**SUDOKU SOLVER**

**A PROJECT REPORT**

**for**

**Introduction to AI (AI101B)**

**Session (2024-25)**

**Submitted by**

**RIDHI RAJPUT (202410116100165)**

**SANKET PUNDHIR (202410116100183)**

**NAMRTA SINGH (202410116100129)**

**Submitted in partial fulfilment of the**

**Requirements for the Degree of**

**MASTER OF COMPUTER APPLICATION**

**Under the Supervision of**

**Ms. Komal Salgotra**

### Assistant Professor



**Submitted to**

**Department Of Computer Applications**

**KIET Group of Institutions, Ghaziabad**

**Uttar Pradesh-201206**

**(April - 2025)**

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| Table of Contents | ii |
| 1. Introduction  1.1 Overview  1.2 Problem Statement  1.3 Objectives  1.4 Scope of the project  1.5 Significance | 3-4  3  3  3  4  4 |
| 2. Methodology | 5-6 |
| 3. Code | 7-9 |
| 4. Result | 10 |
| References | 11 |

**ii**

**INTRODUCTION**

* 1. **Overview**

Sudoku is a popular puzzle game that requires players to fill a 9×9 grid with numbers from 1 to 9, ensuring that no number repeats in any row, column, or 3×3 subgrid. The game enhances logical thinking, pattern recognition, and problem-solving skills. However, solving complex Sudoku puzzles manually can be time-consuming and challenging.

With the increasing role of artificial intelligence and algorithmic problem-solving in various domains, developing an automated Sudoku Solver presents an excellent opportunity to apply computational techniques to real-world problems. This project aims to design a Python-based solution that efficiently solves Sudoku puzzles using a systematic approach.

* 1. **Problem Statement**

Manually solving Sudoku puzzles can be tedious and error-prone, especially for puzzles at higher difficulty levels. This project seeks to implement an automated Sudoku-solving program that can:

* Accurately solve any valid Sudoku puzzle.
* Provide a computationally efficient solution using backtracking and constraint satisfaction techniques.
* Offer a user-friendly interface (UI) for ease of use.
  1. **Objectives**

The primary objectives of this project are:

* To develop a Python-based Sudoku Solver that can handle various difficulty levels.
* To implement the Backtracking Algorithm, ensuring an efficient and systematic solution approach.
* To explore and optimize computational techniques to enhance solving speed.
* To create an intuitive user interface (if applicable) for ease of interaction.
* To provide a detailed analysis of the performance of different solving strategies.

3

* 1. **Scope of the Project**

This project focuses on implementing a Sudoku Solver using Python. The scope includes:

* Understanding and defining the rules of Sudoku.
* Developing an efficient algorithm for solving Sudoku using backtracking.
* Testing the solver with different Sudoku puzzles to evaluate accuracy and performance.
* Exploring potential enhancements, such as heuristic optimizations and graphical interfaces for user interaction
  1. **Significance**

The Sudoku Solver project has educational and practical significance:

* It demonstrates problem-solving techniques in programming and computational mathematics.
* It provides a practical use case for backtracking algorithms, which are widely used in constraint satisfaction problems.
* It serves as a foundation for more advanced AI-based Sudoku solvers, incorporating techniques like constraint propagation and machine learning.
* It highlights the importance of algorithm optimization in real-world problem-solving.

4

**METHODOLOGY**

**1. Problem Analysis & Understanding**

Before developing the solver, it was essential to understand the rules and constraints of Sudoku:

* The puzzle consists of a 9×9 grid, divided into 9 smaller 3×3 subgrids.
* Each row, column, and subgrid must contain the numbers 1 to 9 without repetition.
* Some cells are pre-filled, and the solver must determine the remaining numbers while maintaining the constraints.

Research was conducted on existing Sudoku-solving techniques, including brute force, backtracking, constraint propagation, and heuristic-based methods.

**2. Algorithm Selection**

After analyzing different solving techniques, the Backtracking Algorithm was chosen due to its efficiency and correctness. This algorithm follows a recursive approach:

* It starts by finding an empty cell in the Sudoku grid.
* It attempts to place a number (1-9) that follows the Sudoku rules.
* If a valid number is found, it moves to the next empty cell.
* If a conflict occurs, it backtracks and tries the next possible number.
* The process continues until the puzzle is completely solved or no solution exists.

**3. Implementation Using Python**

The Sudoku Solver was implemented in Python following these key steps:

**A. Grid Representation**

* The Sudoku board was represented as a 2D list (9×9 matrix), where empty cells were assigned a placeholder (e.g., 0).

**B. Backtracking Algorithm**

* Implemented a function to check whether a number can be placed in a given cell.
* Designed a recursive function that fills empty cells using backtracking.
* Optimized the algorithm to improve efficiency and reduce unnecessary computations.

