490. The Maze

Shruti Kavishwar
San Francisco Bay University

Guided By: Dr. Henry Chang

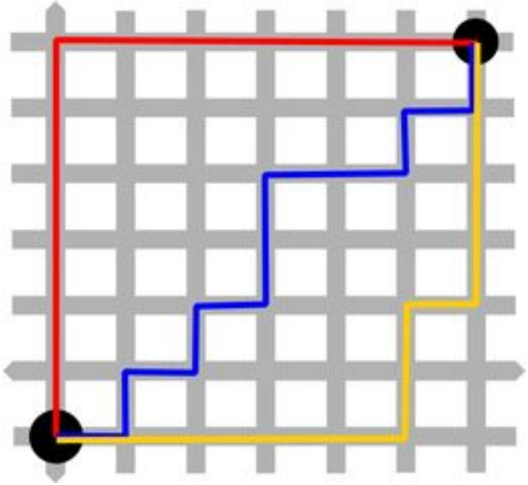
Table of Contents

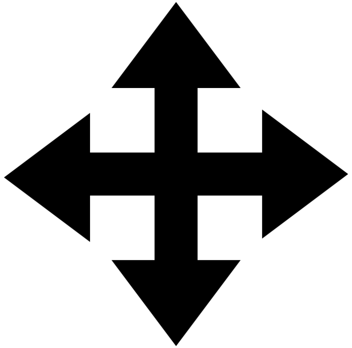
Problem Statement

There is a ball in a maze with empty spaces (represented as 0) and walls (represented as 1). The ball can go through the empty spaces by rolling **up, down, left or right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

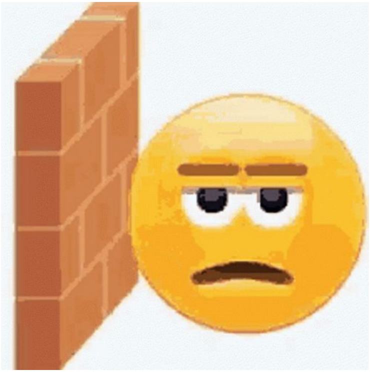
Given the m x n maze, the ball's start position and the destination, where start = [$start_{row}$, $start_{col}$] and destination = [$destination_{row}$, $destination_{col}$], return true if the ball can stop at the destination, otherwise return false.

Understanding the Problem Statement





a. Traversal in manhattan/taxicab geometry
b. Right, Left, Up, Down directions

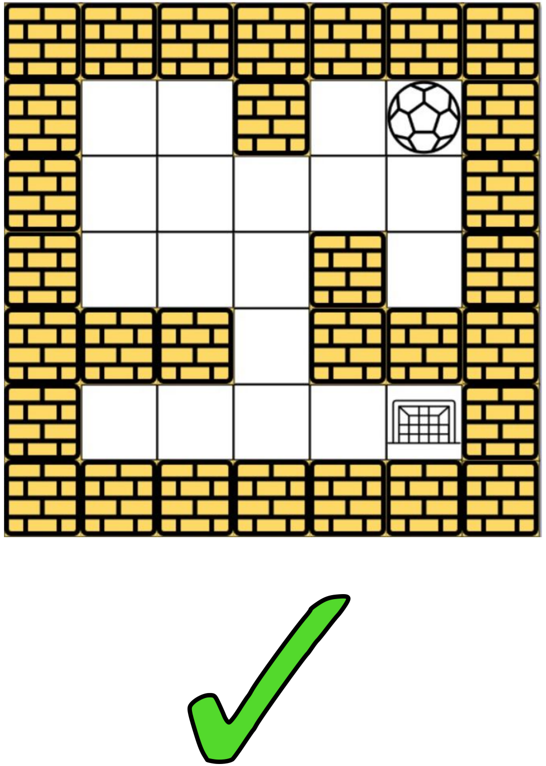


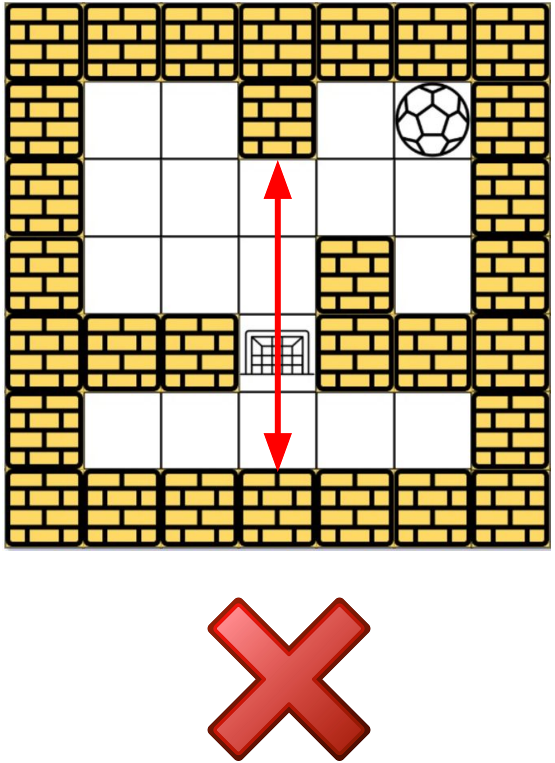
Ball stops when it hits the wall

1 — **Wall**

0 — **Empty Space**

Will the ball reach the goal ?





