# MP1: Search - Lab Report

Bum Jun Kim (*bkim102*)        Samuel Hung (*shung5*)

# Part I: Basic Pathfinding

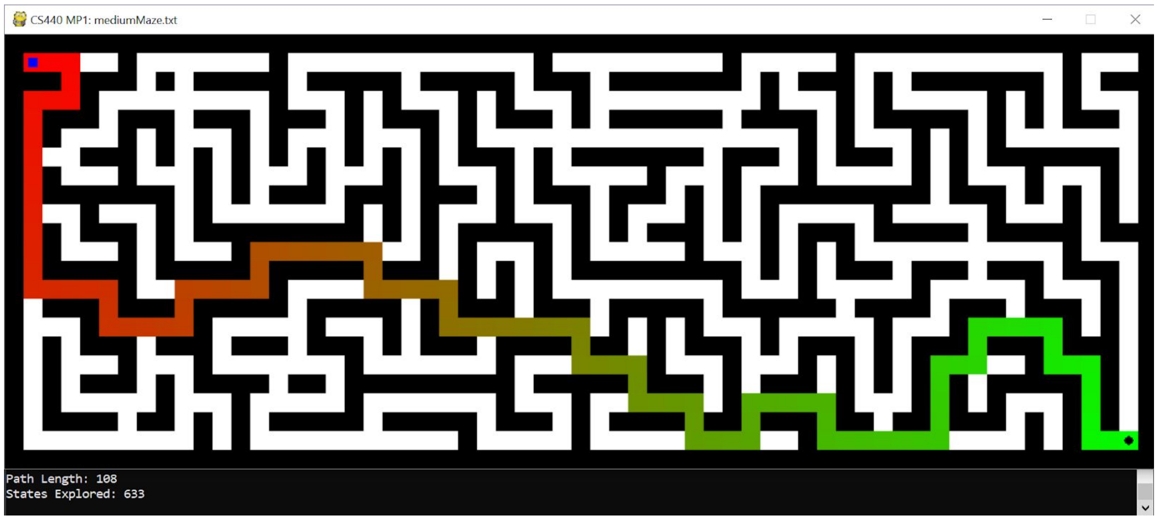
## Breadth-first Search (BFS)

**Implementation**:

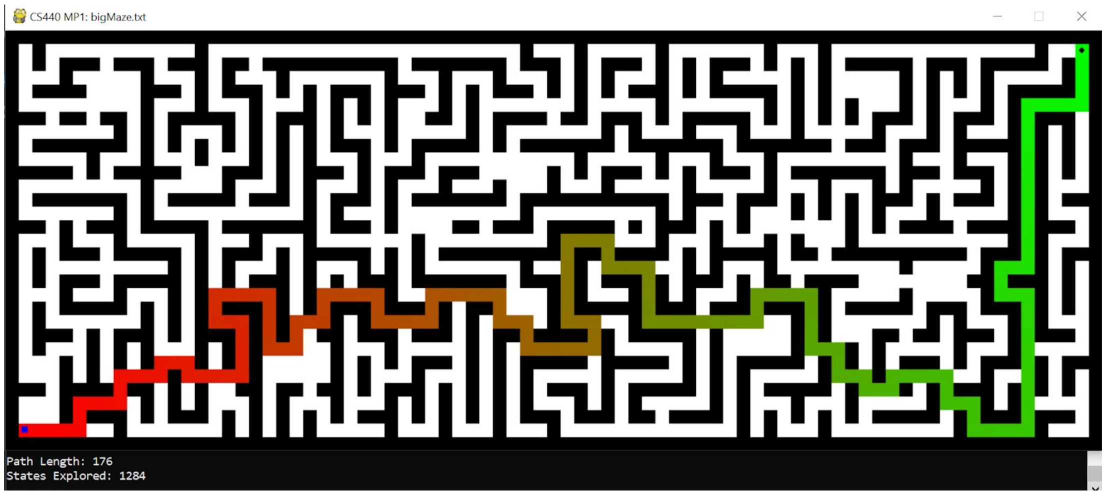
For our BFS algorithm, we use a queue to hold the current state. Once in the current state, we will check the neighbors of that particular state and see if they are valid, as well as unvisited. If so, those neighboring positions will be pushed onto the queue. The current position is then popped off of the queue and we move onto the next element. Having a queue ensures that the first elements that are pushed in get popped out first, allowing for the traversal of one depth level at a time. This process repeats until either no more elements exist inside the queue or the objective is found.

To get the solution path for the maze, we use a list of tuples to store the visited position as well as the position of the next move. We implemented a solution refiner which goes through this list of tuples and connects the objective position to the position before it, all the way back to the starting position. This connected position information is then saved to a seperate list and returned as our solution path.

