CS Project Report

Title : Sudoku Solver

Name
Harshit Shivhare

Class
12th A

Roll Number
19652874

School
Bardsley English Medium Higher Secondary School,Katni

Acknowledgement

I am profoundly grateful to my esteemed Computer Science teacher, **Mr. Ashish Shrivastava** , for their unwavering commitment to education and guidance throughout this project. Their invaluable insights and mentorship have been instrumental in navigating complex technical aspects, fostering a deeper understanding of the subject matter.

I extend my appreciation to our respected school principal, **Mr. Atul Anupam Abraham** , for creating an environment that prioritizes academic excellence and encourages innovative thinking. his leadership has played a crucial role in shaping a culture of learning, and I am grateful for the continuous support received during this project.

I would also like to express my heartfelt thanks to my friends and my parents. Their unwavering encouragement, understanding, and collaborative spirit have been pivotal in maintaining momentum and enhancing the overall project experience. The collective contributions of these remarkable individuals, including my friends and parents, have played a significant role in the successful realization of this project.
I extend my sincere gratitude to all mentioned for their enduring support and valuable contributions.

Certificate

This certificate is hereby awarded to acknowledge the successful completion of the project report titled **"SUDOKU SOLVER"** by **Harshit Kumar Shivhare.** The project was undertaken as part of the academic requirements at **Bardsley English Medium Higher Secondary School,Katni.**

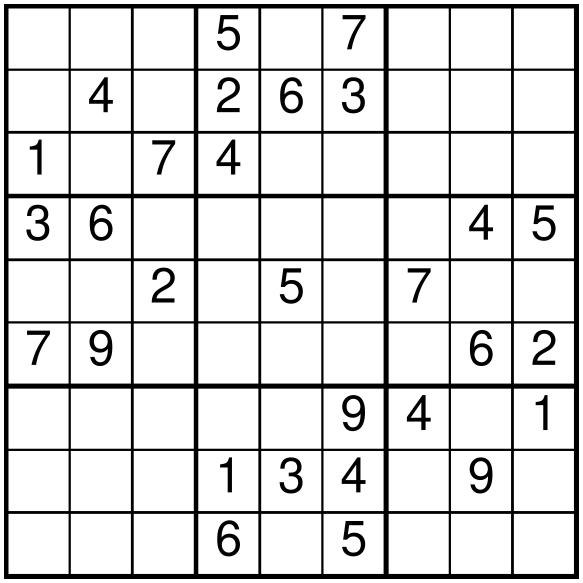
Throughout the duration of this project, **Harshit Kumar Shivhare** exhibited commendable dedication, meticulous effort, and a thorough understanding of the subject matter. The content of the project report reflects original work, and, to the best of my knowledge, has not been submitted for any other academic purpose.

The project was completed under my supervision, and I certify that **Harshit Kumar Shivhare** has met the required standards for academic excellence in their research and presentation. The innovative approach, analytical thinking, and attention to detail demonstrated in this project are noteworthy.

Index

Introduction

What Is Sudoku



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

Sudoku, a logic-based combinatorial number-placement puzzle, has gained immense popularity as a challenging and engaging game. Originating from Switzerland in the 18th century, Sudoku became a global sensation in the early 21st century. The game consists of a 9x9 grid divided into nine 3x3 subgrids, each containing nine cells. The objective is to fill the grid with digits from 1 to 9, ensuring that each row, each column, and each of the nine 3x3 subgrids contains

Sudoku Solving Algorithm

Sudoku-solving algorithms employ various strategies, with Constraint Propagation and the Backtracking Algorithm being prominent.

Brute Force Algorithm:

The brute force approach involves exhaustively checking all possible combinations to find a solution. This method, while straightforward, is computationally expensive, as it systematically explores every potential arrangement without leveraging specific rules or constraints.

Backtracking Algorithm:

In the backtracking approach, the algorithm systematically explores possibilities and backtracks upon encountering conflicts. It efficiently narrows down valid choices, reducing unnecessary exploration. This method proves effective in solving Sudoku puzzles, ensuring adherence to the game's rules.

