CS501 Project: The Maze

By Natnael Haile
ID: 20007

Table of Contents

Introduction

- In this project, our aim is to solve maze problems using both DFT & BFT techniques.
- We will also implement the algorithms using Python with which we will test the test cases on Leetcode.
- Finally, we will compare and contrast the two solution approaches (DFT & BFT) and infer which one is better in terms of time and space complexity

Design

- The problem involves navigating a maze represented as a 2D grid to determine if a path exists from the start to the destination.
- We will discuss two approaches: Depth-First Search (DFS) and Breadth-First Search (BFS), each with its own advantages and trade-offs.

Design cont.

DFT

- Key Components:
  - Starting from the initial position, explore each possible path until reaching the destination or exhausting all options.
  - Backtrack when reaching dead ends to explore alternative paths.
- Implementation:
  - We'll implement DFS recursively to traverse the maze efficiently.

BFT

- Key Components:
  - Explore all possible paths level by level, starting from the initial position.
  - Ensures finding the shortest path first due to its nature of exploring closer nodes before moving farther.
- Implementation:
  - We'll implement BFS using a queue data structure to manage the traversal of the maze.

Design cont.

- Both DFS and BFS have a time complexity of $O(V + E)$, where V is the number of vertices (cells in the maze) and E is the number of edges (possible movements between cells).
- DFS uses less memory as it traverses deeper before backtracking, while BFS explores the maze level by level, potentially using more memory.
- Diagrams (sketches) illustrating how DFS and BFS algorithms traverse the maze will be provided to aid in understanding the traversal process and the solutions to the problems.
- Python code for both DFS and BFS solutions will be presented to clarify the algorithmic logic and implementation details.

