G.E.E.K.
RoboCupJunior Rescue Maze 2021

Neha Vardhaman — *neha.vard@gmail.com*
Sophia Chen — *sophia.nyc2004@gmail.com*

Mentor: Jeremy Desmond

Abstract **2**
Introduction **2**
Project Planning & Development Cycle **2**
Software/Algorithms **2**
Flowchart 2
Search Algorithm 5
Depth-First Search 5
Breadth-First Search 6
Testing 7
Letter Victim Identification 8
K-Nearest Neighbors 8
Letter Recognition 8
Testing 9
Color Victim Identification 9
Serial Communication 10
Hardware Design and Manufacturing **10**
Overall Robot Design 10
Osepp Tank Mechanical Kit 11
Two Tiers 11
Sensor Mounts 11
Rescue Kit Dispenser 12
Overall 12
Spinner and Base 13
Rescue Kit 13
Stepper Motor 13
Ramps 14
Servo 14
DC Motors with Encoder 15
Time of Flight Sensors 15
IMU (Gyro) 16
Stereo Pi and Raspberry Pi Cameras 16
MegaPi 17
Melexis Temperature Sensors 17
Logic Level Shifter 17
Color Sensor 17
Testing 17
Conclusion **17**

Abstract

This paper is about our team's preparation for the RoboCupJunior Rescue Maze 2021 Competition. We present an autonomous robot that can traverse a maze in search of victims. The maze consists of rugged terrain that the robot can move past. The robot is capable of identifying victims represented by letters, colors, and heat, signaling that a victim was found, and delivering rescue kits to specific victims. The robot explores the entire maze while attempting to maximize the efficiency of its path. We built and programmed this robot over the course of approximately 6 months.

Introduction

In this paper, we describe the choices we made during our preparation for the RoboCupJunior Rescue Maze 2021 Competition, as well as the rationale behind the choices. The paper begins with a description of our planning for this task in Section 3. Section 4 details our software design and the algorithms we employed, followed by Section 5 which details our hardware design and the components involved.