Constraint Propagation:

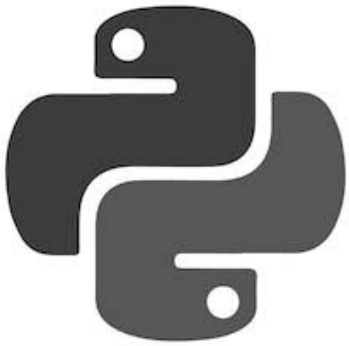
Constraint propagation focuses on iteratively reducing possibilities by applying constraints. This approach involves rules such as elimination and only-choice, continuously narrowing down feasible options. By propagating constraints throughout the Sudoku grid, this method enhances efficiency and reduces the overall solution space.

Objective

The main objective of this project, "Python Sudoku Solver with Visualization," is to develop a sophisticated Sudoku solver using the backtracking algorithm in Python.
The secondary objective is to create an interactive visualization of the Sudoku board, illustrating the step-by-step solving process. This visual representation will serve as a valuable educational tool, allowing users to observe and comprehend the application of the backtracking algorithm.

Tools

Python



Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. Created by Guido van Rossum, Python emphasizes code readability and productivity, making it a popular choice for beginners and experienced developers alike. Its extensive standard library and vibrant community contribute to its widespread use in diverse applications, including web development, data science, artificial intelligence, automation, and more.

External Libraries

1.Random Module

Random module in Python defines a series of functions for generating or manipulating random integers. Python random() is a pseudo-random number generator function that generates a random float number between 0.0 and 1.0, is used by functions in the random module.

2.GUI Library

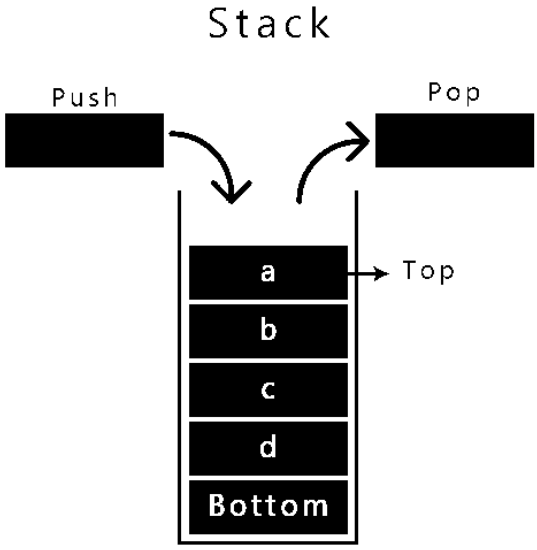
GUI is a visual interface allowing users to interact with software through graphical elements. In Python, libraries like Tkinter facilitate desktop GUI development, while Pygame focuses on game GUIs, offering tools for graphics, events, and sound, making interactive and visually appealing applications.

Methodology

Data Structures

Stack

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear).A stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.



In the Sudoku solver project,the closedset stack keeps track of previously explored cells, allowing the solver to efficiently backtrack when encountering invalid states

2D Array

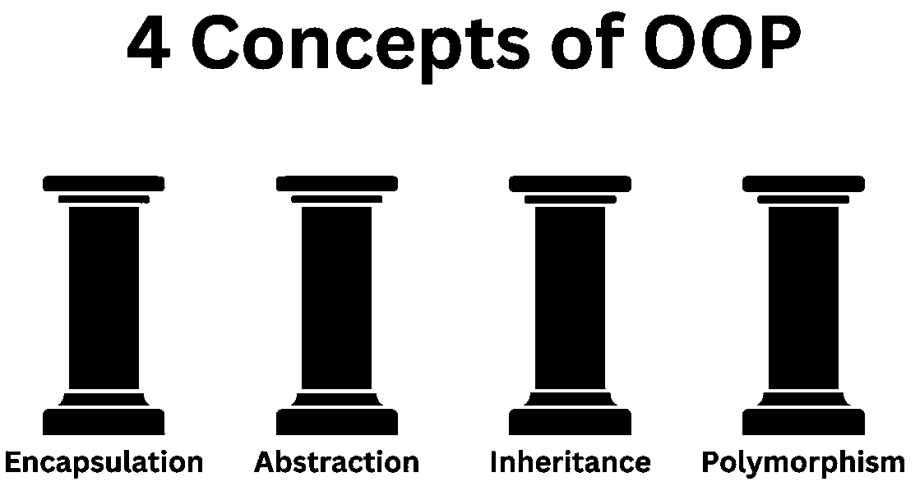
	Col_1	Col_2	Col_3
Row_1	x[0][0]	x[0][1]	x[0][2]
Row_2	x[1][0]	x[1][1]	x[1][2]
Row_3	x[2][0]	x[2][1]	x[2][2]

A two-dimensional array, also known as a 2D array, is a collection of data elements arranged in a grid-like structure with rows and columns. Each element in the array is referred to as a cell and can be accessed by its row and column indices/indexes.

