Joseph Rand
CSC 3001
Dr Staab
December 12, 2020

Maze Generator Project

For this project, I wanted to create a program that would randomly generate a maze. This can be challenging because you need the program to create a maze that would actually have a solution, and not just be random paths leading nowhere. I also wanted to have the program solve the maze, and find the shortest path to the exit if there are multiple ways to solve the maze. Finally, I programmed in an animation that will show a dot moving through the maze to the shortest solution.

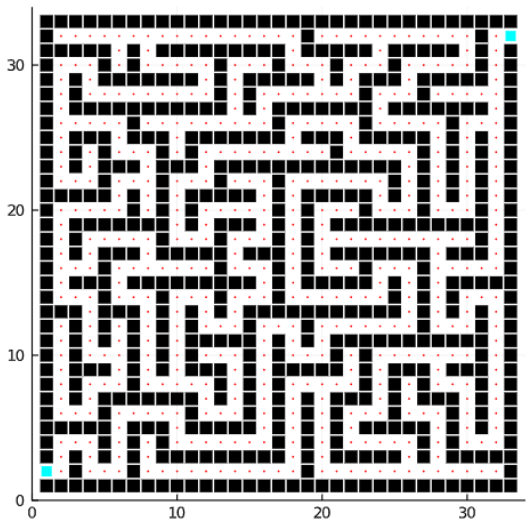
To start the maze, I needed to create a struct of type node. The node would hold the x and y position of the cell in the maze. I also needed to determine if the cell was a wall or a floor. I used a Boolean variable called walkable to store this value. Walkable if true would mean the cell is a floor, and if walkable is false the cell is a wall. Now I needed a struct of type grid to hold an array of nodes that would represent the maze. I would also need a width and a height property to define the size of the array that contains the nodes. The size of the maze would depend on the argument inside the constructor. The user can put a single number to generate a square shaped maze, or put two different values in to make a rectangle shaped maze. The next step is initializing the grids array of nodes. This process will loop through the dimensions passed through the constructor's argument. For each iteration of the loop a node is instantiated and its' position is set to the x/y index of the loop. Each node is also set as a wall, walkable = false, and the perimeter nodes are set to visited. Visited is a Boolean property used in backtracking to keep track if that node has already been visited so that we do not waste time and memory.

Joseph Rand
CSC 3001
Dr Staab
December 12, 2020

Now we can start the backtracking algorithm. A backtracking algorithm attempts to solve a given problem by testing all possible paths towards a solution until a solution is found. Each time a path is tested, if a solution is not found, the algorithm backtracks to test another possible path and so on until the stack is empty. To implement this, we need to create an empty list called stack. We will use the first in last out method to push and pop nodes from the stack. The stack keeps track of the nodes that have possible paths. We start by pushing the start node to the stack. Then, we start a while loop with an exit condition, so that when the stack is empty the loop will exit. In the while loop, we create a node variable called selected node and pop the stack to this variable. With the selected node, we can compare this node to the neighboring nodes to check for an available path. When a path is found, we push the node at the end of the path to the stack. We continue to do this for all available paths, then the loop repeats and the selected node is assigned from the popped node in stack. When the stack is empty, the maze is complete.

```
[6]: maze = MazeGenerator.createMaze(33)
     MazeGenerator.plotMaze(maze)
```

[6]:



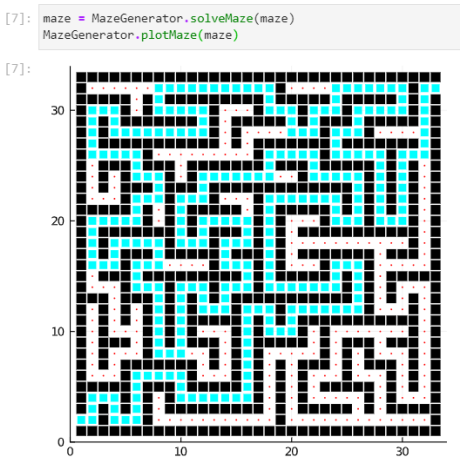
Joseph Rand
CSC 3001
Dr Staab
December 12, 2020

Next is the A* algorithm. The A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs.

Starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until open is empty, or a solution is found. To implement this, we first set the start and target nodes. I hard coded in the start nodes and target nodes to be opposite corners. Then we set the costs. The g cost is the distance between the node and the start node. The h cost or heuristics cost is the distance between the node and the target node. The f cost is the sum of h and g cost. Now that all the costs are set, we create an empty list called open which will store nodes that are ready to be processed and an empty list of nodes called closed which will store the processed nodes. We push the start node to the open list. Now in a while loop with the exit condition being that there are no nodes in open, we assign selected node to the nodes in open with the lowest f cost. If the f costs are the same then the h costs are compared instead. Now we push the selected node to the closed list and remove the selected node from the open list. If the selected node is the target node, then we exit the loop and the solution is found. Otherwise, we get a list of the selected nodes neighbors. For each neighbor node in the list of neighbor nodes, if the neighbor node is not in the closed list, and the node is either not in open or has a better cost than previously evaluated, then update the g cost of neighbor, set the parent of the neighbor to selected node and if it is not in the open list, add it to the open list. When either the target node is found or the open list is empty, the loop ends. If the target node is found, a solution exists. When the target node is found, the target node is sent into a loop where the exit condition of the loop is the node parent is equal

Joseph Rand
CSC 3001
Dr Staab
December 12, 2020

to null. While the parent is not null, push the selected node to a new list, called pathnodes. Then set selected node to the nodes parent, then repeat the loop. When the loop is finished, output the solution.



I was able to reach my goals for this project. The program randomly generates a maze, with given dimensions. I used a backtracking algorithm to generate the maze. I used an A* algorithm to solve the maze. The cool animation that shows a dot moving through the path to highlight the solution was code that I borrowed from the library called plots.