# Navigating a Maze with Visual Victims

Julian Lee and Jeffrey Cheng

Storming Robots, Branchburg NJ, USA

## 1. Introduction

### 1.1. Background

In today's world, it is not uncommon to hear of tragic accidents, such as collapsed buildings, that leave humans stranded. These accidents often make the environment too dangerous for other humans to stage a rescue attempt. This could be due to unstable infrastructure, harmful chemicals in the air, and other life-threatening conditions. In these scenarios, it would be extremely helpful to look toward an alternative source of rescue: robots. If a robot could successfully enter the dangerous environment, rescue or provide aid to those stranded, and return to where it was deployed, no other human lives would be risked, and many victims could be saved. The RoboCup Junior intends to target this goal with its maze competition.

### 1.2. The Task

In the RoboCup Junior maze competition, a robot must autonomously navigate through a maze composed of floors separated by ramps. Each floor is composed of uniform square tiles. The robot is tasked with detecting "visual victims" represented by the letters "H", "S", and "U", which are scattered throughout the maze, so the robot should navigate through every tile in the maze rather than finding a single path through the maze. In this paper, we discuss an algorithm to efficiently navigate every tile of this multidimensional maze as well as two approaches for fast visual victim detection.

## 2. The Algorithm

### 2.1. Defining the Maze

The program must store attributes of each tile including the location of the four walls. To conserve memory, each tile is defined as a byte while each attribute of the tile is defined as a bit - the first 4 bits correspond to whether there is a north, west, south, or east wall respectively. The maze is stored as a one-dimensional array where a specific tile with coordinates (x,y) on floor z corresponds to the index z * length of array * width of array + y * width of array + x.

## 2.2. Section and Orientation

Our robot begins its journey at the center of our map facing in an arbitrary direction. It begins in the middle of the maze so that it can expand in any direction without having to be predefined. We have one variable to keep track which tile we are in and what direction we are facing, and both are updated every time we turn or move between tiles.

## 2.3. Single Floor Navigation

### 2.3.1. Traversal

Our robot begins in a simple traversal algorithm. It moves around the maze with a priority in moving forward as we want to minimize turns to save time and avoid turning errors. Once it cannot move forward, it tries to turn right, then left. It is programmed not to move into already visited tiles to save time and avoid redundancy. However, if the robot has no option of moving into an unvisited tile, a dead end has been reached.

### 2.3.2. Dead End

When a dead end is reached, the robot must consult a more complicated algorithm. We want our robot to find the path to the closest unvisited tile to avoid unnecessarily long paths throughout the maze. To do so, we used a Breadth First Search algorithm. We treated each tile of the maze as a node in a tree, where a parent's children would be a tile's adjacent nodes and a goal node would be any unvisited tile. Our Breadth First Search would then return the shortest path to the closest unvisited tile, and our robot would traverse accordingly.

## 2.4. Multiple Floor Navigation

### 2.4.1. Map Storage

When our robot reaches a different floor, it must also store that floor's data in the map to successfully navigate the floor. However, if the second floor is simply treated as an extension of the first floor, it is likely that information from the first floor will be overwritten if the second floor overlaps the first floor. To compensate for this, we added multiple floors to the map, where a sections whose indices vary by exactly one floor correspond to tiles that are directly above/below each other on the maze. This way, the robot could once again traverse in any direction and never overlap with the first floor.

### 2.4.2. Section Count and Ramp

If there are multiple paths to arrive at a floor, we have to keep our section count accurate. This is so that if we land back on a certain floor, we know we are back to that floor and not a different one. To do so, the section difference between each floor is exactly one full field and the length of the ramp. The difference of a full field ensures that the two separate floors will not overlap in our map, and accounting for the ramp length ensures that if two ramps are of different length, our robot will change our section count accurately.

### 2.4.3. Jumps

Additional ramp tiles are added to the end of the maze array to function as a corridor. We added an attribute, a "jump", which indicates that the current tile can lead to a corridor, separate from the rest of the maze. This corridor represents a ramp. At the end of each corridor, the last tile also has a "jump" attribute which signals that the tile connects to the first tile of the next floor. Accounting for the length of the ramp through the number of tiles in the corridor allows the Breadth First Search algorithm to correctly determine the shortest path to the next tile. A dictionary between tile indices is used to set up jumps between tiles.

