Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Exploración 3D de interiores mediante múltiples MAVs

Tesis que presenta

Luis Alfonso Marín Mota

para obtener el Grado de

Maestro en Ciencias

en Computación

Directora de Tesis: Dra. Sonia Guadalupe Mendoza Chapa

Co-Director de Tesis: Dr. Dominique Decouchant

México, Ciudad de México **Diciembre 2016**

Resumen

El uso de micro vehículos aéreos (MAVs[1], por sus siglas en inglés) en la exploración y el mapeo 3D de locaciones interiores desconocidas puede evitar peligros potenciales para los seres humanos. Entre estos peligros se encuentran derrumbes de edificios, exposición a sustancias tóxicas y activación accidental de explosivos. En sitios de gran tamaño, la exploración puede llevarle mucho tiempo a un solo MAV, cuya duración de vuelo ininterrumpido es de aproximadamente 20 minutos. Dicho aspecto difícilmente puede ser mejorado con la incorporación de baterías de gran tamaño, debido a las capacidades de carga limitadas de los MAVs. El problema principal de realizar exploración y mapeo 3D de locaciones interiores, utilizando múltiples MAVs, es coordinar el movimiento de dichos MAVs para generar un mapa de ocupación 3D de un entorno desconocido. Entre las dificultades que se presentan, al tratar este problema, se encuentran las posibles colisiones entre MAVs durante la exploración de una región o la exploración de una misma región múltiples veces. La construcción del mapa también presenta inconvenientes, debido a que los MAVs pueden llegar a mapearse mutuamente como parte del entorno, obteniendo como consecuencia la adición de elementos externos al mapa. El problema de exploración y mapeo 3D multi-MAV ha sido poco tratado en la literatura relacionada, ya que los trabajos existentes utilizan un solo MAV o se limitan a exploración multi-MAV en dos dimensiones con una altura de vuelo fija. La propuesta presentada en este trabajo de tesis consiste en asignar una región a explorar a cada MAV, utilizando una partición del espacio definida por un diagrama de Voronoi en 3D, y posteriormente construir el mapa de dichas regiones, utilizando un algoritmo de exploración basada en frontera. Debido a que el diagrama de Voronoi genera particiones independientes del espacio para un conjunto de puntos, los MAVs pueden explorar sus respectivas regiones sin riesgo de colisiones entre sí. Puesto que el espacio total a explorar es el mismo, el uso de múltiples MAVs puede reducir el tiempo total de exploración. Con respecto a la construcción del mapa, se propone un proceso de limpieza en el cual se eliminen los rastros dejados por los MAVs mapeados entre sí, a través del algoritmo de relleno por inundación. Este proceso de limpieza es posible debido a que, durante la exploración, los MAVs nunca están en contacto con el resto del mapa. Por último, se realizó una implementación de esta propuesta en la plataforma robótica abierta ROS, utilizando componentes existentes de mapeo, exploración y planificación de movimiento. Dicha implementación se encarga de coordinar la exploración de MAVs ARdrone 2 simulados en Gazebo. En los experimentos realizados se observó que, a medida que se incrementa el número de MAVs, la exploración empeora. Esta stiuación se debe principalmente a la carga computacional de la simulación.

Palabras clave: exploración, mapeo, coordinación, Voronoi, multi-MAV

[1] *Micro Aerial Vehicles*

Abstract

The use of micro aerial vehicles (MAVs) in the 3D exploration of unknown indoor locations can avoid potential hazards for humans. Some of these hazards are the collapse of buildings, the exposure to toxic substances, and the accidental activation of explosives. In large buildings, exploration can long last for a MAV, whose flying time is around of 20 minutes. This aspect can hardly be improved by mounting bigger batteries, due to the payload restrictions of MAVs. The main problem with doing 3D exploration in indoor locations using multiple MAVs is the coordination of the MAVs movement to generate a 3D occupancy map of an unknown destination. Some of the difficulties that appear when solving this problem are the possible collisions among MAVs during the exploration of a region and the exploration of the same area multiple times. The map building also shows issues because MAVs can map to each other as part of the indoor location, obtaining as a consequence the addition of extra elements to the map. The multi-MAV 3D exploration problem has been few solved in the related literature, since the existent proposals use only one MAV or are limited to multi-MAV exploration in two dimensions using a fixed height. The proposal shown in this thesis consists in the assignation of a region to be explored by each MAV using a space partition defined by a 3D Voronoi diagram and then building a map of those regions, using a frontier-based algorithm. Since the Voronoi diagram generates independent partitions of the space for a set of points, the MAVs can explore their perspectives regions without risk of collisions with each other. Since the total space to be explored is the same, the use of multiple MAVs can reduce the total exploration time. Concerning the map building, a cleaning process is proposed in which the traces left by the mapped MAVs are eliminated through the flood fill algorithm. This cleaning process is possible because MAVs are never in contact with the map. Finally, an implementation of our proposal was performed on the open robotics platform ROS, using existing components for mapping, exploration, and motion planning. This implementation coordinates the exploration of simulated MAVs AR.Drone 2.0 in Gazebo. In the experiments performed we observed that as the number of MAVs increases, the exploration deteriorates. This situation is mainly due to the computational requirements of the simulation.

Keywords: exploration, mapping, coordination, Voronoi, multi-MAV

Agradecimientos

Agradezco al CONACYT por el apoyo económico otorgado para la realización de mis estudios. De la misma forma agradezco al CINVESTAV por las facilidades y el apoyo económico brindado.

Agradezco a mis padres por todo lo que han hecho por mi, valoro mucho su compañía y agradezco la oportunidad de estar junto a ellos. También agradezco al resto de mi familia, en especial a mis tíos Jesús y Carlos por su gran apoyo durante momentos difíciles de estos dos años.

A mis asesores de tesis, la Dra. Sonia G. Mendoza Chapa y el Dr. Dominique Decouchant por su apoyo y confianza en el presente trabajo de tesis. Particularmente las revisiones y las correcciones fueron de gran importancia durante el desarrollo de la tesis, sin sus comentarios el resultado habría sido muy diferente.

A la Dra. Xiaoou Li Zhang y el Dr. Sergio Víctor Chapa Vergara por ser los revisores de la presente tesis.

A todos aquellos que laboran en el departamento de computación con los que tuve la oportunidad de platicar. Su apoyo fue muy importante en las diversas etapas de mi estancia.

También como parte del departamento, agradezco a Sofía Reza Cruz por su tiempo, palabras y ayuda durante los trámites escolares. Al Dr. Amilcar Meneses Viveros por su apoyo como coordinador y profesor, confío en que sus lecciones serán de utilidad en mi futuro profesional. Al Dr. Francisco Rodríguez Henríquez por todas sus consideraciones durante estos dos años.

A la Dra. Dolores Lara Cuevas por su tiempo y apoyo durante mi estancia en el departamento, su curso de geometría computacional ayudó a despertar mi interés en la robótica.

Durante este tiempo en el CINVESTAV tuve la oportunidad de conocer a diferentes personas. Entre ellas quiero agradecer por su compañía a Guillermo, Manuel, Diana, Erik, Belem y Eliot con quienes tuve la oportunidad de pasar más tiempo y

diversas experiencias, hicieron que los momentos en la escuela fueran más divertidos y me ayudaron a salir adelante en las materias. También agradezco a Laura R. por los buenos momentos que compartimos, gran parte de mis mejores recuerdos de esta etapa de maestría te los debo a ti.

Por último, pero no menos importante, agradezco a mi amiga Thalia por su apoyo durante este "TT III". Gracias tu amistad, por escucharme y por animarme a avanzar.

Índice general

Índice de figuras

Índice de tablas

Índice de Algoritmos

Lista de archivos

Capítulo 1

Introducción

1.1. Contexto de investigación

El presente trabajo de tesis se sitúa en el área de metodologías de cómputo, de acuerdo con la clasificación ACM [ACM, 2015]. Particularmente este proyecto involucra las sub áreas de planificación robótica, cooperación y coordinación, como se muestra en la figura 1.1.

La planificación robótica o planificación de movimiento es de gran importancia pues permite que un robot pueda tener autonomía al ser capaz de desplazarse en un entorno sin la intervención de un ser humano que le indique como moverse, la planificación robótica involucra un conjunto de problemas relacionados con mover un objeto (robot) desde un punto inicial hasta un punto final evitando colisionar con los posibles obstáculos del entorno [O'Rourke, 1998].

En el campo de la inteligencia artificial distribuida se presentan sistemas formados por un conjunto de agentes. Estos sistemas intentan resolver problemas en los cuales una conducta colectiva aparece más eficiente que una conducta individual. Estos sistemas suelen mejorar los tiempos de respuesta y reducir la complejidad del problema al dividirlo en subtareas. Un agente es un sistema situado dentro de un entorno, que percibe dicho entorno y actúa sobre él a lo largo del tiempo, sus propiedades son las siguientes [Wooldridge and Jennings, 1995].

- Autonomía: es la capacidad de actuar independientemente sin interacción humana.
- Habilidad social: consiste en la interacción o colaboración con otros agentes mediante un lenguaje de comunicación.
- Reactividad: los agentes deben ser capaces de percibir el ambiente y responder oportunamente a los cambios que ocurran en dicho ambiente.

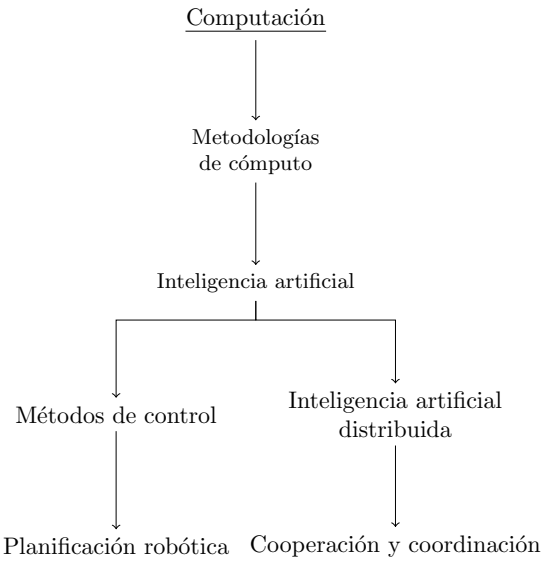


Figura 1.1: Áreas de la computación involucradas en el presente trabajo de tesis de acuerdo con la clasificación ACM 2012.

- Proactividad: los agentes no simplemente actúan en respuesta al ambiente sino que son capaces de tomar la iniciativa y actuar para alcanzar sus metas.

Dentro la robótica los sistemas multi-robot tienen un comportamiento multi-agente en el cual es necesario coordinar las tareas que realiza cada robot para alcanzar consensos resolviendo conflictos de interés. Así como cooperar compartiendo información sobre el problema y la solución desarrollada.

1.2. Motivación

Los MAVs han cobrado relevancia en años recientes, tanto en ambientes comerciales como de investigación. Esta situación se debe a que por su tamaño y capacidades, los MAVs pueden acceder a lugares que los robots terrestres no pueden. Esta flexibilidad permite el desarrollo de aplicaciones de diversos tipos, entre los cuales destaca la exploración de interiores.

A pesar de que la exploración en interiores, usando robots es un tema previamente tratado, la mayoría de las veces se enfoca únicamente en cubrir el piso; con dicho enfoque la complejidad del problema se reduce a explorar un ambiente en dos dimensiones. Shen et al., presentan un método de exploración y mapeo 3D, pero se limita al uso de un solo MAV [Shen et al., 2012]. En las pocas ocasiones en que se han utilizado múltiples MAVs para explorar, se considera una altura fija lo cuál permite dar paso a otro MAV descendiendo al piso en caso de posible colisión.

En locaciones de gran tamaño, la exploración completa del entorno puede llevarle mucho tiempo a un solo MAV. El tiempo de exploración es importante, debido a que los MAVs cuentan con capacidades de carga limitadas, por lo tanto no es posible incorporar baterías grandes que extiendan su tiempo de vuelo. Para algunos MAVs comerciales, el tiempo de vuelo es de aproximadamente veinte minutos, sin considerar tareas adicionales de procesamiento computacional. Aún en el supuesto de que la batería fuera suficiente para cubrir un entorno específico, esto podría consumir mucho tiempo, lo cuál no es aceptable en determinados casos. Esta limitación hace que sea necesario considerar el uso de múltiples MAVs para reducir el tiempo de exploración y extender el área mapeada del entorno.

Es de interés realizar exploración 3D con múltiples MAVs, debido a sus múltiples aplicaciones prácticas, entre las cuales se encuentran el reconocimiento de terreno en situaciones de desastre naturales o producto de ambientes hostiles, como los campos de batalla. Un ejemplo de aplicación práctica puede ser la exploración de un edificio que ha sido afectado por un terremoto y cuya estructura se encuentra en peligro de colapsar. La exploración del edificio permitiría detectar los posibles caminos bloqueados para moverse dentro del inmueble, sin la necesidad de poner en riesgo al personal bomberos o rescate.

1.3. Planteamiento del problema

El problema de exploración y mapeo 3D de locaciones interiores con múltiples MAVs consiste en coordinar el movimiento de dichos MAVs para generar un mapa de ocupación 3D de un entorno desconocido. En la literatura relacionada, las propuestas utilizan un solo MAV para realizar exploración o hacen exploración multi-mav a una altura fija. Entre las dificultades que se presentan al abordar este problema se encuentran las posibles colisiones que pueden ocurrir durante el vuelo de los MAVs. Adicionalmente unas mismas parte del entorno podría ser explorada más de una vez por los MAVs. La construcción del mapa también presenta inconvenientes debido a que un MAV puede ser registrado como parte del entorno por otro MAV, lo anterior tendría como consecuencia la adición de elementos externos al mapa.

1.4. Objetivos

El objetivo de este trabajo de tesis es diseñar e implementar un sistema de exploración que permita coordinar el mapeo y exploración 3D de un entorno desconocido en locaciones interiores, mediante el uso de múltiples MAVs. Este objetivo se apoya en los siguientes objetivos específicos.

1. Diseñar una estrategia de exploración 3D en interiores que admita múltiples MAVs para reducir el tiempo de mapeo del entorno.

2. Desarrollar un sistema de exploración que implemente la estrategia de exploración propuesta.

3. Realizar simulaciones con el sistema de exploración para validar su capacidad de resolver problemas de exploración.

Con el cumplimiento de estos objetivos se pretende validar la siguiente hipótesis de investigación: *es posible reducir el tiempo de exploración 3D en interiores mediante el uso de múltiples MAVs, en comparación con el uso de un solo MAV.*

1.5. Organización de la tesis

El presente documento de tesis se encuentra organizado de la siguiente manera (ver Figura 1.2). En el capítulo 2 se presenta una introducción a los vehículos aéreos no tripulados, su historia y clasificación. Adicionalmente, en dicho capítulo se detalla el concepto de MAV y se realiza una descripción de los MAVs tipo cuadricóptero. En el capítulo 3 se presentan los conceptos de mapeo, planificación de movimiento y exploración robótica. El mapeo describe la forma en como se construyen los mapas utilizando robots. El concepto de planificación de movimiento involucra la forma en como un robot genera trayectorias en el espacio libre. Por último, la exploración robótica describe la manera en que un robot incrementa su conocimiento del mapa al identificar zonas desconocidas. En el capítulo 4 se presentan algunos de los trabajos de exploración multi-robot y exploración mediante MAVs presentes en la literatura. Con respecto a estos trabajos se realiza un análisis comparativo para identificar sus características. En el capítulo 5 se presenta el diseño del sistema de exploración propuesto. Este diseño incluye la arquitectura del sistema y la descripción de los procesos que lo componen. En en el Capítulo 6 se describe la implementación del sistema en ROS y la configuración de las herramientas necesarias para su funcionamiento. En el capítulo 7 se presentan las pruebas realizadas al sistema de exploración así como los resultados de las mismas. Finalmente, en el capítulo 8 se presentan las conclusiones del presente trabajo de tesis y el trabajo futuro.

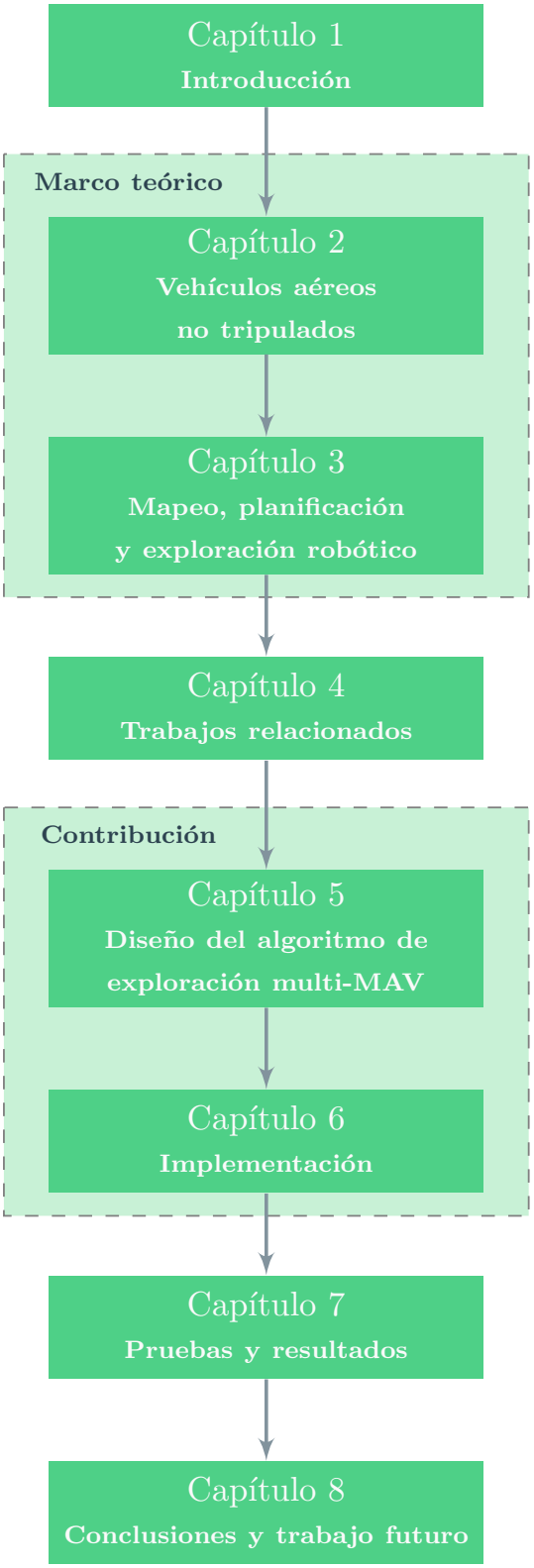


Figura 1.2: Estructura del presente documento de tesis.

Capítulo 2

Vehículos aéreos no tripulados

En este capítulo se realiza una introducción a los vehículos aéreos no tripulados (UAVs[1], por sus siglas en inglés), categoría a la cual pertenecen los MAVs. En la Sección 2.1 se introduce el concepto de UAV y los términos relacionados. En la Sección 2.2 se hace un breve repaso de los antecedentes de los UAVs y sus primeros desarrollos. Así mismo, en la Sección 2.3 se hace una revisión del término **dron**, que se ha vuelto popular para referirse a toda clase de UAVs. En la Sección 2.4 se presentan las clasificaciones de los UAV por su tipo y peso. A continuación, en la Sección 2.5 se describen las arquitecturas multi-UAV, de acuerdo a la forma en que los UAVs cooperan entre sí. Por último, en la Sección 2.6 se presenta la definición de MAV y el esquema general del funcionamiento de un MAV tipo cuadricóptero, en razón de su uso creciente en ambientes comerciales y de investigación.

2.1. Definición

Los UAVs han estado presentes desde hace más de un siglo, aunque inicialmente se destinaron a fines militares y tareas especializadas. En épocas recientes han cobrado relevancia debido a los avances tecnológicos que han permitido su acercamiento al público en general. Cuándo se habla de UAVs, es común encontrarse con una amplia variedad de términos, que en ocasiones, son utilizados para referirse a los mismos conceptos.

Un UAV es una máquina voladora reusable que carece de un piloto humano o pasajeros a bordo. Este tipo de vehículos puede ser controlado remotamente o contar con una computadora que le permita volar de manera autónoma. Además no está clasificado como arma guiada, i.e., como dispositivo para disparo o bombardeo no recuperable, e.g., misiles [JCGUAV, 2007].

[1] *Unmanned Aerial Vehicles*

Diversas entidades gubernamentales han propuesto sus propias definiciones, en las que se conservan conceptos como **no tripulado** y **controlado a distancia** o **con una computadora abordo**, pero se excluyen algunas aeronaves. Por ejemplo la Administración Federal de Aviación (FAA[2], por sus siglas en inglés), entidad responsable de la aviación civil en los Estados Unidos, considera únicamente a aquellas aeronaves que pueden ser controladas en tres ejes o más, con lo cual quedan excluidos los globos [FAA, 2008]. El Departamento de Defensa de los Estados Unidos (DoD[3], por sus siglas en inglés) también excluye los satélites, proyectiles de artillería, minas y aquellos sensores que no cuentan con alguna forma de propulsión [of Defense Office of the Secretary of Defense, 2007].

Sumado a lo anterior, los términos **UAV** y **aeronave no tripulada** (UA[4], por sus siglas en inglés) son utilizados indistintamente para nombrar al mismo tipo de vehículos aéreos. Sin embargo, el término adoptado por algunas agencias gubernamentales como la FAA y la Agencia Europea de Seguridad Aérea (EASA[5], por sus siglas en inglés) es **UA**.

En ocasiones, los términos **UAV** y **sistema aéreo no tripulado** (UAS[6], por sus siglas en inglés) se emplean para referirse al mismo concepto. En este caso, debe considerarse que el UAV es un elemento de un UAS. El Departamento de Defensa de los Estados Unidos reconoce a los vehículos no tripulados (UVs[7], por sus siglas en inglés) como el componente primario de los sistemas no tripulados (US[8], por sus siglas en inglés). Por otra parte, la Agencia Europea de Seguridad Aérea considera que un UAS se compone de una aeronave, una estación de control y todos elementos necesarios para permitir el vuelo de la aeronave, tales como los elementos utilizados para su lanzamiento y recuperación [Valavanis and Vachtsevanos, 2015].

2.2. Historia

El desarrollo de los UAVs ha tenido un largo trayecto, aunque en los últimos 100 años se han presentado los mayores avances. El concepto de UAV tienen sus orígenes en la antigua Grecia y parte de China, aunque en la actualidad está asociado con la electrónica moderna. Con el paso de los años, continuaron surgiendo nuevas máquinas voladoras carentes de tripulación y motivadas por las tecnologías que dominaban la época. Con la introducción de la electrónica, se impulsó la construcción de máquinas

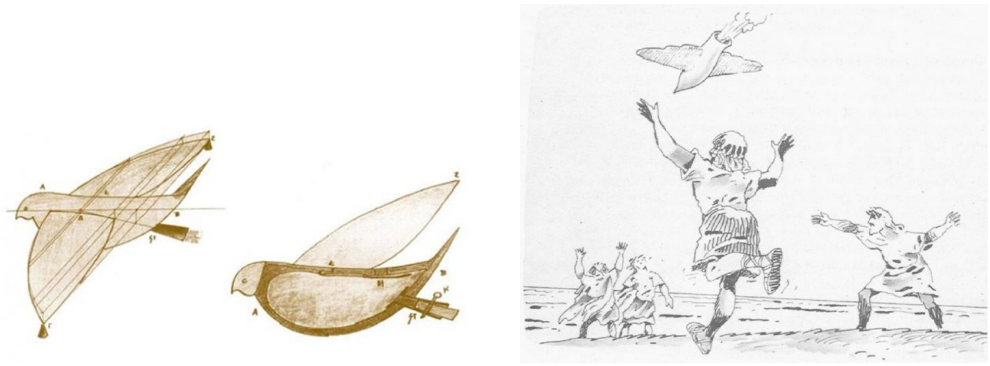
[2] *Federal Aviation Administration*
[3] *United States Department of Defense*
[4] *Unmanned Aircraft*
[5] *European Aviation Safety Agency*
[6] *Unmanned Aircraft System*
[7] *Unnmanned Vehicles*
[8] *Unnmanned Systems*

voladoras más sofisticadas, controladas a distancia, las cuales dieron origen a un desarrollo que continua en la actualidad.

2.2.1. Antecedentes

La historia de la aviación tanto tripulada, como no tripulada se remonta a alrededor de 2500 años atrás. En la antigua Grecia, se desarrollaron prototipos de máquinas voladoras, mientras que en China se experimentó con globos de aire caliente y papalotes utilizados para señalización en el campo de batalla.

La primera máquina voladora autónoma es atribuida a Archytas de Tarantas en el año 425 a. C. Esta máquina, llamada "La paloma" [Dalamagkidis et al., 2012], consistía en una estructura de madera en forma de paloma que era impulsada por vapor en su interior. El cuerpo de la paloma era hueco con forma cilíndrica y poseía alas proyectadas hacía los lados. La parte trasera poseía una abertura que daba acceso al interior y que se conectaba a una caldera sellada herméticamente. La presión del vapor producido por la caldera con el tiempo superaba la resistencia del sello y entonces la paloma emprendía el vuelo [Origins, 2014]. Se dice que el vuelo de la paloma era de hasta 200 metros de distancia (ver Figura 2.1).



(a) (b)

Figura 2.1: Ilustraciones de la paloma creada por Archytas: (a) diagrama de funcionamiento [Origins, 2014] y (b) la paloma en vuelo [Dalamagkidis et al., 2012].

Alrededor del año 200 d. C., el general chino Zhuge Liang creó la linterna voladora de *Kong Ming*. Para ello utilizó globos de papel equipados con lámparas de aceite que calentaban el aire dentro del globo y le permitían elevarse. Zhunge Liang utilizó estas linternas con el fin de asustar a sus enemigos, haciéndolos creer que poseía poderes divinos [Marshall et al., 2015]. En la actualidad estas linternas voladoras son utilizadas en festivales y celebraciones de Asía (ver Figura 2.2).

Leonardo Da Vinci fue otra de las personas que construyó máquinas voladoras. A partir de sus estudios, desarrolló artefactos que le permitirían al hombre planear en el

Figura 2.2: Linterna en el festival de la segunda luna llena, conocido como *Yi Peng*, en Tailandia.

aíre. Una de sus creaciones, ideada para el vuelo no tripulado, fue el "Tornillo aéreo" en 1483 (ver Figura 2.3). Esta máquina, que es considerada el ancestro del helicóptero, tenía un diámetro de 5 metros y para activarla debía hacerse girar el eje hasta alcanzar la fuerza y velocidad necesarias para que se elevara [Dalamagkidis et al., 2012].

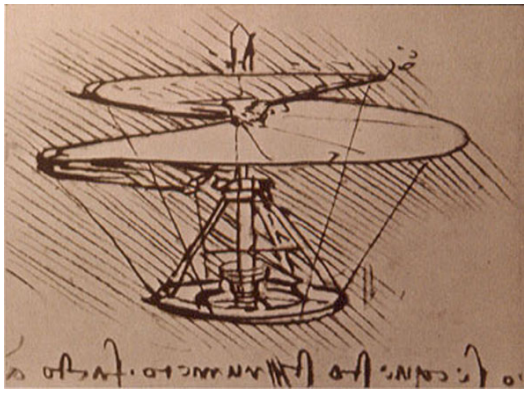


Figura 2.3: Tornillo aéreo de Leonardo Da Vinci.

Durante el siglo XIX, se diseñaron múltiples máquinas voladoras propulsadas por vapor [Valavanis and Kontitsis, 2007]. Entre ellas destaca el carruaje aéreo a vapor (*aerial steam carriage*) creado por William Samuel Henson en 1843 (ver Figura 2.4). El carruaje aéreo estaba planeado para transportar pasajeros, pero finalmente no se llevó a cabo debido a los experimentos fallidos y altos costos involucrados. El prototipo del carruaje aéreo, que Henson presentó, fue un modelo propulsado por vapor que se elevaba en un cable guía [Flyingmachines.org, 2016].

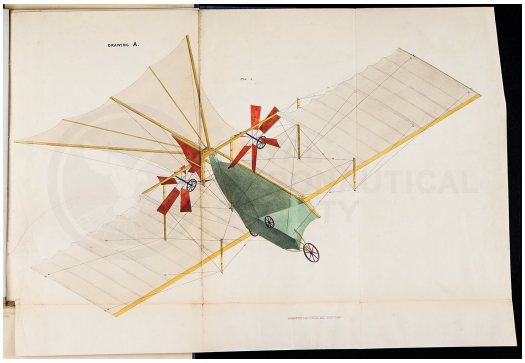


Figura 2.4: Carruaje aéreo a vapor de William Samuel Henson [Aerosocietyheritage.com, 2016].

2.2.2. Primeros UAVs controlados a distancia

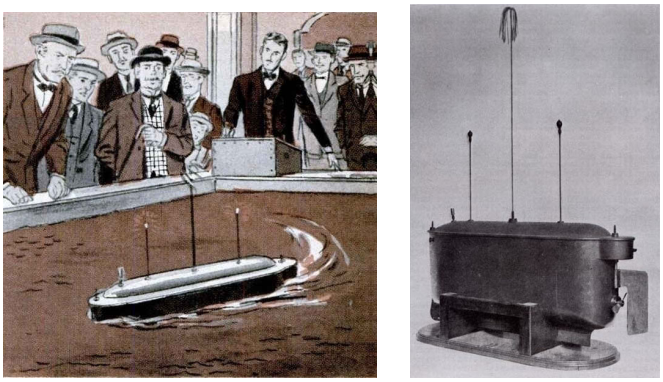
En 1898, Tesla presentó Teleautomaton (ver Figura 2.5), un dispositivo controlado remotamente que tenía la forma de un bote en miniatura [Books, 2016]. Este dispositivo era manipulado por Tesla mediante una caja de control que carecía de algún tipo de conexión visible con el pequeño bote. El dispositivo era manipulado mediante ondas electromagnéticas. En la patente de 1898, Tesla describe algunas de las aplicaciones de su invento.

> "La invención que he descrito será útil en muchas formas. Embarcaciones o vehículos de cualquier tipo podrán ser utilizados para enviar y entregar paquetes, provisiones, instrumentos, objetos y materiales de cualquier tipo. Para establecer comunicación con regiones inaccesibles y explorar las condiciones existentes de estas [...]. Pero el mayor valor de mi invención resultará de su efecto en el campo de guerra y armamento por su uso seguro y poder destructivo."

En 1915, Tesla promovió la idea de aeronaves pilotadas remotamente y equipadas con bombas [Marshall et al., 2015]. Esta idea se vio materializada en el proyecto AT (*Aerial Target*). En 1916, como parte de este proyecto, Henry Philip Folland y Archibald Montgomery Low diseñaron un monoplano a control remoto no tripulado, al cual nombraron *Aerial Target* [NASA, 2016] (ver Figura 2.6). El proyecto contó con el apoyo del Real Cuerpo Aéreo británico (RFC[9], por sus siglas en inglés) [Senson and Ritter, 2011]. Se dice que el nombre de la aeronave fue sugerido por el general Sir David Henderson para encubrir su propósito real, el cuál era servir como sistema de ataque, dado que estaba equipado con bombas [Everett and Toscano, 2015].

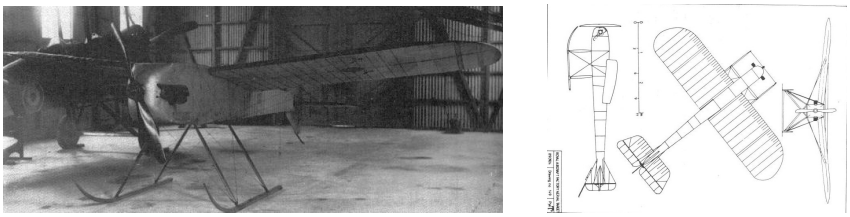
En 1918, se presentó *Aerial Torpedo*, una aeronave desarrollada por Elmer Ambrose Sperry a petición de la marina de los Estados Unidos. Entre las características inicialmente solicitadas por la marina se encontraban que fuera ligero, no tripulado,

[9] *Royal Flying Corps*



(a) (b)

Figura 2.5: Teleautomaton de Nikola Tesla: (a) ilustración de la presentación del dispositivo durante la exhibición eléctrica del Madison *Square Garden* de 1898 y (b) foto del prototipo.



(a) (b)

Figura 2.6: UAV *Aerial target*: (a) foto y (b) diagrama.

capaz de volar 1000 yardas guiado hasta su objetivo y detonar una cabeza armada en un punto lo suficientemente cercano para ser efectivo contra una embarcación [Marshall et al., 2015].

Estos primeros prototipos de aeronaves teledirigidas sentaron los precedentes de muchos otros UAVs construidos durante la primera y segunda guerra mundial.

2.3. Drones

En la actualidad, un término común para referirse a los UAVs es **dron**, una adaptación de la palabra *drone* en inglés, que significa zángano, el cual es el macho de la abeja encargado de la fecundación de la abeja reina. Pero este término tiene sus origenes en los primeros UAVs controlados a distancia. De acuerdo con Steve Zaloga, autor del libro *Unmanned Aerial Vehicles: Robotic Air Warfare 1917-2007* [Zaloga, 2008], el origen de la palabra *drone* está relacionado con una aeronave a control remoto de los años 30, que servía como blanco de pruebas para la artillería antiaérea utilizada por la marina británica [Ignacio Sánchez, 2016, Zimmer, 2016]. En una carta enviada al sitio de noticias *Defense News*, dedicado a tratar temas de política y tecnología de

defensa, Zaloga expone lo siguiente [Mehta, 2013].

> "En 1935 el almirante William Standley, jefe de operaciones navales de Estados Unidos, visitó Reino Unido dónde recibió una demostración por parte de la Marina Real británica (en inglés, *Royal Navy*). En esa demostración se presentó la aeronave remotamente controlada, DH 82B *Queen Bee*[10], utilizada para prácticas de artillería antiaérea. A su regreso, el almirante Standley asignó al comandante Delmer Fahrney, perteneciente a la división de radio del Laboratorio de Investigación Naval (NRL, por sus siglas en inglés[11]), el desarrollo de un sistema similar para prácticas de artillería de la marina de Estados Unidos. El comandante Fahrney adoptó el nombre *drone* para referirse a aquellos sistemas en homenaje a la aeronave *Queen Bee*. El término *drone* se convetiría en la designación oficial de la marina para blancos remotamente controlados por muchas décadas."

La versión anterior parece coincidir con lo explicado en Marshall et al., dónde se menciona la historia de los blancos controlados a distancia (en inglés, *Target drones* y el desarrollo de *Queen Bee* (ver Figura 2.7) para pruebas de artillería por parte de la marina británica y su posterior adopción por la milicia de los Estados Unidos [Marshall et al., 2015].

Figura 2.7: Aeronave *Queen Bee* [Dalamagkidis et al., 2012].

De acuerdo con Fahlstrom y Gleason, el uso del término **dron**, adoptado por la prensa no especializada y público en general, para llamar a cualquier tipo de UAV es técnicamente incorrecto. Lo anterior se debe a que la palabra dron se encuentra relacionada con las primeras aeronaves controladas remotamente, e.g., los blancos teledirigidos. Dichas aeronaves carecían de funciones sofisticadas, como navegación autónoma, por el contrario su vuelo era monótono y su flexibilidad para llevar a cabo misiones era limitada [Fahlstrom and Gleason, 2012].

[10] En español, abeja reina
[11] *Naval Research Laboratory*

2.4. Clasificación de los UAVs

El desarrollo de UAVs ha dado lugar a vehículos de toda clase de dimensiones, es por ello que instituciones de todo el mundo cuentan con regulaciones para identificarlos, de acuerdo a sus características. Las clasificaciones de UAVs se basan en aspectos tales como dimensiones, riesgo de impacto, autonomía o tipo de uso (civil y militar). Sin embargo, estos criterios no están estandarizados a lo largo del mundo. Por ejemplo, van Blyenburgh presenta una clasificación de UAVs de acuerdo a su masa, rango, altitud de vuelo y resistencia (ver Tabla 2.1) [van Blyenburgh, 2006].

Otra de las clasificaciones de UAVs es la propuesta por Nonami et al., en la que se utiliza la configuración aerodinámica como criterio de clasificación y se divide en los siguientes tipos: ala fija, ala rotativa, dirigibles y de aleteo [Nonami et al., 2010]. El tipo ala fija se refiere a los aeroplanos no tripulados, i.e., que tienen alas. Este tipo de UAVs requiere una pista para aterrizar y despegar, o una plataforma de lanzamiento (ver Figura 2.8). No obstante dicho requerimiento, este tipo de UAVs puede realizar grandes recorridos y alcanzar altas velocidades.

(a) (b)

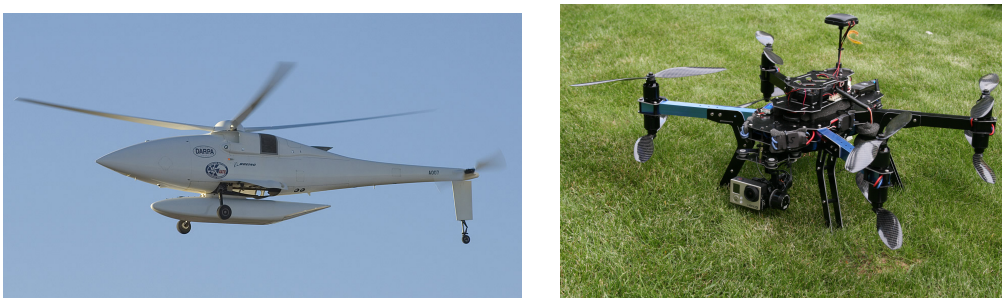
Figura 2.8: UAVs de ala fija en plataforma de lanzamiento: (a) modelo Manta fabricado por Sensintel [Sensintel, 2016] y (b) modelo SCRAB II desarrollado por Scrtargets [Scrtargets.es, 2016].

Los UAVs de tipo ala rotativa son aeronaves VTOL[13] (por sus siglas en inglés), lo que significa que pueden aterrizar y despegar verticalmente, por lo tanto poseen alta maniobrabilidad y son capaces de permanecer suspendidos en el aire. Este tipo de vehículos puede ser encontrado en múltiples configuraciones de rotores, que van desde los helicópteros no tripulados hasta los vehículos multi-rotor, e.g., cuadricóptero, hexacóptero y octacóptero. En la Figura 2.9, se presentan dos UAVs, uno de tipo helicóptero y otro cuadrirotor.

[12] Varia de acuerdo a restricciones legales nacionales
[13] *Vertical Take-Off and Landing*

	Clasificación	**Masa (kg)**	**Rango (km)**	**Altitud de vuelo (m)**	**Resistencia (horas)**
	Micro	<5	<10	250	1
	Mini	<20/25/30/150 [12]	<10	150/250/300	<2
Tácticos	***Close range* (CR)**	25−150	10−30	3,000	2−4
	***Short range* (SR)**	50−250	30−70	3,000	3−6
	***Medium range* (MR)**	150−500	70−200	5,000	6−10
	MR *endurance* (MRE)	500−1,500	>500	8,000	10−18
	***Low altitude deep penetration* (LADP)**	250−2,500	>250	50−9,000	0.5−1
	***Low altitude long endurance* (LALE)**	15−25	>500	3,000	>24
	***Medium altitude long endurance* (MALE)**	1,000−1,500	>500	3,000	24−48
Estratégicos	***High altitude long endurance* (HALE)**	2,500−5,000	>2,000	20,000	24−48
	***Stratospheric* (Strato)**	>2,500	>2,000	>20,000	>48
	Exo-stratospheric (EXO)	Por determinar	Por determinar	>30,500	Por determinar
Tareas especiales	**Unmanned combat AV (UCAV)**	>1,000	1,500	12,000	2
	Lethal (LET)	Por determinar	300	4,000	3−4
	Decoys (DEC)	150−250	0−500	50−5,000	<4

Tabla 2.1: Clasificación de UAVs de acuerdo a su tamaño [van Blyenburgh, 2006].



(a) (b)

Figura 2.9: UAVs de ala rotativa: (a) boeing modelo A160T Hummingbird [Daily, 2016] y (b) 3DR modelo SX8-M [3DR, 2016].

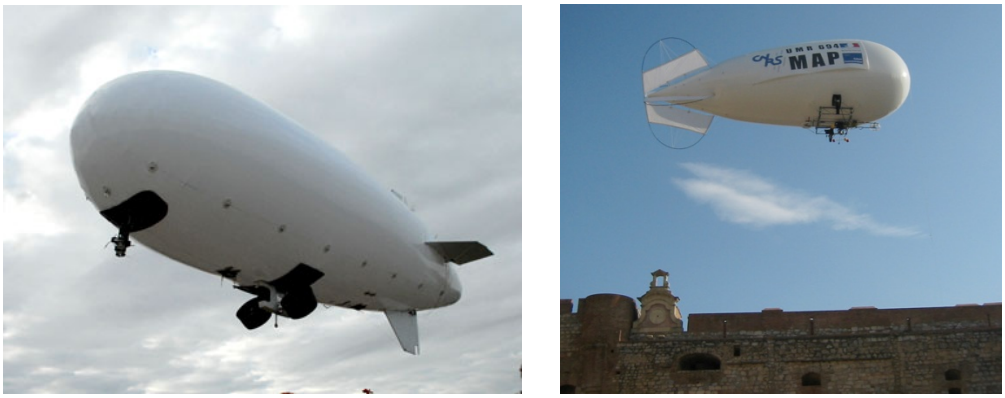
Los UAVs de tipo dirigible son aeronaves que se inflan con un gas más ligero que el aire, lo que les permite flotar. Este tipo de vehículos aéreos puede realizar grandes recorridos, pero a bajas velocidades (ver Figura 2.10). Los dirigibles, por su estructura, pueden ser clasificados en [Grossman, 2016]:

- **rígidos**, deben su forma a una estructura que rodea una o más celdas de gas individuales mas no a la presión del gas utilizado para elevarlo;
- **semirrígidos**, mantienen su forma debido al gas en su interior, pero cuentan con una estructura parcialmente rígida que soporta y distribuye el peso, proveyendo integridad estructural durante las maniobras.
- **no rígidos (*blimps*)**, mantienen su forma únicamente en razón de la presión del gas con el que son llenados.

Los UAVs de aleteo poseen alas con formas inspiradas en pequeñas aves e insectos voladores (ver Figura 2.11). Para el diseño de las alas, se debe considerar el tipo de configuración a utilizar, e.g., alas independientes o pares de alas en cada lado como las de las libélulas. Algunos de los materiales más utilizados en la fabricación de alas son las hojas de PET y los basados en fibra de carbono reforzado [de Croon et al., 2016].

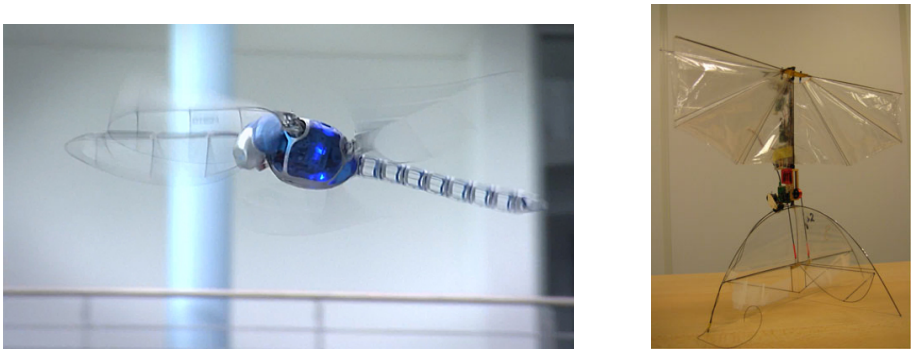
2.5. Arquitecturas multi-UAV

En los sistemas multi-UAVs se requiere considerar los conceptos de coordinación y cooperación. La coordinación permite a un conjunto de robots compartir un mismo espacio, sin riesgo de interacciones peligrosas entre ellos, y también es necesaria cuando se trata de realizar acciones sincronizadas. Por otra parte, el concepto de cooperación está relacionado con la asignación de una tarea, de tal forma que las acciones de los robots contribuyen a la realización de la misma. Esta cooperación puede ser centralizada o descentralizada, dependiendo de dónde se encuentre el conocimiento necesario



(a) (b)

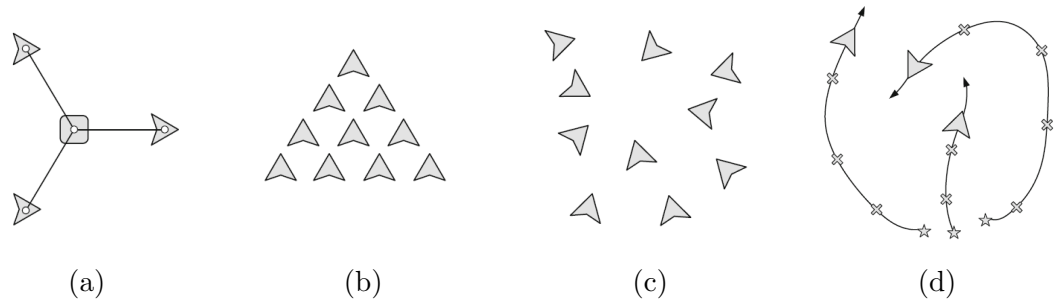
Figura 2.10: UAVs de tipo dirigible: (a) Airship Solutions modelo AS10 [Airship.com.au, 2016] y (b) dirigible no rígido utilizado para la captura de imágenes empleadas en reconstrucción 3D [Deveau, 2016].



(a) (b)

Figura 2.11: UAVs de tipo aleteo: (a) Festo Bionicopter [Festo, 2013] y (b) DelFly [De Croon et al., 2009].

para realizar la tarea. Para elegir el tipo de cooperación se deben tomar en cuenta las capacidades computaciones de cada robot, e.g, capacidad de procesamiento y ancho de banda. De acuerdo a la forma en que los UAVs interactuan entre sí, se definen cuatro clasificaciones (ver Figura 2.12): acoplamiento físico, formaciones, enjambres y cooperación intencional [Maza et al., 2015].



(a) (b) (c) (d)

Figura 2.12: Arquitecturas multi-UAV [Maza et al., 2015]: (a) acoplamiento físico, (b) formaciones de UAVs, (c) enjambre y (d) cooperación intencional.

Cómo lo índica su nombre, en el acoplamiento físico, los UAV están conectados de tal forma que los movimientos de uno afectan al resto. Los UAVs que colaboran bajo estas restricciones son aquellos que cargan o transportan cosas. Por lo tanto, los UAVs están sujetos a las fuerzas generadas por el resto y su comportamiento debe ser el de un todo, lo cual implica desplazarse simultáneamente y evadir obstáculos. Una estrategia para realizar transporte consiste en definir un UAV, que actúe como líder, al cuál el resto debe seguir. Los seguidores no conocen directamente el movimiento del líder, sino que lo estiman a través del movimiento del objeto transportado.

Con respecto a las formaciones de UAVs, el esfuerzo se enfoca en generar modelos que mantengan a los UAVs separados a una distancia previamente definida. La forma de mantener la distancia puede lograrse, informando sus posiciones a través de comunicaciones inalámbricas o identificando a su vecinos. En esta arquitectura también se puede definir un líder, el cual será el único responsable de realizar la **planificación de movimiento** y los seguidores procederán a moverse, utilizando transformaciones de la posición del líder. Por otra parte, el líder puede ser virtual, i.e., un punto de referencia en el espacio, con lo que se permitiría a los UAVs intercambiar posiciones entre ellos.

La arquitectura de tipo enjambre, vista en su mayoría en MAVs, basa su funcionamiento en pequeñas interacciones entre grandes conjuntos de individuos para alcanzar metas globales. Este comportamiento está inspirado en animales de la naturaleza como las aves o los peces. La forma en que los UAVs perciben a sus vecinos es a través de mediciones de distancia o señales de luces de colores. Este tipo de percepción se realiza debido a las limitaciones de las comunicaciones inalámbricas. Una aplicación

potencial de los enjambres de MAVs es la búsqueda de objetos evasivos en una zona confinada, ya que mediante la combinación de los sensores individuales se obtiene un campo más amplio de sensado [Maza et al., 2015].

El último tipo de arquitectura multi-UAV, al cual pertenece el presente trabajo de tesis, es el de cooperación intencional. Esta arquitectura consiste en la asignación de subtareas o metas independientes para cada UAV, dichas subtareas contribuyen a alcanzar una meta global. La asignación de tareas debe ser realizada de tal forma que el equipo complete la meta global de la manera más eficiente posible. Para ello es necesario establecer mecanismos de coordinación que permitan la correcta interacción del conjunto de UAVs. Esta coordinación abarca desde la planificación de trayectorias libres de colisiones, hasta la correcta actualización del estado de los vecinos de cada UAV. El problema de asignación de tareas se vuelve más complicado cuando se introducen dispositivos heterogéneos, e.g., equipos formados por robots terrestres y UAVs.

2.6. MAVs

Los MAVs son una subclasificación de los UAVs que en los últimos años han estado presentes tanto en el campo de la investigación, como en el ámbito comercial. El creciente uso de MAVs se debe principalmente a su pequeño tamaño y bajo costo con respecto al resto de los UAVs. Adicionalmente, las regulaciones del espacio aéreo permiten, en muchas ocasiones, poder utilizarlos sin necesidad de una licencia de vuelo expedida por la agencia correspondiente.

La definición de MAV varía dependiendo de la fuente, debido a los diferentes criterios que se utilizan para clasificar este tipo de UAVs. La Agencia de Investigación de Proyectos Avanzados de la Defensa de Estados Unidos (DARPA[14], por sus siglas en inglés) utiliza las dimensiones de estas aeronaves. Específicamente, la DARPA establece que un MAV debe medir menos de 15 cm a lo largo, ancho y alto [Nonami et al., 2010]. Por otra parte, de acuerdo con van Blyenburgh, la masa de un MAV debe ser inferior a 5 Kg [van Blyenburgh, 2006]. Es esta última definición a la que se apega el presente trabajo de tesis.

Cabe destacar que, a pesar de que en los últimos años se han popularizado los MAVs multi-rotor, no todos los MAVs pertenecen a dicha categoría. Por ejemplo, considerando la definición de MAV, los UAVs Festo Bionicopter y DeFly de tipo aleteo, presentados en la Sección 2.4 también son considerados como MAVs. Sin embargo, es de interés conocer el funcionamiento general de los cuadricópteros, debido a que múltiples trabajos de investigación hacen uso de esta clase de vehículos aéreos.

[14]Defense Advanced Research Projects Agency

Los UAVs de tipo cuadricóptero son una subcategoría de los UAVs VTOL, específicamente de los UAVs multi-rotor, ya que como su nombre lo indica poseen cuatro rotores. Estos vehículos no son nuevos, puesto que desde principios del siglo XX se comenzó a experimentar con aeronaves de este tipo. Su éxito en el ámbito comercial y de investigación puede ser atribuido a su maniobrabilidad y capacidad para llevar pequeñas cargas [Kumar and Michael, 2012]. La maniobrabilidad es afectada por las dimensiones del vehículo, ya que entre más pequeños son presentan un comportamiento más ágil. Otros aspectos de control, como las dinámicas, se mantienen independientemente del tamaño. Por otra parte, su capacidad de carga ha demostrado ser suficiente para el desarrollo de múltiples aplicaciones entres las que se encuentran las de mapeo y exploración, las cuales requieren de equipamiento de sensores tales como cámaras RGB-D y escáneres laser. Sobre estos dos últimos sensores se presenta más información en la Sección 3.1.5.

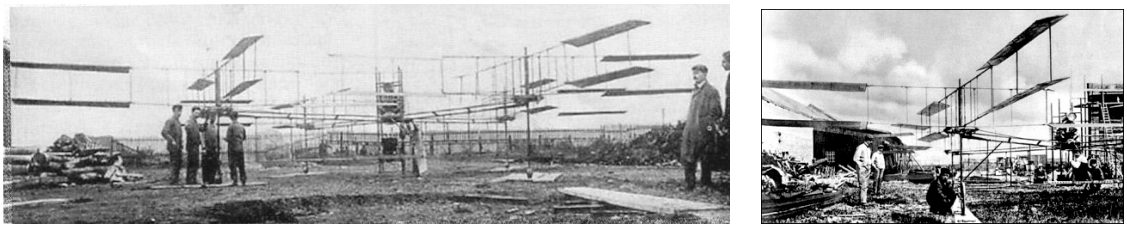
A continuación se presentan más detalles del cuadricóptero. En la Sección 2.6.1 se hace un breve repaso histórico, mediante la presentación de algunas de las primeras aeronaves de este tipo. Por otra parte, en la Sección 2.6.2, se introducen los fundamentos de su estructura, la forma en que se describe su posición y sus dinámicas.

2.6.1. Antecedentes del cuadricóptero

Los UAVs cuadrirotor comparten los origenes de los helicópteros, que son tan antiguos como los primeros diseños de Leonardo Da Vinci vistos en la Sección 2.2.1. Alrededor del año 1800, se presentaron diseños que mejorarían aspectos de sus precursores al incluir una fuente de energía. Tal es el caso del tornillo aéreo a vapor de Gustave Ponton D'Amecourt que incorporaba hélices separadas verticalmente y potenciadas mediante vapor [of flight comission, 2016]. A principios del siglo siguiente, a la par del desarrollo de aeronaves de ala fija, surgieron las primeras propuestas de aeronaves multi-rotor tripuladas. Entre las más importantes, por su similitud con los UAVs cuadrirotor de la actualidad [Domingues, 2009], se encuentran Gyroplane No. 1, Oemichen No. 2, el cuadricóptero de Bothezat y Convertawings Model A.

En el año 1907, en Francia, Louis and Jacques Breguet, en asociación con Charles Richet, construyeron la primera máquina capaz de elevarse del piso verticalmente con un piloto [Aviastar.org, 2016]. El diseño de la aeronave consistía en un estructura metálica que servía de soporte para el piloto y la fuente de energía, basada en gasolina (ver Figura 2.13). Cada esquina de la estructura estaba unida, mediante brazos metálicos, a un rotor con cuatro hélices de biplano. Además, dichas hélices fueros dispuestas de tal forma que un par de rotores opuestos diagonalmente giraba en sentido horario y el otro par en sentido antihorario. Este diseño de rotores se conserva hasta la fecha, como se verá en la Sección 2.6.2. Durante su primer vuelo, el 24 de agosto de 1907, Gyroplane 1 alcanzó la altura de 0.6 metros y un mes más tarde, el

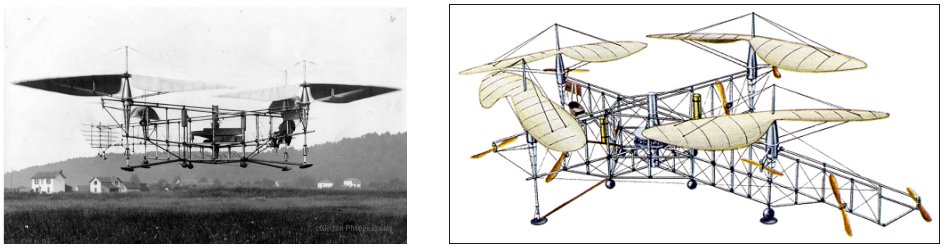
29 de septiembre, se elevó hasta los 1.5 metros, sin embargo nunca pudo desplazarse de manera horizontal. Un año más tarde, Breguet y Richet crearon Gyroplane 2 que montaba un motor Renault de 55 caballos de fuerza y usaba únicamente dos rotores. Gyroplane 2 llegó a tener vuelos exitosos, sin embargo un año más tarde fue destruida por un huracán, lo que alejó a los Breguet de estos desarrollos hasta 1930.



(a) (b)

Figura 2.13: Cuadricóptero Gyroplane No. 1 diseñado por Louis, Jacques Breguet y Charles Richet en 1907 [Aviastar.org, 2016]: (a) al centro se encuentra la estructura encargada de cargar la fuente de energía y el asiento del piloto y (b) rotor con cuatro hélices de biplano.

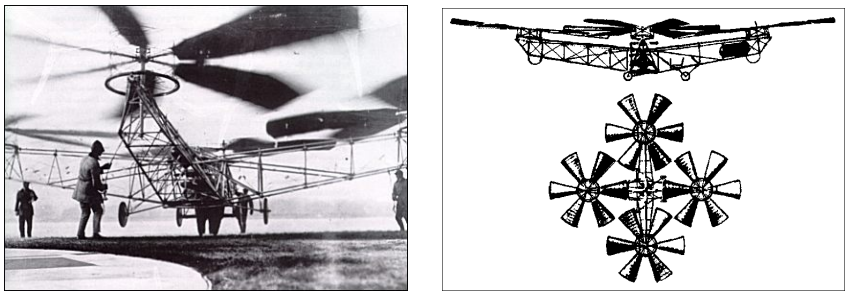
En 1920, Etienne Oemichen trabajó en el diseño de diferentes vehículos aéreos de tipo VTOL, de los cuales el más relevante fue Oemichen No. 2 [Aviastar.org, 2016]. Oemichen No. 1 poseía dos rotores propulsados por un motor de 25 caballos de fuerza, pero los resultados no fueron satisfactorios. En el desarrollo de Oemichen 2 se usó un diseño de estructura metálica en forma de crucifijo (ver Figura 2.14). Como medio de sustentación principal, en cada extremo de la estructura se incorporaron cuatro rotores conectados a una hélice doble con aspas en forma de paleta. En los costados de la aeronave se colocaron hélices de distintos tamaños que servían como medio de estabilización horizontal. Por último, en la parte trasera y delantera se colocaron otro par de hélices para ofrecer propulsión horizontal. Oemichen 2 realizó múltiples vuelos de prueba de 1920 a 1923 y en 1924 rompió diferentes records de distancia. También se le atribuye ser el primer helicóptero en realizar un vuelo de 1 km en un circuito cerrado.



(a) (b)

Figura 2.14: Cuadricóptero Oehmichen No. 2 diseñado por Etienne Oemichen en 1920: (a) en vuelo y (b) esquema [Aviastar.org, 2016].

