MP1 Report

Course: ECE448-LE1

Team: Yuhang Chen

Jiakai Lin

Wenbo Ye

Date: Mar. 11th, 2023

**Section I. Algorithm (Search):**

For dfs, we use recursions to implement the exploration by just randomly choose a direction to explore, returning with the path once the objective is found or return to trace back to the start state to explore in other direction. We use a list called path to serve as stack to record the path the agent is on. Each point in the path is a node. Each node is linked with its state (visited or unvisited). The frontier is the previous node of the current node in the path, where we can pop the path to trace back to get it. And the states are managed by the 2-dimension list called visited. A True/False value in the visited indicates whether a state has been detected.

For bfs, we use the loop to find the objective until all of the states is explored. (In the test there are at least one objective exiting in the map) We use a queue to record the path. And we also use a dictionary to implement the bfs tree. The key-to-value can record the frontier (father of the node) of each node. We maintain the explored states list in the visited queue.

For greedy, the algorithm is almost the same with dfs. However, to determine in which order to do the recursion we use Manhattan distance as the heuristic number to help the agent to judge which direction to go first. Therefore, we use a queue to represent the search states, and use dictionary as the component in the queue. The key of the dictionary represents the heuristic number. And we can just sort the queue by sorting the key each time before to determine which new state to explore.

For astar, beyond Manhattan distance we also use the cost to the current node. By adding these two value together to serve as the judgment argument to determine the new state to explore. We also maintain the visited states in a queue. The frontier is a queue that contains explored nodes with unexplored neighbors. We can trace back with the nodes in the frontier until we get the objectives.

**Section II. Algorithm (A\*, Greedy BFS):**

The search algorithm for a single dot is described in Section I.

For the multiple dots, we use the prim algorithm to generate the minimum distance tree. By minimum distance tree (MST) we can find the relative shortest maze route. To get MST, after one objective is found, we calculate the minimum distance among all the distances between the agent and the objective. Then we choose that relative objective with minimum distance as the next objective to go. Hence, we can get MST and use MST to get all the objective dots in the maze.

**Section III. Results (Basic Pathfinding):**

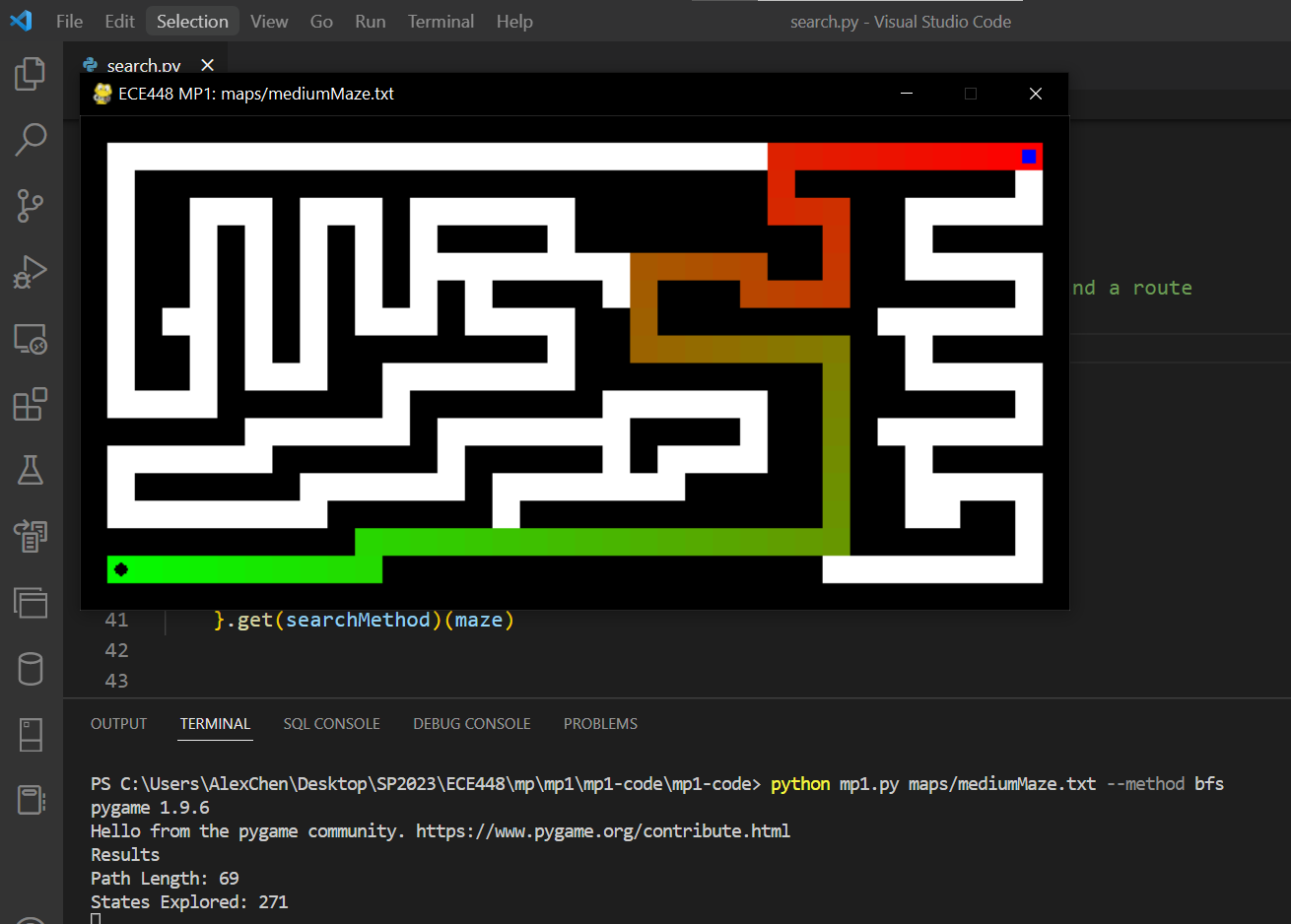
1. **DFS**

Figure 1. The screenshot of Pathfinding, Path Length and States explored by bfs in mediumMaze