**2.4.3.1.** As an extension of the algorithm to make the algorithm function in the case of multiple ramps leading to a single tile, this dictionary should also keep track of the direction of the jump.

## 2.5. Navigating Back to the Start

### 2.5.1. One Floor

After our robot finishes with the maze or we begin to run out of time, we want to return to the beginning. To do so, we rely, once again, on our Breadth First Search. Our Breadth First Search, however, works a little bit differently in this scenario. It only treats adjacent tiles that have already been visited as children to a parent node. This is because we are unsure of the wall situation in unvisited tiles, which could lead to path calculation problems. When returning back to the start, our only goal node is the starting tile, and the Breadth First Search will return the shortest path to the start.

**2.5.1.1.** As the algorithm provides the exact number of tiles that need to be navigated to return to the start, the algorithm can also approximate the time it takes to return to the start. This is practical for the competition where there is a timed run.

#### 2.5.2. Separate Floor

In the event that our robot is on a different floor than it started in, it must find its way to a ramp to lead it back to the beginning floor. Once again, we run our Breadth First Search, but this time, it utilizes the jump attribute of each tile. If the algorithm hits a tile with a jump attribute, we consult our C++ map, which will indicate a jump from the current tile to the beginning of the ramp, and that tile is used as a child node to a parent node. The tile at the end of the ramp also has a jump attribute, which has a jump attribute to the next floor. As a result, our Breadth First Search can reach different floors and still find the shortest path.

### 2.6. Unvisitable Tiles

Occasionally, some tiles have four walls surrounding it, and are thus unreachable. Our program is designed to navigate every tile of the maze until it attempts to return to the start. However, to compensate for unvisitable tiles, we consulted our Breadth First Search again. If and only if our Breadth First Search cannot find an unvisited tile to go to, then we will return to the start.

### 2.7. Alternative Algorithms

A different approach we could have taken would be to redesign our map. Our current map is an array that uses more memory than necessary to avoid presetting the dimensions of the maze and the starting tile (the robot currently starts at the center of an extremely large maze so that it can move many tiles in any direction). An alternative would be to create a structure for each node and have pointers to each of its adjacent nodes. This would allow us to allocate exactly the number of tiles we needed, because a new structure would just be allocated for every new tile we traverse to. This would also allow us to have as many attributes of whatever type we want, instead of being limited to 8 bits. However, this would not work in a scenario where there are multiple paths to a floor: when the robot returns to its original floor through the use of a different ramp after navigating through the other three floors, it will treat the original floor as a new floor (Fig 1).

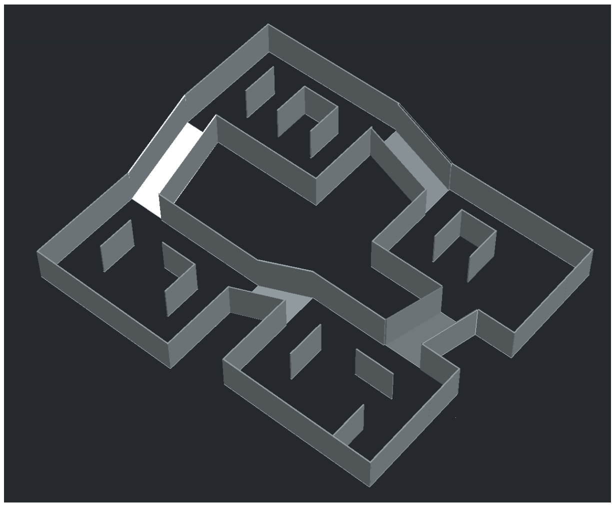

**Fig. 1.** Maze with 4 floors.

## 3. Visual Victims

### 3.1. Hardware

We are using a StereoPi, which allows us to attach two pi cams to the robot. We are using the Waveshare RPi Camera (G) as its fisheye lens maximizes the field of view of the cameras.

