

Project: "490. The Maze" – Leet Code -
Breadth-First Traversal

19819 - Dawit Woldemichael

Professor Henry Chang

Contents

1. Introduction	3
2. Design.....	4
2.1. Problem Understanding	4
2.2. Solution Investigation.....	4
2.3. Solution Selection	4
2.4. Manually Solving the problem	5
3. Implementation.....	7
4. Test	9
5. Enhancement Ideas.....	12
6. Conclusion	14
7. GitHub Link.....	15
8. References.....	16

1. Introduction

The "Maze" problem, as described in LeetCode question 490, poses an intriguing challenge in the realm of pathfinding and graph traversal. This problem revolves around navigating a complex labyrinth-like structure, referred to as a maze, to ascertain the possibility of reaching a specific destination from a designated starting point. The maze itself is represented as a 2D grid, where each cell can take on one of two states: it can either be a "wall" or an "empty cell."

The central task of this problem can be distilled into determining whether a viable path exists from a predefined starting cell to a desired destination cell, while adhering to the rule that movement is solely permissible through empty cells. To address this challenge, an algorithmic approach known as Breadth-First Traversal (BFT) is employed. Breadth-First Traversal, a fundamental technique in graph theory, serves as an ideal strategy due to its distinctive characteristics that align seamlessly with the requirements of this problem.

2. Design

2.1. Problem Understanding

The objective of the problem is to determine if a path exists from the starting point to the destination point, navigating only through empty cells while avoiding walls. The problem can be approached using graph traversal algorithms, with Breadth-First Traversal being a suitable choice due to its ability to find the shortest path.

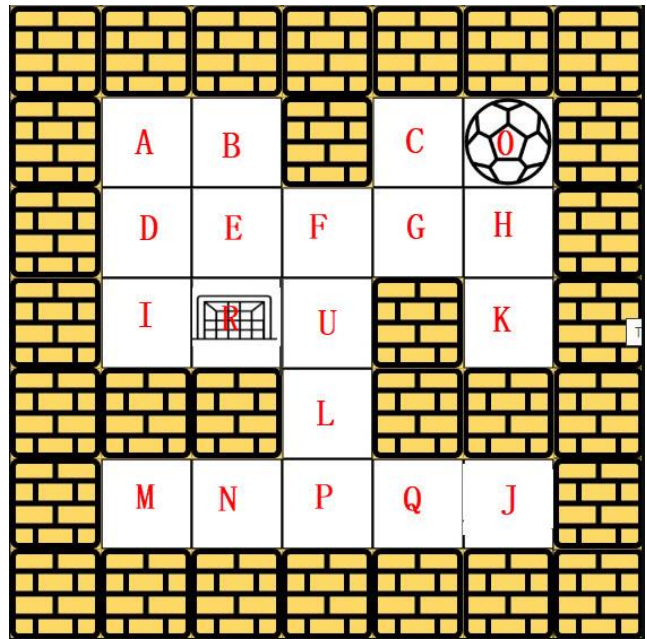
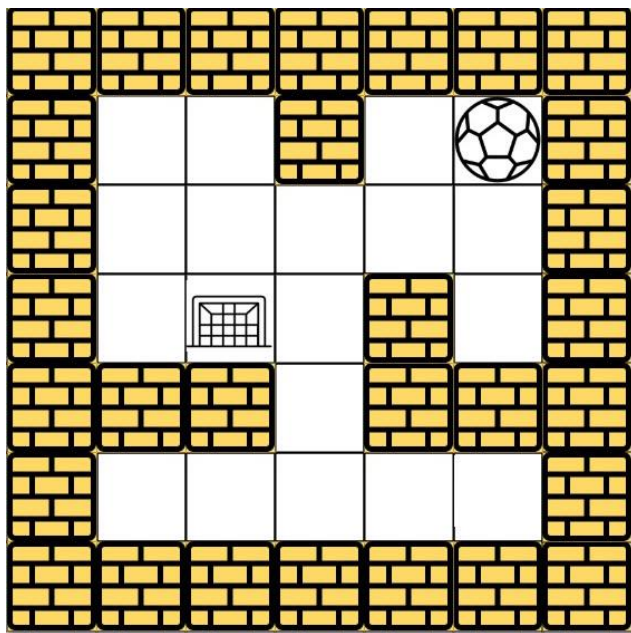
2.2. Solution Investigation

1. **Depth-First Traversal:** One possible solution is to use Depth-First Traversal (DFT), exploring as far as possible along each branch before backtracking. However, DFT may not guarantee finding the shortest path.
2. **Breadth-First Traversal:** BFT explores all neighbors of a node before moving on to their child nodes. This property makes it suitable for finding the shortest path in this problem.