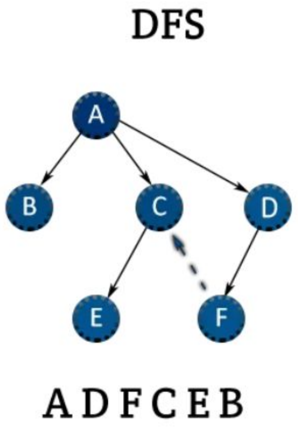
Legged Robot VS Wheeled Robot

- Moves one cell at a time
- Directions are **Right -> Left -> Up -> Down**

- Moves in a direction until it hits the wall
- Directions it can move are **Right -> Left -> Up -> Down**

Depth First Traversal VS Breadth First Traversal

- Uses a top down approach to reach the destination.
- Uses Stack Data Structure which follows Last In First Out (LIFO) approach



- Uses left right (horizontal) approach to reach the destination.
- Uses Queue Data Structure which follows First in First Out (FIFO) approach

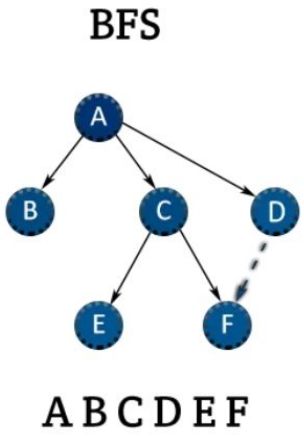
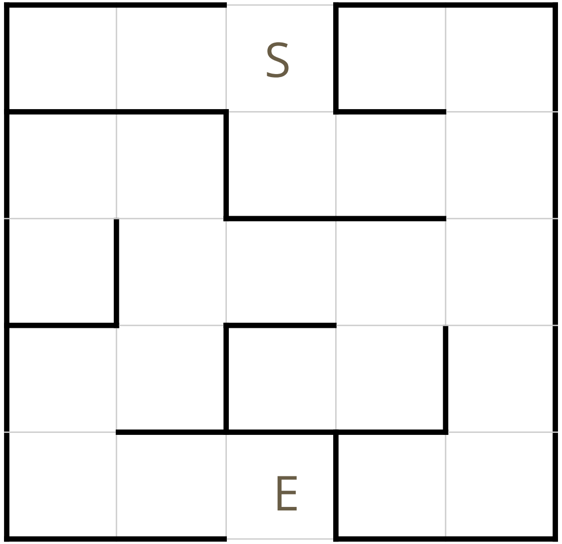


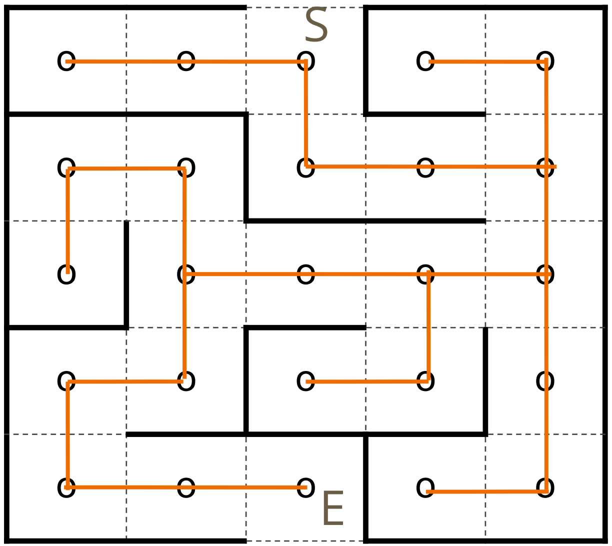
Image credits: [DFS vs BFS Algorithms for Graph Traversal]

Approach 1: Depth First Traversal

- Consider the maze as a grid
- Convert the maze into unit cells and connect the nodes as shown in the figure below



