

18CSC204J – Design and Analysis of Algorithms

Solving snake and ladder game using Breadth first search

A MINI PROJECT REPORT

Submitted by

Reddy jyothi sri D (RA2011028010108)

Jayanth Sai Yarlagadda(RA2011028010054)

Under the guidance of

Ms. Shobana

(Assistant Professor, Department of Computer Science & Engineering) ***in***

partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



SRM

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

S.R.M. Nagar, Kattankulathur, Kancheepuram District

JUNE 2020

SRMUNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled “Solving snake and ladder game using Breadth first search” is the bonafide work of “Reddy jyothi sri D[Reg No: RA2011028010108], Jayanth Sai Yarlagadda [Reg No: RA2011028010108], who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Signature of the Internal Examiner

Signature of the External Examiner

ABSTRACT

This game needs no introduction. We all must have played it in our childhood. Using BFS we can find the Shortest Path to win the game, and, we will state that path and the number of moves of dice it takes too.

The idea is to consider the snakes and ladders board as a directed graph and run Breadth-first search (BFS) from the starting node, vertex 0, as per game rules. Now the problem is reduced to finding the shortest path between two nodes in a directed graph problem. We represent the snakes and ladders board using a map. Snakes and Ladders is a popular game that we all played as children.

For those of you who are new to the game, it's a simple game played on a board that consists of 10×10 squares. The player moves by throwing a dice, you start from square 1, and move your way up to square 100. Some squares have 'ladders' that can propel you forward a few moves, and other squares have 'snakes' that would send you back a few moves. The goal of the game is to reach square 100 first.

There are many graphing algorithms available, but the most appropriate one for us is Breath First Search. This algorithm is used to find the shortest path between two vertices on the graph. Every vertex starts with a root vertex, and every vertex on the graph has an adjacency list. Which is basically a list of vertices that can be reached from that node. Vertices are connected by edges. So to find the shortest path between two vertices you only need to count the number of edges between them.

Table of Contents:

