Universidad de la República

Facultad de Ingeniería

Exploración Colaborativa con Butiá

María Victoria Díaz

Sergio Robaudo

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Supervisado por

Facundo Benavides

Mercedes Marzoa

12 de abril de 2018

Resumen

En el presente trabajo se estudia el problema de optimización del tiempo de exploración de un grupo de robots que busca generar un mapa 2D de un área cerrada desconocida mediante la colaboración entre ellos. Se abordan los problemas de definición de tareas, planificación y navegación; y se asumen como resueltos los problemas de localización y comunicación.

Existe comunicación explícita entre los robots para compartir los datos obtenidos del ambiente y otros agentes, pero la planificación se realiza en forma descentralizada, coordinando implícitamente. Para la exploración se utilizó el enfoque orientado al descubrimiento de fronteras, utilizando el algoritmo Affinity Propagation para clusterizar los puntos de fronteras existentes en el mapa y simplificar así las tareas a procesar al planificar el siguiente objetivo a ser explorado. Para la planificación descentralizada se implementó el algoritmo MinPos en la versión que utiliza Wavefront Propagation para reducir el costo de computo del mismo y haciéndolo apto para ser ejecutado en procesadores de recursos limitados. Para navegación se utiliza el algoritmo A* generando así un camino a seguir desde la ubicación actual del robot hasta el destino escogido al planificar. Para el almacenamiento de los datos recolectados del entorno se utiliza una grilla probabilística, la cual ayuda a la validación de los distintos obstáculos encontrados en el entorno al ser procesados en distintos momentos de la exploración, como a la integración de los datos obtenidos desde otros colaboradores.

Se implementó la solución al problema utilizando el framework OROCOS sobre C++, y la arquitectura de *Data Flow*; teniendo componentes especializadas para cada una de las tareas que realiza el robot (planificación, navegación, control de motores, control de sensores, comunicación) y comunicando paquetes de datos entre ellas para disparar la ejecución de las mismas. La solución fue probada bajo ambientes simulados, con el soporte de la herramienta Morse utilizando mapas simples como un camino cíclico, y complejos como un laberinto, permitiendo estudiar la influencia de la adición de robots en el tiempo final de exploración, así como también el desempeño de una flota a lo largo del tiempo. Se realizaron también pruebas con robots Butiá y sensores de rango RPLidar en escenarios simples, mostrando que el conjunto de algoritmos seleccionado y la arquitectura implementada son viables en hardware real.

Palabras clave: Robótica, Cooperación, Exploración, Butiá, MinPos, Affinity Propagation, Sistemas Multi-robot.

Tabla de contenidos

Capítulo 1

Introducción

El problema a abordar en el presente proyecto es el de minimizar el tiempo de exploración de una flota de robots homogéneos que colaboran en la tarea. La exploración colaborativa multirobot es un área de investigación que abarca numerosas áreas de estudio de la robótica y de la investigación operativa, muchas de ellas complejas en si mismas. El problema de exploración se descompone en subproblemas los cuales se estudian independientemente (figura 1.1).

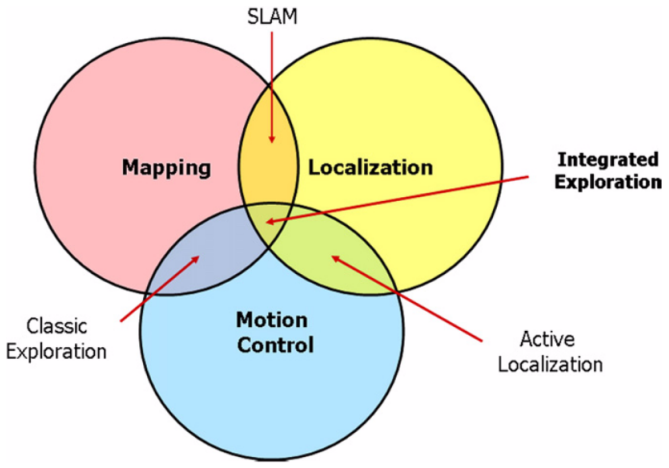


Figura 1.1: Descomposición del problema de exploración es sus subproblemas y sus respectivas interacciones. [1]

En el presente proyecto se analiza el problema desde un punto de vista teórico, para luego llevarlo a una implementación sobre la plataforma Butiá [2] y una simulación del mismo a modo de poder validar el estudio. Se plantea analizar los trabajos de investigación existentes para cada subproblema, determinar cuáles son los más eficientes para el problema de optimización elegido, implementarlos y por último validar los resultados obtenidos tanto en ambientes simulados como reales.

1.1. Motivación

La exploración de áreas desconocidas por robots autónomos puede ser necesaria en situaciones donde el acceso al área es difícil o peligroso [3]. Un ejemplo de ello son las tareas de búsqueda que se realizan luego de los desastres naturales, en los que se envía un robot en búsqueda de personas y animales y en caso de encontrar, reporta dónde se encuentra la persona o animal y cómo se puede llegar hasta el lugar; otro ejemplo son las misiones en las que se envían robots autónomos a otro planeta para que estos investiguen el área y reporten lo visto. En los últimos tiempos, se ha avanzado en la investigación de los sistemas multi-robot autónomos, que exploran el ambiente cooperativamente. Los beneficios de las flotas con múltiples robots sobre las exploraciones con un solo robot son [4]:

- la tarea puede ser muy compleja para ser realizada por un solo robot, o se pueden obtener mejoras en el desempeño usando múltiples robots;
- crear y usar múltiples robots puede ser más sencillo, económico, más flexible y más tolerante a fallas que usando un único robot para cada tarea;
- el enfoque de la robótica móvil cooperativa puede facilitar el estudio de problemas en las ciencias sociales (economía, teoría de la organización) y ciencias de la vida (biología teórica)

Capítulo 2

Estado del arte

En este capítulo se presentan algunas de las soluciones existentes para los subproblemas que surgen a la hora de resolver un problema de exploración cooperativa.

2.1. Exploración cooperativa

En el área de la exploración de ambientes desconocidos cada vez más se tiende hacia la exploración cooperativa y no a exploraciones con un solo robot ya que como se mencionó en 1.1, trabajar con una flota de robot trae grandes ventajas. Por el otro lado, al operar con múltiples robots, surgen nuevos problemas y esto trae algunas desventajas. Estos nuevos problemas son: el agregado de la coordinación, necesario para que la cooperación sea eficiente; y el riesgo de interferencia entre ellos [5]. Esta interferencia se puede presentar de diversas maneras, por ejemplo, si los robots tienen el mismo tipo de sensores, como sensores de ultra sonido, el rendimiento total puede ser reducido debido a la interferencia que pueden generar cuando dos robots están cerca; también se está limitado por los canales de comunicación ya que, al estar constantemente enviando información, se pueden saturar los canales de comunicación y esto generar pérdida de paquetes. Por otro lado, cuanto más grande sea el equipo de robots en una exploración, más veces van a tener que desviarse de su ruta óptima para evitar colisiones, por lo que es muy importante determinar cuántos robots son necesarios para lograr un tiempo óptimo de exploración.

Según Y. Cao et al. [4], el comportamiento cooperativo se define: *Dada una tarea especificada por un diseñador, un sistema de múltiples robots muestra un comportamiento cooperativo si, dado un mecanismo subyacente (por ejemplo, "mecanismo de cooperación"), hay un incremento de la utilidad total del sistema.*

2.2. Sensado y representación del mapa

Uno de los aspectos que se debe resolver como parte del problema de exploración es el mapeo y almacenamiento de los datos. Los mecanismos deben favorecer la expresividad de los datos almacenados, además de proveer estructuras aptas para la ejecución de los distintos algoritmos que asisten en la toma de decisiones.

Una de las estructuras que se utilizan para modelar entornos son las grillas de ocupación [6]. Las mismas, como su nombre lo indica, son grillas que representan una partición del entorno 2D en una cuadrícula de celdas de igual tamaño donde cada una se clasifica comparando su posibilidad de ocupación con la probabilidad inicial asignada a cada celda.
Dadas las probabilidades se puede decir que una celda está:

- Libre, si la probabilidad de ocupación es menor a la probabilidad inicial.
- Desconocida, si la probabilidad de ocupación es igual a la probabilidad inicial.
- Ocupada, si la probabilidad de ocupación es mayor a la probabilidad inicial.

Para la actualización de la grilla se utiliza el algoritmo introducido por Moravec y Elfes [7] que computa la probabilidad de ocupación $p(m)$ para el mapa m. El algoritmo busca maximizar la probabilidad de ocupación para el mapa a partir de las posiciones del robot $x_{1:t}$ y una secuencia de observaciones realizadas por un sensor $z_{1:t}$. Dado que el algoritmo asume que todas la celdas son independientes, la probabilidad del mapa m se puede expresar como el producto de la probabilidad de las celdas.

$$p(m) = \prod_{c \in m} p(c) \tag{2.1}$$

A partir la ecuación 2.1, y teniendo en cuenta que la probabilidad inicial es 0.5 se puede calcular la probabilidad de una celda $c \in m$ a partir de la posición actual del robot y una entrada sensorial:

$$p(c|x_{1:t}, z_{1:t}) = \left[1 + \frac{1 - p(c|x_t, z_t)}{p(c|x_t, z_t)} \frac{1 - p(c|x_{1:t-1}, z_{1:t-1})}{p(c|x_{1:t-1}, z_{1:t-1})}\right]^{-1} \tag{2.2}$$

En el caso de un sensor láser de rango:

$$p(c|z_{t,n}, x_t) = \begin{cases} p_{inicial}, & z_{t,n} \text{ rango máximo de lectura} \\ p_{inicial}, & c \text{ no está cubierto por } z_{t,n} \\ p_{occ}, & \|z_{t,n} - dist(x_t, c)\| < r \\ p_{libre}, & z_{t,n} \geq dist(x_t, c) \end{cases} \tag{2.3}$$

Donde la celda c se corresponde con el rayo n-esimo $z_{t,n}$ de la observación z_t, r es la resolución de la grilla y $0 \leq p_{occ} < p_{inicial} < p_{libre} \leq 1$.

Las grillas probabilistas son una representación del ambiente que permite solucionar problemas de inconsistencia en los datos sensados por distintos agentes de una flota, o incluso diferencias en la información sensada por un mismo robot en distintos momentos en el tiempo. Es muy importante la granularidad o resolución que se utilice para la representación de las celdas, ya que esto puede tener consecuencias tanto sobre el costo de ejecución de los algoritmos sobre la grilla [8], como también sobre la precisión en la navegación, especialmente en ambientes donde el espacio es reducido [9].

Otra estructura utilizada para la representación del mapa es Quad Tree, presentada por R. A. Finkel y J. L. Bentley [10]. Consiste en una estructura dinámica, en la cual cada nodo almacena un registro y tiene exactamente cuatro hijos. Para lograr esto, a medida que se van obteniendo nuevos datos, las regiones se particionan en cuatro partes iguales. Estas particiones pueden tener cualquier forma, pero las más frecuentes son cuadrados o rectángulos. Esta estructura es muy útil a la hora de buscar posibles caminos entre dos puntos.

2.3. Identificación de tareas

La pregunta que se busca responder a la hora de navegar por el ambiente es, dada la información que hay disponible del mundo, ¿para dónde me tengo que mover en el siguiente paso para obtener la mayor cantidad de información posible? Actualmente existen varias formas de afrontar este problema, la más usada y reconocida es la solución basada en el concepto de fronteras propuesto por Yamauchi [11, 12].
La idea central de esta propuesta es que, para ganar la mayor cantidad de información nueva del mundo, es necesario moverse al límite entre el espacio abierto y el territorio inexplorado. Para esto se definen las fronteras, regiones que se encuentran entre el espacio conocido y el inexplorado. Cuando el robot se dirige a estas regiones, puede realizar una búsqueda en profundidad, ver el espacio sin explorar y añadir la nueva información a su mapa. Al hacer esto de manera repetida, el robot está incrementando constantemente su conocimiento y es por esto que se llama exploración basada en fronteras. Si consideramos que un punto en el espacio es accesible si el robot, basado en su mapa, puede navegar hacia el y que todo el espacio accesible es contiguo, dado que tiene que existir un camino que lleve al robot de la posición inicial a cualquier punto accesible del entorno, entonces

cualquier camino que sea parcialmente desconocido va a cruzar una frontera y cuando el robot se dirija hacia ella va a conocer lo que se encontraba luego de esa frontera e integrarlo a su mapa. De esta manera es que, un robot que utilice exploración basada en fronteras eventualmente va a explorar todo el territorio accesible del entorno.

Otra alternativa es la propuesta por Upma Jaina et al. de realizar una partición en círculo [13]. Este acercamiento es especialmente útil para poder lograr un correcto balanceo de la carga de trabajo. Es necesario conocer previamente el tamaño y la forma del espacio a modelar. Para comenzar se divide el ambiente desconocido en subáreas lógicas y así se logra obtener un nivel de dispersión de los robots elevado. Esta partición se realiza haciendo un círculo con centro el centro del área a explorar y un radio de tal manera que cubra todo el espacio. Luego este círculo se subdivide en tantas partes como robots haya y se asignan los robots de manera aleatoria a cada sección. En la Figura 2.1 se puede observar las particiones realizadas para una exploración con cuatro robots en un ambiente determinado.

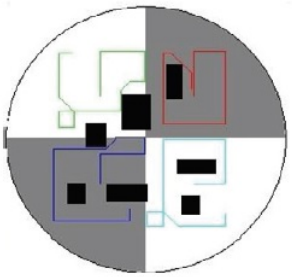


FIGURA 2.1: Cubrimiento de la ejecución del algoritmo de partición en círculo con cuatro robots [13]

Una tercera alternativa es la propuesta por Kai M. Wurm et al. de segmentar el ambiente [5]. Este algoritmo toma en cuenta la estructura del ambiente. En lugar de considerar las fronteras como objetivos, se particiona el ambiente ya explorado y se envía a los robots a determinado segmento para explorarlo. El objetivo es minimizar el tiempo requerido para completar la misión.
La segmentación del mapa se basa en el particionamiento de un grafo para el cual se utilizan los Diagramas de Voronoi [14]. Para la partición del grafo, se utilizan los puntos críticos (puertas), estos se eligen según las siguientes restricciones: los nodos críticos tienen que ser de grado dos, tienen que tener un vecino de grado tres y tienen que ir de una zona conocida a una desconocida. En la Figura 2.2 se puede observar un ejemplo de la selección de los nodos críticos en un ambiente. Una vez particionado el ambiente, cada robot es asignado a una habitación, un pasillo, o parte de un pasillo largo o habitación grande. Para la asignación se utiliza el Método Húngaro [15].

Como se puede observar, estos últimos algoritmos utilizan la idea de particiones para agrupar los puntos de interés a asignar a los robots. En el primer caso, la solución propuesta por Yamauchi, las fronteras son puntos en una grilla. Para poder crear regiones de fronteras, es necesario clusterizar los puntos; una posible opción para esta tarea es

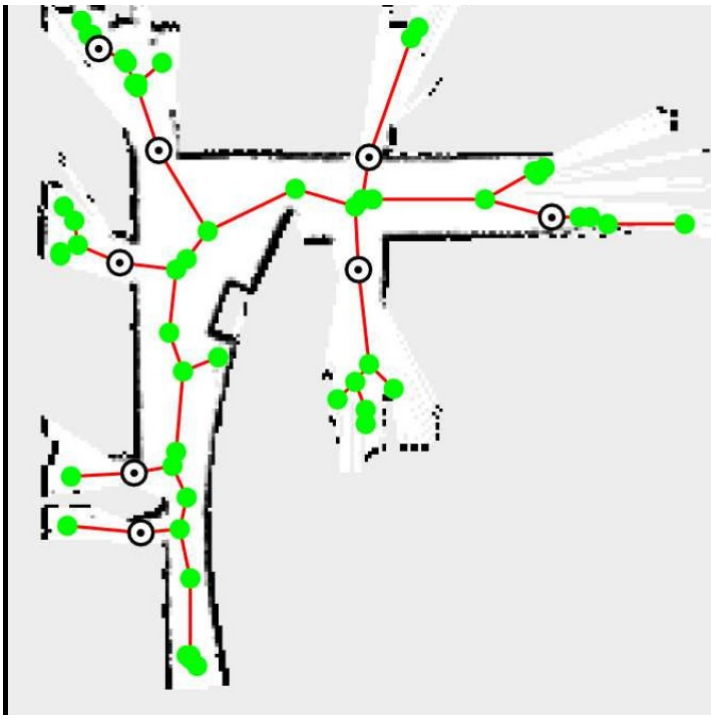


FIGURA 2.2: Ejemplo de segmentación de una pequeña fracción del entorno. Los nodos marcados son los candidatos para particionar el grafo calculado por el algoritmo de Segmentación del ambiente [5]

utilizar el algoritmo k-means [16].

Este es un algoritmo simple para resolver el problema de clusterización, donde, dado un conjunto de datos (celdas en una grilla en este caso) la idea es agrupar dichos datos en un número k de clústers, a priori fijo. La idea principal del algoritmo es que se toman k centros, uno para cada clúster y se los posiciona lo más lejos posible uno de otro. Luego, se asocia cada punto del conjunto de datos al centro más cercano. Una vez que no quedan más puntos a asociar, se recalculan los centroides utilizando el baricentro resultante del conjunto anterior, y se vuelven a asociar los puntos según los nuevos centroides. Este proceso continúa hasta que, al recalcular los centroides, el conjunto nuevo es igual que el conjunto anterior; en ese momento se llegó a la distribución óptima. El objetivo es minimizar la función del error cuadrático, como se muestra a continuación:

$$f(n) = \sum_{i=1}^{C} \sum_{j=1}^{C_i} \|x_i - v_{ij}\|_2^2 \tag{2.4}$$

Donde:

- C es la cantidad de clústers
- C_i es la cantidad de datos en el clúster i
- x_i es el centroide i

- v_{ij} es el dato j del centroide i
- $\|x_i - v_{ij}\|_2$ es la distancia euclídea entre x_i y v_{ij}

Una consideración a tener en cuenta al utilizar k-means en el escenario del problema estudiado, es que el algoritmo requiere saber el número de clústers a generar a priori; y al ser éste un dato desconocido se debe ejecutar el algoritmo múltiples veces evaluando distintos valores hasta encontrar uno que produzca un resultado aceptable.

Otra opción para resolver el problema de clusterización, es el algoritmo Affinity Propagation [17]. Éste algoritmo es muy popular en las áreas de Minería de Datos y Aprendizaje Automático, el mismo se basa en el intercambio de mensajes entre los distintos puntos de datos, y a diferencia de k-means, no requiere conocer de ante mano el número de grupos que se desea obtener; el propio algoritmo se encarga de definirlos en base al descubrimiento de puntos destacados o representativos, denominados *ejemplares*.

Affinitiy Propagation recibe como entradas del algoritmo el conjunto de puntos a clusterizar, y una matriz de similitudes, la cual refleja la semejanza entre toda pareja de puntos de la entrada. Para construir esta matriz, es posible basarse en la distancia entre los puntos, y definir las entradas de la matriz de similitudes como el valor negado del cuadrado de la distancia entre dos puntos; logrando de esta forma que para cualquier terna de puntos *x, y, z*; la similitud entre x e y sea mayor a la de x y z, si x se encuentra más cerca de y que de z. Esta matriz de similitudes tiene una particularidad, y es su diagonal, la cual tiene un significado distinto, se podría incluso considerar como un tercer parámetro de entrada al algoritmo; en la diagonal de la matriz se ingresa el valor de *preferencia*. La *preferencia* define para cada uno de los puntos a clusterizar que tan probable es que el mismo se convierta en un ejemplar; y en caso de todos los valores ser iguales, dependiendo qué tan próximo se encuentra a los valores mínimo y máximo de similitud en la matriz, se encarga de controlar la cantidad de clústers que se generarán. El algoritmo itera ejecutando dos pasos que se encargan de actualizar dos matrices. La primera es la matriz de responsabilidades, la cual se encarga de calcular que tan apropiado es un punto para convertirse en ejemplar para cada uno de los otros. La segunda es la matriz de disponibilidades, la cual se encarga de calcular para cada punto que tan apropiado sería para él, elegir a cada uno de los otros puntos como su ejemplar. Estas matrices son inicializadas con ceros, y pueden ser consideradas tablas de probabilidades logarítmicas.

Primero se actualiza la tabla de responsabilidades:

$$r(i,k) \leftarrow s(i,k) - \min_{j \neq k}\{a(i,j) + s(i,j)\} \tag{2.5}$$

Luego se actualiza la tabla de disponibilidades (*availability*):

$$a(i,k) \leftarrow \min_{i \neq k}\{0, r(k,k) + \sum_{j \neq i, j \neq k} \max(0, r(j,k))\} \tag{2.6}$$

$$a(k,k) \leftarrow \sum_{j \neq k} \max(0, r(j,k)) \tag{2.7}$$

Donde:

- $s(i,k)$ es la entrada correspondiente a los puntos i y k de la matriz de similitudes
- $r(i,k)$ es la entrada correspondiente a los puntos i y k de la matriz de responsabilidades
- $a(i,k)$ es la entrada correspondiente a los puntos i y k de la matriz de disponibilidades

Las iteraciones se repiten hasta que los clústers se estabilizan o hasta que se alcanza un cierto número máximo de iteraciones. Los ejemplares se definen, al finalizar las iteraciones de intercambio de mensajes, como aquellos puntos tales que la suma de sus responsabilidades y disponibilidades consigo mismos es positiva:

$$r(i,i) + a(i,i) > 0 \tag{2.8}$$

2.4. Asignación de tareas

Habiendo decidido cuales van a ser las tareas objetivo, ya sea utilizando fronteras o particiones, el próximo problema a resolver es: ¿cómo se asignan dichas tareas a los robots para ser ejecutadas?

El método Húngaro presentado por Khun [15] fue diseñado para asignar un set de tareas a un set de máquinas dado una matriz de costos fija. Este algoritmo se puede resumir en tres pasos:

- Calcular la matriz de costos reducida, quitándole a cada elemento el elemento minimal de su fila. Realizar lo mismo con las columnas.
- Encontrar el número minimal de las líneas verticales y horizontales requerido para cubrir todos los ceros en la matriz.
- Encontrar el elemento más chico, distinto de cero, en la matriz de costos reducida que no sea cubierto por una línea horizontal o vertical. Restar este valor de cada elemento descubierto de la matriz. Sumar a cada elemento de la matriz de costos

reducida que está cubierto por una línea horizontal y vertical. Continuar con el Paso 2

En el caso de la exploración, cada entrada de la matriz puede ser el largo del camino que el robot tiene que recorrer para llegar a ese objetivo. Como este método requiere tener una igual cantidad de máquinas que de tareas vamos a tener que usar maquinas "dummy" o duplicar los objetivos existentes.

En el caso de tener limitaciones en el alcance de la comunicación en sistemas multi-robot y puede suceder que uno o más robots se muevan fuera de los límites de comunicación y queden aislados, esto puede causar que los robots estén trabajando con los mapas desactualizados. Elizondo et al. [18] plantean un algoritmo basado en ofertas (o subasta autogestionada) para mitigar este problema. Este algoritmo se basa en un proceso de subasta de fronteras, en el que cada robot las calcula de manera independiente. Es completamente descentralizado y asíncrono. Cada robot estima la oferta de los demás y su propia oferta, para así decidir hacia dónde dirigirse.

Para el cálculo de ofertas se tienen en cuenta tres parámetros:

- Explotación: distancia que el robot va a tener que recorrer para alcanzar el objetivo y luego llegar a la frontera.
- Exploración: distancia promedio de una cierta frontera a la posición actual de los otros robots.
- Cohesión: se penaliza una frontera, si para alcanzarla el robot debe romper el puente de comunicación con el resto del equipo.

Para el cálculo de distancias de trayecto en el espacio libre y para obtener la planificación de movimientos requerida, se utiliza el algoritmo referencia EEDT (The Exact Euclidean Distance Transform) [19].
El proceso de planificación que realiza el mecanismo de oferta en la SBX-R es el siguiente:

- Cuando el robot alcanza su objetivo, solicita los mapas y los objetivos de los demás robots mediante el mecanismo de comunicación.
- Con esta información el robot:
  - Actualiza su mapa
  - Establece fronteras candidatas a ser exploradas en el proceso de oferta
  - Calcula la tabla de ofertas y resuelve el problema de emparejamiento óptimo mediante el método Húngaro, con el que el robot elige el objetivo que más beneficia a la exploración

- El robot transmite el nuevo objetivo al resto de los robots mediante el mecanismo de comunicación, para ser notificado si otro robot descubrió ese objetivo antes. En ese caso, vuelve a iniciar el proceso de ofertas
- Una vez seleccionado el objetivo, se planifica la trayectoria con el algoritmo EEDT y se dirige a la frontera seleccionada

Otra alternativa que resuelve estos problemas es la presentada por W. Sheng et al. [20] que se basa en la idea de subastas. Es un algoritmo descentralizado, pensado para un grupo homogéneo de robots, en el que cada robot tiene un rango limitado de comunicación. Se busca maximizar la información obtenida minimizando su costo, donde este costo es la distancia entre la posición actual del robot y la posición objetivo.
En cada instante el robot se encuentra en uno de estos tres estados: sensando y mapeando, ofertando o moviéndose. A la hora de realizar una oferta, el robot calcula la ganancia de información por cada celda frontera basándose en el mapa local actual, la posición actual de los otros robots y la posición destino de los otros robots que se están moviendo. Con esta ganancia de información, el robot hace broadcast de su oferta y espera un tiempo determinado. Si durante ese tiempo no recibe una mejor oferta, el robot gana la subasta. Una vez que se conoce el robot ganador y se le asigna un objetivo, los demás robots recalculan su oferta y se repite el proceso.
El problema de la comunicación se resuelve ya que cada robot se puede comunicar directamente con otro robot dentro de su rango de comunicación o con un robot remoto dentro de la misma subred utilizando robots intermedios.