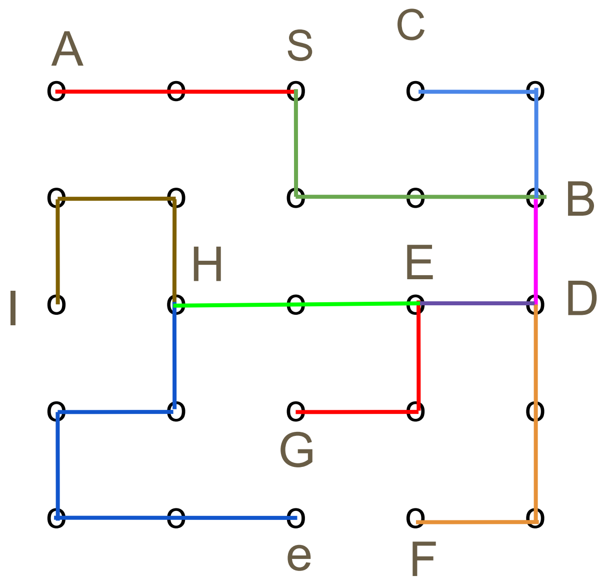
Maze



Maze with unit cells

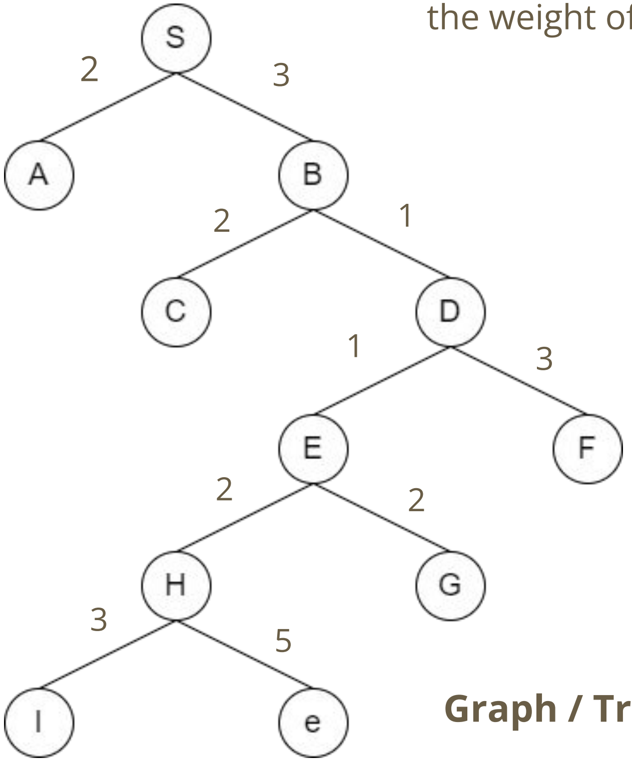
Approach 1: Depth First Traversal

- Each grid cell can be considered as the node of the graph
- Each wall in the maze represents edge of the graph
- Each vertex of the graph is connected to its neighbor
- Label the vertex at every branch point



Edges and Vertices

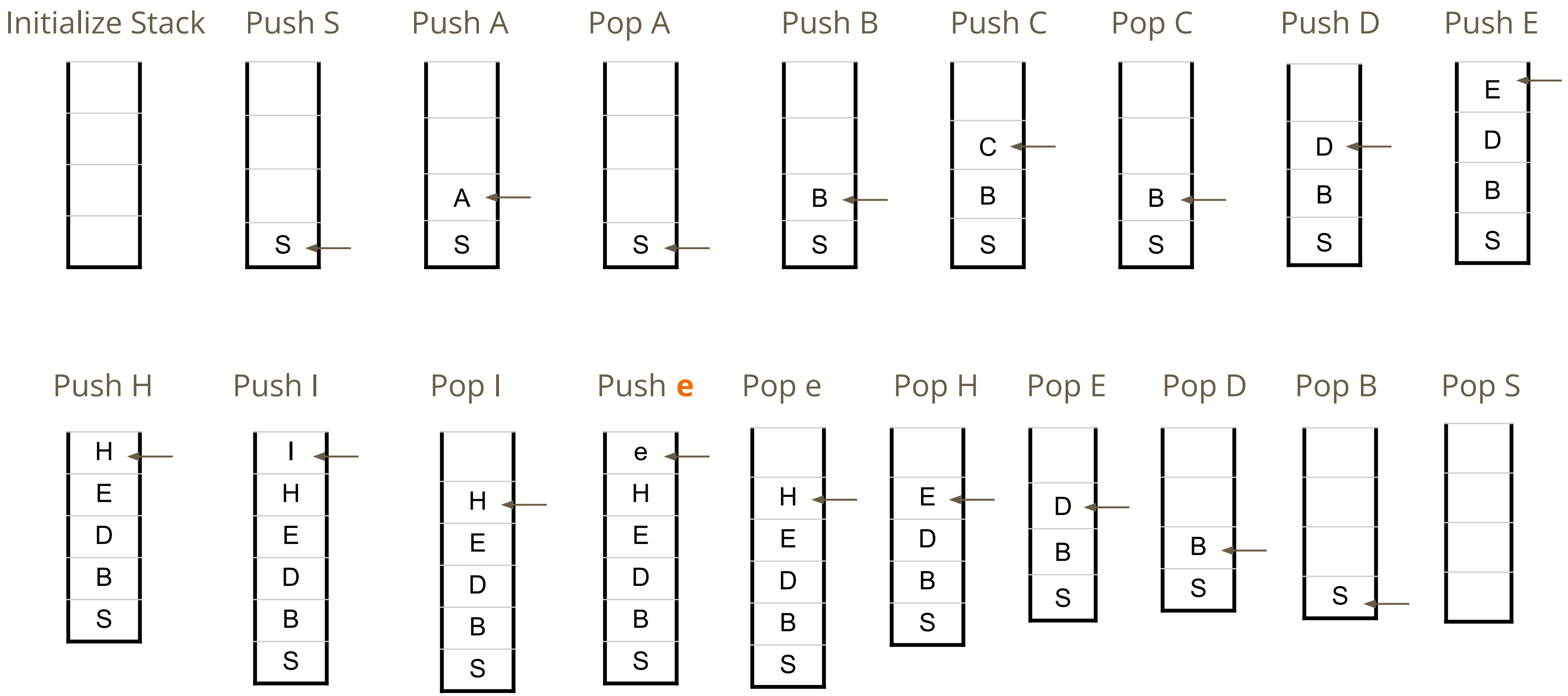
- Add unit cells to calculate the weight of the edge