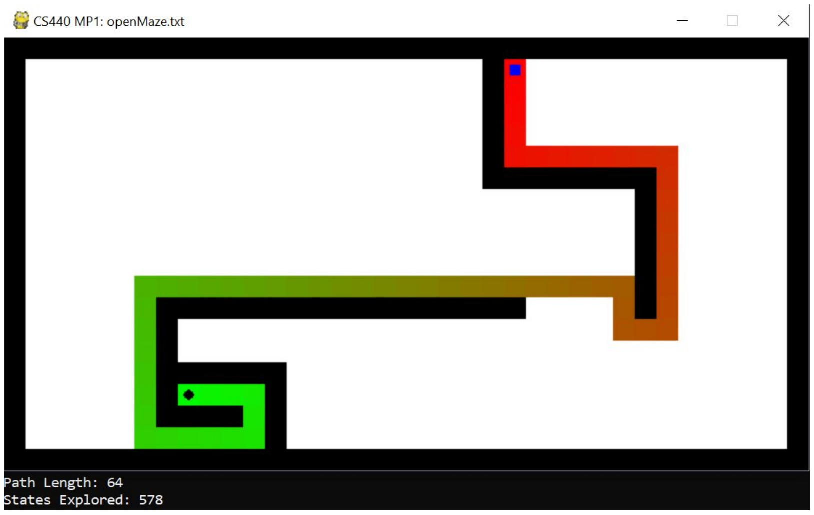
**Results:** (mediumMaze, bigMaze, openMaze) (table provided at the end of part I)



```
Path Length: 108
States Explored: 633
```



```
Path Length: 176
States Explored: 1284
```



```
Path Length: 64
States Explored: 578
```

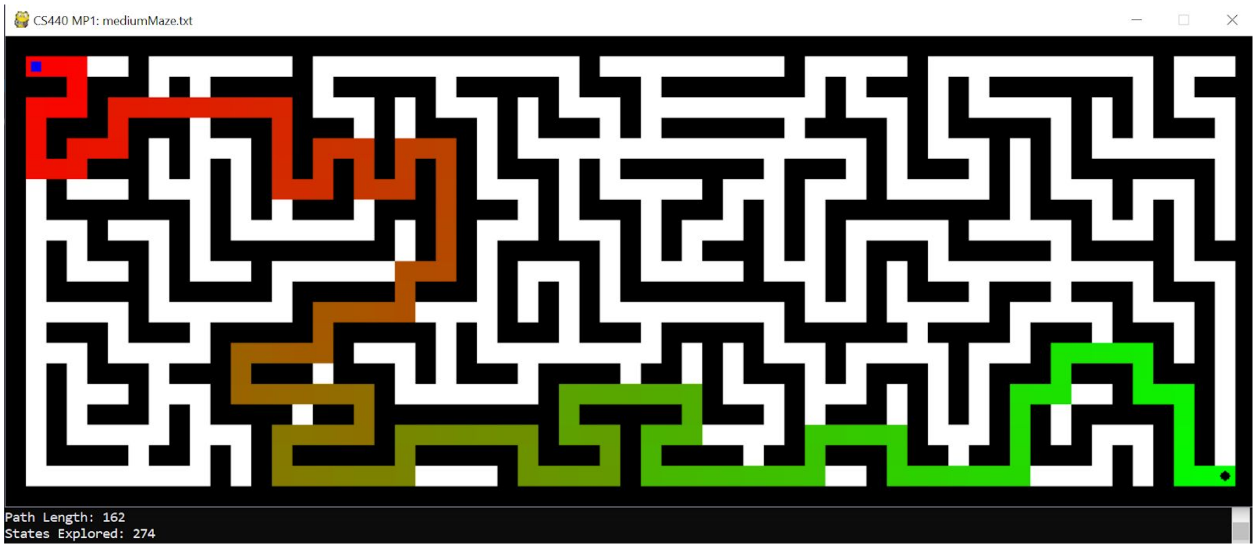
## Depth-first Search (DFS)

**Implementation:**

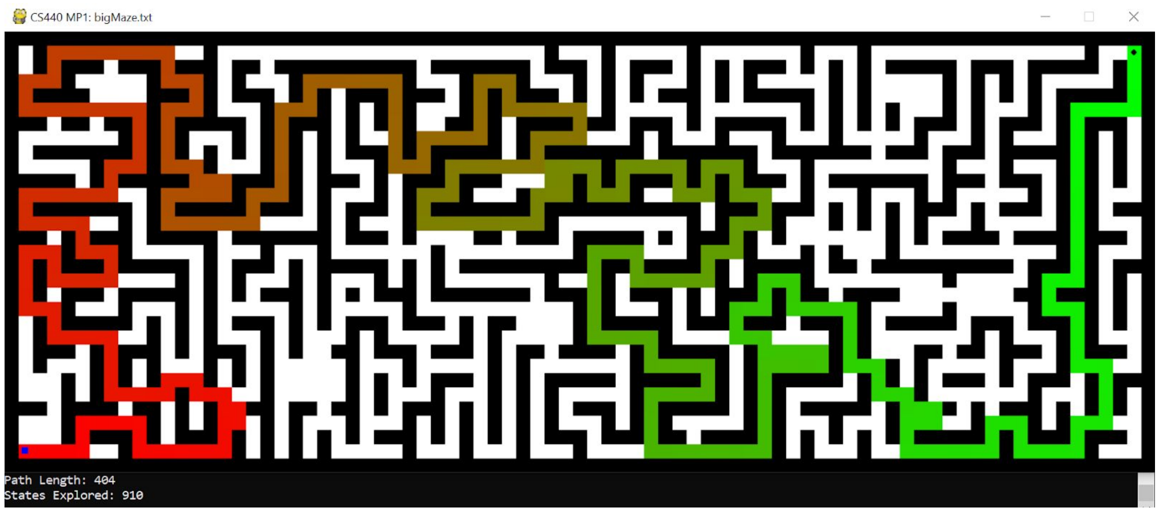
We implemented our DFS algorithm for the maze using a stack. For every current node, we look at its unvisited neighbors and push them onto the stack. The current element gets popped off and we move on to its neighbor, repeating the process until the stack is empty or until the objective is reached.