2.3. Solution Selection

Breadth-First Traversal is chosen as the solution due to its ability to find the shortest path in a maze. Unlike DFT, BFT ensures that the first path found is the shortest. This property aligns well with the problem's requirement to determine whether a path exists between two points in a maze.

2.4. Manually Solving the problem



①

Step 1: Visited:

0

Queue:

Step 2: Visited:

0	1
---	---

Queue: 0

1. Add 0 to the queue
2. Mark 0 as visited

Step 3: Visited:

0	C	K
---	---	---

Queue: C, K

1. Add C & K to the queue
2. Mark C and K as visited

Step 4: Visited:

0	C	K	
---	---	---	--

Queue: K

1. Remove C from queue
2. Print 0, C

Step 5: Visited:

0	C	K	G
---	---	---	---

Queue: K, G

1. Add G to the queue
2. Mark G as visited

Step 6: Visited:

0	C	K	G	
---	---	---	---	--

Queue: G

1. Remove K from the queue
2. Print 0, C, K

Step 7: Visited:

0	C	K	G	
---	---	---	---	--

Queue:

1. Remove G from queue
2. Print 0, C, K, G

Step 8: Visited:

0	C	K	G	D
---	---	---	---	---

Queue: D

1. Add D to the queue
2. Mark D as visited

Step 9: Visited:

0	C	K	G	D	
---	---	---	---	---	--

Queue:

1. Remove D from queue
2. Print 0, C, K, G, D

Step 10:

Visited:

0	C	K	G	D	A	I
---	---	---	---	---	---	---

Queue: A, I

1. Add A, I to the queue
2. Mark A, I as visited

Step 11: Visited:

0	C	K	G	D	A	I
---	---	---	---	---	---	---

Queue: I

1. Remove A from the queue
2. Print 0, C, K, G, D, A

Step 12: Visited:

0	C	K	G	D	A	I	B
---	---	---	---	---	---	---	---

Queue: B

1. Remove I from the queue
2. Print 0, C, K, G, D, A, I

Step 13: Visited:

0	C	K	G	D	A	I	B	R
---	---	---	---	---	---	---	---	---

Queue: R

1. Remove B from the queue
2. Print 0, C, K, G, D, A, I, B

Step 14: Visited:

0	C	K	G	D	A	I	B	R
---	---	---	---	---	---	---	---	---

Queue:

1. Remove R from the queue
2. Print 0, C, K, G, D, A, I, B

Destination cell Reached (R) from 0.

3. Implementation

The Breadth-First Traversal approach involves using a queue to explore nodes level by level. The algorithm starts from the given start point and iteratively explores neighboring cells in the four possible directions: right, left, up, and down. It continues expanding the queue until the destination point is reached or all possible paths are exhausted.

