Solving LeetCode Question 490 - The Maze

Ke Yan

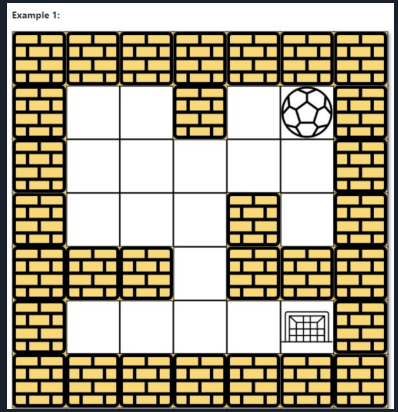
Subjects

- Introduction
- Design
    - Why this approach?
    - Identify and understand the problems.
    - Investigate possible solutions.
    - Theoretical comparison of solutions and selecting the best one.
- Examples
- Implementation
    - How the chosen approach works?
- Enhancement Ideas
- Conclusion

Problem Statement

LeetCode 490 - The Maze:
The Maze problem on LeetCode involves finding a path from the start position to the destination in a maze represented as a 2D matrix.
Each cell in the matrix can be either an obstacle (denoted by 1) or an empty space (denoted by 0).

The objective is to determine if there exists a valid path from the given start position to the destination position.
We'll explore two popular graph traversal algorithms, Breadth-First Search (BFS) and Depth-First Search (DFS), to tackle this maze navigation problem and compare their strengths and weaknesses.



490. The Maze

Medium 1480 151 Add to List Share

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 = [start_row, start_col]` and `destination = [destination_row, destination_col]`, return `true` if the ball can stop at the destination, otherwise return `false`.

You may assume that **the borders of the maze are all walls** (see examples).

Design

- Why Use DFS and BFS?

  DFS: Simple implementation, suitable for small mazes.

  BFS: Finds the shortest path, better for large mazes.

- Identifying and Understanding the Problems

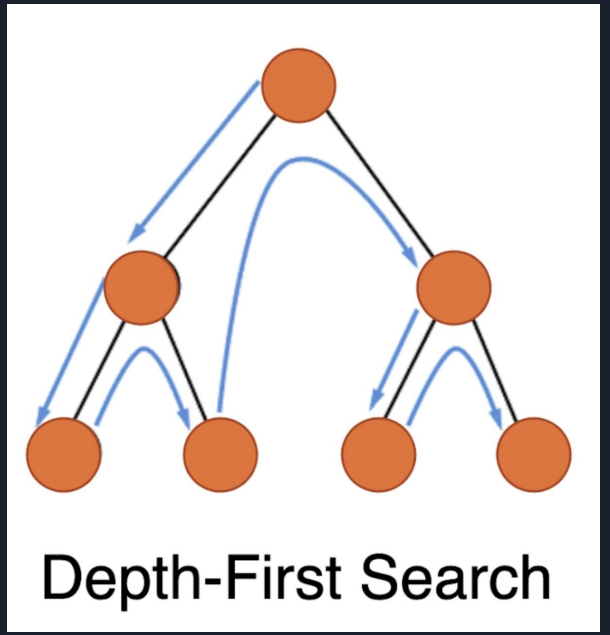
  Maze represented as a 2D matrix with obstacles and empty spaces.

  Goal: Determine if there exists a path from start to destination.

Theoretical Comparison
— DFS: Depth-First Search

- Depth-First Search is an algorithm that explores as far as possible along each branch before backtracking. It starts from the initial node (start position in the maze) and explores as deep as possible along each path before backtracking.

- DFS uses a stack data structure to keep track of visited nodes and explore the neighboring nodes in a last-in-first-out (LIFO) manner.



- Advantages of DFS:

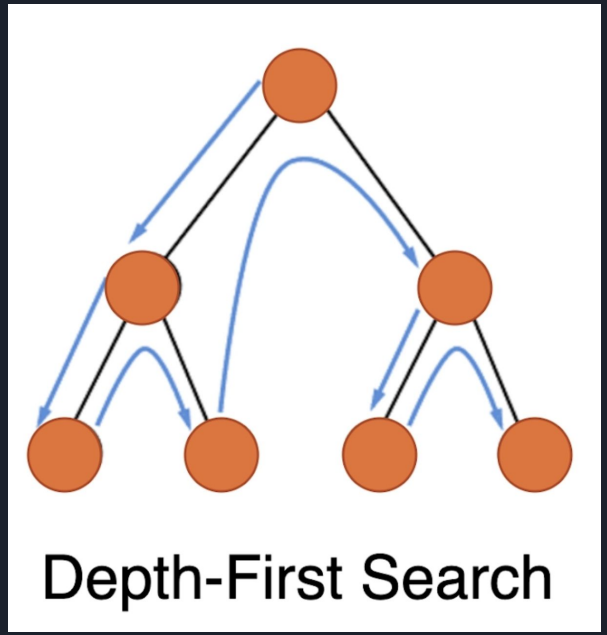
  Simplicity: The DFS algorithm is relatively straightforward to implement and understand.