### 3.2. Finding Potential Contours of Interest

To find the letters within an image, the image is first thresholded. The threshold is determined by taking the average of the darkest 5% of pixels when the camera was facing a white wall, and this value is averaged with the darkest 5% of pixels when the camera faces a letter. The algorithm then uses the OpenCV findContours function to find potential letters in the image.

#### 3.2.1. Avoiding False Detection

Contours that touch the edge of the image are disregarded: these contours either represent noise from the maze, such as walls in different tiles of the maze or letters that have been cut off, in which case the algorithm is likely to incorrectly classify the letter as an “H”, “S”, or “U”. Contours must meet a specific area requirement, and the height to width ratio of the minimum enclosing rectangle (a rotated rectangle) must also be between specific values.

### 3.3. Extracting Regions of Interest

As letters could be tilted at any angle or even upside down, a rotation needed to be applied before determining the correct letter. After this step, the letter is either upside right or rotated 180° (upside down). The next step of the algorithm can account for both these situations.

### 3.3.1. Rotations and Cropping

The algorithm bounds the contours of interest through the use of the OpenCV minAreaRect function, which provides a rotated rectangle representing the desired region of interest. A rotation matrix is created based on the angle and center of the rotated rectangle. In the case that applying a rotation on the frame based on the angle of the rotated rectangle would result in a frame whose height is less than its width (a sideways letter), 90° is added to the angle of the rotated rectangle prior to creating the rotation matrix. This ensures the letters will not be sideways. The rotation matrix is then applied to the frame using the OpenCV warpAffine function. Rotating the points of the rotated rectangle by the same rotation matrix provides the points for the bounding rectangle of the newly rotated contour. The image is then cropped based on this bounding rectangle.

### 3.3.2. Limitations

If the letter's width appears greater than the letter's height in the image either due to a distorted image or physically distorted letters, the algorithm may detect the wrong letter, although it is more likely to fail to detect the letter altogether. To fix this issue, it is necessary to extend the solutions to determine the correct letter detailed below such that they are also applicable for sideways letters.

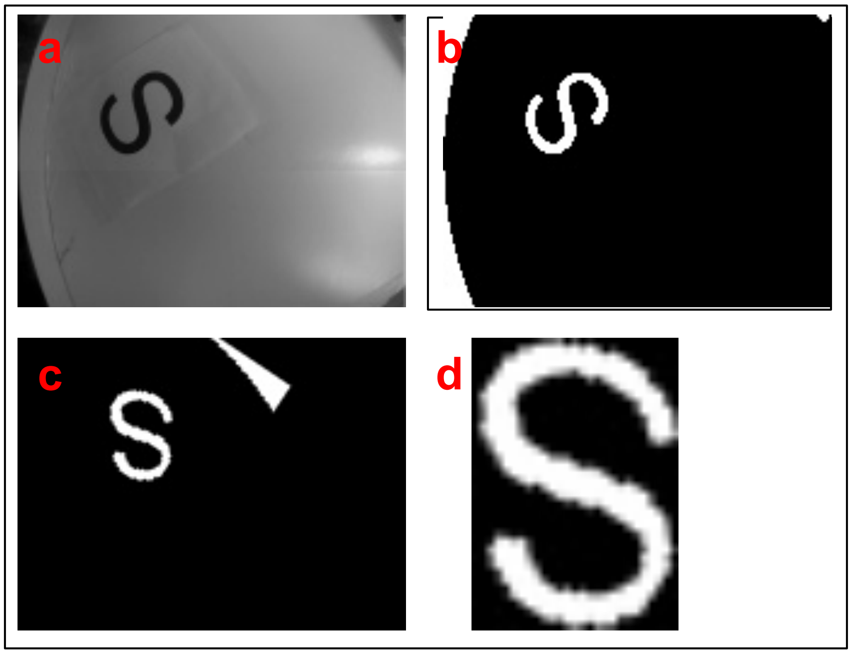

**Fig. 2.** a) Grayscale image captured from the picam. b) Thresholded image. c) Rotated image. d) Cropped image.

## 3.4. Determining the Correct Letter

Two potential methods were examined to determine the correct letter, both which involve dividing the image into regions. As an “H” and “S” appear the same once rotated 180°, the only upside down letter that the algorithm needs to account for in a separate case is an upside down “U”.