Stacks are LIFO so that means the element that gets pushed get processed immediately after the current element. This is how the algorithm is able to traverse downwards in depth every iteration rather than in a lateral fashion like BFS. Like in the BFS algorithm, we store the visited nodes in a list as a tuple containing that current position along with its next position. That way the next position can be used to trace a solution path. This information is passed into our solution refiner function which then saves a connected path into a separate list, which is returned as our solution.

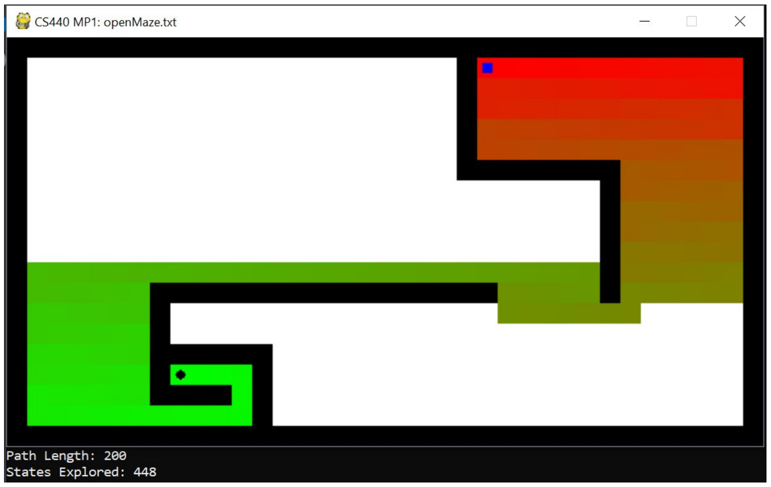
**Results:** (mediumMaze, bigMaze, openMaze) (table provided at the end of part I)



```
Path Length: 162
States Explored: 274
```



```
Path Length: 404
States Explored: 910
```



```
Path Length: 200
States Explored: 448
```

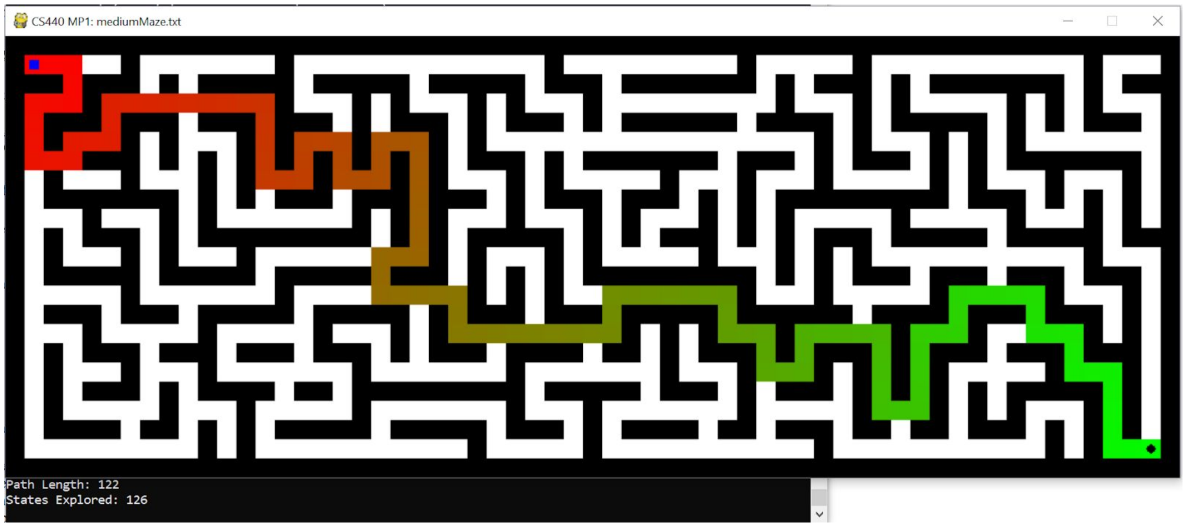
## Greedy best-first Search

**Implementation**:

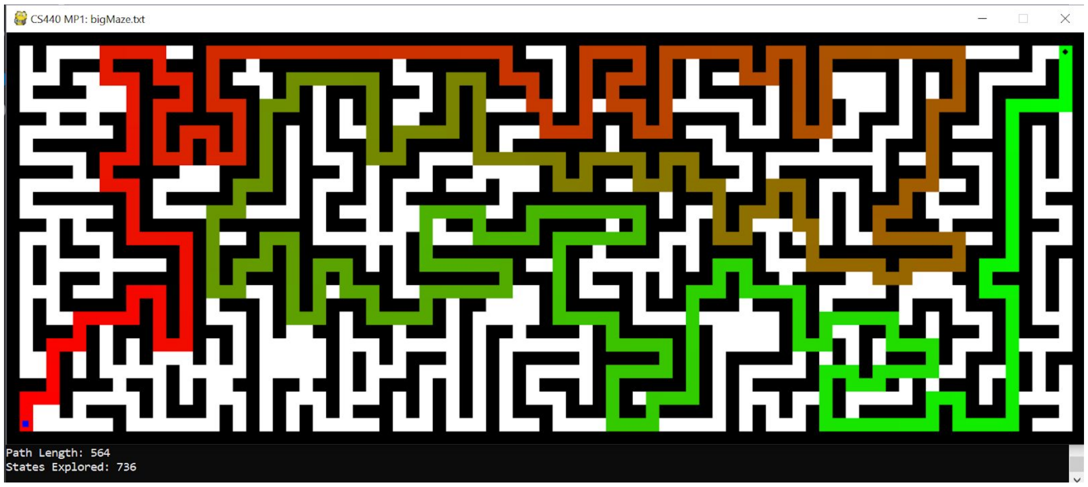
In our Greedy best-first search, we used a stack to hold the unvisited neighboring positions per step.  In addition, we also stored the Manhattan Distance from each of those states to the objective.  Using those distances, we sort the list of neighbors and push that list to the stack, in the order where the neighbor with the best heuristic gets processed first.