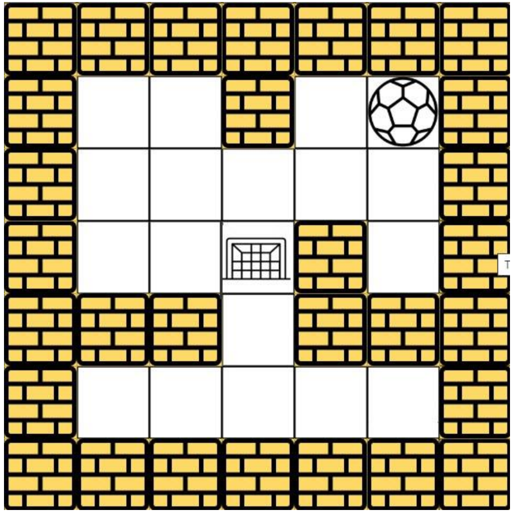
Graph / Tree

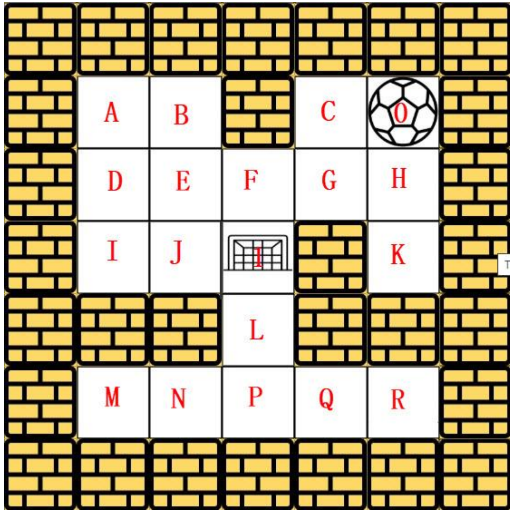
Approach 1: Depth First Traversal Using Legged Robot

- **Find e using Depth First Traversal (Vertical). Use Stack Data Structure**



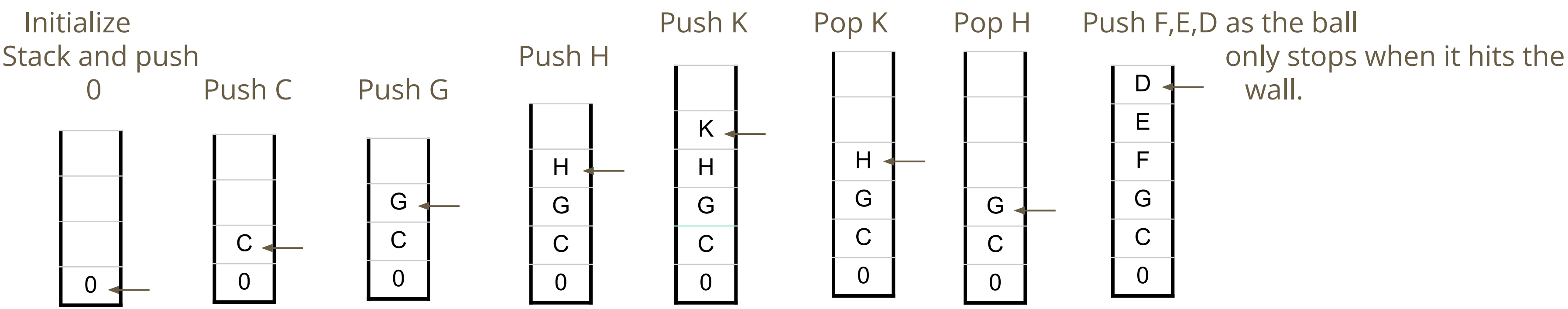
Approach 1: Depth First Traversal Using Wheeled Robot

