RoboCup Junior Rescue Simulation 2023

Team Description Paper

Talos

Abstract

In the RoboCup Junior Rescue Simulation, “the virtual robot is tasked with exploring and mapping a maze with different rooms and identifying victims on its way” (see [1]).

In this case, the robot uses the 360° field of view from the LiDAR scanner alongside a GPS and a Gyro to achieve non-tile-based navigation. Furthermore, three cameras make it possible to identify each fixture even if the images are incomplete.

Index terms

RoboCup Junior, Rescue Simulation, Pathfinding, Deep-first search, A* algorithm, Detection, OpenCV, Mapping, NumPy.

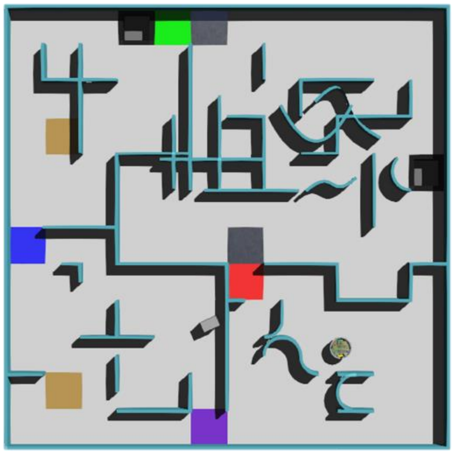


Fig. 1: Simulation on a testing environment

1. Introduction

a. Team

Our team, Talos, chose its name as a homage to the giant automaton in Greek mythology, created to protect Crete by circling its shores.

Biography

We are Talos from the “Instituto de Innovación y Tecnología Aplicada” (IITA) in Salta, Argentina. Although the team was initially formed for the 2022 RoboCup competition, its members have changed: Alejandro de Ugarriza is the only founder still on board, while Ian Dib is a new addition.

Alejandro won first prize in two “Roboliga Simulada” national competitions and participated in two RoboCup categories between 2020 and 2022. On the other hand, Ian has only participated in minor events and is new to international leagues.

Roles

- Alejandro developed a navigation system that abandoned the tile-based algorithm to explore all map areas freely and a fixture detection system that could identify incomplete images.
- Ian took on the state flow in the program to test and research new alternatives that would help prioritize tasks, leading to an overall performance improvement.

b. Project planning

Aim

As Talos’ second take on the competition, we aimed to surpass our previous program: we intended to develop an algorithm that could not only navigate through any area and identify fixtures correctly but also maintain decent performance.

We wanted to challenge ourselves, and now we look forward to exchanging ideas with other groups to share our passion for the competition and robotics in general.

Overall plan (2022)

As for the 2022 RoboCup competition, our initial goal was to implement Machine Learning to some extent. To organize our work, we divided the tasks into gradual milestones:

Milestone #1

The first step was to decide the foundations of our work: we chose the type and number of sensors that would be used (see section 2) and then designed a scheme for the general architecture of the code (see section 3a).

No significant problems were presented while working on these tasks. Afterward, we decided to use the Numpy (see [2]) and OpenCV (see [3]) libraries to work more efficiently.

Milestone #2

Once the foundations of the program had been built, each member worked on a section: Alejandro took on the navigation, whereas Joaquin Rodriguez (a previous member) focused on fixture detection.

By using the A* algorithm (see section 3b), Alejandro could implement proper pathfinding to the robot. This change was implemented after failed tests in which the robot could not complete the maze due to a lack of consistency.

On the other hand, Joaquin finished the detection system after fixing a problem with the camera resolution thanks to an exchange on the RoboCup forum (see [4]).

Milestone #3

As the competition's date closed by, the team focused on improving the remaining part of the navigation, as well as developing the mapping:

- Navigation through a low-level grid was done by the LiDAR sensor, allowing obstacle avoidance while ignoring unnecessary data such as the position of tiles, walls, and vertices.
- Mapping was achieved through a color detection system that used a camera instead of a color sensor —and combined it with LiDAR data to filter out walls—.

The team ended up in sixth place, as the robot could not complete the maze in the real-time limit.

