

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

**DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240030045 |
| **NAME** | HARSHINI J |
| **PROJECT TITLE** | SUDOKU SOLVER USING BACKTRACKING ALGORITHM |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION**

* Artificial Intelligence enables machines to solve complex problems with human-like intelligence. One such classic problem is Sudoku, a logic-based number-placement puzzle.
* The Sudoku Solver project demonstrates how AI search algorithms like Backtracking can automate the solving process.
* The system reads an unsolved Sudoku grid and fills in all missing numbers while following the puzzle constraints — showing how reasoning and constraint satisfaction can be modeled programmatically.

**PROBLEM STATEMENT**

* The challenge is to develop an intelligent system capable of solving any valid Sudoku puzzle using AI-based problem-solving strategies.
* The program should correctly fill in all missing cells following Sudoku rules:  
   1. Each row contains digits 1–9 exactly once.  
   2. Each column contains digits 1–9 exactly once.  
   3. Each 3×3 subgrid contains digits 1–9 exactly once.

**GOAL**

* Implement a **GUI-based Sudoku Solver** using Python’s Tkinter for easy visualization.
* Allow users to **input or use a partially filled Sudoku grid**.
* Use the **Backtracking algorithm** to automatically compute the solution.
* Display the **solved Sudoku grid** interactively on the GUI.

**THEORETICAL BACKGROUND**

* + **Sudoku solving** is a **constraint satisfaction problem** where each cell in the 9×9 grid is treated as a state.
  + **Backtracking Algorithm** is commonly used for solving Sudoku.
  + It fills an empty cell with numbers **1–9** and checks **row, column, and 3×3 subgrid constraints**.
  + The algorithm **recursively explores** the next empty cell and **backtracks** if a contradiction occurs.
  + The advantages are Simple, efficient, and guarantees a solution for **valid 9×9 grids**.
  + Alternative approaches are Constraint Propagation and Stochastic Search (Simulated Annealing).
  + **Justification:** Backtracking is chosen because it is **simple, efficient, deterministic, and guarantees a solution** for standard 9×9 Sudoku grids, making it ideal for small to medium-sized puzzles.

**ALGORITHM EXPLANATION WITH EXAMPLE**

* Initialize a 9×9 Sudoku grid with given numbers, using 0 to represent empty cells.
* Display the grid using **Tkinter** entry boxes, where pre-filled numbers are shown in blue and locked.
* When the user clicks **“Solve Sudoku”**, read all values from the grid into a 2D list.
* Scan the grid to find the **first empty cell** (a cell containing 0).
* For that empty cell, **try placing numbers 1 to 9** sequentially.
* For each number, **check validity** using Sudoku rules — the number must not repeat in the same row, column, or 3×3 subgrid.
* If the number is valid, **place it temporarily** in the current cell and **recursively solve** the remaining grid.
* If a **dead end** is reached (no valid number fits), **backtrack** by resetting the cell to 0 and trying the next number.
* Continue recursively until all cells are filled correctly, meaning the Sudoku is solved.
* Once solved, update the GUI to display the completed grid and show a message saying **“Sudoku Solved Successfully!”**; otherwise, display **“No Solution Exists.”**

**Example:**

* Given a partially filled Sudoku grid, the algorithm starts at the first empty cell (0,2) and tries numbers 1–9, checking row, column, and 3×3 box constraints. Valid numbers are placed, and the algorithm moves to the next empty cell recursively. If no number fits, it **backtracks** to the previous cell and tries the next option. This process continues until the entire grid is correctly filled, producing the complete solution.

**IMPLEMENTATION AND CODE**

import tkinter as tk

from tkinter import messagebox

# ---------------- Sudoku Solver Logic ----------------

def find\_empty(board):

    for i in range(9):

        for j in range(9):

            if board[i][j] == 0:

                return (i, j)

    return None

def is\_valid(board, num, pos):

    row, col = pos

    # Check row

    for j in range(9):

        if board[row][j] == num and col != j:

            return False

    # Check column

    for i in range(9):

        if board[i][col] == num and row != i:

            return False

    # Check 3x3 box

    box\_x = col // 3

    box\_y = row // 3

    for i in range(box\_y\*3, box\_y\*3 + 3):

        for j in range(box\_x\*3, box\_x\*3 + 3):

            if board[i][j] == num and (i, j) != pos:

                return False

    return True

def solve(board):

    empty = find\_empty(board)

    if not empty:

        return True

    row, col = empty

    for num in range(1, 10):

        if is\_valid(board, num, (row, col)):

            board[row][col] = num

            if solve(board):

                return True

            board[row][col] = 0

    return False

# ---------------- GUI Section ----------------

class SudokuGUI:

    def \_\_init\_\_(self, root, initial\_board=None):

        self.root = root

        self.root.title("Sudoku Solver - Backtracking")

        self.entries = [[None for \_ in range(9)] for \_ in range(9)]

        self.fixed\_cells = set()