Like in our BFS and DFS implementations, we also use lists to store previously visited states, as well as the state following it.  The solution refiner method is used again here to connect the dots from the solution all the way back to the beginning, forming a solution path that is returned.

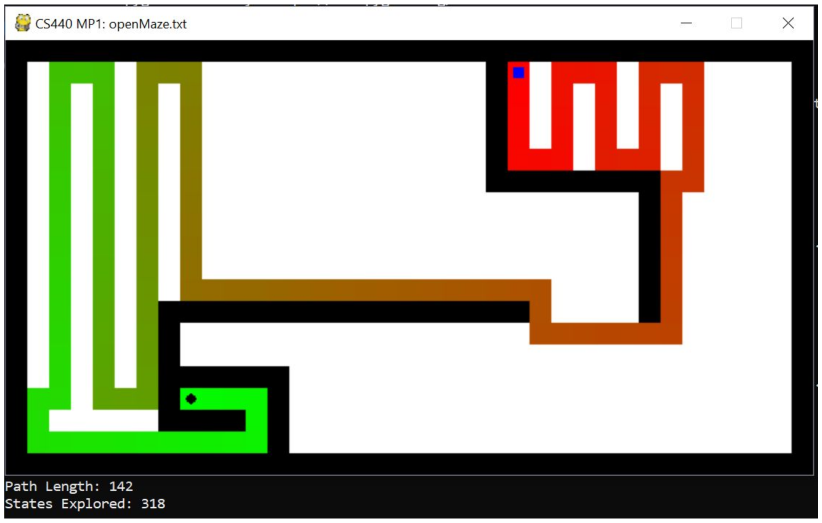
**Results:** (mediumMaze, bigMaze, openMaze) (table provided at the end of part I)



```
Path Length: 122
States Explored: 126
```



```
Path Length: 564
States Explored: 736
```



```
Path Length: 142
States Explored: 318
```

## A* Search

**Implementation**:

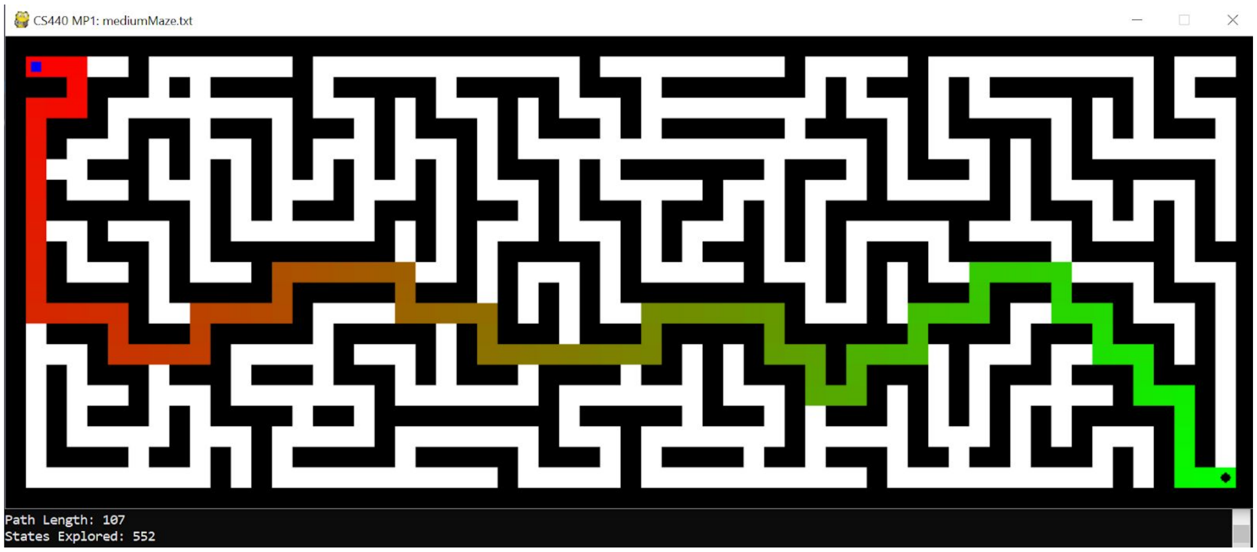
Let’s define f to be the function where **f** = **g** + **h**, where **g** is the distance taken to get to the current node and h is the remaining distance heuristic.

