RNS INSTITUTE OF TECHNOLOGY

BENGALURU - 98

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

DAA Mini Project Work Presentation

SUDOKU

Candidate Names

USN 1: 1RN20IS159 (SIDDHARTH KESARWANI)
USN 2: 1RN20IS160 (SIDHARTH S PAI)
USN 3: 1RN20IS168 (SURABHI S)
USN 4: 1RN20IS184 (VIPUL GAURAV)

Faculty In-charge
Dr. Suresh L
H.O.D
Dept of ISE, RNSIT

AGENDA

- Abstract
- Introduction
- Algorithm Design Technique Used
- Algorithm
- System Requirements
- Implementation / Code
- Discussion of Results-Sample input and Output
- Time Complexity: Analysis of algorithm with other related algorithms
- Real time Applications
- Conclusion and Future Enhancements

ABSTRACT

- Sudoku is a logic-based number based placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 boxes (also called blocks or regions) contains the digits from 1 to 9, only one time each (that is, exclusively). The puzzle setter provides a partially completed grid.
- Many computational methods had been developed in many ways. In this project we acquire the final stage or solution for Sudoku by using BACKTRACKING Algorithm design technique.

INTRODUCTION

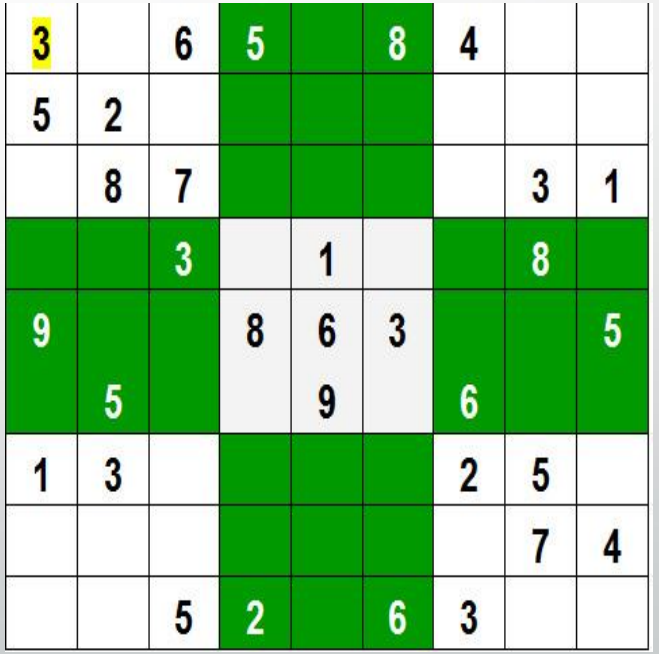
- **Strategies of Sudoku:**

The strategy for solving a puzzle may be regarded as comprising a combination of three processes: scanning, marking up, and analysing. The approach to analysis m ay vary according to the concepts and the representations on which it is based.

- **Scanning Strategy:**

As shown in the figure,

the top right box must contain a 5 and the slot available for the same in the block must be free of any other '5' horizontally or vertically in line with it.



- **Marking up Strategy:**

Scanning stops when no further numerals can be discovered, making it necessary to engage in local analysis. One method to guide the analysis is to mark candidate numerals in the blank analysis.

A contingency when created, a line is checked for numerals for 1-9 and we can determine the missing number although unresolved.

