Depth-First Traversal using Clear Route (Street, Highway) and Unclear Route (Hotel, Hospital)

19819 - Dawit Woldemichael

Professor Henry Chang

Contents

1. Introduction

The '490. The Maze' problem from LeetCode requires finding if there exists a path from a given start cell to a destination cell in a maze. The maze is represented as a 2D matrix, where 0 denotes an empty cell, 1 denotes a wall, and the path can be traversed horizontally or vertically through consecutive empty cells.

In this report, we will present a Python solution based on the Depth-First Traversal approach to address this problem.

2. Design

The Depth-First Traversal (DFS) approach is chosen due to its ability to efficiently explore all possible paths in the maze until it reaches the destination or exhausts all options. The recursive DFS algorithm allows us to backtrack and explore other directions when the current path leads to a dead-end.

The maze represents a graph-like structure with nodes as cells and edges between adjacent cells.

We need to find a path from the start to the destination cell in the maze.

The depth-first traversal is suitable for exploring paths in graphs and is applicable in maze-solving problems.

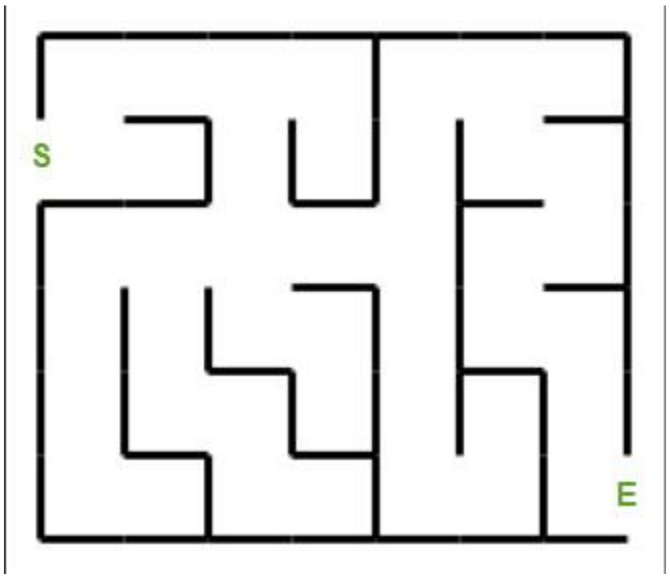
Other possible approaches include Breadth-First Traversal and Dijkstra's algorithm, but DFS is chosen due to its simplicity and effectiveness in exploring maze paths.

Breadth-First Traversal guarantees the shortest path, but it may explore a large number of nodes before finding the destination.

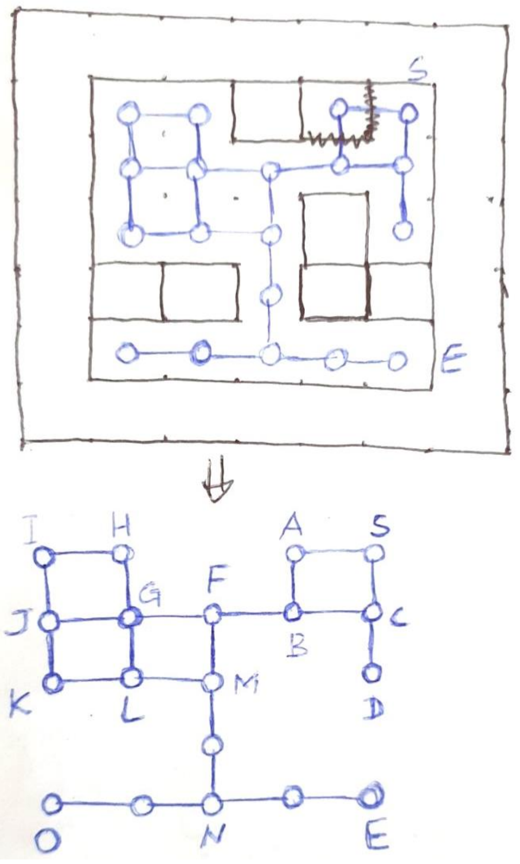
Dijkstra's algorithm considers edge weights and can be used to identify the shortest path between two given points.