En junio de 1921, George de Bothezat e Ivan Jerome fueron contratados por el Cuerpo Aéreo del Ejército de los Estados Unidos (USAAC[15], por sus siglas en inglés) para construir una aeronave capaz de volar verticalmente [Aviastar.org, 2016]. El producto de su trabajo fue una estructura metálica en forma de X, en donde cada extremo poseía un rotor de seis aspas (ver Figura 2.15). Las aspas estaban colocadas de manera separada entre sí e inclinadas hacia el centro de la aeronave. De manera adicional, se colocaron dos hélices horizontales que tenían la función de control direccional. Su motor fue un Bentley de 220 caballos de fuerza, que le permitió alcanzar una altura de 1.8 metros sobre el suelo llevando a una persona a bordo y 1.2 metros con dos personas. Estos resultados no fueron suficientes para los planes del cuerpo aéreo y con el tiempo se perdió interés.



(a) (b)

Figura 2.15: Cuadricóptero Bothezat diseñado por George de Bothezat e Ivan Jerome en 1922 [Aviastar.org, 2016]: (a) despegando y (b) esquema.

2.6.2. Dinámicas del cuadricóptero

El movimiento de un UAV cuadrirotor y por lo tanto de un UAV, es controlado por las velocidades angulares de sus cuatro rotores [Carrillo et al., 2012]. La velocidad angular, de manera similar a la velocidad lineal, mide el desplazamiento por unidad de tiempo para un movimiento circular (i.e., el ángulo de un círculo recorrido durante un segundo) y sus unidades son los radianes/segundo. Cada rotor cuenta con una hélice cuya función es convertir movimiento rotacional en empuje, mediante la diferencia de presión entre sus dos superficies. Este empuje, en el cuadricóptero, produce una fuerza vertical hacia arriba F_i de manera proporcional a la velocidad angular del rotor. No obstante, el empuje no es la única fuerza generada por los rotores, estos también producen un torque T_i, el cual, de acuerdo a la tercera ley de Newton, es la reacción a la acción de hacer girar los rotores en cierto sentido. En otras palabras, si el rotor hace que las hélices giren en sentido horario, también se producirá una fuerza opuesta que tienda a hacer girar al mismo rotor en sentido antihorario.

[15] *United States Army Air Corps*

En un cuadricóptero, los rotores se encuentran colocados en una estructura con forma de cruz y giran en una sola dirección. Si los cuatro giraran en la misma dirección, su torque ocasionaría que el cuadricóptero gire sin control. Con esto en mente, los rotores que giran en la misma dirección se colocan opuestos entre sí, un par en sentido horario y otro en sentido antihorario (ver Figura 2.16). Dicha configuración permite cancelar los efectos producidos por el torque de cada rotor y mantiene a la aeronave en una posición estable.

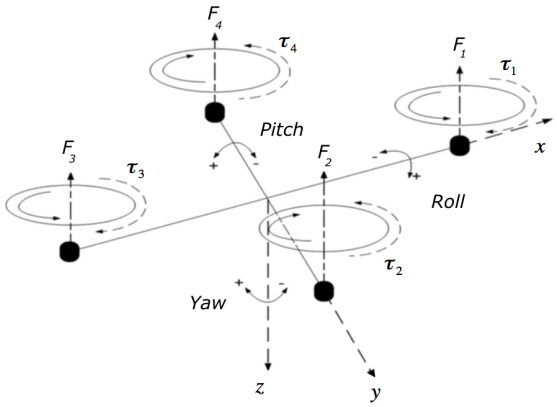


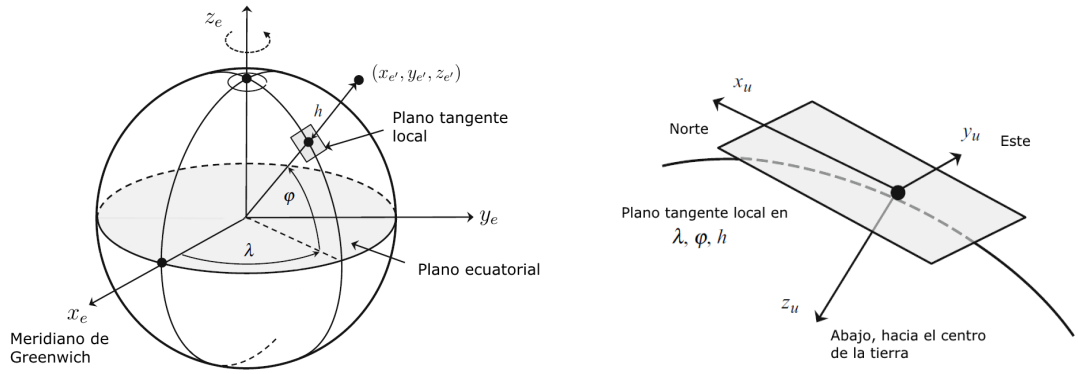
Figura 2.16: Estructura de un UAV cuadrirotor y marco de referencia *body-attached frame* [Selby, 2016].

Para describir la posición y orientación de una aeronave en el espacio, se consideran dos marcos de referencia con origen en el centro de gravedad de la aeronave [Dobrokhodov, 2015]. Ambos marcos son sistemas de coordenadas tridimensionales en el espacio euclidiano y se diferencian entre sí por el objeto al que están alineados. El primer marco, llamado *body-carried frame*, está alineado con los ejes del marco de referencia del plano tangente local (LTP[16], por sus siglas en inglés). Este plano es una superficie cuyos ejes x, y y z apuntan al norte, este y al centro de la tierra, respectivamente (ver Figura 2.17). El segundo marco, llamado *body-atacched frame*, está alineado con el cuerpo de la aeronave y rota con ella. Para definir la posición de una aeronave basta con obtener la traslación del marco *body-carried frame* con respecto al marco de referencia del LPT. En el caso de la orientación, se calculan tres ángulos entre ambos marcos de referencia. Estos ángulos, llamados *roll* (ϕ), *pitch* (θ) y *yaw* (ψ), representan las rotaciones de la aeronave con respecto a los ejes x, y y z del marco *body-carried frame* (ver Figura 2.16).

$(x_{e'}, y_{e'}, z_{e'})$ z_e x_e y_e

Mediante la modifcación de las velocidades angulares de los rotores, se puede desplazar a un cuadricóptero (ver Figura 2.18) [Carrillo et al., 2012]. El ascenso y el descenso son controlados mediante el incremento o disminución simultanea de la ve-

[16] *Local Tangent Plane*



(a) (b)

Figura 2.17: Sistema de coordenadas geodésico y plano tangente local [Dobrokhodov, 2015]: (a) sistema fijo de coordenadas centrado en la tierra (ECEF[17], por sus siglas en inglés) donde λ es la latitud geodésica, φ la longitud geodésica y h la altura altura y (b) plano tangente local.

locidad de los cuatro rotores. Los ángulos *roll* y *pitch* se consigue con una diferencia de velocidades entre dos rotores que giren en el mismo sentido. Mientras se produce esta diferencia, el par restante reduce su velocidad. Para el ángulo *roll* se usan los rotores 2 y 4, mientras que en el ángulo *pitch* los rotores empleados son 1 y 3. El movimiento *yaw* se obtiene como resultado de disminuir la velocidad de un par de rotores similares, mientras se incrementa la velocidad del otro par.

En la práctica estos movimientos producen lo siguiente, *yaw* hace que el rotor gire sobre su eje z, *roll* hace que se mueva a la derecha o izquierda y *pitch* lo hace ir adelante o atrás. Sumando el movimiento de ascenso y descenso, el cuadricóptero puede moverse en los ejes x, y, z y rotar en el eje z (*yaw*). Lo anterior da como resultado cuatro grados de libertad, i.e., no hay rotación sobre el eje x ni sobre el eje y, ya que para alcanzar los seis grados de libertad se necesitan al menos seis rotores [Jiang and Voyles, 2015]. Esta limitación puede ser solventada en determinados casos mediante vuelo agresivo [Mellinger et al., 2012], e.g., para pasar a través de una entrada angosta, este tipo de vuelo utiliza la inercia del movimiento para realizar maniobras que, en condiciones normales, no serían posibles.

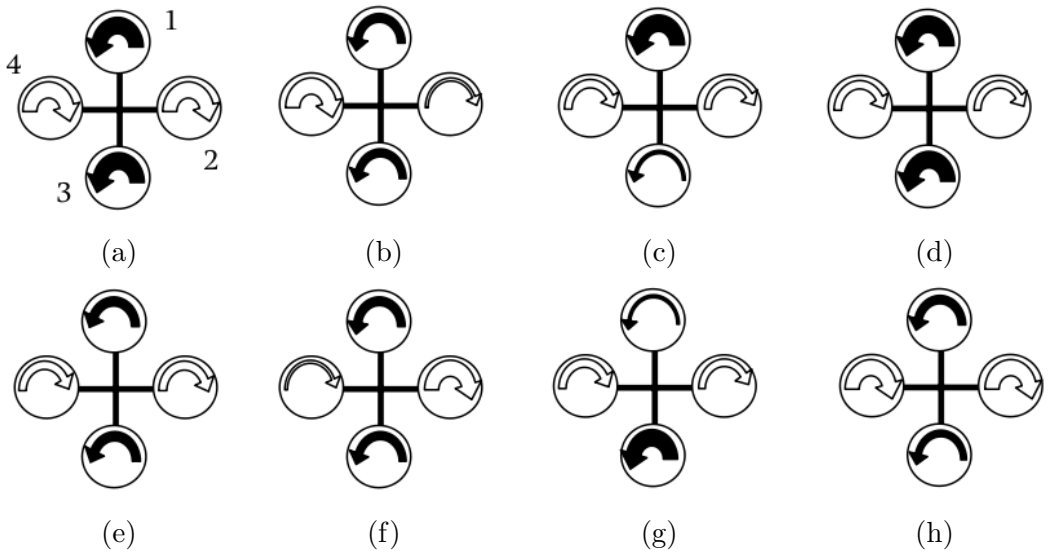


Figura 2.18: Dinámicas de un UAV cuadrirotor [Azfar and Hazry, 2011]: (a) ascenso (b) *roll* sentido horario, (c) *pitch* sentido horario, (d) *yaw* sentido horario, (e) descenso, (f) *roll* sentido antihorario, (g) *pitch* sentido antihorario y (h) *yaw* sentido antihorario.

Capítulo 3

Mapeo, planificación y exploración robótica

En el presente capítulo se introducen los conceptos necesarios para resolver el problema planteado en el Capítulo 1. En la Sección 3.1, se describen las bases de los mapas en robótica, la forma en cómo trabajan los algoritmos que los construyen y las representaciones del entorno. Una vez que un robot cuenta con un mapa del entorno, necesita poder desplazarse en él, procurando evitar colisiones con obstáculos. En robótica, a este concepto de crear un plan de la forma en cómo se moverá un robot, se denomina **planificación de movimiento** y se explicará en la Sección 3.2. Por último, en la Sección 3.3 se presenta el concepto de exploración robótica, el cual consiste en construir un mapa de un ambiente desconocido.

3.1. Mapeo en robótica

En esta Sección se presentan los temas relacionados con el mapeo en el contexto de la robótica. En la Sección 3.1.1 se introduce el concepto de mapeo en el campo de la robótica. En la Sección 3.1.2 se explican las dos representaciones principales del entorno: basadas en coordenadas y relacionales. A continuación, en la Sección 3.1.3 se describe el concepto de odometría visual, el cual es importante en el proceso de construcción de mapas. Con respecto a dicho proceso, en la Sección 3.1.4, se describe el funcionamiento de los algoritmos SLAM, los cuales son responsables de realizar las funciones de mapeo y localización. En la Sección 3.1.5 se describen los dispositivos más utilizados para la adquisición de imágenes en interiores. Dichos dispositivos proporcionan las mediciones que son utilizadas para construir las representaciones del entorno. Por último, en la Sección 3.1.6, se describen las representaciones del entorno en 3D.

3.1.1. Concepto e importancia

El mapeo en robótica se puede describir, de forma general, como la construcción de un mapa del entorno, realizada por un robot, utilizando información espacial obtenida durante el paso del tiempo [Wallgrn, 2009]. Esta información puede provenir de diversas fuentes, entre las cuales se encuentran los sensores externos y los sensores internos; estos últimos pueden proveer información de odometría (en la Sección 3.1.3 se presentará más información acerca de este concepto). El tipo de información que se adquiera influirá en la técnica que se utilice para construir el mapa. Adicionalmente, múltiples fuentes de información pueden ser combinadas para obtener una representación más precisa del entorno. Por otra parte, existen diversas maneras de representar el mapa del entorno, las cuales van desde utilizar directamente la información proporcionada por los sensores hasta construir estructuras de datos específicas para este propósito, e.g., Octomap [Hornung et al., 2013]. Cabe destacar que el término mapeo se encuentra relacionado, con mayor frecuencia, a la construcción de mapas en locaciones interiores, puesto que en exteriores los UAVs pueden disponer de información GPS para navegación y localización, mientras que en interiores, los UAVs deben estimar su posición con base en puntos de referencia del entorno.

Los mapas constituyen una de las partes más importantes en la robótica, debido a que brindan a los robots un modelo de su entorno. En tareas complejas, como de navegación y exploración, los robots necesitan conocer su posición en el entorno, para lograr esta tarea no basta con la información inmediata que proporcionan los sensores externos. En el caso de la navegación, los mapas permiten a un robot encontrar el camino hacia su objetivo y evadir los posibles obstáculos presentes en el entorno. En cuanto a exploración, el uso de mapas permite mantener un registro de las regiones del entorno que ya han sido visitadas por un robot, con el fin de enfocar el desplazamiento del robot hacia aquellas partes desconocidas del entorno.

Aunque el uso de mapas en otras áreas de la ciencia es muy antiguo, e.g., la cartografía en geografía, el mapeo en robótica comenzó a cobrar relevancia a mediados de los años 80. En esta época se presentaron dos enfoques principales, uno utilizando información métrica y otro empleando información topológica [Thrun et al., 2002]. El enfoque métrico se centró en las características geométricas del entorno, mientras el topológico se enfocó en las relaciones existentes entre lugares significativos del mapa. Otra clasificación presente durante aquella época estuvo relacionada con el sistema de coordenadas, centrado en el mundo y centrado en el robot. El sistema de coordenadas centrado en el mundo considera coordenadas globales, lo cual implica que los elementos del mapa no almacenan referencia alguna de cómo llegar a ellos, e.g., a partir de un punto inicial en el mapa. Con respecto al sistema de coordenadas centrado en el robot, se utilizan los datos de las mediciones de los sensores para indicar la forma en que se llega a cada elemento del mapa.

3.1.2. Representaciones del entorno y su clasificación

El constante desarrollo en el área de mapeo en el contexto de la robótica ha permitido que se establezcan clasificaciones posteriores más detalladas, como la presentada por Wallgrn, en la que se agrupan las representaciones del mapa en: basadas en coordenadas (información métrica) y relacionales (información topológica) [Wallgrn, 2009]. La representación basada en coordenadas se divide en: basadas en ocupación, geométricas y basadas en puntos de referencia. Mientras que la clasificación basada en relaciones se divide en grafos de vista y de ruta. Estas dos grandes clasificaciones contienen, de manera implícita, la separación entre coordenadas centradas en el mundo y en el robot.

En los mapas basados en coordenadas, los datos de ocupación, geométricos y de puntos de referencia son considerados en un sistema de coordenadas absoluto. La representación basada en ocupación, propuesta inicialmente por Moravec y Elfes, modela el espacio como un *grid* 2D de grano fino, cuyas celdas pueden estar ocupadas o libres [Moravec and Elfes, 1985]. Por otra parte, en la representación geométrica, se utilizan elementos primitivos como líneas, puntos y curvas, los cuales pueden ser considerados directamente como el mapa del entorno o combinados para formar objetos poligonales. Por último, la representación basada en puntos de referencia (*Landmarks*) considera elementos que sobresalen del entorno, aristas verticales de objetos tales como, ventanas, esquinas y puertas, entre otros. Como se puede observar, a diferencia del enfoque relacional, la representación basada en coordenadas no almacena información alguna que permita conocer la forma en que se llega de un elemento del mapa a otro.

En la representación relacional, las relaciones entre los elementos de un mapa son explícitas, haciendo que resulte natural el uso de grafos. En los grafos de vistas, los nodos (vistas) resultan de la información adquirida por los sensores en determinada posición del entorno. Las aristas representan la adyacencia espacial entre vistas, como resultado de mediciones espaciales adquiridas en orden consecutivo. En el caso de la representación relacional, utilizando un grafo de rutas, los nodos son lugares distintivos del entorno y las aristas denotan los caminos para llegar a ellos. Para determinar lugares distintivos, es posible utilizar puntos de referencia. En ambos casos, los nodos deben ser creados con proximidad suficiente para que el mapa pueda ser utilizado para navegar en el entorno. Sin embargo, dichos nodos deben estar suficientemente separados para que el tamaño computacional del mapa no crezca desmedidamente. Una estrategia para generar los nodos puede ser crearlos únicamente cuando la distancia de la posición actual, con respecto al último nodo, es mayor o igual a cierto valor.

Cada representación presenta diferentes ventajas y desventajas que pueden ser solventadas, mediante su aplicación conjunta para la construcción de un mapa preciso. Por ejemplo, la representación basada en puntos de referencia suele ser compacta y robusta, pero no es conveniente para la exploración o planificación de trayectorias,

debido a que no contiene información del espacio libre. En cambio, los mapas de ocupación sí proporcionan dicha información, pero son más susceptibles a errores en la percepción de los sensores. Estos errores pueden ser mitigados, mediante el uso de una representación relacional que permita integrar correcciones a los datos, ya que se conoce la forma en cómo se llegó a cada elemento del mapa. Estrategias como éstas constituyen la base de los algoritmos de localización y mapeo simultáneo que se describen brevemente en la Sección 3.1.4.

3.1.3. Odometría visual

En el campo de la navegación, el proceso de odometría se refiere a la estimación del cambio de posición de un vehículo, mediante el uso de información obtenida por los sensores de éste. Este tipo de información puede ser la rotación de las ruedas para el caso de los vehículos terrestres. Sin embargo, se pueden producir errores de medición cuando la rotación de las ruedas no produce desplazamiento alguno del vehículo, como es el caso de superficies deslizantes o cuando la rueda deja de estar en contacto con el terreno por irregularidades que este pueda presentar. El proceso de odometría visual es importante por sus aplicaciones al momento de construir mapas.

El proceso de odometría visual consiste en estimar la posición relativa de un vehículo, mediante el análisis de los cambios en imágenes sucesivas tomadas del entorno, mientras el vehículo se encuentra en movimiento. A pesar de que se han hecho avances en el campo de la odometría visual [Bachrach et al., 2012], este proceso consta básicamente de los siguientes pasos [Nistér et al., 2004]:

1. **Detección de características**: consiste en extraer elementos del entorno que sean estables y observables desde diferentes puntos de vista y ángulos. Los tipos de características que pueden ser utilizados, de acuerdo con [Muhammad et al., 2009], son puntos y líneas.

2. **Coincidencia de características**: en esta etapa se comparan características entre pares de imágenes. Se considera que las características de una imagen coinciden con las de otra, cuando existe como máximo un 10 % de diferencia entre estas.

3. **Estimación del movimiento**: se estima la posición relativa del vehículo, comparando tres *frames* de la etapa anterior. Nistér et al, utilizan el algoritmo Random Sample Consensus (RANSAC) para realizar esta tarea. Dicho algoritmo estima parámetros para ajustar un modelo a datos experimentales [Fischler and Bolles, 1981].

3.1.4. Construcción de mapas

La construcción de mapas puede ser referida como el problema de localización y mapeo simultaneo (SLAM[1], por sus siglas en inglés). El problema se compone de dos partes principales. La primera parte involucra la estimación de la posición de un robot en el entorno, dado un conjunto de puntos de referencia. La segunda parte concierne a la creación de puntos de referencia en el entorno, a partir de la posición del robot. Este problema es de tipo "¿El huevo o la gallina?" debido a que se necesita un mapa para conocer la posición del robot y se necesita la posición del robot para generar los puntos de referencia del mapa [Stachniss, 2014]. Adicionalmente, la generación de mapas implica lidiar con cuestiones de incertidumbre en las mediciones de los sensores. Otro problema involucrado es el problema de correspondencia o de asociación de datos, el cual consiste en determinar si dos mediciones tomadas en distintos instantes del tiempo corresponden al mismo elemento físico [Thrun et al., 2002].

El primer problema que se enfrenta cuando se trata con entornos reales es el de incertidumbre. Este término significa que no se puede estar 100 % seguro de que la información que se maneja es correcta. Este problema se presenta en sensores, ubicación de robots en el entorno y puntos de referencia. En los sensores, la incertidumbre se puede deber a ruido en el entorno o en el propio sensor. Como se mencionó anteriormente, la localización de un robot depende del mapa, el cual es desconocido. Una situación similar ocurre con los puntos de referencia que constituyen el mapa, ya que para crearlos se requiere asociar las mediciones de los sensores al mapa, pero esta asociación también es desconocida.

Con respecto al problema de asociación de datos, es difícil determinar si dos observaciones del entorno en distintos tiempos corresponden al mismo elemento. Este problema se debe a los errores acumulativos provocados por la incertidumbre en la posición del robot [Thrun et al., 2002]. El proceso de asociación de datos, descrito de forma general, consiste en tres pasos (ver Figura 3.1). Primero, se realizan observaciones del entorno, las cuales son mediciones utilizando los sensores del robot. A continuación, se utiliza una predicción de la trayectoria del robot para predecir nuevas observaciones. Por último, se seleccionan las observaciones proporcionadas por los sensores que mejor coincidan con las observaciones predichas [Grisetti et al., 2009].

Una consecuencia de no poder determinar con exactitud la correspondencia entre dos observaciones es la dificultad de detectar ciclos. Los ciclos son aquellos casos en los cuales, por la estructura del entorno o el algoritmo de planificación de trayectoria, el robot pasa nuevamente por una región que ya había visitado. El hecho de no detectar un ciclo puede llevar a construir nuevamente partes del entorno y, en el peor de los casos, el robot puede perder la capacidad de ubicarse en el entorno. Sin embargo, la presencia de ciclos en el entorno contribuye a reducir la incertidumbre en

[1] *Simultaneous Localization and Maping*

las estimaciones de los puntos de referencia [Stachniss, 2012a].

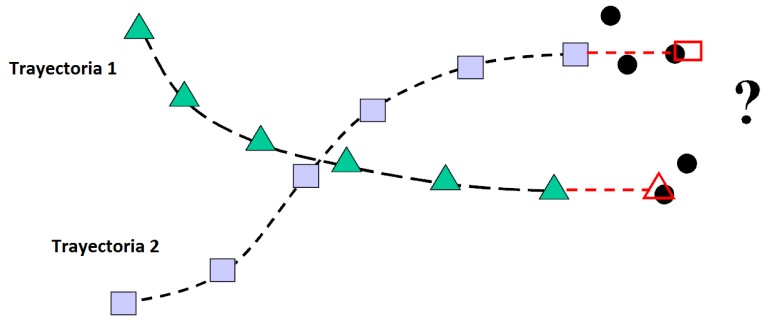


Figura 3.1: Esquema del problema de asociación de datos. Se presentan dos trayectorias distintas. Las líneas punteadas rojas representan trayectorias predichas. Los puntos en negro son observaciones del entorno, utilizando los sensores del robot. El cuadro y el triángulo sin relleno son observaciones predichas. Las observaciones de sensores que se añaden a la trayectoria son aquellas que mejor coinciden con las observaciones predichas [Collins, 2012].

Los algoritmos SLAM enfrentan diversos retos relacionados con la incertidumbre del ambiente. Es por ello que estos algoritmos introducen modelos probabilísticos que consideran explícitamente este aspecto. Estos algoritmos pueden clasificarse en tres tipos principales: basados en grafos, basados en el filtro extendido de Kalman (EKF[2], por sus siglas en inglés), el cual es un tipo de filtro bayesiano, y basados en el filtro de partículas. A continuación, se describe brevemente el algoritmo basado en grafos, debido a su importancia, aunque desde hace algunos años ha crecido el interés de la comunidad en EKF [Thrun and Leonard, 2008].

El algoritmo basado en grafos utiliza información espacial para construir un grafo que modela un error, el cual puede ser minimizado mediante algoritmos de optimización para determinar, con mayor precisión, los elementos del mapa. En un algoritmo basado en grafos de posiciones, existen dos tipos de nodos [Thrun and Leonard, 2008]: 1) los nodos que corresponden a posiciones del robot durante el mapeo del entorno y 2) los nodos asociados a puntos de referencia en el mapa. En el primer tipo de nodos, las aristas son datos espaciales que indican cómo se desplazó el robot de una posición a otra, tales como datos de odometría. En el caso de los puntos de referencia, los nodos del grafo son creados entre cada par de nodos de posición y las aristas representan el error de la observación (ver Figura 3.2). A partir de lo anterior puede aplicarse un algoritmo de minimización de errores, como el de mínimos cuadrados, en las aristas del grafo [Stachniss, 2012b].

[2] *Extended Kallman Filter*

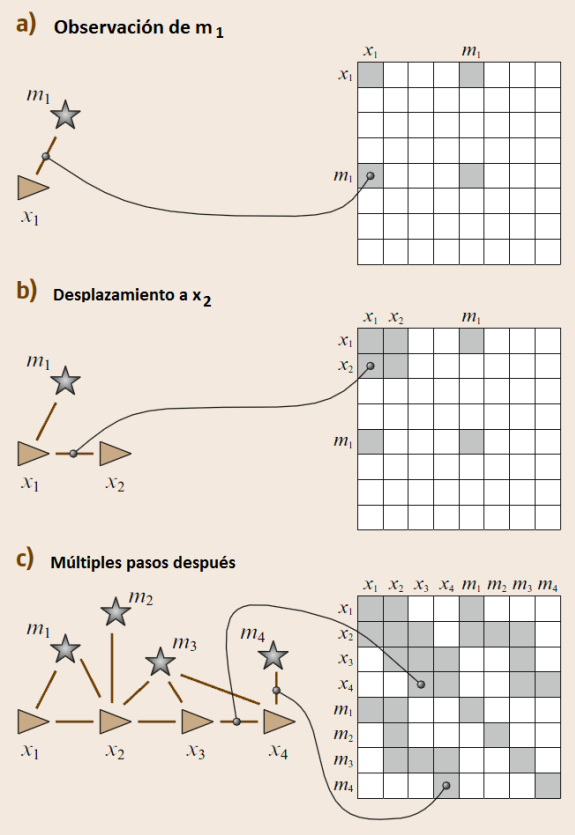


Figura 3.2: Algoritmo SLAM basado en grafo de posiciones, donde x_i es la posición del robot y m_i denota los puntos de referencia del mapa [Thrun and Leonard, 2008]

.

3.1.5. Dispositivos de adquisición de imágenes en interiores

Existe una amplia variedad de sensores que pueden ser montados en robots, pero la cantidad es menor cuando se trata de mapeo en entornos interiores. Dado que los robots que trabajan en este tipo de entornos son pequeños, también poseen menores capacidades de carga en comparación con aquellos que trabajan en exteriores. Entre los sensores más empleados en robots, que cuentan con restricciones de tamaño, se encuentran escáneres láser, cámaras estereoscópicas y recientemente cámaras RGB-D. Estos sensores tienen en común la capacidad de proporcionar información de profundidad, la cual permite construir mapas 3D, aunque la exploración se realice en 2D.

El escáner láser 3D, también llamado **LiDAR** (*Light Detection and Ranging*) o **LADAR** (*Laser Detection and Ranging*) se caracteriza por ofrecer gran precisión y altas tasas de datos. Este dispositivo emite luz láser para medir la distancia a la que se encuentran los objetos. El escáner láser utiliza espejos rotativos para reflejar el haz de luz y con ello extender su percepción del entorno de 1D hasta 3D (ver Figura 3.3) [Nüchter, 2008]. El resultado de escanear una superficie es una nube de puntos, i.e.,

un conjunto de puntos en 3D. En la Sección 3.1.6 se presenta más información acerca de las nubes de puntos y su utilidad en la construcción de mapas.

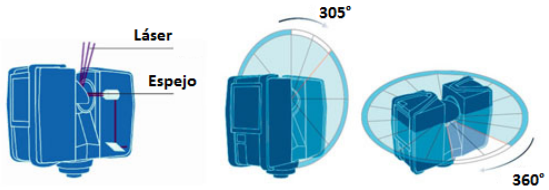


Figura 3.3: Esquema de un escáner láser. Algunos modelos comerciales incorporan dispositivos que le permite rotar horizontal y verticalmente para cubrir una mayor zona [CAP4D, 2016].

Otro dispositivo de percepción del entorno es la cámaras estereoscópica, que permite obtener datos de profundidad de las imágenes que capturan. Este dispositivo se compone de dos cámaras colocadas a una distancia fija conocida. Para determinar la profundidad de los elementos de la imagen 3D, se comparan las imágenes individuales adquiridas por los lados izquierdo y derecho de la cámara estereoscópica. La correspondencia entre imágenes se determina, entre otras cosas, apoyándose en el valor de disparidad de la cámara (la distancia de separación entre una cámara y otra) [Nüchter, 2008]. Este tipo de cámaras es más accesible que los sensores láser en cuestiones de precio y tamaño, pero posee menor precisión.

En años recientes, las cámaras RGB-D han cobrado relevancia en la construcción de mapas para interiores. Este crecimiento en popularidad se debe a que, por su precio, son accesibles para todo tipo de consumidores, e.g., Microsoft Kinect. Este tipo de cámaras proporciona datos de color y profundidad por cada pixel de las imágenes obtenidas [Khoshelham and Elberink, 2012]. Para ello, las cámaras RGB-D se componen de una cámara RGB que captura imágenes a color, un emisor láser que produce un *grid* 2D de luz infraroja y una cámara infraroja que captura la luz emitida y la compara con un patrón de referencia para determinar la distancia a la que se encuentra cada pixel de la imagen (ver Figura 3.4).

3.1.6. Representaciones 3D del entorno

Como se vio en la Sección 3.1.2, existen diversas representaciones del entorno, las cuales pueden ser extendidas a 3D. Incluso algunas trabajan directamente en 3D, como la basada en puntos de referencia, pero no todas proporcionan información acerca del espacio libre y ocupado. Es por ello que los mapas 3D, en su mayoría, son mapas de ocupación, cuyos elementos son llamados vóxeles. Un vóxel es la unidad mínima de un *grid* 3D, regularmente un cubo. Sin embargo, el uso de vóxels puede consumir mucho espacio, es por ello que se han desarrollado diversas representaciones que bus-

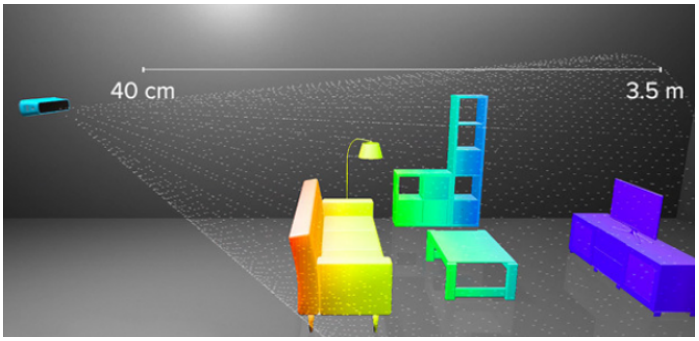
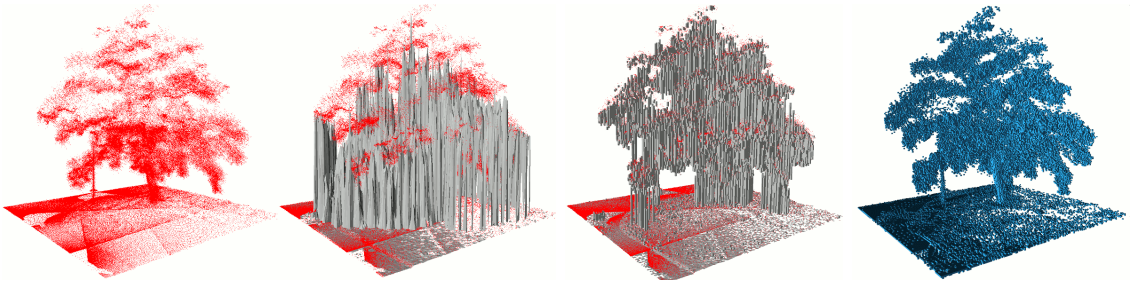


Figura 3.4: Esquema informativo de la cámara RGB-D de Structor Sensor [Occipital, 2016].

can solventar este problema (ver Figura 3.5) [Hornung et al., 2013].



(a) (b) (c) (d)

Figura 3.5: Representaciones de mapas en 3D: (a) nube de puntos, (b) mapa de elevación, (c) superficie multi-nivel y (d) basada en *octrees* [Hornung et al., 2013].

La fuente de información de los mapas 3D es las nubes de puntos, las cuales son mediciones proporcionadas por sensores 3D. Estas mediciones entregan un conjunto de puntos en el plano, con coordenadas x, y, z relativas a la posición del sensor. Los puntos pueden tener información de color, como la obtenida por las cámaras RGB-D. Una primera representación del entorno podría ser realizada, tomando directamente estos puntos. Entre los inconvenientes de emplearlos se encuentran la ineficiencia en espacio de almacenamiento y la incapacidad de representar el espacio libre de manera explícita. Como se mencionó previamente en la Sección 3.1.2, una representación explícita del espacio libre es necesaria para poder distinguir las partes exploradas del entorno, de las inexploradas.

Una de las representaciones que permiten crear mapas en 3D es la basada en mapas de elevación. Estos mapas almacenan la altura estimada de los objetos del entorno en cada celda de un *grid* de ocupación 2D. La forma en que las nubes de puntos son convertidas a alturas es mediante actualización probabilística, utilizando el filtro de Kalman [Burgard et al., 2013]. Sin embargo, este tipo de mapas tiene el inconveniente de representar únicamente un valor de altura por cada celda, por lo

tanto no es capaz de identificar correctamente objetos verticales. Por ejemplo, una mesa se vería solamente como un cubo, ya que se omite el espacio libre entre las patas de la mesa y se mapea únicamente la parte más alta.

Una mejora de los mapas de elevación son los mapas de superficie multi-nivel que incorporan **parches de superficie** (*surface patches*). Los parches de superfie son regiones que se almacenan en cada celda de un *grid* 2D y que están compuestas por la altura promedio, la varianza de la altura y un valor de profundidad [Triebel et al., 2006]. Este mapa tiene la ventaja de poder representar múltiples superficies por celda, obteniendo una representación más precisa del espacio ocupado en comparación con los mapas de elevación (ver Figura 3.6). Una desventaja de los mapas de superficie multi-nivel es que no mapean el espacio desconocido, lo cual se debe a que no mapean explícitamente el espacio libre.

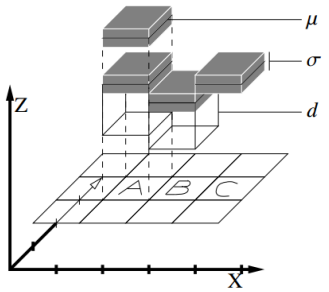


Figura 3.6: Mapa de superficie multi-nivel, donde μ es la altura media, σ la varianza de la altura y d el valor de profundidad [Triebel et al., 2006].

Un último tipo de mapa 3D, inspirado en *grid* de ocupación, es el basado en *octrees*. Esta representación es utilizada en Octomap, una biblioteca que implementa un *grid* de ocupación en 3D para construir un modelo del espacio libre y ocupado del entorno [Hornung et al., 2013]. Los *octrees* [Meagher, 1982] son una estructura que permite representar objetos en 3D. Dicha representación se obtiene considerando al entorno como un cubo formado por ocho cubos internos, cada uno subdividido recursivamente en el mismo número de elementos (ver Figura 3.7). En esta estructura, se puede compactar el espacio de almacenamiento y además es posible consultar el mapa a diferentes resoluciones (ver Figura 3.8).

3.2. Planificación de movimiento en robótica

En esta sección se describen las bases de la **planificación de movimiento** en robótica. En la Sección 3.2.1 se presenta el concepto de **planificación de movimiento** y los términos relacionados. En la Sección 3.2.2 se describe el diagrama de Voronoi por su importancia en el presente trabajo de tesis y por su utilidad para realizar

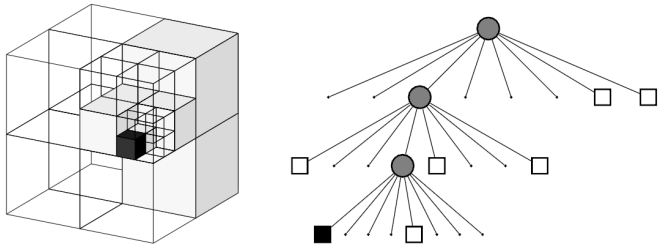
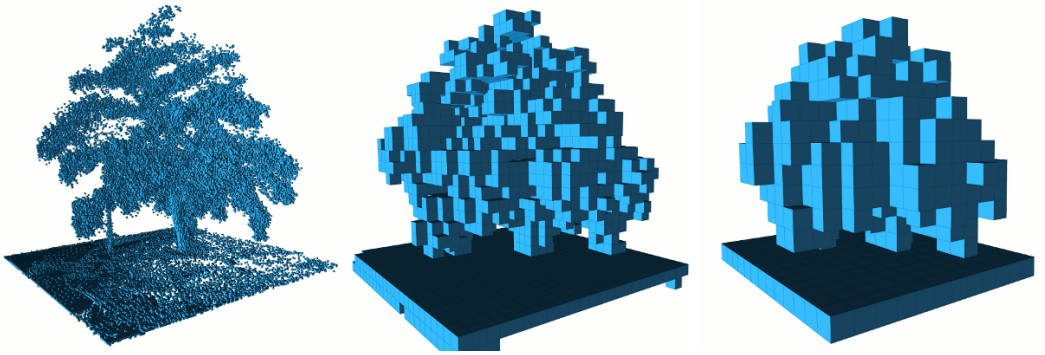


Figura 3.7: Partición basada en *octrees* utilizada en Octomap [Hornung et al., 2013]. A la izquierda, se muestra la representación espacial y a la derecha se ilustra la representación en grafo.



(a) (b) (c)

Figura 3.8: Consultas del mapa producido por Octomap en diferentes resoluciones: (a) 0.08 metros, (b) 0.64 metros y (c) 1.28 metros [Hornung et al., 2013].

planificación de movimiento combinatorio, como se verá en la Sección 3.2.3. En la Sección 3.2.4, se explican los algoritmos basados en muestreo, que a diferencia de los combinatorios, no utilizan representaciones exactas del entorno. Finalmente, en la Sección 3.2.5, se describe la biblioteca OMPL que implementa diversos algoritmos de **planificación de trayectoria** basados en muestreo.

3.2.1. Concepto e importancia

Darle la capacidad a un robot para moverse en un entorno, de forma autónoma, es un problema de gran importancia en robótica, pues está relacionado con su habilidad para ejecutar tareas sin intervención humana durante el proceso. Esto significa que es necesario buscar formas mediante las cuáles un robot, apoyándose de sus sensores, pueda desplazarse hasta su objetivo. Para hacer un plan de movimiento, deben considerarse diversas restricciones. En primer lugar, en situaciones reales, el entorno no está libre de obstáculos, así que el robot podría chocar contra ellos. Además, los robots tienen determinados movimientos permitidos, los cuáles limitan los caminos posibles hacía el objetivo. Y por último, debe tenerse en cuenta el estado actual del robot, e.g., si se encuentra en movimiento, la velocidad a la que se desplace podría

no permitirle girar a tiempo para entrar por una puerta. Este problema ha motivado la creación de diversos algoritmos de planificación de trayectoria, los cuales buscan proporcionar soluciones que consideren dichas restricciones.

El término **planificación de movimiento** (*motion planning*) se refiere a indicarle a un robot como moverse, de un punto inicial a un punto final, evitando las posibles colisiones con los objetos que forman parte del entorno [O'Rourke, 1998]. La **planificación de movimiento** involucra la generación de una ruta libre de obstáculos por la cual pase el robot. A esta actividad se le denomina **planificación de ruta** (*path planning*). Cuando dicha ruta es construida, con base en las dinámicas del robot, se dice que se realiza **planificación de trayectoria** (*trayectory planning*). La distinción entre estos tres términos varía dependiendo del autor [LaValle, 2006, Choset, 2005] y en ocasiones son utilizados como sinónimos. Es por ello que en esta sección, los algoritmos descritos se tratarán indistintamente como algoritmos de **planificación de trayectoria**. Pero cuando se haga referencia al problema, en general, de indicarle a un robot cómo ir de un punto inicial a un punto final, se referirá como **planificación de movimiento**.

El problema de **planificación de movimiento** consiste en realizar una búsqueda en el espacio de configuraciones. Este espacio, también llamado *C-Space*, está compuesto por todas las posibles configuraciones del robot en el ambiente, i.e., las posiciones que el robot puede llegar a tener, de acuerdo a sus movimientos permitidos. Al conjunto de configuraciones en el espacio libre se le denomina C_{free}. Por el contrario, aquellas posiciones en las cuales el robot intersecta a los obstáculos del ambiente, forman el espacio llamado C_{obs}. Cada movimiento independiente del robot es un grado de libertad, por lo tanto la dimensión mínima del espacio de configuraciones es el número de grados de libertad del robot [Choset, 2005].

Los algoritmos de **planificación de trayectoria** consideran ambientes estáticos. Por su funcionamiento, estos algoritmos se dividen en combinatorios y basados en muestreo. Adicionalmente pueden ser clasificados en dos categorías: algoritmos *online* y *offline* [Choset, 2005]. En un algoritmo *online*, el plan de movimiento es creado durante la ejecución del mismo. Por otra parte, en un algoritmo *offline*, el plan se crea con base en un modelo conocido del entorno y se ejecuta posteriormente. En términos prácticos, es posible utilizar un algoritmo *offline* para replanear, a medida que se realiza la ejecución. Sin embargo, los algoritmos *online* son construidos con este enfoque en mente, permitiendo la integración directa de mediciones, mientras se realiza el proceso de planificación.

3.2.2. Diagrama de Voronoi

En robótica, el diagrama de Voronoi puede ser utilizado para tratar problemas de **planificación de movimiento**, como se verá más adelante en la Sección 3.2.3. Es-

te diagrama es una estructura geométrica que subdivide un espacio, asignando cada punto a su sitio más cercano. En un diagrama de Voronoi, los sitios son puntos de interés en el espacio. Por ejemplo, "considere un bosque extenso que debe ser vigilado por múltiples torres de observación, donde el encargado de cada torre es responsable de extinguir el fuego que se encuentre más cercano a su posición, que a cualquier otra torre". En este ejemplo presentado por Joseph O'Rourke, las torres son los sitios del diagrama de Voronoi y las líneas que delimitan las regiones poligonales que vigila cada torre, junto con los sitios, son el diagrama de Voronoi [O'Rourke, 1998]. A todos los puntos que se encuentran en las regiones de cada torre se les denomina vecinos. Debido a que las líneas representan el límite entre regiones, todos los puntos sobre dichas líneas están a la misma distancia de, al menos, dos sitios (ver Figura 3.9).

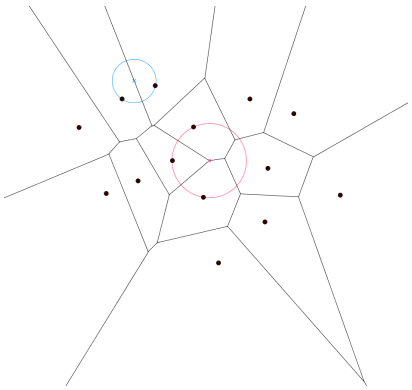


Figura 3.9: Diagrama de Voronoi, cuyos vértices del diagrama están a la misma distancia de tres sitios y las aristas a la misma distancia de dos sitios.

El diagrama de Voronoi se define formalmente de la siguiente manera. Sea $P = \{p_1, p_2, ..., p_n\}$ un conjunto de puntos en el plano euclidiano de dimensión dos, se define la región de Voronoi $V(p_i)$ como $V(p_i) = x : |p_i - x| \leq |p_j - x| \forall j \neq i$. Lo anterior significa que la región de Voronoi de un sitio i está compuesta por todos los puntos del plano que se encuentren a una distancia menor o igual de este sitio, que de cualquier otro sitio. Partiendo de lo anterior, los sitios y el conjunto de puntos, que tienen más de un vecino, son el diagrama de Voronoi V(P) [O'Rourke, 1998].

Una forma de construir un diagrama de Voronoi en 2D es mediante el algoritmo de Fortune, también conocido como algoritmo de barrido de plano, cuya complejidad es $O(n \log n)$ [Berg, 2008]. Este algoritmo utiliza una línea de barrido ℓ para procesar los sitios de forma horizontal, iniciando por el sitio con coordenada y más alta y descendiendo hasta procesar todos los sitios. El algoritmo garantiza que la parte del diagrama, generada por los sitios arriba de la recta, no cambia una vez procesada. En la Figura 3.10, se muestra la línea ℓ utilizada para barrer el plano XY. Durante el barrido, cada sitio se comporta como el foco de una parábola y la línea actúa como directriz. Al conjunto de arcos parabólicos, que aparecen durante el barrido de

la recta, se le llama "línea de playa". Dado que, en una parábola, la distancia del foco a cualquier punto de la parábola es equidistante a la directriz, la intersección de dos arcos formará una arista del diagrama de Voronoi. Además, cuando tres arcos se intersecten, se formará un vértice del diagrama.

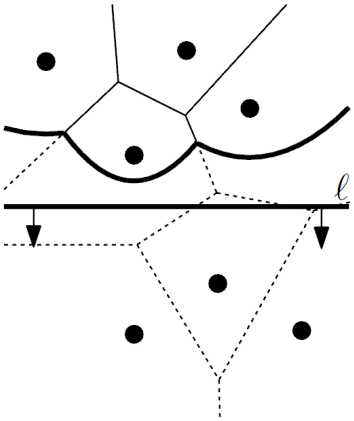


Figura 3.10: Construcción del diagrama de Voronoi, utilizando el algoritmo Fortune [Berg, 2008].

Por la definición del diagrama de Voronoi, es claro que éste crea un partición del espacio, donde las regiones son de tamaños distintos. Para fines prácticos, dicho diagrama puede estar contenido en una rectángulo, sin embargo, cuando carece de un contenedor, las aristas cuyos sitios están en el cierre convexo se prolongan hasta un punto en el infinito. El presente trabajo aprovecha la partición del espacio realizado por el diagrama de Voronoi, en particular las regiones. Empero, para otros trabajos que emplean este diagrama, la información principal reside en las aristas; un ejemplo de esto se presenta en la Sección 3.2.3.

3.2.3. Algoritmos combinatorios

Los algoritmos combinatorios de **planificación de trayectoria** ofrecen soluciones eficientes para situaciones específicas, en las cuales se consideran representaciones 2D exactas del entorno. Sin embargo, este tipo de algoritmos no siempre puede ser implementado de forma práctica. Entre los algoritmos combinatorios más representativos, se encuentran los basados en grafo de visibilidad, descomposición en celdas exactas y diagrama de Voronoi explicado previamente (ver Sección 3.2.2). Estos algoritmos construyen una ruta continua a través del espacio libre. En las descripciones siguientes de algoritmos combinatorios se considera al robot como un punto en el espacio. Dicha consideración es posible asumiendo que el robot es un objeto con forma poligonal convexa y aplicando la suma Minkowski entre el robot y los obstáculos (con forma poligonal no necesariamente convexa) [Berg, 2008]. La idea general de esta operación

es incrementar el tamaño de los obstáculos, adicionando las dimensiones del robot hasta un punto dentro del mismo (ver Figura 3.11).

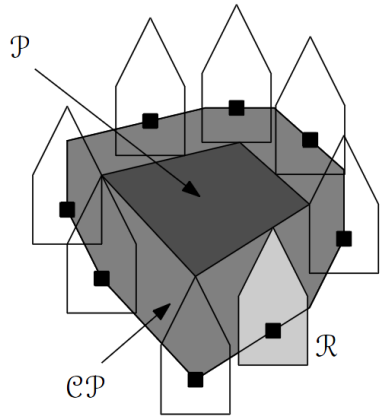


Figura 3.11: Suma Minkowski, donde $\mathcal{R}$ es el robot con forma poligonal convexa, $\mathcal{P}$ es un obstáculo y $\mathcal{CP}$ es el espacio de configuraciones de dicho obstáculo, tal que $\mathcal{R}$ intersecta a $\mathcal{P}$ [Berg, 2008].

Los algoritmos de planificación de trayectoria buscan privilegiar la separación entre los obstáculos y el robot, pero este enfoque no obtiene la ruta más corta. Una estrategia para lograrlo es construir un grafo de visibilidad. Un algoritmo de **planificación de trayectoria**, basado en grafo de visibilidad, considera una representación poligonal simple de los obstáculos presentes en el ambiente. El proceso de construcción del grafo de visibilidad produce un resultado como el que se muestra en la Figura 3.12, donde los nodos son los vértices de los polígonos y las aristas son los segmentos de recta que pueden ser trazados de cada vértice al resto de ellos, sin intersectarse con algún obstáculo [O'Rourke, 1998]. Una vez construido el grafo, basta con aplicar un algoritmo de búsqueda del camino más corto, e.g., el algoritmo de Dijkstra [Dijkstra, 1959], para obtener la ruta que deberá seguir el robot hasta su objetivo. El algoritmo de **planificación de trayectoria** basado en grafo de visibilidad tiene como inconvenientes que requiere una representación poligonal y que la ruta generada pasa muy cerca de los obstáculos, lo cuál podría no ser conveniente por cuestiones de seguridad del robot.

Otro enfoque para generar rutas es la descomposición del espacio libre en celdas exactas, esto se puede realizar de manera triangular, vertical (también llamada descomposición trapezoidal) o cilíndrica [LaValle, 2006], como se muestra en la Figura 3.13. El término **descomposición** se refiere a dar un partición del espacio libre, que facilite el cómputo de caminos entre dos puntos dentro de una misma celda y dos celdas vecinas. También debe permitir determinar, de manera sencilla, a qué celda pertenece un punto dado. Por ejemplo, en la descomposición vertical, se utilizan los vértices de los obstáculos poligonales para trazar semirectas que se extienden vertical-

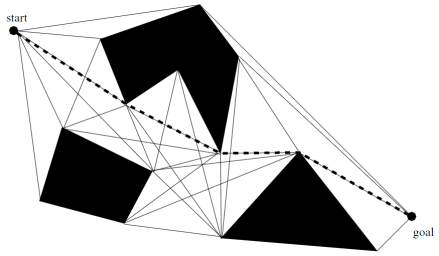
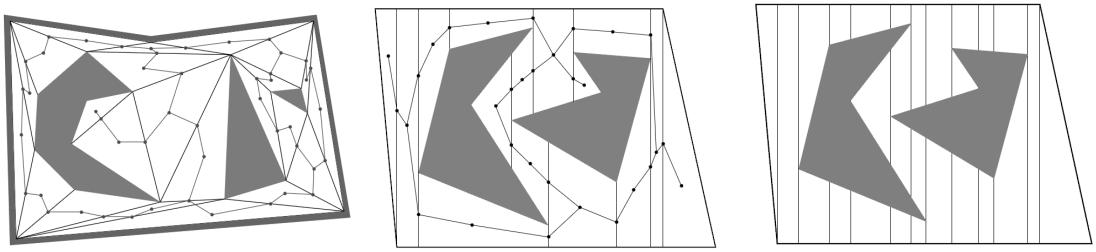


Figura 3.12: Grafo de visibilidad que permite encontrar la ruta más corta entre dos puntos, considerando una representación poligonal del ambiente [Choset, 2005].

mente hasta intersectar a un polígono o la caja contenedora. El resultado son celdas trapezoidales donde los centroides y la mitad de cada lado vertical de las celdas se convierten en nodos de un grafo. Las aristas del grafo corresponden a los segmentos de recta que conectan los nodos en los centroides de cada celda con los nodos presentes en la misma celda. Este algoritmo de planificación de trayectoria, a diferencia del anterior, mantiene al robot a una distancia segura de los obstáculos pero no provee la ruta más corta.



(a) (b) (c)

Figura 3.13: Tipos de descomposición en celdas exactas: (a) triangular, (b) vertical y (c) cilíndrica, en la cual a diferencia del tipo vertical, las líneas continúan hasta cruzar cada polígono [LaValle, 2006].

La última forma para realizar la **planificación de movimiento**, de manera combinatoria, es mediante el diagrama de Voronoi generalizado (GDV[3], por sus siglas en inglés). Como se vió en la Sección 3.2.2, el diagrama de Voronoi es capaz de proporcionar una partición del espacio para un conjunto de puntos, llamados sitios. En un GDV (ver Figura 3.14) las regiones de Voronoi son el conjunto de puntos más cercanos a un obstáculo i que a cualquier otro obstáculo j. Por lo tanto, las aristas son el conjunto de puntos que se encuentran a la misma distancia de dos obstáculos y los nodos son las intersecciones entre dichas aristas [Choset, 2005]. Dadas las características del GDV, un robot puede desplazarse sobre sus aristas hasta alcanzar un

[3]Generalized Voronoi Diagram

determinado objetivo. Las rutas en este diagrama además garantizan una distancia de separación con respecto a los obstáculos, esta característica no está presente en los grafos de visibilidad.

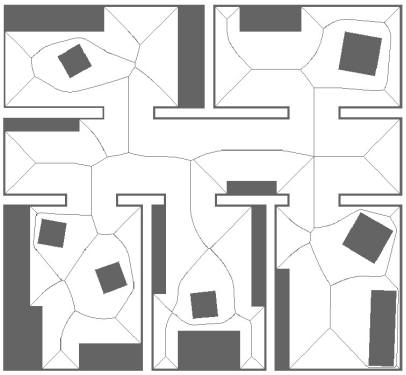


Figura 3.14: Diagrama de Voronoi Generalizado (GDV).

Los algoritmos basados en grafo de visibilidad y descomposición de celdas exactas son algoritmos *offline*, pues requieren de un modelo existente del entorno para generar una ruta. En cambio, el algoritmo basado en GDV es esencialmente *online*, debido a que es posible construir el diagrama, utilizado para planear, a medida que se ejecuta un plan previo. Esta construcción se logra a través de la integración de nuevas mediciones de los sensores [Choset, 2005]. Una construcción progresiva claramente no es posible en el algoritmo basado en grafo de visibilidad, pues se requiere encontrar la ruta más corta entre dos puntos, lo cual implica conocer todo el mapa.

3.2.4. Algoritmos basados en muestreo

Los algoritmos basados en muestreo, a diferencia de los combinatorios, no consideran representaciones exactas del entorno. En cambio, utilizan configuraciones del espacio libre, llamadas muestras, que al ser conectadas producen rutas libres de colisiones [Choset, 2005]. Estos algoritmos se clasifican en **consulta múltiple** (*multi-query*) y **consulta única** (*single-query*) [Kavraki and LaValle, 2008]. En el primer caso, se construye un grafo que contiene rutas en el espacio libre y que puede ser utilizado para realizar consultas. Por otro lado, en los algoritmos de consulta única, el enfoque se centra en construir una estructura que permita alcanzar un objetivo de manera rápida.

Un elemento presente, tanto en los algoritmos de **consulta múltiple** como en los de **consulta única**, es el detector de colisiones. Éste es el responsable de validar que un robot no se intersecte con objetos del entorno ni partes de sí mismo. Esta última intersección puede ocurrir si el robot posee múltiples articulaciones. La detección de colisiones puede ser realizada, de manera general, en dos fases: una llamada **fase amplia** (*broad phase*) y otra llamada **fase angosta** (*narrow phase*) [Choset, 2005]. En la **fase amplia**, se calculan las intersecciones entre las cajas contenedoras del robot

y los obstáculos, con el propósito de evitar una revisión exhaustiva de colisiones entre objetos muy lejanos entre sí. Por otra parte, en la **fase angosta**, si se revisan las colisiones, utilizando la estructura completa de los objetos.

A continuación, se describen el algoritmo PRM y el basado en RRTs. Estas dos propuestas son las más representativas de los algoritmos basados en muestreo. PRM es un algoritmo de **consulta múltiple**, mientras que el basado en RRTs es un algoritmo de **consulta única**.

PRM

Uno de los algoritmos más importantes, basados en muestreo, es el algoritmo PRM (*Proabilistic Road Map*) propuesto por Kavraki et al. Éste se compone de dos fases: la fase de aprendizaje y la fase de consulta. En la fase de aprendizaje se construye un mapa de rutas, muestreando configuraciones del espacio libre. En la fase de consulta, se utiliza el mapa construido para dar solución a problemas de planificación. Debido a que las fases de aprendizaje y de consulta son independientes, es posible reutilizar el mapa de rutas en múltiples consultas [Kavraki et al., 1996].

En la fase de aprendizaje, se crea un grafo no dirigido, cuyos nodos son configuraciones del robot elegidas aleatoriamente en el espacio libre. Por cada nuevo nodo, se eligen k nodos del grafo, ordenados de forma creciente con respecto a la distancia, y se intentan conectar usando un planificador local. En caso de lograrlo se genera una arista entre ambos nodos. Para determinar si una configuración pertenece al espacio libre, ésta se valida con un detector de colisiones. A las aristas en el grafo construido, se les denomina rutas locales y no son almacenadas, bajo la suposición de que pueden ser generadas rápidamente, utilizando un planificador local. Un planificador local puede generar rutas entre dos configuraciones, trazando un segmento de línea y verificando que este no intersecte algún obstáculo. Para hacer esta verificación, se discretiza el segmento en un conjunto de configuraciones que pueden ser probadas por el detector de colisiones.

En la fase de consulta, se utiliza el grafo creado en la fase de aprendizaje, pues este representa la conectividad del espacio libre (ver Figura 3.15). Dada una consulta formada por una configuración inicial y una final del robot, lo único que resta es conectar las configuraciones al grafo para poder aplicar un algoritmo de búsqueda del camino más corto, tal como el algoritmo de Dijkstra [Dijkstra, 1959]. Para conectar las configuraciones inicial y final, estas se insertan como nodos del grafo y a continuación se procede a probar su conectividad con los k nodos más cercanos. Estos nodos vecinos se ordenan de manera creciente, con respecto a una función de distancia. La conectividad es probada utilizando el planificar local. Como se puede ver, no es necesario conectar las configuraciones con todos los nodos vecinos, ya que basta con uno de ellos, pero en caso de no lograrlo la consulta no tendrá solución.

