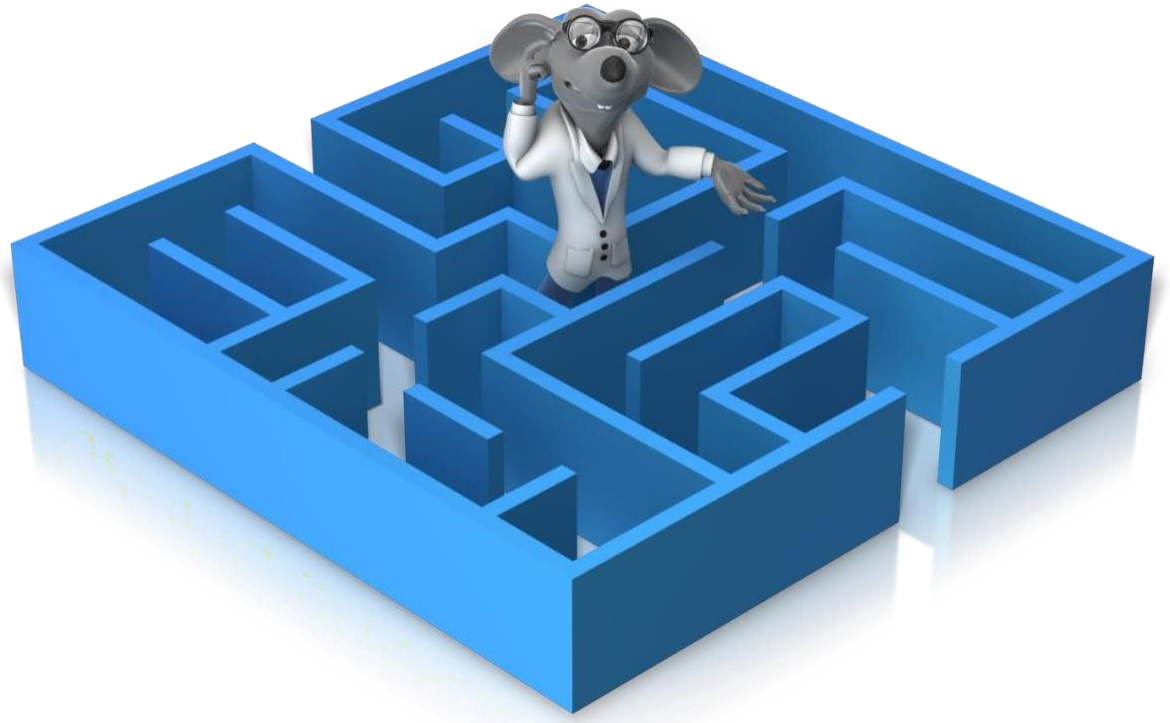
DAA PROJECT

Rat in a Maze



Contents

PROBLEM DEFINITION:

- The rat in a maze problem is one of the famous backtracking problems.
- A rat starts from source and has to reach the destination.
- Different paths a rat can move from source to destinations is found in this problem.
- A rat can move only in 2 directions forward and downward.

PROBLEM EXPLANATION WITH DIAGRAM AND EXAMPLE:

- A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze [0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.
- In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.

Gray blocks are dead ends (value = 0).

Following is a binary matrix representation of the above maze.

