**Practical Application of Algorithms**

**Project: The Maze**

**Course: CS500**

**Professor: Dr. Henry Chang**

**Presented by: Karan Shrestha (20087)**

**Tree Traversal**

**1-BFT(Breadth- First Traversal)**

**2-DFT(Depth- First Traversal):**

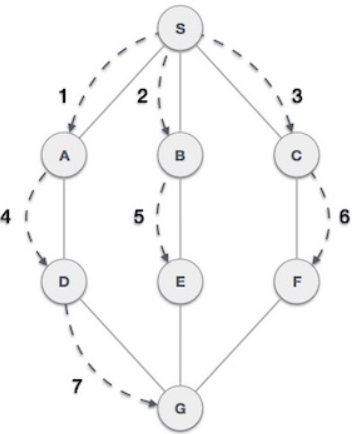
**Outline of Presentation:**

* 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 BFS traversal.

**Algorithm**

**1-BFT(Breadth- First Traversal)**

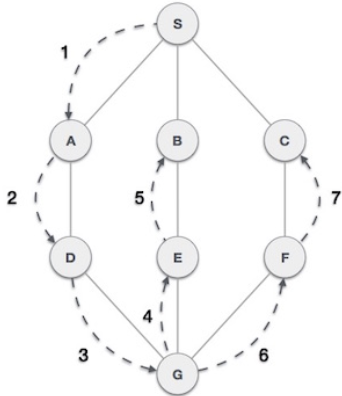
* Breadth-first traversal starts at a given vertex and explores all its neighbors before moving on to the neighbors of those neighbors.
* BFT visits vertices in layers, exploring all vertices at a given depth from the starting vertex before moving on to the next level.
* BFT can be implemented using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



**Algorithm**

**2-DFT(Depth- First Traversal)**

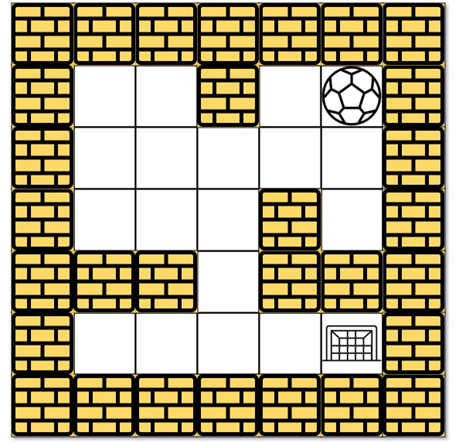
* Depth-first traversal starts at a given vertex and explores as far as possible along each branch before backtracking.
* DFT goes deep into the graph along a single path, visiting all the descendants of a vertex before returning to explore other branches.
* DFT can be implemented using stack data structure.



**The Maze Problem**

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.

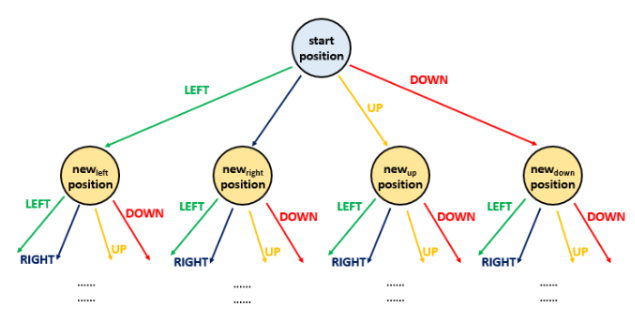
Given the m x n maze, the ball's start position and the destination, where start = [startrow startcol] and destination - [destination row, destination col], return true if the ball can stop at the destination, otherwise return false.



