Sudoku Solver Using AI (Backtracking Algorithm)

- **Project Title**: Sudoku Solver Using AI (Backtracking Algorithm)
- **Submitted By**: Yuvraj Singh
- **Roll No**: 202401100300291
- **Course Name:** Introduction to AI
- **College Name**: KIET Group of Institutions
- **Submission Date**: 11 – 03 – 2025
- **Submitted To:** Mr. Abhishek Shukla

1. Abstract

Sudoku is a widely played number puzzle game that requires logical reasoning and problem-solving skills. The objective is to fill a 9×9 grid such that every row, every column, and each of the nine 3×3 subgrids contain the numbers 1 to 9 without repetition. This project implements a **Sudoku Solver** using the **Backtracking Algorithm**, an AI-based approach that systematically explores potential solutions. The solver efficiently finds valid solutions by recursively testing numbers and backtracking when conflicts arise. This report explains the problem statement, methodology, implementation, and results obtained.

---

2. Introduction

Sudoku is a logic-based puzzle that has gained immense popularity worldwide. The game's complexity increases as more numbers are removed from the initial board, making it a suitable problem for algorithmic solutions. AI techniques such as **constraint satisfaction** and **backtracking search** can be employed to find solutions efficiently. This project presents a Sudoku solver implemented in Python, utilizing the backtracking algorithm to explore possible solutions recursively.

3. Problem Statement

Given a **partially filled Sudoku grid**, the goal is to complete the grid by assigning numbers to empty cells while following these constraints:

1. Each row must contain the digits 1-9 exactly once.
2. Each column must contain the digits 1-9 exactly once.
3. Each 3×3 subgrid must contain the digits 1-9 exactly once.

The challenge is to find an efficient algorithm to solve Sudoku while ensuring correctness and minimal computation time.

4. Methodology

Algorithm Used: Backtracking

The backtracking algorithm is a recursive approach that tries to place numbers in the empty cells one by one and verifies their validity:

1. **Find an empty cell** in the grid.
2. **Try numbers 1-9** in the empty cell.
3. **Check validity** using Sudoku constraints (row, column, and subgrid checks).
4. **Recursively attempt to solve** the remaining puzzle with the placed number.

5. If no valid number exists, **backtrack** and try a different number.
6. Repeat until the entire grid is filled correctly.

---

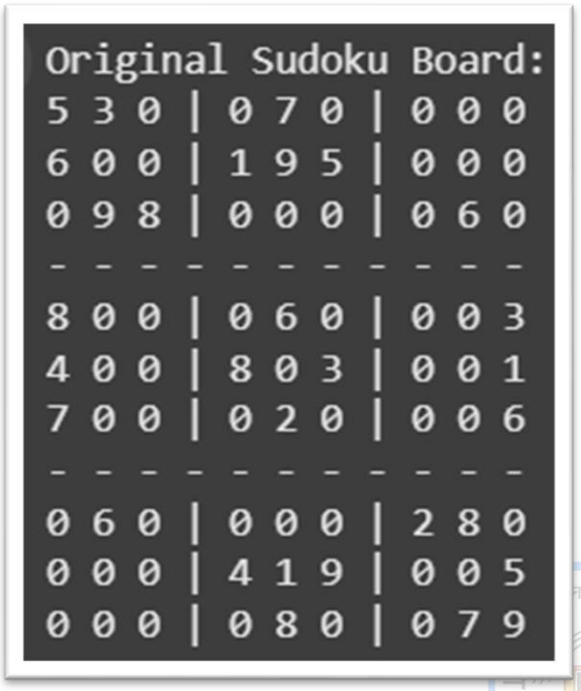
5. Implementation (Code)

Programming Language: Python

The Sudoku solver is implemented in Python using a structured approach with the following key functions:

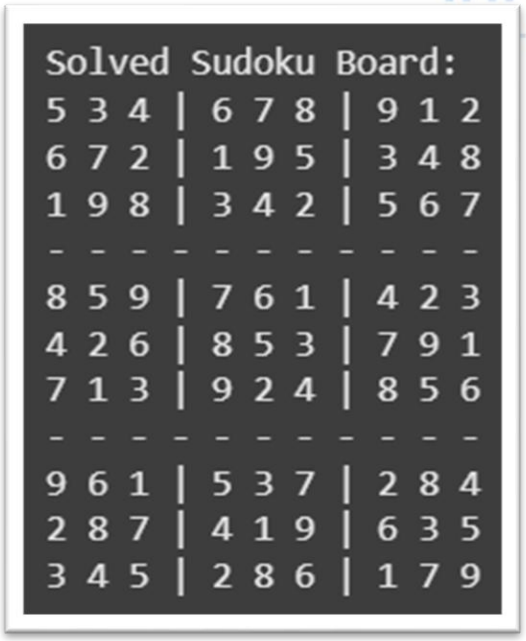
```
def is_valid(board, row, col, num):
    """
    Check if placing 'num' in the given cell (row, col) is valid.
    Ensures that the number is not repeated in the row, column, or
3x3 subgrid.
    """
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def solve_sudoku(board):
    """
    Solve the Sudoku puzzle using a backtracking algorithm.
    This method systematically searches for a valid solution.
    """
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:  # Find an empty cell
                for num in range(1, 10):  # Try numbers 1-9
                    if is_valid(board, row, col, num):
                        board[row][col] = num  # Place the number
                        if solve_sudoku(board):
                            return True  # If solved, return True
                        board[row][col] = 0  # Undo the move (backtrack)
                return False  # No valid number found, backtrack
    return True  # Puzzle solved


def print_board(board):
    """
    Print the Sudoku board in a readable format.
    """
    for row in range(9):
        if row % 3 == 0 and row != 0:
            print("- - - - - - - - - - - -")
        for col in range(9):
            if col % 3 == 0 and col != 0:
                print("|", end=" ")
            print(board[row][col], end=" ")
        print()
```

Self – Generated Unsolved Sudoku Board



```
Original Sudoku Board:
5 3 0 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
- - - - - - - - - - -
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
- - - - - - - - - - -
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9
```

Solved Output given by AI



```
Solved Sudoku Board:
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
- - - - - - - - - - -
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
- - - - - - - - - - -
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```

6. Results & Discussion

- The solver successfully finds the correct solution for given Sudoku puzzles.
- The backtracking algorithm ensures all constraints are met while minimizing unnecessary computations.
- **Time Complexity:** In the worst case, backtracking can have an **exponential time complexity** due to the multiple possibilities for filling empty cells.
- **Advantages:**
  - Guarantees a solution if one exists.
  - Simple and effective approach.
- **Limitations:**
  - Performance decreases for very complex puzzles with multiple empty cells.
  - Can be slow for large Sudoku variants (16×16 or 25×25 grids).

---

7. Conclusion

This project demonstrates how AI techniques such as **backtracking** can be used to efficiently solve Sudoku puzzles. The implementation successfully solves any valid Sudoku board while following all constraints. The project provides insights into constraint satisfaction problems and their applications in AI-based problem-solving. Future improvements can focus on optimizing the algorithm and integrating machine learning for more advanced Sudoku-solving techniques.

8. References

For code – used Chatgpt for refining.

Author - Yuvraj Singh