ARQUITECTURA DEL ROBOT BOMBERO

SIMULADOR IRSIM

Memoria del diseño y programación de un robot autónomo con arquitectura basada en el comportamiento y en el conocimiento

ÁLVARO GARCÍA RUIZ-ESCRIBANO | GRUPO 10

ÍNDICE GENERAL

INTRODUCCIÓN

Este trabajo tiene como objetivo el desarrollo de un robot autónomo capaz de realizar tareas de rescate y extinción de incendios en un entorno cerrado simulado. El robot debe ser capaz de detectar y evitar obstáculos, localizar y apagar incendios, identificar y rescatar personas atrapadas, así como gestionar de forma eficiente su capacidad de agua mediante estaciones de recarga.

Para el control de su comportamiento se implementarán dos arquitecturas complementarias. En primer lugar, se desarrollará una arquitectura basada en el comportamiento, concretamente una arquitectura de subsunción, la cual organiza los distintos comportamientos del robot en niveles jerárquicos según su prioridad. Este enfoque permite una toma de decisiones reactiva y robusta en tiempo real, en función del entorno inmediato. En segundo lugar, se implementará una arquitectura híbrida, que combina el enfoque reactivo anterior con componentes basados en el conocimiento y el uso de mapas. Esta extensión permitirá al robot ejecutar las tareas de forma más eficiente, planificada y deliberada.

El entorno simulado está compuesto por una arena que representa un edificio en llamas y que incluye distintos elementos: luces rojas que simbolizan incendios, luces azules que actúan como estaciones de recarga de agua, luces amarillas que representan iluminación de emergencia, zonas grises que indican la presencia de personas atrapadas, y zonas negras que representan las áreas seguras donde deben ser llevadas.

A lo largo de este documento se describen los pasos seguidos para diseñar la arena, configurar los sensores, definir los niveles de competencia y desarrollar la lógica de control del robot. Asimismo, se presentan los resultados obtenidos, los problemas encontrados durante la implementación y posibles líneas de mejora para futuros desarrollos.

ARENA

El primer paso que lleve a cabo fue el diseño de la arena, incluyendo los elementos que se utilizarán en la tarea. La arena consta de una dimensión de 3x3 metros y representa un espacio cerrado compuesto por pasillos que simulan el interior de un edificio. Su configuración se programa en el archivo de experimento, donde establecí la distribución de las paredes, tal y como se muestra en la siguiente figura:

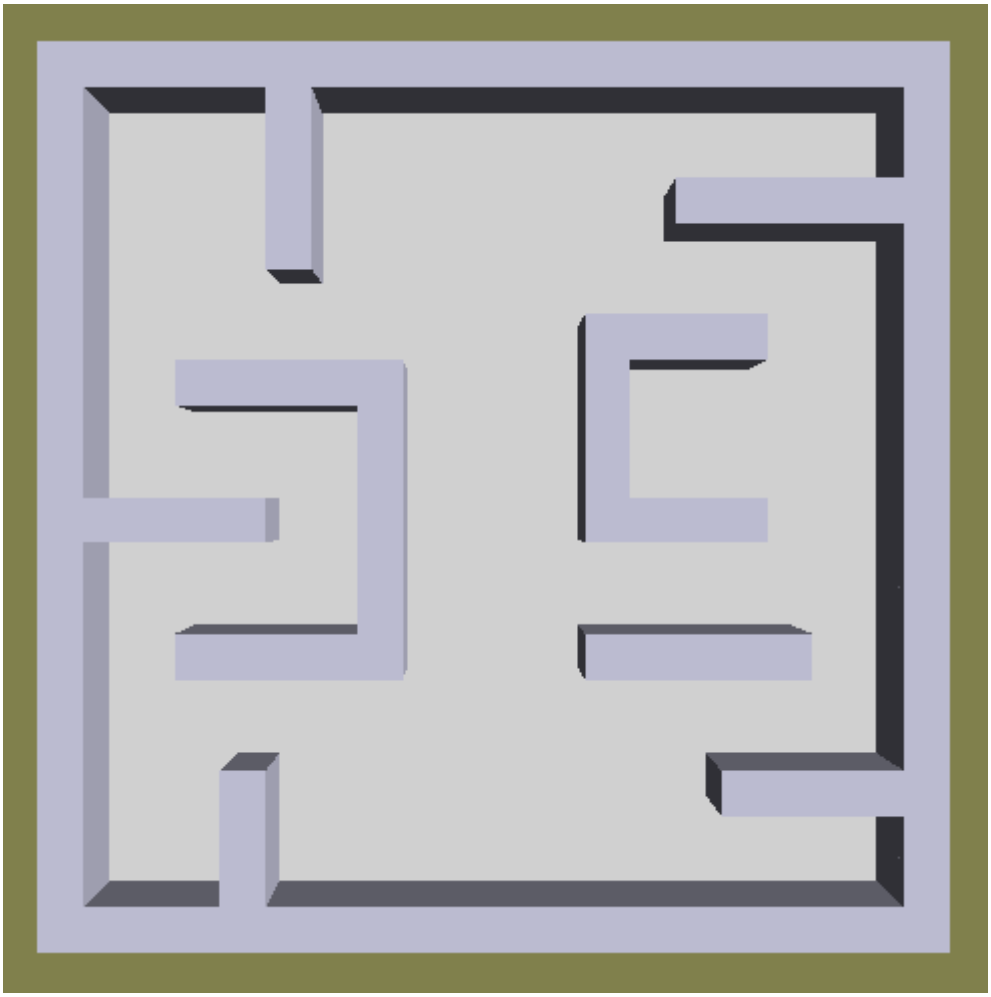

Figura 1: Distribución de la arena.