i Abstract

1. Chapter 1

1.1 Introduction

1.2 Problem Statement

2. Chapter 2

2.1 Definition of BREADTH FIRST SEARCH

2.2 The Algorithm of BFS

2.3 Approach and implimentation

2.4 Algorithm

3. Chapter 3

3.1 Time Complexity Analysis

3.2 CODE

Conclusion

Reference

Chapter 1

Introduction

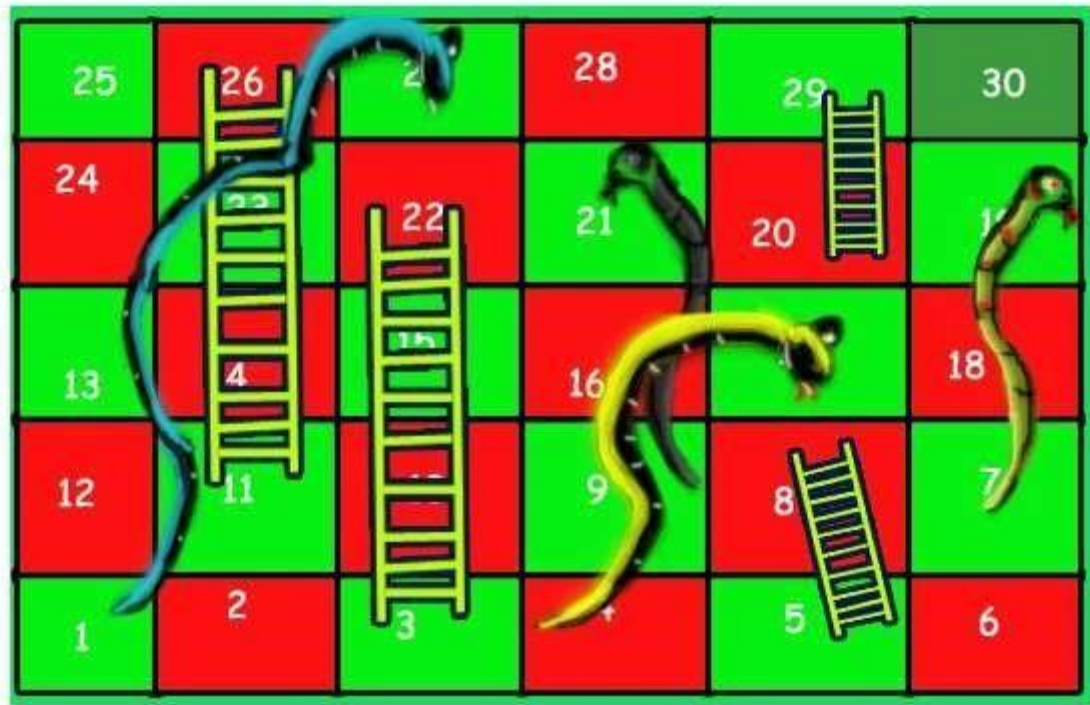


Fig 1. Snake and ladder using BFS

1.1 Problem Statement

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell. If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.

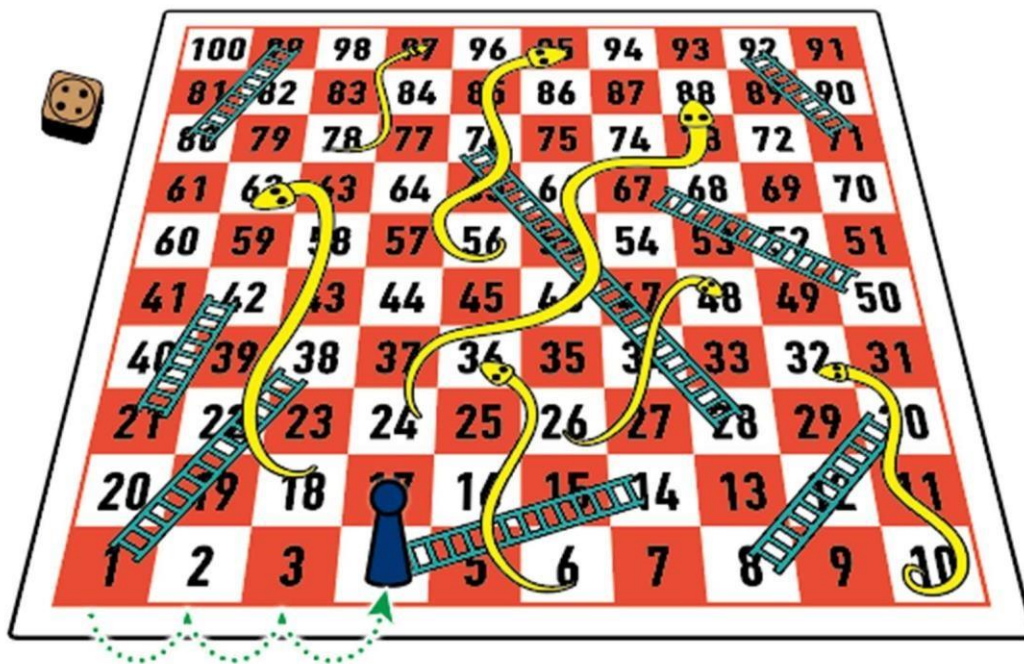
Chapter 2

This chapter will discuss the approach we will be using to solve the game and various ways in which we can get minimum moves using BFS.

2.1 Definition of a Snake and Ladder Game

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph.

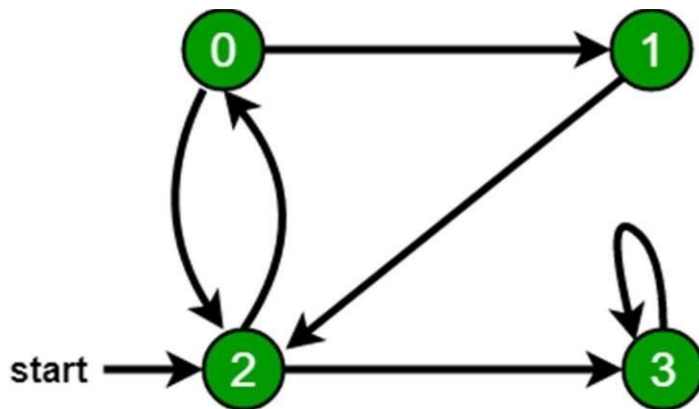
Following is the implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0...N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.



