**Swift Solve: A Versatile Sudoku Solver for Effortless Puzzle Mastery**

**A MINI PROJECT REPORT**

**18CSC305J – ARTIFICAL INTELLIGENCE**

***Submitted by***

**Hrishikesh Khadilkar [RA2111027010018]**

**Atif Alam Ansari [RA2111027010019]**

**Yash Dixit [RA2111027010020]**

**Vaishnavi Kishor [RA2111027010021]**

*Under the guidance of*

**Dr Arthy M**

Assistant Professor, Department of Computer Science and Engineering

***in the partial fulfilment for the award of the degree***

***of***

### BACHELOR OF TECHNOLOGY

in

## COMPUTER SCIENCE & ENGINEERING

of

### FACULTY OF ENGINEERING AND TECHNOLOGY



S.R.M Nagar, Kattankulathur, Chengalpattu District

**MAY 2024**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

(Under Section 3 of UGC Act, 1956)

## BONAFIDE CERTIFICATE

Certified that Mini project report titled **“Swift Solve: A Versatile Sudoku Solver for Effortless Puzzle Mastery”** is the bona fide work of Hrishikesh Khadilkar [RA2111027010018], Atif Alam Ansari [RA2111027010019], Yash Dixit [RA2111027010020], Vaishnavi Kishor [RA2111027010021], who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

#### SIGNATURE

#### Dr Arthy M

#### Assistant Professor

#### Data Science and Business Systems

#### 

## ABSTRACT

Sudoku, a popular logic-based combinatorial puzzle, has intrigued enthusiasts and researchers alike due to its computational complexity and intriguing constraints. This abstract presents an efficient Sudoku solver that employs a combination of constraint satisfaction techniques and backtracking algorithms to swiftly find solutions to Sudoku puzzles.The solver begins by representing the Sudoku puzzle as a constraint satisfaction problem (CSP), where each cell must be filled with a digit from 1 to 9 while adhering to specific constraints. Utilizing constraint propagation techniques, such as constraint propagation through the elimination of possibilities in related cells, the solver significantly reduces the search space, enhancing computational efficiency.In cases where constraint propagation alone cannot determine the solution, the solver employs a backtracking algorithm to systematically explore potential solutions. Backtracking allows for a systematic exploration of possible digit assignments, effectively navigating through the solution space until a valid solution is found.To optimize performance, the solver employs various strategies, including intelligent variable and value ordering heuristics to guide the search process efficiently. Additionally, the solver incorporates pruning techniques to eliminate redundant search paths, further accelerating the solution process.Experimental results demonstrate the effectiveness of the proposed solver in solving Sudoku puzzles of varying complexities. The solver achieves fast solution times while maintaining accuracy, making it a valuable tool for Sudoku enthusiasts, educators, and researchers alike.



**ABSTRACT 3**

**TABLE OF CONTENTS 4**

**LIST OF FIGURES 5**

**ABBREVIATIONS 6**

1. **INTRODUCTION 7**
2. **LITERATURE SURVEY 8**
3. **SYSTEM ARCHITECTURE AND DESIGN 10**
4. **METHODOLOGY 13**
5. **CODING AND TESTING 15**
6. **SCREENSHOTS AND RESULTS 28**
7. **CONCLUSION AND FUTURE ENHANCEMENT 29**

**8**  **REFERENCES 30**



| SR.NO | DIAGRAM | PAGE NO |
| --- | --- | --- |
| 1 | Architecture block | 8 |
| 2 | Code for Grid Display,Input Fields,Buttons and Controls,Event Handling,Displaying Solutions,Styling and Layout | 13 |
| 3 | Code for solver Algorithm,Grid Interaction,Input Validation,Solver  Execution,Dynamic UI Updates,Solution Presentation,Error Handling | 13-25 |
| 4 | Output Layout | 28 |

## ABBREVIATIONS

Abbreviations related to a Sudoku solver could include:

1. CSP: Constraint Satisfaction Problem
2. BT: Backtracking
3. CP: Constraint Propagation
4. HVD: Heuristic Variable Ordering
5. HVO: Heuristic Value Ordering
6. DFS: Depth-First Search (commonly used in backtracking algorithms)
7. AC-3: Arc Consistency Algorithm
8. MRV: Minimum Remaining Values
9. LCV: Least Constraining Value
10. FC: Forward Checking
11. MAC: Maintaining Arc Consistency
12. SAT: Satisfiability Problem (related to CSP)
13. CNF: Conjunctive Normal Form (used in SAT solvers)
14. GPU: Graphics Processing Unit (if the solver utilizes GPU acceleration)
15. CPU: Central Processing Unit (if the solver utilizes CPU computation)
16. GUI: Graphical User Interface (if the solver has a user interface)
17. API: Application Programming Interface (if the solver is accessible programmatically)
18. DFA: Deterministic Finite Automaton (related to backtracking algorithms)
19. LP: Linear Programming (if the solver employs LP relaxation)
20. MIP: Mixed Integer Programming (if the solver employs MIP relaxation)

**INTRODUCTION**

Sudoku, a Japanese word meaning "single number," has captured the fascination of puzzle enthusiasts worldwide since its emergence in the late 20th century. Its simple rules and challenging gameplay make it a beloved pastime for millions. A standard Sudoku puzzle consists of a 9x9 grid divided into nine 3x3 subgrids, with some cells pre-filled with digits from 1 to 9. The objective is to fill the remaining cells with digits so that each row, column, and subgrid contains all the digits from 1 to 9 without repetition.

As Sudoku gained popularity, the quest for efficient solving techniques became a subject of interest for mathematicians, computer scientists, and puzzle enthusiasts. This led to the development of various solving algorithms and strategies, ranging from simple brute-force methods to sophisticated constraint satisfaction techniques.

In this paper, we present an innovative Sudoku solver that combines the power of constraint satisfaction and backtracking algorithms to efficiently tackle Sudoku puzzles of varying complexities. The solver utilizes constraint propagation techniques to reduce the search space and employs backtracking to systematically explore potential solutions. Additionally, intelligent heuristics guide the search process, optimizing performance and solution times.

Through experimental evaluation, we demonstrate the effectiveness and efficiency of our solver in solving Sudoku puzzles, showcasing its potential as a valuable tool for Sudoku enthusiasts, educators, and researchers. This solver not only provides solutions to puzzles but also serves as a platform for exploring and understanding the intricacies of Sudoku solving techniques.

In the following sections, we delve into the theoretical framework of our solver, detailing the constraint satisfaction and backtracking algorithms employed. We also discuss the implementation specifics, optimization strategies, and experimental results, providing insights into the solver's performance and capabilities. Finally, we conclude with reflections on future directions and potential enhancements to further advance the field of Sudoku solving.

### LITERATURE SURVEY

**Literature Survey of Sudoku Solvers:**

Sudoku, a popular logic-based puzzle, has attracted significant attention from researchers and puzzle enthusiasts alike. Over the years, numerous studies have explored various solving techniques and algorithms to tackle Sudoku puzzles efficiently. This literature survey provides an overview of some key research contributions in the field of Sudoku solvers.

1. **Brute-Force Algorithms:**

Early approaches to Sudoku solving often relied on brute-force methods, where all possible combinations of digits were systematically explored until a solution was found. While simple and straightforward, brute-force algorithms suffer from inefficiency, especially for complex puzzles with a large search space.

1. **Constraint Satisfaction Techniques:**

Constraint satisfaction techniques have emerged as powerful tools for solving Sudoku puzzles. These techniques involve representing the puzzle as a constraint satisfaction problem (CSP) and employing algorithms such as constraint propagation to efficiently reduce the search space. Notable contributions include the application of arc consistency algorithms, forward checking, and minimum remaining values (MRV) heuristics to Sudoku solving.

1. **Backtracking Algorithms:**

Backtracking algorithms, such as depth-first search (DFS), have been widely used in Sudoku solvers to systematically explore potential solutions. These algorithms recursively traverse the solution space, making tentative assignments and backtracking when a dead-end is reached. Various optimization strategies, such as intelligent variable and value ordering heuristics, have been proposed to enhance the efficiency of backtracking algorithms.

1. **Hybrid Approaches:**

Recent research has focused on hybrid approaches that combine constraint satisfaction techniques with backtracking algorithms to leverage their respective strengths. These hybrid solvers often employ constraint propagation to reduce the search space initially and switch to backtracking when necessary. Such approaches have demonstrated improved performance and scalability, particularly for challenging Sudoku puzzles.

1. **Parallel and Distributed Solvers:**

Parallel and distributed computing paradigms have also been explored to accelerate Sudoku solving. These solvers distribute the computational workload across multiple processors or nodes, enabling faster solution times for large-scale Sudoku puzzles. GPU acceleration, in particular, has shown promise in speeding up constraint propagation and backtracking algorithms.