In the context of Sudoku solving, the 2D array represents the sudoku grid, with each cell storing a digit and it's grid coordinate. Efficiently navigating and manipulating this array is crucial for implementing the backtracking algorithm.

OOP

Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming.



The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

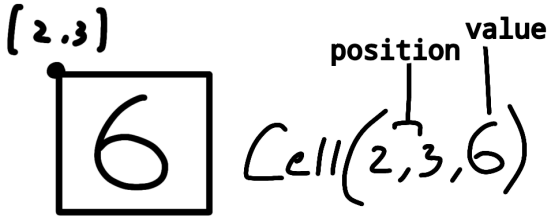
Backtracking

Backtracking, a recursive algorithmic technique, systematically explores and finds solutions to problems with choices and constraints. In the Sudoku solver project, backtracking is pivotal. It guides the systematic filling of Sudoku grid cells. When encountering an invalid state, the algorithm backtracks, undoing choices and exploring alternative values until a valid solution is found or all possibilities are exhausted.

Implementation

Visualizing Sudoku Board

In implementing the sudoku solver the visualization of cell and board plays an important role.
Our objective is to not just create an algorithm that solves give sudoku board,but to breathe life into the process through an intuitive and user friendly visualizatin .It helps understanding how the backtracking algorithm works



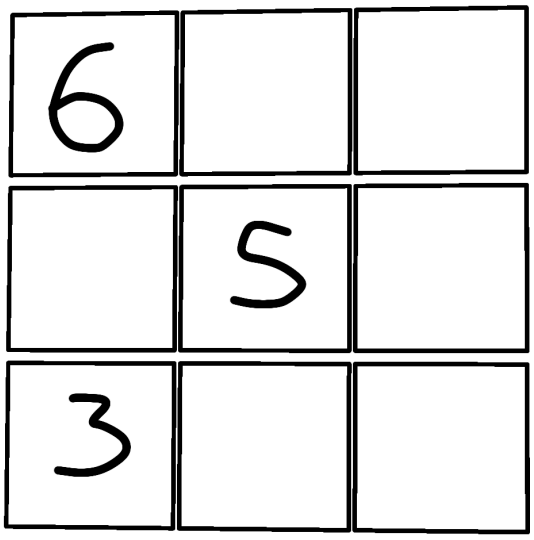
Cells

Each cell instance has position while contains row-column grid coordinates and has a value property while can has any number from 0 to 9 where 0 represents empty cell.

Board

Board instance contains 9x9 2d array of cells.It is initialized with empty cells which then can befilled by user or solver.

```
grid = [
[Cell(0,0,6) , Cell(1,0,0) , Cell(2,0,0)],
[Cell(0,1,0) , Cell(1,1,5) , Cell(2,1,0)],
[Cell(0,2,3) , Cell(1,2,0) , Cell(2,2,0)]
]
```



3x3 grid is show for simplicity

Finding Valid Values

Rows

It loops through the adjacent rows of current cell and removes duplicate value from result

```
def valid_rows(self, cell, result):
    x = cell.x
    y = cell.y
    for i in range(9):
        if x == i:
          continue
        other = self.cells[y][i]
        if other.value in result:
            result.remove(other.value)
```

Columns

It loops through the adjacent column of current cell and removes duplicate value from result

```
def valid_cols(self, cell, result):
    x = cell.x
    y = cell.y
    for j in range(9):
        if y == j:
          continue
        other = self.cells[j][x]
        if other.value in result:
          result.remove(other.value)
```

Sub Grid

It loops through the sub grid of current cell and removes duplicate value from result

```
def valid_subgrid(self, cell, result):
    x = cell.x
    y = cell.y
    startRow = x - x % 3
    startCol = y - y % 3

    for j in range(startCol, startCol + 3):
      for i in range(startRow, startRow + 3):
          if x == i and y == j:
              continue
          other = self.cells[j][i]
          if other.value in result:
              result.remove(other.value)
```

Tracked

It removes duplicate values from result that are present in tracked property of current cell

```
result = []
for i in range(1, 10):
    if i not in curr.tracked:
        result.append(i)
```

Algorithm Implementation

This algorithm run while there are empty cells in the grid.It fills the valid_values in that empty cell,if no values are found it backtracks