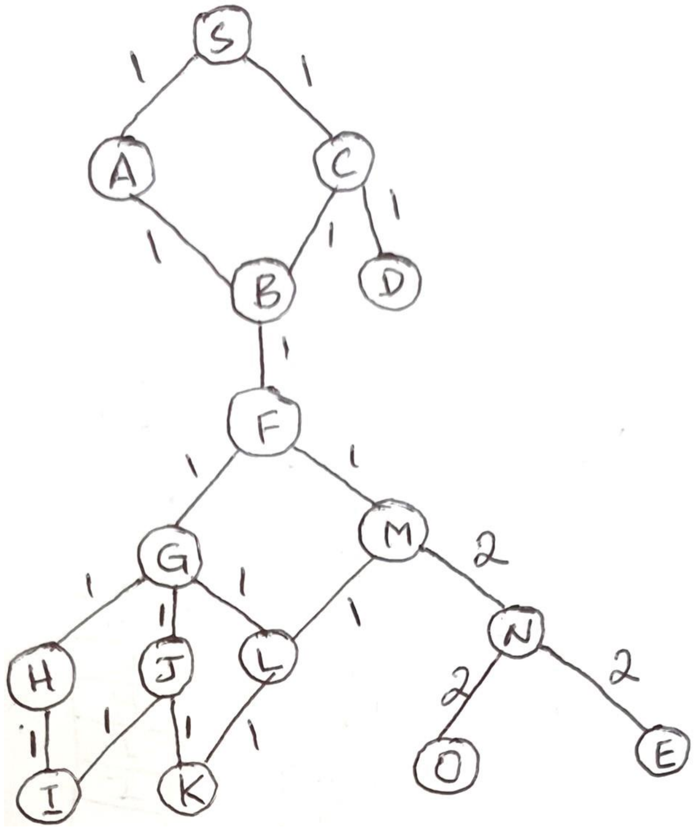
DFS provides a solution if it exists and is more efficient in terms of space complexity compared to BFS.