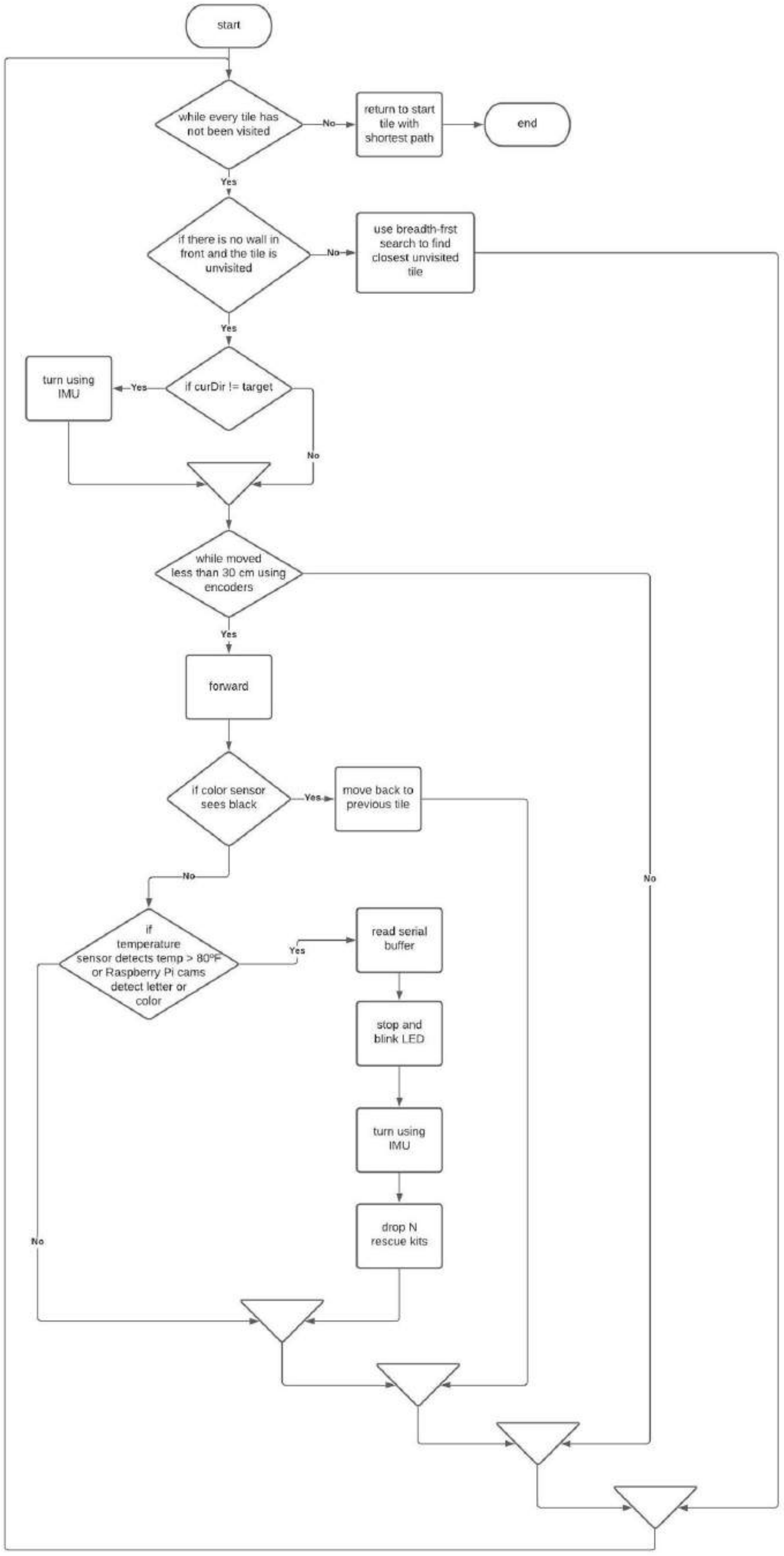
Project Planning & Development Cycle

We divided the overall task into several software and hardware sub-tasks, and set goal dates for the completion of each sub-task, as well as for the integration of the different sub-tasks. We kept more specific accounts of our progress in our Engineering Journal.

Software/Algorithms

1.1. Flowchart

1.1.1. Throughout the development of our algorithms, we referenced back to our flowchart for the overall program (see next page).

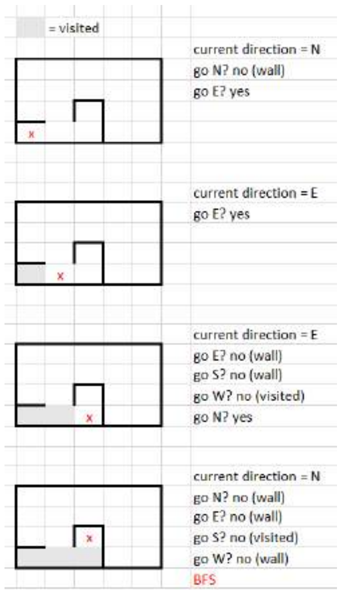

start
while every tile has not been visited
No
return to start tile with shortest path
end
Yes
if there is no wall in front and the tile is unvisited
No
use breadth-frst search to find closest unvisited tile
Yes
if curDir != target
Yes
turn using IMU
No
while moved less than 30 cm using encoders
Yes
forward
if color sensor sees black
Yes
move back to previous tile
No
if temperature sensor detects temp > 80ºF or Raspberry Pi cams detect letter or color
Yes
read serial buffer
stop and blink LED
turn using IMU
drop N rescue kits
No
No

1.2. Search Algorithm

1.2.1. We chose to use a combination of Depth-First Search and Breadth-First Search to traverse the entire maze efficiently.

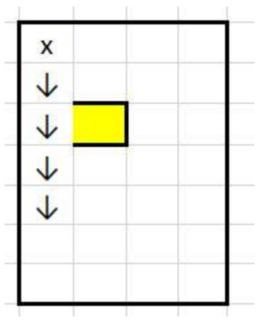
1.2.2. Depth-First Search

1.2.2.1. Depth-First Search is a graph traversal algorithm that starts at a root node and explores along each branch as far as possible before backtracking. We chose to implement this into our search algorithm by using it to visit as many tiles as possible while prioritizing moving in its current direction until all surrounding sides of the tile it's on has a wall or a previously visited tile.

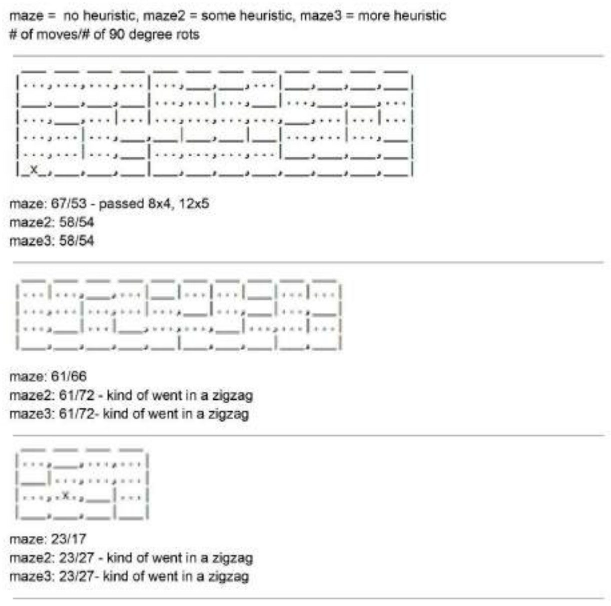
1.2.2.1.1. 

1.2.2.2. Our rationale behind using Depth-First Search for this purpose was that it can traverse unvisited tiles in a simplistic and fast manner.

1.2.2.3. Though we tried two heuristics, we decided not to use it in our final code. A heuristic we tried was if there is a wall one tile distance away during the Depth-First Search to go there in order to avoid passing single "rooms" to have to come back to it later, like in the case shown below where the Depth-First Search passes the yellow tile. The second heuristic is similar but prioritizes moving in the current direction over going to the single room. Our rationale behind not keeping the heuristics was that we tested a few types of heuristics and they did not prove to have a consistent or significant advantage when compared to no heuristic. There were even test cases where the heuristic provided a disadvantage. This is because the robot has no way to tell whether there is actually a single room next to it or if that tile is surrounded by other unvisited tiles.