Aparte de los obstáculos, la arena incluye los siguientes elementos:

- **Luces rojas:** Representan incendios que el robot debe localizar y apagar.
- **Zonas grises:** Simulan áreas donde se encuentran personas atrapadas en el incendio.

- **Zona negra:** Representa una zona segura donde el robot debe depositar a las personas rescatadas.
- **Luz azul:** Indica una estación de recarga para rellenar el depósito de agua del robot.
- **Luz amarilla:** Simula una luz de emergencia. Esta luz estará situada en la misma posición que la zona negra.

Distribuí todos estos elementos en el centro de las casillas de una cuadrícula de 20x20 divisiones. Lo hice así para evitar posibles problemas con casillas conflictivas al implementar la arquitectura híbrida. También situé el robot en el centro de una casilla, con orientación 0, para evitar desviaciones no deseadas, ya que el centro de la arena no coincide con el centro exacto de una celda.

A continuación, muestro un gráfico con las coordenadas de los centros de las casillas libres, que utilicé como referencia para la colocación:

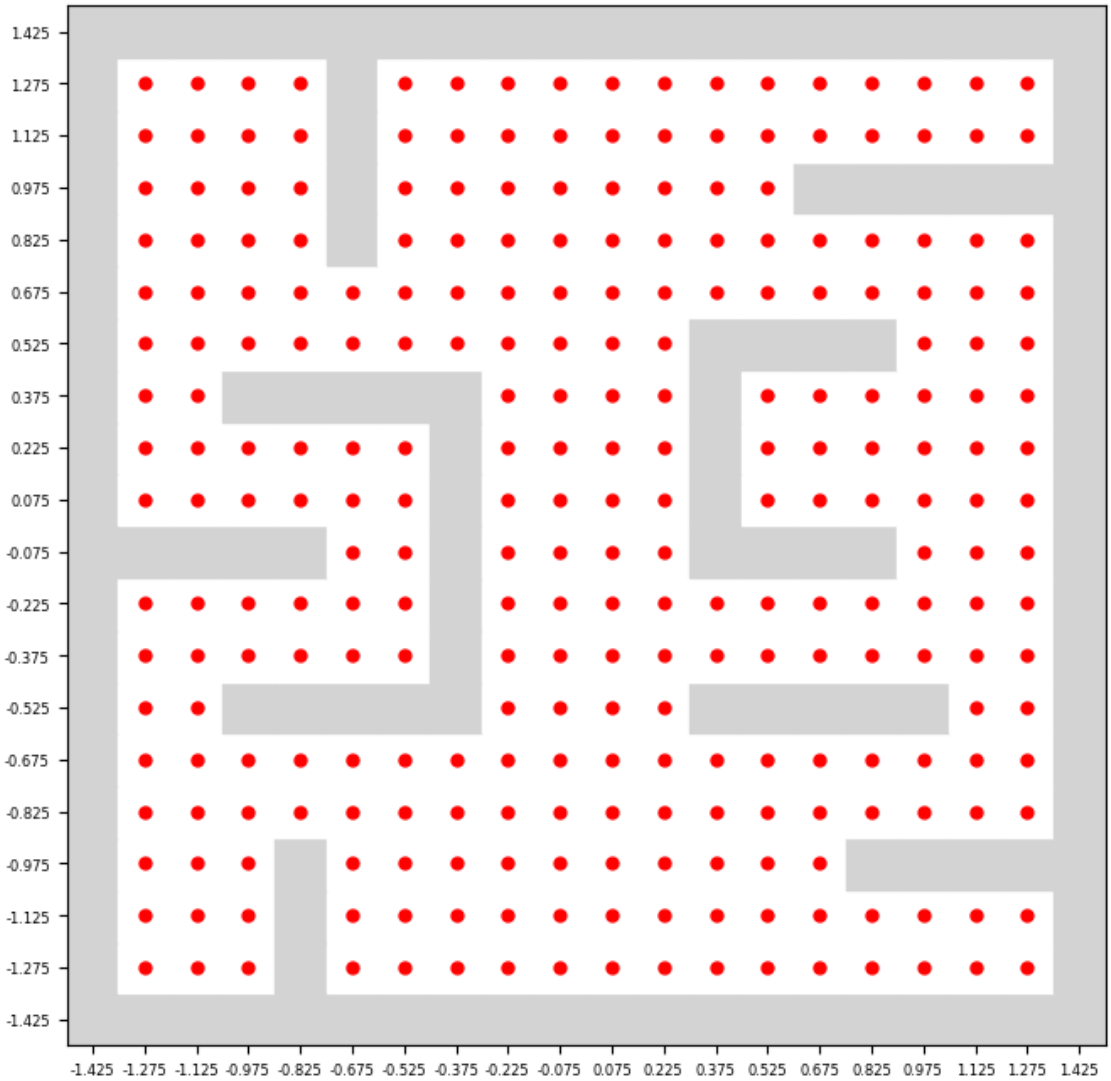

Figura 2: Grafico de las coordenadas de los centros de las casillas libre.