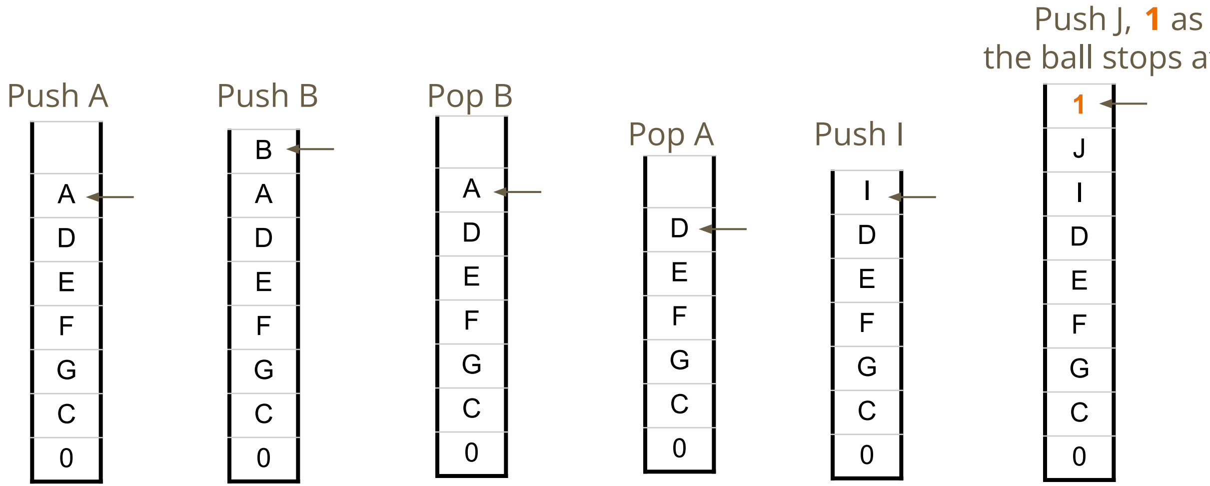




- Will the ball reach the goal when using a Wheeled Robot approach to solve the problem?
- Source = 0 , Destination = 1

Approach 1: Depth First Traversal Using Wheeled Robot





Next , pop element from the stack

- Pop **1**
- Pop J
- Pop I
- Pop D
- Pop E
- Pop F
- Pop G
- Pop C
- Pop 0

Pseudo Code Using DFT

- Initialize an empty stack to keep track of positions to explore for DFT.
- Use a list of directions (Right: (1,0), Left: (-1,0), Up: (0,1), Down: (0,-1)) to iterate through each possible direction.
- Push the starting position onto the stack. Initialize an empty set to keep track of visited positions. Add the starting position to the set of visited positions.
- While the stack is not empty, pop the top position from the stack. Extract the row and column from the position.
- Through each direction, initialize new_row and new_col to the current position.
- If it's possible to move in the current direction without hitting a wall, update new_row and new_col accordingly. Once the loop ends, create a new position (new_pos) with the updated row and column.
- If the new position is the destination, return True.
- If the new position has not been visited, add it to the set of visited positions. Push it onto the stack for further exploration.
- If the loop completes without finding the destination, return False.

Python Code Using DFT

```
from collections import defaultdict
class Solution:
    def hasPath(self, maze: list[list[int]], start: list[int], destination: list[int]) -> bool:
        stack = []
        stack.append(start)
        dirs = [[1,0],[-1,0],[0,1],[0,-1]]
        visited = defaultdict()
        visited[str(start)] = True
        while stack:
            pos = stack.pop()
            row = pos[0]
            col = pos[1]
            for direction in dirs:
                new_row = row
                new_col = col
                while new_row + direction[0] >= 0 and new_col + direction[1] >= 0 and new_row + direction[0] < len(maze) and
new_col + direction[1] < len(maze[0]) and maze[new_row + direction[0]][new_col + direction[1]] != 1:
                    new_row = new_row + direction[0]
                    new_col = new_col + direction[1]

                new_pos = [new_row,new_col]
                if new_pos[0] == destination[0] and new_pos[1] == destination[1]:
                    return True
                if str(new_pos) not in visited:
                    visited[str([new_row, new_col])] = True
                    stack.append(new_pos)
        return False
```

Test Case 1

Input: 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]

Output: true

Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.

```
sol = Solution()
output = sol.hasPath([[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])
#output = sol.hasPath([[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])
#output = sol.hasPath([[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(f"Will the ball reach the destination ?", output)
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_DFT.py"
Will the ball reach the destination ? True
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Test Case 2

Input: 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]

Output: false

Explanation: There is no way for the ball to stop at the destination. Notice that you can pass through the destination but you cannot stop there.