Otra opción posible para la tarea de planificación es el algoritmo MinPos [3]. Este le asigna a cada robot aquella frontera para la cual el mismo tiene mejor posición. Y se define posición de un robot con respecto de una frontera, como la cantidad de otros robots más próximos a la frontera que él; y se considera tener una mejor posición, cuando el valor es menor. En caso de haber más de una frontera disponible según este primer criterio, se utiliza la mínima distancia como criterio de desempate. Éste método de asignación promueve la dispersión de los agentes que colaboran en el mapa; buscando una distribución espacial óptima al hacer que dos robots próximos luego de planificar se dirijan en direcciones distintas.

Algoritmo 1 MinPos

Entrada: R_i, C matriz de costos
Salida: $\alpha_i j$ asignación del robot R_i

1: **para todo** $F_i \in F$ **hacer**
2: $\quad P_i j = \sum_{\forall R_k \in R, k \neq i, C_k j < C_i j} 1$
3: **fin para**
4: $\alpha_i j = 1$ con $j = \underset{\forall F_j \in F}{\text{argmín}}\, P_i j$

Para el *algoritmo 1* se utiliza la siguiente notación:

- R el conjunto de robots, $R : R_1..R_n$ con $n = |R|$ el número total de robots
- F el conjunto de fronteras, $F : F_1..F_m$ con $m = |F|$ el número de fronteras
- C la matriz de costos con C_{ij} el costo asociado con la asignación del robot R_i a la frontera F_j

En su tesis de doctorado, Bautin [21] presenta también una variante del algoritmo MinPos, que intenta introducir las ventajas de las aproximaciones Greedy al problema, logrando un mejor equilibrio en la distribución la asignación entre robots y fronteras. Este algoritmo se llama MPG (MinPos Glouton) e introduce la consideración de las asignaciones de otros robots en la elección de la asignación propia. De esta forma al igual que con MinPos se asignará a un robot la frontera con mejor posición, pero que no haya sido aún asignada a otro robot.

2.5. Planificación de movimientos

Tanto para la asignación de tareas como para la navegación, saber el camino y el costo para ir de un punto a otro es clave. No solo es necesario encontrar un camino, sino que este tiene que ser el de costo mínimo para que la ejecución sea lo más eficiente posible.

Un algoritmo muy utilizado (en robótica, pero también en otras áreas de investigación donde los escenarios presentan un espacio de búsqueda de grandes dimensiones; por ejemplo, Inteligencia Artificial) para resolver este problema es A* [22] o también llamado A estrella. Esto se debe a que es un algoritmo sencillo y del cual se obtienen muy buenos resultados si se utiliza una heurística apropiada. Este algoritmo es una generalización del algoritmo Dijkstra, con la diferencia que A* hace uso de heurísticas para guiar la selección de vértices en el camino mediante la estimación de la distancia hasta el destino, buscando con esto disminuir el tamaño del subespacio explorado en el cálculo del camino óptimo. Y con esto logra que el algoritmo sea más rápido y eficiente.
Se considera el problema de encontrar el camino de menor costo para ir de s a t. Para esto, contamos con la función $\pi_t : V \to \mathbb{R}$ tal que $\pi_t(v)$ nos da un estimado de la distancia para ir de v a t y la función $d_s(v)$ que es la distancia exacta de s a v. En cada paso se va a elegir el v que tenga el menor $k(v) = d_s(v) + \pi_t(v)$. Para el cálculo de $\pi_t(v)$, se puede utilizar la distancia Manhattan que consiste en contar cuantas celdas nos separan del destino horizontal y verticalmente, sin movimientos en diagonal.

Otra solución al problema es el método de propagación de onda [3, 23] para construir el campo de potencial de cada frontera (agrupadas en clústers de fronteras contiguas), lo que da como resultado la distancia, y camino óptimo desde cada frontera a todos los

robots. Con esto, se computa la matriz de costos que utiliza un robot para desplazarse por el mapa. El cálculo utilizando este método no es costoso computacionalmente, y no necesita ser recalculado a menos que información significativa haya aparecido (se recibió actualización de algún otro robot, o se descubrió nueva región del mapa).
En una primera instancia, Bautin et al. [3] definieron una optimización a este algoritmo que consta de que la propagación de la onda desde las fronteras se detiene al encontrar al robot que computa el algoritmo, ya que a esa altura ya son conocidos todos aquellos robots que se encuentran más próximos que él de la frontera.
En un siguiente artículo de Bautin et al. [23] se define el método de propagación de onda simultáneo (SyWaP), de tal manera que permite reducir sustancialmente el tiempo de cómputo necesario para lograr construir los caminos de mínimo costo a las fronteras a explorar, así como decidir cuál es esta frontera óptima para el robot. El algoritmo de propagación de onda es en principio de orden O(n).

2.6. Toma de decisiones síncrona

La sincronización entre los robots es un problema muy importante al momento de considerar los algoritmos de planificación, ya que la gran mayoría de ellos asumen que todos los robots que colaboran planifican al mismo tiempo utilizando la misma información. Pero en la realidad esto rara vez es así, y para lograr que lo sea hay que incurrir en tiempos de ocio, lo que va en contra del objetivo de minimizar el tiempo total de exploración.

Una solución al problema es que los robots planifiquen cada vez que llegan a una frontera. En este caso, los robots han completado una tarea y necesitan una nueva para continuar su exploración. Si desearan sincronizarse y planificar todos a la vez y con la misma información, cada vez que un robot llega a una frontera debe esperar a que todos los demás también lo hagan, compartir la información recolectada por todos, de forma de que todos planifiquen con la misma información, y finalmente ejecutar los algoritmos de planificación.

Otra alternativa sería planificar cada un cierto delta de tiempo. Bajo esta condición, y asumiendo que todos los robots se encuentran con los relojes sincronizados, se puede asumir que todos los robots se van a encontrar planificando al mismo tiempo. En esta modalidad de sincronización, la decisión de que valor darle al delta de tiempo, es muy importante, y puede tener consecuencias notorias en la exploración. Si el delta es muy pequeño, los agentes van a planificar muy seguido, y posiblemente tomando la misma decisión numerosas veces; dado que la operación de planificación es usualmente intensiva en cómputo, y en muchos casos lenta, esto no es una situación deseable. Por otro lado, si el delta que se toma es muy grande, los agentes pueden tener que esperar ociosos ya

habiendo completado sus tareas hasta que el ciclo se cumpla; nuevamente esta situación atenta contra el objetivo de minimización del tiempo total de exploración.

En casos donde estas esperas ociosas no son aceptables, se debe analizar posibles compromisos a tomar para hacer posible sacrificar la sincronicidad de la ejecución de los algoritmos de planificación sin comprometer su integridad. Para esto se debe analizar la naturaleza del algoritmo que se va a utilizar, y ver si bajo ciertas condiciones, aun ejecutando fuera de sincronía, el resultado respeta los objetivos del mismo. Entonces, se podría por ejemplo tomar la decisión de comunicar información cada cierto delta de tiempo (o de información recabada), logrando así que todos los robots que se encuentren colaborando tengan mapas suficientemente sincronizados en todo instante de tiempo. Y luego, cada agente podría planificar inmediatamente después que completa una tarea, de forma de lograr reducir a cero los tiempos de ocio. En caso de que las condiciones anteriores no fuesen suficientes, se podría modificar levemente algún algoritmo para que también acepte como entrada la asignación de tareas del resto de los agentes, y con esto simular mejor que todos los agentes ejecutan sincrónicamente.

Asi como se mencionó en el párrafo anterior que compartir información en forma periódica puede permitir sacrificar la sincronicidad (ejecutar en forma asíncrona), se podría optar también por una solución intermedia donde se hace el esfuerzo de esperar un tiempo X prudencial para que los otros robots completen sus tareas y poder todos planificar en forma sincrónica. Y de acabarse el tiempo de espera, planificar bajo la condición que se presente en ese momento. Esta alternativa permite elegir un valor de X que no represente un tiempo de ocio que perjudique seriamente el tiempo de exploración, a la vez que mejora las chances de ejecutar los algoritmos de planificación en condiciones más próximas a las óptimas. El caso descrito en el párrafo anterior se podría ver como un caso particular dónde ese tiempo de espera es cero.

Finalmente, así como se mencionó la importancia de la sincronización para la ejecución de los algoritmos, se mencionó también otra componente fundamental para la sincronización; que es el compartir e integrar la información obtenida del ambiente. Y para este caso nuevamente se debe analizar con qué frecuencia y en qué momentos compartir la información. Si se comparte información frecuentemente, se logra una mejor sincronía de la información, pero es posible que se sobrecargue la red o utilice demasiados recursos de cómputo en tareas de integración de los datos perjudicando el desempeño de otras tareas. Si se comparte información en forma muy esporádica o solo al completar tareas, por otro lado, se favorece el ahorro recursos, pero es posible que se esté perjudicando la sincronía de los datos entre los agentes; generándose posibles problemas en la asignación de tareas o toma de decisiones, así como posiblemente asignando la misma tarea o tareas similares a más de un miembro del equipo.

A la hora de decidir qué tipo de sincronización utilizar, es importante tener en cuenta que generalmente, la opción ideal va a ser tener sincronización total entre los robots,

desde el punto de vista de los datos y los algoritmos; planificando todos al mismo tiempo y con los mismos datos. Esto debido a que los algoritmos de planificación distribuidos usualmente buscan elegir la tarea ideal para el agente que planifica, asumiendo la planificación que realizarían el resto de los agentes que colaboran; pero para esto se asume también que ellos están planificando en el mismo momento. Pero esto es costoso ya que tiene el potencial de generar tiempo ocioso como ya se mencionó. Hay que tener en cuenta, que a medida que se disminuye el nivel de sincronía de los robots, es necesario evaluar si no se está perjudicando la coordinación entre ellos, y que se siguen manteniendo las propiedades que hacen deseable al algoritmo que está siendo utilizado. De lo contrario, los beneficios que se mencionaron previamente de la coordinación se ponen en riesgo.

2.7. Localización

Para localizar al robot en el entorno y así poder navegar y generar un mapa del mismo hay dos métodos: posicionamiento relativo y posicionamiento global. En el caso de posicionamiento relativo se utiliza la idea de navegación inercial. Esta se basa en el giroscopio y en el acelerómetro para estimar la posición actual a partir de una posición anterior en la que se encontraba. Cuando el robot comienza a moverse se le determina una posición inicial estimada y en base a esta y la información de los sensores, va calculando la posición actual. El grave problema de este método es el error que se acumula en cada estimación de la posición.

Por otro lado, hay diversas maneras de obtener la posición global actual del robot, sin necesidad de conocer cuál era su posición anterior o desde dónde partió. Algunas de las más comunes pueden ser el GPS o una cámara que observe todo el escenario y pueda decir en qué parte del mismo me encuentro.

Otra opción, que se puede utilizar en escenarios donde al menos uno de sus límites es conocido es la trilateración [24]; en la cual se colocan 2 o más puntos de referencia (mojones o faros transmisores/receptores) en posiciones conocidas, y el robot se comunica con ellos para calcular su posición usando triangulación. Este tipo de solución, que logra muy buenos resultados en cuanto a precisión de la posición obtenida, se puede realizar en escenarios no muy grandes en forma relativamente económica utilizando beacons, y en ambientes de mayores dimensiones utilizando cualquier tipo de transmisor/receptor de señales de radio.

2.8. SLAM

La sigla en inglés significa mapeo y localización simultáneos. Como se puede observar en la figura 1.1, SLAM [1] se encuentra en la intersección entre los sub-problemas mapeo y

localización de la exploración cooperativa; y lo que modela es la interacción entre ambos. Mapping es el problema de integrar toda la información recogida por los sensores del robot en una única representación. Los aspectos centrales de este problema son cómo representar el entorno y cómo interpretar la información brindada por los sensores. Por otro lado, la localización se encarga de comunicarle al robot en qué posición se encuentra. Por lo dicho anteriormente, SLAM es el problema de construir un mapa al mismo tiempo que el robot se localiza dentro de ese mapa. Esto es muy importante ya que es la base para crear buenos mapas como resultado de la exploración. Esto se debe a que cuando se realiza una observación del entorno, es necesario saber dónde estoy localizado en el mapa y por otro lado es difícil localizar algo (darle significado a la posición calculada o recibida desde un sensor) sin tener un mapa donde posicionarlo. Este ha sido un tema muy estudiado por la comunidad de la robótica, por lo que, entre otras iniciativas, se creó una página web [1] dónde la gente puede subir sus algoritmos para resolver dicho problema.

[1] `http://www.openslam.org/`

Capítulo 3

Descripción de la solución

En este capítulo se presentan los requerimientos del proyecto, se realiza un análisis de las características del problema y se discute el diseño de la solución propuesta, así como las decisiones tomadas en la ejecución de la misma.

3.1. Requerimientos

Se requiere implementar un sistema robótico multiagente heterogéneo, capaz de colaborar en la tarea de explorar un área desconocida y generar un mapa común de la misma. Para esto es necesario el desarrollo de los componentes que controlan los distintos robots, navegar en el ambiente, resolver problemas de planificación, controladores de sensores y actuadores, generación y mantenimiento de estructuras de datos (mapa) y comunicación.

El sistema deberá ser ejecutado en una plataforma física utilizando el robot Butiá y en una simulada que represente el robot, los sensores y actuadores utilizados en la plataforma física. En ambos casos es necesario que tanto la cantidad de robots como el escenario donde corren puedan ser variables.

El problema de localización de los agentes no va a ser abordado como parte del proyecto, por lo que se asumirá, tanto en la simulación como en el robot real, que existe un sensor que brinda la información relativa a la posición global y orientación del robot en todo momento.

3.2. Análisis y Diseño de la Solución

El problema a resolver plantea el requerimiento de ser compatible con agentes heterogéneos; este hecho es relevante ya que robots con diferentes características físicas van

a poder acceder y navegar en diferentes zonas del espacio explorado. Al momento de implementar los distintos algoritmos que planifiquen las tareas a asignar a un robot, o decidan por dónde debe este navegar; se debe buscar que los mismos sean independientes de este aspecto, de forma de no tener que reimplementar componentes o algoritmos para cada uno, o cada vez que un agente nuevo con fisionomía distinta sea agregado al grupo. Esto implica la necesidad de buscar formas de hacer la implementación suficientemente genérica o parametrizada para adaptarse fácilmente a cada circunstancia.

Es importante probar la solución realizada en robots y escenarios reales, de forma de validar los resultados teóricos en la práctica, y bajo condiciones reales. De igual forma, también es de importancia lograr validar y probar los distintos algoritmos e implementaciones en ambientes simulados, ya que bajo estas condiciones son más simples y ágiles los procesos de desarrollo, búsqueda de errores, y realización de pruebas. A la vez, otra ventaja que presenta la simulación es que permite fácilmente replicar escenarios y condiciones de exploración de forma que se pueda comparar y generar puntos de referencia, para evaluar el desempeño de la implementación realizada con la de terceros.

3.2.1. Orocos

Orocos (Open Robot Control Software) [25] es un framework para desarrollo de componentes para agentes robóticos. El mismo cuenta con librerías y herramientas que simplifican el diseño y desarrollo de sistemas robóticos, brindando todo lo necesario para poder abstraerse del diseño de la arquitectura del software y poder focalizarse en el desarrollo de la funcionalidad necesaria. Orocos utiliza una arquitectura de DataFlow orientada a componentes especializadas que se comunican entre ellas para en conjunto brindar la funcionalidad final del robot.

3.2.2. Propiedades

Dadas las condiciones de heterogeneidad de los robots, y las distintas pruebas que son necesarias realizar para poder evaluar el desempeño de la solución, el sistema tiene que poder adaptarse con facilidad sin la necesidad de tener una versión distinta especializada en cada uno de los potenciales escenarios. Para lograr satisfacer este requerimiento se decidió mantener un archivo de configuración que se lee al iniciar el sistema, dónde se configuran las distintas propiedades que definen el comportamiento deseado. Esto también implica que el sistema debe estar parametrizado en forma apropiada para poder ajustarse y responder a cada una de estas propiedades en la forma deseada. Se decidió utilizar una librería de *PropertyBag* para la implementación, y un archivo de configuración en formato XML.

3.2.3. Arquitectura

Como se mencionó en párrafos anteriores, al utilizar el *framework* Orocos para el desarrollo del sistema, algunas decisiones sobre el diseño de la arquitectura vienen dadas por el mismo. Específicamente, Orocos fomenta el diseño del sistema usando una arquitectura de *DataFlow*, donde existen componentes especializadas en tareas, que se comunican entre ellas. De esta forma, una componente origina una acción (por ejemplo, una componente de sensado recibe información del ambiente) y la comunica a otras componentes; esta información, o nuevos datos generados a partir del procesamiento de los recibidos, es procesada por las componentes que la reciben y comunicada a otras componentes, hasta llegar a un destino.
Una arquitectura de este estilo es especialmente buena en robótica, donde usualmente existen muchas tareas específicas que se pueden modularizar y ejecutar en forma independiente de las demás. Se pueden tener componentes especializadas en cada uno de los tipos distintos de sensores y actuadores con los que cuente el agente robótico, así como también componentes encargadas de tareas de planificación, navegación, coordinación, detección de obstáculos, o cualquier otra tarea relevante para el sistema.
Para la solución planteada se decidió contar con componentes para el sensor de rango, gestión de los motores, localización, comunicación, planificación, navegación, exploración/coordinación y gestión del mapa los cuales se detallan a continuación.

3.2.3.1. Sensores y Actuadores

Para poder generar un mapa en conjunto con otros agentes, es necesario contar con un sistema de referencia global, de forma de ser capaces de correlacionar la información recolectada por distintos individuos. Para esto se cuenta con una componente de localización, responsable de acceder a un sensor o sistema de localización externo responsable de brindar la posición y orientación de un robot en cualquier momento durante la exploración. Esta componente corre en cada robot como un proceso independiente, que con una frecuencia elevada y constante se actualiza con el sistema externo (o sensor) y se encarga de dejar esta información disponible a el resto de las componentes del sistema en todo momento. Para el sistema simulado, la componente se conecta a un sensor GPS de alta precisión. En el sistema real, un GPS no es una solución aceptable debido a que no alcanza la precisión necesaria, especialmente en ambientes cerrados. Una alternativa de bajo costo y con precisión aceptable, como se discute en la sección de Estado del Arte, sería utilizar receptores y emisores de radiofrecuencia -como podrían ser beacons' y luego mediante trilateración calcular la posición global. Debido a que el problema de localización global se dejó fuera del alcance del presente trabajo, en su lugar se optó por utilizar un sistema de cámaras con visión de todo el escenario y figuras adheridas a los robots para poder simular un sistema de localización de alta fidelidad.

Otra pieza fundamental para la generación de mapas es poder percibir información del ambiente; y para lograr hacerlo en forma eficiente es necesario contar con un sensor apropiado. En el caso básico donde simplemente se desea generar un mapa basado en obstáculos (objetos, paredes, etc.) sin un análisis más profundo de ellos, un sensor de distancia es suficiente para la tarea. Pero para que el mismo permita reconocer en forma eficiente el entorno del robot, y al decir eficiente nos referimos a eficiencia en tiempo, el sensor debe ser capaz de realizar una lectura lo más próximo posible a 360 grados en torno al robot. De no ser así, esto implicaría la necesidad de realizar un mayor número de lecturas, o incluso de tener que detener al robot y hacerlo girar sobre si mismo para poder realizar la tarea. Con todas estas consideraciones en cuenta, se decidió el uso de un sensor de rango 360 RPLidar [26] que consiste en un sensor láser de distancia adherido a un motor que gira a frecuencia elevada, y transmite las lecturas de todas las distancias juntos con los ángulos correspondientes a dónde fueron medidas. El sistema robótico cuenta con una componente responsable de comunicarse con este sensor y controlarlo, así como pre-procesar la información recolectada del mismo para compartirla con otras componentes del sistema en forma de una observación en la representación discretizada que se utiliza para modelar el entorno.

La tercera componente fundamental para la exploración es la que permite al robot trasladarse por el ambiente para llevar a cabo sus tareas. En este caso en que se utiliza un robot Butiá para la exploración, se sabe de antemano que el robot es un vehículo sobre ruedas con dos motores para propulsarlas. Nuevamente el sistema cuenta con una componente de software para controlar la tarea. La misma es responsable de mantener los motores funcionando en forma sincronizada; pero a la vez se encarga de decidir qué acciones tomar para trasladar al robot desde la posición actual hasta el punto de destino que recibe. La componente de motor abstrae el *hardware* que hay por debajo, para que el resto de las componentes puedan realizar sus tareas libremente y simplemente le comunican a los motores los puntos de destino a los que desean llegar, con el compromiso de brindar pequeños deltas de navegación, libres de obstáculos. De esta forma la componente de motores se encarga de rotar al robot para posicionarse en dirección a su objetivo, desplazarlo y detenerse una vez que el objetivo es alcanzado.

3.2.3.2. Navegación

La componente de navegación es responsable de controlar todos los aspectos de la navegación, como ser: decisión de caminos, detección y evasión de obstáculos, aborto temprano de tareas y solicitar un nuevo plan y comunicación con motores. El ciclo de vida de la componente cuenta esencialmente de dos estados: navegando y reposo. Cuando el agente no se encuentra navegando hacia un objetivo, el mismo solicita una nueva tarea al planificador mientras espera en estado de reposo. Al recibir un nuevo objetivo, primero se calcula el camino que se desea seguir, para luego traducirlo en pasos

a dar que serán comunicado a los motores. Mientras el robot se encuentre navegando, se controlará constantemente que los pasos a dar sean posibles, que el objetivo aún sea válido, y que no haya obstáculos que requieran recalcular el camino o incluso replanificar. A la vez que se realizan estas tareas, se va llevando registro de ciertas métricas, que van a ayudar a definir los mejores momentos para compartir la información recolectada con otros robots del grupo, así como también integrar la información recibida con la propia. Estos momentos estarán definidos bien por la finalización de una tarea, o por deltas de desplazamiento durante la ejecución de un objetivo. Esta última métrica también se puede ver afectada por obstáculos encontrados.

3.2.3.3. Planificación

La componente dedicada a planificación recibe pedidos de otra componente para calcular una nueva tarea a asignar. Se encarga de definir las fronteras existentes en el mapa más actualizado, agruparlas de forma de definir zonas de frontera, y finalmente, en base a estas zonas de fronteras, y las posiciones más actualizadas de los robots, ejecuta el algoritmo MinPos (en su versión greedy -MPG-) para calcular la nueva tarea a asignar al robot.

3.2.3.4. Comunicación

En el caso de la componente de la comunicación, es la encargada de la comunicación con los otros robots para compartir los mapas y las posiciones de cada uno. Para esto se utiliza el middleware YARP, el que se encarga de conectar los puertos y enviar/recibir los datos. Se utilizaron puertos multicast, donde un mismo puerto puede estar conectado a más de uno, lo que facilita la implementación ya que cada robot cuenta con un único puerto de salida y uno de entrada a la cual se conectan los puertos de los otros robots, sin importar cuantos sean. En el caso del puerto de entrada, se utiliza un buffer circular donde se van almacenando los datos enviados por los otros robots hasta ser leídos.
Dado que se puede tener problemas en la conectividad de la red, es importante agregarle a cada mensaje un *timestamp* para verificar si los mensajes llegaron en orden o si se perdió conexión y los datos están desactualizados.

3.2.3.5. Mapa

Finalmente se define una componente de gestión de mapa, que se va a encargar de mantener la representación local del espacio siendo explorado. Esta componente se responsabiliza de integrar nuevas observaciones al modelo, incorporar los datos de mapas recibidos desde otros robots a la representación local, y de entregar una copia actualizada del mapa generado a las componentes que se lo requieran.

3.2.4. Estructuras de datos

Para la representación y manejo del mapa se utilizó una grilla de ocupación. Como ya se describió en la sección 2.2, cada celda de estas grillas es una probabilidad que nos dice si esa celda es desconocida, ocupada o libre.

3.3. Implementación: decisiones tomadas

En esta sección se detalla cómo se implementó cada componente nombrada en la sección anterior y cada requerimiento, así como también las distintas decisiones tomadas y el porqué de las mismas.

Como se detalla en la sección 3.2.3 y se puede observar en la figura 3.1, el sistema cuenta con las componentes: sensor de rango, gestión de los motores, localización, comunicación, planificación, navegación, exploración/coordinación y gestión del mapa. Para implementar estas componentes se utilizó el método de programación de inversión de control (IoC) [27]; este se basa en la delegación de las tareas hacia otros componentes en los que se espera una respuesta dada pero no se controla cómo se llevan a cabo las mismas. Un ejemplo de esto es el componente de lo motores, desde la navegación se le dice hacia donde se tiene que trasladar y el componente se encarga de hacerlo. Esto permite modularizar el sistema, haciendo más sencilla la tarea de portar los componentes para distintas plataformas y dando lugar al patrón de diseño de la plantilla (*Template*) [27]. Este diseño permite tener una clase base en la que se implementan las funcionalidades que tienen en común y luego tener subclases de esta que implementen las funcionalidades más específicas. Esto es usado para los componentes: sensor de rango, motores y localización; estos componentes, según la plataforma en la que se esté corriendo van a necesitar una implementación distinta. Tomando como ejemplo el componente del sensor de rango, la manera de solicitar los datos del sensado en la simulación es totalmente distinta a como lo es en el robot físico, pero esta diferencia debe de ser transparente para el resto del sistema. Es aquí cuando se tiene una clase que funciona como plantilla que es con la que se comunican los componentes externos; esta se encarga de recibir el pedido de una nueva lectura, procesar la solicitud, darle el formato esperado por los otros componentes y retornar el mismo. En este caso en particular, la plantilla tiene una función para obtener los datos del sensor que luego es sobrecargada por las distintas subclases que implementan esta plantilla.

