RAT IN A MAZE

A MINI PROJECT REPORT

18CSC204J -Design and Analysis of Algorithms Laboratory

Submitted by

Nandhigama Abhinav [RA2011028010125]

V J Venkata rami reddy[RA2011028010142]

Under the guidance of

Dr. B Yamini

Assistant Professor, Department of Networking and Communication

In Partial Fulfillment of the Requirements for the Degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING
with specialization in Cloud computing

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

COLLEGE OF ENGINEERING AND TECHNOLOGY SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR- 603 203

June 2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this mini project report titled "RAT IN A MAZE" is the bonafide work done by NANDHIGAMA ABHINAV (RA2011028010125) and V J VENKATA RAMI REDDY (RA2011028010142) who carried out the mini project work and Laboratory exercises under my supervision for **18CSC204J -Design and Analysis of Algorithms Laboratory**. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

Dr. B Yamini
ASSISTANT PROFESSOR
18CSC204J -Design and Analysis of Algorithms
Course Faculty
Department of Networking and Communications

Signature of the Internal Examiner-I **Signature of the Internal Examiner-II**

ABSTRACT

Maze game is a well-known problem, where we are given a grid of 0’s and 1’s, 0’s corresponds to a place that can be traversed, and 1 corresponds to a place that cannot be traversed (i.e. a wall or barrier); the problem is to find a path from bottom left corner of grid to top right corner; immediate right, immediate left, immediate up and immediate down only are possible (no diagonal moves). We consider a variant of the maze problem where a cost (positive value) or profit (negative value) is attached to visiting each location in the maze, and the problem is to find a path of least cost through the maze.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER-1

PROBLEM DEFINITION

- The Rat in a maze is a famous problem in which we have a N x N matrix (the maze) which consists of two types of cells - one the rat can pass through and the other that the rat cannot pass through.
- The objective of this problem is that the rat will be at a particular cell and we have to find all the possible paths that the rat can take to reach the destination cell from the given source cell.
- Now you will be building a simple react application that will visualize all the possible paths on the web page.

CHAPTER-2

PROBLEM EXPLANATION

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).

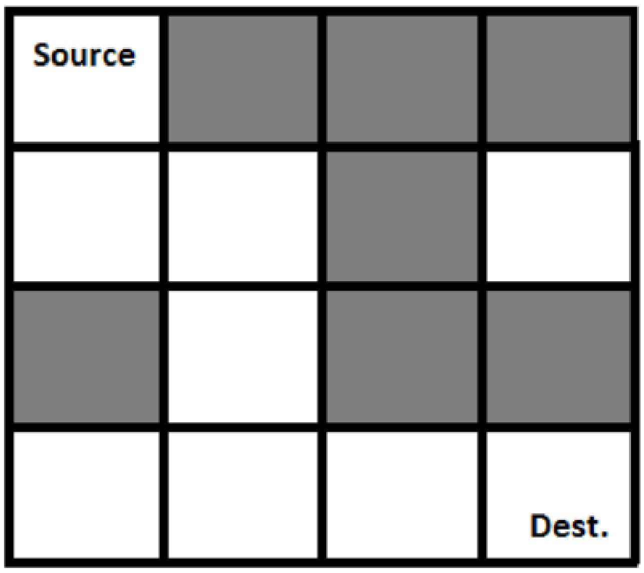

Figure-1

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

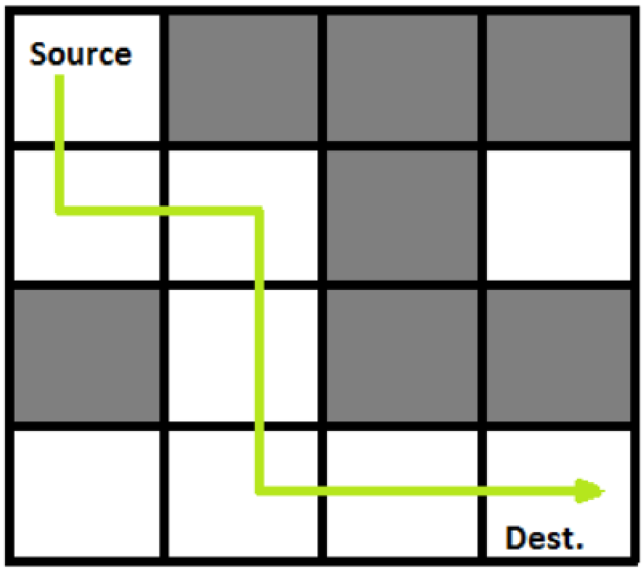

Figure-2

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.

CHAPTER-3

DESIGN TECHNIQUES

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 reaching any level of the search tree) is the process of backtracking.

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the **Brute force search** to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions.

Pros

- Backtracking can almost solve any problems, due to its brute-force nature.
- Can be used to find all the existing solutions if there exists for any problem.
- It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
- Very easy to write the code, and also to debug.

Cons

- It is very slow compared to other solutions.
- Depending on the data that you have, there is the possibility to perform a very large search with **Backtracking** and at the end don't find any match to your search params.
- Very-high computational cost, **Backtracking** is a recursive algorithm that consumes a lot from the memory and from the processor.
- If you need an optimized algorithm, that is able to handle a large volume of data and find all the possible solutions consuming less computational resources and in a shorter interval of time, you may consider using the **branch-and-bound** algorithm.

CHAPTER-4

ALGORITHM FOR THE PROBLEM

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.