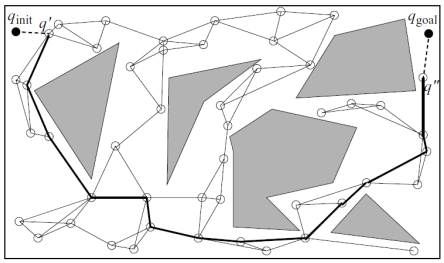


Figura 3.15: Consulta del mapa de rutas construido por el algoritmo PRM [Choset, 2005].

RRTs

Los árboles aleatorios de exploración rápida (RRTs[4], por sus siglas en inglés) propuestos inicialmente por LaValle y generalizados a árboles densos de exploración rápida (RDTs[5], por sus siglas en inglés), son una estructura diseñada para resolver problemas de planificación de trayectorias (ver Figura 3.16). Su funcionamiento se basa en la exploración del *C-Space* (ver Sección 3.2.1), mediante el crecimiento de un árbol a partir de una configuración inicial [LaValle, 1998]. Para ello se vale de configuraciones aleatorias que son conectadas con su vecino más cercano, basado en una métrica de distancia. Este algoritmo es de tipo **consulta única** y, de manera similar al algoritmo PRM, considera el uso de un planificador local para conectar las muestras y un detector de colisiones para validar las configuraciones.

El algoritmo de generación de RRTs utiliza un grafo, cuyo primer nodo es la configuración inicial del robot q_I. A continuación, se procede a generar configuraciones aleatorias, q_r, alrededor de q_I. Con cada nueva q_r, existe la posibilidad de intersección con un obstáculo y es tarea del detector de colisiones determinar esto. En caso de no haber colisión, se procede a intentar conectar la nueva configuración con el nodo más cercano, utilizando un planificador local. Si se logra la conexión, esta se convierte en arista. En el caso en que q_r se intersecte con un objeto, se procede a generar otra configuración aleatoria lo más cerca posible del obstáculo y se repite el proceso de conexión con esta nueva q_r. Este proceso se repite hasta que se alcance un número determinado de configuraciones aleatorias o se llegue a la configuración final.

Para utilizar los RRTs en problemas de planificación de trayectoria, que tienen como objetivo llegar de una configuración inicial a una final, q_F, se debe modificar la forma en cómo se obtienen las configuraciones aleatorias. Esta modificación implica que cada determinado número de iteraciones, e.g., 100, se debe hacer q_r igual a q_F. De esta forma, el planificador local intentará conectar el árbol construido con la configuración final deseada. La realización de dicho intento de conexión en cada iteración no sería viable, puesto que no se exploraría el ambiente y, en caso de haber obstáculos,

[4] *Rapidly Exploring Random Trees*
[5] *Rapidly Exploring Dense Trees*

estos no permitirían conectar las configuraciones inicial y final. Como se puede ver, la exploración parte de q_I, pero es posible utilizar dos árboles, uno partiendo de q_I y otro de q_F, para mejorar los resultados de búsqueda.

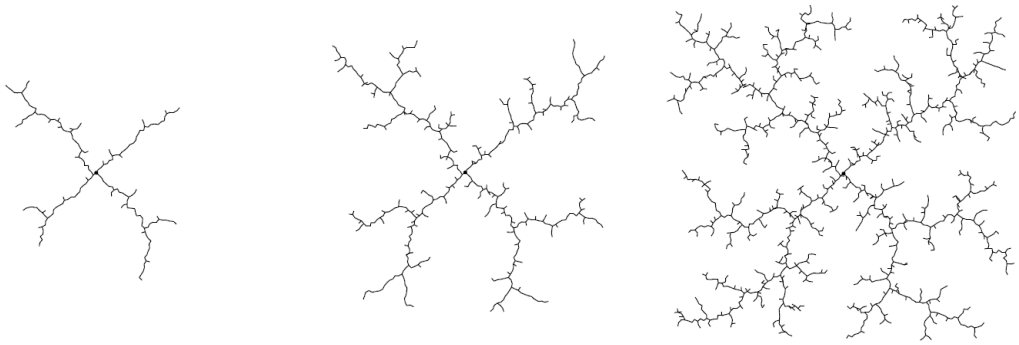


Figura 3.16: Árboles de exploración rápida (RRTs) con distintos niveles de crecimiento [LaValle, 1998].

3.2.5. OMPL

En la sección 3.2 se presentaron algunos de los algoritmos de planificación de trayectoria, tanto combinatorios como basados en muestreo, siendo estos últimos los que gozan de mayor popularidad en entornos industriales y de investigación, gracias a su aplicabilidad en una amplia variedad de problemas del mundo real. Sin embargo, para su implementación en código no basta con la descripción de los algoritmos, ya que es necesario resolver problemas técnicos, lo cual limita el uso de los algoritmos de planificación de trayectoria en la práctica. Este problema no sólo afecta a los usuarios finales, sino también a los investigadores encargados de realizar nuevos planificadores, pues deben crear implementaciones en software de aquellos algoritmos con los cuales desean comparar sus propuestas. Con estos problemas en mente, en el año 2012, se creó la biblioteca abierta de planificación de movimiento (OMPL[6], por sus siglas en inglés) que provee implementaciones de múltiples algoritmos basados en muestreo [Şucan et al., 2012].

OMPL está desarrollada en C++, con el propósito de ser una implementación eficiente, que además provea herramientas para realizar evaluaciones comparativas entre algoritmos. Estas evaluaciones permiten medir el desempeño de nuevos algoritmos, que pueden ser incluidos en la biblioteca a través de la API. El diseño de OMPL hace que sea independiente de algún detector de colisiones o visualizador específico, aunque cuenta con envoltorios para la biblioteca flexible de colisiones (FCL[7], por sus siglas en inglés). De igual manera, la biblioteca no provee una representación de los

[6] *Open Motion Library*
[7] *Flexible Collision Library*

robots, lo que significa que es necesario especificarla, mediante código, o hacer uso de bibliotecas complementarias. Este bajo acoplamiento con otros componentes permite que OMPL se pueda integrar con otras herramientas robóticas, para dotarlas de capacidades de **planificación de movimiento**.

En el presente trabajo de tesis, se utiliza OMPL a través de la interfaz de MoveIt, un software de manipulación robótica. En el Capítulo 6, se presentan más detalles acerca de su forma de uso y configuración, puesto que es necesario indicarle la estructura del robot, en este caso un MAV, a través de un archivo de descripción. En la práctica, esta biblioteca permite configurar el tiempo de búsqueda en el entorno y el algoritmo a utilizar, así como algunos parámetros específicos de cada algoritmo. Es esta posibilidad de emplear complejos planificadores, como si se tratara de una caja negra, la que ha favorecido la difusión de OMPL en ambientes de robótica industriales.

3.3. Exploración robótica

En esta Sección, se explica el término de exploración en el contexto de la robótica y los algoritmos necesarios para realizarla, considerando el uso de un solo robot. En la Sección 3.3.1 se presenta el concepto de exploración. En la Sección 3.3.2 se describe el algoritmo de exploración basado en frontera, el cual es el más representativo de su área. En las secciones 3.3.3 y 3.3.4 se presentan el algoritmo de exploración basado en mapas de cobertura y el basado en características, respectivamente. En la Sección 3.3.5 se describe el algoritmo de relleno por inundación. Este algoritmo es relevante en exploración ya que puede ser aplicado en las celdas de los mapas de ocupación para identificar fronteras inaccesibles [Fraundorfer et al., 2012]. Por último, en la Sección 3.3.6 se describen algunos de los aspectos a tomar en cuenta al momento de decidir entre metas de exploración.

3.3.1. Concepto e importancia

El propósito de la exploración robótica es permitir que un robot adquiera conocimiento de su entorno, mediante su desplazamiento a zonas desconocidas. Este conocimiento se ve reflejado en forma de mapa. Como se vio en la Sección 3.1.4, la construcción de mapas se realiza mediante algoritmos SLAM, pero estos algoritmos no indican al robot los lugares aún no visitados que le sirven para reducir la incertidumbre del entorno. Por otra parte, para realizar exploración es necesario generar rutas que el robot pueda seguir para alcanzar su objetivo, i.e., realizar planificación de movimiento, como se vió en la Sección 3.2. Por lo anterior, el proceso de exploración requiere la aplicación de técnicas de estas dos áreas y además, en el caso de exploración multirobot, se necesita incluir una etapa de coordinación.

La exploración robótica es importante por sus múltiples aplicaciones en lugares riesgosos o inaccesibles para seres humanos. Mediante el uso de robots para exploración, es posible construir mapas de zonas de desastre, pero también brinda a los robots la posibilidad de realizar búsquedas en el entorno. Las búsquedas se logran gracias a un algoritmo de exploración que identifica las zonas del mapa ya visitadas e indica al robot cuáles son los lugares siguientes a investigar.

El proceso de exploración consiste en dos etapas principales, la etapa de definición de regiones a visitar y la etapa de navegación hacia nuevas regiones. En la primera etapa, se proponen y seleccionan locaciones en el mapa a las cuáles el robot puede desplazarse, i.e., configuraciones del espacio libre que no han sido visitadas anteriormente. La segunda etapa consiste en desplazar el robot hacia las locaciones seleccionadas en la etapa previa, evadiendo las posibles colisiones del entorno. Este proceso puede ser realizado mediante algoritmos basados en frontera, basados en mapas de cobertura y en características.

3.3.2. Exploración basada en frontera

El algoritmo de exploración basada en frontera fue propuesto por Yamauchi. Su funcionamiento se basa en la idea de ganar información del ambiente, moviéndose al límite entre el espacio explorado y el inexplorado [Yamauchi, 1997]. La propuesta original fue realizada considerando entornos en 2D, pero trabajos recientes como el presentado por Zhu et al., han extendido la idea a 3D [Zhu et al., 2015]. Este algoritmo es uno de los más populares, ya que fue el primero en introducir el concepto de fronteras para explorar entornos desconocidos.

En este algoritmo se considera al mapa como un *grid* 2D, donde cada celda puede estar "ocupada", "libre" o "desconocida" (ver Figura 3.17). Los valores "ocupado" y "libre" representan al espacio explorado hasta el momento, mientras que las celdas "desconocidas" son aquellas que aún no han sido exploradas. De acuerdo con esta clasificación, una frontera es aquella celda que se encuentra en el límite entre el espacio explorado libre y el espacio desconocido. Cabe destacar que pueden existir múltiples fronteras al mismo tiempo, por lo que es necesario establecer un criterio de selección para determinar a cuál se moverá el robot. En el trabajo de Yamauchi, el criterio es la cercanía al robot.

Una vez que se cuenta con una frontera objetivo, el robot procede a desplazarse a ella y al alcanzarla hace una rotación de 360 grados. Al realizar esta rotación, se crea un mapea del área cercana, cuyo tamaño depende del alcance de los sensores. Una vez finalizado este proceso, se hace la actualización del mapa y se procede a obtener la siguiente frontera, como se describió en el párrafo anterior. En caso de que al navegar hacia una frontera se determine que esta es inaccesible, se procederá a agregarla a una lista de fronteras inaccesibles y se repetirá el proceso de barrido con los sensores.

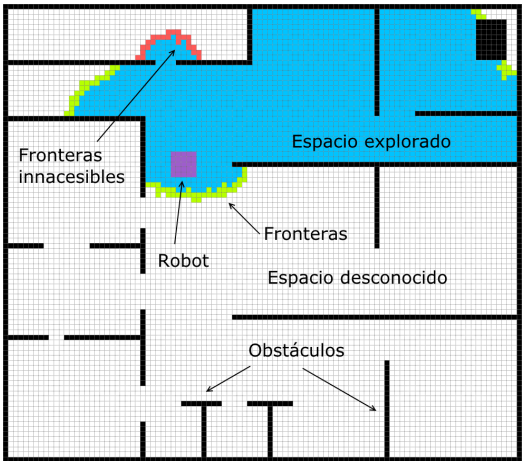


Figura 3.17: Interpretación de la exploración basada en frontera propuesta por Yamauchi [Yamauchi, 1997].

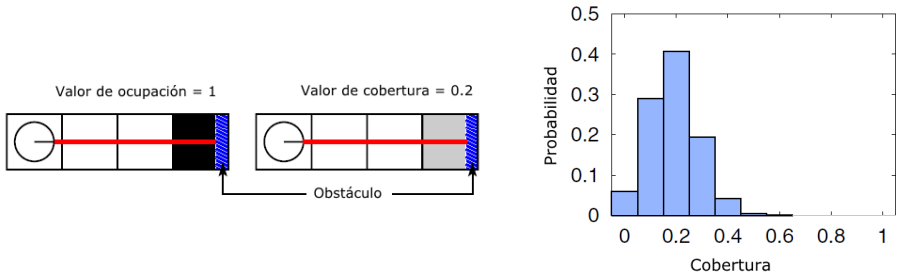
Las fronteras inaccesibles son almacenadas con el propósito de no volver a intentar explorarlas.

3.3.3. Exploración basada en mapas de cobertura

Los mapas de cobertura [Stachniss and Burgard, 2003] son una representación del espacio, similar a los mapas de ocupación, que difieren de estos últimos en los valores almacenados por cada celda. En un mapa de ocupación tradicional, cada celda del *grid* puede tener dos valores, libre u ocupada, pero estos valores no reflejan el estado de aquellas que están parcialmente ocupadas. Con este inconveniente en mente, se crearon los mapas de cobertura en los cuales cada celda almacena una probabilidad *a posteriori*. Esta probabilidad es una creencia del valor de cobertura que puede tener la celda; dicho valor varía en el rango de 0 a 1, donde 0 es libre y 1 totalmente ocupada. La probabilidad de que la celda tenga cierta cobertura es almacenada en forma de histograma (ver Figura 3.18).

Una etapa fundamental en el proceso de exploración es la de identificar las zonas del entorno que requieren ser exploradas. En los mapas de cobertura, esto se logra eligiendo aquellas con mayor incertidumbre. Para ello, primero es necesario tener una medida de la incertidumbre del entorno, a la que se le denomina entropía. Esta medida se calcula utilizando el histograma de cada celda y es cero cuando hay completa seguridad del valor de cobertura de la celda. Finalmente, una celda se considera explorada cuando mediciones consecutivas tienen un cambio de entropía menor a cierto umbral.

La entropía permite identificar aquellas celdas que requieren ser exploradas, pero aún queda pendiente la forma de elegir las celdas a las cuáles el robot debe moverse. Entre las posibles opciones se encuentran: elegir la celda más cercana, seleccionar



(a) (b)

Figura 3.18: Mapas de cobertura: (a) valores almacenados en una celda parcialmente ocupada para mapas de ocupación y mapas de cobertura, y (b) probabilidad *a posteriori* en forma de histograma para la celda de la figura (a) en mapas de cobertura.

aquella con mayor incertidumbre o definir una ventana local, la cual es una zona cercana al robot que debe ser completamente explorada antes de moverse a otra zona. En la Sección 3.3.6 se trata con mayor detalle este aspecto de selección, que puede ser aplicado en todo tipo de exploración. En el caso de los mapas de cobertura, los mejores resultados se obtienen visitando aquellas celdas que minimizan la incertidumbre, i.e., la entropía, y que están más cercanas al robot. Por último, cabe mencionar que los mapas de ocupación pueden considerarse mapas de cobertura, con dos elementos en el histograma. Por lo tanto, también es posible hacer el cálculo de la entropía para cada celda de un mapa de ocupación.

Otro algoritmo que también hace uso de las estimaciones *a posteriori*, pero en una forma diferente es el de exploración basada en filtro de partículas *Rao-Blackwellized*, el cual se aplica en combinación con un algoritmo SLAM basado en el mismo filtro [Stachniss et al., 2005]. En el algoritmo SLAM *Rao-Blackwellized*, cada partícula representa trayectorias potenciales del robot y a su vez cada partícula construye su propio mapa. Dado lo anterior, es posible definir una función de utilidad que considere la ganancia de información, con respecto al costo de realizar dicha trayectoria. En este caso, el significado de las partículas está relacionado con la probabilidad de realizar una trayectoria, dadas ciertas observaciones. Sin embargo, en la Sección 4.2 se presenta otro enfoque, en el cual las partículas son elementos físicos que se dispersan a través del entorno y colisionan con los obstáculos presentes en él.

3.3.4. Exploración basada en características

La exploración basada en características utiliza elementos del entorno para identificar lugares explorables [Newman et al., 2003]. Las características geométricas, como se mencionó en la Sección 3.1.2, pueden ser líneas de paredes, puertas, esquinas, entre otras. Este algoritmo, a diferencia del basado en filtro de partículas *Rao-Blackwellized*, toma como base un algoritmo SLAM cualquiera, que provea un flujo de características

geométricas, datos de posición, orientación e incertidumbre. A partir de las características obtenidas, el algoritmo genera metas de exploración potenciales alrededor de éstas. Los criterios utilizados para definir estas metas son: visitar áreas abiertas, explorar primero localmente antes de considerar zonas lejanas y explorar con distintos niveles de detalle, e.g., adquirir primero información general de los objetos del entorno.

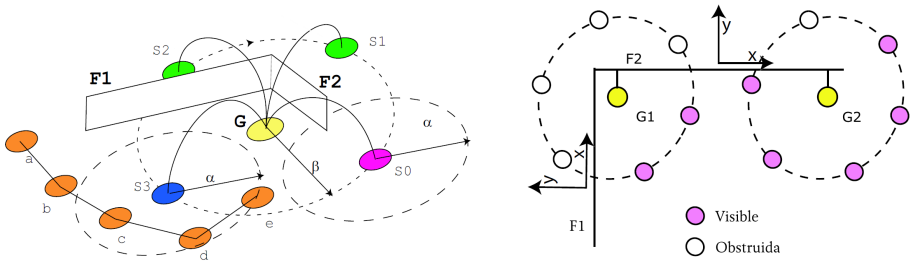
Una vez que se cuenta con las metas, se decide cuál es la que provee la mayor ganancia de información utilizando una función de evaluación que genera muestras alrededor de cada meta. Dichas muestras son puntos distribuidos a un determinado radio de las metas (ver Figura 3.19). La puntuación de cada meta varía en el rango de cero a uno y se calcula contando las muestras que cumplen con dos propiedades. La primera de ellas es que las muestras tengan línea de visión a la meta. La segunda propiedad es que las muestras no se encuentren a una distancia menor a α de los marcadores de trayectoria, donde α es un valor predefinido. Todas aquellas muestras que no cumplan con estas dos propiedades se descartan, i.e., aquellas con las que no se tiene línea de visión o que están a una distancia menor a α de un marcador de trayectoria. La intuición detrás de esta puntuación es que, en regiones altamente exploradas, existirán más características geométricas que obstruyan la línea de visión entre la meta y las muestras, dando como resultado una menor ganancia de información.

Como se indica en el párrafo anterior, este algoritmo considera la línea de visión entre las metas potenciales y los marcadores de trayectoria. La existencia de estos marcadores se debe a que este algoritmo de exploración también sirve para construir un grafo que representa un mapa de rutas. En este mapa, cada marcador es un nodo del grafo que denota una posición del robot. El mapa de rutas contiene los destinos alcanzables por el robot en el espacio libre conocido. La forma de conectar estos destinos es mediante línea de visión. Esta forma es similar a la utilizada con las muestras en el algoritmo PRM de la Sección 3.2.4. Los nuevos marcadores se crean, respetando una distancia mínima con respecto al marcador más cercano en línea de visión. Por último, cuando se genera una meta de exploración, que no está en la línea de visión del robot, se procede a utilizar el grafo para buscar un camino libre.

3.3.5. Algoritmo de relleno por inundación

Este algoritmo es utilizado para rellenar áreas confinadas. Su funcionamiento es similar a la herramienta "cubo de pintura", presente en programas de dibujo por computadora. La representación del espacio de trabajo está dada en píxeles que, para fines prácticos, se considera como un *grid* 2D, donde cada celda tiene un color asignado. Para comenzar, se requiere un punto inicial dentro del área deseada, llamado semilla, un color a reemplazar y otro color para rellenar. Este algoritmo se puede implementar mediante recursividad o una estructura de datos pila [Cohen-Or, 2016].

El algoritmo de relleno por inundación parte de la semilla y rellena las celdas



(a) (b)

Figura 3.19: Exploración basada en características: (a) proceso de creación de metas de exploración, donde s_i son las muestras en un radio β alrededor de la meta G creada por las características F_i, los puntos a, b, c, d, e son marcadores de trayectoria y (b) puntuación de una meta G_i [Newman et al., 2003].

vecinas de forma recursiva (ver Figura 3.20). Existen dos formas de seleccionar estas celdas. La primera es utilizando cuatro de ellas en forma de cruz (arriba, abajo, izquierda, derecha). La segunda consiste en emplear ocho celdas, i.e., todas las que rodean a la semilla. Esta segunda forma tiene el inconveniente de que también rellena las esquinas, lo cual no es deseable en situaciones en que el contenedor tiene forma escalonada. En términos de implementación, una versión recursiva hace un gran número de llamadas a procedimientos, por ello una mejor implementación es la que utiliza pilas (ver Algoritmo 3.1).

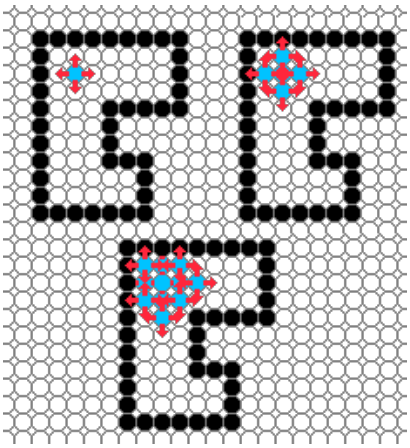


Figura 3.20: Representación gráfica del algoritmo de relleno por inundación.

Una variante eficiente del Algoritmo 3.1 consiste en rellenar por tramos. Un tramo está formado por todas las celdas, del color a reemplazar, que son contiguas en un renglón. El algoritmo consiste en rellenar primero el tramo de la semilla. A continuación, se identifican los inicios de los tramos existentes en los renglones superior e inferior. Por último, se colocan los inicios en la pila y se repite el proceso (ver Algoritmo 3.2) [Inkpen, 2016].

Algoritmo 3.1 Relleno por inundación utilizando una pila

Requiere: Semilla s , color inicial ci, color final cf

```
1: procedure RellenoInundacionPila(s, ci, cf )
2:     P ← ∅                                              ▷ Pila vacía
3:     P.push(s)
4:     while P ≠ ∅ do
5:         cell ← P.pop()
6:         for each v ∈ vecinos de cell do      ▷ cuatro u ocho vecinos en el mapa
7:             if v.color = ci then
8:                 v.color ← cf
9:                 P.push(v)
```

Algoritmo 3.2 Relleno por inundación basado en tramos

Requiere: Semilla s , color inicial ci, color final cf

```
1: procedure RellenoInundacionTramos(s, ci, cf )
2:     P ← ∅                                              ▷ Pila vacía
3:     Inicializar lista IT vacía                         ▷ Inicios de tramos
4:     P.push(s)
5:     while P ≠ ∅ do
6:         cell ← P.pop()
7:         Rellenar las celdas del tramo al que pertenece cell
8:         IT ← Inicios de tramos en los renglones superior e inferior de cell
9:         P.pushAll(IT)                  ▷ Añadir todos los elemento de IT a P
```

3.3.6. Factores en la selección de metas de exploración

En los algoritmos de exploración, varía la forma en cómo se identifican los lugares del entorno que requieren ser explorados, también llamados metas de exploración. Sin embargo, cuando se obtiene más de una meta, todos los algoritmos necesitan un criterio que les indique cuál elegir. Para seleccionar el criterio, es necesario tener en cuenta aspectos del robot tales como sus dinámicas y sensores. Como se mencionó en la Sección 3.3.3, algunos de los criterios son: elegir el destino más cercano al robot, seleccionar el destino con mayor ganancia de información o usar ventanas locales [Stachniss, 2009]. A lo anterior se puede añadir las funciones de recompensa, que premian o penalizan acciones realizadas por el robot, con el objetivo de minimizar parámetros tales como el tiempo de exploración y el número de observaciones [Shade, 2011].

La selección de los puntos más cercanos al robot, produce como resultado, las rutas más cortas posibles, pero no necesariamente el menor tiempo de exploración. Ello se debe a que, en un ambiente real, entran en juego otros factores como la aceleración y rotación del vehículo, los cuáles afectan el tiempo de recorrido, incluso para aquellos puntos ubicados a la misma distancia del robot. La minimización del tiempo de exploración también requiere considerar la forma en cómo se hacen observaciones del entorno, i.e., las mediciones usando los sensores del robot. Por ejemplo, un escáner láser podría requerir que el robot permanezca en su posición durante un par de minutos [Shade, 2011]. En tal caso, resulta preferible elegir rutas de mayor longitud, cuyo punto destino requiera pocas mediciones.

Con respecto a la ganancia de información, lo que se busca es adquirir el mayor conocimiento posible del entorno. Para los mapas de cobertura y, por lo tanto, para los mapas de ocupación, la mayor ganancia de información se logra a través de la selección de las celdas con mayor entropía. Por otra parte, en la exploración basada en frontera, también es posible definir una medida de la ganancia de información, con base en el número de celdas inexploradas que cada frontera tiene como vecinas.

Capítulo 4

Trabajos relacionados

En este capítulo se presentan algunos de los trabajos existentes de exploración multi-robot. Aunque el presente trabajo se centra en MAVs, también es importante conocer las estrategias utilizadas en robots terrestres, ya que algunos de sus conceptos también son aplicables a MAVs. Es por esta razón que en la Sección 4.1 se presentan trabajos de exploración multi-robot terrestre y multi-MAV, en sus variantes de coordinación centralizada y coordinación distribuida. Por otra parte, en la Sección 4.2, el enfoque se centra en la descripción de las propuestas de exploración mediante MAVs. Finalmente, en la Sección 4.3, se realiza un análisis comparativo de las propuestas presentadas en las Secciones 4.1 y 4.2. Primeramente se comparan las propuestas de exploración robótica de ambas secciones, de acuerdo a sus características en común. A continuación se desglosan las características de los trabajos de exploración multi-robot, lo cual incluye a las propuestas multi-MAV. Por último, se analizan las características específicas de los trabajos de exploración mediante MAVs.

4.1. Exploración multi-robot

Para realizar exploración multi-robot, es necesario tomar en cuenta aspectos de coordinación que permitan a los robots interactuar en un mismo ambiente. Entre los factores que afectan la coordinación se encuentran las comunicaciones entre robots, la parte del mapa que cada robot conoce y la forma en que es compartido entre ellos. Adicionalmente, se debe realizar una asignación de tareas que maximice la eficiencia del equipo, ya sea en las rutas recorridas o en la cobertura del entorno. Cuando la exploración con un robot se extiende al uso de múltiples robots, se mantiene el objetivo de obtener la mayor ganancia de información posible. Dicho objetivo consiste en priorizar la exploración de las zonas con una mayor cantidad de elementos desconocidos, e.g., celdas. Dependiendo del lugar donde se realice la asignación de tareas, la coordinación puede ser centralizada o distribuida. En la sección 4.1.1, se presentan algoritmos que hacen uso de un coordinador central, e.g., una estación base. En la Sección 4.1.2 se describen algoritmos en los que robots están encargados de negociar entre sí los lugares del entorno a visitar.

4.1.1. Coordinación centralizada

Simmons et al. presentan un algoritmo de exploración con coordinación centralizada, enfocado en reducir los traslapes entre zonas de ganancia de información. En este algoritmo, cada robot es responsable de generar su propio mapa y de enviarlo al coordinador, en forma de mediciones, para que éste produzca un mapa global. Para coordinar la exploración, cada robot determina los puntos a su alcance con mayor ganancia de información, llamados ofertas, aplicando una función de utilidad. Estos puntos son celdas frontera que cumplen el requisito de tener una separación entre sí, de aproximadamente el tamaño del robot.

Posteriormente las ofertas son enviadas al coordinador central, quien primero asigna la oferta con mayor utilidad al robot que la generó. A continuación, se procede a descartar las ofertas cuya zona de ganancia de información tenga cierto traslape con las ofertas ya asignadas. El traslape se calcula con aproximaciones rectangulares de dichas zonas (ver Figura 4.1). Una vez descartadas las ofertas, se procede a repetir iterativamente el proceso de asignación-descarte hasta que no quedan más robots u ofertas disponibles [Simmons et al., 2000].

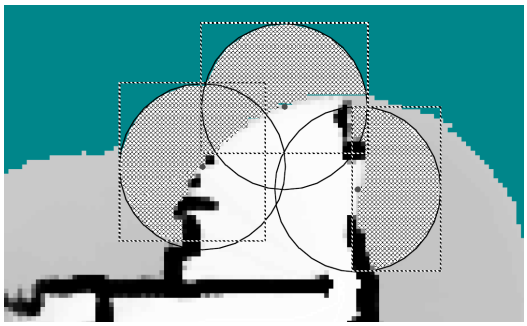


Figura 4.1: Traslape de zonas de ganancia de información [Simmons et al., 2000].

Solanas y García proponen una partición del espacio para realizar exploración multi-robot. Dicha partición consiste en agrupar las celdas desconocidas de un mapa de ocupación en regiones disjuntas mediante el algoritmo *K-means* [Bock, 2007], dónde k denota el número de robots (ver Figura 4.2). Dado que el particionamiento no toma como base a los robots, la forma de asignarlos a las regiones consiste en calcular su distancia a los centroides de éstas y quedarse con la más cercana. A continuación, los robots seleccionan fronteras con base en una función de costos, la cual se encarga de mantenerlos en sus regiones asignadas. Para lograr este resultado se elige la frontera de menor costo. La función asigna costos de dos formas: la primera es para fronteras en la región asignada al robot y la segunda, para fronteras en regiones diferentes a la del robot que las generó. En el primer caso, el costo es la longitud del camino mínimo entre la frontera y el robot. En el segundo caso, el costo es la distancia euclidiana de la frontera al centroide de la región asignada al robot, sumada

a un valor constante de penalización.

Una vez que se asigna una frontera a un robot, el resto de ellas que se encuentren en el rango de sensado del robot también reciben una penalización, con el propósito de evitar traslapes. La exploración de la región continúa hasta que se alcanza un determinado número de celdas; es entonces cuando se aplica nuevamente el algoritmo de agrupamiento y se repite el proceso de asignación. La exploración finaliza cuando no quedan más celdas desconocidas [Solanas and Garcia, 2004]. Posteriormente, Wu et al. proponen una extensión, utilizando una representación poligonal del mapa y el diagrama de Voronoi para generar particiones del espacio desconocido, sin embargo no queda clara la forma en cómo se realiza la exploración [Wu et al., 2007].

Figura 4.2: Exploración de regiones creadas por el algoritmo *K-means*. Las zonas en color negro son las partes exploradas del entorno [Solanas and Garcia, 2004].

Wurm et al. presentan una estrategia de exploración multi-robot que utiliza una segmentación del espacio basada en el diagrama de Voronoi, con el propósito de reducir la interacción entre los robots. El diagrama de Voronoi es calculado usando esqueletización[1] en las celdas del mapa de ocupación, donde cada una de ellas almacena la distancia al obstáculo más cercano. El resultado es un conjunto de aristas rectas, cuyos puntos son equidistantes a dos obstáculos. Por otra parte, los vértices del diagrama son las uniones entre aristas. Una vez construida esta representación, se realiza un partición del diagrama en **puntos críticos** (segmentos), los cuales son regiones del diagrama que representan lugares de acceso, como puertas y pasillos. Estos puntos son nodos de grado dos, cuya distancia a los obstáculos es un mínimo local y, además, tienen como vecino a un nodo de grado tres (ver Figura 4.3).

A partir de los segmentos se obtienen fronteras y, para cada robot, se calcula el costo de desplazarse a un segmento. Para decidir qué robot debe visitar un segmento, se utiliza el método húngaro, un método de asignación óptima de tareas que utiliza una matriz de costos [Wurm et al., 2008]. Esta estrategia privilegia el hecho de que cada robot explore su segmento de manera independiente. En caso de que más de un

[1]Una técnica que produce una estructura delgada en el interior de otra.

robot sea asignado a un segmento, se aplica exploración basada en frontera con una función de costo.

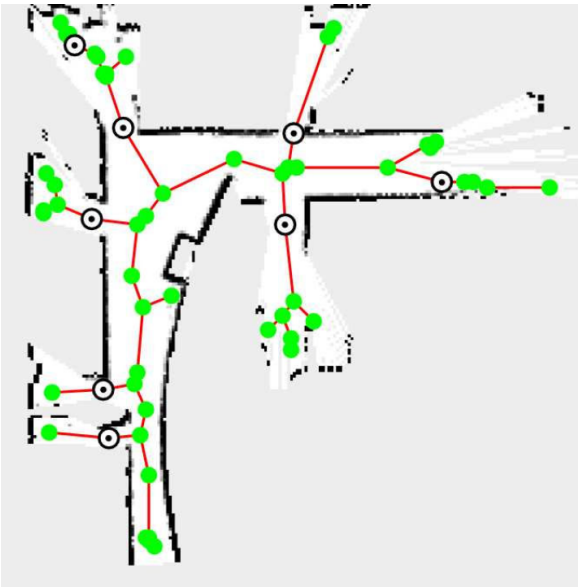


Figura 4.3: Exploración con segmentación del espacio, donde los círculos con un punto en su interior son **puntos críticos** [Wurm et al., 2008].

4.1.2. Coordinación distribuida

Rooker y Birk tratan el problema de exploración autónoma multi-robot bajo las restricciones de alcance de las redes inalámbricas. El objetivo consiste en explorar un ambiente desconocido y conservar la comunicación entre los robots. Para ello se toman como base dos casos. El primero utiliza una estación de control que se encarga de las comunicaciones, por lo tanto la exploración queda limitada al alcance de esta. En el segundo caso, se estudia el uso de paquetes de robots, i.e., que aunque son capaces de moverse más lejos, ahora tienen una distancia máxima de separación entre ellos.

En su propuesta, se utiliza exploración basada en frontera. El movimiento a través de las celdas del *grid* es guiado por una función de utilidad, la cual se encarga de mantener unidos a los robots, puesto que penaliza los movimientos que producen pérdida de comunicación [Rooker and Birk, 2007]. Como resultado, Rooker y Birk obtuvieron que, cuando se trabaja con paquetes de robots, es posible caer en candados mortales (*deadlocks*). Esta situación se presenta cuando los robots se deben alejar más allá del rango de comunicación para continuar la exploración (ver Figura 4.4). La estrategia utilizada para manejar los candados mortales consiste en convertir a un robot en punto de reunión y reagrupar al equipo a su alrededor para después continuar la exploración.

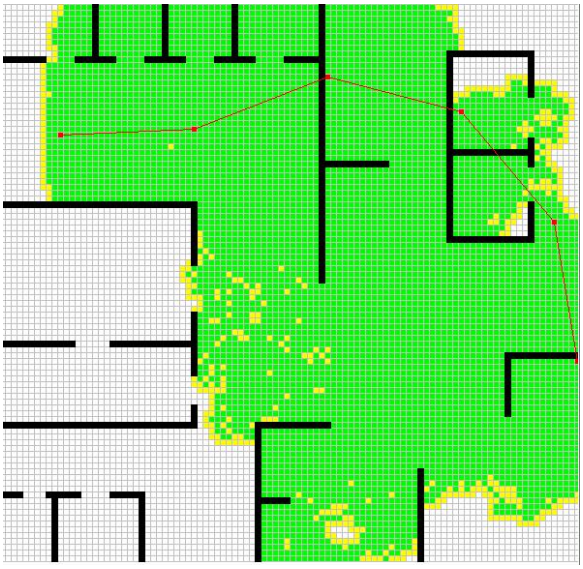


Figura 4.4: Candado mortal en exploración multi-robot. Las líneas rojas representan enlaces de comunicaciones entre robots. [Rooker and Birk, 2007].

Sheng et al. presentan una estrategia de coordinación distribuida para exploración multi-robot con rango limitado de comunicaciones. Por su cercanía, los robots forman subredes mediante las cuales comparten actualizaciones de sus mapas, conforme adquieren nueva información del entorno. El algoritmo utilizado para explorar está basado en frontera y evalúa la ganancia de información de una celda contra el costo de desplazarse a ella. Para determinar a qué celda se mueve cada robot, se utiliza un algoritmo de coordinación basado en ofertas. Una oferta es la mejor de las fronteras de un robot, en términos de ganancia de información y cercanía al resto del equipo.

En el proceso, un robot calcula su oferta y se la propone al resto de sus vecinos. Si ninguna oferta mejor llega en una ventana de tiempo predefinida, el robot se considera ganador. En caso de que llegue una mejor propuesta, el robot espera a que se declare un ganador para recalcular su oferta y volver a proponerla. Finalmente, el ganador se desplaza a la frontera seleccionada, adquiere información del entorno y repite el proceso. Los robots exploran el entorno de manera asíncrona, i.e., en cualquier momento pueden seleccionar una frontera y proponerla a sus vecinos. El proceso de exploración termina cuando no existen nuevas fronteras [Sheng et al., 2006].

Matignon et al. proponen una estrategia de exploración basada en el proceso de decisión de Markov en modo descentralizado (Dec-MDP[2], por sus siglas en inglés). El proceso de decisión de Markov considera la probabilidad de llegar a un estado siguiente, dado que se realiza una acción con incertidumbre en el estado actual. La idea es que cada robot construya su estrategia de exploración basada en las posibles

[2] *Decentralized Markov Decision Process*

interacciones con el resto del equipo. Esto se logra mediante una función de valor distribuida (DVF[3], por sus siglas en inglés) que toma en cuenta los estados de todos los robots en cada paso de exploración.

La función evalúa la ganancia de información que puede obtener un robot al moverse a otro estado y la probabilidad de alcanzarlo, en comparación con el resto del equipo. Este criterio da como resultado que se elija al robot con mejores posibilidades y mejor ganancia de información para ese estado [Matignon et al., 2012]. En la propuesta, los estados son celdas de un *grid* hexagonal, creadas a partir de las celdas de un mapa de ocupación (ver Figura 4.5). Las acciones permitidas a un robot son: movimientos adelante, atrás, izquierda, derecha y seguir la arista del diagrama de Voronoi a la siguiente celda. El diagrama de Voronoi es utilizado en situaciones en las cuales un robot no puede desplazarse a la siguiente celda con los primeros cuatro movimientos, debido a lo estrecho del lugar, e.g., corredores angostos.

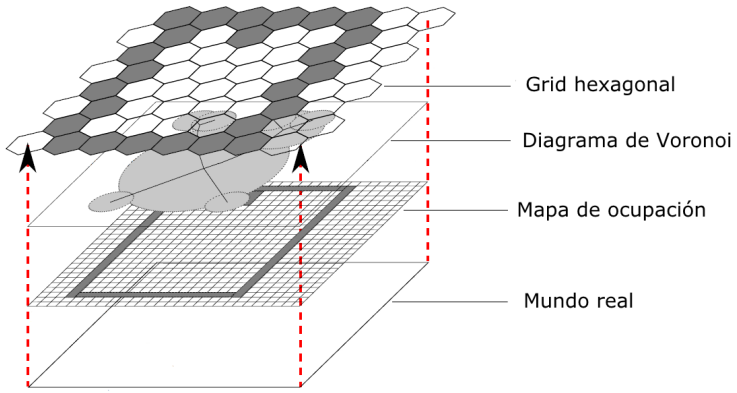


Figura 4.5: Arquitectura en capas para exploración distribuida [Matignon et al., 2012].

Franchi et al. proponen una estrategia descentralizada de exploración que construye un mapa de rutas en forma de SRG (*Sensor-based Random Graph*) con el propósito de maximizar la cobertura. Este tipo de grafo representa el espacio explorado, donde sus nodos son posiciones en el espacio libre y las aristas son rutas libres entre nodos. Estos nodos definen una región local segura (LSR[4], por sus siglas en inglés), cuya área es el rango del sensor. En lugar de existir un SRG global, cada robot almacena la parte que ha generado y la comparte con el resto del equipo cuando se encuentra en rango de comunicación.

En esta estrategia, los bordes de una LSR son clasificados en arcos ocupados, libres y frontera. Un arco frontera es aquel ubicado en una zona libre de la LSR que no tiene traslape con otra LSR (ver Figura 4.6). Para llevar a cabo la exploración, cada

[3] *Distributed Value Function*
[4] *Local Safe Region*

robot selecciona un punto en sus arcos frontera con los que tenga línea de visión. Si los robots del grupo tienen trayectorias que entren en conflicto, se determina un coordinador que las resuelva. La exploración termina cuando no hay más arcos frontera alcanzables en el SRG de los robots [Franchi et al., 2009].

Cesare et al. proponen un algoritmo de exploración distribuido multi-MAV para maximizar el área explorada, considerando restricciones de comunicación y batería. En este trabajo, cada MAV tiene un identificador numérico que indica su prioridad. El algoritmo utiliza cuatro estados: exploración, reunión (*meet*), sacrificio (*sacrifice*) y repetición (*relay*). En el estado de exploración, se utiliza un algoritmo basado en frontera. Cada MAV construye su mapa de ocupación y extrae fronteras, de las cuales selecciona la más cercana y procede a moverse a ella. Cuando dos MAVs seleccionan fronteras muy cercanas entre sí, el MAV de mayor prioridad conserva su meta y el otro genera una nueva lo más cerca posible. En caso de riesgo de colisión, el MAV de menor prioridad desciende para dar paso a su compañero.

Después de un tiempo determinado explorando, los MAVs pasan al estado de reunión. Es posible que, durante la exploración, los MAVs hayan perdido comunicación entre sí. Por esta razón, en el estado de reunión, los MAVs buscan a otro miembro del equipo para intercambiar información del mapa. Cuando encuentran a un miembro, los MAVs establecen los roles de sacrificio y repetición. El MAV con rol sacrificio continua la exploración hasta que su batería desciende a un grado tal, que sólo es suficiente para regresar al punto donde vio al MAV con rol repetición. Este último MAV continua su exploración con la restricción de mantener siempre batería suficiente para regresar a la estación base. Cuando la batería del MAV con rol repetición llega a este punto, desciende y espera hasta que el MAV sacrificio regrese para intercambiar información con él, antes de volver a la estación base [Cesare et al., 2015].

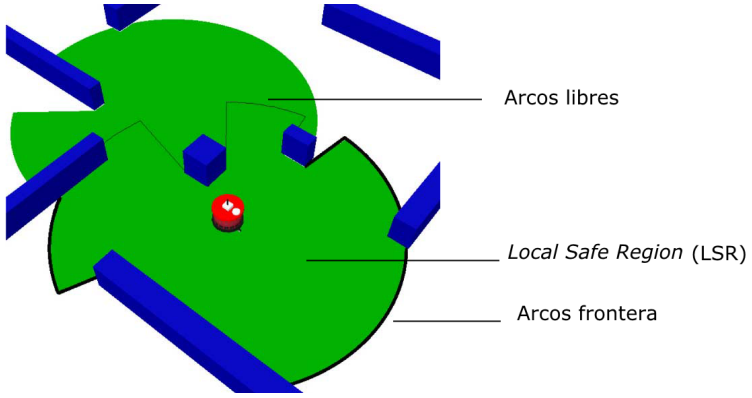


Figura 4.6: Nodos de un SRG. Cada LSR puede contener arcos ocupados, libres y frontera. Los arcos ocupados son aquellos obstruidos por obstáculos [Franchi et al., 2009].

4.2. Exploración mediante MAVs

Las restricciones de procesamiento de los MAVs influyen en los algoritmos utilizados para realizar exploración. Lo anterior se ve reflejado en un menor número de trabajos, en comparación con los de exploración multi-robot. En esta Sección, se presentan propuestas de exploración mediante este tipo de UAVs, de acuerdo a la definición utilizada en el presente trabajo de tesis (ver Sección 2.6). Por la forma del entorno en el que se desplazan los MAVs, este puede ser 2.5D[5] o 3D. Esta consideración se puede realizar manteniendo una altura de vuelo fija.

El algoritmo de exploración multi-MAV propuesto por Cesare et al. (ver Sección 4.1.2) posee características propias de los MAVs que pueden ser analizadas independientemente de la estrategia de coordinación. Estas características son autonomía y tipo de entorno. Debido a que, en su propuesta, los MAVs pueden continuar explorando sin la necesidad de realizar tareas de procesamiento en la estación base, su autonomía es total. Por otra parte, el entorno considerado en la propuesta de Cesare et al. es 2.5D. Este tipo de entorno significa que "el área a mapear es 2D, pero los robots pueden volar a una cierta distancia sobre el suelo" [Cesare et al., 2015]. El algoritmo de exploración basado en frontera, utilizado por Cesare et al, es similar al SRG de Franchi et al. (ver Sección 4.1.2) sin embargo no se proporcionan más detalles al respecto.

Shen et al. proponen un algoritmo de exploración basado en ecuaciones diferenciales estocásticas (SDE[6], por sus siglas en inglés) aplicado a un solo MAV. El algoritmo utiliza partículas, cuyas dinámicas están definidas por estas ecuaciones y tienen el comportamiento de moléculas de un gas ideal. Las partículas representan el espacio libre y son emitidas a medida que el MAV se desplaza en el entorno. El proceso de generación de metas de exploración inicia con la realización de una simulación, en la que se asignan velocidades iniciales aleatorias a las partículas. Dada la forma en cómo se modelan las partículas, las zonas del mapa con menor densidad de estas son aquellas que requieren ser exploradas.

A continuación, se agrupa a las partículas en conjuntos que representan metas de exploración (ver Figura 4.7). Estas metas poseen línea de visión en el rango del sensor, con respecto al espacio explorado e inexplorado. Por último, el MAV visita las metas, emitiendo nuevas partículas y se repite el proceso [Shen et al., 2012]. Un aspecto importante de las partículas es que necesitan ser recicladas cada cierto tiempo, con el fin de mantener un número constante de ellas, lo cual se logra eliminando las más viejas del conjunto.

[5] En un entorno 2.5D, "se consideran paredes verticales y planos horizontales constantes" [Shen et al., 2011].

[6] *Stochastic Differential Equations*

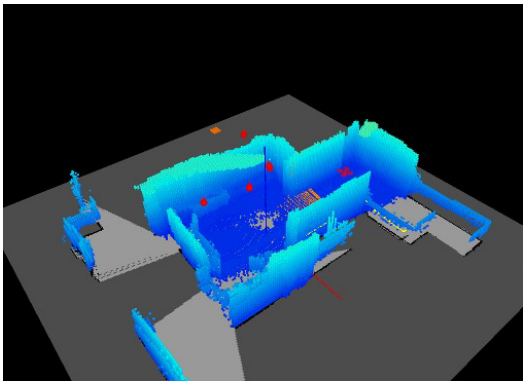


Figura 4.7: Exploración basada en partículas mediante un MAV [Shen et al., 2012].

Bachrach et al. presentan un algoritmo de exploración basado en frontera para un solo MAV en un entorno 2.5D, que conserva la capacidad de localización del MAV. Este algoritmo evalúa un conjunto de muestras, donde cada muestra es una posición posible del MAV en las fronteras (ver Figura 4.8). Para determinar la mejor muestra, que se convertirá en meta de exploración, se utiliza una función de evaluación que toma en cuenta dos aspectos: ganancia de información y localización. En primer lugar, la ganancia de información se estima contando el número de celdas desconocidas que son visibles desde una muestra, de acuerdo a las capacidades de sensado del MAV. A continuación, dicho número se divide entre el número de celdas que normalmente cubre el sensor y el resultado se normaliza en el rango de 0 a 1.

En segundo lugar, se determina la capacidad de localización del MAV en la muestra. Dicho valor se calcula utilizando una métrica de incertidumbre en las mediciones del sensor y también se normaliza en el rango de 0 a 1 [Bachrach et al., 2009]. El cálculo de este valor es importante puesto que, en caso de que las paredes y demás objetos del lugar salgan del rango de sensado, el MAV no podrá determinar su posición en el mapa. La pérdida de localización se debe a que, como se vio en la Sección 3.1.4, un robot requiere comparar observaciones del entorno para determinar su ubicación. Una vez calculada la mejor muestra, el MAV se desplaza a ella, adquiere nueva información y repite el proceso de selección.

Zhu et. al. presentan 3D-FBET, una herramienta de exploración 3D basada en frontera (ver Figura 4.9), la cual procesa únicamente las partes del mapa que cambian. Este algoritmo se basa en un proceso de detección de cambios en el mapa que ocurren con cada nueva observación. En la parte actualizada del mapa, se extraen celdas frontera, las cuales tienen como característica ser vecinas de, al menos, una celda desconocida. Para reducir el número de celdas frontera, éstas se agrupan en fronteras candidatas[7].

[7]Una frontera candidata es el centro geométrico de un conjunto de celdas frontera contiguas

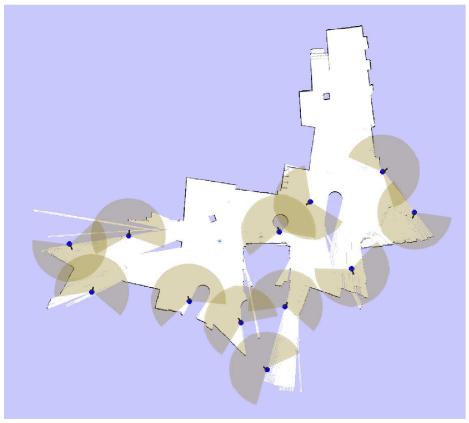


Figura 4.8: Fronteras en exploración mediante un MAV considerando las posiciones del MAV [Bachrach et al., 2009].

Una vez realizada la agrupación en fronteras candidatas, es necesario decidir a cuál de ellas se moverá el MAV. Para realizar esta tarea, se utiliza una función de evaluación que considera la ganancia de información contra el costo de la distancia recorrida. Dicha función se aplica en el momento en que se deja de detectar nuevas fronteras. La ganancia de información de una frontera candidata se calcula contando el número de celdas desconocidas en la esfera de la cual es centro. El radio de dicha esfera es la distancia máxima de sensado del MAV. Una vez que se ha seleccionado una frontera, el MAV procede a desplazarse a ella y continuar la exploración [Zhu et al., 2015].

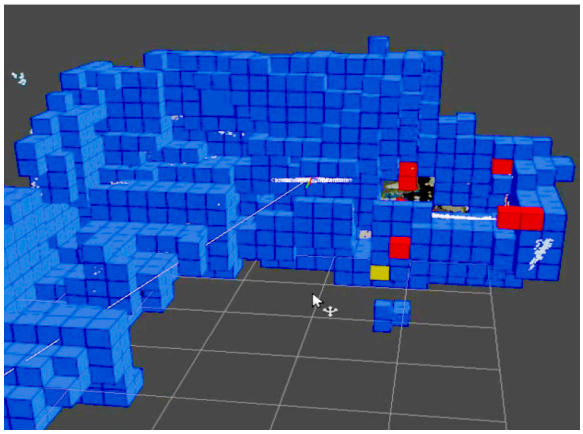


Figura 4.9: Herramienta 3D-FBET de exploración para MAVs. En rojo se muestran las fronteras candidatas y en amarillo se representa la frontera seleccionada [Zhu et al., 2015].

Fraundorfer et al. utilizan un algoritmo de exploración basado en frontera y un algoritmo de seguimiento de paredes, ambos en un entorno 2.5D, para realizar exploración con un MAV. El primer algoritmo agrupa las celdas frontera, etiquetando componentes conexas para producir fronteras. De estas fronteras, se descartan aquellas inaccesibles y cuyo número de celdas sea inferior a un límite preestablecido. Para

determinar la accesibilidad de las fronteras, se aplica el algoritmo de relleno por inundación.

Debido a que cada frontera está compuesta por un conjunto de celdas, es necesario seleccionar un punto a visitar en su interior (i.e., el centroide). La exploración del entorno continúa hasta que no quedan más fronteras por visitar, excepto una. La frontera restante, llamada frontera hogar, es insertada intencionalmente al inicio de la exploración, detrás de la posición del MAV. El algoritmo de seguimiento de paredes utiliza un plano del mapa de ocupación que es paralelo al vector de gravedad. El seguimiento de paredes se produce mediante la navegación hacía tres puntos seleccionados de forma iterativa (ver Figura 4.10). El primer punto se calcula cercano a la pared. El segundo, se calcula a una distancia fija del primero y el tercero se obtiene de forma perpendicular a la pared [Fraundorfer et al., 2012].

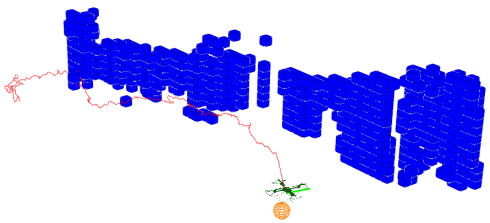


Figura 4.10: Exploración siguiendo una pared [Fraundorfer et al., 2012].

En los trabajos de exploración multi-robot se hace la suposición de que todos comparten el mismo mapa. En la práctica, esta suposición requiere de una técnica para combinar los mapas individuales en uno global. Lawson presenta un algoritmo centralizado de generación cooperativa de mapas en entornos 3D multi-MAV. El algoritmo utiliza, como entrada, las observaciones adquiridas por los MAVs, en forma de nube de puntos. En lugar de enviar todas las observaciones adquiridas a lo largo del tiempo, cada MAV sólo envía la más reciente de ellas. Debido a que las nubes de puntos recibidas por la estación central tienen transformaciones distintas en el espacio, es tarea del algoritmo de fusión de mapas, encontrar una transformación entre ellos.

El proceso de fusión se da en tres etapas: alineación, optimización y concatenación. En la etapa de alineación, se utiliza el algoritmo *Sample Consensus Initial Alignment* (SCIA) [Rusu et al., 2009] para estimar una transformación inicial entre las nubes de puntos. En la segunda etapa se utiliza el algoritmo *Iterative Closest Point* (ICP) [Besl and McKay, 1992], el cual compara los vecinos más cercanos entre los puntos de ambas nubes y proporciona otra transformación a un nivel más fino. Finalmente, tras aplicar esta última transformación, las nubes se colocan juntas en un sólo mapa y se eliminan los puntos redundantes [Lawson, 2015].

4.3. Análisis comparativo

A continuación se presenta un análisis de las características principales de los trabajos de exploración estudiados en las Secciones 4.1 y 4.2 . Los trabajos de ambas secciones se comparan en la Sección 4.3.1 mediante el análisis de sus características generales, puesto que ellas afectan el alcance de las propuestas. En la Sección 4.3.2 se presenta el análisis correspondiente a los trabajos de exploración multi-robot. Finalmente, en la Sección 4.3.3 se analizan las características de los trabajos de exploración mediante MAVs.

4.3.1. Análisis de características generales de exploración robótica

Aunque los trabajos descritos en las Secciones 4.1 y 4.2 tratan de manera diferente el problema de exploración, en general, tienen características en común. Estas características influyen en el nivel de complejidad del problema a solucionar (ver Tabla 4.1). La complejidad de cada una de ellas se ve afectada por suposiciones acerca del entorno, uso compartido de información y uso de componentes existentes[8] o entornos simulados. A continuación se presenta un análisis de dichas características. Por cada una de ellas se realiza una breve descripción y se destacan las diferencias principales entre los trabajos estudiados.

Tipo de exploración

Una de las principales diferencias entre las propuestas analizadas es el tipo y cantidad de robots utilizados para explorar. Estas dos características definen los tipos de exploración: multi-robot, multi-MAV y solo MAVs. Se considera que una propuesta es multi-robot cuando utiliza más de un robot para explorar. De manera similar, una propuesta es multi-MAV si utiliza más de un MAV. Finalmente, es solo con MAVs cuando únicamente utiliza un MAV. Dentro las propuestas de exploración mediante MAVs, se observa que la mayoría usa solamente uno de ellos, a excepción del trabajo de Cesare et al., el cual incorpora múltiples MAVs en el proceso de exploración.

Planificación de movimiento

Esta característica toma en cuenta si la propuesta considera elementos de **planificación de movimiento**, e.g., creación de un mapa de rutas. En este sentido, la mayoría de las propuestas hacen uso de un componente existente que se encarga de realizar esta tarea. Por el contrario, Franchi et al., utilizan una representación del espacio libre a través de un SRG (*Sensor-based Random Graph*) que sirve para planear y

[8]En algunos de los trabajos analizados también se realizan propuestas de algoritmos SLAM o de **planificación de movimiento**. Para efectos del presente análisis, cuando dichos algoritmos no están integrados en el proceso de exploración, también son tratados como componentes existentes.

explorar. El SRG es similar al mapa de rutas utilizado en la exploración basada en características con un solo robot (ver Sección 3.3).

Localización en el entorno

La localización se refiere a la forma en que los robots determinan su posición en el mapa. Esta tarea puede ser realizada mediante elementos propios de los trabajos analizados (e.g., marcadores de trayectoria), datos de simulación o componentes externos (e.g., herramientas de navegación que implementan algoritmos SLAM). En su mayoría, las propuestas analizadas confían la localización a componentes externos. Los trabajos que se diferencian en este aspecto son los de Simmons et al. y nuevamente Franchi et al. En el primero de ellos, se utiliza la posición relativa entre robots para determinar su ubicación. Por otra parte, en la segunda propuesta se utilizan los nodos del SRG, que representan lugares en el espacio libre.

Almacenamiento del mapa

Esta característica se refiere al lugar donde se guarda el mapa del entorno. El almacenamiento puede ser de tres tipos: local, cuando cada robot crea su propio mapa, local compartido, cuando los robots intercambian información o global, cuando todos tienen acceso al mismo mapa. De esta característica depende la forma en cómo se generan las metas de exploración.

En los trabajos que utilizan un solo MAV, el almacenamiento del mapa es local. Por otra parte, en las propuestas multi-robot, el almacenamiento puede ser local compartido o global. En la propuesta de Simmons et al., el almacenamiento es local en los robots y el mapa se comparte con el coordinador cada determinado número de observaciones. Otra forma de realizar esta tarea consiste en compartir el mapa cada que los robots se encuentran en rango de comunicaciones, como lo hacen Matignon et al., Franchi et al. y Cesare et al. Finalmente, el almacenamiento global es común en las propuestas con una implementación simulada.