Las clases que implementan estos componentes del sistema son una extensión de la clase *RTT::TaskContext* de Orocos, lo que nos permite utilizar los servicios, puertos y funciones que provee Orocos para el manejo de componentes. Las funciones principales que provee la clase son:

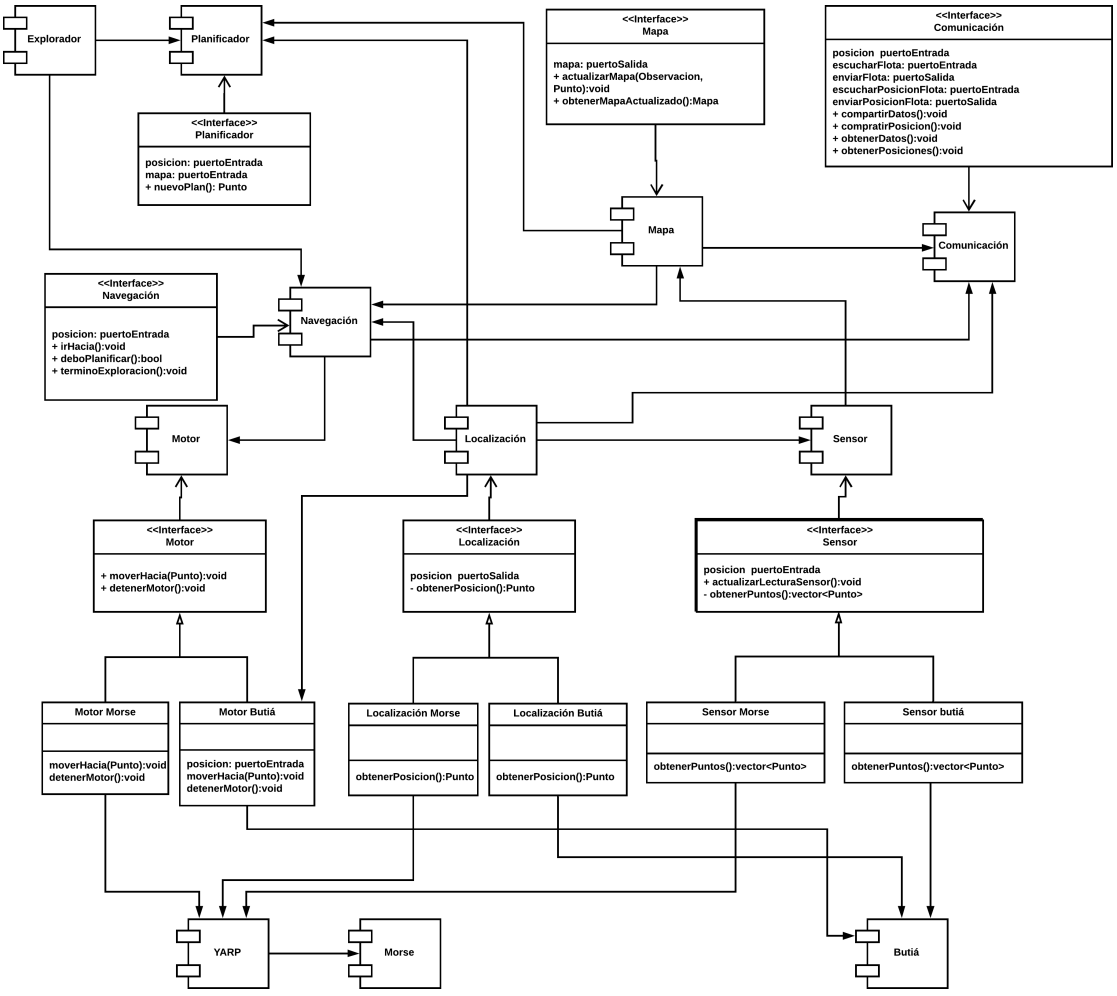


FIGURA 3.1: Diagrama de componentes del sistema con las interfaces con los puertos y operaciones publicadas de cada uno.

- *configureHook*: se verifica que el componente esté correctamente configurado y listo para arrancar. Por ejemplo, se verifica que todos los puertos estén correctamente conectados.
- *startHook*: Se realizan todas las configuraciones necesarias al darle inicio al componente. Por ejemplo, se conectan con los servicios provistos por otros componentes.
- *updateHook*: Esta función corre cada cierto período, configurado en los scripts de Orocos (se hablará de ellos más adelante) y se encargan de realizar todas las tareas recurrentes que ejecute el componente.

Por otro lado, nos permite publicar operaciones para que otros componentes las puedan utilizar en forma de servicios (ver la figura 3.2), así como también puertos de entrada y salida para la comunicación; ambos tipos de comunicación se detallarán más adelante en la sección 3.3.4. Como se mencionó anteriormente, Orocos cuenta con *scripts* que se encargan de la inicialización, configuración y comienzo del programa. En estos se cargan los componentes, se configura cada cuanto deben correr, se conectan los puertos (ya sean

FIGURA 3.2: Diagrama de componentes de un RTT::TaskContext mostrando las opciones que proporciona para comunicarse con otros componentes.

internos o externos) y los componentes entre sí, se llama a la función de configuración de cada componente y se le da comienzo a los mismos.

3.3.1. Heterogeneidad

Uno de los objetivos del proyecto es que el grupo de robots pueda ser heterogéneo, por lo cual puede haber distintos robots con distintas formas y tamaños. En la solución a este problema, una parte importante es tener las propiedades como un archivo XML, independiente del proyecto como se detalla en la sección 3.2.2 y que así cada robot pueda tener su archivo con sus propiedades independientes del resto. Para este problema, lo que nos interesan son las propiedades de ancho y alto que inscriben el robot en un rectángulo con centro el centro del robot y de esta manera cada robot tiene sus propios valores y la implementación se realiza en base a ellos. Como se detalla más adelante, en las secciones 3.3.5 y 3.3.6, las dimensiones del robot se tienen en cuenta a la hora de la búsqueda de caminos para poder determinar si un robot puede pasar por determinado pasillo como también para mantener un margen de distancia con las paredes para evitar estancamientos. En estos algoritmos se utiliza el diámetro del robot, inscribiendo al mismo en un círculo de radio medio diámetro y así independizar la implementación del tamaño y forma del mismo.

3.3.2. Estructuras de datos y Librerías

Como ya se nombró en la sección 3.2.4, para la representación del mapa se utilizaron grillas de ocupación o grillas probabilistas. Para facilitar esta tarea se utilizó MRPT (Mobile Robot Programming Toolkit) [28], el cual es un conjunto de librerías y aplicaciones listas para el uso. En nuestro caso se utilizó dentro del módulo *maps*, el componente *maps::COccupancyGridMap2D* [29]. En esta implementación de las grillas de ocupación se utilizan los valores: 0 para ocupado, 1 para libre y 0.5 para desconocido. Por motivos de optimización, MRPT utiliza el concepto de *log-odds* [6] para almacenar los estados

intermedios de las celdas, donde

$$Odds(x) = p(x)/(1 - p(x)) \tag{3.1}$$

y en caso de necesitar el valor de la probabilidad esta es fácilmente calculable mediante la fórmula

$$p(l) = Odds(l)/(1 + Odds(l)) \tag{3.2}$$

siendo l el valor *log-odds* de la celda. Para hacer este manejo de probabilidades más sencillo, MRPT tiene una tabla donde están almacenados los valores de probabilidades y de *log-odds* haciendo estas conversiones más sencillas. Este módulo no solo se encarga de la representación del mapa sino también del mantenimiento, ya que se le puede agregar una observación -una lectura del sensor- y MRPT se encarga de actualizar la grilla con los nuevos valores de probabilidad de cada celda. Para facilitar el manejo de los datos, también se utiliza el paquete para almacenar la posición del robot utilizando el módulo *poses::CPose2D* que guarda el vector (x, y, phi).
Como se especificó en el párrafo anterior, las clases *mrpt::maps::COccupancyGridMap2D* y *mrpt::poses::CPose2D* se utilizaron a lo largo del sistema para unificar el manejo de los datos de mapas y posiciones. Dado que MRPT utiliza la librería Eigen para el manejo del álgebra de matrices y vectores, en algunas arquitecturas se tiene un error de alineación en tiempos de ejecución a la hora de crear una nueva instancia de manera dinámica de estas clases que son de tamaño fijo como se detalla en el sitio web [1]. Es por esto que se crearon dos clases: *BPose* y *BMap*; estas clases son un simple contenedor que extienden las clases de MRPT a las que se le agrega la macro *EIGEN_MAKE_ALIGNED_OPERATOR_NEW* como dice la documentación para que, a la hora de crear una nueva instancia, siempre se retorne un vector alineado.

Tanto para la comunicación entre robots como para la recepción de los datos de posición enviados en la simulación por MORSE (se verá en la sección 3.3.7), fue necesario utilizar la librería RapidJSON [30] para generar o leer un *JSON* (JavaScript Object Notation). En el caso de la comunicación entre robots, es necesario enviar la información en formato JSON ya que, si se utilizaba la serialización nativa de la librería, cada instancia de comunicación para una grilla 160x160 implicaba 26MB de información contra 0,6MB cuando es JSON. Para realizar esto se utilizó la función que brinda MRPT para transformar el mapa de *maps::COccupancyGridMap2D* a vector y viceversa.

3.3.3. Mapas

A la hora del manejo de mapas hay algunas preguntas a responder: ¿Cuándo y cómo integro los valores leídos por el sensor? ¿Cuándo envío mi mapa a los otros robots? ¿Cuándo y cómo integro mapas de los demás?

[1] `http://eigen.tuxfamily.org/dox-devel/group__TopicStructHavingEigenMembers.html`

Para responder la primera pregunta, ¿Cuándo y cómo integro los valores del sensor al mapa? El cómo ya se contestó en la sección 3.3.2, utilizando la librería MRPT. Para responder el cuándo hay que tener en cuenta el escenario que se va a estar sensando y la frecuencia a la que se va a sensar. Si se observa la figura 3.3 se puede observar como con la frecuencia adecuada, la pérdida de información es muy poca en comparación a la ganancia de cómputo que hay. Por otro lado, se puede observar en la figura 3.4 que, en un escenario más complejo, donde la frecuencia de actualización no es la adecuada, la pérdida de información es mayor ya que el pasillo queda prácticamente sin ser descubierto. En este proyecto, se observó que como el rango del sensor es mayor al tamaño de una celda y se avanza celda a celda, con integrar cuando se llega a una celda nueva es suficiente y se encuentra en el caso de la figura 3.3.

FIGURA 3.3: Las circunferencias grises marcan el área sensada por el robot al llegar a un objetivo y en la imagen de la derecha se muestra en rojo la diferencia respecto a sensar constantemente [21]

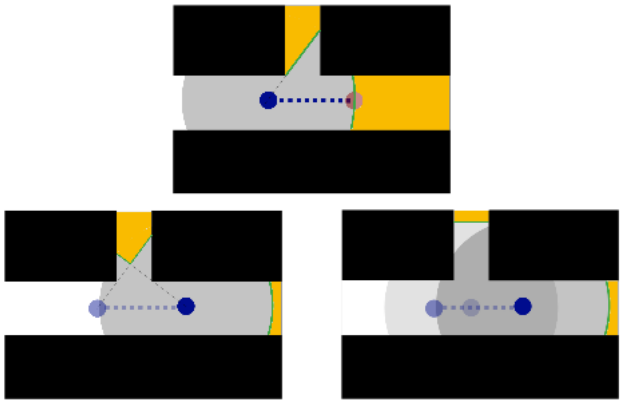


FIGURA 3.4: Al igual que la imagen anterior, lo gris marca lo sensado por el robot, la línea verde es el límite y lo amarillo lo desconocido. Este es un ejemplo en un escenario más complejo y se puede observar en las imágenes de abajo la diferencia entre una baja frecuencia de sensado (izquierda) y una alta (derecha). En este caso la pérdida de información acerca del pasillo es notoria. [21]

¿Cuándo envío mi mapa a los otros robots? En este caso también es importante la frecuencia a la que se envía el mapa porque si la frecuencia es muy alta, se corre el riesgo de sobrecargar la red, pero si la frecuencia es baja, los demás no van a tener la información que yo ya descubrí y se corre el riesgo que otro explore lo que yo ya exploré por desconocimiento. La decisión en este proyecto es realizar el envío de los mapas según dos condiciones: cuando se llega a un objetivo y al avanzar más de una

determinada cantidad de pasos equivalentes al rango del sensor ya que se considera que la ganancia de información es suficiente como para comunicarla.

Por último, queda contestar la pregunta: ¿Cuándo y cómo integro mapas de los demás? Los mapas de los otros robots son integrados a medida que van llegando, esto es para siempre estar con la información actualizada. Uno de los problemas de este enfoque es que puedo sobrecargar mi sistema al hacer integrar constantemente mapas, pero dada la precondición de que la frecuencia de envío de los mapas va a ser adecuada, como se detalló en el párrafo anterior, esto no debería de ser una preocupación para este sistema. Para el cómo se utilizan dos funciones de MRPT sobre los mapas: *setCell* que fuerza el valor de una celda y *updateCell* que integra el nuevo valor a la celda utilizando las probabilidades. El enfoque que se tomó es que para integrar un valor de una celda del mapa de otro robot, esta tiene que tener cierta certeza, no se integran valores cercanos o iguales a desconocido. Para el ejemplo vamos a suponer un robot A que quiere integrar el mapa de un vector de robots $B = B_0, B_1..B_n$. Dada la celda i, si A_i es desconocido (0,5), realizo un *setCell* con el promedio de todas las celdas i de los mapas del vector B que no sean desconocidas. En el caso que A_i sea distinto de 0,5, por cada robot del vector B, si la celda $i > 0{,}8$ hago un *updateCell* con 1 integrándolo como una celda libre, si la celda $i < 0{,}2$ hago un *updateCell* con 0 integrándolo como una celda ocupada, por el contrario, si el valor de la celda i tiene incertidumbre, no se integra al mapa del robot A.

3.3.4. Comunicación

En este proyecto se utilizó la comunicación en distintas instancias: comunicación interna de los componentes de Orocos, comunicación entre los robots, y comunicación de los componentes externos (posición, GPS, motores) a el robot ya sea en el caso de simulación como en el caso del robot físico.
En el caso de la comunicación entre los componentes Orocos se utilizaron dos métodos: la comunicación por puertos y los servicios. La comunicación por puertos se utilizó para la posición y el mapa que son accedidos por distintos componentes en distintos momentos y que son actualizados de manera asíncrona, por lo que es útil publicar los valores en un puerto y que los componentes lean de ellos según lo precisen. Ambos puertos implementan un buffer de tamaño una unidad y cuando se producen datos a mayor velocidad de la que son consumidos, el nuevo dato sobrescribe al viejo, por lo que siempre se encontrará disponible el valor más actualizado. Por otro lado, los servicios fueron utilizados para publicar funciones puntuales como por ejemplo enviarle la instrucción de avanzar a los motores. Estos servicios son bloqueantes por lo que se espera la respuesta de los mismos para continuar la ejecución.
Como se mencionó, los puertos son útiles para el caso en el que hay que acceder datos de manera asíncrona, que una componente actualiza y los otros acceden pero, para

el caso de envío de instrucciones o petición de datos estos no sirven dado que no son instantáneos, puede haber una demora en la escritura/lectura de los puertos y también ocasionalmente se pueden perder mensajes.

Por otro lado, para el caso de la comunicación entre robots (ya sea en la plataforma simulada como en la física) y en la comunicación desde MORSE hacia el robot simulado se utilizó YARP (Yet Another Robot Platform) [31], este es un *middleware* para facilitar la comunicación en plataformas robóticas y ofrece librerías y complementos para MORSE y Orocos que son muy sencillos de utilizar. Inicialmente se tuvo la intención de implementar la comunicación entre robots mediante el método broadcast de mensajes, pero el *middleware* no lo soporta. Debido a esto se cambió el enfoque y en su lugar se utilizó *multicast* para la comunicación, mecanismo soportado en forma nativa que permite comunicar los datos y que todos aquellos robots conectados a la misma subred reciban la información.
Por último, como se muestra en la figura 3.5, en el caso del robot físico para la posición se utilizaron *sockets*. Se tiene una PC que funciona como servidor que le envía la información mediante *sockets* al robot.

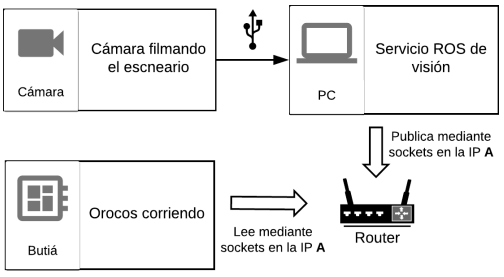


FIGURA 3.5: Diagrama del funcionamiento del sistema de visión para el robot físico donde se tiene una cámara que está filmando el escenario, una PC que actúa como servidor y le comunica al butiá por *socket* la posición actual.

3.3.5. Planificación

Como se mencionó en la sección 2.4, el algoritmo de MinPos [3] tiene varias implementaciones. La más pura es la que utiliza $A*$ como algoritmo para cálculo de camino de menor costo entre las fronteras y los robots. Esta implementación es eficiente cuando se trabajaba con un número bajo de robots y fronteras, pero en cuanto los mismos comienzan a aumentar aparecen problemas de desempeño, lo que hace que no sea una solución apropiada para el problema a resolver en este proyecto. Esto se explica dado que el algoritmo MinPos tiene una complejidad $O(nm)$ -n siendo el número de recursos, y m el número de objetivos- y para cada una de esas instancias se debe ejecutar $A*$ con complejidad $O(p)$ -siendo p el número de celdas del mapa-. Y teniendo en definitiva una complejidad final $O(nmp)$.
En base a los problemas de desempeño del algoritmo, se decidió hacer uso de la implementación alternativa sugerida por Bautín mediante propagación de ondas en paralelo

[23], que permite simplificar la complejidad final independizando el cálculo del número de robots, teniendo complejidad $O(mp)$ en peor caso. Al además utilizar la versión paralelizada con pausado de ondas, permite interrumpir la ejecución del algoritmo en forma temprana al encontrar la solución, sin necesidad de computar todos los caminos desde todas las fronteras a todos los robots; mejorando sustancialmente los tiempos de ejecución. Con este cambio los tiempos de ejecución de los algoritmos son similares en los mejores casos, pero para los peores (número elevado de robots y fronteras) los tiempos mejoran sustancialmente.
Finalmente, un problema que presenta la versión más sencilla de MinPos es que, bajo ciertas circunstancias, es que puede dejar tareas sin asignar a ningún recurso, a la vez que asigna más de un recurso a una misma tarea. Este problema fue resuelto utilizando la variante MPG (MinPos Glouton -o Greedy-) del algoritmo -introducido por Bautín en su tesis de doctorado [21]-, que agrega una condición greedy para las asignaciones, que es las asignaciones de otros robots. Esta variante del algoritmo resuelve estos casos, a la vez que mantiene las propiedades de dispersión de MinPos. MPG tiene complejidad computacional $O(n^2m)$ pero al ser implementado con propagación de ondas en paralelo con pausa, nuevamente es posible independizarse del número de robots y lograr tener complejidad $O(mp)$.

Como se mencionó, unos de los parámetros que recibe el algoritmo de planificación es una lista de fronteras objetivos a asignar a los robots. Para generar esta lista se recorre el mapa validando cuáles de las celdas definen el límite entre zonas conocidas e inexploradas del mapa, a la vez que son alcanzables por el robot. Pero si se hiciese uso de esta lista de celdas como la entrada para el algoritmo de planificación se podrían presentar numerosos problemas. El más evidente es el impacto en el desempeño; si la lista de celdas no es pre-procesada para reducir su número, los tiempos de ejecución del algoritmo aumentan sustancialmente. Otro problema muy importante que puede surgir de no pre-procesar en forma correcta las fronteras de entrada para la planificación, es el impacto en la calidad de la solución, donde varios recursos pueden resultar asignados a objetivos muy próximos.
Para evitar los mencionados problemas, el preprocesamiento que se debe realizar es agrupar las celdas de forma que cada grupo represente una zona de frontera dentro de lo posible separada del resto de las zonas. Y para cada una de estas zonas se debe elegir una celda representativa de la misma; siendo este conjunto de celdas representativas las que se pasarán al algoritmo de planificación para servir de tareas a asignar a los recursos. El algoritmo de agrupación (*clusteurización*) más popular es k-means, y fue el primero en ser implementado para resolver el problema. K-means es un algoritmo de complejidad computacional relativamente baja, lo que lo hace ideal para este tipo de procesamiento; pero para el problema puntual que se está resolviendo presenta algunas desventajas. La principal de ellas es que k-means requiere conocer de antemano el número de grupos que se desea encontrar. Éste es un dato con el que no se cuenta en este problema, por lo que se deben probar varios valores, ejecutando k-means una y otra vez, y evaluando la

calidad de la solución final hasta encontrar una distribución satisfactoria de las celdas de frontera. Este proceso de probar valores y evaluar soluciones hace que la complejidad computacional final del algoritmo sea algo más elevada. Otra desventaja que presenta k-means, es que una vez definidos los grupos, el centro de referencia que define no necesariamente coincide con una de las celdas agrupadas. Esto resulta en que se deba realizar nuevamente otra tarea más para dentro de cada zona generada encontrar al mejor representante. Y aunque cada etapa de procesamiento adicional necesaria no es sustancialmente compleja, el resultado de sumar todas ellas resulta en un algoritmo de agrupación de complejidad media, que hace evidente esto al tener que procesar muchas celdas.
Conociendo estos problemas, e intentando buscar una alternativa que no sufriese de ninguno de estos problemas se encontró el algoritmo *Affinity Propagation*, algoritmo de aprendizaje no supervisado utilizado en el campo del Aprendizaje Automático para realizar agrupación de datos. El algoritmo requiere algo de experimentación para definir funciones a utilizar para calcular la relación entre los datos que se quieren clasificar. En el caso del problema que se busca resolver, esto es relativamente simple ya que la distancia entre las celdas es una buena medida de que tan relacionados deberían estar los puntos. *Affinity Propagation* es un algoritmo iterativo basado en el intercambio de mensajes entre los puntos a clasificar, buscando entre ellos mismos definir quién es un buen candidato para representar a cada otro punto. Como se dijo, este algoritmo no necesita conocer la cantidad de grupos a generar, sino que ellos se definen naturalmente basados en las fuerzas de las relaciones entre los elementos que los conforman. Tampoco necesita post-procesamieto de los grupos para encontrar un buen representante del mismo, ya que el propio algoritmo tiene como salida el conjunto de puntos ejemplares que definen los distintos grupos. Y sumado a estos factores, la complejidad computacional del algoritmo es comparable a la de k-means.
Como era de esperarse al comparar los tiempos de ejecución de ambas implementaciones contra distintos conjuntos y cantidades de celdas, *Affinity Propagation* resultó órdenes de magnitud más veloz que k-means; teniendo además un beneficio no esperado de antemano: la calidad de las soluciones generadas resultó mayor, dando mejor distribución de las fronteras.

En cuanto al manejo de los puntos de frontera y clústers, lo ideal para mejorar los tiempos de procesamiento es en vez de calcularlos cada vez que se quiere planificar, mantener una estructura en memoria, donde luego de cada lectura del sensor se actualice, quitando los puntos que ya no son fronteras o clústers y agregando los nuevos. Si esto no se hace, en el caso de las fronteras es necesario recorrer toda la estructura del mapa cada vez que se quiera planificar y extraer de ella los puntos de frontera actualizados mientras que si se guardan en una estructura simplemente hay que actualizar los puntos del mapa que quedaron dentro del área sensada. En el caso de los clústers si no se guarda la estructura es necesario correr el algoritmo de clusterización con todos los puntos de frontera por cada vez que se quiera planificar; por el otro lado, si se mantienen las

estructuras, simplemente se quitan los centros de clústers que incluían las fronteras que fueron descubiertas y se calculan los nuevos clústers para la nueva frontera descubierta. En la solución de este proyecto no fue posible implementarlo de esta manera, manteniendo las estructuras en memoria, ya que al utilizar MRPT para el manejo de las lecturas del sensor, no había forma sencilla de saber cuáles eran los nuevos puntos que habían sido descubiertos. Para poder implementarlo con las estructuras era necesario realizar un postprocesamiento de las lecturas del sensor, calculando que posiciones que fueron afectadas por los rayos, lo cual implicaba un procesamiento mayor.

3.3.6. Navegación

Una vez que el planificador decidió cual era el próximo destino, el componente de navegación toma el mando del sistema y hasta tanto no se llegue al destino es el encargado de controlar las acciones del robot. La primera decisión a tomar es, dado el nuevo objetivo, ¿cómo llego a él? Para esto se utiliza el algoritmo de $A*$ (sección 2.5) que nos devuelve los puntos intermedios para ir desde la posición actual al objetivo. Para el cálculo del camino, no solo se busca que este sea el más corto, sino que se le agregó una lógica extra al algoritmo para mantener al robot alejado de las paredes y así evitar estancamientos, especialmente a la hora de bordear una pared o de doblar en una esquina que son los lugares más complicados y donde se suele quedar trancado el robot.

