RoboCupJunior Rescue Simulation 2024

Team Description Paper

73 RoBits

Abstract

In RoboCupJr, Rescue Simulation, the virtual robot is set to explore a maze, while identifying wall tokens and saving them inside it. To achieve that, the team developed a virtual robot with a code by using a 360° LiDAR and Cameras to detect obstructions, such as walls and obstacles, and avoid them, alongside GPS and Gyro sensors to help with navigation.

With a non-tile based mapping and a modified Rapidly-exploring Random Trees algorithm on navigation, our robot is able to explore efficiently through the entire map. Moreover, it is able to recognize rotated or warped wall tokens on camera, for high-accuracy token identification.

By exploring different methods and algorithms, the team developed a robot that is capable of accomplishing the tasks set to it with high efficiency and accuracy, being able to perform greatly on very challenging mazes, while using very few computational resources.

1. Introduction

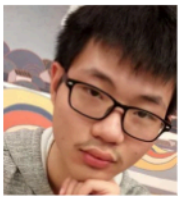
a. Team

- Marcos Menezes Nunes:

Hello, I'm Marcos, 17 years old. I've been a part of this rescue simulation team since 2022, having worked before on the CoSpace league. Besides robotics, I also participate in Programming and Math Olympiads. Those contributed to the early stages of the competition software and the development of new strategies in general. I've worked on the software architecture, the modularity of the code, besides the navigation aspect of the robot, doing functions related to exploration and movement.

- David Icheng Wang Chou:

Hey there! I'm David, a 17 year-old student that likes to code. I joined 73 RoBits in August 2023, participating for the first time in the national stage of the RoboCup Junior. I also participate in other competitions, such as Programming and Math Olympiads. I'm responsible for the robot's mapping system, and also worked on designing new maps, while helping with other robot functionalities.



- Sophia Yara Yano:

Hello! My name is Sophia. I'm a 16-year-old student and I've always loved everything related to robotics. In 2022, I joined the 73 RoBits team where I was able to meet people from different countries who share the same interest for robotics. On the team, I'm responsible for image recognition, design of hardware and documentation, and also creating some maps for testing.

2. Project Planning

a. Overall Project Plan

After having performed really well at last year's competition, the team felt extremely motivated with this year's Robocup Jr, aiming even higher in implementing better algorithms, and hopefully doing well in this year's competition too. To this end, it's fundamental that the robot surpasses its past challenges, and the new ones that appeared as the team progressed.

The main problems noticed from the last Robocup are: inconsistencies when navigating area 4, such as not entering small paths or getting stuck when turning; the misidentification of wall tokens, especially H, S and U; and the difficulties stemming from high code complexity, making the debug process difficult.

It was also essential to adapt to the new competition rules. The team needed to improve the wall token detection system with new strategies for the camera, which can now be customized, but

also dealing with the fact that tokens can appear rotated on the wall. Moreover, the team decided to implement non-tiled based exploration algorithms, such as RRT (Rapidly-Exploring Random Tree, see [6]), to better avoid obstacles and go around the arbitrarily shaped walls in area 4.

With this in mind, these were the milestones decided when planning the team project:

Milestone #0 (Strategies):

The first step in robot development was discussing the team's previous experiences to determine the goals for robot development. The main issues presented earlier were outlined and a general development schedule was set.

Deadline: To be done in January by the whole team.

Milestone #1 (Robot's Hardware):

In order to adapt the robot to the new Erebus/webots update, the next challenge was to configure the new robot hardware, optimizing the placement of parts, and creating a database with the sensor's configurations for easy access in software development.

Deadline: To be done in early February by the whole team.

Milestone #2 (Software Structure):

To simplify the code structure, which had become a bottleneck in development last year, the software was divided into modules (different python files), each with different classes for certain functions (see section 4a). Part of the work included researching how to efficiently do this.

Deadline: Main structure to be done by the end of February by Marcos. Other auxiliary processes were created as development progressed.

Milestone #3 (General Code):

After structuring the software architecture, came developing strategies and algorithms decided on Milestone #0. Since there are 3 main parts of the software, navigation, mapping, and wall token detection, Milestone #3 is divided into .1, .2 and 3.

Milestone #3.1 (Navigation):

Navigation was to be improved by using an RRT algorithm (see [6]). Having a non-tile based algorithm facilitated working in area 4 and identifying obstacles, as mentioned earlier. This milestone included researching and implementing the RRT into the navigation module of the software.

Deadline: To be finished in May by Marcos.

Milestone #3.2 (Mapping):

Alongside the new navigation system, the mapping system also needed to be changed. The walls were to be stored as its coordinates instead of bits in a matrix (non-tile based algorithm). These points are provided by processing the LiDAR rays (see section 4d). There was also the task of mapping different ground tiles, such as connection tiles, blackholes and swamps based on camera information.

