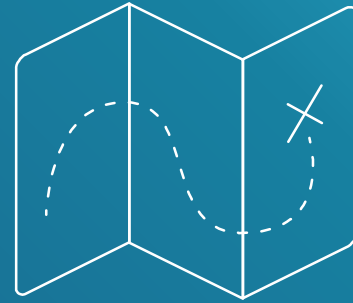




“The Maze” - LC - Breadth-First Traversal and Depth-First Traversal

Contents

- ◇ Introduction
- ◇ Design
- ◇ Implementation
- ◇ Test
- ◇ Enhancement Ideas
- ◇ Conclusion





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

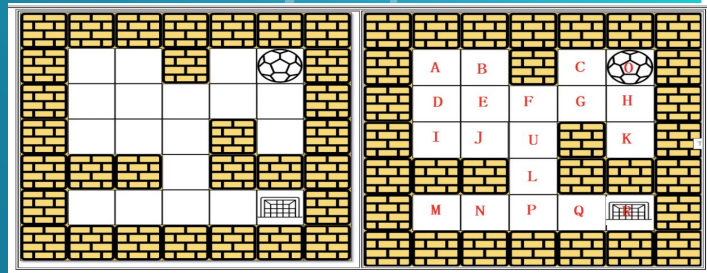
- Depth First Search (DFS) is a core algorithm for systematically exploring and visiting all nodes within a graph, utilizing a depth-first exploration technique.
- DFS traverses deeply into the graph structure, exploring as far as possible along each branch before backtracking, making it well-suited 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	8. Visited - OCHGRFEU Queue - EU Print - OCHGRFF
2) Visited - OCH Queue - CH Print - O	9. Visited - OCHGRFFUDBR Queue - UDBR Print - OCHGRFFE
3) Visited - OCHG Queue - HG Print - OC	10. Add R to the Queue Mark R as visited Print -
4) Visited - OCHGR Queue - GR Print - OCH	11. Visited - Queue - Print -
5) Visited - OCHGRF Queue - RF Print - OCHG	12. Visited - Queue - Print -
6) Visited - OCHGRF Queue - F Print - OCHGR	13. Visited - Queue - Print -
7) Visited - OCHGRF Queue - Print - OCHGRF	

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.