- **Analysing Strategy:**

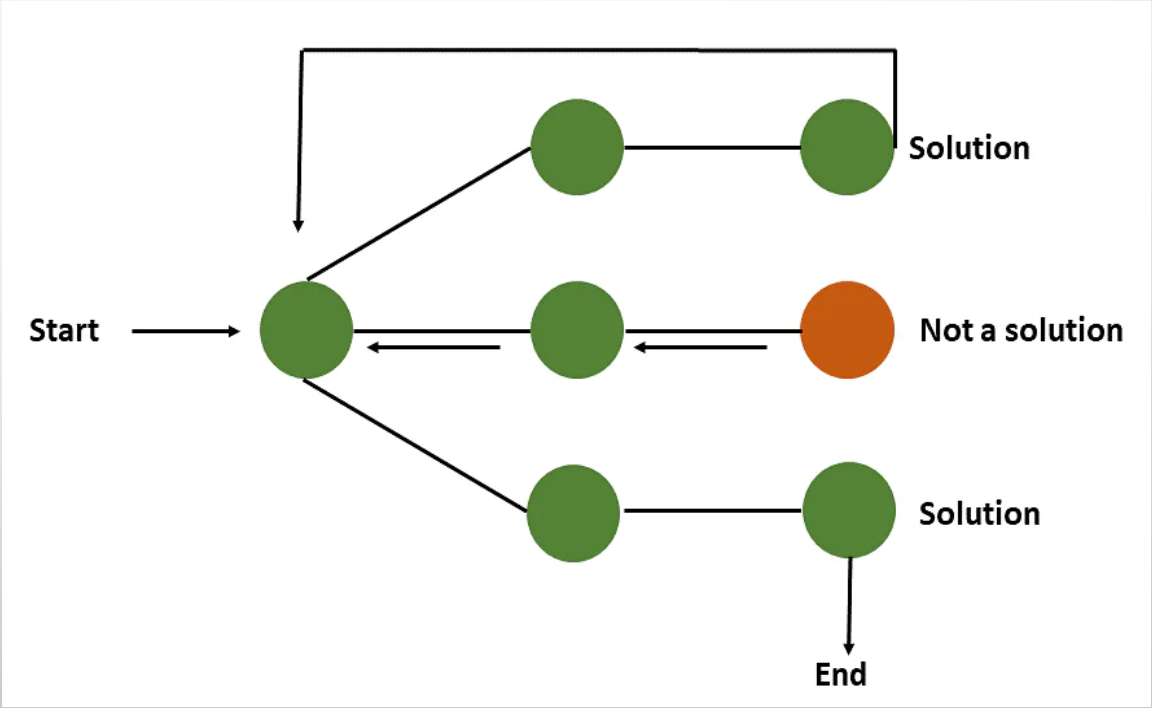
The two main approaches to analysis are "Candidate Elimination"

and "What-if".

ALGORITHM DESIGN TECHNIQUE USED

- BACKTRACKING is a technique based on algorithm to solve SUDOKU 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 doesn't give rise to the solution of the problem based on the constraints given to solve the problem