Rango de comunicaciones

El rango involucra el alcance de las comunicaciones inalámbricas entre robots o con una estación base de control. Existe la posibilidad de que el trabajo analizado tome en cuenta el alcance de las redes inalámbricas, limitando el área explorable si se depende de un coordinador central. Cuando se utilizan múltiples robots, una consecuencia de esta limitación es la necesidad de mantener en comunicación a los robots, como en las propuestas de Rooker y Birk y Sheng et al. Una forma diferente de manejar el límite en el rango de comunicaciones es la propuesta de Cesare et al. Ellos incluyen la posibilidad de que un robot continúe explorando a pesar de haber perdido la comunicación, para después regresar a un punto donde pueda intercambiar información.

Tolerancia a fallas en las comunicaciones

Esta característica se refiere a la capacidad de un robot para recuperarse de problemas en la comunicación con el resto de los robots o con una estación base de control. Entre las posibles fallas se encuentran: pérdidas de mensajes, retrasos en la comunicación y desconexiones. En este sentido se observa que ninguno de los trabajos analizados considera la posibilidad de fallas en las comunicaciones. Por el contrario, se asume que la red es estable, con velocidad suficiente y libre de desconexiones. Esta suposición claramente no ocurre en todas las situaciones reales.

Propuesta	**Tipo de exploración**	**Planificación de movimiento**	**Localización en el entorno**	**Almacenamiento del mapa**	**Rango de comunicaciones**	**Tolerancia a fallas en las comunicaciones**
Simmons et al.	Multi-robot	Componente existente	Relativa entre robots	Local y global en el coordinador	Ilimitado	No
Solanas y Garcia	Multi-robot	Componente existente	Simulada	Global	Ilimitado	No
Wurm et al.	Multi-robot	Componente existente	Componente existente	Global	Ilimitado, pero se puede restringir	No
Rooker y Birk	Multi-robot	Componente existente	Simulada	Global	Limitado	No
Sheng et al.	Multi-robot	Componente existente	Simulada	Global	Limitado	No
Matignon et al.	Multi-robot	Componente existente	Componente existente	Local compartido	No especificado	No
Franchi et al.	Multi-robot	A través del SRG	Nodos del SRG	Local compartido	Limitado	No
Cesare et al.	Multi-MAV	Componente existente	Componente existente	Local compartido	Limitado	No
Shen et al.	MAV	Componente existente	Componente existente	Local	Ilimitado y autónomo	No aplica
Bachrach et al.	MAV	Componente existente	Componente existente	Local	Ilimitado y autónomo	No aplica
Zhu et al.	MAV	Componente existente	Componente existente	Local	Ilimitado y autónomo	No aplica
Fraundorfer et al.	MAV	Componente existente	Componente existente	Local	Ilimitado y autónomo	No aplica

Tabla 4.1: Consideraciones de las propuestas de exploración

4.3.2. Análisis de características propias de la exploración multi-robot

En esta sección se analizan las características presentes en los trabajos de exploración multi-robot, los cuales incluyen terrestres y aéreos, como los MAVs. El análisis se enfoca en la forma en cómo los robots realizan el proceso de exploración. Este proceso incluye desde el algoritmo utilizado para generar las metas de exploración, hasta el criterio de asignación de dichas metas a los robots (ver Tabla 4.2).

Tipo de robot

En los trabajos analizados se distinguen dos tipos principales de robot, terrestres y aéreos. La mayoría de dichos trabajos consideran robots terrestres, ya sean simulados o físicos. Sin embargo, la propuesta de Cesare et al. se distingue por emplear múltiples MAVs, i.e., robots aéreos.

Tipo de coordinación

Cuando se trabaja con múltiples robots es necesario establecer mecanismos de coordinación que les permitan cooperar para completar su objetivo. El tipo de coordinación es determinado por el lugar donde se realiza la asignación de metas. Dicha coordinación puede ser centralizada o distribuida. En el caso de la coordinación centralizada, el encargado de realizar esta tarea es un agente que no pertenece al equipo de robots, como en la propuesta de Simmons et al. Por otra parte, en la coordinación distribuida, los robots están encargados de negociar la asignación de metas. Esta negociación se lleva a cabo mediante ofertas, como en la propuesta de Sheng et al., o considerando un mapa compartido, como en el caso de Matignon et al.

Algoritmo de exploración

Esta característica se refiere el algoritmo de exploración en que se basan las propuestas para identificar las partes del entorno que requieren ser exploradas. El algoritmo más utilizado en los trabajos analizados es el de exploración basada en frontera. Dicho algoritmo también es utilizado en las propuestas de Solanas y Garcia, y de Wurm et al. como elemento complementario. Estás dos propuestas se enfocan en la asignación de regiones independientes a explorar. Solanas y García determinan regiones mediante una partición del espacio desconocido, mientras que Wurm et al., utilizan una segmentación del espacio a través de puntos de acceso. En este sentido, un algoritmo de exploración diferente es el de Fracnhi et al., cuyo mapa de rutas en el espacio libre es almacenado en un SRG, el cual no hace uso explícito de un *grid* 2D para definir fronteras, sino que utiliza los arcos del rango del sensor.

Generación de metas

La generación de metas es la forma en que se seleccionan los puntos del espacio que sirven para extender el modelo del entorno. Las metas de exploración para el algoritmo basado en frontera tienen en común ser celdas en el espacio conocido con, al menos, una celda vecina en el espacio desconocido. Sin embargo, debido al número de celdas frontera, se utiliza un criterio para reducir su número y formar metas de exploración. Los criterios encontrados en los trabajos relacionados son la ganancia de información (cantidad de elementos desconocidos, e.g., celdas) y la distancia al robot. Una celda considerada como meta de exploración reduce la posible ganancia de información de sus celdas vecinas. Un caso diferente ocurre en la propuesta de Wurm et al., en la cual las metas son nodos del diagrama de Voronoi. Se intuye, por sus características, que dichos nodos son puertas o corredores que dan acceso a zonas más grandes, explorables de manera independiente. Finalmente, la propuesta de Franchi et al. utiliza, como metas, los arcos desconocidos del borde del área de sensado.

Asignación de metas

La asignación es la forma en qué se decide a qué robot le corresponde cada meta generada en el punto anterior. Al momento de determinar qué robot explora cada meta, el criterio más utilizado es asignar a los robots las metas que generaron. En este enfoque también se considera la ganancia de información del resto del equipo. Simmons et al. y Solanas y García reducen la ganancia de información de las metas candidatas para un robot, si estas se encuentran muy próximas a las de otro robot. La estrategia de asignar las metas al robot que las generó permite realizar exploración distribuida, ya sea mediante ofertas o mapas compartidos. Cuando se explora con grupos de robots, como en la propuesta de Rooker y Birk, se puede caer en conflictos como los candados mortales, los cuales requieren de un método extra para ser resueltos. Otro tipo de inconvenientes son los conflictos entre trayectorias de robots, tratados por la propuesta de Franchi et al., en la cual se elige un robot coordinador para resolverlos.

Objetivo

El objetivo se refiere al propósito de cada trabajo analizado, el cual se alcanza mediante la aplicación de los criterios de generación y asignación de metas. Aunque los objetivos de las propuestas son variados, todos van orientados hacía la cobertura del espacio. Por ejemplo, Simmons et al., Matignon et al. y Franchi et al. mejoran la cobertura del espacio, al minimizar los traslapes de zonas de ganancia de información y la interacción entre robots. De manera similar, Solanas y García y Wurm et al. reducen los traslapes de zonas de ganancia de información mediante la asignación explícita de regiones a explorar. Por otra parte, Cesare et al. consiguen una mejora en la cobertura, al incorporar un estado de sacrificio en los MAVs, el cual les permite llegar más lejos de acuerdo a su rango de comunicaciones. Propuestas diferentes en

el sentido de la cobertura son las de Rooker y Birk y Sheng et al., quienes buscan mantener la comunicación entre equipos de robots durante la exploración.

Propuesta	**Tipo de robot**	**Tipo de coordinación**	**Algoritmo de exploración**	**Generación de metas**	**Asignación de metas**	**Objetivo**
Simmons et al.	Terestre	Centralizada	Frontera y ofertas	Celdas frontera a una distancia fija	Mayor prioridad al robot que generó la frontera	Minimizar el traslape de información
Solanas y Garcia	Terrestre	Centralizada	Frontera y K-means para partición del espacio desconocido	Celdas frontera	Fronteras cercanas al centroide de la región del robot	Balancear carga y dispersar robots en el entorno
Wurm et al.	Terrestre	Centralizada	Frontera y diagrama de Voronoi generalizado	Nodos del diagrama con distancia mínima a los obstáculos	Método húngaro con metas y distancias	Explorar independiente segmentos del entorno
Rooker y Birk	Terrestre	Distribuida	Frontera	Celdas frontera dentro del rango de comunicaciones	Proximidad	Mantener la comunicación entre robots
Sheng et al.	Terrestre	Distribuida	Frontera	Ganancia de información y distancia a celdas frontera	Robot que generó la frontera con mayor ganancia de información	Mantener la comunicación entre robots
Matignon et al.	Terrestre	Distribuida	Frontera con celdas hexagonales	Celdas frontera	Ganancia de información y probabilidad de alcanzar meta	Maximizar la cobertura y minimizar la interacción entre robots
Franchi et al.	Terrestre	Distribuida	SRG	Arcos frontera en línea de visión	Nodo más cercano del SRG con fronteras accesibles y resolución de conflictos	Maximizar la cobertura
Cesare et al.	Aéreo	Distribuida	Frontera/SRG	No especificado	Proximidad con resolución de conflictos	Maximizar el espacio explorado

Tabla 4.2: Tabla comparativa entre propuestas de exploración multi-robot

4.3.3. Análisis de características propias de la exploración mediante MAVs

Las limitaciones de los MAVs se ven reflejadas en características tales como el tipo de autonomía y el entorno que exploran. Debido a que las propuestas de exploración multi-MAV son reducidas, se realiza la comparación con las propuestas que utilizan un solo MAV (ver Tabla 4.3).

Multi-MAV

Esta característica indica si la propuesta es de exploración multi-MAV o no. Como se mencionó anteriormente, la única propuesta de exploración multi-MAV analizada es la de Cesare et al. Al considerar exploración multi-MAV, la suposición de que el mapa es compartido requiere de un mecanismo para combinar las observaciones de los MAVs. Este problema es tratado por Lawson (ver Sección 4.2) quien se utiliza una estación terrestre para el proceso de fusión de mapas. El número reducido de trabajos de exploración multi-MAV puede deberse a que, en el área robótica, los trabajos multi-MAV se han enfocado en otros objetivos de cooperación. Un ejemplo de estos trabajos es el presentado por Saska et al., cuyo objetivo es la vigilancia cooperativa [Saska et al., 2014].

Tipo de autonomía

Esta característica se refiere a la dependencia de una estación terrestre. En general, cuando la exploración no es guiada por un humano, se considera que ésta es autónoma. Sin embargo, en las propuestas mediante MAVs, se hace énfasis en la autonomía total. Este tipo de autonomía implica que todas las tareas de procesamiento se realicen a bordo de los MAVs. En las propuestas analizadas esta característica se cumple para todos ellas, pero no es una característica general de los trabajos con MAVs.

Como ejemplo de la dependencia de una estación terrestre en trabajos de un solo MAV, se tienen las propuestas de Heng et al. y Bachrach et al., donde se construyen mapas. En la primera propuesta, la estación terrestre es la encargada de procesar las observaciones para agregar textura al mapa. En la segunda propuesta, un algoritmo SLAM se ejecuta en la estación terrestre para corregir las estimaciones de posición del MAV [Bachrach et al., 2012, Heng et al., 2011]. El análisis deja ver que la autonomía de un MAV depende más de los algoritmos SLAM y de planificación de trayectoria que de los algoritmos de exploración. Esta situación se debe a que los primeros son más demandantes en términos computacionales.

Entorno

El tipo de entorno es clasificado de acuerdo a la representación del espacio utilizada. Es 2.5D cuando a una altura fija de vuelo se explora "una *rebanada* 2D del espacio

3D" [Bachrach et al., 2009]. En los trabajos analizados predomina la consideración de un entorno 2.5D, a excepción de las propuestas de Shen et al. y Zhu et al. que utilizan una representación 3D. En la selección del entorno influyen aspectos tales como: el algoritmo SLAM utilizado para construir el mapa y el algoritmo de **planificación de movimiento**. Adicionalmente, dado que la autonomía total es una característica deseada en MAVs, la identificación de fronteras en un entorno 2D permite ahorrar recursos computacionales.

Algoritmo de exploración

De manera similar a las propuestas multi-robot, esta característica indica el algoritmo utilizado para definir metas de exploración. El algoritmo de exploración más utilizado en MAVs es el basado en frontera. Fraundorfer et al. además utilizan un algoritmo de seguimiento de paredes, lo que simplifica el proceso de exploración en entornos abiertos con pocas características (salientes, bordes, marcos de ventanas, entre otros). Un algoritmo diferente al basado en frontera tradicional es el de Shen et al. Ellos proponen el uso de partículas con dinámicas que simulan el comportamiento de un gas ideal para identificar "fronteras", en el sentido de ser lugares que extienden el modelo del entorno.

Generación de metas

Esta característica se refiere a la forma en que se seleccionan los puntos del espacio que sirven para extender el modelo del entorno. El criterio de generación de metas, utilizado por Shen et al., define las regiones que requieren ser exploradas por su baja concentración de partículas. En las propuestas de Zhu et al. y Fraundorfer et al., se agrupan las celdas frontera, lo cual ayuda a reducir el número de lugares a explorar. Por otra parte, Bachrach et al. además consideran las posiciones de un MAV y conservan su capacidad de localización en el entorno al mantenerlo cerca de la estructura del lugar. Las posiciones influyen puesto que estas cambian la orientación del MAV, la cual afecta la ganancia de información. Un ejemplo de esta afectación ocurre cuando el MAV se encuentra orientado hacia una pared. En dicha situación la información que el MAV puede adquirir es muy poca en comparación con una orientación hacia una zona despejada. La capacidad de localización de los MAVs en el entorno es importante debido a que, en la práctica, los algoritmos SLAM requieren de características del ambiente para distinguir dónde se encuentran.

Selección de metas

Después de generar metas candidatas, el algoritmo de exploración mediante MAVs necesita priorizar aquellas a las cuales se desplazará. Para tomar esta decisión, en general los trabajos analizados utilizan como criterio la cercanía entre MAVs y metas. Zhu et al. además incluyen la ganancia de información en forma de celdas desconocidas observadas desde las fronteras.

Propuesta	Multi-MAV	Tipo de autonomía	Entorno	Algoritmo de exploración	Generación de metas	Selección entre metas
Shen et al.	No	Total	3D	Partículas con dinámicas definidas por SDE	Zonas con baja densidad de partículas	Proximidad
Bachrach et al.	No	Total	2.5D	Frontera	Posiciones en las fronteras y capacidad de localización	Proximidad
Zhu et al.	No	Total	3D	Frontera	Agrupamiento de celdas frontera	Proximidad y celdas desconocidas en la frontera
Fraundorfer et al.	No	Total	2.5D	Frontera y seguimiento de paredes	Agrupamiento de celdas frontera	Proximidad
Cesare et al.	Si	Total	2.5D	Frontera y SRG	No especificado	Proximidad

Tabla 4.3: Tabla comparativa entre propuestas de exploración mediante MAVs

Capítulo 5

Diseño del sistema de exploración multi-MAV

En este capítulo, se presenta el diseño del sistema de exploración multi-MAV propuesto en esta tesis. En la Sección 5.1, se describen los procesos de la arquitectura del sistema, así como la interacción entre dichos procesos. En la Sección 5.2, se explican las tareas realizadas por el coordinador, las cuales incluyen la asignación de metas y la generación de regiones para los MAVs. En la Sección 5.3, se detallan las funciones del proceso de exploración que son realizadas por los MAVs y que consisten principalmente en explorar y generar nuevas metas. En la Sección 5.4, se describe el proceso de limpieza del mapa, utilizado para remover las celdas producto del mapeo mutuo entre MAVs. En la Sección 5.5, se explica la condición de término de la exploración multi-MAV, la cual se basa en las respuestas enviadas por los MAVs al coordinador. Por último, en la Sección 5.6 se discuten las características del diseño del sistema de exploración propuesto.

5.1. Arquitectura del sistema

En esta sección se describe la arquitectura del sistema de exploración multi-MAV. En la Sección 5.1.1 se presentan los procesos principales del sistema para una mejor comprensión del mismo y en la Sección 5.1.2 se describe su funcionamiento a nivel general.

5.1.1. Procesos del sistema

El sistema de exploración multi-MAV se compone básicamente de tres elementos: un **proceso de coordinación** que se ejecuta en una estación terrestre, un **proceso de exploración** que se ejecuta en cada MAV y un **proceso de limpieza** del mapa, el cuál se ejecuta en el contexto de un componente existente de mapeo, presente tanto en los MAVs como en la estación terrestre. Este componente de mapeo debe proporcionar funciones para convertir nubes de puntos con un marco de referencia en un mapa

de ocupación. Dichos elementos interactuan entre sí intercambiando información en forma de metas de exploración (ver Figura 5.1).

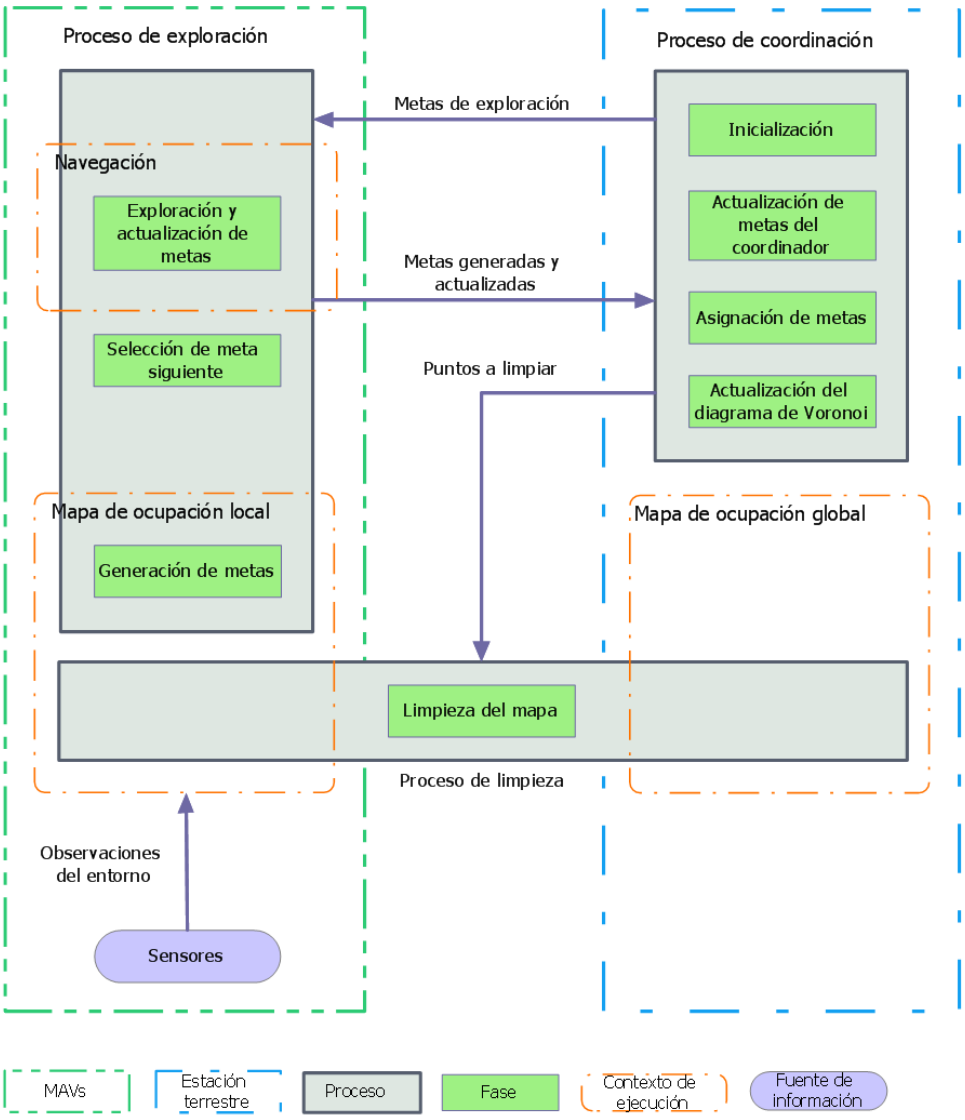


Figura 5.1: Arquitectura del sistema de exploración multi-MAV.

Las **metas de exploración** representan puntos en el espacio 3D que, al ser explorados, extienden el modelo del entorno. Dichas metas, como se verá con mayor detalle en la Sección 5.3.2, se forman a partir de celdas frontera con una distancia mínima entre ellas y poseen un conjunto de datos con información relevante para el proceso de exploración (ver Tabla 5.1). Estos datos permiten conocer la posición de una meta en el espacio, el MAV que la generó, a quién se le asignó y su estado. Los estados

posibles de una meta son: nueva, asignada, descartada, inaccesible y explorada (ver Tabla 5.2). Las transiciones entre dichos estados están definidas por las acciones del coordinador y los MAVs (ver Figura 5.2).

Propiedad	**Tipo de dato**	**Descripción**
`posición`	Punto en 3D	Posición de la meta
`metaID`	Entero positivo	Posición de la meta
`mavAsignadoID`	Entero positivo	Identificador del MAV al que fue asignada la meta
`mavOrigenID`	Entero positivo	Identificador del MAV que generó la meta
`estado`	Entero positivo	Estado de la meta: nueva (0), asignada (1), descartada (2), inaccesible (3), explorada(4)
`cambioEstado`	booleano	Indica si el estado de la meta ha cambiado
`asignada`	booleano	Indica si la meta debe ser explorada o ignorada

Tabla 5.1: Datos de las metas de exploración

Estado	**Descripción**	**Establecido por**
Nueva	Meta que ha sido creada pero aún no tiene asignado un MAV para que la explore	MAVs, con excepción de la etapa de inicialización
Asignada	Meta a la que se le ha asignado el identificador de un MAV en la variable `mavAsignadoID` para su exploración	MAVs y coordinador
Descartada	Meta que resulta de poca utilidad explorar, debido a su proximidad con otras metas	Coordinador
Innaccesible	Meta para la cuál un MAV no fue capaz de encontrar un plan de movimiento	MAVs
Explorada	Meta a cuya posición un MAV se ha desplazado y rotado para adquirir nueva información	MAVs

Tabla 5.2: Estados de las metas de exploración.

Una meta se identifica de manera única mediante su identificador de meta y el identificador del MAV que la generó. Adicionalmente, las metas poseen una propiedad para identificar si su estado ha cambiado, con respecto a la última vez que fueron actualizadas. Dicha propiedad es de utilidad al momento de seleccionar las metas que se envían de los MAVs al coordinador y viceversa.

El **proceso de coordinación** tiene como función principal controlar la exploración a través de pasos de exploración. Un **paso de exploración** es el tiempo transcurrido desde que se asignan regiones y metas a los MAVs hasta que se reciben las respuestas de dichos MAVs. Durante este periodo de tiempo el coordinador reúne la información adquirida por los MAVs, en forma de metas de exploración generadas y actualizadas. Los pasos de exploración además proporcionan una ventana de tiempo en la cual los MAVs pueden explorar metas en su región asignada, con la garantía de

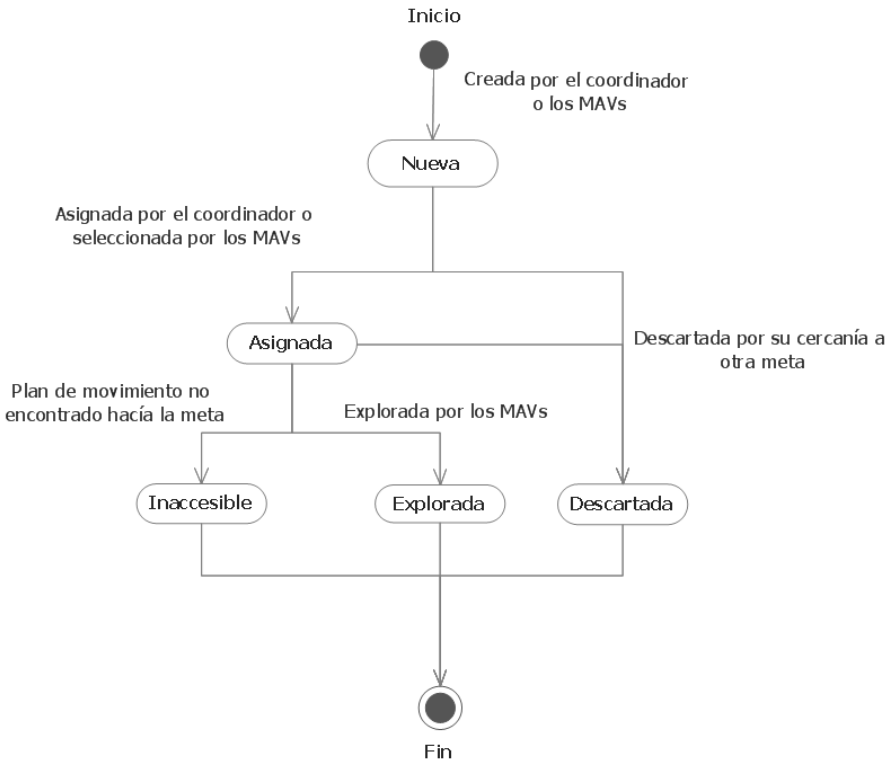


Figura 5.2: Diagrama de estados de una meta de exploración.

que otro MAV no las explorará.

Las regiones asignadas a los MAVs son regiones de un diagrama de Voronoi 3D generado al inicio de cada paso de exploración. A cada MAV le corresponde únicamente una región del diagrama de Voronoi. La posición de las metas que el coordinador asigna a los MAVs, al inicio de un paso de exploración, es la misma que la de los sitios del diagrama de Voronoi. Mediante esta relación se determina que a cada MAV le corresponde la región de Voronoi, cuyo sitio es la posición de la meta que el coordinador le asignó.

El **proceso de limpieza** se encarga de eliminar las celdas ruido presentes en el mapa. Estas celdas son consecuencia del problema de mapeo mutuo, en el cual los MAVs se detectan entre sí como parte del mapa. Las celdas ruido poseen un identificador que ayuda a determinar a qué MAV corresponden. Dichas celdas se marcan como libres y se almacenan en una lista para obtener estadísticas posteriores.

5.1.2. Descripción general del sistema

La exploración del entorno es guiada a través de pasos de exploración. En cada paso, el coordinador interactua con los MAVs de dos formas: enviando metas para su exploración y recibiendo metas nuevas y actualizadas. Por otra parte, los MAVs se encargan de explorar y generar nuevas metas. Durante el desplazamiento de los MAVs, es posible que ocurra mapeo mutuo, por lo tanto al final de cada paso se aplica el **proceso de limpieza** del mapa. Este proceso es aplicado tanto en el mapa del coordinador como en los mapas locales de los MAVs, utilizando las posiciones de las metas exploradas (ver Algoritmo 5.1).

Al inicio de la exploración, el coordinador crea una meta para cada MAV y usa estas metas como sitios para construir un diagrama de Voronoi. Dichas metas son enviadas a los MAV para su exploración (ver línea 7 del Algoritmo 5.1). Mientras los MAVs exploran las metas, el coordinador permanece a la espera de que completen su **proceso de exploración**. Cuando los MAVs finalizan dicho proceso, envían al coordinador, como respuesta, un conjunto de metas nuevas y actualizadas (ver línea 9 del Algoritmo 5.1).

El **proceso de exploración** consiste en explorar y actualizar metas, mientras se generan nuevas metas (ver Figura 5.10). La exploración de una meta consiste en que un MAV se desplace a ella y realice una rotación de 360° con respecto a su eje Z. Esta rotación es de utilidad para adquirir más información del ambiente. El intento de exploración produce un cambio de estado en la meta, ya sea que se pueda explorar o se marque como inaccesible. Durante el desplazamiento a las metas asignadas, los MAVs generan nuevas metas usando un algoritmo de exploración basada en frontera. Los MAVs exploran k metas en su región asignada antes de volver a interactuar con el coordinador, por lo tanto cuentan con un mecanismo de selección de la siguiente meta (ver línea 21 del Algoritmo 5.1).

Una vez exploradas las k metas por parte de los MAVs, estos últimos seleccionan todas aquellas metas que fueron generadas o actualizadas para su envío al coordinador. Cuando el coordinador las recibe da inicio la fase de actualización de metas en la cual descarta aquellas metas que están muy cercanas entre sí. Posteriormente, e. coordinador selecciona nuevas metas a explorar para cada MAV y asigna nuevas regiones actualizando el diagrama de Voronoi (ver línea 15 del Algoritmo 5.1).

El **proceso de coordinación** se repite enviando a los MAVs las metas asignadas y actualizadas, incluyendo aquellas que pertenecen a otros MAVs. El propósito de enviar a todos los MAVs las metas, que fueron actualizadas por otros miembros del equipo, es que puedan explorarlas si se encuentran en su región asignada. La exploración termina cuando las respuestas de todos los MAVs son vacías. En la Sección 5.5 se presenta más información acerca de esta condición de término.

Algoritmo 5.1 Exploracion multi-MAV

Requiere: Lista de MAVs $\mathcal{M}$

procedure COORDINAR($\mathcal{M}$) ▷ Coordinador
  $g \leftarrow \emptyset$ ▷ Lista de metas
  $V, V_{\boldsymbol{ext}} \leftarrow \emptyset$
  $\mathcal{G}_{\mathcal{M}} \leftarrow$ Metas iniciales para cada MAV
  $\mathcal{G}.add(\mathcal{G}_{\mathcal{M}})$
  $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}} \leftarrow ActualizarDiagramaVoronoi(V, V_{\boldsymbol{ext}}, \mathcal{G}_{\mathcal{M}})$
  for each $m \in \mathcal{M}$ **do** $ExplorarEntorno(m, \mathcal{G}_{\mathcal{M}}, \emptyset)$
  do
    $\mathcal{G}_{\mathcal{R}} \leftarrow$ Esperar respuestas de todos los MAVs de $\mathcal{M}$
    if $\mathcal{G}_{\mathcal{R}} \neq \emptyset$ **then**
      ▷ Agregar metas exploradas y generadas y descartar traslapes
      $\mathcal{G} \leftarrow ActualizarMetasCoordinador(\mathcal{G}_{\mathcal{R}}, \mathcal{G})$
      $\mathcal{G}_{\mathcal{M}} \leftarrow AsignarMetasCoordinador(\mathcal{M}, \mathcal{G})$
      $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}} \leftarrow ActualizarDiagramaVoronoi(V, V_{\boldsymbol{ext}}, \mathcal{G}_{\mathcal{M}})$
      $\mathcal{G}_{\mathcal{A}} \leftarrow ObtenerMetasActualizadas(\mathcal{G})$
      for each $m \in \mathcal{M}$ **do** $ExplorarEntorno(m, \mathcal{G}_{\mathcal{M}}, \mathcal{G}_{\mathcal{A}})$
    $LimpiarMapa(\mathcal{G}_{\mathcal{L}})$ ▷ $\mathcal{G}_{\mathcal{L}}$: posiciones de las metas exploradas
  while $\mathcal{E}_{\mathcal{G}} \neq \emptyset$
  Enviar señal de aterrizaje a los MAVs de $\mathcal{M}$
  Desplegar "Exploración finalizada"

function EXPLORARENTORNO($m, \mathcal{G}_{\mathcal{M}}, \mathcal{G}_{\mathcal{A}}$)
  $\mathcal{G} \leftarrow ActualizarMetasMAV(\mathcal{G}, \mathcal{G}_{\mathcal{A}})$ ▷ $\mathcal{G}$ es local al MAV
  $g_{\boldsymbol{m}} \leftarrow$ Meta que le corresponde explorar al MAV m de $\mathcal{G}_{\mathcal{M}}$
  if $g_{\boldsymbol{m}}.asignada$ **then** ▷ Paralelamente $ProcesarCambioMapa(\mathcal{C}_{\mathcal{M}})$
    $\mathcal{V} \leftarrow$ Diagrama de Voronoi usando posiciones de $\mathcal{GM}$ como sitios
    $i \leftarrow 0$
    do
      $\mathcal{G} \leftarrow ExplorarMeta(m, g_{\boldsymbol{m}}, \mathcal{G})$
      $g_m \leftarrow SeleccionarMetaSiguiente(\mathcal{M}, \mathcal{V}, \mathcal{G})$
      $i \leftarrow i + 1$
    while $i < k$ **and** $g \neq NULL$ ▷ k es un número predefinido
  $\mathcal{G}_{\mathcal{R}} \leftarrow ObtenerMetasActualizadas(\mathcal{G})$
  return $\mathcal{G}_{\mathcal{R}}$

5.2. Proceso de coordinación

El sistema de exploración presentado en este capítulo es un sistema de exploración multi-MAV 3D. En dicho sistema, la exploración realizada por los MAVs es coordinada de manera centralizada. Mediante este enfoque, los MAVs puedan explorar de manera independiente regiones asignadas por el coordinador, el cual es una estación de cómputo en tierra.

Con respecto al mapeo y a la localización en el entorno, se hace la suposición de que ambos datos son proporcionados por un componente SLAM existente en cada MAV. Esta suposición implica que la posición de los MAVs en su respectivo mapa local es conocida. Por otra parte, también se hace la suposición de que las posiciones de los MAVs y sus mapas son relativos a un marco de referencia global, i.e., las transformaciones entre los mapas de cada MAV son conocidas. Esta suposición significa que el coordinador central no necesita realizar tareas adicionales de mezcla de mapas, como en la propuesta de Lawson, descrita en la Sección 4.2 [Lawson, 2015].

En el sistema propuesto, no se incluyen mecanismos de tolerancia a fallas en las comunicaciones inalámbricas con el coordinador central. Finalmente, se hace la suposición de que, en todo momento, los MAVs pueden intercambiar información con el coordinador central, i.e., no hay restricciones en el alcance de la comunicación.

En esta sección se describe el **proceso de coordinación**, el cual consta de cuatro fases principales (ver Figura 5.3). La primera fase, **inicialización**, se describe en la Sección 5.2.1 y en ella se crean las primeras metas para los MAVs. La segunda fase es la de **actualización de metas** y se describe en la Sección 5.2.2. En esta fase se procesan las respuestas recibidas de los MAVs, las cuales consisten en las metas que exploraron y generaron. Por otra parte, en la Sección 5.2.3, se describe la tercera fase llamada **asignación de metas**, la cual se enfoca en determinar qué meta, del conjunto de metas disponibles, le corresponde explorar a cada MAV. En la Sección 5.2.4 se describe la última fase, **actualización del diagrama de Voronoi**, que consiste en generar nuevos diagramas después de actualizar sus cajas envolventes. Estas fases, con excepción de la de **inicialización**, se realizan cuando el coordinador recibe las respuestas de los MAVs; durante el resto del tiempo, el coordinador permanece en espera de respuestas (ver Figura 5.4).

5.2.1. Inicialización

El propósito de la fase de **inicialización** es crear los elementos necesarios para comenzar la exploración (ver Algoritmo 5.2). Durante el **proceso de coordinación**, el coordinador mantiene una lista $\mathcal{G}$ en la que almacena las metas que ha enviado y recibido de los MAVs. Al comienzo de la exploración, esta lista está vacía y, excepto en el **paso de exploración** inicial, es actualizada conforme se reciben metas nuevas y exploradas procedentes de los MAVs (ver línea 2 del Algoritmo 5.2). El almacena-

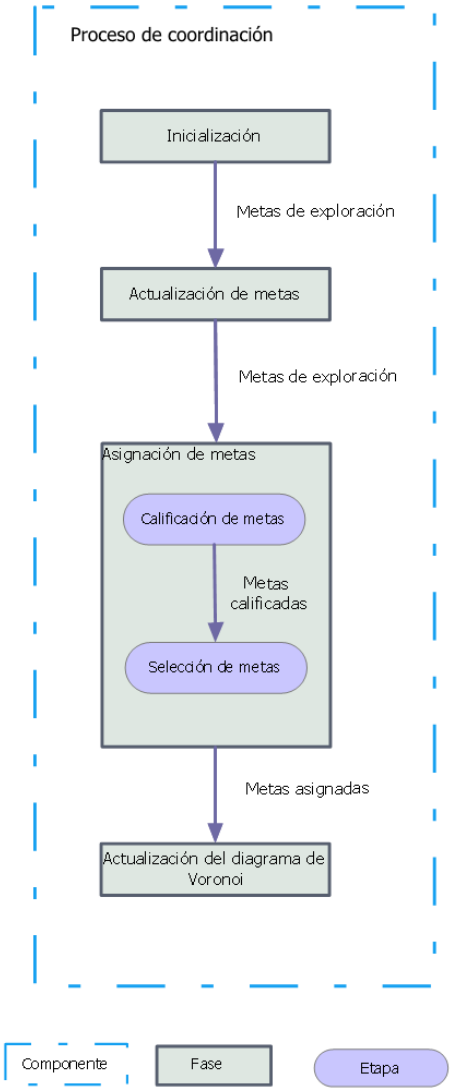


Figura 5.3: Fases del proceso de coordinación.

miento de las metas es necesario para saber cuáles de ellas ya han sido exploradas y cuáles requieren ser exploradas.

En el paso de exploración inicial, el coordinador crea una meta para cada MAV y la agrega a $\mathcal{G}$. Dichas metas son creadas en el espacio libre, a una altura fija sobre los MAVs (ver línea 4 del Algoritmo 5.2). Estas metas les permitirán a los MAVs despegar y comenzar a adquirir información de utilidad para explorar. Con excepción de la fase de inicialización, el coordinador nunca crea metas, solamente se encarga de asignarlas.

Utilizando las metas iniciales, el coordinador crea dos diagramas de Voronoi, $\mathcal{V}$ y

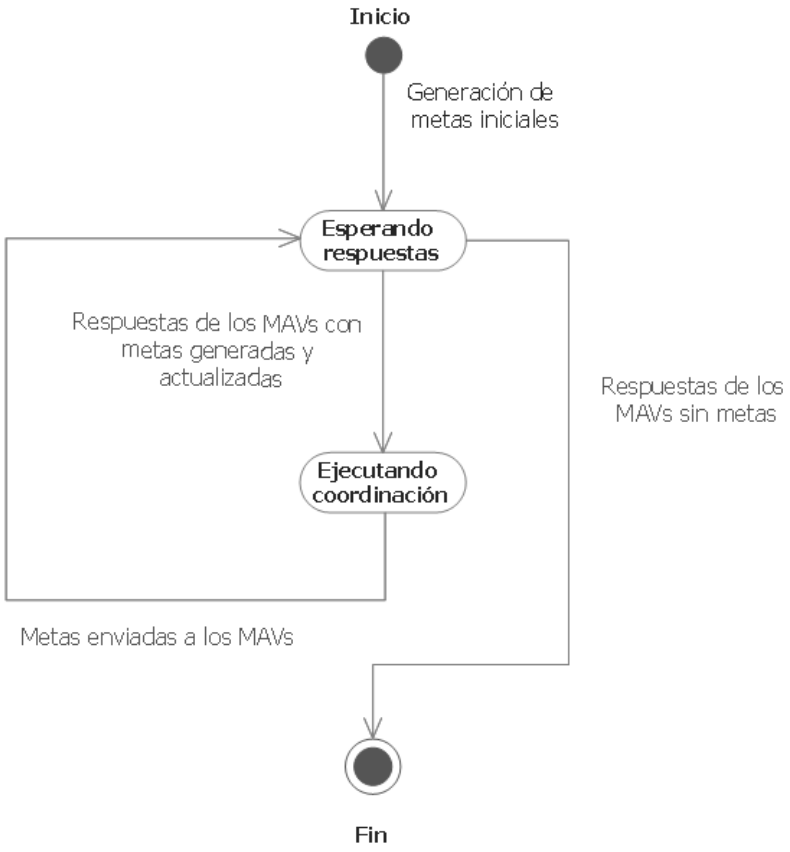


Figura 5.4: Diagrama de estados del coordinador.

$\mathcal{V}_{ext}$, que almacena para su uso posterior en la fase de **asignación de metas** (ver línea 13 del Algoritmo 5.2). La creación de los diagramas es realizada de forma similar a la fase **actualización del diagrama de Voronoi** (ver Sección 5.2.4). La información del diagrama $\mathcal{V}$ es necesaria en los MAVs para identificar las regiones asignadas. El coordinador no envía de manera explícita dicho diagrama a los MAVs, sino únicamente los sitios para su reconstrucción en dichos MAVs.

Al final de esta fase el coordinador envía a los MAVs la información inicial, la cual consiste en las metas que deberán explorar (ver línea 14 del Algoritmo 5.2). A pesar de que cada MAV recibe también las metas del resto de sus compañeros, únicamente explora y almacena la que le corresponde, de acuerdo a la propiedad `mavAsignadoID` de las metas. Es necesario que los MAVs reciban las metas de todo el equipo, puesto que estas representan los sitios del diagrama de Voronoi. Durante la fase de inicialización, los MAVs se encuentran en tierra esperando las metas. Una vez que los MAVs reciben las metas, ellos comienzan el **proceso de exploración**

Algoritmo 5.2 Inicialización del proceso de coordinación

Requiere: Lista de MAVs $\mathcal{M}$

1: **procedure** InicializarCoordinacion($\mathcal{M}$)
2: $g \leftarrow \emptyset$ ▷ Lista de metas generadas por los MAVs
3: $\mathcal{G}_{\mathcal{M}} \leftarrow \emptyset$ ▷ Metas iniciales para los MAVs
4: **for each** $m \in \mathcal{M}$ **do**
5: $g.posicion \leftarrow m.position$
6: $g.posicion.z \leftarrow h$ ▷ h es un valor predefinido
7: $g.mavAsignadoID \leftarrow m.id$
8: $g.mavOrigenID \leftarrow -1$
9: $m.estado \leftarrow$ NUEVA
10: $m.metaID \leftarrow 0$
11: $\mathcal{G}_{\mathcal{M}}.add(m)$
12: $\mathcal{G}.add(\mathcal{G}_{\mathcal{M}})$
13: $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}} \leftarrow ActualizarDiagramaVoronoi(\mathcal{G}_{\mathcal{M}})$
14: **for each** $m \in \mathcal{M}$ **do** $ExplorarEntorno(m, \mathcal{G}_{\mathcal{M}}, \emptyset)$

mientras el coordinador queda en espera de las respuestas de los MAVs.

5.2.2. Actualización de metas

Cada respuesta recibida por el coordinador, de parte de los MAVs, consiste en las metas que generaron y actualizaron. Estas metas pueden ser nuevas o existentes para el coordinador y requieren ser agregadas o actualizadas, respectivamente en su lista de metas. Además, debido a que no hay comunicación entre MAVs, dichas metas pueden encontrarse muy cercanas entre sí (ver Figura 5.5). Esta situación ocasiona que la información, que cada meta puede aportar, sea aproximadamente la misma, por lo tanto es necesario descartar algunas de ellas (ver Algoritmo 5.3).

El criterio para la actualización de metas en el coordinador es el siguiente. Por cada meta recibida $g_{\boldsymbol{r}}$ se hace una búsqueda en la lista de metas, utilizando una esfera con centro $g_{\boldsymbol{r}}$ y radio predefinido $\boldsymbol{r}$ (ver línea 3 del Algoritmo 5.3). Este radio $\boldsymbol{r}$ es la distancia mínima permitida entre metas. A continuación, por cada meta encontrada $g_{\boldsymbol{e}}$ en la esfera especificada, se actualizan los estados de $g_{\boldsymbol{r}}$ y $g_{\boldsymbol{e}}$ de acuerdo a la Tabla 5.3 (ver línea 5 del Algoritmo 5.3). En dicha tabla se muestra el estado que corresponde a ambas metas, de acuerdo a su estado inicial en el formato $g_{\boldsymbol{r}}/g_{\boldsymbol{e}}$. El guión medio (**-**) indica que conservan su mismo estado. En dicha tabla se contemplan dos casos principales, en los cuales puede ocurrir traslape de información: el primero, debido a metas candidatas generadas por MAVs diferentes y el segundo, debido a que una meta candidata puede caer en el rango de otra ya explorada.

Algoritmo 5.3 Actualización de metas en el coordinador

Requiere: Lista de metas recibidas $\mathcal{G}_{\mathcal{R}}$, lista de metas del coordinador $\mathcal{G}$

```
1: procedure ActualizarMetasCoordinador(G_R, G )
2:     for each g_r ∈ G_R do
3:         G_E ← RadiusSearch(g_r, r)                    ▷ r es un valor predefinido
4:         for each g_e ∈ G_R do
5:             resultado ← tablaEstados[g_r.estado][g_e.estado]        ▷ ver Tabla 5.3
6:             g_r.estado ← resultado.g_r
7:             g_e.estado ← resultado.g_e
8:             g_e.cambioEstado ← true si el estado resultante es distinto al inicial
9:         G.add(m)
10:    g_r.cambioEstado ← true
11:    ActualizarListaMetas(g_r)

12: function RadiusSearch(g_r, r )
13:    G_E ← ∅
14:    for each g ∈ G do
15:        if d(g.posicion, g_r.posicion) < r then
16:            G_E.add(g)
17:    return G_E

18: procedure ActualizarListaMetas(g_r )
19:    encontrada ← false
20:    for each g ∈ G do
21:        if g.metaID = g_r.metaID and g.mavOrigenID = g_r.mavOrigenID
   then
22:            g ← g_r
23:            encontrada ← true
24:    if not encontrada then
25:        G.add(g_r)
```

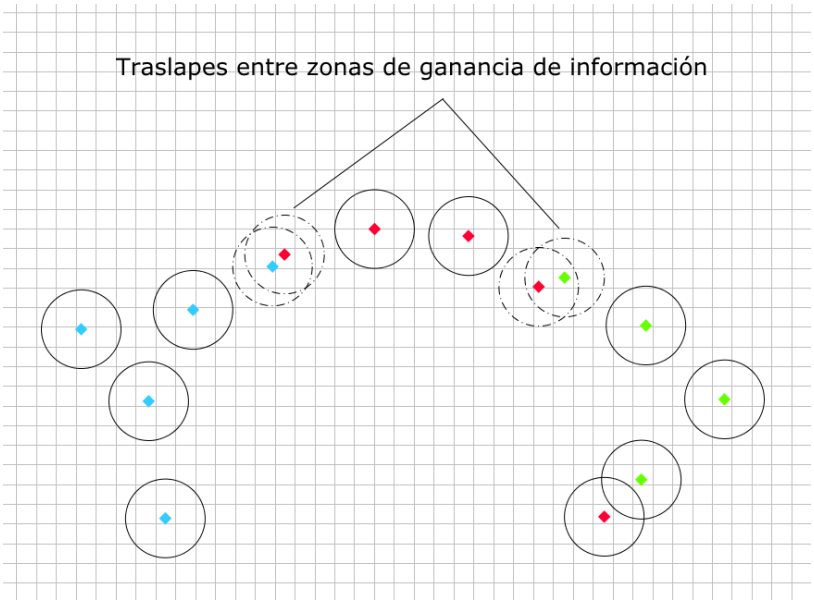


Figura 5.5: Traslape entre zonas de ganancia de información de metas de exploración.

Meta recibida (g_r) \ Meta encontrada (g_e)	**Nueva**	**Asignada**	**Descartada**	**Inaccesible**	**Explorada**
Nueva	-/Descartada	Descartada/-	-/-	-/-	Descartada/-
Asignada	-/Descartada	-/Descartada	-/-	-/-	Descartada/-
Descartada	-/-	-/-	-/-	-/-	-/-
Inaccesible	-/-	-/-	-/-	-/-	-/-
Explorada	-/Descartada	-/Descartada	-/-	-/-	-/-

Tabla 5.3: Tabla de estados resultantes para metas recibidas y existentes en el formato g_r /g_e

La propiedad `cambioEstado` se marca para las metas que adquirieron un estado diferente al que tenían. Esta marca es de utilidad para identificar a las metas que posteriormente serán enviadas a los MAVs (ver línea 8 del Algoritmo 5.3). A continuación, se procede a agregar o actualizar g_r en la lista de metas $\mathcal{G}$, según corresponda (ver línea 18 del Algoritmo 5.3). Cabe destacar que durante la actualización de metas es posible que lleguen las respuestas de otros MAVs, por lo tanto es necesario un mecanismo de control de concurrencia, e.g., *mutex*.

5.2.3. Asignación de metas

La fase de asignación de metas comienza una vez que se reciben y actualizan todas las metas provenientes de los MAVs (ver Algoritmo 5.4). Esta fase consiste en determinar

el siguiente punto al cual se moverán los MAVs y se realiza en dos etapas: calificación de metas y selección de metas. Para lograr este objetivo, en primer lugar se busca la reducción de la interacción entre MAVs y, en segundo lugar, la cooperación para explorar una zona. Mientras el coordinador realiza esta fase de asignación, los MAVs se mantienen volando en su última posición.

Calificación de metas

En esta etapa se utiliza una función de evaluación, que asigna una calificación a cada meta del conjunto de metas disponibles (nuevas) $\mathcal{G}_\mathrm{N}$ (ver línea 5 del Algoritmo 5.4). Dicha función considera la cercanía de las metas a las posiciones de los MAVs y si están contenidas en las regiones asignadas (ver Algoritmo 5.5). Por cada MAV $m \in \mathcal{M}$ se calculan tres puntuaciones $s_{\boldsymbol{g}}$, $s_{\boldsymbol{m}}$ y $s_{\boldsymbol{v}}$ para cada meta $g_{\boldsymbol{n}} \in \mathcal{G}_\mathrm{N}$. La suma ponderada de estas puntuaciones da como resultado la calificación total de la meta. Finalmente, dichas metas son agregadas a una lista $\mathcal{G}_\mathcal{C}$ y ordenadas de manera descendente (ver línea 13 del Algoritmo 5.5).

Las puntuaciones $s_{\boldsymbol{g}}$ y $s_{\boldsymbol{m}}$ se calculan utilizando distancias normalizadas. Los valores con respecto a los cuales se normalizan dichas puntuaciones son las distancias máximas (ver Figura 5.6). Estas distancias son para cada m y g_n, por lo tanto se crean dos conjuntos de dichas distancias, $\mathcal{D}_{m\_max}$ de tamaño $|\mathcal{M}|$ y $\mathcal{D}_{g\_max}$ de tamaño $|\mathcal{G}_\mathrm{N}|$ (ver línea 15 del Algoritmo 5.5). El primer conjunto es obtenido para cada posición del MAV m, con respecto a las posiciones de las metas disponibles $\mathcal{G}_\mathrm{N}$ (ver línea 21 del Algoritmo 5.5). El segundo conjunto es obtenido para cada meta g_n, con respecto a las posiciones de los MAVs (ver línea 23 del Algoritmo 5.5).

La primera puntuación s_g representa la distancia de la posición del MAV m a la posición de la meta g_n. Este valor es la distancia euclidiana $d(m, g_n)$ normalizada en el rango de 0 a 1. Para normalizar dicha distancia, se utiliza la distancia $d_{m\_max} \in \mathcal{D}_{m\_max}$ a la meta más lejana con respecto a la posición del MAV m. De esta forma, cada primera puntuación s_g es el resultado de la división $d(m, g_d)/d_{m\_max}$ (ver línea 8 del Algoritmo 5.5).

La segunda puntuación s_m, similar a s_g, también es la distancia normalizada del MAV m a la meta g_n, pero cambia el valor utilizado para normalizar. En este caso se comparan las distancias del resto de los MAVs a la misma meta, mientras que en el caso anterior son las distancias de las metas al mismo MAV. Por esta razón, el valor utilizado para normalizar es la distancia $d_{g\_max} \in \mathcal{D}_{g\_max}$ del MAV más lejano a la meta $g_{\boldsymbol{n}}$. Finalmente, $s_{\boldsymbol{m}}$ es el resultado de la división $d(m, g_n)/d_{g\_max}$ (ver línea 9 del Algoritmo 5.5).

El último valor, s_v, utilizado para evaluar a una meta consiste en determinar si una meta se encuentra en la región asignada al MAV. Este valor se calcula utilizando

Algoritmo 5.4 Asignación de metas en el coordinador

Requiere: Lista de MAVs $\mathcal{M}$, lista de metas del coordinador $\mathcal{G}$

```
1: function AsignarMetasCoordinador(M, G )
2:     G_N ← Metas de G que tengan estado NUEVO
3:     for each m ∈ M do m.G_N ← G_N
4:     for each m ∈ M do for each g_n ∈ m.G_N do g_n.eval_ID ← m.ID
5:     G_C ← CalificarMetas(M)
6:     G_A ← ∅                                              ▷ Asignaciones
7:     for each g_c ∈ G_C do
8:         encontrada ← false
9:         for each g_a ∈ m.G_A do
10:            mismaMeta ← g_a.metaID = g_c.metaID and g_a.mavOrigenID = g_c.mavOrigenID
11:            if g_a.mavAsignadoID = g_c.eval_ID or mismaMeta then
12:                encontrada ← true
13:        if not encontrada then
14:            g ← g_c
15:            g.mavAsignadoID ← g_c.eval_ID
16:            g.asignada ← true
17:            G_A.add(g)
18:    for each m ∈ M do          ▷ MAVs que no fueron asignados a una meta
19:        encontrado ← false
20:        for each g_a ∈ m.G_A do
21:            if m.ID = g_a.mavAsignadoID then
22:                encontrado ← true
23:        if not encontrado then                  ▷ Se crea una meta sin posición
24:            g.mavAsignadoID ← m.ID
25:            g.asignada ← false
26:            g.posicion ← NULL
27:            G_A.add(g)
28:    return G_A
```

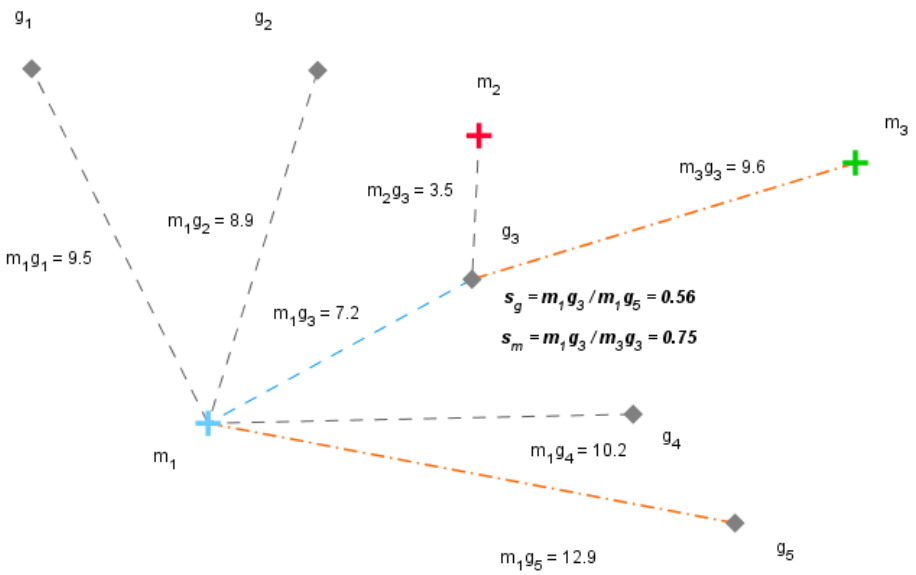


Figura 5.6: Puntuaciones $s_{\boldsymbol{g}}$ y $s_{\boldsymbol{m}}$ de la meta g_3 evaluada para el MAV m_1

el diagrama de Voronoi extendido (ver Sección 5.2.4). Si la meta g_n evaluada para el MAV m se encuentra dentro de la región extendida de m, se asigna el valor 1 a s_v; en caso contrario se asigna cero (ver línea 10 del Algoritmo 5.5).

Selección de metas

Una vez que se tienen las metas calificadas y ordenadas en la lista $\mathcal{G}_{\mathcal{C}}$, se procede a realizar una selección de las mejores para cada MAV. El contenido de dicha selección se almacena en una lista $\mathcal{G}_{\mathcal{A}}$, donde para cada $g_{\boldsymbol{a}} \in \mathcal{G}_{\mathcal{A}}$ se coloca, en la propiedad `mavAsignadoID`, el identificador del MAV que estará encargado de explorarla. En esta fase, es posible que no se asignen metas a algunos MAVs. Esta situación ocurre cuando es mayor el número de MAVs que de metas asignables.

En la lista $\mathcal{G}_{\mathcal{C}}$, las metas se encuentran repetidas $|\mathcal{M}|$ veces, pero difieren en el MAV que las evaluó. Para decidir qué meta le corresponde explorar a cada MAV, se recorre $\mathcal{G}_{\mathcal{C}}$ y se verifica en $\mathcal{G}_{\mathcal{A}}$ si la meta ya está asignada o el MAV ya tiene una meta asignada (ver línea 11 del Algoritmo 5.4). Al realizar esta acción, se están tomando las metas de mayor calificación para cada MAV.