Los pasos devueltos por $A*$ se ejecutan uno a uno, sin juntarlos ya que se observó que de esta manera se controlaba mejor el movimiento del robot tanto en MORSE como en el robot físico, evitando desviaciones. Antes de ejecutar un paso se verifica si el siguiente punto sigue siendo alcanzable, ya que puede haber cambiado una lectura previa que se había hecho de una sección y ahora pase a ser inalcanzable ese punto. En caso de que no sea alcanzable, se descarta ese paso y se continúa con el siguiente. En el peor de los casos se van a descartar todos los pasos y se va a volver a planificar. Esto es necesario ejecutarlo dado que no se cuenta con un algoritmo para evadir obstáculos y se estaría avanzando a ciegas, confiando de que el ambiente inicial en donde se planificó no cambia.

Si el siguiente paso es posible, se manda la acción a los motores y mientras que el robot está en movimiento, trasladándose de un punto a otro se ejecutan dos funciones: una que controla que el robot no se haya quedado trancado y otra para consultar si ya se llegó a la posición objetivo. Para controlar si el robot está trancado simplemente se guarda la posición anterior y si el robot no se movió más del 10 % del tamaño de una celda, se aumenta un contador. Si este contador aumenta más de un cierto número calculado empíricamente, se asume que el robot está trancado, se aborta el paso actual y se pasa al siguiente paso. Al hacer esto se puede correr con la suerte que el siguiente paso sea en otra dirección y el robot se desatranque, pero en caso de que esto no suceda, si ya se abortaron más de 5 pasos se considera que ese objetivo no es alcanzable, se aborta la navegación y se vuelve a planificar. Por otro lado, se considera que el robot

alcanzó el objetivo cuando se encuentra a menos de medio robot más un margen de seguridad de 10cm del objetivo. Este margen es necesario porque muchas veces el punto de frontera que se quiere alcanzar se encuentra muy cercano a una pared y si el robot se acerca mucho puede quedarse estancado; esto es algo que se busca evitar para favorecer la exploración.

Cada vez que se llega a un punto, se comparten el mapa y la posición a los otros robots y se realiza una lectura del sensor integrando la nueva información al mapa. Los puntos se integran siempre, no importa si esa zona ya era conocida o no dado que se trata de sistemas dinámicos donde las condiciones del ambiente pueden cambiar y siempre es necesario tener el mapa actualizado. Al integrar mapas de otros puede suceder que otro robot ya descubrió la frontera a la cual yo estaba yendo, por lo que cada cierta cantidad de pasos que equivalen al rango del sensor se vuelve a verificar si el objetivo sigue siendo una frontera y en caso de que ya no lo sea se abandona ese objetivo y se vuelve a planificar.

3.3.7. Simulación

Para simular tanto el ambiente como el robot con sus componentes se utilizó MORSE [32] con Blender [33] para la interfaz gráfica. La primera es una plataforma de simulación creada con fines educativos por lo que cuenta con una gran variedad de robots preestablecidos y de componentes, lo que hace que la configuración del mismo sea muy sencilla. Se centra en realizar simulaciones 3D realistas, soportando distintos tamaños de escenarios y desde uno a una gran cantidad de robots.
En nuestra implementación, para la simulación del robot se utilizó el iRobot ATVR, el cual se basa en el robot ATVR que distribuye I-Robot. Este difiere del robot Butiá en varios aspectos como la tracción ya que el ATVR cuenta con 4 ruedas mientras que el Butiá cuenta con 2 más una rueda loca y la forma del robot ya que el Butiá es triangular y este es rectangular, pero a efectos de simular los algoritmos y su comportamiento, estas diferencias no nos influyen y se puede utilizar este robot sin comprometer los resultados.
Para la navegación, se cuenta con un componente llamado *Waypoint* al cual se le envía la posición destino y MORSE se encarga de realizar los movimientos necesarios para que el robot alcance esa posición. Esta navegación no siempre es la esperada debido a que a veces tiene bastantes desviaciones lo que hace que una línea recta pase a ser un zigzag y esto hace más lenta la navegación por lo cual es importante controlarlo realizando pasos cortos si es necesario.
Por otro lado, para obtener la posición del robot respecto al ambiente se utilizó la componente *Pose*, la cual tiene como salida una terna (x,y,rotación) tomando como punto de referencia en donde se lo ubique, en nuestro caso se usó la posición por defecto que es el centro del escenario. Este componente es muy sencillo de utilizarlo ya que constantemente está publicando la información y lo único que hace falta hacer es leer la posición

cuando sea necesario. Este es un gran beneficio ya que si hubiese que pedirle al componente la posición y que luego este nos retorne se podrían generar algunas diferencias que harían que no tengamos la posición actualizada.
En el caso del sensor de rango, MORSE cuenta con un componente base, Laser Scanner Sensor, que es el que le otorga el comportamiento al componente y luego tiene 3 variedades del mismo, en nuestro caso se utilizó el *SICK*. Los sensores de rangos tienen varias propiedades que pueden ser personalizadas: el rango de lectura del sensor, la ventana de sensado en grados y la resolución también expresada en grados. En nuestro caso se utilizó una ventana de 360º, con un rango de 6m y una resolución de 1º para simular el sensor que se utilizó en la plataforma física.

En MORSE hay tres maneras de correr las simulaciones:

- levantar todos los robots en una misma instancia y de esta manera poder observar lo que está sucediendo con cada robot;
- correr multi-nodo, es la posibilidad de correr la simulación en una infraestructura distribuida, multi-nodo. La ventaja de correr MORSE de esta manera es que permite simular una gran cantidad de robots sin enlentecer ninguna instancia;
- correr headless, en este caso se corre morse sin interfaz gráfica (GUI). Esto es muy útil a la hora de correr las pruebas, ya que no es necesario estar viendo cada corrida.

Capítulo 4

Pruebas realizadas

En la siguiente sección se presentan las pruebas realizadas y los resultados de las mismas. Primero se describen las pruebas realizadas mediante simulación de escenarios, incluyendo los resultados globales de la exploración, así como las comparaciones entre versiones de algoritmos utilizados. Finalmente se describe la prueba de concepto realizada en un escenario real con el robot Butiá.

4.1. Pruebas simuladas

Como la figura 4.1 lo indica, para realizar las pruebas y el desarrollo en el ambiente simulado se utilizaron 4 escenarios, cada uno con una complejidad distinta. Estos escenarios fueron tomados del artículo [34] y son inspirados en los terrenos utilizados en la competición de rescate de RoboCup [35].

- ***Loop*** - Ese es el escenario más simple, representa una circunvalación, con baja densidad de obstáculos y sin bifurcación de caminos.
- ***Cross*** - Este escenario representa un cruce de caminos, también con una baja densidad de obstáculos y con 5 bifurcaciones de caminos.
- ***ZigZag*** - Este escenario no tiene bifurcaciones de caminos, pero si una mayor densidad de obstáculos. Como se puede observar hay un único camino para llegar de una punta a la otra.
- ***Maze*** - Este escenario es el más complejo, se puede ver como el mapa de una ciudad y tiene una gran cantidad de obstáculos, bifurcaciones de caminos y caminos sin salida.

Los tres escenarios más básicos fueron utilizados para realizar pruebas de concepto e iterar en el desarrollo de las distintas componentes y algoritmos. Las pruebas finales

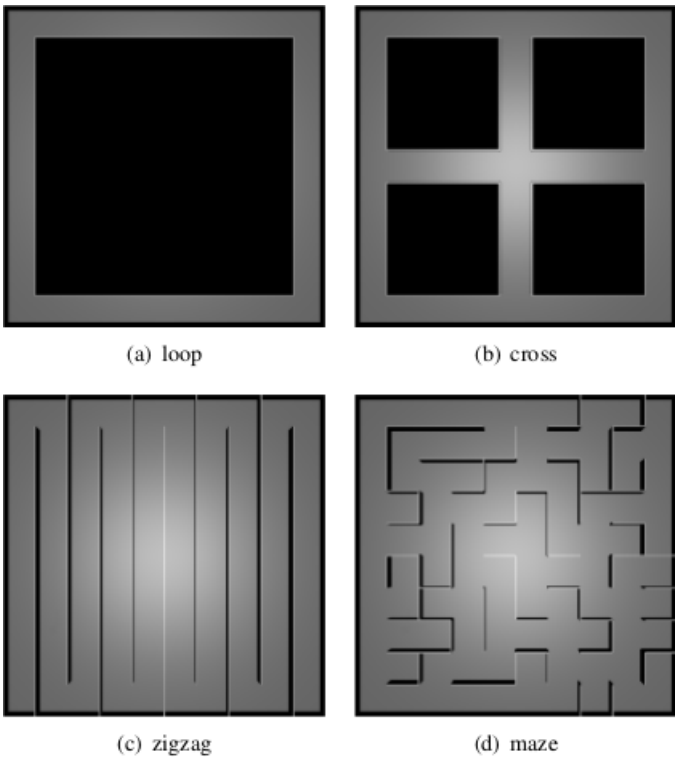


FIGURA 4.1: Cuatro escenarios utilizados para la simulación tomados de [34]

utilizadas para evaluar el desempeño de exploración de distintas configuraciones de robot se realizaron todas en el escenario *Maze*.

4.1.1. Algoritmos de Clusterización

En base a la investigación previa realizada, se optó por dos algoritmos de clusterización para analizar y comparar; estos fueron K-Means [16] (donde, para encontrar el K adecuado se ejecuta el algoritmo sucesivas veces, evaluando distintos valores de K hasta encontrar el indicado) y Affinity Propagation (AP) [17] descritos previamente en la sección 2.3. Para realizar las comparaciones se ejecutaron los algoritmos con distintos escenarios, los cuales tenían distintas cantidades de puntos a agrupar y distribuciones de los mismos. Los escenarios se pueden observar en la figura 4.2. Para realizar las comparaciones entre los algoritmos y tomar una decisión de cuál usar para la solución del proyecto se tomaron en cuenta dos factores: el tiempo de ejecución (cuadro 4.1) y la calidad de los puntos de salida (cuadro 4.2 y figura 4.3), basándose en la cantidad de puntos y en cómo estaban distribuidos respecto a los puntos de frontera usados para la entrada del algoritmo.

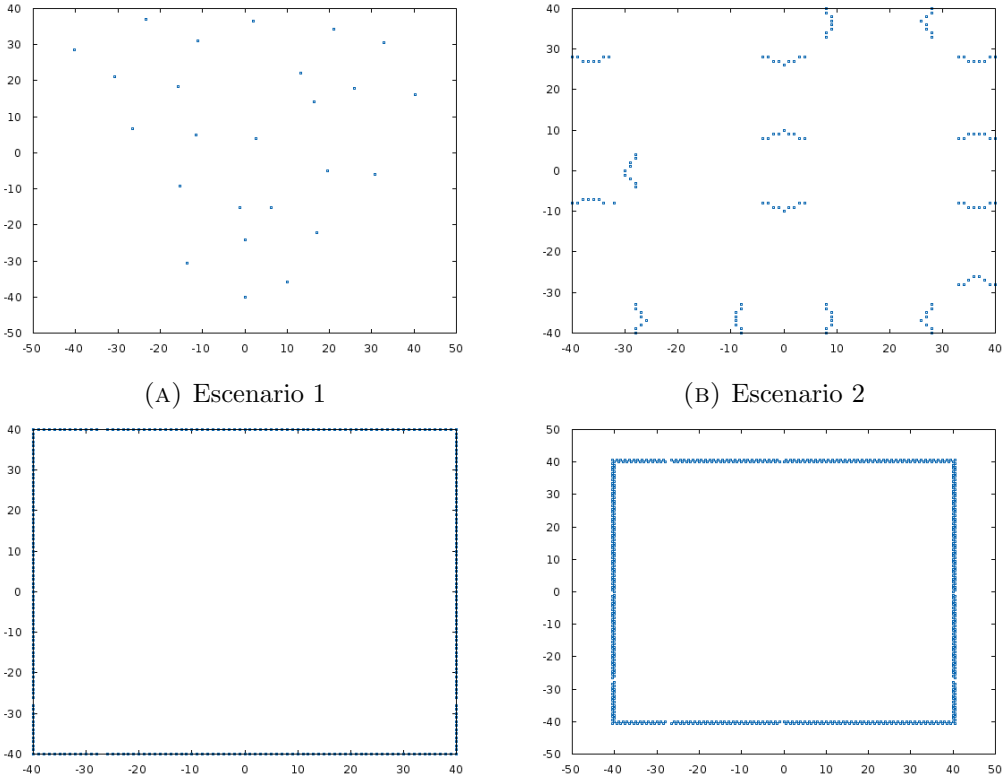
Al analizar los tiempos de clusterización en el cuadro 4.1 se puede observar que en los escenarios del 1 al 3, los tiempos son bastante similares, aunque los de K-Means siempre están por debajo de los de AP, siendo K-Means una mejor solución en cuanto a los tiempos de ejecución. En cambio, cuando crece el número de puntos a clusterizar

		Tiempo (segundos)	
Escenario	Puntos	K-Means	AP
1	24	0.002132	0.004419
2	131	0.010919	0.218573
3	321	0.756446	1.094193
4	643	18.997733	7.771627

CUADRO 4.1: Comparación de tiempos de ejecución entre Affinity Propagation y K-Means con lógica para probar valores de K. Pruebas realizadas para cinco escenarios creados para evaluar situaciones realistas dentro de un ciclo de exploración, así como también casos borde.

como en el escenario 4, se puede observar que no solo K-Means presenta peores tiempos que AP, sino que esta diferencia de tiempos es bastante grande. En base a las pruebas realizadas se puede decir que K-Means, en escenarios con pocos puntos a clusterizar tiene un mejor desempeño que AP pero, cuando el conjunto de puntos crece, el deterioro del desempeño es visible, creciendo rápidamente los tiempos de ejecución de K-Means (en forma aparentemente exponencial). Por otro lado, inicialmente AP no tiene un tan buen desempeño, pero la tasa de deterioro del mismo (aparentemente lineal) es bastante inferior a la de K-Means; lo que hace que una vez que AP iguala a K-Means en tiempos, si se continúan agregando puntos, K-Means rápidamente empeora su desempeño con respecto a AP. Cabe mencionar también, que en los escenarios donde AP presenta peores tiempos que K-Means, ambos algoritmos completan su ejecución en tiempos por debajo del segundo (a veces ambos muy por debajo, en el orden de centésimas de segundo). Mientras que en los escenarios donde se mostró mejor eficiencia por parte de AP, el mismo presentó tiempos siempre por debajo de los diez segundos, mientras que K-Means rondaba los dieciocho a veinte segundos.

Por otro lado, se analizó la calidad de los puntos de salida de los algoritmos donde se compara según la cantidad de los puntos de salida en el cuadro 4.2 y la distribución en el espacio respecto a los puntos de entrada en la figura 4.3. En cuanto a la cantidad de puntos de salida de los algoritmos, la diferencia mayor se da para conjuntos de puntos de entrada grandes como en el escenario 4 donde se puede claramente observar como AP logra mantener un número pequeño de centroides, mientras que K-Means devuelve más del doble de puntos que AP. Para poder analizar estos datos de manera correcta, es muy importante tener en cuenta cómo están distribuidos los puntos de entrada en el espacio. En el escenario 4 que se muestra en la figura 4.3b, donde todos los puntos de entrada son todos contiguos se observa que los puntos retornados por K-Means no se encuentran distribuidos de manera uniforme en el espacio, teniendo zonas con grandes aglomeraciones de puntos de centroides, mientras que, AP no solo logra mantener un número bajo de centroides, sino que se puede observar que están todos distribuidos de manera casi equidistante. Por otro lado, observando el escenario 2 en el cuadro 4.2 se puede observar que la diferencia entre la cantidad de puntos de salida de cada algoritmo no es muy grande, siendo incluso K-Means menor. Pero al observar la distribución de

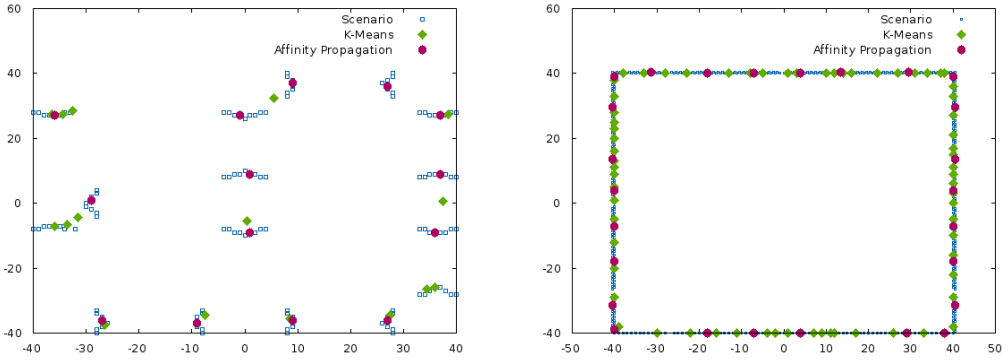


(A) Escenario 1 (B) Escenario 2

(C) Escenario 3 (D) Escenario 4

FIGURA 4.2: Escenarios utilizados para ejecutar las pruebas sobre los algoritmos K-Means y Affinity Propagation.

los puntos en la figura 4.3a, se puede ver como para K-Means hay conjuntos de fronteras para los que no devuelve ningún centroide y otras para los que devuelve hasta tres. Mientras que, por otro lado, AP logra agrupar cada una de las fronteras en forma exacta, y encontrar un centroide para cada una de ellas que es prácticamente el ideal.



(A) Escenario 2 (B) Escenario 4

FIGURA 4.3: Comparación de algoritmos de clusterización Affinity Propagation y K-Means en termino solución generada.

A la hora de decidir con cual algoritmo trabajar, la calidad de los puntos de salida

		Centroides		Centroides P/Frontera		Fronteras S/Centroides	
Escenario	Puntos	K-Means	AP	K-Means	AP	K-Means	AP
1(*)	24	14	4	0.583	0.167	10	20
2	131	14	16	0.875	1.000	4	0
3	321	26	20	1.625	1.250	0	0
4	643	70	28	4.375	1.750	0	0

CUADRO 4.2: Comparación de las soluciones obtenidas comparando Affinity Propagation y K-Means con lógica para probar valores de K.
(*) Escenario poco frecuente. Solo se observa próximo al final de la exploración, si se pretende lograr un 100 % de cubrimiento, y se dejaron puntos solitarios y alejados en rincones o esquinas del mapa.

del algoritmo es muy importante porque, por un lado, cuantos más centroides se tengan, más procesamiento va a requerir el algoritmo de planificación. Por el otro, si los centroides están mal distribuidos y se encuentran muy cercanos unos de otros, o incluso excluyen fronteras válidas, esto podría provocar que dos robots vayan al mismo lugar, comprometiendo el desempeño global de la exploración. En base a esto y a los tiempos de ejecución se optó por AP como solución al problema de clusterización, ya que cuando los conjuntos de fronteras eran pequeños los tiempos de AP eran aceptables a pesar de ser mayores que los de K-Means, pero cuando el número de fronteras crecía, AP era notoriamente más eficiente.

Ya usando el algoritmo AP en la solución final del proyecto, se evaluó el comportamiento y resultados del mismo mediante simulaciones de exploración ejecutadas con configuraciones de uno a nueve robots. Como se observa en la figura 4.4, a lo largo de exploraciones del escenario Maze, el número de celdas de frontera a procesar varió entre cero y poco más de quinientos (aunque en casos excepcionales), y la cantidad de zonas de fronteras reconocidas por AP nunca supero las 30 para ninguna de las configuraciones. Se observa también que, para mismas cantidades de celdas de frontera en distintas ejecuciones, AP reconoció zonas de frontera con variaciones de hasta 10 zonas. Esto es esperable, y hasta deseable, ya que la cantidad de celdas de fronteras no guarda relación con las zonas de fronteras que representan, y demuestra que el algoritmo está teniendo en cuenta otras condiciones en forma apropiada al agrupar celdas en sus respectivas zonas de frontera, como ser la relación que guardan basados en su proximidad.

Luego, en la figura 4.5, se puede observar el tiempo que llevó la ejecución de AP para las distintas configuraciones de robots y celdas de frontera que tuvo que evaluar. Se puede observar que hay cierta relación entre la cantidad de puntos a agrupar y el tiempo de ejecución del algoritmo, pero también se observa una mayor varianza en el mismo a medida que la cantidad de puntos crece. Se podría decir que el algoritmo presenta un comportamiento entre cuasi-lineal y cuadrático en base a su varianza, siendo aproximable por la ecuación $7,52E^{-2} - 1,60E^{-3}x + 1,76E^{-5}x^2$.

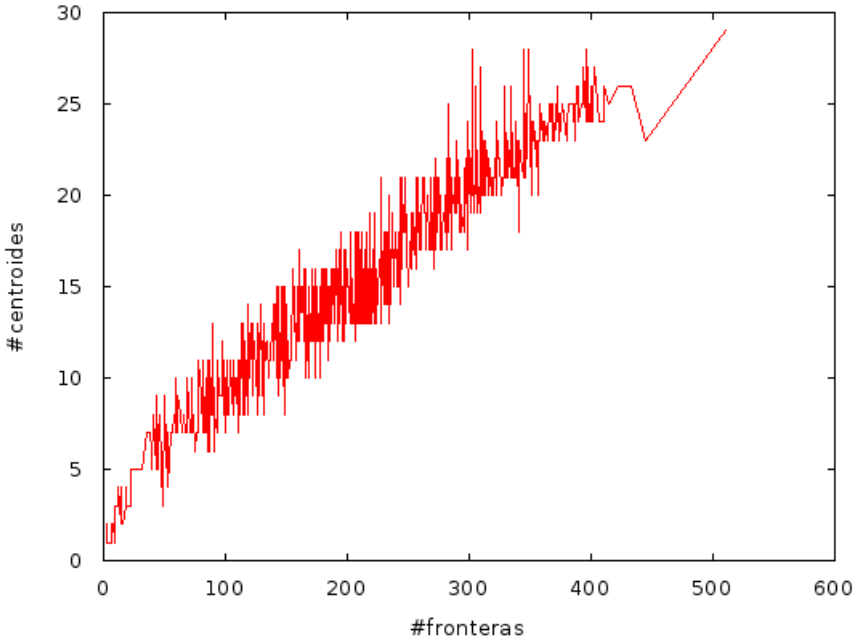


FIGURA 4.4: Gráfica de la cantidad de centroides generados en función de la cantidad inicial de celdas de frontera a agrupar. Datos recolectados experimentalmente a partir de 90 simulaciones variando entre 1 y 9 robots.

4.1.2. Algoritmos de Planificación

Como se mencionó al describir la planificación en el capítulo anterior, se probaron distintos algoritmos e implementaciones de los mismos hasta llegar a la versión final utilizada. La primera implementación que se realizó fue del algoritmo MinPos en su versión más pura siguiendo la descripción teórica del mismo, y haciendo uso del algoritmo A* para el cálculo de camino entre dos puntos. Al correr simulaciones de exploración haciendo uso del mismo, se podía observar que, en los puntos medios de la exploración, los tiempos en que el robot se detenía a planificar eran notorios, y la situación era aún más evidente cuando la configuración de la exploración incluía varios robots en lugar de uno solo.

Dada la naturaleza serial del algoritmo, este comportamiento era esperable, por lo que se decidió implementar la variante de MinPos descrita por Bautin [23] en la que se utiliza propagación de olas como alternativa para el cálculo de camino optimo entre dos puntos y como herramienta para facilitar el cómputo en paralelo de las asignaciones. Cabe notar que esta implementación, a pesar de computar en paralelo distintos caminos y rangos de robots, sigue siendo un algoritmo que ejecuta en forma serial.

Una mejora a futuro para el algoritmo de A* podría ser buscar alternativas para lograr hacer el computo de los caminos en paralelo. Para poder realizarlo, primeramente hay que ver si es posible o si se requiriere incurrir en el uso de técnicas de computación paralela para la optimización, lo cual rompe con la naturaleza serial del algoritmo y hace que los resultados del mismo no sean comparables con la implementación con propagación

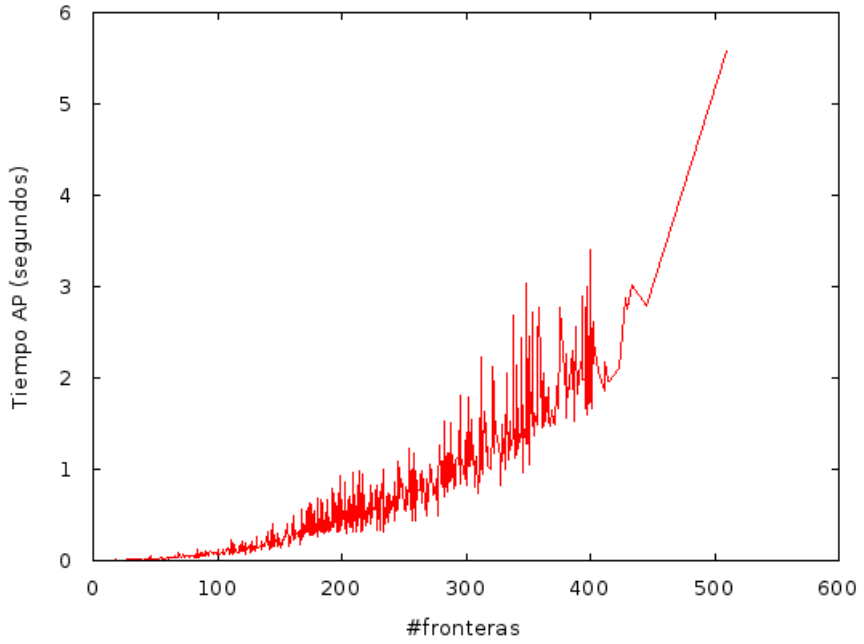


FIGURA 4.5: Tiempos de ejecución del algoritmo de clusterización Affinity Propagation en función de la cantidad de celdas de frontera a agrupar. Datos recolectados experimentalmente a partir de 90 simulaciones variando entre 1 y 9 robots.

de olas. Es por esto, que para el contexto del presente trabajo se optó por el uso de algoritmos ya definidos y la comparación de los mismos para poder decidir cuál utilizar.

