Andrew Hart, Alex Robbins
Intro to Artificial Intelligence
Final Project: Maze Solver

Project Motivation

The goal of the project was to be able to detect and solve a maze from an image. We chose this project because we wanted a project that would touch upon image processing and object detection. The methods used in this project help lay the foundation for an AI to detect objects and interpret what it's seeing in the real world.

Relevant Background

Several different techniques were utilized within the project. The first of which is an image processing technique known as Thresholding. This technique was used in our project to reduce noise within the image. Thresholding is an algorithm that takes the value of each pixel in an image and determines whether it is below or above a certain threshold. It then uses that threshold to determine whether to replace that pixel with either a white or black pixel. This reduces the image to a much simpler form of itself where each pixel is either black or white.

The second technique used is object detection. Object detection is used to find certain objects within an image. We achieved intelligent object detection by using contour detection to first find all the contours (curves that joins all the continuous points along a boundary in an image) of the image. These contours are essentially the outlines of all the individual objects in an image. Each of these object-contours was then fed into an algorithm that finds the smallest rectangular box that fits completely around each contour. The smaller insignificant boxes were assumed to be noise and were ignored, and the larger ones (which were assumed to be bounding boxes of the maze contours) were combined into a single maze outline box. By this point the maze has been isolated within the image.

A second form of object detection was used to find the paths of the maze. Once we had a thresholded image of just the maze cropped from the original image, we could apply erosion to the image to isolate and shrink the paths of the maze-image to make them more manageable. This changes the paths from thick white lines to very skinny white lines.

The next technique used was Hough Line detection. This is another form of edge detection used to find straight lines in an image. We utilized hough line detection to find the path lines in our newly eroded maze-image. These lines were later used to find the start and end points of the maze.

The last technique is path finding algorithms. These types of algorithms are able to find paths between a start and end point of a model made up of paths. The type of path finding algorithm utilized is called A* search. This algorithm works by exploring different least-cost paths

while keeping track of the cost of each path taken (where cost is a combination of previous cost of a path plus some heuristic that estimates the remaining cost of a path). It will return the path that solved the maze and had the lowest cost.

Methods

The program is able to find the maze using the thresholding and object detection techniques described in the above Relevant Background section. Step one is to threshold the image so our application has a much simpler version of the original image to work with. A threshold value provided through the threshold slider of the GUI tells the program what level of pixel color to use as the threshold when determining whether to assign black or white to the pixel. For example, if this value is set to 60, then all pixels with a grayscale value above 60 are converted to white while all other pixels are converted to black (this slider is necessary because different mazes require different threshold values based on what colors they contain). This also acts to get rid of minor noise in the original image. It then takes that image and finds all the edges within it. After finding the edges, we are left with a contour image of all the outlines from the original image. It then surrounds all of these edges detected within individual bounding boxes. All boxes with areas less than the value provided through the minimum bounding box slider in the GUI are ignored, and the remaining boxes are combined into one final bounding box. Everything within this box is assumed to be the maze. This is how the program is able to isolate the maze from the image.

The next step is to crop the contents of the maze bounding box from the thresholded image. The result of this is a black and white image that contains only the maze to be solved. This process is done using simple OpenCV functions that translate the pixels within a possibly rotated rectangular area into a non-rotated rectangle image the exact size of the original rectangular bounding box.

Next, the maze is put through an Erosion function. The Erosion function does not detect the maze as the object but detects the white space between the maze walls as the object. This is so that it will erode away the white space between the maze walls, shrinking the path. This makes the image look as if the walls have grown and the path has become only a few pixels wide. These thin lines essentially make up a skeleton of the original paths of the maze. Using this skeleton, the maze can be solved much more easily and much more quickly.

Our application then uses Hough Line detection to determine the locations of the endpoints of each line that makes up the skeleton. These lines are a representation of each individual segment of the maze paths. Once it has a list of all the endpoints of these lines, it finds the points which are closest to edge of the surrounding box. The points found are used as the start and end point of the maze. Once it has the start and end points, A* search is used to find the shortest path between theses two points. A* search will search the maze by traversing only the white pixels of the maze. This is made easier by two things: First, because there are only two colors in the maze (white and black). This means that to find the successor nodes for

any given node on a path, it just has to search above, below and to the right and left for adjacent white pixels. This is very easy and fast to do. Second, the lines are thinned to only a few pixels wide, which drastically reduces the number of pixels needed to search.

After a solution is found, the application simply overlays the solution onto the original webcam image and displays a few of the intermediate steps (bounding boxes found, and the thresholded/eroded maze image).

Results

The program is able to solve mazes almost in real time. We found that eroding away the path before performing the search drastically improved performance. We attained a speed increase of 66.9% fewer seconds on average per solve, and an 80.9% decrease in the number of nodes expanded by A* search.

There are some issues though. Because Hough Line detection will at some points not be able to find all the lines contained within the skeleton, the lines containing the outermost points which would be used as the start and the end points, are sometimes not found. Thus the program will pick the wrong start and end points and the maze will not be solved correctly. The unreliability of Hough Line detection is also the main reason we were not able to get our program to solve mazes as quickly as we wanted. We had originally hoped to create a list of connected line-segments of the paths of the maze, and search those for a solution rather than searching through pixels. This would have greatly improved speed. However, the inaccuracy of Hough Lines made this approach impossible.