Now, implement a search algorithm to find a solution to a maze using python. The maze is represented as a grid where 'A' represents the starting point, 'B' represents the goal, empty or spaces represent open paths, and “#” represent walls.

Step-01:

First of all import the **“sys”** module, because it is a built-in module that provides access to various system-specific parameters and functions. It is commonly used for interacting with the Python runtime environment and the underlying operating system. When it’s import then some common tasks and functionalities are enabled, for example: access to command-line arguments, exit the script, access to standard streams, access to system-specific information and path manipulation etc.

```python
import sys
```

Step-02:

Now, write down the class. First of all, implement “node” class:

```python
class Node():
   def __init__(self, state, parent, action):
       self.state = state
       self.parent = parent
       self.action = action
```

self.state = state: This line assigns the state parameter passed to the constructor to the state attribute of the node object.

self.parent = parent: This line assigns the parent parameter to the parent attribute of the node object.

self.action = action: This line assigns the action parameter to the action attribute of the node object.

Step-03:

Implement the StackFrontier class. This class implements a stack-based frontier for the search algorithm. It stores nodes and provides methods for adding, checking if it contains a specific state, checking if it's empty, and removing nodes.

```python
class StackFrontier():
   def __init__(self):
       self.frontier = []

   def add(self, node):
       self.frontier.append(node)

   def contains_state(self, state):
       return any(node.state == state for node in self.frontier)
```

```
    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
```

i. __init__(self): This is the class constructor. It initializes an instance of the ‘StackFrontier’ class with an empty list called ‘frontier’. The ‘frontier’ list will be used to store nodes during the search.

ii. add(self, node): This method is used to add a node to the frontier stack. It appends the ‘node’ object to the end of the ‘frontier’ list, effectively pushing it onto the stack.

iii. contains_state(self, state): This method checks if a state is present in any of the nodes within the ‘frontier’. It does so by iterating through the nodes in the ‘frontier’ list and comparing the ‘state’ attribute of each node with the provided ‘state’. If any node in the frontier has a matching state, the method returns **True**; otherwise, it returns **False**.

iv. empty(self): This method checks whether the frontier is empty. It returns **True** if the ‘frontier’ list is empty (indicating that there are no nodes left to explore), and **False** otherwise.

v. remove(self): This method is used to extract and return the top node from the stack. It checks for an empty frontier using ‘empty’. If empty, it raises an "empty frontier" exception; otherwise, it removes and returns the last node with list slicing ([:-1]).

Step-04:

Implement QueueFrontier class and extends StackFrontier to implement a queue-based frontier. It removes nodes in a FIFO (First-In-First-Out) order.

```
class QueueFrontier(StackFrontier):

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```

‘QueueFrontier’ is a subclass of ‘StackFrontier’, inheriting methods and attributes. It modifies the ‘remove’ method to achieve queue behaviour, ensuring the front node is removed and returned. It uses a first-in-first-out (FIFO) approach and raises an "empty frontier" exception if the queue is empty.

Step-05:

Implement **Maze** class. The Maze class reads a maze from a file, validates it, and stores its layout. It tracks the maze's dimensions, walls, start, and goal points. It also prepares to store a solution path if one is found. Here's Maze class functionality:

Implement **Initialization** method under the **Maze** class. Reads a maze from a file and sets its height, width, start position, goal position, and wall locations.

```python
class Maze():

   def __init__(self, filename):

       # Read file and set height and width of maze
       with open(filename) as f:
           contents = f.read()

       # Validate start and goal
       if contents.count("A") != 1:
           raise Exception("maze must have exactly one start point")
       if contents.count("B") != 1:
           raise Exception("maze must have exactly one goal")

       # Determine height and width of maze
       contents = contents.splitlines()
       self.height = len(contents)
       self.width = max(len(line) for line in contents)

       # Keep track of walls
       self.walls = []
       for i in range(self.height):
           row = []
           for j in range(self.width):
               try:
                   if contents[i][j] == "A":
                       self.start = (i, j)
                       row.append(False)
                   elif contents[i][j] == "B":
                       self.goal = (i, j)
                       row.append(False)
                   elif contents[i][j] == " ":
                       row.append(False)
                   else:
                       row.append(True)
```

```
                except IndexError:
                    row.append(False)
            self.walls.append(row)

        self.solution = None
```

def __init__(self, filename): This is the constructor method for the 'Maze' class, which is called when you create a new instance of the class. It takes a 'filename' parameter, which should be the name of a text file that contains the maze's layout.

- It opens the specified file using the 'open' function and reads its contents, storing them in the 'contents' variable.
- It validates the maze by checking if there is exactly one start point ("A") and exactly one goal point ("B") in the maze. If not, it raises an exception with appropriate error messages.
- It splits the 'contents' into lines using 'splitlines()' to determine the height of the maze (the number of lines) and the maximum width of any line.
- It initializes the 'self.height' and 'self.width' attributes to store the maze's dimensions.
- It keeps track of the maze's walls and open paths by iterating over the lines and characters in 'contents'. It creates a 2D list called 'self.walls', where each element represents a cell in the maze. Cells containing walls are marked as **True**, while open cells are marked as **False**. Additionally, it records the starting point as 'self.start' and the goal point as 'self.goal'.
- Finally, it initializes 'self.solution' to 'None', which will be used to store the solution path if found.

<u>Step-06:</u>

Implement **print** methond.It is implement under the **Maze** class. Prints the maze grid to the console, including the start, goal, and solution path (if found).