  Memory Efficiency: DFS uses a limited amount of memory as it explores deeply along each path before considering other alternatives.

- Disadvantages of DFS:

  Completeness: DFS may not always find the optimal path or shortest path between the start and destination. It can get stuck in local paths and miss potential shorter paths elsewhere.

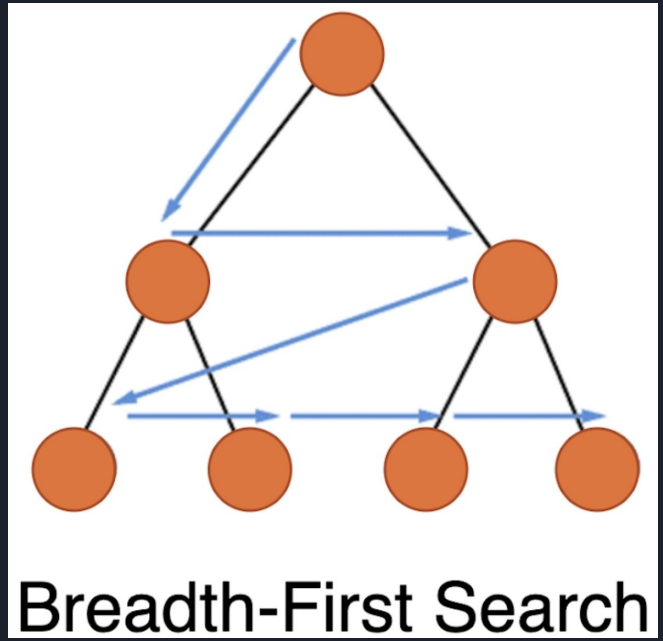
  Non-guaranteed Shortest Path: Due to its nature of exploring deeply, DFS might not guarantee the shortest path in a graph or maze.



Theoretical Comparison
—BFS: Breadth-First Search

- Breadth-First Search is an algorithm that explores all the nodes at the present depth before moving on to the next level. It starts from the initial node (start position in the maze) and explores all its neighbors before moving on to their neighbors.

- BFS uses a queue data structure to keep track of visited nodes and explore the neighboring nodes in a first-in-first-out (FIFO) manner.



- Advantages of BFS:

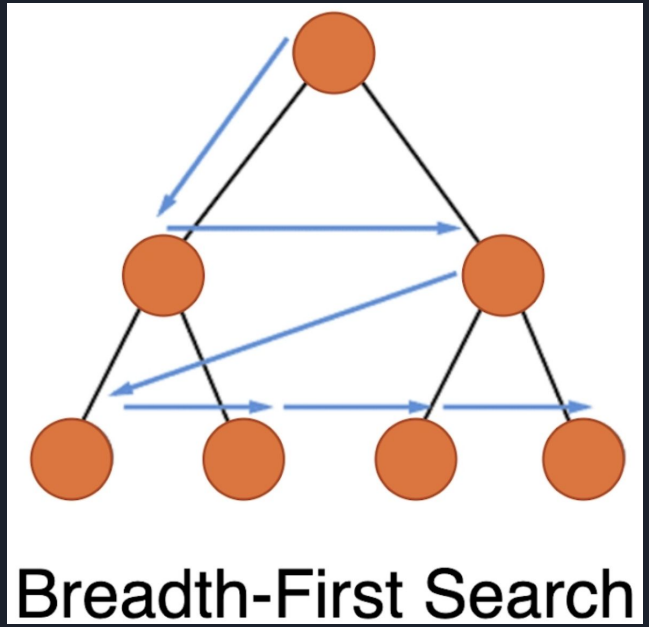
  Completeness: BFS explores all possible paths in a graph, ensuring that it will find the shortest path from the start to the destination (if it exists).

  Shortest Path Guarantee: BFS guarantees the shortest path because it explores the neighbors level by level, and the first occurrence of the destination will be the shortest path.

- Disadvantages of BFS:

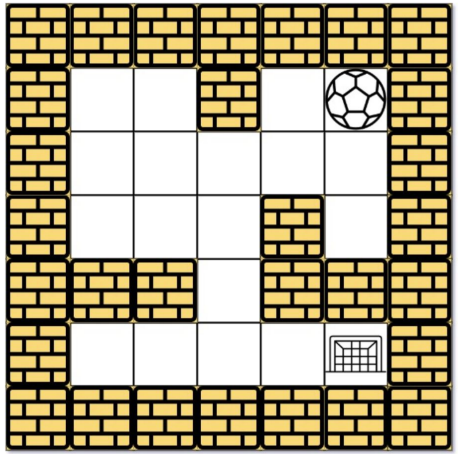
  Memory Requirements: BFS requires more memory than DFS as it keeps all the nodes at the current level in memory.