La ubicación final de los elementos y del robot se define en el archivo de configuración de parámetros, generando la siguiente disposición:

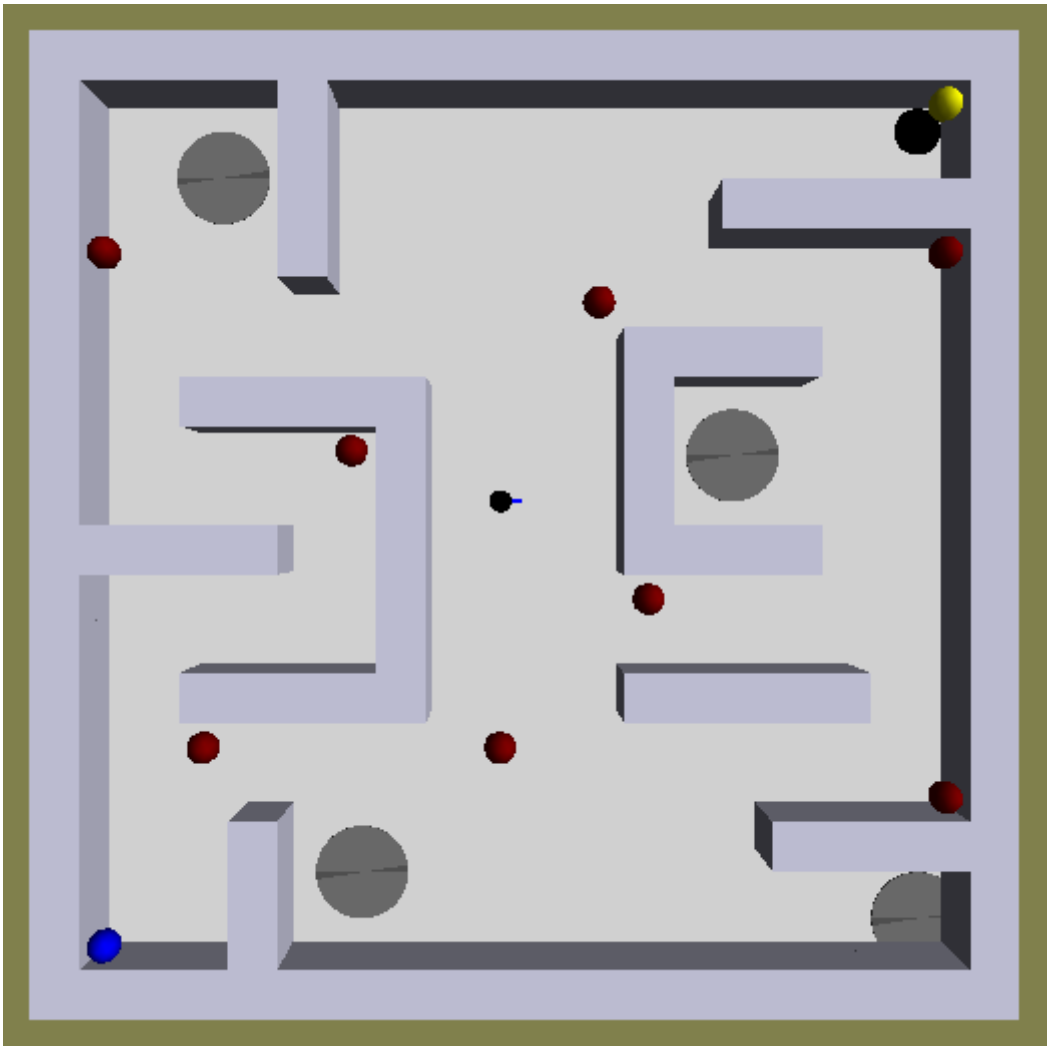

Figura 3: Distribución de los elementos y del robot en la arena.

Como configuración adicional, fue necesario modificar el archivo del objeto redlightobject.cpp para desactivar la secuencia automática de encendido y apagado definida en el método GetTiming. Esta funcionalidad, incluida en la versión del simulador disponible en la web, encendía y apagaba cíclicamente las luces rojas cada 110 Simulation Steps, lo que no resultaba adecuado para el comportamiento deseado del robot.

SENSORES

Una vez diseñada la arena, el siguiente paso consistió en la definición y configuración de los sensores necesarios para el comportamiento del robot. A continuación, se detallan los sensores utilizados, junto con su funcionalidad principal:

- **Sensor de proximidad:** Permite detectar y evitar obstáculos en el entorno.
- **Sensor de suelo:** Detecta las zonas grises (presencia de personas atrapadas) y zonas negras (zonas seguras).
- **Sensor de suelo con memoria:** Identifica si el robot está transportando una persona.
- **Sensor de batería:** Representa el nivel de agua disponible. Se utiliza para determinar cuándo el robot necesita repostar en una estación de recarga.
- **Sensor de luz roja:** Detecta la presencia de incendios, que el robot podrá apagar si dispone de suficiente batería (agua).
- **Sensor de luz azul:** Localiza la estación de recarga de agua.
- **Sensor de luz amarilla:** Detecta la ubicación de la zona segura, a donde deben ser trasladadas las personas rescatadas.

La configuración de estos sensores se realiza en el archivo de parámetros, donde también se definió la distribución de los elementos de la arena. Especifiqué los siguientes valores de configuración:

- **Sensor de luz azul y amarilla**: Rango de detección de 4.5 metros, suficiente para localizar las luces desde cualquier punto de la arena.
- **Sensor de luz roja**: Rango reducido de 0.225 metros, para simular la necesidad de estar muy cerca del fuego para apagarlo.
- **Sensor de batería azul**:
    - Rango: 0.15 metros (equivalente al ancho de una casilla).
    - Coeficiente de carga: 0.25.
    - Coeficiente de descarga: 0.0 (evita que se consuma agua con el movimiento).
- **Sensor de batería amarilla**: También tuve que configurar sus valores, aunque no se use en este experimento, ya que en caso contrario el simulador no interpreta correctamente los parámetros del sensor de batería azul.

  Además de configurar estos sensores en los archivos de parámetros, es necesario declararlos en el controlador del robot para poder utilizarlos en la programación de los comportamientos. No obstante, no fue necesario redefinirlos en los archivos del experimento, ya que utilicé como base los

archivos predefinidos del experimento iri1, los cuales ya incluyen todos los sensores y actuadores necesarios.

