

# Practical Application of Algorithms

## Project: The Maze

Course: CS501

Professor: Dr. Henry Chang

Presented by: Karan Shrestha (20087)

# Abstract



In the realm of maze exploration, traversing from a starting point to a destination presents an intriguing challenge. Given a maze represented as a 2D array where 0s represent open paths and 1s represent walls, the task is to determine whether there exists a path from a designated starting point to a specified destination point.

We approached this problem utilizing both Breadth First Search (BFS) and Depth First Search (DFS) algorithms. These classic graph traversal techniques provide efficient ways to explore the maze's pathways.

**BFS Solution:** We employed a queue-based approach, systematically exploring possible paths from the starting point until reaching the destination or exhausting all options.  
**DFS Solution:** Alternatively, we utilized a recursive approach, delving deeply into each potential pathway, backtracking when necessary, until either reaching the destination or exploring all possibilities.

By applying BFS and DFS algorithms, we effectively tackled the maze exploration problem. These solutions offer versatile approaches to navigating mazes, demonstrating the power of graph traversal techniques in solving real-world challenges

# Table of Content



1. Introduction
2. Algorithm
3. Maze problem using DFT
4. Maze problem using BST
5. Code
6. Testing
7. Conclusion

# Introduction



## Tree Traversal

1-BFT(Breadth- First Traversal)

2-DFT(Depth- First Traversal):

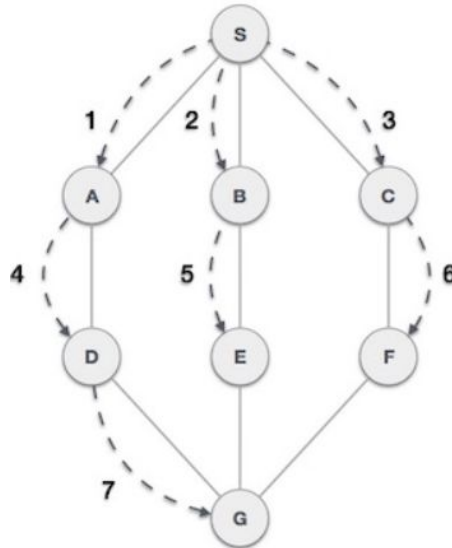
### Outline of Presentation:

- We will explain BFT and DFT traversal.
- In this project we are demonstrating two tree traversal algorithm.
- We will solve the problem with application example of Maze.
- Maze problem
- Solution 1: Solve using DFT traversal.
- Solution 2: Solve using BFT traversal.

# Algorithm

## 1-BFT(Breadth- First Traversal)

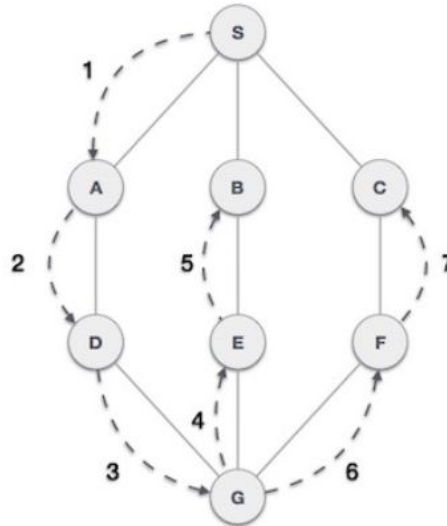
- Breadth-first traversal starts at a given vertex and explores all its neighbors before moving on to the neighbors of those neighbors.
- BFT visits vertices in layers, exploring all vertices at a given depth from the starting vertex before moving on to the next level.
- BFT can be implemented using a queue data structure.