En la gráfica que se muestra en la figura 4.6, se puede observar la notoria diferencia entre ambas implementaciones del algoritmo MinPos a lo largo de una exploración simulada. La primera utilizando A*, claramente dependiente del número de tareas a planificar (número que encuentra su máximo próximo al tiempo medio de la exploración) y alcanzando tiempos en el orden de los segundos. Mientras que la segunda implementación con propagación de olas presenta un comportamiento más errático, presentando principalmente sus máximos hacia el final de la exploración, momento que se asocia con planificación de tareas más lejanas a los recursos. De todas formas, siempre se mantiene en el orden de las décimas de segundos en los peores casos, y en las centésimas o incluso milésimas para la mayoría de las ejecuciones. Esta mejora de los tiempos de ejecución se debe principalmente a la resolución en simultaneo de los distintos cálculos que se requieren para calcular la asignación final, pero también a la posibilidad que brinda de terminar su ejecución en forma temprana una vez que se descubrió la asignación del recurso deseado, independientemente de haber o no acabado de realizar el cálculo para el resto de los recursos.

Cuando se habla de planificación nos centramos en el algoritmo de asignación de tareas a recursos, pero no hay que olvidarse de las dos etapas de preprocesamiento de los datos, que también forman parte de la planificación. La primera es la definición de

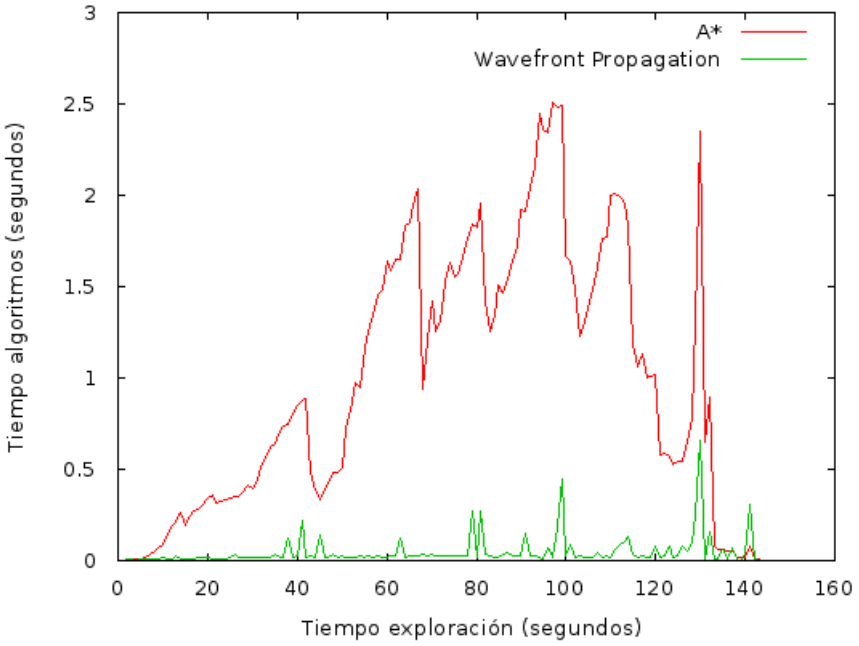


FIGURA 4.6: Comparación de los tiempos de ejecución de las variantes del algoritmo de planificación distribuido MinPos en sus variantes: tradicional utilizando A* para calcular el camino entre dos puntos; y la versión implementada mediante propagación de olas en paralelo. Los datos fueron recolectados a lo largo de una simulación con un robot.

las celdas de frontera, y la segunda es la agrupación de las mismas en zonas de fronteras, que permite simplificar la posterior asignación de tareas, así como asegurarse de evitar errores que lleven a una deficiente distribución de los agentes en el espacio, no por culpa del algoritmo de planificación sino por información de entrada de baja calidad. En las gráficas de las figuras 4.7 y 4.8 se puede observar, para las ejecuciones del ciclo de planificación realizadas durante una misma exploración, cómo los distintos algoritmos intervienen en el tiempo final dedicado a planificar. Se observa que la tarea de definición de celdas de frontera crece en forma continua a lo largo de la exploración, lo que es coherente con el hecho de que las celdas candidatas a ser celdas de frontera y que deben ser evaluadas aumenta con el tiempo a medida que el robot aumenta su conocimiento del ambiente. A pesar de esto, el tiempo dedicado a esa tarea se mantiene siempre bajo. Cuando se observan los algoritmos de asignación de tarea MinPos, se puede observar que en su versión implementada con A* representa aproximadamente la mitad del tiempo del ciclo de planificación, mientras que en su versión implementada con propagación de olas el tiempo que requiere es casi marginal.

Algo que llama la atención al observar estas gráficas, es el porcentaje del tiempo total de planificación que se ocupa en la tarea de agrupación de celdas de fronteras en zonas de fronteras, dónde en uno de los casos está cercano al 50 %, mientras que en el otro es casi la totalidad del tiempo. Al ver esto, se podría plantear si el agrupar las celdas de frontera es realmente beneficioso para la exploración. Como trabajo a futuro se puede realizar el

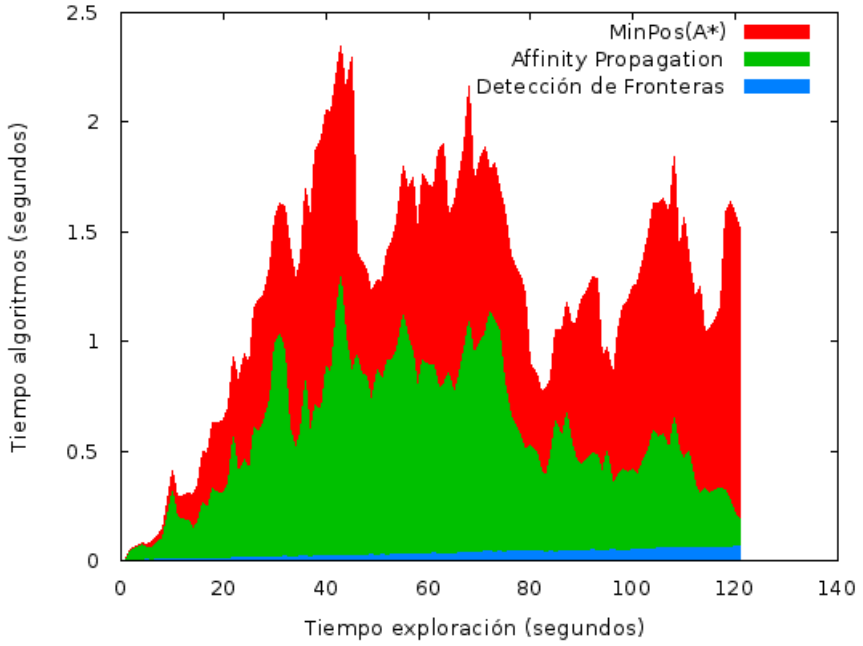


FIGURA 4.7: Distribución de tiempos del planificador en sus tareas de definición de celdas de frontera, clusterización mediante el algoritmo Affinity Propagation, y planificación mediante el algoritmo MinPos en su implementación tradicional utilizando A* para calcular el camino entre dos puntos.

análisis de cómo se comportan los algoritmos de asignación de tareas si no se le realiza un pre procesamiento a los puntos de frontera. Igualmente, observando los tiempos de ejecución de K-Means y Affinity Propagation sobre cientos de puntos (recordar el cuadro 4.1), se puede asegurar que el esfuerzo invertido en la tarea de pre procesamiento no es en vano. Por otro lado, es necesario analizar e investigar los algoritmos para tratar de optimizarlos y así mejorar los tiempos invertidos en clusterizar los puntos de frontera.

Una vez que el desempeño en cuanto a tiempos de ejecución del algoritmo de asignación de tareas ya no era un problema, se pudo observar que para ciertos casos, las asignaciones de tarea realizadas eran subóptimas, asignando en ocasiones múltiples recursos a la misma tarea. Fue entonces cuando se tomó la decisión de implementar el algoritmo MinPos Glouton (MPG) [21], el cual agrega una condición *Greedy* basada en las asignaciones realizadas a otros robots y conserva todas las características deseables de MinPos referentes a la distribución espacial de los recursos. MPG presenta una implementación análoga a la de MinPos mediante el uso de propagación de olas, introduciendo el concepto de pausado de las mismas para permitir la evaluación en forma eficiente de la nueva condición de asignación de los recursos, lo que hizo muy simple el pasaje del uso de un algoritmo al otro.

En la figura 4.9 se puede observar los resultados de la ejecución del algoritmo MPG a lo largo de distintas simulaciones, con distintas configuraciones de cantidad de robots. Se puede observar la varianza e independencia del tiempo de ejecución de la cantidad

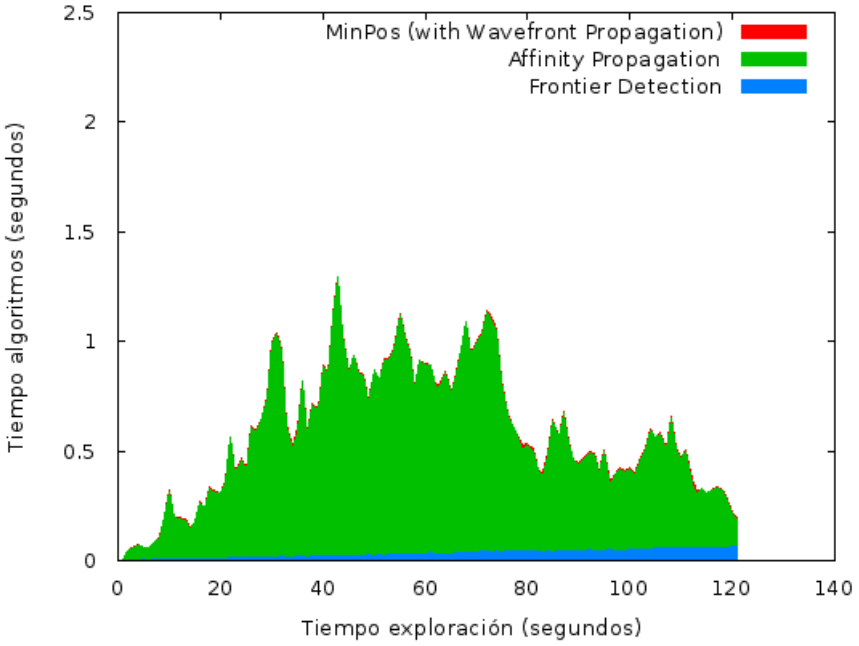


FIGURA 4.8: Distribución de tiempos del planificador en sus tareas de definición de celdas de frontera, clusterización mediante el algoritmo Affinity Propagation, y planificación mediante el algoritmo MinPos en su implementación mediante propagación de olas en paralelo.

de tareas siendo planificadas. Esto se puede relacionar tanto a la independencia de este valor de la distancia a los recursos, como con el hecho de que los datos mostrados no tienen en cuenta las cantidades de recursos evaluados en cada ejecución. Lo que se puede observar, es que para la gran mayoría de las ejecuciones del algoritmo, el tiempo requerido para la misma se mantiene por debajo de la décima de segundo -983 datos se encuentran graficados en la figura 4.9, 867 de ellos no superan la décima de segundo, y 920 no superan las dos décimas-.

En las pruebas realizadas, ya planificando con la versión final del algoritmo, se pudieron verificar los comportamientos esperados, así como observar variaciones del mismo. En la figura 4.10 se muestra la cantidad de zonas de fronteras que se observan a lo largo de un ciclo de exploración. El comportamiento para pocos robots se alinea con lo que se asume de antemano para un escenario estilo laberinto (o planta de un edificio con habitaciones) con numerosas bifurcaciones. Al principio de la exploración se van descubriendo zonas de frontera, más de las que se pueden asignar a los recursos, hasta que se llega a un punto donde ya no se descubren nuevas fronteras y simplemente se barre las zonas desconocidas hasta completar la exploración. Ese comportamiento se aproxima con una parábola con concavidad negativa, con su cima alcanzada aproximadamente a la mitad de la exploración. El comportamiento inicialmente no previsto, se relaciona a las configuraciones de exploración con mayor número de robots. En estos casos, dependiendo si las posiciones iniciales de los mismos se encuentran dentro del alcance de los sensores o no (dependiendo de si los robots se “ven” unos a otros), la cantidad inicial de

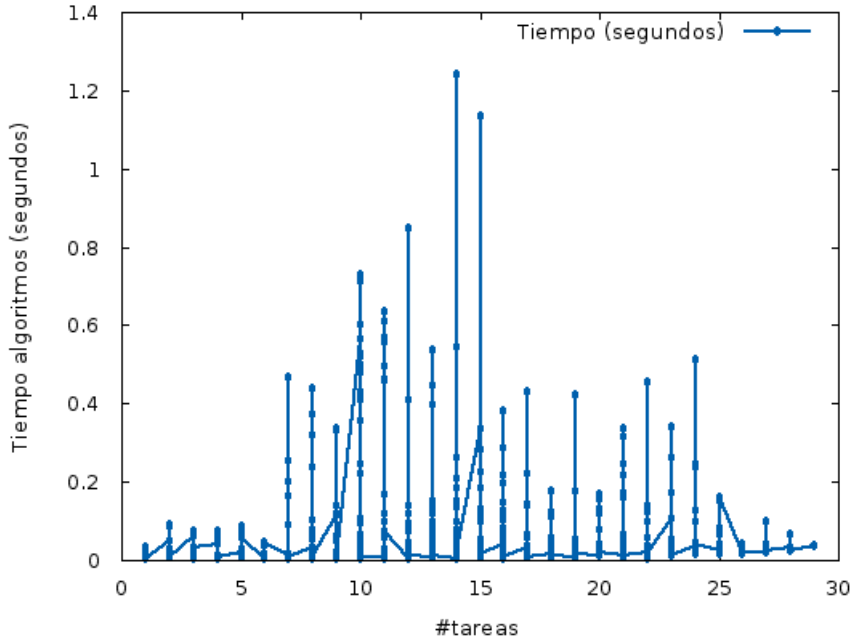


FIGURA 4.9: Tiempos de ejecución del algoritmo de planificación MinPos Glouton en función de la cantidad de tareas. Datos recolectados experimentalmente a partir de 90 simulaciones variando entre 1 y 9 robots.

fronteras observadas varia ampliamente. En los casos donde los robots se ven, el comportamiento es igual al observado con pocos agentes. Pero en los casos donde no es así, la cantidad inicial de fronteras es alta. Debido a esto, y a que los agentes comienzan la exploración en posiciones próximas, en lugar de observase la parábola esperada desde el primer momento, primero se observa un descenso en la cantidad de fronteras, a medida que los robots se encuentran entre ellos y luego sí se observa el comportamiento que se había previsto.

Si observamos la figura 4.11 que muestra los tiempos invertidos a lo largo de la exploración en las principales tareas de planificación (determinación de zonas de frontera y planificación de asignaciones), se puede notar que los mismos, en cierta medida también acompañan este comportamiento de parábola descrito para la cantidad de fronteras descubiertas. En los hechos, el que acompaña principalmente el comportamiento parabólico es el algoritmo de clusterización, que depende en forma más directa del número de fronteras, y al representar la mayor parte del tiempo de planificación genera este efecto. Si nos enfocamos en MPG podemos observar que mantiene el comportamiento errático descrito previamente.
Mirando los tiempos de Affinity Propagation, se puede observar para las configuraciones con mayor número de robots, que al principio de la exploración los tiempos de clusterización son elevados; esto se debe a la distribución inicial de los robots que no se "ven", y como se mencionó previamente, esto genera un gran número de puntos de fronteras al comienzo de la exploración y por consecuencia, un número elevado de zonas de fronteras. Cuando nuevamente nos enfocamos en MPG e intentamos darle sentido a algunas de

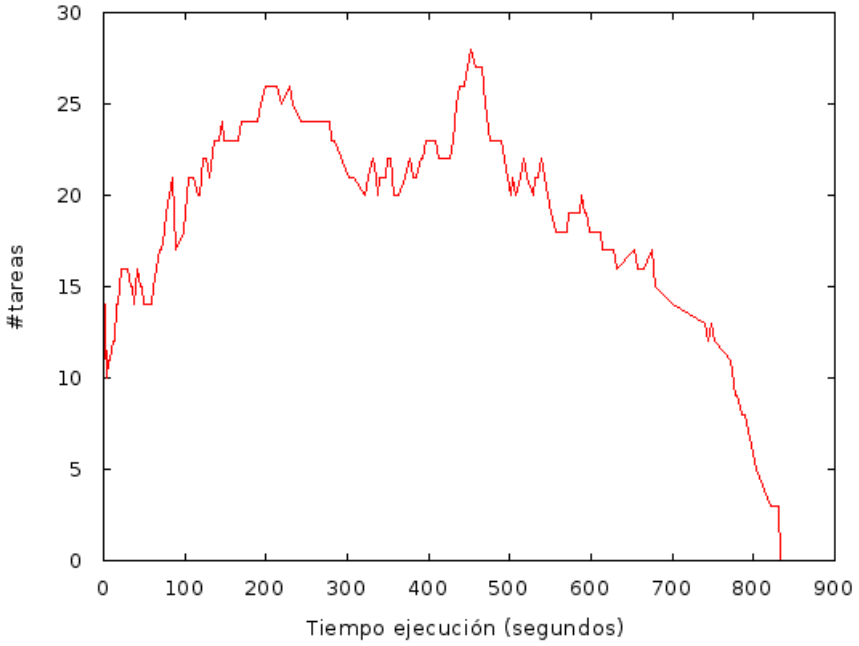


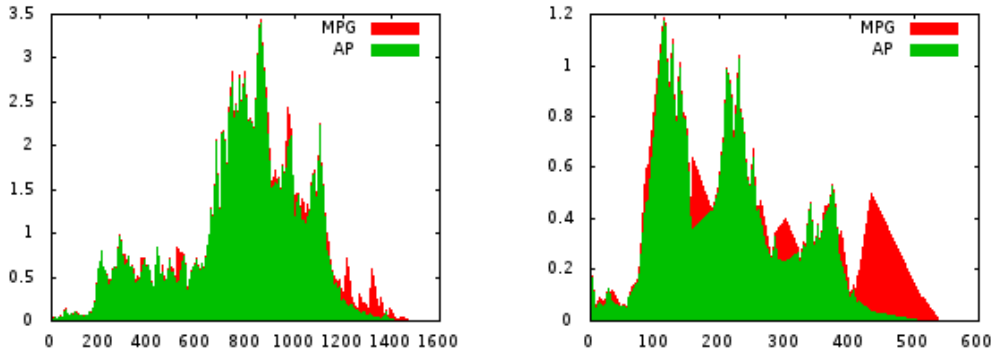
FIGURA 4.10: Cantidad de fronteras a procesar por el planificador a lo largo de la exploración. Datos obtenidos a partir de una simulación realizada con una configuración de dos robots.

sus oscilaciones, resulta de interés notar que para todas las configuraciones se observan tiempos elevados hacia el final de la exploración. Estos tiempos elevados se deben a que en la última etapa de la exploración usualmente quedaron atrás pequeñas regiones o rincones inexplorados, usualmente alejados, y esto hace que se deban planificar caminos más largos, propagando olas de mayor tamaño, que a cada paso generan potencialmente un mayor número de celdas a procesar (cada celda se propaga potencialmente en 8 direcciones) con la respectiva carga computacional que ello implica, e impidiendo en algunas ocasiones aplicar las condiciones de poda en forma temprana que permitían al algoritmo cierto grado de eficiencia adicional.

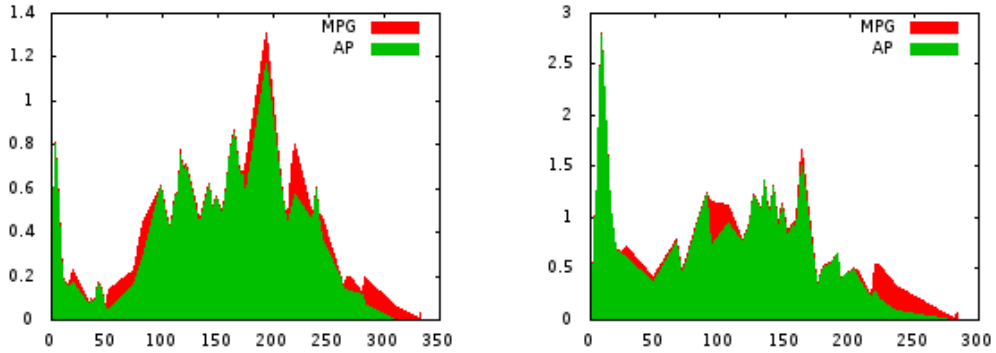
4.1.3. Resultados de Exploración

En la presente sección se introducen los resultados de la simulación a medida que se incorporan agentes a la solución. En estos resultados se logra ver efectivamente cómo la colaboración entre los robots ayuda a reducir el tiempo de exploración de un mismo ambiente en forma sustancial. En la gráfica de la figura 4.12 se puede ver cómo el tiempo del escenario Maze es mejorado desde más de veinticinco minutos al ser explorado por un solo robot, a aproximadamente 5 minutos al ser explorado por cinco o más robots.

Cabe notar que, dadas las características de la simulación, los resultados podrían no ser totalmente fieles. A lo que se hace referencia con esto es, que las simulaciones fueron realizadas corriendo desde una única computadora con recursos compartidos entre



(A) Configuración 1 solo robot. (B) Configuración 3 robots.



(C) Configuración 5 robots. (D) Configuración 8 robots.

FIGURA 4.11: Distribución del tiempo de planificación entre clusterización de celdas de frontera y planificación utilizando algoritmo MinPos Glouton.

todos los robots simulados; por lo que a medida que la cantidad de instancias de robot aumentó, los recursos de los que dispuso cada una de ellas fueron más limitados. De todas formas, esto no se puede utilizar para llegar a una conclusión que asegure que de haber contado con recursos independientes los resultados para simulaciones con mayor número de robots hubiese sido mejores (aunque sí es posible), ya que, aunque recursos como ser los procesadores pasaron a ser compartidos, recursos como ser la red de datos fueron locales a todos los robots y por ende de mayor velocidad y calidad que una simulación real que hiciese uso de una red inalámbrica. Al no contar con mediciones apropiadas para poder cuantificar el efecto de cada uno de estos factores en forma independiente, no es posible saber si los beneficios o perjuicios del ambiente de simulación tienen mayor peso en los resultados. Teniendo en cuenta lo mencionado, y la relatividad de los resultados, se puede de todas formas comentar los mismos sabiendo que son ciertos y aplican para esa realidad.

Observando los resultados graficados, se puede confirmar que efectivamente la colaboración entre los agentes robóticos produjo una mejora sustancial del tiempo total de exploración de un ambiente. Con tiempos iniciales para una configuración de un robot que promediaban los 26,4 minutos, se logró observar mejoras por encima de 45 % al agregar un segundo robot, logrando un tiempo promedio de 14,3 minutos. El agregado

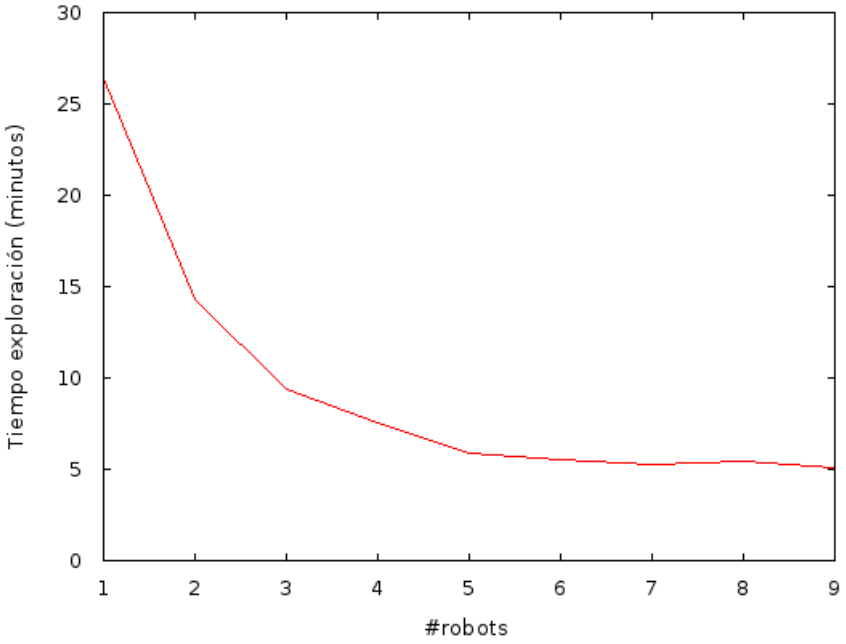


FIGURA 4.12: Evolución del tiempo total de exploración de la solución a medida que se agregan agentes robóticos al sistema.

sucesivo de robots a la exploración hasta llegar a nueve de ellos continuó produciendo mejoras, aunque cada vez menores. Pero teniendo en cuenta los resultados globales, se puede observar que al reducir a 5,1 minutos promedio el tiempo al explorar con nueve robots, se obtuvo una mejora de 80 %.

