# Sudoku Solver

**A PROJECT FILE**
**for**
**INTRODUCTION TO AI (AI201B)**
**Session (2024-25)**

**Submitted by**

**Kunal Prajapati**
**202410116100108**
**Doulat Biswal**
**202410116100070**
**Krishna**
**202410116100103**
**Harsh Aggarwal**
**202410116100081**

**Submitted in partial fulfilment of the**
**Requirements for the Degree of**

**MASTER OF COMPUTER APPLICATION**

**Under the Supervision of**
**Mr. APOORV JAIN**
**Assistant Professor**

**Submitted to**

**DEPARTMENT OF COMPUTER APPLICATIONS**
**KIET Group of Institutions, Ghaziabad**
**Uttar Pradesh-201206**

**(APRIL- 2025)**

# TABLE OF CONTENT

# INTRODUCTION

## 1. Introduction

### 1.1 Background
Sudoku is a popular number puzzle game that requires filling a 9×9 grid with digits from 1 to 9, ensuring that each row, column, and 3×3 subgrid contains all numbers without repetition. Automating the solution process is an interesting computational problem that involves backtracking and constraint satisfaction techniques.

### 1.2 Motivation
Manually solving Sudoku puzzles can be time-consuming and challenging, especially for complex puzzles. The aim of this project is to develop an efficient Sudoku Solver using programming techniques to automate the process.

### 1.3 Objectives
- Implement an algorithm to solve any valid Sudoku puzzle.
- Use backtracking to ensure correctness and efficiency.
- Provide a user-friendly interface for input and output.

### 1.4 Significance
The project demonstrates the application of recursive algorithms, constraint satisfaction problems, and optimization techniques. It also helps in understanding how computational logic can be applied to real-world problem.

# OVERVIEW

Sudoku is a well-known puzzle that consists of a 9×9 grid divided into nine 3×3 subgrids. The objective is to fill the grid with digits from 1 to 9, ensuring that each row, column, and subgrid contains each number exactly once. While simpler puzzles can be solved manually, more complex ones require advanced problem-solving techniques. This project implements a **Sudoku Solver** using a **backtracking algorithm**, which systematically searches for a solution by trying different possibilities and undoing incorrect moves.

The algorithm follows a structured approach. First, it scans the grid for an empty cell. Once an empty cell is found, it attempts to fill it with a number from 1 to 9 while checking if the number violates any Sudoku rules. If the number is valid, it is placed in the cell, and the solver moves to the next empty cell. If at any point, no valid number can be placed, the algorithm **backtracks** by removing the previously placed number and trying the next possible value. This recursive process continues until the entire grid is successfully filled.

The Sudoku Solver is implemented using **Python**, with functions that check the validity of a number, find empty cells, and recursively solve the puzzle. The solution is displayed in a structured format, making it easy to interpret. The algorithm guarantees a correct solution for any valid input, assuming the puzzle has at least one solution.

One of the strengths of this approach is its accuracy—backtracking ensures that only valid solutions are generated. However, the performance depends on the puzzle's difficulty. More advanced techniques, such as **constraint propagation** or **machine learning-based solvers**, could further optimize the solution. This project demonstrates the effectiveness of algorithmic problem-solving in real-world applications.

# METHODOLOGY

**The Sudoku solver follows a structured approach to generate, modify, and solve Sudoku puzzles. The methodology consists of the following steps:**

1. **Generating a Fully Solved Sudoku Board:**
    - A 9x9 grid is initialized with zeros.
    - A few numbers are randomly placed along the diagonal to ensure variation.
    - The Backtracking Algorithm is used to fill the entire board, ensuring a valid Sudoku solution.

2. **Creating a Playable Puzzle:**
    - To create a challenging Sudoku puzzle, numbers are removed from random positions on the solved board.
    - The difficulty level is controlled by specifying how many numbers to remove (e.g., 40 numbers for medium difficulty).

3. **Solving the Puzzle:**
    - The Backtracking Algorithm is applied again to fill in the missing numbers.
    - The algorithm tries placing numbers 1-9 in empty cells while ensuring Sudoku rules are followed.
    - If a number placement leads to an unsolvable state, the algorithm backtracks and tries another number.

4. **Displaying Results:**
    - The program prints the generated Sudoku puzzle before solving.
    - The solved Sudoku grid is displayed for verification.
    - The user is prompted to generate another puzzle or exit the program**.**