```
def print(self):
    solution = self.solution[1] if self.solution is not None else None
    print()
    for i, row in enumerate(self.walls):
        for j, col in enumerate(row):
            if col:
                print("█", end="")
            elif (i, j) == self.start:
                print("A", end="")
            elif (i, j) == self.goal:
                print("B", end="")
            elif solution is not None and (i, j) in solution:
                print("*", end="")
            else:
                print(" ", end="")
        print()
    print()
```

1) It initializes a variable ‘solution’ to the second element of ‘self.solution’ if ‘self.solution’ is not ‘None’. This assumes that ‘self.solution’ is a tuple where the first element may contain actions (not used in this method) and the second element contains the cells of the solution path.
2) It prints a blank line to create space before displaying the maze.
3) It iterates through each row and column of the maze, checking the following conditions for each cell:
    - If ‘col’ is **True**, it prints a filled block character ('█') to represent a wall.
    - If the current cell matches the start position, it prints 'A' to indicate the start point.
    - If the current cell matches the goal position, it prints 'B' to indicate the goal point.
    - If a solution exists (‘solution’ is not ‘None’) and the current cell is part of the solution path (found in ‘solution’), it prints **'*'** to mark that cell on the solution path.
    - Otherwise, it prints a space (' ') to represent an open path.
4) After printing each row, it prints a newline character to move to the next row.
5) Finally, it prints an additional blank line for better formatting.

Step-07:

Implement **Neighbors** method under the **Maze** class. Given a state (position), it returns a list of neighboring states that can be reached from the current state. It checks for valid and unblocked neighbors.

```
def neighbors(self, state):
    row, col = state
    candidates = [
        ("up", (row - 1, col)),
        ("down", (row + 1, col)),
        ("left", (row, col - 1)),
        ("right", (row, col + 1))
    ]

    result = []
    for action, (r, c) in candidates:
            if 0 <= r < self.height and 0 <= c < self.width and not
self.walls[r][c]:
            result.append((action, (r, c)))
    return result
```

The ‘neighbors’ method in the ‘Maze’ class is responsible for finding and returning the neighboring cells of a given state within the maze. Here's a concise explanation of what this method does:

- It takes a ‘state’ as input, which represents a cell's coordinates in the maze (a tuple of ‘(row, col)’).
- It calculates the potential neighboring cells in the four cardinal directions (up, down, left, and right) relative to the given ‘state’.
- It initializes an empty list called ‘result’ to store the valid neighboring cells.
- It iterates through each potential neighboring cell (represented as ‘(action, (r, c))’), where ‘action’ is a string indicating the direction and ‘(r, c)’ are the new coordinates.
- For each potential neighbor, it checks the following conditions:
  => If both ‘r’ and ‘c’ are within the valid range (0 to ‘self.height-1’ for ‘r’ and 0 to ‘self.width-1’ for ‘c’)
  => If the cell at ‘(r, c)’ in the maze is not a wall (as determined by ‘self.walls[r][c]’ being **False**)
- If both conditions are met, it appends the ‘(action, (r, c))’ tuple to the ‘result’ list, indicating a valid neighboring cell.
- Finally, it returns the ‘result’ list containing the valid neighboring cells.

<u>Step-08:</u>

Implement the **solve** method under the **Maze** class. Implements the search algorithm (Breadth-First Search) to find a solution to the maze. It initializes a frontier with the starting position, explores nodes, and backtracks to find the solution path.

```python
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)
    frontier = QueueFrontier()
    frontier.add(start)

    # Initialize an empty explored set
    self.explored = set()

    # Keep looping until solution found
    while True:

        # If nothing left in frontier, then no path
        if frontier.empty():
            raise Exception("no solution")
        # Choose a node from the frontier
        node = frontier.remove()
        self.num_explored += 1

        # If node is the goal, then we have a solution
        if node.state == self.goal:
            actions = []
            cells = []
            while node.parent is not None:
                actions.append(node.action)
                cells.append(node.state)
                node = node.parent
            actions.reverse()
            cells.reverse()
            self.solution = (actions, cells)
            return

