Interactive Sudoku Solver
A Game of Logic Using Backtracking

A PROJECT REPORT

Submitted by

RAJEEV M (AP24122040003)

LOKESH V (AP24122040005)

AML 500

Advanced Algorithms and Analysis

SRM University-AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh - 522 240

Nov 2024

Table of Contents

1. INTRODUCTION

Sudoku is a popular logic-based number puzzle that challenges players to fill a 9x9 grid so that each row, each column, and each of the nine 3x3 subgrids contains all the digits from 1 to 9 exactly once. Originating in the late 18th century as a variation of Latin squares, Sudoku gained global recognition in the 1980s and has since become a widely loved mental exercise. It is a classic example of a problem that blends simplicity in rules with complexity in execution, requiring players to employ logic, deduction, and occasionally trial and error.

At its core, Sudoku involves two fundamental tasks: **solving** a puzzle and **generating** new puzzles. Solving Sudoku often requires recognizing patterns and constraints while testing various possibilities. This process lends itself well to computational techniques, particularly the **backtracking algorithm**, which is a systematic method of exploring all possible configurations to identify a solution. The same logic can be extended to generate valid Sudoku puzzles by first creating a completed grid and then removing numbers to provide varying levels of challenge to the player.

This project addresses both solving and generating Sudoku puzzles programmatically, offering two primary modes of interaction for users:

1. **Mode 1:** The program generates a Sudoku puzzle based on the user's selected difficulty level (easy, medium, or hard). The user is then presented with the challenge of solving the puzzle, providing an interactive experience.
2. **Mode 2:** Users can input a Sudoku puzzle of their choice. The program will analyze the puzzle, validate its correctness, and determine whether a solution exists using the backtracking algorithm.

The significance of this project lies in the elegance of its approach and the versatility of its applications. From recreational use to educational tools that teach algorithmic problem-solving, this implementation of Sudoku serves multiple purposes. It demonstrates the power of backtracking algorithms to navigate constrained search spaces efficiently. Furthermore, it underscores the importance of ensuring validity and uniqueness in generated puzzles—a critical feature to maintain the integrity of the Sudoku experience.

The proposed solution is implemented in Python, leveraging its flexibility and ease of use. The backtracking algorithm systematically explores possible configurations for solving puzzles, while a carefully designed grid generation algorithm ensures that the puzzles produced are solvable and have a unique solution. By integrating user-friendly features, such as puzzle validation and solution feedback, the program caters to Sudoku enthusiasts of varying skill levels.

In conclusion, this project exemplifies the fusion of computer science and recreational problem-solving. It not only offers a robust tool for solving and generating Sudoku puzzles but also showcases the broader application of algorithms to real-world challenges. Through the structured approach and logical rigor of the backtracking algorithm, users can engage with Sudoku puzzles while gaining insights into the underlying computational principles.

2. OBJECTIVES

The objective of this project is to design and implement an interactive Sudoku game with two modes: a pre-defined game mode and a user-input mode. The game should provide a visually appealing and user-friendly interface using Pygame, where users can:

1. **Play the game with varying difficulty levels:**
    - Easy
    - Medium
    - Hard
      The difficulty levels determine the number of pre-filled cells, adding variety to the game.
2. **Manually input numbers to create custom Sudoku puzzles:**
    - Users can place their numbers onto a blank Sudoku grid.
    - The game provides a mechanism to validate the grid and check for solvability.
3. **Solve Sudoku puzzles using a backtracking algorithm:**
    - The implemented algorithm can solve even complex Sudoku grids.
    - The solution is visually displayed, enabling users to understand the logic.
4. **Ensure an intuitive user experience through:**
    - Color-coded cells for easier interaction.
    - Interactive number selection using draggable elements.
    - Feedback for invalid moves.

By achieving these objectives, the project combines logical problem-solving and interactive software development, making it engaging for both casual users and Sudoku enthusiasts.

3. PROBLEM STATEMENT

Sudoku is a popular puzzle game that requires logical reasoning to fill a 9x9 grid with numbers such that each row, column, and 3x3 subgrid contains unique values from 1 to 9. Traditional Sudoku puzzles are often limited to paper formats, which lack interactivity and customization.

The main problems this project addresses include:

1. **Accessibility and Engagement**:
    - Traditional Sudoku lacks an engaging, interactive interface.
    - Users cannot dynamically change the difficulty or input custom puzzles in most applications.
2. **Validation and Assistance**:
    - Users need a mechanism to validate their custom inputs.
    - Automated solutions to assist when users are stuck are often unavailable or poorly integrated.
3. **Programming Challenge**:
    - Implementing a robust algorithm to validate and solve puzzles in real time.
    - Handling user interactions, such as drag-and-drop for number placement.

This project solves these issues by developing an interactive, customizable Sudoku game with features that enhance user experience and provide logical assistance, making it both entertaining and educational.

4. METHODOLOGY

The project follows a systematic development approach, divided into the following stages:

4.1 Requirement Analysis

Understanding the user requirements for an interactive Sudoku game:

- A clear and appealing graphical interface.
- Support for difficulty levels.
- Ability to create custom puzzles.

4.2 Design and Planning

1. **Graphical Interface**:
    - Using **Pygame** for the GUI to create a scalable and interactive game board.
    - Designing menus for game modes and difficulty selection.
2. **Game Logic**:
    - Implementing Sudoku rules (unique numbers in rows, columns, and subgrids).
    - Developing a backtracking algorithm for puzzle-solving.

4.3 Development

1. **Menu System**:
    - Creating a main menu with options for starting the game, selecting difficulty, and exiting.
    - Developing a separate menu for user-input mode.
2. **Gameplay Features**:
    - Implementing drag-and-drop functionality for number placement.
    - Providing real-time feedback for invalid moves.
    - Highlighting cells and enabling an intuitive experience.
3. **Validation and Solving**:

    - Coding a Sudoku validator to check the solvability of the board.
    - Implementing the backtracking algorithm to solve puzzles programmatically.

4.4 Testing and Validation

- Testing each feature (e.g., menu navigation, drag-and-drop interaction).
- Validating the correctness of the solving algorithm with multiple puzzles.

4.5 Deployment

Packaging the game for distribution and ensuring cross-platform compatibility.

Algorithm to Solve the Problem:

The algorithm combines Depth-First Search (DFS) with constraint checking, gradually building up possible solutions. It explores each possible solution path, and whenever a constraint is violated (i.e., a solution doesn't meet the Sudoku rules), it backtracks and tries a different path. This method efficiently narrows down the search space, eliminating invalid solutions early.

Algorithm

The core algorithm used to solve Sudoku combines **Depth-First Search (DFS)** with **constraint checking**. It builds potential solutions incrementally while ensuring that any invalid paths are abandoned early through backtracking. This ensures computational efficiency and adherence to Sudoku's rules.