Overall plan (2023)

Upon analyzing our program's main drawbacks in the past competition, we established efficiency as the main priority since this would improve performance without sacrificing other aspects.

Although we are still in the phase of developing and testing, these have been the milestones regarding the 2023 Competition so far:

Milestone #1

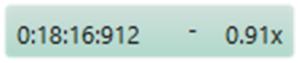

0:18:16:912 - 0.91x

Fig. 2: real-time speed controller

Before working to improve the code, we decided to go back to basics. Therefore, Ian performed a series of standardized tests (see section 4) that controlled how rearrangements in some functions affected the robot's performance.

These changes made the program more efficient by temporarily sending unnecessary functions to a "background" state in our main loop (the real-time speed jumped from 0.3 to almost 1.0).

Milestone #2

After the efficiency improvement, Alejandro focused on exploring new strategies:

- Navigation depends entirely on the points cloud created with the LiDAR, resulting in a non-tile-based algorithm that allows a better exploration, especially in area 4.
- The fixture detection system can recognize victims and hazard zones even if it only detects half of each picture, allowing for an efficient run through the maze.

Although we intended to develop a Machine Learning system for the robot to navigate the maze as Talos did in Crete, we still need to make it work.

2. Robot Design

The design considers the most efficient solution for each task while keeping in mind the budget on the robot customization tool.

To prevent noise generated by the simulated environment from affecting the robot's functionality, we placed the sensors symmetrically (see Fig. 3). This way, the interference is the same on each side and does not cause any malfunctions.

Components

- Motors (x2): used for movement throughout the maze, as they have independent speed and direction. Each connected to a wheel.
- Cameras (x3): visual recognition of fixtures, obstacles and floor type. They replace color sensors to lower the budget (by using color filters). Two are on the sides and one is in the front, allowing an almost 90° field of view.
- GPS: obtains the robot's coordinates, guides it throughout the navigation and re-orientates it in case of LoPs (teleportation). It is located in the center to get accurate measurements.
- Gyroscope: measures the robot's orientation, working as a detector if it is not advancing in a straight line. It is located in the center to get accurate measurements.

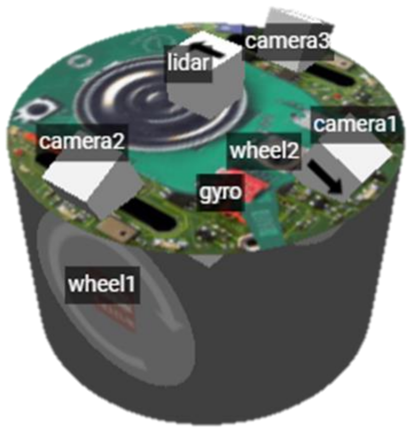


Fig. 3: Robot design on the customization tool

- LiDAR: creates a points cloud that detects objects and obstacles surrounding the robot. In contrast to distance sensors, it has no trouble identifying curved walls, calculating distances, or sampling angles. It is placed on top of the robot, taking full advantage of its 360° detection.

  *The LiDAR can also get raw detection, mainly used to correct the robot's orientation.

3. Software

Given our initial plan to include Machine Learning, our program is divided into two:

- Agent (long-term planner): analyzes a 2d array of nodes to check the most efficient path towards a particular point in the maze (see section 3b). 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.

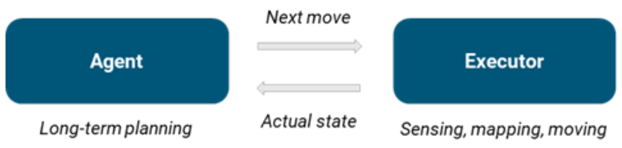


Fig. 4: software architecture diagram

a. General software architecture

We separated the robot’s actions into two groups to boost performance: a main loop with constant tasks and a state machine with temporary functions. The sequence manager controls this last group, allowing them to be run without interrupting the rest of the loop.

Loop

It contains the functions responsible for the robot’s navigation, mapping, and detection. The loop constantly gathers data from the Agent and provides the Executor with a set of tasks to complete every action.

