizar un costmap, mediante la herramienta rviz, que se ha explicado en la sesión de prácticas.

5. ¿Qué es necesario para que pueda funcionar un costmap?

Teniendo en cuenta los requisitos de información del costmap, para que podamos implementar adecuadamente un nodo ROS que incluya manejo de costmap, al menos deben existir en el entorno de ejecución los siguientes nodos ROS:

1. un nodo que publique la matriz de ocupación (occupancy matrix). Esto lo realiza el paquete map_server.
2. un nodo que publique información sensorial y de odometría. Esto lo hace el simulador del paquete stage_ros.
3. el propio nodo que implementa el costmap, al que se deben pasar los parámetros de configuración en un fichero (más adelante se describen estos parámetros)
4. Usar un nodo para la visualización (rviz), esto nos permitirá visualizar adecuadamente toda la información publicada, incluida la información del costmap.
5. Usar un nodo para publicar información sobre la localización del robot y poder visualizar correctamente al robot en rviz (paquete fake_localization).

A partir de ahora se hará en algunas partes referencia a los ficheros que hay en el paquete mis_costmaps.zip subido en PRADO. Descargar y compilar para entender mejor esta guía.

6. ¿Cómo se maneja un costmap?

Es importante tener en cuenta que hay dos formas distintas de inicializar un costmap y nos lleva **distinguir dos tipos fundamentales de costmaps**, que se declaran con la misma clase, pero que tienen comportamientos distintos:

1. **Global costmap.** Un global costmap guarda información sobre obstáculos a partir del mapa global del entorno donde está situado el robot. Se inicializa alimentándolo a partir de un mapa estático generado por el usuario y que se convierte en una matriz de ocupación mediante el paquete map_server. En este caso el costmap se inicializa automáticamente para ajustarse a las dimensiones del mapa estático y recibe la información sobre ocupación de la matriz de ocupación del mapa (ver map_server).
2. **Local costmap.** Un local costmap guarda información de los obstáculos en el entorno local del robot. El entorno local se define a partir de una altura y anchura (razonablemente pequeñas) y poniendo a true el parámetro "rolling window" del fichero de configuración del costmap. El parámetro "rolling window" hace que el robot se mantenga en el centro del costmap conforme se mueve en el entorno, además la información sobre obstáculos y espacio libre se va actualizando acordemente. En este sentido, el robot tiene un mapa completamente actualizado de su entorno más cercano y, por tanto, un costmap local es una herramienta muy adecuada para implementar comportamientos de navegación local.

Ficheros de configur□ción de costm□ps.

Los parámetros de un costmap están definidos y explicados en http://wiki.ros.org/costmap_2d#Component_API. Podemos capturar sus valores si nos interesara con nh getparam() pero la API de ROS para costmaps suministra funciones para manejar estos elementos. A continuación un ejemplo del fichero launch "miscostmaps_fake_5cm.launch" donde se lanza el nodo con el código fuente anterior y, además, nodos map_server, stage_ros, fake_localization y rviz.

```
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>
  <!-- include file="$(find navigation_stage)/move_base_config/move_base.xml"/ -->
   <node name="miscostmaps_node" pkg="miscostmaps" type="miscostmaps_node" output="screen">
      <rosparam file="$(find miscostmaps)/configuration/costmap_common_params.yaml" command="load"
ns="global_costmap" />
          <rosparam file="$(find miscostmaps)/configuration/costmap_common_params.yaml" command="load"
ns="local_costmap" />
          <rosparam file="$(find miscostmaps)/configuration/local_costmap_params.yaml"  command="load" />
          <rosparam file="$(find miscostmaps)/configuration/global_costmap_params.yaml" command="load" />
        </node>
   <node name="map_server" pkg="map_server" type="map_server" args="$(find
miscostmaps)/maps/simple_rooms.yaml" respawn="false"/>


  <node pkg="stage_ros" type="stageros" name="stageros" args="$(find
mistage1516)/configuracion/mundos/mi-simplerooms.world" respawn="false">
              <param name="base_watchdog_timeout" value="0.2"/>
        </node>
 <node name="fake_localization" pkg="fake_localization" type="fake_localization" respawn="false" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find navigation_stage)/single_robot.rviz" />
</launch>
```

En el código ejemplo para llamar al nodo utilizamos 3 ficheros de parámetros (ver su contenido en el paquete miscostmaps directorio “configuration” subido a la plataforma PRADO):

1. Costmap_common_params.yaml: se configuran parámetros comunes a ambos costmaps como, por ejemplo, qué **topics** relativos a fuentes sensoriales (observation_sources) se van a usar (especificando el topic y tipo de mensaje entre otras cosas), qué **valor de coste del costmap usamos como umbral** para, a partir de él, considerar que la celda que contiene ese valor es un obstáculo, qué **radio consideramos para hacer la inflación** (en este ejemplo 55 cm, es decir, los valores de las casillas a menos de 55 cm de un obstáculo tendrán un valor distinto al de FREE_SPACE), qué **dimensiones tiene el footprint del robot** (especificado en este caso como los puntos de un pentágono “con pico”), etc.
2. Local_costmap_params.yaml: parámetros exclusivos del costmap local:
    a. a qué frecuencia se modifica el costmap (**update frequency**). Este valor es **importante** porque hay que asignarlo considerando la frecuencia a la que se actualiza el escaneo láser y la frecuencia a la que se envían órdenes al robot.
    b. **Frecuencia de publicación**: este valor es **fundamental** porque por defecto está a 0 (significa no publicar el costmap) puede dar dolores de cabeza si no se pone a un valor adecuado, por ejemplo 2 o 5 Hz.
    c. Configuración del **rolling window** (ventana que avanza junto al robot):
        i. rolling_window:**tiene que estar a true**,
        ii. las **dimensiones de la ventana** (width y height, en metros),
        iii. el **origen de coordenadas** (dejarlo a origin_x = origin_y = 0)

iv. **la resolución** (resolution en metros/celda) indica "**cómo de grandes son las celdas del costmap"**, a mayor resolución, las celdas serán más pequeñas, pero esto afectará al tamaño del costmap y por tanto a la eficiencia de cualquier algoritmo que lo recorra.

3. Global_costmap_params.yaml: son los mismos parámetros que los del local costmap pero con valores distintos, especialmente **rolling_window tiene que ser false** (el global costmap no se desplza con el robot), por tanto es estático (**static_map = true**) y es importante especificar en **qué topic se publica el mapa** del que se nutre el costmap local que en nuestro caso es "/map".

Consultar esos ficheros en el código suministrado para observar la sintaxis sencilla (el fichero contiene comentarios) para asignar valores a los parámetros. Hacer a continuación lo siguiente:

1. Ejecutar el fichero launch "miscostmaps_fake_5cm.launch".
2. Añadir en rviz un nuevo display de tipo Map y asociar el topic "miscotmaps_node/local_costmap/costmap" observar que aparece el costmap local.
3. Hacer lo mismo con el topic "miscotmaps_node/global_costmap/costmap

Configur□ción dinámic□ de p□rámetros: p□quete rqt_reconfigure.

Los parámetros configurados pueden cambiarse dinámicamente en tiempo de ejecución con la herramienta "rqt_reconfigure" del paquete rqt_reconfigure http://wiki.ros.org/rqt_reconfigure .

Ejecutar rosrun rqt_reconfigure rqt_reconfigure, observar la ventana que aparece y cambiar los parámetros. Por ejemplo, observar cómo al cambiar la resolución del costmap podemos tener un costmap más o menos pixelado, dependiendo de la menor o mayor resolución, respectivamente.