RoboCupJunior Rescue (Maze) 2023

Team Description Paper

ED Robotics

Abstract

This paper describes our robot, as well as the development process of it. The robot is a junior rescue maze robot capable of completing most tasks that are within the rules. It is a four-wheeled encoder-motor driven robot controlled by a Raspberry Pi and a Makeblock Me Auriga (Arduino Mega). Positioning is done using two ultrasonic sensors on each side and one in front, and the rescue kit mechanism is in the back. For detecting victims, the robot uses a camera looking down on two mirrors, which split the image into two, enabling us to see both walls simultaneously. The strong sides of the robot, besides being cute, are advanced pose estimation, aligning itself well to the maze, optimization of memory usage and a reliable rescue kit mechanism.

1. Introduction

a. Team

The name of our team is ED Robotics, and our members and their roles are as follows:

QUAK123 – Responsible for rescue kit mechanism. Prior experiences include programming games, mobile and web applications. Worked on CAD projects in school.

apollen85 – Responsible for the motion system. This includes the hardware design (excluding the rescue kit mechanism), electronics and local maze navigation (for example turning exactly 90^{O}). Experience was gained through experimenting with simple Arduino projects and doing CAD projects at home and in school, but organized and complicated projects like this one have not been attempted before. Programming was self-taught.

c4tm4n/"bohrmeister" – Responsible for the vision system and the raspberry Pi. This includes detecting coloured and visual victims and handling the raspberry Pi. Past experiences include python programming and Linux skills.

MrK – Responsible for maze navigation and mapping. This is also the master code which communicates with the other programs, and it also navigates the robot through the maze, maps the maze, and gets the robot back to the start tile. Had previous experience in programming, both from self-learning, for instance from making games, and from school.

2. Project Planning

a. Overall Project Plan

As this is our first extensive and collaborative project, our overall aim for the competition is learning about collaboration and robotics as well as building a well-performing robot.

During the early stages of development, not much of a plan existed. However, we realised that we needed to spend much time as early as possible. Later on, our work became more organized. A schedule was made and evaluations were made throughout. Different roles were assigned and defined, which enabled us to work more efficiently. The plan was to be mostly done with the robot a few weeks before the competition to give time for extensive testing and bug-fixing.

Some of our milestones have been to get the components to work independently, so that the robot could be tested in its complete form. Other than that, not many milestones were set.

As we have no past performances, these have not influenced our development process. Inspiration has been taken from other teams in designing our solutions, but a lot has been done without any prior research or experience.

b. Integration Plan

The system is divided into many parts. Processing is done by a Raspberry Pi, running navigation code and vision code, and an Arduino, running low-level code. These communicate via serial communication and communication between programs on the same Pi is done via socket communication. The low-level code handles all hardware interaction, which mainly consists of driving the four encoder motors, reading ultrasonic distance data, reading gyro data and controlling lights. The rescue kit mechanism is on its own removable plate giving modularity, and the only electric connection is the servo wires.

3. Hardware

The robot is a four-wheeled robot built on a baseplate, on which everything rests. The two arcs support sensors, computers and more and the battery has been mounted on the underside to lower the centre of gravity. A large open space in the middle of the robot on the top enables us to have a dual-mirror system, through which the camera on top can see the wall on both sides of the robot simultaneously. This has forced us to squish everything out of the field of view of the camera, which has proven to be difficult at times. To make assembly easier, we use M4 screws and nuts for everything big enough and M2 for everything smaller. Electrical connections have been made using either direct soldering, dupont-connectors or other connectors specific to the sensors/components used. The dupont-connectors have been secured, for example by using hot glue, to prevent accidental disconnections.

