# SUDOKU SOLVER AND GENERATOR

# MICRO PROJECT REPORT

# Submitted by

#### MEGAPRANITHAA M

#### 23ITR096

#### MOUNITH D

#### 23ITR103

#### KARTHIKEYAN R

#### 23ITR081

#### *in partial fulfilment of the requirements*

#### *for the award of the degree*

#### *of*

#### BACHELOR OF TECHNOLOGY

**IN**

**INFORMATION TECHNOLOGY**

## DEPARTMENT OF INFORMATION TECHNOLOGY

<

width1.00” >



# KONGU ENGINEERING COLLEGE

# (Autonomous)

**PERUNDURAI ERODE – 638 060**

**NOVEMBER 2024**

## DEPARTMENT OF INFORIMATION TECHNOLOGY

## KONGU ENGINEERING COLLEGE

**(Autonomous)**

**PERUNDURAI ERODE – 638060**

**NOVEMBER 2024**

**BONAFIED CERTIFICATE**

This is to certify that the Project report entitled Sudoku Solver and Generator is the bonafide record of project work done by **Megapranithaa M (23ITR096), Mounith D (23ITR103)** and **Karthikeyan R (23ITR081)** for **22ITT32 PYTHON PROGRAMMING AND FRAMEWORKS** during the year 2024–2025.

**COURSE IN CHARGE HEAD OF THE DEPARTMENT**

**(Signature with seal)**

Date:

Submitted for the final viva voce examination held on 18.11.2024.

## DEPARTMENT OF INFORMATION TECHNOLOGY

## KONGU ENGINEERING COLLEGE

**(Autonomous)**

**PERUNDURAI ERODE – 638060**

**NOVEMBER 2024**

**DECLARATION**

We affirm that the Project Report titled Sudoku Solver and Generator being submitted in partial fulfilment of the requirements for the award of Bachelor of Technology is the original work carried out by us. It has not formed the part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

**Date:18.11.2024**

**MEGAPRANITHAA M**

**23ITR096**

**MOUNITH D**

**23ITR103**

**KARTHIKEYAN R**

**23ITR081**

I certify that the declaration made by the above candidates is true to the best of my knowledge.

Date: 18.11.2024

Name and Signature of the course in charge

## ABSTRACT

The Sudoku Solver and Generator application is a powerful and user-friendly program designed to solve existing Sudoku puzzles and generate new ones with customizable difficulty levels. The application provides an engaging interface, making it suitable for both casual users and those interested in exploring algorithmic problem-solving techniques.

The solver leverages the **backtracking algorithm**, a recursive approach that efficiently identifies solutions by systematically testing numbers and backtracking to resolve conflicts. This ensures every puzzle is solved accurately and logically. The generator, using randomization techniques, creates puzzles by selectively removing numbers from a pre-filled grid while validating the uniqueness of the solution, thus offering a variety of difficulty levels.

Key libraries enhance the application’s functionality:

* **NumPy** enables efficient handling of grids and operations, crucial for managing the 9x9 puzzle structure.
* **random** facilitates controlled randomization, balancing the challenge and solvability of generated puzzles.
* **Tkinter** provides an intuitive GUI, allowing users to input puzzles, view solutions, and generate new games interactively.
* **Pillow (PIL)** supports saving puzzles as images, enabling users to share or print them.
* **Pygame** adds optional advanced graphical features, such as animations and theme customization, elevating the user experience.

The development environment, **Visual Studio Code (VS Code)**, ensures an efficient coding workflow with robust debugging and testing tools.

This project not only serves as an educational tool for algorithm enthusiasts but also as a comprehensive application for Sudoku lovers. Its modular design allows for future enhancements, such as leaderboards, multiplayer modes, or integration with mobile platforms. Combining computational efficiency with an appealing design, the application exemplifies the synergy between algorithmic precision and user-centered software development.

## ACKNOWLEDGEMENT

## First and foremost, we acknowledge the abundant grace and presence of Almighty throughout different phases of the project and its successful completion.

## We wish to express our gratefulness to our beloved Correspondent Thiru.A.K.ILANGO B.Com., M.B.A., LLB., and all the trust members of Kongu Vellalar Institute of Technology Trust for providing all the necessary facilities to complete the project successfully.

## We express our deep sense of gratitude to our beloved Principal Dr.V.BALUSAMY B.E.(Hons)., M.Tech., Ph.D., for providing us an opportunity to complete the project.

## We express our gratitude to Dr. S. ANANDAMURUGAN M.E., Ph.D., Head of the Department, Department of Information Technology for his valuable suggestions.

## We are highly indebted to Mrs. P. VANITHA B.E., M.E., Department of Information Technology for her valuable supervision and advice for the fruitful completion of the project.

## We are thankful to the faculty members of the Department of Information Technology for their valuable guidance and support.

## TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER No** | **TITLE** | **PAGE No** |

1. **Introduction 1**

**1.1Overview of Sudoku**

**1.2Purpose of the Application**

1. **System Requirements 2**

**2.1 Software Requirements**

**3. Design and Architecture 6  
 3.1 Application Workflow  
 3.2 System Components**

**a.Solver Module**

**bGenerator Module**

**c.Graphical User Interface (GUI)**

**4. Algorithmic Approach 10  
 4.1 Backtracking Algorithm**

**for Solving Sudoku  
 4.2 Randomization Techniques**

**for Generating Puzzles**

** 5. Technology Stack 13  
 5.1 Programming Language: Python  
 5.2 Libraries and Frameworks Used**

**a.NumPy**

**b.random**

**c.Tkinter**

**d.Pillow (PIL)**

**e.Pygame**

**6. Implementation Details 15  
 6.1 Sudoku Solver Implementation  
 6.2 Sudoku Generator Implementation  
 6.3 GUI Development**

**7. Features 18  
 7.1 Solving Existing Puzzles  
 7.2 Generating New Puzzles  
 7.3 Saving and Sharing Puzzles**

**8. Testing and Validation 20  
 8.1 Test Cases for Solver  
 8.2 Validation of Generated Puzzles**

**9. Future Enhancements 23  
 9.1 Adding Leaderboards  
 9.2 Mobile Platform Integration  
 9.3 Multiplayer Mode**

**10. Conclusion 26**

**11. Coding 27**

**12. Snapshots 55**

** 13. Reference 59**

## CHAPTER 1

## INTRODUCTION

**1.1 Overview of Sudoku**  
Sudoku is a popular logic-based number-placement puzzle that has captivated enthusiasts worldwide. The standard 9x9 grid, divided into smaller 3x3 subgrids, challenges players to fill empty cells with digits from 1 to 9. The catch is that each digit must appear only once in every row, column, and subgrid. Sudoku puzzles range in difficulty, testing logical reasoning and pattern recognition skills.

**1.2 Purpose of the Application**  
The Sudoku Solver and Generator application aims to simplify the solving process and create new puzzles for enthusiasts of all skill levels. The solver provides instant solutions to existing puzzles using an efficient backtracking algorithm, while the generator produces puzzles with adjustable difficulty levels to cater to beginners and experts alike.

This application not only enhances the Sudoku experience but also serves as an educational tool for understanding algorithms, matrix operations, and graphical user interface development.

## CHAPTER 2

**SOFTWARE REQUIRMENTS**

2.1 Software Requirements

The Sudoku Solver and Generator application relies on a carefully selected set of software components to ensure optimal performance, compatibility, and user experience. This section elaborates on the software prerequisites required for both development and execution.

1. Operating System

The application is cross-platform and can be developed and executed on the following operating systems:

* Windows: Supports Windows 10 or later versions. Windows provides a stable environment for Python-based applications, and it is widely used for development due to its compatibility with tools like Visual Studio Code.
* macOS: Compatible with macOS Mojave (10.14) or later. macOS offers a Unix-based environment, making it suitable for Python development with native support for essential libraries.
* Linux: Distributions such as Ubuntu (18.04 or newer), Fedora, and others are supported. Linux is particularly preferred by developers for its command-line tools, package management, and efficient resource usage.

