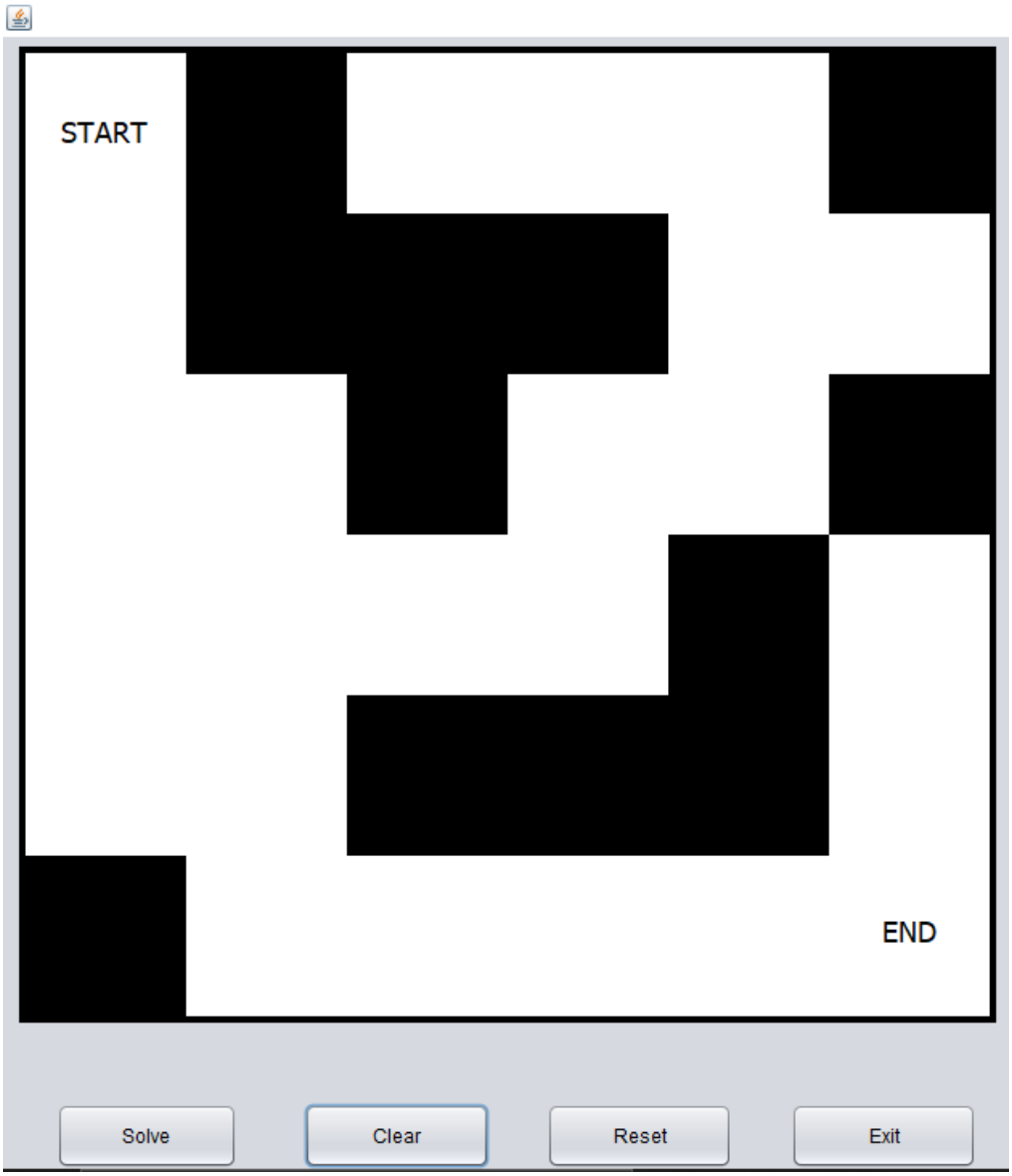
Problem definition:

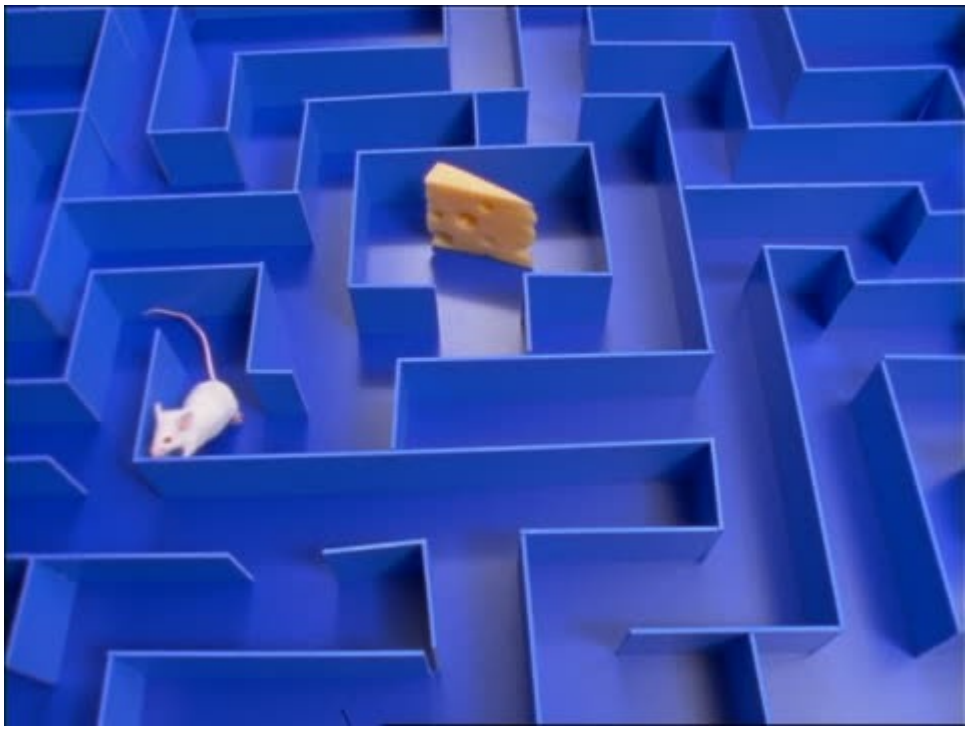
The rat must go from its origin to its goal. Only forward and downward movement is possible for the rat. In the labyrinth matrix, a 0 indicates that the block is a dead end, while a value of 1 indicates that the block is usable on the route from source to destination. The goal of this issue is to determine all of the different routes the rat may take to reach the destination cell from the specified source cell, assuming that the rat will start in a certain cell.



Problem Explanation with diagram and example

The location of the start is in the top-left cell, and the location of the destination is in the bottom-right cell. There are certain cells that can be moved, but there are also other cells that cannot be moved. If the rats begins from the starting vertex to the ending vertex, we need to determine whether or not there is any possibility for the route to be finished, and if so, we need to

designate the appropriate path for the rat to take. The instructions for the labyrinth are presented in the form of a binary matrix. If a cell has the value 1 assigned to it, then that cell represents a legitimate route; otherwise, it has the value 0. A labyrinth has the shape of a 2D in which some cells or blocks are obscured. It is necessary for us to locate a way to go from the origin to the destination without entering any of the cells that are closed off. The image that can be seen below depicts an unsolved labyrinth. The cells that are represented in grey represent dead ends, while the cells that are shown in white represent cells that can be reached.



Design Technique Used:

Backtracking is a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as

it determines that the candidate cannot possibly be completed to a valid solution.
The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.
Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute-force enumeration of all complete candidates, since it can eliminate many candidates with a single test.
Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing,[3] for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.
Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates. It is therefore a metaheuristic rather than a specific algorithm – although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.

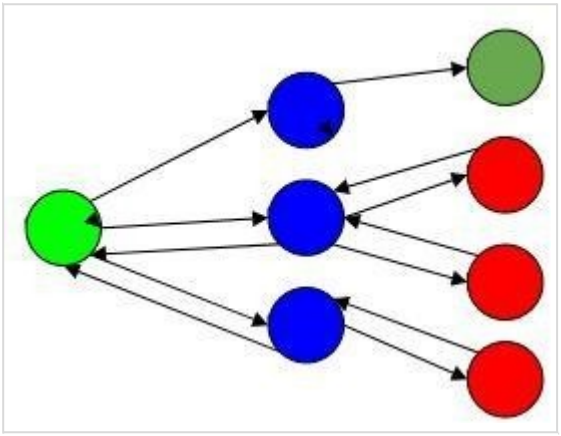
The term "backtrack" was coined by American mathematician

D. H. Lehmer in the 1950s.[4] The pioneer string-processing language SNOBOL (1962) may have been the first to provide a built-in general backtracking facility.The backtracking technique is used to solve this problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don't give rise to the problem's solution based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem is used to find a feasible solution of the problem.
- Optimization problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.
Example,



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is the end solution.
Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find the next point to find solution.