        # Mark node as explored
        self.explored.add(node.state)

        # Add neighbors to frontier
        for action, state in self.neighbors(node.state):
            if not frontier.contains_state(state) and state not in
self.explored:
                child = Node(state=state, parent=node, action=action)
                frontier.add(child)
```

Explanation of **solve** method:

- It initializes a counter ‘self.num_explored’ to keep track of the number of states explored during the search.
- It creates a starting node (‘start’) using the initial state (‘self.start’) and adds it to a queue-based frontier (‘frontier’) using the ‘QueueFrontier’.
- It initializes an empty set called ‘self.explored’ to keep track of explored states.
- It enters a loop that continues until a solution is found (or the maze is determined to have no solution):
    - ❖ If the frontier is empty, it raises an "no solution" exception, indicating that no path to the goal exists.
    - ❖ It selects a node from the frontier using the ‘remove’ method.
    - ❖ It increments the ‘self.num_explored’ counter to track the number of states explored.
    - ❖ If the selected node's state matches the goal state (‘self.goal’), it means a solution has been found. It reconstructs the solution path by backtracking from the goal node to the start node, collecting actions and cells along the way. The resulting path is stored in ‘self.solution’.
    - ❖ It marks the current node's state as explored by adding it to the ‘self.explored’ set.

    - ❖ It examines the neighboring cells of the current node using the ‘neighbors’ method and checks if they are suitable for exploration. If a neighboring cell meets the criteria (not already in the frontier and not explored), it creates a child node and adds it to the frontier for further exploration.
- Once a solution is found, the method returns, and the solution path is stored in ‘self.solution’.

Step-09:

Implement the **output_image** method under the **Maze** class. Creates an image representation of the maze with various colours for walls, start, goal, solution path, and explored cells. It saves the image to a file.

Requirement: If not install Pillow,then install Pillow in the Virtual Environment

Run this command in the terminal: **pip install Pillow**

```
def output_image(self, filename, show_solution=True, show_explored=False):
   from PIL import Image, ImageDraw
   cell_size = 50
   cell_border = 2

   # Create a blank canvas
   img = Image.new(
       "RGBA",
       (self.width * cell_size, self.height * cell_size),
       "black"
   )
   draw = ImageDraw.Draw(img)

   solution = self.solution[1] if self.solution is not None else None
   for i, row in enumerate(self.walls):
       for j, col in enumerate(row):

           # Walls
           if col:
               fill = (40, 40, 40)

           # Start
           elif (i, j) == self.start:
               fill = (255, 0, 0)

           # Goal
           elif (i, j) == self.goal:
               fill = (0, 171, 28)
```

```
            # Solution
            elif solution is not None and show_solution and (i, j) in
solution:
                fill = (220, 235, 113)

            # Explored
            elif solution is not None and show_explored and (i, j) in
self.explored:
                fill = (212, 97, 85)

            # Empty cell
            else:
                fill = (237, 240, 252)

            # Draw cell
            draw.rectangle(
                ([(j * cell_size + cell_border, i * cell_size + cell_border),
                    ((j + 1) * cell_size - cell_border, (i + 1) * cell_size -
cell_border)]),
                fill=fill
            )

    img.save(filename)
```

Here's explanation of **output_image** method:

- It takes three parameters: ‘filename’ (the name of the image file to save), ‘show_solution’(a boolean indicating whether to highlight the solution path), and ‘show_explored’ (a boolean indicating whether to highlight explored cells).
- It imports the necessary libraries, including the Python Imaging Library (PIL) for image creation.
- It defines ‘cell_size’ and ‘cell_border’ variables to control the size and border width of each cell in the generated image.
- It creates a blank image canvas (‘img’) with dimensions based on the maze's width and height, as specified by ‘cell_size’. The background color is set to black.
- It checks if a solution exists (‘self.solution’ is not ‘None’) and assigns the solution cells to the ‘solution’ variable.
- It iterates through each cell in the maze, determining its appearance based on several conditions:
    - ❖ Walls are filled with a dark color (‘(40, 40, 40)’).
    - ❖ The start point is marked with red (‘(255, 0, 0)’).
    - ❖ The goal point is marked with green (‘(0, 171, 28)’).
    - ❖ If ‘show_solution’ is **True** and the cell is part of the solution path, it is marked with a specific color (‘(220, 235, 113)’).
    - ❖ If ‘show_explored’ is **True** and the cell has been explored, it is marked with another color (‘(212, 97, 85)’).

- For each cell, it draws a rectangle on the image canvas, using the specified ‘fill’ color. The rectangle's position and size are calculated based on the cell's coordinates and ‘cell_size’.
- Finally, it saves the generated image to the specified ‘filename’.

  End the implement all needed method under the Maze class.

<u>Step-10:</u>

Implement **if condition**. This code checks if the script is provided with exactly two command-line arguments. If the count is not two, it exits the script and shows a usage message, instructing how to correctly use the script, including providing a filename (e.g., "Prob.txt").