We implemented using a stack to keep (next position, **g**, **f**, path)  just like how the DFS was implemented, except we sorted the list according to the **f** value in the stack, so the first to be popped next would be the one with the smallest value of **f**.

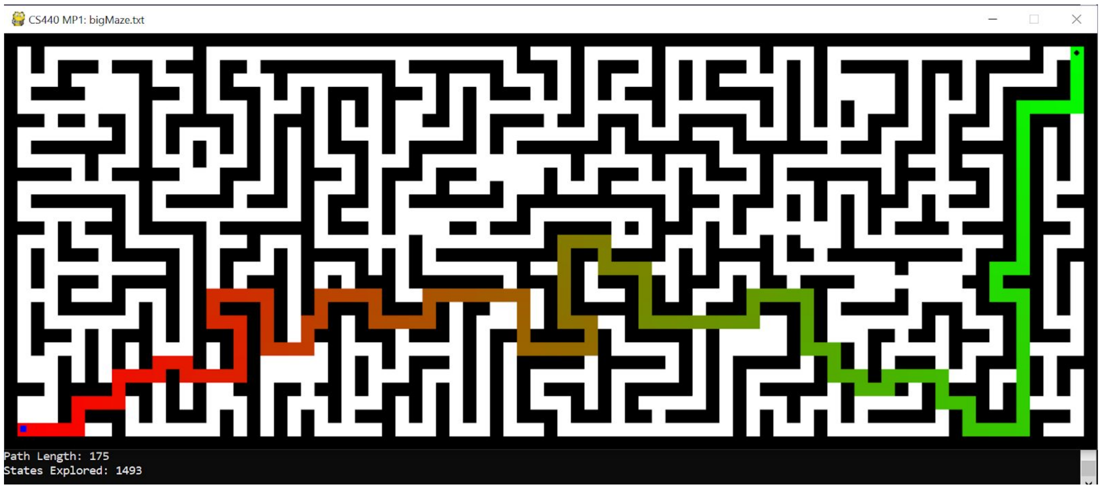
Code structure is essentially the same as in the previous algorithms.  We have a outer loop that does computations for the current node.  There is a for loop that processes each available neighbor and adds it to a data structure (a stack for the case of A*).

As mentioned above, this stack stores four elements: the position, distance taken to the current path, the distance to current path + Manhattan Distance (**g** + **h**), and the path taken to get to the current position of the maze.
**g** (distance to the current path) was calculated by adding 1 to the previous position’s **g**. **h** was calculated by the Manhattan Distance from the goal. We stored the path leading up to that current position for each state, so when we reach the final position, it would automatically give us the best path we found since we are constantly choosing the option with the lowest cost.

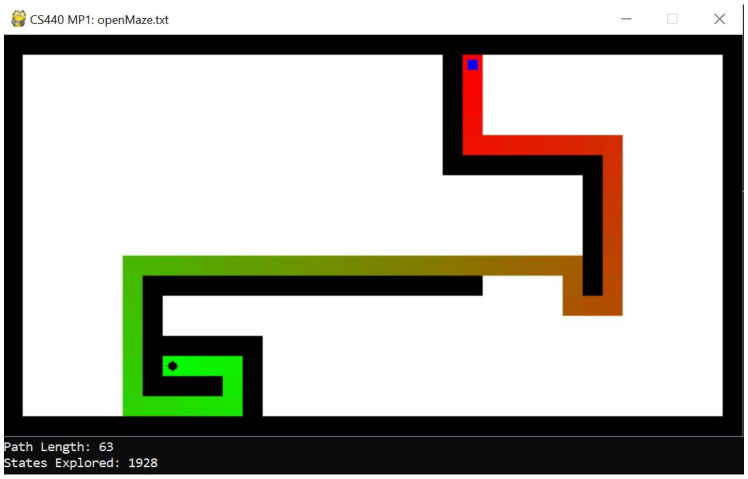
**Results:** (mediumMaze, bigMaze, openMaze) (table provided at the end of part I)



```
Path Length: 107
States Explored: 552
```



```
Path Length: 175
States Explored: 1493
```



```
Path Length: 63
States Explored: 1928
```

## Part I: Final Results

| **Path Length** | mediumMaze | bigMaze | openMaze |
|---|---|---|---|
| BFS | 108 | 176 | 64 |
| DFS | 162 | 404 | 200 |
| Greedy | 122 | 564 | 142 |
| A* | 107 | 175 | 63 |

| **States Explored** | mediumMaze | bigMaze | openMaze |
|---|---|---|---|
| BFS | 663 | 1284 | 578 |
| DFS | 274 | 910 | 448 |
| Greedy | 126 | 736 | 318 |
| A* | 552 | 1493 | 1928 |

# Part II: Search with Multiple Dots

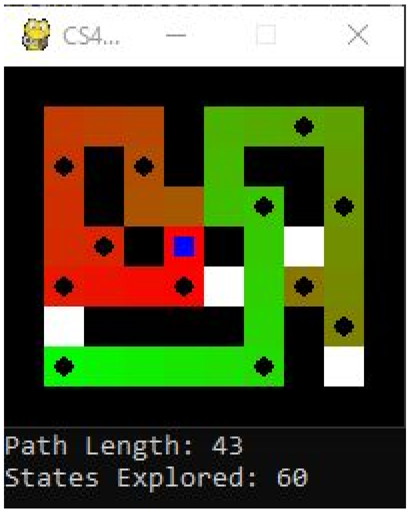
**Implementation**:

Building upon the A* search for a singular target, we store a static list of all the objectives as well as a dynamic list of the objectives that supports the removal of an objective after it has been reached. For each objective remaining in the dynamic list, we calculate the Manhattan distance from the current position to that objective.