Algorithm:

Step 1 − if current_position is goal, return
success Step 2 − else,
Step 3 − if current_position is an end point, return failed.
Step 4 − else, if current_position is not the end point, explore and repeat above steps.

Explanation:

Create a solution matrix, initially filled with 0's.

Create a recursive function, which takes initial matrix, output matrix and position of rat(i, j).
if the position is out of the matrix or the position is not valid then return. 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.
Recursively call for position (i-1,j), (I,j-1), (i+1, j) and (i, j+1). Unmark position (i, j), i.e output[i][j] = 0

Complexity Analysis:

Time Complexity: O(2^(n^2)).

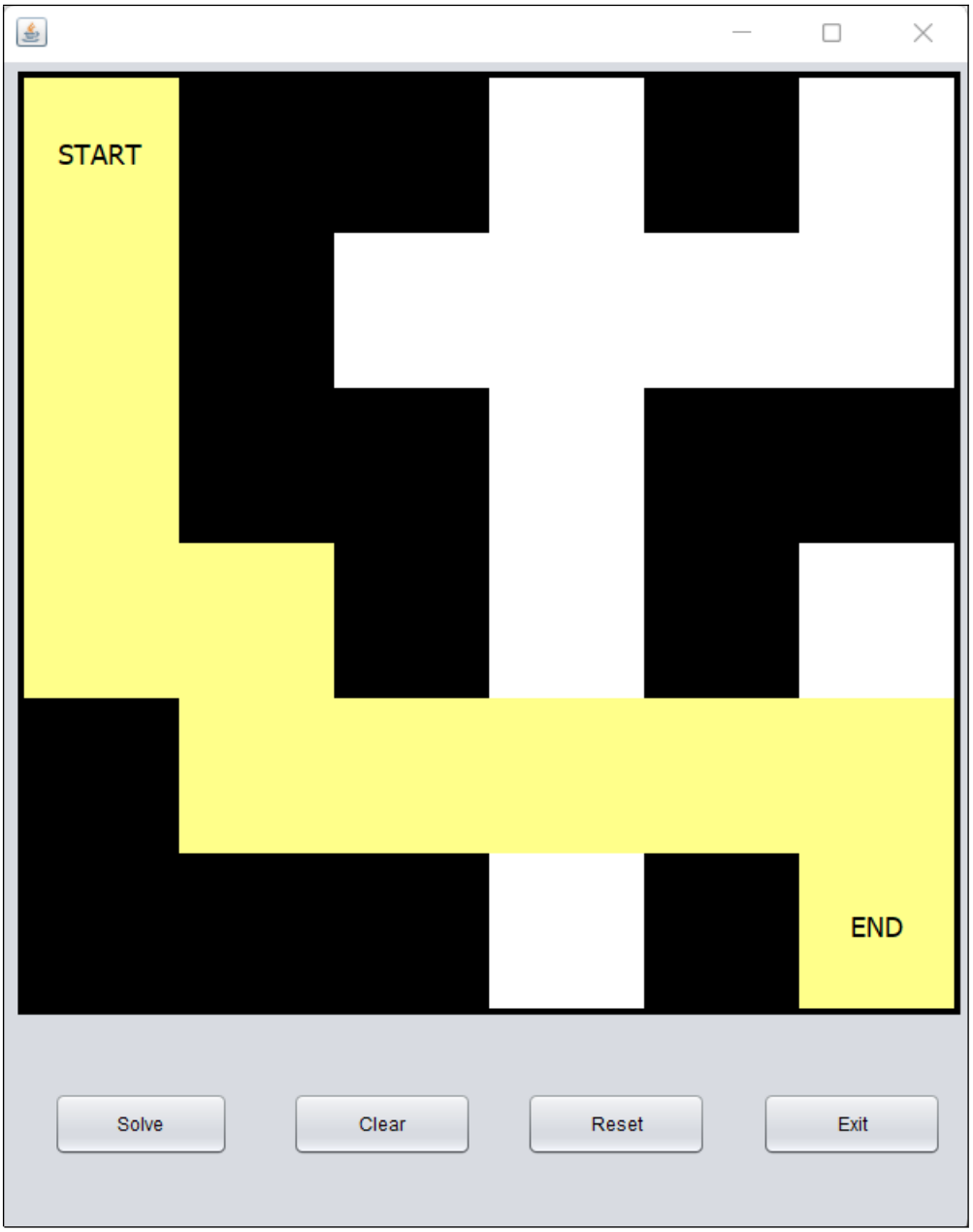
The recursion can run upper-bound
2^(n^2) times.