5

**C. User Input Handling**

* Allowed users to input Sudoku puzzles manually.
* Ensured validation of input data to prevent invalid grids.

**D. Output Display**

* The solved Sudoku puzzle was displayed in a structured format.
* Enhanced the visualization of the solution using Python libraries.

**4. Testing & Debugging**

* The solver was tested with Sudoku puzzles of different difficulty levels (easy, medium, hard, expert).
* Debugging was performed to identify and fix logical errors or incorrect outputs.
* The algorithm's performance was evaluated in terms of execution time and accuracy.

**5. Optimization & Performance Enhancement**

To improve efficiency, various optimizations were considered:

* **Row, Column, and Subgrid Checks**: Pre-checking possible values before recursion to reduce unnecessary computations.
* **Heuristic Strategies:** Prioritizing cells with the least possibilities first to speed up the solving process.
* **Constraint Propagation**: Reducing the number of choices by analyzing dependencies between cells.

**6. User Interface**

* A UI was explored to allow users to input puzzles easily.
* Python libraries like ipywidgets, matplotlib.pyplot, pytesseract and google.colab.patches.cv2\_imshow were used for proper working.

6

**CODE**

!sudo apt install tesseract-ocr -y

!pip install pytesseract opencv-python ipywidgets

import cv2

import numpy as np

import pytesseract

import matplotlib.pyplot as plt

from google.colab.patches import cv2\_imshow

from google.colab import files

import ipywidgets as widgets

from IPython.display import display, clear\_output

# Set Tesseract path for Colab

pytesseract.pytesseract.tesseract\_cmd = "/usr/bin/tesseract"

uploaded\_image\_path = None

# === IMAGE PROCESSING ===

def preprocess\_image(image\_path):

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

image = cv2.resize(image, (450, 450))

blurred = cv2.GaussianBlur(image, (5, 5), 0)

threshold = cv2.adaptiveThreshold(

blurred, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,

cv2.THRESH\_BINARY\_INV, 11, 2

)

return threshold

# === GRID EXTRACTION ===

def extract\_sudoku\_grid(image):

grid = np.zeros((9, 9), dtype=int)

h, w = image.shape

cell\_h, cell\_w = h // 9, w // 9

for i in range(9):

for j in range(9):

y1, y2 = i \* cell\_h, (i + 1) \* cell\_h

x1, x2 = j \* cell\_w, (j + 1) \* cell\_w

cell = image[y1:y2, x1:x2]

cell = cv2.resize(cell, (100, 100))

cell = cv2.GaussianBlur(cell, (3, 3), 0)

\_, cell = cv2.threshold(cell, 128, 255, cv2.THRESH\_BINARY | cv2.THRESH\_OTSU)

cell = cell[15:85, 15:85] # crop center

7

digit = pytesseract.image\_to\_string(

cell, config='--psm 10 --oem 3 -c tessedit\_char\_whitelist=123456789'

)

digit = ''.join(filter(str.isdigit, digit))

grid[i][j] = int(digit) if digit else 0

return grid

# === SUDOKU SOLVER ===

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

for i in range(9):

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

board[row//3\*3 + i//3][col//3\*3 + i%3] == num:

return False

return True

def solve\_sudoku(board):

for i in range(9):

for j in range(9):

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

for num in range(1, 10):

if is\_valid(board, i, j, num):

board[i][j] = num

if solve\_sudoku(board):

return True

board[i][j] = 0

return False

return True

# === DISPLAY ===

def display\_grid(grid, title):

print(f"\n{title}:")

for row in grid:

print(" ".join(str(num) if num != 0 else "." for num in row))

# === UI ===

def on\_upload\_change(change):

global uploaded\_image\_path

uploaded\_file = list(change['new'].values())[0]

uploaded\_image\_path = "/content/uploaded\_sudoku.png"

with open(uploaded\_image\_path, 'wb') as f:

f.write(uploaded\_file['content'])

print("\n✅ Image uploaded!")

image = cv2.imread(uploaded\_image\_path)

cv2\_imshow(image)

def on\_solve\_clicked(b):

clear\_output()

display(select\_input, upload\_button, solve\_button) 8

if select\_input.value == "Manual":

grid = [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9],

]

display\_grid(grid, "Manual Input")

if solve\_sudoku(grid):

display\_grid(grid, "Solved Sudoku")

else:

print("❌ Could not solve the Sudoku.")

elif select\_input.value == "Upload" and uploaded\_image\_path:

processed = preprocess\_image(uploaded\_image\_path)

cv2\_imshow(processed)

grid = extract\_sudoku\_grid(processed)

display\_grid(grid, "Extracted Grid")

if solve\_sudoku(grid):

display\_grid(grid, "Solved Sudoku")

else:

print("❌ Could not solve the Sudoku.")

else:

print("⚠ Please upload an image first!")