Steps/Algorithm to Solve the Problem

The following steps outline the logic used to solve the Sudoku puzzle:

1. **Check for Valid Placement (Constraint Checking):**
    - Before placing a number in an empty cell, validate that the placement adheres to Sudoku rules:
        - The number should not exist in the same row.
        - The number should not exist in the same column.

        - The number should not exist in the same 3x3 subgrid.
    - If any constraint is violated, discard this number as a potential solution for the current cell.

2. **Depth-First Search (DFS) with Backtracking:**
    - The algorithm utilizes DFS to explore all potential configurations for the grid.
    - Each number from 1 to 9 is checked for validity in the current cell. If valid, the number is placed temporarily.
    - The algorithm then moves to the next empty cell, recursively repeating this process.
3. **Recursive Backtracking Function:**
    - **Base Case:**
        - If no unassigned cells are left in the grid, the puzzle is solved, and the function returns true.
    - **Recursive Case:**
        - Identify the first unassigned (empty) cell.
        - Try numbers from 1 to 9 for this cell:
            1. Validate the placement using the constraint-checking function.
            2. If valid, place the number in the cell and recursively attempt to solve the remaining grid.
            3. If the recursive attempt succeeds, return true.
            4. If unsuccessful, undo the placement (backtrack) and try the next number.
    - If no numbers from 1 to 9 lead to a valid solution, return false (indicating no solution exists for the current configuration).

5. IMPLEMENTATION

Pseudocode

Here is the pseudocode for the backtracking algorithm:

```
def solve_sudoku(grid):
    # Find the first unassigned cell
    row, col = find_empty_cell(grid)
    if not row:  # Base Case: No unassigned cells
        return True

    for num in range(1, 10):  # Try numbers 1 through 9
        if is_valid(grid, row, col, num):  # Check if placement is valid
            grid[row][col] = num  # Place the number
            if solve_sudoku(grid):  # Recursive call
                return True
            grid[row][col] = 0  # Backtrack: Remove the number

    return False  # No solution found for current configuration

def is_valid(grid, row, col, num):
    # Check row, column, and 3x3 subgrid for validity
    return not (
        num in grid[row] or
        num in [grid[i][col] for i in range(9)] or
        num in get_subgrid(grid, row, col)
    )
```

Program Code (Python):

Sudoku Menu Page

```
import pygame
import sys
import sudoku_game
import sudoku_user_mode
#from sudoku_user_mode import start_game  # Import the user mode script

# Initialize Pygame
pygame.init()

# Screen setup
screen_width, screen_height = 700, 500
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Sudoku Game")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)

# Fonts
font = pygame.font.Font(None, 36)

# Game variables
mode = "menu"
difficulty_blanks = 2 # Default difficulty (Easy)

# Function to display the main menu
def display_main_menu():
    screen.fill(WHITE)
    title_text = font.render("Sudoku Game", True, BLACK)
    mode1_text = font.render("1. Start Game", True, BLACK)
    mode2_text = font.render("2. User Mode", True, BLACK)
    exit_text = font.render("3. Exit", True, BLACK)

    screen.blit(title_text, (screen_width // 2 - title_text.get_width() //
2, 50))
    screen.blit(mode1_text, (screen_width // 2 - mode1_text.get_width() //
2, 150))
    screen.blit(mode2_text, (screen_width // 2 - mode2_text.get_width() //
2, 200))
    screen.blit(exit_text, (screen_width // 2 - exit_text.get_width() // 2,
250))

# Function to display the difficulty menu
def display_difficulty_menu():
    screen.fill(WHITE)
    title_text = font.render("Select Difficulty", True, BLACK)
    easy_text = font.render("1. Easy", True, BLACK)
    medium_text = font.render("2. Medium", True, BLACK)
    hard_text = font.render("3. Hard", True, BLACK)
    back_text = font.render("4. Back to Main Menu", True, BLACK)

    screen.blit(title_text, (screen_width // 2 - title_text.get_width() //
2, 50))
    screen.blit(easy_text, (screen_width // 2 - easy_text.get_width() // 2,
150))
    screen.blit(medium_text, (screen_width // 2 - medium_text.get_width()
// 2, 200))
    screen.blit(hard_text, (screen_width // 2 - hard_text.get_width() // 2,
250))
    screen.blit(back_text, (screen_width // 2 - back_text.get_width() // 2,
300))

# Function to exit the game after a delay
def exit_game_with_delay(message, delay=500):
    print(message)   # Print the selected difficulty

    # Display the goodbye message
    screen.fill(WHITE)
    goodbye_text = font.render("Goodbye!", True, BLACK)
    screen.blit(goodbye_text, (screen_width // 2 - goodbye_text.get_width()
// 2, screen_height // 2 - goodbye_text.get_height() // 2))
    pygame.display.flip()   # Update the display
    pygame.time.delay(2000)   # Show goodbye message for 2 seconds

    pygame.quit()   # Clean up and close the window
    sys.exit()   # Exit the program

# Game loop
running = True
while running:
    if mode == "menu":   # Main menu
        display_main_menu()
    elif mode == "difficulty":   # Difficulty selection
        display_difficulty_menu()
    elif mode == "user_mode":   # User mode
        print("User mode is selected")
        sudoku_user_mode.start_game()   # Call the function from the us.py
file

    pygame.display.flip()   # Update the display

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
            if mode == "menu":   # Main menu interaction
                if 150 < mouse_y < 180:   # Start Game
                    mode = "difficulty"
                elif 200 < mouse_y < 230:   # User Mode
                    mode = "user_mode"   # Switch to user mode
                elif 250 < mouse_y < 280:   # Exit
                    running = False
            elif mode == "difficulty":   # Difficulty selection
                if 150 < mouse_y < 180:
                    # Easy
                    sudoku_game.start_game("Easy")
                    exit_game_with_delay("Selected Difficulty: Easy")   #
```

```
Print and exit
                elif 200 < mouse_y < 230:

                    sudoku_game.start_game("Medium")
                    exit_game_with_delay("Selected Difficulty: Medium")   #
Print and exit
                elif 250 < mouse_y < 280:

                    sudoku_game.start_game("Hard")
                    exit_game_with_delay("Selected Difficulty: Hard")   #
Print and exit
                elif 300 < mouse_y < 330:  # Back to Main Menu
                    mode = "menu"

pygame.quit()
```