1.1. Manual process to demonstrate concepts.

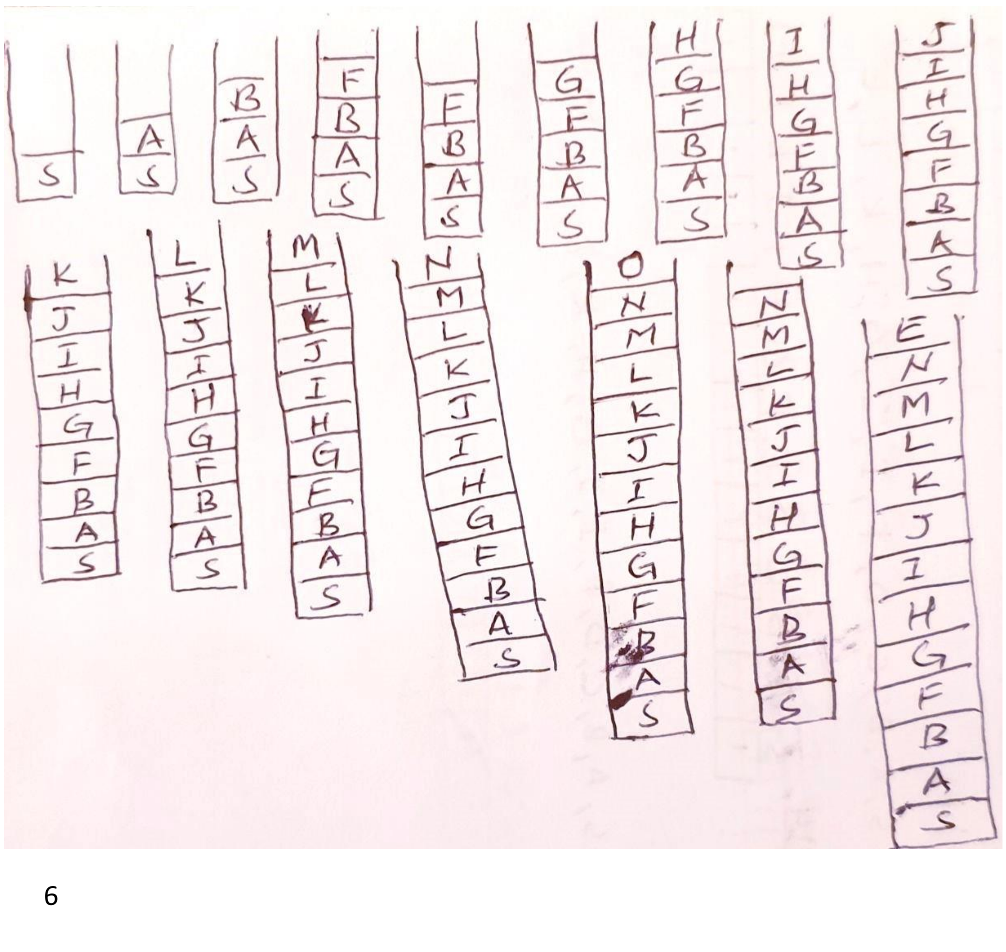


1.1.1. Tree



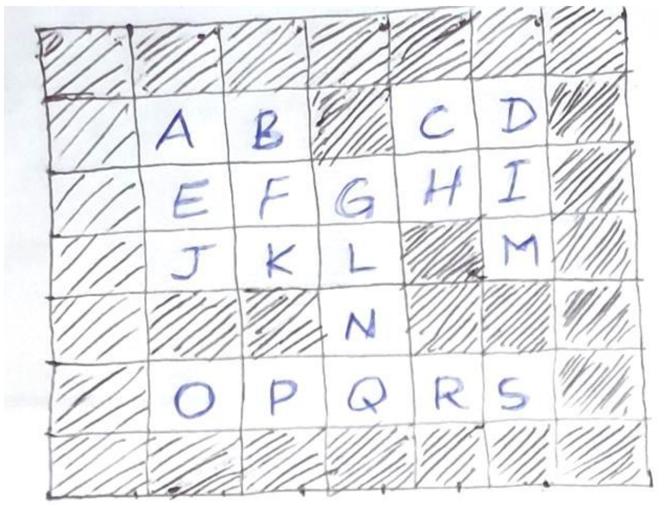


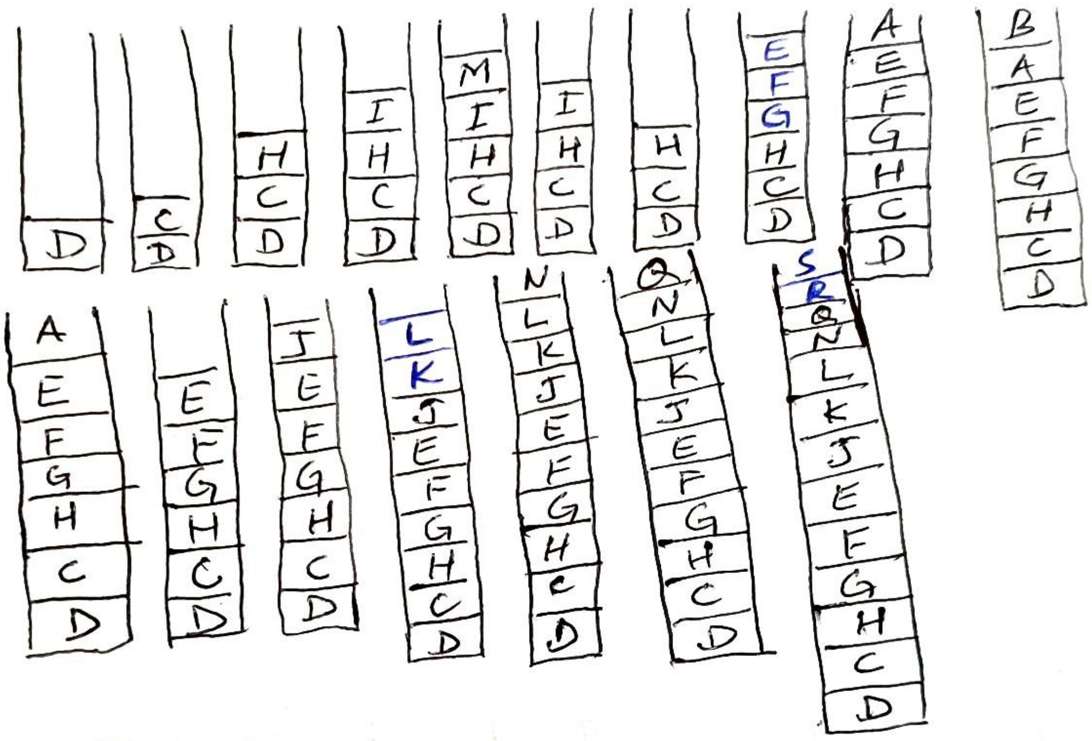
S
A
C
B
D
F
G
M
H
J
L
N
I
K
O
E
1
2



S
A
B
F
G
H
I
J
K
L
M
N
O
E

1.1.2. Matrix





3. Implementation

I used Python for solving the problem using DFS approach. The Python code implementation for solving the problem involves the following steps:

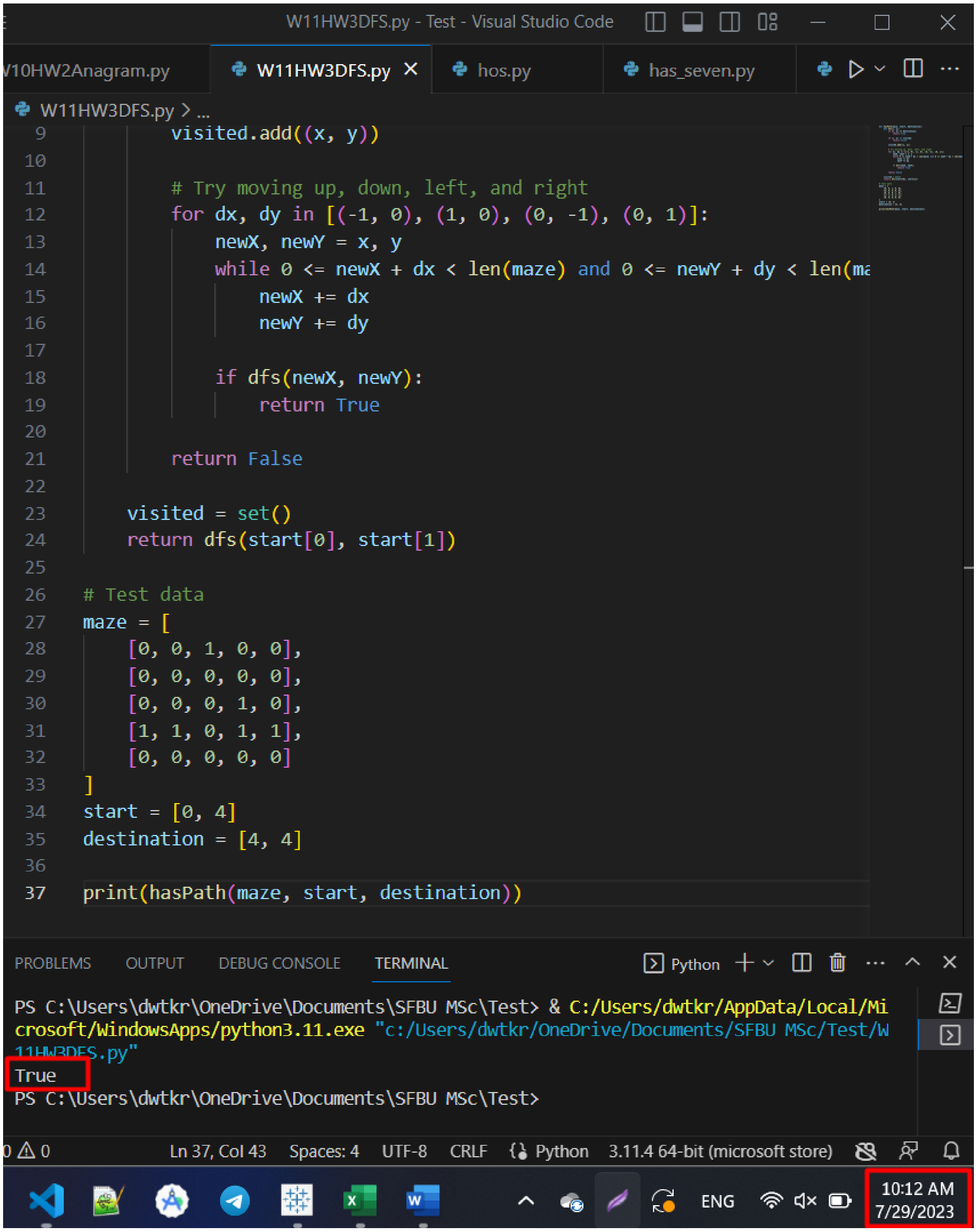
- Define the four possible directions: right, left, up, down.
- Implement a recursive dfs function to explore the maze.
- Mark visited cells as '2' to avoid revisiting them.
- Backtrack and explore other directions if a dead-end is reached.

```python
def hasPath(maze, start, destination):
    # Define directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def dfs(x, y):
        if [x, y] == destination:
            return True

        # Mark the current cell as visited
        maze[x][y] = 2

        for dx, dy in directions:
            # Keep moving in a certain direction until you hit a wall or reach the boundary
            newX, newY = x + dx, y + dy
            while 0 <= newX < len(maze) and 0 <= newY < len(maze[0]) and maze[newX][newY] != 1:
                newX += dx
                newY += dy

            # Backtrack and explore other directions
            if dfs(newX - dx, newY - dy):
                return True

        return False

    return dfs(start[0], start[1])

# Test data
maze = [[0, 0, 1, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 1, 0], [1, 1, 0, 1, 1], [0, 0, 0, 0, 0]]
start = [0, 4]
```

```
destination = [4, 4]

print(hasPath(maze, start, destination))  # Output: True
```