# Algorithm

## 2-DFT(Depth- First Traversal)

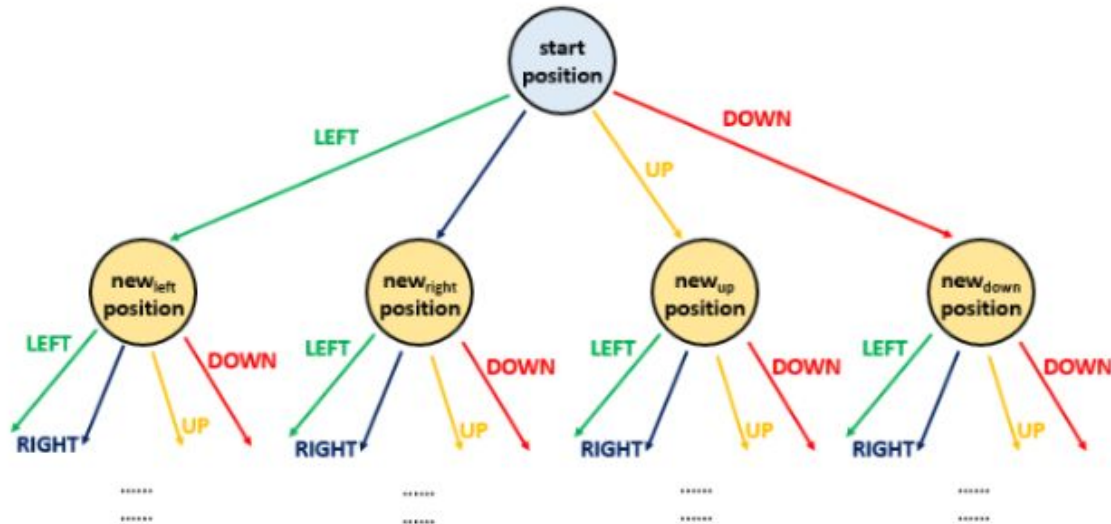
- Depth-first traversal starts at a given vertex and explores as far as possible along each branch before backtracking.
- DFT goes deep into the graph along a single path, visiting all the descendants of a vertex before returning to explore other branches.
- DFT can be implemented using stack data structure.



# The Maze Problem Tree Traversal

The ball can move **Right, Left, Up, Down**

So, we can draw tree as below for more clear explanation.

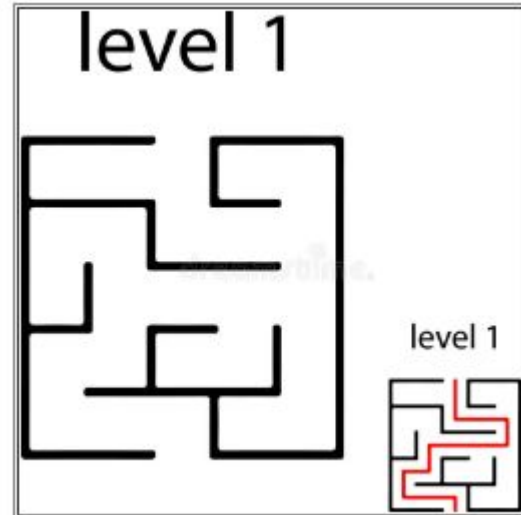


# The Maze Problem 1- Using DFT

## Approach 1: Without Wheel(Legged Robot)

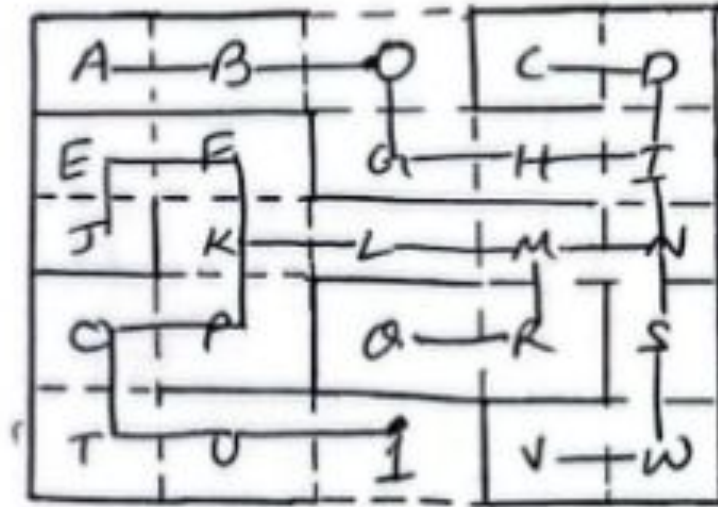
Clear Route (Street, Highway)

Robot Can move 1 step at a time.