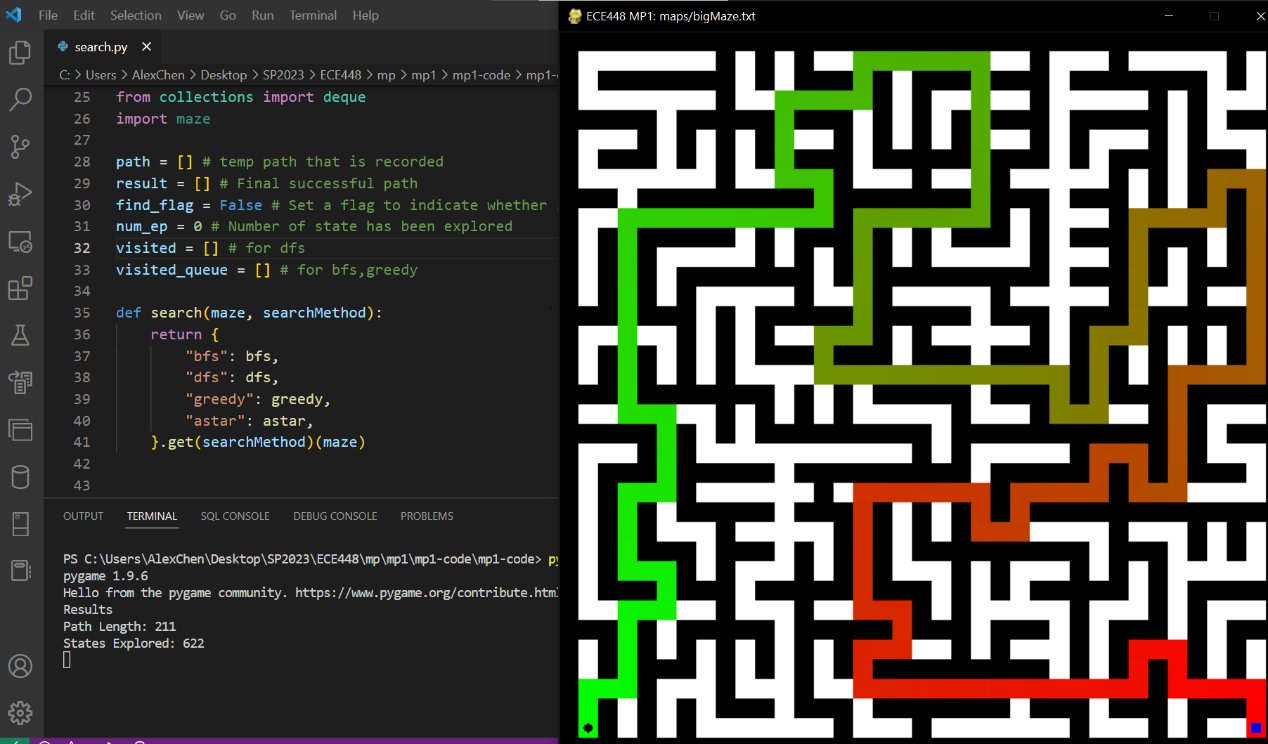
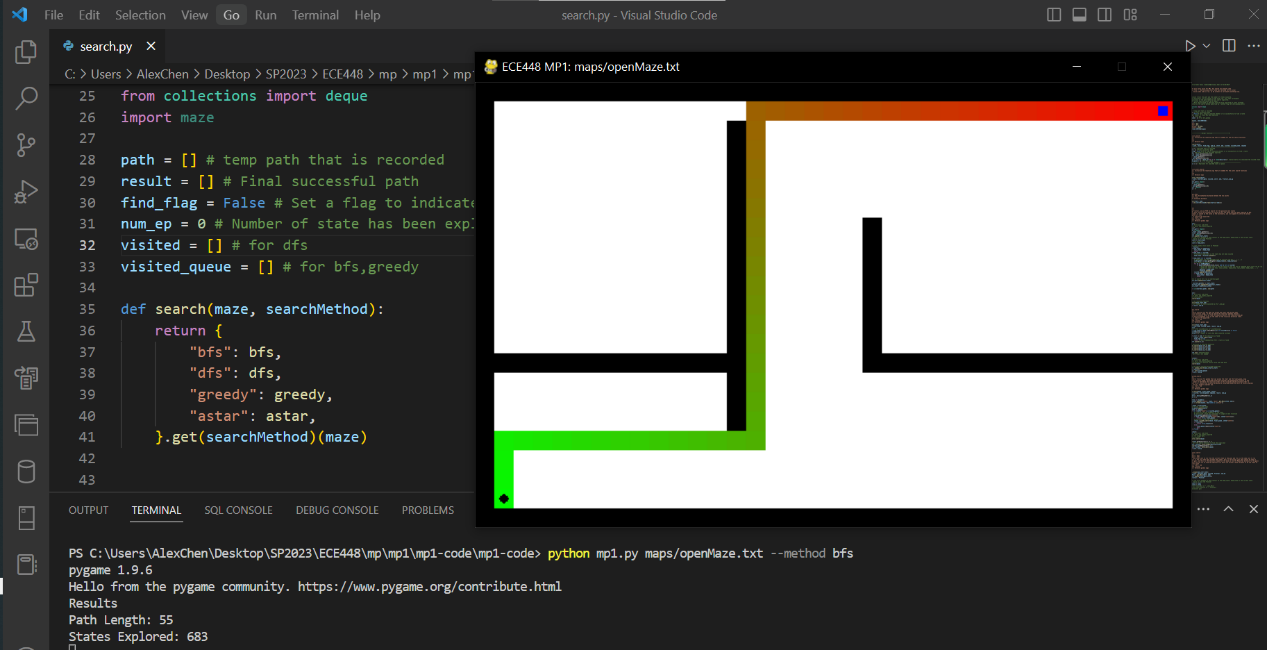
****

