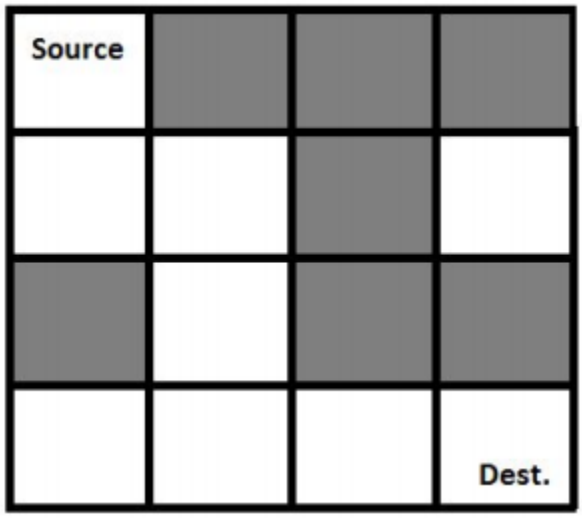
PROJECT TITLE : Rat in Maze

Overview

A maze is in the form of a 2D matrix in which some cells/blocks are blocked. One of the cells is termed as a source cell, from where we have to start. And another one of them is termed as a destination cell, where we have to reach. We have to find a path from the source to the destination without moving into any of the blocked cells.

PROBLEM STATEMENT :



We have to find a path from the source to the destination without moving into any of the blocked cells. A picture of an unsolved maze is shown above , where grey cells denote the dead ends and white cells denote the cells which can be accessed.

ABSTRACTION:

To solve these types of puzzle, we first start with the source cell and move in a direction where the path is not blocked. If the path taken makes us reach the destination, then the puzzle is solved. Otherwise, we come back and change our direction of the path taken.

METHODOLOGY USED :

Backtracking is a famous algorithmic-technique for solving/resolving 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 reaching any level of the search tree) is the process of backtracking.

ALGORITHM :

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes an initial matrix (Maze), output matrix (Solution) 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 the destination or not. If the destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.

CODE :

```
#include <bits/stdc++.h>

using namespace std;

#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++)

cout<<" "<<sol[i][j]<<" ";

cout<<endl;

}}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) {
                cout<<"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;
```

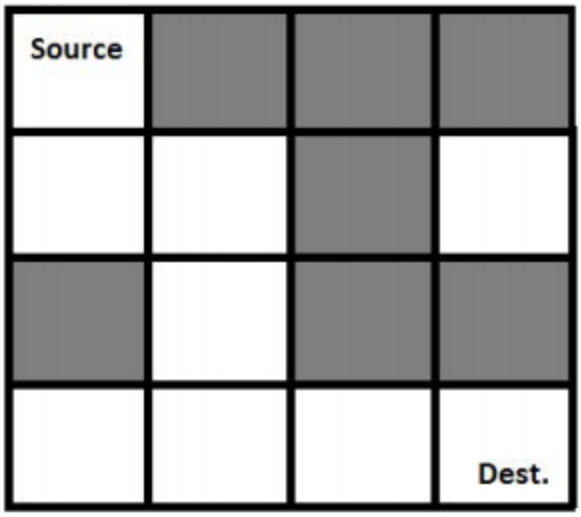
```
                if (solveMazeUtil(maze, x, y + 1, sol) == true)
                        return true;
                sol[x][y] = 0;
                return false;
        }
        return false;

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;
}
```

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.

EXAMPLE FOR RAT MAZE PROBLEM :



SOLUTION