2.2 The Breadth first search Algorithm

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



Approach:

- Consider each as a vertex in [directed graph](#).
- From cell 1 you can go to cells 2, 3, 4, 5, 6, 7 so vertex 1 will have directed edge towards vertex 2, vertex 3....vertex 7. Similarly consider this for rest of the cells.
- For snake- connect directed edge from head of snake vertex to tail of snake vertex. (See example image above- snake from 12 to 2. So directed edge from vertex 12 to vertex 2)
- For ladder- connect directed edge from bottom of ladder vertex to top of the ladder vertex.
- Now problem is reduced to Shorted path problem. So by [Breadth-First Search](#) (using queue) we can solve the problem.

Implementation:

- Each vertex will store 2 information, cell number and number of moves required to reach to that cell. (cell, moves)
- Start from cell (vertex) 1, add it to the queue.
- For any index = i, Remove vertex 'i' from queue and add all the vertices to which can be reached from vertex 'i' by throwing the dice once and update the moves for each vertex (moves = moves to reach cell 'i' + 1 if no snake or ladder is present else moves = cell to which snake or ladder will leads to)
- Remove a vertex from queue and follow the previous step.
- Maintain visited[] array to avoid going in loops. Once reach to the end(destination vertex), stop

2.3 ALGORITHM

Begin initially mark all cell
as unvisited define queue q
mark the starting vertex as
visited

for starting vertex the vertex number := 0 and distance
:= 0 add starting vertex s into q while q is not empty,
do qVert := front element of the queue v := vertex
number of qVert if v = cell -1, then //when it is last
vertex break the loop delete one item from queue
for j := v + 1, to v + 6 and j < cell, increase j by
1, do if j is not visited, then
newVert.dist := (qVert.dist +
1) mark v as visited if there is
snake or ladder, then
newVert.vert := move[j] //jump to that location
else newVert.vert := j insert
newVert into queue done done return
qVert.dist
End

Chapter 3

This chapter covers the time complexity analysis for the approach used above.

3.1 Time Complexity Analysis

Time complexity of the above solution is $O(N)$ as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes $O(1)$ time.

Code:

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include <iostream>
#include <queue> using
namespace std;
// An entry in queue used in BFS struct queueEntry
{
    int v;
// Vertex number
    int dist;
// Distance of this vertex from source
};
// This function returns minimum number of dice throws required to //
Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i // takes to. int
getMinDiceThrows(int move[], int N)
{
// The graph has N vertices. Mark all the vertices as
// not visited
bool *visited = new bool[N];
for (int i = 0; i < N; i++)
    visited[i] = false; // Create a
    queue for BFS
    queue<queueEntry>q;
// Mark the node 0 as visited and enqueue it.
    visited[0] = true; queueEntry s = {0, 0}; //
    distance of 0't vertex is also 0 q.push(s);
// Enqueue 0'th vertex
// Do a BFS starting from vertex at index 0 queueEntry
    qe;
// A queue entry (qe) while
    (!q.empty())
```

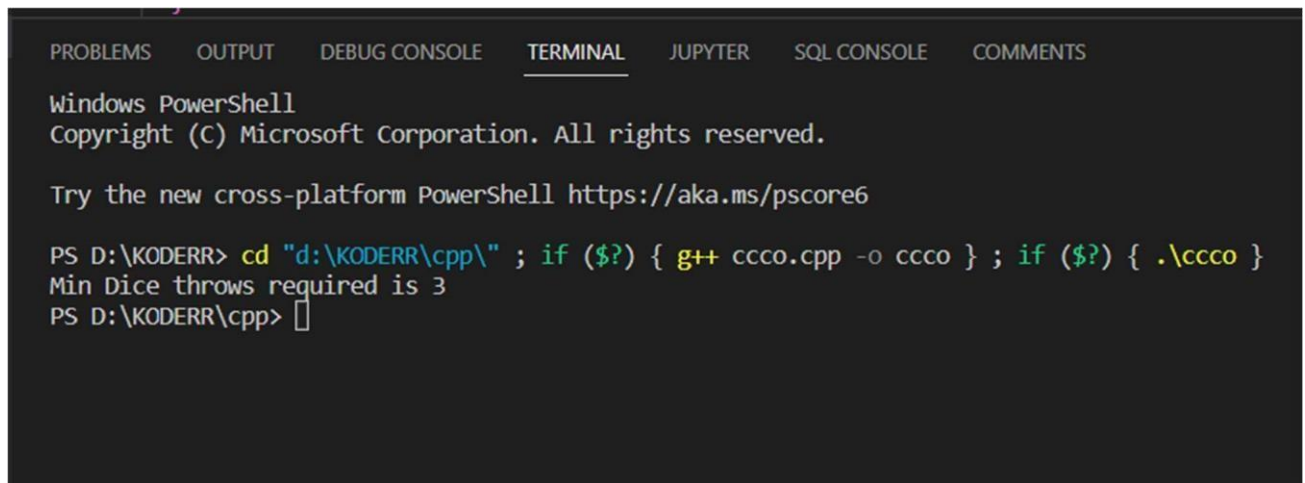
```