Space Complexity: O(n^2).

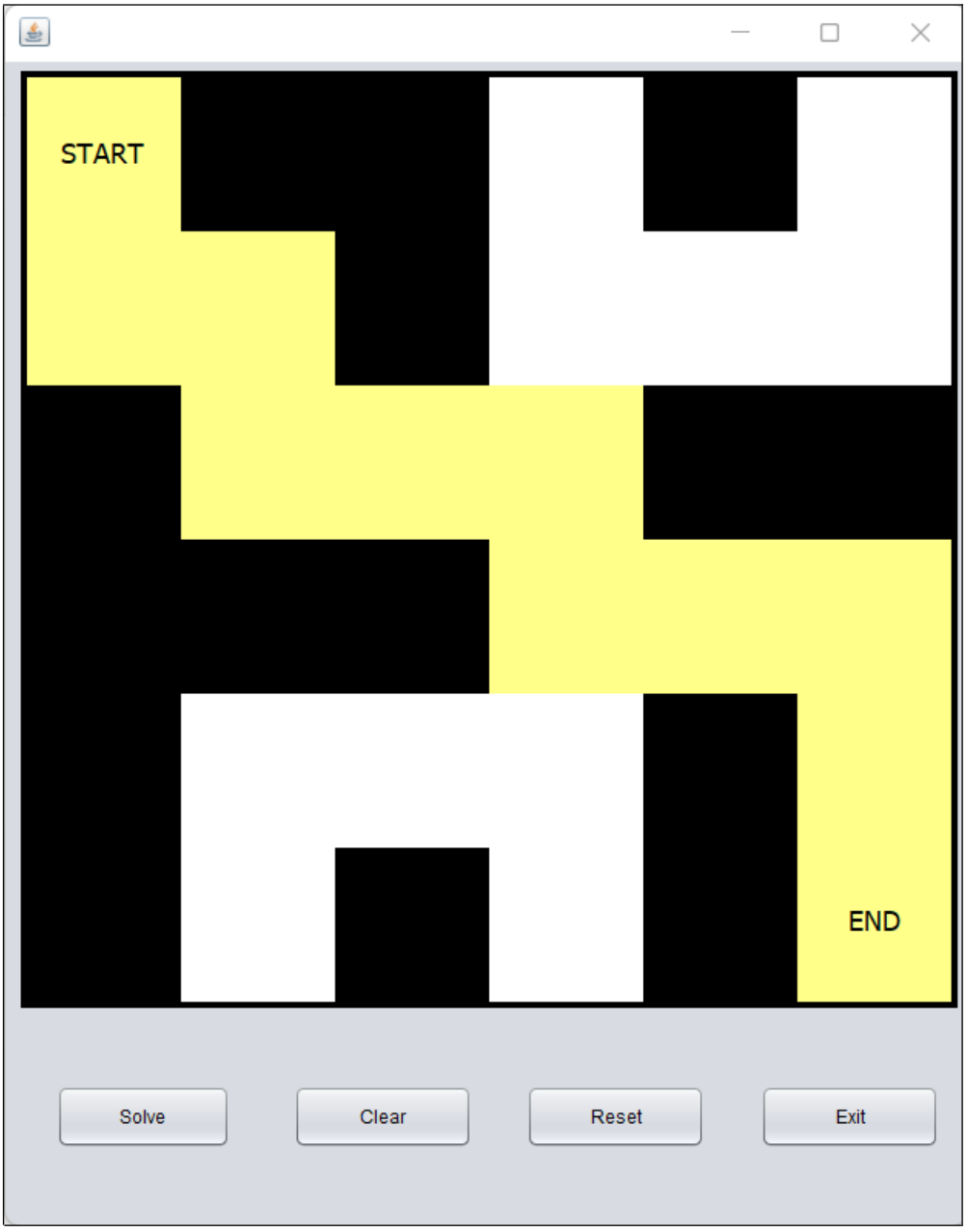
Output matrix is required so an extra space of size n*n is needed. Considering a grid of size MxN, and rat has to move from 0,0 to M-1,N-1 (some cells may be blocked and rat can move either down or right) Now at every point rat has two choices, go down or go left and to reach destination, rat has to make (M+N) moves, so total recursion complexity
turns out as O(2^(M+N)) which is very high, but there are many overlapping recursive calls, so using DP reduces the complexity to O(MN).