```python
sol = Solution()
#output = sol.hasPath([[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])
output = sol.hasPath([[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])
#output = sol.hasPath([[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(f"Will the ball reach the destination ?", output)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_DFT.py"
Will the ball reach the destination ? False
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Test Case 3

Input: 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]

Output: false

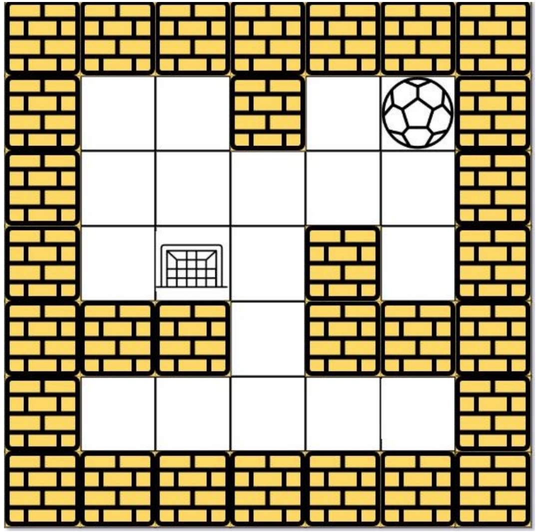
```python
sol = Solution()
#output = sol.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],[0,4],[4,4])
#output = sol.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],[0,4],[3,2])
output = sol.hasPath([[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]],[4,3],[0,1])
print(f"Will the ball reach the destination ?", output)
```

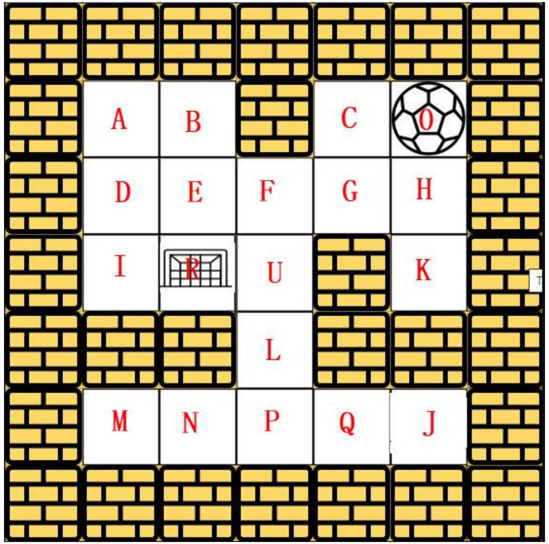
ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_DFT.py"
Will the ball reach the destination ? False
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Approach 2: Breadth First Traversal





- Will the ball reach the goal when using a Wheeled Robot approach to solve the problem?
- Source = 0 , Destination = R

Approach 2: Breadth First Traversal Using Legged Robot

Visited: 0
0
Queue: (empty)
Initialize an queue

Visited: 0
1
Queue: 0
a. Add 0 to Queue
b. Mark 0 as visited

Visited: 0
1
Queue: **Print: 0**
a. Remove 0 from Queue
b. Print 0

Visited: 0 C H
1 1 1
Queue: C H **Print: 0**
a. Add C,H to Queue
b. Mark C, H visited

Visited: 0 C H
1 1 1
Queue: H **Print: 0 C**
a. Remove C from Queue
b. Print C

Visited: 0 C H G
1 1 1 1
Queue: H G **Print: 0 C**
a. Add G to the Queue
b. Mark G as visited

Visited: 0 C H G
1 1 1 1
Queue: G **Print: 0 C H**
a. Remove H from Queue
b. Print H

Visited: 0 C H G H
1 1 1 1 1
Queue: G K **Print: 0 C H**
a. Add K to the Queue
b. Mark K as visited

Visited: 0 C H G H
1 1 1 1 1
Queue: K **Print: 0 C H G**
a. Remove G from Queue
b. Print G

Visited: 0 C H G H F
1 1 1 1 1 1
Queue: K F **Print: 0 C H G**
a. Add F to the Queue
b. Mark F as visited

Approach 2: Breadth First Traversal Using Legged Robot

Visited: 0 C H G H F
1 1 1 1 1 1
Queue: F Print: 0 C H G K

a. Remove K from Queue
b. Print K

Visited: 0 C H G H F
1 1 1 1 1 1
Queue: Print: 0 C H G K F

a. Remove F from Queue
b. Print F

Visited: 0 C H G H F E U
1 1 1 1 1 1 1 1
Queue: E U Print: 0 C H G K F

a. Add E, U to the Queue
b. Mark E, U as visited

Visited: 0 C H G H F E U
1 1 1 1 1 1 1 1
Queue: U Print: 0 C H G K F E

a. Remove E from the Queue
b. Print E

Visited: 0 C H G H F E U D B R
1 1 1 1 1 1 1 1 1 1 1
Queue: U D B R Print: 0 C H G K F E

a. Add D, B, R to the Queue
b. Mark D, B, R as visited

Visited: 0 C H G H F E U D B R
1 1 1 1 1 1 1 1 1 1 1
Queue: D B R Print: 0 C H G K F E U