1.2.2.3.1. 

1.2.2.3.2. We analyzed the algorithms with and without different heuristics by keeping track of the number of moves and turns that occurred to complete the search.



maze = no heuristic, maze2 = some heuristic, maze3 = more heuristic
# of moves/# of 90 degree rots

```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|...,...,...,...|...,___,___,...|___,___,___,___|
|___,___,___,___|...,...|...,___|...,___,___,...|
|...,___,...|...|...,...,...,...,___,...|...|...|
|...,...|...,___,___|___,___|___|...,...|...,___|
|...,...|...,___|...,...,...,...|___,___,___,___|
|_x_,___,___,___|___,___,___,___,___,___,___,___|
```

maze: 67/53 - passed 8x4, 12x5
maze2: 58/54
maze3: 58/54

```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|...|...,___,...|___|...|...|___|...|...|
|...,...|...,...|...,___|...,___|...,___|
|...,___|...|___,...,...,___|...,...|...|
|___,___,___,___,___|___,___,___|___,___|
```

maze: 61/66
maze2: 61/72 - kind of went in a zigzag
maze3: 61/72- kind of went in a zigzag

```
 ___ ___ ___ ___
|...,___,...,...|
|___|...,...,...|
|...,.x.,___|...|
|___,___,___|___|
```

maze: 23/17
maze2: 23/27 - kind of went in a zigzag
maze3: 23/27- kind of went in a zigzag

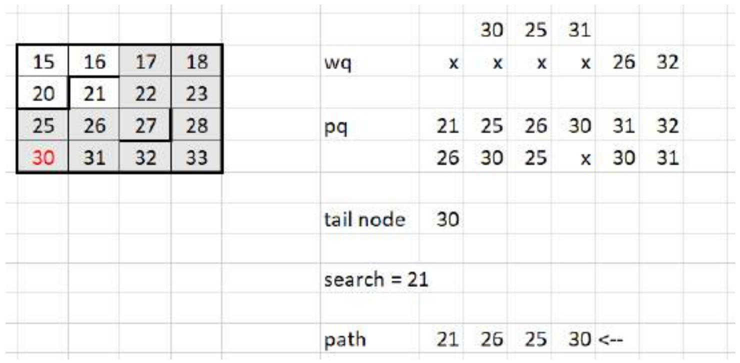
1.2.2.3.2.1.

1.2.3. Breadth-First Search

1.2.3.1. Breadth-First Search is a graph traversal algorithm that starts at a root node and explores all of its neighboring nodes at its current depth before moving on to nodes at the next depth. We chose to implement this into our search algorithm by using it when Depth-First Search is completed to find the shortest path to a tile that has not yet been visited. The Breadth-First Search uses a worker queue (FIFO) and a parent array. First, the current tile the robot is on when Breadth-First Search is called is enqueued to the worker queue. Next, all 4 adjacent tiles are checked to see if the robot can move there (no wall or black tile), and if that tile's parent queue index is not set. If both of these things are true, the tile is enqueued to the worker queue, and its index is set to the tile number of the tail node. After all 4 adjacent tiles are checked, the tail node is enqueued, and the process is repeated until the worker queue is empty or an unvisited tile is reached. If the worker queue is empty, that means all of the tiles in the maze have been visited. If it reaches an unvisited tile, then the parent queue is used to form the path by reading it from left to right starting at the index of the current tile the robot is on, finding the index with that index, and so on until the unvisited tile's index is reached. The path can be followed backwards to reach the unvisited tile.

1.2.3.2. Our rationale behind using Breadth-First Search for this purpose was that it would minimize the amount of revisits to tiles because it always finds the shortest path.

1.2.3.3. Developing the Breadth-First Search algorithm involved working out sample cases such as this one

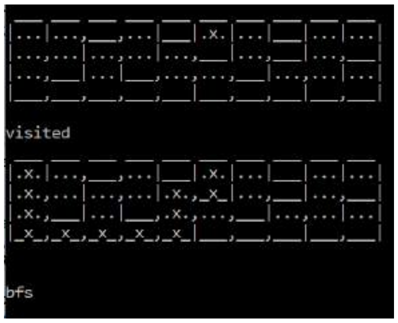


15	16	17	18
20	21	22	23
25	26	27	28
30	31	32	33

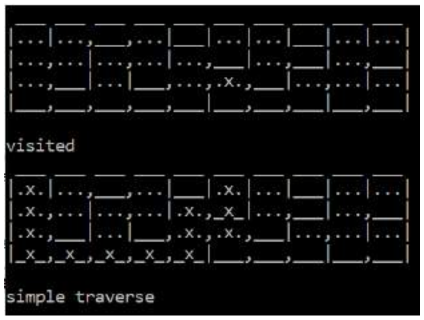
		30	25	31		
wq	x	x	x	x	26	32
pq	21	25	26	30	31	32
	26	30	25	x	30	31
tail node	30					
search = 21						
path	21	26	25	30 <--		

