Sudoku Solver

Submitted in partial fulfilment of the requirements of the degree

BACHELOR OF ENGINEERING IN COMPUTER ENGINEERING

By

Niyati Savant – 2104156

Sawant Tanishqa - 2104157

Soweda Mohammed Kasim– 2104171

Supervisor

Prof. Sakshi Surve

Computer Engineering Department

Thadomal Shahani Engineering College

Bandra(W), Mumbai - 400 050

University of Mumbai

(AY 2022-23)

Overview

The Sudoku Solver project in Python aims to implement a program that can solve Sudoku puzzles using efficient algorithms and techniques. The scope of this project may include the following features:

Input and Output:

The program should be able to take input in the form of a 9x9 Sudoku grid, either as a string or a list of lists, and display the solved Sudoku grid as output. The input may also include incomplete Sudoku puzzles with missing numbers that need to be filled in.

Sudoku Solving Algorithm:

The project should implement an efficient Sudoku solving algorithm that can solve Sudoku puzzles of varying difficulty levels. This may involve backtracking, a common algorithm used for Sudoku solving, which involves placing numbers in the grid and recursively checking if they satisfy the rules of Sudoku until a solution is found.

User Interface:

The project includes a simple user interface that allows users to input and display Sudoku grids, and view the solution. The user interface is implemented using a graphical user interface (GUI) library Tkinter.

Validation and Error Handling:

The program is be able to validate the input Sudoku grid to ensure that it conforms to the rules of Sudoku, such as having no repeated numbers in rows, columns, or 3x3 sub-grids. The program also handle errors gracefully, such as providing meaningful error messages for invalid input or when a Sudoku puzzle has no solution.

Packages implemented in the Project

Python Tkinter

Python Tkinter is a popular GUI (Graphical User Interface) library that comes with Python as a standard library. It provides a simple and easy-to-use interface for creating desktop applications with graphical components such as buttons, labels, text boxes, and more. Tkinter allows developers to create cross-platform GUI applications that can run on various operating systems, including Windows, macOS, and Linux. In this report, we will discuss some of the popular Tkinter packages available for Python, their features, and use cases. Tkinter is the standard Tkinter library that comes with Python. It provides the basic functionality for creating GUI applications with Tkinter. It includes classes for creating windows, frames, buttons, labels, text boxes, and other graphical components. Tkinter also provides event handling mechanisms for capturing user actions such as button clicks or key presses.

Creating a window using Tkinter:

Windows contains components such as label, radiobutton, text etc.

Syntax:

```
import tkinter as tk

# Create the main window

root = tk.Tk()

root.title("My Window")

root.geometry("300x200")

# Start the main event loop

root.mainloop()
```

Explanation:

root.mainloop(): The window will stay open and the main event loop will keep running until the user closes the window or the mainloop() method is explicitly stopped.

root.title(): To set the title of the window

root.geometry(): To set the dimensions of the window.

Methods:

Entry():

In Tkinter, the Entry widget is used to create a text box where users can enter input. The Entry widget allows you to retrieve the input entered by the user, and you can configure its appearance and behavior using various properties and methods.

We need to first create an instance of the Entry widget by calling the Entry() function, which takes the parent window as the first argument, and optional parameters such as width to specify the width of the Entry widget in characters.

Grid():

The grid method in Tkinter is a geometry manager that allows you to create a grid-like layout for placing widgets (such as buttons, labels, etc.) in rows and columns in a window. You can specify the row and column positions of the widgets, as well as additional options for alignment, padding, and resizing. Firstly we need to create an instance of the widget (such as Label and Button) that you want to place in the grid layout.

We can also specify additional options for alignment, padding, and resizing using keyword arguments in the grid method. For example, you can use sticky parameter to specify the alignment of the widget within its grid cell (e.g., sticky="w" to align left, sticky="e" to align right, sticky="n" to align top, sticky="s" to align bottom). You can use padx and pady parameters to specify the horizontal and vertical padding around the widget, and columnspan and rowspan parameters to specify the number of columns and rows that the widget spans in the grid.

Finally, we start the main event loop by calling the mainloop() method of the Tk class. This method waits for user input and handles events, such as button clicks or key presses, until the window is closed by the user.

