Green University of Bangladesh

Department of Computer Science and Engineering (CSE)
Semester: (Spring, Year: 2025), B.Sc. in CSE (Day)

Sudoku – solver

Course Title: Artificial Intelligence Lab
Course Code:CSE 316
Section: 221-D10

Students Details

Name	ID
Md Sohel	221002173
Moriam Khatun	221002197
Abu Sayed	221002387

Submission Date: 21-05-2025
Course Teacher's Name: Md Fahimul Islam

Contents

Chapter 1

Introduction

1.1 Overview

This project aims to develop a Sudoku Solver using Backtracking and Constraint Satisfaction techniques in Python. The solver will efficiently fill a 9×9 Sudoku grid by ensuring that numbers 1-9 appear only once in each row, column, and 3×3 subgrid. The implementation will demonstrate core AI problem-solving strategies.This project is an AI-powered Sudoku Solver that allows users to input or capture Sudoku puzzles and get accurate, instant solutions. It integrates backtracking algorithms, constraint satisfaction (CSP), and additional AI features like hints and explanation tracking.

1.2 Motivation

Sudoku puzzles are globally popular for cognitive training. However, complex puzzles can be frustrating. This tool aims to assist users with solving them efficiently and understanding the solving process—bridging entertainment with AI-powered learning.

- Implementing Backtracking and Constraint Satisfaction helps in learning fundamental AI algorithms.
- Enhances logical reasoning and optimization techniques.
- Sudoku solving strategies apply to scheduling, resource allocation, and combinatorial problems.
- Strengthens Python programming skills, especially in recursion and constraint handling.
- Allows analysis of efficiency between Backtracking and Constraint Satisfaction methods.
- Prepares for more complex AI problems like CSP (Constraint Satisfaction Problems).

1.3 Problem Definition

1.3.1 Problem Statement

Manual Sudoku solving can be time-consuming and difficult for non-experts. Users need a tool that not only solves puzzles but explains the steps and helps with learning.

- Ensure that the initial Sudoku puzzle follows the rules (no duplicates in rows, columns, or sub-grid).
- Develop a recursive backtracking algorithm to systematically explore possible solutions.
- Apply constraint propagation to efficiently reduce the search space.
- Ensure that the solver works correctly to vary difficulty levels while optimizing speed.

1.3.2 Complex Engineering Problem

The project handles a complex problem: real-time Sudoku puzzle recognition, constraint-based decision-making, and user interaction through a GUI. It addresses dynamic constraint propagation, backtracking efficiency, and UI responsiveness.

Table 1.1: Summary of the attributes touched by the mentioned projects

Name of the P Attributess	**Explain how to address**
P1: Depth of knowledge required	Requires Python fundamentals + AI techniques (Backtracking, Constraint Satisfaction).
P2: Range of conflicting requirements	Backtracking ensures completeness but is slow; CSP improves speed but needs careful implementation
P3: Depth of analysis required	Must evaluate time complexity, test on puzzles of varying difficulty, and compare methods.
P4: Familiarity of issues	Recursion limits in backtracking, incorrect constraint propagation in CSP.
P5: Extent of applicable codes	Backtracking template and CSP functions can be adapted for similar problems.
P6: Extent of stakeholder involvement and conflicting requirements	This Sudoku solver project involves minimal academic stakeholder input but must balance key tradeoffs between backtracking's completeness and CSP's efficiency while maintaining educational clarity and potential for practical application.
P7: Interdependence	CSP reduces backtracking's workload; grid validation must run before solving.

1.4 Design Goals/Objectives

- Ensure that the solver correctly solves all valid Sudoku puzzles using backtracking and CSP techniques.
- Minimize the solving time for puzzles of varying difficulty levels.
- Develop reusable components for scalability.
- Provide clear input/output options.
- Use AI to solve any legitimate Sudoku puzzle.
- Permit GUI or voice commands for input.
- Describe the problem-solving procedure in detail.
- Provide performance indicators and guidance.
- Provide an interface that is easy to use and has accessibility features.