Pseudo-Code

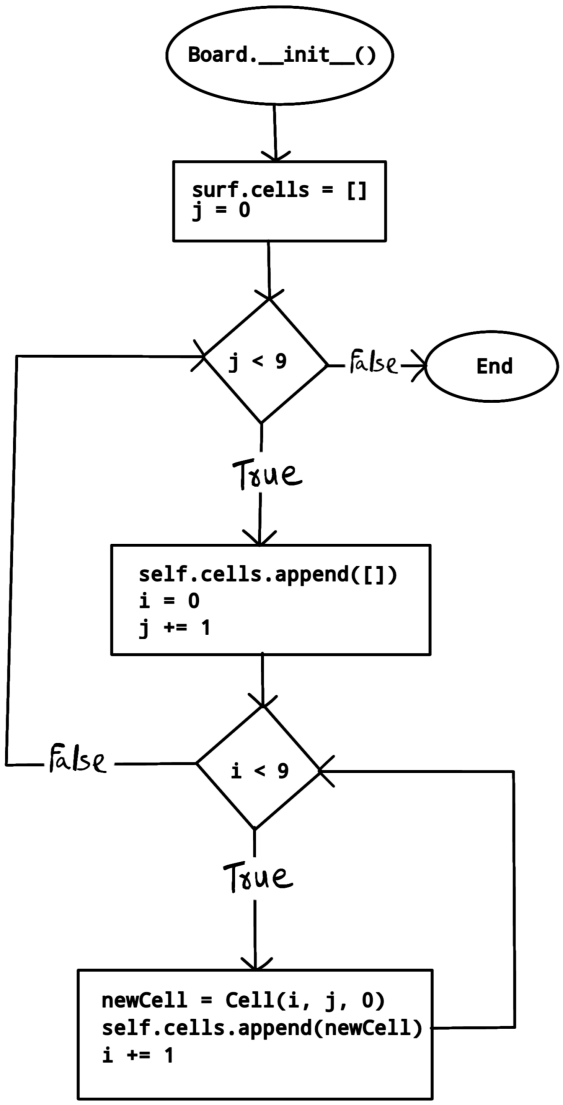
- While grid has empty cells
- Find valid_values for the empty cell
- if no valid values found : backtrack
- else :
    - fill one of valid_values to the empty cell
    - append the cell to closedset
    - append that value to its tracked property

Iterative Backtracking

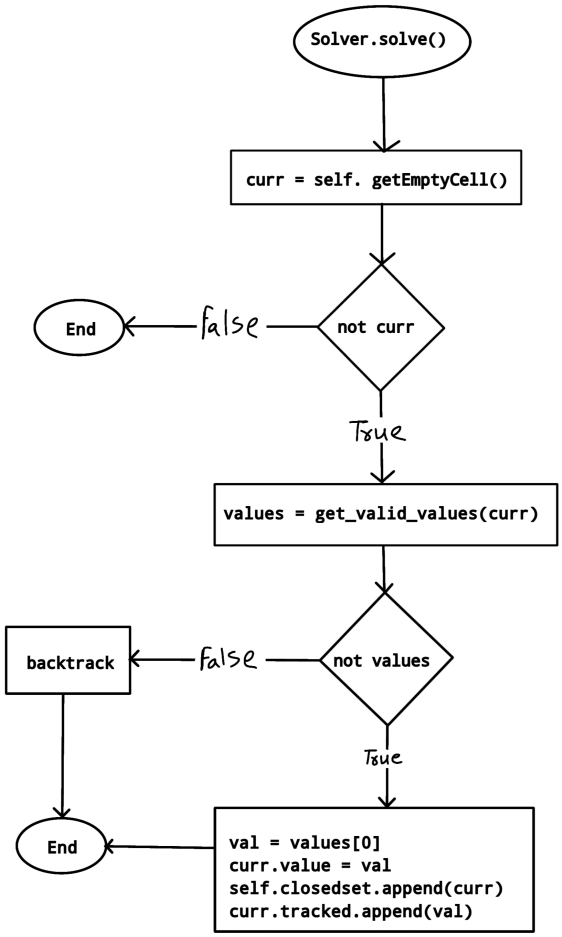
Backtrack function is called when there are no valid_values left for the current cell.It undo previous choices to explore other possibilities

```
def backtrack(self, curr):
    if not self.closedset:
        print("no solution")
        return
    curr.tracked = []
    prev = self.closedset.pop()
    prev.value = 0
```

