RoboCupJuniorRescue(Line/Maze)2024

Team Description Paper

# ATLAS-7810



# Abstract

● El robot del equipo Atlas-7810 ha pasado por diversas modificaciones durante el proceso de preparación para la competencia RoboCup Junior Rescue 2024. Este robot, en la fase de simulación, tiene la tarea de recorrer el mapa en busca de víctimas, detectándolas y clasificándolas adecuadamente. Al concluir el recorrido inicial, antes de regresar al punto de partida, ejecutará un segundo recorrido con el objetivo de verificar la detección completa de todas las víctimas. Además de estas capacidades, el robot desempeña la función crucial de mapeo. Mientras realiza las tareas mencionadas, lleva a cabo un seguimiento detallado del entorno, generando un mapa que será devuelto en forma de matriz. Este mapa incluirá las ubicaciones precisas de las víctimas detectadas, así como su clasificación correspondiente. Este doble recorrido y la precisión en la detección y mapeo son fundamentales para garantizar la eficiencia y efectividad del robot en situaciones de rescate. Gracias a estas modificaciones hechas, ninguna víctima quedará sin detectar y el mapeo del área será lo más preciso posible.

# 1. Introduction a. Team

* Brief description of each team member’s roles, their past experiences, and what they contributed to the team.
* Suggestion of 20 - 100 words per team member description.

# 2. Project Planning

# a. Overall Project Plan

**Objetivo:** Siendo el primer intento del equipo Atlas en la competencia, nuestro objetivo es superar los logros obtenidos en la por nuestros compañeros de la institución en la anterior edición de la competencia: Nuestro objetivo era mantener la navegación eficiente ya establecida, pero centrar nuestros esfuerzos en mejorar el mapeo y detección de victimas para aumentar la cantidad de puntos obtenidos**.**

Fig. 1: Simulación en un entorno de prueba

Queremos comunicarnos con otros equipos y conocer como ellos desarrollaron sus programas, no solo para aprender más y mejorar nuestro programa a futuro, sino para disfrutar la oportunidad única de cooperación y diversión.

**Trabajos previos:** Durante los años 2022 y 2023 otros compañeros de nuestra institución fueron trabajando sus programas para sus respectivas competencias. Realizaron mejoras en la navegación y en el mapeo, al igual que implementar nuevas estrategias de detección y eficiencia a la hora de recorrer el mapa. De esta manera lograr la obtención de buenos resultados.

**Plan general (2024):** Para la robocup 2024 se utilizó como base el programa anteriormente desarrollado y para eso se realizó un análisis previo de aquellos ámbitos o inconvenientes que el programa anterior no estaba abarcando. Por lo que se estableció como prioridad la maximización de la puntuación, es decir obtener el mejor resultado posible.

Teniendo en cuenta que aún se continúa desarrollando el código los hitos acontecidos fueron:

**Hito#1:** Teniendo en cuenta que ningún participante previo quedo en el equipo, se decidió realizar una capacitación general del funcionamiento del programa previo, en donde cada uno de los participantes fue comprendiendo la organización general de cada parte del programa

**Hito#2:** Una vez el programa fue entendido por todas las partes participantes, se realizó la identificación de aquellas áreas que se podía mejorar, logrando visualizar que el programa tenia ciertos problemas en la detección de la victimas y carteles, y en la elaboración de la rejilla de bonificación.

Se decidió investigar mas a fondo que estaba pasando con la rejilla de bonificación, y se determino que el programa no estaba agregando las victimas y carteles, que en algunas situaciones no mapeaba bien la pared, y que se estaba entregándolo con una extensión mas grande de lo que correspondía.