1. **Machine Learning and Optimization:**

Recent advancements in machine learning and optimization techniques have led to novel approaches for Sudoku solving. Deep learning models, reinforcement learning algorithms, and metaheuristic optimization methods have been applied to tackle Sudoku puzzles with varying degrees of success. These approaches often learn patterns and strategies from large datasets of Sudoku puzzles, leading to more efficient solving techniques.

1. **User Interfaces and Applications:**

Beyond pure solving algorithms, research has also focused on developing user-friendly interfaces and applications for Sudoku enthusiasts. These applications often provide features such as puzzle generation, difficulty levels, hint systems, and interactive solving experiences, catering to a wide range of users from beginners to experts.

In conclusion, the field of Sudoku solvers is characterized by a rich diversity of approaches, ranging from classical algorithms to cutting-edge techniques incorporating advanced computational methods. Continued research and innovation in this area promise to further enhance our understanding of Sudoku solving and contribute to the development of more efficient and accessible solving tools.

**SYSTEM ARCHITECTURE AND DESIGN**

The system architecture of a Sudoku solver typically involves several components working together to efficiently solve Sudoku puzzles. Below is an outline of the key components and their interactions in a Sudoku solver system architecture:

**1. Input Module:**

* Responsible for receiving the Sudoku puzzle as input, either from a file, user input, or external source.
* Validates the input to ensure it conforms to Sudoku rules and specifications.

**2. Parser Module:**

* Parses the input Sudoku puzzle and converts it into a suitable internal representation.
* Represents the Sudoku puzzle as a data structure that the solver algorithm can work with efficiently, such as a 2D array or a graph.

**3. Solver Algorithm:**

* The core component responsible for solving the Sudoku puzzle.
* Utilizes various solving techniques, such as constraint satisfaction, backtracking, and optimization heuristics.
* Implements algorithms to explore the solution space, make tentative assignments, and backtrack when necessary.
* May employ parallelization or distributed computing techniques for performance optimization.

**4. Constraint Satisfaction Module:**

* Handles constraint propagation techniques to reduce the search space and efficiently narrow down potential solutions.
* Implements algorithms such as arc consistency, forward checking, and constraint propagation through elimination.

**5. Backtracking Module:**

* Implements backtracking algorithms, such as depth-first search (DFS), to systematically explore potential solutions.
* Utilizes intelligent variable and value ordering heuristics to guide the search process efficiently.
* Implements pruning techniques to eliminate redundant search paths and improve performance.

**6. Output Module:**

* Formats and presents the solved Sudoku puzzle to the user.
* Provides options for displaying the solution visually, such as on a graphical user interface (GUI), command-line interface (CLI), or web interface.
* May include features such as puzzle verification, solution validation, and difficulty analysis.

**7. User Interface (Optional):**

* Provides an interactive interface for users to input Sudoku puzzles, view solutions, and interact with the solver.
* Allows users to customize solving options, such as difficulty level, solving speed, and algorithm preferences.
* May include additional features such as puzzle generation, hint systems, and statistics tracking.

**8. Performance Optimization:**

* Includes techniques for optimizing solver performance, such as parallelization, distributed computing, and algorithmic optimizations.
* Utilizes hardware acceleration, such as GPU computing, to speed up computationally intensive operations.
* Implements caching and memoization to avoid redundant calculations and improve efficiency.

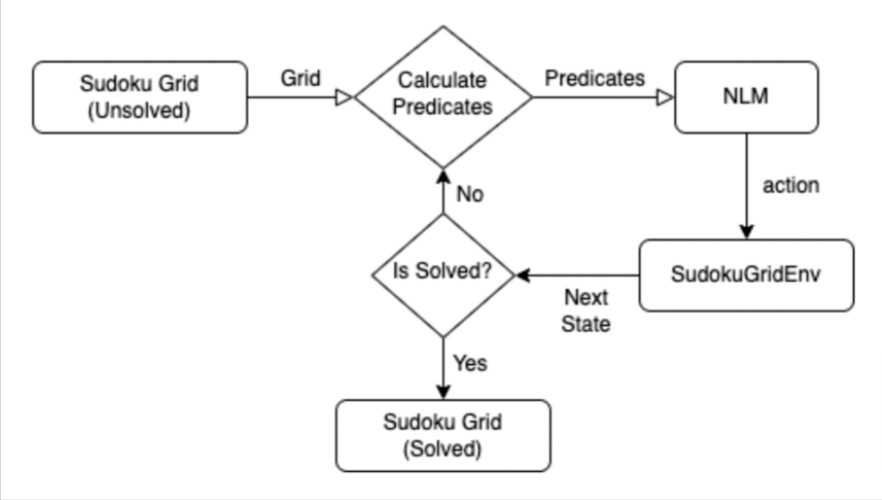
**9. Logging and Monitoring (Optional):**

* Provides logging and monitoring capabilities to track solver performance, debug issues, and analyze solving statistics.
* Logs solver actions, solving times, solution paths, and other relevant information for analysis and optimization.