2. Programming Language

The application is developed entirely in Python, a high-level, interpreted programming language known for its simplicity and vast library ecosystem. The recommended version is Python 3.9 or later, as it ensures compatibility with the latest features and libraries used in this application.

3. Integrated Development Environment (IDE)

The development process benefits significantly from a robust IDE, enabling efficient coding, debugging, and execution:

* Visual Studio Code (VS Code): The primary IDE recommended for this project due to its lightweight nature, extensive plugin ecosystem, and support for Python extensions like Pylance, linting tools, and IntelliSense.
* Alternative IDEs:
  + PyCharm: Offers advanced debugging and integrated tools for Python developers.
  + Jupyter Notebook: Suitable for exploratory coding, especially during algorithm testing and development.
  + Any Python-Compatible Editor: Editors such as Sublime Text and Atom can also be used with appropriate plugins.

4. Python Libraries

A variety of libraries are integrated into the application to perform specific tasks efficiently:

* NumPy: Essential for handling arrays and matrices, which form the backbone of Sudoku grids. Its fast and memory-efficient operations enable seamless manipulation of the 9x9 grid.
* random: Used for generating random configurations during Sudoku puzzle creation, ensuring diversity in puzzle designs.
* Tkinter: A built-in Python library for GUI development. It simplifies the creation of user interfaces, including puzzle grids, buttons, and interactive input fields.
* Pillow (PIL): Enables image handling features, such as saving puzzles as image files. This is particularly useful for sharing or archiving generated puzzles.
* Pygame (optional): Enhances graphical interactivity, allowing for animations, smooth transitions, and customizable themes for an engaging user experience.

5. Package Management

To install and manage the required libraries, the application uses:

* pip (Python Package Installer): Bundled with Python, pip simplifies the installation of external libraries like NumPy, Pillow, and Pygame.
* Virtual Environments: Tools like venv or conda are highly recommended to isolate project dependencies, preventing conflicts between different Python projects on the same system.

6. Version Control System

* Git: A version control system is crucial for tracking changes, especially when working in teams. Git enables collaborative development, ensuring changes are merged and managed efficiently. Repositories can be hosted on platforms like GitHub or GitLab for backup and teamwork.

7. Command Line/Terminal

A functional command-line interface is essential for executing scripts, installing dependencies, and managing the project environment. Windows Command Prompt, macOS Terminal, or Linux Shell can all serve this purpose.

8. Testing and Debugging Tools

Testing and debugging tools are essential for ensuring the application’s reliability:

* Python’s unittest Framework: Used for creating and running test cases to validate the Sudoku solver and generator functionalities.
* Debugging Plugins for VS Code: Extensions like Python Debugger allow real-time code inspection and troubleshooting.

9. Optional Tools

For enhancing development efficiency:

* Docker: Provides a containerized environment to package the application along with its dependencies, ensuring consistency across different systems.
* Automated Testing Frameworks: Tools like pytest can be integrated for advanced testing scenarios.

## CHAPTER 3

## DESIGN AND ARCHITECTURE

**3. Design and Architecture**

The Sudoku Solver and Generator application is designed with modularity, user-friendliness, and efficiency in mind. The application integrates multiple components that work together to provide a seamless experience for solving and generating Sudoku puzzles. The architecture ensures scalability, allowing easy modifications and enhancements in the future.

**3.1 Application Workflow**

The application workflow is divided into two primary functionalities: solving existing puzzles and generating new ones. Below is a step-by-step breakdown of the workflow:

**1. Input Stage**

* **Solver Module**:
  + The user inputs an incomplete Sudoku puzzle through the graphical user interface (GUI).
  + The application validates the grid to ensure it adheres to Sudoku rules (e.g., numbers range from 1 to 9, no duplicates in rows, columns, or subgrids).
* **Generator Module**:
  + The user selects the desired difficulty level (e.g., Easy, Medium, Hard).
  + The application uses randomization techniques to create a new puzzle grid with pre-filled cells, ensuring that the puzzle has a unique solution.

**2. Processing Stage**

* **Solver Module**:
  + The backtracking algorithm is invoked to solve the puzzle.
  + The algorithm iteratively fills the grid by testing numbers in empty cells and backtracking when conflicts arise.
* **Generator Module**:
  + A complete puzzle grid is initially generated.
  + Numbers are selectively removed based on the difficulty level, ensuring logical solvability.

**3. Output Stage**

* **Solver Module**:
  + The solved puzzle is displayed on the GUI.
  + Optionally, the solution can be saved as an image or printed.
* **Generator Module**:
  + The newly generated puzzle is displayed for the user to play.
  + The user has the option to save, print, or solve the puzzle immediately.

**4. User Interaction**

* The user interacts with the puzzle grid through the GUI, which allows actions like entering numbers, erasing inputs, or resetting the grid.

**3.2 System Components**

The application is composed of three primary components:

**1. Solver Module**

The solver module is responsible for solving user-provided Sudoku puzzles.

* **Core Algorithm**:
  + The backtracking algorithm systematically tries all possible numbers in empty cells.
  + It ensures adherence to Sudoku rules by checking rows, columns, and subgrids for conflicts.
* **Validation**:
  + Input grids are validated before processing to detect errors such as duplicates or invalid numbers.
* **Performance Optimization**:
  + The algorithm uses efficient grid traversal techniques to minimize redundant checks.
  + NumPy arrays are employed for faster operations on the grid.

**2. Generator Module**

The generator module creates new Sudoku puzzles based on user-selected difficulty levels.

* **Grid Initialization**:
  + A complete, valid Sudoku grid is generated using a randomized fill algorithm.
* **Number Removal**:
  + Numbers are strategically removed from the grid while ensuring a single unique solution.
  + Difficulty levels are determined by the number of cells removed and the logical complexity required to solve the puzzle.
* **Validation**:
  + After generating the puzzle, the grid is tested for uniqueness of the solution using the solver module.

**3. Graphical User Interface (GUI)**

The GUI provides an interactive platform for users to input puzzles, view solutions, and generate new puzzles.

* **Framework**:
  + Built using **Tkinter**, the GUI is lightweight and integrates seamlessly with Python’s backend logic.
* **Key Features**:
  + **Grid Display**: A 9x9 interactive grid where users can input and modify numbers.
  + **Buttons and Controls**: Options to solve, reset, or generate puzzles, along with difficulty selection.
  + **Error Feedback**: Highlights invalid entries or conflicts in the grid.
* **Optional Enhancements**:
  + **Themes and Animations**: Using Pygame, the interface can be enhanced with customizable themes, animations, and sound effects.
  + **Export Functionality**: Allows users to save puzzles and solutions as image files using Pillow (PIL).

## CHAPTER 4

## ALGORITHM

**4.1 Backtracking Algorithm for Solving Sudoku**

The backtracking algorithm is a recursive, brute-force approach to solving Sudoku puzzles. It systematically tries to fill the grid by placing numbers and backtracking when a conflict arises.

Steps:

1. **Find an Empty Cell:**
   * Traverse the grid to locate the first empty cell (usually represented by 0 or a blank).
2. **Try a Candidate Number:**
   * For the empty cell, attempt to place a number (1 through 9).
3. **Check Validity:**
   * Ensure the number does not conflict with existing numbers in the:
     + Row.
     + Column.
     + 3x3 subgrid.
4. **Recursive Attempt:**
   * Recursively attempt to solve the rest of the puzzle with the current placement.
   * If a solution is found, return it.
5. **Backtrack if Necessary:**
   * If no valid number can be placed, reset the cell (empty it) and return to the previous step to try another number.
6. **Repeat Until Solved:**
   * The algorithm continues until all cells are filled or no solution exists.

**Advantages:**

* Guarantees finding a solution (if it exists).
* Straightforward implementation.

**Disadvantages:**

* Can be slow for very complex puzzles.
* Inefficient for puzzles with multiple solutions unless optimized.

**4.2 Randomization Techniques for Generating Sudoku Puzzles**

Generating Sudoku puzzles involves creating a valid grid and removing numbers while ensuring the puzzle remains solvable and retains a unique solution.