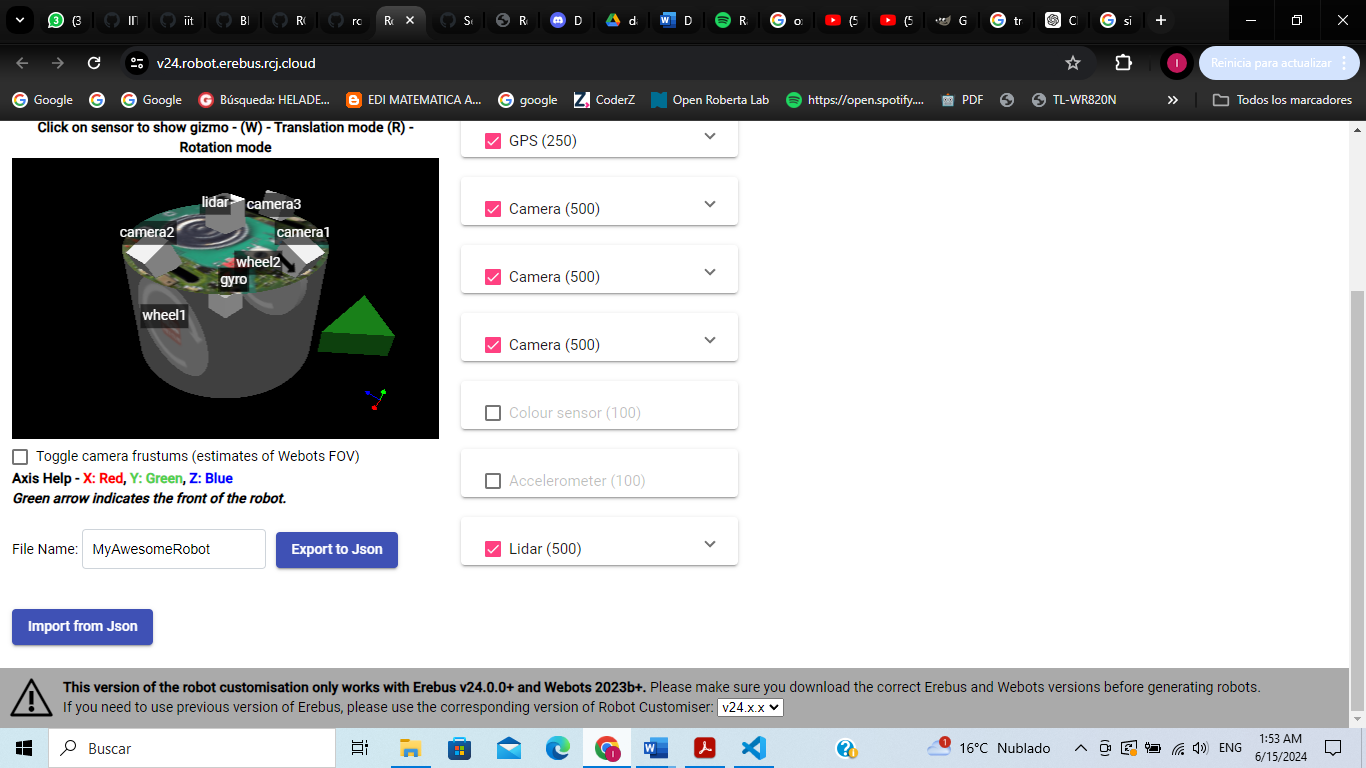
Se logro corregir la extensión de la rejilla y corregir las paredes, pero no agregar las victimas

**Hito#3:** Con el nuevo reglamento, el robot ya no poseía la máxima resolución de pixeles de las cámaras a costo 0, y debido al diseño de nuestro robot tuvimos que conformarnos con la resolución mínima. Esto provoco que apareciera una nueva gran problemática. La modificación de la resolución de las cámaras ocasiono que el programa no solo no pudiera detectar víctimas, sino que también afecto a la correcta navegación. Se plantearon numerosas estrategias para arreglar este problema, como por ejemplo eliminar la cámara frontal (que nos obligaría a cambiar toda la navegación a fondo). Sin embargo, para suerte de nosotros, con la modificación de algunas variables pudimos readaptar el código.

Luego se avanzó con la mejora de la detección de victimas mediante el estudio del sistema YOLO (JOACO AGREGA REFERCIA ACA DE YOLO), pero aun no se implemento al programa final. Por último, después de varios intentos mediante diferentes métodos, se logró añadir las victimas a la rejilla de bonificación, aunque se va buscar mejorar la eficiencia de este sistema.

- Mediante el análisis de datos se determino que se logro una mejora en los puntos obtenidos por la ejecución del programa. Sin embargo, creemos que aún se puede mejorar mucho mas el rendimiento si se siguen desarrollando los cambios esperados para el programa

## 3. Robot Design

El diseño busca permitir cumplir cada tarea de la manera más eficiente posible, teniendo en cuenta el presupuesto en la herramienta de personalización del robot. Para evitar que el ruido generado por el entorno simulado afecte la funcionalidad del robot, colocamos los sensores de manera simétrica (ver Fig. 3). De esta forma, la interferencia es la misma en cada lado y no causa ningún mal funcionamiento.

#### **Componentes:**

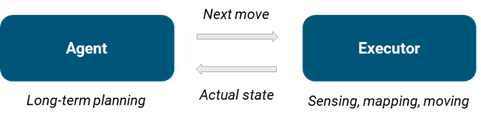
* **Motores (x2)**: Permiten el movimiento a lo largo del laberinto, ya que tienen velocidad y dirección independientes. Cada uno conectado a una rueda.