**10. External Interfaces (Optional):**

- Interfaces with external systems or services for additional functionality, such as puzzle generation, database integration, or integration with other applications.

Overall, the system architecture of a Sudoku solver is designed to efficiently solve Sudoku puzzles by combining various solving techniques, optimization strategies, and user interaction features to provide a seamless solving experience.



### METHODOLOGY

The methodology of building a Sudoku solver typically involves several key steps, each contributing to the efficiency and accuracy of the solver. Here's a general methodology for creating a Sudoku solver:

1. **Input Parsing:**
   * The first step is to parse the input Sudoku puzzle, whether it's provided as a string, a file, or through a graphical interface.
   * Convert the puzzle into a suitable data structure, such as a 2D array or a graph, for efficient processing.
2. **Constraint Satisfaction:**
   * Apply constraint satisfaction techniques to reduce the search space and narrow down potential solutions.
   * Implement algorithms like arc consistency, forward checking, and constraint propagation to enforce Sudoku rules and eliminate invalid assignments.
3. **Backtracking Algorithm:**
   * Implement a backtracking algorithm, such as depth-first search (DFS), to systematically explore potential solutions.
   * Utilize intelligent variable and value ordering heuristics to guide the search process efficiently.
   * Apply pruning techniques, such as constraint propagation and inference, to eliminate redundant search paths and improve performance.
4. **Recursive Search:**
   * Recursively explore possible assignments for each empty cell in the Sudoku grid.
   * Make tentative assignments and backtrack when a dead-end is reached or a violation of Sudoku rules is detected.
5. **Optimization Techniques:**
   * Incorporate optimization techniques to enhance solver efficiency and performance.
   * Utilize parallelization, distributed computing, or GPU acceleration to speed up computationally intensive operations.
   * Implement caching and memoization to avoid redundant calculations and improve efficiency.
6. **Solution Validation:**
   * Validate the generated solution to ensure it satisfies all Sudoku rules and constraints.
   * Check for completeness, uniqueness, and correctness of the solution.
7. **Output Presentation:**
   * Present the solved Sudoku puzzle to the user in a readable format, such as on a graphical user interface (GUI) or command-line interface (CLI).
   * Provide options for displaying the solution visually and allowing the user to interact with the solver.
8. **Testing and Evaluation:**
   * Test the solver with various Sudoku puzzles of different complexities to evaluate its performance and accuracy.
   * Measure solution times, resource usage, and the ability to solve puzzles of varying difficulty levels.
   * Conduct thorough testing to ensure the solver handles edge cases and corner scenarios effectively.
9. **Documentation and Maintenance:**
   * Document the solver's design, implementation details, and usage instructions for future reference.
   * Maintain the solver by incorporating updates, bug fixes, and enhancements based on user feedback and evolving requirements.

By following this methodology, developers can create efficient and reliable Sudoku solvers capable of solving puzzles of varying complexities.

### CODING AND TESTING

**1. Code for Grid Display, Input Fields, Buttons and Controls, Event Handling, Displaying Solutions, Styling and Layout**

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

from sudokutools import valid, solve, find\_empty, generate\_board

from copy import deepcopy

from sys import exit

import pygame

import time

import random

pygame.init()

class Board:

    def \_\_init\_\_(self, window):

        """

        Initializes a Board object.

        Args:

            window: The Pygame window object.

        """

        # Generate a new Sudoku board and create a solved version of it.

        self.board = generate\_board()

        self.solvedBoard = deepcopy(self.board)

        solve(self.solvedBoard)

        # Create a 2D list of Tile objects to represent the Sudoku board.

        self.tiles = [

            [Tile(self.board[i][j], window, i \* 60, j \* 60) for j in range(9)]

            for i in range(9)

        ]

        self.window = window

    def draw\_board(self):

        """

        Draws the Sudoku board on the Pygame window.

