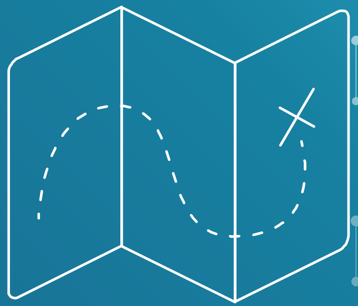
“ The Maze" - LC - Breadth-First Traversal and Depth-First Traversal

Contents



Introduction

Breadth-First Traversal

- BFS systematically explores and visits all nodes in a graph, starting from a chosen node and traversing neighboring nodes level by level.
- Maze Navigation: In mazes, BFS efficiently explores and navigates through passages and corridors, ensuring comprehensive coverage of the maze's layout.
- Shortest Path Discovery: BFS excels at finding the shortest path from the maze entrance to the exit by prioritizing nodes based on their distance from the starting point, leveraging a queue-based approach for efficient traversal.

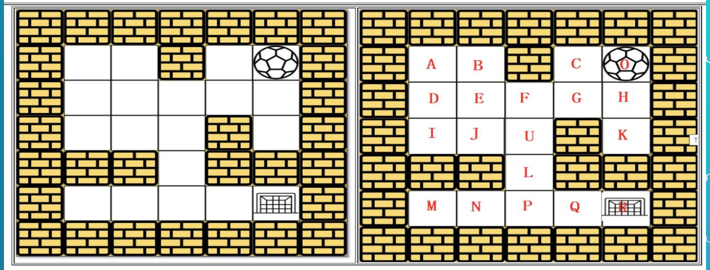
Depth-First Traversal

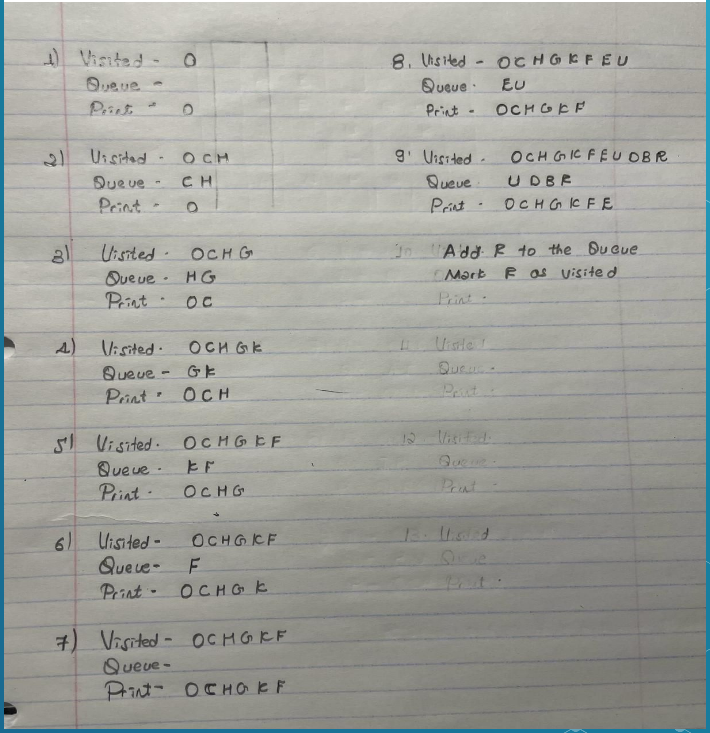
- Depth First Search (DFS) is a core algorithm for systematically exploring and visiting all nodes within a graph, utilizing a depth-first exploration technique.
- DFS traverses deeply into the graph structure, exploring as far as possible along each branch before backtracking, making it well-suited for maze-solving and similar scenarios.
- While DFS may not prioritize finding the shortest path, its exhaustive search approach often reveals alternative routes and potential dead-ends within the maze, offering valuable insights in maze-solving strategies.

Design

Breadth-First Traversal-Without Wheel (Legged Robot)

- Right, Left, Up , bottom
- The ball can only move one cell at a time.