*Fig. 2: Robot design on the customization tool*

* **Cámaras (x3)**: Permiten el reconocimiento visual de estructuras, obstáculos y tipo de suelo. Dos están en los lados y una en la parte frontal, permitiendo un campo de visión de casi 90°. Todos tienen una resolución de Width y de Height de 40 para no pasarse del presupuesto
* **GPS**: obtiene las coordenadas del robot, lo guía a lo largo de la navegación y lo reorienta en caso de pérdida de posición (teleportación). Está ubicado en el centro para obtener mediciones precisas.
* **Giroscopio**: Mide la orientación del robot, funcionando como un detector si no avanza en línea recta. Está ubicado en el centro para obtener mediciones precisas.
* **LiDAR**: Permite crea una nube de puntos que detecta objetos y obstáculos alrededor del robot. No posee problemas para identificar paredes curvas, calcular distancias o muestrear ángulos. Está ubicado en la parte superior del robot, aprovechando al máximo su detección de 360°.

# 4. Software

Nuestro código se encuentra dividió en 2 partes, la parte de Agente y la parte de Ejecutor los cuales trabajan en conjunto. Se opto por este tipo de arquitectura para optimizar la navegación y mejorar la toma de decisiones en entornos complejos

 **Agente (planificador a largo plazo)**: analiza una matriz 2D de nodos para determinar cuál es el camino más eficiente hacia un punto en específico dentro del laberinto (ver sección 3b). Es decir, proporciona instrucciones al Ejecutor.

*Fig. 3: software architecture*

 **Ejecutor (actor a corto plazo)**: Está a cargo del movimiento siguiendo las instrucciones del Agente. Luego procesa los datos recopilados de los sensores para actualizar la matriz del Agente, permitiendo que le proporcione nuevas instrucciones.

# General software architecture

Se decidió separar las acciones del robot en dos grupos: un bucle principal con tareas constantes y una máquina de estados con funciones temporales. Un gestor de secuencias controla este último grupo, permitiendo que se ejecuten sin interrumpir el resto del bucle.

**Bucle:** Contiene las funciones encargadas de la navegación, mapeo y detección del robot. El bucle recopila datos del Agente y ordena al Ejecutor una serie de tareas a realizar por cada acción.

Además, verifica estados de emergencia (Atascado, Fin, o Reportar\_víctima) que podrían hacer que el programa tenga un error. Una vez que detecta cualquier cambio, envía una señal para iniciar la máquina de estados.

**Máquina de estado**s: Existen diferentes estados específicos para cada situación o cambio detectado en el bucle principal. Incluye:

- **Init:** Calibra la posición y rotación del robot, y realiza tareas de inicio (se ejecuta al inicio del programa y después cambia al estado explorar).

**- Explorar:** Envía la cuadrícula LiDAR al Agente y solicita instrucciones (mientras aplica corrección de posición de bajo nivel). Funciona como un ciclo: la cuadrícula se actualiza una vez que se ejecutan las instrucciones.

- **Reportar\_víctima:** Se activa cuando se detecta una víctima, detiene el robot durante un segundo, informa de la víctima detectada al supervisor –cambiando el estado de la víctima en el mapa a reportado– y vuelve a Explorar.

- **Enviar mapa:** Estado en donde se envía al supervisor el mapa obtenido con el mapeo

- **Detener:** Estado de depuración (sin causa aparente), vuelve a Explorar una vez que se resuelve el problema.

- **Atascado**: Estado de depuración por si algo bloquea el robot (pero no las ruedas), vuelve al estado Explorar una vez que el robot logra desatascarse.

- **Teletransportado:** En caso de pérdida de posición se recalibra la posición, y vuelve a Explorar

- **Fin:** Envía un fin de juego después de completar la navegación, mapeo y detección.

# b. Navigation

En la navegación, se utilizan las (clases/programas/otra opción) de Agente y Ejecutor, los cuales son algoritmos que se encargaran de planificar los posibles caminos y de reaccionar con movimientos. La planificación generalmente es más lenta, pero da mejores resultados; el movimiento es generalmente más rápido, pero puede atascarse. (ver [5]).

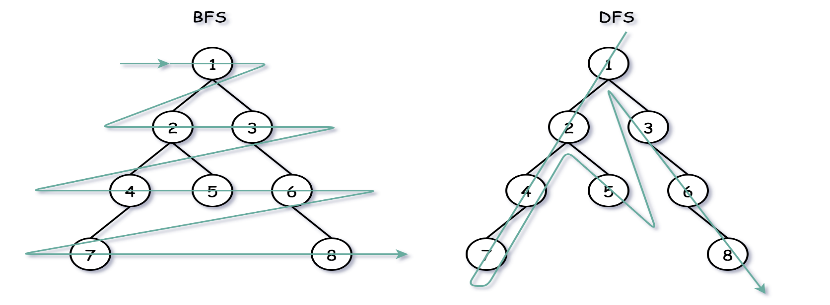
El Agente es el responsable de la búsqueda de caminos: en lugar de probar cada trayectoria, determina cuál es la mejor opción. Por otro lado, el Ejecutor se encarga del movimiento: sigue la dirección elegida por el Agente, enviando a su vez nuevos datos para el siguiente análisis de la nueva posible ruta.