Figure 2. The screenshot of Pathfinding, Path Length and States explored by bfs in bigMaze

Figure 3. The screenshot of Pathfinding, Path Length and States explored by bfs in openMaze

****

1. **BFS**

Figure 4. The screenshot of Pathfinding, Path Length and States explored by dfs in mediumMaze

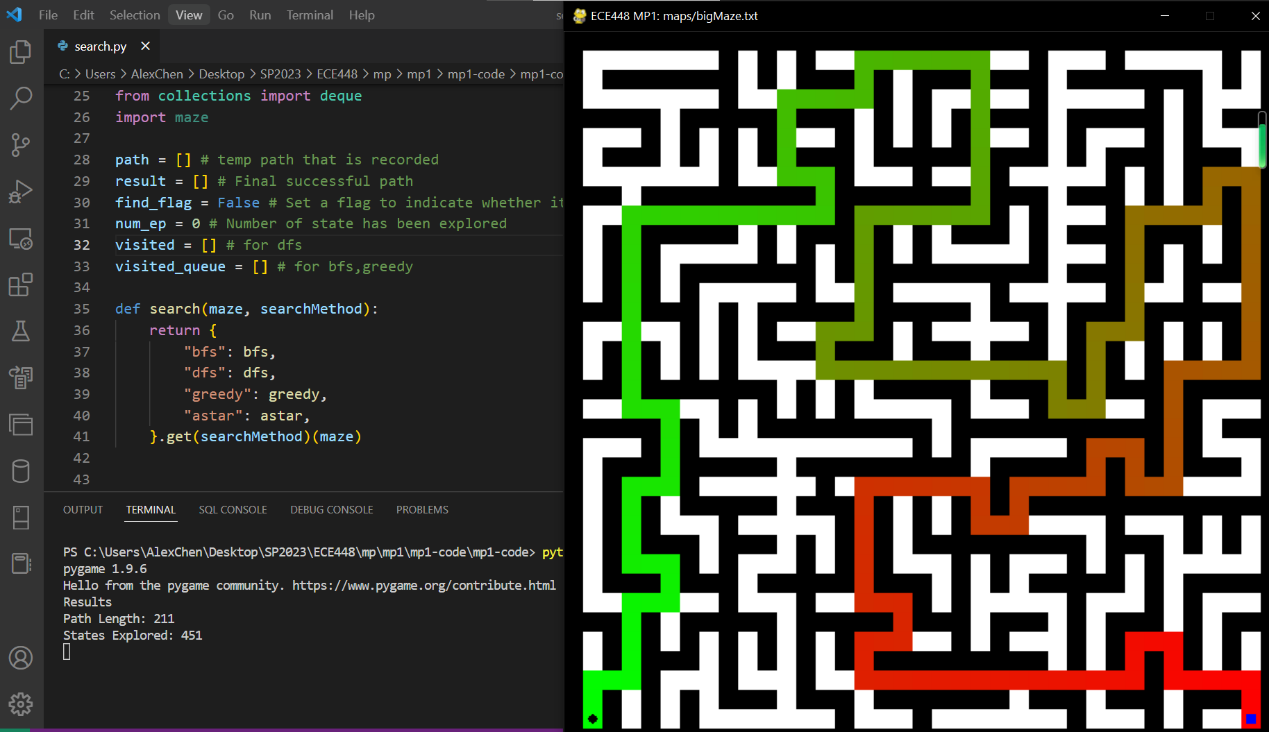
****

Figure 5. The screenshot of Pathfinding, Path Length and States explored by dfs in bigMaze

