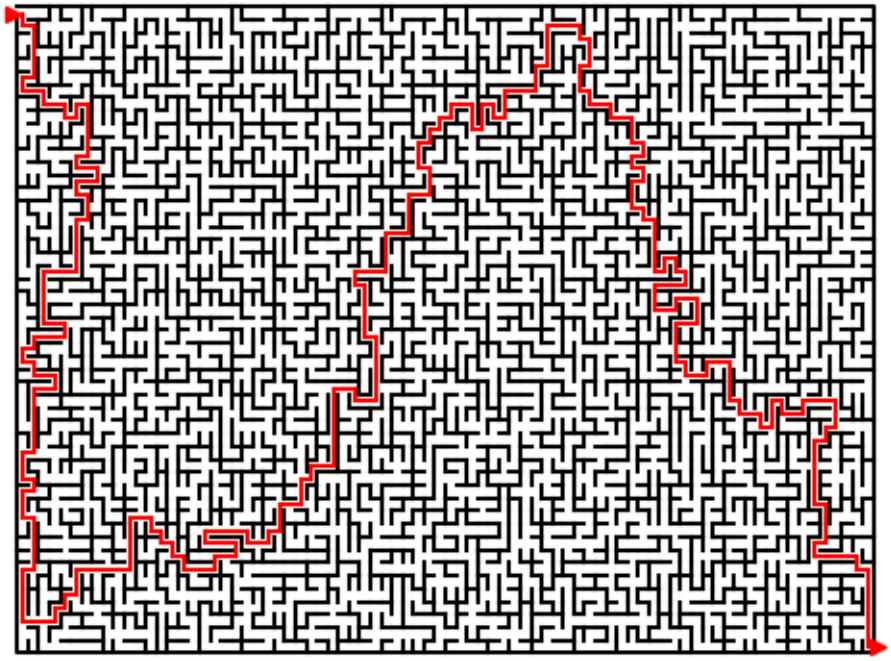
Maze Solver

CSCI112, Lab 10

March 21, 2023



File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

Documentation: Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

Solving a maze: Now that you know how to generate a maze, solving a maze with depth-first search is easy!

In your maze generation program, you found the walls of the maze by repeatedly joining two adjacent cells, putting a "door" between them and erasing the wall. You saved the walls so you could draw them.

Now we want to save all the doors. This will be a list of all pairs of cells that are connected to each other by a door. These pairs form the edges of a graph. Transform this list of pairs into an adjacency list representation of the graph, which can be a dictionary with cells as keys and lists of other cells that are connected to them.

If there is a door between cell a and cell b, be sure to add an edge from a to b and one from b to a!

Depth first search: You can now use a simple stack to search for a solution to the maze. When each node is placed on the "gray" stack, you can record its predecessor in another dictionary. The start node has predecessor `None`.

When the end node has a predecessor, you can stop the search!

Random seed: If you want to draw the same maze every time, set the random seed to the same value before shuffling the edges: `random.seed(1234)`, for example.

Drawing the solution: This predecessor list will enable you to draw the solution by starting at the end point, and drawing a line from predecessor to predecessor, until there are no more predecessors.

Feel free to use my drawing code. I've modified the `drawMaze` routine to facilitate drawing the solution path. (It also helps with some "off by one" problems some students had with my original drawing routine.)

After both of these routines have been called, call the `plt.show()` routine to show the drawing.

```
def drawMaze(walls, mazeSize):
    nrows, ncols = mazeSize
    fig, ax = plt.subplots()
    ax.axis("off")
    # draw outer box
    lo = -0.5
    hix = ncols-0.5
    hiy = nrows-0.5
    plt.plot((lo, lo, hix), (hiy-1, lo, lo), color='black')
    plt.plot((lo, hix, hix), (hiy, hiy, lo+1), color='black')
    # draw walls
    for wall in walls:
        a,b = wall
        y1,x1 = a
        y2,x2 = b
        if x1 == x2:
            y = (y1+y2)*0.5
            plt.plot((x1-0.5,x1+0.5), (y,y),  color='black')
        else:
            x = (x1+x2)*0.5
            plt.plot((x,x),(y1-0.5,y1+0.5),  color='black')

def drawPath(predecessors, mazeSize):
    nrows, ncols = mazeSize
    pos = (0, ncols-1)
    while predecessors[pos] is not None:
        y1,x1 = pos
        y2,x2 = predecessors[pos]
        plt.plot((x1,x2), (y1,y2), color='red')
        pos = predecessors[pos]
    plt.plot((ncols),(0), color='red', marker='>')
    plt.plot((-1), (nrows-1) ,color='red', marker='>')
```