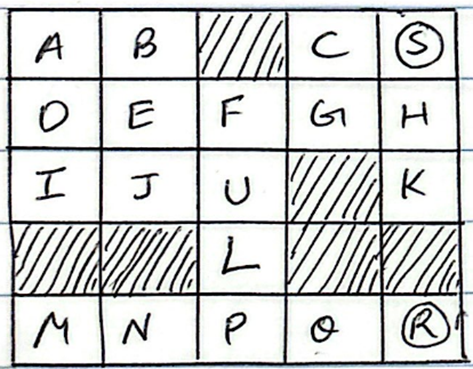
**The Maze Problem**

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

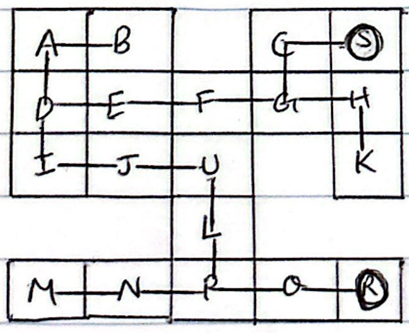
**So, we can draw tree as below for more clear explanation.**



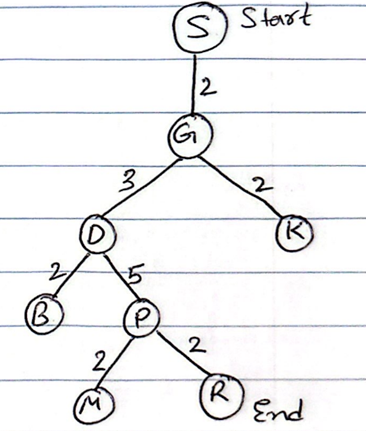
**Assigning each block with Node name as below:**



**Joining the Node Paths:**

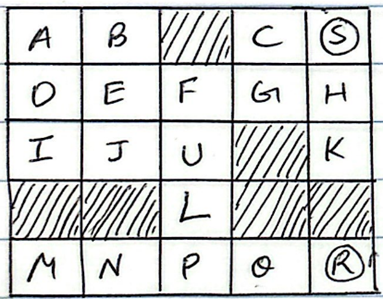


**Tree Diagram:**



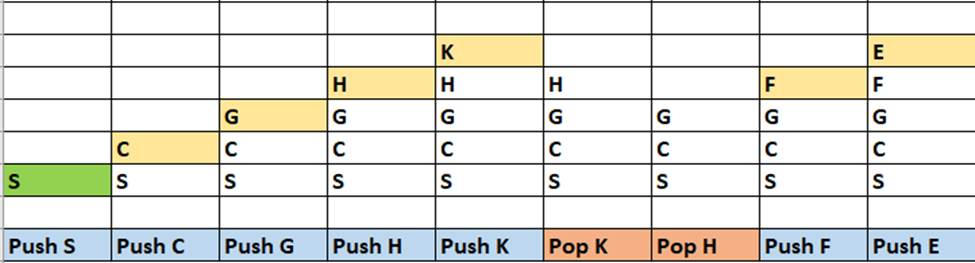
**Solution 1: Using DFS(Depth First Traversal)**

1. Legged robots moves in : DFS
2. The ball can only move one cell at a time.
3. Depth-First Traversal - Right, Left, Up, Down
4. Starting Node is S
5. Ending Node is R
6. Starting Stack one by one step:

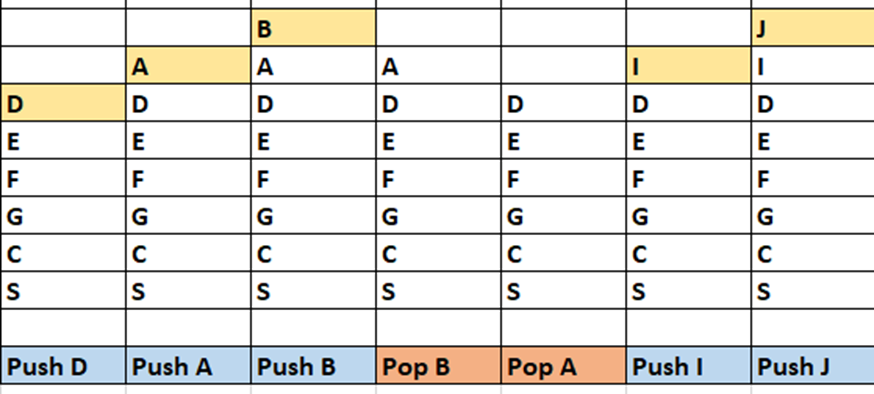


**Solution 1: Using DFS(Depth First Traversal)**

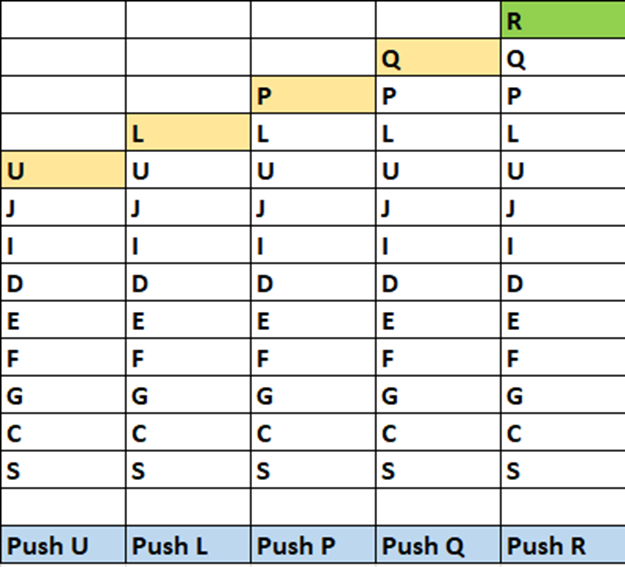
Starting From Node S: Stack Diagram



Stack diagram: Continue….



Stack diagram: Continue….

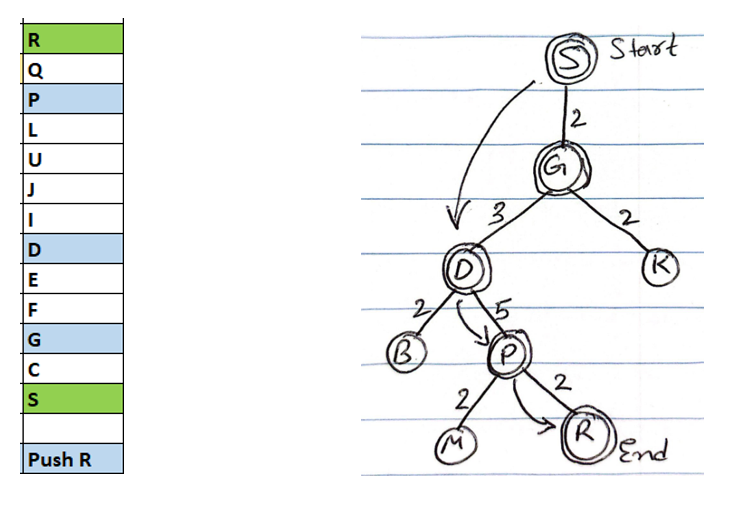


**Solution 1: Using DFS(Depth First Traversal)**

Showing Final Stack from Starting to Ending Path and Tree.

Now we found the path to Target Node(R):