4. Test

The provided test data is used to validate the correctness of the implemented solution. The test data includes a sample maze, start cell, and destination cell with the expected output.



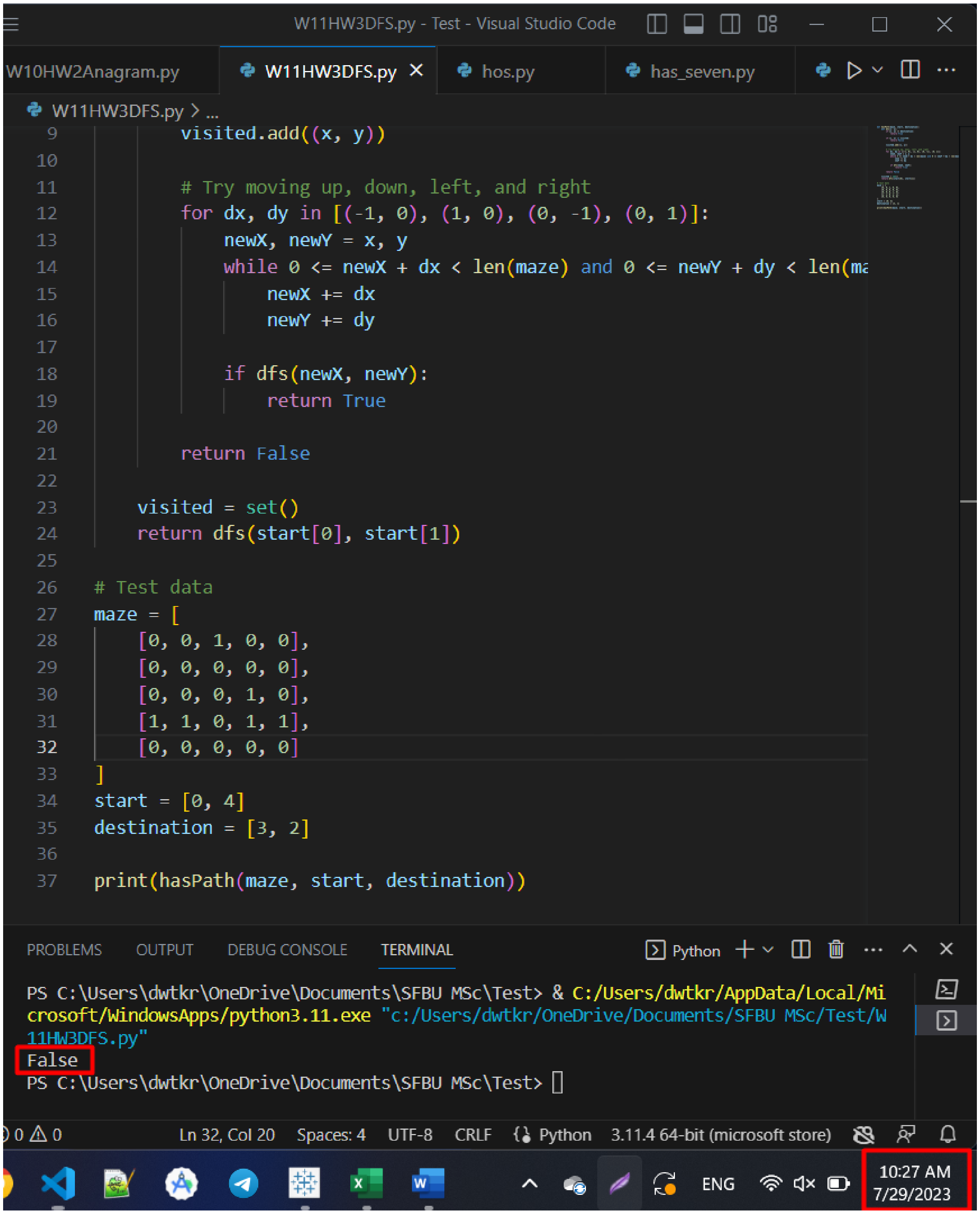
W11HW3DFS.py - Test - Visual Studio Code

```
            visited.add((x, y))

            # Try moving up, down, left, and right
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                newX, newY = x, y
                while 0 <= newX + dx < len(maze) and 0 <= newY + dy < len(ma
                    newX += dx
                    newY += dy

                if dfs(newX, newY):
                    return True

            return False

        visited = set()
        return dfs(start[0], start[1])

# Test data
maze = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 1],
    [0, 0, 0, 0, 0]
]
start = [0, 4]
destination = [4, 4]

print(hasPath(maze, start, destination))
```

```
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test> & C:/Users/dwtkr/AppData/Local/Mi
crosoft/WindowsApps/python3.11.exe "c:/Users/dwtkr/OneDrive/Documents/SFBU MSc/Test/W
11HW3DFS.py"
True
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test>
```