```
if len(sys.argv) != 2:
   sys.exit("Usage: python ey712.py Prob.txt")
```

<u>Step-11:</u>

```
m = Maze(sys.argv[1])
print("Maze:")
m.print()
print("Solving...")
m.solve()
print("States Explored:", m.num_explored)
print("Solution:")
m.print()
m.output_image("ProblemSolution.png", show_explored=True)
```

m = Maze(sys.argv[1]): It creates an instance of the ‘Maze’ class, initializing it with a filename obtained from the first command-line argument (‘sys.argv[1]’).

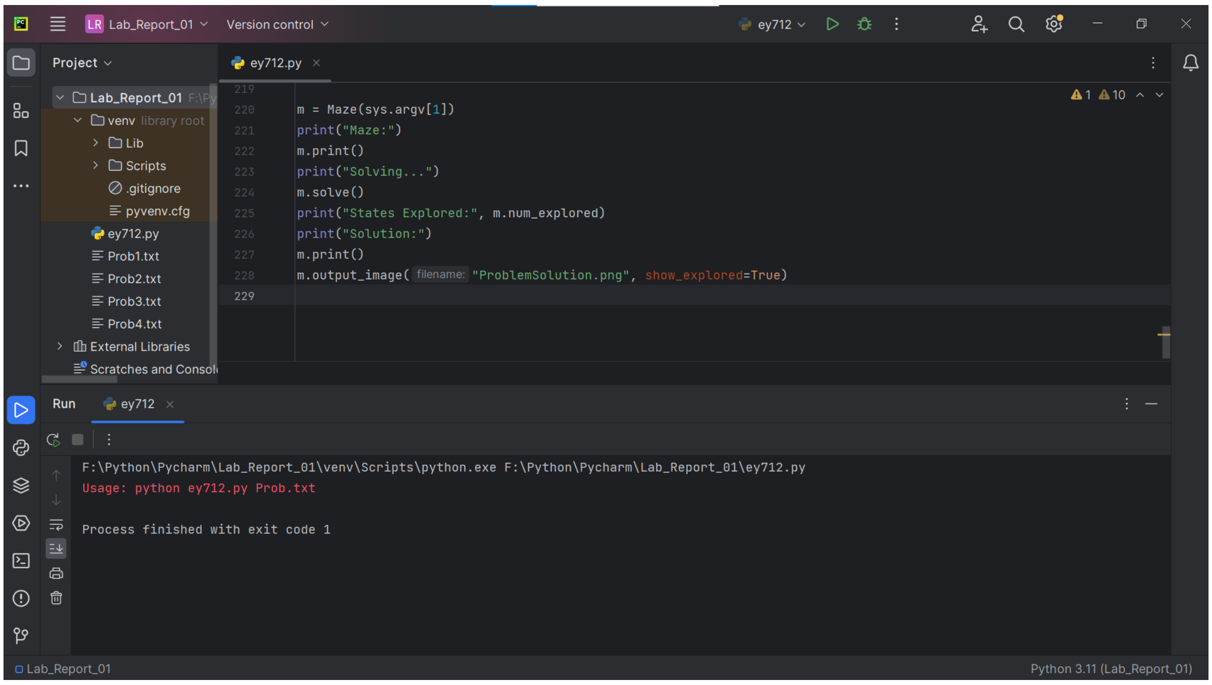
m.solve(): It invokes the ‘solve’ method of the ‘Maze’ instance to find a solution to the maze.

print("States Explored:", m.num_explored): It prints the number of states explored during the maze solving process.

m.print(): It again calls the ‘print’ method of the ‘Maze’ instance, displaying the maze layout with the solution path (if found) on the console.

m.output_image("ProblemSolution.png", show_explored=True): It generates an image of the maze with the solution path and explored cells highlighted and saves it as "ProblemSolution.png" in the current directory.

Now run the code:



```
m = Maze(sys.argv[1])
print("Maze:")
m.print()
print("Solving...")
m.solve()
print("States Explored:", m.num_explored)
print("Solution:")
m.print()
m.output_image(filename: "ProblemSolution.png", show_explored=True)
```

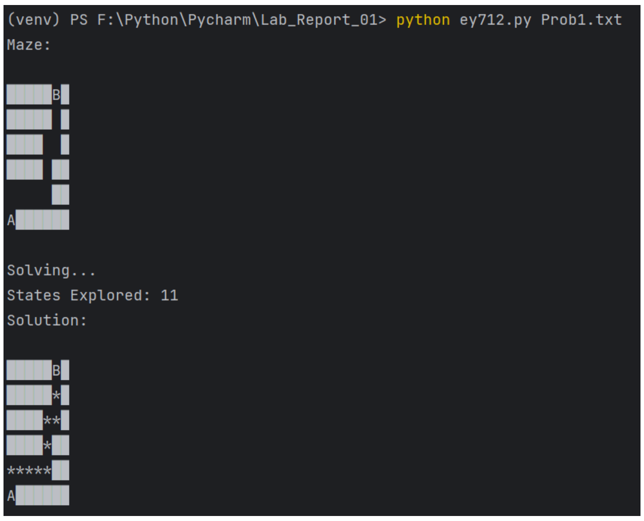
```
F:\Python\Pycharm\Lab_Report_01\venv\Scripts\python.exe F:\Python\Pycharm\Lab_Report_01\ey712.py
Usage: python ey712.py Prob.txt

