Final Project: Shortest Path Line Maze Solver

Akmarzhan Abylay

CS152 Spring 2020

Problem Definition

People have been interested in solving mazes ever since they were created back in the 5th century BC (Krystec, 2001). Following the general hype around maze solving, I decided to construct a robot, which could solve simple mazes using the shortest path available.

Below is the problem definition:

	Description
Initial State	A robot is placed at the start of the maze (i.e., a combination of paths represented by a black tape) facing the center of the maze itself.
Possible Actions/Results	Depending on the type of intersection, the robot can either go straight, go back, turn right, turn left or stop, if it reaches the end of the maze. It will do it by adjusting the speed of the motors and changing its orientation, after which moving in a line.
State Space	There are infinitely many states, since we are dealing with a continuous line of tape, in addition, to numerous orientations (i.e., around 360 degrees), since there are no limits. However, in an idealized situation, we will have 4 orientations (i.e., N, S, W, E) and numerous points throughout the lines.
Goal Test	The end of the maze is the goal state. It is represented through a rectangular box made from black tape. The robot will identify it through its sensors.

This problem is exciting for numerous reasons:

- It is an opportunity to apply theoretical knowledge from class in practice (i.e., building the robot from physical materials);
- This is not an easy problem and requires thorough consideration, as well as possibly can be used on larger scale problems (e.g., material handling, warehouse management, pipe inspection, and bomb disposal);
- It can be further used in other classes (i.e., for CS162 FP).

AI approaches are a good fit here because we are trying to create a tool that performs functions considered intelligent by humans, such as reasoning, knowledge representation, learning, and inference (i.e., applying logic rules to the available data to reach conclusions). Maze solver robots involve rational acting, as they act to achieve the best outcome (or best-expected outcome when there is uncertainty), which is reaching the end of the maze in this situation. It doesn't only try to reach a solution but also optimizes learning from past mistakes (i.e., first exploration phase) to perform more effectively in the future, requiring less time and steps to solve the same problem. It acts autonomously in the environment it was placed in (i.e., maze), perceives the environment and pursues a goal (i.e., end of maze). The robot modifies its behavior based on the environment and as "AI is concerned with intelligent behavior in artifacts", maze solver robots are good examples of intelligent robots (Russell & Norvig, 2016)[1].

[1] **#aiconcepts** - analyzed artificial intelligence concepts with an appropriate level of detail while placing the reasoning in the context of the problem and literature.

Solution Specification[2]

List of materials

Function	Materials
Microcontroller	• Arduino UNO
Actuators	• Motor Driver Shield for working with actuators • DC Motor and Wheels (2x)
Sensors	• 4 Line Sensor Modules - I made 2 of these myself using a breadboard, TCRT5000, wires and 2 resistors (220, and 10k Ohm).
Power Supply	• 9V batteries with holders (2x)
Other materials	• Jumper Wires (male-to-male, male-to-female and female-to-female) • Acrylic Sheet

Wiring

Below I drew the wiring. The pins got messed up in the process, so I specified that the leftmost sensor is connected to A2, rightmost to A5, center left to A3, and center-right to A4. We will use the IR sensors since our line is made from black tape, and it will be easy to detect it as the black color doesn't reflect the light.