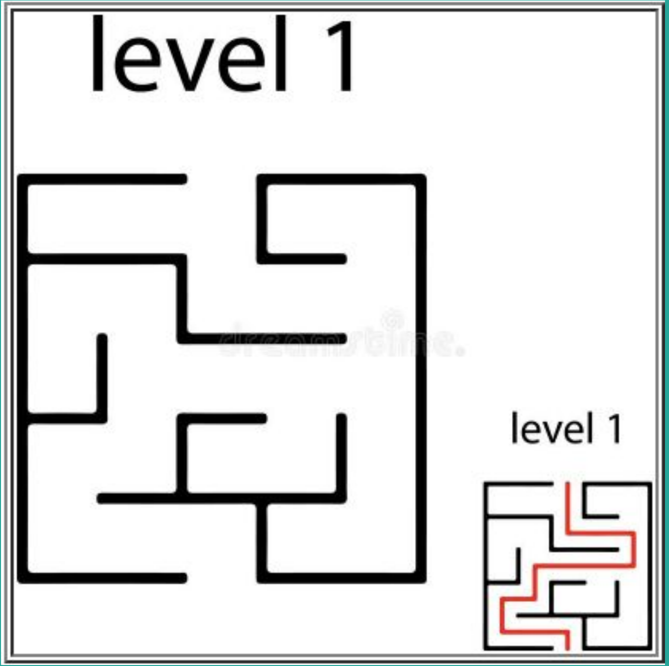
Implementation & Testing

1. DFT

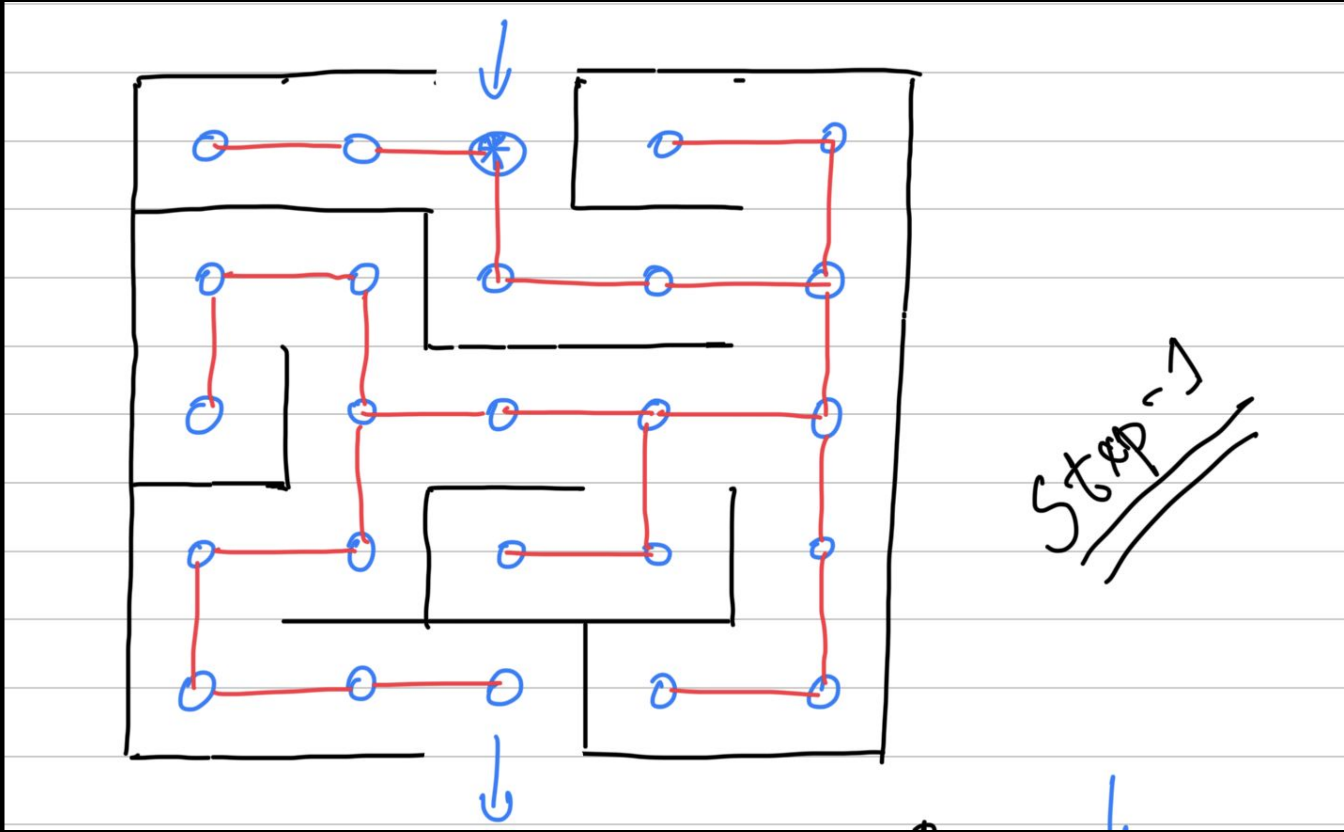
- Manual Solution for Problem 1
- Manual Solution for Problem 2
- LeetCode Solution - Python
- Test Cases Execution Screenshot