I have done the implementation using Python programming language.

```
from collections import deque

def hasPath(maze, start, destination):
    rows = len(maze)
    cols = len(maze[0])
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up

    queue = deque([start])
    visited = set(start)

    while queue:
        current_row, current_col = queue.popleft()

        if [current_row, current_col] == destination:
            return True

        for dr, dc in directions:
            new_row, new_col = current_row + dr, current_col + dc

            # Move in the current direction until you hit a wall or go out of
            # bounds
            while 0 <= new_row < rows and 0 <= new_col < cols and
            maze[new_row][new_col] == 0:
                new_row += dr
                new_col += dc

            # Adjust back to valid cell
            new_row -= dr
            new_col -= dc

            if (new_row, new_col) not in visited:
                queue.append((new_row, new_col))
                visited.add((new_row, new_col))
```

```
        return False

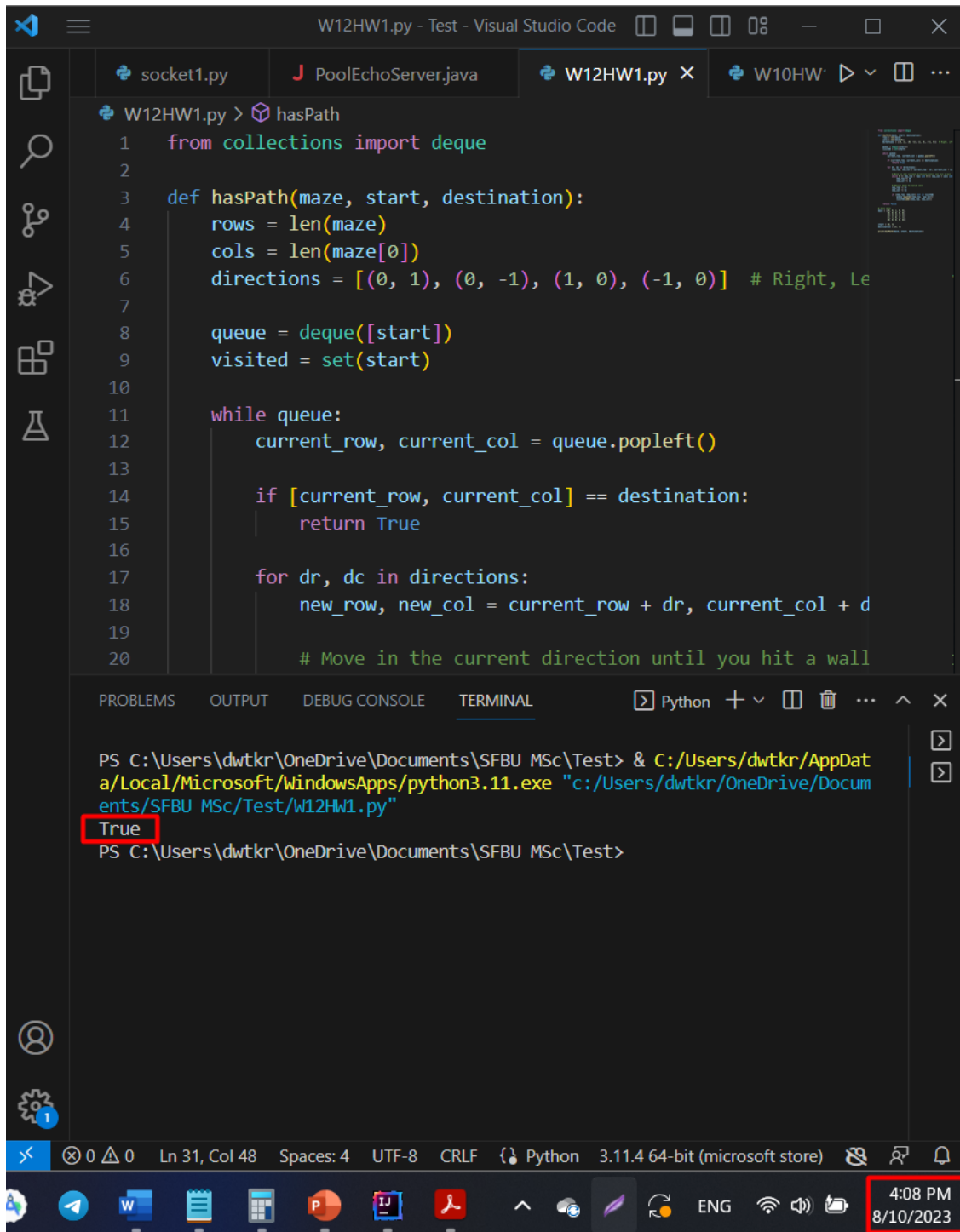
# Test data
maze = [[0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0],
        [1, 1, 0, 1, 1],
        [0, 0, 0, 0, 0]]

start = [0, 4]
destination = [4, 4]

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


4. Test

The implementation is tested using the provided test data as follows:



```
W12HW1.py > hasPath
1 from collections import deque
2
3 def hasPath(maze, start, destination):
4     rows = len(maze)
5     cols = len(maze[0])
6     directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up
7
8     queue = deque([start])
9     visited = set(start)
10
11     while queue:
12         current_row, current_col = queue.popleft()
13
14         if [current_row, current_col] == destination:
15             return True
16
17         for dr, dc in directions:
18             new_row, new_col = current_row + dr, current_col + dc
19
20             # Move in the current direction until you hit a wall
```