Sudoku User Mode

```
import pygame
import numpy as np
from copy import deepcopy

# Initialize pygame
pygame.init()

# Screen setup for 700x500
screen_width, screen_height = 700, 500
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Sudoku Game - User Mode")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
GREEN = (99, 200, 167)
RED = (255, 0, 0)
LIGHT_BLUE = (173, 216, 230)
HOVER_COLOR = (220, 220, 220)

# Fonts
font = pygame.font.Font(None, 30)

# Size variables
cell_size = 50
offset_x, offset_y = 20, 20  # Offset to center board in 700x500


# Function to check if the board is valid
def is_valid_board(board):
    for row in range(9):
        for col in range(9):
            num = board[row][col]
            if num != 0:
                board[row][col] = 0
                if not is_valid(board, row, col, num):
                    board[row][col] = num  # Restore the value before
returning
                    return False
                board[row][col] = num
    return True
```

```python
# Function to check if placing a number is valid
def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False
    box_x, box_y = row // 3 * 3, col // 3 * 3
    for i in range(3):
        for j in range(3):
            if board[box_x + i][box_y + j] == num:
                return False
    return True


# Function to solve Sudoku using backtracking
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0
                return False
    return True


# Main menu function
def main_menu():
    menu_running = True

    while menu_running:
        screen.fill(WHITE)

        # Display the title
        title_text = font.render("Sudoku Game", True, BLACK)
        screen.blit(title_text, (screen_width // 2 - title_text.get_width()
// 2, screen_height // 4))

        # Create Start Game button
        start_button = pygame.Rect(screen_width // 2 - 75, screen_height //
2, 150, 50)
        pygame.draw.rect(screen, GREEN, start_button)
        start_text = font.render("Start Game", True, BLACK)
        text_rect_start = start_text.get_rect(center=start_button.center)
        screen.blit(start_text, text_rect_start)

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                menu_running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                mouse_x, mouse_y = event.pos
                if start_button.collidepoint(mouse_x, mouse_y):
                    menu_running = False  # Exit menu and start the game

        pygame.display.flip()

    start_game()
```

```
# Start the game with a new board
def start_game():
    initial_board = [[0] * 9 for _ in range(9)]
    solved_board = deepcopy(initial_board)

    # Game loop
    running = True
    dragging = False
    selected_num = None
    mouse_x, mouse_y = 0, 0
    solution_found = False
    invalid_board = False

    while running:
        screen.fill(WHITE)

        # Draw Sudoku grid with empty cells clearly visible
        for row in range(9):
            for col in range(9):
                rect = pygame.Rect(offset_x + col * cell_size, offset_y + 
row * cell_size, cell_size, cell_size)
                pygame.draw.rect(screen, LIGHT_BLUE if (row // 3 + col // 
3) % 2 == 0 else WHITE, rect)
                pygame.draw.rect(screen, BLACK, rect, 1)

                if initial_board[row][col] != 0:
                    num_text = font.render(str(initial_board[row][col]), 
True, BLACK)
                    screen.blit(num_text, (offset_x + col * cell_size + 15, 
offset_y + row * cell_size + 10))

        # Display draggable number buttons from 1 to 9 in a 2x5 grid format
        for i in range(1, 10):
            row = (i - 1) // 2
            col = (i - 1) % 2
            x_pos = screen_width - 160 + col * 60
            y_pos = offset_y + 30 + row * 50  # Adjusted y position for 
buttons
            num_text = font.render(str(i), True, BLACK)
            num_rect = pygame.Rect(x_pos - 15, y_pos - 15, 50, 50)

            # Highlight the button if hovered over
            if num_rect.collidepoint(pygame.mouse.get_pos()):
                pygame.draw.rect(screen, HOVER_COLOR, num_rect)
            else:
                pygame.draw.rect(screen, GRAY, num_rect)

            pygame.draw.rect(screen, BLACK, num_rect, 2)  # Border for each 
button
            # Center the text
            text_rect = num_text.get_rect(center=num_rect.center)
            screen.blit(num_text, text_rect)

        # Draw Find Solution button below number buttons
        find_solution_button = pygame.Rect(screen_width - 190, offset_y + 
300, 155, 40)
        pygame.draw.rect(screen, GREEN if is_valid_board(initial_board) 
else RED, find_solution_button)
        solution_text = font.render(" Find Solution", True, BLACK)
        text_rect_solution =
solution_text.get_rect(center=find_solution_button.center)
        screen.blit(solution_text, text_rect_solution)

        # Draw Exit button below Find Solution button
        exit_button = pygame.Rect(screen_width - 170, offset_y + 350, 120,
40)
        pygame.draw.rect(screen, GRAY, exit_button)
        exit_text = font.render("Exit", True, BLACK)
        text_rect_exit = exit_text.get_rect(center=exit_button.center)
        screen.blit(exit_text, text_rect_exit)

        # Handle dragging and dropping numbers into empty cells
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                mouse_x, mouse_y = event.pos
                # Check if a number button is clicked to start dragging
                for i in range(1, 10):
                    row = (i - 1) // 2  # Adjusted to match button layout
                    col = (i - 1) % 2
                    x_pos = screen_width - 160 + col * 60
                    y_pos = offset_y + 30 + row * 50
                    num_rect = pygame.Rect(x_pos - 15, y_pos - 15, 50, 50)
                    if num_rect.collidepoint(mouse_x, mouse_y):
                        dragging = True
                        selected_num = i
                        break
                # Check if Find Solution button is clicked
                if find_solution_button.collidepoint(mouse_x, mouse_y):
                    if is_valid_board(initial_board):
                        solution_board = deepcopy(initial_board)
                        solution_found = solve_sudoku(solution_board)
                        if solution_found:
                            solved_board = solution_board
                        else:
                            print("No solution exists")
                    else:
                        invalid_board = True
                # Check if Exit button is clicked
                if exit_button.collidepoint(mouse_x, mouse_y):
                    running = False
            elif event.type == pygame.MOUSEBUTTONUP:
                if dragging and selected_num is not None:
                    dragging = False
                    grid_x = (mouse_x - offset_x) // cell_size
                    grid_y = (mouse_y - offset_y) // cell_size
                    if 0 <= grid_x < 9 and 0 <= grid_y < 9:
                        # Update the board with the selected number,
replacing any existing value
                        initial_board[grid_y][grid_x] = selected_num
                    selected_num = None
            elif event.type == pygame.MOUSEMOTION:
                mouse_x, mouse_y = pygame.mouse.get_pos()

        # Display solution if found
        if solution_found:
            for row in range(9):
                for col in range(9):
                    if solved_board[row][col] != 0:
                        # Use the same font and color as user-entered
numbers
                        num_text = font.render(str(solved_board[row][col]),
True, BLACK)
                        screen.blit(num_text, (offset_x + col * cell_size +
15, offset_y + row * cell_size + 10))
        elif invalid_board:
            invalid_text = font.render("Invalid Board!", True, RED)
            screen.blit(invalid_text, (screen_width // 2 -
invalid_text.get_width() // 2, screen_height // 2))

        pygame.display.flip()

    pygame.quit()


# Start the application
if __name__ == "__main__":
    main_menu()
```