Visited -
Queue -
Print -

	A	B		C	⑥
	D	E	F	G	H
	I	R	U		K
			L		
	M	N	P	Q	J

1) Visited - A D
Queue -
Print - 0

2) Visited - A C K
Queue - C K
Print - 0

3) Visited - A C K G
Queue - K G
Print - D C

4) Visited - A C K G
Queue - G
Print - D C K

5) Visited - A C K G
Queue -
Print - D C K G

6) Visited - A D C K G D
Q - D
P -

7) Visited - A D C K G D A I
Q - A I
P - D C K G D

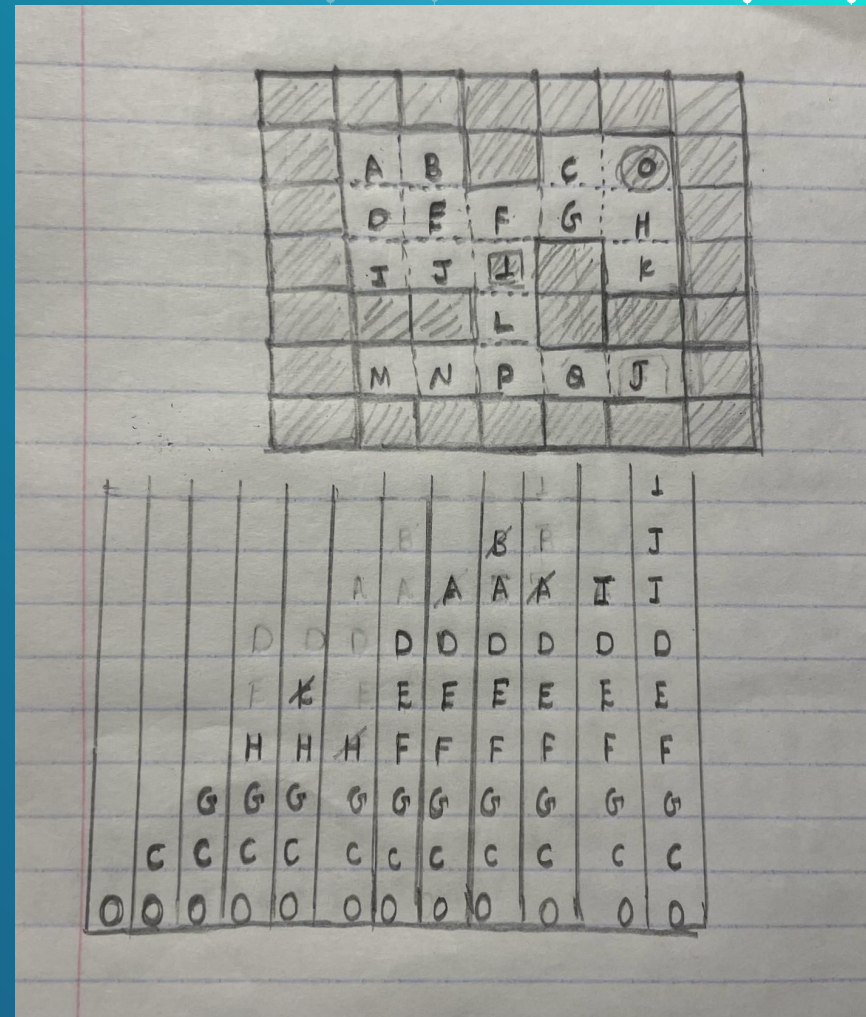
8) Visited - A D C K G D A I B
Q - I B
P - D C K G D A

9) Visited - A D C K G D A I B U
Q - B U
P - D C K G D A I

R is visited as it is between I AND U

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

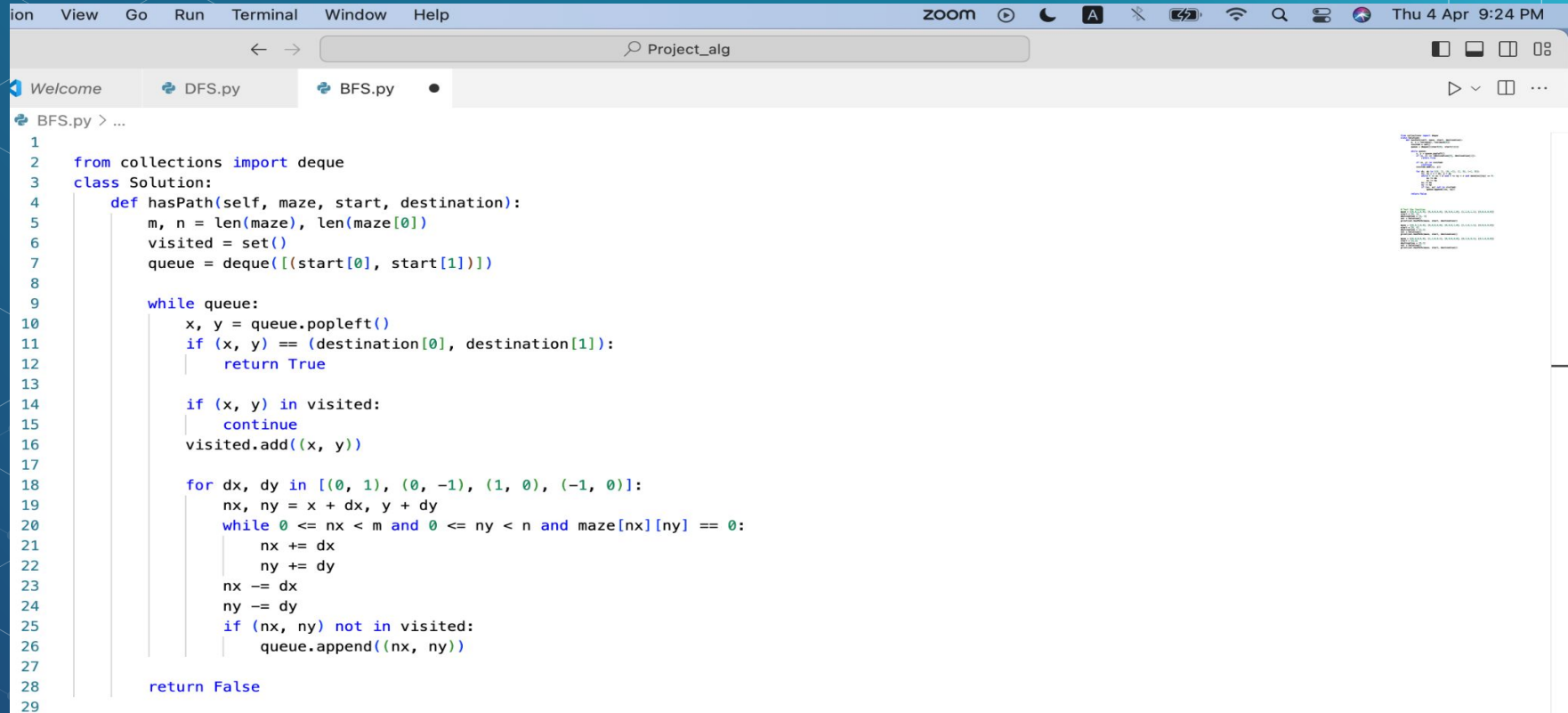
- Right, Left, Up , bottom
- The ball can go through the empty spaces by rolling right, left, up, down, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.





