**Design and Analysis of Algorithms**

**Rat IN A MAZE**

A Project Submitted

in Partial Fulfilment of the Requirements

for the Degree of

**Bachelor of Technology**

in

**Computer science and engineering**

**As part of “CSE 2719 - Design and Analysis of Algorithms” course**

BY

**Y SRI PRANEETH [200348]**

**CH. ROHITH REDDY [200378]**

**SRI SIVA SATYA [200377]**

**ANVITHA MARAM [200399]**

**KHUSHI TELUGU [200400]**



SCHOOL OF ENGINEERING AND TECHNOLOGY

BML MUNJAL UNIVERSITY GURGAON, 2021

Abstract

For our design and analysis of algorithms course, we opted to solve a rat in a maze problem. The backtracking algorithm is used to solve the rat in a maze problem. The rat in a maze problem occurs when a rat is present in a square maze, either at the source or at any place, with the goal of reaching its destination. It has the ability to take any path and come up with multiple solutions. To solve the problem, we applied the Python programming language. Now we'll look at the specifics of how it's done.

features implemented:

Initially, the rat could only move in two directions: right and down, but we have now developed it into four directions, allowing the rat to move in all four directions: up, down, right, and left.

algorithm used:

In this problem we use back tracking algorithm. Firstly, we will make a matrix to represent the maze, and the elements of the matrix will be either 0 or 1. Where, 1 represents the blocked cell and 0 represent the cells in which we can move. The matrix for the maze shown above is:

0 1 0 1 1

0 0 0 0 0

1 0 1 0 1

0 0 1 0 0

1 0 0 1 0

Now, we will make one more matrix of the same dimension to store the solution. Its elements will also be either 0 or 1. 1 will represent the cells in our path and rest of the cells will be 0. The matrix representing the solution is:

1 0 0 0 0

1 1 1 1 0

0 0 0 1 0

0 0 0 1 1

0 0 0 0 1

Thus, we now have our matrices. Next, we will find a path from the source cell to the destination cell and the steps we will take are:

1. Check for the current cell, if it is the destination cell, then the puzzle is solved.
2. If not, then we will try to move downward and see if we can move in the downward cell or not (to move in a cell it must be vacant and not already present in the path).
3. If we can move there, then we will continue with the path taken to the next downward cell.
4. If not, we will try to move to the rightward cell. And if it is blocked or taken, we will move upward.
5. Similarly, if we can't move up as well, we will simply move to the left cell.
6. If none of the four moves (down, right, up, or left) are possible, we will simply move back and change our current path (backtracking).

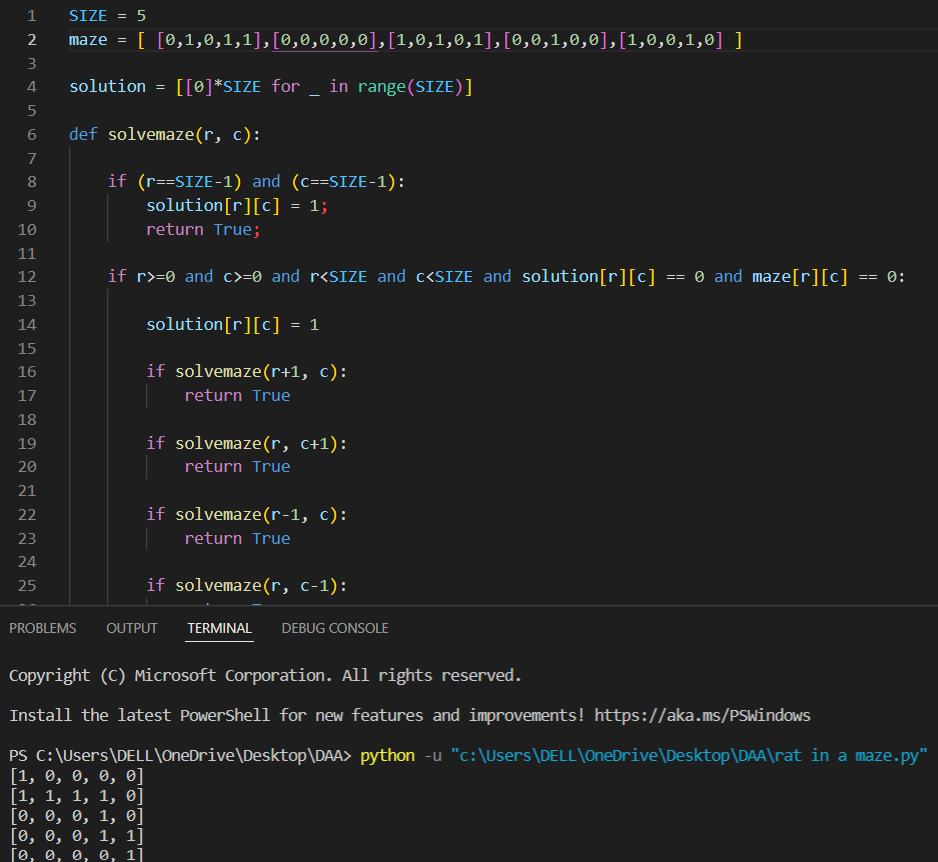
Thus, the summary is that we try to move to the other cell (down, right, up, and left) from the current cell and if no movement is possible, then just come back and change the direction of the path to another cell.