Deadline: Wall mapping to be finished by May, by David. Components that could only be detected by cameras (see Milestone #3.3) were to be done by early June, by Sophia and David.

Milestone #3.3 (Wall Token Detection/Identification):

The detection of wall tokens and identification follows the same general strategy as the one used last year (see section 4c), mostly requiring fixes in the formulas, as the cameras are now.

The team also tried to develop a machine-learning algorithm as an alternative to victim identification; however, the low accuracy of this kind of algorithm made it worse than the strategies previously used.

Deadline: To be finished late May by Sophia.

Milestone #4 (Bug hunting and required documentations):

To have a robust code, it is necessary to create a thorough testing routine that can identify corner cases in the code, as well as possible bugs, improving performance before the competition (see section 5). Besides that, it was necessary to write the TDP and the poster as they are required documents, which can greatly affect the end-game performance.

Deadline: To be done in June and July. Assigned to the whole team.

As the work progressed between the milestones, there were changes to the original plans, due to tasks requiring more or less time to be done than what was planned, or new strategies that appeared as we developed the robot, to be talked about later on the TDP.

3. Robot Design

In designing the robot, the team needed to have a general overview of the code's tasks, as to plan which sensors would work best in executing each of the software's functions. The hardware needs to support the robot's movement, mapping, locating and computer vision.

Firstly, and also most simply, the robot uses wheels on its left and right side, making it possible to move forward or backwards and rotate itself.The robot uses LiDAR for mapping, instead of normal distance sensors, because it allows for more information and flexibility, showing 512 evenly spaced rays of distance each time it is updated. It was placed on the top front of the robot, in the same X and Z positions as the cameras, thus simplifying the trigonometric formulas which transform pixel positions into map coordinates (see section 4c).

The next hardware task is to know the location of the robot. For this, the robot has a GPS and a Gyroscope. The former is positioned on the robot's center for easier calculations on the space occupied by the robot, which can be obtained by just adding the radius of the robot to the GPS in whichever direction is desired. With the Gyroscope in the center, problems in measuring the angles are also avoided.

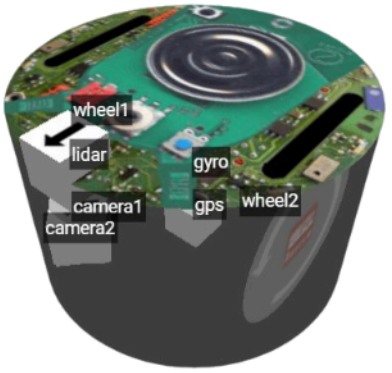


Figure 1: Robot Design on the Erebus Robot Customisation 2024

Finally, the robot uses cameras to find different objects in the map. To determine the optimal number, resolution and placement for them, the team created a gallery of images captured by different settings. From that, it was noticed that the vertical amplitude of the camera is less important than the horizontal one, because increasing the former would only allow it to capture more of the sky's area, completely irrelevant for the robot's objective; while increasing the latter contributes in identifying more area of the walls, wall tokens and other important details on the map at any given moment. Thus, the final composition is of two 128x40 resolution cameras, enhancing the robot's side view and at the same time allowing for the visualization of the color tiles by the robot, essential in both navigation and mapping. Additionally, the Y-axis position of the cameras were adjusted to be the same as the tokens on the walls for accurate identification, and the FOV was set to max on the software, providing almost 180° of vision.

4. Software

In this section we present an overview of the software's organization, as well as an in-depth analysis of all main functions of the code, separated into their respective subsections, which reflect the code's modular structure.

a. General software architecture

The software is divided into different modules, i.e. python files, each of which is composed of multiple classes. Besides keeping the code organized by separating variables and functions according to their usage, this modularity makes it easy to change certain functions during development and testing without getting the whole code inoperant.

If a new function gets added by the navigation module, for instance, the team could disable the camera class, making it easier to test this new function. Or if a problem occurs, turning off different modules makes it really easy to get to the root of the problem, speeding drastically the debugging process.