1. Manually solve the following problem using DFT (Depth First Traversal):

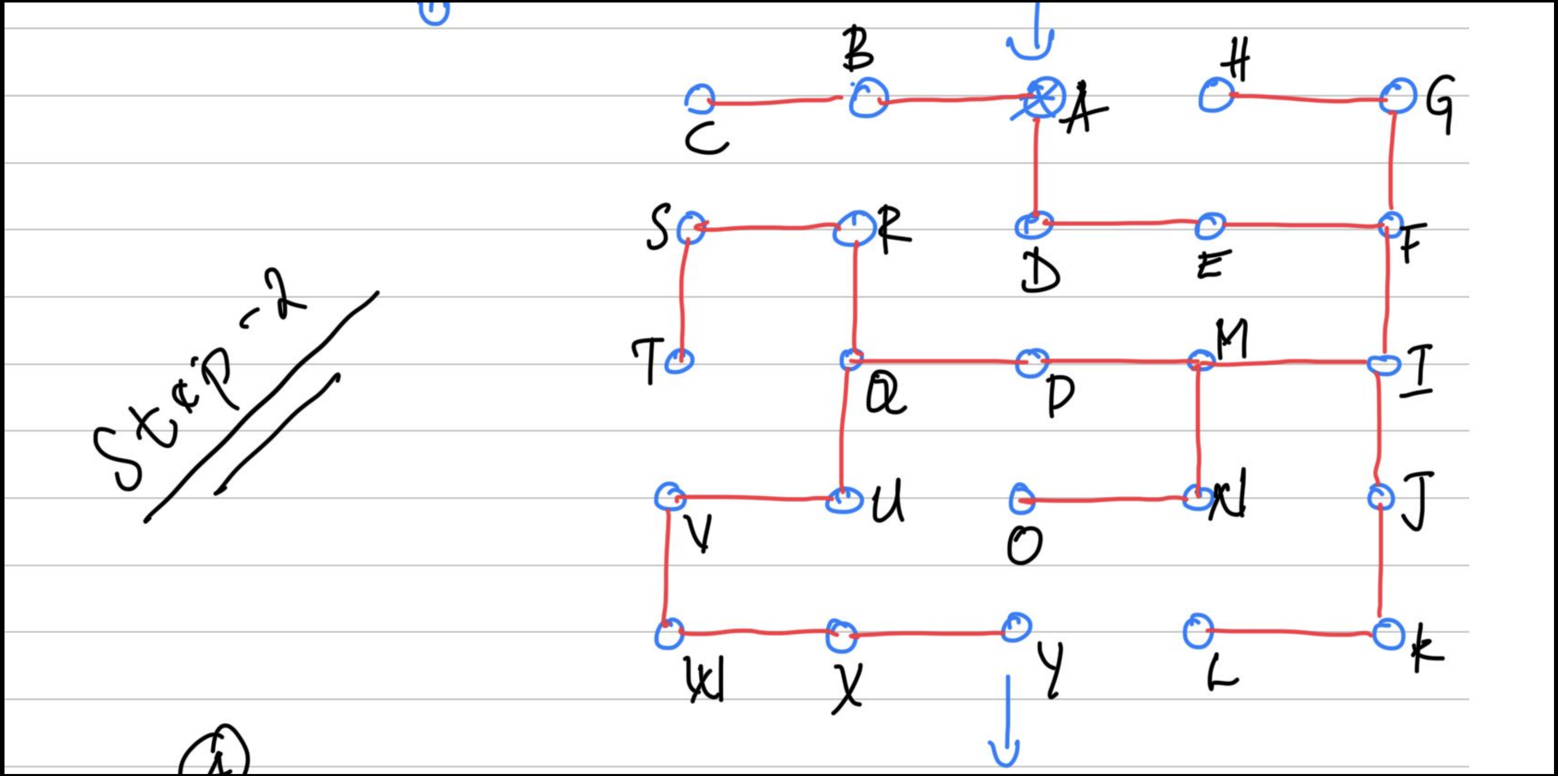
- Without Wheel



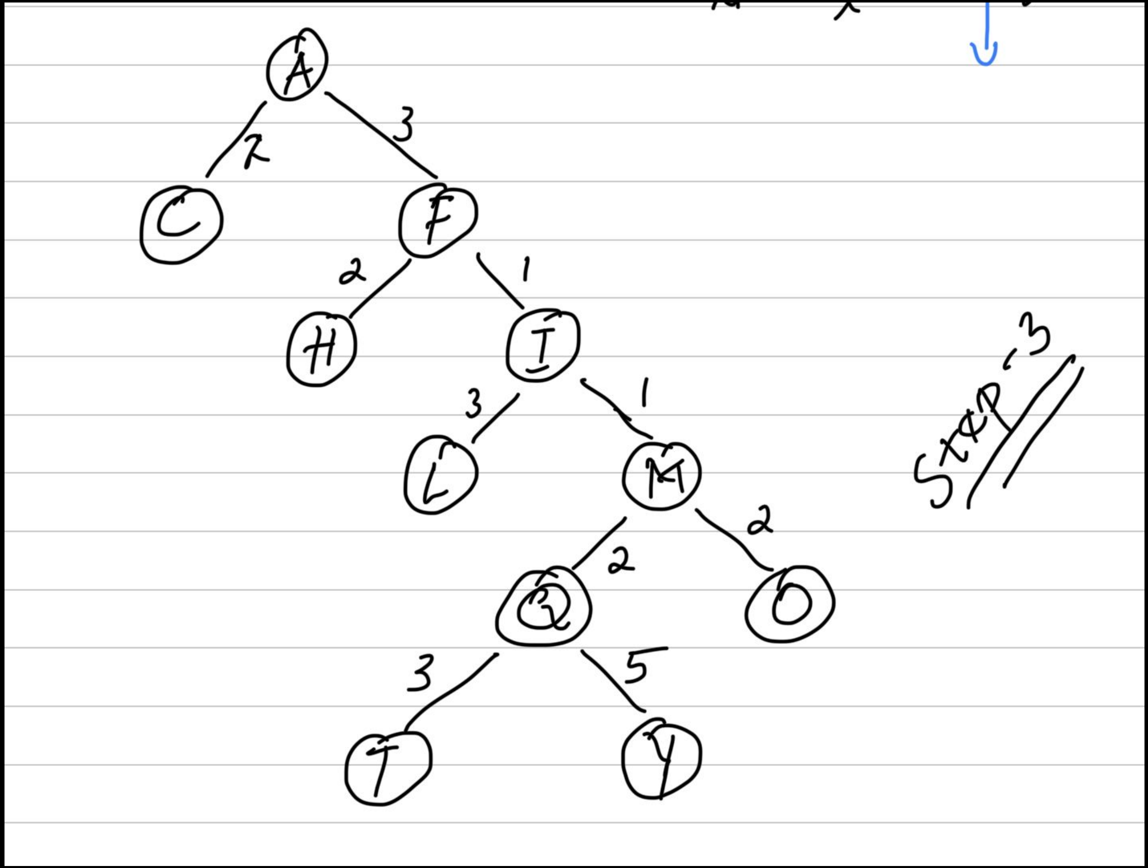
Build the tree first:



Build the tree first: cont.



Build the tree first: cont.

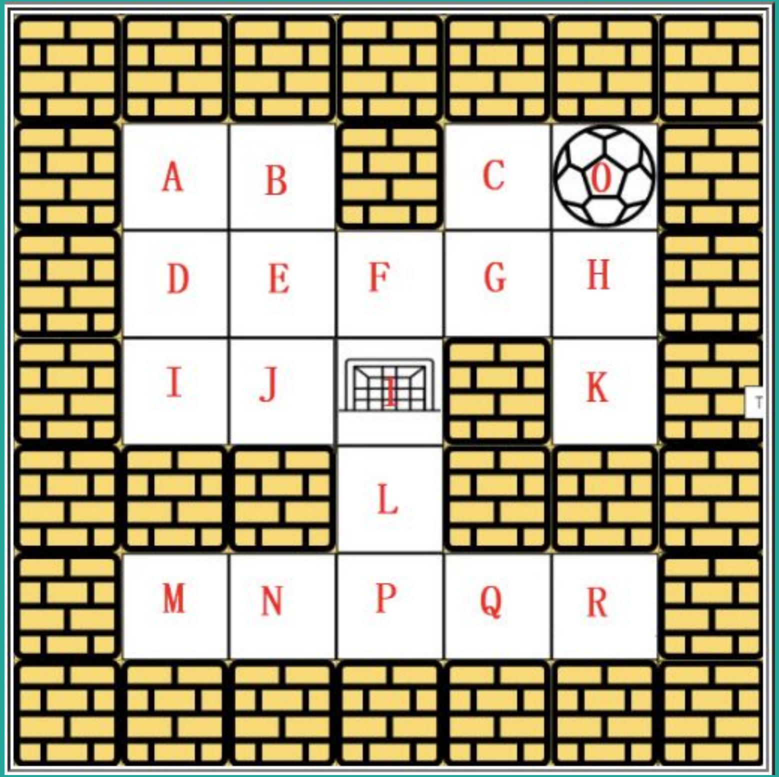


Now using DFT, the below table shows the traversed leaves added and popped from a stack.