a. Remove U from the Queue
b. Print U

Visited: 0 C H G H F E U D B R L
1 1 1 1 1 1 1 1 1 1 1 1
Queue: D B R Print: 0 C H G K F E U

a. Add L to the Queue
b. Mark L as visited

Approach 2: Breadth First Traversal Using Legged Robot

Visited: 0 C H G H F E U D B R L
1 1 1 1 1 1 1 1 1 1 1 1
Queue: B R L Print: 0 C H G K F E U D

a. Remove D from the Queue
b. Print D

Visited: 0 C H G H F E U D B R L A I
1 1 1 1 1 1 1 1 1 1 1 1 1 1
Queue: B R L A I Print: 0 C H G K F E U D

a. Add A, I to the Queue
b. Mark A,I as visited

Visited: 0 C H G H F E U D B R L A I
1 1 1 1 1 1 1 1 1 1 1 1 1 1
Queue: R L A I Print: 0 C H G K F E U D B

a. Remove B from the Queue
b. Print B

Visited: 0 C H G H F E U D B R L A I
1 1 1 1 1 1 1 1 1 1 1 1 1 1
Queue: R L A I Print: 0 C H G K F E U D B R

a. Remove R from the Queue
b. Print R

Approach 2: Breadth First Traversal Using Wheeled Robot

Visited: 0
0
Queue: (empty)
Initialize an queue

Visited: 0
1
Queue: 0
a. Add 0 to Queue
b. Mark 0 as visited

Visited: 0
1
Queue: **Print: 0**
a. Remove 0 from Queue
b. Print 0

Visited: 0 C K
1 1 1
Queue: C K **Print: 0**
a. Add C,K to Queue
b. Mark C, K visited

Visited: 0 C K
1 1 1
Queue: K **Print: 0 C**
a. Remove C from Queue
b. Print C

Visited: 0 C K G
1 1 1 1
Queue: K G **Print: 0 C**
a. Add G to the Queue
b. Mark G as visited

Visited: 0 C K G
1 1 1 1 1
Queue: G **Print: 0 C K**
a. Remove K from Queue
b. Print K

Visited: 0 C K G
1 1 1 1
Queue: **Print: 0 C K G**
a. Remove G from Queue
b. Print G

Visited: 0 C K G D
1 1 1 1 1
Queue: D **Print: 0 C K G**
a. Add D to the Queue
b. Mark D visited

Visited: 0 C K G D
1 1 1 1 1
Queue: **Print: 0 C K G D**
a. Remove D from Queue
b. Print D

Approach 2: Breadth First Traversal Using Wheeled Robot

Visited: 0 C K G D A I
1 1 1 1 1 1 1
Queue: A I Print: 0 C K G D

a. Add A, I to the Queue
b. Mark A, I as visited

Visited: 0 C K G D A I
1 1 1 1 1 1 1
Queue: I Print: 0 C K G D A

a. Remove A from Queue
b. Print A

Visited: 0 C K G D A I B
1 1 1 1 1 1 1 1
Queue: I B Print: 0 C K G D A

a. Add B to the Queue
b. Mark B as visited

Visited: 0 C K G D A I B
1 1 1 1 1 1 1 1
Queue: B Print: 0 C K G D A I

a. Remove I from the Queue
b. Print I

Visited: 0 C K G D A I B U
1 1 1 1 1 1 1 1 1
Queue: B U Print: 0 C K G D A I

a. Add U to the queue
b. Mark U as visited

Visited: 0 C K G D A I B U
1 1 1 1 1 1 1 1 1
Queue: U Print: 0 C K G D A I B

a. Remove B from the queue
b. Print B

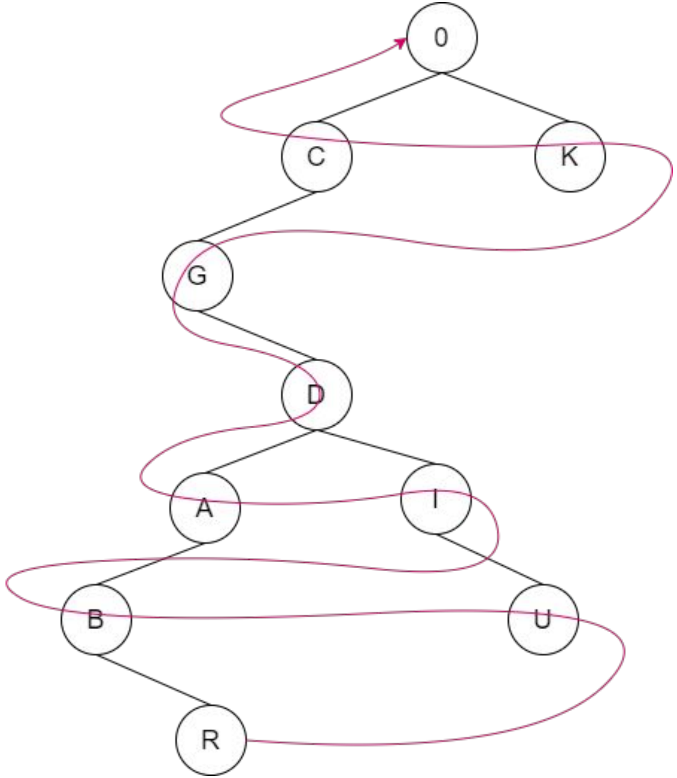
Visited: 0 C K G D A I B U R
1 1 1 1 1 1 1 1 1 1
Queue: U R Print: 0 C K G D A I B