Process finished with exit code 1
```

So this code executes successfully.

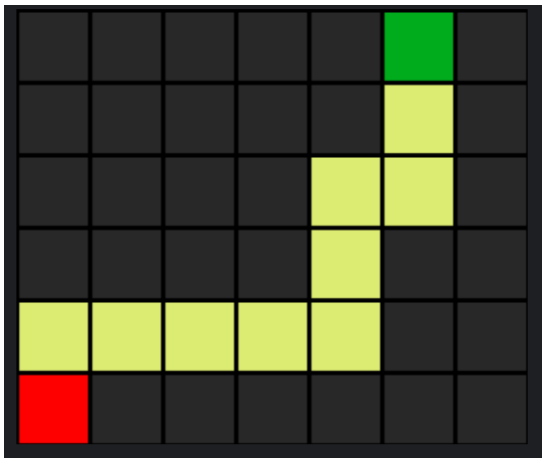
Problem-01:

Now solve the first maze problem **Prob1.txt** using BFS. First of all, Open the terminal and run the command:

python ey712.py Prob1.txt

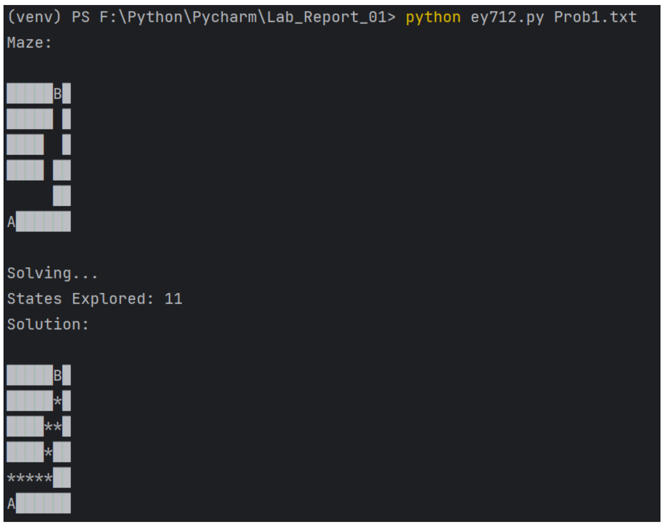


```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob1.txt
Maze:

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

Solving...
States Explored: 11
Solution:

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

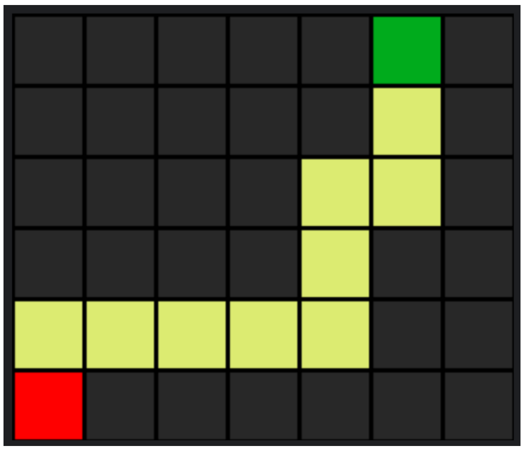


So,we can see after solving this problem using the BFS method, States Explored is 11.