Code explanation:

Let us now clearly know about everything that we have written in our code.

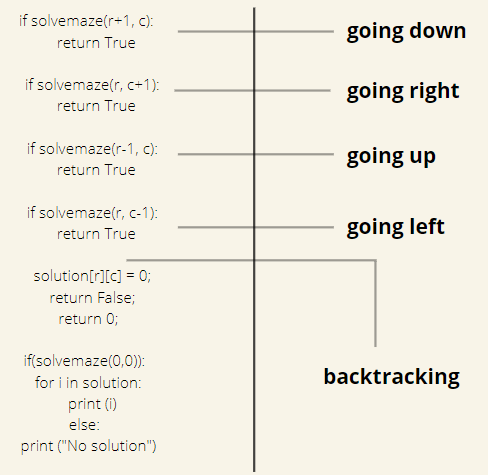
First and foremost, we will declare the matrix size. After giving the input maze, we then create a list too store the solution matrix. Then, we define a function “**solvemaze**”.

This is the actual function where the backtracking algorithm is implemented.



Firstly, we check if our cell is the destination cell or not if (r = = SIZE-1) and

(c = = SIZE-1). If it's the destination cell, we've already solved the riddle. If that's not the case, we'll see if it's a valid cell to transfer. Following are the conditions that must be checked. A valid cell must be in the matrix i.e., indices must be between 0 to SIZE-1 r>=0 && c>=0 && r<size, must not be blocked maze [r] [c] = = 0 and must not be taken in the path solution [r] [c] = = 0. Firstly, we will try the downward cell if (solvemaze (r+1, c)). If it doesn't give us the solution then we will move to the rightward cell, and similarly to the upward and the leftward cells. If all of the cells fail to give us the solution, we will leave the cell solution [r] [c] = 0 and go to some other cell.



The above one explains the transverse of rat in the 4 directions and mentions backtracking.

time complexity:

**O(4^(n^2))**

Because on every cell we need to try 4 different directions.

Space complexity:

**O(n^2)**

Output matrix is required so an extra space of size n\*n is needed. It has large space complexity because we are using recursion.

gITHUB REPOSITORY LINK:

<https://github.com/Praneethyakkala/Rat-in-a-maze_-group>

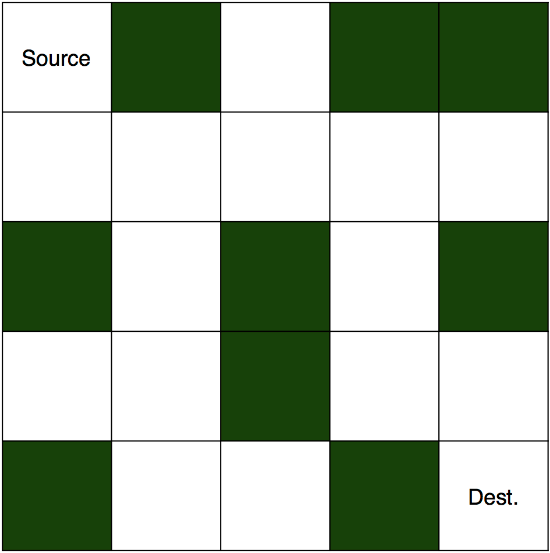
future work:

We'll try to write code for the identical problem, but with lesser complexity.

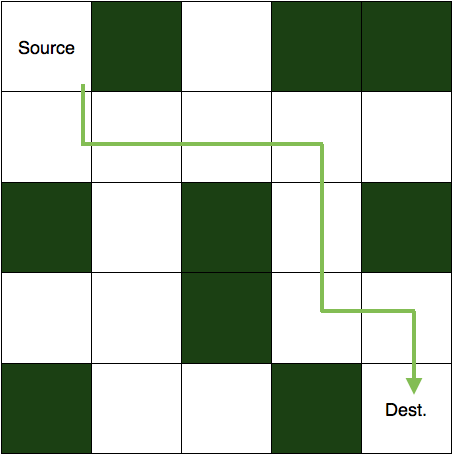
The current code only works with square rat mazes. We'll work on several rat maze shapes and try to build code for them.

demo:

A maze is a 2D matrix in which some cells are blocked. One of the cells is the source cell, from where we have to start. And another one of them is the destination, where we must reach. 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 below.



And this is its solution.



To solve this puzzle, we first start with the source cell and move in a direction where the path is not blocked. If taken path makes us reach to the destination then the puzzle is solved else, we come back and change our direction of the path taken. We are going to implement the same logic in our code also. So, let's see how.

screenshots:

Text

Description automatically generated

acknowledgements:

First and foremost, we would want to express our gratitude to Nitin Varyani, our course instructor, for helping us through this project. We were able to complete our job correctly and on time thanks to his coaching. Finally, we'd want to express our gratitude to all of the members who contributed to the project's success.