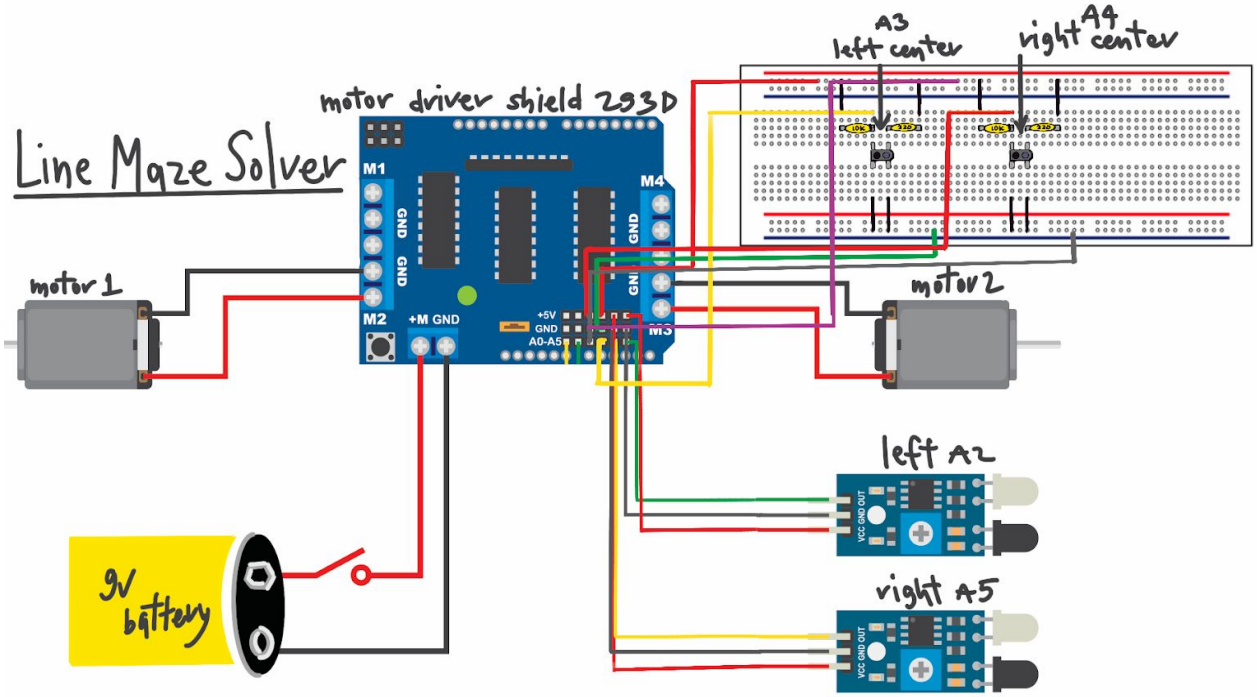

Figure 1. Wiring of the Line Maze Solver

[2] **#robotics** - accurately selected relevant sensors, actuators and control algorithms and provides a clear justification (e.g., IR sensors are required because maze is made of black type which doesn't reflect light); justified why the robot was intelligent and described why this particular solution works in the context of the problem.

Code Implementation

For the implementation of this line maze solver, I developed a framework in which the sensor and actuators interact (i.e., a combination of if-else statements) in order for the robot to follow the line, turn in the right place, and stop when it reached the end of the maze. Also, I identified a way to store the path that the robot took (i.e., an array filled with the type of actions, such as "L" for turning left and so on), and created a shortcut based on which it will find the shortest solution given the first exploration phase. The shortcut for the shortest path (i.e., optimization) is better explained through examples in the section below.

Analysis of Solution

Through trials and errors, the robot started to work. Below are the two mazes that I tried it on (i.e., it is the same maze, only the starting point changed).

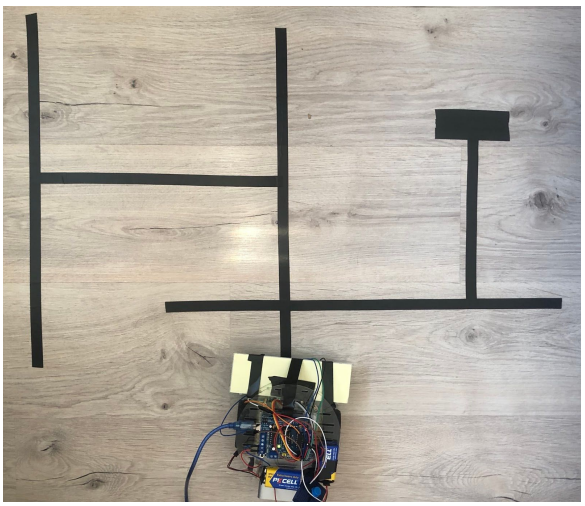

Figure 2. Maze 1

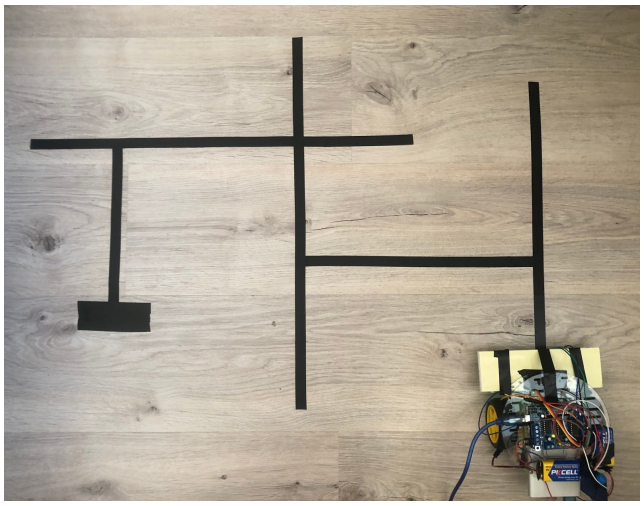

Figure 3. Maze 2

We should first think about how we would solve this maze in theory. Just randomly wandering around will be of no help, which is why it is better to implement a heuristic to help find a solution for sure. There are numerous heuristics, such as dead-end filling, Trémaux's algorithm, the pledge algorithm, or the shortest path algorithms (e.g., A*), but given the simplicity of the mazes we are going to solve, and no particular limit in time, we will use the wall follower heuristic, and specifically, the left-hand rule (LHR), where we hypothetically put our left hand on the left wall and follow the path always turning left, and going back if it is a dead-end (Wikipedia, n.d.). Given that all walls are connected in our mazes, this heuristic is complete but not optimal, as we are guaranteed to find a solution, but it won't be the shortest one. However, that is of no problem, since we are going to store each action that the robot takes at each step and then decode the path using the following shortcuts. In other words, if we turn left, then go back and turn right, it is equivalent to just going backward. Following the same logic, we can make other shortcuts shown in the table below.