1.5 Application

This Sudoku solver project serves as a practical implementation of AI techniques, demonstrating how backtracking and constraint satisfaction can solve combinatorial problems. It can be used as a teaching tool in AI/algorithm courses to illustrate recursive problem solving and optimization. Beyond academics, the core logic can be adapted for scheduling, resource allocation, or puzzle-based gaming applications. The modular design allows integration with larger AI systems, while efficiency benchmarks provide insight for real-world CSP implementations. Future extensions could include a web/mobile interface for greater accessibility.

- Educational tools for learning Sudoku.
- Mobile/desktop apps for cognitive skill development.
- Integration into learning management systems.
- Visual/voice-based accessibility tools.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

The general architecture, design, and implementation stages of the AI-based Sudoku Human-computer interaction, algorithmic problem-solving, user interface design, and artificial intelligence are all combined in the creation of the AI-based Sudoku Solver. Project setup, architecture, front-end design, back-end logic, data flow, user experience features, and optional system integrations are all covered in detail in this chapter.

The main goal was to create a clever yet user-friendly platform that would enable both novices and experts to solve Sudoku puzzles quickly and comprehend the reasoning behind each step. To ensure ease of use and wide compatibility, we utilized Tkinter for the graphical user interface and Python for the backend.

2.2 Project Details

The project is organized into modular Python files with specific responsibilities:

- main.py: Initializes the GUI, connects UI events to backend functions.
- solver.py: Contains the core solving logic, including backtracking and constraint satisfaction methods.
- utils.py: Helper functions such as validating puzzle states, checking grid constraints, and formatting input/output.

System Architecture

- The architecture of the project is divided into the following layers:
- Presentation Layer: Built using Tkinter to provide a user-friendly interface for entering and solving Sudoku puzzles.

- Application Logic Layer: Handles user input, grid state management, and communicates between the UI and solver.
- AI Solver Layer: Implements algorithms such as backtracking, heuristics, and constraint satisfaction to solve puzzles.
- Voice/Text-to-Speech Integration: Allows user interaction through speech and provides auditory feedback for accessibility.

2.3 Implementation

In order to provide a smooth user experience, the contribution and volunteer platform's implementation centers on the combination of contemporary web development tools with a well-structured database architecture.