GUI for Rat In a Maze:

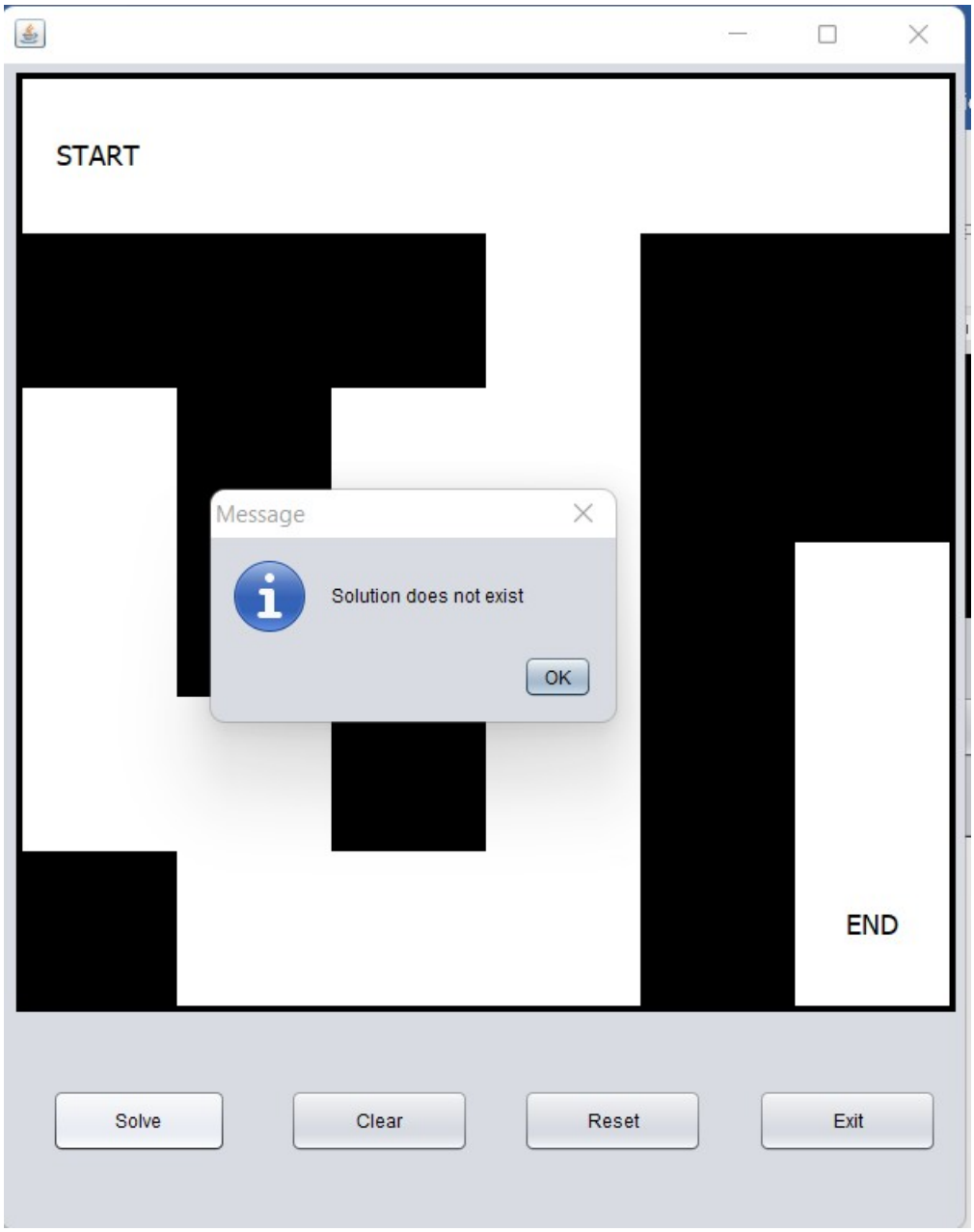
1.