Implementation and Test

Breadth-First Traversal



The screenshot shows a code editor window titled 'Project_alg' with a menu bar (ion, View, Go, Run, Terminal, Window, Help) and a status bar (zoom, Thu 4 Apr 9:24 PM). The editor has tabs for 'Welcome', 'DFS.py', and 'BFS.py'. The 'BFS.py' tab is active, showing a Python script for finding a path in a maze using Breadth-First Traversal. The code is as follows:

```
1 from collections import deque
2 class Solution:
3     def hasPath(self, maze, start, destination):
4         m, n = len(maze), len(maze[0])
5         visited = set()
6         queue = deque([(start[0], start[1])])
7
8         while queue:
9             x, y = queue.popleft()
10            if (x, y) == (destination[0], destination[1]):
11                return True
12
13            if (x, y) in visited:
14                continue
15            visited.add((x, y))
16
17            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
18                nx, ny = x + dx, y + dy
19                while 0 <= nx < m and 0 <= ny < n and maze[nx][ny] == 0:
20                    nx += dx
21                    ny += dy
22                nx -= dx
23                ny -= dy
24                if (nx, ny) not in visited:
25                    queue.append((nx, ny))
26
27            return False
28
29
```

Click to add a breakpoint. on

```
36 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]]
37 start = [0, 4]
38 destination = [4, 4]
39 sol = Solution()
40 print(sol.hasPath(maze, start, destination))
41
42 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]]
43 start = [0, 4]
44 destination = [3,2]
45 sol = Solution()
46 print(sol.hasPath(maze, start, destination))
47
48 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]]
49 start = [4,3]
50 destination = [0,1]
51 sol = Solution()
52 print(sol.hasPath(maze, start, destination))
```