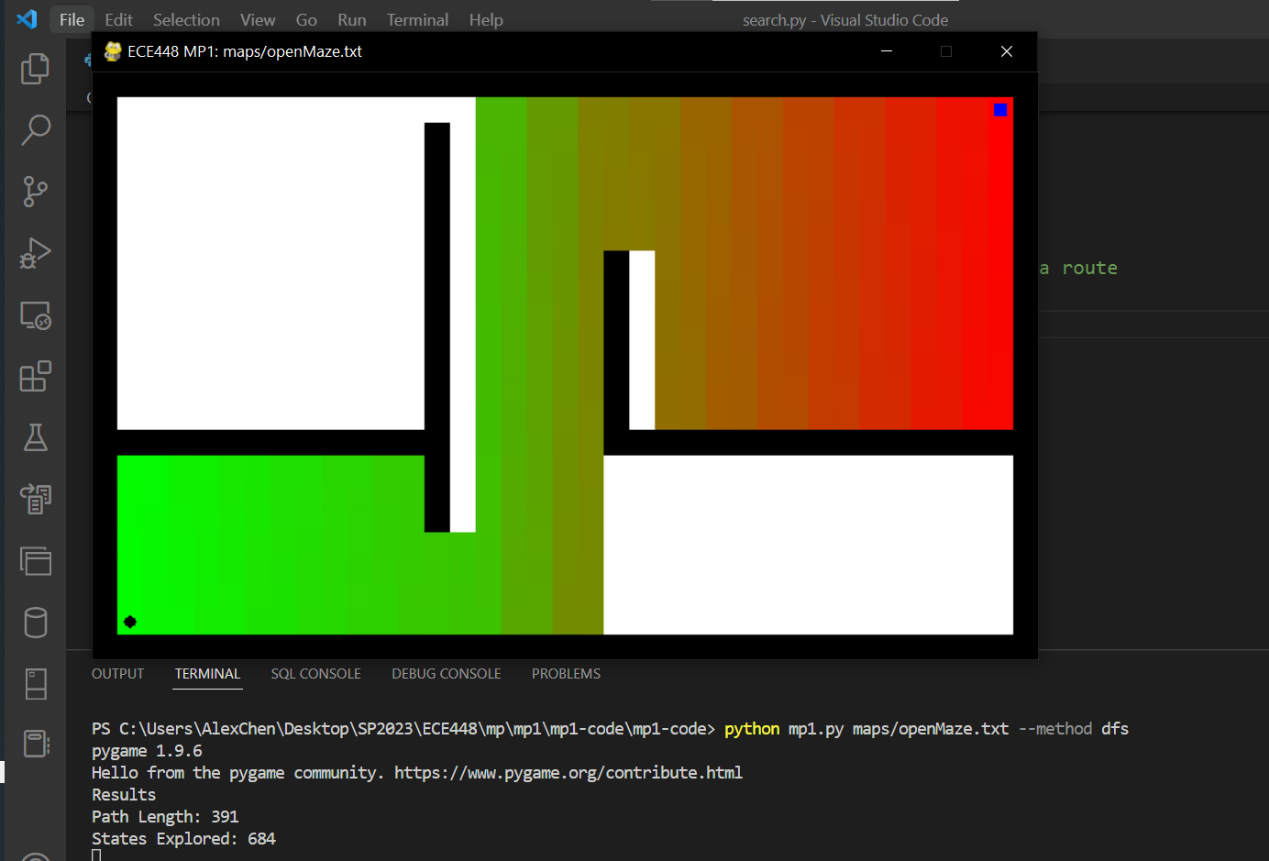
****

Figure 6. The screenshot of Pathfinding, Path Length and States explored by dfs in openMaze

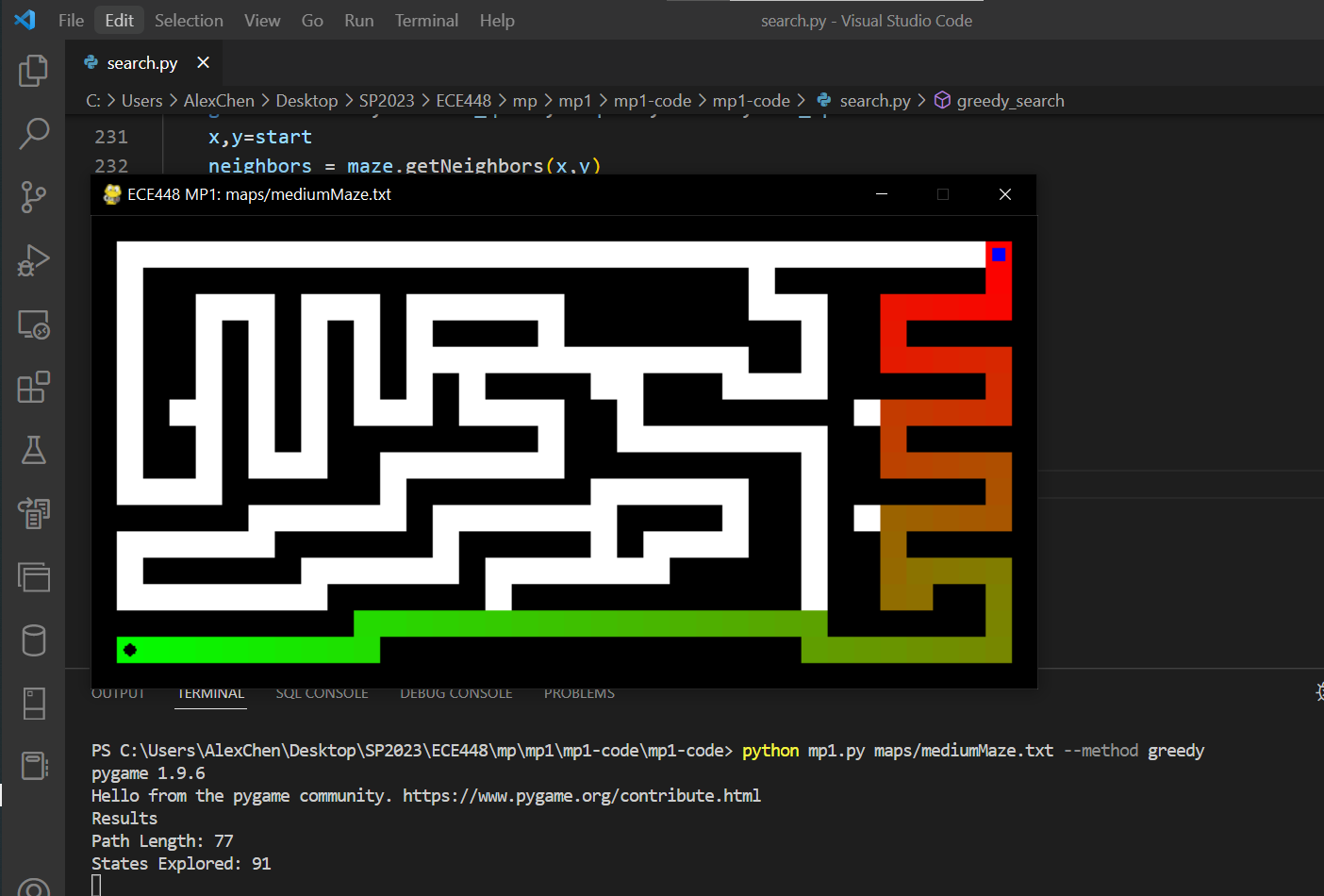
1. **Greedy**

Figure 7. The screenshot of Pathfinding, Path Length and States explored by greedy in mediumMaze

****

Figure 8. The screenshot of Pathfinding, Path Length and States explored by greedy in bigMaze

