Abstract:

The main idea of our project is to solve mazes and find the fastest solution with relatively small time complexity. We’re starting with a realistic goal to solve two dimensional mazes with O(n) complexity where n is the number of intersection points in the maze.

Data Structures:

The data structures we’ll be using for this algorithm are graphs, stacks, and arrays. The graph will be used to structure the maze in the computer’s memory and connecting the nodes in an efficient and useful way for our purposes. The stack will hold the path of traversal where a node will be added every time the algorithm traverses, and popped every time it back tracks. The array will hold the directions as ints (1-4) and upon finding the best solution will output the path as directions (N,E,S,W) respectively.

Read in:

The way our code will process a maze starts off by reading in a black and white jpg/png file (TBD based on ease of coding). So far, the picture must be grid-like in the way that the maze is set up, so the walls are the same size as the paths. Upon read-in, the user will set a pixel scale which will specify how large the wall/path units are. Each ‘pixel’ will be read in as a Boolean called wall. Where ever the code can find a point surrounded by 3 or 4 not walls, it will label the point as an intersection node. Intersection being a string within the node struct specifying the type of node. Wherever the program finds a point surrounded by only 1 not wall, a node will be created of type edge. Points with 0 or 2 surrounding white spaces will be disregarded, however each node created will count the number of units that must be traveled to get to the next node. This will be saved into an int of the struct of which there are 4 specific variables: northDistance, eastDistance, southDistanct, westDistance. Often, not all of these variables will be used so for each node they will default to 0. These nodes will all be connected as a graph with the edges being the paths between intersection points. Start and finish nodes will be specified by the user after scale specification upon calling the inread function. For while loop implementation, a dimensionless start node will be connected to the indicated start node.

Solutions:

First, we will create a brute force method which will most likely have O(n2) time complexity. It will be a ‘hug the left wall’ technique where we use a depth first graph traversal to make our way to the finish node. The traversal will keep track of directions (north=1, east=2, south=3, west=4) in a stack to keep track of the path traveled. When the traversal finds an edge node it will backtrack to the last intersection node and using the visited array, choose a separate path. If all paths are already visited the node will be labeled as a deadEnd and the traversal will back track to the previous intersection node. Once the finish node is reached, the distance travelled will be calculated using the sum of distances between nodes in the current path. This value will be compared to the global variable fastestDistance, if this variable is empty or greater than the calculated currentDistance it will be rewritten and the path will be saved to the other global variable, an int\* fastestPath. Then, the current node will backtrack to the last intersection node and look for a different path to the finish in the same way. This process will repeat until the traversal back tracks all the way to the super imposed start node.

A findDistance function will be implemented with an int output being the total distance from start to finish along the given path. It will take the source node (start) and the stack array of directions created. Using the array of directions, it will traverse the maze along the given path and for every traversal take the corresponding distance to the direction travelled and add it to a sum which after traversing the entire array will be outputted.

A backtrack function will also be implemented with a node\* output and an input of the current direction stack and current node\*. The function will pop the most recent direction and travel the opposite direction to get back to the last intersection node. Or we will keep track of the previous intersection node as a global variable and the backtrack function will simply pop the most recent direction, add the current node to the visited array and reassign current to previous node.

Next, we have a couple ideas for finding the shortest path in less time. One idea is to create a line from the start to finish node. This would require add coordinates to the nodes upon read-in which wouldn’t be difficult, it would just change the type of image inreader we use. In this case, the traversal would start at the indicated instead of the super imposed start node. The traversal would be compared to the line at every intersection node and use its relative position to make a movement decision. (If left of the line, go right when possible and vice versa.)

The other method we have discussed is slightly harder to implement. It would require calculating the distance of each node to the finish node and making movement decisions based on that. So, at an intersection it would check each adjacent node’s distance to finish and traverse to the node with the least value for that variable.

Conclusion:

We have already started work on the brute force method and it the code itself seems rather simple so we will definitely be able to create a maze solver of O(n2) in a week. The difficulty comes from designing the inread function. We are currently researching the best image processor function to use for our purposes. Also, creating the graph will be difficult as it is the most recent structure we have learned. Hopefully this week’s homework assignment will give us the practice we need for an efficient implementation. Once we do some preliminary testing with the brute force method, we will move on to creating our own algorithm. Depending on time we will decide which method is rational to develop and begin working on that piece by Wednesday next week. Our current goal is to have our algorithm solve for the maze’s best solution in O(n) time complexity and we are on track to produce at least this quality of a product.