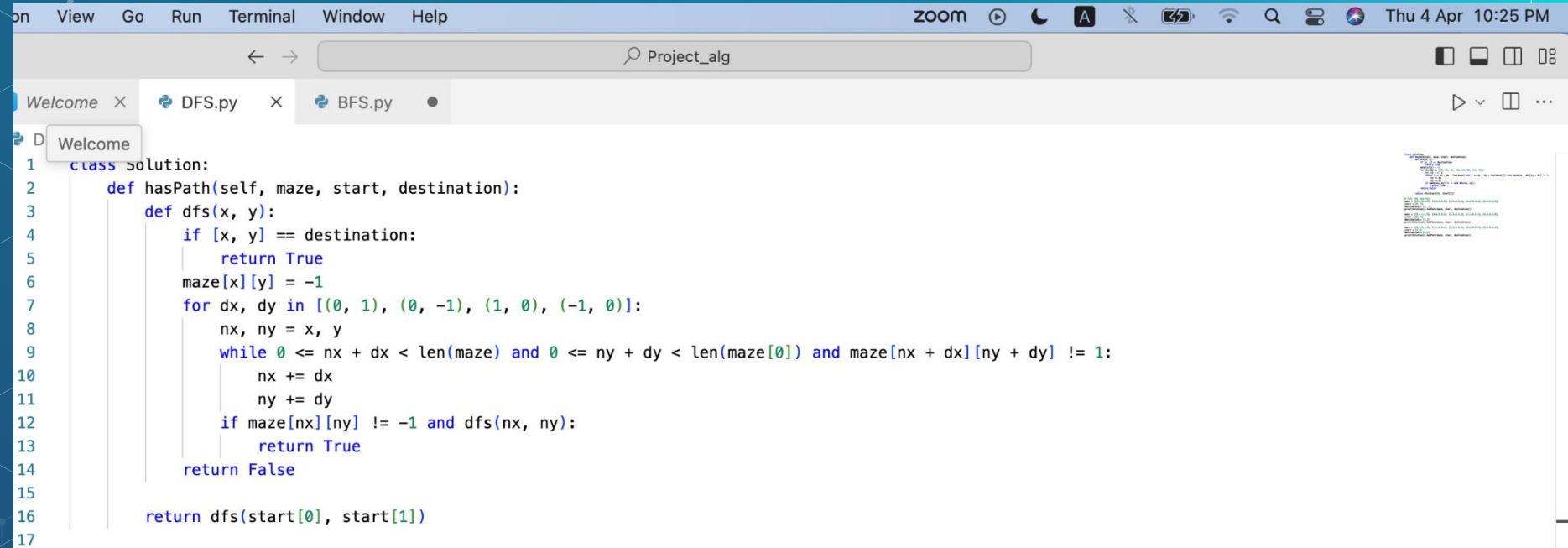
PROBLEMS OUTPUT TERMINAL

^ X

> > > 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



The screenshot shows a code editor window with a menu bar (File, View, Go, Run, Terminal, Window, Help) and a toolbar with icons for zoom, dark mode, font size, and search. The address bar shows 'Project_alg'. The editor has three tabs: 'Welcome', 'DFS.py', and 'BFS.py'. The 'Welcome' tab is active, displaying a Python class named 'Solution' with a 'hasPath' method. The method uses a recursive 'dfs' function to explore the maze. The maze is represented as a 2D array where -1 indicates a wall. The 'dfs' function checks if the current position is the destination, marks the current cell as visited, and recursively explores the four adjacent cells. The 'hasPath' method returns the result of the 'dfs' function starting from the given start coordinates.

```
1 class Solution:
2     def hasPath(self, maze, start, destination):
3         def dfs(x, y):
4             if [x, y] == destination:
5                 return True
6             maze[x][y] = -1
7             for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
8                 nx, ny = x, y
9                 while 0 <= nx + dx < len(maze) and 0 <= ny + dy < len(maze[0]) and maze[nx + dx][ny + dy] != 1:
10                     nx += dx
11                     ny += dy
12                 if maze[nx][ny] != -1 and dfs(nx, ny):
13                     return True
14             return False
15
16         return dfs(start[0], start[1])
17
```



```

17
18 # Test the function
19 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]]
20 start = [0, 4]
21 destination = [4, 4]
22 print(Solution().hasPath(maze, start, destination))
23
24 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]]
25 start = [0, 4]
26 destination = [3,2]
27 print(Solution().hasPath(maze, start, destination))
28
29 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]]
30 start = [4,3]
31 destination = [0,1]
32 print(Solution().hasPath(maze, start, destination))
33

```

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
• Project_al

```



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**