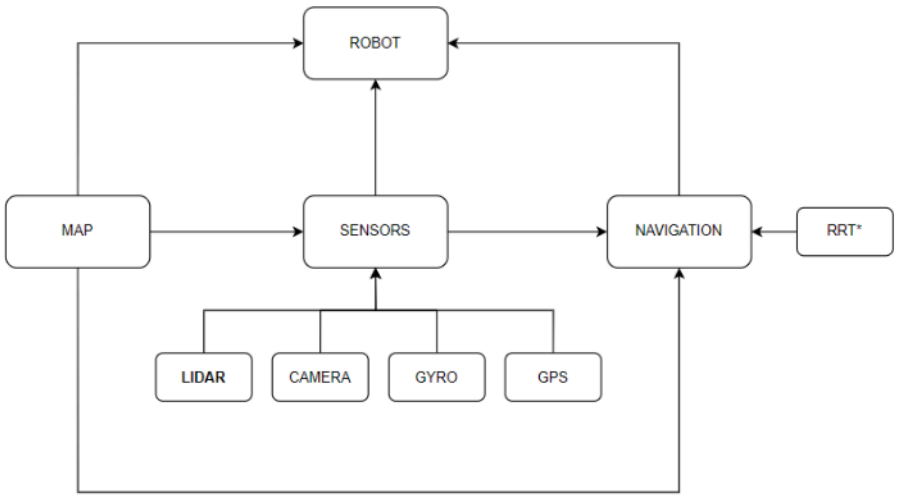


Figure 2: Flowchart of modules and classes of robot

The main module is "robot.py", containing the "robot", "sensors", "gps", "gyro", "camera" and "lidar" classes. There's also a navigation module with "RRT*" and "navigation" classes, and a mapping module with the "map" class.The main loop of the simulation lies in the robot class of the robot module. Initially it calls the ***run_calibration*** function that gets the initial position and angulation of the robot, ensuring that the robot's variables are well set before exploring the map.

After that, until the end of the round, the main loop repeatedly calls for the ***run_simulation*** function, which acts as the 'brain' of the robot. It decides which type of action the robot needs to do and how it should be done. The actions are divided into two types: routine actions and emergency actions.

As the name implies, routine actions are the main ones acted out by the robot, and include instructions such as exploring the map. "Routine" also expresses the fact that they are more regular in their execution than other parts of the code. They are:

- "**explore**" - Find new unexplored areas of the map and go to them.
- "**collect**" - Once a wall sign is seen through the camera, walk to it and try to collect after identifying its type.

On the other hand, the emergency actions are triggered by situations that could crash or delay the software, such as a LOP (Lack of Progress), being stuck by a wall or obstacle, or even entering an endless loop that would not tick the simulation. As their name implies, if called, an emergency action will enter as the next action for the robot, delaying or stopping the current one. They are:

- "**LOP**" - Recalculate robots angulation like ***run_calibration***
- "**stuck**" - Divided into two levels. The first one tries to walk back to the last position, marking the current one as explored. If it ends up stuck again is the second level where it stops for 20 seconds, forcing a LOP.
- "**while**" - Forcibly ends the current tick. If called multiple times ends up at a '**LOP**' or '**stuck**' action.

b. Navigation

The navigation part of the software involves both exploring the map and collecting wall tokens, and is based on a generic movement command called "***walk***".

The collection of wall tokens is done by the ***run_simulation*** function, which iterates through the tokens previously identified by the camera, checking whether it's possible to go to them from the

robot's current position. If so, a "**collect**" action is added and a ***walk*** command is sent to the navigation class, which executes it as a high priority action.

Meanwhile, the map exploration on the "**explore**" action was designed to solve the difficulties noticed from previous competitions. Beforehand, the code used a greedy search-based navigation system alongside a backtracking algorithm on the tiles of the map. That was the simplest algorithm that could walk through most parts of the map. However, that didn't properly work on the 4th area of the map, in which many corner cases would appear, such as paths with non-trivial angles and tight passages that didn't follow the tile structure.

This year, the team decided to use RRTs (see [6]): a sampling based navigation algorithm (see [12]) that works by creating random points in bulk.

More specifically, it repeats a routine of creating random points on the map, finding the closest previously sampled point and connecting the new point as a child of that node. This slowly creates a space-filling tree that reaches unexplored areas, leaving a path of points and parents that the robot can traverse. It is a simple but highly efficient algorithm which can quickly find paths in very detailed spaces, all while depending very little on the structure of the obstacles.

The lack of a grid on the points gives a huge advantage over other common search based algorithms such as **A*** (see [8]), **BFS** (see [9]) or **Dijkstra** (see [10]) since the robot can explore in any direction, not needing an extremely detailed map matrix to mark the tiles, which would hugely increase code complexity.

Adapting this general algorithm to the competition's restraints, the robot keeps track of the map with a matrix representing the areas accessible with the paths created by the RRT and the areas yet to be explored. Points reaching the latter areas are called frontier points.

When deciding to explore on the ***run_simulation*** function, the RRT is set to generate new points until it samples a frontier point, allowing for the navigation functions to be called, generating a "**walk**" list. Then, the robot passes through all points on the "**walk**" list until it reaches the frontier one, where it will once again update the RRT, keeping the navigation loop.