ARQUITECTURA SUBSUNCIÓN

He optado por una arquitectura de subsunción, ya que considero que, siendo funcional, es la arquitectura basada en el comportamiento más sencilla para diseñar la arquitectura del robot. Esta arquitectura permite organizar los distintos comportamientos del robot en niveles jerárquicos, priorizando aquellos que requieren una respuesta más inmediata.

La arquitectura está compuesta por los siguientes comportamientos, organizados de menor a mayor nivel de competencia:

- **Evitar obstáculos**: Si el robot detecta un obstáculo cercano, girará para sortearlo.
- **Rescate**: Si el robot pasa sobre una zona gris, recogerá a la persona atrapada, "borrará" la zona gris y se dirigirá a la zona negra para dejarla. Para alcanzarla hará uso del sensor de luz de forma que se orientará hacia la luz amarilla de emergencia, ubicada sobre la zona negra. Mientras el robot se encuentre en modo rescate la arquitectura inhibirá las entradas de recargar agua y apagar fuego.
- **Recargar agua**: En este escenario, la batería del robot no representa su energía, sino su capacidad de agua. No disminuirá con el movimiento, sino únicamente al apagar un fuego. Cada fuego consumirá la mitad del total de la batería. Si el robot se queda sin agua suficiente para apagar un fuego, se dirigirá a la luz azul para recargarla antes de continuar.
- **Apagar fuego**: Si el robot detecta una luz roja cercana y tiene suficiente batería (agua), apagará el fuego y reducirá 50% de la carga de su batería.
- **Navegación**: Si el robot no está afectado por ningún otro comportamiento de mayor prioridad, avanzará en línea recta.

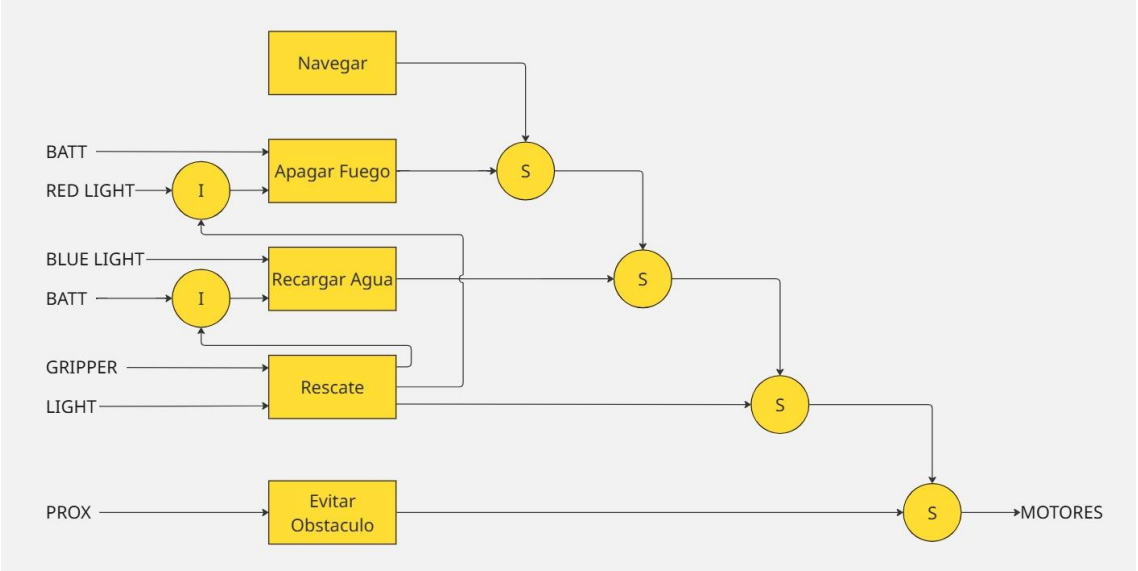

Figura 4: Arquitectura subsunción del robot bombero

Coordinador de comportamientos

Dado que se trabaja en un entorno secuencial, los niveles de competencia no se ejecutan en paralelo. Por ello, es necesario implementar un coordinador que evalúe los resultados de cada comportamiento e inhiba los de menor prioridad cuando sea necesario.

Este coordinador utiliza una tabla de activación en la que cada fila corresponde a un nivel de competencia. En ella se almacena:

- Las velocidades propuestas para las ruedas izquierda y derecha.
- Un flag que indica si ese comportamiento debe activarse y suprimir los inferiores.

	Left Speed	*Right Speed*	*Flag*
ObstacleAvoidance	V_l^0	V_r^0	T/F
Rescue	V_l^1	V_r^1	T/F
LoadWater	V_l^2	V_r^2	T/F
FightFire	V_l^3	V_r^3	T/F
Navigate	V_l^4	V_r^4	T/F

Figura 5: Tabla del coordinador del experimento Robot Bombero

Al finalizar la ejecución de todos los comportamientos, el coordinador recorre la tabla desde el nivel más alto de prioridad (ObstacleAvoidance) hasta el más bajo (Navigate). En cuanto encuentra una tarea activada (flag a TRUE), aplica las velocidades asociadas a los actuadores del robot.

En los siguientes apartados se describe en detalle el funcionamiento de cada nivel de competencia y las condiciones que deben cumplirse para su activación.

ObstacleAvoidance

Este nivel de competencia es el encargado de hacer que el robot evite obstáculos. El nivel de competencia comprueba si hay un obstáculo delante del robot. En caso afirmativo, suprime la señal que viene de los otros niveles de competencia, calcula un vector en dirección opuesta al obstáculo que será enviado a los motores. El nivel de competencia lee los sensores de proximidad, calcula el máximo y genera un vector opuesto a la dirección del obstáculo. Si el máximo de los sensores se encuentra por encima de un umbral (PROXIMITY THRESHOLD), se considera que existe un obstáculo. Además, se pone el robot en color verde y se calcula una velocidad lineal y angular en función del vector obtenido anteriormente. A partir de esos valores se calcula las velocidades de la rueda derecha e izquierda y se activa el flag del nivel de competencia.