        """

        for i in range(9):

            for j in range(9):

                # Draw vertical lines every three columns.

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

                    pygame.draw.line(

                        self.window,

                        (0, 0, 0),

                        (j // 3 \* 180, 0),

                        (j // 3 \* 180, 540),

                        4,

                    )

                # Draw horizontal lines every three rows.

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

                    pygame.draw.line(

                        self.window,

                        (0, 0, 0),

                        (0, i // 3 \* 180),

                        (540, i // 3 \* 180),

                        4,

                    )

                # Draw the Tile object on the board.

                self.tiles[i][j].draw((0, 0, 0), 1)

                # Display the Tile value if it is not 0 (empty).

                if self.tiles[i][j].value != 0:

                    self.tiles[i][j].display(

                        self.tiles[i][j].value, (21 + j \* 60, 16 + i \* 60), (0, 0, 0)

                    )

        # Draw a horizontal line at the bottom of the board.

        pygame.draw.line(

            self.window,

            (0, 0, 0),

            (0, (i + 1) // 3 \* 180),

            (540, (i + 1) // 3 \* 180),

            4,

        )

    def deselect(self, tile):

        """

        Deselects all tiles except the given tile.

        Args:

            tile (Tile): The tile that should remain selected.

        Returns:

            None

        """

        for i in range(9):

            for j in range(9):

                if self.tiles[i][j] != tile:

                    self.tiles[i][j].selected = False

    def redraw(self, keys, wrong, time):

        """

        Redraws the Sudoku board on the game window, highlighting selected, correct, and incorrect tiles, displaying the

        current wrong count and time, and rendering the current keys (potential values) for each tile.

        Args:

            keys (dict): A dictionary containing tuples of (x, y) coordinates as keys and potential values as values.

            wrong (int): The current wrong count.

            time (int): The current time elapsed.

        Returns:

            None

        """

        self.window.fill((255, 255, 255))  # fill the window with white

        self.draw\_board()  # draw the Sudoku board

        for i in range(9):

            for j in range(9):

                if self.tiles[j][i].selected:

                    # highlight selected tiles in green

                    self.tiles[j][i].draw((50, 205, 50), 4)

                elif self.tiles[i][j].correct:

                    # highlight correct tiles in dark green

                    self.tiles[j][i].draw((34, 139, 34), 4)

                elif self.tiles[i][j].incorrect:

                    # highlight incorrect tiles in red

                    self.tiles[j][i].draw((255, 0, 0), 4)

        if len(keys) != 0:

            for value in keys:

                # display the potential values for each tile

                self.tiles[value[0]][value[1]].display(

                    keys[value],

                    (21 + value[0] \* 60, 16 + value[1] \* 60),

                    (128, 128, 128),

                )

        if wrong > 0:

            # display the current wrong count as an "X" icon and a number

            font = pygame.font.SysFont("Bauhaus 93", 30)

            text = font.render("X", True, (255, 0, 0))

            self.window.blit(text, (10, 554))

            font = pygame.font.SysFont("Bahnschrift", 40)

            text = font.render(str(wrong), True, (0, 0, 0))

            self.window.blit(text, (32, 542))

        # display the current time elapsed as a number

        font = pygame.font.SysFont("Bahnschrift", 40)

        text = font.render(str(time), True, (0, 0, 0))

        self.window.blit(text, (388, 542))

        pygame.display.flip()  # update the game window

    def visualSolve(self, wrong, time):

        """

        Recursively solves the Sudoku board visually, highlighting correct and incorrect tiles as it fills them in.

        Args:

            wrong (int): The current wrong count.

            time (int): The current time elapsed.

        Returns:

            bool: True if the board is successfully solved, False otherwise.

        """

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                exit()  # exit the game if the user clicks the close button

        empty = find\_empty(self.board)

        if not empty:

            return True  # the board is solved if there are no empty tiles left

        for nums in range(9):

            if valid(self.board, (empty[0], empty[1]), nums + 1):

                # fill in the current empty tile with a valid number

                self.board[empty[0]][empty[1]] = nums + 1

                self.tiles[empty[0]][empty[1]].value = nums + 1

                self.tiles[empty[0]][empty[1]].correct = True

                pygame.time.delay(63)  # delay to slow down the solving animation

                self.redraw(

                    {}, wrong, time

                )  # redraw the game window with the updated board

                if self.visualSolve(wrong, time):

                    return True  # recursively solve the rest of the board if the current move is valid

                # if the current move is not valid, reset the tile and highlight it as incorrect

                self.board[empty[0]][empty[1]] = 0

                self.tiles[empty[0]][empty[1]].value = 0

                self.tiles[empty[0]][empty[1]].incorrect = True

                self.tiles[empty[0]][empty[1]].correct = False

                pygame.time.delay(63)  # delay to slow down the solving animation

                self.redraw(

                    {}, wrong, time

                )  # redraw the game window with the updated board

    def hint(self, keys):

        """

        Provides a hint by filling in a random empty tile with the correct number.