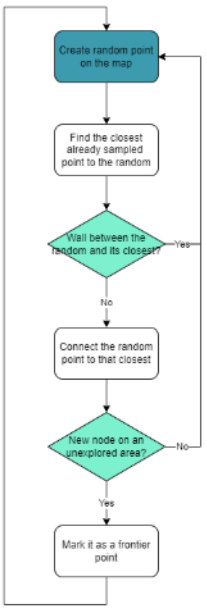


Figure 3: Flowchart of RRT

As of now, explored areas are those which have been seen through the cameras' images. This excludes the possibility of exploring areas that don't contain anything new or important to the robot, such as wall signs and colored tiles, decreasing significantly the time to run the full map.

To further increase exploration efficiency, the robot uses a modified RRT algorithm, called RRT* (see [7]): These slightly change the way to connect points by considering the distance to the initial point on the neighboring area, making navigation smoother.

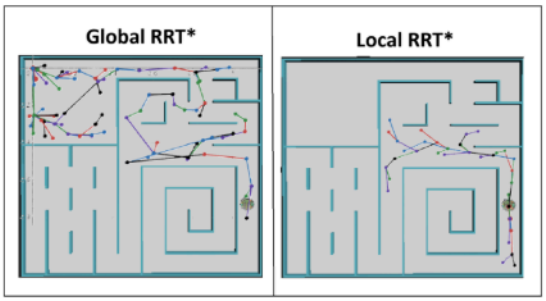


Figure 4: Comparison between Global RRT* and Local RRT*

It was also noticed that the randomness of the RRT would sometimes create deviations on exploration. To solve this issue, based on a researched idea (see [14]), two RRTs were implemented, instead of one: a global RRT, which doesn't reset throughout exploration, and a local RRT, which is reset every time the robot tries to find non explored areas on the ***run_simulation*** function. This increases the probability of frontier points being found closer to the robot.

To integrate this new algorithm into the navigation module, the team exploited the software's highly modular architecture; disabling unused modules and focusing on testing this one. To test the RRT, points were generated and taken into a graph maker, which was able to check its correctness before integrating into the movement of the robot. During the more advanced testing, the team created maps containing all previously identified problems, like small spaces and weirdly shaped obstacles, as well as some other theoreticized corner cases. The current navigation code is able to run through all these maps without issues.

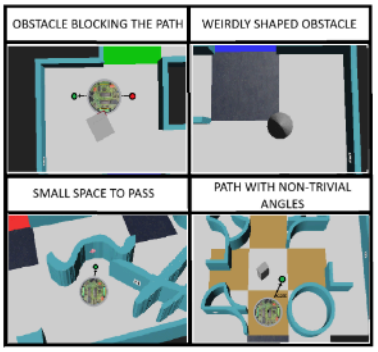


Figure 5: Previously identified corner cases

c. Wall Token detection

To make the object detection on walls, we use the OpenCV library from Python (see [2]). The code first joins both camera images into a contiguous one, to be analyzed.

To identify a wall token, the code checks through all camera pixels, starting on the top left corner and going in the right-down direction. It verifies if the pixel has the color of wall or not, searching vertically for a pattern of "non wall - wall - wall" . Observe that, if this pattern occurs at least two times, the first one will be because of the sky and the second one because of a one token (image x), that means that there is a token on this wall column.

The only exception to this happens when the robot is so close to the wall it can't see the sky (image x+1), which can be solved by simply checking if the uppermost pixel on the column is a wall. That process continues for all wall tokens and returns a list of coordinates of token positions in the image, called "**deltas**".

Based on the "**deltas**" list, the robot gets information of the size, position and angle of the respective tokens. Firstly, the robot finds the angle of the center position of it in relation to the robot angulation, then uses the closest LiDAR ray to that angle to get the token position (image x+2).

It also calculates the wall angulation using close by LiDAR rays. This information will be used to know which side of the wall the token is at, facilitating its collection later (image x+3). The acquired information is saved on a list of tokens, to be accessed later by the robot.

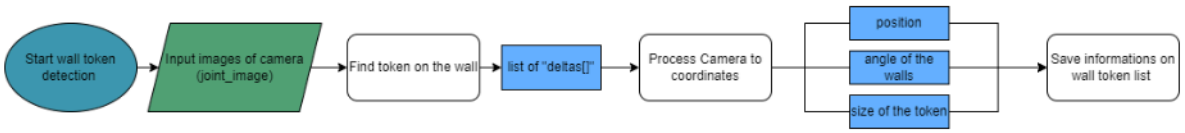


Figure 6: Flowchart of saving wall token

Whenever one of the marked points is decided to be accessible by the navigation algorithm, the robot calculates a path to it, in order to collect it.

After getting close to the wall sign, the robot does a full turn, constantly checking the middle pixel of the camera's image. If that pixel is not a wall and the LiDAR identifies a close object in that direction, the robot then gets as close as it can to the token, starting the victim type identification.