a. Add **R** to the queue
b. Mark R as visited

Visited: 0 C K G D A I B U R
1 1 1 1 1 1 1 1 1 1
Queue: Print: 0 C K G D A I B U R

a. Remove U , Print U
b. Next Remove R and Print R

Approach 2: Breadth First Traversal Using Wheeled Robot



- **This is the part of the full tree which represents the path from source 0 to destination R**

Pseudo Code for BFT

- Create a queue to perform BFS traversal.
- Use a list of directions (Right: (1,0), Left: (-1,0), Up: (0,1), Down: (0,-1)) to iterate through each possible direction.
- Maintain a dictionary called visited to keep track of visited positions.
- If the queue is not empty, dequeue first position and explore all possible directions from that dequeued position.
- Move in each direction until ball hits a wall.
- If the new position after movement is the destination, return True.
- Otherwise, if the new position hasn't been visited yet, mark it as visited and enqueue it.
- If no path is found after exploring all possible paths, return False.

Python Code Using BFT

```
from collections import deque, defaultdict

class Solution:
    def hasPath(self, maze: list[list[int]], start: list[int], destination: list[int]) -> bool:
        queue = deque()
        queue.append(tuple(start))
        dirs = [[1, 0], [-1, 0], [0, 1], [0, -1]]
        visited = defaultdict(bool)

        while queue:
            row, col = queue.popleft()
            for direction in dirs:
                new_row, new_col = row, col
                while 0 <= new_row + direction[0] < len(maze) and 0 <= new_col + direction[1] < len(maze[0]) and maze[new_row +
direction[0]][new_col + direction[1]] != 1:
                    new_row += direction[0]
                    new_col += direction[1]

                new_pos = (new_row, new_col)
                if new_pos == tuple(destination):
                    return True
                if not visited[new_pos]:
                    visited[new_pos] = True
                    queue.append(new_pos)

        return False
```

Test Case 1

Input: 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]

Output: true

Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.

```python
sol = Solution()
output = sol.hasPath([[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])
#output = sol.hasPath([[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])
#output = sol.hasPath([[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(f"Will the ball reach the destination (BFS) ?", output)
```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_BFT.py"
Will the ball reach the destination (BFS) ? True
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Test Case 2

Input: 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]

Output: false

Explanation: There is no way for the ball to stop at the destination. Notice that you can pass through the destination but you cannot stop there.

```python
sol = Solution()
#output = sol.hasPath([[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])
output = sol.hasPath([[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])
#output = sol.hasPath([[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(f"Will the ball reach the destination (BFS) ?", output)
```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_BFT.py"
Will the ball reach the destination (BFS) ? False
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Test Case 3

Input: 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]

Output: false

```python
sol = Solution()
#output = sol.hasPath([[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])
#output = sol.hasPath([[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])
output = sol.hasPath([[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(f"Will the ball reach the destination (BFS) ?", output)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

TERMINAL

```
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> cd "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project"
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project> & C:/Users/kavis/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/kavis/OneDrive/Documents/Trimester 2 Jan 24 - April 24/Practical Applications of Algorithms/Project/490_Maze_BFT.py"
Will the ball reach the destination (BFS) ? False
PS C:\Users\kavis\OneDrive\Documents\Trimester 2 Jan 24 - April 24\Practical Applications of Algorithms\Project>
```

Next steps...

- If the maze is very large, **DFS** is indeed preferred over BFS. Based on the size of the maze, either of the algorithms will be preferred.
- If the problem is modified to find the shortest path, only **BFS** gives the right answer.
- Other algorithms like A* and Dijkstra can be explored.

Summary

- Both **BFS** and *DFS* have same big O time complexity (***O(N*M)***) (N, M are rows and columns in the maze)
- **BFS** will generally use **more memory** since it keeps track of all the paths from the visited nodes, while *DFS* keeps track of *only one path* at a time.
- **BFS** will give the **shortest path** while *DFS* gives *a path* (not necessarily shortest one).

References

- [Difference between BFS and DFS - GeeksforGeeks]
- [Wheels Are Better Than Feet for Legged Robots - IEEE Spectrum]
- [Leet Code 490. The Maze — Explained Python3 Solution | by Edward Zhou | Tech Life & Fun | Medium]
-