2.3.1 main.py :

```
import customtkinter as ctk
# import threading
import time
# from datetime import datetime
from solver import solve, is_valid, solve_csp
from utils import generate_board

ctk.set_appearance_mode("light")
ctk.set_default_color_theme("blue")

class SudokuGUI(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.title(" AI Sudoku Solver")
        self.geometry("1000x740")
        self.strategy = "Backtracking"
        self.entries = [[None for _ in range(9)] for _ in range(9)]
        self.board = [[0 for _ in range(9)] for _ in range(9)]
        self.voice_enabled = True
        self.start_time = None

        self.main_frame = ctk.CTkFrame(self)
        self.main_frame.pack(padx=10, pady=10, fill="both", expand=True)

        self.left_frame = ctk.CTkFrame(self.main_frame)
        self.left_frame.grid(row=0, column=0, padx=10, sticky="n")

        self.right_frame = ctk.CTkFrame(self.main_frame)
        self.right_frame.grid(row=0, column=1, padx=10, sticky="n")
```

```
        self.grid_frame = ctk.CTkFrame(self.left_frame)
        self.grid_frame.pack(pady=10)
        self.create_grid()

        self.button_frame = ctk.CTkFrame(self.left_frame)
        self.button_frame.pack(pady=5)
        self.create_buttons()

        self.message_label = ctk.CTkLabel(self.left_frame, text=" Ready", text_col
        self.message_label.pack(pady=6)

        self.timer_label = ctk.CTkLabel(self.left_frame, text=" Timer: 0s", font=c
        self.timer_label.pack(pady=2)
        self.update_timer()

        self.explanation_box = ctk.CTkTextbox(self.left_frame, width=500, height=1
        self.explanation_box.pack(pady=5)
        self.explanation_box.insert("1.0", " Explanation log will appear here.\n")
        self.explanation_box.configure(state="disabled")

    def create_grid(self):
        for i in range(9):
            for j in range(9):
                var = ctk.StringVar()
                entry = ctk.CTkEntry(self.grid_frame, width=45, height=45, font=ctk
                                     justify="center", textvariable=var)
                entry.grid(row=i, column=j, padx=1, pady=1)
                var.trace_add("write", lambda *args, r=i, c=j: self.validate_cell(
                self.entries[i][j] = entry

    def create_buttons(self):
        self.difficulty_var = ctk.StringVar(value="Easy")
        ctk.CTkOptionMenu(self.button_frame, variable=self.difficulty_var, values=
        ctk.CTkButton(self.button_frame, text=" Generate", command=self.generate_pu
        ctk.CTkButton(self.button_frame, text=" Solve", command=self.solve_sudoku,
        ctk.CTkButton(self.button_frame, text=" Hint", command=self.give_hint, widt
        ctk.CTkButton(self.button_frame, text=" Strategy", command=self.toggle_stra
        ctk.CTkButton(self.button_frame, text=" Explain Move", command=self.explai



    def update_timer(self):
        if self.start_time:
            elapsed = int(time.time() - self.start_time)
            self.timer_label.configure(text=f" Timer: {elapsed}s")
        self.after(1000, self.update_timer)

    def get_board(self):
```

```
    for i in range(9):
        for j in range(9):
            val = self.entries[i][j].get()
            self.board[i][j] = int(val) if val.isdigit() else 0

def fill_board(self):
    for i in range(9):
        for j in range(9):
            self.entries[i][j].delete(0, ctk.END)
            if self.board[i][j] != 0:
                self.entries[i][j].insert(0, str(self.board[i][j]))

def generate_puzzle(self):
    difficulty = self.difficulty_var.get().lower()
    self.board = generate_board(difficulty)
    self.start_time = time.time()
    self.fill_board()
    self.message_label.configure(text=f" Puzzle generated ({difficulty.title()


def toggle_strategy(self):
    self.strategy = "CSP" if self.strategy == "Backtracking" else "Backtracking
    self.message_label.configure(text=f" Strategy set to: {self.strategy}")

def validate_cell(self, row, col, *_):
    value = self.entries[row][col].get()
    if not value.isdigit():
        self.entries[row][col].configure(fg_color="white")
        return
    value = int(value)
    temp_board = [[int(self.entries[i][j].get() or 0) for j in range(9)] for i
    temp_board[row][col] = 0  # Exclude current cell value for self-check
    if not is_valid(temp_board, row, col, value):
        self.entries[row][col].configure(fg_color="misty rose")
        self.message_label.configure(text=f" Invalid: {value} conflicts.")
    else:
        self.entries[row][col].configure(fg_color="light green")
        self.message_label.configure(text=f" {value} is valid.")


    if self.is_board_complete() and self.is_board_valid():
        self.message_label.configure(text=" Congratulations! Puzzle completed!

    if self.is_board_complete() and not self.is_board_valid():
        self.message_label.configure(text="You are not able to solve the puzzl


def is_board_complete(self):
```

```
        for i in range(9):
            for j in range(9):
                val = self.entries[i][j].get()
                if not val.isdigit() or int(val) == 0:
                    return False
        return True

    def is_board_valid(self):
        board = [[int(self.entries[i][j].get() or 0) for j in range(9)] for i in ra
        for i in range(9):
            for j in range(9):
                value = board[i][j]
                if value == 0:
                    continue
                board[i][j] = 0
                if not is_valid(board, i, j, value):
                    return False
                board[i][j] = value
        return True


    def solve_sudoku(self):
        self.get_board()
        self.start_time = time.time()
        if self.strategy == "CSP":
            result, steps, duration = solve_csp(self.board)
        else:
            result, steps, duration = solve(self.board)

        if result:
            self.fill_board()
            self.message_label.configure(text=f" Solved in {steps} steps and {dura
        else:
            self.message_label.configure(text=" No solution found.")

    def give_hint(self):
        self.get_board()
        for i in range(9):
            for j in range(9):
                if self.board[i][j] == 0:
                    possible_values = [num for num in range(1, 10) if is_valid(sel
                    if possible_values:
                        self.entries[i][j].delete(0, ctk.END)
                        self.entries[i][j].insert(0, str(possible_values[0]))
                        self.entries[i][j].configure(fg_color="lightyellow")
                        hint = f" Hint: Try {possible_values[0]} at ({i+1},{j+1})"
                        self.message_label.configure(text=hint)
                        return
```

```
        self.message_label.configure(text=" Puzzle might already be solved.")

    def explain_move(self):
        self.get_board()
        trace = []

        def trace_solve(board):
            for i in range(9):
                for j in range(9):
                    if board[i][j] == 0:
                        for num in range(1, 10):
                            if is_valid(board, i, j, num):
                                trace.append(f" Trying {num} at ({i+1},{j+1}) - va
                                board[i][j] = num
                                if trace_solve(board):
                                    return True
                                board[i][j] = 0
                                trace.append(f" Backtracking from {num} at ({i+1},
                            else:
                                trace.append(f" {num} at ({i+1},{j+1}) - invalid.")
                        return False
            return True

        trace_solve([row[:] for row in self.board])
        self.explanation_box.configure(state="normal")
        self.explanation_box.delete("1.0", ctk.END)
        self.explanation_box.insert(ctk.END, "\n".join(trace))
        self.explanation_box.configure(state="disabled")
        self.message_label.configure(text=" Explanation ready!")

if __name__ == "__main__":
    app = SudokuGUI()
    app.mainloop()
```