Avoiding the possibility of the token being rotated, the code finds the corners of the rectangle or rhombus wall token, and fixes them in the closest 90° to identify its type. To do that, the algorithm uses the warpPerspective function from OpenCV (see [3]). Then it differentiates victims from hazmats by checking the size of the wall token since hazmats are generally smaller.

If it determines that the token is a victim, it's possible to check the highest and lowest pixels of the middle column of the image, because if the highest and lowest are black the victim is a 'S', if only the lowest one is black, the victim is a 'U', and if none of them is black, the victim is a 'H'. If the token is a hazmat, the amount of yellow, red, black and white pixels are calculated. Having yellow pixels means that the token is an Organic Peroxide, if not, the red amount indicates a Flammable Gas, a considerable amount of black indicates a Corrosive, otherwise it's a Poison hazmat.

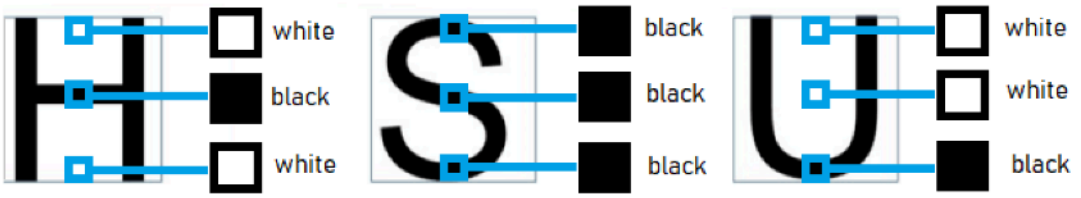


Figure 7: Process to identify the type of a victim

Continuing the collection, the robot gets closer to the token to send it using the emitter, and after a few seconds, returns to the original position.

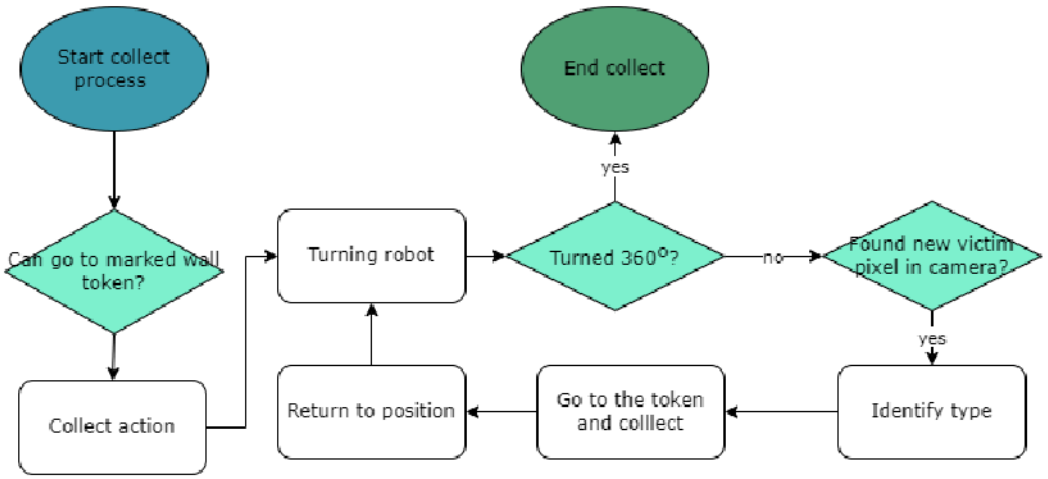


Figure 8: Flowchart of collect process

To develop this process, the team needed to do a lot of testing and research, as a lot of small bugs and mistakes were present from last year's code. First, a database of camera pictures was made with OpenCV, which could be analyzed individually for better bug-testing (see [4]). The team was using an innovative solution to check wall pixels by transforming the RGB colors in vectors. Also, we made maps with different victim placements (especially tokens on area 4 that gives more points), making it possible to test in different scenarios.