Button():

btn = tk.Button(root, text="text", command=perform_task)

text=" text": Specifies the text to be displayed on the button.

command =on_button_click: Specifies the function to be called when the button is clicked.

configure():

The configure() method in Tkinter is used to modify the properties or options of a widget after it has been created. It allows you to change various attributes of a widget, such as text, color, font, size, and more. Here is the syntax for the configure() method:

Syntax:

widget.configure(option=value, ...)

widget: Specifies the Tkinter widget for which you want to modify the properties.

option: Specifies the option or property that you want to change.

value: Specifies the new value for the option.

validatecommand():

The validatecommand option in Tkinter is used to specify a callback function that is called to perform validation on user input in a widget, such as an Entry widget.

Syntax :

validatecommand=(validate_function, validate_mode, validate_data)

validate_function: Specifies the callback function that is called to perform the validation. This function should return a boolean value (True or False) to indicate whether the input is valid or not.

validate_mode: Specifies the validation mode, which determines when the validation function is called.

validate_data: Specifies additional data that can be passed to the validation function as arguments, if needed.

Code:

Sudoku.py

```
from tkinter import *
from solver import solver


root=Tk()
root.title("Sudoku Solver")
root.geometry("425x600")


label=Label(root,text="Enter the numbers").grid(row=0,column=1,columnspan=10)
error_label =Label(root,text="",fg="red")
error_label.grid(row=15,column=1,columnspan=10,pady=5)
solved_label =Label(root,text="",fg="green")
solved_label.grid(row=15,column=1,columnspan=10,pady=5)
cells= {}


def ValidateNumber(P):
    out=(P.isdigit() or P =="")and len(P) < 2
    return out


reg=root.register(ValidateNumber)


def draw3x3grid(row,column,bgcolor) :
    for i in range(3):
        for j in range(3):


e=Entry(root,width=5,bg=bgcolor,justify="center",validate="key",validatecommand=(reg,
"%P"))
```

```
            e.grid(row=row+i+1,column=column+j+1,sticky="nsew",padx=1,pady=1,ipady=5)
            cells[(row+i+1,column+j+1)]=e

def draw9x9Grid():
    color="#FAFA33"
    for rowNo in range(1,10,3):
        for colNo in range(0,9,3):
            draw3x3grid(rowNo,colNo,color)
            if color=="#FAFA33":
                color="#87CEEB"
            else:
                color="#FAFA33"

def clearValues():
    error_label.configure(text="")
    solved_label.configure(text="")
    for row in range(2,11):
        for col in range(1,10):
            cell=cells[(row,col)]
            cell.delete(0,"end")

def getValues():
    board=[]
    error_label.configure(text="")
    solved_label.configure(text="")
    for row in range(2,11):
        rows=[]
```

```
    for col in range(1,10):
        val=cells[(row,col)].get()
        if val == "":
            rows.append(0)
        else:
            rows.append(int(val))
      board.append(rows)
  updateValues(board)

btn=Button(root,command=getValues,text="Solve",width=10)
btn.grid(row=20,column=1,columnspan=5,pady=20,padx=30)
btn=Button(root,command=clearValues,text="Clear",width=10)
btn.grid(row=20,column=3,columnspan=5,pady=20,padx=60)

def updateValues(s):
  sol = solver(s)
  if sol != "No":
    for rows in range(2, 11):
      for col in range(1, 10):
        cells[(rows,col)].delete(0, "end")
        cells[(rows,col)].insert(0, sol[rows-2][col-1])
    solved_label.configure(text="Sudoku solved!")
  else:
    error_label.configure(text="Solution does not exist for this sudoku")

draw9x9Grid()
root.mainloop()
```

Solver.py

```
N = 9
def isSafe(sudoku, row, col, num):
    #to check if number is in same row
    for i in range(9):
        if sudoku[row][i] == num:
            return False

    #to check if number is in same col
    for i in range(9):
        if sudoku[i][col] == num:
            return False

    #To check if the number is present in it's 3x3 grid
    startRow = row - row%3
    startCol = col - col%3
    for i in range(3):
        for j in range(3):
            if sudoku[startRow+i][startCol+j] == num:
                return False
            return True

def solveSudoku(sudoku, row, col):
    if ((row == N-1)and(col == N)):
        return True
```

```
  if col == N:
      row += 1
      col = 0

    if sudoku[row][col] > 0:
      return solveSudoku(sudoku, row, col+1)
    for num in range(1, N + 1):
      if isSafe(sudoku, row, col, num):
        sudoku[row][col] = num
        #checking possibility with next column
        if solveSudoku(sudoku, row, col+1):
          return True
      sudoku[row][col] = 0
    return False

def solver(sudoku):
    if solveSudoku(sudoku, 0, 0):
      return sudoku
    else:
      return "No"
```