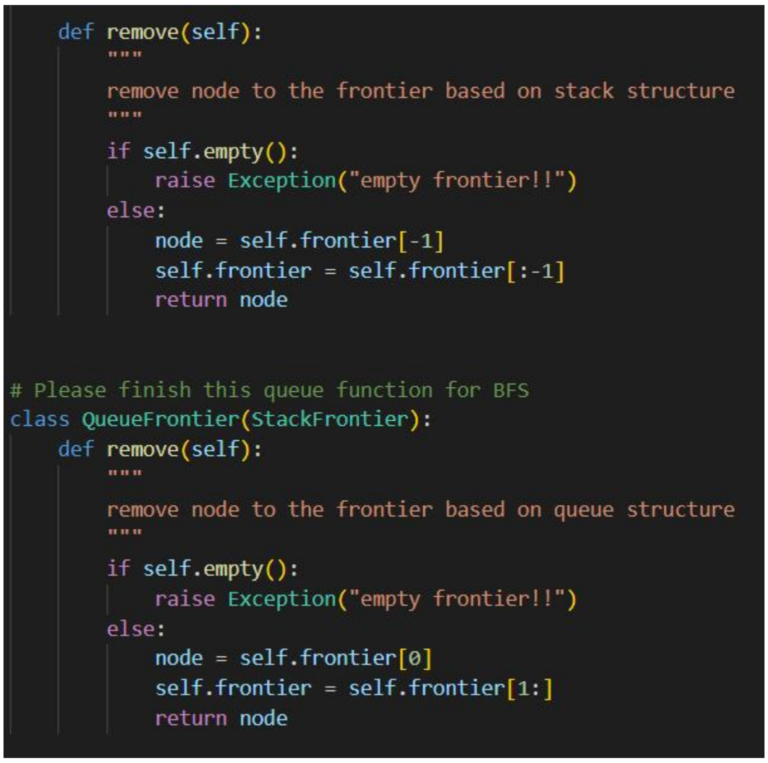
121090406 马腾飞

I. Programming Part

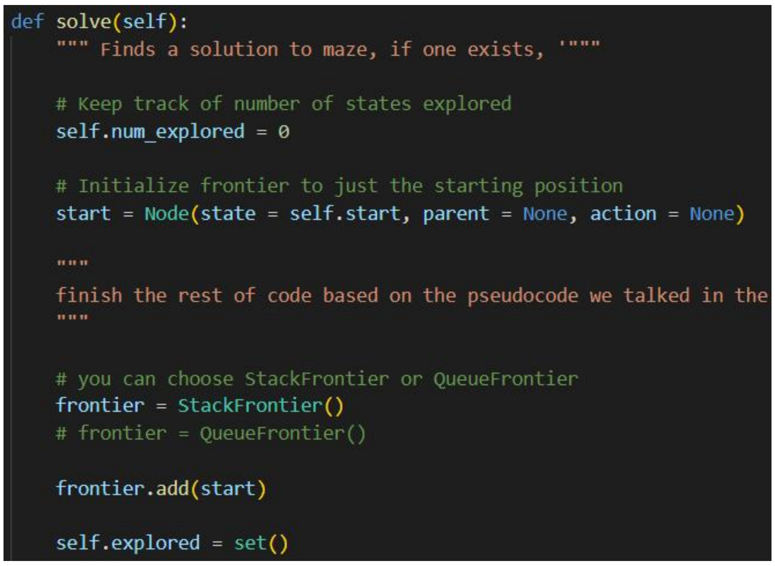
Key code:

1. The only different between DFS and BFS is “remove” of frontier, and DFS uses StackFrontier, BFS uses QueueFrontier.



```
    def remove(self):
        """
        remove node to the frontier based on stack structure
        """
        if self.empty():
            raise Exception("empty frontier!!")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node


# Please finish this queue function for BFS
class QueueFrontier(StackFrontier):
    def remove(self):
        """
        remove node to the frontier based on queue structure
        """
        if self.empty():
            raise Exception("empty frontier!!")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```

2. For “solve” part, firstly initialize a frontier and a explored set, and put start point into frontier.