### Búsqueda de Caminos

La búsqueda de caminos se determina de acuerdo a la información obtenida durante el mapeo. Ambos procesos co-ocurren.

Al recibir información de la matriz 2D, el Agente verifica qué nodos son transitables. Para ello, se utiliza un algoritmo de recorrido de árboles (ver [6] y ver [7]).

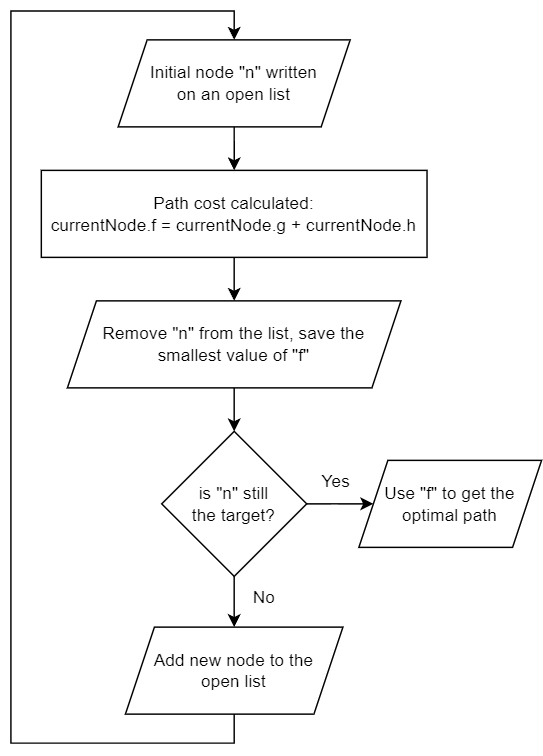
Aunque una búsqueda en profundidad (ver [8]) permite que el robot analice y atraviese la mayoría de los caminos, la exploración rama por rama es innecesaria y más lenta ya que la idea es elegir el camino más factible en el menor tiempo posible. Por lo tanto, se eligió una búsqueda en anchura (ver [9]) para explorar los nodos nivel por nivel, siguiendo directamente hasta el nodo objetivo.



*Fig. 4 & 5: Imagen ilustrativa de cada algoritmo y el orden en el cual se visitan los nodos (source:* ([10])

En cuanto a encontrar el camino más corto, consideramos dos algoritmos diferentes:

**Algoritmo de Dijkstra:** El robot visita cada vértice detectado en la matriz 2D, agrupando todos los nodos no visitados, verificando repetidamente el más cercano que aún no ha sido examinado y agregando las nuevas detecciones al conjunto. De esta manera la cuadrícula se expande hacia afuera: desde el punto de partida hasta alcanzar el objetivo (ver [11]).

Si bien el algoritmo de Dijkstra pareciera ser la mejor opción para encontrar el camino más corto al nodo objetivo, a veces no suele hacerlo con el menor costo ya que únicamente considera la distancia con el nodo inicial y no la existencia de posibles obstáculos. Por lo tanto, se decidió adoptar un enfoque diferente.

**Algoritmo A**\*: El robot mantiene un árbol de caminos que se originan en el nodo inicial y se extienden hasta que uno de ellos alcanza el nodo objetivo. Es un algoritmo más inteligente. Su objetivo es encontrar el camino con el menor costo, es decir el que implica recorrer la menor distancia posible, formulado en gráficos ponderados (ver [12]).