**Steps:**

1. **Generate a Fully Solved Grid**:
   * Use a **backtracking algorithm** or another method to create a fully populated, valid Sudoku grid.
   * Randomly shuffle numbers (or use a randomized starting pattern) to ensure variety.
2. **Remove Numbers**:
   * Randomly remove cells while maintaining the grid's **uniqueness** (ensuring there’s only one solution).
   * Use **randomization techniques**:
     + Randomly choose cells to empty.
     + Check solvability after each removal.
     + Stop when a desired level of difficulty is achieved.
3. **Validate Difficulty**:
   * Test the puzzle using a solver to ensure it meets the desired difficulty level (easy, medium, hard, etc.).
   * Difficulty can be adjusted by controlling the number of clues (filled cells) or the logical complexity of the solution path.

**Randomization Techniques:**

* **Shuffling**:
  + Shuffle rows and columns within the same band (groups of three rows or columns) to maintain validity.
  + Permute numbers (e.g., swap all instances of 1 with 9) to create variation.
* **Selective Removal**:
  + Use heuristics to remove numbers while keeping the puzzle solvable.

**Advantages:**

* Can generate puzzles of varying difficulty.
* Introduces randomness to avoid repetitive patterns.

**Disadvantages:**

* Ensuring uniqueness can be computationally expensive.
* Requires careful balancing to prevent overly easy or unsolvable puzzles.

**CHAPTER 5**

## TECHNOLOGY STACK

**5.1 Programming Language: Python**

Python is selected for its simplicity, extensive library support, and suitability for implementing algorithms and developing user interfaces. It offers robust tools for rapid prototyping and debugging, making it ideal for a Sudoku application.

**5.2 Libraries and Frameworks Used**

**a. NumPy**  
NumPy is used for efficient manipulation of 2D arrays, which represent the Sudoku grid. It enables quick access to matrix operations for validating rows, columns, and subgrids. NumPy's speed and flexibility make it a vital component for handling numerical data in the application.

**b. random**  
The random module introduces randomness in generating Sudoku puzzles. It is used for shuffling numbers, rows, or columns to ensure variation and uniqueness in the puzzles.

**c. Tkinter**  
Tkinter is utilized to build the graphical user interface (GUI) for the application. It allows the creation of an interactive Sudoku interface with widgets like buttons, text boxes, and labels. Its simplicity makes it a popular choice for developing desktop applications.

**d. Pillow (PIL)**  
Pillow (PIL) is employed for image processing and customization within the application. It enables the creation of visual elements such as gridlines, numbers, and highlighted cells, enhancing the visual appeal of the Sudoku interface.

**e. Pygame**  
Pygame is used to design a more interactive and visually dynamic Sudoku experience. It provides advanced capabilities for graphics, animations, and event handling, making the interface engaging and responsive.

## CHAPTER 6

## IMPLEMENTATION

**6.1 Sudoku Solver Implementation**

The **Sudoku Solver** is implemented using the **backtracking algorithm**. The backtracking approach works by trying different possibilities and backtracking when an invalid state is encountered. Here's how the solution is structured:

1. **Input Representation**: The Sudoku puzzle is represented as a 9x9 grid, where each cell can either contain a number (1-9) or be empty (represented by 0 or null).
2. **Recursive Algorithm**: The solver recursively attempts to fill each empty cell with numbers from 1 to 9. After placing a number, it checks if the current configuration is valid by ensuring that:
   * No number is repeated in the same row.
   * No number is repeated in the same column.
   * No number is repeated in the same 3x3 subgrid.
3. **Backtracking Process**: If a number placement is valid, the algorithm moves to the next empty cell and repeats the process. If no valid number can be placed, the algorithm backtracks by removing the last placed number and tries the next possible number.
4. **Termination**: The process continues until the grid is completely filled, indicating that the puzzle is solved. If the algorithm exhausts all possibilities and cannot find a valid number for an empty cell, the puzzle has no solution.

**6.2 Sudoku Generator Implementation**

The **Sudoku Generator** creates random puzzles by first generating a fully solved Sudoku grid and then removing numbers to form the puzzle. The key steps in the generation process include:

1. **Grid Generation**: A fully solved Sudoku grid is generated using the same **backtracking algorithm** that is used in the solver. This ensures that the grid is valid and follows all Sudoku rules.
2. **Random Removal**: After generating a solved grid, the generator randomly removes numbers from the grid to create a puzzle. This is done by selecting random cells and setting them to zero while checking that the puzzle still has a unique solution after each removal.
3. **Solvability Check**: After each number removal, the generator ensures that the puzzle still has a unique solution by running the solver algorithm on the partially filled grid. If the grid has no solution or multiple solutions, the algorithm will attempt to remove fewer numbers or adjust the puzzle until a valid puzzle is created.
4. **Difficulty Level**: The difficulty of the generated puzzle is controlled by adjusting the number of given clues (the numbers that are already placed in the grid). Fewer clues generally make the puzzle harder, while more clues make it easier. Additionally, the complexity of the logical steps required to solve the puzzle can also be used to adjust the difficulty.

**6.3 GUI Development**

The **Graphical User Interface (GUI)** for the Sudoku application is built using **Tkinter** for its simplicity and ease of use. The GUI includes the following key components:

1. **Grid Layout**: The Sudoku grid is displayed as a 9x9 grid of cells. Each cell can either contain a number or be empty. Empty cells are represented by editable fields where the user can input values.
2. **Buttons**: There are buttons for:
   * **Solving the Puzzle**: A button that triggers the solver algorithm to fill in the missing numbers.
   * **Generating a Puzzle**: A button that triggers the puzzle generator to create a new Sudoku puzzle.
   * **Clearing the Grid**: A button to reset the grid to its initial state, allowing the user to input new values or start over.
3. **User Interaction**: Users can input numbers manually into the grid. The system validates inputs to ensure that they follow Sudoku rules, and provides visual feedback for incorrect entries (e.g., highlighting incorrect numbers).
4. **Styling and Visuals**: The design is kept simple yet visually appealing. The grid is neatly bordered with different colors for rows, columns, and 3x3 subgrids to improve readability. Additionally, the numbers are displayed in bold, and error cells can be highlighted in red to alert the user of mistakes.
5. **Visual Feedback**: The solver highlights the cells it is filling as it progresses, giving users a visual sense of the solving process. The grid also dynamically updates when the user inputs values.
6. **Event Handling**: The GUI includes handling for mouse and keyboard events, such as clicking on a grid cell to input a number or clicking a button to solve the puzzle.

**CHAPTER 7**

**FEATURES**

**7.1 Solving Existing Puzzles**

This feature allows users to input and solve Sudoku puzzles, either manually or by loading pre-defined grids. Key aspects include:

* **Manual Input**: Users can input their puzzles directly into the 9x9 grid through an intuitive interface.
* **Automatic Solver**: The backtracking algorithm quickly solves the entered puzzle, displaying the completed grid.
* **Error Detection**: If the input grid violates Sudoku rules (e.g., duplicates in rows, columns, or subgrids), the system provides immediate feedback by highlighting problematic cells.
* **Progress Visualization**: The solving process can be visualized step-by-step, allowing users to understand how the algorithm works.

**7.2 Generating New Puzzles**

This feature enables the creation of new Sudoku puzzles with adjustable difficulty levels. Key functionalities include:

* **Random Puzzle Generation**: The system generates a fully solved grid and removes numbers to create a valid puzzle with a unique solution.
* **Difficulty Levels**: Users can choose between levels such as Easy, Medium, and Hard, which determine the number of given clues and the complexity of the logical steps required to solve the puzzle.
* **Variety**: Randomization techniques ensure that each puzzle is unique, avoiding repetitive patterns.
* **Preview Option**: Generated puzzles can be previewed before saving or sharing.

**7.3 Saving and Sharing Puzzles**

Users can save their progress or share puzzles with others for collaboration or challenges. Key functionalities include:

* **Save Feature**: Puzzles, whether solved or unsolved, can be saved to a file or database for later use.
* **Load Feature**: Previously saved puzzles can be reloaded into the application for continued solving or review.
* **Export as Image or File**: Puzzles can be exported as image files (using Pillow) or text-based formats for sharing.
* **Sharing Options**: Users can share puzzles via email or social media platforms directly from the application.
* **QR Code Integration (Optional)**: A QR code representing the puzzle can be generated for easy sharing and scanning.