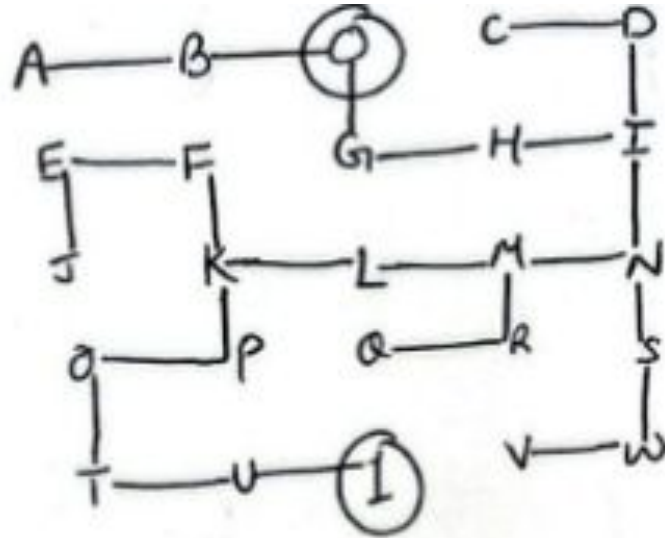




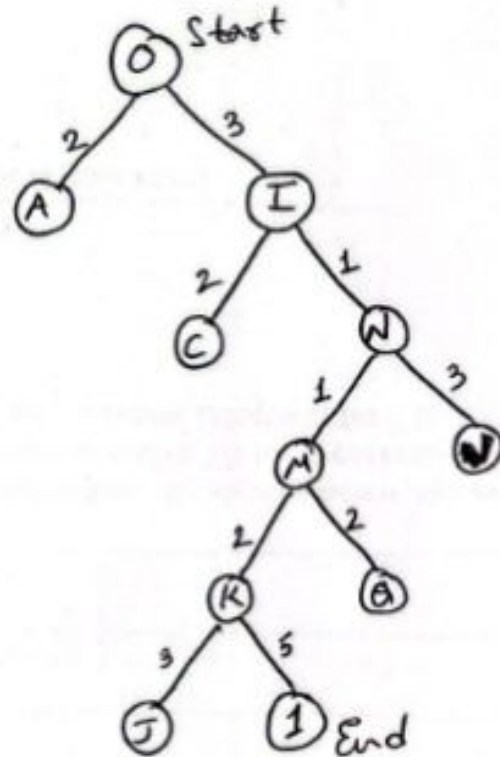
Assigning each block with Node name as below:



## Joining the Node Paths:



## Tree Diagram:





# Stack

Starting From Node 0: Stack Diagram(One step at a time)

									C
								D	D
							I	I	I
		A				H	H	H	H
	B	B	B		G	G	G	G	G
0	0	0	0	0	0	0	0	0	0
Push 0	Push B	Push A	Pop A	Pop B	Push G	Push H	Push I	Push D	Push C



# Stack

Stacking continue...

								J
							E	E
						F	F	F
					K	K	K	K
				L	L	L	L	L
			M	M	M	M	M	M
D		N	N	N	N	N	N	N
I	I	I	I	I	I	I	I	I
H	H	H	H	H	H	H	H	H
G	G	G	G	G	G	G	G	G
O	O	O	O	O	O	O	O	O
Pop C	Pop D	Push N	Push M	Push L	Push K	Push F	Push E	Push J



# Stack

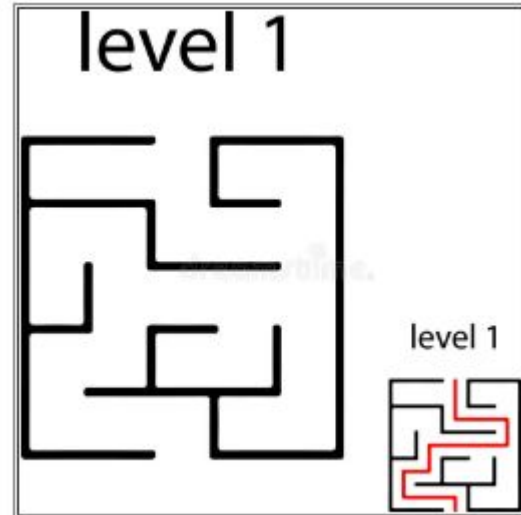
Stacking continue...