```
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/W12HW1.py"
True
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test>
```

Ln 31, Col 48 Spaces: 4 UTF-8 CRLF Python 3.11.4 64-bit (microsoft store) 4:08 PM 8/10/2023

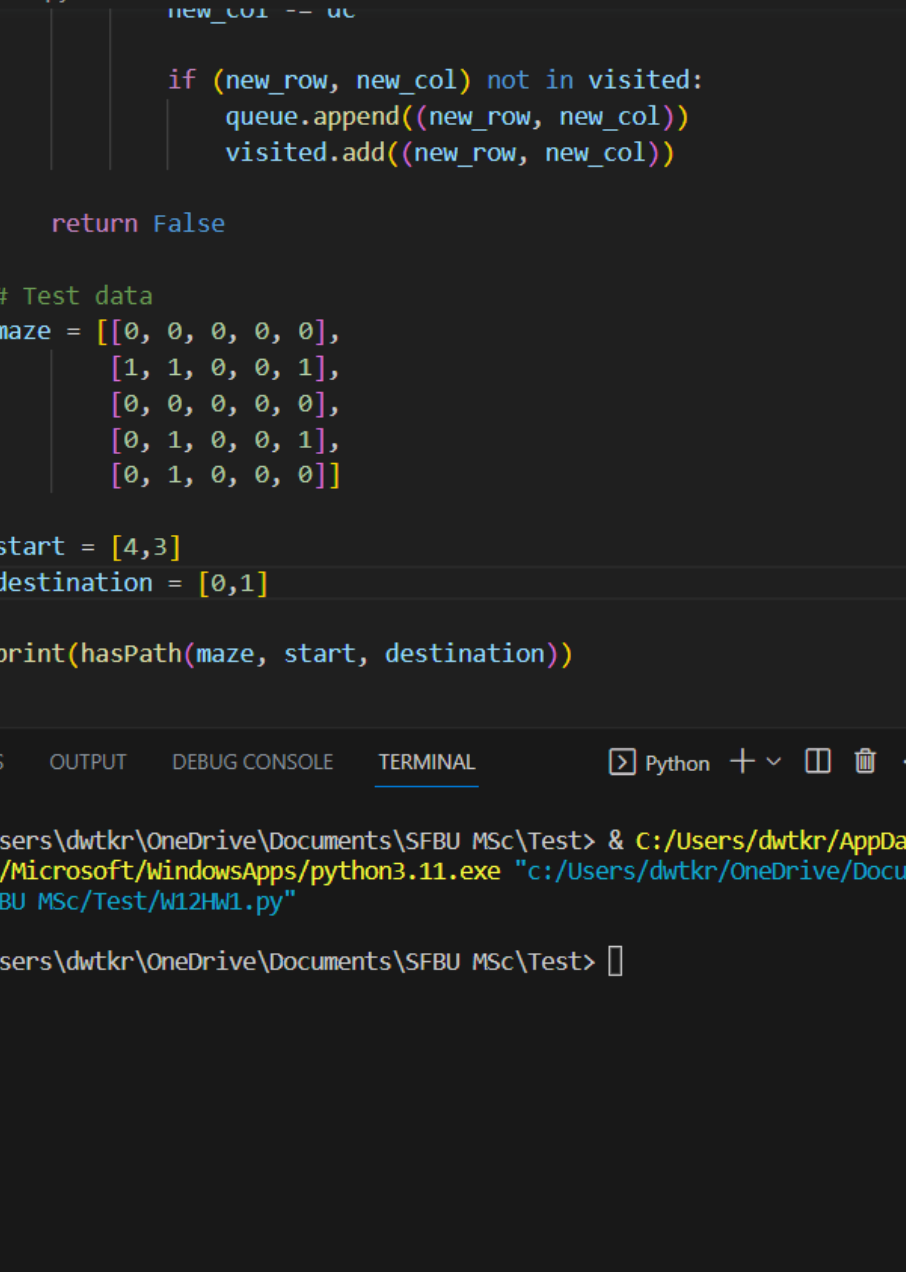
The image shows a Visual Studio Code editor window with the file 'W12HW1.py' open. The code is a Python script for a maze pathfinding problem. It defines a maze, a start point, and a destination, and then prints the result of a pathfinding function.