```
def solve(self):
    """ Finds a solution to maze, if one exists, '"""

    # Keep track of number of states explored
    self.num_explored = 0

    # Initialize frontier to just the starting position
    start = Node(state = self.start, parent = None, action = None)

    """
    finish the rest of code based on the pseudocode we talked in the
    """

    # you can choose StackFrontier or QueueFrontier
    frontier = StackFrontier()
    # frontier = QueueFrontier()

    frontier.add(start)

    self.explored = set()
```

After that, repeat the following procedures until frontier is empty.

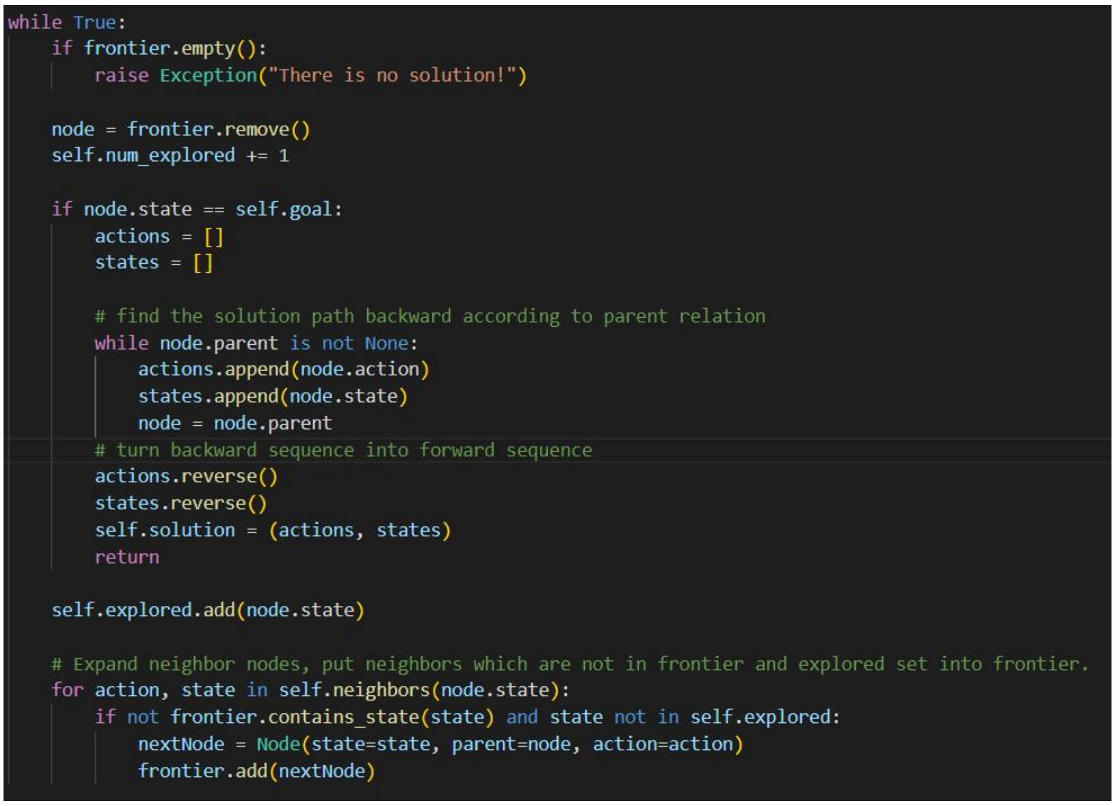
(i) Check whether frontier is empty. If it is empty, there is no solution.

(ii) Remove a node from frontier and check whether the node is goal. If it is goal, return solution.

(iii) Put the removed node into explored set.

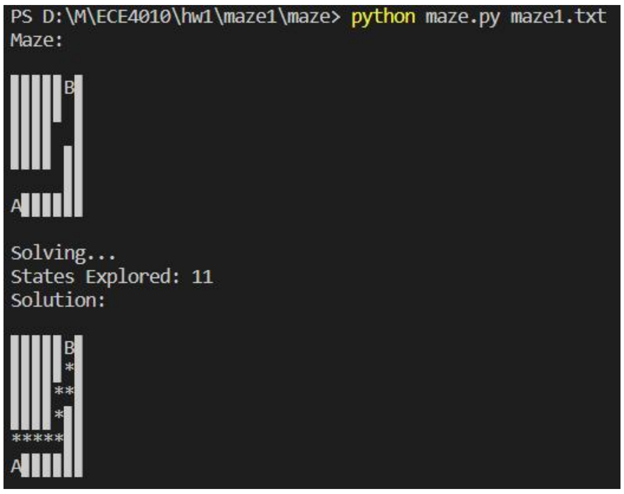
121090406 马腾飞

(iv) Expand neighbor nodes, put neighbors which are not in frontier and explored set into frontier.



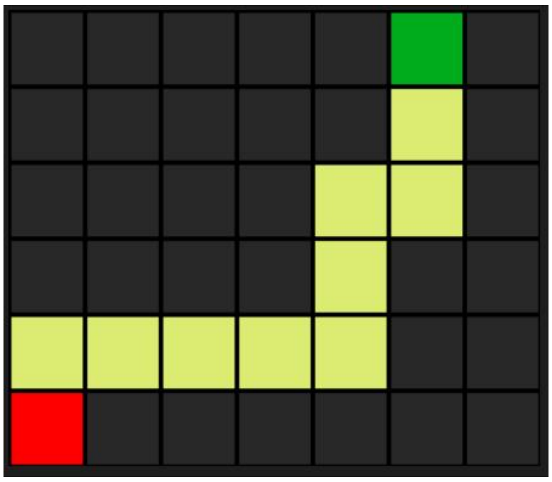
```python
while True:
    if frontier.empty():
        raise Exception("There is no solution!")

