CS 424

Fall 2018

Machine Problem 2

Tomb Raider Robot

Hongyi Li (hli106)
Yangge Li (li213)
Michael McGuire (mlm3)
Dipayan Mukherjee (dipayan2)

Introduction

In this machine problem, we are supposed to make the robot search through a maze. The robot should be able to follow the wall of the maze using the wall signal. The robot should also be able to keep a distance from the wall without bumping or losing the wall. In addition, the robot is supposed to plot a contour for the trace it passes. Finally, the robot should be able to take pictures of it's environment using the camera and identify specific items that appear in the database using openCV.

System Overview

The Overall Structure of the Code

In the main function, we will first make sure that the robot can enter into the maze in any random angles and adjust itself parallel to the wall. The main function will then enter a while loop that won't finish until we press the PLAY button. Inside the while loop, it will use motion logic to navigate mazes. This includes following the wall, left turn and right turn. The program will also check if the robot is in a safe state during every iteration of the while loop.

Whenever the robot stops moving forward and starts rotating, the main program will report the distances the robot traveled. At every turn, the main program will report the angles the robot turned. With these information, the contour plotting thread can plot the layout of the routes correctly.

Besides these, we also have threads for song playing and camera. Both of them will interact with the main program by mutex locks or semaphores. In this way, the song will only be played when unsafe conditions pop up and the camera will stop taking pictures when the robot finishes the maze.

Finally, the robot will do image identifications when it finishes the maze and the main program gets out of the while loop.

Thread Information

In total, we have 4 threads. The main program is the main thread. In addition, we have three threads for contour plotting, song playing and taking pictures respectively. The contour plotting thread interacts with the main thread through semaphores, while the song playing and camera thread interact with the main thread through mutexes.

We made these decisions because the song thread will only run when the main thread encounters unsafe conditions and thus stops running. Similarly, the main program won't obtain the lock and thus stops the camera thread until it finishes the whole maze. As we can see, the running of the song and camera threads don't have too much overlapping. However, we need to plot the contour as the robot is navigating. As a result, the running of the contour plotting thread overlaps a lot with the main thread. By using a semaphore between the contour plotting and main thread, we can make sure that the execution of the main thread won't be blocked by

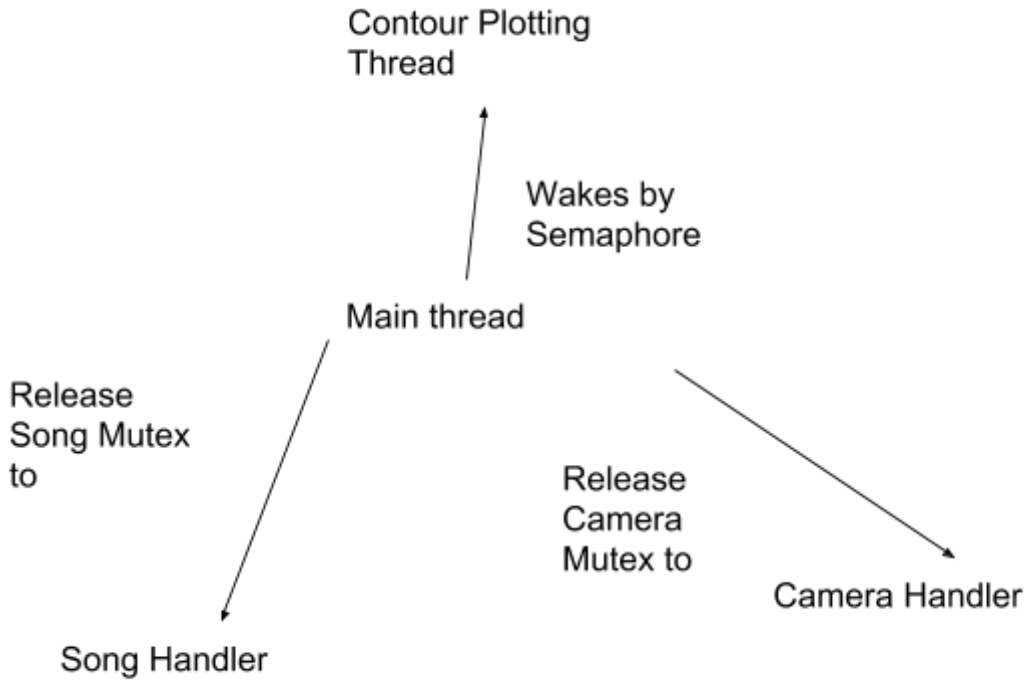
the contour plotting thread. This is because we can make programs waiting for semaphores go to sleep.