Sudoku Game

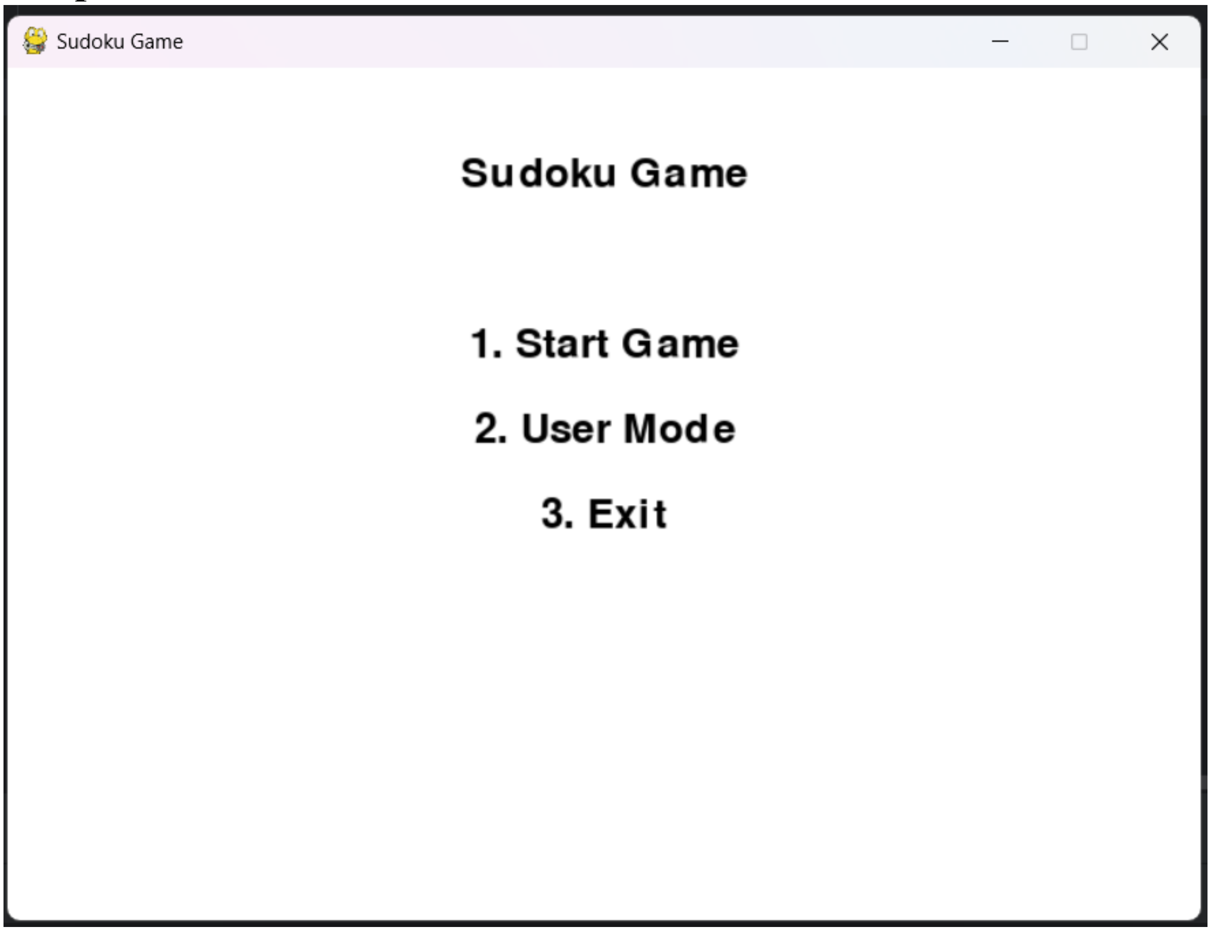
```
# sudoku_game.py
import pygame
from random import sample
from copy import deepcopy

# Initialize pygame
pygame.init()

# Screen setup
screen_width, screen_height = 700, 500
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Sudoku Game")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
GREEN = (99, 200,167)
RED = (255, 0, 0)
LIGHT_BLUE = (173, 216, 230)
HOVER_COLOR = (220, 220, 220)  # Color for hover effect on buttons

# Fonts
font = pygame.font.Font(None, 30)
win_font = pygame.font.Font(None, 60)

# Size variables
cell_size = 50
offset_x, offset_y = 30, 30

# Generate a random 9x9 Sudoku grid
def generate_sudoku():
    base = 3
    side = base * base

    def pattern(r, c): return (base * (r % base) + r // base + c) % side

    def shuffle(s): return sample(s, len(s))
    r_base = range(base)
    rows = [g * base + r for g in shuffle(r_base) for r in shuffle(r_base)]
    cols = [g * base + c for g in shuffle(r_base) for c in shuffle(r_base)]
    nums = shuffle(range(1, side + 1))
    board = [[nums[pattern(r, c)] for c in cols] for r in rows]
    return board

# Create a partial board with some cells set to 0 (blank)
def make_partial_board(board, blanks):
    partial_board = deepcopy(board)
    for _ in range(blanks):
        x, y = sample(range(9), 2)
        partial_board[x][y] = 0
    return partial_board

# Start the game with the selected difficulty level
def start_game(selected_difficulty):
    # Set the number of blanks based on difficulty level
    if selected_difficulty == 'Easy':
        blanks = 2
    elif selected_difficulty == 'Medium':
        blanks = 50
    else:  # Hard
        blanks = 65

    full_board = generate_sudoku()  # Generate the full solved board
    initial_board = make_partial_board(deepcopy(full_board), blanks)  #
Create a playable board with blanks
    solved_board = deepcopy(initial_board)  # Initialize the player's board

    # Helper functions
    def draw_grid():
        for row in range(9):
            for col in range(9):
                rect = pygame.Rect(offset_x + col * cell_size, offset_y +
row * cell_size, cell_size, cell_size)
                pygame.draw.rect(screen, LIGHT_BLUE if (row // 3 + col //
3) % 2 == 0 else WHITE, rect)
                pygame.draw.rect(screen, BLACK, rect, 1)
                if initial_board[row][col] != 0:
                    num_text = font.render(str(initial_board[row][col]),
True, BLACK)
                    screen.blit(num_text, (offset_x + col * cell_size + 15,
offset_y + row * cell_size + 10))
                elif solved_board[row][col] != 0:
                    color = GREEN if solved_board[row][col] ==
full_board[row][col] else RED
                    num_text = font.render(str(solved_board[row][col]),
True, color)
                    screen.blit(num_text, (offset_x + col * cell_size + 15,
offset_y + row * cell_size + 10))

        # Draw thicker lines for 3x3 subgrids
        for i in range(0, 10, 3):
            pygame.draw.line(screen, BLACK, (offset_x, offset_y + i *
cell_size),
                             (offset_x + 9 * cell_size, offset_y + i *
cell_size), 2)
            pygame.draw.line(screen, BLACK, (offset_x + i * cell_size,
offset_y),
                             (offset_x + i * cell_size, offset_y + 9 *
cell_size), 2)

    def draw_buttons():
        # Display draggable number buttons from 1 to 9 in a 2x5 grid format
        for i in range(1, 10):
            row = (i - 1) // 2
            col = (i - 1) % 2
            x_pos = 540 + col * 60
            y_pos = 40 + row * 60
            num_text = font.render(str(i), True, BLACK)
            num_rect = pygame.Rect(x_pos - 15, y_pos - 15, 50, 50)

            # Highlight the button if hovered over
            if num_rect.collidepoint(pygame.mouse.get_pos()):
                pygame.draw.rect(screen, HOVER_COLOR, num_rect)
            else:
                pygame.draw.rect(screen, GRAY, num_rect)

            pygame.draw.rect(screen, BLACK, num_rect, 2)  # Border for each
button
            # Center the text
            text_rect = num_text.get_rect(center=num_rect.center)
            screen.blit(num_text, text_rect)

    def check_win():
        for i in range(9):
            for j in range(9):
                if solved_board[i][j] != full_board[i][j]:
                    return False
        return True

    # Game loop
    running = True
    dragging = False
    selected_num = None
    mouse_x, mouse_y = 0, 0

    while running:
        screen.fill(WHITE)
        draw_grid()
        draw_buttons()

        # Display the dragged number if dragging
        if dragging and selected_num is not None:
            num_text = font.render(str(selected_num), True, RED)
            screen.blit(num_text, (mouse_x - 15, mouse_y - 15))

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                mouse_x, mouse_y = pygame.mouse.get_pos()
                # Check if a number button is clicked to start dragging
                for i in range(1, 10):
                    row = (i - 1) // 2
                    col = (i - 1) % 2
                    x_pos = 540 + col * 60
                    y_pos = 40 + row * 60
                    num_rect = pygame.Rect(x_pos - 15, y_pos - 15, 50, 50)
                    if num_rect.collidepoint(mouse_x, mouse_y):
                        dragging = True
                        selected_num = i
                        break
            elif event.type == pygame.MOUSEBUTTONUP:
                if dragging:
                    dragging = False
                    if selected_num is not None:
                        # Calculate the grid cell where the number is
dropped
                        grid_x = (mouse_x - offset_x) // cell_size
                        grid_y = (mouse_y - offset_y) // cell_size
                        # Place the number if in a valid cell
                        if 0 <= grid_x < 9 and 0 <= grid_y < 9 and
initial_board[grid_y][grid_x] == 0:
                            solved_board[grid_y][grid_x] = selected_num
                            if check_win():
                                win_text = win_font.render("You Win!",
True, GREEN)
                                screen.blit(win_text, (200, 220))
                                pygame.display.flip()
                                pygame.time.delay(5000)
                                running = False
                        selected_num = None
            elif event.type == pygame.MOUSEMOTION:
                mouse_x, mouse_y = pygame.mouse.get_pos()

        pygame.display.flip()

    pygame.quit()
```