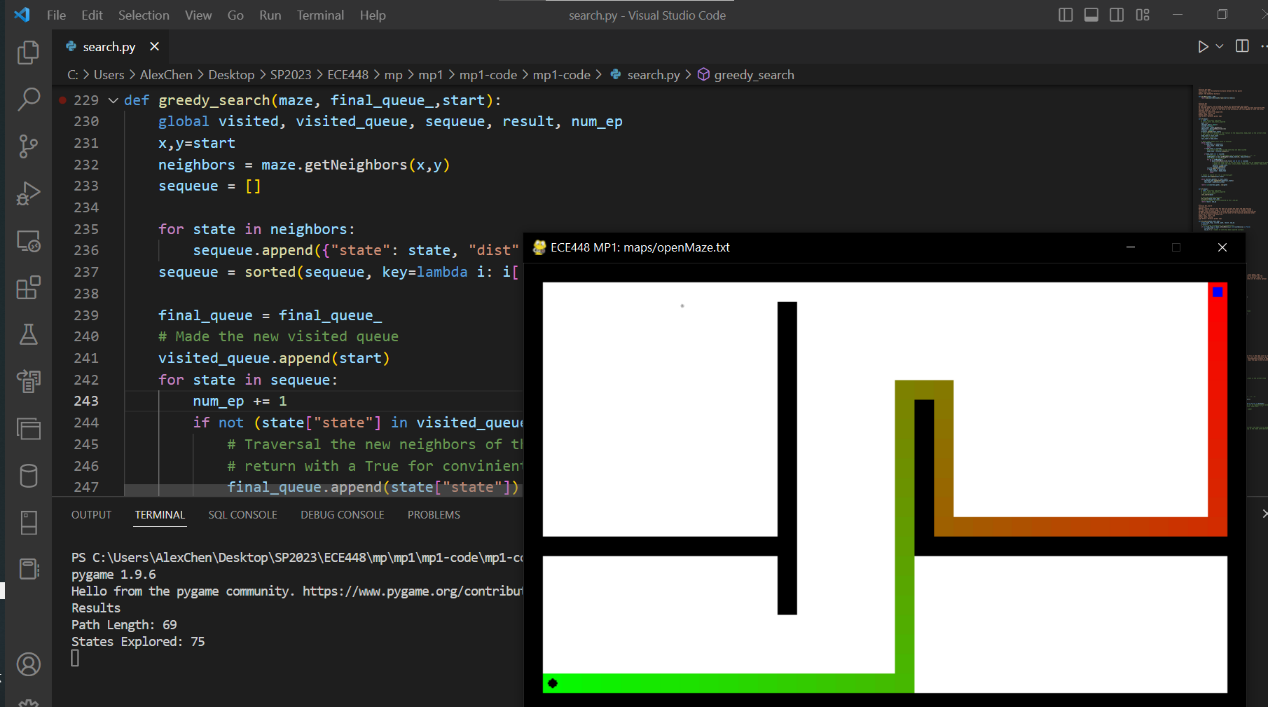
****

Figure 9. The screenshot of Pathfinding, Path Length and States explored by greedy in openMaze

1. **Astar**

Figure 10. The screenshot of Pathfinding, Path Length and States explored by astar in mediumMaze

****

Figure 11. The screenshot of Pathfinding, Path Length and States explored by astar in bigMaze