													T		Y					
										O		Q	Q	Q	Q	Q				
							L		M	M	M	M	M	M	M	M	M			
				H		I	I	I	I	I	I	I	I	I	I	I	I	I		
	C		F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

2. Manually solve the following problem using DFT (Depth First Traversal):

- With Wheel



Using DFT, the following table shows the traversed leaves added and popped from a stack.

													1
									B				J
								A	A	A		I	I
							D	D	D	D	D	D	D
				K			E	E	E	E	E	E	E
			H	H	H		F	F	F	F	F	F	F
		G	G	G	G	G	G	G	G	G	G	G	G
	C	C	C	C	C	C	C	C	C	C	C	C	C
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Python Code Solution for Leetcode Problem 490.

```python
def hasPath(maze, start, destination) -> bool:
    if not maze:
        return False
    m, n = len(maze), len(maze[0])
    visited: set[tuple[int, int]] = set()
    def dfs(x: int, y: int):
        if [x, y] == destination:
            return True
        if (x, y) in visited:
            return False
        visited.add((x, y))
        # Try moving in all four directions
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for dx, dy in directions:
            newX: int = x + dx
            newY: int = y + dy
        # Keep moving until hitting a wall or the boundary
            while 0 <= newX < m and 0 <= newY < n and
maze[newX][newY] == 0:
                newX += dx
                newY += dy
            # When hitting a wall or boundary,
            # perform DFS from the new position
            if dfs(newX - dx, newY - dy):
                return True
        return False
    return dfs(start[0], start[1])
```

Execution/Output Screenshot

Welcome   501.py

Users > nati > Documents > sfbu > CS501 > homeworks > project > 501.py > hasPath

```python
def hasPath(maze: list[list[int]], start: list[int], destination: list[int]):
    if not maze:
        return False
    m, n = len(maze), len(maze[0])
    visited: set[tuple[int, int]] = set()
    def dfs(x: int, y: int):
        if [x, y] == destination:
            return True
        if (x, y) in visited:
            return False
        visited.add((x, y))
        # Try moving in all four directions
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for dx, dy in directions:
            newX: int = x + dx
            newY: int = y + dy
            # Keep moving until hitting a wall or the boundary
            while 0 <= newX < m and 0 <= newY < n and maze[newX][newY] == 0:
                newX += dx
                newY += dy
            # When hitting a wall or boundary, perform DFS from the new position
            if dfs(newX - dx, newY - dy):
                return True
        return False
    return dfs(start[0], start[1])
# Test the function
maze, start, destination = [[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]], [0, 4], [4, 4]
print(hasPath(maze, start, destination))  # Output: True
maze, start, destination = [[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]], [0, 4], [3, 2]
print(hasPath(maze, start, destination))  # Output: False
maze, start, destination = [[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]], [4, 3], [0, 1]
print(hasPath(maze, start, destination))  # Output: False
```