1.2.4. Testing

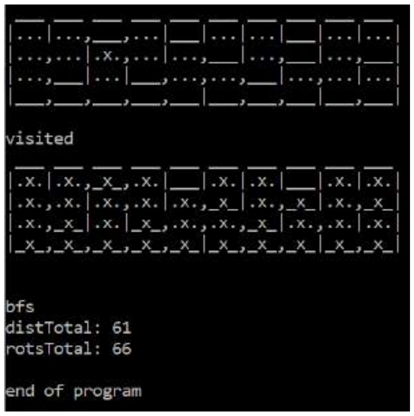
1.2.4.1. We tested the search algorithm by simulating the robot going through the maze that is typed out in a text file. The robot's movements are stimulated by updating the position of the robot on the map when the enter key is pressed. We also print whether it is doing the Depth-First Search or Breadth-First Search, and which tiles the robot has visited to ensure that the robot visited all tiles before finishing the program. We tested several different mazes to check the accuracy and efficiency of our search algorithm.



```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|...|...,___,...|___|.x.|...|___|...|...|
|...,...|...,...|...,___|...,___|...,___|
|...,___|...|___,...,...,___|...,...|...|
|___,___,___,___,___|___,___,___|___,___|

visited
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|.x.|...,___,...|___|.x.|...|___|...|...|
|.x.,...|...,...|.x.,_x_|...,___|...,___|
|.x.,___|...|___,.x.,...,___|...,...|...|
|_x_,_x_,_x_,_x_,_x_|___,___,___|___,___|


bfs
```



```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|...|...,___,...|___|...|...|___|...|...|
|...,...|...,...|...,___|...,___|...,___|
|...,___|...|___,...,.x.,___|...,...|...|
|___,___,___,___,___|___,___,___|___,___|

visited
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|.x.|...,___,...|___|.x.|...|___|...|...|
|.x.,...|...,...|.x.,_x_|...,___|...,___|
|.x.,___|...|___,.x.,.x.,___|...,...|...|
|_x_,_x_,_x_,_x_,_x_|___,___,___|___,___|

simple traverse
```



```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|...|...,___,...|___|...|...|___|...|...|
|...,...|.x.,...|...,___|...,___|...,___|
|...,___|...|___,...,...,___|...,...|...|
|___,___,___,___,___|___,___,___|___,___|

visited
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|.x.|.x.,_x_,.x.|___|.x.|.x.|___|.x.|.x.|
|.x.,.x.|.x.,.x.|.x.,_x_|.x.,_x_|.x.,_x_|
|.x.,_x_|.x.|_x_,.x.,.x.,_x_|.x.,.x.|.x.|
|_x_,_x_,_x_,_x_,_x_|_x_,_x_,_x_|_x_,_x_|


bfs
distTotal: 61
rotsTotal: 66

end of program
```

1.3. Letter Victim Identification

1.3.1. K-Nearest Neighbors

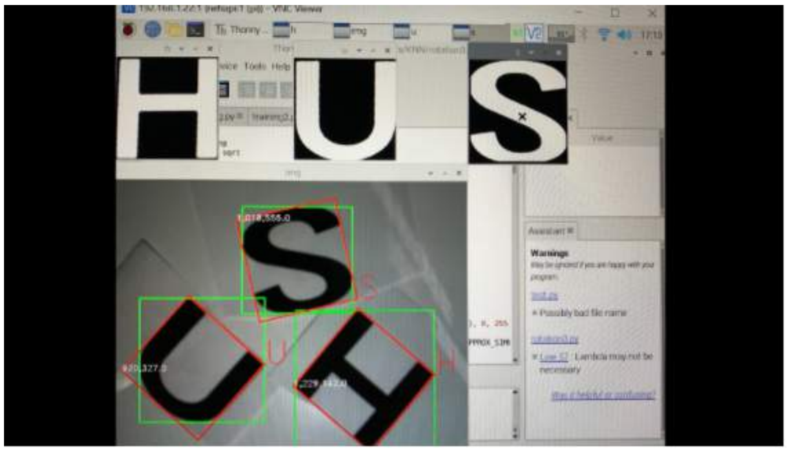
1.3.1.1. K-Nearest Neighbors is a supervised machine learning algorithm that outputs the most frequent classification in the k-nearest neighbors. We chose to implement this in our letter victim identification algorithm by training the KNN object from the OpenCV library using H, S, and U image data that was processed in the same way as the letter recognition code. We used a K value of 7 because we determined by printing out the neighbors returned by KNN that it provided the most accurate results for our data set. We use the output of the KNN function to identify if the image most closely resembles H, S, or U, and then take into account the distance, which is a measure of how different the image is from the training data, to determine if the image is actually the letter H, S, or U, or just resembles it. To choose a maximum distance value, we tested the code on all of the other letters of the alphabet because they relatively resemble H, S, or U.

1.3.2. Letter Recognition

1.3.2.1. Before passing the letter image into KNN, we needed to recognize and prepare it so that it is as close to the training data as possible. We perform a GaussianBlur to smooth out the image. We chose Gaussian Blur as opposed to the regular blur because since it does not smooth out edges as much, it is best for getting rid of noise from the camera. We convert the smoothed out image to grayscale and take the threshold because we want each pixel to be black or white as the color or grayscale value is irrelevant to classifying the letter. Then we find the contours with the RETR_EXTERNAL mode, which gives the outermost contour if there are nested contours, and CHAIN_APPROX_SIMPLE, which approximates the shape and saves memory. We filter out contours that are not between the minArea and maxArea which are values we predetermined by printing the area of the contour when we placed the robot next to the letter as close as it will get and as far as it will get while it remains in the tile. For the remaining contours, for each one at a time we call minBoundingRect and save the angle, since we do not assume that the letter victims are upright. We call boundingRect and use its returned position and dimensions to slice the image. We rotate the sliced image based on the angle we found earlier with padding so that the letter isn't cut off. Finally, we call KNN on the 4 possible orientations of the letter and consider the one with the lowest distance value.