Furthermore, it checks for emergency states (Stuck, End, or Report_vicitm) that could make the program malfunction. Once it detects any changes, it sends a signal to start the state machine.

State Machine

There is a specific state for each situation or change detected in the main loop. It includes:

- Init: calibrates the position and rotation of the robot and does start-up tasks (it only runs when the program is started and changes to the Explore state immediately after).
- Explore: sends the LiDAR grid to the Agent and requests instructions (while applying low-level position correction). It works as a cycle: the grid is updated once the instructions are executed.
- Report_vicitm: triggered when a victim is detected, it stops the robot for one second, reports the victim to the supervisor –changing the victim’s state in mapping to reported– 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 (but not the wheels), changes back to Explore once the robot is 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.

b. Navigation

During the navigation, both the Agent and the Executor are used. “There’s a tradeoff between planning with pathfinders and reacting with movement algorithms. Planning generally is slower but gives better results; movement is generally faster but can get stuck” (see [5]).

The Agent is in charge of the pathfinding: instead of trying every trajectory, it plans which way will give the best tradeoff. On the other hand, the Executor is in charge of the movement: it moves in the direction chosen by the Agent, sending new information for analysis.

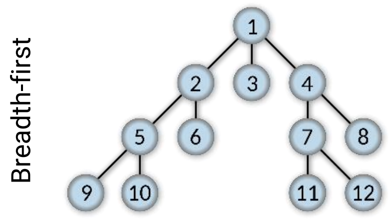
Pathfinding

The pathfinding depends on the information detected during the mapping. Both co-occur.

Upon receiving information from the 2d array, the Agent checks which nodes are traversable. To do so, a Tree-traversal algorithm (see [6]) is used.

Although a Depth-first search (see [7]) would have allowed the robot to go through most paths, the branch-by-branch exploration is unnecessary and slower. Thus, a Breadth-first search (see [8]) was chosen to explore nodes level by level, following the goal node directly.

As for finding the shortest path, we looked into two different algorithms:



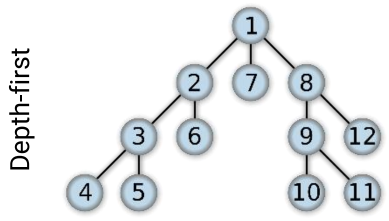


Fig. 5 & 6: Order in which the nodes are visited in each algorithm (source: ([7.1], [8.1])

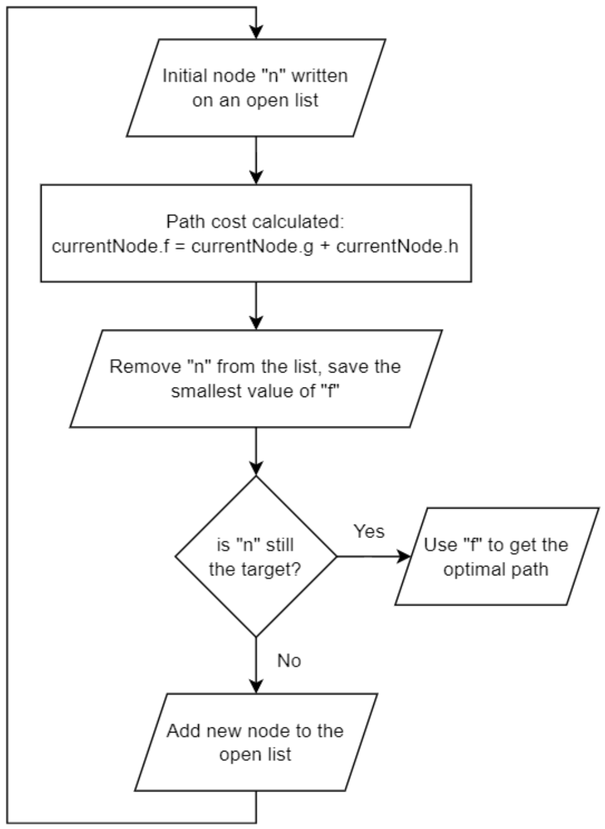
Dijkstra's algorithm

The robot visits every vertex detected on the 2d array, repeatedly checking the closest not-yet-examined one and adding the new detections to the set. The grid expands outwards: from the starting point until reaching the goal (see [9]).