Also, I didn't implement the PID control, as for this simple implementation, it wasn't necessary. However, in the future, to improve the accuracy of the robot and the way it follows the reference path, we might include it after some tuning.

Shortcut	**Original path**
B - backward	LBR, RBL, SBS
R - right	SBL, LBS
S - straight	LBL

This shortcut will allow us to find the optimal path (i.e., the shortest) since we store each move the robot takes and just keep the needed steps. For example, in mazes 1 and 2, the robot will follow the next steps.

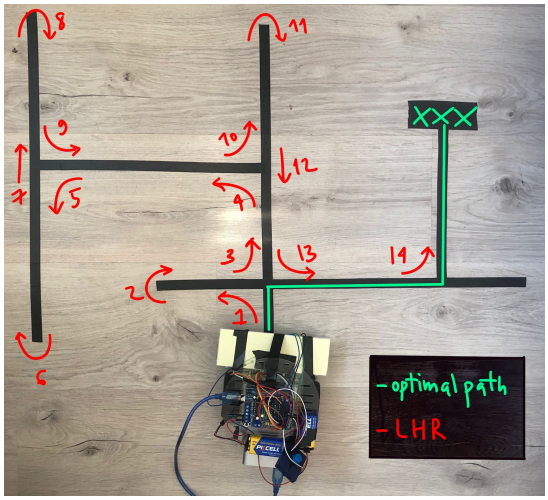

Figure 4. Maze 1 - LHR

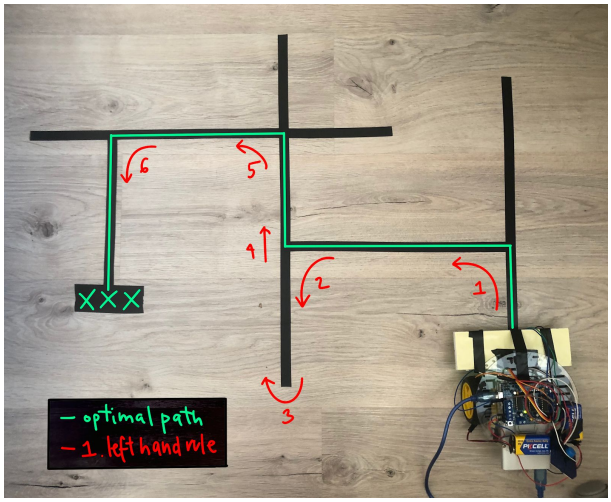

Figure 5. Maze 2 - LHR

The shortcuts will substantially cut the path and we will get:

- **Maze 1**: [*LBLLLBSBLLBSLL*] → [*SLLBSBLLBSLL*] → [*SLRBLLBSLL*] → [*SLBLBSLL*] → [*SSBSLL*] → [*SBLL*] → ***RL***, which is our optimal solution.
- **Maze 2**: [*LLBSLL*] → ***LRLL,*** which is our optimal solution.

This is a youtube video with the robot solving these two mazes. The videos are also submitted as a secondary file zipped with the code.

Conclusion

In this paper, I have created a simple maze solving robot, which optimizes its path after exploring the maze. Instead of implementing a search algorithm, which would consume additional memory and energy, our robot used a shortcut heuristic to solve the maze in the most efficient way. This robot could have been improved through additional sensors, actuators, and new heuristics or software (i.e., drawing out a map of the maze and, after the exploration, go in a direct line from the start to end using diagonals and so on). In the future, I might use this robot in further courses, such as CS162 to develop software to control the robot distantly[3].

[3] This turned out to be more than 3 pages, since I was adding the figures and specifications into the main sections. However, in plain text, this is roughly 3 pages.

References

Adafruit. (2012). Adafruit Motor shield V1 firmware with basic Microstepping support. Retrieved from https://github.com/adafruit/Adafruit-Motor-Shield-library.

Krystec, L. (2001). Amazing Mazes. The Museum of Unnatural Mystery. Retrieved from http://www.unmuseum.org/maze.htm.

Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach (3rd ed.). Boston: Pearson.

Wikipedia contributors. (n.d.). Maze Solving Algorithms. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Maze_solving_algorithm.

Appendix

The code was submitted as an additional file, but you can also find it on Github here.

Proposal

Problem Definition: I will solve a problem of a robot stuck in a maze that tries to find a way out. The routes will be represented through black lines on the floor (i.e., tape) and the robot will have two trials: the first one to explore the maze and lay out the map (e.g., backtracking is possible), and the second one to find the shortest way out.

Targeted LOs: #robotics, #aiconcepts and #aicoding

Proposed Solution & Deliverables: I am going to use an Arduino UNO with sensors (e.g., IR sensors), actuators (e.g., DC motors) and some additional modules to create the robot itself. I will write the code in an Arduino IDE and use a maze solving heuristic to find the shortest path from the start to end of the maze. I will submit the code, the report and the video of the robot in action.