```
W12HW1.py > ...
27         new_col = new_col + 1
28
29         if (new_row, new_col) not in visited:
30             queue.append((new_row, new_col))
31             visited.add((new_row, new_col))
32
33     return False
34
35 # Test data
36 maze = [[0, 0, 1, 0, 0],
37         [0, 0, 0, 0, 0],
38         [0, 0, 0, 1, 0],
39         [1, 1, 0, 1, 1],
40         [0, 0, 0, 0, 0]]
41
42 start = [0, 4]
43 destination = [3, 2]
44
45 print(hasPath(maze, start, destination))
46
```

The terminal window shows the command to run the script and the output:

```
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/w12HW1.py"
False
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSc\Test>
```

The status bar at the bottom indicates the file is at line 46, column 1, with 4 spaces. The Python version is 3.11.4 64-bit (microsoft store). The system clock shows 4:09 PM on 8/10/2023.



The screenshot shows the Visual Studio Code interface. The editor window displays a Python file named `W12HW1.py` with the following code:

```
27         new_col = new_col + 1
28
29         if (new_row, new_col) not in visited:
30             queue.append((new_row, new_col))
31             visited.add((new_row, new_col))
32
33     return False
34
35 # Test data
36 maze = [[0, 0, 0, 0, 0],
37         [1, 1, 0, 0, 1],
38         [0, 0, 0, 0, 0],
39         [0, 1, 0, 0, 1],
40         [0, 1, 0, 0, 0]]
41
42 start = [4,3]
43 destination = [0,1]
44
45 print(hasPath(maze, start, destination))
46
```

The `TERMINAL` panel at the bottom shows the command prompt output:

```
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/W12HW1.py"
False
PS C:\Users\dwtkr\OneDrive\Documents\SFBU MSC\Test>
```

The status bar at the bottom indicates the file is at line 43, column 19, with 4 spaces, UTF-8 encoding, CRLF line endings, and the Python 3.11.4 64-bit (microsoft store) interpreter.

5. Enhancement Ideas

5.1. Efficiency: Avoiding Redundant Visits

While the Breadth-First Traversal (BFT) algorithm efficiently explores the maze by systematically moving level by level, there is an opportunity to enhance its performance by minimizing the number of visits to cells that have already been explored.

When applying BFT, each cell's neighbors are added to the traversal queue for exploration. However, as the traversal progresses, some cells may become reachable through multiple paths, leading to redundant visits. This redundancy can increase the algorithm's time complexity and computational resources.

To address this issue, a mechanism can be introduced to keep track of the cells that have already been visited. This can be achieved through the utilization of a data structure such as a hash set or a boolean grid. Whenever a cell is added to the traversal queue, it should also be marked as visited in the data structure. Before adding a cell to the queue, the algorithm should check if it has already been visited. If so, the cell can be skipped, avoiding the unnecessary duplication of efforts.

By implementing this optimization, the algorithm's efficiency can be significantly improved, reducing the number of operations required and accelerating the overall pathfinding process. This enhancement is particularly valuable in larger mazes or scenarios where computational resources are constrained.

5.2. Path Finding: Returning the Actual Path

An additional enhancement idea is to modify the algorithm to not only determine if a path exists but also to return to the actual path itself. This can be accomplished by maintaining a parent-child relationship between cells as they are explored. When the destination cell is reached, this relationship can be used to backtrack from the destination to the starting cell, reconstructing the optimal path taken.

While this enhancement introduces some complexity, it provides valuable information beyond the mere existence of a path. It empowers users with insights into the specific route that can be taken, aiding in visualization, analysis, and decision-making in applications where the path is crucial.

These enhancement ideas offer ways for refining the Breadth-First Traversal-based solution to the Maze problem. By fine-tuning the algorithm's efficiency and expanding its capabilities, the solution can become even more robust and versatile, accommodating a wider range of scenarios and requirements.

6. Conclusion

The Breadth-First Traversal approach proves effective in solving the Maze problem. It ensures that the shortest path between two points in a maze is found, providing a reliable solution for determining path existence. By following this approach, we can navigate through the maze's empty cells efficiently, making informed decisions based on the principles of graph traversal.

7. GitHub Link

<https://github.com/dawit-kiros/CS455AlgorithmBFT>

8. References

https://hc.labnet.sfbu.edu/~henry/npu/classes/algorithm/graph_alg/slide/maze.html#Approach%205:%20Wheeled%20robots%20move%20in%20a%20Hotel:%20BFS