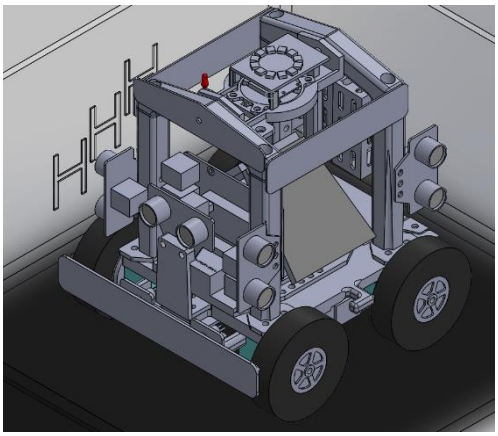

Fig. 3.1 – CAD model of the robot and surrounding maze

a. Mechanical Design and Manufacturing

The main structure of the robot consists of 3D-printed parts. Everything is connected by the baseplate. From the baseplate, 2 arcs come out supporting the sideways-facing ultrasonic sensors and leaving the middle open for the dual-mirror system. The top of each arc is removable to facilitate access to components under the mirrors, which are held in place partly by interfacing with the board below.

The power train consists of four encoder motors from makeblock. The wheels are neoprene foam tires with a 3D-printed mounting adaptor attaching it to the motor, but textured silicone tires are available as a backup. The size of the tires was decided by adjusting the size in our CAD model and then, using the appropriate constraints/mates, the ground clearance was checked when driving onto and leaving ramps from above and below. For ground clearance regarding speedbumps, simple calculations were made to calculate the distance from the ground to the lowest point of the robot. Placement of the motors and thereby wheels was made to minimize skidding while turning (put close together along the length of the robot to minimize lateral movement), while also leaving enough space below for the battery and not making the robot too wide. For the outer part of the backup wheels (the tire), some different materials were tested. Slick rings worked alright when clean, but they got dirty quickly and lost grip. Some thicker and wider tires with a more pronounced structure (our backup) were also tested and yielded

better results. Mitigations for the wheels sliding along speedbumps at shallow angles include perforated edges, software fixes and potentially protrusions on the sides of the wheels.

The main concept of the rescue kit mechanism is to drop rescue-kits by pushing them over the edge of the robot, using a racket and pinion mechanism. The rescue kits are stacked on top of each other and stored in a container. The container is designed in a robust manner to withstand bumps in the case of the robot overturning and a support for the container can be applied to make it less fragile while being transported. The entire rescue kit mechanism is built on its own plate that then is mounted on the main baseplate of the robot. This is done intentionally to avoid having to print the whole baseplate of the robot when making small changes that only affect the rescue kit mechanism, thus saving time and materials.

One aspect of our robot which we believe to be fairly unusual is the dual-mirror system. It is our chosen solution to the problem that the Raspberry Pi only has one camera port, while we want to see both walls simultaneously. Because we had the hardware for the mirror system and did not know about the existence of dual-camera solutions, we went with the mirror system. Other considerations like the extra processing power needed to process two images were also taken into account. The size, positioning, shape and angle of the mirrors were determined by trial and error. In the CAD model, lines were drawn from the camera lens centre to the vertices (corners) of the mirror, to the wall of the maze the robot was positioned in.

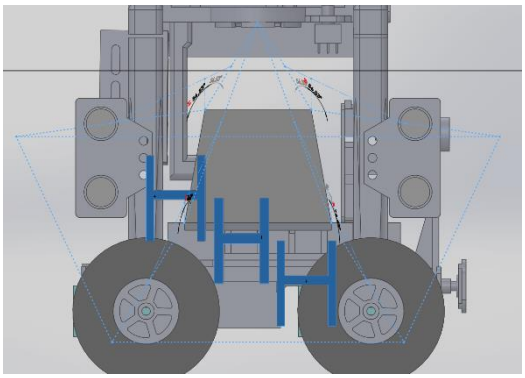

Fig. 3.a.1 – Camera FOV

In order to satisfy the law of reflection, constraints were added to set the angle between the outgoing line from the camera and the line normal to the mirror and the angle between the outgoing line from the mirror and the line normal to the mirror to equal values. This resulted in 4 intersection points between the outgoing lines and the maze wall. When connected by lines, the area created forms the FOV (Field Of View) of the camera. As mentioned earlier, the characteristics of the mirrors were changed to get an optimal view of the wall and the victims painted thereon. To make this process easier, victims were drawn on the maze wall in the CAD software at the ideal height as well as on the maximum and minimum height, ensuring that all legitimate victim positions were covered.