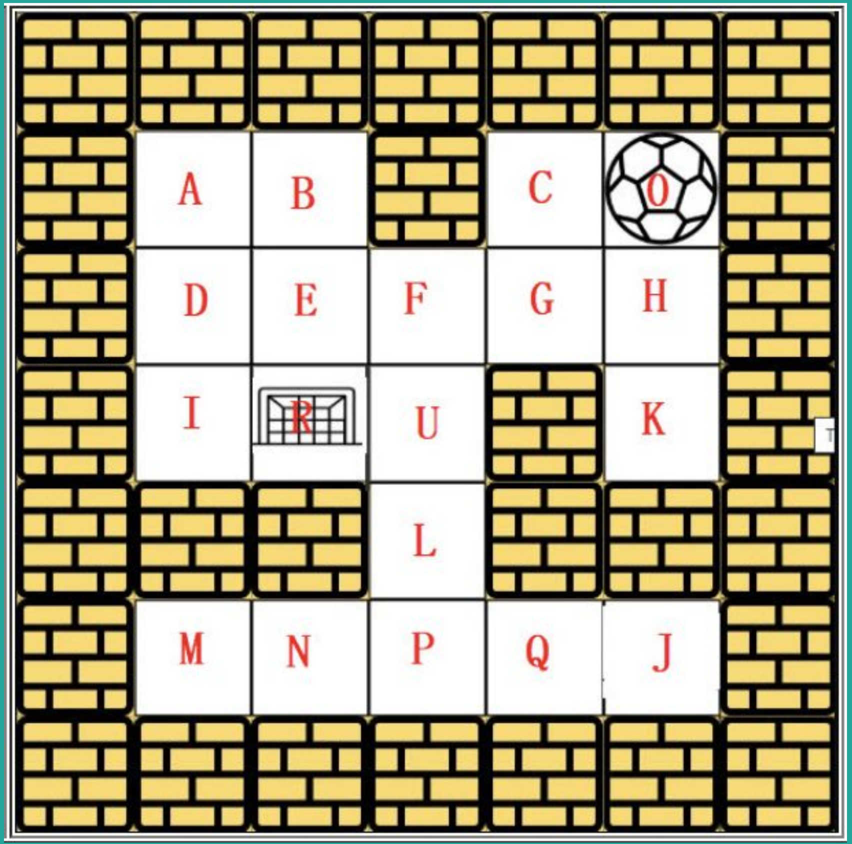
PROBLEMS   OUTPUT   TERMINAL   PORTS   SERIAL MONITOR   DEBUG CONSOLE

```
cd /Users/nati/Documents/sfbu/CS501/homeworks/project
/usr/bin/python3 /Users/nati/Documents/sfbu/CS501/homeworks/project/501.py
nati@Natis-Macbook ~ % cd /Users/nati/Documents/sfbu/CS501/homeworks/project
nati@Natis-Macbook project % /usr/bin/python3 /Users/nati/Documents/sfbu/CS501/homeworks/project/501.py
True
False
False
nati@Natis-Macbook project %
```

2. BFT

- Manual Solution for Problem 1
- LeetCode Solution - Python
- Test Cases Execution Screenshot

- Manually solve the following problem using BFT (Breadth First Traversal):
- With Wheel



Using BFT, the following table shows the traversed cells of the maze added and removed from a queue.

```
Visited: 0
         0
Queue:
```

```
Visited: 0
         1
Queue: 0
1. Add 0 to the queue
2. Mark 0 as visited
```

```
Visited: 0
         1
Queue:
1. Remove 0 from the queue
2. Print 0
```

```
Visited: 0 C K
         1 1 1
Queue: C K
1. Add C and K to the queue
2. Mark C and K as visited
```

```
Visited: 0 C K
         1 1 1
Queue: K
1. Remove C from the queue
2. Print 0 C
```

```
Visited: 0 C K G
         1 1 1 1
Queue: K G
1. Add G to the queue
2. Mark G as visited
```

```
Visited: 0 C K G
         1 1 1 1
Queue: G
1. Remove K from the queue
2. Print: 0 C K
```

```
Visited: 0 C K G
         1 1 1 1
Queue:
1. Remove G from the queue
2. Print 0 C K G
```

```
Visited: 0 C K G D
         1 1 1 1 1
Queue: D
1. Add D to the queue
2. Mark D as visited
```

```
Visited: 0 C K G D
         1 1 1 1 1
Queue:
1. Remove D from the queue
2. Print: 0 C K G D
```

```
Visited: 0 C K G D A I
         1 1 1 1 1 1 1
Queue: A I
1. Add A, I to the queue
2. Mark A, I as visited
Visited: 0 C K G D A I
         1 1 1 1 1 1 1
Queue: I
1. Remove A from the queue
2. Print: 0 C K G D A
```

```
Visited: 0 C K G D A I B
         1 1 1 1 1 1 1 1
Queue: I B
1. Add B to the queue
2. Mark B as visited
```

```
Visited: 0 C K G D A I B
         1 1 1 1 1 1 1 1
Queue: B
1. Remove I from the queue
2. Print: 0 C K G D A I
```

```
Visited: 0 C K G D A I B R
         1 1 1 1 1 1 1 1 1
Queue: B R
1. Add R to the queue
2. Mark R as visited
```