Flow Chart



Flow-Chart 1



Flow-Chart 2

Source Code

```
import sys
import pygame as pg
from random import random, randint

DIM = 720
BG_COLOR = "#EAE2B7"
WHITE = "#F6F2DE"
GRAY = "#5E412F"
BLACK = "#323031"
ORANGE = "#F07818"
CYAN = "#78C0A8"
YELLOW = "#F0A830"

def clamp(n, minn, maxn):
    return max(min(maxn, n), minn)

pg.font.init()
font = pg.font.Font("mono.ttf", fonsize)
numbers = []
for txt in ["", 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    numbers.append(font.render(str(txt), True, BLACK))

class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

class Solver:
    def __init__(self, program):
        self.cells = program.board.cells
        self.program = program
        self.closedset = []

    def get_empty(self):
        for j in range(9):
            for i in range(9):
                cell = self.cells[j][i]
                if cell.value == 0:
                    return cell
        return False

    def valid_rows(self, cell, result):
        x = cell.x
        y = cell.y

        for i in range(9):
            if x == i:
                continue
            other = self.cells[y][i]
            if other.value in result:
                result.remove(other.value)
```

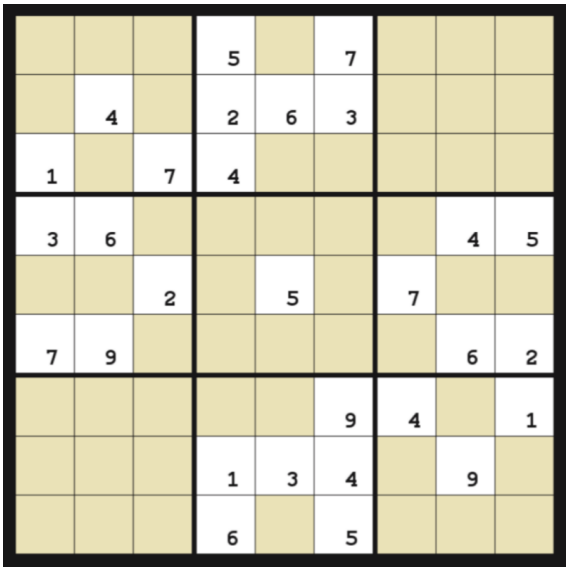
```
    def valid_cols(self, cell, result):
        x = cell.x
        y = cell.y

        for j in range(9):
            if y == j:
                continue
            other = self.cells[j][x]
            if other.value in result:
                result.remove(other.value)

    def valid_subgrid(self, cell, result):
        x = cell.x
        y = cell.y
        startRow = x - x % 3
        startCol = y - y % 3

        for j in range(startCol, startCol + 3):
            for i in range(startRow, startRow + 3):
                if x == i and y == j:
                    continue
                other = self.cells[j][i]
                if other.value in result:
                    result.remove(other.value)

    def backtrack(self, curr):
        if not self.closedset:
            self.program.state = 3
            return
        curr.tracked = []
        prev = self.closedset.pop()
        prev.value = 0

    def solve(self):
        if self.program.state in [0, 2, 3]:
            return
        curr = self.get_empty()
        if not curr:
            self.program.state = 2
            return
        self.program.current = curr

        values = []
        for i in range(1, 10):
            if i not in curr.tracked:
                values.append(i)
        self.valid_rows(curr, values)
        self.valid_cols(curr, values)
        self.valid_subgrid(curr, values)

        if not values:
            return self.backtrack(curr)

        curr.value = values[randint(0, len(values) - 1)]
        curr.tracked.append(curr.value)
        self.closedset.append(curr)
```

```
class Cell:
    def __init__(self, x, y, value=0):
        self.x = x
        self.y = y

        self.value = value
        self.tracked = []

    def render(self, surf, size, offset, color=WHITE):
        weight = 0.5
        x = self.x * size + offset.x + self.x // 3 * weight * 10
        y = self.y * size + offset.y + self.y // 3 * weight * 10

        pg.draw.rect(
            surf,color,
            (x + weight / 2, y + weight / 2, size - weight, size - weight),
        )

        surf.blit(numbers[self.value], (x + size / 2, y + size / 2))

class Board:
    def __init__(self):
        self.cells = []
        self.size = DIM / 11
        self.offset = Vector(self.size, self.size)
        self.size = int(self.size)

        for j in range(9):
            self.cells.append([])
            for i in range(9):
                self.cells[j].append(Cell(i, j))

    def get_coords(self, mpos):
        x = int((mpos[0] - self.offset.x) / self.size)
        y = int((mpos[1] - self.offset.y) / self.size)

        return Vector(clamp(x, 0, 8), clamp(y, 0, 8))

    def render_one(self, surf, cell, color=YELLOW):
        cell.render(surf, self.size, self.offset, color)

    def render_many(self, surf, cells, color=CYAN):
        for cell in cells:
            self.render_one(surf, cell, color)

    def render(self, surf, color=WHITE):
        for j in range(9):
            for i in range(9):
                cell = self.cells[j][i]
                if cell.value == 0:
                    cell.render(surf, self.size, self.offset, GRAY)
                else:
                    cell.render(surf, self.size, self.offset, color)
```