6. RESULTS AND DISCUSSION

6.1. Solving Puzzles

- Mode 1: Users appreciate the range of difficulty levels and clear grid presentation.
- Mode 2: Users find the validation and solving feedback useful for understanding their puzzles.

Output:

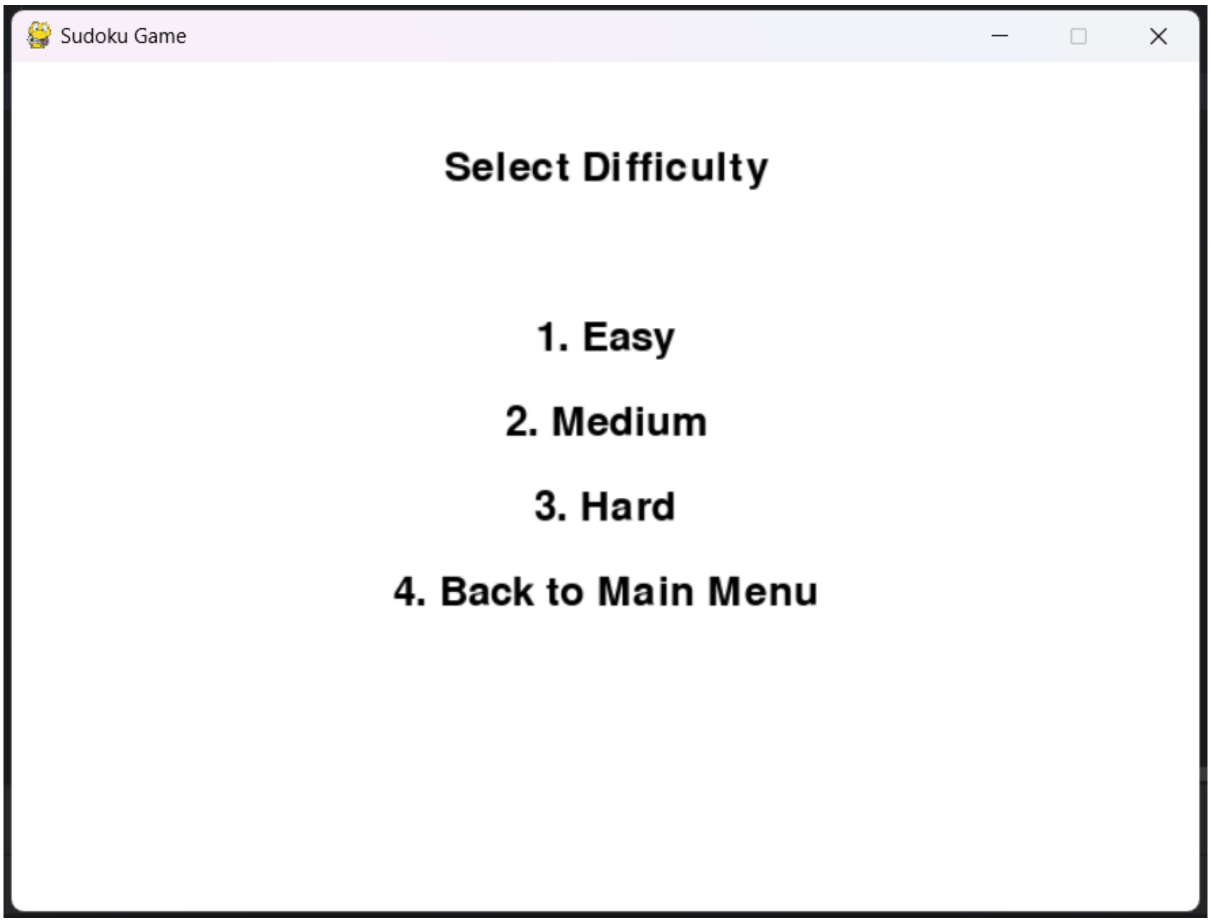


The above output is the sudoku main page in which user can select the start game,user mode and the exit.