A continuación, se procede a verificar que todos los MAVs tengan una meta asignada. En caso de que no sea así, se les asigna una meta falsa (ver línea 21 del Algoritmo 5.4). Esta meta falsa no contiene posición y su propiedad `asignada` tiene un valor falso. Esta acción permite que el **proceso de coordinación** pueda seguir un flujo similar que el que se tendría si todas las metas fueran asignadas. Sin embargo, el **proceso de exploración** es el encargado de identificar estas metas y realizar las

Algoritmo 5.5 Calificación de metas en el coordinador

Requiere: Lista de MAVs $\mathcal{M}$
Asegura: Los elementos de la lista $\mathcal{M}$ deben contener las metas a calificar $\mathcal{G}_\mathcal{N}$

function CALIFICARMETAS($\mathcal{M}$)
  $\mathcal{G}_\mathcal{C} \leftarrow \emptyset$ ▷ Metas calificadas
  $\mathcal{D}_{\boldsymbol{m\_max}}, \mathcal{D}_{\boldsymbol{g\_max}} \leftarrow CalcularDistanciasMaximas(\mathcal{G}_\mathcal{N}, \mathcal{M})$
  for $i = 0$ **to** $\mathcal{M}.size - 1$ **do**
    $m \leftarrow \mathcal{M}[i]$
    for $j = 0$ **to** $m.\mathcal{G}_\mathcal{N}.size - 1$ **do**
      $g_{\boldsymbol{n}} \leftarrow m.\mathcal{G}_\mathcal{N}[j]$
      $s_{\boldsymbol{g}} \leftarrow d(m.posicion, g_{\boldsymbol{n}}.posicion)/\mathcal{D}_{\boldsymbol{m\_max}}[i]$ ▷ Relativa a otras metas
      $s_{\boldsymbol{m}} \leftarrow d(m.posicion, g_{\boldsymbol{n}}.posicion)/\mathcal{D}_{\boldsymbol{g\_max}}[j]$ ▷ Relativa a otros MAVs
      $s_{\boldsymbol{v}} \leftarrow \mathcal{V}_{\boldsymbol{ext}}.enRegion(m, g_{\boldsymbol{n}})$ ▷ 1, si $g_{\boldsymbol{n}} \in m.region_{\boldsymbol{ext}}$; 0, si no
      $m.\mathcal{G}_\mathcal{N}[j].s_{\boldsymbol{total}} \leftarrow w_{\boldsymbol{g}} s_{\boldsymbol{g}} + w_{\boldsymbol{m}} s_{\boldsymbol{m}} + w_{\boldsymbol{v}} s_{\boldsymbol{v}}$ ▷ $w_{\boldsymbol{g}}, w_{\boldsymbol{m}}, w_{\boldsymbol{v}}$ son predefinidos
  for each $m \in \mathcal{M}$ **do** $\mathcal{G}_\mathcal{C}.add(m.\mathcal{G}_\mathcal{N})$
  return $\mathcal{G}_\mathcal{C}$ ordenado por $s_{\boldsymbol{total}}$ de manera descendente

function CALCULARDISTANCIASMAXIMAS($\mathcal{G}_\mathcal{N}, \mathcal{M}$)
  $\mathcal{D}_{\boldsymbol{m\_max}}[0, 0, 0, ...]$ ▷ Arreglo de dimensión $|\mathcal{M}|$
  $\mathcal{D}_{\boldsymbol{g\_max}}[0, 0, 0, ...]$ ▷ Arreglo de dimensión $|\mathcal{G}|$
  for $i = 0$ **to** $\mathcal{M}.size - 1$ **do**
    for $j = 0$ **to** $\mathcal{G}_\mathcal{N}.size - 1$ **do**
      $distance \leftarrow d(\mathcal{M}[i], \mathcal{G}_\mathcal{N}[j])$
      if $distance > \mathcal{D}_{\boldsymbol{m\_max}}[i]$ **then**
        $\mathcal{D}_{\boldsymbol{m\_max}}[i] \leftarrow distance$
      if $distance > \mathcal{D}_{\boldsymbol{g\_max}}[j]$ **then**
        $\mathcal{D}_{\boldsymbol{g\_max}}[j] \leftarrow distance$
  return $\mathcal{D}_{\boldsymbol{m\_max}}, \mathcal{D}_{\boldsymbol{g\_max}}$

acciones pertinentes.

5.2.4. Actualización del diagrama de Voronoi

Antes de enviar las metas asignadas a los MAVs, el coordinador genera un diagrama de Voronoi $\mathcal{V}$, usándolas como sitios (ver Algoritmo 5.6). Dicho diagrama define las regiones asignadas de los MAVs y se encuentra contenido en una caja envolvente. Durante esta fase de generación de $\mathcal{V}$, también se genera un diagrama de Voronoi extendido $\mathcal{V}_{\boldsymbol{ext}}$, que es de utilidad en la fase de **asignación de metas** (ver Figura 5.7).

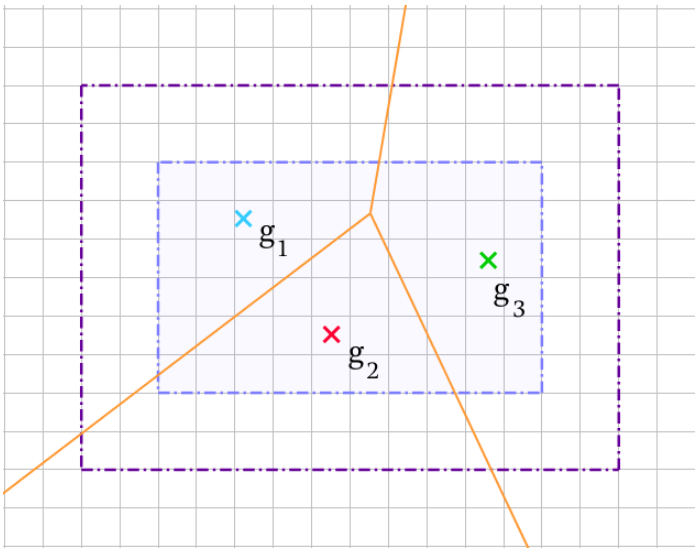


Figura 5.7: Diagrama de Voronoi 2D en cajas contenedoras. La caja interna es utilizada para contener al diagrama de Voronoi $\mathcal{V}$, mientras que la caja externa contiene al diagrama de Voronoi extendido $\mathcal{V}_{\boldsymbol{ext}}$. Las metas asignadas g_i son los sitios de ambos diagramas.

Para contener al diagrama de Voronoi, se utiliza una caja envolvente cuyos límites están dados por las posiciones de las metas que han sido asignadas. La caja envolvente tiene como propósito limitar el tamaño del espacio explorable en el que los MAVs seleccionan sus k metas. Debido a que la posición de las metas es solo un punto en 3D, en las primeras iteraciones dichos puntos no son suficiente para definir una caja del tamaño necesario que le permita a los MAVs explorar. Por esta razón, cada meta asignada se considera como una esfera con radio igual al rango del sensor de los MAVs (ver Figura 5.8). Las dimensiones de la caja son actualizadas con las mínimas y máximas posiciones de dichas metas en los ejes, X, Y y Z (ver línea 16 del Algoritmo 5.6).

La caja contenedora del diagrama de Voronoi extendido $\mathcal{V}_{\boldsymbol{ext}}$ se calcula incrementando el tamaño de cada lado de la caja envolvente de $\mathcal{V}$, en una distancia igual al

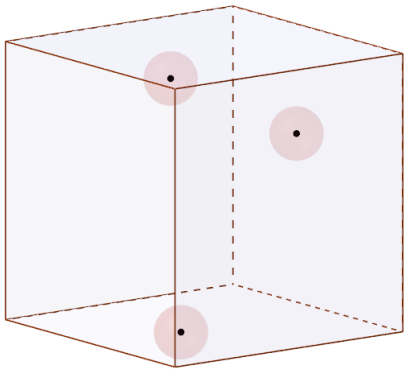


Figura 5.8: Caja envolvente del diagrama de Voronoi. Las esferas representan posiciones de metas que han sido asignadas a los MAVs durante el proceso de exploración. Su radio es igual al rango del sensor de los MAVs.

Algoritmo 5.6 Actualización del diagrama de Voronoi

Requiere: Diagrama de Voronoi $\mathcal{V}$, diagrama de Voronoi extendido $\mathcal{V}_{\boldsymbol{ext}}$, metas asignadas $\mathcal{G}_{\mathcal{M}}$

function ActualizarDiagramaVoronoi($\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}}, \mathcal{G}_{\mathcal{M}}$)
    Remover de $\mathcal{G}_{\mathcal{M}}$ las metas con el campo $asignada = false$
    $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}} \leftarrow ActualizarCajaEnvolvente(\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}}, \mathcal{G}_{\mathcal{M}})$
    $V.Generate$(posiciones de $\mathcal{G}_{\mathcal{M}}$)
    $V_{\boldsymbol{ext}}.Generate$(posiciones de $\mathcal{G}_{\mathcal{M}}$)
    return $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}}$

function ActualizarCajaEnvolvente($\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}}, \mathcal{G}_{\mathcal{M}}$)
    $box \leftarrow \mathcal{V}.boundingBox$
    for each $eje \in Ejes$ **do** ▷ Ejes X, Y, Z
        for each $g_{\boldsymbol{m}} \in \mathcal{G}_{\mathcal{M}}$ **do**
            if $g_{\boldsymbol{m}}.posicion.Coordenada(eje) + r_{\boldsymbol{s}} > box.Max(eje)$ **then**
                $box.Coordenada(eje) \leftarrow g_{\boldsymbol{m}}.posicion.Coordenada(eje) + r_{\boldsymbol{s}}$
            if $g_{\boldsymbol{m}}.posicion.Coordenada(eje) - r_{\boldsymbol{s}} < box.Min(eje)$ **then**
                $box.Min(eje) \leftarrow g_{\boldsymbol{m}}.posicion.Coordenada(eje) - r_{\boldsymbol{s}}$
    $extendedBox \leftarrow box$
    for each $eje \in Ejes$ **do**
        $extendedBox.Max(eje) \leftarrow box.Max(eje) + r_{\boldsymbol{s}}$
        $extendedBox.Min(eje) \leftarrow box.Min(eje) - r_{\boldsymbol{s}}$
    $\mathcal{V}.boundingBox \leftarrow box$
    $\mathcal{V}_{\boldsymbol{ext}}.boundingBox \leftarrow extendedBox$
    return $\mathcal{V}, \mathcal{V}_{\boldsymbol{ext}}$

rango del sensor r_s de los MAVs (ver línea 9 del Algoritmo 5.6). El propósito de extender el diagrama de esta forma es que cada meta pertenezca a la región de un MAV. Esta condición se cumple debido a que los MAVs únicamente exploran metas dentro de su región asignada, por lo tanto la meta más lejana que pueden generar se encuentra a una distancia r_s de dicha región.

5.3. Proceso de exploración

El algoritmo de exploración que utilizan los MAVs está basado en frontera. i.e., se seleccionan puntos entre el espacio conocido y el desconocido. Este espacio en que se desplazan los MAVs es 3D, por lo tanto las metas a las que dichos MAVs navegan también pertenecen a un espacio 3D.

En este trabajo se hace la suposición de que los MAVs cuenta con las capacidades necesarias para extraer fronteras y definir metas de exploración. De manera adicional, se considera que los MAVs son capaces de navegar de manera autónoma hacía dichas metas, i.e., dado un punto en el espacio, los MAVs pueden llegar a él, evadiendo las posibles colisiones con objetos del entorno. Por esta razón, en el sistema propuesto no se indica a los MAVs explícitamente la ruta a seguir para alcanzar su destino.

Cada MAV cuenta con sus propios sensores que le permiten adquirir información del entorno. Los MAVs construyen su mapa interno, usando las observaciones de sus sensores. Estas observaciones también son enviadas al coordinador para que genere un mapa global. Estos mapas no se comparten entre MAVs de forma explícita, ni en forma de mediciones. Por esta razón, cada MAV sólo es "consciente" de la parte del entorno que ha explorado.

En el proceso de exploración, los MAVs, se desplazan y adquieren información un número predefinido k de metas (ver Figura 5.9). Simultáneamente, los MAVs utilizan dicha información para generar nuevas metas. En la Sección 5.3.1 se describe la fase de **actualización y exploración de metas**, mientras que en la Sección 5.3.2 se detalla la fase de **generación de metas**. Cabe destacar que esta fase únicamente se realiza durante la exploración, i.e., durante los momentos en que se produce la mayor adquisición de información. Una vez explorada una meta, los MAVs efectúan la fase de **selección de la siguiente meta**, descrita en la Sección 5.3.3, la cual consiste en identificar otra meta para explorar dentro de las regiones asignadas a los MAVs. Cuando los MAVs no están explorando, se mantienen volando en su última posición a la espera de metas determinadas por el coordinador (ver Figura 5.10).

5.3.1. Actualización y exploración de metas

Como se mencionó en la Sección 5.1, cuando se reciben las metas a explorar de parte del coordinador, los MAVs actualizan su lista local de metas (ver línea 22 del

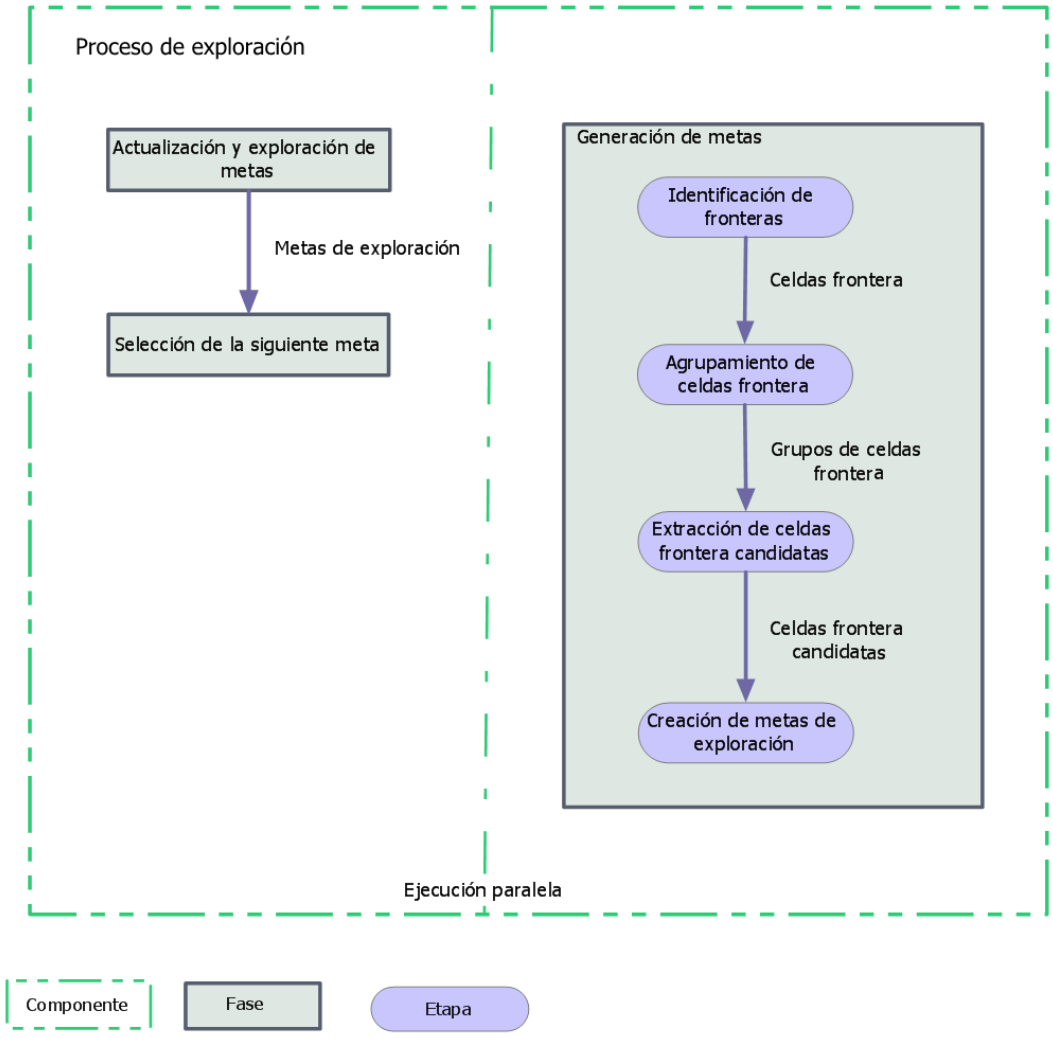


Figura 5.9: Fases del proceso de exploración.

Algoritmo 5.1). Esta actualización es similar a la realizada en el **proceso de coordinación** (ver línea 18 del Algoritmo 5.3). Sin embargo, aquellas metas cuya propiedad `asignada` tiene un valor falso, no son exploradas, ya que fueron creadas como metas falsas (ver Sección 5.2.3). Como consecuencia, dichas metas no se agregan a la lista de metas de los MAVs.

Aunque los MAVs no exploran cuando reciben una meta falsa, su fase de **generación de metas** sí es ejecutada (ver Sección 5.3.2). Esta acción les permite continuar adquiriendo información del entorno. En la práctica, el uso de metas falsas brinda la oportunidad de procesar observaciones faltantes del **paso de exploración** anterior. Dichas observaciones pueden contribuir a crear nuevas metas; en tal caso, estas metas no se exploran, sino que se envían al coordinador para su asignación.

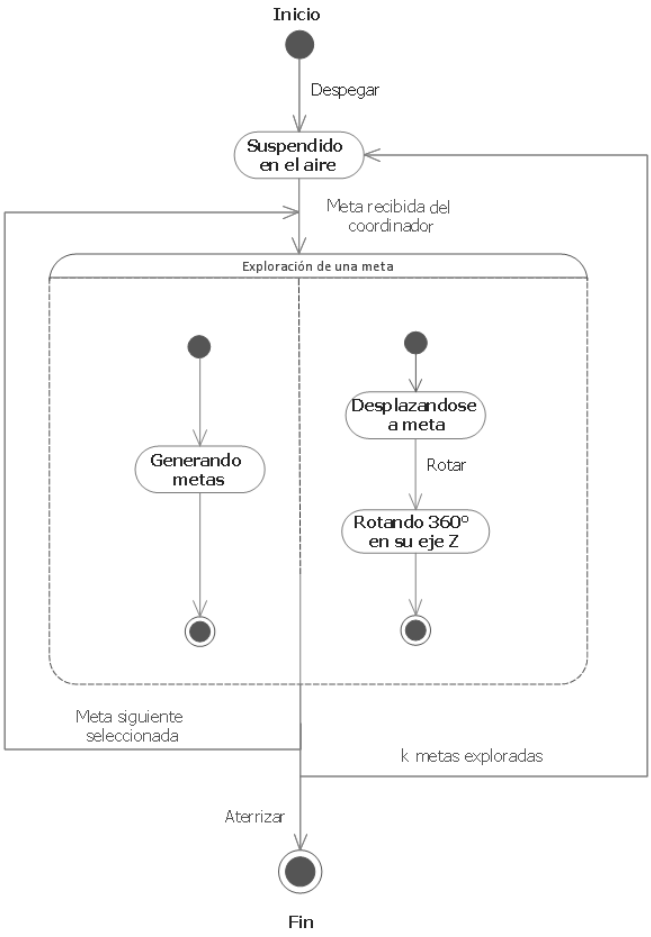


Figura 5.10: Diagrama de estados de los MAVs.

Para las metas, cuya propiedad `asignada` tiene un valor verdadero, se aplica la fase de **exploración de metas** (ver Algoritmo 5.7). Dicha fase consiste en desplazar al MAV m a la posición de la meta $g_{\boldsymbol{m}}$ (ver línea 3 del Algoritmo 5.7). Para producir este desplazamiento, primero se crea un plan de movimiento que genera la ruta a seguir hasta la posición de $g_{\boldsymbol{m}}$. En caso de poder crear dicho plan, se procede a su ejecución, i.e., el coordinador envía los comandos necesarios de vuelo al MAV para alcanzar su meta (ver línea 5 del Algoritmo 5.7). Cuando el MAV alcanza su objetivo, rota 360° con respecto a su eje Z para adquirir más información del entorno. Por último, el MAV cambia el estado de $g_{\boldsymbol{m}}$ a *explorada*. En caso de que no se haya podido crear el plan de movimiento, el estado de la meta pasa a ser *inaccesible*. Finalmente, se actualiza $g_{\boldsymbol{m}}$ en la lista de metas $\mathcal{G}$.

Algoritmo 5.7 Exploración de una meta

Requiere: MAV m, meta a explorar $g_{\boldsymbol{m}}$, lista de metas $\mathcal{G}$ del MAV m

1: **function** ExplorarMeta($m, g_{\boldsymbol{m}}, \mathcal{G}$)
2: $plan \leftarrow$ Generar plan de movimiento de $m.posicion$ a $g_{\boldsymbol{m}}.posicion$
3: En caso de que m esté en el suelo, enviar señal de despegue a m
4: **if** $plan$ **not** NULL **then**
5: Ejecutar plan de movimiento
6: Rotar 360° con respecto al eje Z de m
7: $g_{\boldsymbol{m}}.estado \leftarrow explorada$
8: **else**
9: $g_{\boldsymbol{m}}.estado \leftarrow inaccesible$
10: $g_{\boldsymbol{m}}.cambioEstado \leftarrow$ true
11: Actualizar $g_{\boldsymbol{m}}$ en $\mathcal{G}$
12: **return** $\mathcal{G}$

5.3.2. Generación de metas

Para la generación de nuevas metas, se hace uso del algoritmo de exploración basado en frontera, en particular la versión presentada por Zhu et. al. (ver Sección 4.2). En dicha versión, se extraen fronteras únicamente de las partes actualizadas del mapa, las cuales son determinadas a medida que se adquieren nuevas observaciones del entorno. La fase de generación de metas de exploración se compone de 4 etapas: identificación de fronteras, agrupamiento de celdas frontera, extracción de fronteras candidatas y finalmente, creación de metas de exploración (ver Algoritmo 5.8).

Algoritmo 5.8 Fase de generación de metas, realizada con cada cambio en el mapa

Requiere: Celdas que cambiaron en el mapa $\mathcal{C}_{\mathcal{M}}$

1: **procedure** ProcesarCambioMapa($\mathcal{C}_{\mathcal{M}}$)
2: $\mathcal{C}_{\mathcal{F}} \leftarrow IdentificarFronteras(\mathcal{C}_{\mathcal{M}})$
3: $\mathcal{C}_{\mathcal{G}} \leftarrow AgruparFronteras(\mathcal{C}_{\mathcal{F}})$
4: $\mathcal{C}_{\mathcal{C}} \leftarrow ExtraerCeldasCandidatas(\mathcal{C}_{\mathcal{G}})$
5: $\mathcal{G} \leftarrow CrearMetasExploracion(\mathcal{C}_{\mathcal{C}}, \mathcal{G})$

Identificación de fronteras

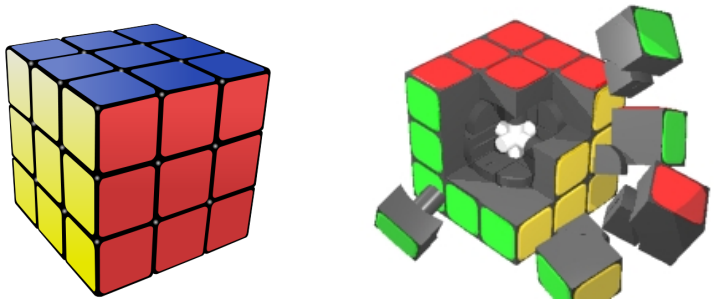
En esta etapa, se identifican fronteras a partir de las celdas que cambiaron en el mapa $\mathcal{C}_{\mathcal{M}}$ (ver Algoritmo 5.9). Para realizar esta identificación, por cada $c_m \in \mathcal{C}_{\mathcal{M}}$ se deben obtener sus celdas vecinas (ver línea 6 del Algoritmo 5.9). Se consideran celdas vecinas a aquellas que tienen al menos un punto en común con c_m, por lo tanto cada c_m tiene

26 celdas vecinas. Un ejemplo de estas celdas es el cubo Rubik. En dicho cubo, el centro puede considerarse como la celda c_m y todos los cubos que la rodean como sus celdas vecinas (ver Figura 5.11).

Algoritmo 5.9 Identificación de celdas frontera

Requiere: Celdas que cambiaron en el mapa $\mathcal{C}_\mathcal{M}$

```
1:  function IdentificarFronteras(C_M )
2:      C_F ← ∅                                              ▷ Celdas frontera
3:      for each c_m ∈ C_M do
4:          if not c_m.occupied then
5:              anyFree, anyUnknown, nextToObstacle ← false
6:              N ← c_m.GetNeighbors()
7:              for each n ∈ N do
8:                  if n.isUnknown() then
9:                      anyUnknown ← true
10:                 else if n.occupied then
11:                     nextToObstacle ← true
12:                 else
13:                     anyFree ← true
14:             if anyFree or anyUnknown and not nextToObstacle then
15:                 C_F.add(c_m)
16:     return C_F
```



(a) (b)

Figura 5.11: Celdas vecinas ejemplificadas en el cubo Rubik: (a) exterior [Commons, 2016b] e (b) interior del cubo [Commons, 2016a].

Las características que una celda debe cumplir para considerarse frontera son: estar libre, tener al menos una celda vecina desconocida, no tener celdas vecinas ocupadas y finalmente, tener al menos una celda vecina libre (ver Figura 5.12 y línea 14 del Algoritmo 5.9).

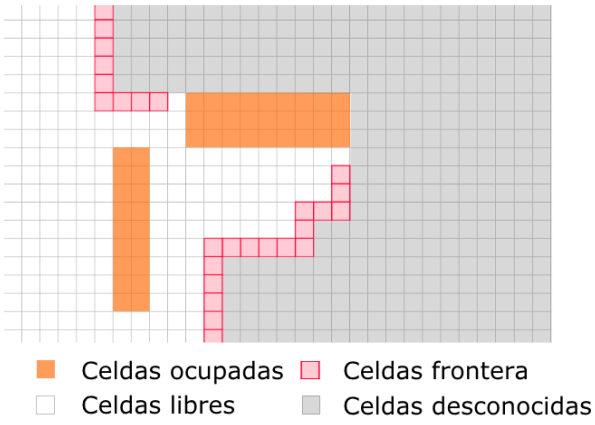


Figura 5.12: Celdas frontera en el mapa.

Agrupamiento de celdas frontera

Una vez obtenido el conjunto de celdas frontera $\mathcal{C}_{\mathcal{F}}$, se crean grupos con aquellas fronteras que son vecinas (ver Algoritmo 5.10). Esta tarea se realiza mientras existan celdas frontera agrupables (ver línea 3 del Algoritmo 5.10). La creación de grupos se hace de manera similar al algoritmo de relleno por inundación (ver Sección 3.3.5), i.e., a partir de una semilla se seleccionan todos los elementos vecinos. La semilla obtenida de $\mathcal{C}_{\mathcal{F}}$ es agregada a una pila $\mathcal{C}_{\mathcal{P}}$ que tiene como propósito almacenar celdas vecinas (ver línea 7 del Algoritmo 5.10). A continuación se extraen celdas $c_p \in \mathcal{C}_{\mathcal{P}}$ de la pila y se obtienen sus celdas vecinas $\mathcal{N}$. En caso de que alguna celda $n \in \mathcal{N}$ también pertenezca a $\mathcal{C}_{\mathcal{F}}$, se elimina de $\mathcal{C}_{\mathcal{F}}$ y se agrega a $\mathcal{C}_{\mathcal{P}}$ y al grupo de la semilla (ver línea 12 del Algoritmo 5.10). Finalmente, los grupos generados se almacenan en una lista $\mathcal{C}_{\mathcal{G}}$ para su procesamiento posterior.

Extracción de celdas frontera candidatas

La extracción de fronteras candidatas $\mathcal{C}_{\mathcal{C}}$ se realiza eligiendo una celda representativa de cada grupo de celdas frontera $\mathcal{C}_{\mathcal{G}}$ (ver Algoritmo 5.11). En este caso se utiliza la celda más cercana al centroide de cada grupo. El centroide es calculado como la media aritmética de las coordenadas x, y y z (ver línea 8 del Algoritmo 5.11), mientras que la cercanía se determina usando la distancia euclidiana.

Creación de metas de exploración

Para crear metas de exploración se utilizan las celdas candidatas (ver Algoritmo 5.12). Este proceso asegura que las metas tienen una distancia mínima r entre sí, que reduce el traslape entre zonas de ganancia de información. Por cada $c_c \in \mathcal{C}_{\mathcal{C}}$ se obtienen las metas de exploración, cuya posición se encuentra en la esfera definida por $c_c.posicion$ y r. Esta búsqueda en la esfera es similar a la realizada durante la actualización de

Algoritmo 5.10 Agrupamiento de celdas frontera

Requiere: Celdas frontera $\mathcal{C}_{\mathcal{F}}$

```
1: function AgruparFronteras(C_F )
2:     C_G ← ∅                                   ▷ Grupos de celdas frontera
3:     while C_F not empty do
4:         grupo ← ∅
5:         C_P ← ∅                  ▷ Pila para almacenar vecinos del mismo grupo
6:         c_f ← C_F.pop()
7:         C_P.push(c_f)
8:         while C_P not empty do
9:             c_p ← C_P.pop()
10:            N ← c_p.GetNeighbors()
11:            for each n ∈ N do
12:                if n ∈ C_F then
13:                    C_F.delete(n)
14:                    grupo.add(n)
15:                    C_P.push(n)
16:        C_G.add(grupo)
17:    return C_G
```

metas en el coordinador (ver línea 12 del Algoritmo 5.3). En caso de no encontrar meta alguna en la esfera de c_c, se crea una nueva meta en la posición de c_c.

5.3.3. Selección de la siguiente meta

Esta fase se realiza cuando un MAV termina de explorar una meta y su número de metas exploradas es menor a un valor predefinido k (ver Algoritmo 5.13). La meta siguiente g_{sig} se selecciona del conjunto de metas $\mathcal{G}$ del MAV m, por lo tanto no se requiere comunicación alguna con el coordinador. Para hacer esta selección, se asigna una calificación a cada meta $g \in \mathcal{G}$, cuyo estado sea nueva y además se encuentre en la región asignada del MAV m. Finalmente, se elige como g_{sig} la meta de mayor calificación.

La ganancia de información de una meta g se mide contando el número de celdas desconocidas que están cercanas a su posición (ver Algoritmo 5.14). Una forma de realizar este conteo explícitamente es utilizando las celdas desconocidas en la esfera definida por la posición de g y el radio r. Sin embargo, dependiendo del valor r, este cálculo puede ser muy costoso. Una forma alternativa de realizar dicho conteo requiere tomar seis puntos en la superficie de la misma. Estos seis puntos se obtienen sumando y restando r a las coordenadas en cada eje de la posición de g, e.g., $p_1(g.posicion.x+r$, $g.posicion.y$, $g.posicion.z)$, $p_2(g.posicion.x - r$, $g.posicion.y$, $g.posicion.z)$ (ver línea

Algoritmo 5.11 Creación de celdas candidatas

Requiere: Grupos de celdas frontera $\mathcal{C}_\mathcal{G}$

function ExtraerCeldasCandidatas($\mathcal{C}_\mathcal{G}$)
    $\mathcal{C}_\mathcal{C} \leftarrow \emptyset$ ▷ Celdas candidatas
    for each $c_g \in \mathcal{C}_\mathcal{G}$ **do**
        $centroide \leftarrow CalcularCentroide(c_g)$
        $c_c \leftarrow CalcularCeldaCercana(c_g, centroide)$
        $\mathcal{C}_\mathcal{C}.add(c_c)$
    return $\mathcal{C}_\mathcal{C}$

function CalcularCentroide(c_g)
    $centroide \leftarrow 0,0,0$ ▷ Punto en 3D
    for each $c \in c_g$ **do**
        $centroide.x \leftarrow centroide.x + c.x$
        $centroide.y \leftarrow centroide.y + c.y$
        $centroide.z \leftarrow centroide.z + c.z$
    $centroide.x \leftarrow centroide.x/|c_g|$
    $centroide.y \leftarrow centroide.y/|c_g|$
    $centroide.z \leftarrow centroide.z/|c_g|$
    return *centroide*

function CalcularCeldaCercana($c_g, centroide$)
    $c_c \leftarrow c_g.first$
    for each $c_g \in \mathcal{C}_\mathcal{G}$ **do**
        if $d(c_g, centroide) < d(c_c, centroide)$ **then**
            $c_c \leftarrow c_g$
    return $c_\mathbf{c}$

Algoritmo 5.12 Creación de metas de exploración

Requiere: MAV m, celdas candidatas $\mathcal{C}_{\mathcal{E}}$, metas $\mathcal{G}$ del MAV m*

```
1: function CrearMetasExploracion(m, C_E, G)
2:     for each c_c ∈ C_E do
3:         G_E ← RadiusSearch(c_c, r)
4:         if G_E = ∅ then
5:             g ← Nueva meta
6:             g.metaID ← lastId
7:             g.posicion ← c_c.posicion
8:             g.mavAsignadoId ← NULL
9:             g.mavOrigenId ← m.ID
10:            g.estado ← NUEVA
11:            g.cambioEstado ← false
12:            G.add(g)
13:            lastId ← lastId + 1
14:    return G
```

Algoritmo 5.13 Selección de meta siguiente

Requiere: MAV m, metas $\mathcal{G}$ del MAV m

```
1: function SeleccionarMetaSiguiente(m, G)
2:     distanciaMaxima ← ObtenerDistanciaMaxima(m, G)
3:     g_sig ← NULL
4:     s_max ← 0
5:     for each g ∈ G do
6:         if g.estado = NUEVA and V.enRegion(m, g) then
7:             totalCeldas, celdasDesconocidas ← ContarCeldas(g, r)
8:             p_c ← celdasDesconocidas/totalCeldas
9:             p_d ← 1 − (d(g.posicion, m.posicion)/distanciaMaxima)
10:            s ← w_c p_c + w_d p_d          ▷ w_c y w_d son valores predefinidos
11:            if s > s_max then
12:                s_max ← s
13:                g_sig ← g
14:    return g_sig

15: function ObtenerDistanciaMaxima(m, G)
16:    distanciaMaxima ← 0
17:    for each g ∈ G do
18:        if d(g.posicion, m.posicion) > maxDistance then
19:            maxDistance ← d(g.posicion, m.posicion)
20:    return distanciaMaxima
```

5 del Algoritmo 5.14).

Para contar las celdas desconocidas, se utiliza un rayo trazado desde el centro de la esfera a cada uno de los seis puntos (ver línea 9 del Algoritmo 5.14). Por cada celda c en el conjunto de celdas intersectadas $\mathcal{C}$ se verifica si es desconocida y, en caso de serlo, se agrega a la cuenta de celdas desconocidas (ver línea 14 del Algoritmo 5.14). Durante este proceso, también se mantiene la cuenta del número total de celdas, que es de utilidad para normalizar.

Para calificar a una meta g, se utilizan dos puntuaciones: p_c y p_d. La primera, p_c, es el resultado de dividir el número de celdas desconocidas entre el total de celdas obtenidas durante el conteo. El segundo valor, p_d, se obtiene dividiendo la distancia del MAV m a la meta g entre la distancia a la meta más lejana con respecto a m (ver línea 15 del Algoritmo 5.14). Finalmente, la calificación de g es el resultado de la suma ponderada de p_c y p_d (ver línea 10 del Algoritmo 5.14).

Algoritmo 5.14 Conteo de celdas en la posición de una meta

Requiere: Meta g y radio de ganancia de información r

```
1: function ContarCeldas(g, r)                          ▷ Celdas totales y desconocidas
2:     n_total, n_desc ← 0
3:     for each eje in Ejes do      ▷ X,Y,Z; por cada eje, calcular con r positiva y r negativa
4:         p_destino ← g.posicion
5:         p_destino.Coordenada(eje) ← g.posicion.Coordenada(eje) + r
6:         n_subT, n_subD ← ContarCeldasDesconocidas(g.posicion, p_destino)
7:         Agregar n_subT, n_subD a n_total y n_desc respectivamente
8:     return n_total, n_desc

9: function ContarCeldasDesconocidas(puntoOrigen, puntoDestino)
10:    numDesconocidas ← 0
11:    C ← trazar rayo de puntoOrigen hacia puntoDestino
12:    for each c ∈ C do                          ▷ Celdas intersectadas por el rayo
13:        if c.isUnknown() then
14:            numDesconocidas ← numDesconocidas + 1
15:    return |C|, numDesconocidas
```

5.4. Limpieza del mapa

Para eliminar los elementos ocupados del mapa, que se producen cuando un MAV pasa frente a otro, se lleva a cabo un proceso de limpieza al final de cada **paso de exploración**. Dado que los MAVs no conocen las posiciones ni la información del resto de los miembros del equipo durante la exploración, es tarea del coordinador seleccionar los puntos a limpiar. El proceso de limpieza consiste en marcar, como libres, los vóxeles alrededor de los puntos indicados (ver Algoritmo 5.15).

Las posiciones a limpiar $\mathcal{G}_{\mathcal{L}}$, seleccionadas por el coordinador, son las posiciones de las metas exploradas. Dichas metas son posiciones en las cuáles estuvieron los MAVs, por lo tanto tienen riesgo de mapeo mutuo. Cabe destacar que no basta con limpiar estas posiciones una sola vez, debido a los diferentes planes de movimiento que pueden ser generados por los MAVs. Estos planes hacen que, para llegar de un punto a otro, un MAV pueda pasar cerca de una meta ya explorada. Por esta razón, al final de cada paso de exploración, se limpian nuevamente todas las metas exploradas.

A partir de $\mathcal{G}_{\mathcal{L}}$ se utiliza un algoritmo de relleno por inundación basado en pila (ver Sección 3.3.5) para limpiar las celdas ruido (ver Algoritmo 5.15). Debido a que el problema de mapeo mutuo no ocurre con exactitud en la celda ubicada en la posición de $g_{\boldsymbol{l}} \in \mathcal{G}_{\mathcal{L}}$, se extiende el rango de búsqueda. La forma de extenderlo es utilizando las celdas ocupadas $\mathcal{C}_{\mathcal{O}}$ en la caja con centro p y lado $2l$ (ver línea 17 del Algoritmo 5.15).

El algoritmo de relleno por inundación toma como semilla una celda $c_{\boldsymbol{o}} \in \mathcal{C}_{\mathcal{O}}$. A continuación, por cada celda vecina $n \in \mathcal{N}$, se verifica si está ocupada y, en caso de estarlo, se limpia (ver línea 10 del Algoritmo 5.15). Finalmente, se asigna un identificador a n para determinar de qué MAV provino y se agrega tanto a $\mathcal{C}_{\mathcal{O}}$ como a un conjunto de celdas ruido $\mathcal{C}_{\mathcal{R}}$. El proceso continua mientras haya celdas vecinas ocupadas y no se alcance un límite de celdas ruido. Este límite tiene como propósito evitar que se eliminen por completo todas las celdas del mapa, en caso de que una meta se encuentre cercana a una pared.

5.5. Término de la exploración

El encargado de determinar cuándo termina la exploración es el coordinador, con base en las respuestas de los MAVs. Esta situación ocurre cuando dichas respuestas son vacías. Como se mencionó anteriormente, las respuestas de los MAVs consisten en el conjunto de metas que generaron y actualizaron. Por esta razón, las respuestas vacías significan que no se adquirió información suficiente para crear nuevas metas y que los MAVs no recibieron meta alguna para explorar por parte del coordinador. Esta

Algoritmo 5.15 Limpieza del mapa

Requiere: Metas a limpiar $\mathcal{G}_{\mathcal{L}}$, mapa de ocupación *map*

```
1: function LimpiarMapa(G_L, map )
2:     R ← ∅                                              ▷ Celdas ruido
3:     for each g_l ∈ G_L do
4:         C_O ← boxSearch(g_l.posicion, l, map)          ▷ l es un valor predefinido
5:         C_R ← ∅                                        ▷ Celdas ruido del punto p
6:         while C_O not empty and |C_R| < maxCells do
7:             c_o ← C_O.pop()
8:             N ← c_o.GetNeighbors()
9:             for each n ∈ N do
10:                if n.occupied then
11:                    n.occupied ← false
12:                    n.ID ← g_l.mavAsignadoID
13:                    C_R.add(n)
14:                    C_O.push(n)
15:        R.add(C_R)
16:    return R

17: function BoxSearch(p, l, map)
18:    cellSize ← map.resolution
19:    C_O ← ∅
20:    for x = p.x − l to p.x + l do
21:        for y = p.y − l to p.y + l do
22:            for x = p.z − l to p.z + l do
23:                c ← map.getCellAtPoint(x, y, z)
24:                if c.isOccupied() then
25:                    C_O.push(c)
26:                z ← z + cellSize
27:            y ← y + cellSize
28:        x ← x + cellSize
29:    return C_O
```

última afirmación es correcta, ya que en caso de que los MAVs hubieran recibido metas a explorar, las respuestas contendrían al menos una meta con cambio de estado, *explorada* o *inaccesible* (ver Sección 5.3.1).

Como se puede ver en el párrafo anterior, el fin de la exploración está relacionado con el hecho de que el coordinador no tenga metas disponibles para asignar. Este hecho se usa con frecuencia en los trabajos relacionados como condición de término de la exploración (ver Secciones 4.1 y 4.2). Sin embargo, los MAVs pueden llegar a generar nuevas metas en su última posición, debido a posibles cambios sin procesar en el mapa (ver Sección 5.3.1). Por esta razón, aunque no queden metas disponibles para asignar, se envían metas falsas a los MAVs.

A lo largo del **proceso de coordinación** es común el envió de metas falsas, e.g., cuando hay menos metas disponibles que MAVs. Durante el proceso de exploración puede ocurrir múltiples veces que todos los MAVs reciban metas falsas. Sin embargo, después de cada ocasión en que se realice esta acción, los MAVs deberán generar nuevas metas ya que, de no ser así, la exploración del entorno terminará. Esta condición permite que la exploración pueda recuperarse de situaciones en las que no se detectaron correctamente los cambios en el mapa al final de la exploración de una meta. Esta situación puede ocurrir debido a factores tales como: velocidad del MAV al rotar, estabilidad y posición.

Por otra parte, la exploración termina debido a que se realiza en un entorno cerrado. Puesto que las metas son consideradas como esferas con una separación mínima entre sí (ver Sección 5.3.2) después de cierto tiempo no habrá más espacio para seguir agregándolas. Dado que el número de metas es limitado, basta con que los MAVs continúen explorándolas para que llegue el momento en que no haya más metas disponibles para asignar por parte del coordinador y la exploración termine.

5.6. Discusión

En esta sección se discute la estrategia de exploración propuesta. En la Sección 5.6.1 se analiza la forma en cómo se coordina la exploración multi-MAV, mediante las regiones del diagrama de Voronoi, y en la Sección 5.6.2 se presenta un breve análisis de la ganancia de información proporcionada por las metas de exploración.

5.6.1. Análisis de la exploración multi-MAV utilizando regiones del diagrama de Voronoi

Mediante la exploración de un número limitado de metas, en las regiones asignadas a los MAVs, se favorece la cooperación entre ellos. En caso de que se intentaran explorar las regiones completamente existirían dos problemas. El primero de ellos es

cómo determinar cuando se ha explorado por completo el diagrama de Voronoi. Este problema es ocasionado por los lugares inaccesibles dentro de las regiones. El segundo problema se debe a los tamaños diferentes de las regiones, ya que aún en el supuesto de que se puede explorar una región por completo, las regiones más pequeñas se terminarían más rápido. Esta situación dejaría inactivos a los MAVs asignados a dichas regiones, hasta que los MAVs con regiones más grandes terminaran de explorar las suyas. En ese momento, el coordinador les enviaría una nueva meta, pero se habrían desaprovechado sus.

Finalmente, debido a que en cada **paso de exploración** los MAVs crean y exploran metas en lugares que extienden el mapa del entorno, de acuerdo a la información disponible, se puede considerar que la estrategia de exploración propuesta se comporta como un algoritmo voraz. Puesto que el agrupamiento de fronteras no mantiene completamente la ganancia de información de las celdas agrupadas, esta estrategia no proporciona la mejor cobertura posible del mapa. A cambio de la pérdida de cobertura, las metas reducen la distancia que necesitan recorrer los MAVs para explorar el entorno, en comparación con la exploración de todas las celdas frontera.

5.6.2. Análisis de la ganancia de información de las metas de exploración

Las ubicaciones de las meta, a pesar de no ser las mejores, obtienen una buena ganancia de información con respecto al costo de explorarlas. Esta situación se debe a dos aspectos principales, el uso de celdas frontera y su agrupación en metas de exploración.

En primer lugar, las fronteras proporcionan la mayor ganancia de información en el mapa, i.e., celdas desconocidas. Esta afirmación se basa en el hecho de que para cada celda libre c_l del mapa, que no es frontera, es posible encontrar una celda c_f que si lo es, la cual proporciona una mayor ganancia de información que c_l. Se utilizan las celdas libres, debido a que son los lugares donde los MAVs pueden colocarse sin chocar con obstáculos. En caso de que existiera alguna celda c_l con mayor ganancia de información que una celda frontera c_f, una parte de su zona de ganancia de información tendría que estar dentro de las celdas libres u ocupadas y otra parte dentro de las celdas desconocidas. A medida que la posición de c_l se acerca a las celdas desconocidas, la zona de ganancia de información de c_l crece. Si se elige c_l en la posición más cercana a las celdas desconocidas, esta será una celda libre que tenga como vecinas a las celdas desconocidas, por lo tanto c_l será una celda frontera c_f.

En segundo lugar, al utilizar el centroide para agrupar las fronteras conectadas, también se agrupa la cantidad de información que proporcionan. Sin embargo, ya que los grupos de fronteras dependen de las observaciones adquiridas, los centroides de dichos grupos pueden estar muy cercanos entre sí. Por esta razón, se utiliza la

distancia mínima entre metas, la cual reduce el traslape entre zonas de ganancia de información. Aunque esta distancia proporciona ventajas en la reducción de dichos traslapes, también puede disminuir el nivel de detalle del mapa, en comparación con la exploración de todas las fronteras.

Capítulo 6

Implementación

En esta Sección se presenta la implementación del sistema de exploración multi-MAV. En la Sección 6.1, se describe la arquitectura del sistema implementada en ROS. Esta arquitectura incluye los componentes del sistema y la interacción entre nodos de ROS. En la Sección 6.2, se describe la configuración realizada del entorno de simulación para permitir el uso de múltiples MAVs simulados. En la Sección 6.3 se presenta la configuración del entorno de planificación de movimiento y ejecución de trayectorias. Para esta tarea se utilizó la herramienta MoveIt!, sin embargo fueron necesarias adecuaciones para permitir su funcionamiento con múltiples MAVs. En la Sección 6.4, se describen los componentes relacionados con el **proceso de coordinación** y se describen los mensajes creados para realizar el intercambio de información entre los procesos de coordinación y exploración. En la Sección 6.5, se describen los componentes implementados para realizar el **proceso de exploración** y la visualización de las **metas de exploración**. En la Sección 6.6 se presenta el funcionamiento del componente de limpieza, así como las acciones realizadas para incluirlo en los elementos de generación de mapas. Finalmente, en la Sección 6.7 se discute la implementación del sistema en ROS haciendo énfasis en los componentes utilizados.

6.1. Arquitectura del sistema en ROS

En esta sección, se describe la arquitectura implementada en ROS del sistema propuesto. En la Sección 6.1.1, se presenta un panorama general de la implementación y las herramientas utilizadas. En la Sección 6.1.2, se describen los componentes del sistema, de forma modular a nivel de paquetes de ROS. En la Sección 6.1.3, se presentan los nodos implementados en ROS y la forma en que estos interactúan entre sí.

6.1.1. Herramientas utilizadas

El sistema de exploración multi-MAV se implementó en la plataforma de desarrollo robótico, *Robot Operating System* (ROS) en su distribución Indigo (ver Apéndice

A.1). Los MAVs fueron simulados en Gazebo (ver Apéndice A.3), utilizando como base el proyecto `tum_simulator` que proporciona un modelo del MAV Parrot AR.Drone 2.0. A los MAVs se les montaron sensores RGB-D Microsoft Kinect simulados. Estos sensores proveen una nube de puntos que es convertida por la biblioteca Octomap [Hornung et al., 2013] en un mapa de ocupación 3D. Los MAVs son coordinados por un coordinador que interactua con ellos a través del envío y recepción de metas de exploración. Para solventar el problema de mapeo mutuo se limpian puntos de los mapas del coordinador y de los MAVs. Dichos puntos son proporcionados por el coordinador (ver Figura 6.1).

En la simulación se utilizan transformaciones absolutas, con respecto a un marco de referencia global, tanto para las nubes de puntos, como para las posiciones de los MAVs. Debido a esta situación, no se hace uso de herramientas SLAM para proveer mapeo y localización en el entorno. Otra consecuencia del marco de referencia global, es que tampoco es necesario realizar procesos de fusión de nubes de puntos para hallar la relación entre los marcos de referencia locales de los MAVs.

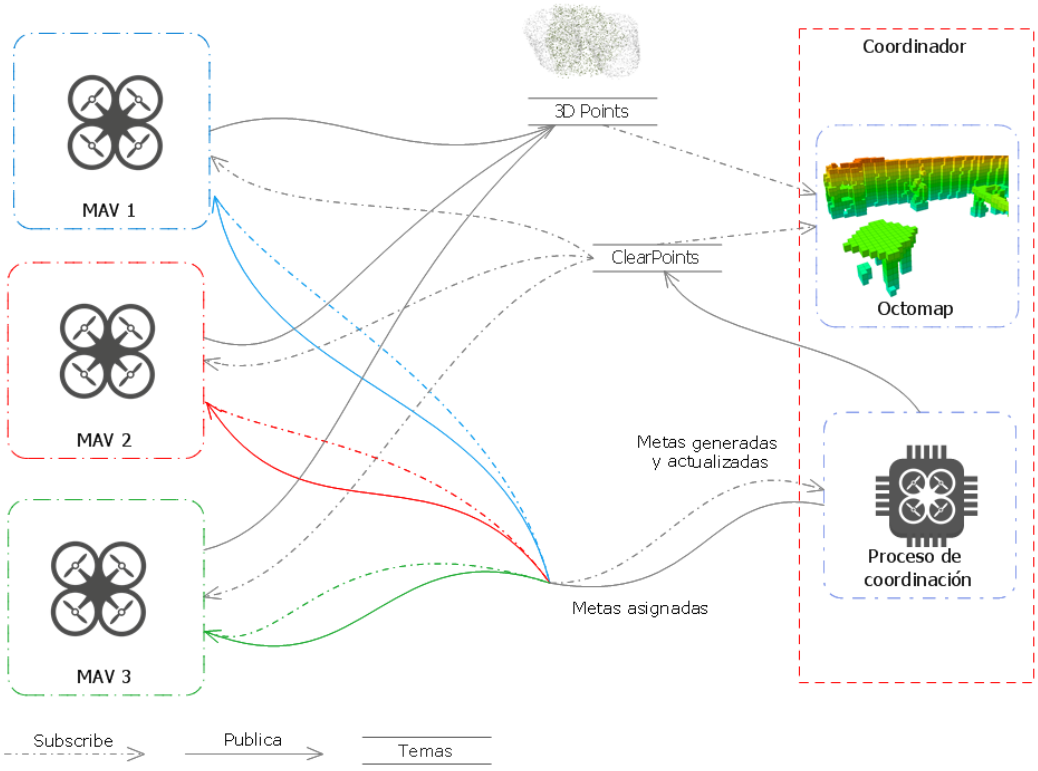


Figura 6.1: Arquitectura del sistema de exploración propuesto, implementado en ROS.

Los MAVs simulados deben su capacidad de navegación autónoma al *plugin* MoveIt! para ROS. Este *plugin* incorpora funciones de planificación de movimiento. Por

cada MAV, se define un **grupo de movimiento** (*move group*). Este grupo es el encargado de generar trayectorias libres de obstáculos en el entorno. Para realizar esta tarea, cada grupo de movimiento construye un mapa de ocupación local, mediante Octomap, utilizando la nube de puntos provista por el sensor Kinect. Dicho mapa únicamente es modificado por el **proceso de limpieza** y no se comparte con el coordinador.

Un elemento que es parte de la generación de metas es la identificación de celdas frontera. Esta tarea de identificación se realiza utilizando una versión modificada de la herramienta 3D-FBET [Zhu et al., 2015]. Para lograr la extracción de fronteras, se mantiene un mapa local en los MAVs. El mapa de cada MAV es creado a partir de la nube de puntos 3D del sensor Kinect correspondiente. Este mapa, aunque en estructura es simular al utilizado por MoveIt! para realizar la planificación de movimiento, no es el mismo.

El diagrama de Voronoi, que representa las **regiones asignadas** a los MAVs, se crea en el coordinador con la biblioteca Voro++. Este diagrama no es enviado de forma explicita a los MAVs, los cuales únicamente reciben los sitios del diagrama y lo reconstruyen, utilizando la misma biblioteca.

Finalmente, como elemento de supervisión del sistema de exploración, el coordinador y los MAVs crean **marcadores de visualización** de la herramienta de visualización 3D RViz (RViz *markers*). Estos marcadores representan las metas de exploración, los diagramas de Voronoi y las **celdas ruido**. Dichos marcadores pueden ser observados en RViz (ver Apéndice A.2). Esta herramienta también se utiliza para visualizar los mapas generados por Octomap.

6.1.2. Componentes

La implementación del sistema fue realizada en el lenguaje C++ a través de dos paquetes para ROS (ver Figura 6.2). El primer paquete, `mv_explorer`, se encarga de las funciones de exploración y coordinación que corresponden a los procesos de exploración y coordinación, respectivamente (ver Secciones 5.3 y 5.2). El segundo paquete, `mv_octomap`, construye un mapa de ocupación y además proporciona funciones de limpieza para mapas creados con Octomap.

Los componentes del paquete `mv_explorer` relacionados con los MAVs son: `MAVExplorer`, `GoalGenerator`, `MAVGoalContainer`, `MotionExecutor` y `GoalAndMapMarkerPublisher`. `MAVExplorer` controla el flujo de la exploración en los MAVs, i.e., recepción y exploración de metas recibidas del coordinador. `GoalGenerator` realiza la identificación y el agrupamiento de celdas frontera (ver Sección 5.3.2) en un mapa local creado con Octomap. Por otro lado, los centroides de los grupos de celdas frontera

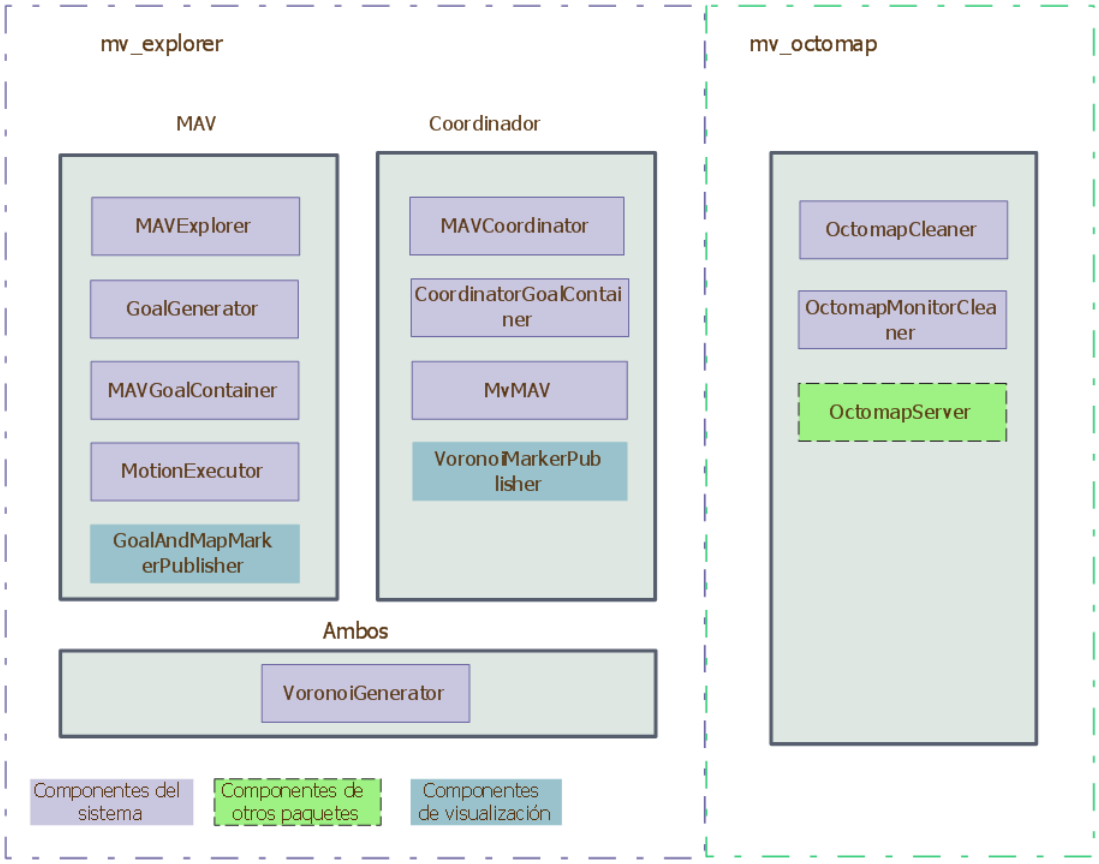


Figura 6.2: Componentes de los paquetes de ROS desarrollados.

son posibles metas de exploración. Por esta razón se agregan al contenedor de metas `MAVGoalContainer`, el cual garantiza una distancia mínima entre ellas (ver Sección 5.3.2). Este contenedor, además de almacenar metas, también es responsable de obtener la meta siguiente, dada la posición del MAV (ver Sección 5.3.3). El componente `MotionExecutor`, es el encargado de generar y ejecutar los planes de movimiento que permiten desplazarse a una meta, mediante la API del grupo de movimiento de MoveIt!. Por último, el componente `GoalAndMapMarkerPublisher` crea marcadores de visualización, utilizando las posiciones de las metas y las celdas del mapa local.

Los componentes del paquete `mv_explorer`, que se ejecutan en el coordinador, son: `MAVCoordinator`, `CoordinatorGoalContainer`, `MvMAV` y `VoronoiMarkerPublisher`. `MAVCoordinator` se encarga de coordinar la exploración de los MAVs, apoyándose del resto de los componentes (ver Sección 5.2). El componente `CoordinatorGoalContainer` tiene como funciones principales el almacenamiento y la actualización de estado de las metas del coordinador (ver Sección 5.2.2). Por otro lado, `MvMAV` almacena la información referente a los MAVs del equipo de exploración. Este componente sirve como elemento de abstracción para que el proceso de coordinación se enfoque en la asignación de metas a los MAVs, mas no en la forma en como serán enviadas. Por

último, el componente `VoronoiMarkerPublisher` tiene como función la creación de marcadores de visualización que representan las aristas y sitios del diagrama de Voronoi, así como su caja envolvente.

El componente `VoronoiGenerator` es utilizado por el coordinador y los MAVs. Este componente actúa como interfaz con la biblioteca Voro++ y proporciona funciones que simplifican la consulta del diagrama, e.g., verificar si una celda está en la región de un MAV.

El paquete `mv_octomap` utiliza la biblioteca Octomap para convertir las nubes de puntos en celdas de un mapa de ocupación. En particular, el componente `OctomapServer` utilizado pertenece al paquete `octomap_server`, el cual implementa funciones de la biblioteca Octomap, mediante elementos de ROS (nodos y temas). El componente `OctomapCleaner` contiene las funciones para la limpieza de las celdas ruido, presentes en el mapa (ver Sección 5.4). Por último, `OctomapMonitorCleaner` sirve para realizar el proceso de limpieza en el mapa del grupo de movimiento de MoveIt!, particularmente en el componente de percepción `occupancy_map_monitor`.

