RAJALAKSHMI INSTITUTE OF TECHNOLOGY
(An Autonomous Institution, Affiliated to Anna University, Chennai)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

ACADEMIC YEAR 2025 - 2026

SEMESTER III

ARTIFICIAL INTELLIGENCE LABORATORY

MINI PROJECT REPORT

REGISTER NUMBER	2117240070366
NAME	Yamunadevi Prakash
PROJECT TITLE	Sudoku Solver as a CSP framed like a game
DATE OF SUBMISSION	
FACULTY IN-CHARGE	**Mrs. M. Divya**

Signature of Faculty In-charge

INTRODUCTION

Artificial Intelligence (AI) is the science of developing intelligent agents capable of performing tasks that require reasoning, learning, and decision-making. One significant area of AI is **Constraint Satisfaction Problems (CSPs)**, where problems are defined in terms of variables, their possible values (domains), and constraints that restrict which combinations of values are valid.

This project, titled *"Sudoku Solver as a CSP Game"*, demonstrates how a complex human reasoning task — solving Sudoku — can be modeled as a CSP and efficiently solved using **backtracking with heuristics**. Sudoku is a 9×9 logic puzzle where each row, column, and 3×3 box must contain the digits 1–9 exactly once.

The project allows users to **interact with the Sudoku grid**, fill cells manually, and then watch the system solve it automatically using AI-based constraint reasoning. This highlights the power of CSP techniques for logical and combinatorial problem solving.

PROBLEM STATEMENT

To design and implement a **Sudoku Solver** in Python that models the puzzle as a **Constraint Satisfaction Problem (CSP)** and solves it efficiently using **backtracking search** combined with **constraint propagation** and **Minimum Remaining Value (MRV) heuristic**, while providing a simple interactive game-like interface.

GOAL

The primary goals of this project are:

- To represent Sudoku as a CSP with variables, domains, and constraints.
- To apply AI-based search algorithms (backtracking + MRV).
- To allow user interaction (modify cells, auto-solve option).
- To showcase constraint reasoning and efficient search techniques in AI.

THEORETICAL BACKGROUND

A **Constraint Satisfaction Problem (CSP)** consists of:

- **Variables:** Each cell in the Sudoku grid (81 in total).
- **Domains:** Possible values {1,2,3,4,5,6,7,8,9}.
- **Constraints:** Each number must appear once per row, column, and subgrid.

Algorithm Overview

- **Backtracking Search:**
  A depth-first search method that assigns values to variables one by one. If a violation occurs, it backtracks to try alternate assignments.
- **Constraint Propagation:**
  The domains of unassigned variables are reduced dynamically after each assignment, ensuring that no future assignments violate constraints.
- **MRV Heuristic (Minimum Remaining Values):**
  Chooses the variable with the smallest domain first, thereby reducing branching and improving efficiency.

Literature Survey

- Russell & Norvig, *Artificial Intelligence: A Modern Approach*, emphasize CSPs for structured logical problem solving.
- Dechter (2003) in *Constraint Processing* highlights efficiency improvements from heuristics like MRV and forward checking.
- Other methods (e.g., genetic algorithms, simulated annealing) exist but CSPs remain the most interpretable and consistent for Sudoku.

Justification for Choosing CSP

Sudoku fits naturally as a CSP because all constraints are explicit and deterministic. CSP solvers handle such structured constraints more efficiently than heuristic search or brute force methods.

ALGORITHM EXPLANATION WITH EXAMPLE

Steps of the CSP Backtracking Algorithm with MRV:

1. Select the unfilled cell with the **minimum remaining valid values (MRV)**.
2. Generate the domain (1–9) for that cell by checking Sudoku constraints.
3. Assign a value and propagate the effect (update possible domains of others).
4. Continue recursively until all cells are filled.
5. If any cell has no valid values, **backtrack** to the previous assignment.

Example:
If a cell in row 1, column 3 can only take {1,4,6}, MRV ensures this cell is picked first. When 1 is placed, all other conflicting cells' domains are updated. If inconsistency arises, it backtracks and tries 4 or 6.