Despite Dijkstra's algorithm being the best option to find the shortest path to the goal node, it does not always do it with the smallest cost. Thus, we chose to take a different approach instead.

A algorithm*

The robot maintains a tree of paths originating at the star node and extending until one of them reaches the goal node. It aims to find the path with the least cost —in this case, the least distance traveled— formulated in weighted graphs (see [10]).

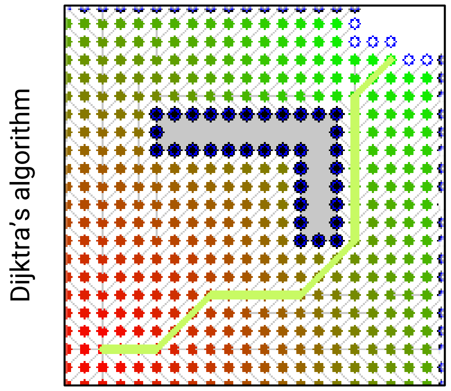

Initial node "n" written on an open list
Path cost calculated: currentNode.f = currentNode.g + currentNode.h
Remove "n" from the list, save the smallest value of "f"
is "n" still the target?
Yes
Use "f" to get the optimal path
No
Add new node to the open list
Fig. 7: Flow chart of the A algorithm*

To achieve that, the total cost of the node, f, is calculated through the formula:

$$currentNode.f = currentNode.g + currentNode.h$$

With *g* being the distance between the current node and the start node, and *h* (heuristic) being the estimated distance from the current node to the end node. The heuristic must always be an underestimation of the total distance. Otherwise, the robot would search for nodes that may not have the smallest cost.

Once the calculation is done, the Agent chooses the best path based on the smallest value of f and orders the movement. The impact of this f variable in pathfinding is reflected in the following illustrations (see Fig. 8 and 9).



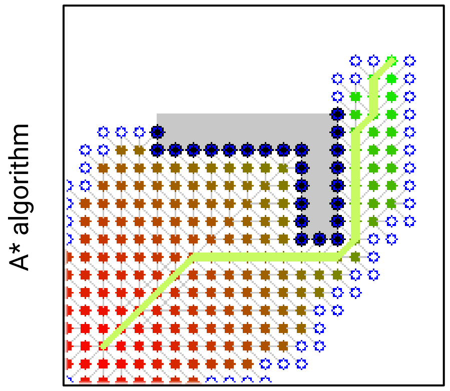


Fig. 8 & 9: Illustration of pathfinding algorithms (source: ([9.1], [10.1])

Therefore, the A* algorithm can find the shortest path to the goal node and do so in the least distance traveled. Even still, the A* algorithm does not always find the path that takes the shortest time to cover (by not avoiding swamps, for instance), thus being a feature that still needs to be polished.

Movement

The Executor controls the movement itself. Mostly, it follows the instructions given by the Agent after analyzing the data collected.

Pre-pathfinding

Given that the robot does not always spawn in the center of the initial node, the Executor instructs the robot to move around the starting node before the pathfinding begins. This movement allows for sensor calibration, preventing the difference from tampering with the GPS and Gyro original values.

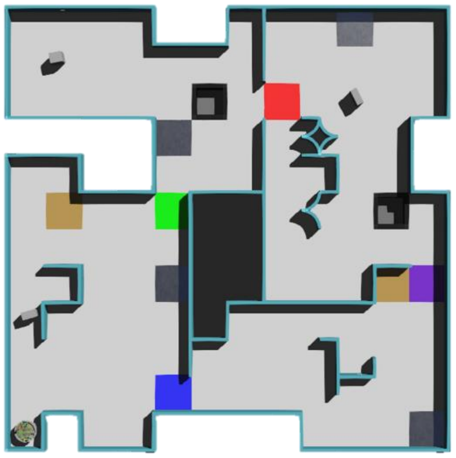
Once the pre-instructed movement finishes, the robot is well-centered and updates the sensors to start with the pathfinding.