Again this problem is solved using DFS and run the same command in the terminal:



```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob1.txt
Maze:

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

Solving...
States Explored: 11
Solution:

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

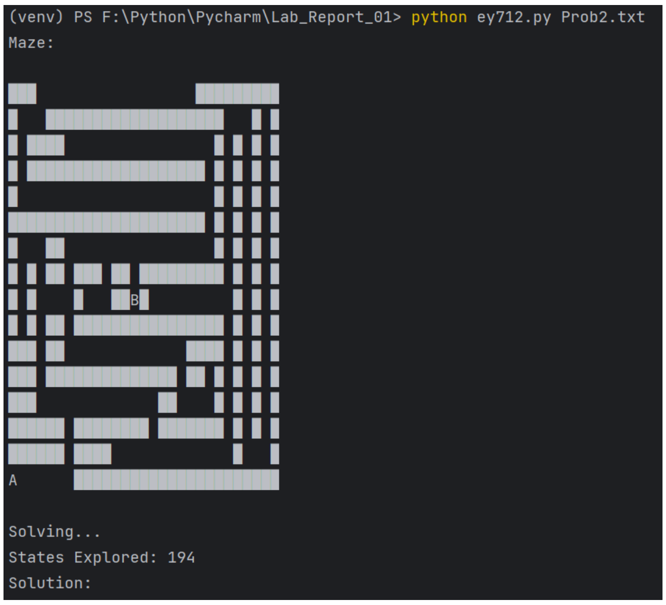


Again,we can see after solving this problem using the DFS method, States Explored is 11. Here using BFS and DFS both, States Explored is 11. So, using BFS or DFS, we can see the same result.Both algorithms will provide the best solution for this problem.

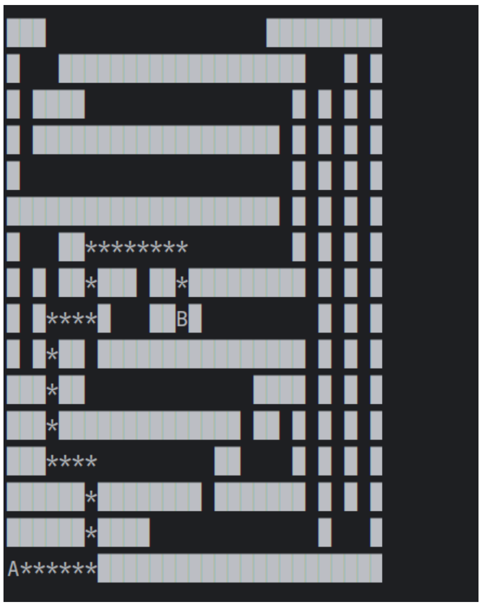
Problem-02:

Now solve the 2nd maze problem **Prob2.txt** using DFS. First of all, Open the terminal and run the command:

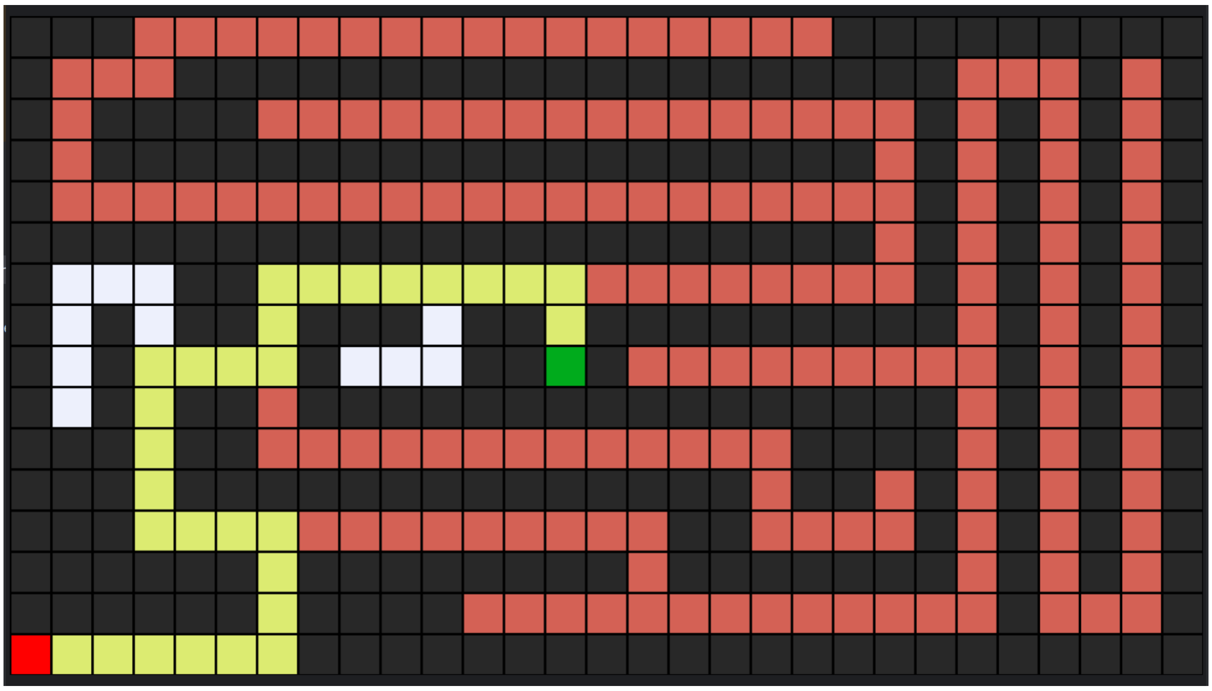
python ey712.py Prob2.txt



```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob2.txt
Maze:

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

Solving...
States Explored: 194
Solution:
```

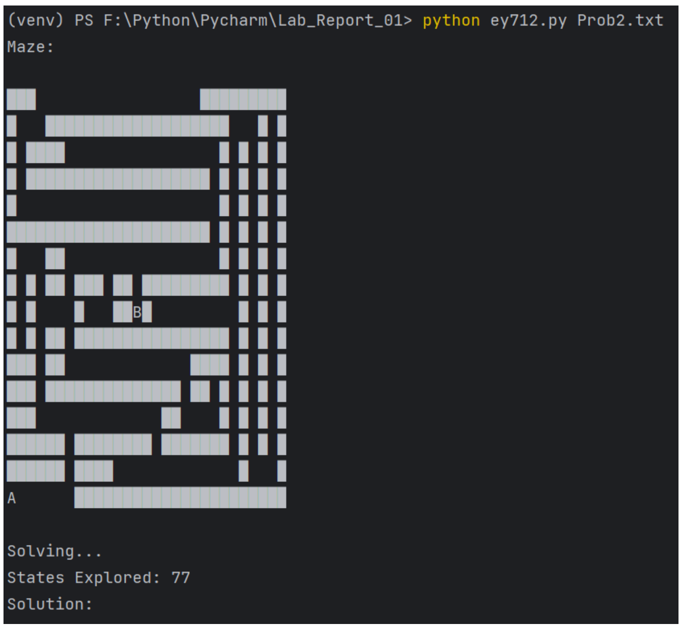