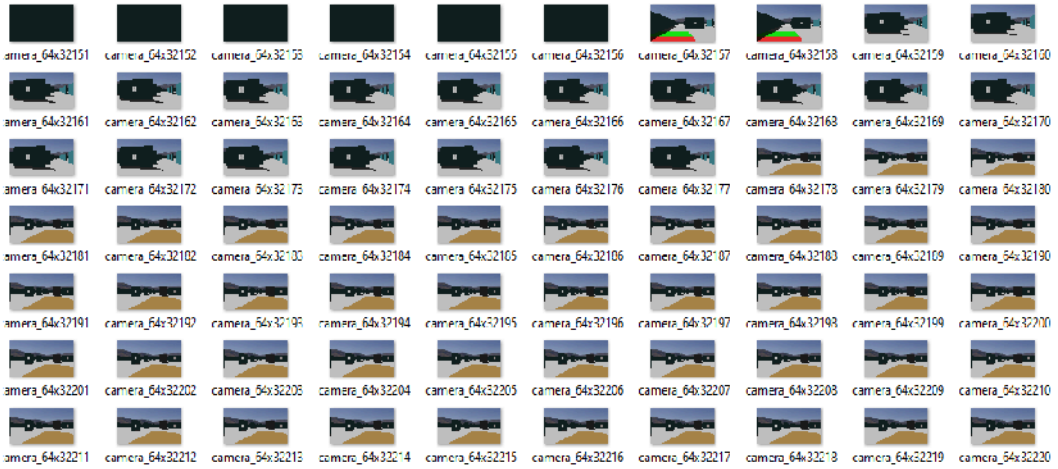


Figure 9: Files of camera tests' resolution

d. Mapping

The mapping system the team devised consists of four main parts: wall mapping, floor mapping, obstacle mapping and map emitting. This is because walls are mainly detected by the LiDAR; ground colors, such as black holes, wall tokens and connection tiles, mapped by cameras.

- **Wall Mapping**

The team uses LiDAR to detect where the walls are. As mentioned before, the LiDAR has 512 rays uniformly distributed around the robot, each of which detects the distance to the nearest

physical object in its direction. By using trigonometry, it is possible to save their coordinates relative to the starting tile into a list, forming a point cloud of obstacles around the robot.

The team's first idea to process this information was just to store them into a list, however this made the code very slow, as the code needed to iterate through the whole list whenever it tried to find a point. To solve this, in the robot's main code, there is a matrix whose entries represent half-tiles of the explorable maze. This matrix starts off only containing the initial tile, progressively increasing its size to store data on newly explored areas; whenever a new half-tile is detected, the matrix is expanded horizontally and vertically until it encompasses it. On each of the matrix's spaces, it's created a list that contains all wall points that have been detected within it.

This way, to check if there is a wall somewhere, one can access the half-tile's respective matrix index and get all points previously stored. Also, when adding points to the matrix, we avoid saving points that are near to each other, which wouldn't affect the navigation and only worsen the time used to detect if there is a wall near them.

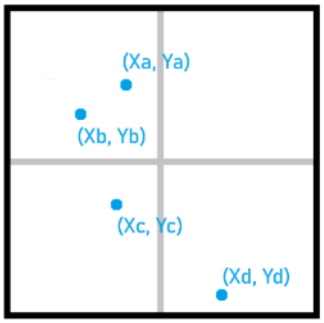


Figure 10: A tile with four wall points

On the left image, the black square represents the border of an tile, that is divided into 4 half-tiles by the gray lines, and four point in position (Xa, Ya), (Xb, Yb), (Xc, Yc) and (Xd, Yd). Each of these is stored in its respective half-tile for later access.

In comparison with last year's mapping (Appendix A), which was made by dividing each tile into a 2x2 or 12x12 depending on the area, marking an entire block as a wall if a point is detected within it, the new method has a higher accuracy, as it marks the precise information of all previously found points.

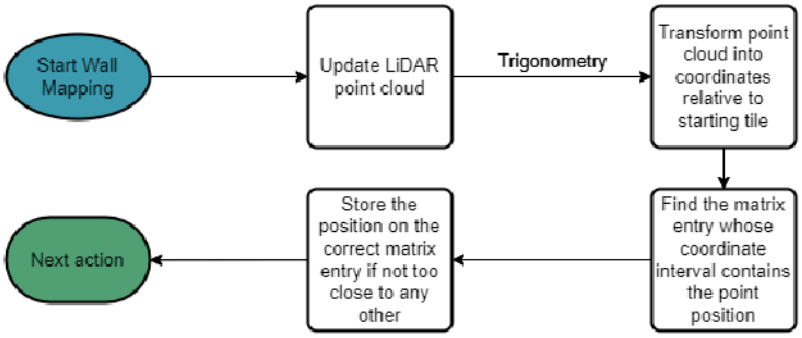


Figure 11: Flowchart of wall mapping function

- **Floor Mapping**

Using information coming from the cameras, the robot can detect connection tiles, swamps, black holes and checkpoints.

Each image generated by the camera is scanned from its bottom-up until a wall or the top of the image is reached while looking for the colors that represent the aforementioned map features. This way, every pixel that represents a ground object, i. e. an object below the wall or sky on the camera's image, is analyzed.