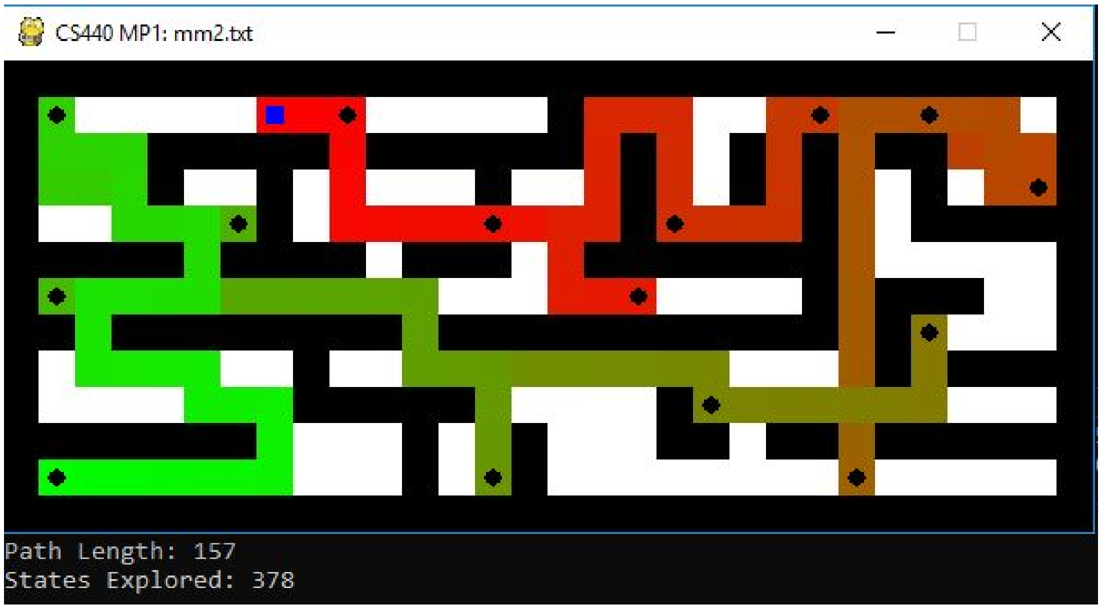
Within this loop, we also keep track of the minimum value — we use this value to help us select the path we will take. In this loop, we also store the path up to that node using a two-dimensional list. Admittedly, this is very memory intensive; however, this allows us to have immediate access to the solution path when the final objective is reached.

As for book-keeping, we have two lists — the first list holds all of the visited nodes, while the second does the same for each objective but clears itself after an objective is reached. This allows us to go back to previously visited states in case we need to use them again as part of our solution path.

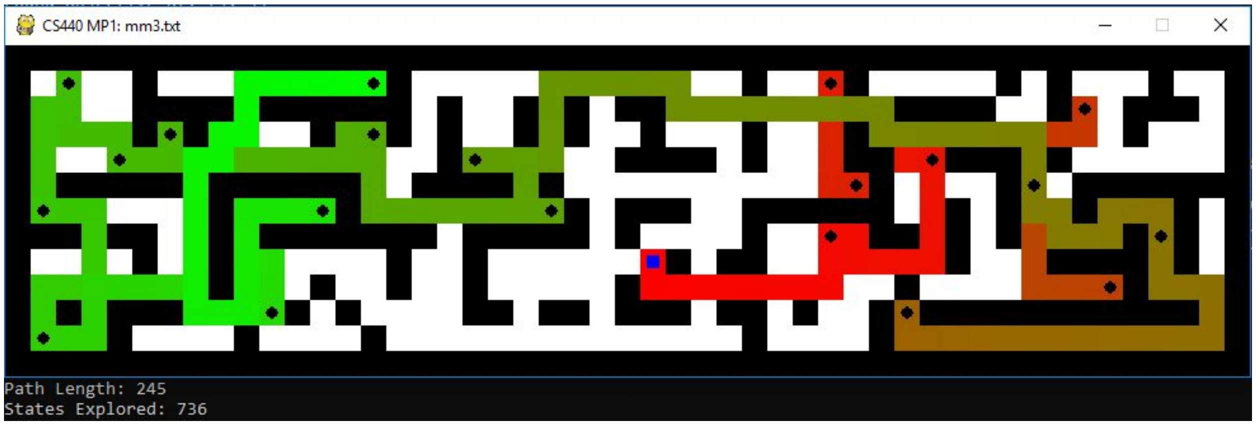
**Results:**(tinySearch, smallSearch, mediumSearch)



```
Path Length: 43
States Explored: 60
```



```
Path Length: 157
States Explored: 378
```



```
Path Length: 245
States Explored: 736
```

| **A*** | tinySearch | smallSearch | mediumSearch |
|---|---|---|---|
| Path Length | 43 | 157 | 378 |
| State Explored | 60 | 378 | 736 |