Post-pathfinding

Unlike the program presented in 2022, the robot moves following non-tile-based navigation. Thus, 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 us to explore Area 4 further.

As stated before, the GPS and the Gyroscope are used to obtain data regarding the robot's position. However, the reference point is always taken from the GPS since the Gyroscope cannot detect a change in orientation after being teleported (LoPs).

During this stage, the robot continuously uses the LiDAR scanner and sends the information to the Agent, creating a points cloud and developing two grids necessary for mapping (see section 3d).



```
• robot_position: Position2D(-0.7200000571946387,0.599999732195614)
• positionOffsets:  Position2D(0.1199999428053613,0.119999732195614)
• registered start position: Position2D(-0.6000001143892774,0.719999464391228)
```

Fig. 10: Calibration in the 2nd testing environment

c. Detection

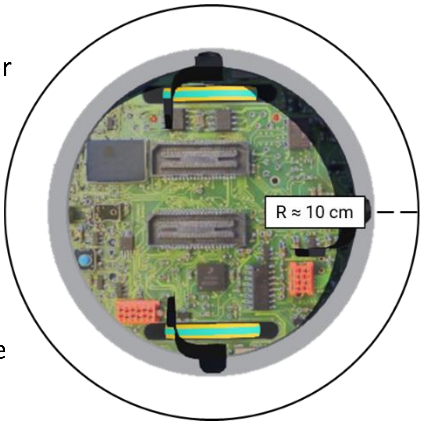


Fig. 11: Approximate detection range

The Robot Design (see section 2) shows that our robot does not use color sensors to identify the fixtures or the tile color. Instead, we implemented a recognition system using data collected by the cameras –controlled by the Agent– that identifies objects based on color, shape, and size.

Regarding color detection, three filters are applied to the camera (red, white, and yellow), and each one is configured to take a specific color according to its hue and saturation –avoiding as much noise as possible.

As for shape and size, the Agent checks the number of pixels in the image –in the perimeter and the area– and classifies them by color.

Victims and Hazard signs

Fixture detection is a continuous process: the robot constantly collects and analyzes camera data if a wall is in range (~10 cm radius). Therefore, using color filters, the Agent can differentiate fixtures from the environment.

Once a fixture is detected, the OpenCV library gets a clear perspective by calling the cv2.warpPerspective() function (see [11]) to correct distortions (see Fig. 12). Then, the shape comes into play. By analyzing the amount and disposition of the pixels in the image's perimeter, the Agent can determine the position of the image's corners and, thus, distinguish between victims and hazard signs.

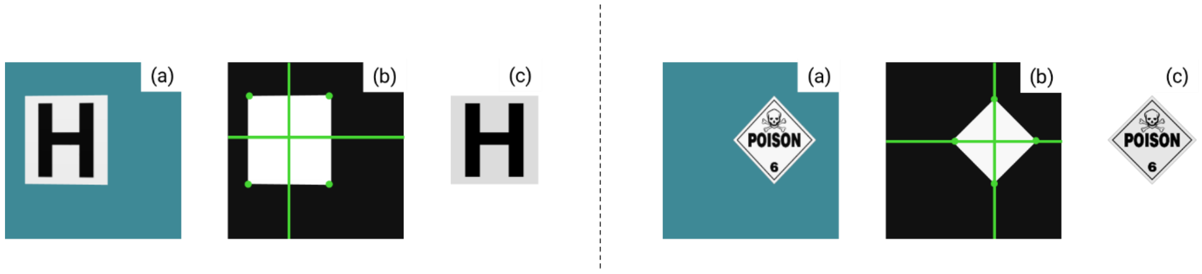


Fig. 12: Fixtures, (a) original image, (b) corners detection, (c) perspective transformation

Victims

If the fixture detected is determined to be a victim, the cv2.cvtColor() function (see [12]) eases the analysis by modifying its colors. Next, the cv2.findContours() function (see [13]) detects and crops the contour of the picture.