6.2. Generating Puzzles

The generated puzzles are tested to ensure:

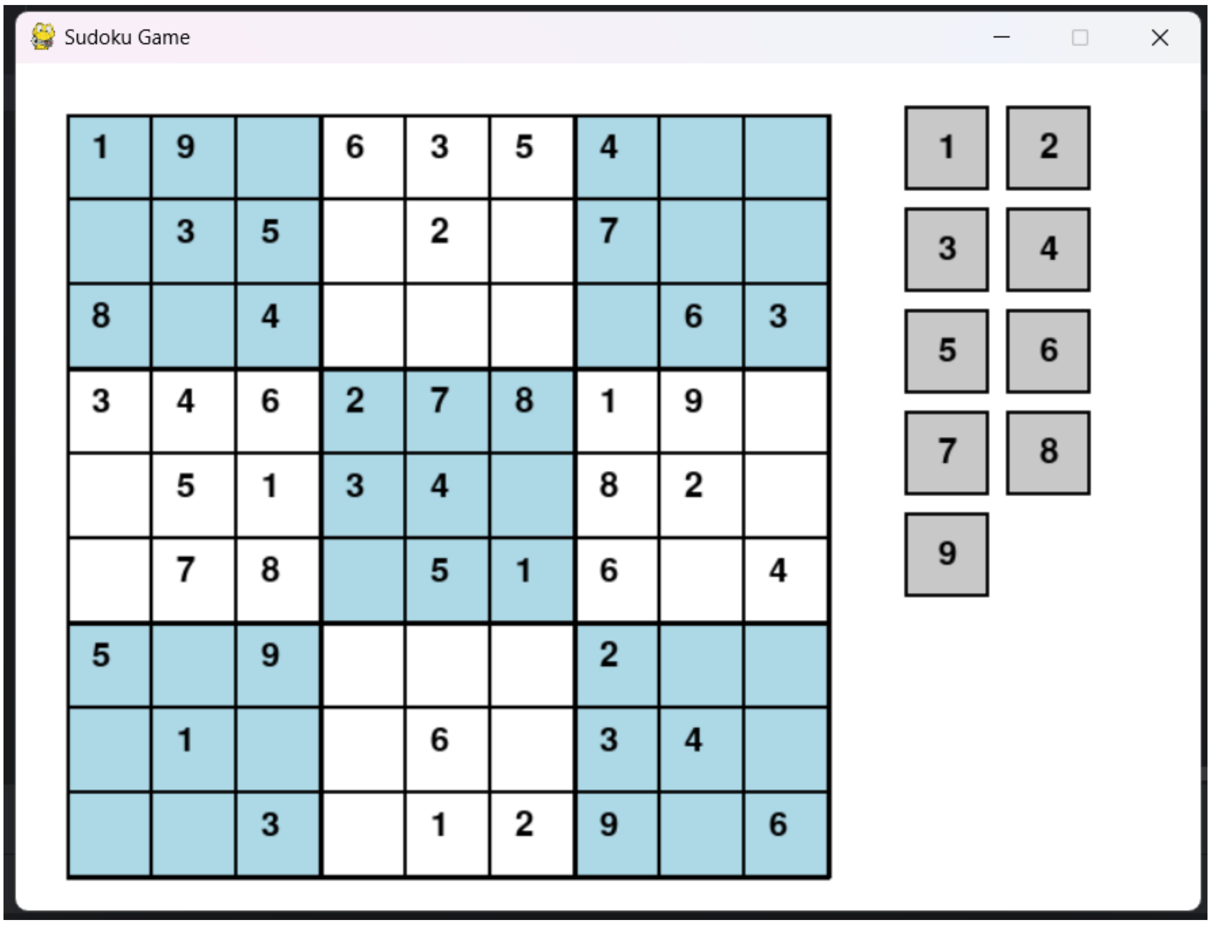
- They adhere to Sudoku rules.
- They have unique solutions.
- The difficulty levels align with the expected complexity for users.



When the user selects the start game it displays the different difficulty levels in the game like Easy,Medium,Hard as shown above.

Playing:

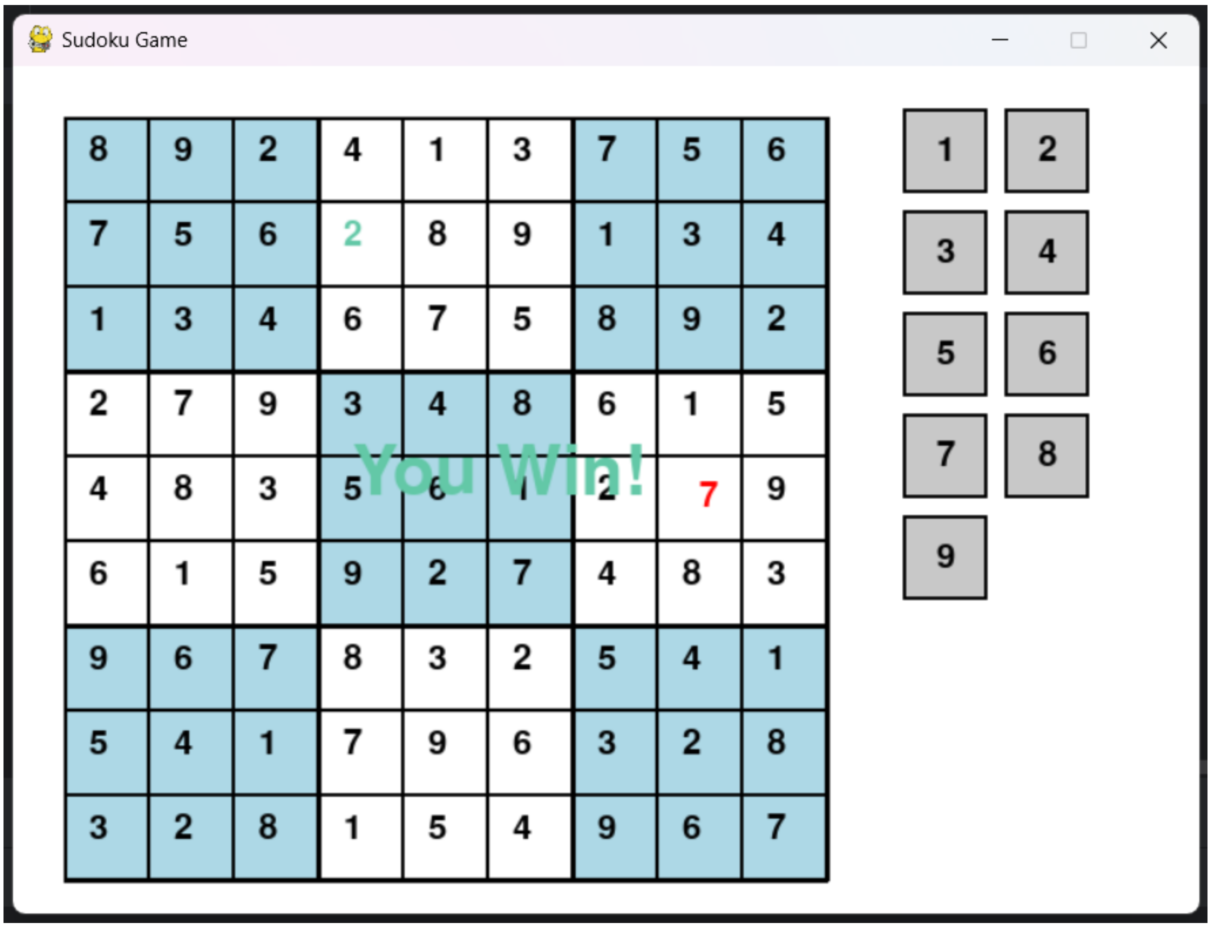
Based on the difficulty mode the sudoku game will be displayed and user can start playing the game.