CHAPTER-5

EXPLANATION OF ALGORITHM

ALGORITHM EXPLANATION:

```
 public class RatMaze
{
final int N = 6;
 /* A utility function to print solution matrix
 sol[N][N] */
 void printSolution(int sol[][])
 {
 for (int i = 0; i < N; i++)
 {
 for (int j = 0; j < N; j++)
System.out.print(" " + sol[i][j] +" ");
System.out.println();
}
 }
 /* A utility function to check if x,y is valid index for N*N maze */
boolean isSafe(int maze[][], int x, int y)
 {
 // if (x,y outside maze) return false
 return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
}
 boolean solveMaze(int maze[][])
 {
 int sol[][] = {{0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0},
 };
 if (solveMazeUtil(maze, 0, 0, sol) == false)
 {
/*for(int i=0;i<6;i++)
{
For(int j=0;j<6;j++)
System.out.print(sol[i]]j]+" ");
System.out.println();
}*/
System.out.print(“Solution doesn’t exist”);
Return false;
}
printSolution(sol);
 return true;
 }
 /* A recursive utility function to solve Maze problem */
 boolean solveMazeUtil(int maze[][], int x, int y, int sol[][])
 {
 // if (x,y is goal) return true
 if (x == N - 1 && y == N - 1)
 {
 sol[x][y] = 1;
 return true;
}
 // Check if maze[x][y] is valid
 if (isSafe(maze, x, y) == true)
 {
 // mark x,y as part of solution path sol[x][y] = 1;
 /* Move forward in x direction */
 if (solveMazeUtil(maze, x + 1, y, sol)) return true;
 //if (solveMazeUtil(maze,x-1,y,sol))
 // return true;
 /* If moving in x direction doesn't give solution then Move down in y direction */
 if (solveMazeUtil(maze, x, y + 1, sol)) return true;
 sol[x][y] = 0; return false;
 }
 return false;
 }
 public static void main(String args[])
 {
RatMaze rat = new RatMaze();
 int maze[][] = { {1,0,1,1,1,0 },
 {1,0,0,1,1,1 },
{1,1,0,1,0,1 },
{1,0,0,0,0,1 },
 {1,1,0,0,0,1 },
{0,1,1,1,1,1 },
 };
 rat.solveMaze(maze);
}
}
```

PSEUDO CODE:-

```
        isValid(x, y)
Input: x and y point in the maze.
Output: True if the path  (x,y) place is valid, otherwise false.
Begin
 if x and y are in range and (x,y) place is not blocked, then
            return true
          return false
        End
solveRatMaze(x, y)
Input: The starting point x and y.
Output: The path to be followed  by the rat to reach the
destination, otherwise false.
Begin
if (x,y) is the bottom right corner, then
            mark the place as 1
            return true
          if isValidPlace(x, y) = true, then
            mark (x, y) place as 1
          /* For the forward movement*/  if solveRatMaze(x+1,
        y) = true, then
              return true
  /*For the downward movement */ if solveRatMaze(x, y+1) =
true, then
              return true
            mark (x,y) as 0 when backtracks
            return false
          return false,End
```

CHAPTER-6

COMPLEXITY ANALYSIS

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.

CHAPTER-7

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.

REFERENCES

1) https://www.geeksforgeeks.org/backtracking-introduction

2) Rat-in-a-Maze-mini-project/rat in a maze(mini project) at main · VivekBurugadda/Rat-in-a-Maze-mini-project · GitHub

3) https://www.youtube.com/watch?v=elyagipwO8w

4)https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/#:~:text=The%20N%20Queen%20is%20the,two%20queens%20attack%20each%20other.&text=The%20expected%20output%20is%20a,for%20above%204%20queen%20solution.

5)https://pt.wikipedia.org/wiki/Backtracking

6)https://towardsdatascience.com/genetic-algorithm-vs-backtracking-n-queen-problem-cdf38e15d73f

7)https://intellipaat.com/community/52752/what-is-backtracking-in-artificial-intelligence

8)https://www.baeldung.com/cs/backtracking-algorithms

9)https://www.quora.com/What-are-the-advantages-and-disadvantages-of-a-backtracking-algorithm

APPENDIX

CODE

1. CODE

```
#include <stdio.h>
#include <stdbool.h>
// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y,int sol[N][N]);

// A utility function to print solution matrix 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");
   }
}

// A utility function to check if x, y is valid index for
// N*N maze
bool isSafe(int maze[N][N], int x, int y)
{
   // if (x, y outside maze) return false
   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;
}

// A recursive utility function to solve Maze problem
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // if (x, y is goal) return true
    if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
        sol[x][y] = 1;
        return true;
    }
    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true) {
        // Check if the current block is already part of
        // solution path.
        if (sol[x][y] == 1)
            return false;
        // mark x, y as part of solution path
        sol[x][y] = 1;
        /* Move forward in x direction */
        if (solveMazeUtil(maze, x + 1, y, sol) == true)
            return true;
        // If moving in x direction doesn't give solution
        // then Move down in y direction
        if (solveMazeUtil(maze, x, y + 1, sol) == true)
            return true;
        // If none of the above movements work then
        // BACKTRACK: unmark x, y as part of solution path
        sol[x][y] = 0;
        return false;
    }
    return false;
}
// driver program to test above function
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:-

```
1  0  0  0
1  1  0  0
0  1  0  0
0  1  1  1
```

- **printMaze():** This function is just printing the original maze matrix.

- **printSolution():** This function is just printing the solution matrix derived after applying the backtracking algorithm.

- **solveMaze():** This is the main function where we are implementing our backtracking algorithm. First, we are checking if our cell/block is the destination cell or not if (r==SIZE-1) and (c==SIZE-1). If it is the destination cell then our maze is already solved. If not, then we search whether it is a valid cell to move or not. A valid cell must be in the 2D-matrix i.e., indices must between 0 to SIZE-1 i.e r>=0 && c>=0 && r