En esta sección se describieron los paquetes creados, aunque existen otros paquetes que también son fundamentales para el funcionamiento del sistema. Los paquetes `cvg_sim_gazebo`, `cvg_sim_gazebo_plugins`, `cvg_sim_msgs` y `message_to_tf` son utilizados para la simulación de los MAVs (ver Sección 6.2). Los paquetes `ardrone2_moveit_config`, `moveit_ros/perception` y `moveit_simple_controller_manager` sirven para las funciones de planificación de movimiento (ver Sección 6.3). En este sentido, el paquete que permite la ejecución de dicho movimiento es `multidof_action_controller`. Por último, el paquete `robot_pose_publisher` proporciona la posición de los MAVs con colores. Algunos de estos paquetes fueron modificados con respecto a sus versiones originales.

6.1.3. Interacción entre nodos

Para una mejor comprensión del funcionamiento de una implementación en ROS, es importante conocer los nodos y temas que utiliza, así como la forma en que tales elementos interactuan entre sí. Dado que todos los MAVs poseen los mismos nombres de nodos y temas, es necesario un mecanismo para diferenciarlos, en este caso se utilizan espacios de nombres (*namespaces*) de ROS. En este sistema, la nomenclatura utilizada para dichos espacios es las siglas `ard`[1] seguidas de un número entero que sirve para identificar al MAV, e.g., `ard1` y `ard2`.

Los paquetes desarrollados crean tres nodos principales `/mv_explorer_ard`, `/mv_coordinator` y `/mv_octomap`. Sin embargo, existen otros nodos indispensables para

[1]El nombre de espacio de nombres `ard`, es por las siglas del AR.Drone 2.0 simulado

la exploración, los cuales son provistos por paquetes de ROS. Estos últimos nodos son responsables de los MAVs simulados, la planificación de movimiento y la ejecución de trayectorias (ver Figura 6.3). Cabe destacar que en el nodo `/mv_explorer_ard`, el prefijo `ard` no es un espacio de nombres, sino que es parte del nombre del nodo. Esta situación se debe a limitaciones técnicas en la resolución de espacios de nombres por parte de la API del grupo de movimiento de MoveIt!. Este grupo es la clase `MoveGroup` de MoveIt! encargada de proporcionar funciones de planificación de movimiento y ejecución de trayectorias.

Los nodos se comunican entre sí enviando mensajes a través de temas en dos modelos: el modelo cliente-servidor y el modelo publicación-subscripción, siendo este último el más utilizado. Los nodos pueden utilizar ambos modelos de comunicación al mismo tiempo pero, en general, entre cada par de nodos se utiliza solamente un modelo de comunicación a la vez.

En la implementación del sistema, el modelo cliente-servidor se aplica mediante los **servidores de acciones** (*Action Servers*) para acciones que toman determinado tiempo en ejecutarse en los nodos. Este modelo es utilizado en la comunicación entre los nodos `/mv_explorer_ard` (proceso de exploración) y `/mv_explorer/coordinator` (proceso de coordinación). En este caso, los MAVs actúan como servidores puesto que ofrecen el servicio de exploración de metas y proporcionan como resultado un conjunto de metas generadas y exploradas. El segundo caso donde se aplica el modelo cliente-servidor es entre los nodos `/mv_explorer_ard`, `/ard/move_group` y `/ard/multi_dof_joint_trajectory_action`. Entre los primeros dos nodos es utilizado para realizar la planificación de movimiento y entre los últimos dos nodos para la ejecución de trayectorias. En este caso los nodos `/ard/move_group` y `/ard/multi_dof_joint_trajectory_action` actúan como servidores al ofrecer los servicios de planificación y ejecución de trayectorias.

El modelo de publicación-subscripción es empleado por todos los nodos del sistema. Este modelo se aplica mediante la publicación de información por parte de los nodos en temas de ROS. El ejemplo más representativo del uso de este modelo de comunicación en el sistema desarrollado son los temas de nubes de puntos. Dichos temas se pueden ver en la comunicación entre los nodos `/gazebo`, `/mv_explorer_ard`, `/ard/move_group`, `/ard/relay` y `/mv_octomap`. Las nubes de puntos tienen como origen al nodo `/gazebo` porque en dicho nodo se simulan los MAVs. Estas nubes son utilizadas por el resto de los nodos para construir mapas de ocupación, con excepción del nodo `/ard/relay`, que se encarga de concentrar las nubes de todos los MAVs hacia un tema en común para la creación del mapa global en el nodo `/mv_octomap`.

Por último, los nodos `/ard/robot_state_publisher`, `/ard/ground_truth_to_tf` y `/ard/robot_pose_publisher` están encargados de publicar información del estado y la posición del MAV. En primer nodo, `/ard/robot_state_publisher`, publica trans-

formaciones de las posiciones de las articulaciones de los MAVs. Aunque los MAVs únicamente poseen una unión virtual configurada por MoveIt!, se conserva el nodo para posibles usos futuros. El segundo nodo, `/ard/ground_truth_to_tf`, convierte los mensajes con las posiciones de los MAVs en transformaciones, las cuales permiten el uso de un marco de referencia común. El último nodo, `/ard/robot_pose_publisher`,es utilizado para publicar las posiciones junto con marcadores de visualización de RViz, que ayudan a asignar colores a los MAVs para una mejor interpretación del proceso de exploración.

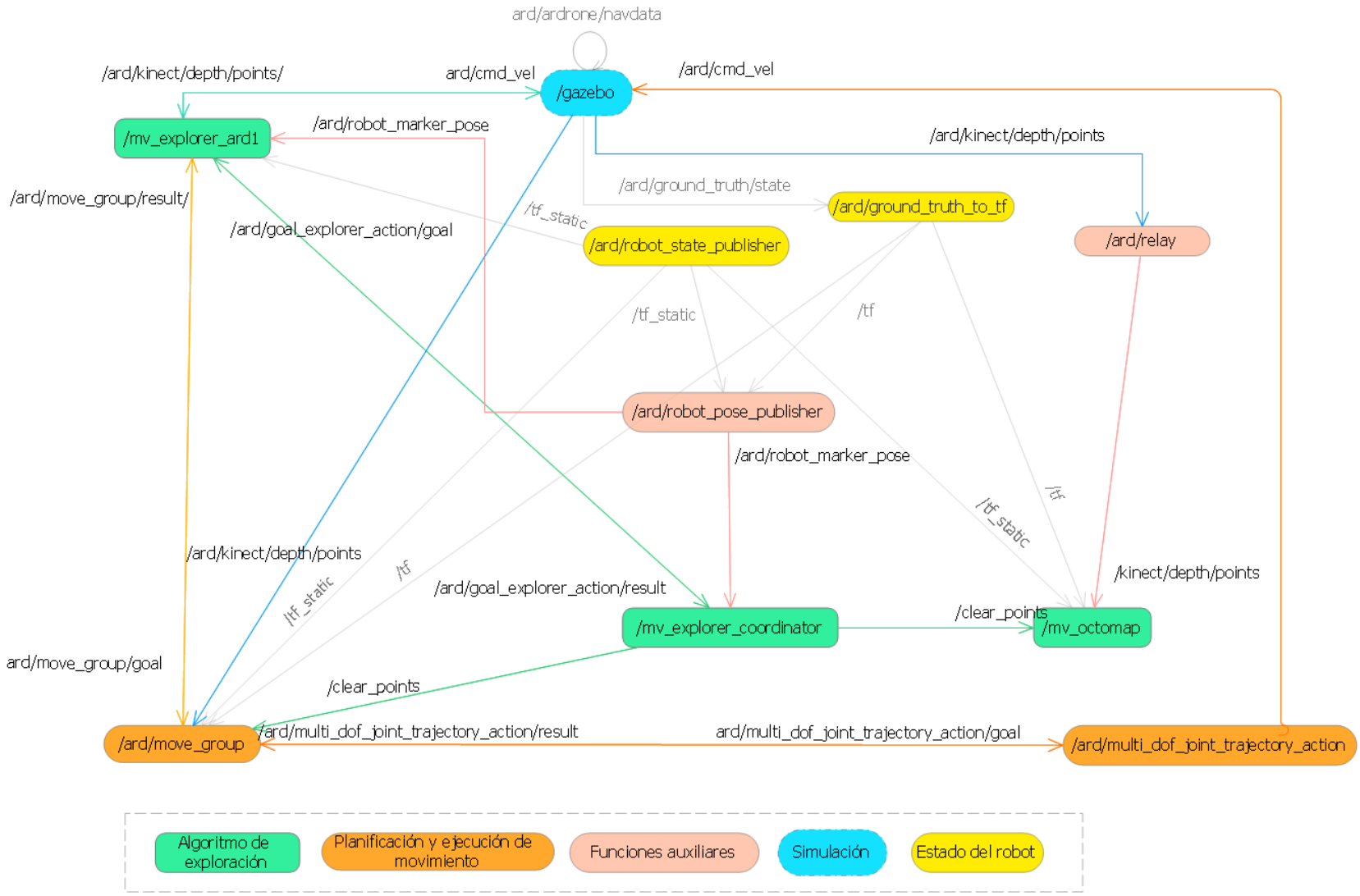


Figura 6.3: Interacción entre nodos de ROS, las aristas son los temas principales mediante los cuales se comunican.

6.2. Configuración de la simulación

En esta sección, se describen las acciones realizadas para configurar la simulación. Esta simulación en Gazebo es iniciada mediante un conjunto de archivos de lanzamiento, los cuáles se describen en la Sección 6.2.1. En Gazebo se coloca un conjunto de MAVs simulados, los cuales tienen como base un modelo comercial. Los elementos necesarios para utilizar estos MAVs se presentan en la Sección 6.2.2. En la Sección 6.2.3, se describe la forma en cómo los MAVs obtienen las observaciones del entorno simulado. Finalmente, en la Sección 6.2.4, se describe el nodo encargado de publicar las posiciones a color de los MAVs.

6.2.1. Archivos de lanzamiento de la simulación

La simulación es iniciada con la ejecución de un conjunto de nodos a través de archivos de lanzamiento. Estos archivos permiten lanzar múltiples nodos de manera sencilla y brindan mecanismos de configuración de parámetros. Los archivos de lanzamiento creados para la simulación son: `start_simulator.launch`, `world_map.launch`, `multi_mav.launch` y `one_mav.launch` (ver Figura 6.4).

El archivo `start_simulator.launch` es responsable de las llamadas a los archivos de lanzamiento `empty_world.launch`, `world_map.launch` y `multi_mav.launch`. El archivo `empty_world.launch` pertenece a Gazebo y es el encargado de iniciar el entorno de simulación. Por su parte, `world_map.launch` inicia el nodo `mv_octomap` que contiene la implementación de `OctomapServer` con funciones de limpieza. El archivo `multi_mav.launch` configura y lanza los archivos `one_mav.launch`. Entre las configuraciones se encuentran: la posición inicial de los MAVs, el color asignado a sus marcadores de posición y su espacio de nombres.

El archivo `one_mav.launch` ejecuta los nodos de la simulación relacionados con el MAV. Los nodos son: `spawn_robot`, `robot_pose_publisher`, `robot_state_publisher`, `relay`, `ground_truth_to_tf` y `multi_dof_joint_trajectory_action`. De estos nodos, `spwan_robot`, del tipo `spwawn_model`, es el que carga el modelo de los MAVs y lo coloca en el mundo simulado. Del resto de los nodos se hablará con más detalle en la Sección 6.5.

6.2.2. Simulación de los MAVs

Los MAVs, que son utilizados en la simulación del presente trabajo, pertenecen al paquete de ROS `tum_simulator` [Huang and Sturm, 2016]. Este paquete fue diseñado para simular un solo MAV AR.Drone 2.0 y permitir su control mediante un *joystick* (ver Figura 6.5). Un dato extra acerca del paquete `tum_simulator` es que, en realidad, es un metapaquete. Los metapaquetes se pueden considerar como paquetes virtuales, que únicamente referencian a otros paquetes. En el caso de `tum_simulator`, los pa-

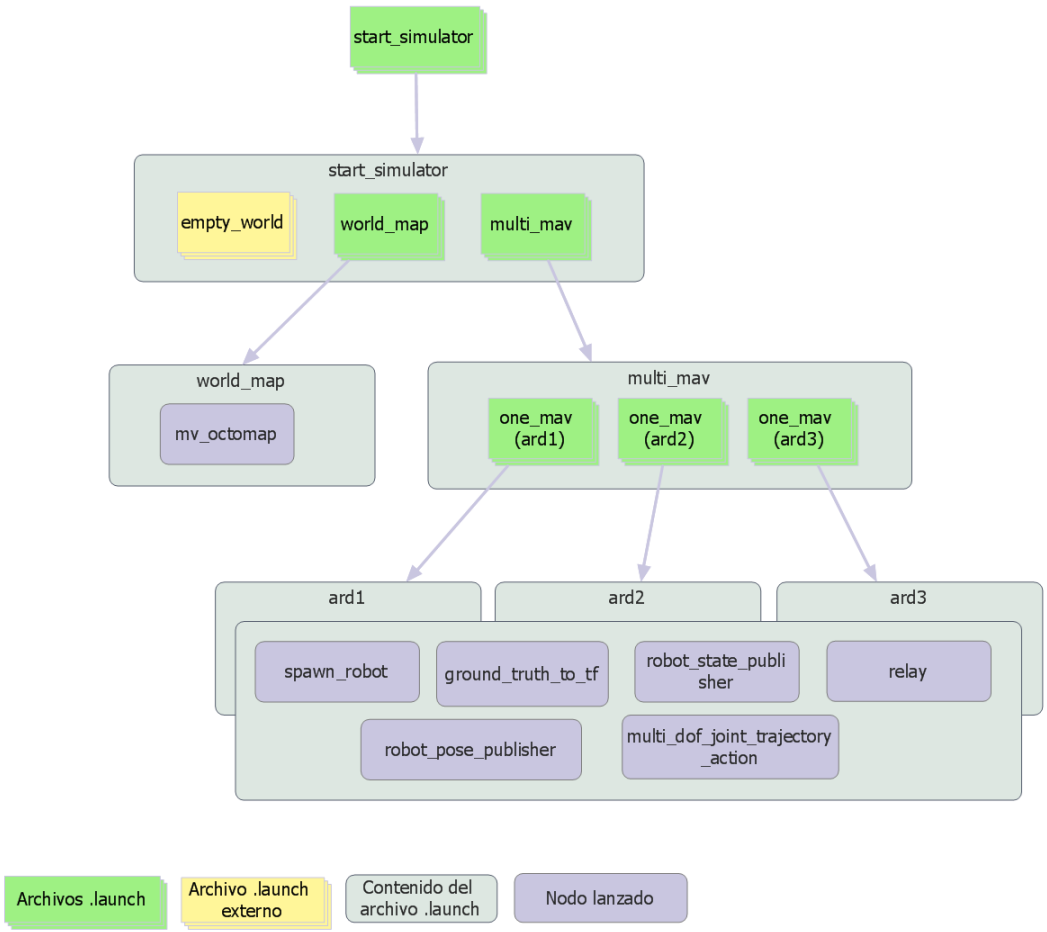


Figura 6.4: Archivos de lanzamiento de la simulación.

quetes referenciados son: `cvg_sim_gazebo`, `cvg_sim_gazebo_plugins`, `cvg_sim_msgs` y `message_to_tf`.

Para lograr los objetivos del presente trabajo de tesis, fue necesario hacer modificaciones que permitieran su extensión a múltiples MAVs mediante otro tipo de control. Entre estas modificaciones se incluyen el uso de *namespaces* y transformaciones con un marco de referencia común. El archivo `.xacro`, que contiene la configuración del modelo del AR.Drone 2.0, llamado `ardrone2_kinect.urdf.xacro`, se ubica en la carpeta `urdf` del paquete `cvg_sim_gazebo`.

Para permitir el uso correcto de espacios de nombres en `tum_simulator`, se modificaron los archivos `cvg_sim_gazebo_plugins/src/quadrotor_simple_controller.cpp`, `cvg_sim_gazebo_plugins/src/quadrotor_state_controller.cpp` y `cvg_sim_gazebo_plugins/urdf/quadrotor_sensors.urdf.xacro` para remover los espacios

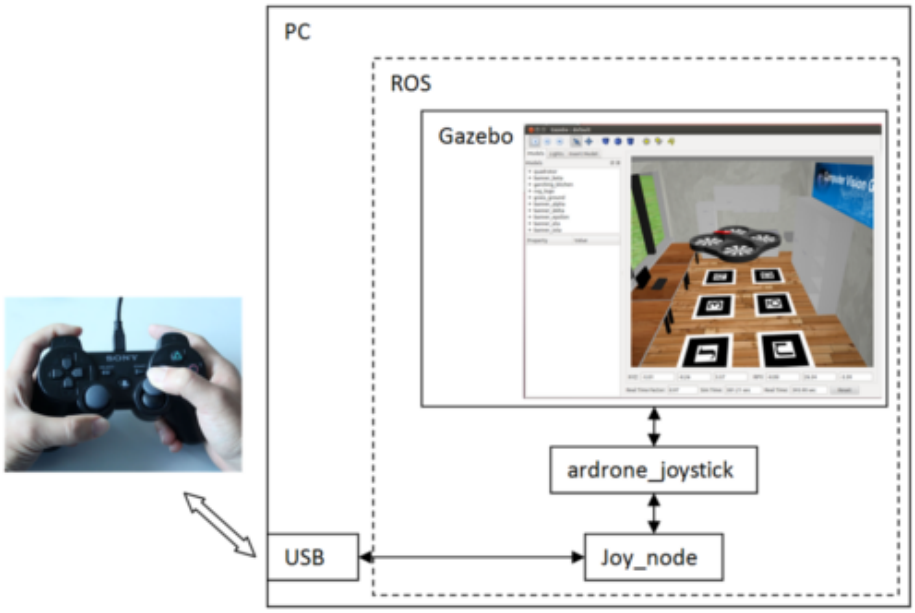


Figura 6.5: Estructura del paquete tum_simulator [Huang and Sturm, 2016].

de nombres globales (/). Con esta modificación, es posible colocar, en el archivo `multi_mav.launch`, las llamadas a los archivos `one_mav.launch`, dentro de grupos cuyo espacio de nombres sigue la convención `ard` previamente definida.

Con el propósito de generar un árbol de transformaciones conectado, se definió el marco `world` como marco de referencia global (ver Figura 6.6) . Para las transformaciones de los mensajes con las posiciones de los MAVs, proporcionados por Gazebo, se utiliza el nodo `ground_truth_to_tf` (del tipo `message_to_tf`). En este nodo, el marco global de referencia se establece en el parámetro `frame_id`. Adicionalmente, es necesario establecer el parámetro `tf_prefix` de los grupos del archivo `multi_mav.launch`, para evitar que los nombres de transformaciones se traslapen entre sí. En el trabajo desarrollado, este parámetro sigue la convención `ard_tf` donde, de igual forma que en los espacios de nombres, después de la palabra `ard` se coloca un número entero positivo que identifica a cada MAV.

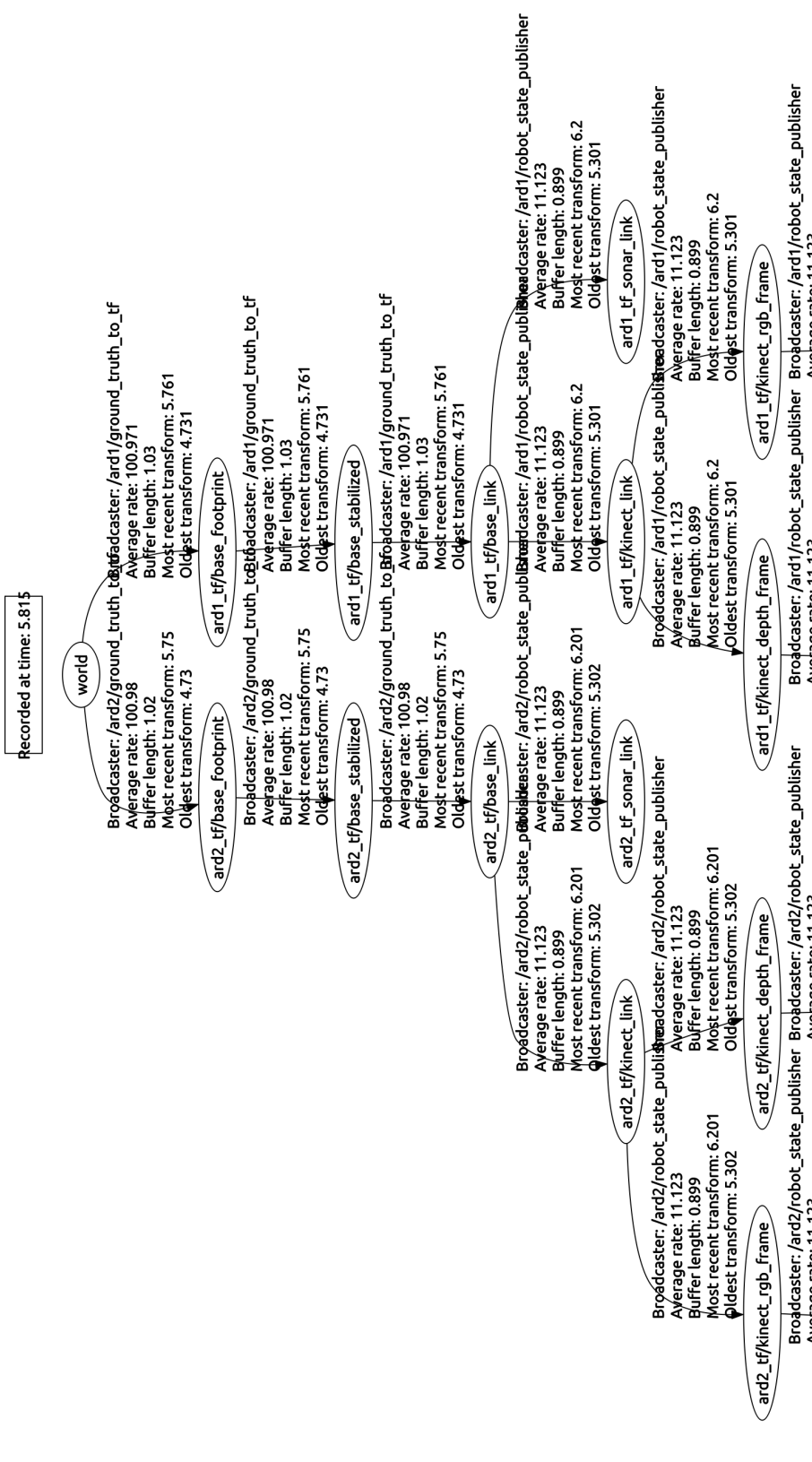


Figura 6.6: Transformaciones entre marcos de referencia

6.2.3. Adquisición de información del entorno

Para que los MAVs puedan construir un modelo del entorno simulado, necesitan un sensor que brinde información de profundidad. Con este propósito, se acopló a los MAVs un sensor RGB-D que simula un dispositivo Microsoft Kinect. La referencia al modelo del sensor se encuentra en el archivo `ardrone2_kinect.urdf.xacro` y está fijo al marco `base_link` del cuerpo del MAV.

La definición del modelo del sensor se ubica en el archivo `cvg_sim_gazebo/urdf/-kinect_camera_custom.urdf`. Este modelo, además de tener las texturas del dispositivo Kinect, utiliza el *plugin* de Gazebo `libgazebo_ros_openni_kinect` para proveer la información del sensor con la misma interfaz que se utiliza para un sensor Microsoft Kinect real. El sensor provee distintos tipos de información, pero la información utilizada por el sistema desarrollado es una nube de puntos. Esta nube es publicada mediante mensajes, cuyos marcos de referencia también son afectados por el parámetro `tf_prefix` (ver Figura 6.6).

Mediante el uso de este sensor, los MAVs adquieren información del entorno, la cual es publicada en el tema `/ard/kinect/depth/points`. Esta forma de publicar las nubes de puntos es idéntica a la forma en como se haría con MAVs reales, i.e., cada MAV montaría su propio sensor y proporcionaría una nube de puntos independiente. Las nubes de puntos de los MAVs son utilizadas localmentente para construir un mapa de ocupación que sirve para la planificación de movimiento (ver Sección 6.3) y la identificación de celdas frontera (ver Sección 6.5).

Con el propósito de mantener un sistema débilmente acoplado, el nodo `mv_octomap`, encargado de la construcción del mapa en el coordinador, no se comunica directamente con los temas donde los MAVs publican las nubes de puntos. En su lugar los MAVs utilizan la herramienta `relay` del paquete `topic_tools`, para reenviar las nubes de puntos a un tema en común llamado `/kinect/depth/points` (ver Figura 6.7).

Una vez que el nodo `mv_octomap` tiene acceso a la nube de puntos, puede construir un mapa de ocupación global utilizando la biblioteca Octomap. A nivel de código C++, el nodo `mv_octomap` es creado por la clase `MvOctomapServer` que hereda de `TrackingOctomapServer`, la cual pertenece al paquete `octomap_server`. Este paquete es una implementación para ROS que se subscribe a una nube de puntos y publica un mapa de ocupación.

6.2.4. Posición de los MAVs

Con el propósito de distinguir a los MAVs y a las metas que éstos generan, se modificó el paquete `robot_pose_publisher` [Toris, 2016] para publicar marcadores de visualización con la posición de los MAVs. Los nodos encargados de la publicación de marcadores se ejecutan en los mismos espacios de nombres de los MAVs, por lo tanto

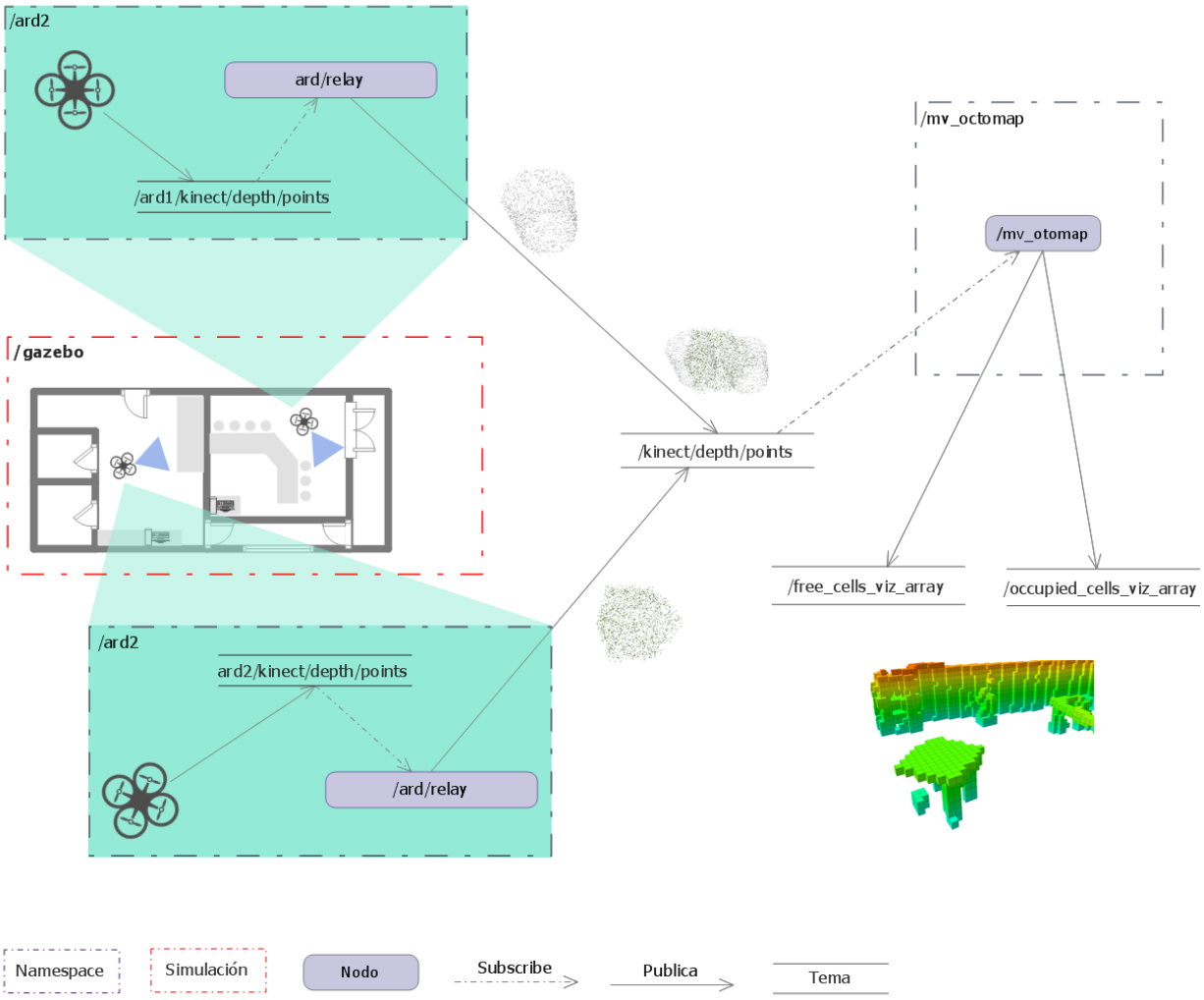


Figura 6.7: Adquisición de información por parte de los MAVs para el mapa global.

son independientes entre sí.

El paquete `robot_pose_publisher`, como su nombre lo indica, está diseñado para publicar la posición de un robot. En el caso del sistema desarrollado, el robot es un MAV y se extiende a múltiples MAVs mediante archivos de lanzamiento. El paquete `robot_pose_publisher` usa la función `lookupTransform()` de la clase `tf:transformer` para obtener la transformación entre el marco de referencia global y el marco de referencia de un MAV. Dicha transformación contiene métodos para acceder a la posición y orientación del marco de referencia del MAV.

Utilizando la posición provista por la transformación, se crean marcadores de visualización del tipo flecha, cuyo color es asignado en el archivo de lanzamiento `multi_mav.launch`. Este color es transferido al archivo `one_mav.launch` y finalmente, pasado como parámetro al nodo `robot_pose_publisher`. El color es de importancia, puesto que es el mismo que se asigna a las metas generadas por los MAVs.

6.3. Configuración del entorno de planificación de movimiento

En esta sección se describen las acciones realizadas para dotar de capacidades de planificación de movimiento y ejecución de trayectorias a los MAVs simulados. En la Sección 6.3.1 se detalla la configuración del paquete de MoveIt!, el cual sirve para ejecutar un grupo de movimiento configurado de acuerdo a las dinámicas del MAV. En la Sección 6.3.2 se presenta la configuración de los archivos del paquete de MoveIt!. En la Sección 6.3.3 se presentan las acciones realizadas para permitir el funcionamiento del paquete de MoveIt! con múltiples MAVs, ya que MoveIt! está diseñado para trabajar con un solo robot. En la Sección 6.3.4 se describe el funcionamiento del nodo encargado de ejecutar los planes de movimiento necesarios para explorar metas.

6.3.1. Paquete de configuración de MoveIt!

MoveIt! [Ioan A. and Sachin, 2016] es un *framework* de planificación de movimiento de ROS, cuyo nodo principal es el grupo de movimiento `move_group`. Los paquetes creados para MoveIt! constan de una serie de archivos de configuración que utilizan el modelo de un robot. MoveIt! no está diseñado para trabajar con múltiples robots, por lo tanto la configuración se realiza con uno solo de ellos. Aunque el sitio oficial de MoveIt! provee paquetes para algunos robots especializados, este no es el caso del AR.Drone 2.0. Sin embargo, es posible crear un paquete de MoveIt! mediante el uso de su asistente (ver Figura 6.8).

La configuración del paquete se realizó utilizando el modelo `urdf` del MAV, ubicado en el archivo `ardrone2_kinect.urdf.xacro`. La matriz de auto-colisión se generó con los parámetros por defecto. Esta matriz indica las posibles colisiones del robot consigo y su uso es común en robots con múltiples articulaciones. MoveIt! requiere una articulación virtual entre el robot y un marco de referencia fijo. Para el MAV simulado, se utilizaron el marco `base_link` como referencia del robot y el marco global `/world` como marco fijo. El nombre del grupo de planificación utilizado es `ardrone2_group` y contiene a la articulación virtual. Este grupo es diferente al grupo de movimiento y sirve para definir partes del MAV.

Los efectores finales y artículaciones pasivas no se configuraron en el MAV. Estos efectores son elementos de los robots, por lo general el final de los brazos, los cuales

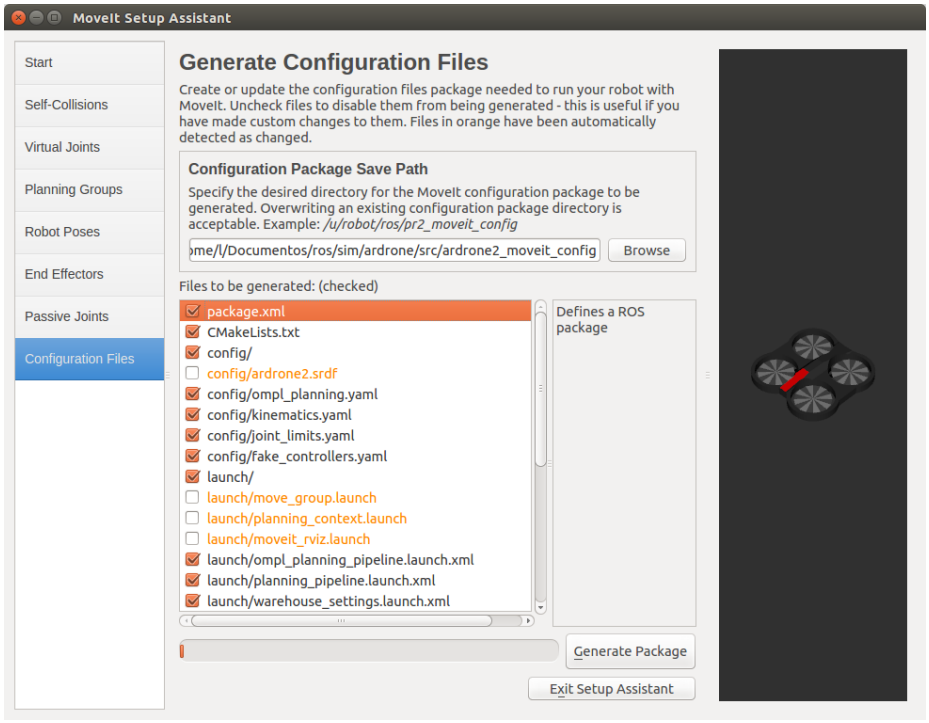


Figura 6.8: Configuración del paquete de MoveIt! para el MAV AR.Drone 2.0 simulado.

sirven para interactuar con objetos mediante agarre o manipulación. Por otra parte, las articulaciones pasivas son articulaciones para las cuales no se debe crear un plan de movimiento.

6.3.2. Configuración de los archivos del paquete de MoveIt!

El paquete producido por MoveIt! para el MAV es `ardrone2_moveit_config` y consta de múltiples archivos de lanzamiento y configuración divididos en las carpetas `config` y `launch`, sin embargo únicamente se modificaron algunos de ellos para configurar el entorno (ve Figura 6.9). Dichos archivos cuentan con la información necesaria del modelo para realizar planificación de movimiento, pero no poseen información alguna para comunicarse con un MAV. Es importante notar que hasta este punto MoveIt! no distingue si el modelo es de un MAV real o simulado.

Con el propósito de especificar el nodo encargado de ejecutar las trayectorias generadas por el planificador de movimiento, se modificó el archivo `ardrone2_moveit_config/config/controllers.yaml` (ver Archivo 6.1). En este archivo, se indica el nodo del servidor de acciones (parámetro `name`), el nombre de la especificación de la acción (parámetro `type`) y las articulaciones del robot. Para el MAV simulado, el nodo que actúa como servidor de acciones es `multi_dof_joint_trajectory_action` y la acción es `MultiDofFollowJointTrajectory` (ver Sección 6.3.4). La única articulación del

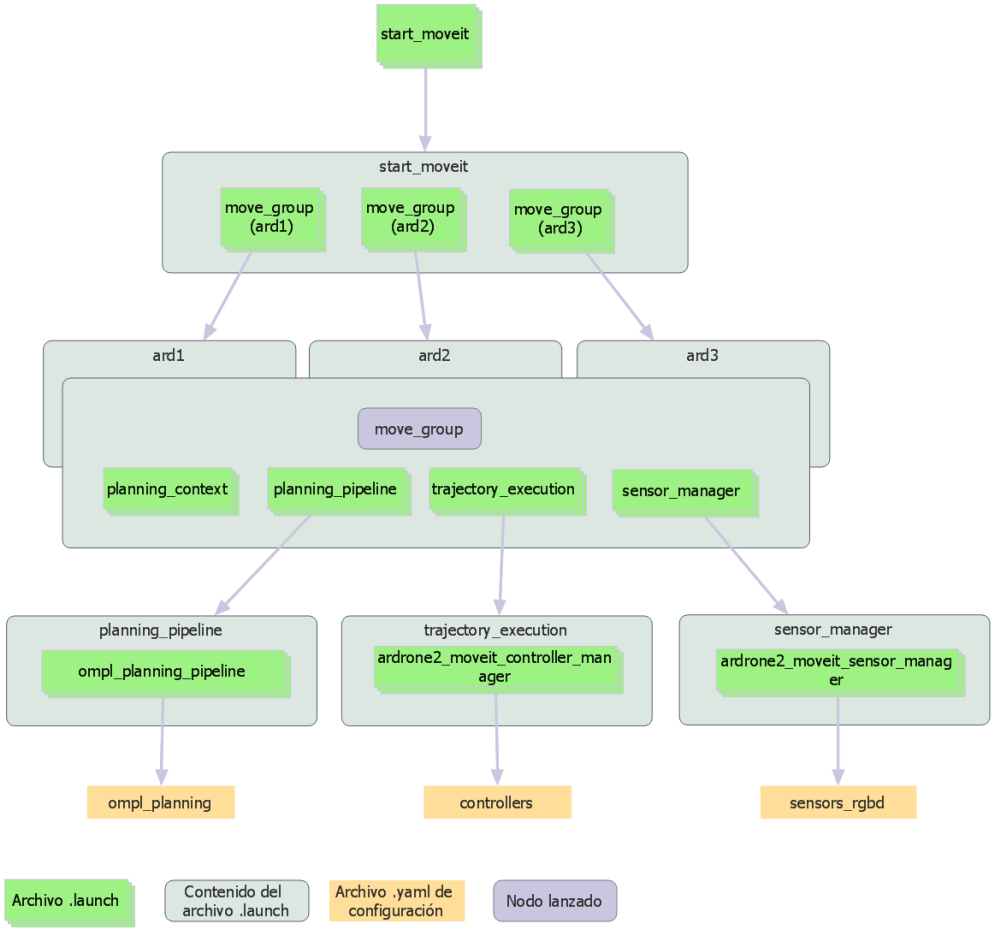


Figura 6.9: Archivos de lanzamiento para el paquete de MoveIt!

MAV es la articulación virtual definida en el paquete de MoveIt! (ver Sección 6.3.1).

Archivo 6.1: controllers.yaml

```
1 controller_list:
2   - name: multi_dof_joint_trajectory_action
3     type: MultiDofFollowJointTrajectory
4     default: true
5     joints: [virtual_joint]
```

Como paso siguiente se modificó el archivo `ardrone2_moveit_controller_manager.launch.xml` en la carpeta `launch` (ver Archivo 6.2). En este archivo únicamente se especificó el nombre del paquete de MoveIt! creado para permitir que el manejador de controladores encuentre el controlador correspondiente al MAV simulado.

Para que el grupo de movimiento construya un mapa del entorno, que le permita planificar rutas libres de colisiones, debe tener una fuente de información. Esta fuente

Archivo 6.2: ardrone2_moveit_controller_manager.launch.xml

```
1 <?xml version="1.0"?>
2
3 <launch>
4        <arg name="moveit_controller_manager"
5       default="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
6        <param name="moveit_controller_manager"
7         value="$(arg moveit_controller_manager)"/>
8        <rosparam file="$(find ardrone2_moveit_config)/config/controllers.yaml"/>
9 </launch>
```

se asigna en el parámetro `point_cloud_topic` del archivo `sensors_rgbd.yaml`. Dado que estos archivos de MoveIt! están diseñados para un solo robot, el tema asignado es `kinect/depth/points`, sin espacio de nombres.

6.3.3. Configuración del paquete de MoveIt! para múltiples MAVs

MoveIt! no está diseñado para trabajar con múltiples robots, por lo tanto fue necesario crear un archivo de lanzamiento, `start_moveit.launch`, que permita realizar esta tarea. Un problema adicional es que el grupo de movimiento no permite especificar el parámetro `tf_prefix`, por esta razón fue necesario modificar líneas del código fuente de MoveIt!. Este parámetro permite distinguir entre transformaciones de marcos de referencia.

El archivo llamado `start_moveit.launch` se compone de tres grupos dentro de los cuales se realiza la llamada al archivo de lanzamiento `move_group.launch`. En estos grupos se definen los espacios de nombres `ard` y se establece el valor falso al parámetro `trajectory_execution/execution_duration_monitoring`. Este parámetro es el encargado de limitar el tiempo de ejecución de las trayectorias de los MAV. En caso de conservarse esta limitación, los MAVs podrían no alcanzar su objetivo a tiempo.

En el archivo `move_group.launch` se encuentran la llamada al nodo del mismo nombre y los archivos de lanzamiento: `planning_context.launch`, `planning_pipeline.launch`, `trajectory_execution.launch` y `sensor_manager.launch` (ver Figura 6.9). Estos últimos dos archivos actúan como envoltorios para llamar a los archivos específicos del MAV, mientras que `planning_pipeline.launch` carga el *plugin* responsable de la planificación de movimiento. Este *plugin*, `ompl_planning_pipeline`, como su nombre lo indica, es el *plugin* de la biblioteca abierta de planificación de movimiento (OMPL) para MoveIt! (ver Sección 3.2.5) y utiliza el archivo `ompl_planning.yaml` para obtener la configuración de sus planificadores.

Cuando se incluyen los espacios de nombres `ard` directamente en el archivo `start-`

_moveit.launch, el nodo move_group se vuelve incapaz de encontrar los temas del resto de los archivos de lanzamiento. La solución a este problema consiste en utilizar diferentes espacios de nombres para dichos archivos y para el nodo move_group. El espacio de nombres de los archivos de lanzamiento debe ser ard/move_group, mientras que el del nodo move_group debe ser ard.

Los marcos de referencia de los MAVs simulados se diferencian entre sí mediante el parámetro tf_prefix, sin embargo este parámetro no es utilizado en MoveIt!. Este problema se resolvió modificando el paquete planning de MoveIt!. Específicamente, se incluyó el prefijo en el archivo planning_scene_monitor.cpp de la carpeta planning_scene_monitor. El prefijo falso se crea concatenando el espacio de nombres del nodo raiz (en que se ejecuta el código de dicho archivo) con el texto _tf. El único lugar donde debe agregarse este prefijo es en la función getShapeTransformCache del mismo archivo. En esta función, el prefijo se concatena al nombre de las primeras consultas de transformaciones (ver Archivo 6.3).

Archivo 6.3: Modificaciones al archivo planning_scene_monitor.cpp

```
1  void planning_scene_monitor::PlanningSceneMonitor::startSceneMonitor(...)
2  {
3    ...
4    if(!this->namespace_scene_display_.empty())
5      this->fake_tf_prefix_ = this->namespace_scene_display_ + "_tf/";
6    else
7    {
8      std::string aux = root_nh_.getNamespace();
9      boost::algorithm::trim_left_if(aux,boost::algorithm::is_any_of("/"));
10     boost::algorithm::trim_right_if(aux,boost::algorithm::is_any_of("/"));
11     this->fake_tf_prefix_ = aux + "_tf/";
12   }
13   ...
14 }
15
16
17 bool planning_scene_monitor::PlanningSceneMonitor::getShapeTransformCache(...)
18 {
19         ...
20         for (LinkShapeHandles::const_iterator it ...)
21         {
22           tf::StampedTransform tr;
23           sourceAuxFrame = this->fake_tf_prefix_ + it->first->getName();
24           tf_->waitForTransform(target_frame, sourceAuxFrame, ...);
25           ...
26
27 }
```

6.3.4. Ejecución de trayectorias

El plan de movimiento generado por el *plugin* de planificación crea un conjunto de puntos destino llamados **metas de posición** (*pose goals*). Estas metas no están relacionadas con las **metas de exploración** del sistema propuesto en el presente trabajo. Las metas de posición son enviadas al nodo responsable de la ejecución de trayectorias, el cual implementa un **servidor de acciones**. Este servidor convierte las trayectorias en comandos de vuelo para el MAV (ver Figura 6.10).

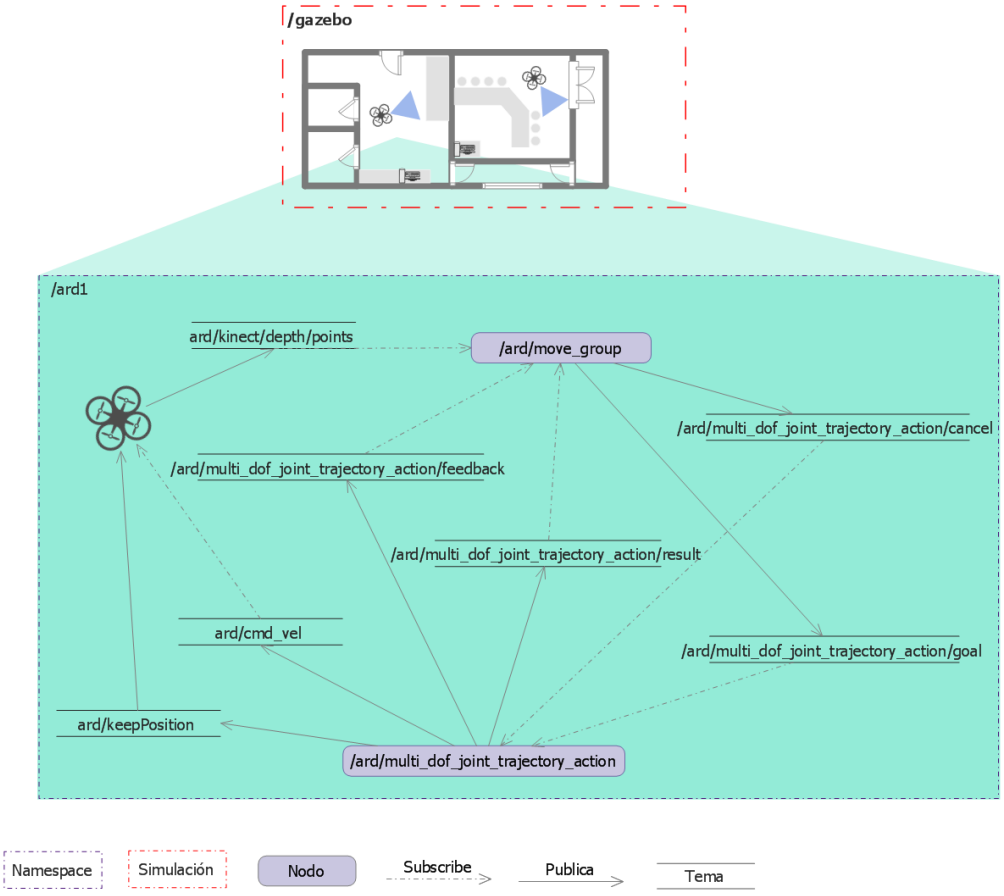


Figura 6.10: Nodos involucrados en la planificación de movimiento y ejecución de trayectorias.

El nodo utilizado para ejecutar trayectorias es `multidof_action_controller`, el cual fue creado utilizando como base el proyecto Autonomous Flight ROS [Tonioni, 2016]. El cambio principal con respecto a dicho proyecto fue la actualización del paquete `action_controller` de la herramienta de compilación `rosmake` a `catkin`, aunque a la fecha del presente documento de tesis ya existe una versión para `catkin`. Adicionalmente, se utilizó el paquete `moveit_simple_controller_manager` del proyecto de

Alessio Tonioni. Este paquete permite a MoveIt! reconocer la acción `MultiDofFollowJointTrajectoryAction` y comunicarse con el servidor de acciones especificado. La necesidad de este paquete surge debido a que MoveIt! no ofrece un controlador por defecto para articulaciones de múltiples grados de libertad, como es el caso del MAV.

El nodo `multidof_action_controller` utiliza las metas de posición recibidas para generar mensajes del tipo `geometry_msgs::Twist`. Estos mensajes se componen de tres velocidades lineales y tres angulares. Estos mensajes son publicados en el tema `cmd_vel` del MAV, el cual realiza los movimientos correspondientes mediante el paquete `ardrone_autonomy`. Este paquete proporciona un `driver` para el MAV AR.Drone 2.0 y está implementado en el paquete `tum_simulator` que provee los MAVs simulados.

Al final de la ejecución de una trayectoria recibida, el nodo `multidof_action_controller` envía un mensaje de estabilización al MAV correspondiente. Este mensaje del tipo `std_msg::empty` es un mensaje vacío publicado en el tema `/ard/keepPosition`. Este tema es publicado por el controlador del MAV y al recibir un mensaje mantiene el MAV a una altura fija.

6.4. Proceso de coordinación

En esta sección se presentan los componentes del paquete `mv_explorer` involucrados con el **proceso de coordinación** (ver Sección 5.2). En la Sección 6.4.1 se presenta la definición de las **metas de exploración** como mensaje de ROS. En la Sección 6.4.2 se detalla la acción utilizada para asignar tareas a los MAVs. Mediante dicha acción se intercambian metas y se da seguimiento a la exploración. En la Sección 6.4.3, se describe el funcionamiento de los componentes involucrados en la asignación y actualización de metas. Finalmente, en la Sección 6.4.4, se describen las acciones realizadas para crear y publicar diagramas de Voronoi.

6.4.1. Metas de exploración

Las **metas de exploración** fueron definidas como mensajes personalizados de ROS. Estos mensajes representan información a intercambiar. Para definir un mensaje personalizado en ROS, se utiliza un archivo de descripción, en el cual se indican los nombres y tipos de variables. El mensaje correspondiente a las metas de exploración es `MvGoal`. Este mensaje es usado tanto por el coordinador como por los MAVs y con excepción de la propiedad `asignada`, la definición es idéntica a la presentada en el Capítulo 5 (ver Sección 5.1.1). La propiedad `asignada` se implementó de manera interna en el código.

El mensaje de las metas de exploración se compone del identificador de la meta, su posición, el identificador del MAV que la generó, el identificador del MAV al que está

asignada, el estado de la meta y un campo que indica si dicho estado ha cambiado. El identificador de la meta, `goalId`, es un número entero positivo. La posición de la meta, `pose`, representa la ubicación de la meta y está compuesta por la posición en 3D (x, y y z) y la orientación en forma de cuaternión (x, y, z y w). El identificador del MAV, que generó la meta `mavOriginId`, es el identificador del MAV que la creó durante el **proceso de exploración**. El identificador del MAV al que se asignó la meta, `mavAssignedId`, es el identificador del MAV que está encargado de explorarla. Por otra parte, el estado, `status`, indica la condición en que se encuentra la meta:

- `NEW`: no se ha hecho acción alguna sobre la meta;
- `ASSIGNED`: la meta ha sido asignada a un MAV por el coordinador o seleccionada por dicho MAV;
- `DISCARDED`: el coordinador descartó la meta por su cercanía a otras metas;
- `INACCESSIBLE`: el MAV que tiene asignada la meta no pudo generar un plan de movimiento hacía ella y
- `EXPLORED`: la meta ha sido explorada por el MAV al que se le asignó.

Archivo 6.4: Mensaje de las **metas de exploración** definido en el archivo MvGoal.msg.

```
1  uint32 goalId
2  geometry_msgs/Pose pose
3  uint8 mavOriginId
4  uint8 mavAssignedId
5  uint32 status
6  bool statusChanged
7
8  #Goal status codes
9  uint32 NEW = 0 #Not assigned
10 uint32 ASSIGNED = 1 #Assigned
11 uint32 DISCARDED = 2 #Discarded
12 uint32 INACCESSIBLE = 3 #Inaccessible
13 uint32 EXPLORED = 4 #Completed
```

6.4.2. Acción de exploración

El método para que el coordinador y los MAVs puedan intercambiar información en forma de **metas de navegación** requiere de la definición de una acción. Las acciones son tareas que se ejecutan en un nodo que actúa como servidor de acciones. Estas tareas son llamadas metas, pero no deben confundirse con las **metas de exploración** del sistema propuesto. El archivo de definición de acción se compone de tres partes separadas por guiones "`---`". En la primera parte se especifica la meta que será ejecutada por el servidor. En la segunda parte, se indica la respuesta del servidor y en la última parte, se coloca el progreso de la tarea (*feedback*).

Meta de la acción

La meta de la acción es el mensaje que reciben los MAVs. Dicha meta se compone de los límites de la caja envolvente del diagrama de Voronoi, las **metas de exploración** utilizadas como sitios del diagrama, las metas que fueron actualizadas en el coordinador, la meta asignada al MAV, el máximo número de metas a explorar y el tipo de exploración a realizar (ver línea 1 del Código 6.5). La caja envolvente, `boundingBox`, es necesaria ya que el coordinador únicamente envía los sitios del diagrama de Voronoi, `goalSites`. Por otra parte, las metas actualizadas, `updatedGoals`, y la meta a explorar, `goal`, son necesarias para realizar la exploración. El número máximo de metas a explorar, `maxExploreGoals`, se utiliza para limitar la cantidad de metas que los MAVs exploran en su región asignada. El tipo de exploración, `explorationTye`, indica la acción que deben realizar los MAVs con la meta recibida:

- `GOAL_ASSIGNED`: explorar la meta;
- `NO_GOAL`: ignorar la meta, pero adquirir información en su posición y
- `FINISHED`: terminar la exploración y aterrizar.

Al terminar la exploración, la acción por defecto es aterrizar, sin embargo para la presente implementación no hay una diferencia significativa en ordenarles que vuelvan al punto donde despegaron, ya que la planificación de movimiento corre a cargo de MoveIt!.

Archivo 6.5: Definición de la acción para intercambio de metas en el archivo ExploreGoal.action

```
1  # Goal
2  mv_explorer/BoundingBox boundingBox
3  mv_explorer/MvGoal[] goalSites
4  mv_explorer/MvGoal[] updatedGoals
5  mv_explorer/MvGoal goal
6  uint32 maxExploreGoals
7  uint32 explorationType
8
9  #Exploration types
10 uint32 GOAL_ASSIGNED = 0
11 uint32 NO_GOAL = 1
12 uint32 FINISHED = 2
13 ---
14 # Result
15
16 uint32 mavId
17 uint32 exploredGoals
18 mv_explorer/MvGoal[] resultGoals
19 uint32 resultCode
20
21 # Result codes
```

```
22 uint32 ERROR = 0
23 uint32 MAX_GOALS_REACHED = 1
24 uint32 GOALS_RAN_OUT = 2
25 ---
26 # Feedback
27 uint32 status_code
28
29 # Status codes
30 uint32 RECEIVED = 0
31 uint32 TRAVELLING = 1
32 uint32 SCANNING = 2
33 uint32 NEW_SELECTED = 3
```

Resultado de la acción

El resultado de la acción es recibido por el coordinador. Este resultado se compone del identificador del MAV, el número de metas que exploraron, las metas que generaron y actualizaron y un código de resultado de la exploración (ver línea 14 del Código 6.5). El identificador del MAV, `mavId`, ayuda a saber de qué MAV proviene la respuesta, puesto que se reciben como *callbacks* por parte de los servidores de acciones. El número de metas exploradas, `exploredGoals`, es de carácter informativo y para conocer cuántas metas exploró el MAV, incluyendo la meta enviada por el coordinador. Las metas actualizadas por el MAV, `resultGoals`, son la lista de metas que el MAV generó y actualizó, i.e., metas que fueron creadas o cuyo estado cambió. El código de resultado, `resultCode`, proporciona información acerca del motivo por el cuál terminó la exploración:

- `ERROR`: ocurrió un problema en la exploración, e.g., excepción en el código;
- `MAV_GOALS_REACHED`: se alcanzó el número máximo de metas para explorar en la región asignada y
- `GOALS_RAN_OUT`: no hay más metas para explorar en la región asignada.

Progreso de la acción

Los mensajes de progreso de la acción ejecutada en los MAVs son recibidos por el coordinador. Estos mensajes se componen únicamente de un código de estado (ver línea 26 del Código 6.5). El código de estado, `status_code`, brinda información acerca del estado del MAV:

- `RECEIVED`: el MAV recibió la meta;
- `TRAVELLING`: el MAV está generando un plan de movimiento y ejecutando la trayectoria;

- `SCANNING`: el MAV está adquiriendo información del entorno mediante una rotación de 360° con respecto a su eje Z y
- `NEW_SELECTED`: el MAV eligió otra meta para explorar en su región asignada.

6.4.3. Asignación y actualización de metas

Las metas correspondientes a los MAVs son definidas en el componente `MAVCoordinator`. Este componente es una clase C++ que implementa las fases de inicialización, asignación de metas y actualización del diagrama de Voronoi (ver Sección 5.2). Para enviar metas a los MAVs, el sistema hace uso del componente `MvMAV`. Este componente crea un cliente de acciones que proporciona las funciones necesarias para conectarse con los servidores de acciones de los MAVs, a través de los mensajes definidos para la acción de exploración.

Los clientes de acciones necesitan definir una función que sirva como *callback* y otra como *feedback*. La función *feedback* se encuentra en el componente `MvMAV`, pero la función *callback* pertenece a `MAVCoordinator`. Esta estructura se debe a que el coordinador debe procesar las respuestas de los MAVs para agregarlas a su contenedor de metas.

La comunicación entre el cliente (coordinador) y los servidores de acciones (MAVs) se hace mediante tres temas (ver Figura 6.11). El servidor de acciones recibe las tareas a través del nodo `/ard/goal_explorer_action/goal`. El resultado de dicha tarea es devuelto mediante el tema `/ard/goal_explorer_action/result`. El tema utilizado para recibir el progreso de las metas es `/ard/goal_explorer_action/feedback`. A pesar de que un servidor de acciones por defecto cuenta con un tema para cancelar las metas, `/ard/goal_explorer_action/cancel`, en el presente sistema no se utiliza. Cabe mencionar que la suscripción y publicación de los temas no es explícita, sino a través de la interfaz de la biblioteca `actionlib` de ROS.