In terms of priority levels, the main thread is at a higher priority level than all the others. The contour plotting thread can only run when it gets the semaphore. However, the main thread will only release the semaphore when the robot starts a new line or turn. As a result, the contour plotting thread will always stop when the main thread is reporting new navigation data.

The song handler thread can only run when it has the song mutex. However, the main thread won't release the mutex until it encounters unsafe conditions. in this way, the main thread is always in a higher priority level in safe conditions.

The camera thread can only run when it owns the camera mutex and it first gets the mutex before taking each picture and releasing the mutex after taking each picture. Nevertheless, the main thread will acquire the mutex after it finishes the maze. As a result, the main thread is at a higher priority level after the robot finishes the maze. This is also the time when the image identification process is supposed to begin.

In conclustion, the architecture of our program is as below:



Contour Plot Algorithm

In order to plot the contour, we used two global vectors, one for distances and one for angles. Every time the robot stopped to make a 90 degree turn clockwise or counterclockwise, we would add the distance traveled since its last turn to the distances vector, and would record 90.0 or -90.0 degrees to angles, depending on if the turn was clockwise or counterclockwise, respectively. At the end we add the last

distance to the distances vector and 0.0 to the angles vector. These vectors are used to dynamically generate the contour graph from start to finish.

Wall Tracking Algorithm

In order to track the wall, the robot has to keep a distance from the wall. It should neither get too far from or too close to the wall. It should also be able to handle left and right turns around corners. The robot is running at speed 200 for both straight and turning. Since some of the turning are hard coded, we are not able to easily modify the running speed. In order to satisfy the requirements above, we design the following algorithm:

- When the robot encounters a left corner. Since the wall sensor is only placed at the right side of the robot. The robot will not be able to see the wall in front of it. Therefore, the bump sensor has to be used to identify a wall in front of the robot.

  When the robot receives a bump signal, it notices it encounters a left corner. In this case, the robot will perform the following action:
  - Move backward to get away from the wall in front.
  - Turn counter clockwise for 800 milliseconds such that the robot is about to be parallel to the next wall.
  - Keep turning until the robot wall signal is below a certain threshold such that the robot is parallel to the next wall.
- When the robot encounters a right corner, the wall signal will go to 0. However, since the wall sensor is not that accurate, we use 10 as the threshold. Therefore, when the wall signal is below 10, the robot will know it encounters a right corner. When encountering a right corner, since the wall signal is totally lost, it's impossible for the robot to use wall signal to properly track the wall. In this case, we just hard code the turning.