1) Visited - O
Queue -
Print - O

2) Visited - O C H
Queue - C H
Print - O

3) Visited - O C H G
Queue - H G
Print - O C

4) Visited - O C H G K
Queue - G K
Print - O C H

5) Visited - O C H G K F
Queue - K F
Print - O C H G

6) Visited - O C H G K F
Queue - F
Print - O C H G K

7) Visited - O C H G K F
Queue -
Print - O C H G K F

8. Visited - O C H G K F E U
Queue - E U
Print - O C H G K F

9. Visited - O C H G K F E U D B R
Queue - U D B R
Print - O C H G K F E

10 Add R to the Queue
Mark R as visited
Print -

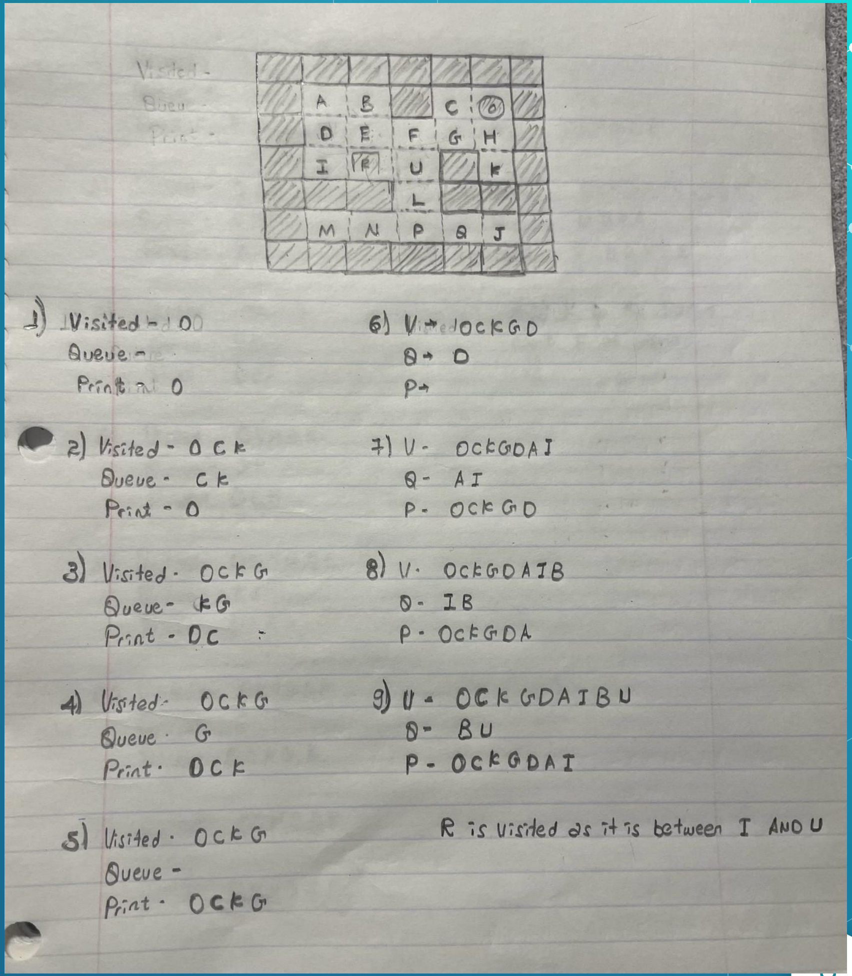
11 Visited
Queue -
Print -

12 Visited
Queue -
Print -

13 Visited
Queue
Print -

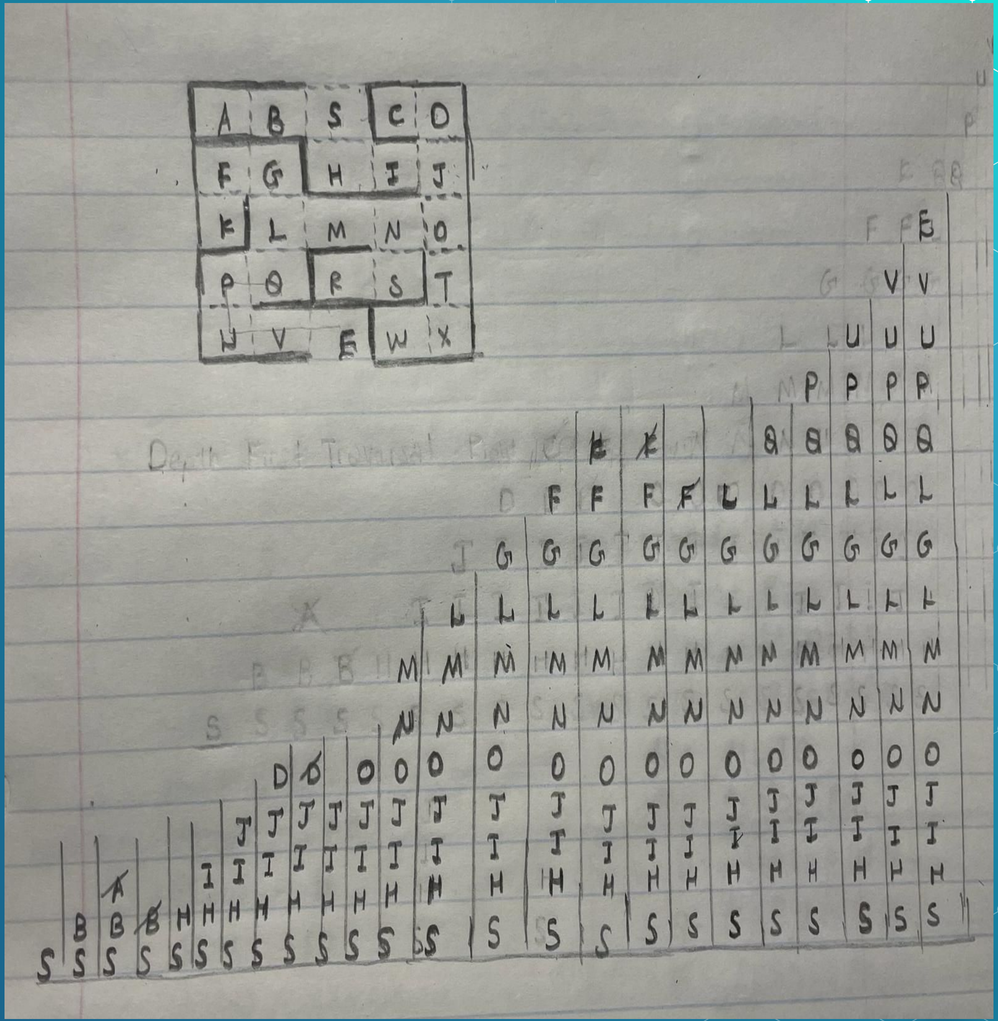
Breadth-First Traversal-With Wheel (Self-driving Car)

- Right, Left, Up , bottom
- 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.



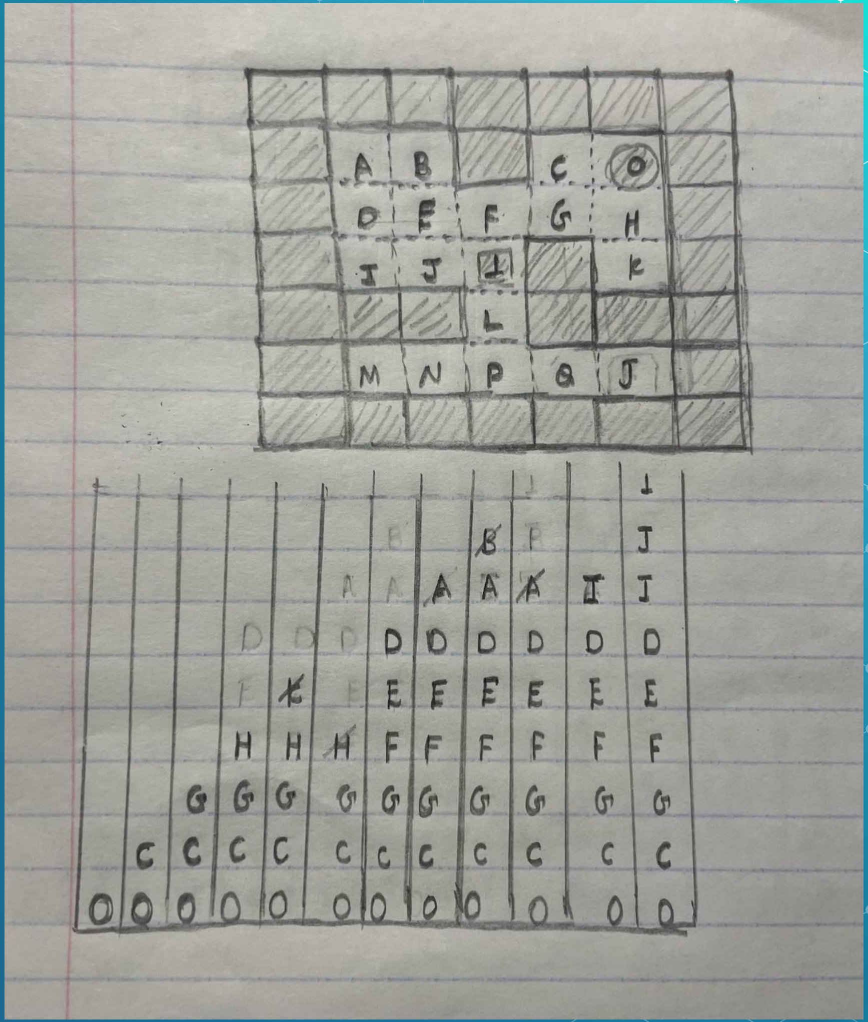
Depth-First Traversal-Without Wheel (Legged Robot)