State Space Tree in a BACKTRACKING Algorithm

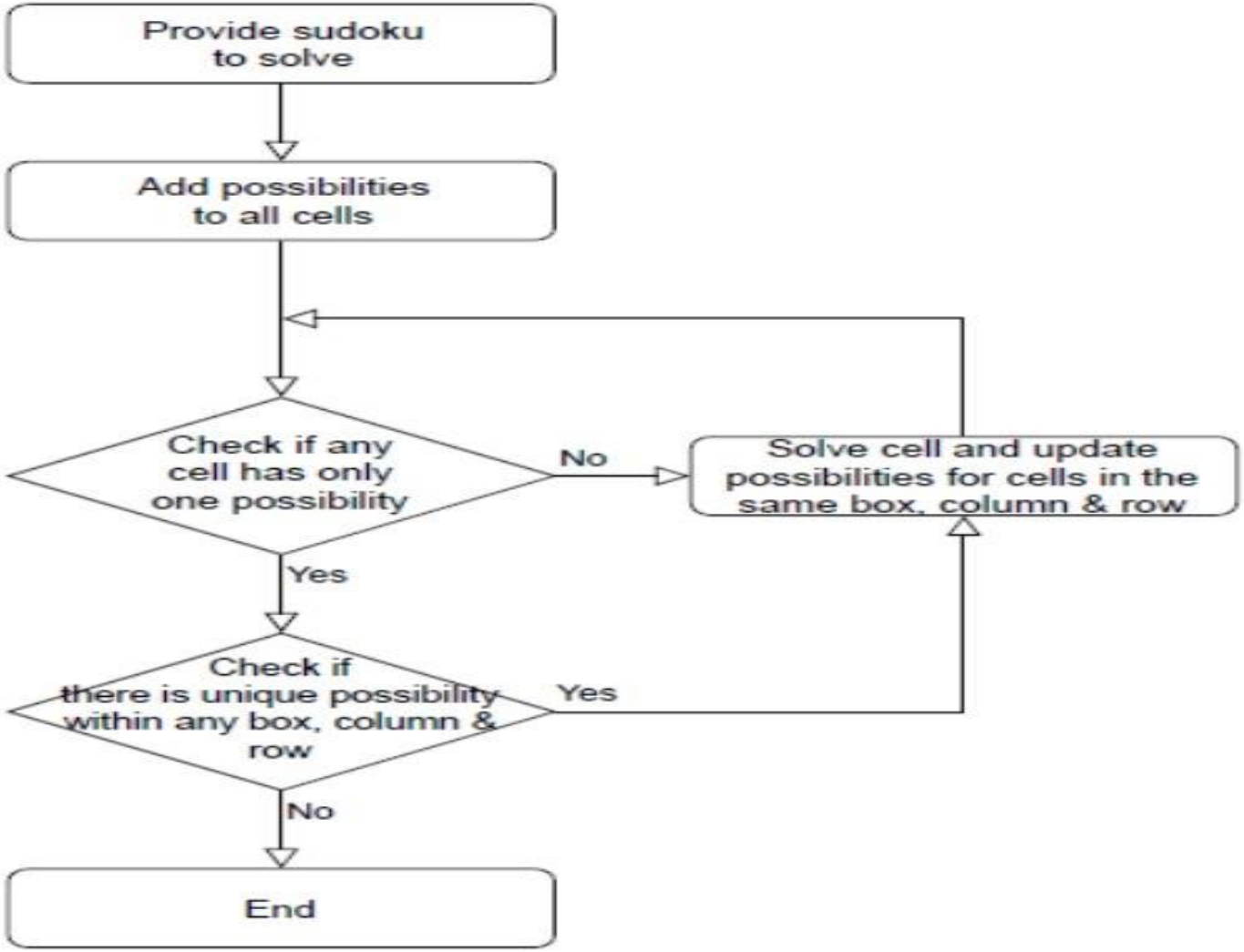


ALGORITHM

sudokuSolver(grid)

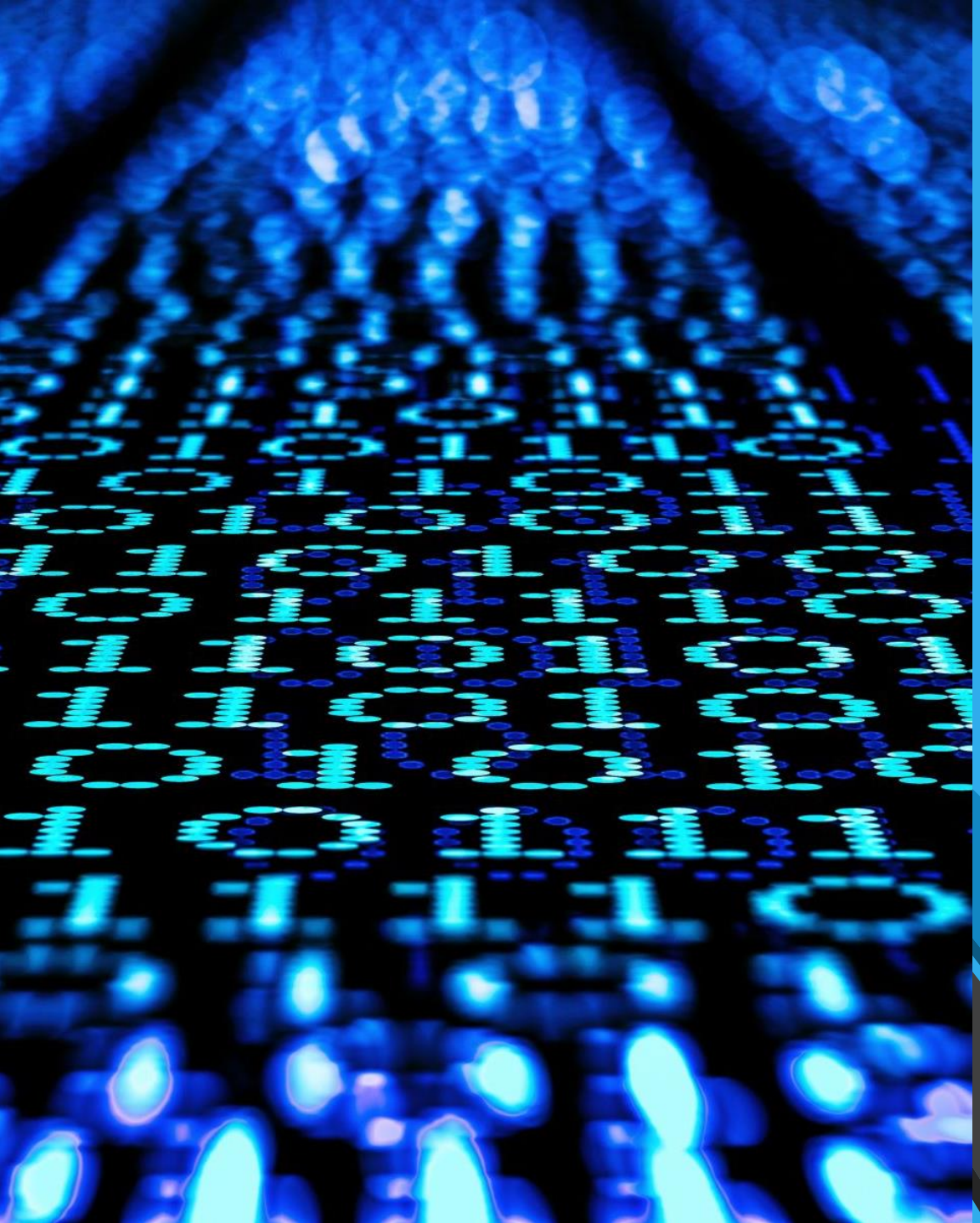
1. Find an unfilled cell *(i, j)* in *grid*
2. If all the cells are filled then
3. A valid sudoku is obtained hence ***return true***
4. For each ***num*** in 1 to 9
5. If the cell *(i, j)* can be filled with ***num*** then fill it with num temporarily to check
6. If *sudokuSolver(grid)* is true then return true
7. If the cell *(i, j)* can't be filled with ***num*** the mark it as unfilled to trigger backtracking
8. If none of the numbers from 1 to 9 can be filled in cell *(i, j)* then ***return false*** as there is no solution for this sudoku