1.3.3. Testing

1.3.3.1. To check that the image was being processed the way that we intended, we used imshow to display the thresholded image, contours, minBoundingRect and boundingRect, sliced image, and rotated image, and to check that we identified the letter correctly, the classification of the victim.



1.4. Color Victim Identification

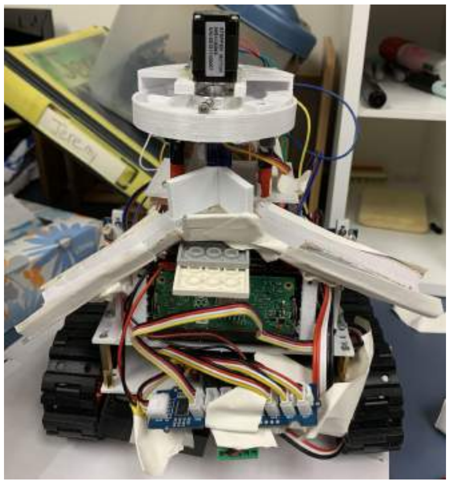
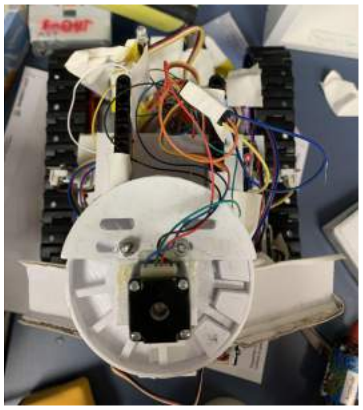
1.4.1. In our algorithm to detect color victims, we first approximated HSV (Hue Saturation Value) values, an alternative to RGB, by testing different values on the color victims. We then used the values as thresholds in the inRange function, which returns the pixels that are in the range of the thresholds. Using that, we find contours and if the area is above the minArea, we identify a color victim.

1.5. Serial Communication

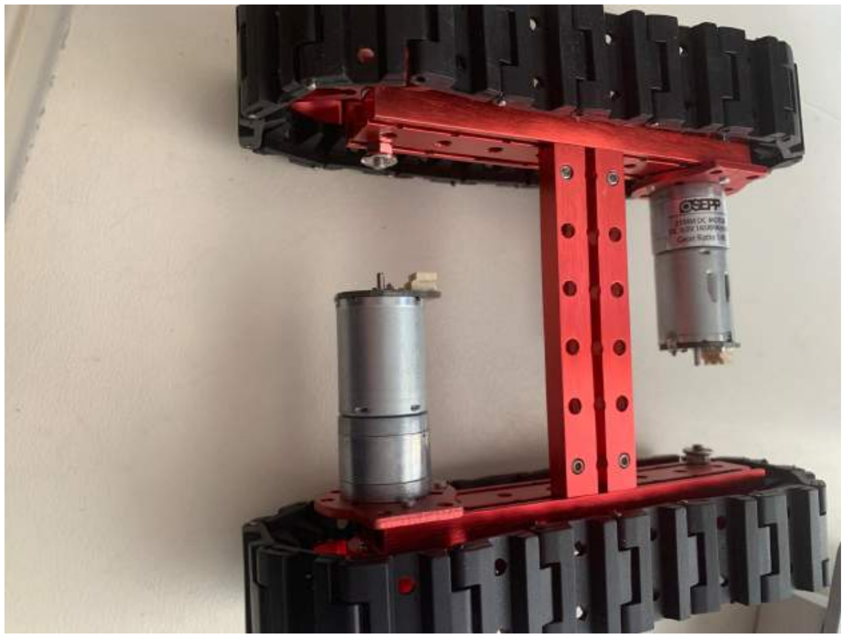
1.5.1. The robot's tasks are split between two boards or essentially “brains.” The StereoPi controls the Raspberry Pi cameras that are responsible for detecting letter and color victims. The MegaPi controls the TOF sensors, IMU, temperature sensors, DC motors, and the stepper motor. In order for the robot to perform all of these individual tasks together cohesively, the two boards will need to communicate with another. We chose to use the MegaPi as the main control of the robot because it already controls most of the sensor and motor functions; the Stereo Pi needs to send less information over to the MegaPi than the MegaPi would need to send to the Stereo Pi. We also wrote our depth first search and breadth first search algorithm in C rather than Python. We used serial communication to communicate between the two boards. Serial communication is the process of sending data, one bit at a time, over a communication channel. Whenever the Raspberry Pi cameras detect a letter or color victim, it sends a code character over to the arduino through serial communication. The MegaPi will then read in the character through the serial port and complete whatever tasks it needs to deliver the rescue kit(s) to the victims.