Rescue

Este nivel de competencia se encarga de detectar y rescatar personas atrapadas, representadas por las zonas grises, y transportarlas hasta la zona segura, que corresponde a una zona negra en la arena. Para ello, utilizo el sensor GroundMemory, que me informa si el robot ha pasado por una zona gris: en ese caso, su valor pasa a ser 1.0, lo que activa el modo rescate.

Una vez en modo rescate, el robot se orienta hacia la luz amarilla, situada sobre la zona negra, utilizando los valores proporcionados por los sensores de luz. Si los sensores izquierdo (L0) y derecho (L7) están activados simultáneamente, significa que el robot ya está bien orientado hacia la luz, y por tanto se sale de la ejecución del comportamiento sin activar el flag, ya que no necesita modificar su movimiento actual.

En caso contrario, el comportamiento se activa y el robot calcula la suma de los valores de los sensores de luz de su lado izquierdo (L0 a L3) y de su lado derecho (L4 a L7). Si la luz se encuentra a la izquierda, el robot gira hacia la izquierda; si está a la derecha, gira hacia la derecha. Además, se encienden los LEDs en color amarillo para indicar visualmente que está en proceso de rescate, y se inhiben los comportamientos de recarga de agua (LoadWater) y extinción de incendios (FightFire) para que no interfieran en la tarea actual.

Una vez que el robot llega a la zona negra, el comportamiento deja de ejecutarse.

Para reflejar que la persona ha sido rescatada, implementé un método llamado SwitchNearestGroundArea en arena.cpp, inspirado en el método SwitchNearestLight. Este método busca el objeto GroundArea más cercano a la

posición actual del robot y, si su color es gris (valor 0.5), lo reemplaza por blanco (1.0) utilizando SetColor. Además, creé una función homónima en el sensor de suelo (GroundSensor) que guarda la posición del robot e invoca el método de la arena. En el controlador, llamo a esta función justo cuando el robot pisa una zona gris:

```
m_seGround->SwitchNearestGroundArea(1.0);
```

De este modo, la zona gris queda "borrada" una vez que ha sido rescatada la persona correspondiente.

LoadWater

Este comportamiento gestiona la recarga de agua del robot, que está representada por el valor del sensor de batería azul. Esta batería no disminuye con el movimiento, sino únicamente al apagar incendios. Si el valor cae por debajo de 0.499999 (no aproximamo a 0.5 por problemas de precisión) y el comportamiento Rescue no está activo, se activa este nivel.

Cuando se activa:

- El robot se ilumina en color azul.
- Se orienta hacia la luz azul, que representa la estación de recarga, utilizando el mismo esquema de orientación descrito en Rescue.
- Una vez alcanzada la estación, la batería se recarga completamente.

FightFire

Este nivel de competencia tiene como objetivo apagar fuegos, representados por las luces rojas. Para activarse, deben cumplirse tres condiciones:

1. La suma de las lecturas de los sensores de luz roja debe superar el umbral FIGHT_FIRE_THRESHOLD.
2. El nivel de batería debe ser mayor que BATTERY_THRESHOLD.
3. El comportamiento Rescue no debe estar activo.

Si se cumplen estas condiciones:

- El robot se ilumina en color rojo.
- Se apaga la luz roja más cercana mediante SwitchNearestLight(0).
- Se reduce la batería en 0.5, simulando el consumo de agua.

La gestión del consumo se realiza mediante el método SetBatteryLevel, implementado en el controlador del sensor de batería azul. Este método asigna directamente el nuevo nivel de batería:

```
newBattery = battery[0] - 0.5;

m_seBlueBattery->SetBatteryLevel(newBattery);
```

Navigate

Este comportamiento es el de menor prioridad. Funciona de forma que, si ningún otro comportamiento se activa, el robot avanzará en línea recta a una velocidad SPEED, definida como 500.

REGISTRO DE RESULTADOS

Para facilitar el análisis posterior del experimento, implementé varios mecanismos de registro mediante escritura en ficheros. En cada instante de muestreo se almacenan los siguientes datos:

- **Posición, orientación y tiempo** en outputFiles/robotPosition, desde el método SimulationStep.
- **Velocidad de las ruedas** en outputFiles/robotWheels, también desde SimulationStep.
- **Nivel de competencia activo** en outputFiles/coordinatorOutput, desde el método Coordinator.

Adicionalmente, cada tarea imprime sus valores sensoriales, si se activa o no la tarea y las velocidades propuestas para las ruedas en su propio fichero:

- avoidOutput
- rescueOutput
- waterOutput
- fireOutput
- navigateOutput

Estos archivos permiten un análisis detallado y visual del funcionamiento del robot a lo largo del experimento.

Duración del experimento

El primer dato que analicé fue la duración total del experimento, es decir, el tiempo que tarda el robot en rescatar a todas las personas y apagar todos los fuegos. Esta información puede obtenerse desde cualquiera de los ficheros, ya que todos incluyen la marca temporal. La duración total fue de 2117,4 segundos, es decir, aproximadamente 35,29 minutos de tiempo de simulación. Es importante aclarar que este tiempo no equivale al tiempo real de ejecución.