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



**Solution 1: Using DFS(Depth First Traversal)**

**def hasPathDFS(maze, start, destination):**

**if not maze or not maze[0]:**

**return False**

**m, n = len(maze), len(maze[0])**

**visited = set()**

**def dfs(x, y):**

**if (x, y) in visited:**

**return False**

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

**if [x, y] == destination:**

**return True**

**directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]**

**for dx, dy in directions:**

**nx, ny = x + dx, y + dy**

**while 0 <= nx < m and 0 <= ny < n and maze[nx][ny] == 0:**

**nx += dx**

**ny += dy**

**nx -= dx**

**ny -= dy**

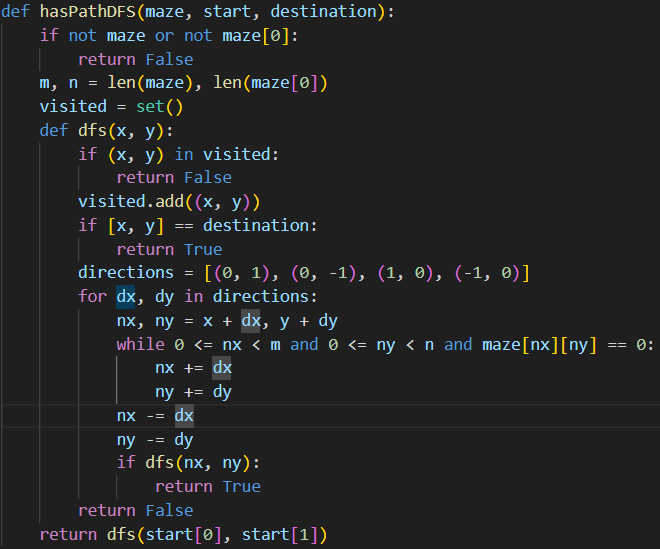
**if dfs(nx, ny):**

**return True**

**return False**

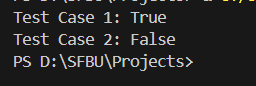
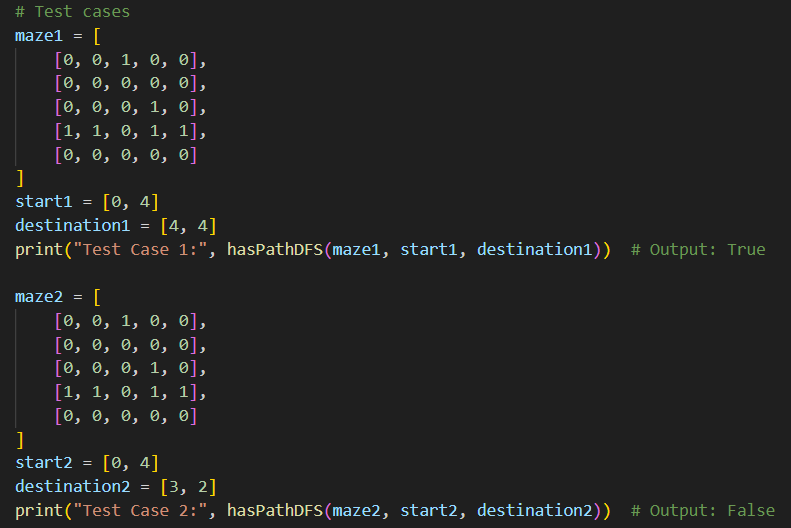
**return dfs(start[0], start[1])**

**Code:**



**Solution 1: Using DFS(Depth First Traversal)**

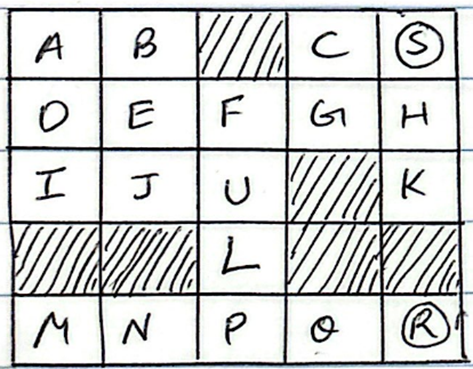
**Testing:**



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

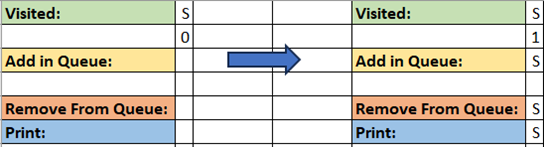
Breadth-First Traversal (**Wheeled robots approach**)

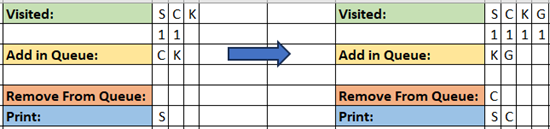
The ball(Node S) can go through the empty spaces by rolling right, left, up, down, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.



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

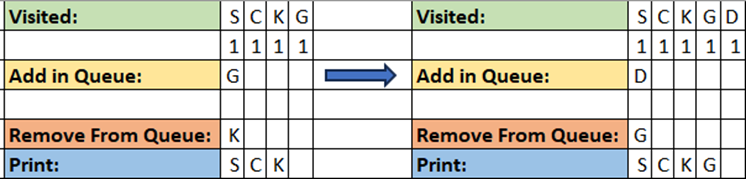
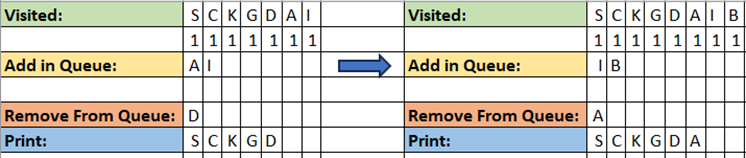
Starting BFT- With **Queue**:





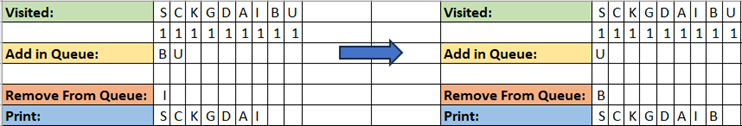
**Solution 2: Using BFT(Breadth-First Traversal)**

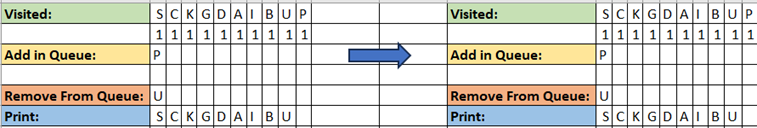
**Queue** Continue…

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

**Queue** Continue…



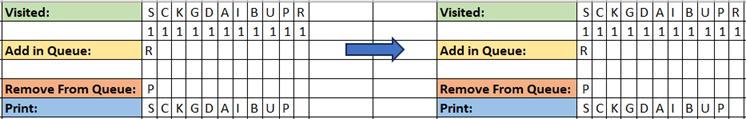


**Solution 2: Using BFT(Breadth-First Traversal)**

**Queue** Continue…

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

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



**Solution 2: 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**

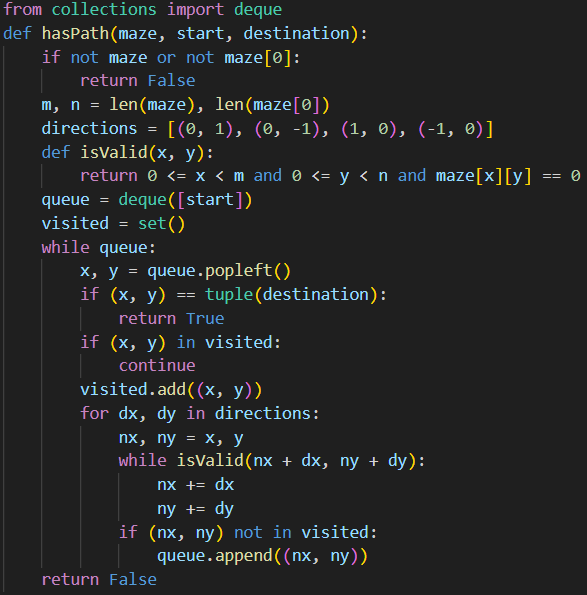
**ny += dy**

**if (nx, ny) not in visited:**

**queue.append((nx, ny))**

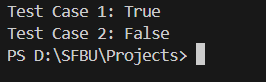
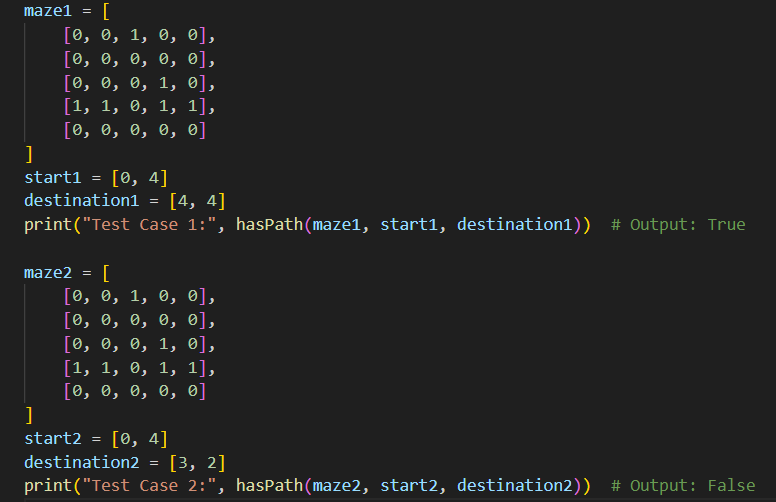
**return False**

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



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

**Test Case:**



**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**