Hardware Design and Manufacturing

1.6. Overall Robot Design

1.6.1.  

1.6.2. Osepp Tank Mechanical Kit

1.6.2.1. 

1.6.2.2. We chose this for our chassis because the metal frame pieces and wheels provide a sturdy base for our robot.

1.6.2.3. We chose to use the treads so that the wheels do not slip on a smooth surface and so the robot can get enough traction to climb stairs and a ramp.

1.6.3. Two Tiers

1.6.3.1. Our robot has two tiers in order to fit the two batteries, the MegaPi, the StereoPi, and all of the sensors. The top plate is mounted to the bottom plate using metal standoffs.

1.6.3.2. We designed and 3D printed custom plates for size and hole placement.

1.6.3.2.1. The bottom plate has two temperature sensors on either side, a battery, and the IMU mounted to it. We chose to place the IMU on the bottom plate because it is less likely to move while driving as it is directly mounted to the metal base.

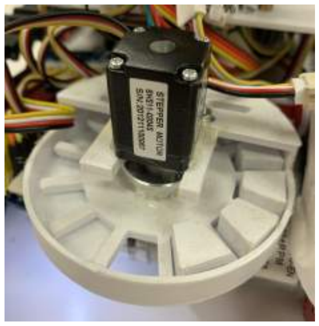
1.6.3.2.2. The top plate has two TOF sensors on either side facing the front, two TOF sensors on either side facing the sides, and two Raspberry Pi cameras facing the sides. It has a built-in vertical mount for the StereoPi that conserves space, and two triangular standoffs for the Raspberry Pi cameras to angle them downwards so they can see the visual victims above and below 7 cm.

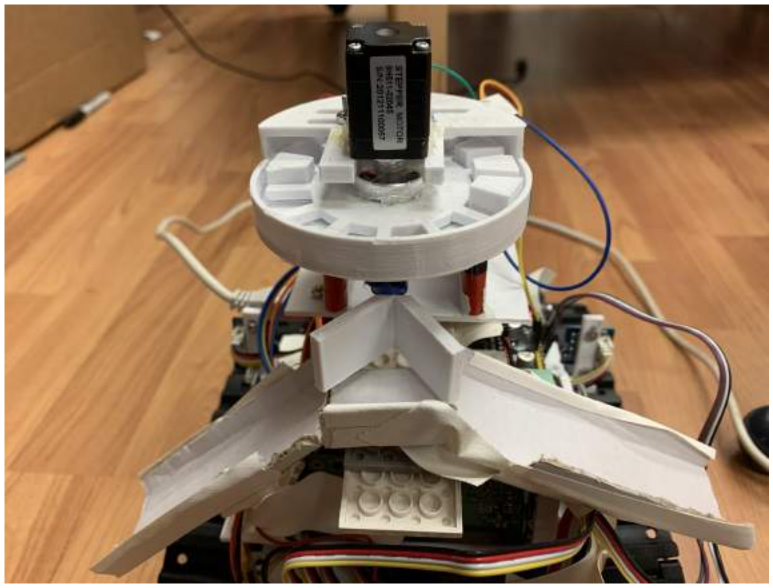
1.7. Sensor Mounts

1.7.1. We designed and 3D printed custom sensor mounts for the TOFs, temperature sensors, and Raspberry Pi cameras. We included holes in all of them to be able to bolt on the sensor to the mount and the mount to one of the plates or robot base.

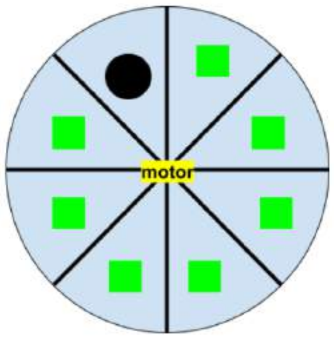
1.8. Rescue Kit Dispenser

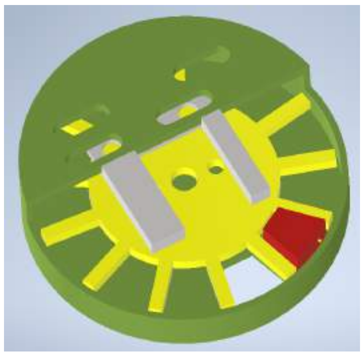
1.8.1. Overall





1.8.1.1. We designed the Spinner and Base in CAD based off of our initial idea sketch.





1.8.1.2. How our Rescue Kit Dispenser works is that before starting the robot, the 12 rescue kits are placed in 12 out of the 13 slots. When the spinner is rotated 28 degrees by a stepper Motor, one rescue kit falls to the ground because it becomes positioned above a hole. We are able to control which side the rescue kit falls on by rotating the robot.

1.8.1.3. We chose this design because it is compact, easy to control how many rescue kits are dropped, and consistent. We also initially created a plastic prototype powered by an EV3 motor that successfully dispensed a rescue kit-like object.