Comportamientos activos

A partir del fichero coordinatorOutput, generé el siguiente diagrama temporal de los comportamientos activos, que permite observar la secuencia y duración de cada tarea desde el inicio hasta la finalización del experimento:

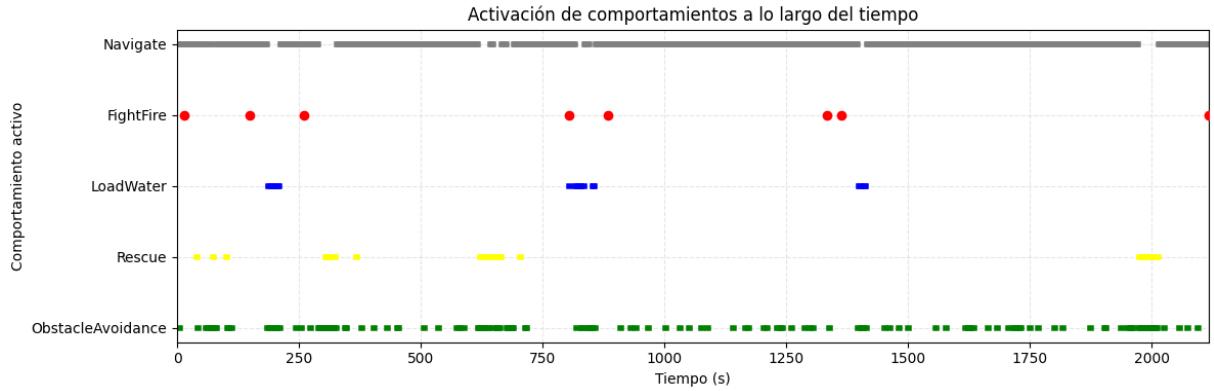

Figura 6: Línea de tiempo de los comportamientos activos

Del análisis de este gráfico se extraen varias conclusiones relevantes:

1. El robot rescata primero a todas las personas antes de apagar completamente los incendios.
2. Detecté un comportamiento anómalo: una de las luces rojas se apagaba sin que se registre la activación del comportamiento FightFire. Tras analizar el problema, caí en la conclusión de que esto ocurría debido a que, al apagarse la luz, el robot estaba colisionando contra una pared, activando así el comportamiento ObstacleAvoidance que suprimía al resto, impidiendo que se registrara correctamente FightFire. Aunque esto no debería ocurrir, porque por código se activa el flag antes de apagar la luz, es la única explicación que veo lógica. La única solución que encontré, sin tener que modificar el código, fue mover la luz.
3. Otro comportamiento interesante se observa en la interacción entre la recarga de agua y el apagado de fuegos. Aunque cada fuego consume la

mitad del depósito, el gráfico muestra tres activaciones de FightFire sin una activación intermedia de LoadWater. Esto ocurre porque el robot recarga automáticamente al pasar por la estación sin que el comportamiento LoadWater se active (ya que el nivel de batería no llegó a estar por debajo del umbral). Esta situación se confirma con los datos del nivel de batería en el fichero waterOutput, que dio lugar al siguiente gráfico:

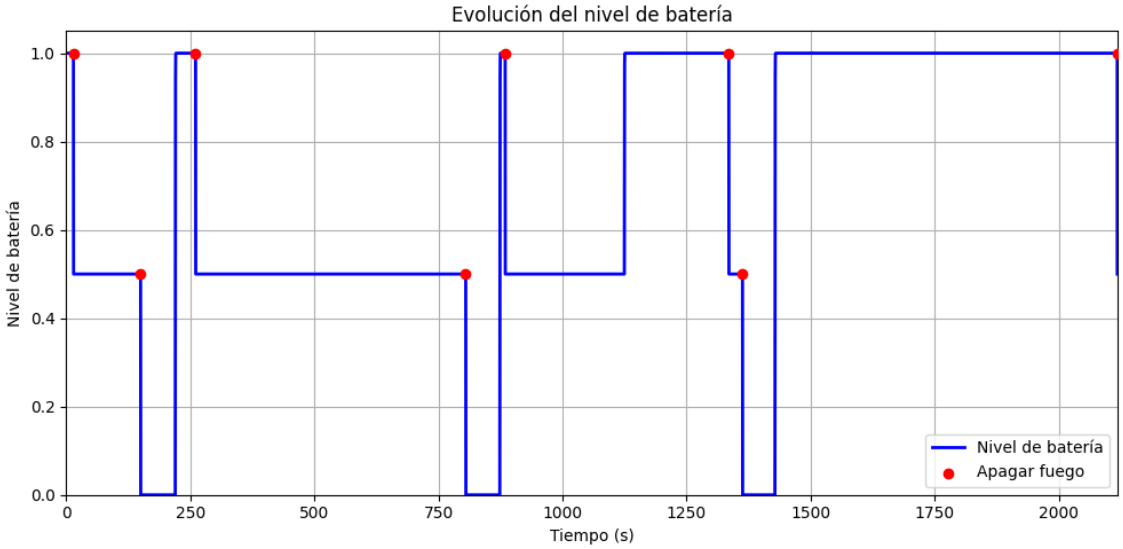

Figura 7: Línea de tiempo de la evolución del nivel de batería

Se observa claramente cómo ocurren recargas sin agotar completamente la batería, lo que explica la ausencia de activación explícita del comportamiento LoadWater en algunos casos.

Trayectoria del robot

Otro resultado relevante es la trayectoria seguida por el robot en la arena, representada a partir de los datos del fichero robotPosition.