El componente `CoordinatorGoalContainer` implementa la fase de actualización de metas del coordinador y es invocado por `MAVCoordinator` (ver Sección 5.2.2). El coordinador utiliza el componente `CoordinatorGoalContainer` para almacenar las metas que recibe de los MAVs. Este componente utiliza *mutex* pertenecientes al paquete `ecl` de ROS para permitir la actualización concurrente de la lista de metas. Este tipo de actualización se requiere, debido a que las respuestas de los MAVs pueden llegar simultáneamente.

6.4.4. Generación y visualización del diagrama de Voronoi

La generación del diagrama de Voronoi que define las regiones asignadas, se realiza en el componente `VoronoiGenerator`. Este componente utiliza la biblioteca Voro++ para generar diagramas de Voronoi 3D [Rycroft, 2016]. En esencia, el componente

`VoronoiGenerator` únicamente enmascara las funciones de Voro++, en caso de que se necesite reemplazarlo. Voro++ permite exportar el diagrama como un conjunto de puntos y aristas en un formato de texto apto para la herramienta `Gnuplot`. Este formato es utilizado por el componente `VoronoiMarkerPublisher` para crear marcadores de visualización.

Para observar las regiones del diagrama de Voronoi en las que los MAVs generan metas de exploración, se creó el componente `VoronoiMarkerPublisher`. Este componente lee las celdas exportadas por Voro++ y construye marcadores de visualización en forma de líneas para las aristas del diagrama, incluyendo la caja contenedora. Para construir los marcadores de los sitios del diagrama, se utilizan los puntos exportados por Voro++. Estos marcadores de visualización son publicados por una función invocada por un objeto `timer` de ROS.

6.5. Proceso de exploración

En esta sección, se describe el funcionamiento de los componentes del sistema relacionados con el **proceso de exploración** (ver Sección 5.3). En la Sección 6.5.1 se explica la forma en cómo se realiza el proceso de exploración, utilizando los componentes desarrollados. En la Sección 6.5.2, se presentan las acciones realizadas para permitir que los MAVs puedan desplazarse a las posiciones de las metas. En la Sección 6.5.3 se describe el funcionamiento de los componentes utilizados para extraer fronteras y generar metas de exploración. En la Sección 6.5.4, se presenta la generación de marcadores de visualización realizada para representar a las metas.

6.5.1. Exploración de las metas

Para que los MAVs sean capaces de interactuar con el coordinador, se definió un **servidor de acciones**, el cual utiliza la definición de la acción de exploración para recibir tareas, enviar resultados y notificar el progreso de la tarea (ver Sección 6.4.1). El servidor de acciones se encuentra definido en la clase `MAVExplorer`, la cual controla el proceso de exploración (ver Sección 5.3). Cuando se recibe una tarea del coordinador, se crea un hilo para atenderla en la función `goalCallBack` (ver Sección 5.3.1).

El nodo creado para exploración por parte de los MAVs es `/mv_explorer_ard`. A diferencia del resto de los nodos utilizados por los MAVs, `/mv_explorer_ard` no utiliza un espacio de nombres `ard`. Esta situación se debe a la API de MoveIt! utilizada en el componente `MotionExecutor`. Dicha API se vuelve incapaz de encontrar los nodos de los grupos de movimiento de los MAVs, cuando el nodo que la utiliza se encuentra dentro de un espacio de nombres. Este nodo también es el encargado de funcionar como servidor de acciones.

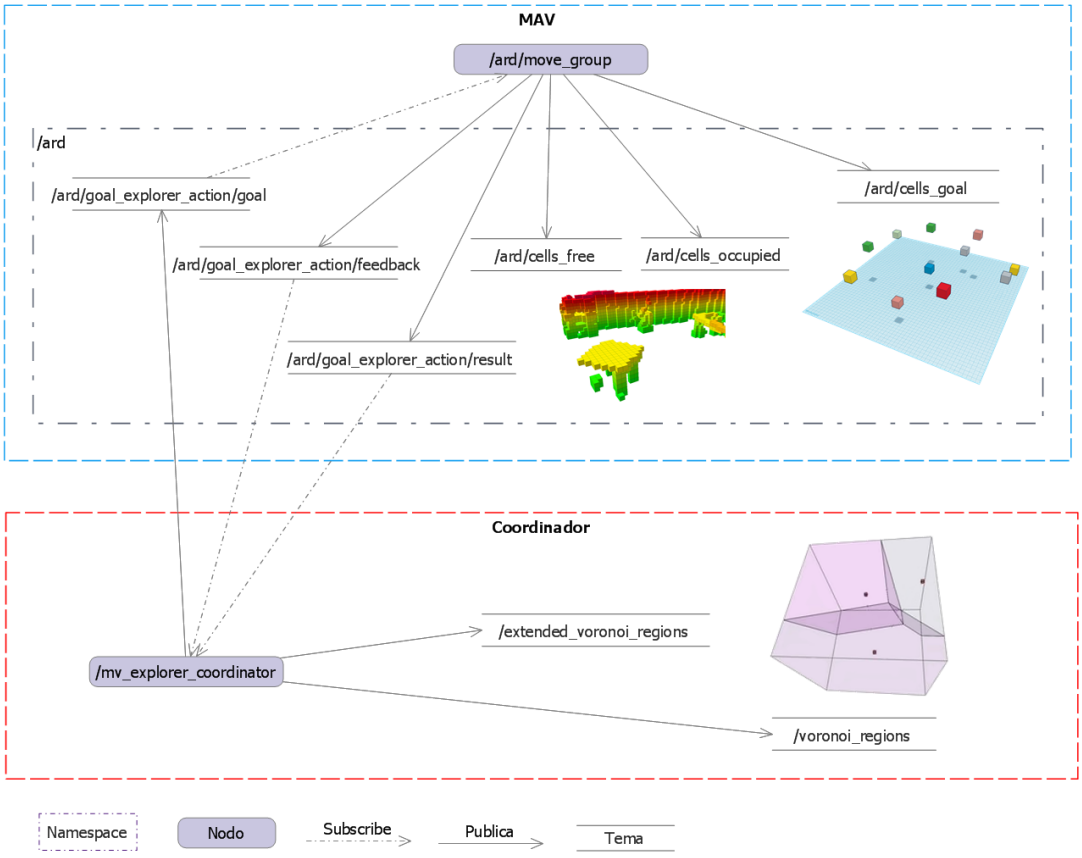


Figura 6.11: Nodos y temas para los procesos de coordinación y exploración.

Los MAVs se desplazan a las metas mediante el componente `MotionExecutor` y adquieren información rotando en su eje Z. Esta rotación en los MAVs no corre a cargo del componente `MotionExecutor`, sino que se envían directamente mensajes `geometry_msgs::Twist` al tema `/ard/cmd_vel` de los MAVs. En estos mensajes se asigna la velocidad angular en el eje Z y el número de mensajes está calculado para que el MAV rote aproximadamente 360°.

Cuando un MAV ha explorado una meta, debe seleccionar otra en su región asignada (ver Algoritmo 5.13 de la Sección 5.3.3). Esta fase se encuentra implementada en el componente `MAVGoalContainer`, el cual es invocado desde el componente `MAVExplorer`. `MAVGoalContainer` utiliza el mapa local construido con Octomap para medir la ganancia de información en cada meta y verifica si un MAV se encuentra en su región asignada, mediante el componente `VoronoiGenerator`.

El componente `MAVGoalContainer` es una clase C++ que contiene la lista de metas

locales del MAV. Esta clase además mantiene una copia de las posiciones de las metas en una variable de tipo `pcl:octree`, la cual permite realizar consultas espaciales de una manera más sencilla, e.g., búsqueda de puntos cercanos o puntos en un radio determinado. `MAVGoalContainer` también implementa mecanismos de concurrencia para el acceso a la lista de metas. Por esta razón, todas las modificaciones a la lista de metas son realizadas a través de los métodos que ofrece dicho componente.

6.5.2. Planificación y ejecución de movimiento en los MAVs

El componente que se encarga de desplazar el MAV hasta una meta es `MotionExecutor`, el cual es una clase de C++ que utiliza la API del grupo de movimiento de MoveIt!. Para utilizar esta API, primero es necesario crear y configurar un paquete de MoveIt! (ver Sección 6.3.1). La API proporciona un objeto `MoveGroup` que brinda las funciones necesarias para realizar planificación de movimiento y ejecución de trayectorias. Para generar un plan de movimiento, se debe especificar la posición deseada e invocar a la función `plan` del objeto `MoveGroup`. Si la creación del plan resulta exitosa, se puede invocar a la función `execute` para que el grupo de movimiento asigne la tarea al nodo responsable de la ejecución de trayectorias (ver Sección 6.3.4). En caso de que no se pueda generar el plan de movimiento, se procede a marcar la meta como inaccesible.

Para poder utilizar la API, se requiere que el nodo correspondiente al grupo de movimiento, `/ard/move_group`, se encuentre en ejecución. Adicionalmente, al crear el objeto `MoveGroup`, es necesario especificar un **manejador de nodo** (`NodeHandle`) que posea el mismo espacio de nombres que el nodo del grupo de movimiento. Esta configuración del objeto es indispensable para poder planificar movimiento y ejecutar trayectorias.

El objeto `MoveGroup` permite especificar el planificador de movimiento y el **espacio de trabajo** (*workspace*). Para el caso del sistema propuesto en el presente trabajo, dicho espacio coincide con la caja envolvente del diagrama de Voronoi. El espacio de trabajo constituye los límites de la zona en que el planificador de movimiento genera trayectorias posibles. Los límites de este espacio ayudan a reducir el tiempo de planificación y sobre todo previenen que los MAVs se desplacen más allá de dichos límites.

El componente `MotionExecutor` también implementa métodos para publicar mensajes de despegue y aterrizaje del MAV. Estos mensajes son enviados a los temas `/ard/ardrone/land` y `/ard/ardrone/takeoff`. Los mensajes que estos temas utilizan son mensajes vacíos del tipo `std_msgs::Empty`. El mensaje de despegue es enviado al inicio de la exploración, mientras que el mensaje de aterrizaje es enviado al final de la exploración.

6.5.3. Generación de metas

Para realizar el proceso de generación de metas (ver Sección 5.3.2) se utiliza el componente `GoalGenerator`. Este componente es una clase C++, la cual usa parte del código del proyecto FBET 3D para identificar fronteras. FBET 3D está implementado con el código del paquete `OctomapServer`, por lo tanto su funcionamiento es similar al de dicho paquete. Las principales modificaciones que se hicieron a FBET 3D fueron: obtener correctamente el centroide de los grupos de fronteras y agregar **fronteras candidatas** al contenedor local de **metas de exploración**. Otros cambios menores implicaron: modificar la forma en cómo se identifican fronteras, descartar la selección de la mejor frontera candidata, agregar *mutex* al mapa para permitir su limpieza y establecer límites en el eje Z para las fronteras candidatas.

Aunque FBET 3D dice utilizar el centroide de los grupos de celdas frontera para producir celdas candidatas, en la implementación el centroide no es obtenido correctamente. En su lugar la función `find_center` utiliza la primera celda de dichos grupos. Para obtener el centroide correctamente, las celdas de cada grupo se agregaron a un objeto `octree` de la biblioteca PCL (*Point Cloud Library*). A continuación, se utilizó la función `compute3DCentroid` para obtener el punto centroide. Este centroide no necesariamente coincide con la posición de una celda frontera del grupo. Por esta razón, se buscó la celda del grupo más cercana a dicho centroide, mediante la función `nearestKSearch` del `octree`.

Las fronteras candidatas son agregadas al contenedor de metas, implementado en la clase `MAVGoalContainer`, con el fin de convertirlas en metas de exploración con una distancia mínima entre sí (ver Algoritmo 5.12 de la Sección 5.3.2). Para verificar la distancia entre metas, se utiliza la función `radiusSearch` del objeto `octree` que almacena las posiciones de las metas en la clase `MAVGoalContainer`.

Ya que FBET 3D está basado en el código de `OctomapServer`, también publica las celdas libres y ocupadas de su mapa interno. Las funciones correspondientes a dicha publicación fueron colocadas en el componente `GoalAndMarkerPublisher`. Estas celdas son publicadas como arreglos de marcadores de visualización de RViz en los temas: `/ard/cells_occupied` y `/ard/cells_free`.

6.5.4. Visualización de metas

Con el propósito de observar el proceso de exploración, se creó el componente `GoalAndMarkerPublisher`, el cual es responsable de publicar las metas y el mapa local de los MAVs. Tanto las metas como el mapa local son publicados en la forma de marcadores de visualización de RViz. La función de publicación es invocada por un objeto `Timer` de ROS en la clase `MAVExplorer`.

Dado que los MAVs almacenan, tanto las metas que han generado como las metas

Algoritmo 6.1 Publicación de metas de exploración

Requiere: Conjunto de metas $\mathcal{G}$ del MAV m y MAV m

```
1: procedure PublicarMetasMAV(𝒢, m )
2:     𝒱_𝕄 ← ∅                                          ▷ Marcadores de visualización
3:     𝒱_color ← GenearColoresMarcadores(m)
4:     for each g ∈ 𝒢 do
5:         e ← g.estado
6:         mId ← m.mavID
7:         new ← e = NUEVA and g.mavOrigenId = mId
8:         discarded ← e = DESCARTADA and g.mavOrigenId = mId
9:         inaccessible ← e = INACCESIBLE and g.mavOrigenId = mId
10:        assigned ← e = ASIGNADA and g.mavAsidnadoId = mId
11:        explored ← e = EXPLORADA and g.mavAsignadoId = mId
12:        if new or assigned or discarded or inaccessible or explored then
13:            v ← Crear marcador de visualización con la posición de g
14:            v.color ← 𝒱_color.getColor(g.estado)
15:            𝒱.add(v)
16:    Publicar 𝒱 en el tema /ard/cell_goals

17: function GenerarColoresMarcadores(m)
18:    𝒱_color ← ∅                                      ▷ Colores para los estados de las metas
19:    𝒱.add(m.color, NUEVA)
20:    𝒱.add(amarillo, ASIGNADA)
21:    𝒱.add(gris, DESCARTADA)
22:    𝒱.add(guinda, INACCESIBLE)
23:    𝒱.add(m.color con transparencia, EXPLORADA)
24:    return 𝒱_color
```

que han recibido de otros MAVs a través del coordinador, es necesario seleccionar un conjunto de ellas para publicar. Este conjunto de metas se compone básicamente de las metas que los MAV crearon o les fueron asignadas (ver Algoritmo 6.1). Para identificar las metas creadas por los MAVs, se utiliza la propiedad `mavOrigenId`. Las metas cuyo estado es nueva, descartada o inaccesible son publicadas por el MAV que las generó (ver línea 7 del Algoritmo 6.1). Por otra parte, las metas, cuyo estado es asignada o explorada, son publicadas por el MAV que las tiene asignadas (ver línea 10 del Algoritmo 6.1).

Los marcadores, que representan metas de exploración, tienen diferentes colores para distinguir su estado (ver línea 17 del Algoritmo 6.1). El color de las metas nuevas es el mismo que el de los MAVs. Este color es adquirido de los marcadores de visualización publicados por el nodo `/ard/robot_pose_publisher` (ver Sección 6.2.4). El color de las metas asignadas es amarillo, mientras que los de las metas descartadas e innaccesibles son gris y guinda, respectivamente. Finalmente, para las metas exploradas se utiliza el mismo color del MAV, pero con transparencia.

6.6. Limpieza del mapa

La limpieza de las celdas producidas por el problema de mapeo mutuo, se realiza en todos los componentes del sistema que construyen un mapa. Estos componentes pertenecen a los paquetes `mv_octomap`, `mv_explorer` y `planning` de MoveIt!. En la Sección 6.6.1 se presenta la definición del mensaje utilizado para limpiar puntos del mapa. En la Sección 6.6.2 se describe el funcionamiento del componente de limpieza y en la Sección 6.6.3 se presenta la forma de integrar dicho componente de limpieza a los componentes de generación de mapas.

6.6.1. Puntos del mapa a limpiar

Los puntos del mapa a limpiar fueron definidos como mensajes independientes de las metas. Esta implementación elimina la dependencia entre el componente de limpieza y las metas de exploración y permite utilizar dicho componente con diferentes puntos.

El mensaje de los puntos de limpieza consta de la posición a limpiar, una distancia radial de limpieza, el color con el que marcarán las **celdas ruido** y un identificador para dichas celdas (ver Archivo 6.6). La posición a limpiar, `pose`, representa la posición de un punto en 3D. La distancia de limpieza, `radius`, es la distancia a la cuál se buscarán celdas ocupadas, las cuales se insertarán en la pila del algoritmo de relleno de inundación (ver Sección 3.3.5). El color para marcar las celdas ruido, `outputColor`, es el color que se asignará a todas las celdas detectadas, mediante el algoritmo de relleno por inundación en la posición especificada. El identificador de estas celdas, `outputId`, sirve para identificar a qué grupo de celdas ruido pertenecen, cada grupo le corresponde a un MAV mapeado.

Archivo 6.6: Mensaje de punto sujeto a limpieza, definido en el archivo ClearPoint.msg.

```
1 geometry_msgs/Pose pose
2 float64 radius
3 std_msgs/ColorRGBA outputColor
4 uint32 outputId
```

6.6.2. Componente de limpieza

El componente de limpieza es una clase en C++, `OctomapCleaner`, la cual aplica el proceso de limpieza en cada lista de puntos recibida (ver Sección 5.4). Para este proceso, no basta con utilizar las celdas que son publicadas por los nodos que construyen mapas, sino que es necesario acceder a los objetos de C++ donde se almacenan dichos mapas. Por esta razón, la clase `OctomapCleaner` contiene una referencia al mapa a limpiar y a un *mutex* que es bloqueado al momento de eliminar las celdas ruido.

Para adquirir los puntos a limpiar, la clase `OctomapCleaner` se subscribe al tema `/clear_points`, donde recibe mensajes del tipo `ClearPoint`. Estos mensajes son publicados por el **proceso de coordinación** implementado en el nodo `/mv_explorer_coordinator`. A pesar de que los mapas limpiados pertenecen a diferentes nodos, el tema donde se publican dichos puntos es global, lo cual favorece el diseño desacoplado. Los nodos que se subscriben al tema `/clear_points` son: `/ard/move_group`, `/mv_explorer_ard` y `/mv_octomap`. Los dos primeros nodos están presentes en todos los MAVs con su correspondiente espacio de nombres `ard` y el nodo `/mv_octomap`, corresponde al mapa global del coordinador (ver Figura 6.12).

Con el propósito de limpiar el mapa usado para planificación de movimiento en MoveIt!, se implementó el componente `OctomapMonitorCleaner`. Dicho componente es una clase en C++ que hereda de `OctomapCleaner`. `OctomapMonitorCleaner` se diferencia de su clase padre en el método utilizado para bloquear las actualizaciones del mapa. Mientras que `OctomapCleaner` hace uso de *mutex* perteneciente al paquete `ecl` de ROS, `OctomapMonitorCleaner` invoca al método de bloqueo del `octree`, el cual contiene el mapa en MoveIt!.

Las celdas ruido, eliminadas del mapa, proporcionan información de los lugares donde ocurrió el problema de mapeo mutuo. Una vez que el mapa es limpiado, estas celdas son almacenadas en forma de marcadores de visualización. Dichos marcadores son publicados con el color enviado por el proceso de coordinación que se ejecuta en el nodo `/mv_explorer_coordinator`.

6.6.3. Integración del componente de limpieza

El componente de limpieza requiere ser integrado a los diferentes lugares del sistema donde se generan mapas. Estos lugares son el mapa global en el coordinador, los mapas

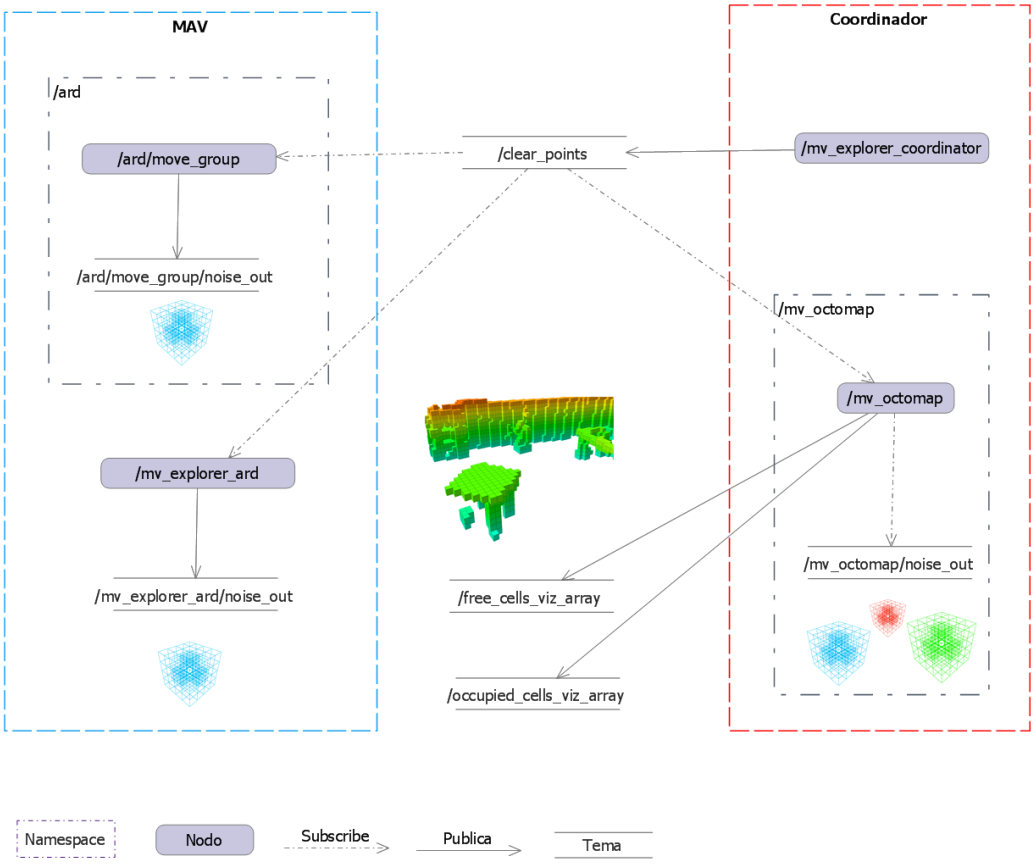


Figura 6.12: Nodos utilizados en el proceso de limpieza del mapa.

locales en los MAVs y los mapas utilizados para la planificación de movimiento en MoveIt!. En todos los casos se requiere agregar la referencia al paquete `mv_octomap`.

Mapa global

Dado que se requiere acceso a la variable donde se almacena el mapa, se implementó la clase `MvOctomapServer`, la cuál hereda de la clase `TrackingOctomapServer` del paquete `octomap_server`. La clase `TrackingOctomapServer` es una implementación del servidor de Octomap, por lo tanto almacena una variable con el mapa generado. Por esta razón, el nodo encargado de mantener el mapa del coordinador es el nodo `/mv_octomap` creado en la clase `MvOctomapServer`. Los temas publicados por dicho nodo son: `/free_cells_viz_array` y `/occupied_cells_viz_array`. En estos temas se publican arreglos de marcadores de visualización de RViz, los cuales representan las celdas libres y celdas ocupadas del mapa, respectivamente.

Para la integración del componente de limpieza a la clase `MvOctomapServer`, se

agregó un atributo con una instancia de la clase `OctomapCleaner`. Esta instancia fue creada con la referencia al mapa global y la referencia al *mutex* encargado de bloquear el mapa cada vez que es modificado. Para bloquear el mapa basta con implementar el método virtual `insertScan()` y llamar a la función `lock()` del *mutex*, antes de invocar al método `insertScan()` de la clase padre. Al terminar esta acción, se debe llamar a la función `unlock()` del *mutex*.

Mapa local

El mapa local de los MAVs es almacenado en el componente `GoalGenerator`. Ya que este componente posee una implementación de `OctomapServer` (ver Sección 6.5.3), el proceso de integración es similar al del mapa global, i.e., agregar un atributo a la clase `GoalGenerator` con una instancia de `OctomapCleaner`. Al igual que en `MvOctomapServer`, es necesario definir un *mutex* que bloquee las operaciones sobre el mapa para manejar la concurrencia.

Mapa de MoveIt

Para colocar el componente de limpieza en el mapa utilizado para planificar movimiento, es necesario modificar el paquete `perception` de MoveIt!. Dentro de este paquete, se encuentra la clase `occupancy_map_monitor`, responsable de almacenar el mapa en MoveIt!. En esta clase se agregó un atributo del tipo `OctomapMonitorCleaner`, el cual hereda de la clase `OctomapCleaner`. Mediante este atributo se puede aplicar el mismo proceso que con los mapas anteriores, ya que `occupancy_map_monitor` implementa los mecanismos necesarios de bloqueo del mapa, los cuales son aprovechados por `OctomapMonitorCleaner`.

6.7. Discusion

En esta sección se discute brevemente la implementación del sistema propuesto. Dicha implementación se realizó en ROS y se utilizaron componentes existentes de planificación de movimiento, exploración y mapeo. En la implementación realizada, se creó un paquete de MoveIt! utilizando el archivo donde está definido el modelo del MAV simulado. Dicho paquete contiene los archivos necesarios para que un MAV pueda realizar planificación de movimiento. Estos archivos fueron modificados para permitir su uso con múltiples MAVs.

Una forma alternativa de realizar la planificación de movimiento y ejecución de trayectorias con los MAVs consiste en crear un paquete de configuración de MoveIt! para todos ellos. Para crear este paquete se requiere un modelo con todos los MAVs para los cuales se planificará, i.e., un archivo sdf donde se coloque múltiples veces el modelo de un MAV. Utilizando este modelo, el asistente para la creación de paquetes de configuración de MoveIt! consideraría a los MAVs como articulaciones de un solo

robot.

En caso de que se utilizara un paquete de MoveIt! configurado para todos los MAVs, el proceso de limpieza de las celdas del mapa, utilizando el algoritmo de relleno por inundación, no sería necesario. Esta situación se debe a que MoveIt! cuenta con sus propios procesos para limpiar las nubes de puntos de los elementos del robot. Dichas nubes limpiadas son publicadas por el grupo de movimiento y con ellas sería posible construir un mapa libre de celdas ruido.

En este caso, el nodo del grupo de movimiento debería ejecutarse en un elemento al cual los MAVs tengan acceso común. Este elemento podría ser el coordinador, sin embargo aumentaría la cantidad de mensajes que deberían intercambiar los MAVs y dicho coordinador. Este diseño además incrementaría el acoplamiento del sistema, ya que la planificación de movimiento dejaría de residir en los MAVs.

Capítulo 7

Pruebas y resultados

En este capítulo se presentan las pruebas realizadas al sistema de exploración multi-MAV. En la Sección 7.1 se describe el entorno en que se realizaron las pruebas. En la Sección 7.2 se presentan los resultados de los experimentos realizados. Finalmente, en la Sección 7.3 se discuten los resultados de la exploración.

7.1. Entorno de pruebas

Para la realización de las pruebas se utilizó la plataforma ROS y el simulador Gazebo. Las pruebas se realizaron utilizando un modelo de Gazebo, el cuál tiene la estructura de un edificio.

7.1.1. Equipo

El equipo de pruebas es una computadora de escritorio con el sistema operativo Ubuntu 14.04. A este equipo se le instaló la distribución ROS Indigo que utiliza la versión 2.2 de Gazebo. Sus características son: procesador AMD Phenom II X6 1100T a 3.3 Ghz, 8 GB de RAM DDR 3, disco de estado sólido Samsung EVO 850 de 500 GB y tarjeta de video NVIDIA GeForce GTX 750 TI.

7.1.2. Escenario

El escenario seleccionado para las pruebas es Willow Garage, el cuál es uno de los modelos instalados por defecto con Gazebo 2.2 de ROS Indigo. No se intentó explorar completamente este modelo, en cambio se seleccionó únicamente el área central y se delimitó por paredes de tipo `nist_maze_wall_240` (ver Figura 7.1). Dado que el modelo seleccionado para las pruebas es de un entorno abierto, se estableció un límite de tres metros de altura para las metas generadas. También se estableció un límite inferior de un metro de altura para que los MAVs no se desplacen a metas muy cercanas al piso que los lleven a estrellarse.

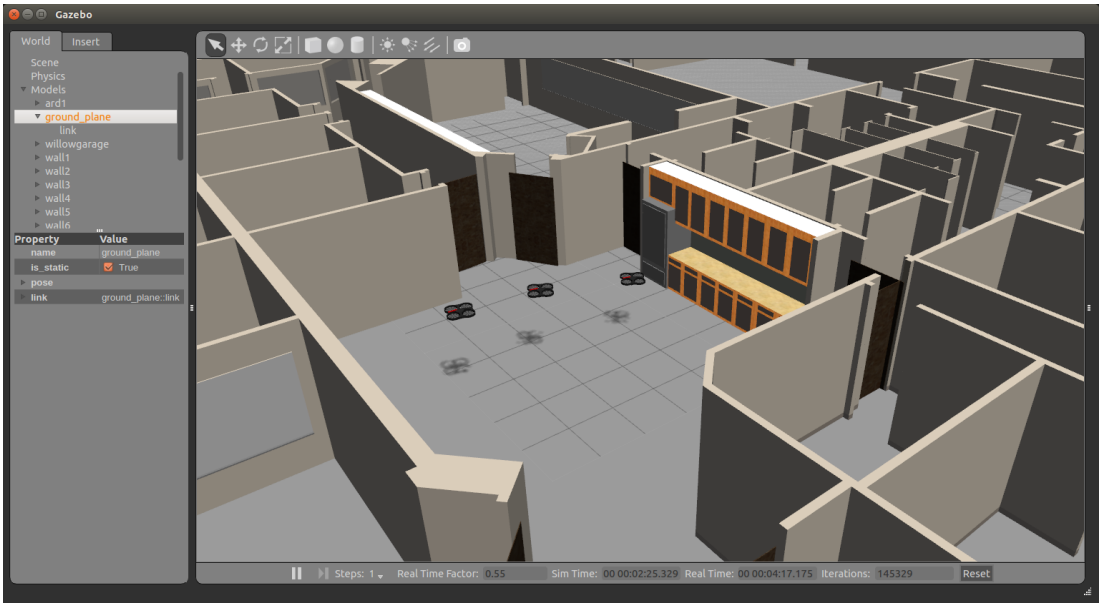


Figura 7.1: Escenario de pruebas utilizando el modelo Willow Garage

7.2. Experimentos de exploración multi-MAV

En esta sección se describen los experimentos de exploración multi-MAV. En la Sección 7.2.1 se describen los criterios utilizados para evaluar los experimentos y en la Sección 7.2.2 se presentan los resultados de los mismos.

7.2.1. Descripción de experimentos

Para realizar la evaluación del sistema propuesto, se midió el tiempo en que los MAVs simulados finalizan la exploración del escenario de pruebas, utilizando conjuntos de uno, dos y tres MAVs. Por cada conjunto de MAVs se realizaron 30 ejecuciones y el máximo rango de sensado de cada MAV fue establecido en 4 metros. El número de metas que los MAVs exploran en su región asignada es dos (ver Figura 7.2). El tiempo fue medido desde que el coordinador ejecuta la fase de inicialización hasta que la exploración se da por terminada, como consecuencia de falta de metas para explorar.

Los tiempos fueron medidos utilizando la función `ros::Time`, la cual utiliza el tiempo del simulador. El uso de este tiempo proporciona medidas más confiables con respecto a las mediciones, utilizando el tiempo de la máquina `ros::WallTime`. Esta situación se debe a que en el tiempo de simulación ya se están considerando las posibles variaciones en la velocidad a la que se ejecuta dicha simulación.

Además del tiempo, también se contaron el número de celdas ocupadas y libres en el mapa global. La resolución del mapa, i.e., tamaño de celda es de 0.1 metros

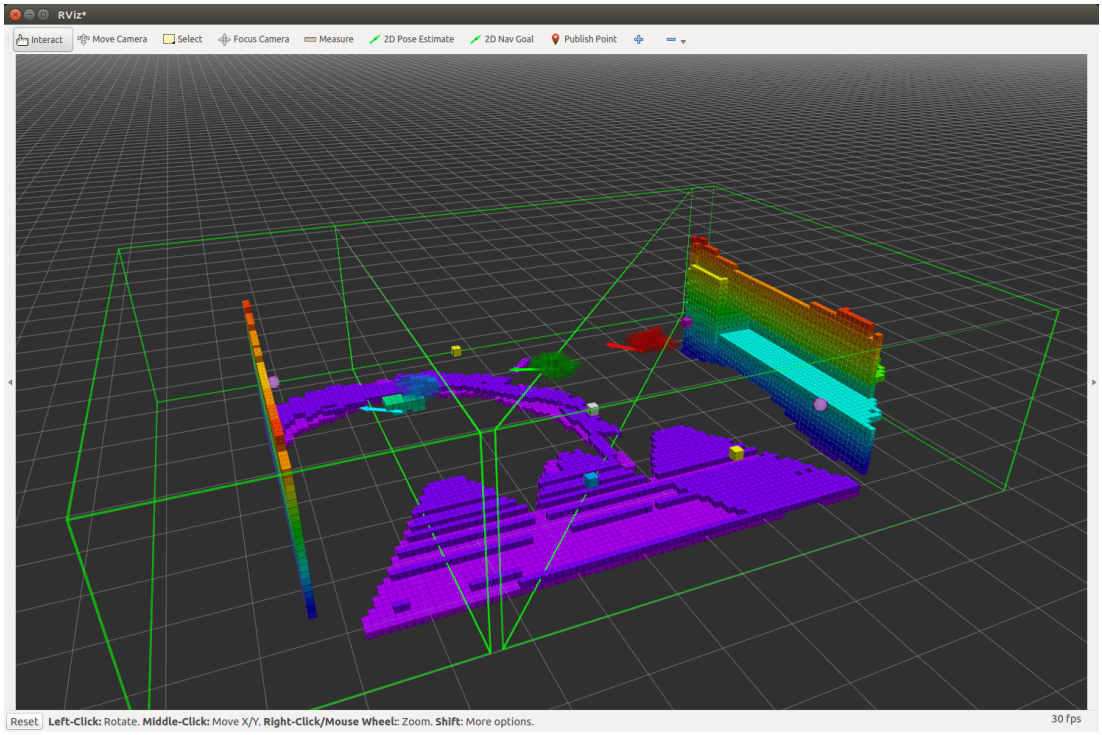


Figura 7.2: Regiones asignadas de los MAVs

cúbicos. Esta medición se llevó a cabo utilizando la función `waitForMessage` en el coordinador. Dicha función espera por un mensaje en los temas en que son publicados los marcadores de visualización del mapa. De manera similar, se midió el número de celdas producidas como consecuencia del problema de mapeo mutuo. La medición se realizó cada 10 segundos utilizando el tiempo del simulador. Se seleccionó este tiempo de medición para no imponer una carga de procesamiento excesiva en el equipo de cómputo.

7.2.2. Resultados de los experimentos

A continuación se presentan los resultados de la exploración multi-MAV para conjuntos de uno, dos y tres MAVs. El número promedio de celdas ocupadas, libres y ruido es presentado en esta sección con respecto al tiempo de exploración. Ya que los experimentos de exploración tienen una duración distinta, independientemente del número de MAVs, el número promedio de celdas para cada tiempo t fue calculado utilizando los datos disponibles en dicho t. Este cálculo implica que en aquellos tiempos, cercanos a la duración máxima observada, el promedio de celdas haya sido obtenido utilizando un menor número de datos dividido entre los 30 experimentos.

Exploración con un MAV

Cuando se explora utilizando un MAV, la duración promedio de la exploración es de 7 minutos y 12 segundos. El número promedio de celdas ocupadas es de 185110.68 y el número promedio de celdas libres es de 9817.68. La exploración del entorno dura como máximo 10 minutos (ver Figura 7.3).

La exploración con un MAV produce un mapa visualmente similar al modelo de Willow Garage (ver Figura 7.4). Esta similitud significa que los obstáculos presentes en el modelo también están presentes en el mapa generado. Adicionalmente se observa que las metas exploradas están distribuidas uniformemente en el mapa.

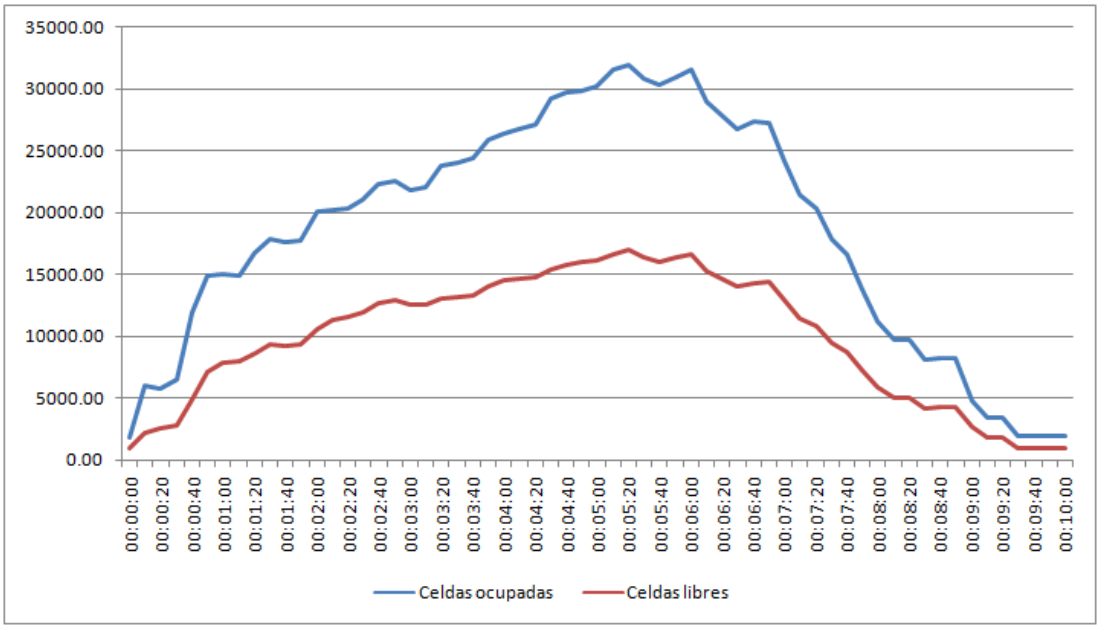


Figura 7.3: Resultados de la exploración con un MAV

Exploración con dos MAVs

En la exploración con dos MAVs se tiene un tiempo promedio de 4 minutos y 15 segundos. El número promedio de celdas ocupadas es de 17164.43 y el número promedio de celdas libres es de 737.30. A diferencia de la exploración con un MAV, se tienen celdas ruido, cuyo número promedio es de 147.94. La exploración del entorno tiene una duración máxima de aproximadamente 4 minutos (ver Figura 7.5).

Durante los experimentos con dos MAVs se observó que, en una ocasión, el grupo de movimiento no respondió a la solicitud de un plan. Ya que la respuesta a la solicitud nunca llegó al MAV este se quedó esperándola, incluso cuando su compañero ya había terminado de explorar sus metas. Esta situación ocasionó que se tuviera que detener

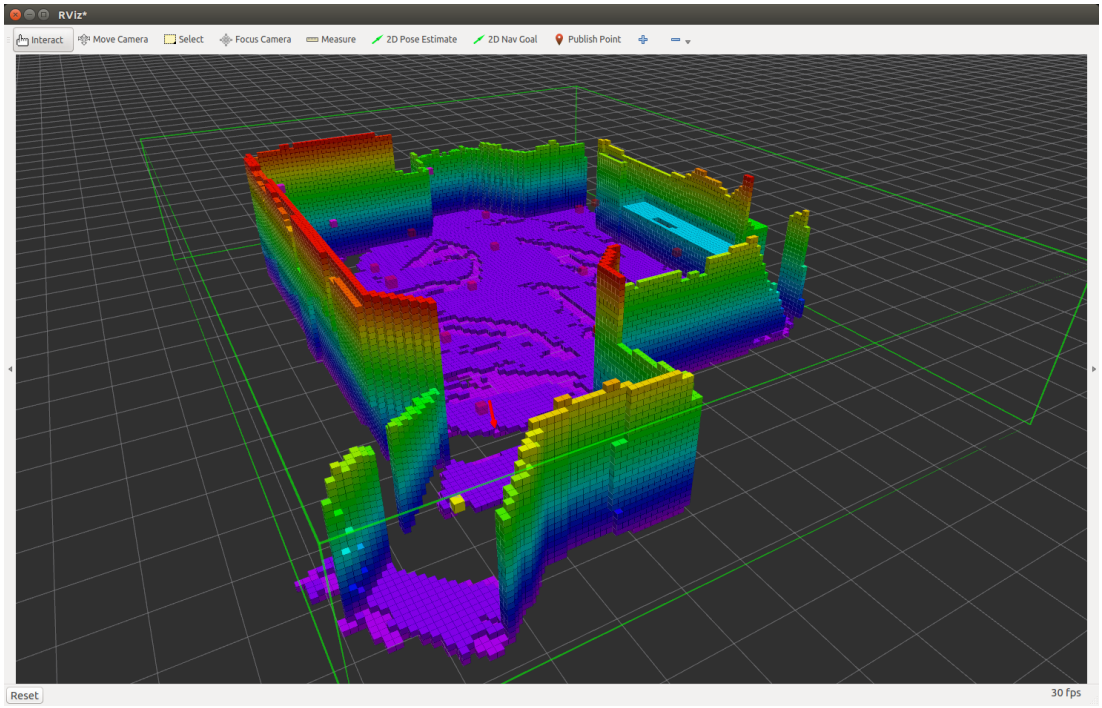


Figura 7.4: Eliminación de una parte del mapa al limpiar celdas ruido.

la exploración manualmente. Este problema se repitió con mayor frecuencia al pasar a tres MAVs.

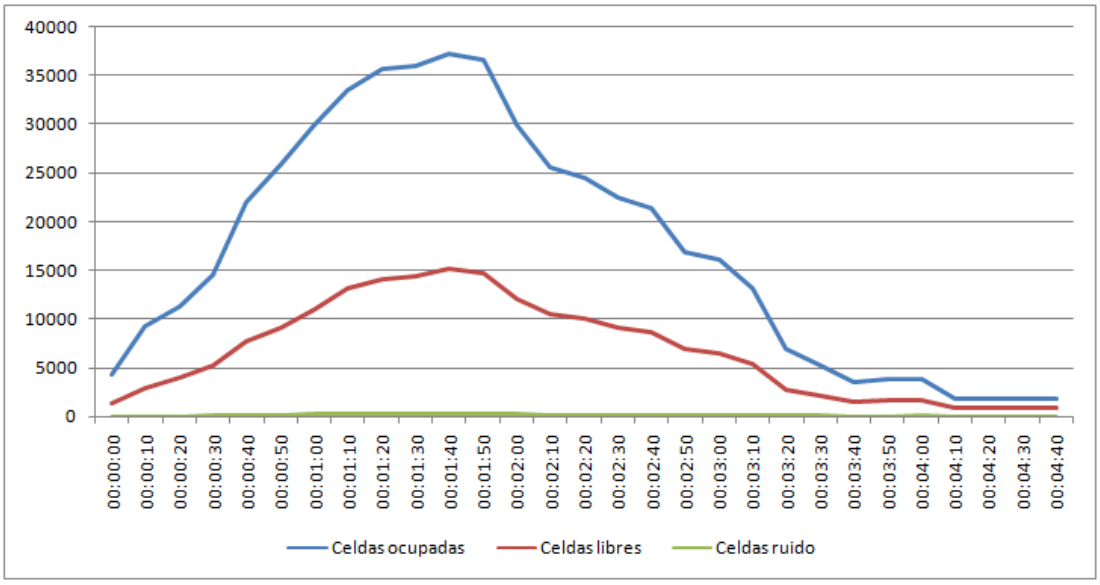


Figura 7.5: Resultados de la exploración con dos MAVs

Exploración con tres MAVs

En la exploración con tres MAVs la duración promedio de la exploración es de 2 minutos y 14 segundos. El número promedio de celdas ocupadas es de 12678.1 y el número de celdas libres es de 4470.25. Puesto que se explora con tres MAVs existe el problema de mapeo mutuo y por lo tanto se presentan celdas ruido, las cuales en promedio son 154.78. La duración máxima de la exploración es de aproximadamente 2 minutos (ver Figura 7.6)

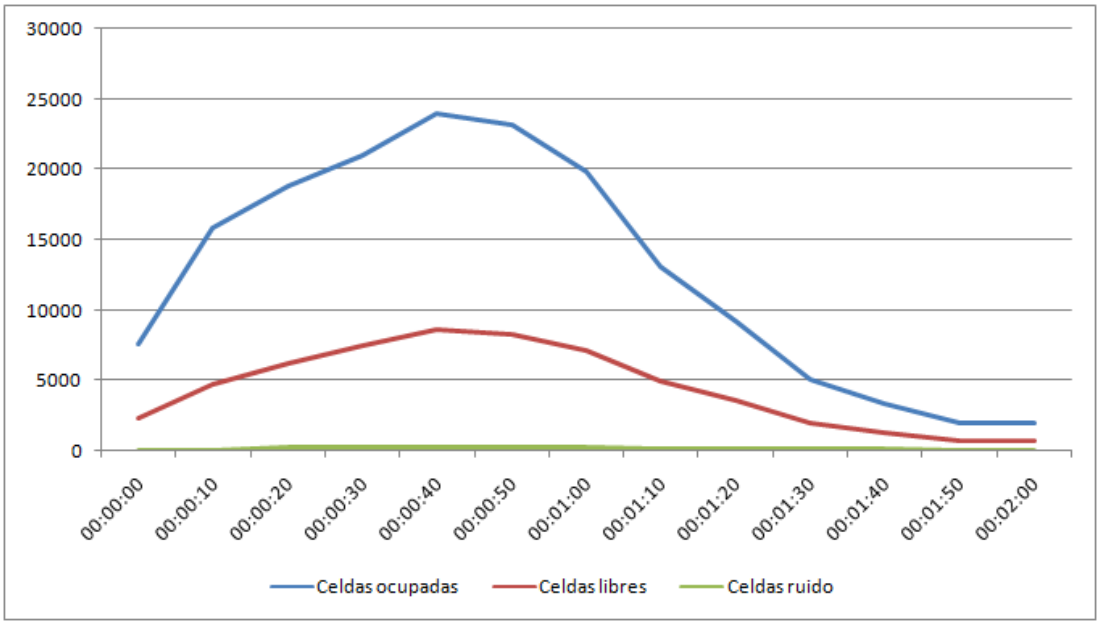


Figura 7.6: Resultados de la exploración con tres MAVs

El problema de mapeo mutuo se vuelve más importante conforme se incrementa el número de MAVs. Cuando las metas que exploran los MAVs se encuentran demasiado cercanas a las celdas ocupadas del mapa, el componente de limpieza elimina partes de dicho mapa (ver Figura 7.7). En las pruebas realizadas se observó que este problema ocurre con mayor frecuencia cuando se utilizan tres MAVs en comparación con el uso de dos MAVs.

Otro problema que se observó, en la exploración con tres MAVs, es que llegaron a colisionar con los obstáculos del entorno. Este problema se debe a que los MAVs se desplazan hacia zonas que no han sido detectadas como ocupadas. Esta situación ocurre cuando se genera un plan de movimiento que pasa por una zona cuyas celdas son desconocidas, por lo tanto no se sabe si hay obstáculos. El problema podría solventarse si el obstáculo apareciera en el rango de sensado de los MAVs durante su desplazamiento a las metas.

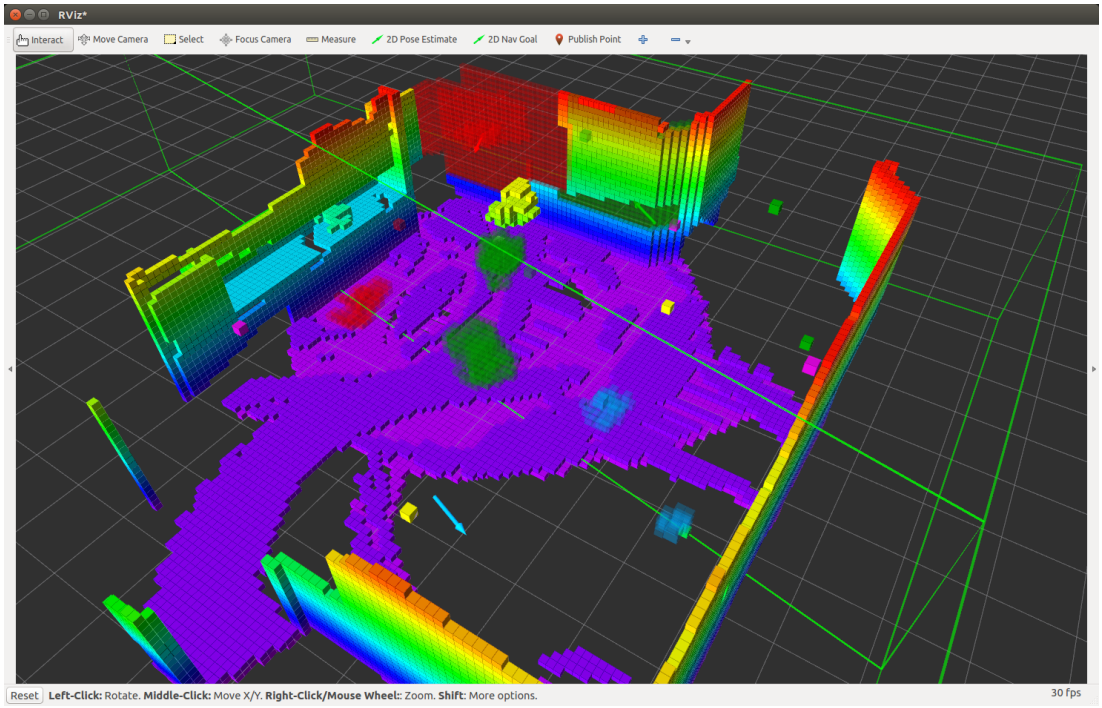


Figura 7.7: Eliminación de una parte del mapa al limpiar celdas ruido.

7.3. Discusión

En los experimentos realizados se observó que, al incrementar la cantidad de MAVs, la exploración termina en un menor tiempo. Sin embargo, el número promedio de celdas ocupadas también disminuye (ver Figura 7.8). Este resultado es contrario a lo que podría esperarse al incrementar el número de MAVs, ya que es deseable que se explore la misma cantidad de espacio en un menor tiempo.

El resultado de los experimentos muestra que el tiempo máximo de exploración ocurre con un MAV, sin embargo también es aquel que obtiene el mayor número de celdas ocupadas y libres. i.e., en promedio con un MAV se crea el mapa más completo del entorno (ver Tabla 7.1). Esta situación es una consecuencia directa de la realización de los experimentos mediante una simulación en un solo equipo.

MAVs	**Duración de la exploración**	**Celdas ocupadas**	**Celdas libres**	**Celdas ruido**
1	00:07:12	18511.68	9817.68	0
2	00:04:15	17164.43	6737.30	147.94
3	00:02:14	12678.1	4470.25	154.78

Tabla 7.1: Resultados promedio de la exploración multi-MAV

Debido a que la simulación se ejecuta en un solo equipo de cómputo, al incre-

mentar el número de MAVs la capacidad para procesar los cambios en el mapa se ve reducida. Esta reducción implica que se detectan menos cambios en el mapa. Un menor número de cambios en el mapa ocasiona que haya menos celdas para evaluar, por lo tanto la cantidad de metas de exploración que puede generarse también disminuye. El número de metas influye tanto en la duración de la exploración, como en la cantidad de celdas exploradas.

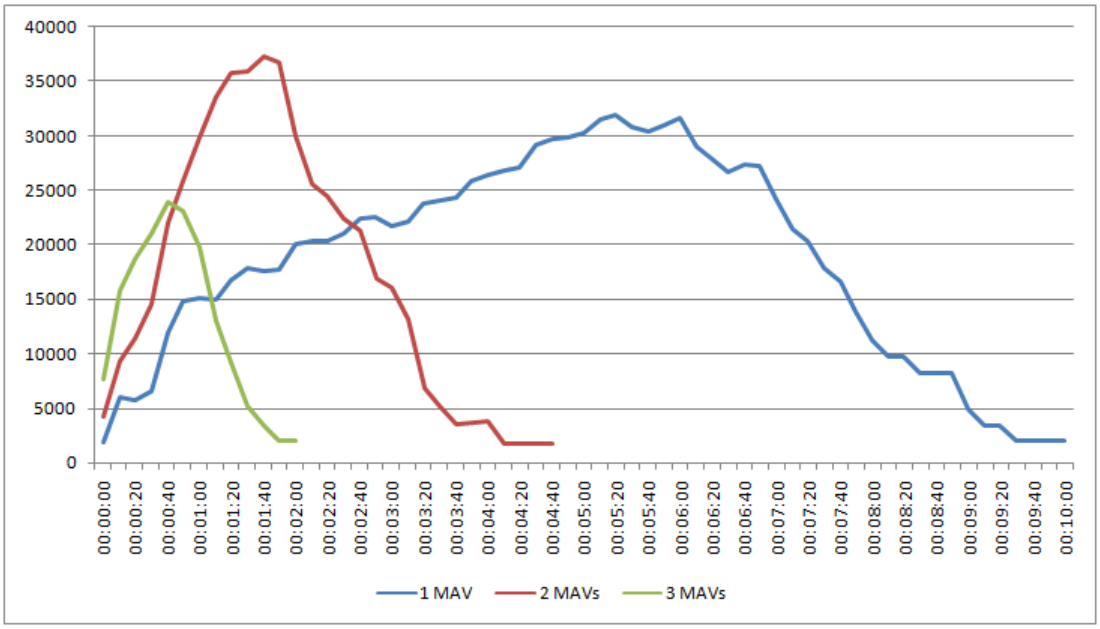


Figura 7.8: Celdas ocupadas para uno, dos y tres MAVs

Capítulo 8

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado. En la Sección 8.1 se hace una breve recapitulación del problema tratado. En la Sección 8.2 se describen las contribuciones y limitaciones principales del sistema de exploración multi-MAV propuesto. Finalmente, en la Sección 8.3 se presentan los trabajos futuros que pueden derivar de la presente tesis.

8.1. Recapitulación del problema

En el presente trabajo de tesis se trató el problema de exploración multi-MAV. Este problema consiste en utilizar más de un MAV para explorar un entorno desconocido, sin instrucciones de un operador humano. El resultado de esta exploración es un mapa de ocupación 3D, el cuál es de utilidad para conocer la estructura del entorno.

A pesar de que, en el campo de la robótica, la exploración es un problema ampliamente estudiado, la mayoría de las propuestas se enfocan en robots terrestres. Aún dentro de las propuestas de exploración con MAVs, el uso de múltiples de ellos es algo poco común. En la única propuesta multi-MAV, encontrada en la literatura, se hace la simplificación del entorno a 2.5D. i.e., se considera un entorno 2D a una altura fija. Esta simplificación limita las capacidades de exploración de los MAVs.

Cuando se realiza exploración multi-MAV, existe una serie de problemas. El primero de ellos consiste en que los MAVs pueden colisionar entre sí, con consecuencias que impedirían continuar la exploración. Este caso es contrario a lo que ocurre con los robots terrestres que no tienen mayor problema al colisionar. El segundo problema es que los MAVs exploren una misma zona múltiples veces, por lo tanto es necesario introducir mecanismos de coordinación. Por último, la exploración robótica comúnmente se realiza en entornos estáticos, lo cual se debe principalmente a la naturaleza de los algoritmos de localización y mapeo. Sin embargo, un inconveniente extra es que los MAVs se pueden mapear mutuamente, introduciendo con ello ruido en el mapa. Este ruido se presenta en forma de celdas ocupadas que en realidad están

libres.

8.2. Conclusiones

En este trabajo de tesis se partió de la hipótesis siguiente: *es posible reducir el tiempo de exploración 3D en interiores mediante el uso de múltiples MAVs, en comparación con el uso de un solo MAV*. A partir de los resultados de los experimentos realizados, se puede concluir que la hipótesis es falsa, i.e., cuando se incrementa el número de MAVs la exploración no mejora. Sin embargo, dichos resultados pueden deberse a las capacidades de procesamiento del equipo de cómputo en el cual se realizó la simulación (ver Sección 7.3). Por este motivo, existen razones para creer que los resultados pueden mejorar si se utilizan más equipos de cómputo.

En la Sección 8.2.1 se describen las contribuciones de la propuesta de exploración multi-MAV y en la Sección 8.2.2 se presentan sus limitaciones principales.

8.2.1. Contribuciones

El presente trabajo de tesis tiene dos contribuciones principales: a) una estrategia de exploración multi-MAV utilizando coordinación centralizada, que no está ligada a una implementación específica y b) un sistema de exploración multi-MAV que implementa dicha estrategia de exploración.

La estrategia de exploración propuesta permite a múltiples MAVs explorar un entorno 3D desconocido. Esta estrategia define lugares a explorar llamados metas de exploración, las cuales son creadas a partir de grupos de celdas frontera. Dicha estrategia reduce las posibles colisiones entre MAVs, gracias al uso de un coordinador central que asigna regiones independientes a explorar de un diagrama de Voronoi. El uso de estas regiones, junto con las metas de exploración, reduce los traslapes de zonas de ganancia de información y, por lo tanto, la cantidad de lugares que los MAVs deben explorar.

La segunda contribución del presente trabajo de tesis es un sistema de exploración multi-MAV implementado en la plataforma de desarrollo robótico ROS. Este sistema además combina componentes existentes de planificación de movimiento, ejecución de trayectorias y mapeo. En el aspecto técnico, el sistema muestra que es posible utilizar el *plugin* MoveIt! con múltiples robots, aún cuando dicho *plugin* no fue diseñado específicamente para ello. Por último, gracias a que el código del sistema está escrito en una plataforma de código abierto, sus componentes pueden ser analizados y reutilizados para favorecer el desarrollo de otros sistemas de exploración.