b. Electronic Design and Manufacturing

The computing power of the robot is provided by a Raspberry Pi 4 and a Makeblock Me Auriga, which is a modified Arduino Mega. The Pi is the main brain of the robot, responsible for maze navigation and vision processing due to its greater processing capabilities, while the Auriga handles sensor interfacing and maze navigation from tile to tile due to it being easier to integrate with the hardware and it giving the precise timings needed for measuring distances with ultrasonic sensors or reading encoders accurately, for example. Both of these boards are powered by a single 1800mAh 3-cell (3S) LiPo battery through two voltage regulators (XL4015) connected to the battery in parallel. The Pi receives its power through a cut USB-C cord with a voltage of 5,1V (as per the specification of the Pi) and the Auriga is connected by a barrel connector supplying 9V (specification says 6-12V). The reason for powering the Auriga with 9V instead of 5V like the raspberry is that the motors need the higher voltage for operation and the board cannot step the voltage up, only down. Both of these voltages are regulated down from the 10-12,6V which the battery can supply (10V is our safety minimum at roughly

3,33V/cell). The battery was placed under the baseplate to keep the centre of gravity low, aiding in the stability of the robot. Communication between the boards is done by serial communication but is complicated by that the Auriga runs on 5V logic and the Pi on 3,3V logic. This is solved by having a logic level shifter between the connections, which can convert logic signals between two given voltages (5 and 3.3V in this case).

The robot has several sensors. For driving in alignment with the maze, it has encoders on the motors, ultrasonic distance measurement sensors and a gyroscope. Each wheel has an encoder built in and there is one gyroscope built into the computer responsible for the low-level tasks. The encoders are the default distance measurement sensors for knowing if the robot has driven a "step" (30cm, one tile) but due to their unreliability regarding tire slippage and possible debris, other methods later described are used in conjunction with the encoders. The ultrasonic sensors are used to align the robot to the walls of the maze. There are two on each side, meaning that even with just one wall being present we can calculate the sideways position and angle relative to the wall (fig. 4.a.1). However, the ultrasonic sensors were observed to be noisy (values jumping up and down), so to smooth the values out, a running average for the measurement of each sensor is kept. A sensor in front is also used to determine the distance to the wall in front and stop when close enough, and it may be used for determining distance driven instead of using encoders. Because of the relatively slow time of flight for the sound waves produced by the ultrasonic sensor, interference and dead time on the processor were real problems. Therefore, the sensors are queried in a certain order with active delays in between. The gyroscope becomes useful when turning and when driving without a wall. It can keep track of the rotation of the robot very precisely for the short amount of time it is used, as drift does not have time to pick up and a new reference is established every time before the program relies on the gyroscope.

Another sensor is the colour sensor near the ground on the front of the robot. It sits as close to the ground as possible while still being safe and guarded from speedbumps, debris and steps, and it is used to determine the colour of the ground under or slightly in front of the robot. It is an Adafruit TCS34725 and the integration time is set to 60ms to get updates frequently enough so that the robot does not accidentally drive too far into a black tile before driving back. The camera could also be considered a sensor, but the details of that assembly has already been given in 3.a.

For powering and controlling the encoder motors, two new encoder motor drivers were needed. Because there are only two built-in encoder motor controllers on the Auriga and we needed four of them, as driving one side per controller did not work good enough, we found a way to use two more. There is another board from Makeblock called the MegaPi, for which encoder motor controllers with the appropriate port and the same library are available. Through reverse-engineering the drivers on the board using schematics for the Auriga as well as the MegaPi, combined with manual measurement, we were able to connect the two new drivers to the Auriga, define two new encoder motor ports in the library and then use these identically to the original ones.