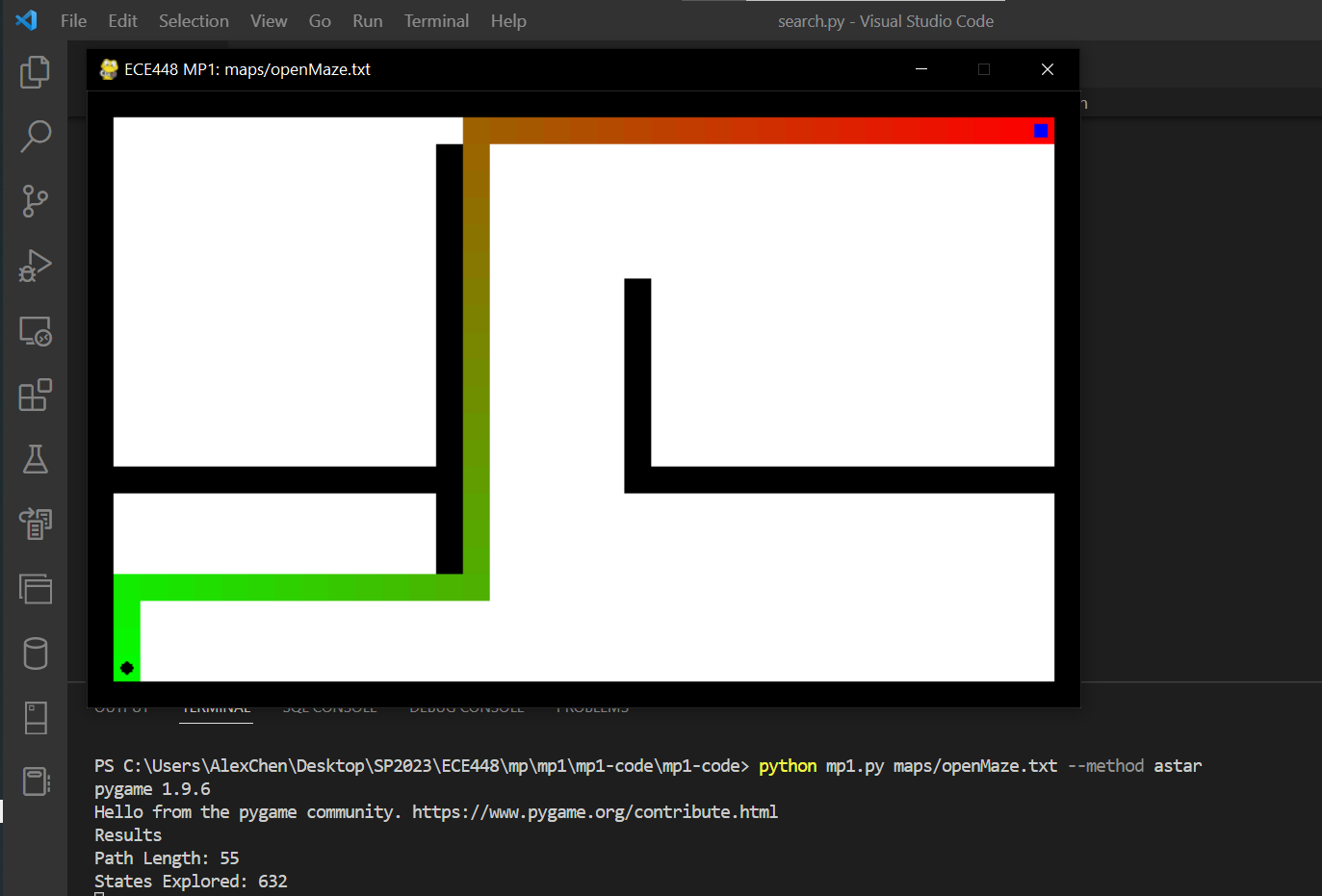
****

Figure 12. The screenshot of Pathfinding, Path Length and States explored by astar in openMaze

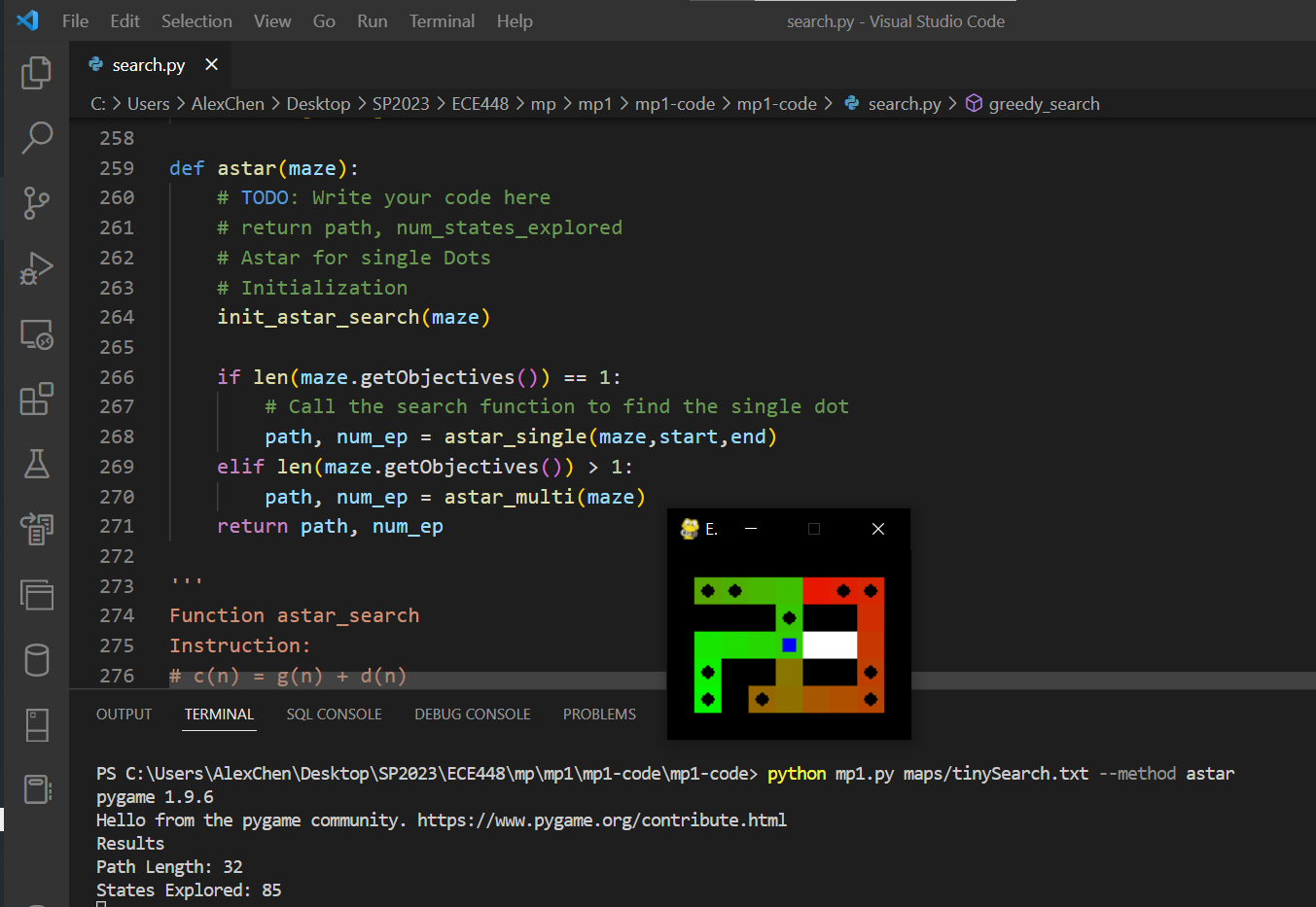
**Section IV. Results (Search with multiple dots):**

Figure 13. The screenshot of Pathfinding, Path Length and States explored by astar in tinySearch