Algo relevante a mencionar es que, a partir del quinto robot agregado, las mejoras obtenidas fueron cada vez más marginales. Con la configuración de cinco robots los tiempos ya se habían mejorado en un 77,5 % lo que significa que el agregado de los últimos cuatro robots produjo tan solo una mejora con respecto a esta configuración de 47 segundos. Incluso se observa que la configuración de ocho robots fue levemente peor en términos de tiempo que la de siete (7 segundos más lenta), mientras que la configuración de nueve robots fue 10 segundos más rápida que la de siete robots. Esto nos hace pensar que, a partir de cierto punto se ha alcanzado un comportamiento asintótico donde, el agregado de más agentes a la exploración no va a traducirse necesariamente en una mejora del tiempo final de la exploración. Vale la pena notar que, para las simulaciones y el escenario donde fueron corridas, este número a partir del cual se observa el comportamiento asintótico es de cinco o seis robots, pero este número va a variar dependiendo de las dimensiones y características de cada escenario. Lo que es interesante concluir sobre esto, es que este comportamiento existe, y que el agregado de robots a la exploración no va a mejorar el tiempo de exploración infinitamente, sino que la mejora que van a aportar los nuevos robots va a estar acotada. Eventualmente, el agregado de nuevos agentes robóticos podría terminar perjudicando al objetivo de optimización de tiempo; los resultados

no lo llegan a mostrar, pero es posible que, si se siguiesen agregando robots, el tiempo en lugar de mostrar un comportamiento asintótico demostrase un comportamiento parabólico. Esto se podría justificar ya sea porque los robots se estorben unos a otros durante la exploración, como por el efecto que el agregado de nuevos robots tiene en los algoritmos de planificación e integración y comunicación de datos de los mapas y posiciones.

Observando ahora las figuras 4.13 y 4.14 se analiza la evolución del tiempo de exploración en función de su descomposición entre tiempo invertido en planificación, y tiempo invertido en navegación. En la figura 4.13 se observa como los tiempos tanto de planificación, como de navegación presentan un comportamiento asintótico a medida que se agregan más robots a la exploración. La planificación se aproxima al valor asintótico en forma más temprana que la navegación. Esto se podría explicar contemplando que, a mayor número de robots, se requieren menos ciclos de planificación, pero cada uno de ellos es más complejo. Por otro lado, el tiempo de navegación presenta poca mejora a partir del cuarto robot agregado al equipo; este hecho es contrainutivo, ya que cada robot debería estar cubriendo menor área de exploración y desplazándose menor distancia en total; lo que da lugar a pensar que esta información muestra potencial de mejora en esta área. Al analizar la relación entre estos dos tiempos, al variar la configuración

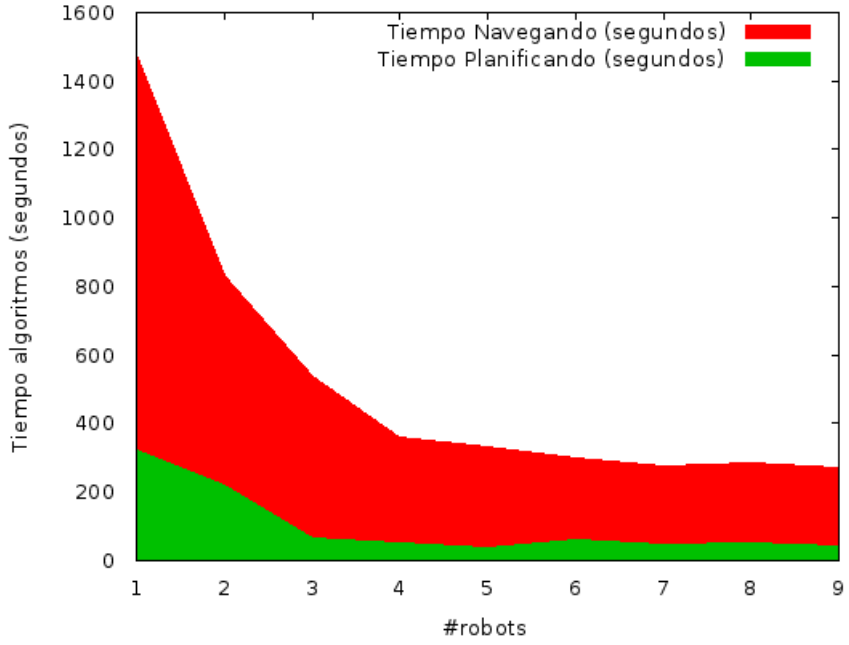


FIGURA 4.13: Distribución de tiempos del componente de navegación entre sus tiempos de espera por un objetivo, y sus tiempos de navegación hacia el objetivo recibido. Datos graficados en función de la cantidad de robots para 90 simulaciones realizadas (10 de ellas para cada número de robots).

de número de robots, se ve que la misma oscila alcanzando su mínimo de 13 % en la configuración de cinco robots, y el máximo de 35 % con dos robots. Esto no permite generar conclusiones en base a esta relación. Si se analiza la figura 4.14 que muestra la

distribución entre tiempo de planificación y navegación para una simulación de exploración, nuevamente se puede ver cómo los mayores tiempos de planificación se dan hacia la mitad de la exploración cuando hay un mayor número de fronteras. Desde el punto de vista de la navegación se observan dos aspectos de interés: el primero es el mismo ya mencionado dónde la mayoría del tiempo de exploración es invertido en navegación. Esto desde el punto de vista del proyecto es deseable, ya que se buscó minimizar el tiempo de exploración en base a los algoritmos de planificación utilizados dejando de lado en parte los aspectos relacionados a la navegación. El segundo aspecto a observar es que hacia el final de la exploración, se ve un incremento sustancial en la relación entre tiempo de navegación y tiempo de planificación, en favor del tiempo de navegación. Este aspecto se condice con el hecho de buscar objetivos más distantes, que se da al final de los ciclos de exploración debido a que en el camino se dejaron regiones inexploradas. Pensando

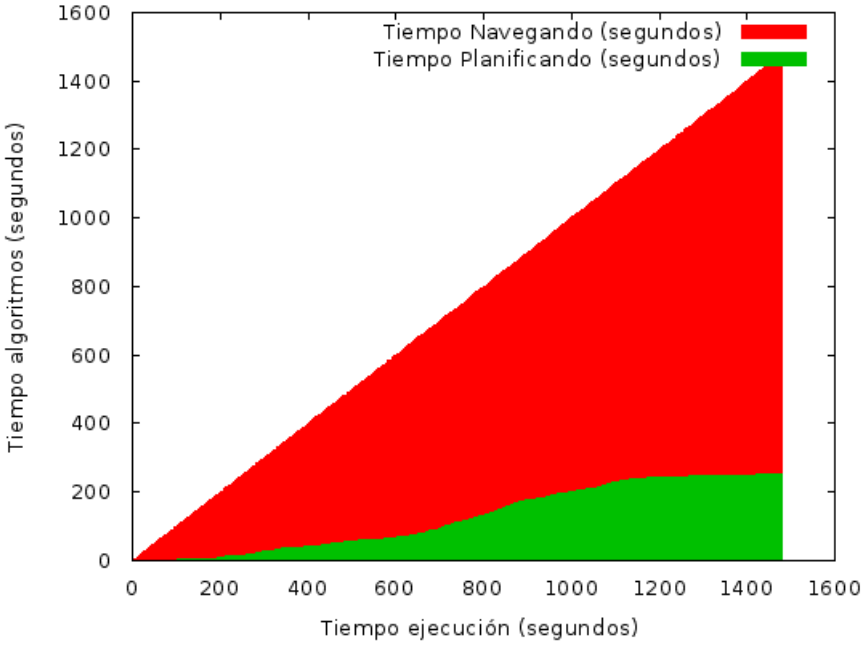


FIGURA 4.14: Distribución de tiempos del componente de navegación entre sus tiempos de espera por un objetivo, y sus tiempos de navegación hacia el objetivo recibido. Datos recolectados para la exploración de 1 robot.

en posibles mejoras a realizar para evitar estas situaciones, hay que distinguir entre dos posibles escenarios. Por un lado, estas regiones inexploradas dejadas atrás podrían corresponderse con grandes zonas inexploradas, en cuyo caso, quizá mejores decisiones al planificar objetivos podrían haber prevenido la situación. Por otro lado, estas regiones podrían corresponderse con una situación que se observa frecuentemente: pequeñas regiones asociadas a esquinas o rincones que no fueron alcanzados por los sensores al explorar una región. En este segundo caso, explorar estas regiones podría no aportar valor desde el punto de la información que agregarán a la solución final. Distintas estrategias podrían ser aplicadas para optimizar este aspecto. En caso de conocer el área final de exploración del ambiente, se podría optar por una estrategia de exploración que

apunte a completar un porcentaje del área en lugar de su totalidad (por ejemplo 98 %), de esta forma en caso de haber dejado atrás pequeñas regiones al explorar el mapa, estas deberían ser ignoradas y no tenidas en cuenta descartando los últimos ciclos de planificación y navegación. En caso de no conocer el área total a ser explorada, se podría utilizar una estrategia que haga pre-procesamiento del área detrás de las fronteras, calculando su potencial de información a incorporar; de esta forma cuando una frontera es planificada y la misma es lejana, se realiza el cálculo de su potencial y en caso que, la información que pueda llegar a aportar la frontera no sea significativa, la misma se descarta logrando de esta forma ahorrar el tiempo de navegación hacia la misma.

4.2. Prueba de concepto con Butiá

En la prueba de concepto sobre una plataforma física, se requiere utilizar la plataforma Butiá para robótica educativa [2]. Se hará uso de la versión 3 de la misma, quedando para un futuro las pruebas con más de un robot en un sistema heterogéneo, utilizando también el Butiá 2. La versión 2 de Butiá es rectangular en el plano $< x, y >$, con veinticinco centímetros de lado; mientras que la versión 3 es triangular e inscrita en una circunferencia de diámetro veinticinco centímetros.
Para la plataforma física no solo es necesario saber que estructura se va a utilizar (en nuestro caso el Butiá) sino también definir que hardware/software se va a utilizar para los distintos componentes: placa, sensor de rango, visión y motores.

En el caso de los motores se utilizaron servo motores. Estos se componen de un motor DC que junto con engranajes y un circuito de control le permite controlar la posición del eje en un momento dado y mantenerse fijo en ella. Gracias a esto se puede fácilmente controlar la velocidad a la que gira el motor y dejarlo fijo en caso de que se requiera que el robot quede quieto. Para controlar la velocidad y habilitar o deshabilitar el motor se escribe en ciertos archivos del sistema, específicos para estos motores, por lo que el control sobre los mismos es sencillo y no requiere de ningún tipo de comunicación o paquete para manejarlos.
A diferencia de la simulación (sección 3.3.7) que se le envía el punto destino mediante un puerto a MORSE y este se encarga de trasladar el robot hasta ese punto, en el robot físico lo único que tenemos es control sobre el movimiento de los motores. Para implementar la navegación hacia el punto destino se realizan dos pasos:

- se rota el robot para que quede alineado con el nuevo punto destino, en caso de que sea necesario y
- se avanza en línea recta.

Como se muestra en la figura 4.15, para la rotación primero se calcula el ángulo (*phi*) a rotar, luego en base al mismo se identifica si es necesario rotar hacia la derecha o hacia

la izquierda, dependiendo si el valor es mayor o menor a la rotación actual del robot; y por último se rota con centro el eje de las ruedas controlando con la posición cuando se llegue al ángulo deseado. Por el lugar donde están ubicadas las ruedas, no es posible rotar con centro el centro del robot, lo más cercano que se puede estar, al rotar ambas ruedas (una para delante y la otra para atrás) es rotar según el centro del eje de las ruedas delanteras.

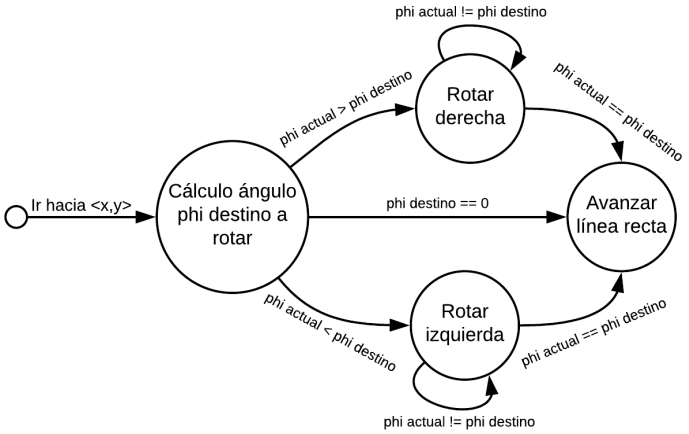


FIGURA 4.15: Máquina de estados que muestra cómo la componente de los motores del robot físico resuelve la navegación hacia un punto destino

Para utilizar como placa computadora se plantearon dos opciones: Raspberry Pi 3 (RPi) [36] o BeagleBone Black (BBB) [37]. Para la implementación de este proyecto en un principio se comenzó utilizando la RPi pero luego de varios inconvenientes con la misma se optó por usar la BBB con una versión de Ubuntu 16.04. Algunas de las ventajas que tiene la BBB sobre la RPi son:

- La BBB tiene un eMMC incorporado, de la cual inicia haciéndola más estable en comparación con la RPi que inicia desde la SD.
- La BBB tiene un botón para prender y otro para reiniciar, lo cual frente a un fallo en la conexión SSH permite hacer un apagado seguro de la placa sin correr riesgo de corromper el sistema como sucede con la RPi.
- La BBB permite conectarla a una PC mediante un USB no solo para darle corriente, sino que al hacer esto se genera una conexión con la placa y se puede fácilmente conectarse por SSH utilizando el puerto 192.168.7.2. En la RPi esto no es posible, lo que hace engorrosa la configuración inicial, ya que es necesario conectarlo a un monitor y utilizar un teclado.

El sensor de rango 360º utilizado es el RPLIDAR A1 [38]. Este es de los sensores más económicos del mercado y con un muy buen rendimiento. Algunas especificaciones del producto son: la distancia máxima de sensado son 6m, se puede variar la frecuencia a la cual gira el sensor y tiene un error de 0,2 cm. Para integrar el sensor a nuestro sistema

se utilizó un SDK en C++ provisto por los desarrolladores del sensor, el cual es sencillo de utilizar y devuelve un buffer con todas las lecturas realizadas por el sensor en una vuelta, una entrada al sistema muy parecida al que se obtiene desde la simulación.

El último componente a definir es el sistema de visión, el cual actúa como GPS del robot, entregando la posición en el plano $< x, y >$ y la rotación phi. Para este proyecto se utilizó un paquete de ROS de visión que implementa el sistema ALVAR [39]. La localización se realiza basado en marcadores, cada uno de ellos es generado por el paquete y tiene un identificador único; en la figura 4.16 se puede observar la salida de 6 marcadores que van del 0 al 5. Estos deben de estar ubicados en lugares completamente visibles y fijos, es decir no pueden rotar (independientemente del robot que los lleva) y nada los puede tapar. En nuestro caso se utilizó un contenedor ROS de ALVAR, permitiendo una fácil utilización e integración.
El sistema de visión corre desde una PC, la cual actúa como servidor y publica, mediante un script de Python que se comunica con ROS, para cada marcador sus posiciones en un puerto distinto que luego son consumidos desde los robots, siempre respetando los identificadores de manera tal de que cada robot sepa que datos son los que corresponden con su localización. Para que la lectura se haga de manera correcta es necesario calibrar la cámara; esto se hace utilizando un paquete que provee ROS. La calibración es sencilla ya que lo único que hay que hacer es imprimir el patrón de la figura 4.17, iniciar el paquete y mover esta imagen en distintas direcciones hasta que se calibre. Estos datos son guardados en un archivo por lo que es necesario hacerlo solo una vez por cada cámara.

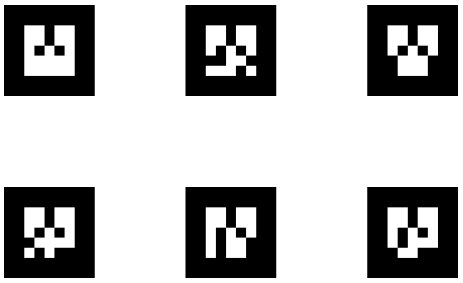


Figura 4.16

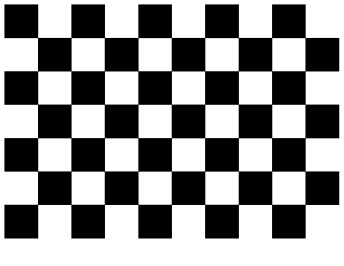


Figura 4.17

4.3. Diseño físico del Butiá

Como se puede observar en las figuras 4.18 y 4.19 si observamos el robot de abajo hacia arriba, el robot cuenta con dos ruedas en la parte delantera del robot con dos servo motores, una a cada lado, que son las que le dan la dirección y hacen avanzar al robot; ademas cuenta con un tercer punto de apoyo en la parte trasera. Luego cuenta con un hub USB, necesario ya que la placa cuenta con un único puerto USB y es necesario conectar un adaptador WiFi y el sensor de rango. Este hub está conectado a la BBB junto con una *shield* creada para conectar los motores a la placa y alimentarlos de manera sencilla e independiente. El sensor de rango se encuentra en la parte superior del robot ya que tiene que tener una visión 360º y está levemente elevado para que el cable del mismo pueda entrar sin inconvenientes. Por último, se realizó un soporte en alambre visible en la figura 4.19 para poder colocar el marcador utilizado para la visión ya que como se dijo anteriormente, este debe de estar sobre una superficie fija y se debe de tener una completa visión sobre él. El soporte de alambre tiene una pequeña incidencia sobre la lectura del sensor, generando puntos ciegos y disminuyendo el rango sensado levemente. Pero esto no influye en la ejecución del programa ya que el sensor tiene una distancia mínima de sensado de 15cm y todo lo que esté más cerca es considerado datos inválidos devolviendo 0 en los ángulos de incidencia de los alambres, y descartándolo automáticamente sin necesidad de programar código ad-hoc para la tarea.

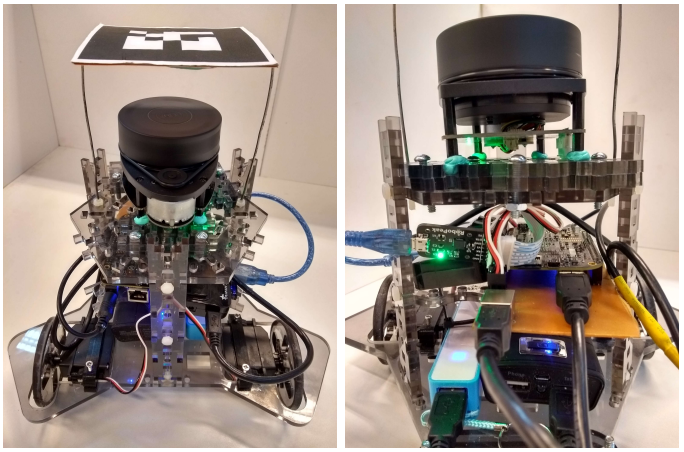


FIGURA 4.18: Imágenes de la estructura utilizada para implementar la plataforma física Butiá

4.4. Escenario

Dadas las limitaciones ofrecidas por el ambiente donde se instaló el escenario y los requerimientos necesarios para que el sistema de visión funcionase en forma apropiada, el tamaño del escenario utilizado para la realización de las pruebas no fue de las dimensiones

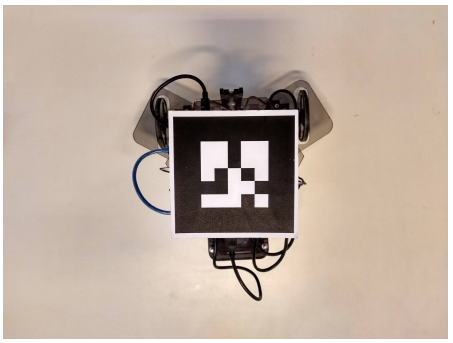


FIGURA 4.19: Marcador utilizado para la visión con el sistema ALVAR

inicialmente planificadas. En la imagen 4.20 se puede observar el escenario, el cual cuenta con una pared interna como obstáculo.

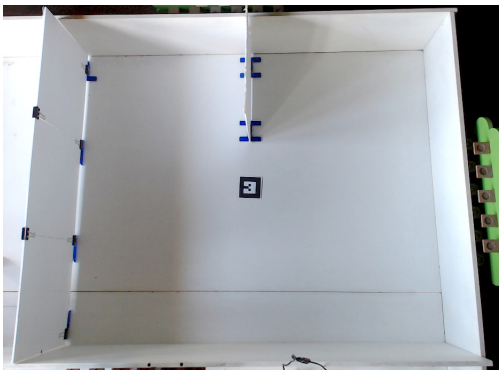


FIGURA 4.20: Escenario utilizado para las pruebas en el ambiente físico. Sus dimensiones son 150cm x 120cm con una pared interna de 40cm. En esta imagen se puede observar el marcador utilizado para marcar el centro del escenario y los ejes x e y

4.5. Resultados

A la hora de llevar un sistema que funciona en una plataforma simulada a una plataforma física aparecen nuevos problemas a resolver. En una plataforma simulada se trabaja en un ambiente controlado, en el cual el usuario decide el error sobre el que va a trabajar; en cambio, en una plataforma real, cada componente físico tiene un error y puede ofrecer dificultades, apareciendo situaciones que no se tuvo en cuenta a la hora de programar el sistema o que solo suceden en ambientes con limitaciones y condiciones reales.

El primer paso para esta transición es configurar el ambiente: instalación de *frameworks*, *middleware* y paquetes necesarios; en la placa a utilizar. Por tratarse de un *hardware* (HW) distinto al de una PC, pueden surgir incompatibilidades con las versiones de los paquetes que se están utilizando. Como sucedió en el caso de este proyecto, a veces hay que cambiar de placa y probar distintos sistemas operativos hasta encontrar el adecuado para ese caso en particular. Una vez que se decidió la placa a utilizar hay que

definir y configurar los otros componentes de HW a utilizar como los sensores y motores. Es importante para este paso, generar pequeños programas que sirvan para probar y calibrar cada componente por separado antes de ser integrados al sistema. Un ejemplo son los motores, que hay que encontrar los valores a los que se mueven, determinar las distintas velocidades y calibrarlos ya que a veces los motores tienen algunas pequeñas diferencias que hace que al avanzar no lo hagan en una línea recta.

Una vez que los componentes físicos están funcionando por separado, hay que integrarlos al sistema previo. En este proyecto como ya se mencionó, se utilizó el patrón de diseño de plantilla para facilitar esta tarea y que los componentes del sistema que se encargan de los sensores y motores sean independientes. Por último, pero no menos importante, es necesario definir el escenario a utilizar y los tamaños de las celdas a utilizar para las pruebas; estos tamaños deben de estar relacionados con el tamaño del robot, como se manejan las celdas durante la exploración y el alcance de la localización y comunicación.
En el caso de este proyecto, el tamaño del escenario estuvo fuertemente limitado por el espacio físico donde se realizaron las pruebas y el sistema de visión utilizado para la localización. Esta última restricción fue principalmente causada porque el sistema de visión no soportaba que la cámara esté en ángulo ni distancias muy grandes porque esto ocasionaba que se perdiera visibilidad sobre el robot, quedando fuera de alcance; estas restricciones no se conocían de antemano por lo que fue necesario realizar reiteradas pruebas hasta lograr un escenario final.
Otros factores que son necesarios tener en cuenta a la hora de correr el sistema en un ambiente físico son la movilidad del robot, en qué tipo de superficie puede desplazarse correctamente; y la luz, especialmente cuando se cuenta con un sistema de localización basado en cámaras, es necesario tener en cuenta la cantidad y el tipo de luz ya sea natural o artificial.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

El proyecto realizado contribuyó en la investigación y comparación de distintos algoritmos necesarios para resolver problemas de asignación de tareas y clusterización de regiones de frontera relacionados a exploración robótica. A partir de esta investigación se determinó que el algoritmo MinPos Glouton implementado con Wavefront Propagation tiene muy buenos resultados independientemente de la cantidad de tareas y de recursos que haya para la asignación de las mismas. En cuanto a los algoritmos utilizados para la clusterización de puntos de frontera se puede concluir que el algoritmo Affinity Propagation es más eficiente que K-means en términos generales; generando conjuntos de centros de clústers de mejores características y con menores tiempos de ejecución para grandes cantidades de puntos a agrupar; por otro lado, K-means muestra mejores tiempos de ejecución para pequeños grupos de puntos de fronteras y al tratarse de pequeños grupos, la calidad de los conjuntos de clústers generados no tiene un impacto tan importante como si sucede con al agrupar grandes cantidades de puntos.

Un objetivo del proyecto consistía en no solo ser capaces de ejecutar con múltiples robots simultáneamente, sino que estos tenían que poder comunicarse y cooperar durante la exploración para ayudar a minimizar el tiempo total de la tarea. En base a esto, se analizó la influencia del agregado progresivo de robots a la flota para la exploración de un mismo mapa; se observó que los primeros robots agregados impactan en forma importante representando una gran mejora en el tiempo total de exploración mientras que luego de cierta cantidad de robots se llega a una meseta. Aunque no se probó, los resultados pueden llevar a pensar que si se continúan agregando robots, eventualmente se llega a un punto en el que agregar un robot más a la exploración genera un impacto negativo en el tiempo total de exploración. Es importante poder determinar el número de robots a partir de cuando comienza a suceder, para poder optimizar la configuración de la flota de robots al máximo. El decaimiento en la mejora del tiempo de exploración

se pude dar por distintos factores como por ejemplo: el espacio físico donde los robots se tienen que desplazar, pudiendo implicar que se estorben unos a otros, teniendo que por ejemplo invertir tiempo en tareas de evasión; o el canal de comunicación puede ser saturado, entre otros.