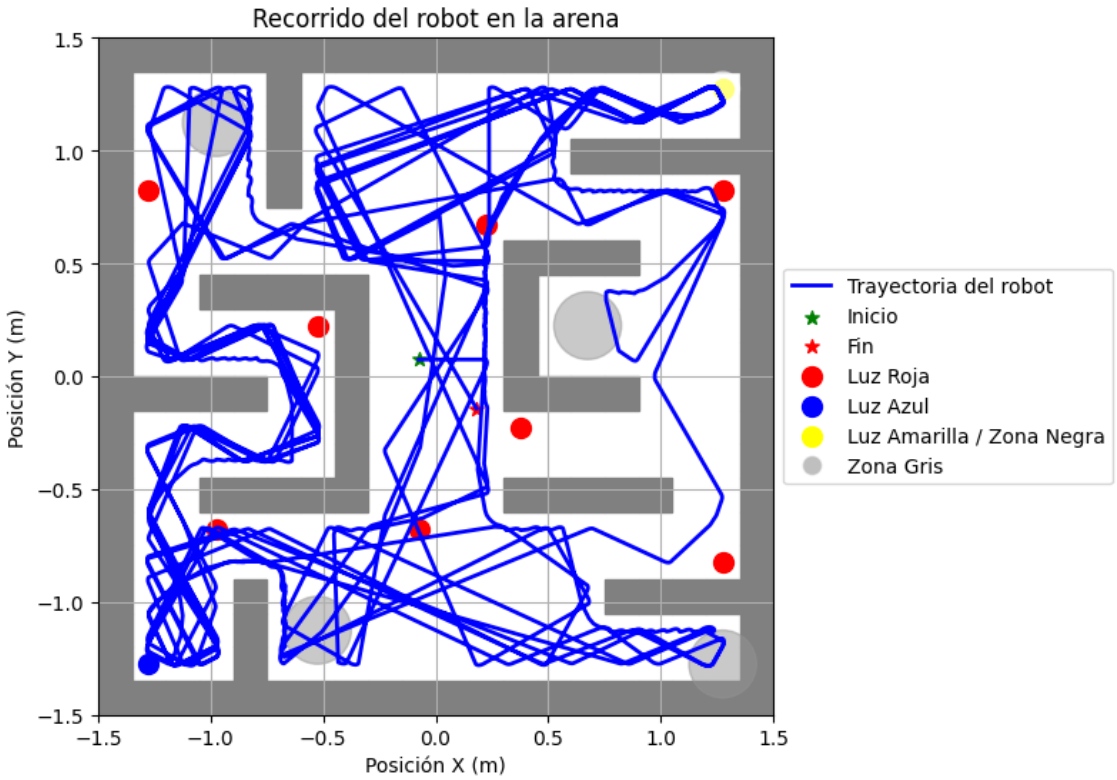

Figura 8: Trayectoria del robot por la arena

En el gráfico se aprecian zonas más transitadas que otras. En especial, la parte izquierda de la arena muestra una mayor densidad de trayectorias, en contraste con la parte derecha, que en algunas zonas ni siquiera llegan a pisarse. Esta diferencia se explica por la distribución de los muros y, en particular, por la posición de la estación de recarga, colocada en la esquina inferior izquierda. Esto provoca que el robot transite repetidamente por ese lateral. Este comportamiento es también observable con el siguiente mapa de calor.