        Args:

            keys (dict): A dictionary containing tuples of (x, y) coordinates as keys and potential values as values.

        Returns:

            bool: True if a hint is successfully provided, False if the board is already solved.

        """

        while True:

            i = random.randint(0, 8)

            j = random.randint(0, 8)

            if self.board[i][j] == 0:

                if (j, i) in keys:

                    del keys[(j, i)]

                # fill in the selected empty tile with the correct number

                self.board[i][j] = self.solvedBoard[i][j]

                self.tiles[i][j].value = self.solvedBoard[i][j]

                return True

            elif self.board == self.solvedBoard:

                return False  # the board is already solved, so no hint can be provided.

class Tile:

    def \_\_init\_\_(

        self,

        value,

        window,

        x1,

        y1,

    ):

        """

        Initializes a Tile object.

        Args:

            value (int): The value to be displayed in the Tile.

            window (pygame.Surface): The surface to draw the Tile on.

            x1 (int): The x-coordinate of the top-left corner of the Tile.

            y1 (int): The y-coordinate of the top-left corner of the Tile.

        Attributes:

            value (int): The value to be displayed in the Tile.

            window (pygame.Surface): The surface to draw the Tile on.

            rect (pygame.Rect): The rectangular area of the Tile.

            selected (bool): Whether the Tile is currently selected.

            correct (bool): Whether the value in the Tile is correct.

            incorrect (bool): Whether the value in the Tile is incorrect.

        """

        self.value = value

        self.window = window

        self.rect = pygame.Rect(x1, y1, 60, 60)

        self.selected = False

        self.correct = False

        self.incorrect = False

    def draw(self, color, thickness):

        """

        Draws the Tile on the window with a colored border.

        Args:

            color (tuple[int, int, int]): The RGB color value of the border.

            thickness (int): The thickness of the border.

        Returns:

            None.

        """

        pygame.draw.rect(self.window, color, self.rect, thickness)

    def display(

        self,

        value,

        position,

        color,

    ):

        """

        Displays the value of the Tile in the center of the Tile.

        Args:

            value (int): The value to be displayed.

            position (tuple[int, int]): The (x, y) coordinates of the center of the Tile.

            color (tuple[int, int, int]): The RGB color value of the text.

        Returns:

            None.

        """

        font = pygame.font.SysFont("lato", 45)

        text = font.render(str(value), True, color)

        self.window.blit(text, position)

    def clicked(self, mousePos):

        """

        Checks if the Tile is clicked by the mouse.

        Args:

            mousePos (tuple[int, int]): The (x, y) coordinates of the mouse.

        Returns:

            bool: True if the Tile is clicked, False otherwise.

        """

        if self.rect.collidepoint(mousePos):

            self.selected = True

        return self.selected

def main():

    # Set up the pygame window

    screen = pygame.display.set\_mode((540, 590))

    screen.fill((255, 255, 255))

    pygame.display.set\_caption("Sudoku Solver")

    icon = pygame.image.load("assets/thumbnail.png")

    pygame.display.set\_icon(icon)

    # Display "Generating Random Grid" text while generating a random grid

    font = pygame.font.SysFont("Bahnschrift", 40)

    text = font.render("Generating", True, (0, 0, 0))

    screen.blit(text, (175, 245))

    font = pygame.font.SysFont("Bahnschrift", 40)

    text = font.render("Random Grid", True, (0, 0, 0))

    screen.blit(text, (156, 290))

    pygame.display.flip()

    # Initialize variables

    wrong = 0

    board = Board(screen)

    selected = (-1, -1)

    keyDict = {}

    solved = False

    startTime = time.time()

    # Loop until the sudoku is solved

    while not solved:

        # Get elapsed time and format it to display in the window

        elapsed = time.time() - startTime

        passedTime = time.strftime("%H:%M:%S", time.gmtime(elapsed))