FLOWCHART:


Provide sudoku to solve
Add possibilities to all cells
Check if any cell has only one possibility
No
Solve cell and update possibilities for cells in the same box, column & row
Yes
Check if there is unique possibility within any box, column & row
Yes
No
End

SYSTEM REQUIREMENTS

- **Hardware Requirements** • Processor: Intel Core2 Quad @ 2.4Ghz on Windows® Vista 64-Bit / Windows® 7 64-Bit / Windows® 8 64-Bit / Windows® 8.1 64-Bit. • RAM: 2GB of RAM • Memory: 256GB Hard drive • Keyboard: MS compatible keyboard • Mouse: MS compatible mouse
- **Software Requirements** • Operating system: Windows® Vista 64-Bit / Windows® 7 64-Bit / Windows® 8 64-Bit / Windows® 8.1 64-Bit. •Programming language : Java • IDE: Eclipse



IMPLEMENTATION/ CODE

```java
package sudoku;
public class GFG
{
    public static boolean isSafe(int[][] board,
                                 int row, int col,
                                 int num)
    {

        for (int d = 0; d < board.length; d++)
        {

            if (board[row][d] == num) {
                return false;
            }
        }


        for (int r = 0; r < board.length; r++)
        {

            if (board[r][col] == num)
            {
                return false;
            }
        }

        int sqrt = (int)Math.sqrt(board.length);
        int boxRowStart = row - row % sqrt;
        int boxColStart = col - col % sqrt;

        for (int r = boxRowStart;
             r < boxRowStart + sqrt; r++)
        {
            for (int d = boxColStart;
                 d < boxColStart + sqrt; d++)
            {
                if (board[r][d] == num)
                {
```


```
39                      return false;
40                  }
41              }
42          }
43
44
45          return true;
46      }
47
48      public static boolean solveSudoku(
49          int[][] board, int n)
50      {
51          int row = -1;
52          int col = -1;
53          boolean isEmpty = true;
54          for (int i = 0; i < n; i++)
55          {
56              for (int j = 0; j < n; j++)
57              {
58                  if (board[i][j] == 0)
59                  {
60                      row = i;
61                      col = j;
62
63
64                      isEmpty = false;
65                      break;
66                  }
67              }
68              if (!isEmpty) {
69                  break;
70              }
71          }
```

```
74          if (isEmpty)
75          {
76              return true;
77          }
78
79
80          for (int num = 1; num <= n; num++)
81          {
82              if (isSafe(board, row, col, num))
83              {
84                  board[row][col] = num;
85                  if (solveSudoku(board, n))
86                  {
87
88                      return true;
89                  }
90                  else
91                  {
92
93                      board[row][col] = 0;
94                  }
95              }
96          }
97          return false;
98      }
99
100     public static void print(
101         int[][] board, int N)
102     {
103
104         for (int r = 0; r < N; r++)
105         {
106             for (int d = 0; d < N; d++)
107             {
108                 System.out.print(board[r][d]);
109                 System.out.print(" ");
110             }
111             System.out.print("\n");
```

```
113                 if ((r + 1) % (int)Math.sqrt(N) == 0)
114                 {
115                     System.out.print("");
116                 }
117             }
118         }
119
120         public static void main(String args[])
121         {
122
123             int[][] board = new int[][] {
124                 { 3, 0, 6, 5, 0, 8, 4, 0, 0 },
125                 { 5, 2, 0, 0, 0, 0, 0, 0, 0 },
126                 { 0, 8, 7, 0, 0, 0, 0, 3, 1 },
127                 { 0, 0, 3, 0, 1, 0, 0, 8, 0 },
128                 { 9, 0, 0, 8, 6, 3, 0, 0, 5 },
129                 { 0, 5, 0, 0, 9, 0, 6, 0, 0 },
130                 { 1, 3, 0, 0, 0, 0, 2, 5, 0 },
131                 { 0, 0, 0, 0, 0, 0, 0, 7, 4 },
132                 { 0, 0, 5, 2, 0, 6, 3, 0, 0 }
133             };
134             int N = board.length;
135
136             if (solveSudoku(board, N))
137             {
138                 print(board, N);
139             }
140             else {
141                 System.out.println("No solution");
142             }
143         }
144 }
```

RESULT: SAMPLE INPUT AND OUTPUT

- **INPUT**

```
grid = { {3, 0, 6, 5, 0, 8, 4, 0, 0},
         {5, 2, 0, 0, 0, 0, 0, 0, 0},
         {0, 8, 7, 0, 0, 0, 0, 3, 1},
         {0, 0, 3, 0, 1, 0, 0, 8, 0},
         {9, 0, 0, 8, 6, 3, 0, 0, 5},
         {0, 5, 0, 0, 9, 0, 6, 0, 0},
         {1, 3, 0, 0, 0, 0, 2, 5, 0},
         {0, 0, 0, 0, 0, 0, 0, 7, 4},
         {0, 0, 5, 2, 0, 6, 3, 0, 0} }
```

- **OUTPUT**

```
Output:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

COMPLEXITY ANALYSIS

- Time complexity: O(9^(n*n)).
- For every unassigned index, there are 9 possible options so the time complexity is O(9^(n*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.
- Space Complexity: O(n*n).
- To store the output array a matrix is needed.

REAL TIME APPLICATIONS

- Examples where BACKTRACKING can be used to solve puzzles or problems include:

Eight Queens Puzzle, Crosswords, Verbal Arithmetic, Sudoku and Peg Solitaire.

- Combinational optimization problems such as parsing and the knapsack problem.
- To find all Hamiltonian paths present in a graph.

CONCLUSION

- We started with the concepts, rules and the strategies of Sudoku, thus even if you have never heard about the game, you should get a brief idea in your mind about it now.
- The Sudoku is just a member of puzzles, by introducing to the BACKTRACKING method, we could be able to construct even more difficult puzzles in the early future. Besides the puzzle solving, there are many other areas also applicable for the BACKTRACKING method, and which could be adapted much better than the conventional computational models.

FUTURE ENHANCEMENTS

There are some of the main suggestions given by the user for improvement.

1. Instant help by giving hints and users progress while playing.
2. Score system based on time and accuracy, and database to keep track of the top ten record.

Thank You!