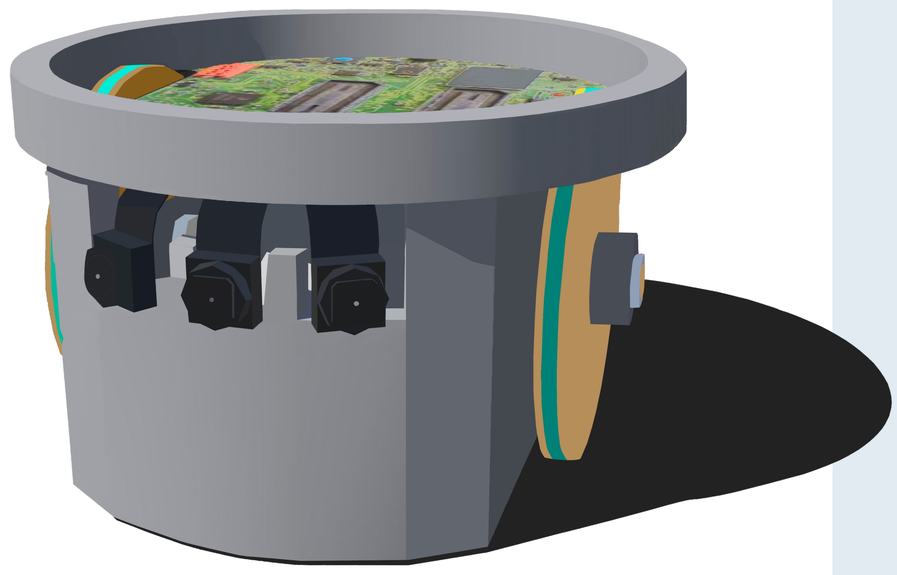
RoboCup 2023 - Junior Rescue Simulation

TALOS



Design

Strategy

Sensors were placed symmetrically so that the noise interference from the simulated environment didn't affect the robot's functionality.

No distance sensors were used, as the LiDAR scanner is a better alternative: it has no trouble identifying curved walls, calculating distances, or sampling angles. Likewise, no color sensors were needed since the cameras already have a filter for color distinction and can also be used for fixture detection.

Components

Motors (x2): used for movement throughout the maze, as they have independent speed and direction. Each is connected to a wheel.

Cameras (x3): visual recognition of fixtures and floor color. Two are on the sides, and one is in the front, allowing an almost 90° field view.

GPS: obtains the robot's coordinates, guides it during the navigation (and reorientates it in case of a reset). It is in the center to get accurate measurements.

Gyroscope: measures the robot's orientation, checking that it advances in a straight line. Located in the center to get accurate measurements.

LiDAR: creates a point cloud that detects objects and obstacles surrounding the robot. Placed on top of the robot, taking full advantage of its 360° detection.

Main loop

Init: calibrates the position and rotation of the robot. Only runs when the program starts, changes to Explore afterwards.

Explore: sends the LiDAR grid to the Agent and requests instructions. Works as a cycle: the grid is updated once instructions are executed.

Report_victim: triggered when a victim is detected, it stops the robot for one second, reports the victim to the supervisor and changes back to Explore.

Stop: debug state (no apparent cause), changes back to Explore once the problem is solved.

Stuck: debug state if something is blocking the robot, changes back to Explore once unstuck.

Teleported: debug state in case of LoP, changes back to Explore after recalibrating the position.

End: sends an end of play after completing the navigation, mapping, and detection.

Algorithms

Language

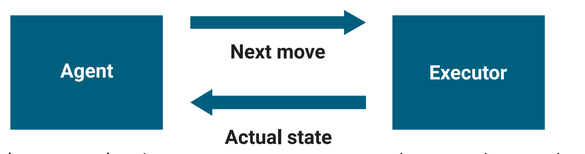
Python (discarded attempts to implement C++).

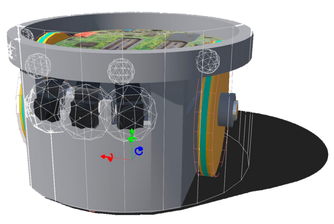
Architecture

The program has two parts:

Agent (long-term planner): analyzes a 2d array of nodes to check the most efficient path towards a particular point in the maze. It gives instructions to the Executor.

Executor (short-term actor): processes the data collected from the sensors to update the Agent's array. In charge of simple movement by following the Agent's instructions.





Algorithms for Detection

Instead of using color sensors, we implemented a method based on the data collected by cameras that identifies targets based on their color, shape, and size.

Fixtures

Ithe robot constantly collects and analyzes camera data if a wall is in range (~10 cm radius). Using color filters, the Agent can differentiate fixtures from the environment.