        # Create 9x9 grid of entry boxes

        for i in range(9):

            for j in range(9):

                entry = tk.Entry(root, width=3, font=("Arial", 18), justify="center")

                entry.grid(row=i, column=j, padx=3, pady=3)

                # Add thicker borders to separate 3x3 boxes

                if i % 3 == 0 and i != 0:

                    entry.grid(pady=(10, 3))

                if j % 3 == 0 and j != 0:

                    entry.grid(padx=(10, 3))

                self.entries[i][j] = entry

        # Single Solve Button

        solve\_btn = tk.Button(root, text="Solve Sudoku", command=self.solve\_gui,

                              bg="lightgreen", font=("Arial", 12, "bold"))

        solve\_btn.grid(row=9, column=0, columnspan=9, sticky="we", pady=10)

        # Load initial board if given

        if initial\_board:

            self.set\_board(initial\_board)

    def get\_board(self):

        board = []

        for i in range(9):

            row = []

            for j in range(9):

                val = self.entries[i][j].get()

                if val == "":

                    row.append(0)

                else:

                    try:

                        num = int(val)

                        if 1 <= num <= 9:

                            row.append(num)

                        else:

                            row.append(0)

                    except:

                        row.append(0)

            board.append(row)

        return board

    def set\_board(self, board):

        for i in range(9):

            for j in range(9):

                entry = self.entries[i][j]

                entry.delete(0, tk.END)

                if board[i][j] != 0:

                    entry.insert(0, str(board[i][j]))

                    entry.config(fg="blue", state="disabled")  # fixed cells

                    self.fixed\_cells.add((i, j))

                else:

                    entry.config(state="normal", fg="black")

    def solve\_gui(self):

        board = self.get\_board()

        if solve(board):

            self.set\_board(board)

            messagebox.showinfo("Sudoku Solver", "Sudoku Solved Successfully!")

        else:

            messagebox.showerror("Sudoku Solver", "No solution exists!")

# ---------------- Main Program ----------------

if \_\_name\_\_ == "\_\_main\_\_":

    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]

    ]

    root = tk.Tk()

    gui = SudokuGUI(root, puzzle)

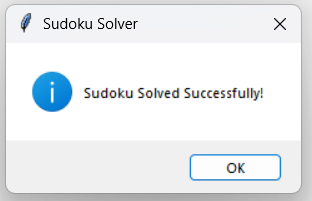
    root.mainloop()

**OUTPUT**

* The program **opens a 9×9 Sudoku grid** with pre-filled numbers from the puzzle. Pre-filled numbers are displayed which are **uneditable**..
* Clicking the **“Solve Sudoku”** button runs the **backtracking algorithm** to fill in empty cells.
* **Solved numbers** are displayed in the grid (currently in black, editable) while pre-filled numbers remain **locked.**



* If the puzzle is successfully solved, a **message box** appears:  
  **“Sudoku Solved Successfully!”**



* If the puzzle is unsolvable, a message box appears:  
  “ No solution exists!”
* The final grid clearly shows a complete and valid Sudoku solution, satisfying all row, column, and 3×3 subgrid constraints.

**RESULTS AND FUTURE ENHANCEMENT**

* **Results**
* The program successfully displays a **9×9 Sudoku grid** with pre-filled and empty cells.
* Pre-filled numbers are clearly **uneditable**.
* Users can enter numbers in empty cells to create custom puzzles.
* Clicking **“Solve Sudoku”** automatically solves the puzzle using **backtracking**.
* The algorithm fills all empty cells correctly, adhering to **row, column, and 3×3 subgrid constraints**.
* A **message box confirms** when the puzzle is successfully solved.
* If the puzzle is unsolvable, the program alerts the user with a **“No solution exists”** message.
* The program demonstrates the **efficiency and reliability of backtracking** for standard 9×9 Sudoku puzzles.
* **Future enhancements**
* Add a **hint feature** to suggest possible numbers for empty cells.
* Implement **auto-checking** for incorrect user inputs in real-time.
* Include **different difficulty levels** (easy, medium, hard) by generating puzzles programmatically.
* Add **timer and scoring system** for competitive Sudoku solving.
* Incorporate **color-coded highlights** for conflicts or repeated numbers.
* Extend the program to handle larger Sudoku Grids(e.g.,16x16).
* Integrate a **puzzle generator** to create random solvable Sudoku puzzles automatically.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | **https://github.com/HarshiniJayakumar/HARSHINI-J** |

**REFERENCES**

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson.

[2] “Sudoku Solver using Backtracking in Python,” GeeksforGeeks. [Online]. Available: <https://www.geeksforgeeks.org/sudoku-backtracking-7/>

[3] “Python Sudoku Solver,” Real Python. [Online]. Available: <https://realpython.com/python-sudoku-solver/>

[4] Python Official Documentation, Tkinter Library. [Online]. Available: <https://docs.python.org/3/library/tkinter.html>

[5] “Solving Sudoku with Backtracking,” Towards Data Science. [Online]. Available: <https://towardsdatascience.com/solving-sudoku-with-backtracking-196e2f0c31d6>