Un objetivo importante del proyecto y que se cumplió es que la implementación no fuese dependiente de una plataforma en particular, sino que se pudiese adaptar fácilmente para correr en distintas plataformas físicas y simuladas. Esto fue logrado con ayuda del *framework* utilizado, Orocos, que sigue una arquitectura de *Data Flow*, teniendo componentes especializadas para cada una de las tareas que realiza el robot; el diseño basado en plantillas (patrón de diseño *Template Method*), que permite abstraer cada componente de la implementación específica de la plataforma a correr los robots; y por último, la utilización de un archivo XML de propiedades, donde se tienen las configuraciones comunes a todas las plataformas así como las específicas. Algo a tener en cuenta es que a pesar de haber diseñado el sistema de forma de facilitar el agregado de una nueva plataforma con sus componentes, cada nueva plataforma tiene sus propias complejidades que hacen que haya en ciertas ocasiones la necesidad de ajustar algunas partes de los algoritmos que no se habían tenido en cuenta previamente.

Durante el desarrollo del proyecto, las dos principales razones de bloqueo y retrasos que se enfrentaron fueron: el uso de *frameworks* nuevos, que eran desconocidos para el equipo, por lo que hubo que investigar previamente para poder luego desarrollar haciendo uso de los mismos; y la configuración de los ambientes, ya sea en la PC como en la SBC de la BBB, donde hubo que instalar los distintos *frameworks* y *middlewares*, así como lograr su correcto funcionamiento, lo que implica en ciertos casos reimplementar o actualizar ciertas componente debido a limitaciones o variaciones en las versiones disponibles de librerías para las distintas arquitecturas y sistemas operativos.

5.2. Trabajo futuro

Por cuestiones de tiempo, quedaron algunas mejoras pendientes para realizar al sistema, las cuales podrían mejorar aún más los tiempos de exploración y mejorar la eficiencia del sistema.

La primera mejora es sobre la planificación; en este proyecto, cuando un robot planifica cuál es su siguiente objetivo, calcula y asume a que fronteras van a estar asignados los otros robots. Una mejora a este algoritmo es que los robots compartan cuál es su frontera objetivo y utilizar este dato para optimizar el algoritmo de MinPos, partiendo de la base que ya se sabe a dónde va a ir o está yendo el resto del equipo. Al hacer esto dejaría de ser una planificación sincronizada de manera implícita y pasaría a ser explícita.
Una segunda mejora sobre la planificación es sobre los algoritmos de clusterización.

Como se puede observar en la sección de experimentación 4.1, los algoritmos de *K-means* [16] y *Affinity Propagation* [17] varían su comportamiento dependiendo de la cantidad de puntos a clusterizar; *K-means* tiene mejores tiempos cuando el conjunto de puntos es reducido mientras que *Affinity Propagation* tiene mejores tiempos y resultados sobre conjuntos grandes. Una mejora a la solución propuesta es variar que algoritmo se utiliza, dependiendo del conjunto de puntos de frontera a clusterizar; de esta manera, se optimizan los tiempos de planificación totales.
Por último, como se mencionó en la sección 4.1.2, se pueden buscar alternativas para ejecutar MinPos con A* de forma paralela y así mejorar los tiempos de ejecución y que el algoritmo no se vea tan perjudicado cuando crece el número de robots y la cantidad de tareas a asignar. Por otro lado, también se puede mejorar el algoritmo de MinPos con propagación de olas si en vez de realizar la ejecución de las olas de manera serial, se busca la manera de realizar estas propagaciones de manera simultánea en paralelo.

Por otro lado, se puede mejorar el mapa que se obtiene como salida de una exploración teniendo en cuenta donde están los otros robots y en caso de que sense a uno, no agregarlo a mi mapa. En este proyecto se asume que ese robot se va a ir y esa área va a volver a ser sensada cuando esté libre y el mapa se corregiría, pero esto no siempre es así, puede suceder que esa área no vuelva a ser sensada o que no se vuelva a sensar la cantidad de veces necesaria para eliminar al robot del mapa. El inconveniente con este punto es que al utilizar MRPT no sabemos cuáles fueron los nuevos puntos del mapa sensados en una lectura del sensor. Para solucionar esto se pueden tener dos soluciones: calcular en una lectura del sensor, a qué punto en el mapa corresponde cada rayo y consultar si hay algún robot en esa posición; por otro lado, se puede realizar un post procesamiento, marcando como libre las celdas correspondientes a la posición de cada robot. Ambos algoritmos agregan una carga de procesamiento a la integración de una lectura al mapa que debería de ser rápida. En el caso del post procesamiento, es importante tener en cuenta no solo la celda que representa el centro del robot sino también las celdas que lo contienen lo cual dificulta la tarea y los algoritmos ya que no solo hay que tener en cuenta la posición, sino la rotación.

A la hora de la navegación hay dos comportamientos que pueden ser agregados para que la navegación sea más ágil. El primero es evasión de obstáculos, el cual evita que el robot se tranque en caso de que se encuentre con un obstáculo que no había sido previsto o con otro robot. El segundo es des atascamiento utilizando campos potenciales En este caso, lo que se realiza una vez que el robot está atascado, es analizar cómo es el escenario en la zona donde se encuentra y se mueve marcha atrás hacia el lugar donde haya menos obstáculos.
Por otro lado, en la navegación, para facilitar la búsqueda de caminos entre dos puntos en la grilla, se puede tener una representación paralela del escenario utilizando Quadtree [10]. Esta representación del ambiente facilita la búsqueda de los nodos vecinos y de las áreas libres/ocupadas, haciendo más eficiente los algoritmos de búsqueda de caminos.

Una posible mejora, que no aplica directamente sobre la navegación, pero que permite mejorar el tiempo invertido en la misma, es agregar preprocesamiento de las áreas detrás de las fronteras para estudiar su potencial de información a aportar, y decidir si vale la pena el esfuerzo que requiera navegar hasta ellas.

Otra mejora para la exploración, especialmente en lugares estrechos o donde el robot puede haber ciertas posiciones a las que no pueda acceder es el filtrado de tareas a la hora de planificar. Si el robot se encuentra estancado y ya intentó ir a una posición sin éxito, esta posición no la tengo en cuenta para la próxima planificación. De esta manera se evita que la planificación lo siga mandando al mismo punto al que no puede acceder y le envíe uno nuevo al que pueda ir y desbloquearse.

En cuanto a las pruebas con en la plataforma física, se deja como trabajo a futuro realizar pruebas en ambientes más complejos, no solo más extensos sino con más obstáculos o paredes, logrando acercarse a una situación real. También queda para probar con más de un robot como se planteó al comienzo del proyecto, de manera de que colaboren para recorrer el ambiente. Por último, sería recomendable utilizar un sistema de visión más preciso, sin errores y portable hacia otros ambientes donde una cámara no es posible; un ejemplo de esto es un sistema de localización utilizando *beacons* [24].

Bibliografía

[1] M. Juliá, Óscar Reinoso, A. Gil, M. Ballesta, and L. Payá, "A hybrid solution to the multi-robot integrated exploration problem," *Engineering Applications of Artificial Intelligence, 23*, p. 473–486, 2010.

[2] Proyecto butiá. Grupo MINA Facultad de Ingeniería UDELAR. [Online]. Available: https://www.fing.edu.uy/inco/proyectos/butia/

[3] A. Bautin, O. Simonin, and F. Charpillet, "Minpos : A novel frontier allocation algorithm for multi-robot exploration," *ICIRA - 5th International Conference on Intelligent Robotics and Applications*, pp. 496–508, 2012.

[4] Y. Cao, A. Kahng, and A. Fukunaga, "Cooperative mobile robotics: Antecedents and directions." *Autonomous Robots*, vol. 4, no. 1, pp. 7–27, 1997.

[5] K. M. Wurm, C. Stachniss, and W. Burgard, "Coordinated multi-robot exploration using a segmentation of the environment," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, France, 2008.

[6] C. Stachniss, *Robotic Mapping and Exploration.*, ser. Springer Tracts in Advanced Robotics, Albert-Ludwigs-University of Freiburg, Institute of Computer Science, Autonomous Intelligent Systems, 2009, vol. 55.

[7] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, 1985, pp. 116–121.

[8] R. Dia, J. Mottin, T. Rakotovao, D. Puschini, and S. Lesecq, "Evaluation of occupancy grid resolution through a novel approach for inverse sensor modeling," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 13 841 – 13 847, 2017, 20th IFAC World Congress.

[9] E. Einhorn, C. Schröter, and H. M. Gross, "Finding the adequate resolution for grid mapping - cell sizes locally adapting on-the-fly," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1843–1848.

[10] R. A. Finkel and J. L. Bentley, "Quad trees, a data structure for retrieval on composite keys," *Springer-Verlag 1974*, 1974.

[11] B. Yamauchi, "A frontier-based approach for autonomous exploration," *International Symposium on Computational Intelligence in Robotics and Automation.*

[12] ——, "Frontier-based exploration using multiple robots," *Naval Research Laboratory*, pp. 146–151, 1997.

[13] U. Jaina, R. Tiwarib, S. Majumdarc, and S. Sharmad, "Multi robot area exploration using circle partitioning method," *Procedia Engineering 41*, pp. 383–387, 2012.

[14] H. Choset and J. Burdick, "Sensor based planning, part i: The generalized voronoi graph," in *In Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, Nagoya, Japan, 1995.

[15] H. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly, 2*, p. 83–97, 1995.

[16] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics.* Berkeley, Calif.: University of California Press, 1967, pp. 281–297.

[17] F. Brendan J. and D. Delbert, "Clustering by passing messages between data points." *Science*, 2007.

[18] J. C. Elizondo, E. R. G. Ramírez, and J. R. Martínez, "Multi-robot exploration using self-biddings under constraints on communication range," *IEEE LATIN AMERICA TRANSACTIONS, VOL. 14, NO. 2*, p. 971–982, 2016.

[19] E.-L. Juan Carlos, P.-G. Ezra Federico, and R.-T. José Gabriel, "The exact euclidean distance transform: A new algorithm for universal path planning." *International Journal of Advanced Robotic Systems, Vol 10, Iss 6 (2013)*, no. 6, 2013.

[20] W. Sheng, Q. Yang, J. Tan, and N. Xi, "Distributed multi-robot coordination in area exploration," *Robotics and Autonomous Systems 54*, p. 945–955, 2006.

[21] A. Bautin, "Stratégie d'exploration multirobot fondée sur le calcul de champs de potentiels," Ph.D. dissertation, Universite de Lorraine, 2013-10-03.

[22] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," 2004.

[23] A. Bautin, O. Simonin, and F. Charpillet, "Sywap: Synchronized wavefront propagation for multi-robot assignment of spatially-situated tasks," *Universite de Lorraine, LORIA*, 2013.

[24] L. E. Navarro-Serment, C. Paredis, and P. Khosla, "A beacon system for the localization of distributed robotic teams," in *Proceedings of the International Conference on Field and Service Robotics (FSR '99)*, August 1999.

[25] The orocos project, smarter control in robotics & automation! [Online]. Available: http://www.orocos.org/

[26] Rplidar. SLAMTEC. [Online]. Available: https://www.slamtec.com/en/Lidar/A3

[27] "Designing reusable classes," *Journal Object Oriented Program.*, vol. 1, no. 2, pp. 22–35. [Online]. Available: http://www.laputan.org/drc/drc.html

[28] Mrpt, empowering c++ development in robotics. MRPT. [Online]. Available: https://www.mrpt.org/

[29] mrpt::maps::coccupancygridmap2d class reference. MRPT. [Online]. Available: http://mrpt.ual.es/reference/1.4.0/classmrpt_1_1maps_1_1_c_occupancy_grid_map2_d.html

[30] Rapidjson. Tencent. [Online]. Available: http://rapidjson.org/

[31] Yarp. YARP. [Online]. Available: http://www.yarp.it/

[32] A. Degroote, S. Lemaignan, P. Koch, L. N.Y., J. Nicola, C.-E. Hrabia, M. Herrb, and H. Khambhaita. The morse simulator documentation. [Online]. Available: https://www.openrobots.org/morse/doc/stable/morse.html

[33] Blender. [Online]. Available: https://www.blender.org/

[34] Z. Yan, L. Fabresse, J. Laval, and N. Bouraqadi, "Metrics for performance benchmarking of multi-robot exploration," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2015, pp. 3407–3414.

[35] Robocuprescue robot league. [Online]. Available: http://wiki.robocup.org/wiki/Robot_League#Field_Description

[36] Raspberry pi 3 model b. Raspberry Pi Foundation. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[37] Beaglebone black. BeagleBoard.org Foundation. [Online]. Available: https://beagleboard.org/black

[38] Rplidar a1. SLAMTEC. [Online]. Available: https://www.slamtec.com/en/Lidar/A1

[39] Augmented reality / 3d tracking. VTT Technical Research Centre of Finland Ltd. [Online]. Available: http://virtual.vtt.fi/virtual/proj2/multimedia/index.html

Capítulo 6

Anexos

6.1. Código

6.1.1. Script para instalación de máquina virtual y placa

VM: APT GIT OPENROBOTS ROBOTPKG YARP MORSE MORSEYARP PYMORSE OCL OROCOS KDL PYOROCOSKDL PYYARP UPDATEPATH OROCOS_YARP_TRANSPORT MRPT HEADLESS ROS_CAMERA

VM_SPECIFIC: APT GIT OPENROBOTS ROBOTPKG_SPECIFIC YARP MORSE MORSEYARP PYMORSE OCL OROCOS KDL PYOROCOSKDL PYYARP UPDATEPATH OROCOS_YARP_TRANSPORT_SPECIFIC MRPT HEADLESS

BEAGLEBONE: APT GIT OPENROBOTS ROBOTPKG YARP EXTRA_OCL_DEPS OCL OROCOS KDL PYOROCOSKDL UPDATEPATH OROCOS_YARP_TRANSPORT MRPT RPLIDAR_SDK

APT:

```
sudo apt-get update
```

GIT:

```
sudo apt-get -y install git
```

OPENROBOTS:

```
cd /opt/; sudo mkdir openrobots
cd /opt/; sudo chmod 777 openrobots
```

ROBOTPKG:

```
cd /opt/openrobots/; git clone https://git.openrobots.org/robots/robotpkg
  .git
cd /opt/openrobots/robotpkg/bootstrap/; ./bootstrap
```

ROBOTPKG_SPECIFIC:

```
cd /opt/openrobots/; git clone https://git.openrobots.org/robots/robotpkg
  .git
```

```
  cd /opt/openrobots/robotpkg/; git checkout 4
    c6d8c6754f07beb925623ca27c5a9b7bdc4dfab
  cd /opt/openrobots/robotpkg/bootstrap/; ./bootstrap
```

YARP:

```
  sudo apt-get -y install bzip2 libbz2-dev zlib1g-dev libssl-dev libncurses
    -dev pax tar libace-dev
  sudo apt-get -y install cmake g++ libgtk2.0-dev libgtkmm-2.4-dev
    libglademm-2.4-dev libjpeg-dev lua5.2 liblua5.2-dev libopencv-dev libcv
    -dev libhighgui-dev qtbase5-dev qtdeclarative5-dev qtmultimedia5-dev
    libreadline-dev sqlite3 libtinyxml-dev
  cd /opt/openrobots/robotpkg/middleware/yarp/; make update
```

MORSE:

```
  sudo add-apt-repository ppa:fkrull/deadsnakes
  sudo apt-get update
  sudo apt-get -y install blender python3-numpy python3-dev
  cd /opt/openrobots/robotpkg/simulation/morse/; make update
```

MORSEYARP:

```
  sudo apt-get -y install swig
  cd /opt/openrobots/robotpkg/simulation/morse-yarp/; make update
```

EXTRA_OCL_DEPS:

```
  sudo apt-get -y install bzip2 libbz2-dev libssl-dev libncurses-dev pax
    tar libreadline-dev cmake
```

OCL:

```
  sudo apt-get -y install libboost-dev libboost-filesystem-dev libboost-
    iostreams-dev libboost-math-dev libboost-thread-dev doxygen
    liblog4cxx10-dev libnetcdf-dev libxerces-c-dev libboost-program-options
    -dev
  cd /opt/openrobots/robotpkg/architecture/orocos-ocl/; make update
```

OROCOS:

```
  cd /opt/openrobots/robotpkg/meta-pkgs/orocos-toolchain/; make update
```

KDL:

```
  sudo apt-get -y install libeigen3-dev
  cd /opt/openrobots/robotpkg/motion/orocos-kdl/; make update
```

PYOROCOSKDL:

```
  sudo apt-get -y install python-sip-dev python2.7-dev
  cd /opt/openrobots/robotpkg/motion/py-orocos-kdl/; make update
```

PYYARP:

```
  cd /opt/openrobots/robotpkg/middleware/py-yarp/; make update
```

UPDATEPATH:

```
  echo 'export PATH="/opt/openrobots/bin:$PATH"' >> ~/.bashrc
  export PATH="/opt/openrobots/bin:$PATH"
```

OROCOS_YARP_TRANSPORT:

```
cd /opt/; sudo mkdir orocos-yarp-transport
cd /opt/; sudo chmod 777 orocos-yarp-transport
git clone https://github.com/adegroote/orocos-yarp-transport.git /opt/
  orocos-yarp-transport
cd /opt/orocos-yarp-transport/; cmake . -DCREATE_SHARED_LIBRARY=ON -
  DCREATE_YARPSERVER3=ON -DCREATE_GUIS=ON
cd /opt/orocos-yarp-transport/; sudo make install
cd /usr/local/lib/orocos/gnulinux/yarp/plugins/; sudo cp *.* /opt/
  openrobots/lib/orocos/gnulinux/types/
```

OROCOS_YARP_TRANSPORT_SPECIFIC:

```
cd /opt/; sudo mkdir orocos-yarp-transport
cd /opt/; sudo chmod 777 orocos-yarp-transport
git clone https://github.com/adegroote/orocos-yarp-transport.git /opt/
  orocos-yarp-transport
cd /opt/orocos-yarp-transport/; git checkout
  c28e3ca4c9998a8be127fa46d6b740cc87326e06
cd /opt/orocos-yarp-transport/; cmake . -DCREATE_SHARED_LIBRARY=ON -
  DCREATE_YARPSERVER3=ON -DCREATE_GUIS=ON
cd /opt/orocos-yarp-transport/; sudo make install
cd /usr/local/lib/orocos/gnulinux/yarp/plugins/; sudo cp *.* /opt/
  openrobots/lib/orocos/gnulinux/types/
```

MRPT:

```
sudo add-apt-repository ppa:joseluisblancoc/mrpt
sudo apt-get update
sudo apt-get -y install libmrpt-dev mrpt-apps
```

HEADLESS: MESA XVFB

MESA:

```
sudo apt-get -y install llvm-dev scons python-mako libedit-dev
sudo apt-get -y build-dep mesa
cd /opt/; sudo wget ftp://ftp.freedesktop.org/pub/mesa/mesa-17.2.0-rc2.
  tar.gz
cd /opt/; sudo tar -xvf mesa-17.2.0-rc2.tar.gz
sudo mv /opt/mesa-17.2.0-rc2/ /opt/mesa/
cd /opt/; sudo rm mesa-17.2.0-rc2.tar.gz
cd /opt/; sudo chmod -R 777 mesa
cd /opt/mesa/; scons build=release llvm=yes libgl-xlib
```

XVFB:

```
sudo apt-get -y install xvfb
```

PYMORSE:

```
cd /opt/; sudo wget https://pypi.python.org/packages/60/c9/0691395964
  caf265a4f6e5d81e9bedad7c44c019af7d4158a031469e394c/pymorse-1.4.tar.gz
cd /opt/; sudo tar -xvf pymorse-1.4.tar.gz
cd /opt/; sudo rm pymorse-1.4.tar.gz
```

```
  cd /opt/; sudo chmod -R 777 pymorse-1.4
  cd /opt/pymorse-1.4/; sudo python setup.py install

RPLIDAR_SDK:
  sudo apt-get -y install unzip
  cd /opt/; sudo wget http://bucket.download.slamtec.com/95452
    c5f32181b75ef151dbc62a3e2f59155de4c/rplidar_sdk_v1.5.7.zip
  cd /opt/; sudo unzip rplidar_sdk_v1.5.7.zip -d rplidar
  cd /opt/; sudo rm rplidar_sdk_v1.5.7.zip
  cd /opt/; sudo chmod -R 777 rplidar
  cd /opt/rplidar/sdk/; make
  cp "/opt/rplidar/sdk/output/Linux/Release/librplidar_sdk.a" "/opt/
    openrobots/lib/librplidar_sdk.a"
  cd /opt/; sudo rm -r rplidar/*
  cp "/opt/openrobots/lib/librplidar_sdk.a" "/opt/rplidar/librplidar_sdk.a"

ROS_CAMERA: ROS_KINETIC USB_CAMERA ROS_SETUP

ROS_KINETIC:
  sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -
    sc) main" > /etc/apt/sources.list.d/ros-latest.list'
  sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-
    key 421C365BD9FF1F717815A3895523BAEEB01FA116
  sudo apt-get update
  sudo apt-get install ros-kinetic-desktop-full
  sudo rosdep init
  rosdep update
  echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
  source ~/.bashrc

USB_CAMERA:
  sudo apt-get install python-rosinstall python-rosinstall-generator python
    -wstool build-essential
  sudo apt-get install ros-kinetic-usb-cam
  sudo apt-get install ros-kinetic-camera-calibration

ROS_SETUP:
  mkdir -p ~/catkin_ws/src
  cd ~/catkin_ws/; catkin_make
  cd ~/catkin_ws/; source devel/setup.bash
  echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
  echo "----- Copy your package folder to ~/catkin_ws/src and run
    catkin_make -----"
```

6.1.2. Script para ejecutar morse

6.1.2.1. Script para ejecutar la simulación de todos los robots en un mismo nodo

```
from morse.builder import *
import os

# robot positions up to 9 robots
robot_positions = [[-36,-36,0], [-26,-36,0], [-36,-26,0], [-16,-36,0],
    [-36,-16,0], [-6,-36,0], [-36,-6,0], [4,-36,0], [-36,4,0]]

# ROBOT_COUNT is an environment variable with the number of robots <= 9
robot_count = int(os.environ["ROBOT_COUNT"])

if (robot_count < 1 or robot_count > 9):
    print ('ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 9')

else:
    delta = (robot_count + 1) %2
    for id in range(0, robot_count):
        # creates a new instance of the robot
        robot = ATRV("butia-" + str(id))
        robot.rotate(0, 0, 1.5708) # radians
        robot.translate(robot_positions[id + delta][0], robot_positions[id
    + delta][1], robot_positions[id + delta][2])
        robot.properties(GroundRobot = True)

        ## retrieves the actual position of the robot
        # OUT: timestamp x y z yaw(rot z) pitch(rot y) roll(rot x)
        pose = Pose("pose-" + str(id))
        pose.add_stream("yarp") # PORT: /morse/robot/pose/out
        robot.append(pose)

        ## controls the robot's engines
        # IN: {"id": "<id>", "component":"robot.motion", "service", "<
    service>", "params": [<params>]}
        motion = Waypoint("motion-" + str(id))
        motion.properties(FreeZ = True)
        motion.properties(ObstacleAvoidance = True)
        motion.properties(Speed = 2.0)
        motion.add_service("yarp_json") # PORT: /ors/services/robot.motion/
    request or reply
        robot.append(motion)

        ## laser range sensor
        # OUT: dictionary of the current sensor data
        sick = Sick("sick-" + str(id))
        sick.translate(0, 0, 0)
        sick.rotate(0, 0, 0)
        sick.properties(scan_window = 360)
        sick.properties(laser_range = 10)
        sick.properties(resolution = 1)
        sick.properties(Visible_arc = False)
```

```
        sick.add_service('yarp_json') # PORT: /ors/services/robot.sick/
    request or reply
        robot.append(sick)

    ## environment and camera setup
    # fastmode = true to remove graphics
    env = Environment('custom_maps/maze', fastmode = True)
    env.set_camera_location([0, 0, 100])
    env.set_camera_rotation([0, 0, 0])
    env.show_framerate(True)
    env.use_internal_syncer()
```