2.3.2 Solver.py :

```
import time

# Global step counter
solve_steps = 0

def is_valid(board, row, col, num):
    if num in board[row]:
        return False
    if num in [board[i][col] for i in range(9)]:
        return False
```

```
    box_row, box_col = row // 3 * 3, col // 3 * 3
    for i in range(3):
        for j in range(3):
            if board[box_row + i][box_col + j] == num:
                return False
    return True

def solve(board):
    global solve_steps
    solve_steps = 0
    start_time = time.time()
    result = _solve(board)
    end_time = time.time()
    return result, solve_steps, round(end_time - start_time, 4)

def _solve(board):
    global solve_steps
    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
                        solve_steps += 1
                        if _solve(board):
                            return True
                        board[row][col] = 0
                return False
    return True


def select_unassigned_variable_mrv(board, domains):
    min_domain_size = 10  # Larger than any possible domain
    selected = None

    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                domain_size = len(domains[row][col])
                if domain_size < min_domain_size:
                    min_domain_size = domain_size
                    selected = (row, col)
    return selected



def csp_backtrack(board, domains):
    global solve_steps
```

```
    empty = select_unassigned_variable_mrv(board, domains)
    if not empty:
        return True  # Solved

    row, col = empty

    for value in domains[row][col]:
        if is_valid(board, row, col, value):
            board[row][col] = value
            solve_steps += 1

            # Create a deep copy of domains to simulate forward checking
            new_domains = [ [list(domains[r][c]) for c in range(9)] for r in range
            new_domains[row][col] = [value]  # Set current cell domain to only the

            result = csp_backtrack(board, new_domains)
            if result:
                return True

            board[row][col] = 0

    return False



def solve_csp(board):
    global solve_steps
    solve_steps = 0
    import time
    start = time.time()

    domains = [[[i for i in range(1, 10)] for _ in range(9)] for _ in range(9)]

    result = csp_backtrack(board, domains)

    end = time.time()
    duration = end - start

    return result, solve_steps, duration
```