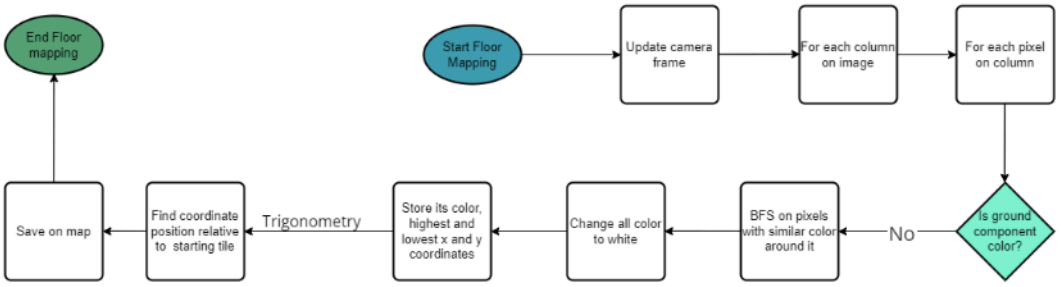


Figure 12: Flowchart of floor mapping function

While processing a pixel, the code transforms it from BGRA to HSV, as, in some sense, shadows do not change too much the "hue" of its color. Then, by comparing its hue and saturation to the values of the known map features, it is possible to know if the color represents a specific ground component.

If the pixel is not of a normal ground tile, i.e. connection tiles, swamp or black hole, the code does a BFS (see [9]), checking all pixels around it that have the same color category, saving this area's highest and lowest x and y pixel coordinates in a list, then changing this frame's pixels HSV values to represent white (the normal ground color) so that it doesn't get processed multiple times.

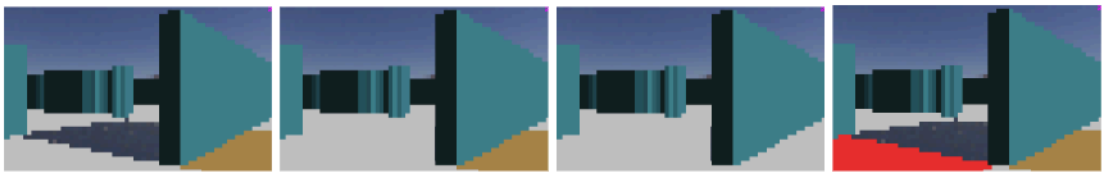


Figure 13: Images processed by robot after BFS passing through them and changing to ground color

After getting all component's highest and lowest (x, y) pixel coordinates the code combines each component's height on the image with the camera's height and vertical field of view, getting a angle that can be used to calculate the distance from the camera to this tile's midpoint and then find its coordinate on the map.

By testing, we found out that if two or more ground components were connected, the BFS would mark all the tiles with the same color around it as only one, making it return the wrong mid points, making the mapping of the area incorrect. To avoid this problem, we instead check for the proximity between colors, which differentiates features.

In last year's code (Appendix A), the ground mapping was made only by detecting the exact blocks ahead of the robot, making it necessary for the robot to pass through each tile to identify each ground component. In comparison to it, the new method made it possible for tiles to be mapped as long as they are detected by the camera, improving the robot's path planning.

- **Obstacle Mapping**

In order to correctly detect the obstacle's shape, the team decided to use two vertically different ray angles, so that when the robot gets close to an obstacle, the normal LiDAR ray is able to detect the distance to the obstacle's top while its lower ray would get the distance from its base. This way, the robot would mark as "unwalkable" every point coordinate that was detected, avoiding colliding with normal obstacles or even weirdly shaped ones like conical or spherical.

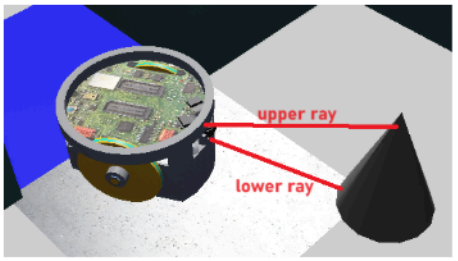


Figure 14: Different lidar angles detecting conical object

In last year's competition, there were a few problems related to the robot only using the horizontal LiDAR rays on obstacle's detection, resulting in it colliding with conical or spherical shaped ones, or even Lack Of Progress (LOP) errors. By using the new mapping system, these problems are avoided, as the robot is able to detect different shapes, having more accuracy on marking them as "unwalkable".

- **Map emitting**

Finally, after mapping the whole maze, it is necessary to emit it to the main supervisor in order to gain map bonus points.

As the wall mapping is done by creating lists of wall points, the code needs to transform into a correct matrix format first so that it can be submitted. The robot does this by creating a temporary matrix, which is filled with '1' and '0' according to the presence of points in corresponding half-tiles. After that, the robot adds other map features based on previously stored information (see section 4c), and does a cleanup of suspicious map features (disconnected walls, etc.).

5. Performance evaluation

In order to evaluate the robot's performance, it is necessary to test it on a lot of different maps, searching for corner cases, and then debugging them to be able to fix the problems that could appear during the competition.

- Testing:

The maps used to test the robot were both reused maps from last year's competition, made by either the team or the Erebus Rescue community, and new ones created by the team and focused on the past noticed problems (see section 2a) or new exceptional situations devised by the team.