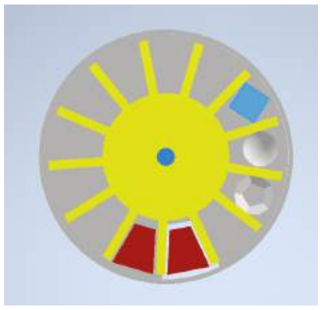
1.8.2. Spinner and Base

1.8.2.1. The Spinner has 13 slots so that it can hold 12 rescue kits since 1 slot will start positioned above the hole.

1.8.2.2. The Base has a partial cover as a way to mount a motor to the spinner from above, but not a full cover so that we can easily load our Rescue Kits into it. It has a hole in the bottom that is bigger than the rescue kit to ensure that the rescue kit does not get stuck, but not too big to ensure that the adjacent rescue kit does not fall prematurely.

1.8.3. Rescue Kit

1.8.3.1. We chose a 1 cm3 trapezoidal prism shape for our Rescue Kit because it fits in the slot of our spinner and does not roll away when it hits the ground. We were able to determine that this shape fits best by placing other shapes, including a sphere, cube, and dodecahedron into our CAD model.

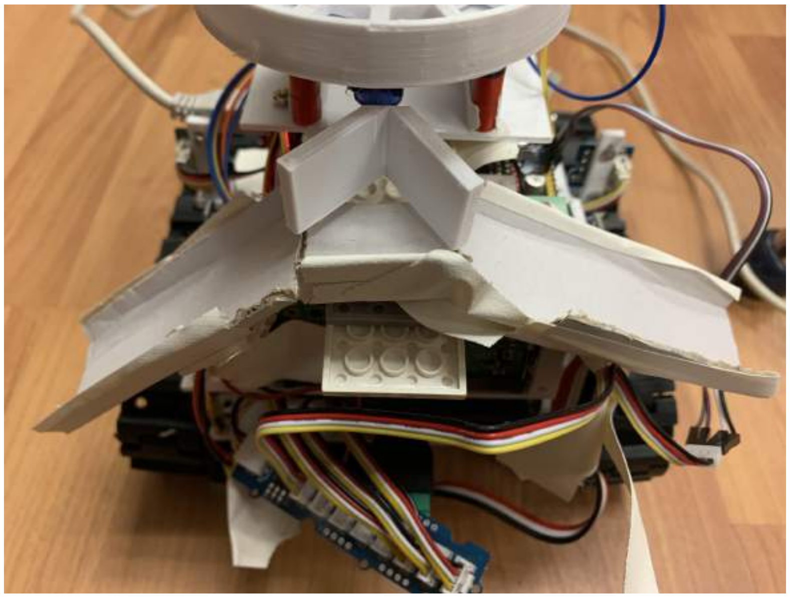


1.8.4. Stepper Motor

1.8.4.1. Our Rescue Kit Dispenser is controlled by a stepper motor with 200 steps.

1.8.4.1.1. We chose to use a stepper motor because we need the Spinner to move a certain amount each time to drop one Rescue Kit, which we can do by telling the stepper motor to move a certain number of steps.

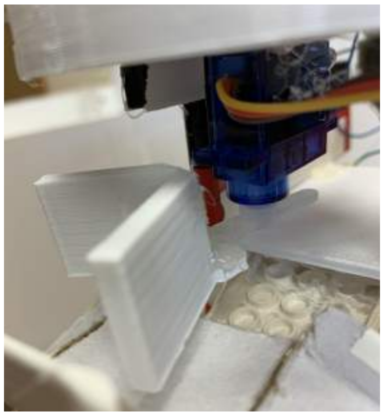
1.8.5. Ramps

1.8.5.1. 

1.8.5.2. The ramps going down each side of the robot serve to drop the Rescue Kit on the side where the victim is located. It is attached to a platform where the Rescue Kit lands after it is dispensed.

1.8.5.2.1. We chose to construct the ramps out of cardboard because our Rescue Kits are plastic and slide down the surface smoothly.

1.8.6. Servo

1.8.6.1. 

1.8.6.2. The Servo controls which ramp the Rescue Kit slides down. It is attached to a 3D printed V-shaped piece that pushes the Rescue Kit left or right, depending on which side the victim is on.

- 1.8.6.2.1. We chose to incorporate this mechanism to save time while dispensing the Rescue Kits, as it allows us to dispense them on the correct side without turning the entire robot.
- 1.8.6.2.2. We chose to use a 180 Servo because it is compact and we can control it by setting it to the position that we want.
- 1.8.6.2.3. To determine the best angle of the V-shaped piece, we first made a prototype out of cardboard.
  - 1.8.6.2.3.1.

1.9. DC Motors with Encoder

1.9.1. Even with a perfect depth first search algorithm, the robot would still be blind in the maze if it does not know which tile it is on in the maze. In order to travel from one tile to the next accurately, without moving too much or too little, we decided to use motor encoders to tell the robot how far to move.

- 1.9.1.1. We first considered figuring out the exact time it would take for the robot to move one tile and run the motors at a certain speed for a certain amount of time; however, we concluded that this approach would be unreliable since the robot's speed may vary based on how fully charged the battery is. A motor encoder provides a more reliable method of tracking how far a robot has moved since the number of rotations of the motor is less prone to unpredictable factors.

1.10. Time of Flight Sensors

1.10.1. Our robot has a total of 6 time of flight (TOF) sensors. Time of Flight sensors return a value that represents the distance from the robot to an object. The greater the value is, the farther away the sensor is from the object. Two time flight sensors were mounted on the front and sides of the robot.