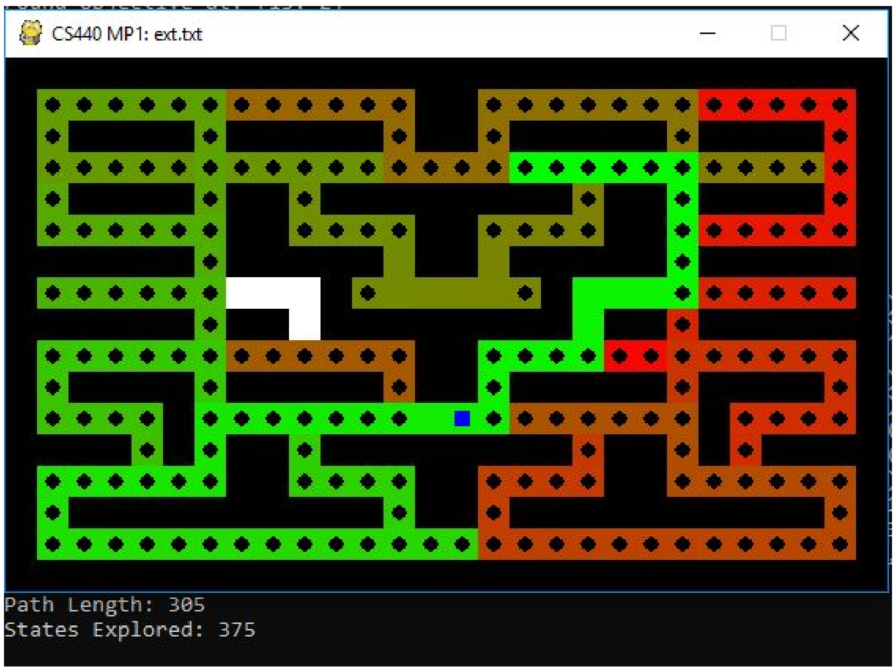
## Extra Credit

**Implementation:**

Adding to the A* search for multiple dots, we implemented a linear scaling on the heuristic depending on the wall density of the maze(f = 20*density*h + g). The wall density is simply just the number of walls in the maze divided by the total maze area.  The idea is that the denser the maze is, there becomes a higher possibility of running into complications and at a much higher frequency.  Thus, denser maps often result in a longer solution path to the goal. We concluded that scaling (upwards) the expected path length to reach the goal seemed reasonable.

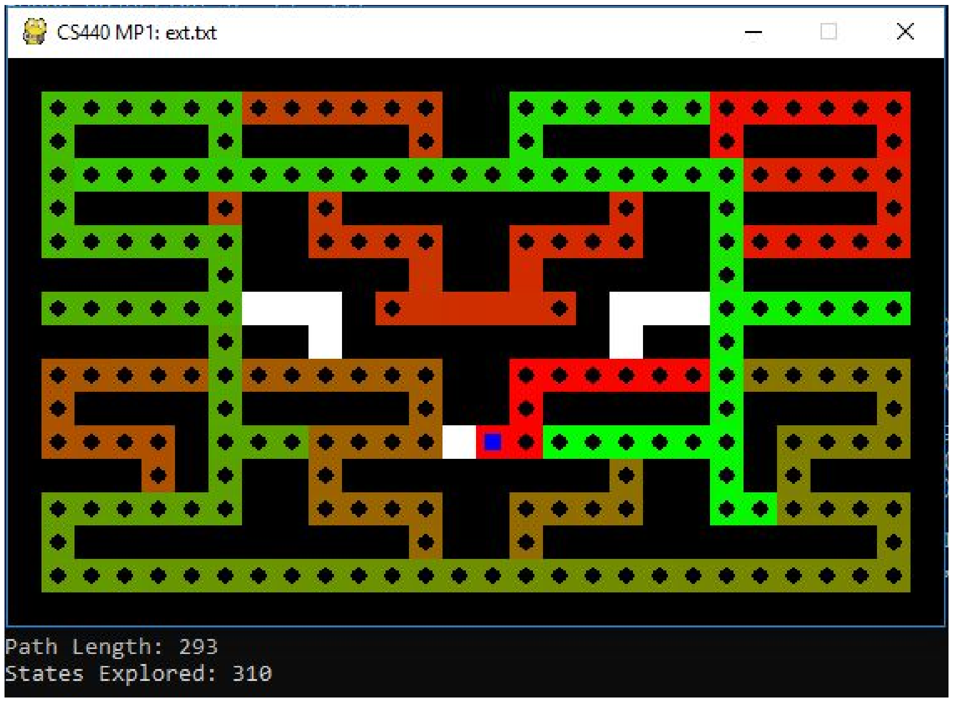
The results from the bigDots maze show that the heuristic had a decrease on both the path length and states explored. As the maze scaled from  sizes tiny, small, to medium, it resulted in not only decreasing the path length but it also significantly reduced the states explored. The map density heuristic proves to reduce the number of states explored by 50% for the larger mazes like bigDots.