### 3.4.1. Using Contours

This method involves looking at sections of the frame and counting the number of contours in each section. To avoid very small regions created by noise from being counted as a contour, only contours that are larger than one-fourth of the largest contour in the section are counted. The algorithm first examines the left half and right half of the frame. If both the left half and right half of the frame have two contours, the letter must be an S (Fig. 3). In order to differentiate between a “U” and an “H”, the algorithm then counts the number of contours in the top third and bottom third of the image. If there are two contours in both these regions, the letter must be an “H” (Fig. 4). Otherwise, the letter is a “U”.

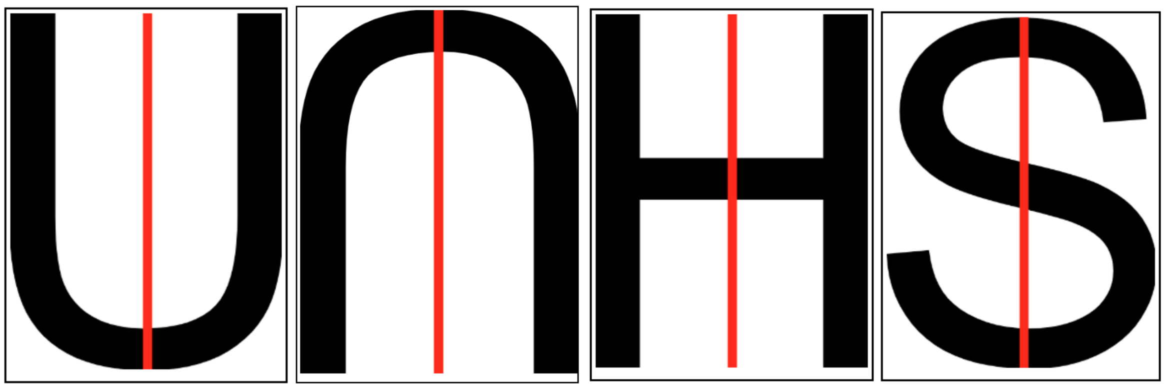

**Fig. 3.** Note that a "U" and an "H" has one contour on each side while the "S" has two contours on either side.

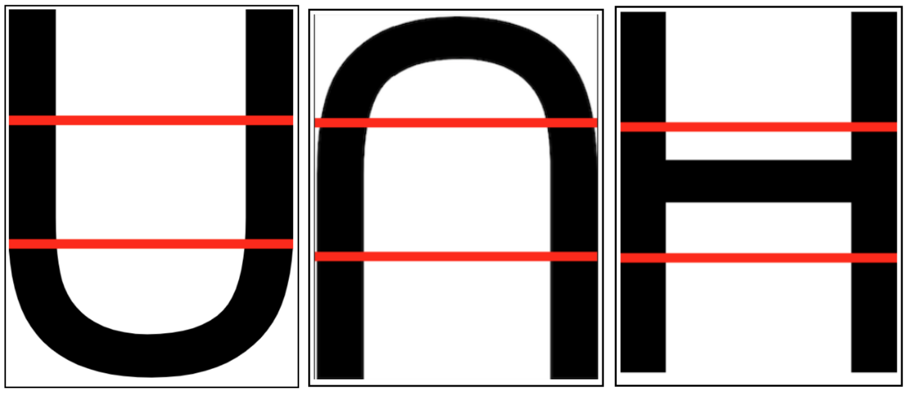

**Fig. 4.** Note that an "H" always has two contours in the top third and bottom third of the image while the "U" has one contour in either the top third or bottom third of the image

**3.4.2. Finding the "fill" of the image** This method looks at the percentage of black pixels in specific regions of the image rather than counting the contours in each region. The rightmost and leftmost fifth of the image for an "H" and a "U" have a much higher percentage of black pixels compared to an "S" (Fig. 5). The algorithm then differentiates between an "H" and a "U" by examining the middle region: the "H" has a higher percentage of black pixels than the "U" in this section (Fig. 6). In some fonts, the bridge for the "H" can be fairly narrow which can make it difficult to differentiate between the "U" and "H", so more calibration is required. Warped letters due to the fisheye lens being used can lower the fill of the rightmost and leftmost fifth of the image for the letters "H" and "U", so this could also be problematic. For this reason, the contours method for differentiating between letters is preferable.