- Right, Left, Up , bottom
- The ball can only move one cell at a time.



Depth-First Traversal- With Wheel (Self-driving Car)

- Right, Left, Up , bottom
- 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.



Implementation and Test

Breadth-First Traversal

```python
from collections import deque
class Solution:
    def hasPath(self, maze, start, destination):
        m, n = len(maze), len(maze[0])
        visited = set()
        queue = deque([(start[0], start[1])])

        while queue:
            x, y = queue.popleft()
            if (x, y) == (destination[0], destination[1]):
                return True

            if (x, y) in visited:
                continue
            visited.add((x, y))

            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                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 (nx, ny) not in visited:
                    queue.append((nx, ny))

        return False
```

Click to add a breakpoint.

```python
    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]
    sol = Solution()
    print(sol.hasPath(maze, start, destination))

    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]
    sol = Solution()
    print(sol.hasPath(maze, start, destination))

    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]
    sol = Solution()
    print(sol.hasPath(maze, start, destination))
```

PROBLEMS OUTPUT TERMINAL

TERMINAL

```
→ Project_alg cd /Users/belsabel/Desktop/Project_alg
/usr/local/bin/python3 /Users/belsabel/Desktop/Project_alg/BFS.py
→ Project_alg /usr/local/bin/python3 /Users/belsabel/Desktop/Project_alg/BFS.py
True
False
False
```

Depth-First Traversal

```python
class Solution:
    def hasPath(self, maze, start, destination):
        def dfs(x, y):
            if [x, y] == destination:
                return True
            maze[x][y] = -1
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nx, ny = x, y
                while 0 <= nx + dx < len(maze) and 0 <= ny + dy < len(maze[0]) and maze[nx + dx][ny + dy] != 1:
                    nx += dx
                    ny += dy
                if maze[nx][ny] != -1 and dfs(nx, ny):
                    return True
            return False

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

```python
# Test the function
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(Solution().hasPath(maze, start, destination))

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]
print(Solution().hasPath(maze, start, destination))

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]
print(Solution().hasPath(maze, start, destination))
```

PROBLEMS OUTPUT TERMINAL

TERMINAL

```
False
➜  Project_alg cd /Users/belsabel/Desktop/Project_alg
➜  Project_alg /usr/local/bin/python3 /Users/belsabel/Desktop/Project_alg/DFS.py
True
False
False
```

Enhancement Ideas

- **User Interaction**: Provide user-friendly interfaces to input maze configurations or interactively solve mazes using mouse clicks or keyboard inputs.
- **Performance Metrics:** Display metrics like the number of steps taken, time elapsed, and path length to help users understand the efficiency of the solution.
- **Interactive Features:** Add interactive elements to the maze-solving application, such as the ability to pause, resume, or step through the solving process step by step. This allows users to observe how BFS explores the maze in detail.
- **Integration with Games or Simulations:** Integrate the maze-solving application with games or simulations, where players can interact with the maze environment, solve puzzles, or compete against each other using BFS ,DFS and other algorithms.

Conclusion

- Manual calculation of maze solutions using BFS and DFS provided insight into algorithmic behavior and traversal strategies.
- Comparing manually derived solutions with Python implementations offered a practical understanding of algorithmic efficiency and accuracy.
- Implementing BFS and DFS solutions in Python showcased proficiency in algorithmic translation to code and facilitated validation of manual calculations.

References

- Graph Traversals - BFS & DFS -Breadth First Search and Depth First Search
- Breadth First Search (BFS): Visualized and Explained
- Search A Maze For Any Path - Depth First Search Fundamentals