4. Software

a. General software architecture

The software is divided into three different programs. On the Auriga, we have the low-level code, which is responsible for hardware interaction (except for the camera) and navigating from one tile to the next, through ramps and turning. The two other programs, which run on the Raspberry Pi, are the vision program, which detects victims and sends that data, and the navigational program, which makes decisions about navigating through

the maze, stores information about the maze, and decides when rescue kits should be dropped.

The navigational program can send commands to the low-level program, which is constantly listening. During execution of certain commands, the low-level program can accept interrupts for stopping and dropping rescue kits, after which to finish the command. When a command is finished, the low-level program sends a command to the navigational program so that it can continue the process. This interface is held constant and the implementation of commands does not matter to the other party. For extra redundancy, some features have been implemented, like always answering commands for confirmation.

The navigational program also receives information about detected victims from the vision program, like how many kits we should drop, and on which side the victim is on. The data is checked, to make sure that there is a wall to that side, and then forwarded to the low-level program, which blinks and drops the kits. To make sure that the navigational program always listens to the vision program, a separate thread is used for listening.

Orientation in the maze is done by the low-level code using primarily the ultrasonic distance sensors. Because two sensors are used for each side, we can theoretically get an exact value for the distance and angle of the robot in relation to the wall:

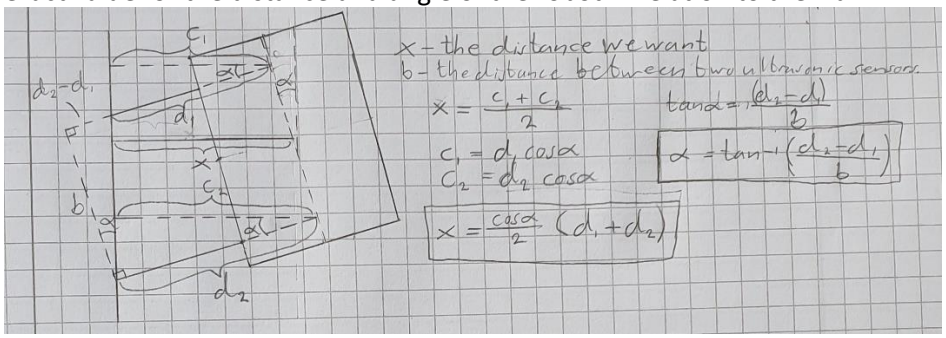

Fig. 4.a.1 – Calculation used for pose estimation

This is very good in theory, but due to some inaccuracies in the ultrasonic sensor readings there are slight fluctuations. Thankfully, this did not prove to be a problem, partly due to the smoothing of ultrasonic sensor measurements using a running average. To position ourselves in the middle of tiles in the sideward direction, the distance to the wall is used in a PD-loop (PID without integral term) to give a wanted angle, which is in turn fed into a PD-loop which gives the wanted motor speed corrections. The robot angle is also used to calculate the distance driven parallel to the wall. Using trigonometry we can calculate this distance with the robot angle and encoders' measure of the distance driven.

Ground colours are detected by the low-level code using a colour sensor which gives values for red, green, blue and clear. Each colour that we should be able to detect has a reference colour with the four values and a radius. A colour can be seen as a point in a four-dimensional coordinate system, and thus, when a colour is read we can calculate the euclidean distance between the colours, the parameter distance. Distance is calculated to all reference colours and the radius is subtracted, after which the closest one is picked. The radius is a measure of how spread out the values are. This code is wrapped in a class for modularity. For detecting blue and reflective tiles, a high enough share of measurements on the new tile have to confirm the decision.