    node = frontier.remove()
    self.num_explored += 1

    if node.state == self.goal:
        actions = []
        states = []

        # find the solution path backward according to parent relation
        while node.parent is not None:
            actions.append(node.action)
            states.append(node.state)
            node = node.parent
        # turn backward sequence into forward sequence
        actions.reverse()
        states.reverse()
        self.solution = (actions, states)
        return

    self.explored.add(node.state)

    # Expand neighbor nodes, put neighbors which are not in frontier and explored set into frontier.
    for action, state in self.neighbors(node.state):
        if not frontier.contains_state(state) and state not in self.explored:
            nextNode = Node(state=state, parent=node, action=action)
            frontier.add(nextNode)
```

Results:
For DFS(StackFrontier):
Maze1:

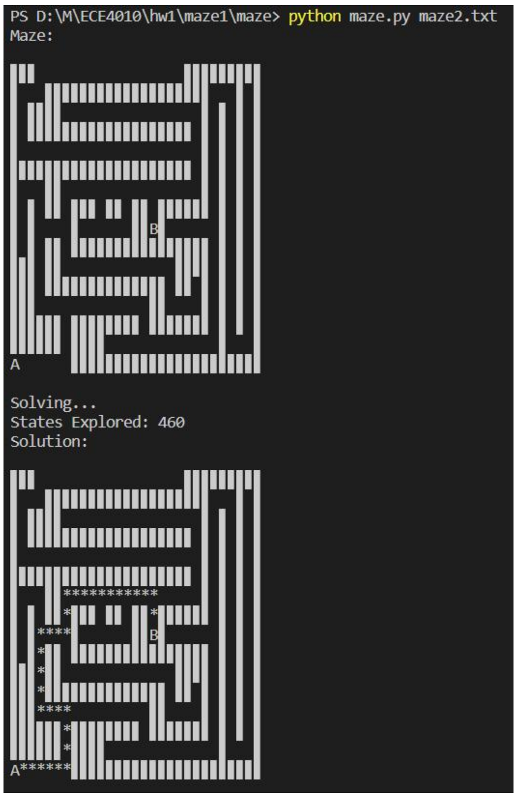


```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze1.txt
Maze:

█████B█
█████ █
████  █
████ ██
     ██
A██████

Solving...
States Explored: 11
Solution:

█████B█
█████*█
████**█
████*██
*****██
A██████
```

121090406 马腾飞

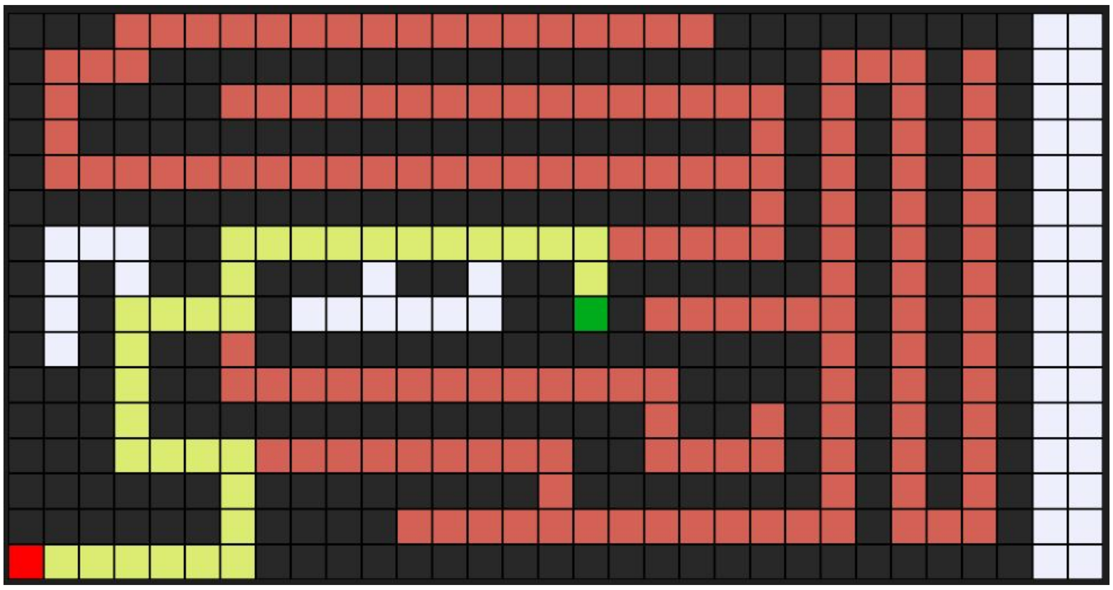


Maze2:

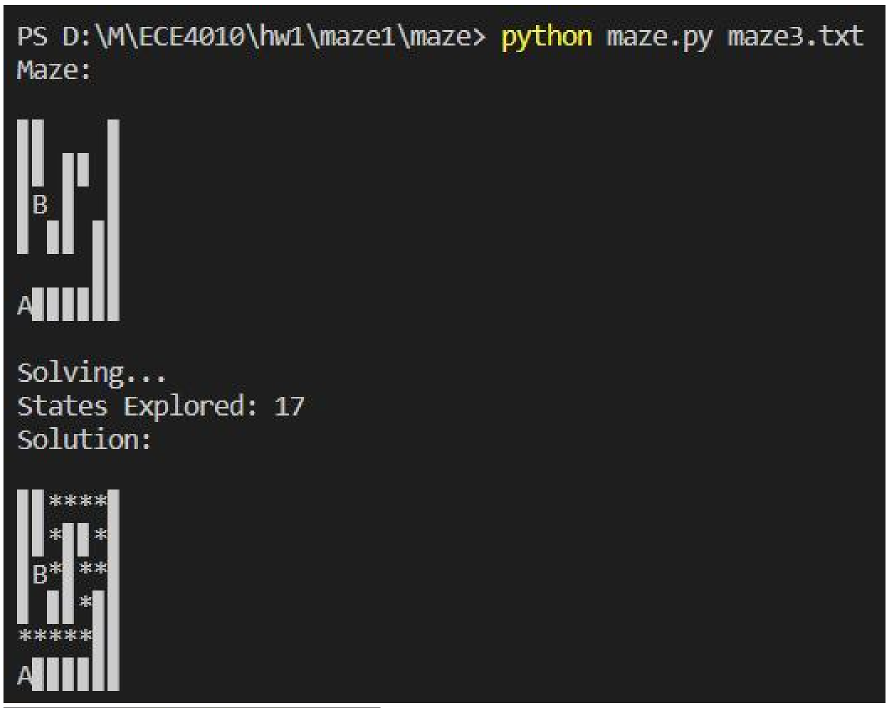


```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze2.txt
Maze:

███                 █████████
█   ███████████████████   █ █
█ ████                █ █ █ █
█ ███████████████████ █ █ █ █
█                     █ █ █ █
█████████████████████ █ █ █ █
█   ██                █ █ █ █
█ █ ██ ███ ██ ██ ██████ █ █ █
█ █    █      ██B█      █ █ █
█ █ ██ ████████████████ █ █ █
███ ██             ████ █ █ █
███ ██████████████ ██ █ █ █ █
███             ██    █ █ █ █
██████ ████████ ███████ █ █ █
██████ ████             █   █
A      ██████████████████████

Solving...
States Explored: 460
Solution:

███                 █████████
█   ███████████████████   █ █
█ ████                █ █ █ █
█ ███████████████████ █ █ █ █
█                     █ █ █ █
█████████████████████ █ █ █ █
█   ██***********     █ █ █ █
█ █ ██*███ ██ ██*██████ █ █ █
█ █****█      ██B█      █ █ █
█ █*██ ████████████████ █ █ █
███*██             ████ █ █ █
███*██████████████ ██ █ █ █ █
███****         ██    █ █ █ █
██████*████████ ███████ █ █ █
██████*████             █   █
A******██████████████████████
```

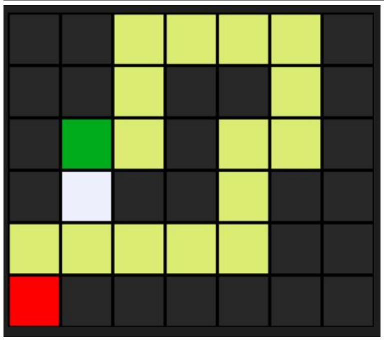
121090406 马腾飞



Maze3:



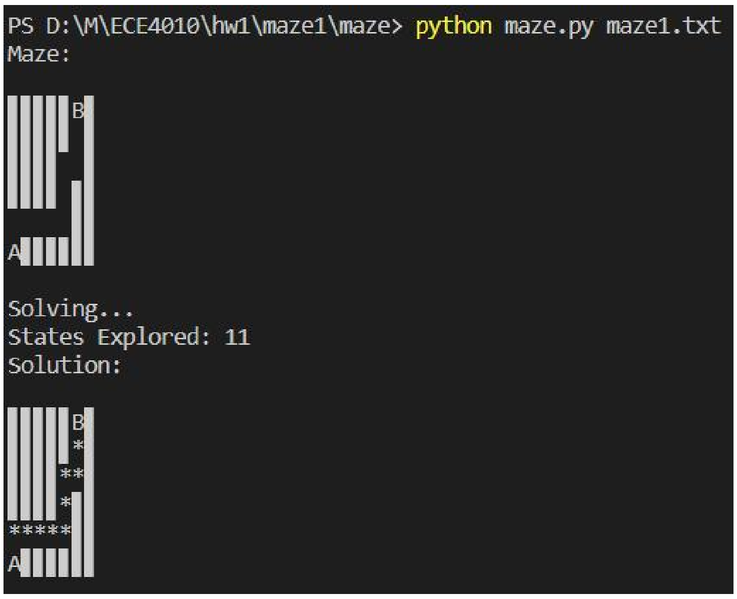
```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze3.txt
Maze:

██    █
██ ██ █
█B █  █
█ ██ ██
     ██
A██████

Solving...
States Explored: 17
Solution:

██****█
██*██*█
█B*█**█
█ ██*██
*****██
A██████
```



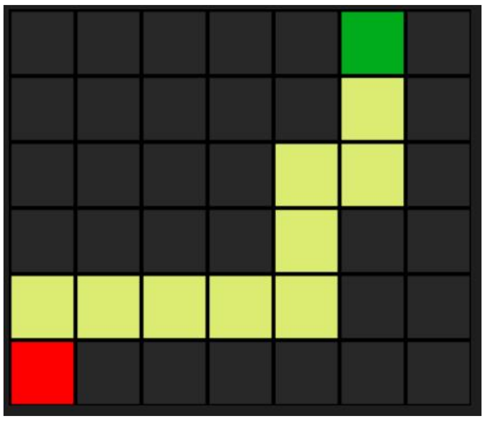
For BFS(QueueFrontier):

Maze1:

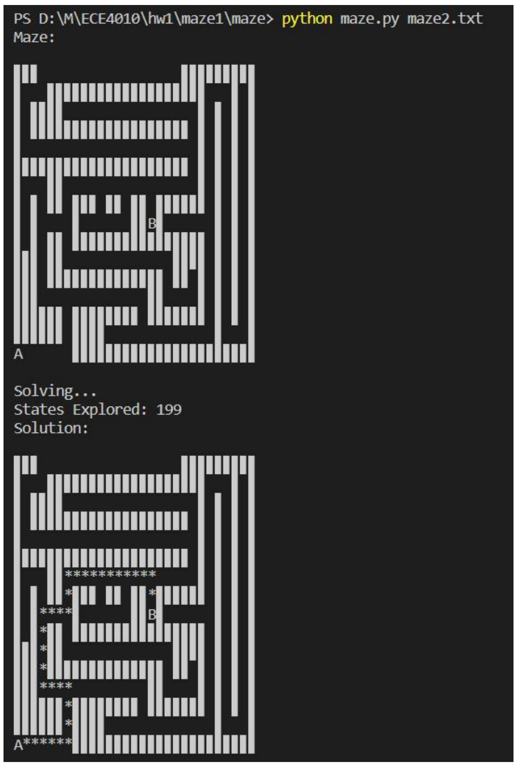
121090406 马腾飞