1.10.1.1. The time of flight sensors serve multiple purposes. An important function of the sensors to sense if the robot is in front of or adjacent to a wall. This information is important because it is needed for the robot to create a map of the maze that is used for navigation (DFS and BFS); whether or not there is a wall in front of the robot will affect whether it moves forward or the direction in which it turns. For instance, if there is a wall in front of the robot but no walls on either side, the robot would turn to the right rather than continuing forward. If there were walls on either side of the robot but none in front, then the robot would move forward. If there were walls in front of and two the sides of the robot, it would turn 180º around. (For this reason, we decided we did not need any TOF sensors on the back of the robot). Only one time of flight sensors on each side are necessary to detect walls, the reason why there are two tof sensors on each side is to make sure the robot is properly oriented. The two TOF sensors on each side of the robot are spaced so that a difference in measurements from the two sensors can be used to tell when the robot may be crooked. The two sensors should read a similar value while the robot is still or moving forward (not turning), and so a difference in values would signal that one sensor is closer to the wall than the other, or in other words, one end of the robot is closer to the wall than the other. This benefit from having two TOF sensors is also useful in sensing obstacles, which is why there are also two tof sensors on the front of the robot. The TOF sensors will help the robot determine if there is an obstacle in front of it and which way to turn in order to avoid the obstacle.

1.11. IMU (Gyro)

1.11.1. Figuring out how to get the robot to make accurate 90º and 180º turns was an important issue we had to tackle. We came up with a few solutions for this issue such as using the motor encoder but we ultimately decided that an IMU would provide the most accurate solution after several tests. We used the gyroscope on the IMU to read the angle that the robot turns. At one point during testing, the sensor did not provide accurate readings, and the robot constantly turned too much or not enough. In an effort to narrow down the bug, one of the things we tried was disabling the magnetometer. We suspected that the motors with encoders and the battery, which were in close proximity to the IMU, might have interfered with the sensor readings, which are rooted in magnetic field strength (relative to Earth's magnetic north).

1.12. Stereo Pi and Raspberry Pi Cameras

1.12.1. We decided to mount two Raspberry Pi cameras on either side of our robot to detect color and letter victims.

1.12.1.1. We decided that having two cameras was a more efficient solution than having just one camera and rotating our robot at each tile or having one camera mounted on a servo motor that can be rotated. We originally planned on using one Raspberry Pi camera and one Arducam, a USB camera which is compatible with the Raspberry Pi. Unfortunately, we encountered several difficulties with the Arducam when testing the cameras together after switching to a battery as a power source (rather than the wall outlet). The port numbers of the Raspberry Pi camera and Arducam were jumbled and unpredictable. We were eventually able to open the Raspberry Pi camera successfully, but we were unable to open the Arducam. As a result, we decided it would be best to use two Raspberry Pi cameras rather than one Raspberry Pi camera and one Arducam. In order to do this, we could not use our Raspberry Pi because it only had one port for a Raspberry Pi camera. Instead, we needed to use a StereoPi, which has two ports for Raspberry Pi cameras.

1.13. MegaPi

1.13.1. Many of the sensors we used, such as the TOF sensors, IMU, and temperature sensors are Arduino sensors. We used a MegaPi in order to run these sensors, but also because it can be connected to and communicate with the StereoPi (through a process called Serial Communication). Though the MegaPi does not have enough I2C ports to support all our sensors, and because multiple of our sensors had the same address, we used an I2C Multiplexer.

1.14. Melexis Temperature Sensors

1.14.1. We have two temperature sensors on each side of our robot to detect heat victims. We used Melexis temperature sensors, which are non-contact infrared sensors that can obtain the temperature of an object.

1.15. Logic Level Shifter

1.15.1. We attached the temperature sensors to the MegaPi through a logic level shifter, and used the StereoPi to supply the 3V.

1.15.1.1. We did this because the Melexis temperature sensors we used were the 3V versions, and so we could not connect them directly to the MegaPi because the board could only supply 5V, which may burn out the sensors. The StereoPi has a 3V pin unlike the MegaPi.

1.16. Color Sensor

1.16.1. We used an RGB color sensor to detect when the robot is on black. It is mounted at the very front of our robot so that the robot quickly recognizes the black tile before it goes more than half way onto it.

1.17. Testing

1.17.1. To test and debug the sensors we used, we printed their values to the Serial Monitor in separate programs and in the main code. We also printed error messages when the Arduino could not communicate with the sensor, which indicated to us that there is a problem, such as mixed up wiring.

Conclusion

This paper presents what our team was able to accomplish for the RoboCupJunior Rescue Maze 2021 Competition. However, it is important to recognize the constraints we faced and improvements we would like to make in the future. COVID-19 was a major constraint for our team as we were unable to regularly meet in person until April. This delayed our hardware construction, which in turn delayed the integration process. In addition, the hardware design process was lengthier than expected due to a limitation of experience; we ran into challenges where we had to experiment with certain hardware components because they did not perform as well as we expected or were not compatible with our other components. In the future, our main goal is to improve the accuracy of our robot's movements, such as going forward a certain amount and turning a certain number of degrees, while also improving the speed of these movements.