The read function in the library for the colour sensor is blocking, meaning that it blocks other code from executing during that time. This is problematic because we cannot have any "dead" time in our control loop. In the read function, there was just a simple delay equal to the integration time followed by reading the sensor over I2C, so to fix this, the

delay was removed and instead it is checked every time the read function is called whether the integration time has passed since the last reading.

When dropping rescue kits, the low-level code blinks simultaneously to dropping the kits to save time. This is achieved by using similar methods to the colour sensor, with there being a while loop containing if-statements, which are only executed if enough time has passed since the last execution of that if-statement.

Driving a step is done as follows:

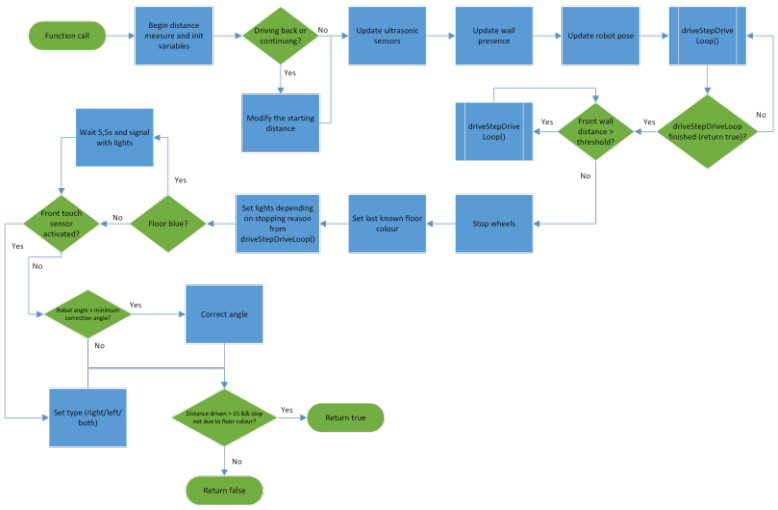

Fig. 4.a.2 – Flowchart for driving a step

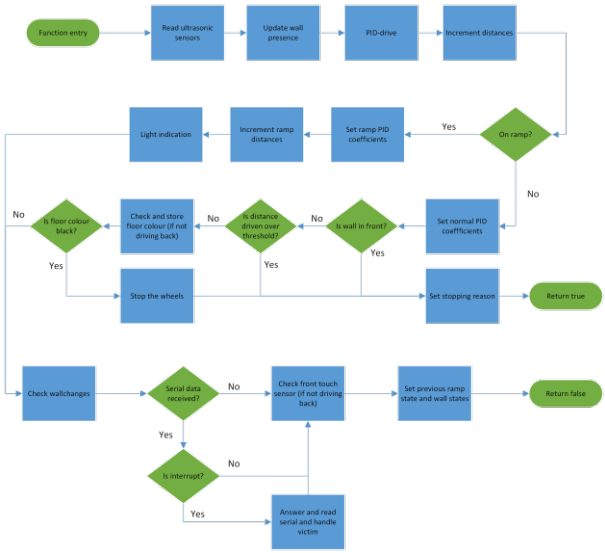

Fig. 4.a.3 – Clarifying flowchart for the driveStepDriveLoop() subroutine

To store data about the maze, the navigational program uses a list of a class called Map, which contains a 50 by 50 unsigned 16-bit integer two-dimensional array/matrix, where each position in the array represents a tile. For each level we visit, we add an instance of it to the list, so that we can easily change our position in the maze and level by changing the index in the list and in the map array. We store our position with three integers, one which keeps track of which map we are on, one that contains our X-position, or “left and right”-position, and one which contains our Z-position, or “forward and backward”-position. To keep track of which direction we are facing, we use an integer which can have four possible values for front (=0), left (=1), back (=2) and right (=3), which have the same values as the bit indexes in the stored map, so that I can convert directions directly to walls. If we for instance turned left, the direction would be incremented by 1. Since we do not want the value of the integer to be larger than 3, or smaller than 1, we make sure that we add or subtract 4 to it, if it is larger or smaller than that, after each time we update the direction. Each bit in the 16-bit unsigned integer is used as a boolean, which is possible due to bitwise operations, as we can use AND when checking for a value and OR, or NOT AND when setting a value in the integer to true or false respectively. If we were to utilize all the 16 bits, it would be about 90% more memory efficient (as 1 bit is 1/8th of a byte) than using regular booleans.