As for the victim recognition itself, the cropped image is divided into three vertical zones. The Agent then checks the proportion of white pixels to black pixels in each zone –area comparison– and contrasts the result with the pre-loaded ratios corresponding to each victim's fixture (see Fig. 13).

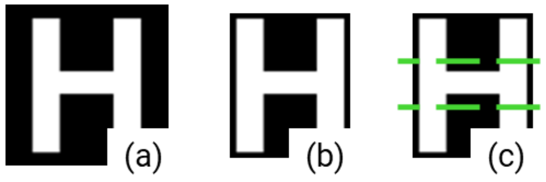


Fig. 13: Victims, (a) black and white, (b) cropped corners, (c) vertical zones

Hazard signs

On the other hand, if the sign detected is determined to be a hazard zone, the cv2.getPerspectiveTransform() function is called to rotate the images (see Fig. a). Before comparing pixel proportions in the images' areas, the cv2.mean() function is called to determine its color channels.

If the average color between the channels is gray, the hazard sign can be "Poison" or "Corrosive". Instead, if the color channels are not equal, the symbol can be "Flammable Gas" or "Organic Peroxide". In both cases, the cv2.cvtColor() function is called to suppress colors, as with the victim signs.

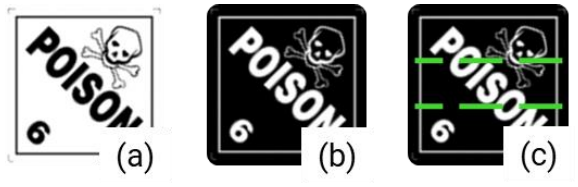


Fig. 14: Hazard Signs, (a) 45° rotation, (b) black and white, (c) vertical zones

The sign recognition is still done by dividing the image into three zones and contrasting the proportion of white pixels to black pixels (see Fig 14).

This determination by area comparisons has one significant advantage: ultimately, the robot does not need to see the victim entirely, allowing fast and effective detection. Unlike the 2022 program, the new comparison matrix optimized the program in both time and computational cost.

Floor type

Floor type recognition only depends on color since its shape and size do not vary inside each area (the robot stores its position by recognizing the color tile that marks the start of each new area).

The robot sees the floor as it navigates thanks to a slight downward inclination on the three cameras. The image is then corrected by the cv2.warpPerspective() function.

Therefore, the robot is constantly using the cv2.mean() function to determine the color channels of the floor around it. This information is then compared to pre-loaded values to determine each tile's type, allowing the Agent to plan a response.

For instance, if a black hole is detected, the shortest path is recalculated by considering that point as an inaccessible node (see Fig 15). In the case of swamps, however, we noticed that recalculating the shortest path sometimes led to an even more significant loss of time than telling the robot to go through them. Thus, we didn't prohibit the robot from entering the swamps, though this is not the best answer (see Fig. 16). We plan to study these cases further to keep optimizing the program.

Either way, the information regarding tile type is also used in the mapping process (see section 3d). So after its immediate analysis, the data is stored to create the final grids.

*Note: this section was "Tile type", but since navigation is no longer tile-based, neither is the detection.

Obstacles

As with fixtures, obstacles are recognized by considering the color, shape, and size of every camera detection. The cameras' inclination makes detecting objects connected to the floor easier.

Firstly, a specific color filter allows us to differentiate obstacles from the environment and fixtures (since obstacles are colorless and easier to recognize). The Agent then checks the number of pixels in the contour and area of the obstacle, getting its size and recalculating the shortest path.