**CHAPTER 8**

**TESTING AND VALIDATION**

**8.1 Test Cases for Solver**

The Sudoku solver undergoes rigorous testing to ensure correctness, efficiency, and robustness. Key test cases include:

1. **Basic Valid Puzzle**:
   * **Input**: A partially filled valid Sudoku puzzle.
   * **Expected Output**: The solver fills the grid correctly, adhering to Sudoku rules.
2. **Completed Puzzle**:
   * **Input**: A fully solved Sudoku puzzle.
   * **Expected Output**: The solver identifies the puzzle as already solved and makes no changes.
3. **Invalid Puzzle**:
   * **Input**: A puzzle with conflicts (e.g., duplicate numbers in a row, column, or subgrid).
   * **Expected Output**: The solver detects the conflict and indicates that the puzzle is unsolvable.
4. **Empty Puzzle**:
   * **Input**: An empty 9x9 grid.
   * **Expected Output**: The solver generates a valid solution, filling all cells correctly.
5. **Edge Cases**:
   * **Input**: Puzzles with minimal clues (e.g., only 17 clues, the theoretical minimum for a unique solution).
   * **Expected Output**: The solver successfully completes the puzzle within a reasonable time.
6. **Performance Testing**:
   * **Input**: Complex puzzles with high computational requirements.
   * **Expected Output**: The solver completes within acceptable time limits.

**8.2 Validation of Generated Puzzles**

Validation ensures that generated Sudoku puzzles are unique, solvable, and aligned with the specified difficulty level. Key validation steps include:

1. **Uniqueness Check**:
   * **Method**: After generating a puzzle, the solver algorithm runs to verify that there is exactly one solution.
   * **Outcome**: If multiple solutions are detected, adjustments are made to the puzzle until uniqueness is guaranteed.
2. **Solvability Check**:
   * **Method**: The generated puzzle is tested with the solver to confirm it can be completed without violating Sudoku rules.
   * **Outcome**: The puzzle is accepted only if it is solvable.
3. **Difficulty Level Verification**:
   * **Method**: The puzzle is analyzed for the complexity of logical steps required to solve it, ensuring alignment with the chosen difficulty level.
   * **Outcome**: Puzzles are classified into Easy, Medium, or Hard based on the number of clues and logical complexity.
4. **Randomness Test**:
   * **Method**: Multiple puzzles are generated and checked for repetition or predictable patterns.
   * **Outcome**: The system avoids producing repetitive or overly similar puzzles.
5. **Stress Testing**:
   * **Method**: The generator is tasked with creating a large batch of puzzles to test stability and efficiency.
   * **Outcome**: The generator consistently produces valid and unique puzzles without errors.

**CHAPTER 9**

**FUTURE ENHANCEMENT**

**9.1 Adding Leaderboards**

**Objective: Introduce a competitive element by tracking and displaying user performance.**

* **Features:**
  + **Track metrics such as time taken to solve puzzles, difficulty levels completed, and accuracy.**
  + **Display a global leaderboard for all users and a local leaderboard for individual players.**
  + **Include options to filter scores by date, difficulty, or region.**
* **Benefits:**
  + **Encourages engagement and competition among players.**
  + **Provides users with goals to improve their performance.**
  + **Fosters a sense of community within the application.**

**9.2 Mobile Platform Integration**

**Objective: Extend the application's accessibility by developing mobile versions for iOS and Android.**

* **Features:**
  + **Cross-platform compatibility, allowing seamless synchronization of progress between desktop and mobile devices.**
  + **Optimized touch-screen controls for intuitive grid interactions, such as tapping to input numbers and swiping for navigation.**
  + **Offline mode for solving puzzles without an internet connection.**
* **Benefits:**
  + **Expands the user base by catering to on-the-go users.**
  + **Enhances convenience, making Sudoku accessible anytime, anywhere.**
  + **Boosts user retention by integrating with popular app stores.**

**9.3 Multiplayer Mode**

**Objective: Introduce a collaborative and competitive mode for real-time multiplayer Sudoku challenges.**

* **Features:**
  + **Competitive Mode:**
    - **Players compete to solve the same puzzle simultaneously, with live updates showing progress.**
    - **Scoring based on speed and accuracy.**
  + **Collaborative Mode:**
    - **Multiple players work together on a single puzzle, sharing inputs in real time.**
    - **Chat or emoji reactions for in-game communication.**
  + **Matchmaking System:**
    - **Pair players based on skill level or difficulty preferences.**
* **Benefits:**
  + **Adds social interaction, making the game more engaging.**
  + **Encourages teamwork and friendly competition.**
  + **Increases user activity and session duration.**

**CONCLUSION**

The development of the Sudoku application combines powerful algorithms, user-friendly interfaces, and robust validation processes to deliver a comprehensive tool for Sudoku enthusiasts. The solver, built on a backtracking algorithm, ensures accurate and efficient puzzle-solving. The generator produces unique puzzles with adjustable difficulty levels, catering to a wide range of users. The GUI enhances user interaction by providing an intuitive and visually appealing platform for solving and creating puzzles.

By implementing advanced features such as saving, sharing, and puzzle customization, the application offers versatility and convenience. Rigorous testing ensures the reliability and quality of both the solver and the generator. Future enhancements, including leaderboards, mobile integration, and multiplayer modes, aim to broaden accessibility, foster competition, and encourage social interaction.

This project demonstrates the effective integration of algorithms, technology, and design to create a functional, engaging, and scalable Sudoku solution for users of all skill levels.

**CODING**

import tkinter as tk

from tkinter import messagebox

from PIL import Image, ImageTk

import os

import random

import mysql.connector

from tkinter import PhotoImage

from copy import deepcopy

# Helper function to handle DB connection

def connect\_to\_db():

try:

db = mysql.connector.connect(

host="localhost",

user="root",

password="",

database="sudoku"

)

return db

except mysql.connector.Error as err:

messagebox.showerror("Error", f"Database connection error: {err}")

return None

# Main Page for launching the app

class SudokuFrontPage:

def \_init\_(self):

self.root = tk.Tk()

self.root.title("Sudoku")

self.bg\_color = "#f0e68c"

self.root.configure(bg=self.bg\_color)

self.root.attributes("-fullscreen", True)

# Load and resize background image

self.background\_image = Image.open(r"C:\Users\Mounith\Desktop\study\Project\rosy\sudu01.png")

self.background\_image = self.background\_image.resize((self.root.winfo\_screenwidth(), self.root.winfo\_screenheight()))

self.background\_image = ImageTk.PhotoImage(self.background\_image)

# Create a Canvas to display the background image

self.canvas = tk.Canvas(self.root, width=self.root.winfo\_screenwidth(), height=self.root.winfo\_screenheight())

self.canvas.pack(fill="both", expand=True)

self.canvas.create\_image(0, 0, image=self.background\_image, anchor="nw")

# Display text directly on the canvas

self.canvas.create\_text(self.root.winfo\_screenwidth() // 2, self.root.winfo\_screenheight() // 12,

text="Welcome to Sudoku!", font=("Arial", 36, "bold"), fill="black")

# Buttons

self.play\_button = tk.Button(self.root, text="Play", font=("Arial", 20), bg="#4CAF50", fg="white", command=self.show\_login\_page)

self.quit\_button = tk.Button(self.root, text="Quit", bg="red", fg="white", font=("Arial", 20), command=self.root.quit)

# Pack buttons after canvas

self.play\_button.place(x=self.root.winfo\_screenwidth() // 2 - 70, y=self.root.winfo\_screenheight() // 3+50)

self.quit\_button.place(x=self.root.winfo\_screenwidth() // 2 - 70, y=self.root.winfo\_screenheight() // 2 + 60)

self.root.mainloop()

def show\_login\_page(self):

self.root.withdraw()

LoginPage(self.root)

# Login Page class

class LoginPage:

def \_init\_(self, root):

self.root = root

self.login\_window = tk.Toplevel(self.root)

self.login\_window.title("Login Page")

self.login\_window.configure(bg="#f0e68c")

self.login\_window.attributes("-fullscreen", True)