2.



3.


START
Message
Solution does not exist
OK
END
Solve
Clear
Reset
Exit

CODE Illustration:

- Logic for backtracking along with initialization of Initial 6X6 matrix

```
 */
public class Maze6X6 extends javax.swing.JFrame {

        Panel[][] panelArray = new Panel[6][6];
        Color bl = new Color(0, 0, 0);
        Color yl=new Color(255,255,132);
        final int N = 6;
    /**
     * Creates new form Maze6X6
     */
    public Maze6X6() {
        initComponents();
        Arr();


    }
//{ start of maze solution

    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);
    }

    int[][] 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},
        };
```

- To check if the adjacent matrix cell is empty

```
        if (solveMazeUtil(maze, 0, 0, sol) == false)
        {
            JOptionPane.showMessageDialog(this,"Solution does not exist");
//
//              System.out.print("Solution doesn't exist");
            return sol;
            //return false;
        }

//          printSolution(sol);
        return sol;
    }

    /* 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, y + 1, sol))
                return true;



            sol[x][y] = 0;
            return false;
        }

        return false;
    }
```

- Code to change the background colour of Jpanel in java which determines whether a particular cell is blocked or open for movement.

```
public void state(Panel p)
{
   Color bg=p.getBackground();
   if(bg.equals(bl))
    p.setBackground(new java.awt.Color(255,255,255));
    else
    p.setBackground(new java.awt.Color(0,0,0));
}
```

```
private void panel1MouseClicked(java.awt.event.MouseEvent evt) {
    state(panel1);
}
```

- Solution and colouring of optimal path in the Jpanel matrix

```
private void SolveActionPerformed(java.awt.event.ActionEvent evt) {

Maze6X6 rat = new Maze6X6();
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 },
                 {1,1,0,1,0,1 },
                 {0,1,1,1,1,1 },
               };
for(int i=0;i<6;i++)
{
    for(int j=0;j<6;j++)
    {
        Color bg=panelArray[i][j].getBackground();
        if(bg.equals(bl))
            maze[i][j]=0;
        else
            maze[i][j]=1;
    }
}
int sol[][]=rat.solveMaze(maze);

for(int i=0;i<6;i++)
    {
        for(int j=0;j<6;j++)
        {
            if(sol[i][j]==1)
                panelArray[i][j].setBackground(new java.awt.Color(255,255,132));
        }
    }
}
```

- Clear and reset button action statements

```
    private void Clear1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(0);         // TODO add your handling code here:
    }

    private void Clear2ActionPerformed(java.awt.event.ActionEvent evt) {
    for(int i=0;i<6;i++)
    {
        for(int j=0;j<6;j++)
        {
            Color gc=panelArray[i][j].getBackground();
            if(gc.equals(yl))
                panelArray[i][j].setBackground(new java.awt.Color(255,255,255));
        }
    }
        //clearing the yellow part


        // TODO add your handling code here:
    }
```

- Initialising all 36 Jpanels and adding its properties to a 6X6 matrix.

```
public void Arr()
{
        panelArray[0][0]=panel1;
        panelArray[0][1]=panel2;
        panelArray[0][2]=panel3;
        panelArray[0][3]=panel4;
        panelArray[0][4]=panel5;
        panelArray[0][5]=panel6;

        panelArray[1][0]=panel7;
        panelArray[1][1]=panel8;
        panelArray[1][2]=panel9;
        panelArray[1][3]=panel10;
        panelArray[1][4]=panel11;
        panelArray[1][5]=panel12;

        panelArray[2][0]=panel13;
        panelArray[2][1]=panel14;
        panelArray[2][2]=panel15;
        panelArray[2][3]=panel16;
        panelArray[2][4]=panel17;
        panelArray[2][5]=panel18;

        panelArray[3][0]=panel19;
        panelArray[3][1]=panel20;
        panelArray[3][2]=panel21;
        panelArray[3][3]=panel22;
        panelArray[3][4]=panel23;
        panelArray[3][5]=panel24;

        panelArray[4][0]=panel25;
        panelArray[4][1]=panel26;
        panelArray[4][2]=panel27;
        panelArray[4][3]=panel28;
        panelArray[4][4]=panel29;
        panelArray[4][5]=panel30;

        panelArray[5][0]=panel31;
        panelArray[5][1]=panel32;
        panelArray[5][2]=panel33;
        panelArray[5][3]=panel34;
        panelArray[5][4]=panel35;
        panelArray[5][5]=panel36;

}
```