8.2.2. Limitaciones

La limitación más importante del presente trabajo es que fue probado en un entorno simulado. Este entorno tiene como consecuencia que las dinámicas de los MAVs no son exactamente las mismas que en un entorno real. Por ejemplo, un problema en un entorno real es que los MAVs son más sensibles a colisiones con los obstáculos del entorno, las cuales pueden llegar a ocurrir debido a que el componente de planificación de movimiento genera rutas en un espacio de trabajo que puede incluir espacio inexplorado.

La planificación de movimiento provista por MoveIt! es responsabilidad de la biblioteca OMPL. Dicha planificación está sujeta a las observaciones del entorno. La principal consecuencia de este hecho es que cuando un MAV se desplaza de un lugar a otro puede chocar con obstáculos que no han sido mapeados. Este no es el caso de los obstáculos que son vistos de frente por el sensor Kinect, porque se pueden utilizar mecanismos de replaneción. Sin embargo, cuando el plan de movimiento requiere que el MAV se desplace de lado o hacia atrás no hay forma de detectar los obstáculos. Para solventar este aspecto se requiere de un sensor con cobertura de 360° o diseñar un planificador que considere la dirección del sensor.

Una limitación de nivel técnico es que el *plugin* de MoveIt! en ocasiones no responde la solicitud del plan de movimiento. Este problema ha sido discutido anteriormente en la comunidad de MoveIt! por otros usuarios. A pesar de que se implementaron las sugerencias de los foros como son: ejecutar en un hilo independiente el proceso del servidor de acciones y utilizar `ros queues` independientes, el problema permanece. Las `ros queues` son colas donde se almacenan las *callbacks* utilizadas en el modelo publicador-suscriptor.

El sistema de exploración desarrollado considera que el coordinador es una estación de cómputo terrestre. Esta limitación se debe principalmente a que puede ser necesario implementar un proceso de fusión de metas en el coordinador. Este proceso puede ser costoso en términos computacionales, por lo tanto es complicado ejecutarlo en un MAV que actue como coordinador, aunque se requieren pruebas en MAVs reales para determinar su viabilidad.

Por otra parte, en este sistema se hace la suposición de que el mapeo y la localización en el entorno son proporcionadas por un componente SLAM existente. Este componente es el encargado de dar las transformaciones de las nubes de puntos con respecto a un marco de referencia común. Este componente no se utiliza en el sistema, ya que la simulación proporciona dichas transformaciones.

dado que los MAVs son simulados, es suficiente con aplicar un torque específico a los motores para mantenerlo estático en el aire. En una implementación real sería necesario utilizar un sensor de altura para determinar cuánta velocidad se debe aplicar

a los motores para conservar al MAV estático.

Otra limitación es que no se incluyen mecanismos de tolerancia a fallas en la exploración ni en las comunicaciones. Esta limitación implica que el sistema no es capaz de recuperarse de errores ocasionados por colisiones de los MAVs, ni por fallas en las comunicaciones. Los mecanismos de recuperación de las comunicaciones en un entorno real correrían a cargo de la plataforma ROS.

Finalmente, en el sistema propuesto se asumen que el rango de comunicaciones es ilimitado, lo cual no ocurre en entornos reales. En caso de que se utilizara el sistema en un entorno real, su alcance quedaría limitado al de la señal inalámbrica entre el coordinador y los MAVs.

8.3. Trabajo futuro

Con el propósito de continuar probando el sistema de exploración propuesto, utilizando una simulación, se pueden agregar más equipos de cómputo. Una forma de incluir más equipos de cómputo consiste en utilizar un equipo para mantener la simulación y otros para actuar como coordinador y MAVs. En dicha arquitectura, los equipos que actuen como MAVs y coordinador deberían adquirir la información de las nubes de puntos publicadas por el equipo que ejecute la simulación. Gracias a la estructura de ROS, este proceso se puede realizar de manera transparente, mediante la configuración del núcleo `roscore`.

Entre los trabajos futuros destaca el de implementar el sistema de exploración en MAVs físicos. Para lograr este propósito, es necesario realizar la instalación de ROS en los MAVs y colocarles un paquete SLAM, e.g., RGBD SLAM V2. En caso de que se utilice un MAV AR.Drone 2.0, el sistema de exploración ya podría enviarle comandos de vuelo, debido a que hace uso del paquete `Autonomy` que proporciona un *driver* de dicho MAV. En caso de utilizar otro MAV, sería necesario implementar un nuevo componente de ejecución de trayectorias.

Los algoritmos SLAM multi-robot, aplicados a MAVs, pueden eliminar la necesidad del componente de limpieza. En la literatura, estos algoritmos son utilizados en robots terrestres. Aunque su implementación en MAVs puede ser compleja, debido a las capacidades de procesamiento de estos últimos, su investigación y aplicación es de interés.

Con el propósito de mejorar la experiencia de uso para un ser humano, que se encargue de dar seguimiento a la exploración, se pueden agregar funciones al sistema de exploración. Entre estas funciones se encuentra la consulta de estadísticas de exploración tales como: tiempo de exploración, número de metas generadas, número de celdas mapeadas por MAV, número de celdas totales y número de celdas ruido. Adicionalmente, en determinadas situaciones, podría ser deseable interrumpir la ex-

ploración, ordenar a los MAVs regresar a la base e incluso generar metas manualmente.

Para agregar estas funciones al sistema de exploración, se puede crear una interfaz de usuario o un *plugin* para RViz. En el primer caso, sería necesario utilizar la biblioteca de RViz para acceder a las funciones de visualización, mediante una aplicación del *framework* QT. En el segundo caso, las funciones del sistema de exploración recaerían en un *plugin* de RViz. Mediante dicho *plugin* podrían aprovecharse todas las funciones existentes de la interfaz de RViz como son agregar elementos de visualización de otros temas y seleccionar el tipo de cámara de RViz.

Apéndice A

Herramientas de desarrollo

En este apéndice se describen las herramientas utilizadas para la implementación del sistema propuesto en el presente trabajo de tesis. En la Sección A.1 se presenta el *framework* ROS sobre el cuál se ejecutan todos los componentes del sistema. En la Sección A.2 se describe la herramienta de visualización utilizada para observar el mapa, las **metas de exploración** y las regiones del diagrama de Voronoi. En la Sección A.3 se presenta el simulador Gazebo y algunos de sus elementos principales.

A.1. ROS

En esta sección se presenta el *framework* ROS utilizado en el desarrollo del presente trabajo de tesis. En la Sección A.1.1 se realiza una introducción a ROS, describiendo su funcionamiento general y orígenes. En la Sección A.1.2 se presentan los elementos principales de ROS, entre los cuales se encuentran los nodos y temas. Por último, en la Sección A.1.3, se describe brevemente el proceso de desarrollo de aplicaciones en ROS y la forma de ejecutar una aplicación de ROS.

A.1.1. Introducción a ROS

El sistema operativo robótico (ROS, por sus siglas en inglés[1]) es un *framework* para el desarrollo de software robótico, distribuido bajo la licencia *open source*, BSD (por sus siglas en inglés[2]). ROS fue desarrollado originalmente en 2007 por el Laboratorio de Inteligencia Artificial de la Universidad de Stanford. Actualmente es mantenido por el Instituto de Investigación Robótica Willow Garage. ROS provee un conjunto de bibliotecas y herramientas que ofrecen abstracción de hardware, controladores de dispositivos y paso de mensajes. Las capacidades de ROS pueden ser extendidas mediante la instalación de paquetes compatibles. ROS puede funcionar en modo distribuido, por lo tanto puede ser utilizado para comunicar nodos en diferentes máquinas.

[1] *Robot Operating System*
[2] *Berkeley Software Distribution*

Una vez que ROS está configurado en este modo, es capaz de realizar resolución de nombres de temas y nodos de manera transparente.

Las distribuciones[3] de ROS son soportadas oficialmente para versiones específicas del sistema operativo Ubuntu. Adicionalmente es posible instalarlo de forma experimental en los sistemas operativos OS X, Gentoo Linux, Arch Linux, Android y Windows. Cada distribución de ROS se encuentra asociada a una letra del alfabeto, por ejemplo: Hydro Medusa, Indigo Igloo y Jade Turtle. Por último la mascota de ROS es la tortuga y existen diferentes versiones de este animal para cada distribución (ver Figura A.1).

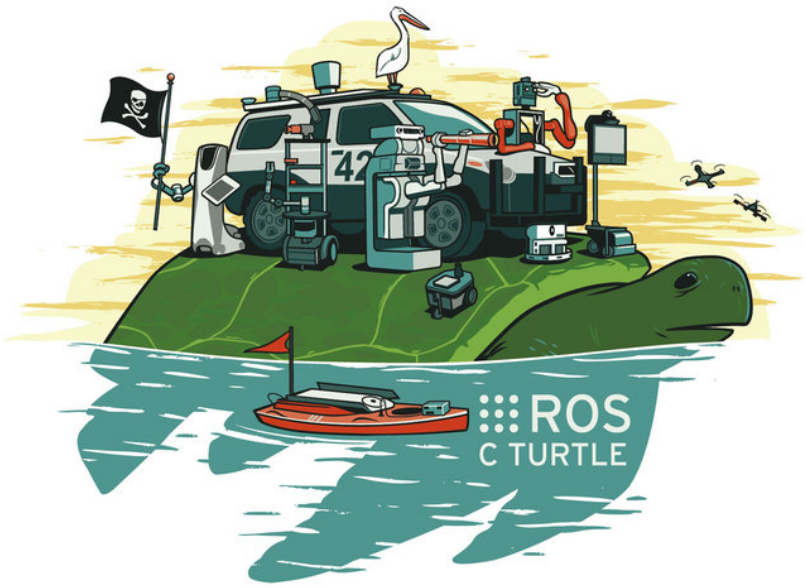


Figura A.1: Mascota de ROS en la distribución C Turtle.

A.1.2. Elementos de ROS

Los elementos principales que componen ROS son los nodos y temas. El núcleo proporciona todos los elementos necesarios para permitir la comunicación entre estos elementos a través de mensajes. Las acciones permiten la ejecución de tareas largas, sin embargo son diferentes de los servicios de ROS. En los servicios, los nodos envían una solicitud y reciben una respuesta.

Núcleo

El proceso encargado de gestionar la comunicación entre nodos es el núcleo `roscore`, por lo tanto debe haber una instancia de este núcleo ejecutándose en todo momento

[3]Versiones del conjunto de paquetes de ROS

(ver Sección A.2). Todo proceso que se ejecuta en ROS se registra en `roscore` para que el resto de los elementos pueda tener acceso a él. Una vez que el núcleo está, activo es posible utilizar un conjunto de herramientas para monitoreo y administración de elementos de ROS desde la consola. Entre dichas herramientas se encuentran: `rqt_graph` (ver Figura A.3), para mostrar un grafo con los nodos en ejecución (ver Figura A.3); `rostopic`, para obtener y publicar información en los temas y `rqt_tf_tree`, para observar el grafo de transformaciones entre marcos de referencia.

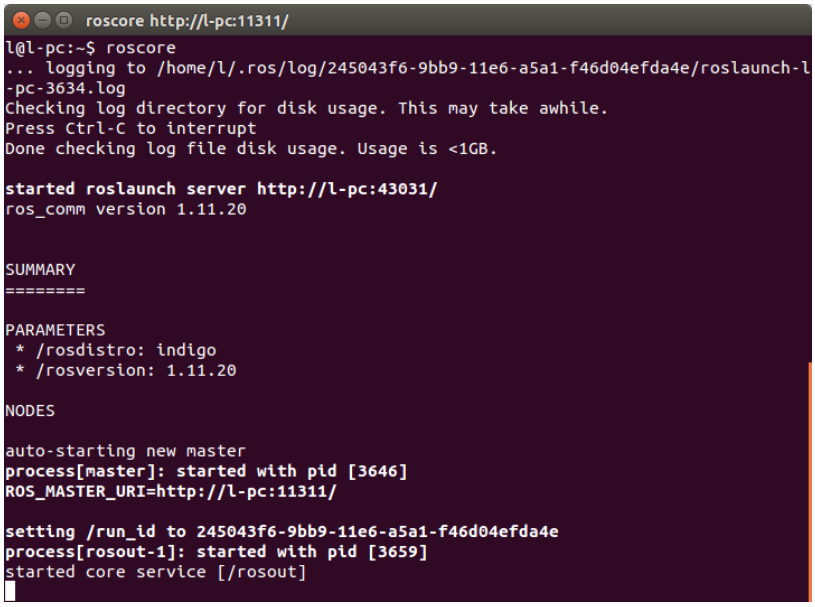


Figura A.2: Roscore en ejecución.

Nodos

Los nodos son programas en lenguaje C++ o Python que se ejecutan en el contexto de ROS. Un robot puede estar compuesto de múltiples nodos que se encargan de obtener información de los diversos elementos que lo componen, e.g., información de sensores. Los nodos se comunican entre sí principalmente mediante el modelo de comunicación publicador-subscriptor. Aunque también es posible utilizar el modelo cliente-servidor. Los nodos son supervisados por el proceso `roscore` y los mensajes de estado que se registran de ellos pueden ser consultados en la herramienta `rqt_console`.

Mensajes

Los mensajes son una estructura de datos que puede ser anidada, i.e., mensajes compuestos de otros mensajes. Los mensajes se compone de un conjunto de campos definidos por un tipo de mensaje y una descripción. Los tipos de mensajes cuentan con primitivas similares a los del lenguaje C (integer, boolean, string, entre otros). Con la instalación de ROS, se incluyen tipos de mensajes que son de utilidad en un entorno

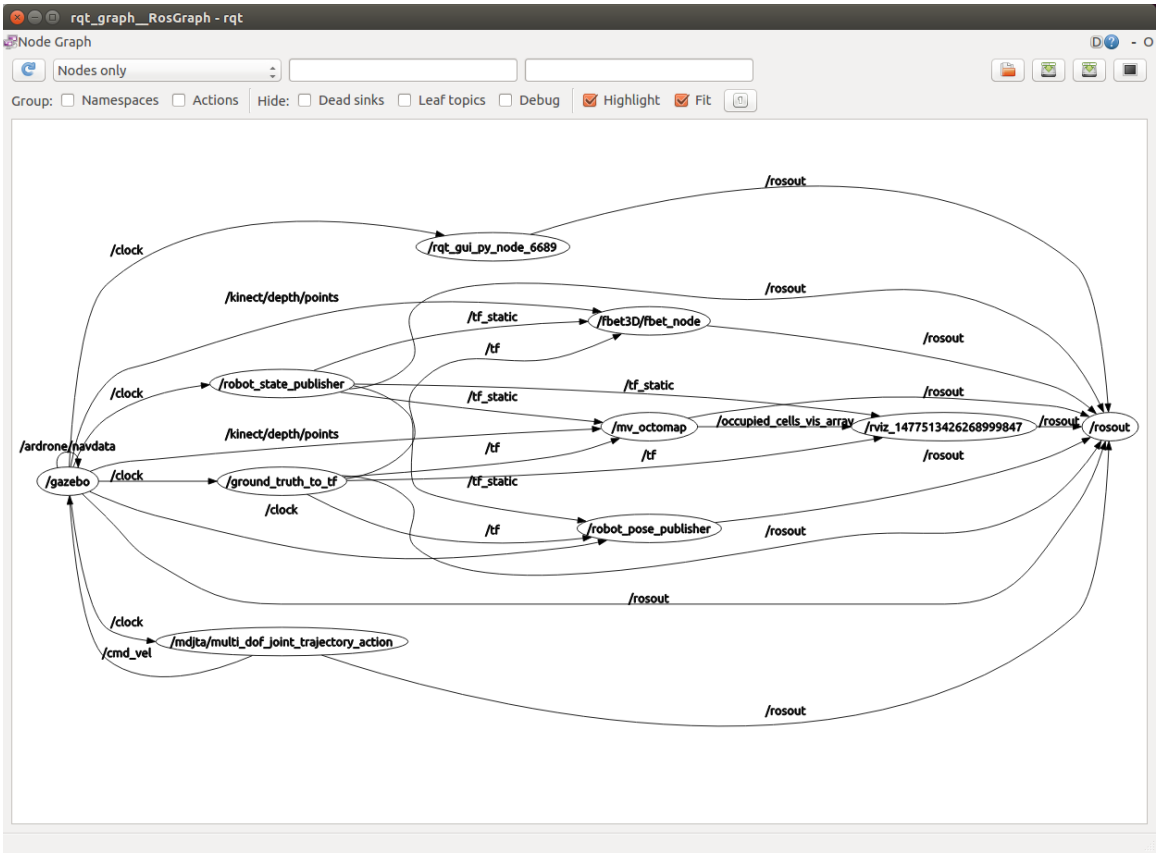


Figura A.3: Herramienta rqt_graph para visualización de nodos y temas.

robótico. Entre estos tipos de mensajes se encuentran: mensajes de velocidad, mensajes de información de profundidad, encabezados de mensajes, marcas de tiempo y datos de posición, entre otros. De manera adicional, los desarrolladores pueden crear tipos de mensajes personalizados que se ajusten a sus necesidades.

Temas

Uno de los principales métodos de comunicación entre nodos son los temas (*topics*), los cuales son creados, en su mayoría, cuando un nodo anuncia que quiere publicar información. Los temas también se crean mediante **servidores de acciones**, aunque en este caso no es posible definir el nombre del tema. En el modelo publicador-subscriptor, los nodos pueden interactuar con los temas, mediante publicación de mensajes o subscripción para recibir los mensajes cuando estén disponibles. Los temas tienen nombres únicos y si dos nodos anuncian que publicarán información en temas con el mismo nombre se considera que es el mismo tema.

Acciones

Las acciones son proporcionadas por el paquete `actionlib` y están diseñadas para ejecutar acciones que pueden tardar mucho tiempo en completarse. Para utilizar acciones es necesario definirlas previamente en un archivo dividido en tres partes: meta, resultado y progreso. La meta es la tarea a ejecutar, el resultado, como su nombre lo indica, es la información obtenida al ejecutar la tarea y el progreso es el estado de la tarea a lo largo del tiempo. Las acciones funcionan con el modelo de comunicación cliente-servidor. El servidor, llamado **servidor de acciones** (*action server*), se subscribe a los temas `nombreAccion/goal` y `nombreAccion/cancel` y publica en los temas `nombreAccion/feedback` y `nombreAccion/result`. El cliente, llamado **cliente de acciones** (*action client*), se subcribe a estos dos últimos temas para obtener información de la tarea solicitada y publica información en los dos primeros temas para asignar o cancelar tareas.

A.1.3. Creación de aplicaciones en ROS

Para el desarrollo de aplicaciones en ROS, se pueden utilizar los lenguajes de programación C++ y Python. Las bibliotecas de ROS permiten la creación de nodos, publicación de mensajes y transformación de datos, entre otros. ROS se apoya en el sistema de construcción[4] CMake, responsable de generar los elementos solicitados a partir de código fuente; estos elementos pueden ser bibliotecas, programas ejecutables, scripts, etc.

Catkin

A partir de la versión Hydro Medusa, se introdujo un nuevo sistema de construcción llamado `catkin`, como reemplazo del anterior, `rosbuild`, que combina macros de CMake y *scripts* de Python. Catkin está diseñado para ofrecer una mejor distribución de paquetes y portabilidad. Además `catkin` agregó soporte para la búsqueda de paquetes, lo cual es de utilidad cuando se construyen múltiples proyectos dependientes a la vez. En ROS, el código fuente está organizado en los llamados "paquetes"; cada paquete puede contener uno o más elementos cuando es construido. Un ejemplo del contenido de un paquete es los programas ejecutables que actúan como nodos de ROS.

Espacios de trabajo de catkin

Los espacios de trabajo de `catkin` son carpetas donde se almacenan los elementos necesarios para compilar y ejecutar un conjunto de paquetes. Estas carpetas, a su vez, se componen de tres subcarpetas principales:

- **build**: es una carpeta autogenerada por el comando `catkin_make` al construir un espacio de trabajo; dentro de esta carpeta se coloca el resultado de la construcción de los paquetes, como pueden ser los binarios;

[4] En inglés, *build system*

- **devel**: en esta carpeta, el comando `catkin_make` coloca las bibliotecas y archivos de cabecera utilizados por los paquetes; adicionalmente se puede encontrar el archivo `setup.bash` que debe ser registrado en el entorno para permitir la localización de los paquetes ROS desarrollados y

- **src**: dentro de esta carpeta, se colocan los paquetes que se requiere construir.

Paquetes

Cómo se mencionó anteriormente, un paquete es una colección de elementos a ser construido. Los paquetes pueden contener múltiples archivos de código, definiciones de mensajes, definiciones de acciones y archivos de *script*, entre otros. Los paquetes compilados se pueden componer de múltiples nodos, los cuales son identificados mediante el nombre del paquete. Los paquetes son los componentes funcionales de ROS y permiten extender las capacidades del *framework* mediante nuevos robots, modelos, sensores y funcionalidades diversas. La comunidad ROS proporciona múltiples paquete,s sin embargo algunos de ellos son específicos de ciertas distribuciones de ROS. Esta situación se debe a los cambios que ocurren entre versiones de ROS, los cuales pueden dejar funciones obsoletas o cambiar la forma en cómo se realizan las cosas.

Archivos de lanzamiento

Los nodos de ROS se ejecutan mediante el comando `rosrun`, sin embargo cuando se inician múltiples nodos es de utilidad agruparlos mediante **archivos de lanzamiento** (*launch files*). Este tipo de archivos son de texto plano donde se especifica un conjunto de nodos a ejecutar y otros archivos de lanzamiento. Estos archivos también permiten especificar argumentos para los nodos y leer información del **servidor de parámetros**. Los archivos de lanzamiento son ejecutados desde la terminal con el comando `roslaunch`. Finalmente, estos archivos son de utilidad para reducir el número de terminales utilizadas para la ejecución de nodos. Además integran mecanismos de configuración que proporcionan tolerancia a fallos, e.g., reiniciar un nodo en caso de fallo y finalizar el resto de los nodos si alguno de importancia produce un error.

A.2. RViz

En esta sección se describen los elementos principales de la herramienta RViz. En la Sección A.2.1 se explica brevemente el funcionamiento de RViz. En la Sección A.2.2, se presentan los **marcadores de visualización** de RViz. Estos marcadores son mensajes de primitivas en 3D, las cuales pueden ser observadas en RViz. Finalmente, en la Sección A.2.3, se presenta el componente de MoveIt! para RViz; este componente es de utilidad para interactuar con el **grupo de movimiento** de MoveIt!.

A.2.1. Introducción a RViz

RViz es una herramienta de visualización de ROS que permite observar información en 3D (ver Figura A.4). RViz puede ser iniciado mediante el comando `rviz` y admite archivos de configuración que son ejecutados mediante archivos de lanzamiento. Cabe destacar que las funciones de RViz únicamente incluyen la visualización de objetos creados por otros componentes, por lo tanto RViz no posee funciones de simulación.

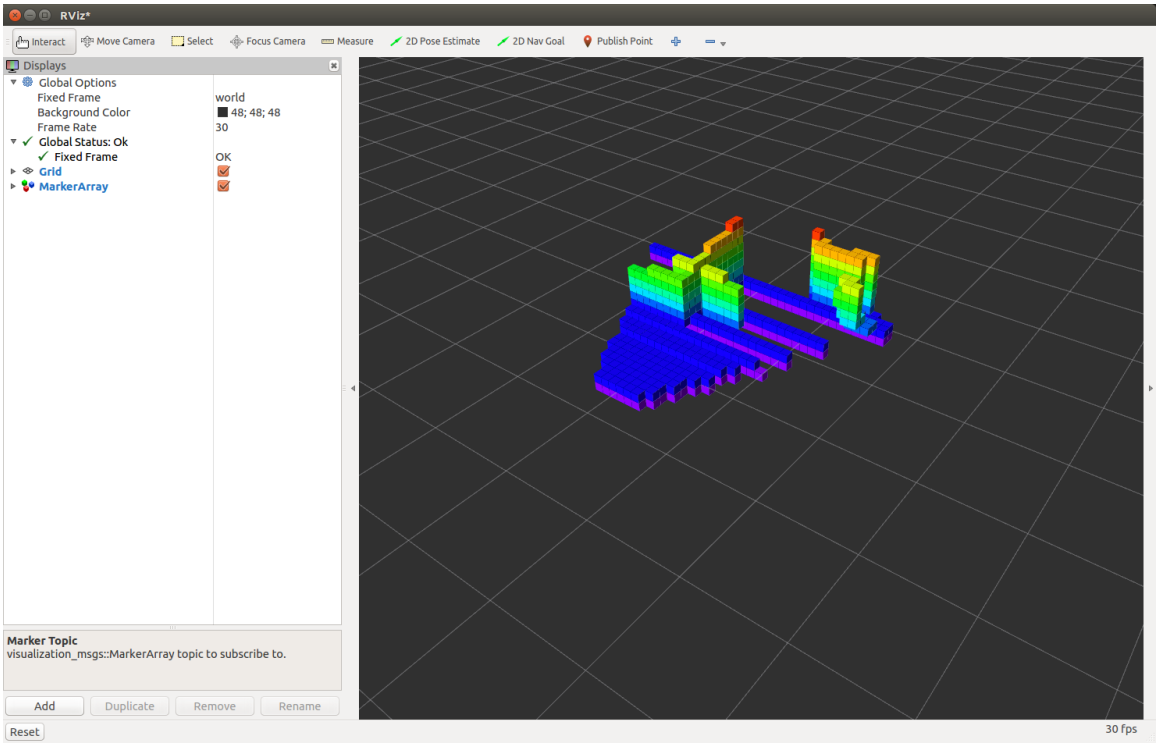


Figura A.4: Herramienta de visualización de ROS, RViz.

Pantallas

Las **pantallas** (*displays*) en general son elementos que pueden ser desplegados en un plano 3D. Entre estos elementos se encuentran los marcadores de visualización, las nubes de puntos, mediciones láser, rutas y transformaciones. En el panel *displays* pueden agregarse dichos elementos además de *plugins*, como el de planificación de movimiento de MoveIt!.

Vistas

El panel de vistas permite intercambiar los controladores de vista utilizados para manipular la vista de la escena. Los controladores de vista funcionan mediante un sistema de *plugins*. Gracias a este sistema es posible incluir otros controladores para

distintos tipos de interacciones. Algunos de estos controladores proporcionan funciones para publicar las posiciones de las vistas como temas de ROS. Entre los tipos de controladores por defecto se encuentran: Orbit, el controlador por defecto; FPS, una cámara en primera persona y XY Orbit, que únicamente enfoca el plano XY.

A.2.2. Marcadores de visualización

Los marcadores de visualización (RViz *markers*) son mensajes del tipo `visualization_msgs/Marker` utilizados para representar primitivas en 3D. Estos mensajes pueden ser visualizados en la herramienta RViz y se componen de un encabezado, una ubicación, un tipo de marcador, una acción y un color.

Para utilizar los marcadores, se debe crear un objeto `visualization_msgs::Marker`. En este objeto se debe llenar el encabezado, el cual se compone de una marca de tiempo y un marco de referencia global. La marca de tiempo permite actualizar los cambios de posición y el marco de referencia sirve para colocar al marcador en la posición correcta. La ubicación del marcador está dada mediante un campo de tipo `geometry_msg::Pose`, el cual se compone de la posición y la orientación del marcador en forma de cuaternión. Los tipos de marcadores son: flecha, cubo, esfera, cilindro, *line strip*, lista de líneas, lista de cubos, lista de esferas, puntos, texto, mallas y listas de triángulos.

Las acciones que se pueden realizar con los marcadores son cuatro: agregar o modificar, eliminar y eliminar todos. Estas acciones se ven reflejadas en RViz. La acción de agregar ocurre cuando un marcador es publicado por primera vez. Cuando el mismo marcador es publicado con una diferente posición se actualiza su posición. La acción eliminar remueve el marcador y finalmente eliminar todos quita todos los marcadores.

A.2.3. Plugin de MoveIt! para RViz

El software de planificación robótica, MoveIt!, ofrece un *plugin* para RViz que permite interactuar con el **grupo de movimiento** (ver Figura A.5). Este grupo es el elemento principal de MoveIt! y controla la generación de planes de movimiento, adquisición de datos y ejecución de trayectorias. Para poder usar el *plugin*, se requiere la configuración previa de un paquete de MoveIt!.

El *plugin* de MoveIt! carga el modelo del robot y lo presenta en pantalla para permitir que el usuario pueda establecer las posiciones hacia las cuales debe planear. El *plugin* brinda acceso a las funciones de planeación de movimiento, permitiendo especificar el espacio de trabajo, el tiempo de planificación, los intentos de planificación y las restricciones.

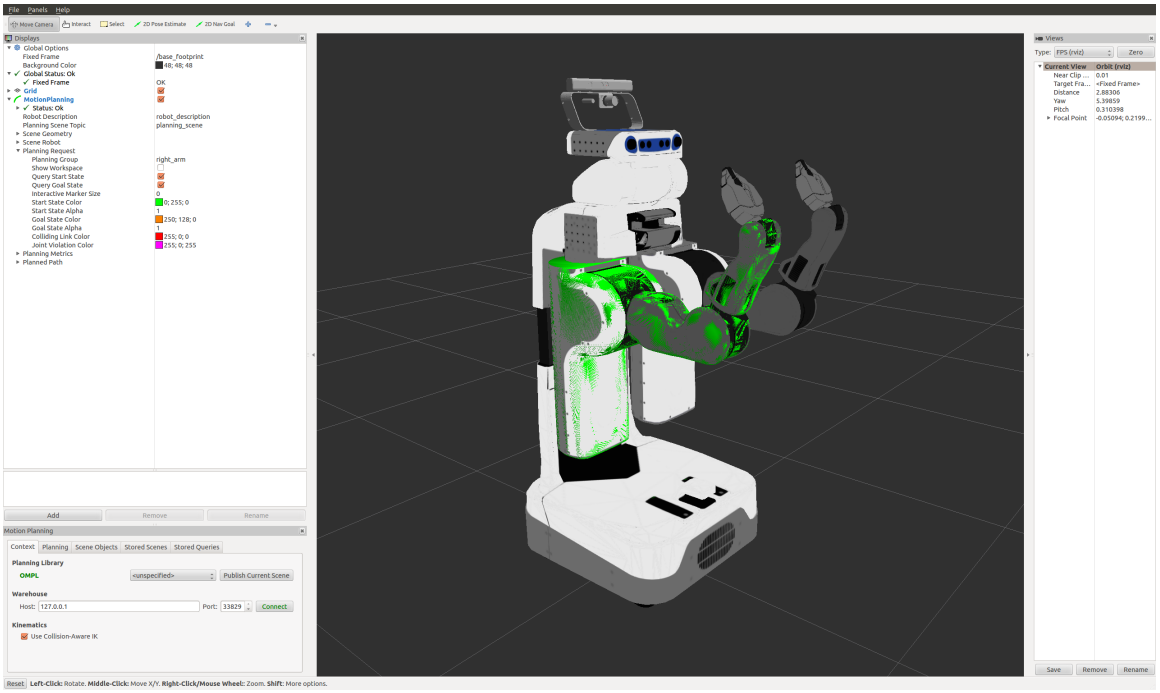


Figura A.5: Plugin de planificación de movimiento de MoveIt! [International, 2016].

En el panel *displays*, ubicado del lado izquierdo, se pueden configurar opciones del *plugin* como *Move Group Namespace* y *Planning Scene Topic*. La primera opción permite definir el **espacio de nombres** (*namespace*) en el que se encuentra el grupo de movimiento. Esta opción es importante cuando se trabaja con múltiples grupos de movimiento con el mismo nombre, pero en diferente espacio de nombres. Por otra parte, *Planning Scene Topic* indica el tema en el cual se está publicando el mapa usado para planificación de movimiento. Este tema no puede ser consultado directamente desde RViz, como se hace con los arreglos de marcadores publicados por el componente OctomapServer.

El *plugin* de MoveIt! cuenta con las siguientes categorías de opciones: *Scene geometry*, *Scene Robot*, *Planning Request*, *Planning Metrics* y *Planned Path*. *Scene geometry* agrupa las opciones de visualización del mapa usado para la planificación, e.g., estilo de coloreado. En *Scene Robot* es posible configurar la vista del robobot, e.g., mostrar el robot, cambiar el color del cuerpo, definir el enlace principal con el marco de referencia. En *Planning Request* se puede configurar el estilo de las posiciones iniciales y finales usadas para la planificación, además de mostrar el espacio de trabajo para la planificación. *Planning Metrics* agrupa funciones para mostrar u ocultar torques de las articulaciones y límite de peso. Por último, *Planned Path* proporciona opciones para cambiar el tema donde se publica la trayectoria planificada para el robot y para mostrar las colisiones de dicho robot.

A.3. Gazebo

En esta sección se presenta el simulador Gazebo utilizado ampliamente en robótica. En la Sección A.3.1 se presenta una breve introducción a la herramienta y sus componentes. En la Sección A.3.2 se describen los formatos de archivos utilizados para representar modelos de robots. Por último, en la Sección A.3.3 se presentan los sensores utilizados para adquirir información en Gazebo.

A.3.1. Introducción a Gazebo

Gazebo es una herramienta de simulación robótica en 3D creada en la Universidad del Sur de California por Andrew Howard y Nate Koenig [OSRF, 2016]. Actualmente Gazebo es desarrollado por la *Open Source Robotics Foundation* (OSRF) y aunque existen versiones de integradas a ROS, Gazebo se distribuye como un software independiente. Esta herramienta hace uso del motor gráfico OGRE3D para representar entornos 3D, e.g., texturas e iluminación.

Gazebo se compone de un servidor, `gzserver` y un cliente, `gzclient`. El servidor se encarga de procesar archivos de descripción para realizar la simulación. El cliente se conecta al servidor para visualizar los elementos simulados mediante una interfaz gráfica. También es posible iniciar el cliente y el servidor en un solo elemento mediante el comando `gazebo`. La versión de Gazebo integrada a ROS es un paquete llamado `gazebo_ros`. Este paquete incluye archivos de lanzamiento que permiten iniciar el simulador y cargar un mundo.

La interfaz de Gazebo se compone de un área de visualización, una barra de herramientas y un panel de propiedades (ver Figura A.6). En el área de visualización se puede interactuar con los elementos simulados mediante el uso del ratón. La barra de herramientas permite mover estos elementos, rotarlos o cambiarlos de tamaño. Esta barra también permite agregar tres primitivas: cubo, esfera y cilindro, sin embargo esta no es la forma principal para agregar elementos. En el panel de propiedades se presentan los nombres modelos de los elementos que componen la simulación. En este panel se encuentra la pestaña *insert* que da acceso a la lista de modelos disponibles. La complejidad de estos modelos varía desde los más sencillos, e.g., una lata de refresco, hasta los más complicados como el robot PR2 que incluye sensores láser.

A.3.2. Modelos de robots

Existen diferentes tipos de archivos en los cuales se pueden definir modelos de robots para Gazebo, entre ellos se encuentran los archivos SDF y URDF. Los archivos URDF (*Universal Robotic Description Format*) son archivos de texto en formato XML que contienen la descripción de los elementos de un robot. Por otra parte, los archivos SDF (*Simulation Description Format*) también son archivos en formato XML, pero

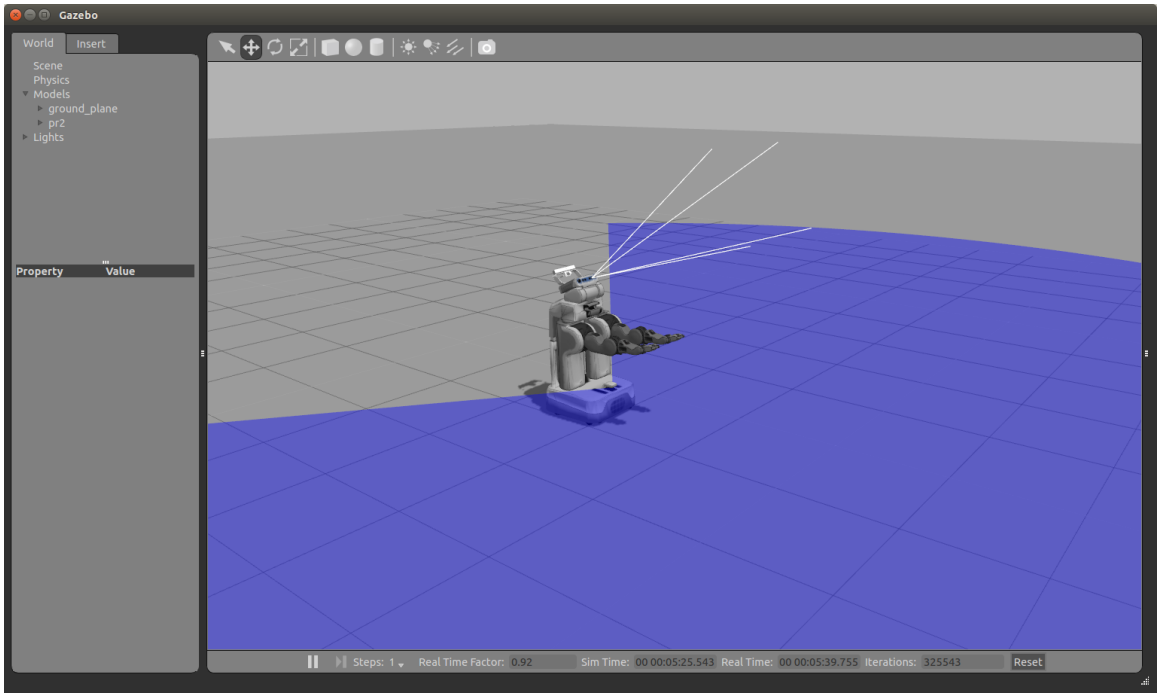


Figura A.6: Simulador Gazebo con el robot PR2.

son específicos para describir robots y entornos simulados; además los archivos URDF pueden ser convertidos a SDF.

En un archivo URDF, el robot es descrito como un conjunto de **enlaces** (*links*), conectados mediante **articulaciones** (*joints*). Los enlaces describen la forma de los elementos: geometría, origen, material, colisiones e inercia. Las articulaciones especifican las uniones entre enlaces: padre, hijo, origen, límite de movimiento y fricción. Los archivos URDF pueden ser simplificados mediante el lenguaje Xacro (XML Macros). Este lenguaje proporciona un conjunto de expresiones que permiten construir archivos XML. Por esta razón, es común encontrar archivos de descripción de robots con la extensión `.urdf.xacro`.

Los archivos SDF permiten especificar todos los elementos de la simulación. Ejemplo de estos elementos son: *world*, *model*, *actor* y *light*. *World* permite definir: física, escena, modelo, luces. *Model* incluye enlaces, articulaciones y elementos de agarre. *Actor* se compone de animaciones y scripts. Por último, *light* permite definir tipos de luces, difusión y atenuación, entre otros. Como se puede ver, existen opciones para definir diversos elementos de la simulación y el modelo del robot solamente es una parte de dicha definición. Por esta razón, es posible convertir un archivo URDF a uno SDF.

A.3.3. Sensores

Gazebo cuenta con diferentes tipos de sensores útiles para adquirir información del entorno simulado. Estos sensores son incorporados a los robots por medio de *plugins*. Algunos de los sensores disponibles son: cámaras, cámaras de profundidad, láser e IMU (*Inertial Measurement Unit*). Estos sensores pueden ser configurados con propiedades similares a las de los sensores reales.

Los *plugins* de sensores pueden ser agregados tanto en archivos SDF como URDF. Cuando se agregan en archivos URDF deben ser colocados dentro de una sección del XML llamada `gazebo`. Ya que los archivos URDF con convertidos a SDF por Gazebo, la sección `gazebo` es insertada directamente en la sección `model` del archivo SDF resultante de la conversión. Los *plugins* no incluyen un modelo del sensor representado, i.e., solamente son un cubo gris en la simulación. Estos modelos suelen encontrarse de manera independiente y son creados con el propósito de que la simulación luzca más realista.

En robótica, uno de los sensores más utilizados para percepción en interiores es el sensor RGB-D y Gazebo cuenta con un *plugin* de este tipo. Este sensor proporciona información de profundidad a color en forma de nubes de puntos. El sensor RGB-D más conocido es Microsoft Kinect, el cual es utilizado en entornos reales mediante el SDK OpenNI. Este sensor está implementado en Gazebo a través del *plugin* `libgazebo_ros_openni_kinect.so` que utiliza el mismo SDK.

Bibliografía

[3DR, 2016] 3DR (2016). X8-m — 3dr drone and uav technology. https://3dr.com/kb/x8-m-support/.

[ACM, 2015] ACM (2015). The 2012 acm computing classification system, association for computing machinery. http://www.acm.org/about/class/2012.

[Aerosocietyheritage.com, 2016] Aerosocietyheritage.com (2016). Royal aeronautical society — heritage — ws henson's 'aerial steam carriage': Specification (1842-1843). http://aerosocietyheritage.com/collections/hensons-arial-steam-carriage-specification/#Gallery.

[Airship.com.au, 2016] Airship.com.au (2016). Airship solutions - products. http://www.airship.com.au/products/large_airships.html.

[Aviastar.org, 2016] Aviastar.org (2016). All the world's rotorcraft. http://www.aviastar.org/.

[Azfar and Hazry, 2011] Azfar, A. Z. and Hazry, D. (2011). Simple gui design for monitoring of a remotely operated quadcopter unmanned aerial vehicle. In *2011 IEEE 7 th International Colloquium on Signal Processing and its Applications*.

[Bachrach et al., 2009] Bachrach, A., He, R., and Roy, N. (2009). Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228.

[Bachrach et al., 2012] Bachrach, A., Prentice, S., He, R., Henry, P., Huang, A. S., Krainin, M., Maturana, D., Fox, D., and Roy, N. (2012). Estimation, planning, and mapping for autonomous flight using an rgb-d camera in gps-denied environments. *The International Journal of Robotics Research*, 31(11):1320–1343.

[Berg, 2008] Berg, M. D. (2008). *Computational geometry*. Springer.

[Besl and McKay, 1992] Besl, P. J. and McKay, N. D. (1992). Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics.

[Bock, 2007] Bock, H.-H. (2007). *Clustering Methods: A History of k-Means Algorithms*, pages 161–172. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Books, 2016] Books, G. (2016). Patent us613809 method of and apparatus for controlling mechanism of moving vessels or vehicles. https://www.google.com/patents/US613809.

[Burgard et al., 2013] Burgard, W., Stachniss, C., Bennewitz, M., Tipaldi, D., and Spinello, L. (2013). Techniques for 3d mapping. http://ais.informatik.uni-freiburg.de/teaching/ss13/robotics/slides/17-3dmapping.pdf.

[CAP4D, 2016] CAP4D (2016). Techonology cap4d.

[Carrillo et al., 2012] Carrillo, L. R. G., López, A. E. D., Lozano, R., and Pégard, C. (2012). *Quad rotorcraft control: vision-based hovering and navigation*. Springer Science & Business Media.

[Cesare et al., 2015] Cesare, K., Skeele, R., Yoo, S.-H., Zhang, Y., and Hollinger, G. (2015). Multi-uav exploration with limited communication and battery. In *Robotics and Automation (ICRA), IEEE International Conference on*, pages 2230–2235. IEEE.

[Choset, 2005] Choset, H. M. (2005). *Principles of robot motion: theory, algorithms, and implementation*. MIT press.

[Cohen-Or, 2016] Cohen-Or, D. (2016). Flood-fill. http://www.math.tau.ac.il/~dcor/Graphics/cg-slides/flood-fill03.pdf.

[Collins, 2012] Collins, B. (2012). Introduction to data association. http://www.cse.psu.edu/~rtc12/CSE598C/datassocPart1.pdf.

[Commons, 2016a] Commons, W. (2016a). Interior del cubo. https://upload.wikimedia.org/wikipedia/commons/c/c3/Abierto.jpg.

[Commons, 2016b] Commons, W. (2016b). Rubik's cube. https://upload.wikimedia.org/wikipedia/commons/4/46/Rubik%27s_cube_resolved.svg.

[Daily, 2016] Daily, D. I. (2016). A160 hummingbird: Boeings variable-rotor vtuav. http://www.defenseindustrydaily.com/a160-hummingbird-boeings-variable-rotor-vtuav-03989/.

[Dalamagkidis et al., 2012] Dalamagkidis, K., Valavanis, K. P., and Piegl, L. A. (2012). *On Integrating Unmanned Aircraft Systems into the National Airspace System: Issues, Challenges, Operational Restrictions, Certification, and Recommendations*, chapter Aviation History and Unmanned Flight, pages 11–42. Springer Netherlands, Dordrecht.

[De Croon et al., 2009] De Croon, G., De Clercq, K., Ruijsink, Remes, and De Wagter, C. (2009). Design, aerodynamics, and vision-based control of the delfly. *International Journal of Micro Air Vehicles*, 1(2):71–97.

[de Croon et al., 2016] de Croon, G., Perçin, M., Remes, B., Ruijsink, R., and De Wagter, C. (2016). Introduction to flapping wing design. In *The DelFly*, pages 9–29. Springer.

[Deveau, 2016] Deveau, M. (2016). Image recording from aerial vectors. `http://www.map.archi.fr/aibm/Portal_of_Architectural_Image-Based-Modeling/Article-Deveau.html`.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[Dobrokhodov, 2015] Dobrokhodov, V. (2015). *Kinematics and Dynamics of Fixed-Wing UAVs*, pages 243–277. Springer Netherlands, Dordrecht.

[Domingues, 2009] Domingues, J. M. B. (2009). Quadrotor prototype. *Uneversidade Tecnica deLisboa. Dissertacio*.

[Everett and Toscano, 2015] Everett, H. and Toscano, M. (2015). *Unmanned Systems of World Wars I and II*. MIT Press.

[FAA, 2008] FAA, F. A. A. (2008). ft systems operations in the u. s. national airspace system.

[Fahlstrom and Gleason, 2012] Fahlstrom, P. and Gleason, T. (2012). *Introduction to UAV systems*. John Wiley & Sons.

[Festo, 2013] Festo, A. (2013). Bionicopter.

[Fischler and Bolles, 1981] Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395.

[Flyingmachines.org, 2016] Flyingmachines.org (2016). Flying machines william samuel henson. `http://www.flyingmachines.org/hens.html`.

[Franchi et al., 2009] Franchi, A., Freda, L., Oriolo, G., and Vendittelli, M. (2009). The sensor-based random graph method for cooperative robot exploration. *IEEE/ASME Transactions on Mechatronics*, 14(2):163–175.

[Fraundorfer et al., 2012] Fraundorfer, F., Heng, L., Honegger, D., Lee, G. H., Meier, L., Tanskanen, P., and Pollefeys, M. (2012). Vision-based autonomous mapping and exploration using a quadrotor mav. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4557–4564. IEEE.

[Grisetti et al., 2009] Grisetti, G., Stachniss, C., Arras, K., and Burgard, W. (2009). Data association. `http://ais.informatik.uni-freiburg.de/teaching/ws09/robotics2/pdfs/rob2-11-dataassociation.pdf`.

[Grossman, 2016] Grossman, D. (2016). Airships, dirigibles, zeppelins, and blimps. `http://www.airships.net/dirigible`.

[Heng et al., 2011] Heng, L., Lee, G. H., Fraundorfer, F., and Pollefeys, M. (2011). Real-time photo-realistic 3d mapping for micro aerial vehicles. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4012–4019. IEEE.

[Hornung et al., 2013] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206.

[Huang and Sturm, 2016] Huang, H. and Sturm, J. (2016). tum_simulator. `http://wiki.ros.org/tum_simulator`.

[Ignacio Sánchez, 2016] Ignacio Sánchez, J. (2016). De dónde procede el término dron. `http://www.nosolosig.com/articulos/270-de-donde-procede-el-termino-dron`.

[Inkpen, 2016] Inkpen, K. (2016). Filling algorithms. `http://www.cs.sfu.ca/CourseCentral/361/inkpen/Notes/361_lecture6.pdf`.

[International, 2016] International, S. (2016). Moveit rviz tutorial. `http://docs.ros.org/hydro/api/moveit_ros_visualization/html/doc/tutorial.html`.

[Ioan A. and Sachin, 2016] Ioan A., S. and Sachin, C. (2016). Moveit! `http://moveit.ros.org/`.

[JCGUAV, 2007] JCGUAV, J. C. G. o. U. A. V. (2007). Unmanned aerial vehicle systems airworthiness requirements (usar).

[Jiang and Voyles, 2015] Jiang, G. and Voyles, R. M. (2015). *Dexterous UAVs for Precision Low-Altitude Flight*, pages 207–237. Springer Netherlands, Dordrecht.

[Kavraki and LaValle, 2008] Kavraki, L. E. and LaValle, S. M. (2008). *Motion Planning*, pages 109–131. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Kavraki et al., 1996] Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566–580.

[Khoshelham and Elberink, 2012] Khoshelham, K. and Elberink, S. O. (2012). Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454.

[Kumar and Michael, 2012] Kumar, V. and Michael, N. (2012). Opportunities and challenges with autonomous micro aerial vehicles. *The International Journal of Robotics Research*, 31(11):1279–1291.

[LaValle, 1998] LaValle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning.

[LaValle, 2006] LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

[Lawson, 2015] Lawson, A. E. (2015). Cooperative 3-d map generation using multiple uavs.

[Marshall et al., 2015] Marshall, D. M., Barnhart, R. K., Shappee, E., and Most, M. T. (2015). *Introduction to unmanned aircraft systems*. Crc Press.

[Matignon et al., 2012] Matignon, L., Jeanpierre, L., and Mouaddib, A.-I. (2012). Distributed value functions for multi-robot exploration. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1544–1550. IEEE.

[Maza et al., 2015] Maza, I., Ollero, A., Casado, E., and Scarlatti, D. (2015). *Classification of Multi-UAV Architectures*, pages 953–975. Springer Netherlands, Dordrecht.

[Meagher, 1982] Meagher, D. (1982). Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147.

[Mehta, 2013] Mehta, A. (2013). History tuesday: The origin of the term drone. `http://intercepts.defensenews.com/2013/05/the-origin-of-drone-and-why-it-should-be-ok-to-use/`.

[Mellinger et al., 2012] Mellinger, D., Michael, N., and Kumar, V. (2012). Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, page 0278364911434236.

[Moravec and Elfes, 1985] Moravec, H. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 116–121. IEEE.

[Muhammad et al., 2009] Muhammad, N., Fofi, D., and Ainouz, S. (2009). Current state of the art of vision based slam. In *IS&T/SPIE Electronic Imaging*, pages 72510F–72510F. International Society for Optics and Photonics.

[NASA, 2016] NASA (2016). Aeronautics and astronautics chronology 1915 a 1919. `http://www.hq.nasa.gov/office/pao/History/Timeline/1915-19.html`.

[Newman et al., 2003] Newman, P., Bosse, M., and Leonard, J. (2003). Autonomous feature-based exploration. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 1, pages 1234–1240. IEEE.

[Nistér et al., 2004] Nistér, D., Naroditsky, O., and Bergen, J. (2004). Visual odometry. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–652. IEEE.

[Nonami et al., 2010] Nonami, K., Kendoul, F., Suzuki, S., Wang, W., and Nakazawa, D. (2010). *Autonomous Flying Robots: Unmanned Aerial Vehicles and Micro Aerial Vehicles*. Springer Science & Business Media.

[Nüchter, 2008] Nüchter, A. (2008). *3D robotic mapping: the simultaneous localization and mapping problem with six degrees of freedom*, volume 52. Springer.

[Occipital, 2016] Occipital, I. (2016). Structure sensor. url=http://structure.io/.

[of Defense Office of the Secretary of Defense, 2007] of Defense Office of the Secretary of Defense, U. D. (2007). Unmanned systems roadmap 2007-2032.

[of flight comission, 2016] of flight comission, U. C. (2016). History of flight. `http://www.centennialofflight.net/timeline/search_timeline.htm`.

[Origins, 2014] Origins, A. (2014). The steam-powered pigeon of archytas, the flying machine of antiquity. `http://www.ancient-origins.net/ancient-technology/steam-powered-pigeon-archytas-flying-machine-antiquity-002179`.

[O'Rourke, 1998] O'Rourke, J. (1998). *Computational geometry in C*, chapter 8, page 294. Cambridge university press.

[OSRF, 2016] OSRF (2016). Gazebo. `http://gazebosim.org/`.

[Rooker and Birk, 2007] Rooker, M. N. and Birk, A. (2007). Multi-robot exploration under the constraints of wireless networking. *Control Engineering Practice*, 15(4):435–445.

[Rusu et al., 2009] Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. IEEE.

[Rycroft, 2016] Rycroft, C. H. (2016). Voro++. `http://math.lbl.gov/voro++/`.

[Saska et al., 2014] Saska, M., Chudoba, J., Přeučil, L., Thomas, J., Loianno, G., Třešňák, A., Vonásek, V., and Kumar, V. (2014). Autonomous deployment of swarms of micro-aerial vehicles in cooperative surveillance. In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 584–595. IEEE.

[Scrtargets.es, 2016] Scrtargets.es (2016). Scrab ii. `http://www.scrtargets.es/index.php/en/products/scra`.

[Selby, 2016] Selby, W. (2016). System modeling. `http://wilselby.herokuapp.com/sample-page/research/arducopter/modeling/`.

[Sensintel, 2016] Sensintel (2016). Portable launchers. `http://2014.sensintel.com.php53-14.ord1-1.websitetestlink.com/UAV/launcher/`.

[Senson and Ritter, 2011] Senson, B. and Ritter, J. (2011). *Aerospace engineering From the ground up*. Delmar Cengage Learning.

[Shade, 2011] Shade, R. (2011). *Choosing Where To Go: Mobile Robot Exploration*. PhD thesis, University of Oxford, Oxford, United Kingdom.

[Shen et al., 2011] Shen, S., Michael, N., and Kumar, V. (2011). Autonomous multi-floor indoor navigation with a computationally constrained mav. In *Robotics and automation (ICRA), 2011 IEEE international conference on*, pages 20–25. IEEE.

[Shen et al., 2012] Shen, S., Michael, N., and Kumar, V. (2012). Autonomous indoor 3d exploration with a micro-aerial vehicle. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 9–15. IEEE.

[Sheng et al., 2006] Sheng, W., Yang, Q., Tan, J., and Xi, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, 54(12):945–955.

[Simmons et al., 2000] Simmons, R., Apfelbaum, D., Burgard, W., Fox, D., Moors, M., Thrun, S., and Younes, H. (2000). Coordination for multi-robot exploration and mapping. In *AAAI/IAAI*, pages 852–858.

[Solanas and Garcia, 2004] Solanas, A. and Garcia, M. A. (2004). Coordinated multi-robot exploration through unsupervised clustering of unknown space. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 1, pages 717–721. IEEE.

[Stachniss, 2009] Stachniss, C. (2009). *Robotic mapping and exploration*, volume 55. Springer.

[Stachniss, 2012a] Stachniss, C. (2012a). Ekf slam. `http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam04-ekf-slam.pdf`.

[Stachniss, 2012b] Stachniss, C. (2012b). Least squares approach to slam. `http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam15-ls-slam.pdf`.

[Stachniss, 2014] Stachniss, C. (2014). Introduction to robot mapping. `http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam01-intro.pdf`.

[Stachniss and Burgard, 2003] Stachniss, C. and Burgard, W. (2003). Mapping and exploration with mobile robots using coverage maps. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 467–472. IEEE.

[Stachniss et al., 2005] Stachniss, C., Grisetti, G., and Burgard, W. (2005). Information gain-based exploration using rao-blackwellized particle filters. In *Robotics: Science and Systems*, volume 2, pages 65–72.

[Şucan et al., 2012] Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82. `http://ompl.kavrakilab.org`.

[Thrun et al., 2002] Thrun, S. et al. (2002). Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, 1:1–35.

[Thrun and Leonard, 2008] Thrun, S. and Leonard, J. J. (2008). *Simultaneous Localization and Mapping*, pages 871–889. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Tonioni, 2016] Tonioni, A. (2016). Autonomous flight ros. `https://github.com/AlessioTonioni/Autonomous-Flight-ROS`.

[Toris, 2016] Toris, R. (2016). robot_pose_publisher. `http://wiki.ros.org/robot_pose_publisher`.

[Triebel et al., 2006] Triebel, R., Pfaff, P., and Burgard, W. (2006). Multi-level surface maps for outdoor terrain mapping and loop closing. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2276–2282.

[Valavanis and Vachtsevanos, 2015] Valavanis, K. and Vachtsevanos, G. (2015). Introduction to the handbook on uavs. In Valavanis, K. P. and Vachtsevanos, G. J., editors, *Handbook of Unmanned Aerial Vehicles*, pages 5–42. Springer Netherlands.

[Valavanis and Kontitsis, 2007] Valavanis, K. P. and Kontitsis, M. (2007). *Advances in Unmanned Aerial Vehicles: State of the Art and the Road to Autonomy*, chapter A Historical Perspective on Unmanned Aerial Vehicles, pages 15–46. Springer Netherlands, Dordrecht.

[van Blyenburgh, 2006] van Blyenburgh, P. (2006). Uav systems: global review. In *Conference, Amsterdam, The Netherlands*.

[Wallgrn, 2009] Wallgrn, J. O. (2009). *Hierarchical Voronoi Graphs: Spatial Representation and Reasoning for Mobile Robots*. Springer Publishing Company, Incorporated, 1st edition.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152.

[Wu et al., 2007] Wu, L., García García, M. Á., Puig Valls, D., and Solé Ribalta, A. (2007). Voronoi-based space partitioning for coordinated multi-robot exploration.

[Wurm et al., 2008] Wurm, K. M., Stachniss, C., and Burgard, W. (2008). Coordinated multi-robot exploration using a segmentation of the environment. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1160–1165. IEEE.

[Yamauchi, 1997] Yamauchi, B. (1997). A frontier-based approach for autonomous exploration. In *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, pages 146–151. IEEE.

[Zaloga, 2008] Zaloga, S. J. (2008). *Unmanned Aerial Vehicles: Robotic Air Warfare 1917-2007*, volume 144. Osprey Publishing.

[Zhu et al., 2015] Zhu, C., Ding, R., Lin, M., and Wu, Y. (2015). A 3d frontier-based exploration tool for mavs. In *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on*, pages 348–352. IEEE.

[Zimmer, 2016] Zimmer, B. (2016). The flight of 'drone' from bees to planes. `http://www.wsj.com/news/articles/SB10001424127887324110404578625803736954968`.