2.3.3 utils.py :

```
import random
from solver import solve, is_valid

def generate_board(difficulty='easy'):
    board = [[0] * 9 for _ in range(9)]

    # Randomly fill a few starting cells
    filled = 0
    while filled < 10:
        row, col = random.randint(0, 8), random.randint(0, 8)
        num = random.randint(1, 9)
        if board[row][col] == 0 and is_valid(board, row, col, num):
            board[row][col] = num
            filled += 1

    solve(board)  # Get a fully solved board

    # Decide how many cells to remove based on difficulty
    if difficulty == 'easy':
        cells_to_remove = 30
    elif difficulty == 'medium':
        cells_to_remove = 40
    elif difficulty == 'hard':
        cells_to_remove = 55
    else:
        cells_to_remove = 40  # default to medium

    # Randomly remove cells
    removed = 0
    while removed < cells_to_remove:
        row, col = random.randint(0, 8), random.randint(0, 8)
        if board[row][col] != 0:
            board[row][col] = 0
            removed += 1

    return board
```

Chapter 3

Performance Evaluation

3.1 Simulation Environment/ Simulation Procedure

To evaluate the effectiveness and performance of the AI-based Sudoku Solver, various test environments and scenarios were designed. The application was tested on a standard desktop setup:

- Operating System: Windows 11 / Ubuntu 22.04
- Processor: Intel Core i5 / AMD Ryzen 5 (or equivalent)
- RAM: 8 GB or higher
- Python Version: 3.10+

Libraries Used: tkinter, pyttsx3, speech$_{r}ecognition, threading, timeTheperformancewasevaluatedbasedo$

- Puzzle Difficulty: Easy, Medium, Hard, and Evil-level puzzles
- Solving Time: Time taken by the AI to solve the puzzle from start to finish
- Accuracy: The ability to correctly solve the puzzle without errors
- Response Time: Time taken for hint generation and speech-to-text recognition
- Resource Usage: CPU and memory consumption during execution

3.2 Results Analysis/Testing

3.2.1 Generate :

This Sudoku system offers three difficulty modes: Easy, Medium, and Hard. To start, you select one of the modes depending on your skill level. Once the mode is selected, you press the Generate button. The system then creates a new Sudoku puzzle that matches the selected difficulty. An Easy puzzle will have more pre-filled numbers, while a Hard puzzle will have fewer clues and be more challenging. The generated puzzle appears on the grid, ready to be solved.