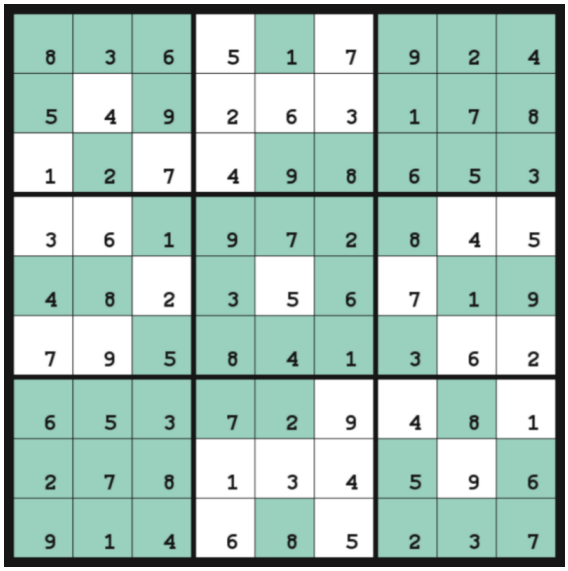
```
class Main:
    def __init__(self):
        pg.init()
        pg.display.set_caption("Sudoku Solver")
        self.screen = pg.display.set_mode((DIM, DIM), pg.RESIZABLE)
        self.clock = pg.time.Clock()
        self.restart()
        self.current = self.board.cells[0][0]

    def restart(self):
        self.board = Board()
        self.solver = Solver(self)
        self.state = 0

    def run(self):
        while True:
            mpos = pg.mouse.get_pos()
            if self.state == 0:
                bpos = self.board.get_coords(mpos)
                self.current = self.board.cells[bpos.y][bpos.x]

            for event in pg.event.get():
                if event.type == pg.KEYDOWN:
                    if event.key == pg.K_s and self.state == 0:
                        print("Starting")
                        self.state = 1
                    digit = event.key - 48

                    if self.state == 0 and digit > -1 and digit < 10:
                        self.current.value = digit

                if event.type == pg.QUIT or event.key == pg.K_q:
                    pg.quit()
                    sys.exit()
                    break

            self.solver.solve()
            self.screen.fill(BG_COLOR)
            self.board.render(self.screen)
            self.board.render_many(self.screen, self.solver.closedset)
            self.board.render_one(self.screen, self.current)
            pg.display.update()
            self.clock.tick(60)

if __name__ == "__main__":
    Main().run()
```

Conclusion

In conclusion, the Python Sudoku Solver project has achieved a harmonious fusion of the backtracking algorithm and Object-Oriented Programming (OOP). The solver not only efficiently cracks Sudoku puzzles but also prioritizes user engagement through intuitive visualization. This strategic blend of algorithmic precision and dynamic interaction marks the project's success in delivering a comprehensive, user-friendly solution for Sudoku enthusiasts of varying skill levels.



Input



Output

References

- **Python Software Foundation** https://www.python.org/
- **GUI Programming in Python** https://wiki.python.org/moin/GuiProgramming
- **Python's Pygame Library** https://pyga.me/
- **Stack** https://www.geeksforgeeks.org/stack-data-structure/
- **2d Array** https://computersciencewiki.org/index.php/Two-dimensional_arrays
- **OOP** https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/
- **What is Sudoku** https://en.wikipedia.org/wiki/Sudoku
- **Some Sudoku Solving Algorithms** https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
- **What is Backtracking** https://en.wikipedia.org/wiki/Backtracking