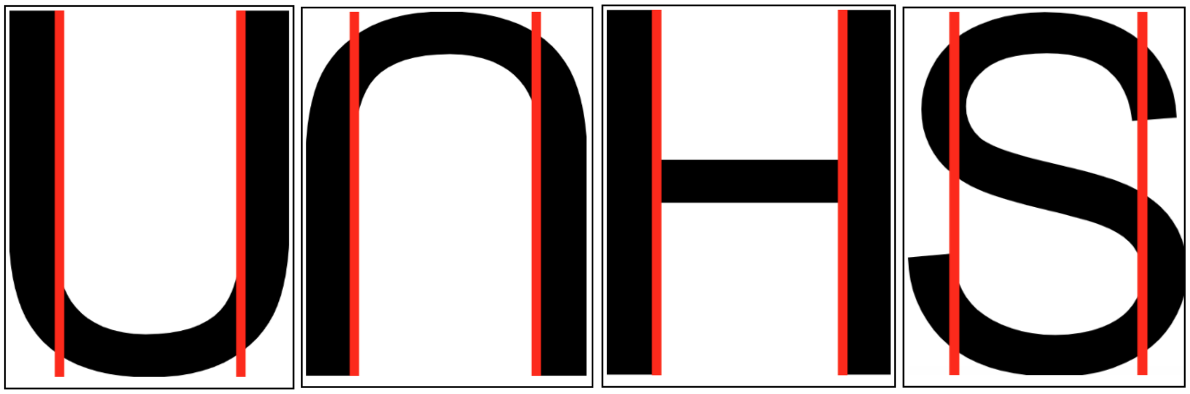
**Fig. 5.** The letters "H" and "U" have an 80-100% fill on the rightmost fifth and the leftmost fifth of the image while these sections have under a 60% fill for the letter "S".

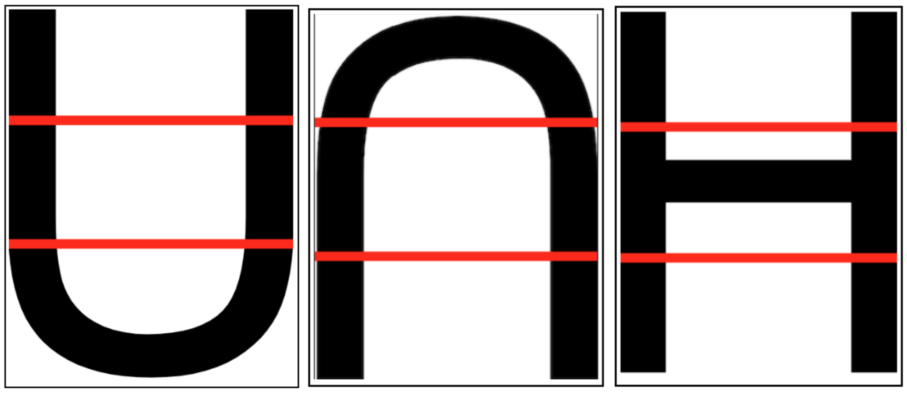

**Fig. 6.** The letter "H" has a much higher "fill" in the middle section of the image relative to the letter "U".

## 4. Conclusion

We have discussed algorithms to solve two challenges presented in the RoboCup Junior Maze. One algorithm allows for successful complete navigation of a multi-floor maze while maintaining a map to allow for calculating the fastest path back to the start of the maze. The second part of the paper discusses an algorithm with two variations for quick letter recognition using solely the OpenCV library. Future improvements can be made for both algorithms: the memory required for the navigation algorithm can be reduced by only allocating bytes to tiles that exist, and the letter detection algorithm can be modified to account for more cases of camera distortions and distorted letters. We hope to implement some of these improvements in the future.