When we travel up or down a ramp, the low-level program will send the height and length of the ramp, which is calculated using trigonometry. The navigational program uses this to calculate how high up we are compared to the first level, and how many tiles forward we moved. This way, we can figure out whether a ramp leads to a new or known map, and we can keep track of where we are on a level even if there are multiple ramps to that level, as well as reduce the amount of memory required.

To navigate the maze and to ensure that we can explore the entire maze if there is enough time, the navigational algorithm based on Trémaux's algorithm. The algorithm is typically used as a maze-solving algorithm, but if the maze has no solution, as the case is in this competition, it always explores the entire maze. As described in figure 4. a.4, our implementation of it starts by looking at the tiles around it for unexplored tiles. If there are any, they are saved in a list in the current map-object to keep track of which level the map belongs to. Then, the program turns and drives to one of the tiles. It first tries the tile on the left, then the tile in front, and lastly the tile to the right. After it has driven it checks the tiles around the new tile and the cycle continues. When it comes to a tile that does not have any unexplored tiles around it, it finds a path to the last saved tile using a shortest path algorithm and goes to that tile. When there are no unexplored tiles left in the map, it returns to the start tile, or ramps that are not fully explored, as I mark if we are not on the first level.

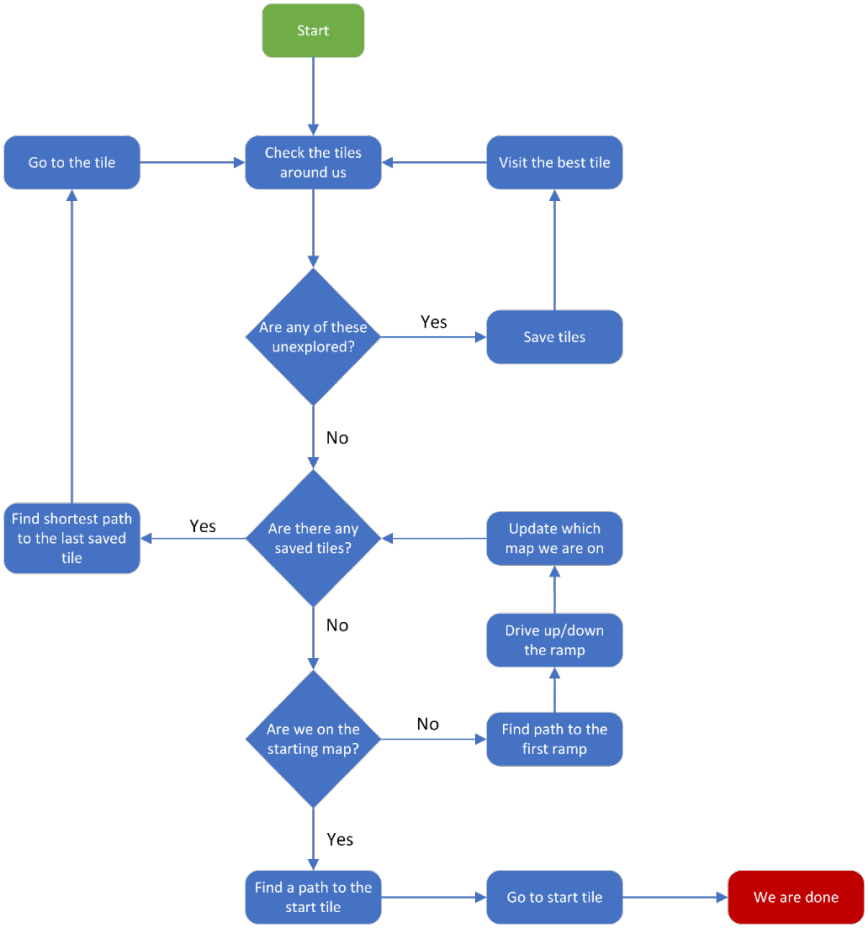

Fig. 4.a.4 - Flowchart for the navigational algorithm

To find a path back to the start when the time is almost up, or to any tile in the navigational algorithm, a recursive path-finding algorithm was developed from scratch. The algorithm works by checking the tiles around it, starting with the one closest to the goal tile. If there is no wall in the way, we check if the tile is explored, not black, not a ramp or if it is the tile we are trying to find, and if either of those are true, we move