```
███                 █████████
█   ███████████████   █ █ █
█ ████                █ █ █ █
█ ███████████████████ █ █ █ █
█                     █ █ █ █
█████████████████████ █ █ █ █
█   ██********        █ █ █ █
█ █ ██*███ ██*█████████ █ █ █
█ █****█   ██B█         █ █ █
█ █*██ ████████████████ █ █ █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

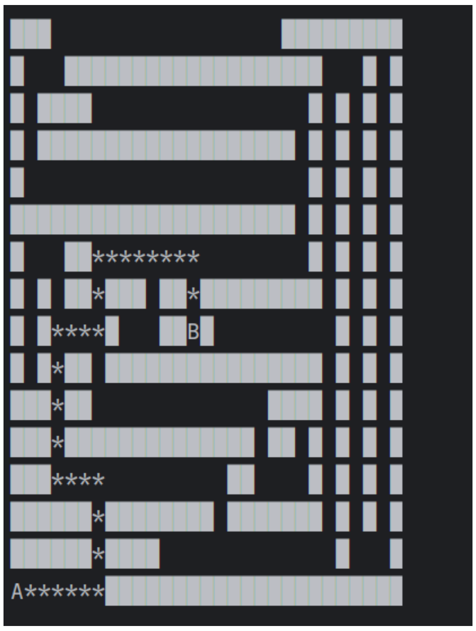


So,we can see after solving this problem using the DFS method, States Explored is 194.

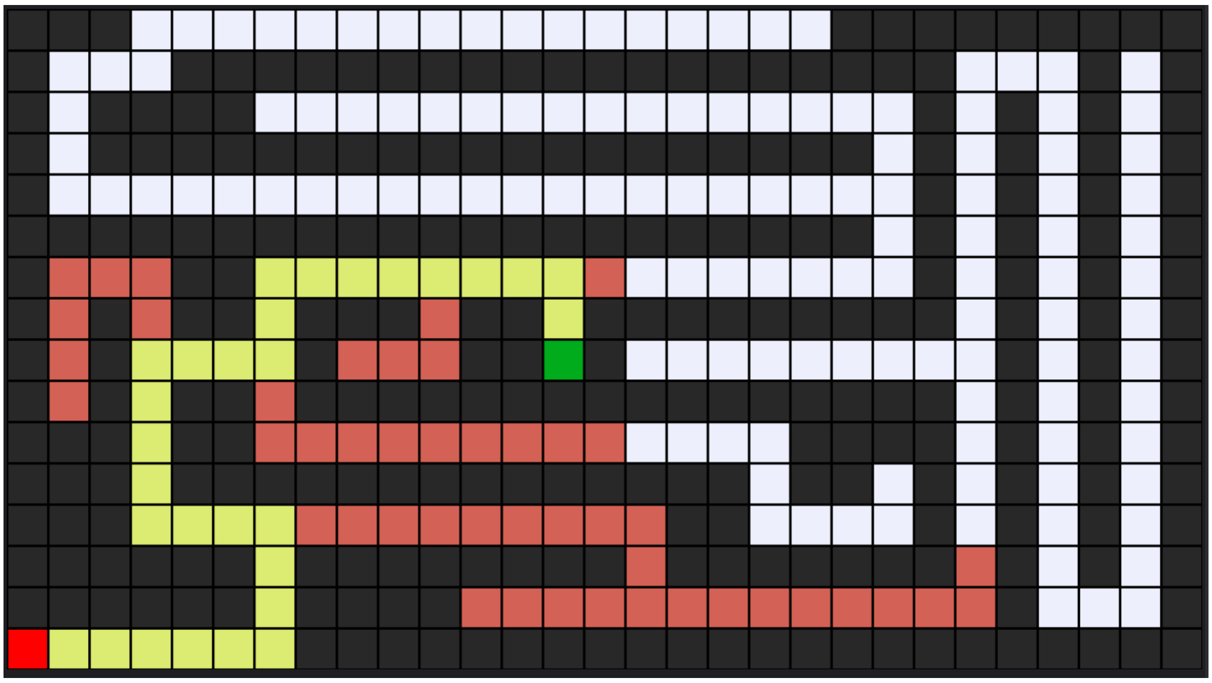
Again this problem is solved using BFS and run the same command in the terminal:



```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob2.txt
Maze:

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


Solving...
States Explored: 77
Solution:
```



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



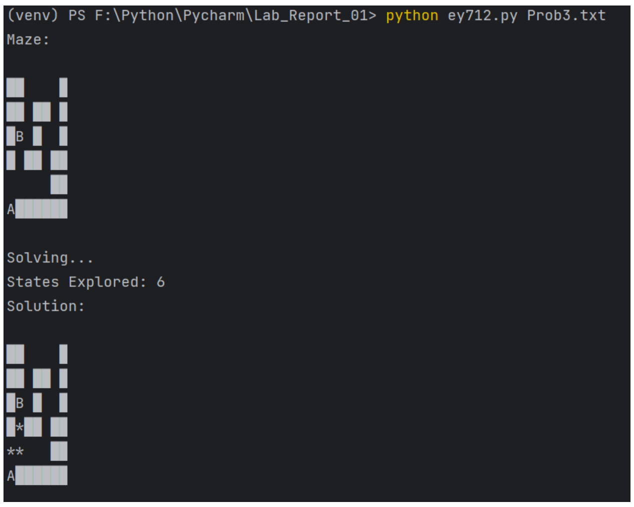
Again,we can see after solving this problem using the BFS method, States Explored is only 77.