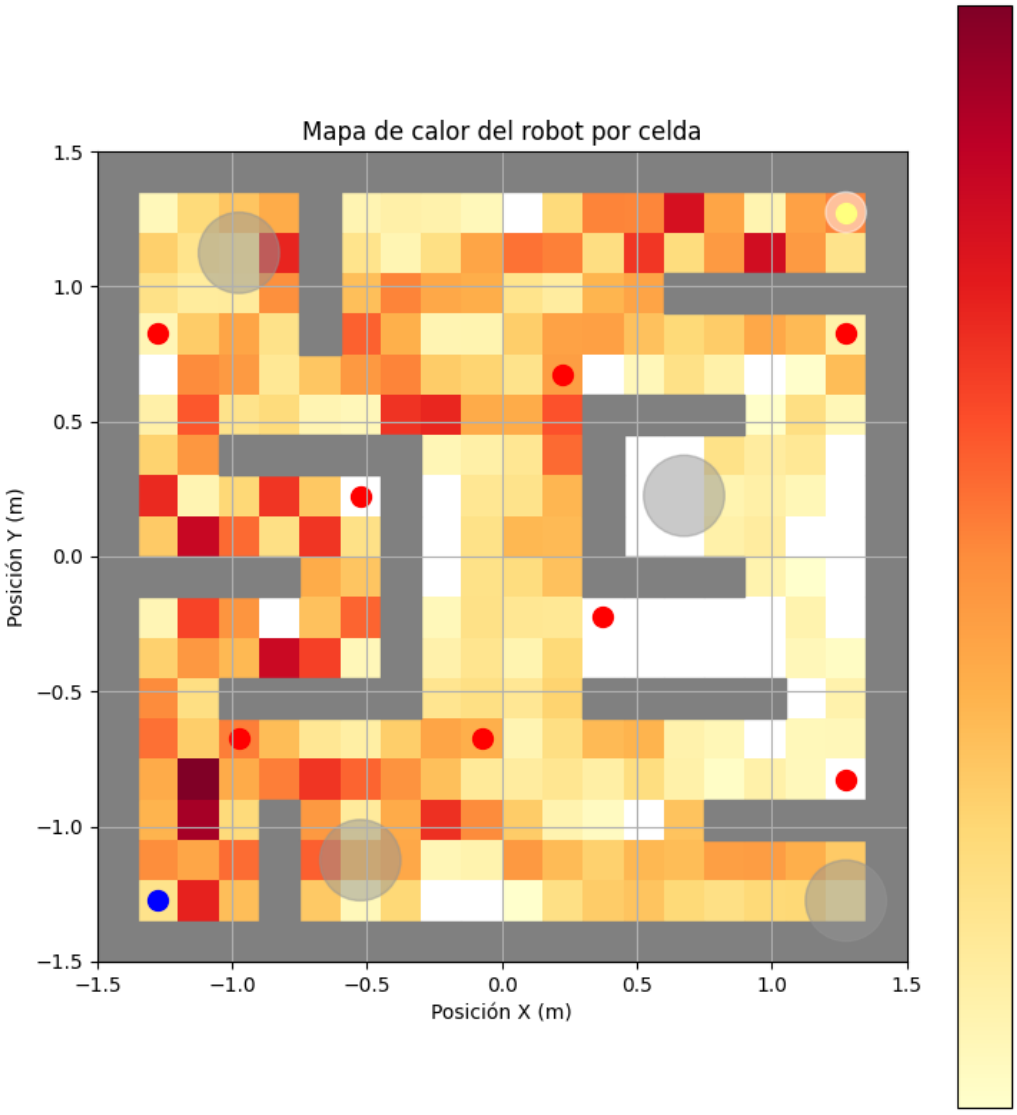

Figura 9: Mapa de calor por celdas

La distribución de los muros fue el aspecto que más veces tuve que modificar. Comencé con una disposición más compleja, con pasillos estrechos y zonas tipo "habitaciones" con accesos reducidos. Sin embargo, durante la implementación, fui simplificando progresivamente la estructura. Esto se debió a que el robot se quedaba atascado en "zonas muertas" al intentar acceder a las luces amarilla y azul. Además, el robot repetía patrones de trayectoria que eran innecesarios. Para evitar estas situaciones, probé distintas configuraciones de muros y fui ajustando el umbral de proximidad, hasta obtener una distribución funcional, que es la que se presenta en esta memoria.

Otra posible mejora que valoré durante el desarrollo fue modificar la forma en la que el robot se aproxima a las luces, incorporando información sobre la presencia de obstáculos. Actualmente, la lógica se basa únicamente en si el robot está

orientado hacia la luz, girando en consecuencia. Sin embargo, este enfoque puede provocar que el robot quede bloqueado si hay un obstáculo entre él y la fuente de luz.

Por lo que, una alternativa más robusta sería implementar una aproximación más inteligente que combine la dirección de la luz con el análisis de obstáculos, permitiendo al robot rodearlos si es necesario. No obstante, consideré que esta solución requería una lógica considerablemente más compleja. Por tanto, opté por simplificar el diseño de la arena para evitar dichas situaciones, manteniendo el equilibrio entre funcionalidad y complejidad del código.

Tabla resumen

Para finalizar el análisis de resultados, he compilado un resumen estadístico en la siguiente tabla.

Métrica	***Valor***
Tiempo total (simulador)	2117.4 s
Número de fuegos apagados	8
Personas rescatadas	4
Activaciones de FightFire	8
Activaciones de LoadWater	318
Activaciones de Rescue	618
Activaciones de AvoidObstacle	3349
Activaciones de Navigate	16882

Figura 10: Tabla resumen resultados

ARQUITECTURA HIBRIDA

El problema de alcanzar zonas específicas del mapa de manera ineficiente y de hacer repeticiones periódicas de trayectorias innecesarias, es fácilmente solucionable si implementamos una arquitectura hibrida. Para ello, añadiría a la arquitectura basada en el comportamiento una capa deliberativa, es decir, una

arquitectura basada en el conocimiento que combinase el enfoque reactivo de los comportamientos con un conocimiento progresivo del entorno.

La idea se basa en la arquitectura híbrida presentada en clase, aunque adaptada a las particularidades de este experimento. En primer lugar, dado que las zonas grises (que representan personas atrapadas) desaparecen una vez que han sido rescatadas, no tendría sentido almacenarlas en el mapa como puntos de interés permanentes. Por el contrario, la zona negra (la zona segura donde se dejan a las personas) sí debe mantenerse registrada, ya que el robot debe volver a ella tras cada rescate.

Otro punto de interés clave es la estación de recarga de agua (luz azul). Debería almacenar también su posición en el mapa, ya que esto permitiría al robot dirigirse a recargar de forma eficiente cuando su nivel de batería lo requiera.

Además, como los elementos del entorno están colocados en el centro de cada casilla de una cuadrícula 20x20, considero que sería eficiente implementar una estrategia de exploración sistemática que recorra todas las casillas, asegurando que se rescaten todas las personas y se apaguen todos los fuegos. Una opción que he valorado es que el robot explore la arena de arriba abajo, casilla por casilla, registrando en el mapa los elementos encontrados.

En el caso de las luces rojas, también sería necesario aplicar una lógica específica en su registro. Si el robot detecta una luz roja y no puede apagarla, ya sea por falta de batería o porque está transportando a una persona y el comportamiento *Rescue* inhibe la acción, se debería marcar esa casilla como pendiente de visitar. En cambio, si el robot logra apagar la luz, esa casilla podría considerarse libre.

Estoy convencido de que esta implementación permitiría ejecutar las tareas de forma mucho más eficiente, reduciendo considerablemente el tiempo necesario para completar el experimento. Quería haber desarrollado esta parte, pero por falta de tiempo no he podido implementarla. Aun así, considero que representa una evolución natural del trabajo y una posible línea de mejora para futuras versiones del proyecto.

ARCHIVOS MODIFICADOS

1. Iir1exp.cpp
2. Iri1exp.h
3. O1param.txt
4. Iri1controller.cpp
5. Iri1controller.h

6. Bluebatterysensor.cpp
7. Bluebatterysensor.cpp
8. Arena.cpp
9. Arena.h
10. Groundsensor.cpp
11. Groundsensor.h
12. Redlightobject.cpp

BIBLIOGRAFÍA

Material de Introducción a la Robótica Inteligente - https://www.robolabo.etsit.upm.es/subjects.php?subj=irin&tab=tab1