onto the next step. The tile can also not be in the current path. In this step, we add the tile to a list, which represents the current path, as a byte array which stores the tile's X- and Z-coordinates. From here, we search the tiles next to this tile, and then the tiles next to those and so on, until we find the tile we are looking for. If we return without having

found a path, we remove this tile from the path and mark it as a fully explored tile, which means that we will stop searching onto that tile. The shortest possible distance is between two tiles is adding the difference between the X- and Z-coordinates between the two tiles since we can only move in four directions. When we find our goal tile, we check if the distance to this tile is the shortest possible, and if so, we are done and return the found path. If this was not the shortest path, we begin returning to where we started, but we do not mark the tiles from the found path as fully searched, because there could be a shorter path that uses part of this path. When we are going back to the start and we are not on the starting level, or when we have explored an entire map, we also need to "jump between ramps" to find paths, by taking our previously used ramps to get back to the starting area. It would also be possible to use the travel time instead of tile amount for more optimization.

The vision code is creating an HSV and a binary image out of the captured image. The binary image is used to find visual victims, first it finds contours in the image which are compared with combined sample images. The combined sample images have been created by adding together samples for each victim using bitwise operators (AND & OR). Resulting in one sample with all pixels the victims needs and one with pixels it may have.

Fig. 4.a.5 – Combined sample victims for the visual victim "U".

The coloured victim identification uses HSV-thresholds to create masks. By analysing the mask, it makes sure it can be a victim by including factors such as size, position and difference to the surroundings. Furthermore, it also compares it with previous frames to make sure it keeps its shape.