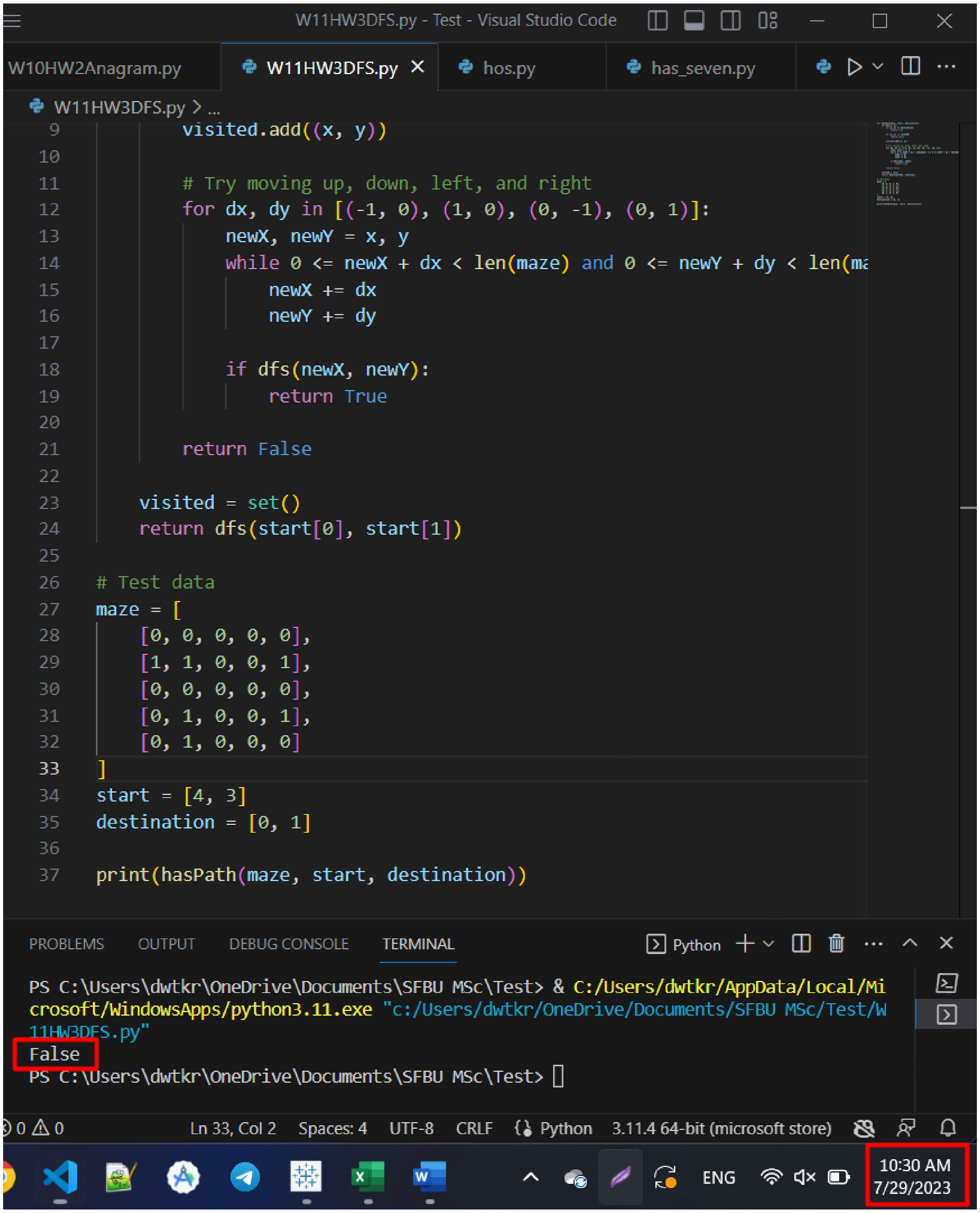
Ln 37, Col 43 Spaces: 4 UTF-8 CRLF Python 3.11.4 64-bit (microsoft store)