IMPLEMENTATION AND CODE

"""

Sudoku Solver as a CSP Game

A Constraint Satisfaction Problem approach to solving Sudoku puzzles.

```
"""
class SudokuCSP:
    def __init__(self, grid):
        """Initialize the Sudoku CSP with a 9x9 grid (0 represents empty cells)."""
        self.grid = [row[:] for row in grid]  # Copy grid
        self.size = 9
        self.box_size = 3


    def is_valid(self, row, col, num):
        """Check if placing num at (row, col) violates constraints."""
        # Row constraint
        if num in self.grid[row]:
            return False
        # Column constraint
        if num in [self.grid[r][col] for r in range(self.size)]:
            return False
        # Box constraint (3x3 subgrid)
        box_row, box_col = (row // 3) * 3, (col // 3) * 3
        for r in range(box_row, box_row + 3):
            for c in range(box_col, box_col + 3):
                if self.grid[r][c] == num:
                    return False
        return True
```

```
def get_domain(self, row, col):
    """Get valid domain (possible values) for a cell using constraint propagation."""
    if self.grid[row][col] != 0:
        return []
    return [num for num in range(1, 10) if self.is_valid(row, col, num)]
def find_mrv_cell(self):
    """Find empty cell with Minimum Remaining Values (MRV heuristic)."""
    min_domain_size = 10
    best_cell = None
    for row in range(self.size):
        for col in range(self.size):
            if self.grid[row][col] == 0:
                domain = self.get_domain(row, col)
                if len(domain) < min_domain_size:
                    min_domain_size = len(domain)
                    best_cell = (row, col, domain)
    return best_cell


def solve(self):
    """Solve Sudoku using backtracking with MRV and constraint propagation."""
    # Find empty cell with minimum remaining values
    result = self.find_mrv_cell()
    if result is None:
```

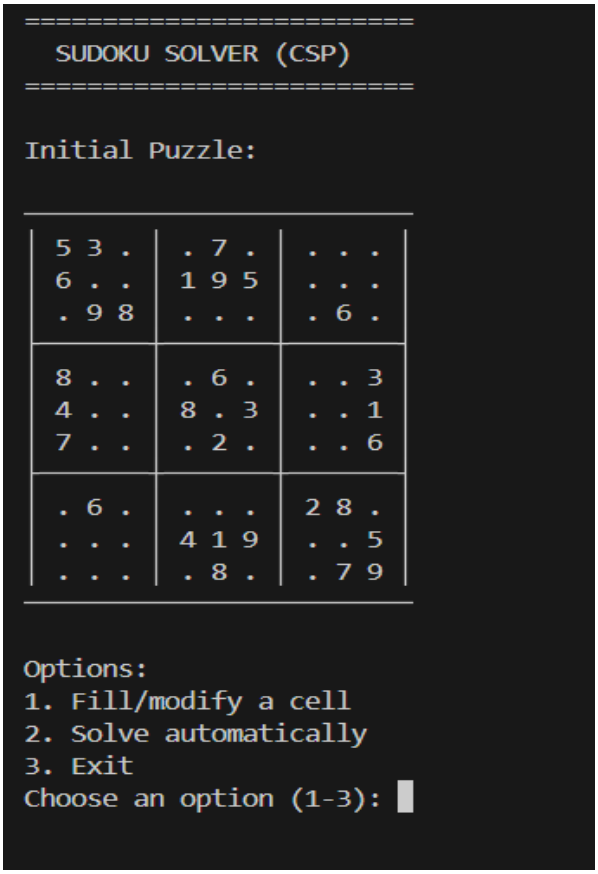
```
            return True  # All cells filled, puzzle solved
        row, col, domain = result
        if not domain:
            return False  # No valid values, backtrack
        # Try each value in the domain
        for num in domain:
            self.grid[row][col] = num
            if self.solve():
                return True
            # Backtrack
            self.grid[row][col] = 0
        return False


def display_grid(grid):
    """Display the Sudoku grid in a neat format."""
    print("\n" + "─" * 25)
    for i, row in enumerate(grid):
        if i % 3 == 0 and i != 0:
            print("├" + "─" * 7 + "┼" + "─" * 7 + "┼" + "─" * 7 + "┤")
        row_str = "│ "
        for j, num in enumerate(row):
            if j % 3 == 0 and j != 0:
                row_str += "│ "
```