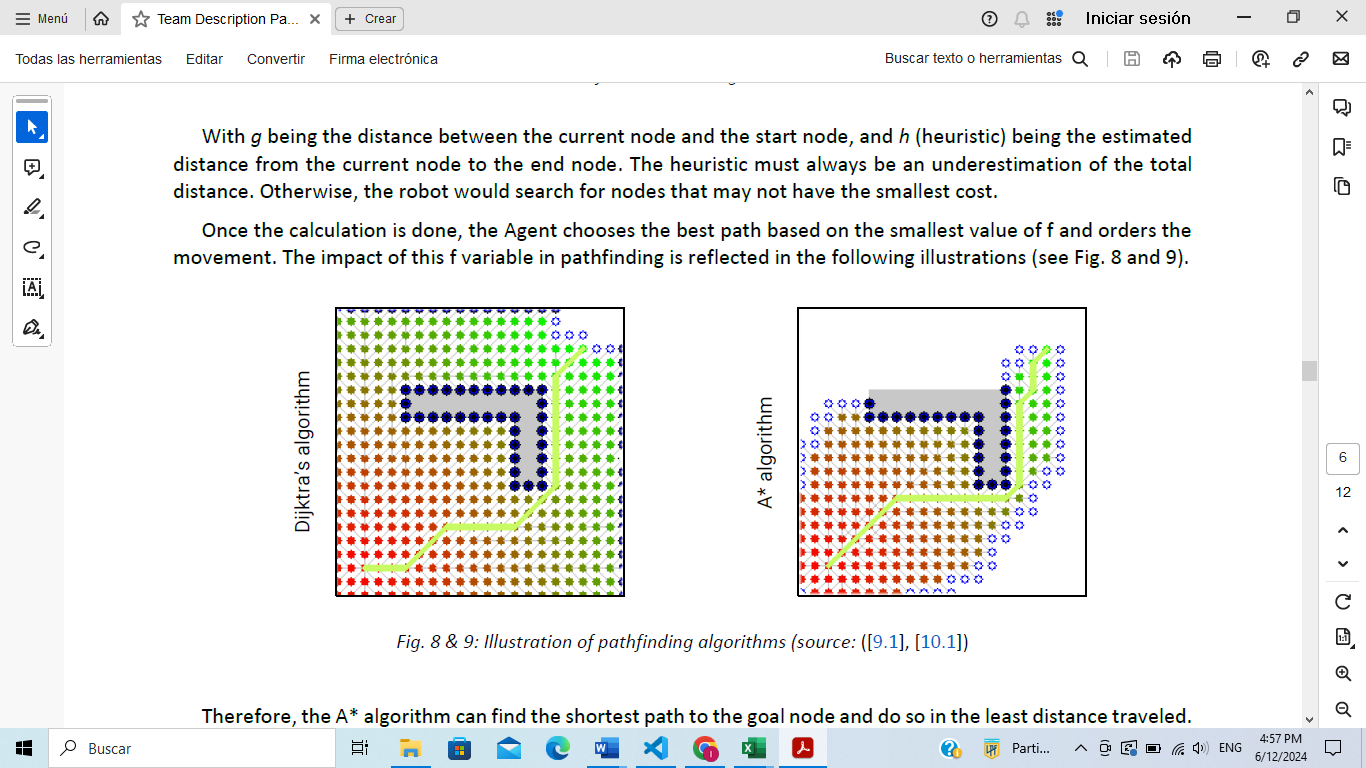
Para lograr esto, el costo total del nodo, f, se calcula a través de la fórmula:

*currentNode.f = currentNode.g + currentNode.h*

Donde g es la distancia entre el nodo actual y el nodo de inicio, y h (heurística) es la distancia estimada desde el nodo actual hasta el nodo final. La heurística es una estimación de la distancia total para que el robot no busque nodos que no tienen el menor costo.

**Fig. 6:** Diagrama de flujo del algoritmo A\*

Realizado dicho cálculo, el Agente elige el mejor camino basado en el valor más pequeño de f y ordena el movimiento.