10:12 AM
7/29/2023



```python
            visited.add((x, y))

            # Try moving up, down, left, and right
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                newX, newY = x, y
                while 0 <= newX + dx < len(maze) and 0 <= newY + dy < len(ma
                    newX += dx
                    newY += dy

                if dfs(newX, newY):
                    return True

            return False

        visited = set()
        return dfs(start[0], start[1])

# Test data
maze = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 1],
    [0, 0, 0, 0, 0]
]
start = [0, 4]
destination = [3, 2]

print(hasPath(maze, start, destination))
```

```
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test> & C:/Users/dwtkr/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/dwtkr/OneDrive/Documents/SFBU MSc/Test/W11HW3DFS.py"
False
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test> 
```



W11HW3DFS.py - Test - Visual Studio Code

W10HW2Anagram.py | W11HW3DFS.py | hos.py | has_seven.py

W11HW3DFS.py > ...

```python
            visited.add((x, y))

            # Try moving up, down, left, and right
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                newX, newY = x, y
                while 0 <= newX + dx < len(maze) and 0 <= newY + dy < len(ma
                    newX += dx
                    newY += dy

                if dfs(newX, newY):
                    return True

            return False

        visited = set()
        return dfs(start[0], start[1])

# Test data
maze = [
    [0, 0, 0, 0, 0],
    [1, 1, 0, 0, 1],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 0, 1],
    [0, 1, 0, 0, 0]
]
start = [4, 3]
destination = [0, 1]

print(hasPath(maze, start, destination))
```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL

```
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test> & C:/Users/dwtkr/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/dwtkr/OneDrive/Documents/SFBU MSc/Test/W11HW3DFS.py"
False
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test> 
```

0 ⚠ 0 Ln 33, Col 2 Spaces: 4 UTF-8 CRLF { } Python 3.11.4 64-bit (microsoft store)

ENG 10:30 AM 7/29/2023

5. Enhancement Ideas

The current implementation is functional, but there are potential enhancements:

- Implementing a non-recursive version of DFS to avoid potential stack overflow for large mazes.
- Optimizing the DFS algorithm by using memorization or dynamic programming techniques to avoid redundant computations in case of overlapping paths.

6. Conclusion

The Depth-First Traversal approach, implemented in Python, successfully solves the '490. The Maze' problem on LeetCode for the provided test data. The DFS algorithm explores the maze and determines if there is a path from the start to the destination. The solution provides a straightforward and efficient way to tackle maze-solving problems, making it a suitable choice for similar scenarios. Potential enhancements can be considered to further optimize the algorithm's performance for more extensive mazes.

7. Bibliography / References

https://hc.labnet.sfbu.edu/~henry/npu/classes/algorithm/tutorialpoints_dsa/slide/depth_first_traversal.html

http://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

8. GitHub Link

https://github.com/dawit-kiros/CS455Algorithm