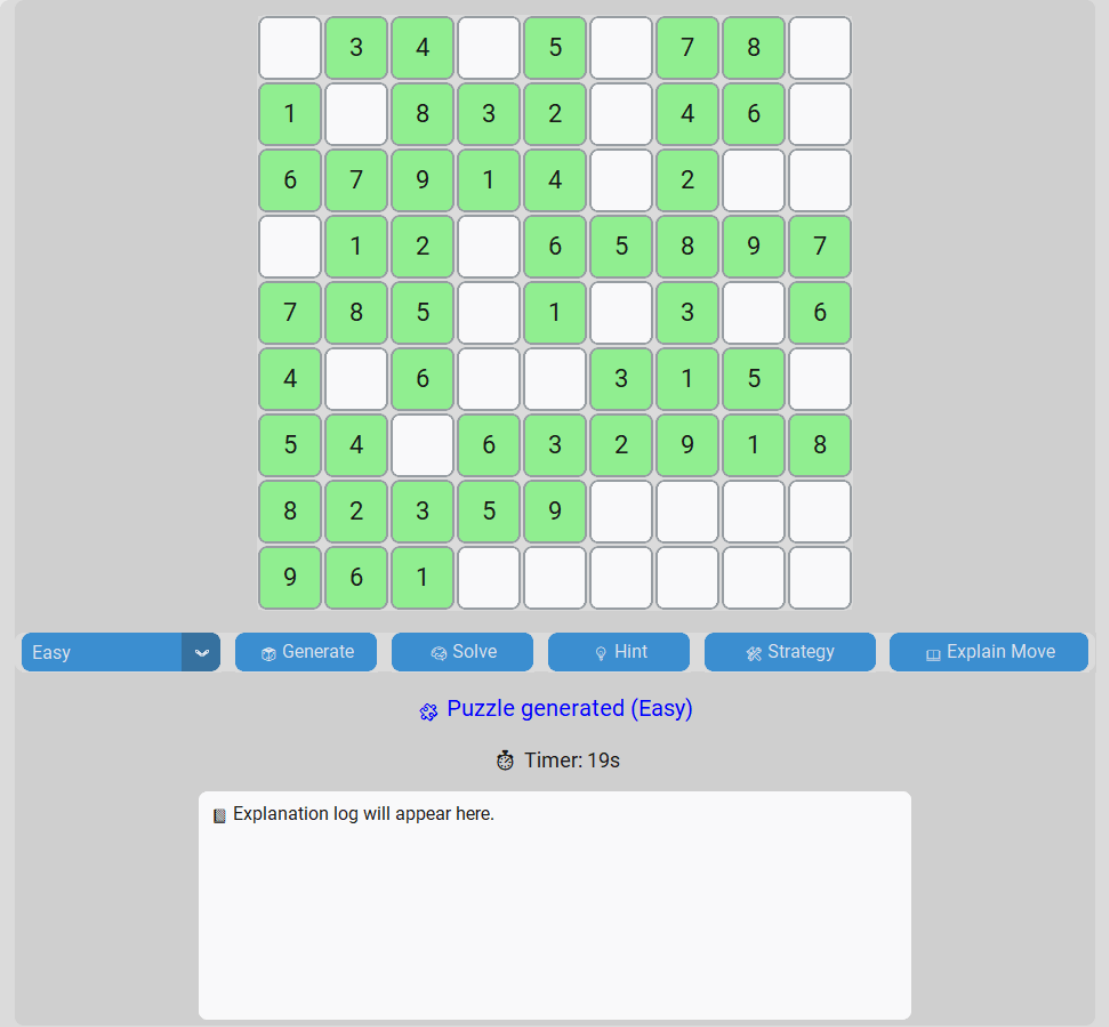

Figure 3.1: Generate

3.2.2 Hint System :

When solving manually, if you're unsure what number to put, pressing the Hint button shows the correct number and its exact position (row and column) to help you continue.

Figure 3.2: Hint System

3.2.3 Automatic Solve system

When the Solve button is pressed, the system automatically completes the Sudoku puzzle using an internal algorithm. It instantly fills all the correct values in the grid, turns the cells green to indicate correctness, and shows a message stating how many steps it took and how much time was required to solve it.