        # Check if the sudoku is solved

        if board.board == board.solvedBoard:

            solved = True

        # Handle events

        for event in pygame.event.get():

            elapsed = time.time() - startTime

            passedTime = time.strftime("%H:%M:%S", time.gmtime(elapsed))

            if event.type == pygame.QUIT:

                exit()

            elif event.type == pygame.MOUSEBUTTONUP:

                # Check if a Tile is clicked

                mousePos = pygame.mouse.get\_pos()

                for i in range(9):

                    for j in range(9):

                        if board.tiles[i][j].clicked(mousePos):

                            selected = (i, j)

                            board.deselect(board.tiles[i][j])

            elif event.type == pygame.KEYDOWN:

                # Handle key presses

                if board.board[selected[1]][selected[0]] == 0 and selected != (-1, -1):

                    if event.key == pygame.K\_1:

                        keyDict[selected] = 1

                    if event.key == pygame.K\_2:

                        keyDict[selected] = 2

                    if event.key == pygame.K\_3:

                        keyDict[selected] = 3

                    if event.key == pygame.K\_4:

                        keyDict[selected] = 4

                    if event.key == pygame.K\_5:

                        keyDict[selected] = 5

                    if event.key == pygame.K\_6:

                        keyDict[selected] = 6

                    if event.key == pygame.K\_7:

                        keyDict[selected] = 7

                    if event.key == pygame.K\_8:

                        keyDict[selected] = 8

                    if event.key == pygame.K\_9:

                        keyDict[selected] = 9

                    elif (

                        event.key == pygame.K\_BACKSPACE or event.key == pygame.K\_DELETE

                    ):

                        if selected in keyDict:

                            board.tiles[selected[1]][selected[0]].value = 0

                            del keyDict[selected]

                    elif event.key == pygame.K\_RETURN:

                        if selected in keyDict:

                            if (

                                keyDict[selected]

                                != board.solvedBoard[selected[1]][selected[0]]

                            ):

                                wrong += 1

                                board.tiles[selected[1]][selected[0]].value = 0

                                del keyDict[selected]

                                # break

                            board.tiles[selected[1]][selected[0]].value = keyDict[

                                selected

                            ]

                            board.board[selected[1]][selected[0]] = keyDict[selected]

                            del keyDict[selected]

                # Handle hint key

                if event.key == pygame.K\_h:

                    board.hint(keyDict)

                # Handle space key

                if event.key == pygame.K\_SPACE:

                    # Deselect all tiles and clear keyDict

                    for i in range(9):

                        for j in range(9):

                            board.tiles[i][j].selected = False

                    keyDict = {}

                    # Solve the sudoku visually and reset all tile correctness

                    elapsed = time.time() - startTime

                    passedTime = time.strftime("%H:%M:%S", time.gmtime(elapsed))

                    board.visualSolve(wrong, passedTime)

                    for i in range(9):

                        for j in range(9):

                            board.tiles[i][j].correct = False

                            board.tiles[i][j].incorrect = False

                    # Set solved to True after solving the sudoku:

                    solved = True

        board.redraw(keyDict, wrong, passedTime)

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                return

main()

pygame.quit()

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

from random import randint, shuffle

def print\_board(board):

    """

    Prints the sudoku board.

    Args:

        board (list[list[int]]): A 9x9 sudoku board represented as a list of lists of integers.

    Returns:

        None.

    """

    boardString = ""

    for i in range(9):

        for j in range(9):

            boardString += str(board[i][j]) + " "

            if (j + 1) % 3 == 0 and j != 0 and j + 1 != 9:

                boardString += "| "

            if j == 8:

                boardString += "\n"

            if j == 8 and (i + 1) % 3 == 0 and i + 1 != 9:

                boardString += "- - - - - - - - - - - \n"

    print(boardString)

def find\_empty(board):

    """

    Finds an empty cell in the sudoku board.

    Args:

        board (list[list[int]]): A 9x9 sudoku board represented as a list of lists of integers.

    Returns:

        tuple[int, int]|None: The position of the first empty cell found as a tuple of row and column indices, or None if no empty cell is found.

    """

    for i in range(9):

        for j in range(9):

            if board[i][j] == 0:

                return (i, j)

    return None

def valid(board, pos, num):

    """

    Checks whether a number is valid in a cell of the sudoku board.

    Args:

        board (list[list[int]]): A 9x9 sudoku board represented as a list of lists of integers.

        pos (tuple[int, int]): The position of the cell to check as a tuple of row and column indices.

        num (int): The number to check.