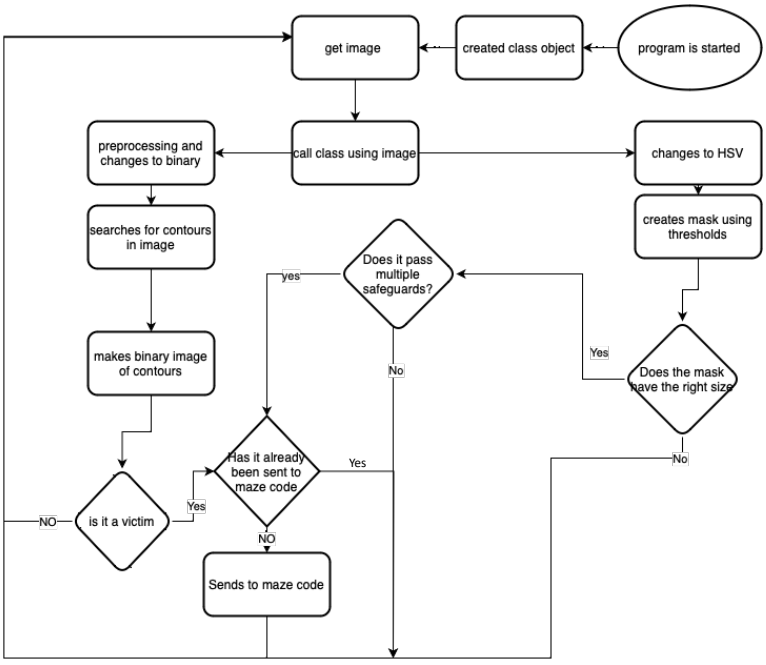

Fig. 4.a.6 – Flowchart for victim identification

b. Innovative solutions

One important aspect of our odometry is something which we have named “wallchanges”. A wallchange is defined as when a wall is “lost” or “found”, that is, when the wall presence switches from true to false or vice versa. Because all walls begin or end at tile intersections, these detected changes in wall presence can be used to correct the odometry provided by the encoders and front ultrasonic sensor, which would otherwise be unreliable. Wallchanges give some stable point of reference. Wallchanges are calculated individually for every sidewards facing sensor which means that problems could arise if the robot angle is too great, as the wallchanges would be detected at the wrong distances. This is solved by using the current known robot angle and trigonometry to calculate the “true” position of the wallchange. When a wallchange is detected, we have a known reference to the robot position. That can be used to look at changes in the front ultrasonic sensor distance to determine how far the robot has driven.

Different testing procedures were implemented for the three distinct different parts of the code. In the low-level code, most new features were tested from the ground up, meaning that the function of the most fundamental functions was confirmed first, thereafter the functions “above” them and so on. This makes it easier to find errors as it narrows down the possible sources. Another type of testing is of the functionality of the robot that we actually want, and not the functionality that builds up to the goal. This testing was done by just letting the robot drive and when it encountered errors, it was stopped and the reason for the error was investigated. Elimination of behaviours correlated with errors was attempted with varying degrees of success. Not much raw data was collected due to the difficult nature of transferring it from the Auriga to the computer (an attempt was made with an ESP8266 but we never went through with it), but data was collected for setting thresholds (like distances to walls, measuring how far the robot drives, etc.).

There have been multiple approaches for testing the vision code. One has been testing it on runs with the robot, other options have been testing while standing and changing different factors. But the most common used is testing with previous captured log images. Sometimes the entire code was tested and sometimes only specific parts.

To test new features in the maze navigation code, there were two different possible procedures. Later, it was possible to test new features directly by running the code and trying to get the robot in situations that could test them and using a log file to find bugs and get data. Before the hardware was working, though, it was much harder to test the code. To counter this, a simulation of the competition was created, where code could be written and tested much quicker and easier. Without it we would likely not have time to fully develop the path-finding algorithm or the navigational algorithm.

5. Performance evaluation

The performance of the robot is not as reliable as we had hoped for. We believe that the robot is a good prototype, but that it is not as polished as it should be. One of the big problems is that the robot is very reliant on the low-level code being able to drive perfectly, which is not the case. This makes the robot too complex for its own good, as a minor error in the low-level code could have negative consequences later. Mitigations for this have been tried, and work to an extent, but it could be better. Potential fixes would be to either make the maze navigation less susceptible to errors low down or drastic improvements to the low-level code, like better sensors and movement system (handling of stairs and speedbumps).

Testing has been done partly by testing the individual systems like described in 4b, but also by testing the robot as a whole similarly to how it would be during a scoring run. These tests were then analysed, and common failure points mitigated.

6. Conclusion

To conclude, this project has taught us a lot. Our goals were to learn about robotics, collaboration and having a well-performing robot in the end, and we are of the opinion that these goals have been reached. A lot of improvements were found on the way, but due to a lack of time, they could not be implemented, and thus there is a lot of room for improvement. The robot has many good features, as outlined in this document, but there is much that can be done in terms of robustness and new features.

References

- http://docs.makeblock.com/diy-platform/en/electronic-modules/main-control-boards/me-auriga.html
- https://www.arduino.cc/reference/en/
- https://docs.arduino.cc/tutorials/generic/basic-servo-control