When making new maps for testing, the team focused on applying the new map rules, such as different connection tile colors, obstacles positioned less than 8 cm of walls and rotated wall tokens. With this, the team created a set of maps exploiting all rules of RCJ's Rescue Simulation (see [16]).

Besides focusing on adding the new rules, lower restrictions and updates on the maps, we also tried to find corner cases that could decrease the robot's final score. Some of the corner cases are blackholes surrounding a wall, causing LiDAR to not detect it and marking it as ground; a bad angle for victim detection due to some wall positioned in front of it and some others that were already shown previously in specific testings of navigation, mapping and wall token detection.

During the more advanced testing phases, the team created a set of 70 maps (25 newly created by the team, and 45 from previous competitions) and also a google spreadsheet (see [17]), containing the information of each code test, including the score, map percentage and some extra observations (i.e. "Webots starts lagging around 02:30").

- Debugging:

Usually, when a bug is detected, a lot of checkpoint prints are sent to the console. To support these, the code can use its modularization (see section 4a), and furthermore some functions, such as ***print_map*** (print the entire map the way it is submitted), that would print general variables that could have been affected by a problem.

For debugging image related issues, the code used the python library OpenCV's ***imwrite()*** function (see [4]), which gets an image matrix and downloads it into the computer as a png file, making it easier to inspect a single frame. This function was also used for testing LiDAR point clouds when mapping, returning a black and white image of what is seen by LiDAR.

On navigation, debugging is a bit harder as RRT* has the unique feature of being random, meaning that each time the map is being simulated, the robot will always explore it using different paths, making it harder to repeat the same specific error. To overcome this, we created a way that the robot could print the RRT* graph on the simulator console in such a way that it could be almost copy-pasted to Desmos Graphing Calculator (see [3]), making it way easier to be debugged. It works by taking advantage of the browser console, using a JavaScript code to inject new lines between each node and its parent.

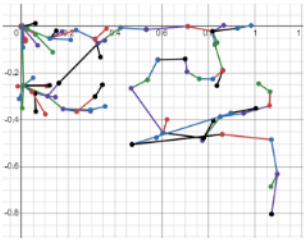


Figure 15: RRT* Graph on Desmos

6. Conclusion

The team's main aim was to improve the performance of our project while learning and developing new strategies and algorithms to get through the competition's challenges. Although still not perfect, the team's current solution can be said to have achieved these goals, by excelling in implementing more refined strategies and delving deeper into research throughout development.

That said, the team hopes to get a good performance in the competition and, more importantly, to gather more knowledge and experiences by interacting with other participants, one of the unique opportunities provided by the RoboCupJr.

Appendix A

The last year's TDP is available through the following link:
https://docs.google.com/document/d/1t1RAISkVmp_BduKqEr-1Frtv9oGIZT7Q/

References

[1] Numpy Library. Available at: https://numpy.org

[2] OpenCV Library. Available at: https://opencv.org

[3] OpenCV, Geometric Transformations of Images. Available at:
https://docs.opencv.org/4.x/da/d6e/tutorial_py_geometric_transformations.html

[4] OpenCV, Image file reading and writing. Available at:
https://docs.opencv.org/4.x/d4/da8/group__imgcodecs.html

[5] Stickytape Library. Available at: https://pypi.org/project/stickytape/

[6] Rapidly exploring random Trees. Available at:
https://medium.com/geekculture/rapidly-exploring-random-trees-rrt-and-their-much-nicer-properties-7b5d983e5b18

[7] RRT* Algorithm. Available at: https://www.youtube.com/watch?v=JM7kmWE8Gtc

[8] A* Algorithm. Available at: https://www.geeksforgeeks.org/a-search-algorithm/

[9] Breadth-first search Algorithm. Available at:
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[10] Dijkstra Algorithm. Available at:
https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

[11] A*, BFS, Dijkstra Comparison. Available at:
https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[12] A. Choudhary, "Sampling-based Path Planning Algorithms: A Survey". Available at:
https://arxiv.org/pdf/2304.14839

[13] Desmos Calculator. Available at: https://www.desmos.com/calculator

[14] H. Umari and S. Mukhopadhyay, "Autonomous Robotic Exploration Based on Multiple Rapidly-exploring Randomized Trees". Available at:
https://hasauino.github.io/assets/papers/rrtexploration.pdf

[15] Forum question on rotated signs. Available at:
https://junior.forum.robocup.org/t/victim-or-distractor/3664

[16] Robocup Junior Rescue Simulation 2024 Rules. Available at: https://rescue.rcj.cloud/rules/2024/RCJRescueSimulation2024-final.pdf

[17] Test Datas. Available at: https://docs.google.com/spreadsheets/d/1WRDsI9UFoxH_Gc0jU_cPRtAVwUQ1iHCxo6R_UMwDQ88