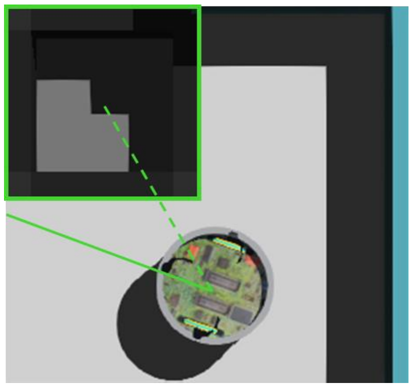


Fig. 15: Shortest path recalculation due to the hole blocking the node

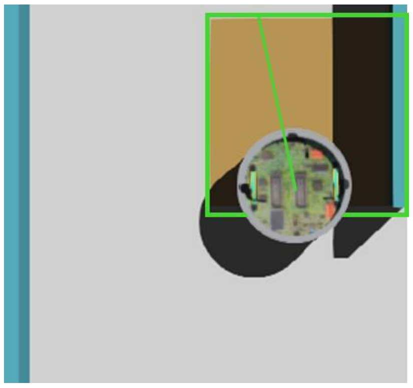


Fig. 16: Greater loss of time on the shortest path due to swamps

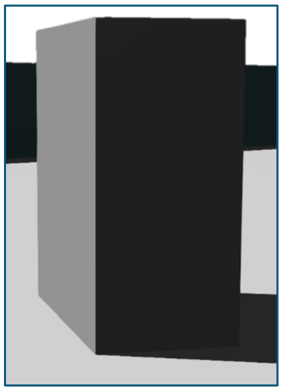


Fig. 17: Obstacles

The Agent calculates the distance between the robot and the obstacle, thanks to LiDAR detection (see section 3d). Once the obstacle's position is defined, the information is stored to be used in the mapping process, the same way as with the floor type.

d. Mapping

Navigation and detection play significant roles in mapping.

As the robot is navigating the maze, the Agent uses the LiDAR scanner to create a points cloud (refer to section 2). It stores the position of walls and obstacles, using this information to create a 2-dimensional NumPy array.

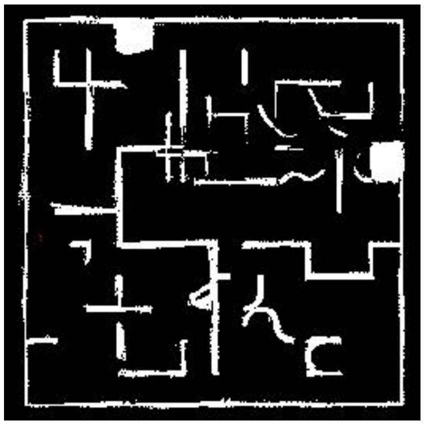


Fig. 18: Integer's grid after the simulation of the 1st testing environment

Integers grid

Every detection made during the run is stored in a resizable NumPy array of integers (see Fig. 18) –later called "first grid". In other words, the number of points on the grid increases with each detection.

Nevertheless, our main improvement for this competition is also our most significant drawback. Since the robot's navigation is no longer tiled-based (refer to section 3b), the mapping cannot be achieved precisely. Therefore, the borders between tiles are unclear, affecting the following grid.

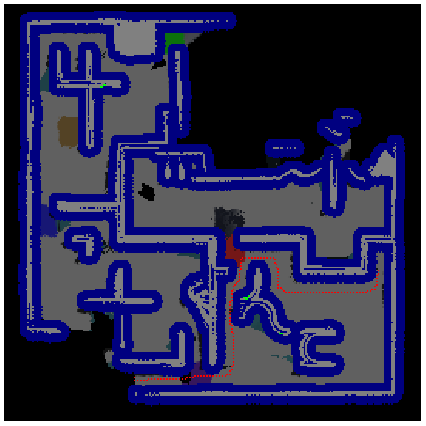


Fig. 19: Granular grid halfway through a run on the 1st testing environment

Granular grid

The Agent develops a new grid by crossing the first LiDAR array with other information collected, such as floor type and fixtures position. The result is a "granular" grid with all the relevant information, but still not in the bonus grid's format.

Even if the 2d array created in the mapping process is a result of crossing the LiDAR grids, both processes co-occur, meaning that the visual representation of the map expands dynamically as the robot explores the maze. This result can be seen during the simulation (see Fig. 19).

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

Unlike the previous version of our code –which used a Nodes grid instead of the Granular– the current version can properly detect curved walls, as it no longer returns false positives.

4. Performance evaluation

Generally, the performance evaluation started with analyzing the program to mark improvable features. Then, research and evaluation of alternatives helped filter the ideas. Lastly, standardized testing allowed us to measure the changes' effects.

