Rat In A Maze GFG

This program is designed to find all possible paths from the source to the destination in a given maze using a backtracking approach. The maze is represented as a grid where each cell can be either open (1) or blocked (0). The goal is to determine all valid paths that can be taken to reach the destination cell from the source cell.

Approach 1: Function to find all possible paths in the maze from source to destination using Backtracking

1. The program starts by defining the **isPossible** function, which checks if a given cell is a valid move (within the maze boundaries, unvisited, and not blocked).
2. The **solvePath** function is the core of the backtracking approach. It takes the maze grid, its size **n**, the current path being explored, the current position **(srcX, srcY)**, a vector of strings to store the answer **ans**, and a matrix to track visited cells **visited**.
3. The base case of **solvePath** is when the current position **(srcX, srcY)** is the destination **(n - 1, n - 1)**. If this condition is met, the current path is added to the answer **ans**.
4. The program marks the current cell as visited by setting **visited[srcX][srcY]** to 1.
5. The program then explores four possible directions: Downwards (**D**), Leftwards (**L**), Rightwards (**R**), and Upwards (**U**). For each direction, it checks if the move is possible using the **isPossible** function. If it's possible, the direction is added to the current path, and the **solvePath** function is called recursively with the updated position. After the recursive call, the last character of the path is removed to backtrack.
6. Once all possible paths from the current position are explored, the current cell is marked as unvisited by setting **visited[srcX][srcY]** back to 0.
7. The **findPath** function initializes the source cell **(0, 0)** and an empty path. It then creates the **visited** matrix and initializes it with all 0s.
8. The **findPath** function first checks if the source or destination cell is blocked. If either is blocked, it returns an empty answer vector since no paths are possible.
9. The **findPath** function then calls **solvePath** with the initial parameters and returns the final answer.
10. In the **main** function, a sample maze grid is defined. The **findPath** function is called to find all paths from the source **(0, 0)** to the destination **(n - 1, n - 1)**. The program then prints the found paths.
11. **Time Complexity: In the program, for each cell in the maze, we can explore up to four directions (up, down, left, right) to determine if a valid path exists. In the worst case, we could potentially explore all possible paths from each cell. Considering an NxN maze, the maximum number of cells is N^2. Thus, the time**

complexity becomes O(4^(N^2)) because we have a branching factor of 4 for each cell, and we potentially explore up to N^2 cells.

12. **Space Complexity: O(N^2) due to the space required by the call stack and the space used to store the paths.**