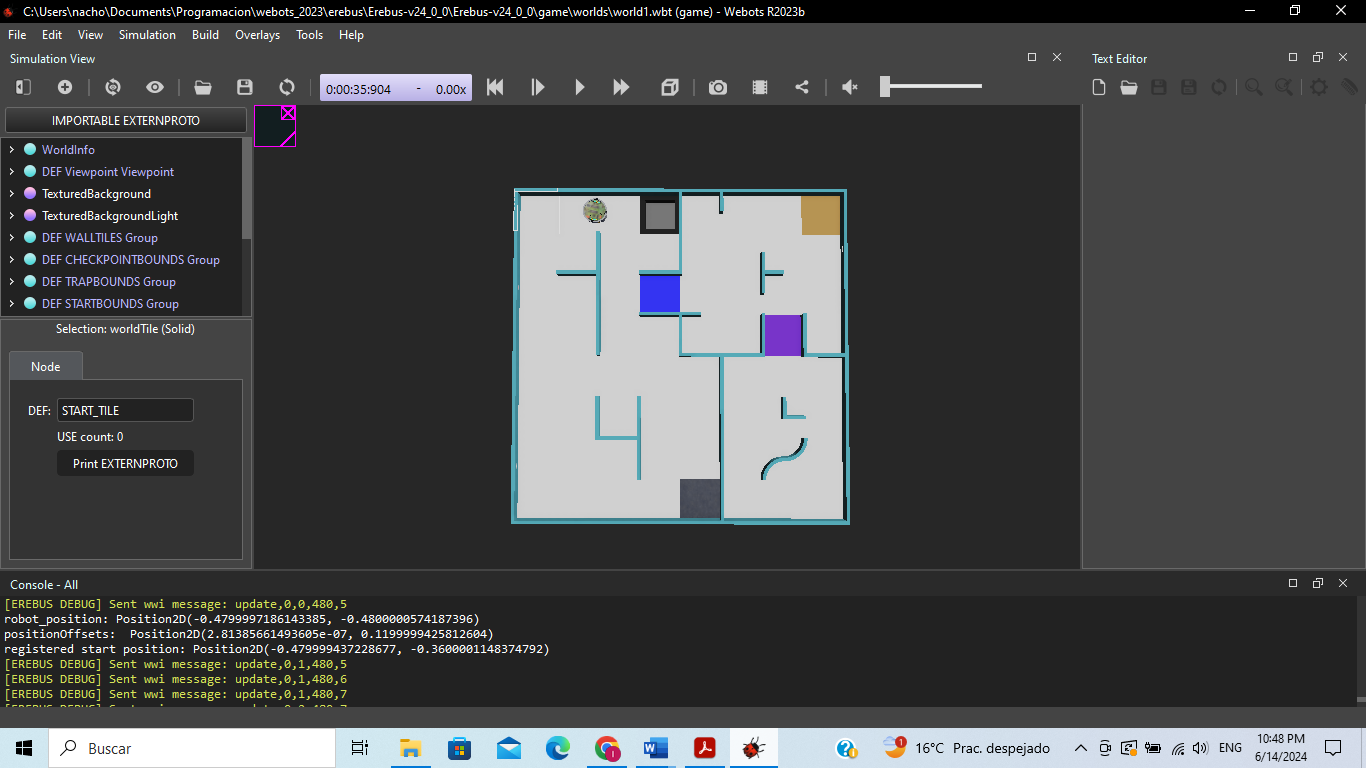


*Fig. 7 & 8: Ilustration of pathfinding algorithms (source:* ([11.1], [12.1]])

Con las siguientes ilustraciones se demuestra que el algoritmo A\* puede encontrar el camino más corto hacia el nodo objetivo y hacerlo en la menor distancia recorrida. Aun así, el algoritmo A\* no siempre encuentra el camino que toma el menor tiempo en cubrir (por ejemplo, no evitando pantanos), por lo que es una característica que aún necesita ser perfeccionada.

### Movimiento

De acuerdo a las instrucciones dadas por el Agente después de analizar los datos recopilados, el ejecutor controla el movimiento

**Pre-búsqueda de caminos**: Debido a que el robot no siempre aparece en el centro exacto del nodo inicial, el Ejecutor ordena al robot para que se mueva alrededor del nodo inicial antes de comenzar la búsqueda de los posibles caminos para que este bien centrado. Realizando esto se logra calibrar los sensores, evitando que se alteren los valores originales del GPS y el giroscopio.

**Post-búsqueda de caminos**: El robot se mueve siguiendo una navegación basadas nodos. Como resultado, la navegación del laberinto es más fluida.

*Fig. 9: Calibration in the 2nd testing environment*

El GPS y el giroscopio se utilizan para obtener datos sobre la posición del robot. Sin embargo, el punto de referencia es el GPS, ya que con el giroscopio no es posible detectar un cambio en la orientación después de ser teletransportado (LoPs).

También se utiliza continuamente el escáner LiDAR, que envía información al Agente, creando una nube de puntos y desarrollando dos cuadrículas necesarias para el mapeo (ver sección 3d).

# c. Wall Token detection

Se implemento un sistema de reconocimiento utilizando datos recogidos por las 3 cámaras, controladas por el Agente, que identifica objetos basándose en el color, la forma y el tamaño.

Se aplican a la cámara tres filtros (rojo, blanco y amarillo), y cada uno configurado para detectar un color específico según su tono y saturación. El Agente verifica el número de píxeles en la imagen, tanto en el perímetro como en el área, y los clasifica por color.

**Víctimas y señales de peligro**: El robot analiza constantemente los datos de las cámaras y si una pared está dentro del rango (~10 cm de radio), usando los filtros de color, el Agente puede diferenciar los accesorios del entorno.

Una vez detectado un accesorio, la biblioteca OpenCV obtiene una vision clara llamando a la función cv2.warpPerspective() (ver [13]) para corregir distorsiones. Luego se analiza la cantidad y disposición de los píxeles en el perímetro de la imagen, el Agente determinar las esquinas de la imagen y, diferencia si es una víctimas o señales de peligro.