Furthermore, resources such as the RoboCup forum or Stack Overflow were checked regularly to ask for references on facing the problems and constraints shown during the tests.

Testing procedure 2022

As the changes applied to the program became more frequent, checking whether they made an actual improvement became unclear. Thus, we implemented standardized testing (see [14]).

This methodology relies on reproducibility: by making maps on the map creator tool, it was possible to run each version of the program on different scenarios and compare their performance (virtual-time speed, map completion, and correctness, etc.).

Therefore, we initially estimated that 11 maps would be enough. However, during the day of the competition, the overall virtual-time speed calculated on the tests was not reflected in the round's simulation, proving that a more sophisticated procedure would have been needed to accurately predict the robot's performance.

Testing procedure 2023

Upon realizing the constraints during the 2022 RoboCup, we made 20 sample maps to get more accurate measurements (see [15]). Therefore, the data collected along the maps could be filtered, calculating more precise overall values.

Despite taking more time, this testing procedure was essential to check whether the changes made regarding pathfinding were reflected during navigation and how it affected the mapping algorithm.

General performance

The robot can navigate the first three areas and usually avoids LoPs, although its performance in the fourth area can still be improved. It can detect all fixtures and floor color using three cameras that have proven worth it, even with a limited budget. Furthermore, the new algorithms generally allow for map correctness of around 90%.

Overall, the new program focuses mainly on efficiency: completing every task to its maximum possible extent without sacrificing performance. Thanks to the improved structures in the main loop, the simulation speed is not so far behind the real-time speed as in the past competition.

5. Conclusion

Given our past performance in the RoboCup Rescue Simulation, we intended to improve our general performance while developing new innovative ideas and algorithms. Even if a Machine Learning system could not yet be included, we managed to implement some other improvements: the navigation is no longer tile-based, fixtures can be detected with only half the image, floor color detection does not need a color sensor, and the simulated-speed can catch up to real-time speed.

Therefore, we succeeded in many of our goals for the competition while still having room for improvement. Furthermore, as a team, we got to research and learn various new topics and alternatives that we are eager to discuss with other participants to expand our knowledge.

Acknowledgments

The authors would like to thank the RoboCup committee for keeping the simulated category after the pandemic since it represents a significant opportunity for teams that have fewer chances of getting proper pieces for other competitions, like ours.

We would also like to thank the IITA Institute for its unconditional support throughout this journey and Salta's government authorities for giving us a chance to travel and compete. It is great being able to share enthusiasm and knowledge with teams all around the world.

Appendix

For additional information, our progress can be found in the following GitHub repository:

<https://github.com/iita-robotica/rescate_laberinto/tree/master>

Despite we were not able to implement Machine Learning, all our research can be found in:

<https://github.com/CoolRobotsAndStuff/machine-learning-for-maze-exploration>

References

[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 algorithm research

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

[7] Depth-first search algorithm research

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

[7.1] Alexander Drichel, CC BY-SA 3.0, via Wikimedia Commons

<https://upload.wikimedia.org/wikipedia/commons/1/1f/Depth-first-tree.svg>

[8] Breadth- first search algorithm research

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

[8.1] Alexander Drichel, CC BY 3.0, via Wikimedia Commons

<https://upload.wikimedia.org/wikipedia/commons/3/33/Breadth-first-tree.svg>

[9] Dijktra's algorithm research

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

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

<https://upload.wikimedia.org/wikipedia/commons/2/23/Dijkstras_progress_animation.gif>

[10] A* algorithm research

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

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

<https://upload.wikimedia.org/wikipedia/commons/5/5d/Astar_progress_animation.gif>

[11] OpenCV library, geometric transformation

<https://docs.opencv.org/4.x/da/d6e/tutorial_py_geometric_transformations.html>

[12] OpenCV library, changing color space

<https://docs.opencv.org/4.x/df/d9d/tutorial_py_colorspaces.html>

[13] OpenCV library, contour approximation

<https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html>

[14] Talos' previous testing environments

<https://github.com/CoolRobotsAndStuff/erebus_testing_environment>

[15] 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.