								1
							U	U
						T	T	T
E				Q	Q	Q	Q	Q
F	F		P	P	P	P	P	P
K	K	K	K	K	K	K	K	K
L	L	L	L	L	L	L	L	L
M	M	M	M	M	M	M	M	M
N	N	N	N	N	N	N	N	N
I	I	I	I	I	I	I	I	I
H	H	H	H	H	H	H	H	H
G	G	G	G	G	G	G	G	G
O	O	O	O	O	O	O	O	O
Pop J	Pop E	Pop F	Push P	Push Q	Push T	Push U	Push 1	

# The Maze Problem 1- Using DFT

## Approach 2: With Wheel (Self-driving Car)

Assuming the robot can go through the empty spaces by rolling.





# Stack

Starting From Node 0: Stack Diagram (Robot will roll till the wall end)

					V
					W
			C		S
			D		N
		I	I	I	I
A		H	H	H	H
B		G	G	G	G
0	0	0	0	0	0
Push (0,B,A)	Pop (A,B)	Push (G,H,I)	Push (D,C)	Pop (C,D)	Push (N,S,W,V)



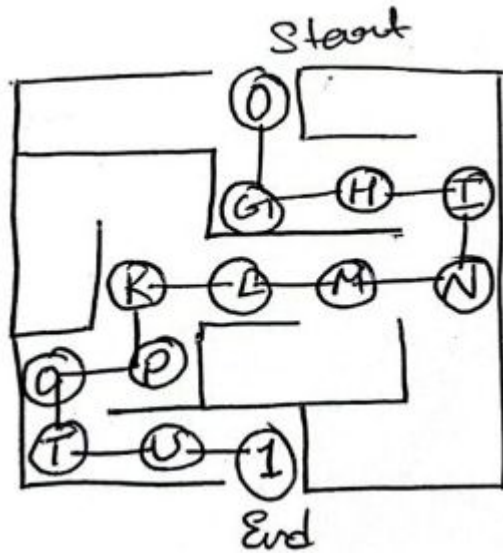


# Stack

Stacking continue...

				1
				U
		J		T
		E		O
		F		P
	K	K	K	K
	L	L	L	L
	M	M	M	M
N	N	N	N	N
I	I	I	I	I
H	H	H	H	H
G	G	G	G	G
O	O	O	O	O
Pop (V,W,S)	Push (M,L,K)	Push(F,E,J)	Pop (J,E,F)	Push (P,O,T,U,1)