# Load background image

self.bg\_image = PhotoImage(file="sudu02.png") # Replace with the actual path

self.bg\_label = tk.Label(self.login\_window, image=self.bg\_image)

self.bg\_label.place(relwidth=1, relheight=1) # Make background fill entire window

# Create a frame for better layout management

frame = tk.Frame(self.login\_window, bg="black")

frame.pack(expand=True)

# Username label and entry

self.username\_label = tk.Label(frame, text="Username", font=("Arial", 14), bg="#f0e68c")

self.username\_label.pack(pady=10)

self.username\_entry = tk.Entry(frame, font=("Arial", 14))

self.username\_entry.pack(pady=10)

# Password label and entry

self.password\_label = tk.Label(frame, text="Password", font=("Arial", 14), bg="#f0e68c")

self.password\_label.pack(pady=10)

self.password\_entry = tk.Entry(frame, show="\*", font=("Arial", 14))

self.password\_entry.pack(pady=10)

# Login button

self.login\_button = tk.Button(frame, text="Login", font=("Arial", 14), command=self.login)

self.login\_button.pack(pady=10)

# Sign Up button

self.signup\_button = tk.Button(frame, text="Sign Up", font=("Arial", 14), command=self.show\_signup\_page)

self.signup\_button.pack(pady=10)

# Back button

self.back\_button = tk.Button(frame, text="Back", font=("Arial", 14), command=self.login\_window.quit)

self.back\_button.pack(pady=10)

def login(self):

username = self.username\_entry.get()

password = self.password\_entry.get()

db = connect\_to\_db()

if db is None:

return

cursor = db.cursor()

cursor.execute("SELECT \* FROM users WHERE username = %s AND password = %s", (username, password))

user = cursor.fetchone()

if user:

messagebox.showinfo("Success", "Login successful!")

self.login\_window.destroy()

self.start\_game(username) # Pass the username to the game

else:

messagebox.showerror("Error", "Invalid username or password.")

cursor.close()

db.close()

def show\_signup\_page(self):

self.login\_window.withdraw()

SignupPage(self.root)

def start\_game(self, username): # Accept username parameter

self.root.withdraw() # Hide the main window

game = SudokuGame(self.root, username) # Pass the username to SudokuGame

game.difficulty\_selection()

# Signup Page class

class SignupPage:

def \_init\_(self, root):

self.root = root

self.signup\_window = tk.Toplevel(self.root)

self.signup\_window.title("Signup Page")

self.signup\_window.configure(bg="#f0e68c")

self.signup\_window.attributes("-fullscreen", True)

# Load background image for design enhancement

self.bg\_image = PhotoImage(file="sudu03.png") # Replace with the actual path to your background image

self.bg\_label = tk.Label(self.signup\_window, image=self.bg\_image)

self.bg\_label.place(relwidth=1, relheight=1) # Make background fill the entire window

# Create a frame for form styling

frame = tk.Frame(self.signup\_window, bg="#f0e68c")

frame.place(relx=0.5, rely=0.4, anchor="center")

# Title label for the signup page

self.title\_label = tk.Label(frame, text="Create a New Account", font=("Arial", 24, "bold"), bg="#f0e68c", fg="blue")

self.title\_label.grid(row=0, column=0, columnspan=2, pady=20)

# Username label and entry field

self.username\_label = tk.Label(frame, text="Username", font=("Arial", 14), bg="#f0e68c", fg="black")

self.username\_label.grid(row=1, column=0, padx=10, pady=5, sticky="e")

self.username\_entry = tk.Entry(frame, font=("Arial", 14), width=25, bd=2, relief="solid")

self.username\_entry.grid(row=1, column=1, padx=10, pady=5)

# Password label and entry field

self.password\_label = tk.Label(frame, text="Password", font=("Arial", 14), bg="#f0e68c", fg="black")

self.password\_label.grid(row=2, column=0, padx=10, pady=5, sticky="e")

self.password\_entry = tk.Entry(frame, show="\*", font=("Arial", 14), width=25, bd=2, relief="solid")

self.password\_entry.grid(row=2, column=1, padx=10, pady=5)

# Confirm Password label and entry field

self.confirm\_password\_label = tk.Label(frame, text="Confirm Password", font=("Arial", 14), bg="#f0e68c", fg="black")

self.confirm\_password\_label.grid(row=3, column=0, padx=10, pady=5, sticky="e")

self.confirm\_password\_entry = tk.Entry(frame, show="\*", font=("Arial", 14), width=25, bd=2, relief="solid")

self.confirm\_password\_entry.grid(row=3, column=1, padx=10, pady=5)

# Sign Up Button

self.signup\_button = tk.Button(frame, text="Sign Up", font=("Arial", 14, "bold"), bg="#4CAF50", fg="white", command=self.signup)

self.signup\_button.grid(row=4, column=0, columnspan=2, pady=20)

# Back to Login Button

self.back\_button = tk.Button(frame, text="Back to Login", font=("Arial", 14), command=self.go\_back\_to\_login, bg="orange", fg="black")

self.back\_button.grid(row=5, column=0, columnspan=2, pady=10)

def signup(self):

username = self.username\_entry.get()

password = self.password\_entry.get()

confirm\_password = self.confirm\_password\_entry.get()

if password != confirm\_password:

messagebox.showerror("Error", "Passwords do not match!")

return

db = connect\_to\_db()

if db is None:

return

cursor = db.cursor()

cursor.execute("SELECT \* FROM users WHERE username = %s", (username,))

existing\_user = cursor.fetchone()

if existing\_user:

messagebox.showerror("Error", "Username already exists!")

return

cursor.execute("INSERT INTO users (username, password) VALUES (%s, %s)", (username, password))

db.commit()

messagebox.showinfo("Success", "Account created successfully!")

self.signup\_window.destroy()

LoginPage(self.root)

cursor.close()

db.close()

def go\_back\_to\_login(self):

self.signup\_window.destroy()

LoginPage(self.root)

# Main Sudoku Game

class SudokuGame:

def \_init\_(self, root, username):

self.root = root

self.username=username

self.game\_window = None

self.board = None

self.solved\_board = None

self.original\_board = None

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

self.selected\_difficulty = None

self.bg\_color = "#f0e68c"

self.fixed\_bg\_color = "white"

self.fixed\_text\_color = "black"

self.entry\_bg\_color = "white"

self.entry\_text\_color = "black"

self.entry\_t\_color = "green"

self.button\_color = "#4CAF50"

self.time\_elapsed = 0

self.timer\_running = False

self.time\_limit = 3600 # 1 hour = 3600 seconds

self.score = 0

self.high\_score = 0

def update\_high\_score\_in\_db(self):

db = connect\_to\_db()

if db is None:

return

cursor = db.cursor()

cursor.execute("SELECT score FROM users WHERE username = %s", (self.username,))

user = cursor.fetchone()

if user:

# User exists, update the high score if necessary

current\_high\_score = user[0]

if self.score > current\_high\_score:

cursor.execute("UPDATE users SET score = %s WHERE username = %s", (self.score, self.username))

db.commit()

messagebox.showinfo("Success", "High score updated!")

else:

messagebox.showerror("Error", "User not found in the database.")

cursor.close()

db.close()

def is\_valid(self, board, row, col, num):

for i in range(9):

if board[row][i] == num or board[i][col] == num:

return False

box\_row\_start = (row // 3) \* 3

box\_col\_start = (col // 3) \* 3

for i in range(3):

for j in range(3):

if board[box\_row\_start + i][box\_col\_start + j] == num:

return False

return True

def solve\_sudoku(self, board):

for row in range(9):

for col in range(9):

if board[row][col] == 0:

shuffled\_numbers = random.sample(range(1, 10), 9)

for num in shuffled\_numbers:

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

board[row][col] = num

if self.solve\_sudoku(board):

return True

board[row][col] = 0

return False

return True

def create\_solved\_board(self):

board = [[0]\*9 for \_ in range(9)]

self.solve\_sudoku(board)

return board

def remove\_cells(self, board, num\_cells):

cells\_removed = 0

while cells\_removed < num\_cells:

row = random.randint(0, 8)

col = random.randint(0, 8)

if board[row][col] != 0:

board[row][col] = 0

cells\_removed += 1

def generate\_sudoku(self, difficulty):

self.solved\_board = self.create\_solved\_board()

self.board = deepcopy(self.solved\_board)

if difficulty == "easy":

num\_cells\_removed = 20

elif difficulty == "medium":

num\_cells\_removed = 40

elif difficulty == "hard":

num\_cells\_removed = 55

self.remove\_cells(self.board, num\_cells\_removed)

self.original\_board = deepcopy(self.board)

def display\_board(self):

center\_frame = tk.Frame(self.game\_window, bg="black")

center\_frame.place(relx=0.5, rely=0.4, anchor="center")

for i in range(9):

for j in range(9):

e = tk.Entry(center\_frame, width=4, justify="center", font=("Arial", 14))

self.entries[i][j] = e

if self.board[i][j] != 0:

e.insert(0, self.board[i][j])

e.config(state="readonly", disabledbackground=self.fixed\_bg\_color, disabledforeground=self.fixed\_text\_color)

else:

e.config(bg=self.entry\_bg\_color, fg=self.entry\_text\_color)

e.grid(row=i, column=j, padx=2, pady=2)

def hint(self):

empty\_cells = [(i, j) for i in range(9) for j in range(9) if self.board[i][j] == 0]

if empty\_cells:

row, col = random.choice(empty\_cells)

self.board[row][col] = self.solved\_board[row][col]

self.entries[row][col].insert(0, self.solved\_board[row][col])

self.entries[row][col].config(state="readonly", disabledbackground=self.fixed\_bg\_color, disabledforeground=self.fixed\_text\_color)

def check\_puzzle(self):

correct = True

for i in range(9):

for j in range(9):

if self.board[i][j] == 0:

user\_input = self.entries[i][j].get()

if not user\_input.isdigit() or int(user\_input) != self.solved\_board[i][j]:

correct = False

self.entries[i][j].config(bg="lightcoral")

else:

self.score += 1

self.entries[i][j].config(bg="lightgreen")

if correct:

self.stop\_timer()

if self.score > self.high\_score:

self.high\_score = self.score

messagebox.showinfo("Success", f"Puzzle solved correctly! Your score: {self.score}\nHigh Score: {self.high\_score}")

# Update the high score in the database

self.update\_high\_score\_in\_db()

# Generate the next puzzle

self.next\_puzzle()

def next\_puzzle(self):

"""Generate and display a new puzzle after the current one is solved."""

self.reset\_puzzle() # Reset the current puzzle

self.generate\_sudoku(self.selected\_difficulty) # Generate a new puzzle

self.display\_board() # Display the new puzzle

self.score = 0 # Reset score for the new puzzle

self.start\_timer() # Restart the timer

def reveal\_puzzle(self):

for i in range(9):

for j in range(9):

self.board[i][j] = self.solved\_board[i][j]

self.entries[i][j].delete(0, tk.END)

self.entries[i][j].insert(0, self.solved\_board[i][j])

self.entries[i][j].config(state="readonly", disabledbackground=self.fixed\_bg\_color, disabledforeground=self.fixed\_text\_color)

def reset\_puzzle(self):

self.board = deepcopy(self.original\_board)

self.score = 0

for i in range(9):

for j in range(9):

self.entries[i][j].config(state="normal", bg=self.entry\_bg\_color, fg=self.entry\_text\_color)

self.entries[i][j].delete(0, tk.END)

if self.board[i][j] != 0:

self.entries[i][j].insert(0, self.board[i][j])

self.entries[i][j].config(state="readonly", disabledbackground=self.fixed\_bg\_color, disabledforeground=self.fixed\_text\_color)

def setup\_buttons(self):

button\_frame = tk.Frame(self.game\_window, bg="black")

button\_frame.place(relx=0.5, rely=0.8, anchor="center")

tk.Button(button\_frame, text="Hint", bg="grey", fg="white", font=("Arial", 14), command=self.hint).grid(row=0, column=0, padx=10)

tk.Button(button\_frame, text="Check Puzzle", bg="grey", fg="white", font=("Arial", 14), command=self.check\_puzzle).grid(row=0, column=1, padx=10)

tk.Button(button\_frame, text="Reveal Puzzle", bg="grey", fg="white", font=("Arial", 14), command=self.reveal\_puzzle).grid(row=0, column=2, padx=10)

tk.Button(button\_frame, text="Reset Puzzle", bg="grey", fg="white", font=("Arial", 14), command=self.reset\_puzzle).grid(row=0, column=3, padx=10)

tk.Button(button\_frame, text="Back", bg="grey", fg="white", font=("Arial", 14), command=self.go\_back\_to\_difficulty\_selection).grid(row=0, column=4, padx=10)

tk.Button(button\_frame, text="Save Solved Board", bg="grey", fg="white", font=("Arial", 14), command=self.save\_solved\_board).grid(row=0, column=5, padx=10)

def start\_timer(self):

self.timer\_running = True

self.update\_timer()

def stop\_timer(self):

self.timer\_running = False

def update\_timer(self):

if self.timer\_running:

if self.time\_elapsed < self.time\_limit:

minutes, seconds = divmod(self.time\_elapsed, 60)

self.timer\_label.config(text=f"Time Elapsed: {minutes:02}:{seconds:02} | Score: {self.score} | High Score: {self.high\_score}",bg="grey")

self.time\_elapsed += 1

self.game\_window.after(1000, self.update\_timer)

else:

self.stop\_timer()

messagebox.showinfo("Time Up", "1 hour has passed! Time's up!")

def start\_game(self):

if self.selected\_difficulty:

self.game\_window = tk.Toplevel(self.root) # Create a new window for the game

self.game\_window.title("Sudoku Game")

self.game\_window.configure(bg="black")

self.game\_window.attributes("-fullscreen", True) # Set to fullscreen

self.background\_image = Image.open(r"C:\Users\Mounith\Desktop\study\Project\rosy\sudu3.jpg") # Update path as needed

self.background\_image = self.background\_image.resize((self.root.winfo\_screenwidth(), self.root.winfo\_screenheight()))

self.background\_image = ImageTk.PhotoImage(self.background\_image)

background\_label = tk.Label(self.game\_window, image=self.background\_image)

background\_label.place(relwidth=1, relheight=1)

username\_label = tk.Label(self.game\_window, text=f"Welcome, {self.username}!", font=("Arial", 14), bg="black", fg="white")

username\_label.place(relx=0.5, rely=0.05, anchor="center")

self.generate\_sudoku(self.selected\_difficulty)

self.display\_board()

self.setup\_buttons()

self.timer\_label = tk.Label(self.game\_window, bg="grey", fg="black", font=("Arial", 14))

self.timer\_label.place(relx=0.5, rely=0.1, anchor="center")

self.start\_timer()

def set\_difficulty(self, difficulty):

self.selected\_difficulty = difficulty

self.start\_game()

def difficulty\_selection(self):

# Create a new window for difficulty selection

self.game\_window = tk.Toplevel(self.root)

self.game\_window.title("Sudoku - Choose Difficulty")

self.game\_window.configure(bg=self.bg\_color)

self.game\_window.attributes("-fullscreen", True)

# Load and resize the background image

self.background\_image = Image.open(r"C:\Users\Mounith\Desktop\study\Project\rosy\sudu7.png")

self.background\_image = self.background\_image.resize(

(self.root.winfo\_screenwidth(), self.root.winfo\_screenheight())

)

self.background\_image = ImageTk.PhotoImage(self.background\_image)

# Create a Canvas to display the background image and text

self.canvas = tk.Canvas(self.game\_window, bg=self.bg\_color, bd=0, highlightthickness=0)

self.canvas.place(relwidth=1, relheight=1) # Fill the entire window

self.canvas.create\_image(0, 0, image=self.background\_image, anchor="nw")

# Ensure window size is available after it is rendered

self.game\_window.after(100, self.add\_text\_and\_buttons)

def add\_text\_and\_buttons(self):