Once a fixture is detected, the OpenCV library is used to recognize contours and isolate symbols for analysis, differentiating victims and hazard signs.

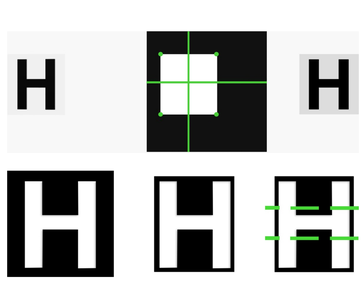
Victims

As for victim recognition, the cropped image is divided into three vertical zones. By contrasting the proportion of white to black pixels in each zone, the Agent can differentiate each type of victim.

Hazard signs

Likewise, Hazard sign recognition also requires dividing the image into three zones. If the average color between the channels is gray, the hazard sign can be “Poison” or “Corrosive.” Otherwise, the symbol can be “Flammable Gas” or “Organic Peroxide.”

Either way, an OpenCV function is called to suppress colors in the image, allowing the Agent to compare the proportion of white to black pixels.

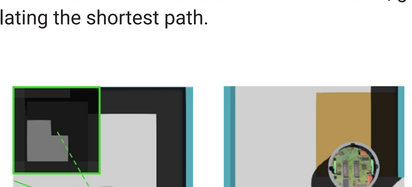


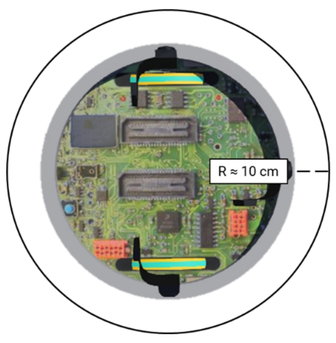
Floor-type

The robot sees the floor as it navigates, thanks to a slight downward inclination on the three cameras. The image is then corrected, and its color channels are determined through two OpenCV functions. This information is then compared to pre-loaded values to select each tile's type, allowing the agent to plan a response.

Obstacles

The cameras' inclination makes detecting objects connected to the floor easier. As obstacles are colorless, a specific color filter can help distinguish them from the environment. The agent then checks the number of pixels in the contour and area of the obstacle, getting its size and recalculating the shortest path.





Algorithms for Navigation

The Agent is in charge of the pathfinding: it plans which way will give the best tradeoff. The Executor takes care of the movement: it moves in the direction the Agent chooses and sends back new information.

Pathfinding

Instead, a Depth-first search with a branch-by-branch exploration would have been unnecessary and slower.

As for finding the best path, we went with the A* algorithm, which aims to find the path with the least cost (least distance traveled). In the following diagram, g is the distance between the current node and the start node, h is the estimated distance from the current node to the end node, and f represents the path's efficiency.

We also checked Dijkstra's algorithm, and although it always found the shortest path to the goal node, it didn't always correlate with the least distance traveled, resulting in a loss of time.

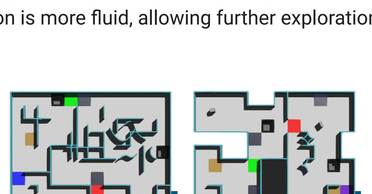
Movement

Pre-pathfinding

As the robot doesn't always spawn in the center of the initial node, the Executor commands a series of movements for sensor calibration, preventing the original difference from tampering with the GPS and Gyro.

Post-pathfinding

Thanks to the GPS and Gyroscope data, the robot can move following non-tile-based navigation. This means the instructions given by the Agent and received by the Executor are not tied to tiles but to nodes. As a result, maze exploration is more fluid, allowing further exploration.



Flowchart of the “A” algorithm

Algorithms for Mapping

The Agent uses the LiDAR scanner to create a points cloud, storing the position of walls and obstacles throughout the maze. The information collected is then stored and used in 2-dimensional NumPy arrays.

Integers grid

Every detection made during the run is stored in a resizable NumPy array of integers (the number of points on the grid increases with each detection). Still, as the robot's navigation is no longer tiled-based, the mapping cannot be achieved with the best precision.

Granular grid

The Agent develops a new grid by crossing the first array with other information collected, such as floor type and fixtures position. The result is a “granular” grid with all the relevant information.

Once the simulation stops and the granular grid is complete, the agent converts it to the bonus grid's format: a matrix (a rectangular array of numbers).

Team

ID: Talos from the IITA institute in Argentina.

Members: Alejandro de Ugarriza, Ian Dib.

Background: Alejandro won first prize in two national competitions and participated in two RoboCup categories between 2020 and 2022. Ian only participated in minor events and is new to international leagues.

Roles: Alejandro developed a non-tiled-based navigation system to explore all map areas freely and an improved fixture detection system to identify incomplete images. Ian took on the state flow in the program to test and research new innovations, leading to an overall performance improvement.

Argentina

Our code is open source