*Fig. 10: Approximate detection range*



*Fig. 11: Fixtures, (a) original image, (b) corners detection, (c) perspective transformation*

**Víctimas:** Si es una víctima, la función cv2.cvtColor() (ver [14]) facilita el análisis modificando sus colores. A continuación, la función cv2.findContours() (ver [15]) detecta y recorta el contorno de la imagen.

*Fig. 12: Victims, (a) black and white, (b) cropped corners, (c) vertical zones*

La imagen recortada se divide en tres zonas verticales. Se verifica la proporción de píxeles blancos a píxeles negros en cada zona, comparando el área, y contrasta el resultado con las proporciones precargadas de cada posible víctima (ver Fig. 12).

**Señales de peligro:** Si se detecta una zona de peligro, se llama a la función cv2.getPerspectiveTransform() para rotar las imágenes (ver Fig. a). Luego se llama a la función cv2.mean() para determinar sus canales de color.

*Fig. 13: Hazard Signs, (a) 45° rotation, (b) black and white, (c) vertical zones*

Si el color promedio es gris, la señal de peligro puede ser "Veneno" o "Corrosivo". En cambio, si los canales de color no son iguales, el símbolo puede ser "Gas Inflamable" o "Peróxido Orgánico". En ambos casos, se llama a la función cv2.cvtColor() para suprimir colores.

El reconocimiento de señales logra dividiendo la imagen en tres zonas y contrastando la proporción de píxeles blancos a píxeles negros (ver Fig 15).

**Tipo de suelo:** El reconocimiento del tipo de suelo solo depende del color (el robot almacena su posición reconociendo la baldosa de color que marca el inicio de cada nueva área).

El robot le es posible ver suelo mientras navega gracias a una mínima inclinación hacia abajo en las tres cámaras. La imagen se corrige luego con la función cv2.warpPerspective().

Por lo tanto, el robot usa constantemente la función cv2.mean() para determinar los canales de color del suelo. Esta información se compara con valores pre-cargados para determinar el tipo de cada baldosa.

La información sobre el tipo de baldosa también se utiliza en el proceso de mapeo (ver sección 3d). Por lo que los datos se almacenan para crear las cuadrículas finales.

**Obstáculos:** Los obstáculos se reconocen teniendo en cuenta el color, la forma y el tamaño de cada detección de la cámara.

Primero, un filtro de color específico diferencia los obstáculos del entorno y de los accesorios (ya que los obstáculos son incoloros). El Agente verifica el contorno del obstáculo contando en número de pixeles y calculando su área, obteniendo su tamaño y ajustando el cálculo del camino más corto.

*Fig. 16: Obstacles*

*Fig. 15: Greater loss of time on the shortest path due to swamps*

*Fig. 14: Shortest path recalculation due to the hole blocking the node*

El Agente calcula la distancia entre el robot y el obstáculo, gracias a la detección LiDAR (ver sección 3d). Una vez definida la posición del obstáculo, la información se almacena para el proceso de mapeo

**Problema con la resolución de la cámara:** Surgió un inconveniente con las nuevas resoluciones de las cámaras, ya que pasamos de tener la máxima a la mínima, provocando que todo lo anteriormente desarrollado quede obsoleto. Para la solución de esto se buscó readaptar el código para evitar perder mucho tiempo. De esta manera se descubrió que modificando la variable min\_fixture\_height() de un valor de 16 a uno de 10 el programa volvía a funcionar, teniendo que cambiar únicamente los valores precargados del tipo de suelo

# d. Mapping

**Navegación y detección**: La navegación y la detección son muy importantes para el mapeo. Mientras el robot navega por el laberinto, el Agente utiliza el escáner LiDAR para crear una nube de puntos (consulta la sección 2). Guarda la posición de paredes y obstáculos, y con dicha información para crear un array de NumPy bidimensional.

*Fig. 17: Integer’s grid after the simulation of the 1st testing environment*

**Rejilla de enteros:** Cada detección realizada durante la ejecución se almacena en un array redimensionable de NumPy de enteros (ver Fig. 17), denominado más tarde como "primera rejilla". En términos más sencillos, el número de puntos en la rejilla aumenta con cada detección.

*Fig. 18: Granular grid halfway through a run on the 1st testing environment*

**Rejilla granular**: El Agente desarrolla una nueva rejilla cruzando el primer array LiDAR con otra información recopilada, como el tipo de suelo y la posición de los accesorios. El resultado es una rejilla "granular" con toda la información necesaria, pero aún no en el formato de la rejilla de bonificación.

Incluso si el array 2D creado en el proceso de mapeo es el resultado de cruzar las rejillas LiDAR, ambos procesos coexisten, lo que significa que la representación visual del mapa se expande dinámicamente a medida que el robot explora el laberinto. Este resultado se puede ver durante la simulación (ver Fig. 18).

