Project Report
Sudoku Solver using Backtracking

Subject Name – Design and Analysis of Algorithms

Subject Code – 23CSH-301

Submitted To:

Er. Mohammad Shaqlain

(E17211)

Submitted By:

Name:Mandeep Kaur

UID: 23bcs10854

Section: KRG_2-B

An implementation of recursive backtracking algorithm to solve 9×9 Sudoku puzzles

1. Aim

To develop and analyze the complexity of a program to solve a Sudoku puzzle using the Backtracking algorithm.

2. Objective

The objective is to implement a Sudoku Solver that fills a 9×9 grid by assigning digits (1–9) such that:

- Each row, column, and 3×3 subgrid contains all digits exactly once.
- The program efficiently backtracks to correct conflicts.

3. Input / Apparatus Used

- **Programming Language:** C++
- **IDE Used:** Code::Blocks / VS Code / Visual Studio
- **Input:** 9×9 grid with 0s as empty cells

4. Procedure / Algorithm

Algorithm: Sudoku Solver (Backtracking)

```
1. Start
2. Find an empty cell (value = 0)
3. Try digits 1 to 9:
      • Check if safe in row, column, and 3×3 subgrid
4. If safe, place and recurse
5. If conflict later, backtrack (remove and try next)
6. Repeat until board is full or no solution
7. Stop
```

Recursive Equation:
Let n = current cell index

$$SolveSudoku(n) = \{ \quad SolveSudoku(n+1) ifvalidplacementbacktrack \& trynextotherwise$$

5. C++ Program

```cpp
#include <bits/stdc++.h>
using namespace std;

#define N 9

bool isSafe(int grid[N][N], int row, int col, int num) {
    for (int x = 0; x < 9; x++)
        if (grid[row][x] == num || grid[x][col] == num)
            return false;

    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + startRow][j + startCol] == num)
                return false;

    return true;
}

bool solveSudoku(int grid[N][N], int row, int col) {
    if (row == N - 1 && col == N)
        return true;

    if (col == N) {
        row++;
        col = 0;
    }

    if (grid[row][col] != 0)
        return solveSudoku(grid, row, col + 1);

    for (int num = 1; num <= 9; num++)
        { if (isSafe(grid, row, col, num))
        {
            grid[row][col] = num;
            if (solveSudoku(grid, row, col + 1))
                return true;
            grid[row][col] = 0;
        }
    }
}   return false;

void printGrid(int grid[N][N]) {
    for (int r = 0; r < N; r++) {
        for (int d = 0; d < N; d++) {
            cout << grid[r][d] << " ";
        }
        cout << endl;
    }
}

int main() {
    int grid[N][N] = {
        {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}
    };

    if (solveSudoku(grid, 0, 0))
        printGrid(grid);
    else
        cout << "No solution exists";
    return 0;
}
```

Listing 1: Sudoku Solver in C++

Sample 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
```

6. Complexity Analysis

- **Worst Case Time:** $O(9^{N\times N})$ (empty grid)
- **Average Case:** Much faster due to pruning
- **Space Complexity:** $O(N \times N)$ (grid + recursion stack)

7. Result

The program successfully solves the given Sudoku puzzle using backtracking. It efficiently fills all empty cells while satisfying all constraints.

8. Conclusion

- Demonstrates recursive backtracking for constraint satisfaction problems.
- Intelligent pruning reduces exponential search space.
- Strengthens understanding of algorithm design and recursion in C++.