1	9		6	3	5	4		
	3	5		2		7		
8		4					6	3
3	4	6	2	7	8	1	9	
	5	1	3	4		8	2	
	7	8		5	1	6		4
5		9				2		
	1			6		3	4	
		3		1	2	9		6

WIN:

If Output obey all condition ...

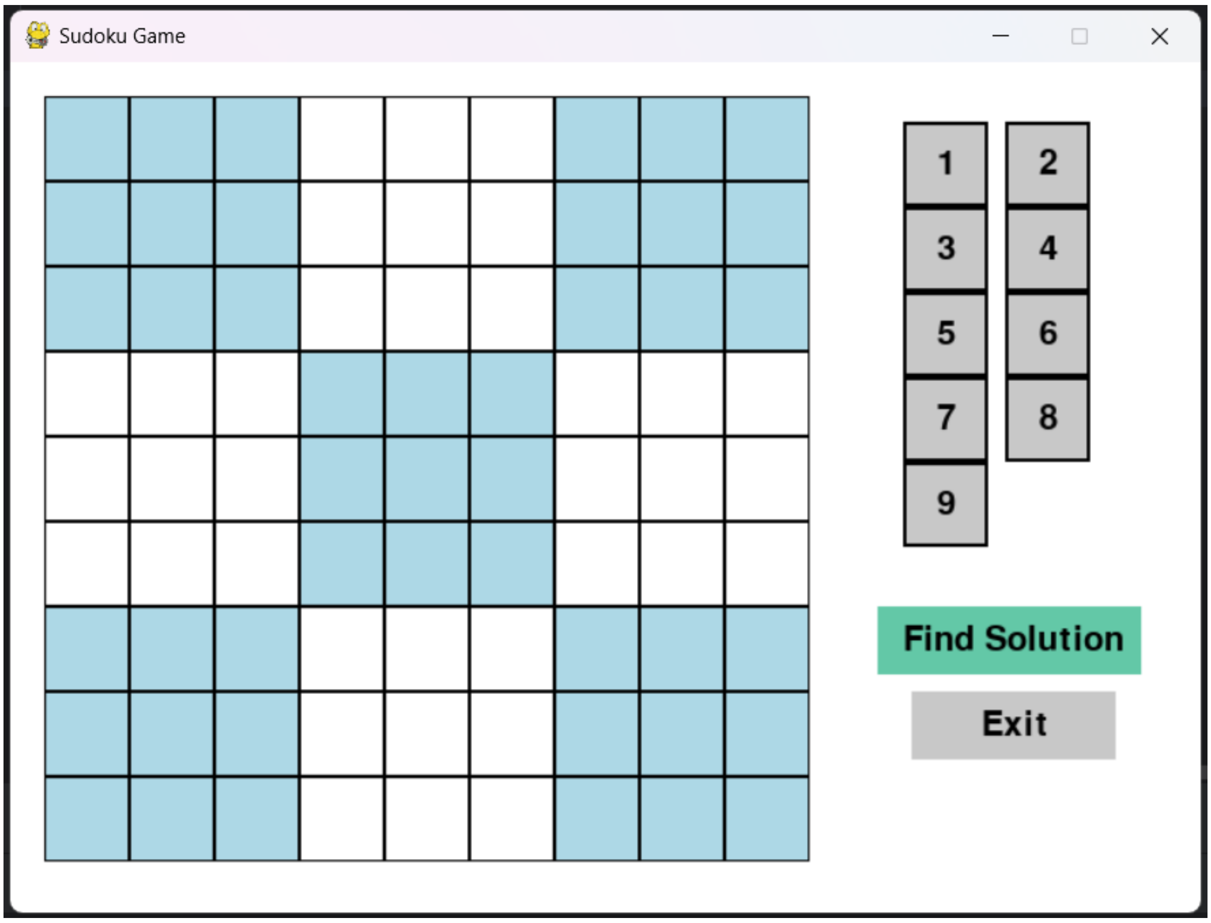


8	9	2	4	1	3	7	5	6
7	5	6	2	8	9	1	3	4
1	3	4	6	7	5	8	9	2
2	7	9	3	4	8	6	1	5
4	8	3	5	6	1	2	7	9
6	1	5	9	2	7	4	8	3
9	6	7	8	3	2	5	4	1
5	4	1	7	9	6	3	2	8
3	2	8	1	5	4	9	6	7

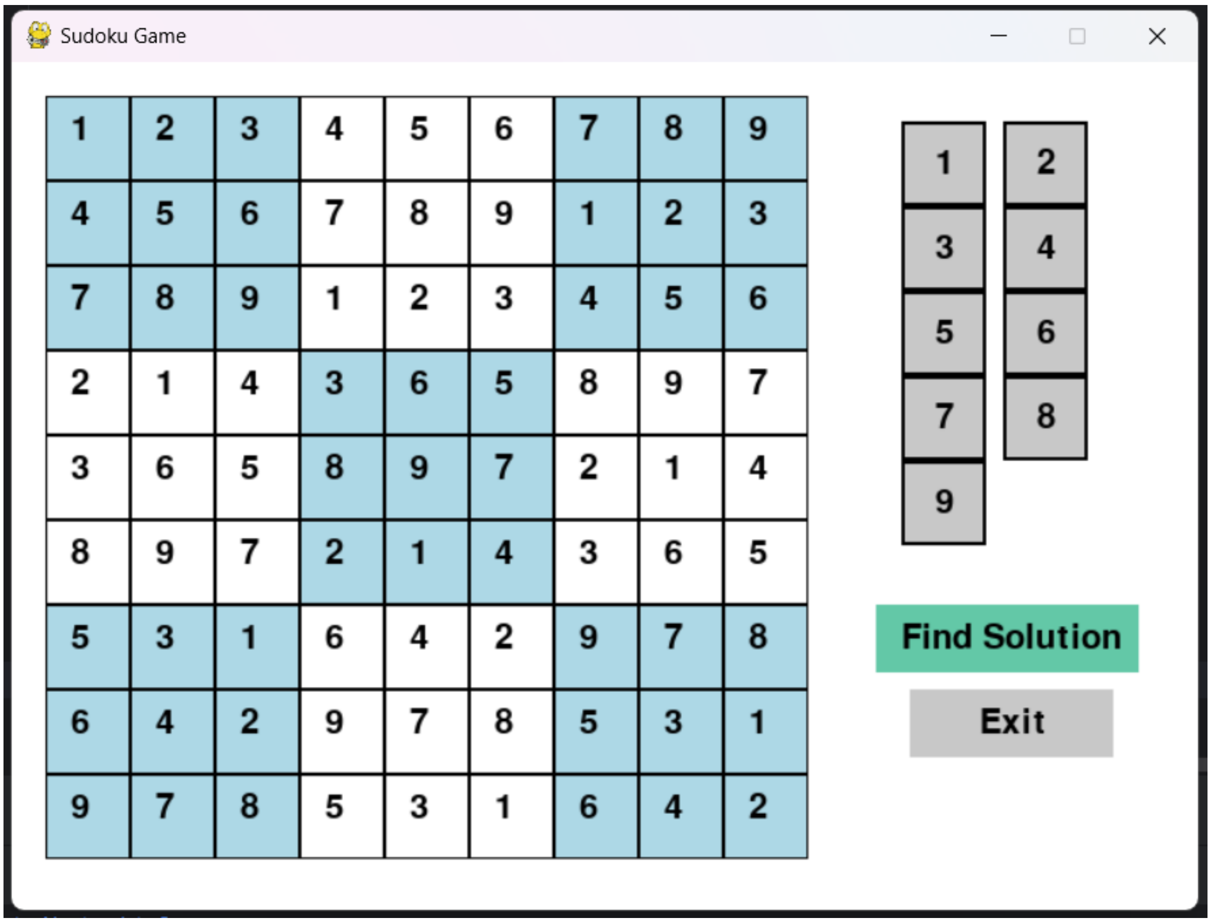
If the number entered by the user is correct, then it will display in green color, else in red color. Once the user finishes the game, then it will display “You Win!”