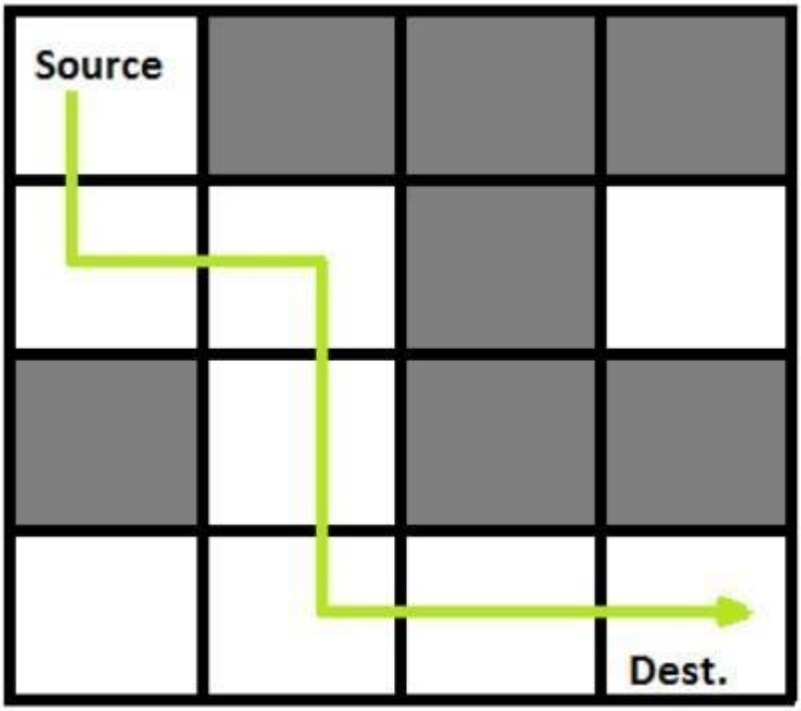
{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All entries in solution path are marked as 1.

DESIGN TECHNIQUES USED:

Backtracking:

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally.
Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reachingany level of the search tree) is the process of backtracking.

ALGORITHM:

Backtracking technique is used in this problem.

Approach:
Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination, then backtrack and try other paths.

Algorithm:

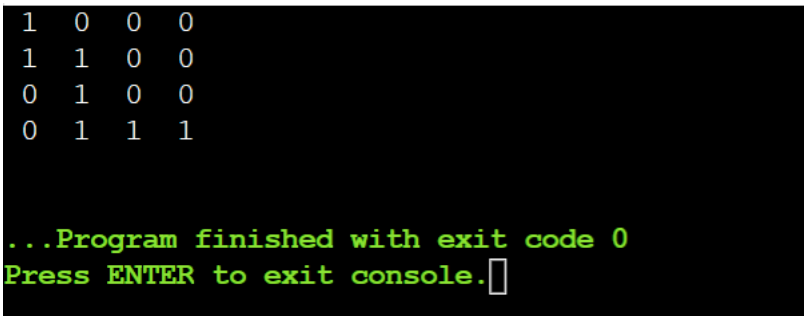
1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i-1,j), (I,j-1), (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e., output[i][j] = 0.

ALGORITHM EXPLANATION:

```c
#include <stdio.h>
#include <stdbool.h>
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y,int sol[N][N]);
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}
bool isSafe(int maze[N][N], int x, int y)
{
    if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;
    return false;
}
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };
    if (solveMazeUtil(maze, 0, 0, sol) == false) {
        printf("Solution doesn't exist");
        return false;
    }
    printSolution(sol);
    return true;
}
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
        sol[x][y] = 1;
        return true;
    }
    if (isSafe(maze, x, y) == true) {
        if (sol[x][y] == 1)
            return false;
        sol[x][y] = 1;
        if (solveMazeUtil(maze, x + 1, y, sol) == true)
            return true;
        if (solveMazeUtil(maze, x, y + 1, sol) == true)
            return true;
        sol[x][y] = 0;
        return false;
    }
    return false;
}
int main()
{
    int maze[N][N] = { { 1, 0, 0, 0 },
                       { 1, 1, 0, 1 },
                       { 0, 1, 0, 0 },
                       { 1, 1, 1, 1 } };
    solveMaze(maze);
    return 0;
}
```

OUTPUT:-



COMPLEXITY ANALYSIS:

- **Time Complexity:** O(2^(n^2)).
  The recursion can run upper-bound 2^(n^2) times.
- **Space Complexity:** O(n^2).
  Output matrix is required so an extra space of size n*n is needed.

CONCLUSION:

We have created a solution for a real-life scenario completely from scratch. It helps us to write logic and maintain clean structure in code. This helps in creating solutions for real time problems in our day-to-day life.