    { qe =
    q.front(); int
    v = qe.v;
    // vertex no. of queue entry
    // If front vertex is the destination vertex,
    // we are done if (v
    == N-1) break;
    // Otherwise dequeue the front vertex and enqueue
    // its adjacent vertices (or cell numbers reachable
    // through a dice throw) q.pop(); for (int j=v+1;
    j<=(v+6) && j<N; ++j)
    {
    // If this cell is already visited, then ignore if (!visited[j])
    {
    // Otherwise calculate its distance and mark it as visited queueEntry
    a;
    a.dist = (qe.dist + 1); visited[j]
    = true;
    // Check if there a snake or ladder at 'j'
    // then tail of snake or top of ladder
    // become the adjacent of 'i' if
    (move[j] != -1) a.v = move[j];
    else a.v = j; q.push(a);
    }
    }
    }
    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe' return
    qe.dist;
    }
    // Driver program to test methods of graph
    class int main()
    {
    // Let us construct the board given in above diagram
    int N = 30; int
    moves[N];
    for (int i = 0; i<N; i++) moves[i] = -1;

```

```
// Ladders moves[2] = 21; moves[4] = 7; moves[10] = 25; moves[19] = 28; //  
Snakes moves[26] = 0; moves[20] = 8; moves[16] = 3; moves[18] = 6; cout  
<< "Min Dice throws required is " << getMinDiceThrows(moves, N); return  
0;  
}
```

Output:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER SQL CONSOLE COMMENTS  
Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Try the new cross-platform PowerShell https://aka.ms/pscore6  
  
PS D:\KODERR> cd "d:\KODERR\cpp\" ; if ($?) { g++ ccco.cpp -o ccco } ; if ($?) { .\ccco }  
Min Dice throws required is 3  
PS D:\KODERR\cpp> 
```

Conclusion

We have implemented Breadth first Search algorithm to find the minimum steps to win the Snake and ladder game.

Reference

- <https://www.geeksforgeeks.org/snake-ladder-problem-2/>
- <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

RAT IN A MAZE

A MINOR PROJECT REPORT

Submitted by

Reddy jyothi sri D (RA2011028010108)
Jayanth Sai Yarlagaadda (RA2011028010054)