User Mode

The Below image is the User mode where user can create their own Sudoku game.



Based on the numbers placed by the user in the cells, it will find the solution, if the solution exists.



1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

7. TIME COMPLEXITY AND SPACE COMPLEXITY

Time Complexity:

The time complexity of solving a Sudoku puzzle using backtracking is determined by the number of recursive calls made and the operations performed at each step. Here's the detailed derivation:

1. **Recursive Exploration:**
   At every step, the algorithm attempts to fill an empty cell with a number from 1 to 9.
   - If there are nnn empty cells, the worst-case scenario involves trying 9n9^n9n combinations.
   - However, due to constraint checking (validating the row, column, and 3x3 subgrid), invalid combinations are pruned early. This significantly reduces the search space but does not eliminate the exponential growth in the worst case.
2. **Constraint Checking:**
   For each placement attempt, the algorithm validates the number against the row, column, and subgrid constraints.
   - Checking all constraints for a single cell takes O(9)O(9)O(9), as each row, column, and subgrid contains a maximum of 9 cells.
   - Therefore, the complexity per placement attempt is constant, O(1)O(1)O(1).
3. **Overall Time Complexity:**
   Combining the above, the **worst-case time complexity** is:

O(9n)where n is the number of empty cells.O(9^n) \quad \text{where } n \text{ is the number of empty cells.}O(9n)where n is the number of empty cells.

- In practice, the complexity is much lower due to pruning from constraint checks and early termination when a solution is found.

Space Complexity

The space complexity is determined by the storage used during the execution of the algorithm.

1. **Recursive Stack Space:**
   The backtracking approach uses recursion to explore all possible solutions.
   - In the worst case, the recursion depth is equal to the number of empty cells, nnn, as each recursive call fills one cell.
   - Hence, the space required for the stack is O(n)O(n)O(n).
2. **Grid Storage:**
   The Sudoku grid is stored in memory and requires constant space, O(81)O(81)O(81), as the grid size is fixed at 9x9. For larger grids, the storage requirement is O(k2)O(k^2)O(k2), where kkk is the grid dimension.
3. **Auxiliary Space:**
   No additional data structures are used beyond the grid and the stack. Thus, auxiliary space usage is minimal.
4. **Overall Space Complexity:**
   Combining the stack space and grid storage:

   $O(n)+O(81) \approx O(n)$ for a standard 9x9 grid.

8. REFERENCES

[1] Maji A K, Pal R K. Sudoku solver using mini grid based backtracking [C]. 2014 IEEE

International Advance Computing Conference (IACC), Gurgaon, India, 2014, pp. 36-44, doi:

10.1109/IAdCC.2014.6779291.

[2] Wang, C. et al. A Novel Evolutionary Algorithm with Column and Sub-Block Local Search for Sudoku

Puzzles [J]. IEEE Transactions on Games, 2023, 1–11, doi: 10.1109/TG.2023.3236490.

[3] Gupta K, Khatri S, Khan M H. A Novel Automated Solver for Sudoku Images [C]. 2019 IEEE 5th

International Conference for Convergence in Technology (I2CT), Bombay, India, 2019, pp. 1-6, doi:

10.1109/I2CT45611.2019.9033860.

[4] Nwulu E, Bisandu D, Dunka B. A hybrid backtracking and pencil and paper sudoku solver [J].

International Journal of Computer Applications, 2019, 181, 975-8887. 10.5120/ijca2019918642.

[5] Dheeraj G S, Lakshmi K B V, Krishna K A. Computer Vision based Sudoku Solving with Augmented

Reality [J]. International Research Journal of Engineering and Technology (IRJET), 2020, 7(10):

1489–1493.

[6] Syed A, Merugu S, Kumar V. Augmented Reality on Sudoku Puzzle Using Computer Vision and Deep Learning [J]. Augmented Reality on Sudoku Puzzle Using Computer Vision and Deep Learning,

[7] Sudoku solving using patterns and graph theory [J]. International Journal of Emerging

Technologies and Innovative Research, 2021, 8(3): 2219-2226.

[8] Iyer R, Jhaveri A, Parab K. A Review of Sudoku Solving using Patterns [J]. International Journal of

Scientific and Research Publications, 2013, 3(5).

[9] Musliu N, Winter F. A Hybrid Approach for the Sudoku Problem: Using Constraint Programming in

Iterated Local Search [J]. IEEE Intelligent Systems, 2017, 32(2): 52-62, doi: 10.1109/MIS.2017.29

[10] Singh V, Sharma V, Bachchas V. Sudoku Solving Using Quantum Computer [J]. Ijraset Journal for

Research in Applied Science and Engineering Technology, 2023, 128: 8.

[11] Herimanto P, Sitorus P, Zamzami E M. An Implementation of Backtracking Algorithm for Solving a

Sudoku-Puzzle Based on Android [J]. J. Phys.: Conf. Ser., 2020, 1566: 012038.

[12] A Survey on Sudoku Solver Using Various Algorithms [J]. International Journal of Emerging Trends

Technology in Computer Science (IJETTCS), 2017, 6(4): 123-129.

[13] Norvig P, Russel S. Artificial Intelligence: A Modern Approach. Third Edition [D]. Harlow: Pearson

Education, 2016.

[14] Knuth, D. Dancing links [J]. arXiv Preprint, 2000.

[15] Harrysson M, Laestander H. Solving Sudoku efficiently with Dancing Links. (Degree Project in

Computer Science, DD143X) [D]. Stockholm, Sweden: Stockholm University, 2014.