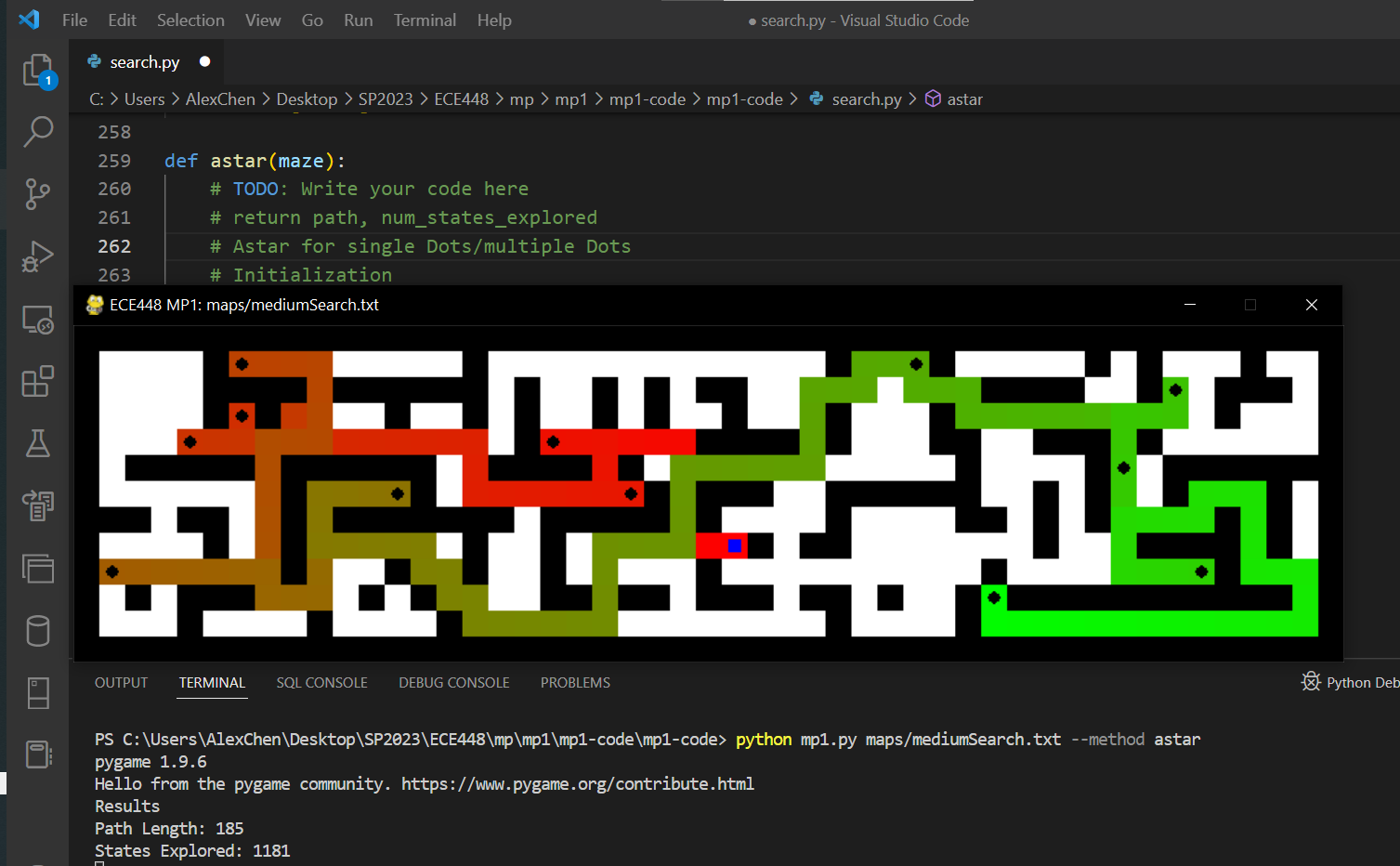
****

Figure 14. The screenshot of Pathfinding, Path Length and States explored by astar in mediumSearch