Python Code Solution for Leetcode Problem 490.

```python
from collections import deque

def hasPath(maze, start, destination) -> bool:
    if not maze or not maze[0]:
        return False
    rows, cols = len(maze), len(maze[0])
    visited: set[tuple[int, int]] = set()
    queue = deque([tuple(start)])
    directions: list[tuple[int, int]] = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    while queue:
        i, j = queue.popleft()
        if [i, j] == destination:
            return True
        if (i, j) in visited:
            continue
        visited.add((i, j))
        for dx, dy in directions:
            x, y = i, j
            while 0 <= x + dx < rows and 0 <= y + dy < cols and maze[x + dx][y + dy] == 0:
                x += dx
                y += dy
            if (x, y) not in visited:
                queue.append((x, y))
    return False
```

Execution/Output Screenshot

bft   dft.py

bft > hasPath

```python
from collections import deque

def hasPath(maze: list[list[int]], start: list[int], destination: list[int]) -> bool:
    if not maze or not maze[0]:
        return False
    rows, cols = len(maze), len(maze[0])
    visited: set[tuple[int, int]] = set()
    queue = deque([tuple(start)])
    directions: list[tuple[int, int]] = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    while queue:
        i, j = queue.popleft()
        if [i, j] == destination:
            return True
        if (i, j) in visited:
            continue
        visited.add((i, j))
        for dx, dy in directions:
            x, y = i, j
            while 0 <= x + dx < rows and 0 <= y + dy < cols and maze[x + dx][y + dy] == 0:
                x += dx
                y += dy
            if (x, y) not in visited:
                queue.append((x, y))
    return False
# Test the function
maze, start, destination = [[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]], [0, 4], [4, 4]
print(hasPath(maze, start, destination))  # Output: True
maze, start, destination = [[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]], [0, 4], [3, 2]
print(hasPath(maze, start, destination))  # Output: False
maze, start, destination = [[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]], [4, 3], [0, 1]
print(hasPath(maze, start, destination))  # Output: False
```

PROBLEMS   OUTPUT   TERMINAL   PORTS   SERIAL MONITOR   DEBUG CONSOLE

```
cd /Users/nati/Documents/sfbu/CS501/homeworks/project
/usr/bin/python3 /Users/nati/Documents/sfbu/CS501/homeworks/project/bft
nati@Natis-Macbook project % cd /Users/nati/Documents/sfbu/CS501/homeworks/project
nati@Natis-Macbook project % /usr/bin/python3 /Users/nati/Documents/sfbu/CS501/homeworks/project/bft
True
False
False
nati@Natis-Macbook project %
```

Enhancement Ideas

- Implement optimization techniques such as memoization to improve the efficiency of DFS and BFS algorithms.
- Explore parallelization strategies by dividing the maze into smaller sections to accelerate maze traversal.
- Utilize dynamic programming techniques to optimize the computation of solutions.

Conclusion

- Our analysis of this project using DFS and BFS techniques reveals their distinct approaches to maze traversal.
- DFS efficiently explores paths until a solution is found or a dead end is reached, while BFS systematically explores all possible paths level by level, prioritizing the shortest path.

Conclusion cont.

- We've outlined key components, implementation strategies, and complexities for both approaches.
- Visual aids and Python code enhance comprehension and implementation.
- With these approaches, we gain versatile tools for solving maze navigation problems efficiently, whether prioritizing simplicity and memory efficiency with DFS or guaranteeing the shortest path with BFS.
- Our exploration equips us with valuable problem-solving skills applicable to diverse scenarios.

References

- robot path planning
- Robotic Path Planning
- Breadth-First and Depth-First Search for Path Planning
- A framework for robot path finding in unstructured environments
- A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games

Github Link

https://github.com/cur10usityDrives/Algorithms/tree/main


THANK YOU