6.1.2.2. Script para ejecutar la simulación de cada robot en un nodo distinto

```
from morse.builder import *
import os

# robot positions up to 9 robots
robot_positions = [[-36,-36,0], [-26,-36,0], [-36,-26,0], [-16,-36,0],
    [-36,-16,0], [-6,-36,0], [-36,-6,0], [4,-36,0], [-36,4,0]]

# morse nodes
morse_nodes = ['node0', 'node1', 'node2', 'node3', 'node4', 'node5', 'node6
    ', 'node7', 'node8']
nodes_dist = {}

# ROBOT_COUNT is an environment variable with the number of robots <= 9
robot_count = int(os.environ["ROBOT_COUNT"])

if (robot_count < 1 or robot_count > 9):
    print ('ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 9')

else:
    delta = (robot_count + 1) %2
    for id in range(0, robot_count):
        # creates a new instance of the robot
        robot = ATRV("butia-" + str(id))
        robot.rotate(0, 0, 1.5708) # radians
        robot.translate(robot_positions[id + delta][0], robot_positions[id
    + delta][1], robot_positions[id + delta][2])
        robot.properties(GroundRobot = True)

        ## retrieves the actual position of the robot
        # OUT: timestamp x y z yaw(rot z) pitch(rot y) roll(rot x)
        pose = Pose("pose-" + str(id))
        pose.add_stream("yarp") # PORT: /morse/robot/pose/out
        robot.append(pose)

        ## controls the robot's engines
```

```
        # IN: {"id": "<id>", "component":"robot.motion", "service", "<
    service>", "params": [<params>]}
        motion = Waypoint("motion-" + str(id))
        motion.properties(FreeZ = True)
        motion.properties(ObstacleAvoidance = True)
        motion.properties(Speed = 2.0)
        motion.add_service("yarp_json") # PORT: /ors/services/robot.motion/
    request or reply
        robot.append(motion)

        ## laser range sensor
        # OUT: dictionary of the current sensor data
        sick = Sick("sick-" + str(id))
        sick.translate(0, 0, 0)
        sick.rotate(0, 0, 0)
        sick.properties(scan_window = 360)
        sick.properties(laser_range = 10)
        sick.properties(resolution = 1)
        sick.properties(Visible_arc = False)
        sick.add_service('yarp_json') # PORT: /ors/services/robot.sick/
    request or reply
        robot.append(sick)

        nodes_dist[morse_nodes[id]] = [robot.name]

    ## environment and camera setup
    # fastmode = true to remove graphics
    env = Environment('custom_maps/maze', fastmode = True)
    env.set_camera_location([0, 0, 100])
    env.set_camera_rotation([0, 0, 0])
    env.use_internal_syncer()
    env.configure_multinode(protocol='socket', distribution=nodes_dist)
```

6.1.3. Script para ejecutar la simulación

6.1.3.1. Script para ejecutar la simulación de todos los robots en un mismo nodo

```
robot_count=$1

if (( !robot_count || robot_count < 1 || robot_count > 9 ))
then
  echo 'ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 9'
  exit
fi

echo '========== SIMULATION Collaborative multirobot exploration
    =========='
echo ''
```

```
echo '========== START YARPSERVER ==========='
gnome-terminal -t "YARPSERVER3" -e "bash -c \" yarpserver3 --write \"" &
sleep 2

echo '========== START MORSE ==========='
gnome-terminal -t "MORSE" -e "bash -c \"export ROBOT_COUNT=$robot_count;
    morse run simulator-morse/simulator_multirobot.py\"" &
sleep 5

echo '========== START DEPLOYER OROCOS ==========='
for (( i=0; i < $robot_count; ++i ))
do
  echo "========== OROCOS FOR ROBOT $i ==========="
  gnome-terminal -t "DEPLOYER OROCOS $i" -e "bash -c \"export ROBOT_COUNT=
    $robot_count; export ROBOT_ID=$i; export RTT_COMPONENT_PATH=$(pwd)/
    install/lib/orocos:$RTT_COMPONENT_PATH; deployer-gnulinux -ldebug -s
    robot-exploration/OrocosScripts/Simulated/butia$i.ops Deployer$i\"" &
  # REAL-TIME LOGGING # gnome-terminal -t "DEPLOYER OROCOS $i" -e "bash -c
    \"export ROBOT_COUNT=$robot_count; export ROBOT_ID=$i; export
    RTT_COMPONENT_PATH=$(pwd)/install/lib/orocos:$RTT_COMPONENT_PATH;
    deployer-gnulinux -lrealtime -s robot-exploration/butia$i.ops
    Deployer$i\"" &
  sleep 2
done
sleep 3

# get name of the machine:
nameVM=`uname -n`

echo 'Press any key to connect Orocos and Morse YARP ports. Or press "q" to
     quit.'
read touche
while [ "$touche" != "q" ]
do
  for (( p=0; p < $robot_count; ++p ))
  do
    echo "========== CONNECT FOR ROBOT $p ==========="

    for (( m=0; m < $robot_count; ++m ))
    do
      if (( m != p ))
      then
        yarp connect /$nameVM/communication$p/broadcastToSwarm/out /$nameVM
    /communication$m/listenToSwarm/in mcast
        yarp connect /$nameVM/communication$p/broadcastPositionToSwarm/out
    /$nameVM/communication$m/listenToSwarmPositions/in mcast
      fi
    done

    yarp connect /morse/butia-$p/pose-$p/out /$nameVM/gps$p/readPosition/in
```

```
    yarp connect /ors/services/butia-$p.sick-$p/reply /$nameVM/radar$p/
    replyRadar/in
    yarp connect /$nameVM/radar$p/requestRadar/out /ors/services/butia-$p.
    sick-$p/request
    yarp connect /$nameVM/engine$p/writeWaypoint/out /ors/services/butia-$p
    .motion-$p/request
  done

  echo 'Press "q" to quit.'
  read touche
done

killall gnome-terminal
killall gnome-terminal-server
```

6.1.3.2. Script para ejecutar la simulación de cada robot en un nodo distinto

```
robot_count=$1

if (( !robot_count || robot_count < 1 || robot_count > 9 ))
then
  echo 'ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 9'
  exit
fi

echo '========== SIMULATION Collaborative multirobot exploration
    =========='
echo ''
echo '========== SATART YARPSERVER =========='
gnome-terminal -t "YARPSERVER3" -e "bash -c yarpserver3" &
sleep 2

echo '========== SATART MORSE =========='
gnome-terminal -t "MULTINODE SERVER" -e "bash -c \"export PYTHONPATH=
    $PYTHONPATH:/usr/local/lib/python2.7/dist-packages/; multinode_server\"
    "
sleep 2

gnome-terminal -t "MORSE node0" -e "bash -c \"export MORSE_NODE=node0;
    export ROBOT_COUNT=$robot_count; Xvfb -screen 0 100x100x8 :1 &
    LD_LIBRARY_PATH=/opt/mesa/build/linux-x86_64/gallium/targets/libgl-xlib
     DISPLAY=:1 morse run simulator-morse/simulator_multinode.py\"" &
for (( i=1; i < $robot_count; ++i ))
do
  echo "========== MORSE NODE FOR ROBOT $i =========="
  gnome-terminal -t "MORSE node$i" -e "bash -c \"export MORSE_NODE=node$i;
    export ROBOT_COUNT=$robot_count; Xvfb -screen 0 100x100x8 :1 &
    LD_LIBRARY_PATH=/opt/mesa/build/linux-x86_64/gallium/targets/libgl-xlib
     DISPLAY=:1 morse run simulator-morse/simulator_multinode.py\"" &
```

```
  sleep 1
done
sleep 2

echo '========== SATART DEPLOYER OROCOS ==========='
for (( i=0; i < $robot_count; ++i ))
do
  echo "========== OROCOS FOR ROBOT $i ==========="
  gnome-terminal -t "DEPLOYER OROCOS $i" -e "bash -c \"export ROBOT_COUNT=
    $robot_count; export ROBOT_ID=$i; export RTT_COMPONENT_PATH=$(pwd)/
    install/lib/orocos:$RTT_COMPONENT_PATH; deployer-gnulinux -ldebug -s
    robot-exploration/OrocosScripts/Simulated/butia$i.ops Deployer$i\"" &
  sleep 2
done
sleep 3

# get name of the machine:
nameVM=`uname -n`

echo 'Press any key to connect Orocos and Morse YARP ports. Or press "q" to
     quit.'
read touche
while [ "$touche" != "q" ]
do
  for (( p=0; p < $robot_count; ++p ))
  do
    echo "========== CONNECT FOR ROBOT $p ==========="

    for (( m=0; m < $robot_count; ++m ))
    do
      if (( m != p ))
      then
        yarp connect /$nameVM/communication$p/broadcastToSwarm/out /$nameVM
    /communication$m/listenToSwarm/in mcast
        yarp connect /$nameVM/communication$p/broadcastPositionToSwarm/out
    /$nameVM/communication$m/listenToSwarmPositions/in mcast
      fi
    done

    yarp connect /morse/butia-$p/pose-$p/out /$nameVM/gps$p/readPosition/in
    yarp connect /ors/services/butia-$p.sick-$p/reply /$nameVM/radar$p/
    replyRadar/in
    yarp connect /$nameVM/radar$p/requestRadar/out /ors/services/butia-$p.
    sick-$p/request
    yarp connect /$nameVM/engine$p/writeWaypoint/out /ors/services/butia-$p
    .motion-$p/request
  done

  echo 'Press "q" to quit.'
  read touche
done
```

```
killall gnome-terminal
killall gnome-terminal-server
```

6.1.3.3. Script para ejecutar la simulación sin interfaz gráfica

```
robot_count=$1

if (( !robot_count || robot_count < 1 || robot_count > 9 ))
then
  echo 'ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 9'
  exit
fi

echo '========== SIMULATION Collaborative multirobot exploration
    =========='
echo ''
echo '========== START YARPSERVER ==========='
gnome-terminal -t "YARPSERVER3" -e "bash -c yarpserver3" &
sleep 2

echo '========== START MORSE ==========='
gnome-terminal -t "MORSE" -e "bash -c \"export ROBOT_COUNT=$robot_count;
    Xvfb -screen 0 100x100x8 :1 & LD_LIBRARY_PATH=/opt/mesa/build/linux-
    x86_64/gallium/targets/libgl-xlib DISPLAY=:1 morse run simulator-morse/
    simulator_multirobot.py\"" &
sleep 3

echo '========== START DEPLOYER OROCOS ==========='
for (( i=0; i < $robot_count; ++i ))
do
  echo "========== OROCOS FOR ROBOT $i ==========="
  gnome-terminal -t "DEPLOYER OROCOS $i" -e "bash -c \"export ROBOT_COUNT=
    $robot_count; export ROBOT_ID=$i; export RTT_COMPONENT_PATH=$(pwd)/
    install/lib/orocos:$RTT_COMPONENT_PATH; deployer-gnulinux -ldebug -s
    robot-exploration/OrocosScripts/Simulated/butia$i.ops Deployer$i\"" &
  sleep 1
done
sleep 3

# get name of the machine:
nameVM=`uname -n`

echo 'Press any key to connect Orocos and Morse YARP ports. Or press "q" to
    quit.'
read touche
while [ "$touche" != "q" ]
do
  for (( p=0; p < $robot_count; ++p ))
```

```
  do
    echo "========== CONNECT FOR ROBOT $p ==========="

    for (( m=0; m < $robot_count; ++m ))
    do
      if (( m != p ))
      then
        yarp connect /$nameVM/communication$p/broadcastToSwarm/out /$nameVM
    /communication$m/listenToSwarm/in mcast
        yarp connect /$nameVM/communication$p/broadcastPositionToSwarm/out
    /$nameVM/communication$m/listenToSwarmPositions/in mcast
      fi
    done

    yarp connect /morse/butia-$p/pose-$p/out /$nameVM/gps$p/readPosition/in
    yarp connect /ors/services/butia-$p.sick-$p/reply /$nameVM/radar$p/
    replyRadar/in
    yarp connect /$nameVM/radar$p/requestRadar/out /ors/services/butia-$p.
    sick-$p/request
    yarp connect /$nameVM/engine$p/writeWaypoint/out /ors/services/butia-$p
    .motion-$p/request
  done

  echo 'Press "q" to quit.'
  read touche
done

killall gnome-terminal
killall gnome-terminal-server
```

6.1.4. Scripts para ejecutar en el Butiá

6.1.4.1. Script para ejecutar el servidor

```
robots_ips="1 192.168.1.101"

echo '========== BUTIA SERVER YARP AND LOCATION ==========='
echo ''

echo '========== START YARP ==========='
gnome-terminal -t "YARPSERVER3" -e "bash -c \" yarpserver3 \" " &
sleep 5

echo '========== START VISION ==========='
gnome-terminal -t "VISION" -e "bash -c \" roslaunch butia_location vision.
    launch \" " &
sleep 2

echo '========== START RVIZ ==========='
```

```
gnome-terminal -t "RVIZ" -e "bash -c rviz" &
sleep 5

echo '========== START CALIBRATION ==========='
gnome-terminal -t "CALIBRATION" -e "bash -c \" rostopic pub /
    ar_calibration_marker/enable_detection std_msgs/Bool false \" " &
sleep 5

echo '========== START POSITION PROXY ==========='
gnome-terminal -t "PROXY" -e "bash -c \" rosrun butia_location
    position_proxy.py $robots_ips \" " &
sleep 2

echo 'Press "q" to quit.'
read touche
while [ "$touche" != "q" ]
do
  read touche
done

killall gnome-terminal
killall gnome-terminal-server
```

6.1.4.2. Script para ejecutar el proyecto en la placa

```
robot_count=$1
robot_id=$2

if (( !robot_count || robot_count < 1 || robot_count > 4 ))
then
  echo 'ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 4'
  exit
fi

if (( !robot_id ))
then
  echo 'ERROR: you should set the id of the robot > 1'
  exit
fi

echo 'You should now run runButiaConnections.sh to connect the YARP ports'
echo ''

echo '========== START DEPLOYER OROCOS ==========='
bash -c \"export ROBOT_COUNT=$robot_count; export ROBOT_ID=$robot_id;
    export RTT_COMPONENT_PATH=$(pwd)/install/lib/orocos:$RTT_COMPONENT_PATH
    ; deployer-gnulinux -ldebug -s robot-exploration/OrocosScripts/Butia/
    butia_board$robot_id.ops \"
sleep 2
```

6.1.4.3. Script para conectar los puertos de los distintos robots

```
robot_count=$1
robot_id=$2

if (( !robot_count || robot_count < 1 || robot_count > 4 ))
then
  echo 'ERROR: Non-acceptable number of robots, 1 <= ROBOT_COUNT <= 4'
  exit
fi

if (( !robot_id ))
then
  echo 'ERROR: you should set the id of the robot >= 1'
  exit
fi

# name of the machine:
nameBoard='arm'

for (( m=1; m <= $robot_count; ++m ))
do
  if (( m != $robot_id ))
  then
    yarp connect /$nameBoard/communication$robot_id/broadcastToSwarm/out /
    $nameBoard/communication$m/listenToSwarm/in mcast
    yarp connect /$nameBoard/communication$robot_id/
    broadcastPositionToSwarm/out /$nameBoard/communication$m/
    listenToSwarmPositions/in mcast
  fi
done
```

6.1.5. Archivo de propiedades

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
  <!--
  <simple name="IParam" type="short">
    <description>Param Description</description>
    <value>5</value>
  </simple>
  <simple name="DParam" type="double">
    <description>Param Description</description>
    <value>-3.0</value>
  </simple>
  <struct name="SubBag" type="PropertyBag">
    <description>SubBag Description</description>
    <simple name="SParam" type="string">
```

```
      <description>Param Description</description>
      <value>The String</value>
    </simple>
    <simple name="BParam" type="boolean">
      <description>Param Description</description>
      <value>0</value>
    </simple>
  </struct>
  -->
<!-- shared properties between simulated and physical environments -->
  <struct name="Shared" type="PropertyBag">
    <simple name="IS_SIMULATION" type="boolean">
      <description>True if run as simulation</description>
      <value>0</value>
    </simple>
    <simple name="GRID_CELL_SIZE" type="double">
      <description>Size of a cell in the grid in meters</description>
      <value>0.1</value>
    </simple>
    <simple name="GRID_WIDTH" type="double">
      <description>Width of the grid in meters</description>
      <value>1.5</value>
    </simple>
    <simple name="GRID_HEIGHT" type="double">
      <description>Height of the grid in meters</description>
      <value>1.8</value>
    </simple>
    <simple name="ENABLE_LOGGING" type="boolean">
      <description>Enables or disables logging.</description>
      <value>1</value>
    </simple>
    <simple name="LOG_STATISTICS" type="boolean">
      <description>Enables or disables logging the statistics through time.
    </description>
      <value>1</value>
    </simple>
    <simple name="DRAW_MAPS" type="boolean">
      <description>Enables or disables drawing maps.</description>
      <value>1</value>
    </simple>
    <simple name="LOG_PATH" type="boolean">
      <description>Enables or disables logging the path followed by the
    robot.</description>
      <value>1</value>
    </simple>
    <simple name="ROBOT_ID" type="double">
      <description>ID of the robot relative to the swarm.</description>
      <value>0</value>
    </simple>
    <simple name="ROBOT_COUNT" type="double">
      <description>Number of robots within the swarm.</description>
```

```
      <value>1</value>
    </simple>
  </struct>

<!-- property values for simulation -->
  <struct name="Simulation" type="PropertyBag">
  <!-- RADAR -->
    <simple name="RADAR_ANGLE" type="double">
      <description>Value of the radar sensor resolution in radian</
    description>
      <value>0.0174533</value> <!-- 1 degree -->
    </simple>
    <simple name="RADAR_RANGE" type="double">
      <description>Value of the radar sensor range lecture in meters</
    description>
      <value>10</value>
    </simple>
    <simple name="RADAR_RIGHT_TO_LEFT" type="boolean">
      <description>True if the radar recture is from right to left</
    description>
      <value>1</value>
    </simple>

  <!-- ENGINE -->
    <simple name="ENGINE_SPEED" type="double">
      <description>The speed the robot should move at</description>
      <value>2.0</value>
    </simple>
    <simple name="ENGINE_PRECISION" type="double">
      <description>The precision to use to determine if the robot arrived
    to a certain location</description>
      <value>0.05</value>
    </simple>

  <!-- NAVIGATOR -->
    <simple name="ROBOT_LENGTH" type="double">
      <description>The length of the robot in meters. Needed to know when
    to stop.</description>
      <value>1.14</value>
    </simple>
    <simple name="ROBOT_WIDTH" type="double">
      <description>The width of the robot in meters. Needed to decide
    feasible paths for the robot.</description>
      <value>0.99062</value>
    </simple>
  </struct>

<!-- property values for physical -->
  <struct name="Physical" type="PropertyBag">
  <!-- RADAR -->
    <simple name="RADAR_ANGLE" type="double">
```

```
    <description>Value of the radar sensor resolution in radian</
  description>
    <value>0.0174533</value> <!-- 1 degree -->
  </simple>
  <simple name="RADAR_RANGE" type="double">
    <description>Value of the radar sensor range lecture in meters</
  description>
    <value>1</value>
  </simple>
  <simple name="RADAR_RIGHT_TO_LEFT" type="boolean">
    <description>True if the radar recture is from right to left</
  description>
    <value>0</value>
  </simple>
  <simple name="COM_BAUDRATE" type="double">
    <description>Value of the radar sensor Baud rate</description>
    <value>115200</value>
  </simple>
  <simple name="COM_PORT" type="string">
    <description>Value of the port to which the radar sensor is connected
  </description>
    <value>/dev/ttyUSB0</value>
  </simple>

<!-- NAVIGATOR -->
  <simple name="ROBOT_LENGTH" type="double">
    <description>The length of the robot in meters. Needed to know when
  to stop.</description>
    <value>0.25</value>
  </simple>
  <simple name="ROBOT_WIDTH" type="double">
    <description>The width of the robot in meters. Needed to decide
  feasible paths for the robot.</description>
    <value>0.29</value>
  </simple>

<!-- ENGINE -->
  <simple name="ENGINE_SPEED" type="double">
    <description>The speed the robot should move</description>
    <value>0</value>
  </simple>
  <simple name="ENGINE_PRECISION" type="double">
    <description>The precision to use to determine if the robot arrived
  to a certain location</description>
    <value>0.05</value>
  </simple>

  <simple name="LEFT_SERVO_PATH" type="string">
    <description>The path to the left pwm controller</description>
    <value>/sys/class/pwm/pwmchip6/pwm1</value> <!-- P8.13 -->
  </simple>
```

```
    <simple name="RIGHT_SERVO_PATH" type="string">
      <description>The path to the right pwm controller</description>
      <value>/sys/class/pwm/pwmchip3/pwm0</value> <!-- P9.14 -->
    </simple>

    <simple name="LEFT_SERVO_MULTIPLIER" type="double">
      <description>Multiplier for the velocity of the left servo, 1 to use
    the default one</description>
      <value>1.0</value>
    </simple>
    <simple name="RIGHT_SERVO_MULTIPLIER" type="double">
      <description>Multiplier for the velocity of the right servo, 1 to use
     the default one</description>
      <value>1</value>
    </simple>

  <!-- LOCATION -->
    <simple name="LOCATION_SERVER_IP" type="string">
      <description>The ip where the location is published</description>
      <value>0.0.0.0</value>
    </simple>
    <simple name="LOCATION_BASE_PORT" type="double">
      <description>The base port where the location is published</
    description>
      <value>7000</value>
    </simple>
  </struct>

</properties>
```

6.2. Manual del usuario

6.2.1. Repositorio

La estructura de carpetas en el repositorio es la siguiente:

- **butia-location**
  Configuración para el sistema de visión.
- **code-documentation**
  Documentación del código en formato HTML, LaTex y PDF
- **environment-setup**
  Makefile para instalación y SDK para el sensor de rango RPLidar
- **robot-exploration**
  Código principal del proyecto.

- **simulator-morse**
  Scripts y mapas para la simulación con MORSE.Scripts y mapas para la simulación con MORSE.

6.2.2. Manual de instalación en máquina virtual

Los sistemas operativos que fueron probados y son soportados son Ubuntu 14.04 y ubuntu 16.04. Para instalar los paquetes necesarios simplemente es necesario navegar hasta la carpeta *environment-setup* y correr el comando *make VM*. En caso de que ocurra un error, los mismos se pueden deber a que algunos de los repositorios que se clonaron hayan tenido cambios que requieren nuevas librerías, o nuevas versiones de las mismas. En ese caso se puede correr: *make VM_SPECIFIC*; este target utilizará la versión exacta de los repositorios existente al momento de realizar el script de automatización de configuración del ambiente.

6.2.3. Manual de instalación en máquina placa BBB

1. Instalación de Ubuntu 16.04
   - descargar imagen de https://rcn-ee.com/rootfs/2017-10-12/microsd/bone-ubuntu-16.04.3-console-armhf-2017-10-12-2gb.img.xz
   - Extraer la imagen y grabarla en la tarjeta SD
   - Apretar el botón de boot de la placa y enchufarla usando el USB a la PC y soltarlo cuando las luces comiencen a titilar
   - Una vez haya terminado el proceso se va a prender la placa y va a quedar accesible para uso en el puerto 192.168.7.2
     usuario: ubuntu
     contraseña: temppwd
2. Configurar el WiFi
3. Expandir la partición para que ocupe todo el espacio de la SD.

```
cd /opt/scripts/tools
git pull
sudo ./grow_partition.sh
sudo reboot
```

4. Crear una partición de swap.
   Utilizar la configuración *bs=1K count=4M*
5. Configurar el timezone y en caso de no tener acceso a internet es importante configurarle la hora ya que las BBB cuando se apagan pierden el reloj.

6. Una vez que la placa está configurada, se copia el makefile en *environment-setup* hacia la placa y se corre el comando *make BEAGLEBONE*.
   Si al instalar YARP tira el error **"error: narrowing conversion of '-1' from 'int' to 'char' inside [-Wnarrowing]"**, usar version YARP 2.3.70 (2017-06-15). Para esto es necesario:

   - descargar el archivo https://github.com/robotology/yarp/archive/v2.3.70.tar.gz
   - copiarlo en la carpeta */opt/openrobots/robotpkg/distfiles con el nombre v2.3.70.tar.gz*
   - en el Makefile en */opt/openrobots/robotpkg/middleware/yarp cambiar la versión a 2.3.70*
   - ejecutar *"make NO_CHECKSUM=yes install"*

7. Archivos necesarios para copiar desde el proyecto hacia la placa:

   - environment-setup/makefile
   - runButia.sh
   - runButiaConnections.sh
   - configPINS.sh
   - la carpeta robot-exploration: *rsync -avzhe ssh –delete –exclude "build/". ubuntu@192.168.7.2:/home/ubuntu/*

6.2.4. Compilar el proyecto

Dentro de la carpeta *robot-exploration* se encuentran los archivos: *CMakeLists_butia.txt* y *CMakeLists_simulation.txt*. A la hora de compilar, según para que plataforma se esté compilando es necesario copiar el archivo correspondiente a *CMakeLists.txt*.

6.2.5. Ejecutar la simulación

Para ejecutar la simulación primero es necesario configurar el archivo de propiedades en *robot-exploration/helpers/properties/properties.cpf* con los parámetros correspondientes y luego, parado en la raíz del proyecto ejecutar el comando y seguir los pasos que se indican:

- En caso de querer ejecutar la simulación con todos los robots ejecutando en un mismo nodo: *./runMultiple.sh #ROBOT_COUNT*
- En caso de querer ejecutar la simulación con cada robot ejecutando sobre un nodo distinto: *./runMultiNode.sh #ROBOT_COUNT*
- En caso de querer ejecutar la simulación sin interfaz gráfica, utilizada para las pruebas: *./runHeadless.sh #ROBOT_COUNT*

6.2.6. Ejecutar en el Butiá

1. Una vez que la placa está encendida y se verifica que el sensor de rango también lo está, es necesario copiar a la placa y correr el archivo que se encuentra en la raíz del proyecto: *configPINS.sh*; este se encarga de dar los permisos necesarios para poder controlar los motores y el sensor de rango.

2. Colocar el marcador número 0 en el escenario marcando el centro y los ejes. En la PC que va a actuar como servidor, modificar el script que se encuentra en la raíz del proyecto *./runButiaServer.sh*, listando las IP de los distintos robots: *robots_ips="#ROBOT_COUNT IP_ROBOT_1 IP_ROBOT_ .."* y ejecutarlo.

3. en la placa, configurar el archivo de propiedades en *robot-exploration/helpers/properties/properties.*
y luego ejecutar el script *./runButia #ROBOT_COUNT #ROBOT_IP*

4. una vez que están todos los robots iniciados, correr en cada uno el script *./runButiaConnections #ROBOT_COUNT #ROBOT_IP*, encargado de generar las conexiones necesarias entre los robots para la comunicación de los mapas y las posiciones.