# CODE

```
import random

def is_valid(board, row, col, num):
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):
for row in range(9):
for col in range(9):
if board[row][col] == 0:
for num in range(1, 10):
if is_valid(board, row, col, num):
board[row][col] = num
if solve_sudoku(board):
return True
board[row][col] = 0
return False
return True

def print_board(board):
for row in board:
print(" ".join(str(num) for num in row))

def generate_full_sudoku():
board = [[0 for _ in range(9)] for _ in range(9)]
numbers = list(range(1, 10))

for i in range(9):
num = numbers.pop(random.randint(0, len(numbers) - 1))
board[i][i] = num

solve_sudoku(board)
return board

def remove_numbers(board, difficulty=40):
puzzle = [row[:] for row in board]
for _ in range(difficulty):
row, col = random.randint(0, 8), random.randint(0, 8)
```

```
while puzzle[row][col] == 0:
row, col = random.randint(0, 8), random.randint(0, 8)
puzzle[row][col] = 0
return puzzle

def play_sudoku():
while True:
print("Generating a random Sudoku puzzle...")
full_board = generate_full_sudoku()
puzzle_board = remove_numbers(full_board)

print("Sudoku Puzzle:")
print_board(puzzle_board)

if solve_sudoku(puzzle_board):
print("Solved Sudoku:")
print_board(puzzle_board)
else:
print("No solution exists.")

play_again = input("Do you want to generate another Sudoku? (yes/no): ").strip().lower()
if play_again != 'yes':
break

play_sudoku()
```

# OUTPUT

```
Generating a random Sudoku puzzle...
Sudoku Puzzle:
3 0 2 0 5 6 0 0 9
5 0 6 1 0 0 0 3 4
0 8 0 0 3 0 0 6 5
1 0 3 8 0 0 0 9 7
0 0 7 9 1 0 3 0 8
0 0 9 0 6 7 0 1 2
2 0 0 0 0 0 0 0 1
6 0 0 7 0 1 9 0 0
9 7 1 0 2 0 0 4 6
Solved Sudoku:
3 1 2 4 5 6 7 8 9
5 9 6 1 7 8 2 3 4
7 8 4 2 3 9 1 6 5
1 2 3 8 4 5 6 9 7
4 6 7 9 1 2 3 5 8
8 5 9 3 6 7 4 1 2
2 3 8 6 9 4 5 7 1
6 4 5 7 8 1 9 2 3
9 7 1 5 2 3 8 4 6
Do you want to generate another Sudoku? (yes/no): yes
```

```
9 7 1 5 2 3 8 4 6
Do you want to generate another Sudoku? (yes/no): yes
Generating a random Sudoku puzzle...
Sudoku Puzzle:
0 2 3 1 0 4 0 0 0
4 0 6 7 8 9 0 3 5
8 0 7 0 0 0 0 0 0
1 4 2 0 0 0 0 9 0
3 0 8 0 4 1 5 0 0
0 5 0 6 2 8 3 0 0
0 3 0 0 0 6 0 0 1
9 0 5 4 1 0 0 6 0
0 8 1 5 9 3 4 0 0
Solved Sudoku:
5 2 3 1 6 4 7 8 9
4 1 6 7 8 9 2 3 5
8 9 7 2 3 5 1 4 6
1 4 2 3 5 7 6 9 8
3 6 8 9 4 1 5 2 7
7 5 9 6 2 8 3 1 4
2 3 4 8 7 6 9 5 1
9 7 5 4 1 2 8 6 3
6 8 1 5 9 3 4 7 2
Do you want to generate another Sudoku? (yes/no): no
```

# Conclusion

The Sudoku Solver project successfully demonstrates the implementation of algorithmic problem-solving using **backtracking**. Sudoku is a complex logical puzzle that requires systematic trial and error, and the backtracking algorithm efficiently finds a solution by eliminating incorrect choices and recursively exploring valid possibilities. By ensuring that each row, column, and 3×3 subgrid adheres to Sudoku's fundamental constraints, the solver is able to determine a correct solution for any solvable puzzle.

One of the key strengths of this approach is its **accuracy and reliability**. The backtracking algorithm guarantees a valid solution whenever one exists. Additionally, the implementation is **simple yet powerful**, making it suitable for solving a wide range of Sudoku puzzles, from easy to extremely difficult. The solution is systematically built by attempting to place numbers in empty cells while continuously checking for rule violations, ensuring that the logic remains sound throughout the process.

However, the project also highlights some **limitations**. The backtracking algorithm, while effective, can become computationally expensive for particularly complex puzzles. As the difficulty of the Sudoku puzzle increases, the number of recursive calls required to find a solution also rises, leading to potential performance bottlenecks. Future improvements could involve **optimizing the solver** using techniques such as **constraint propagation**, which reduces the search space by ruling out impossible values early on, or **heuristic-based approaches**, which prioritize filling cells based on the most constrained choices.

Another possible enhancement is the integration of **machine learning** or **AI-based solvers**, which could learn patterns from previously solved puzzles to make predictions and solve puzzles faster. This would be particularly useful for solving Sudoku variations or optimizing puzzle generation.

In conclusion, the Sudoku Solver effectively showcases the power of **algorithmic thinking and computational problem-solving**. It serves as an excellent example of how programming can be used to tackle logical puzzles efficiently. While backtracking provides a strong foundation, further enhancements using AI and optimization techniques could make Sudoku solving even more efficient and scalable for larger or more challenging puzzles.