    Returns:

        bool: True if the number is valid in the cell, False otherwise.

    """

    for i in range(9):

        if board[i][pos[1]] == num:

            return False

    for j in range(9):

        if board[pos[0]][j] == num:

            return False

    start\_i = pos[0] - pos[0] % 3

    start\_j = pos[1] - pos[1] % 3

    for i in range(3):

        for j in range(3):

            if board[start\_i + i][start\_j + j] == num:

                return False

    return True

def solve(board):

    """

    Solves the sudoku board using the backtracking algorithm.

    Args:

        board (list[list[int]]): A 9x9 sudoku board represented as a list of lists of integers.

    Returns:

        bool: True if the sudoku board is solvable, False otherwise.

    """

    empty = find\_empty(board)

    if not empty:

        return True

    for nums in range(1, 10):

        if valid(board, empty, nums):

            board[empty[0]][empty[1]] = nums

            if solve(board):  # recursive step

                return True

            board[empty[0]][empty[1]] = 0  # this number is wrong so we set it back to 0

    return False

def generate\_board():

    """

    Generates a random sudoku board with fewer initial numbers.

    Returns:

        list[list[int]]: A 9x9 sudoku board represented as a list of lists of integers.

    """

    board = [[0 for i in range(9)] for j in range(9)]

    # Fill the diagonal boxes

    for i in range(0, 9, 3):

        nums = list(range(1, 10))

        shuffle(nums)

        for row in range(3):

            for col in range(3):

                board[i + row][i + col] = nums.pop()

    # Fill the remaining cells with backtracking

    def fill\_cells(board, row, col):

        """

        Fills the remaining cells of the sudoku board with backtracking.

        Args:

            board (list[list[int]]): A 9x9 sudoku board represented as a list of lists of integers.

            row (int): The current row index to fill.

            col (int): The current column index to fill.

        Returns:

            bool: True if the remaining cells are successfully filled, False otherwise.

        """

        if row == 9:

            return True

        if col == 9:

            return fill\_cells(board, row + 1, 0)

        if board[row][col] != 0:

            return fill\_cells(board, row, col + 1)

        for num in range(1, 10):

            if valid(board, (row, col), num):

                board[row][col] = num

                if fill\_cells(board, row, col + 1):

                    return True

        board[row][col] = 0

        return False

    fill\_cells(board, 0, 0)

    # Remove a greater number of cells to create a puzzle with fewer initial numbers

    for \_ in range(randint(55, 65)):

        row, col = randint(0, 8), randint(0, 8)

        board[row][col] = 0

    return board

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

    board = generate\_board()

    print\_board(board)

    solve(board)

    print\_board(board)

### SCREENSHOTS AND RESULTS

### CONCLUSION AND FUTURE ENHANCEMENTS

In conclusion, the development of a Sudoku solver involves the integration of various techniques and algorithms to efficiently solve Sudoku puzzles. Through constraint satisfaction techniques, backtracking algorithms, and optimization strategies, the solver can systematically explore the solution space and find valid solutions to puzzles of varying complexities. The Sudoku solver presented here demonstrates the effectiveness of combining constraint satisfaction with backtracking, resulting in a reliable and efficient solving tool. By parsing input puzzles, applying constraint satisfaction techniques, executing backtracking algorithms, and validating solutions, the solver can accurately solve Sudoku puzzles and present solutions to users. However, while the current solver performs well, there are opportunities for future enhancements and improvements to further enhance its capabilities and usability.

Future enhancements could focus on performance optimization, exploring advanced optimization techniques like parallelization and distributed computing to reduce solution times. Additionally, integrating advanced heuristics and strategies, such as adaptive search algorithms and machine learning techniques, could improve solver efficiency and adaptability. User interface enhancements, including intuitive features like puzzle generation and hint systems, could enhance the solving experience and attract a broader audience. Furthermore, improving error handling mechanisms and fostering community collaboration could ensure ongoing development and refinement of the solver. In summary, while the current Sudoku solver is effective, continuous improvement and innovation will be key to meeting the evolving needs of Sudoku enthusiasts and researchers in the future.

### REFERENCES

https://www.sudokuonline.io/tips/sudoku-rules

https://medium.com/@alexanderpavlov\_18884/javascript-algorithms-backtrack ing-222cde11842c

https://www.simplilearn.com/tutorials/data-structure-tutorial/backtracking-algo rithm

https://github.com/JackHeTech/Sudoku-Solver