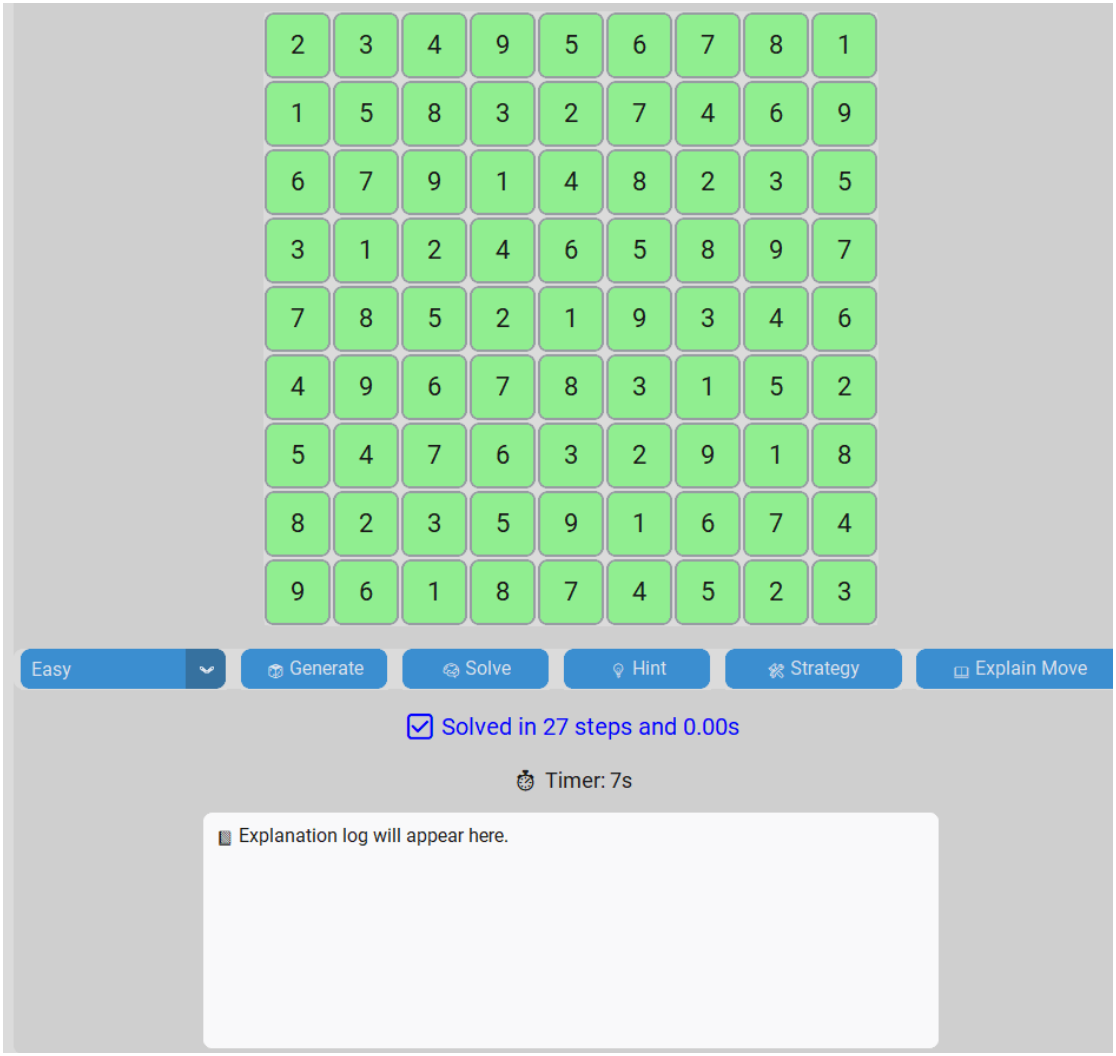

Figure 3.3: Solve button press

3.2.4 Solve Manually

when you solve the Sudoku puzzle manually, it shows a message at the end saying "Congratulations! Puzzle completed!"It also displays the total time taken to solve the puzzle, in this case, 271 seconds.