Faculty Name: Ms Shobana

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

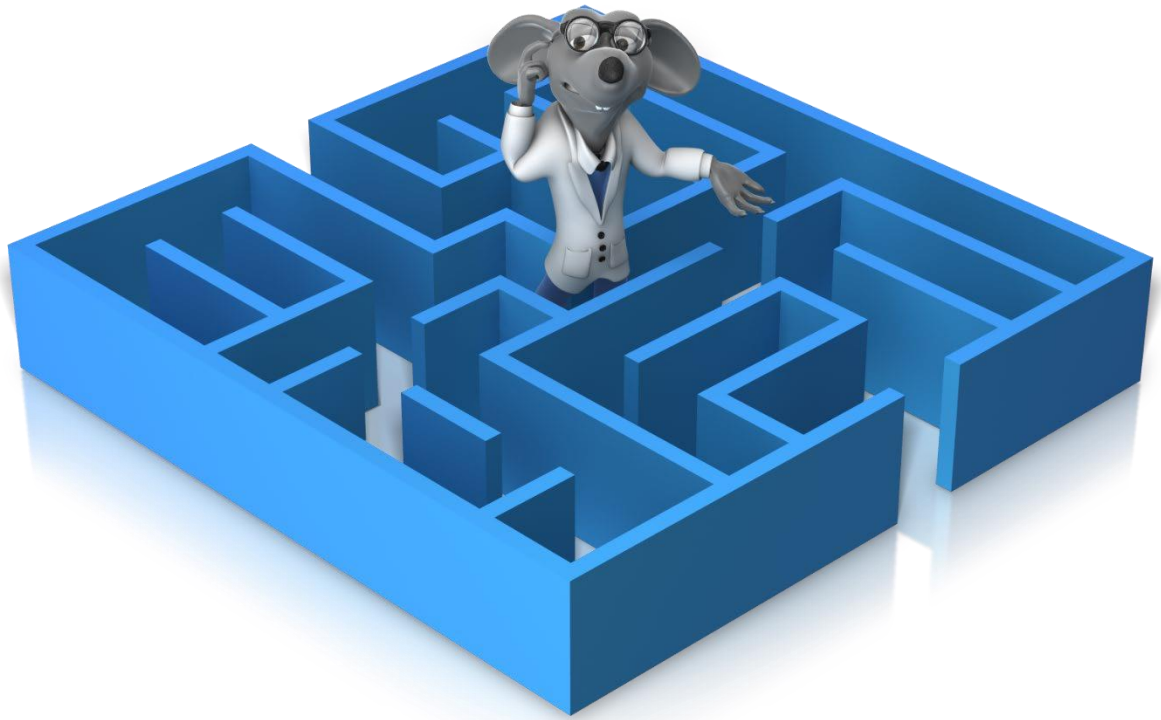


S.R.M. Nagar, Kattankulathur, Kancheepuram District

June,2022

DAA PROJECT

Rat in a Maze



Contents

PROBLEM DEFINITION: 4

**PROBLEM EXPLANATION WITH DIAGRAM AND
EXAMPLE: 4**

ALGORITHM:..... Error! Bookmark not defined.

DESIGN TECHNIQUE:.....9

COMPLEXITY ANALYSIS:.....10

CONCLUSION:.....10

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 $N \times N$ 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).

Source			
			Dest.

Following is a binary matrix representation of the above maze.

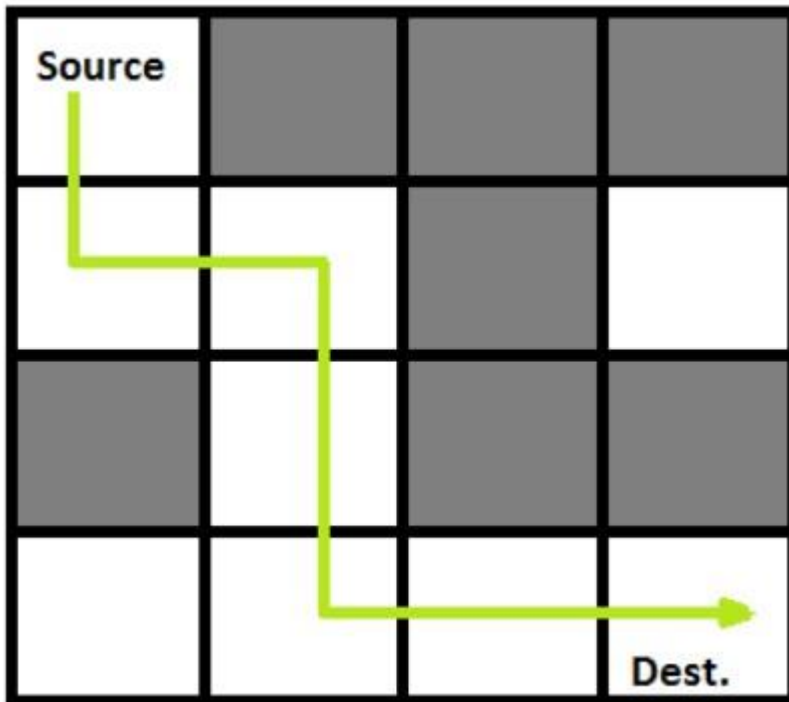
{1, 0, 0, 0}

{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 reaching any 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:

```
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);  
    }  
}
```

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.