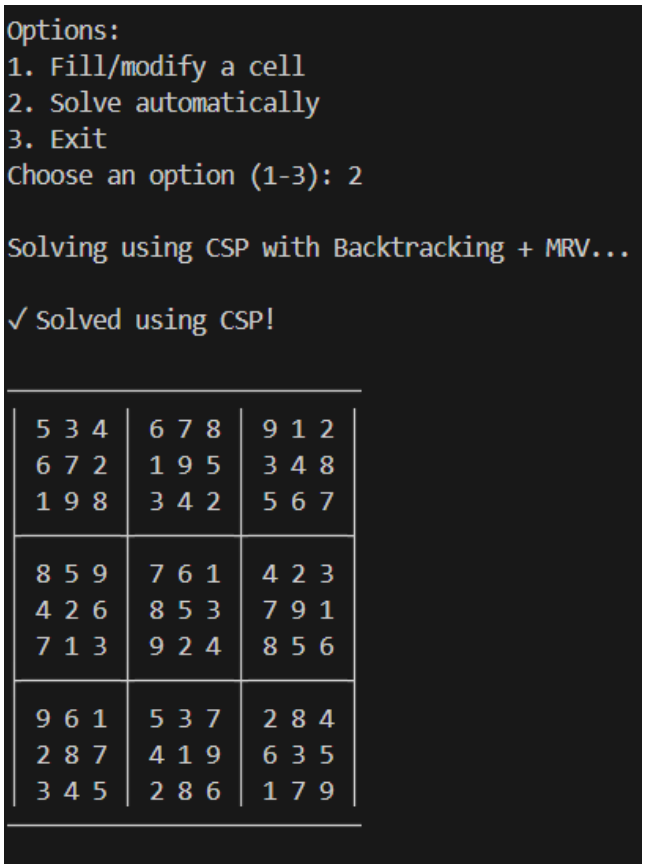
```
            row_str += str(num if num != 0 else ".") + " "
        row_str += "┃ "
        print(row_str)
    print("━" * 25 + "\n")


def get_user_input():
    """Get user's choice for game interaction."""
    print("Options:")
    print("1. Fill/modify a cell")
    print("2. Solve automatically")
    print("3. Exit")
    return input("Choose an option (1-3): ").strip()


def main():
    """Main game loop."""
    # Sample Sudoku puzzle (0 represents empty cells)
    puzzle = [
        [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]
]
print("=" * 25)
print("  SUDOKU SOLVER (CSP)")
print("=" * 25)
print("\nInitial Puzzle:")
display_grid(puzzle)
while True:
    choice = get_user_input()
    if choice == "1":
        try:
            row = int(input("Enter row (1-9): ")) - 1
            col = int(input("Enter column (1-9): ")) - 1
            num = int(input("Enter number (0-9, 0 for empty): "))
            if 0 <= row < 9 and 0 <= col < 9 and 0 <= num <= 9:
                puzzle[row][col] = num
                print("\nUpdated Grid:")
                display_grid(puzzle)
            else:
                print("Invalid input! Use values 1-9 for position and 0-9 for number.")
        except ValueError:
```

```
            print("Invalid input! Please enter numbers only.")
        elif choice == "2":
            print("\nSolving using CSP with Backtracking + MRV...\n")
            solver = SudokuCSP(puzzle)
            if solver.solve():
                print("✓ Solved using CSP!")
                display_grid(solver.grid)
            else:
                print("✗ No solution exists for this puzzle.")
            break
        elif choice == "3":
            print("Thanks for playing!")
            break
        else:
            print("Invalid choice! Please select 1, 2, or 3.")

if __name__ == "__main__":
    main()
```

OUTPUT





RESULTS AND FUTURE ENHANCEMENT

The CSP-based approach successfully solves Sudoku puzzles with efficiency and clarity. The MRV heuristic and constraint propagation reduce backtracking and make the algorithm intelligent and systematic.

Future Enhancements:

- Add **graphical user interface (GUI)** using Tkinter.
- Implement **forward checking** for deeper propagation.
- Add random Sudoku puzzle generation and difficulty levels.
- Integrate timing and performance statistics for educational insight.

Git Hub Link of the project and report	https://github.com/YamunadeviPrakash/Sudoku-Solver.git

REFERENCES

1. Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th Ed.). Pearson.
2. Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.
3. Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.
4. Towards Data Science Blog – "Solving Sudoku with CSP and Backtracking in Python."
5. GeeksforGeeks – "Sudoku Solver using Backtracking Algorithm."