# Display the text "Choose difficulty level"

center\_x = self.game\_window.winfo\_width() // 2

center\_y = self.game\_window.winfo\_height() // 2

self.canvas.create\_text(center\_x, center\_y - 400, text="Choose difficulty level", font=("Arial", 24), fill=self.entry\_t\_color)

self.canvas.create\_text(center\_x, center\_y - 100, text="Test your logic and problem-solving skills!", font=("Arial", 16), fill=self.entry\_t\_color)

# Difficulty buttons

button\_width = 10

tk.Button(self.game\_window, text="Easy", font=("Arial", 14), width=button\_width, command=lambda: self.set\_difficulty("easy")).place(relx=0.2, rely=0.5, anchor="center")

tk.Button(self.game\_window, text="Medium", font=("Arial", 14), width=button\_width, command=lambda: self.set\_difficulty("medium")).place(relx=0.5, rely=0.5, anchor="center")

tk.Button(self.game\_window, text="Hard", font=("Arial", 14), width=button\_width, command=lambda: self.set\_difficulty("hard")).place(relx=0.8, rely=0.5, anchor="center")

# Quit button

tk.Button(self.game\_window, text="Quit", bg="red", font=("Arial", 14), width=button\_width, command=self.root.quit).place(relx=0.5, rely=0.8, anchor="center")

def go\_back\_to\_difficulty\_selection(self):

self.game\_window.destroy()

self.difficulty\_selection()

def save\_solved\_board(self):

try:

# Generate a unique filename

documents\_folder = os.path.expanduser("~/Documents/sudoku")

base\_filename = "sudoku"

file\_extension = ".txt"

count = 1

# Find the next available filename

while True:

file\_path = os.path.join(documents\_folder, f"{base\_filename}{count}{file\_extension}")

if not os.path.exists(file\_path):

break

count += 1

# Save the solved board to the generated filename

with open(file\_path, "w") as f:

for row in self.solved\_board:

f.write(" ".join(map(str, row)) + "\n")

messagebox.showinfo("Saved", f"Solved Sudoku board saved to '{file\_path}'")

except Exception as e:

messagebox.showerror("Error", f"An error occurred while saving: {e}")

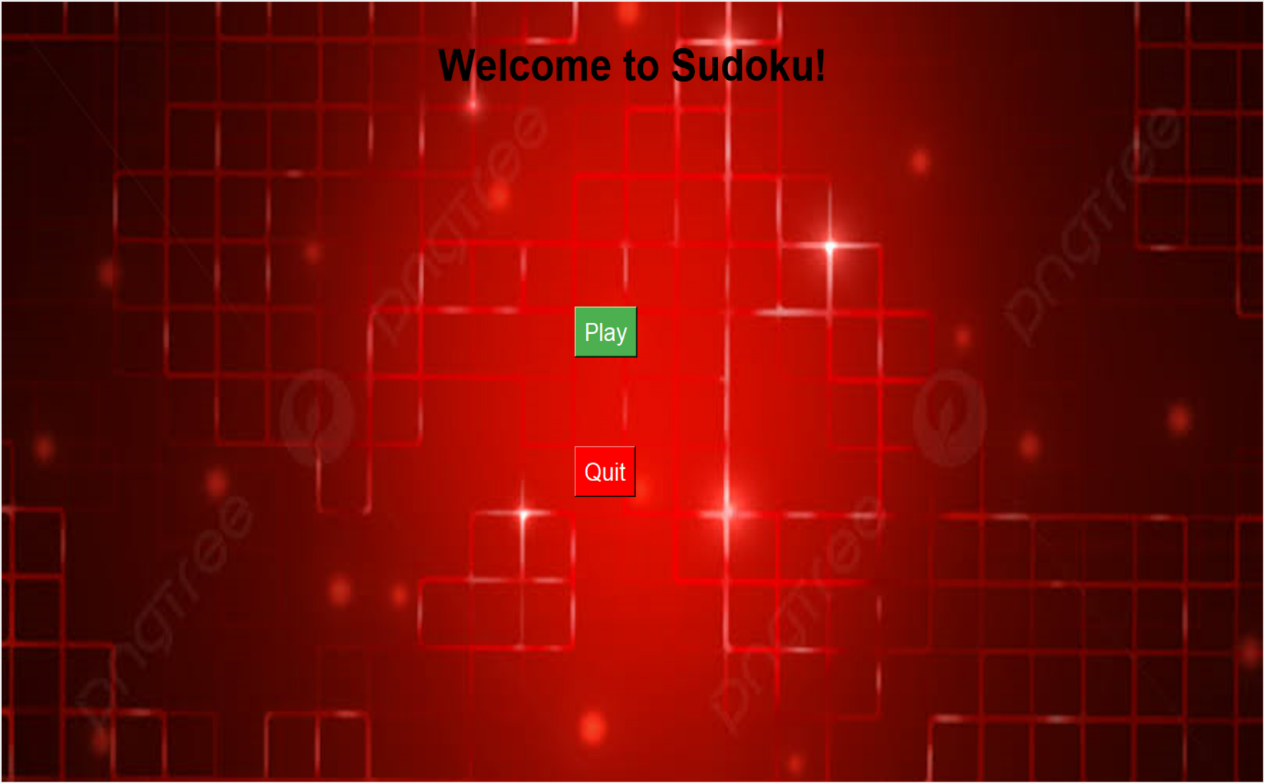
# Run the application

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

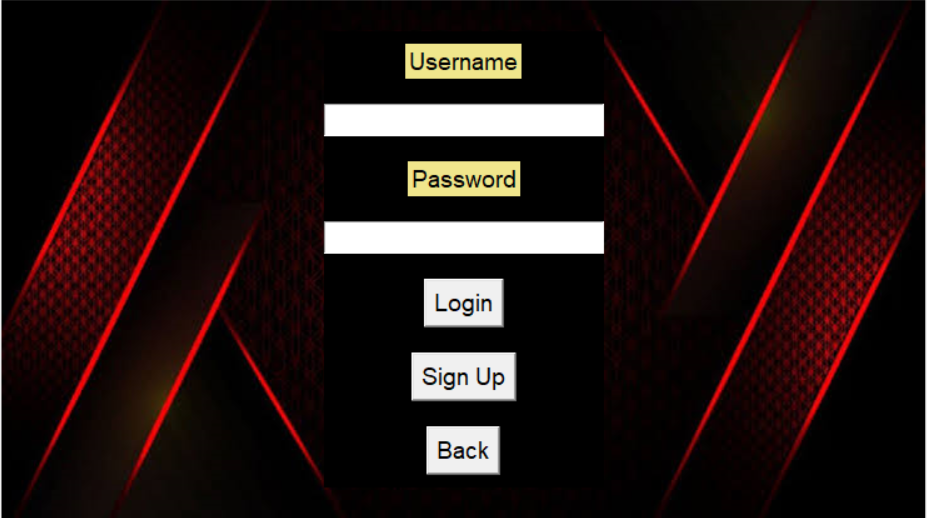
SudokuFrontPage()

SNAPSHOTS

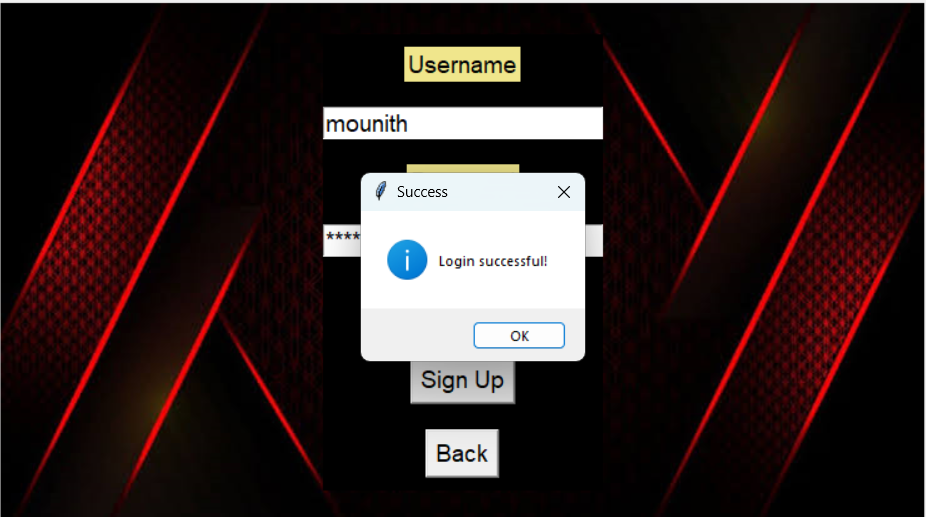
WELCOME PAGE



LOGIN PAGE:



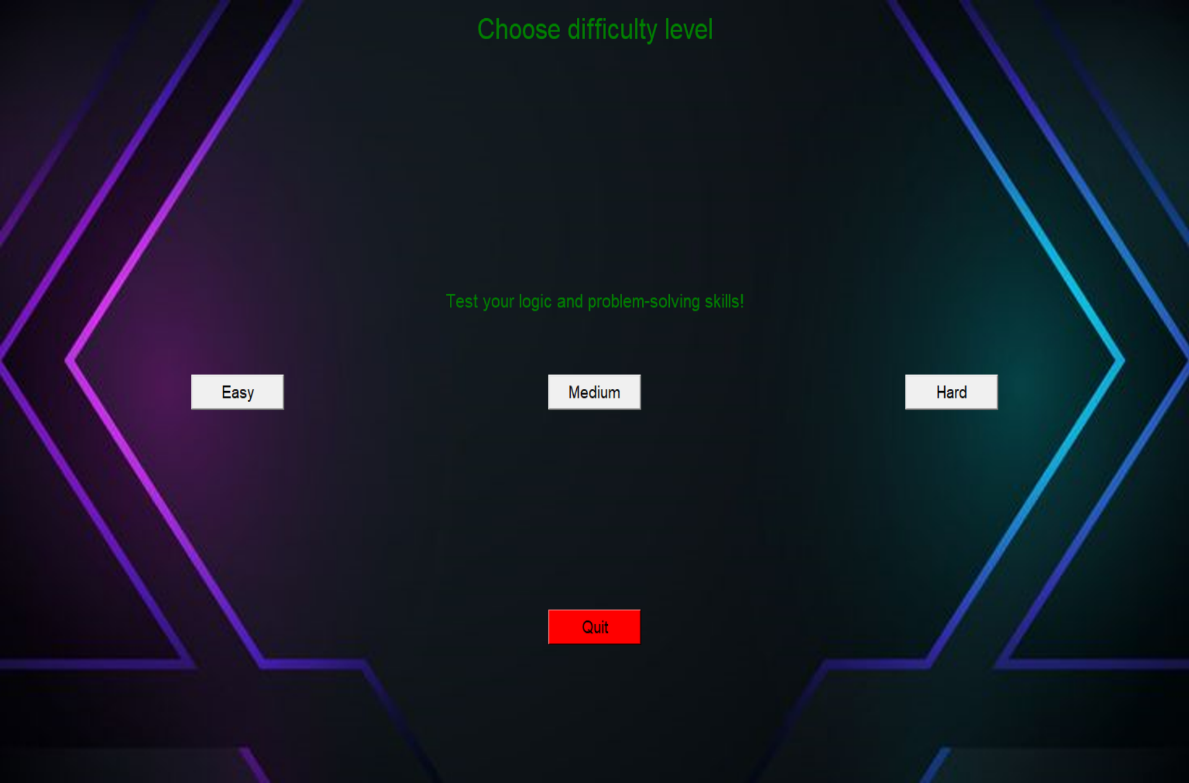
LOGIN VALIDATION:



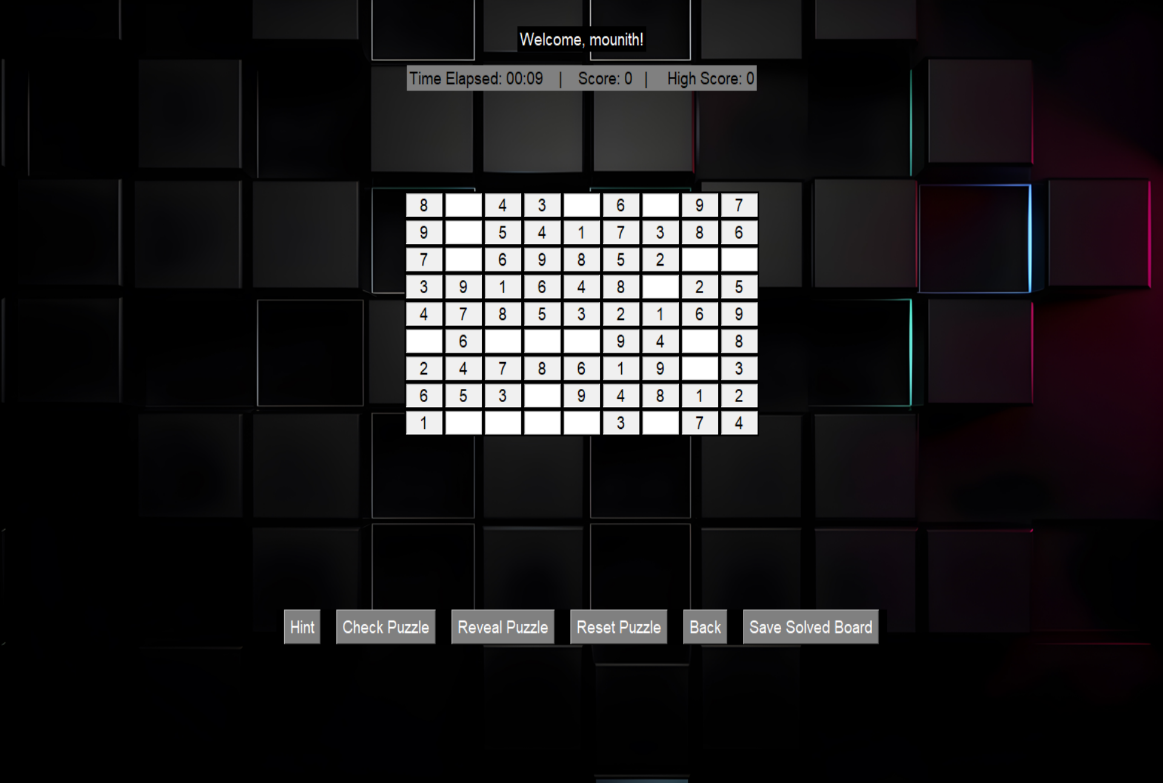
CREATE NEW ACCOUNT:



DIIFICULTY LEVEL:



GAME PAGE:

**REFERENCE**

* Peter Norvig,Director of Research at Google,Reference: Sudoku Solver using Constraint Propagation and Search -[Peter Norvig's Sudoku Guide](<https://norvig.com/sudoku.html>)
* Dan Bader,Software Developer, Author, and Founder of Real Python,Reference:Sudoku Solver and Generator in Python- [Real Python Guide](<https://realpython.com/python-sudoku-solver/>)
* Jose Portilla ,Head of Data Science at Pierian Data ,Reference: Creating and Solving Sudoku Puzzles in Python- [Towards DataScience](<https://towardsdatascience.com/creating-and-solving-sudoku-puzzles-in-python-3e9d6aae4811>)
* GeeksforGeeks Team ,Technical Content Team, GeeksforGeeks ,Reference: Sudoku Solver in Python using Backtracking - [GeeksforGeeks](<https://www.geeksforgeeks.org/sudoku-backtracking-7/>)
* Siemen,Data Scientist, Python Enthusiast,Reference:How to Solve a Sudoku with Python (Backtracking Algorithm) - [Medium](<https://medium.com/@siemen/python-backtracking-solving-sudoku-82f1d850d5a>)
* "Automate the Boring Stuff with Python" by Al Sweigart
* "Python Crash Course" by Eric Matthes
* "Think Python: How to Think Like a Computer Scientist" by Allen B. Downey
* "Python for Data Analysis" by Wes McKinney
* https://github.com/Mounith2005/SudokuSolver.git