```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze1.txt
Maze:

█████B█
█████ █
████  █
████ ██
     ██
A██████

Solving...
States Explored: 11
Solution:

█████B█
█████*█
████**█
████*██
*****██
A██████
```

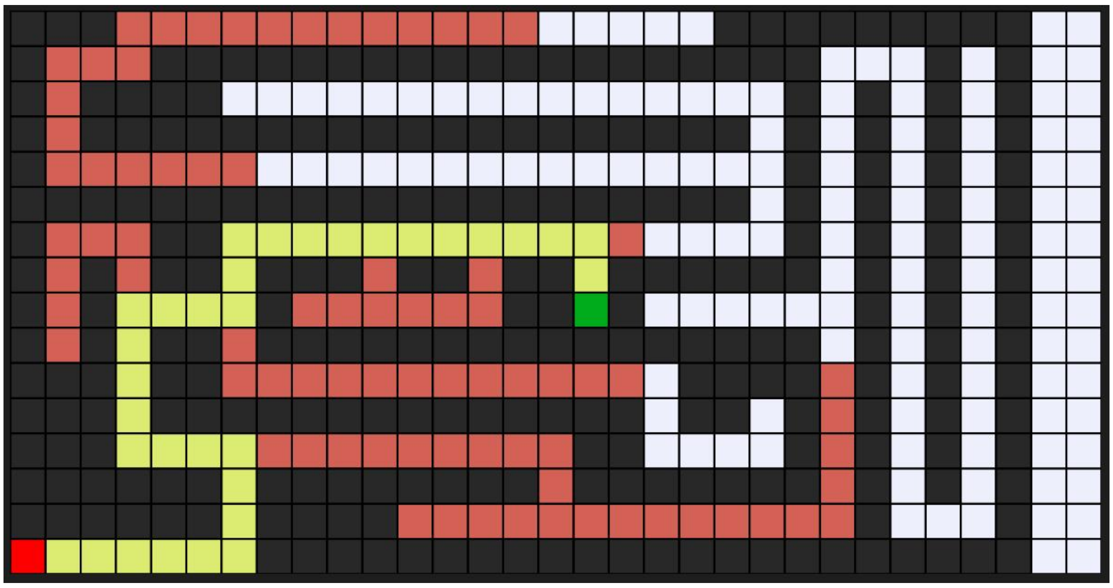


Maze2:

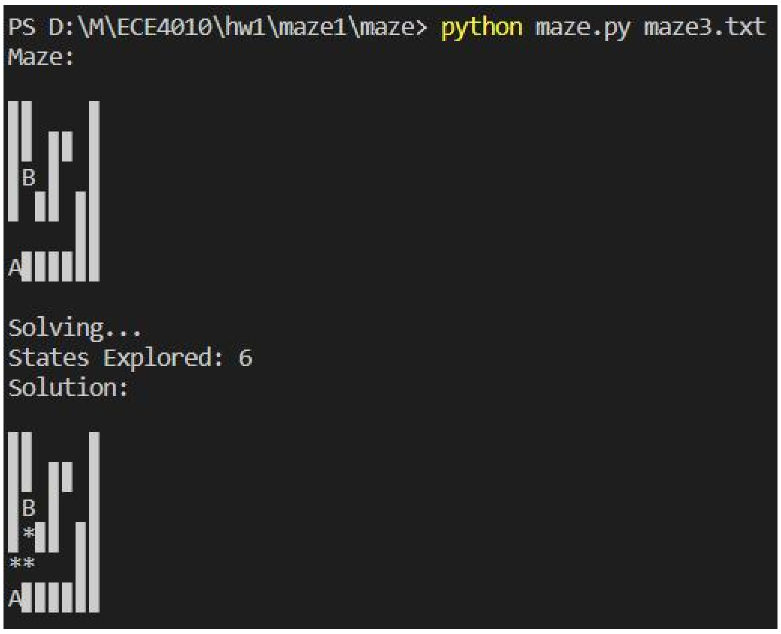


```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze2.txt
Maze:

███                 █████████
█   ███████████████████   █ █
█ ████                █ █ █ █
█ ███████████████████ █ █ █ █
█                     █ █ █ █
█████████████████████ █ █ █ █
█   ██                █ █ █ █
█ █ ██ ███ ██ ██ ██████ █ █ █
█ █    █      ██B█      █ █ █
█ █ ██ ████████████████ █ █ █
███ ██             ████ █ █ █
███ ██████████████ ██ █ █ █ █
███             ██    █ █ █ █
██████ ████████ ███████ █ █ █
██████ ████             █   █
A      ██████████████████████

Solving...
States Explored: 199
Solution:

███                 █████████
█   ███████████████████   █ █
█ ████                █ █ █ █
█ ███████████████████ █ █ █ █
█                     █ █ █ █
█████████████████████ █ █ █ █
█   ██***********     █ █ █ █
█ █ ██*███ ██ ██*██████ █ █ █
█ █****█      ██B█      █ █ █
█ █*██ ████████████████ █ █ █
███*██             ████ █ █ █
███*██████████████ ██ █ █ █ █
███****         ██    █ █ █ █
██████*████████ ███████ █ █ █
██████*████             █   █
A******██████████████████████
```

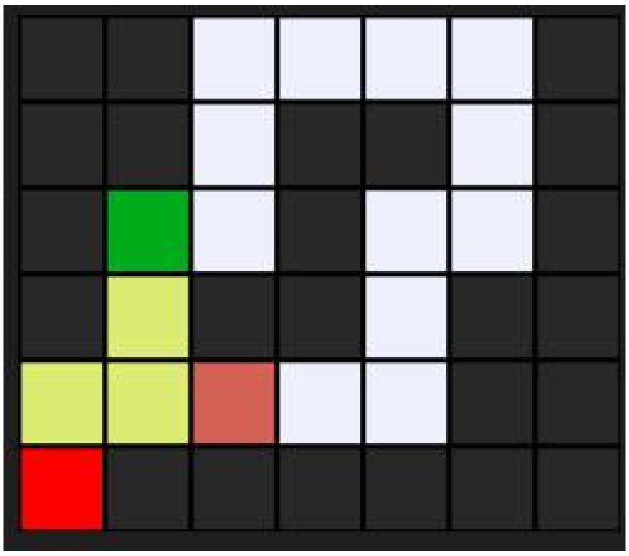
121090406 马腾飞



Maze3:



```
PS D:\M\ECE4010\hw1\maze1\maze> python maze.py maze3.txt
Maze:

██    █
██ ██ █
█B █  █
█ ██ ██
     ██
A██████

Solving...
States Explored: 6
Solution:

██    █
██ ██ █
█B █  █
█*██ ██
**   ██
A██████
```



121090406 马腾飞

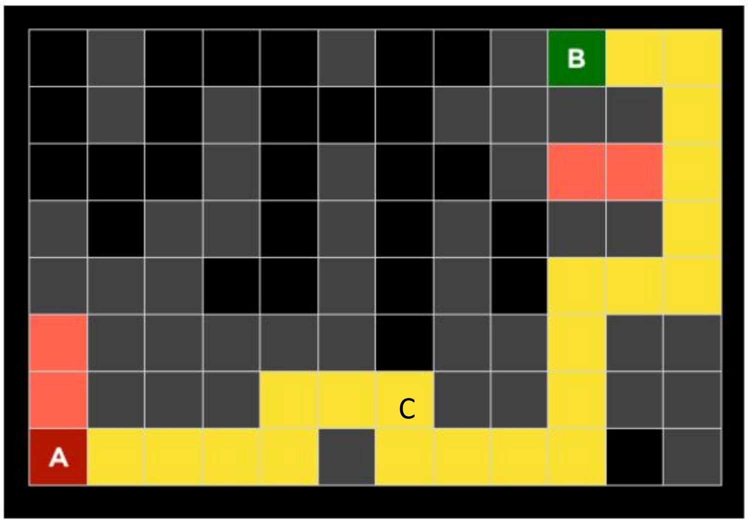
II. Written Part

Q1. [5 pts] Between depth first search (DFS) and breadth first search (BFS), which will find a shorter path through a maze?

BFS，details are as follows.

Breadth First Search (BFS): BFS starts at the beginning point and first explores all adjacent points, then expands layer by layer outward. This method ensures that when we reach the end point for the first time, the path found is the shortest. Therefore, in problems where finding the shortest path is the goal, BFS is the better choice.

Q2. [5pts] The following question will ask you about the below maze. Grey cells indicate walls. A search algorithm was run on this maze, and found the yellow highlighted path from point A to B. In doing so, the red highlighted cells were the states explored but that did not lead to the goal.



Of the four search algorithms discussed in lecture — depth-first search, breadth-first search, greedy best-first search with Manhattan distance heuristic, and A* search with Manhattan distance heuristic — which one (or multiple, if multiple are possible) could be the algorithm used?

Only DFS could be the algorithm used.

1. At point C, there are no branches, which indicates breadth-first search is not used. Because BFS will expand all possible paths, this algorithm obviously is not to expand upward at point C.
2. At point C, the algorithm expands downward, which means that the algorithm does not expand according to minimizing Manhattan distance or (Manhattan distance + cost, namely, like A*). So greedy best-first search with Manhattan distance heuristic and A* search with Manhattan distance heuristic are not used.
3. As we can see in the picture above, every wrong path reaches dead end, and not all

the paths are attempted. This highly corresponds to the features of DFS. DFS explores as deeply as possible into the maze's branches until it reaches a dead end, and then backtracks to the nearest fork. So the algorithm can be DFS.

Q3. [5 pts] Why is depth-limited minimax sometimes preferable to minimax without a depth limit?

Depth-limited Minimax is sometimes preferred over Minimax without a depth limit mainly due to efficiency and practicality considerations.

1. Computational Efficiency: In the Minimax algorithm without a depth limit, the computer needs to explore all possible game states until the end of the game. This is often impractical because the number of possible states can be enormous, leading to **excessive computation time and memory usage.** Depth limitation significantly reduces the number of nodes that need to be evaluated, thus improving the algorithm's computational efficiency.

2. Real-time Decision Making: In some real-time or time-constrained games, making quick decisions is crucial. Depth-limited Minimax, by limiting the search depth, can provide good decisions within a reasonable time frame, rather than trying to find the perfect decision, which is more practical in real-world applications.

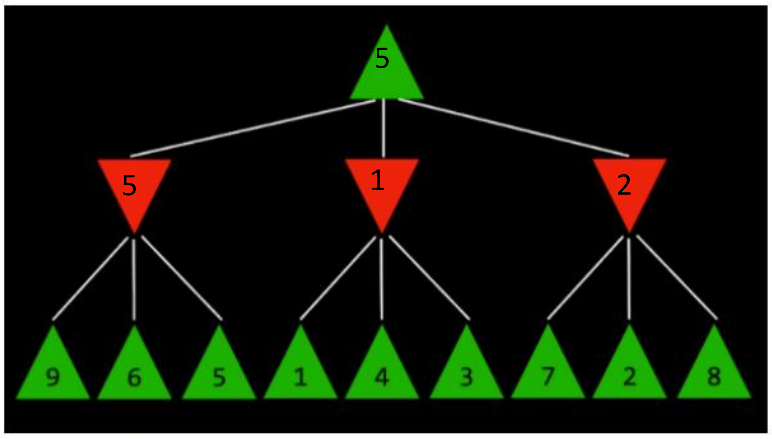
3. Avoiding the Horizon Effect: In certain cases, a shallower search might avoid the so-called "horizon effect," where the algorithm might only see immediate gains and overlook long-term benefits. By appropriately limiting the search depth, this phenomenon can be somewhat reduced.

4. Heuristic Evaluation: Depth-limited Minimax is often used in conjunction with heuristic evaluation functions. These functions can estimate the current game situation without reaching an endgame state, allowing the algorithm to make reasonable decisions even with limited depth.

In summary, due to its higher computational efficiency, suitability for real-time decision-making, reduction of the horizon effect, and effective combination with heuristic evaluations, depth-limited Minimax is often more popular in many practical applications than Minimax without a depth limit.

121090406 马腾飞

Q4. [10 pts] The following question will ask you about the Minimax tree below, where the green up arrows indicate the MAX player and red down arrows indicate the MIN player. The leaf nodes are each labelled with their value.



a. [5 pts] What are the value of the nodes in the above figure, if we use Minimax?
b. [5 pts] If you are the MAX player, what is your best action (left, middle, or right) based on the above figure

a. Because min(9,6,5) = 5 , min(1,4,3) = 1, min(7,2,8) = 2, max(5,1,2) = 5, the value of nodes are as what I mark in the figure above.
b. Left is my best action, because the value of left is maximum among the three nodes.