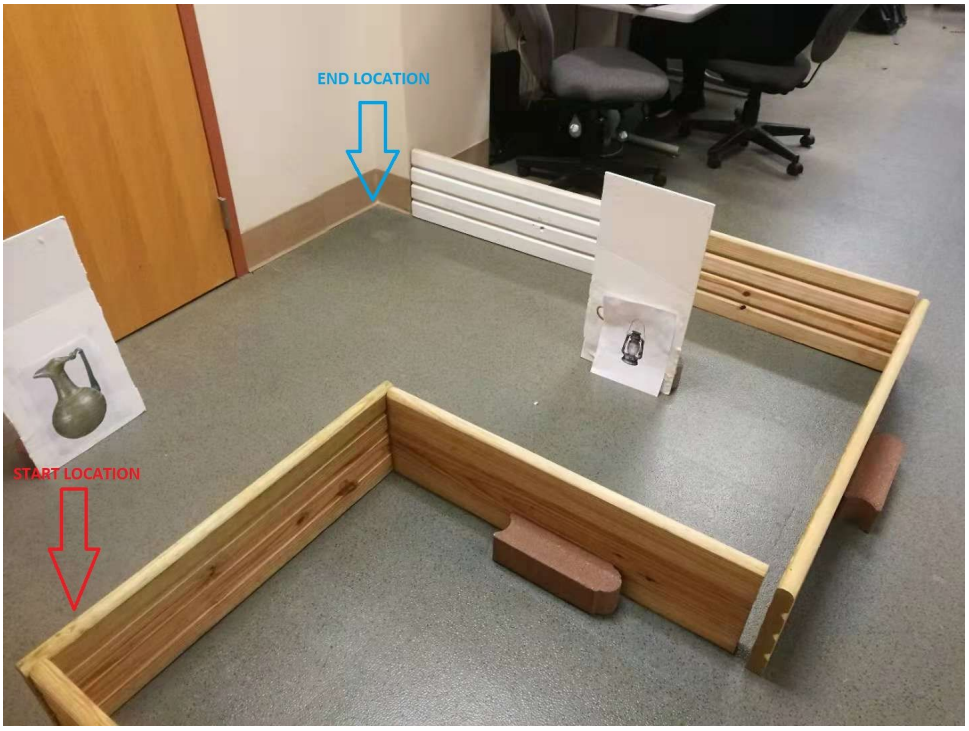
  When the robot encounters a right corner, it will perform the following step:
  - Moving forward for 1250 milliseconds to move out of the first wall.
  - Turn clockwise for 900 milliseconds to make a 90 degree right turn.
  - Move forward for 1100 milliseconds to move into second wall to regain wall signal.
- A threshold is set to indicate that the robot is too far away from the wall. When the wall signal is smaller than 20, the robot will know it's too far from the wall.

  When the robot is too far away from the wall, the robot will perform the following steps:
  - The robot will turn clockwise to adjust its direction.
  - The robot will stop turning when one of the following condition is met:
    - The wall signal is larger than 30.
    - The wall signal just reaches a local maximum.

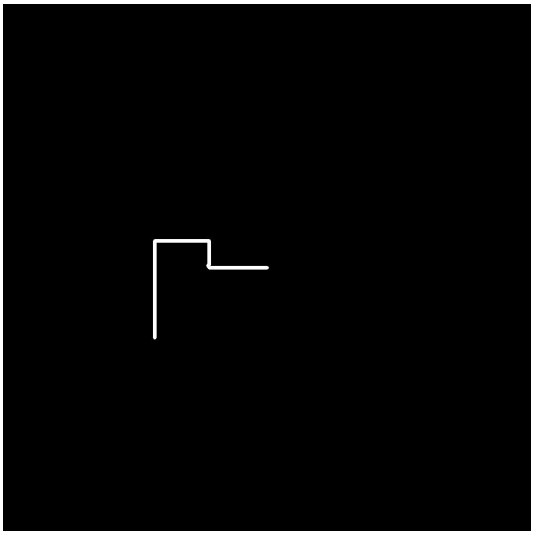
- Another threshold is set to indicate that the robot is too close to the wall. When the wall signal is larger than 110, the robot will know it's too close to the wall. In this case, the robot will perform the following steps:
    - The robot will turn counter-clockwise to adjust its direction.
    - The robot will stop turning when the wall signal is smaller than 100.
- In other cases, the robot will just move straight.

Result

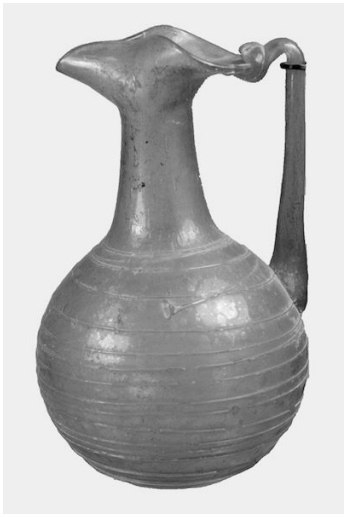
To test our software, we run the robot in the maze shown in the picture below:



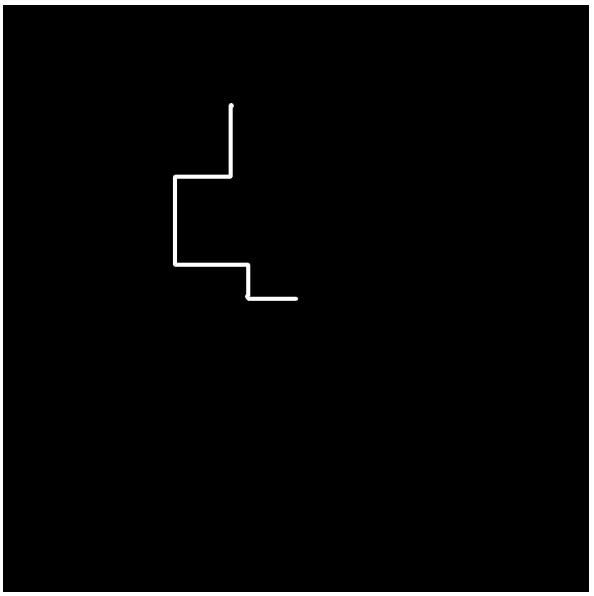
The maze is constructed by only 5 walls and has two objects to be detected. The robot is able to follow the wall of the maze and the plotted contour is shown below:

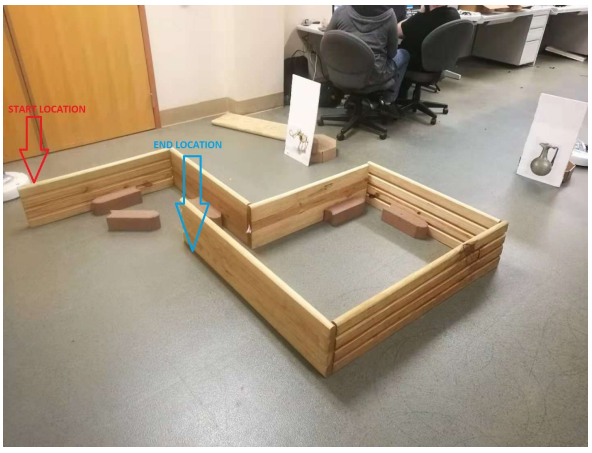


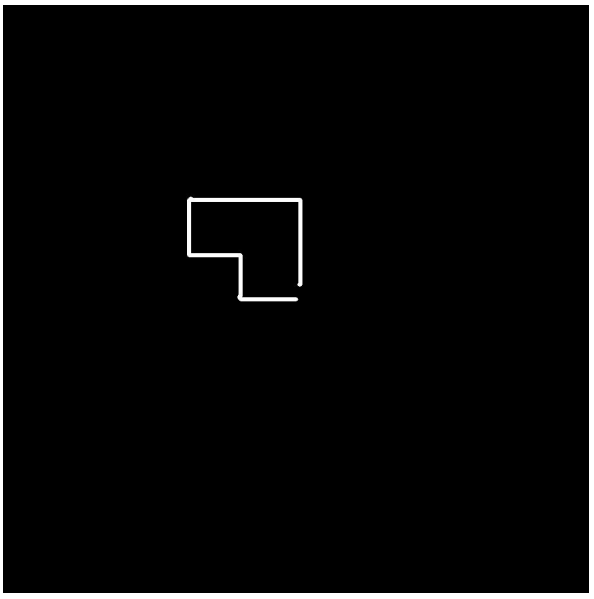
The robot is able to identify the two objects in the maze. The results for the identification are shown below:



We run the software we have on two additional different mazes. The picture of the maze and the corresponding contour are shown below:







Conclusion

When designing the navigation algorithms, we were initially not very sure about how to pick up those threshold values. As a result, the robot often turned too much or too less at some critical points. To solve this, we did a large amount of trials and kept changing those values to finally determine the optimal threshold parameters.

The hardest part about the project was integrating all of the separate parts into one coherent framework. The image detection and the navigation would work perfectly fine in isolation, but as soon as the two were brought together, neither would function properly. We remedied this situation by simplifying our design and minimizing the number of complex tasks the robot performed simultaneously, while keeping the threads independent of each other.

Initially our code was complicated and difficult to understand, compounding our bugs. We had to redesign the code into a few critical blocks. That way, we could look at a certain block and knew for sure what had to be done at that block in terms of navigation, contour logging, danger control, and image processing.

We learned how to distribute the separate tasks to different team members, and then coordinate as we integrated the many parts together. We learned a lot about multithreading in C++ and the issues competing threads tend to run in. We also learned a lot about version control.