Una vez que la simulación se detiene y la rejilla granular está completa, se convierte al formato de la rejilla de bonificación: una matriz (un array rectangular de números).

**Inclusión de víctimas a la grilla de bonificación**

Visto que el programa en primer lugar inicia con una rejilla que va almacenando datos y luego es redimensionada, resulta imposible agregar las victimas detectadas a la rejilla de bonificación (matriz). Por lo que decidimos recopilar la información de la posición y el tipo de victimas detectadas en un array único para luego formar una rejilla granular aparte que contenga estos datos.

Luego se compara la rejilla granular principal (la formada con los datos de las paredes y suelos) con la rejilla granular de víctimas. De esta manera se determina la posición correcta de las víctimas en la rejilla de bonificación

Para identificar que tipo de victima que corresponde en cada posición decidimos que al mismo tiempo que se detecta la posición, se almacene en un array diferente el tipo de víctima (la posición de la víctima y la posición del tipo de víctima no coincide por lo que se deben realizar estas 2 operaciones por aparte), identificada con un numero que depende del orden en el cual se detecto dicha victima. Con esta primera rejilla de enteros se forma una rejilla granular la cual se transforma a su vez en una grilla de bonificación que contiene solo los tipos de víctimas.

Posteriormente se comparan las 2 rejillas de bonificación, agregando el tipo de victima (letra) a la rejilla la cual contiene la posición de las misma (rejilla principal)

**Corrección de la grilla de bonificación**

Terminada la rejilla de bonificación se le realiza un análisis a la misma, para determinar si hay posibles errores en su creación, más específicamente en la creación de las paredes. Para solucionar esto se aplicaron distintas funciones, que toman como base la posición en donde puede ubicarse el error y luego analizan sus alrededores. Debido a que puede haber paredes en cualquier parte de la rejilla se crearon funciones que corrigen las paredes de los bordes y las paredes interiores en cualquier posición, incluidas las paredes curvas. De esta manera si se logra identificar un error, el programa cambiara el valor de dicha posición a uno correspondiente y en caso de no serlo seguirá analizando el resto de la rejilla.

Agregar anexos (si es necesario) sobre cómo funciona OpenCV y otras librerías

Ajustar imágenes y resolución de las mismas

Corregir la presentación del documento (formato, tipo de letra, interlineado, etc.)

**Referencias**

[1] RoboCup Rescue Simulation website

<https://junior.robocup.org/robocupjuniorrescue-league-simulation/>

[2] NumPy library official website

<https://numpy.org/>

[3] OpenCV library official website

<https://opencv.org/>

[4] Talos’ first forum exchange (2022)

<https://junior.forum.robocup.org/t/erebus-simulation-platform-new-release-camera-resolution-change/2319>

[5] Red Blob Games (2017), Introduction to A\*

<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

[6] Tree-traversal:

<https://runestone.academy/ns/books/published/pythoned/Trees/RecorridosDeArboles.html>

[7] Tree-traversal algorithm research

<https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

[8] Depth-first search algorithm research

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>

[9] Breadth- first search algorithm research

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> 12

[10] Hasha Mangena, via harshamangena.hashnode.dev

< https://aman.ai/code/assets/code/dfs\_bfs2.png>

[11] Dijktra’s algorithm research

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

[11.1] Subh83, CC BY 3.0, via Wikimedia Commons

<https://upload.wikimedia.org/wikipedia/commons/2/23/Dijkstras\_progress\_animation.gif>

[12] A\* algorithm research

<https://www.geeksforgeeks.org/a-search-algorithm/>

[12.1] Subh83, CC BY 3.0, via Wikimedia Commons

<https://upload.wikimedia.org/wikipedia/commons/5/5d/Astar\_progress\_animation.gif>

[13] OpenCV library, geometric transformation

<https://docs.opencv.org/4.x/da/d6e/tutorial\_py\_geometric\_transformations.html>

[14] OpenCV library, changing color space

<https://docs.opencv.org/4.x/df/d9d/tutorial\_py\_colorspaces.html>

[15] OpenCV library, contour approximation

<https://docs.opencv.org/3.4/d4/d73/tutorial\_py\_contours\_begin.html>

[16] Talos’ previous testing environments

<https://github.com/CoolRobotsAndStuff/erebus\_testing\_environment>

[17] Talos’ current testing environments

<https://github.com/iita-robotica/testing\_simulator>

*Additional resources*

A\* algorithm research.

G. Tang, C. Tang, C. Claramunt, X. Hu and P. Zhou, "Geometric A-Star Algorithm: An Improved A-Star Algorithm for AGV Path Planning in a Port Environment," in IEEE Access, vol. 9.