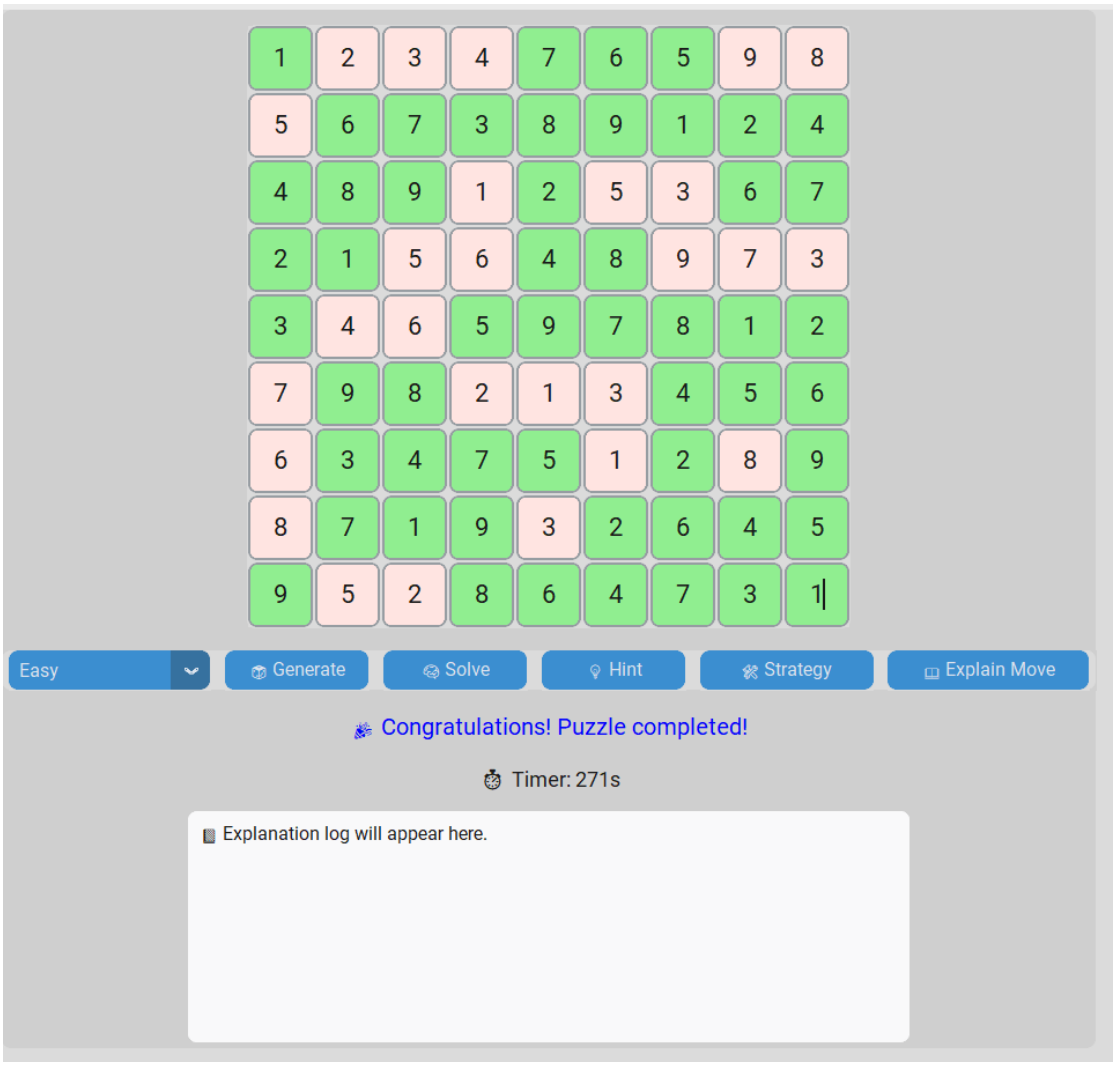

Figure 3.4: Solve manually

Chapter 4

Conclusion

4.1 Discussion

The AI Sudoku Solver accomplishes its goal of offering a clever, user-friendly, and instructive platform for resolving Sudoku puzzles. It combines pedagogical techniques like hinting and logic explanation, contemporary interface elements like speech input and audio feedback, and a traditional AI approach (backtracking with limitations).

4.2 Limitations

The system's present version has certain restrictions despite its functionality:

- Absence of Web/Mobile Support: The program solely uses Python to run on
- No Puzzle Generator: It solely solves Sudoku puzzles; it doesn't create new ones.
- Limited Error Handling: Errors in user input are identified but not thoroughly examined.
- Speech Accuracy in Noisy Environments: Clear speech and a calm environment are necessary for voice input.
- No Save Feature: Neither history nor current progress may be saved.

4.3 Scope of Future Work

There are several ways to grow this project:

- Web and Mobile Version: Making use of frameworks such as Flutter for mobile or React for web.
- Add a random problem generating algorithm that is depending on difficulty.

- Compete with other players to solve problems in multiplayer mode.
- User Profile System: Store individual puzzles, progress, and scores.
- Database Integration: Keep track of user information, problem solutions, and AI judgments.
- Advanced AI Algorithms: Include machine learning to solve problems based on predictions or identify patterns.
- OCR Integration: Enable the scanning of paper Sudoku puzzles with an uploaded image or webcam.