# Widgets

select\_input = widgets.Dropdown(options=["Manual", "Upload"], description='Mode:')

upload\_button = widgets.FileUpload(accept='image/\*', multiple=False)

solve\_button = widgets.Button(description="Solve Sudoku")

upload\_button.observe(on\_upload\_change, names='value')

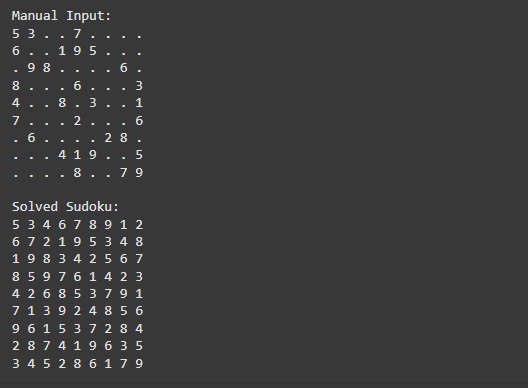
solve\_button.on\_click(on\_solve\_clicked)

display(select\_input, upload\_button, solve\_button)

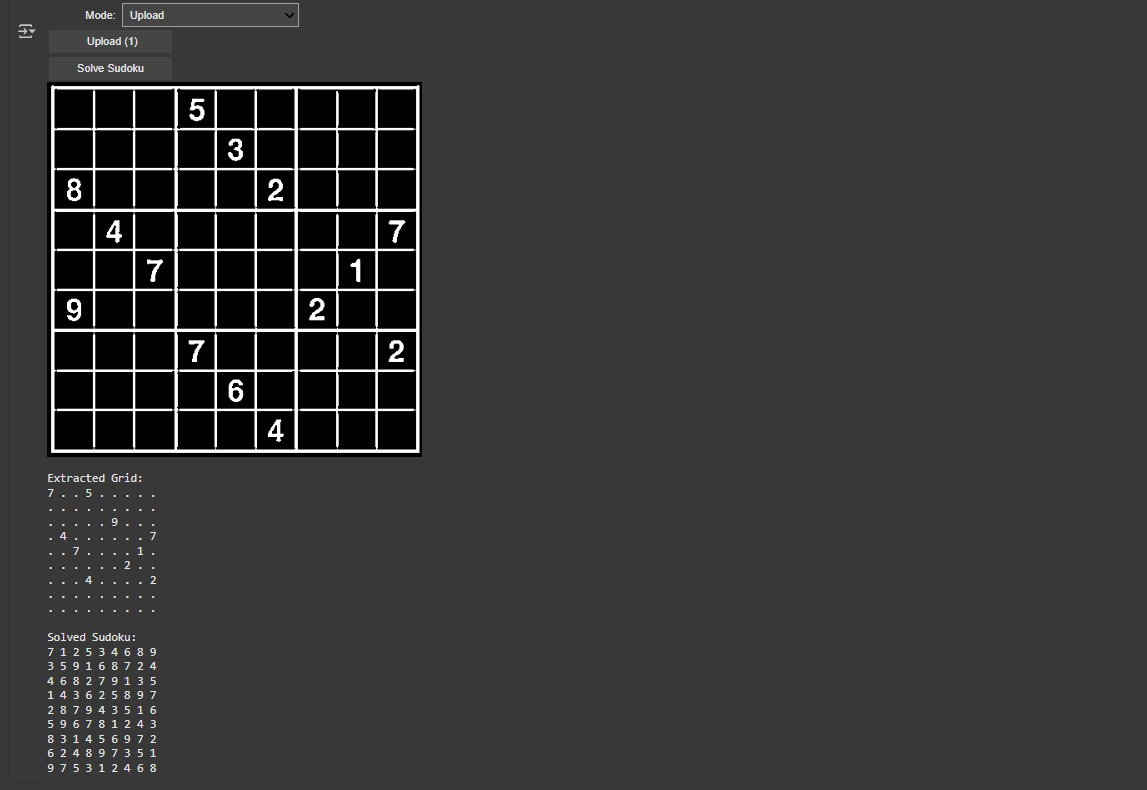
9

**RESULT**

**1. Manual Input**

****

**2. Input by Image**



10

**REFERENCES**

* 1. Sudoku.com- The rules of Sudoku
  2. Github.co- Sudoku Solver in Python
  3. GeeksforGeeks.com- Backtracking Algorithm Explained
  4. W3Schools.com- Python Backtracking Examples

11