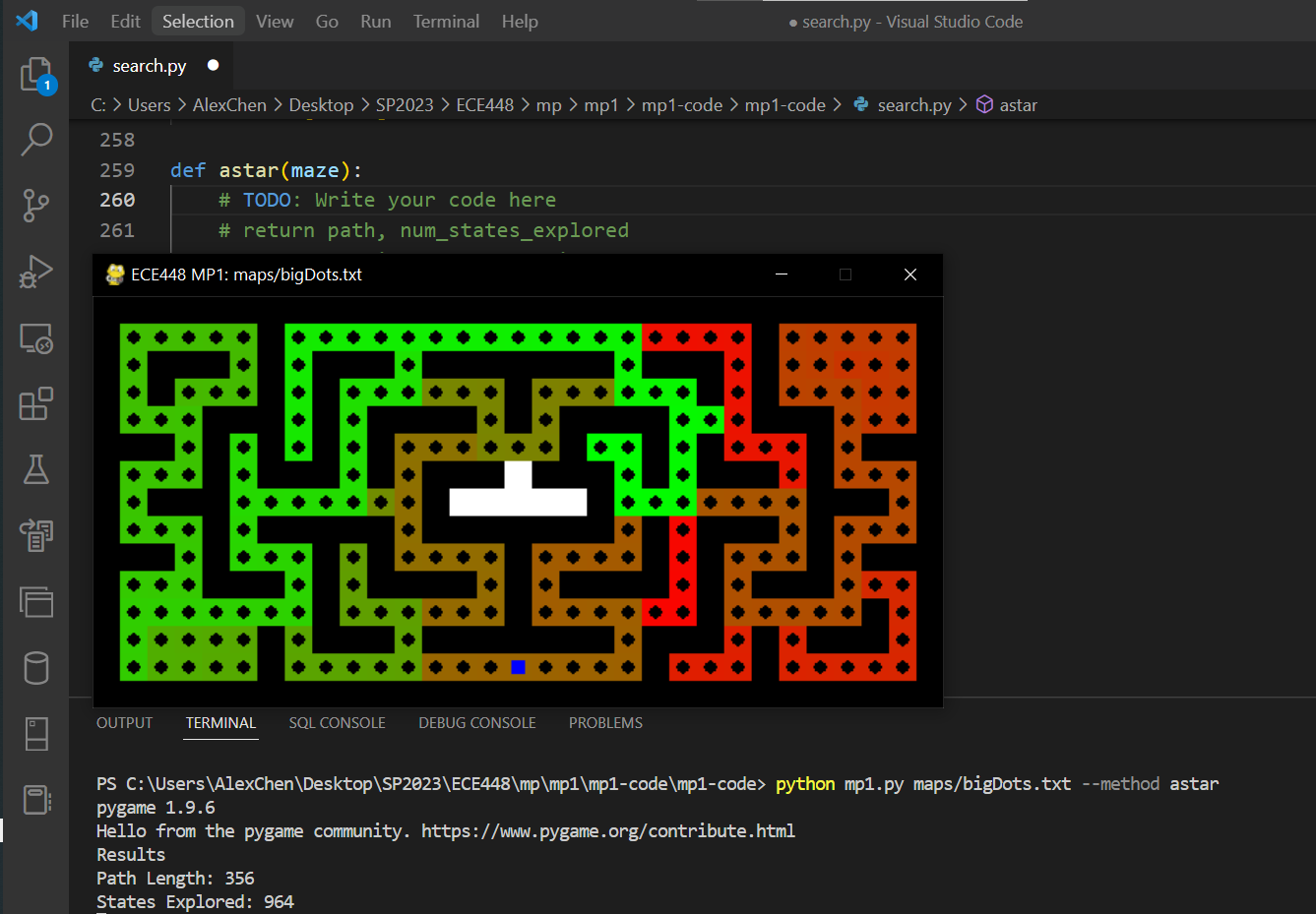
****

Figure 15. The screenshot of Pathfinding, Path Length and States explored by astar in bigDots

**Extra Credit:**

For the extra part, we think that we can optimize the A\* algorithm by changing weight of the heuristic number in the calculation of the idea cost (fn)

For example, to search multiple dots in the maze, the agent need to go through the same route more than one times. Therefore the weight of hn need to be increased (where we use MST to represent hn) And we use twice of the hn to replace hn in the original formular. fn = hn +cn => fn = 2\*hn + cn

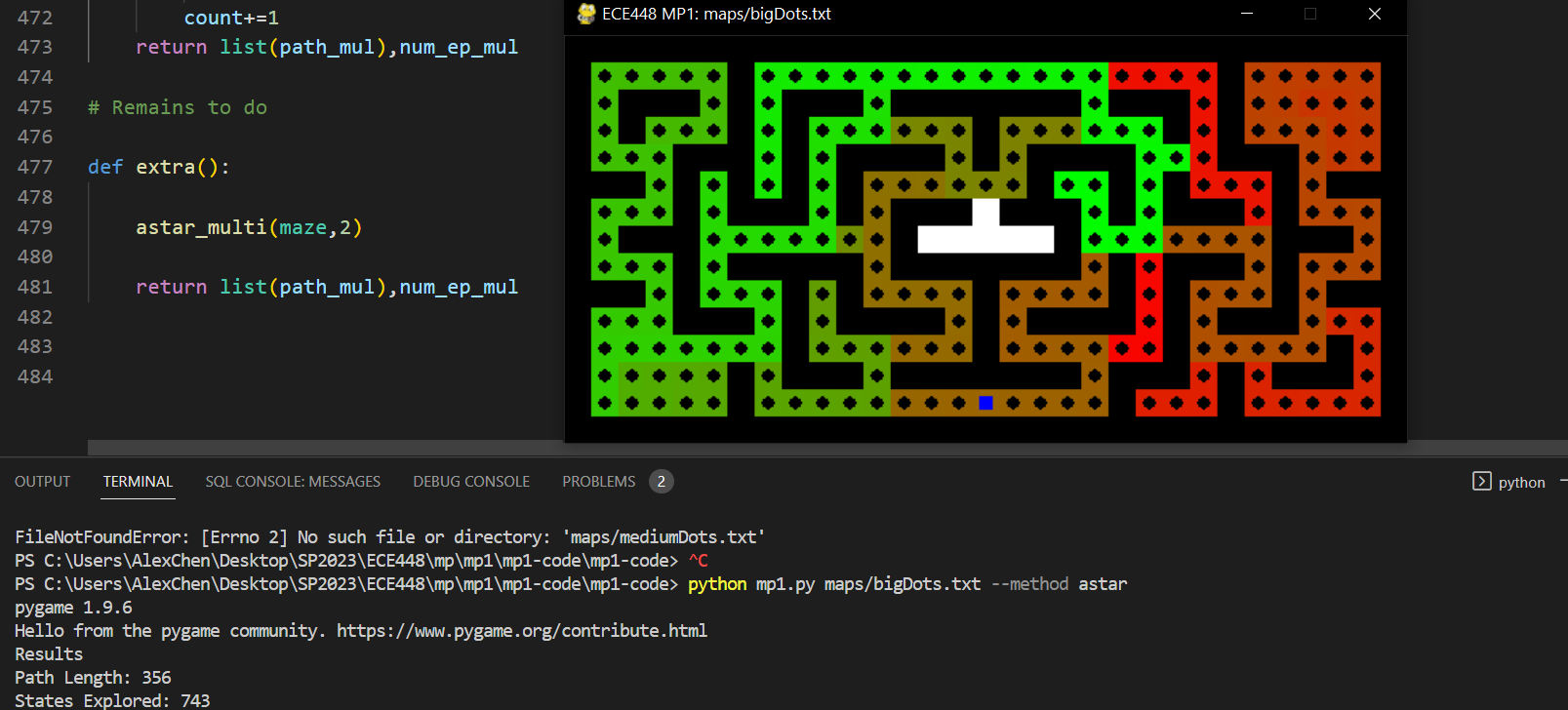
****The code’s running result shows as follow. As we can find, the state explored decreased. Hence, the algorithm to be applied in the task like BigDots is optimized.

Figure 16. The screenshot of Pathfinding, Path Length and States explored by extra in bigDots

**Statement of Contribution:**

Yuhang and Wenbo both implemented the DFS function, Wenbo's code never ran successfully, so Yuhang's code was submitted.

Wenbo implemented the BFS function.

Jiakai and Yuhang both implemented the greedy function; their results were compared for debugging and Yuhang's code was submitted.

Yuhang implemented the Astar function.

Jiakai implemented the part of extra credit.