# Path

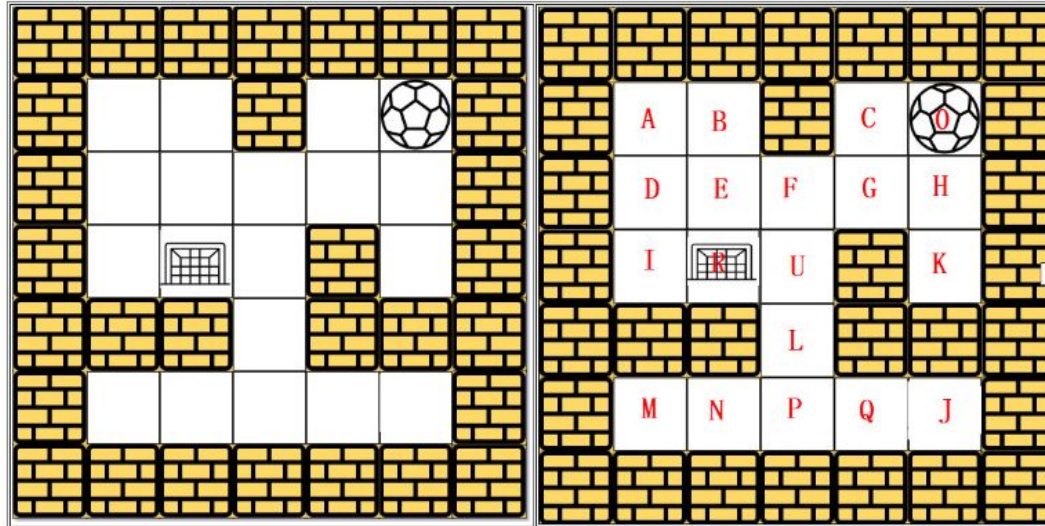


1
U
T
O
P
K
L
M
N
I
H
G
0
Push (P,O,T,U,1)

# The Maze Problem- 2

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  $m \times n$  maze, the ball's start position and the destination, where start = [startrow startcol] and destination - [destination row, destination col], return true if the ball can stop at the destination, otherwise return false.



## Starting BFT- With Queue:

The ball can go through the empty spaces by rolling right, left, up, down, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

[illegible]

## Queue Continue...

[illegible][illegible]

## Solution 2: Using BFT(Breadth-First Traversal)

[illegible]

# Solution: Using BFT(Breadth-First Traversal)


Queue Continue...

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

Now we found the Target Node(R) as a **Visited**:

So we Can return output: **True** (Ball can stop at the de

# Solution: Using BFT(Breadth-First Traversal)



```
from collections import deque
def hasPath(maze, start, destination):
    if not maze or not maze[0]:
        return False
    m, n = len(maze), len(maze[0])
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    def isValid(x, y):
        return 0 <= x < m and 0 <= y < n and maze[x][y] == 0
    queue = deque([start])
    visited = set()
    while queue:
        x, y = queue.popleft()
        if (x, y) == tuple(destination):
            return True
        if (x, y) in visited:
            continue
        visited.add((x, y))
        for dx, dy in directions:
            nx, ny = x, y
            while isValid(nx + dx, ny + dy):
                nx += dx
                ny += dy
            if (nx, ny) not in visited:
                queue.append((nx, ny))
    return False
```



# Solution : Using BFT(Breadth-First Traversal)

Code:

```
from collections import deque
def hasPath(maze, start, destination):
    if not maze or not maze[0]:
        return False
    m, n = len(maze), len(maze[0])
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    def isValid(x, y):
        return 0 <= x < m and 0 <= y < n and maze[x][y] == 0
    queue = deque([start])
    visited = set()
    while queue:
        x, y = queue.popleft()
        if (x, y) == tuple(destination):
            return True
        if (x, y) in visited:
            continue
        visited.add((x, y))
        for dx, dy in directions:
            nx, ny = x, y
            while isValid(nx + dx, ny + dy):
                nx += dx
                ny += dy
            if (nx, ny) not in visited:
                queue.append((nx, ny))
    return False
```

# Solution: Using BFT(Breadth-First Traversal)

Test Case:

```
maze1 = [  
    [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]  
]  
  
start1 = [0, 4]  
destination1 = [4, 4]  
print("Test Case 1:", hasPath(maze1, start1, destination1)) # Output: True  
  
maze2 = [  
    [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]  
]  
  
start2 = [0, 4]  
destination2 = [3, 2]  
print("Test Case 2:", hasPath(maze2, start2, destination2)) # Output: False
```

```
Test Case 1: True  
Test Case 2: False  
PS D:\SFBU\Projects>
```



# Conclusion

## Complexity Analysis (DFS):

Time Complexity:  $O(mn)$ . Complete traversal of maze will be done in the worst case. Here,  $m$  and  $n$  refers to the number of rows and columns of the maze.

Space Complexity:  $O(mn)$ . Visited array of size  $m*n$  is used.

## Complexity Analysis (BFS):

Time Complexity:  $O(mn)$ . Complete traversal of maze will be done in the worst case. Here,  $m$  and  $n$  refers to the number of rows and columns of the maze.

Space Complexity:  $O(mn)$ . Visited array of size  $m*n$  is used and queue size can grow upto  $m * n$  in worst case.

Depth-First Traversal - It does not find the Shortest Path

Breadth-First Traversal - It does find the Shortest Path