  Implementation Complexity: Implementing BFS can be slightly more complex than DFS due to its use of a queue.



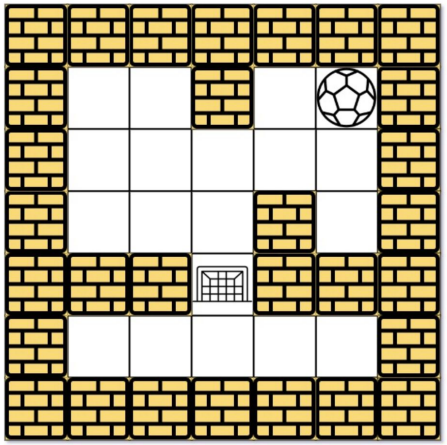
Examples

Example 1:



Input: 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]
Output: true
Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.

Example 2:



Input: 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]
Output: false
Explanation: There is no way for the ball to stop at the destination. Notice that you can pass through the destination but you cannot stop there.

Example 3:

```
Input: 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]
Output: false
```

Implementation & Test – DFS

```python
def hasPath_dfs(maze, start, destination):
    rows = len(maze)
    cols = len(maze[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]

    def dfs(x,y):
        if visited[x][y] == True:
            return False
        visited[x][y] = True

        if [x,y] == destination:
            return True

        left = y
        while (left > 0 and maze[x][left-1]==0):
            left -= 1
        if dfs(x,left):
            return True

        right = y
        while(right<cols-1 and maze[x][right + 1]==0):
            right +=1
        if dfs(x,right):
            return True

        up = x
        while(up>0 and maze[up-1][y] == 0):
            up -= 1
        if dfs(up,y):
            return True

        down = x
        while( down < rows-1 and maze[down+1][y] == 0):
            down += 1
        if dfs(down,y):
            return True

        return False
```

```python
# Test case 1
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(hasPath_dfs(maze, start, destination))  #output: True

# Test case 2
start2 = [0, 4]
destination2 = [3, 2]
print(hasPath_dfs(maze, start2, destination2)) #output: False

# Test case 3
maze3 = [
    [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]
]
start3 = [4, 3]
destination3 = [0, 1]
print(hasPath_dfs(maze3, start3, destination3)) #output: False
```

Implementation & Test – BFS

```python
from collections import deque
def hasPath_bfs(maze,start,destination):
    def is_valid(x,y):
        return 0<= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 0

    directions = [(0,1),(0,-1),(1,0),(-1,0)]
    queue = deque([start])
    visited = set()

    while(queue):
        x,y = queue.popleft()
        visited.add((x,y))

        if[x,y] == destination:
            return True

        for(dx,dy) in directions:
            nx = x
            ny = y
            while(is_valid(nx+dx,ny+dy)):
                nx += dx
                ny += dy
            if(nx,ny) not in visited:
                queue.append((nx,ny))

    return False
```

```python
# Test case 1
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(hasPath_bfs(maze, start, destination))

# Test case 2
start2 = [0, 4]
destination2 = [3, 2]
print(hasPath_bfs(maze, start2, destination2))

# Test case 3
maze3 = [
    [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]
]

start3 = [4, 3]
destination3 = [0, 1]
print(hasPath_bfs(maze3, start3, destination3))
```

Conclusion

- DFS, utilizing a stack for traversal, is relatively simple and memory-efficient, making it suitable for small mazes. However, it may not guarantee the shortest path and can explore unnecessary paths.On the other hand, BFS, employing a queue for traversal, explores all possible paths level by level, ensuring that it finds the shortest path if it exists. While BFS guarantees optimality, it requires more memory and slightly more complex implementation compared to DFS.

- After a theoretical comparison, we identified that the choice between DFS and BFS depends on the problem's requirements, the size of the maze, and the importance of finding the shortest path.

- For smaller mazes and scenarios where memory efficiency is crucial, DFS might be the preferred approach. However, when dealing with larger mazes or when finding the shortest path is a priority, BFS is the superior choice as it guarantees optimality and shortest path discovery.

- Throughout the implementation and testing, we verified the effectiveness of the chosen approach and its ability to navigate mazes efficiently.As with any problem-solving process, there are always potential enhancement ideas that can be explored to improve the solution further.

- In conclusion, both DFS and BFS offer valuable insights into maze navigation, and understanding their strengths and limitations empowers us to tackle maze-related challenges effectively.