- Component Initialisation Code

```
private void initComponents() {

    Solve = new javax.swing.JButton();
    jPanel1 = new javax.swing.JPanel();
    panel1 = new java.awt.Panel();
    START = new javax.swing.JLabel();
    panel2 = new java.awt.Panel();
    panel3 = new java.awt.Panel();
    panel6 = new java.awt.Panel();
    panel4 = new java.awt.Panel();
    panel7 = new java.awt.Panel();
    panel11 = new java.awt.Panel();
    panel8 = new java.awt.Panel();
    panel9 = new java.awt.Panel();
    panel10 = new java.awt.Panel();
    panel13 = new java.awt.Panel();
    panel5 = new java.awt.Panel();
    panel19 = new java.awt.Panel();
    panel25 = new java.awt.Panel();
    panel31 = new java.awt.Panel();
    panel14 = new java.awt.Panel();
    panel20 = new java.awt.Panel();
    panel26 = new java.awt.Panel();
    panel32 = new java.awt.Panel();
    panel15 = new java.awt.Panel();
    panel21 = new java.awt.Panel();
    panel27 = new java.awt.Panel();
    panel33 = new java.awt.Panel();
    panel16 = new java.awt.Panel();
    panel22 = new java.awt.Panel();
    panel28 = new java.awt.Panel();
    panel34 = new java.awt.Panel();
    panel17 = new java.awt.Panel();
    panel23 = new java.awt.Panel();
    panel29 = new java.awt.Panel();
    panel12 = new java.awt.Panel();
    panel18 = new java.awt.Panel();
    panel35 = new java.awt.Panel();
    panel24 = new java.awt.Panel();
    panel30 = new java.awt.Panel();
    panel36 = new java.awt.Panel();
    jLabel1 = new javax.swing.JLabel();
    Clear = new javax.swing.JButton();
    Clear1 = new javax.swing.JButton();
    Clear2 = new javax.swing.JButton();
```

- Setting the alignment properties of different GUI swing widgets.

```
javax.swing.GroupLayout panel2Layout = new javax.swing.GroupLayout(panel2);
panel2.setLayout(panel2Layout);
panel2Layout.setHorizontalGroup(
    panel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGap(0, 100, Short.MAX_VALUE)
);
panel2Layout.setVerticalGroup(
    panel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGap(0, 100, Short.MAX_VALUE)
);

panel3.setBackground(new java.awt.Color(255, 255, 255));
panel3.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent evt) {
        panel3MouseClicked(evt);
    }
});
```