For problem-02, here using the BFS method, states explored are only 77.And on other hand, same problem solved using DFS method, we can see states explored are 194. So, using BFS and DFS both methods, we can see the different result. In this problem, BFS is the best algorithm, because when using the BFS method it reduces more time complexity for finding the final state and BFS states explored value is less than DFS states explored value.
Finally,for Prob2.txt, BFS is the best search algorithm method better than DFS algorithm.

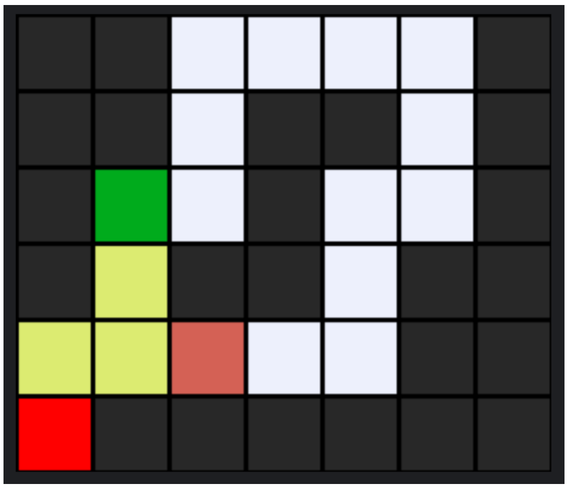
Problem-03:

Now solve the 3nd maze problem **Prob3.txt** using BFS. First of all, Open the terminal and run the command:

```
python ey712.py Prob3.txt
```

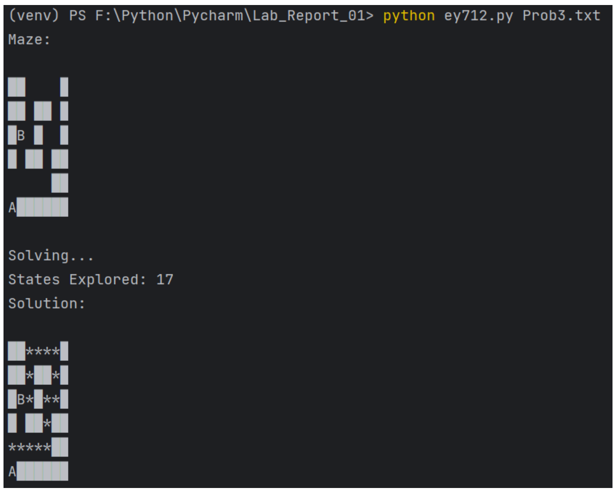


```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob3.txt
Maze:

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

Solving...
States Explored: 6
Solution:

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

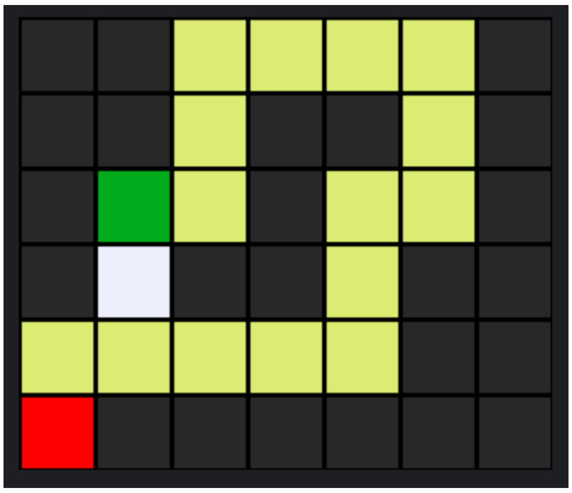


So,we can see after solving this problem using the BFS method, the States Explored value is only 6.

Again this problem is solved using DFS and run the same command in the terminal:



```
(venv) PS F:\Python\Pycharm\Lab_Report_01> python ey712.py Prob3.txt
Maze:

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

Solving...
States Explored: 17
Solution:

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



Again,we can see after solving this problem using the DFS method, the States Explored value is 17.

For problem-03, here using the BFS method, states explored are only 6.And on other hand, same problem solved using DFS method, we can see states explored are 17. So, using BFS and DFS both methods, we can see the different result. In this problem, BFS is the best algorithm, because when using the BFS method it reduces more time complexity for finding the final state and BFS states explored value is less than DFS states explored value.
Finally,for Prob3.txt, BFS is the best search algorithm method better than DFS algorithm.