Project: "490. The Maze" - LC - Breadth-First Traversal

NAME: FEVEN BELAY ARAYA
ID: 20027

TABLE OF CONTENT

1. INTRODUCTION

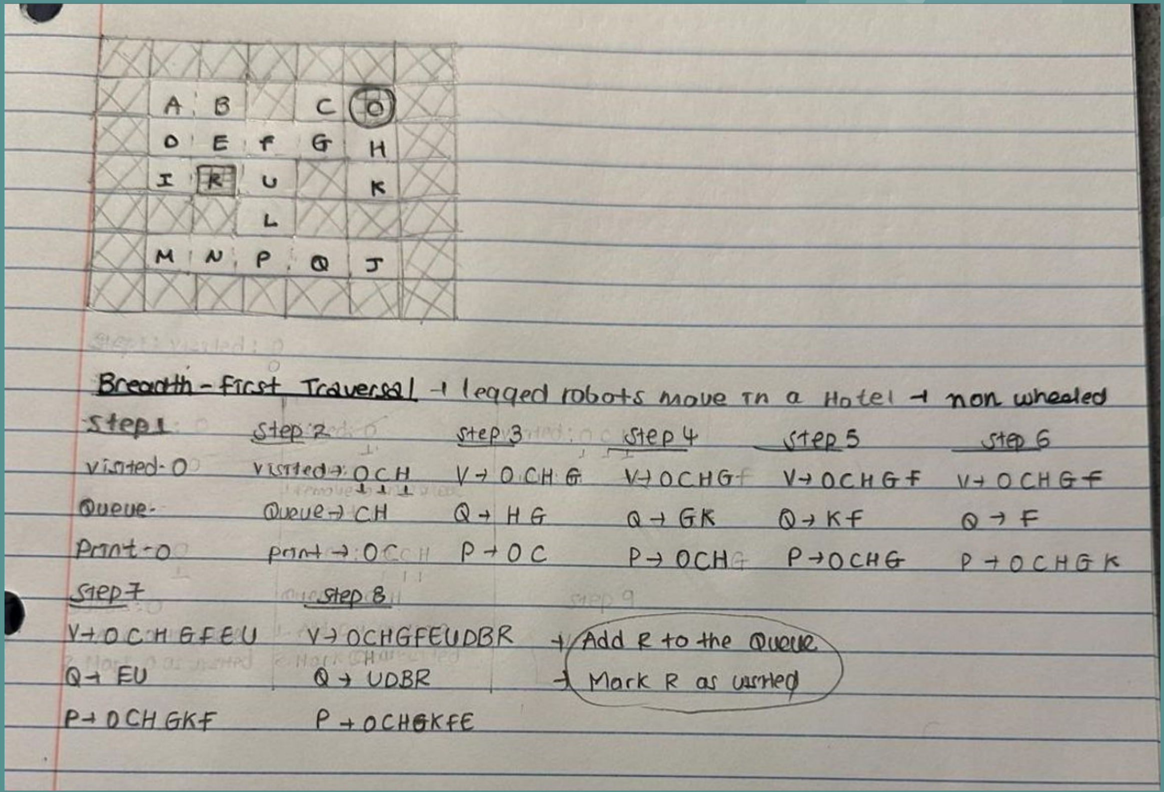
•Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures.

•It starts at a selected node (often at the root in the case of trees, or some arbitrary node for graphs).

•It explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

2. DESIGN

2.1. Without Wheel (Legged Robot)



2.2. With Wheel (Self-driving Car)

BFT → wheeled

1) Visited → O
Queue -
Print - O

2. visited - O C K
Queue - C K
Print - O

3. visited - O C K G
Queue - K G
Print - O C

4. visited - OCKG
Queue - G
Print - OCK

5. V → OCKG
Q →
P → OCKG

6. V → O C K G D
Q → D
P →

7) V → OCKGDAI
Q → AI
P → OCKGD

8) V → OCKGDAIB
Q → IB
P → OCKGDA

9) V → OCKGDAIBU
Q → BU
P → OCKGDAI

Add R to the Queue as It is already visited

3. IMPLEMENTATION

Users > fevenbelay > Desktop > FevDesktop > SFBU-Semester2 > Practical algorithm > BFS-maze.py > ...

```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

```

4. TEST

```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]
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 DEBUG CONSOLE TERMINAL

```
/usr/bin/python3 "/Users/fevenbelay/Desktop/FevDesktop/SFBU-Semester2/Practical algorithm/BFS-maze.py"
(base) fevenbelay@Fevens-MacBook-Air ~ % /usr/bin/python3 "/Users/fevenbelay/Desktop/FevDesktop/SFBU-Semester2/Practical algorithm/BFS-maze.py"
True
False
False
(base) fevenbelay@Fevens-MacBook-Air ~ %
```

5. ENHANCEMENT IDEAS

1. Space-Efficient Data Structures: Using more space-efficient data structures for the queue and visited set, such as compressed bit vectors, can significantly reduce the memory footprint of BFS.

2. Parallel BFS: BFS can be parallelized by processing multiple nodes at the same level concurrently, using parallel computing resources.

3. Sparse BFS: For sparse graphs, an adjacency list can be more efficient than an adjacency matrix, reducing the time complexity of checking for neighboring nodes from O(V) to O(deg(v)), where V is the number of vertices and deg(v) is the degree of a vertex.

4. Dynamic BFS: For graphs that change over time (dynamic graphs), maintaining BFS trees and updating them incrementally can be more efficient than re-computing the BFS from scratch after every change.

5. External BFS: For extremely large graphs that do not fit into memory, employing external memory algorithms can allow BFS to run on graphs stored on disk.

6. CONCLUSION

- This approach is ideal for mazes because it can find the shortest path without getting trapped in dead ends or visiting any part of the maze unnecessarily.

- Each position is visited at most once, ensuring the process is efficient and terminates once it either finds the destination or exhausts all possible paths.