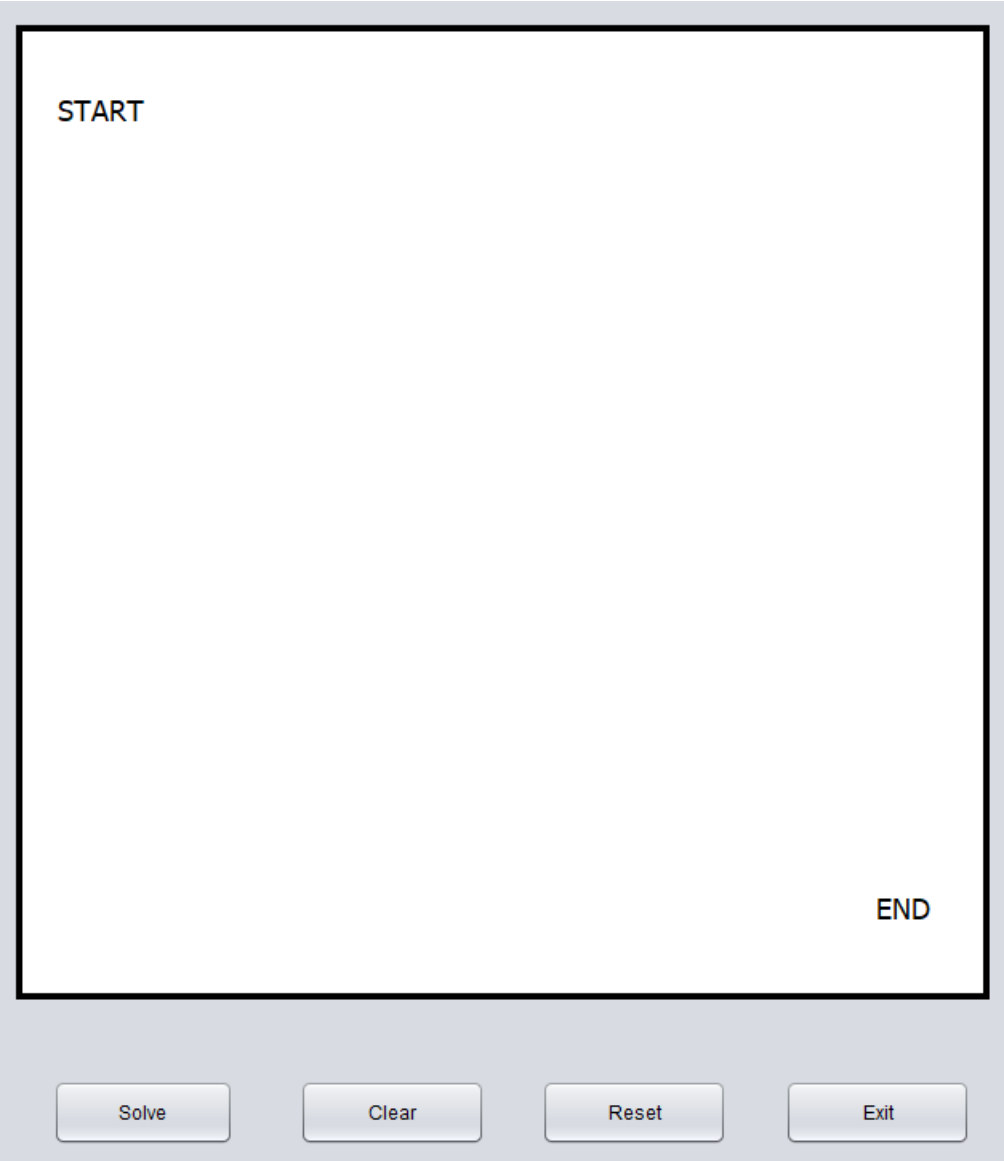
- Adding event Listeners to Different Action Buttons.

```
Clear.setText("Reset");
Clear.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        ClearActionPerformed(evt);
    }
});

Clear1.setText("Exit");
Clear1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        Clear1ActionPerformed(evt);
    }
});

Clear2.setText("Clear");
Clear2.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        Clear2ActionPerformed(evt);
    }
});
```

- Final Output Screen



Algorithm Explanation :

Input Array

```
int maze[][] = { { 1,0,0,1,0,1 },
                 { 1,1,0,1,1,1 },
                 { 0,1,1,1,0,0 },
```

```
{ 0,1,0,1,1,1 },
{ 0,1,1,1,0,1 },
{ 0,1,0,1,0,1 },
};
```

From the above array we will starting at index maze[0][0] and we will be reaching to index[6][6]

In the above array 1 is represented as free space and 0 is represented as occupied space

Now from index[0][0] we go to index[1][0], we can't remain on index[0][0] so we will be going to index[1][0]. From index[1][0] we can go to index[1][1], from there we go to index[2][1], from where we have two options but we can't go to index[2][0] or index[3][1], but index[3][1] is not explored as out car would be blocked in future moves, so we go to index and traverse the rest of the remaining matrix in a similar manner and reach the final index[6][6] position and the path used in highlighted using yellow colour.

Conclusion:

The problem of solving the maze by the rat to reach the cheeze in the end is solved using the backtracking method and an example is provided with the algorithm and the user interface. This method can be used to sove mazes in a highly effective manner without making the state space tree of all the possible outcomes.

References:

- Java-Oracle
- Swing Components
- AWT components
- Backtracking
- Rat in a maze