A Maze Solver

Kyle Zhou, Erin Morrow
https://github.com/zhouzhouthezhou/MatLabMazeSolver

Abstract:
One application that showcases the power of MATLAB is the concept of a maze solver. This project explores different methods of solving mazes, an implementation of a shortest path maze solver in MATLAB, and future prospects for the project.

Introduction:

Over the years, many different ways of solving mazes have been created. The most basic of which is the right hand rule, wherein you put your hand on the right wall of the maze and follow it to the end. This, however only works for mazes where the walls are connected to the outer wall of the maze and finds a very messy solution and is also just generally inefficient. The method that we used finds the shortest path from the start to the finish, working for non-connected mazes with multiple solutions.

Theory:

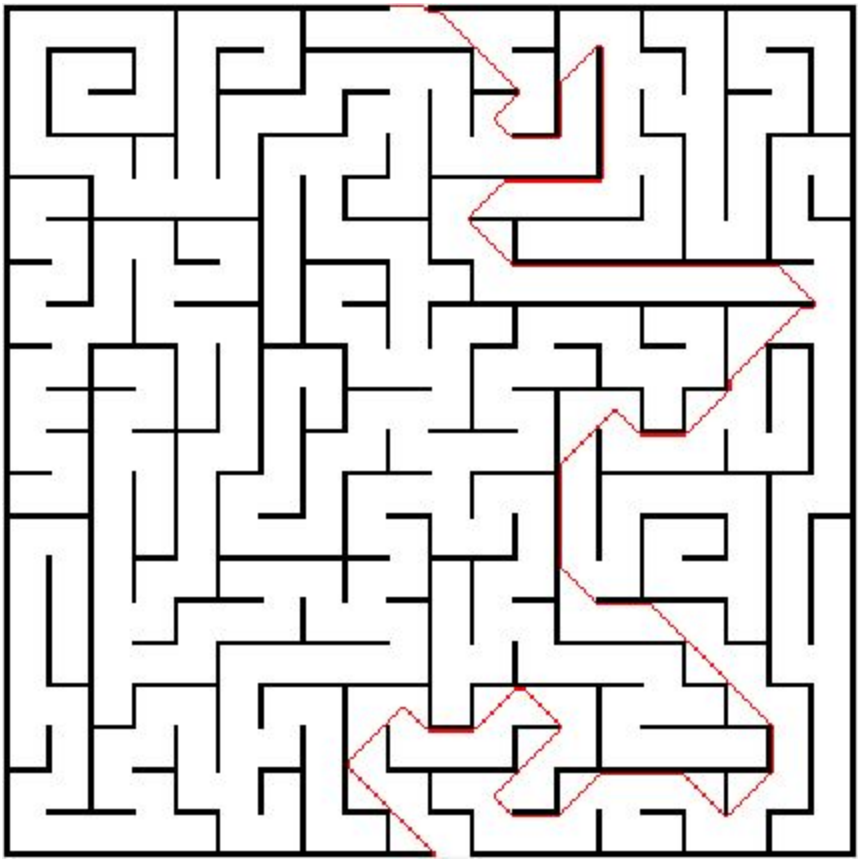
The shortest-path method makes use of MATLAB's shortest path function, which takes in a graph and two nodes and finds the shortest path between the two nodes. So now the problem is generating a graph representation for any arbitrary maze, calculate the index of the start and finish nodes, and display a graphical representation of the solved maze.

Building the Graph:

Using MATLAB's image processing library, we parsed the maze image into a n*m*3 tensor, representing the rgb values of the image. If the image was greyscale, we would construct the n*m*3 tensor using the greyscale n*m matrix. Then we extracted the red values and used it as a representation on the maze, where a r value of not 0 represented a wall and a r value of 0 represented the path of the maze. MATLAB also has convenient data structures built in, so we used the Graph object and created a graph with n*m nodes, with each node's name being a number from 1 to n*m, representing the index of each pixel in the image. So a 322*322 pixel image will result in a graph of 103684 nodes. The using some clever math each node was connected in a lattice formation to mimic the image matrix, with each node in their corresponding location according to their index. The using the find function we found all the pixels that were non 0 and added their index to a vector which was then used to remove their corresponding nodes from the graph. This also will remove all edges connected to the removed node, leaving only the nodes that represent the paths of the maze.

Solving for the Shortest Path and Display:

Interestingly, the hardest part about solving for the shortest path is dynamically finding the start and endpoints. This was accomplished with an algorithm that took the r matrix of the image and ripped out the edge vectors, looking for zero values in each of the 4 edge vectors. This will find two vectors and return which edge vector they are (top, bottom, left, or right) as well as the start index and end index of the 0 values. Then using some clever math, we were able to take this information and project the vector indices on to the original maze-space to obtain the indices of the start and end of the maze. Then we just pass the generated graph and start/end indices into the shortest path function to obtain a vector of the indices that correspond to the shortest path through the graph and subsequently the maze. We then rebuild the rgb tensor, manipulating the indices of the shortest path to the rgb values of red. When displayed, the resultant image will show the shortest path through the maze.



Problems, Constraints, and Future Prospects:

Currently the maze solver is only able to work for rectangular n*m mazes whose walls have no gap between the edge of the image and the edge of the maze. There was not enough time to implement detection of the outer wall of the maze. This is a big problem, however, since most mazes will have that gap between the edge of the maze and the edge of the image. Furthermore, if we could get this edge detection to work, our algorithm would work for any arbitrary maze,

not just rectangular ones. This would also allow for us to use MATLAB's ability to process a real time camera feed to overlay maze solutions on the feed real time.

Another issue is the efficiency of the algorithm. As you may have noticed, just running a 322*322 pixel maze forces the graph to generate an absurd about of nodes. While taking only ~2 seconds to run for a 3232*322 pixel maze, the 1802*1802 test maze takes over a minute to solve. While this is fine if you just want to obtain a solution to the maze, it makes real time solving impossible. If we had more time we could figure out a way to optimize the graph generation to bring the absurd numbers down and make the function faster.

Instructions for Running:

Running this script is very straight forward. After navigating to the directory where the m file is stored, you just need to run the command mazesolver('path_to_maze_picutre_name'). This will start maze generation and after some time the solved maze will display. In the git repo, there are 3 test mazes that we used. One simple maze, maze_small.png, a very very large maze, maze.png, and a rectangular maze with multiple solutions, maze_rect.png. Feel free to use these or your own maze so long as they follow the constraints listed in the previous section.