**Results:**



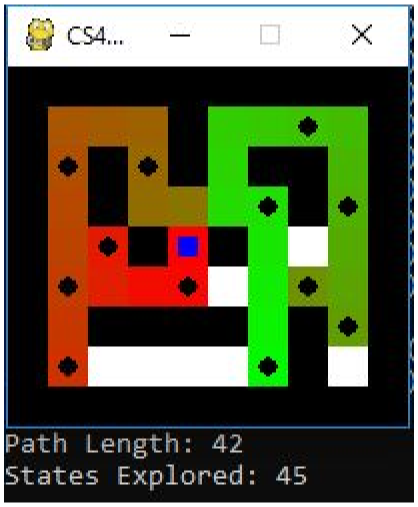
```
Path Length: 305
States Explored: 375
```

bigDots (admissible)



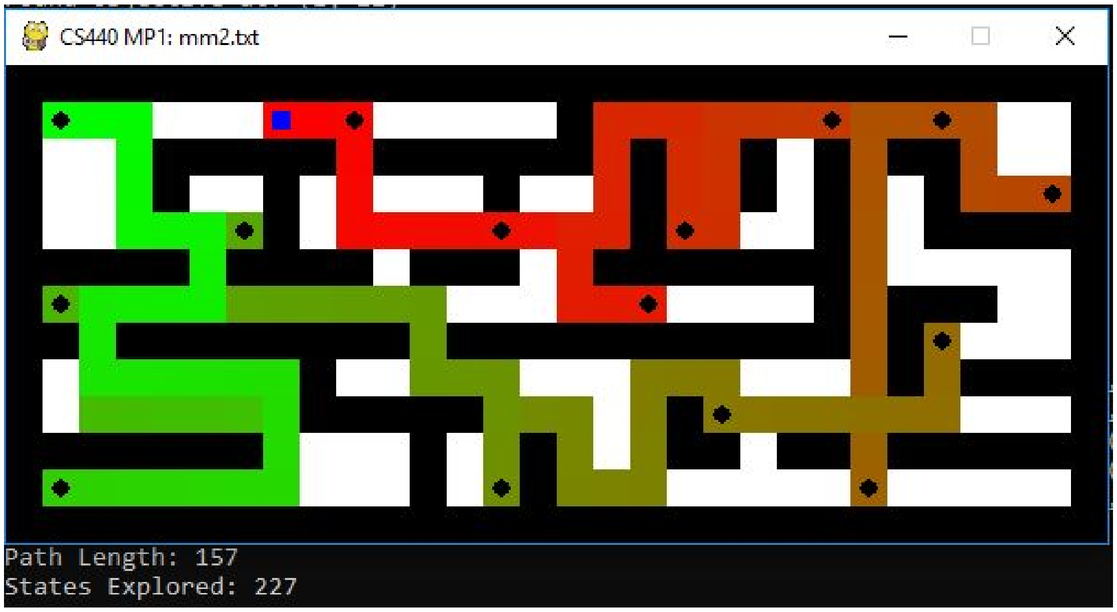
```
Path Length: 293
States Explored: 310
```

bigDots(non-admissible)



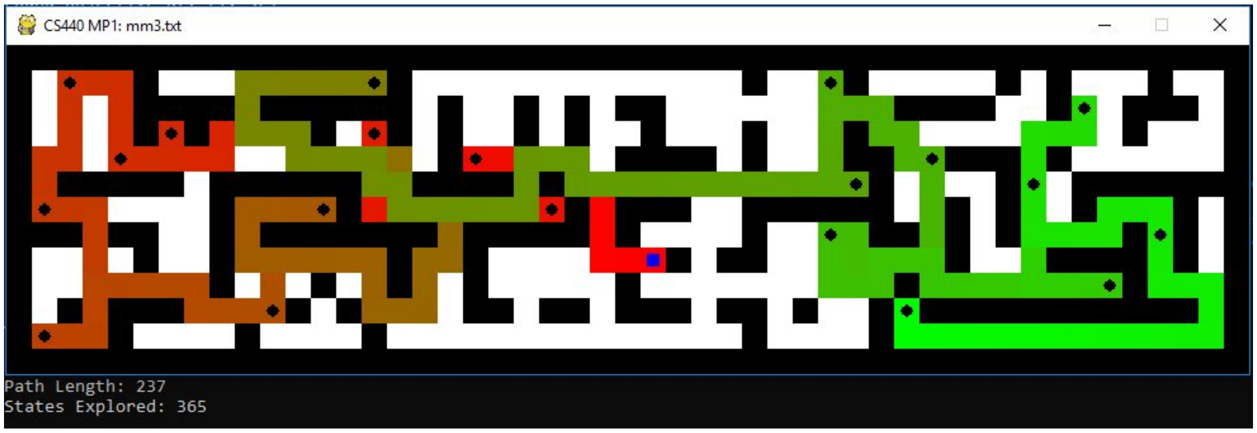
```
Path Length: 42
States Explored: 45
```

tinySearch (non-admissible)



```
Path Length: 157
States Explored: 227
```

smallSearch (non-admissible)



```
Path Length: 237
States Explored: 365
```

mediumSearch (non-admissible)

| | tinySearch | | smallSearch | | mediumSearch | |
|---|---|---|---|---|---|---|
| | path | SE | path | SE | path | SE |
| A* | 43 | 60 | 157 | 378 | 245 | 736 |
| A* with non-admissible heuristic | 42 | 45 | 157 | 227 | 237 | 365 |

*(SE = State Explored)*

## Statement of Individual Contribution

**Sam**: *implementing BFS, DFS, methods to refine solution path for BFS-DFS-Greedy, map-density heuristic, overall code structure.*

**Bum Jun**: *implementing A* search, modifying for multiple dots, methods of producing the final shortest path and other statistics.*

Overall, we think we worked well as a group and we both spent an equal amount of effort and time into this assignment.