8puzzle

Project 2: Sudoku.

Description

This program uses backtracking algorithm to solve a sudoku board. We programmed this project using Python due to it's syntactical simplicity.

You pass in a document with the initial state and it returns the solution of the sudoku board.

The board class created for this project creates a 3d structure that represents the remaining legal values for all the unassigned cells. This was important in order to calculate the necessary heuristics.

Sudoku was defined as a constrain satisfaction problem, where each cell in the board is associated so a set of 10 different numbers that can be assigned to the cell.

It is a constraint satisfaction problem because the value that you put in each cell will depend on the values that neighbouring cells have.

Variables: Cells in the sudoku board.
Domain for Variables: the set of numbers from 0 to 9.
Constraints: Each cell can't have the same number as any other cell in the same row or in the same column. Also, each cell has to have a different value from the block of 9 cells that it is located at in the board.

Resources Used

We had to look at the official Python documentation.

Running the Project

It is recommended that you use a computer running Linux or macOS to run this code. Also ensure that the computer has a clean copy of `Python >3.10` installed. This project has been tested with all releases of Python till the latest one, `Python 3.10`.

This code has been tested on Ubuntu, Arch, and macOS. Keep in mind that you need to have Python installed.

You need to invoke this file from the terminal using one command line parameter, the path to input file.

```
$ python3 sudoku.py path/to/input/file.txt
```

For example, if you wanted to call this file on `Input1.txt`, and both files were in the same directory, you would invoke:

```
$ python3 sudoku.py Input1.txt 1
```

If the argument is ignored, the file will throw an error.

Files Submitted

As stated above, this project has been categorised into multiple module files that add layers of abstraction to better keep track of what is going on.

File	Purpose
board.py	Holds the entire Board object, including the heuristic functions.
sudoku.py	The main file that you should run. This takes the path of the input file as an input and returns the final output as a file. It also has the code of the backtracking algorithm, which ultimately solves the sudoku board.

Miscellaneous

Everything has been commented in Google Docstings.

There are additional comments in the files to explain the obscure blocks of code, to help the graders.

Overall, this project was quite interesting to finish. We had fun after we got through the hurdle of setting the problem up.

Authors

Juan Jose Castano Moreno
jc10536@nyu.edu
@juanjomoreno11

Rishyak Panchal
rishyak@nyu.edu
@rishyak

Course

NYU Tandon's CS-UY 4613 Intro to Artificial Intelligence.

Solutions Generated

These are the solutions generated for all files.

output1.txt: Input 1

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

output2.txt: Input 2

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

output3.txt: Input 3

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

Source Code

This is all the source code of our program.

board.py

```
#!/usr/bin/python3

""" Object file for the Sudoku board.

    This is imported in sudoku.py.

Authors:
    Juan Jose Castano Moreno
    jc10536@nyu.edu

    Rishyak Panchal
    rishyak@nyu.edu
"""

class Board:
    """ Board Class

    Attributes:
        board (list[list[int]]): List symbolising state of the puzzle.
        boxes (list[list[list[int]]]): List symbolising the state of the puzzle,
            but instead of the assignments, it contains the remaining legal
            values of each unassigned cell
    """
    def __init__(self, board) -> None:
        self.board = board

        fullSet = [1,2,3,4,5,6,7,8,9]
        block = []
        self.boxes = []

        for line in board:
            for cell in line:
                if cell == 0:
                    block.append(fullSet[:])
                else:
                    block.append(None)
            self.boxes.append(block[:][:])
            block = []


    def getNeighbors(self, cell : tuple[int]) -> list[int]:
        """ Gets the assigned neighbors of a cell in the sudoku board.
```

```
        Gets the assigned neighbors of a cell in the sudoku board.

    Args:
        cell (tuple[int]): The cell we want to find the neighbors of.

    Returns:
        list[int] : The neighbors of a cell.
    """
    neighbors = []
    row = cell[0]
    col = cell[1]

    for i in range(9):
        if self.board[i][col] != 0:
            neighbors.append(self.board[i][col])
        if self.board[row][i] != 0:
            neighbors.append(self.board[row][i])

    box_row, box_col = row //3  * 3, col //3  * 3
    for i in range(box_row, box_row + 3):
        for j in range(box_col, box_col + 3):
            if i == row or j == col:
                continue
            if self.board[i][j] != 0:
                neighbors.append(self.board[i][j])

    return neighbors


def __str__(self):
    """ Print method mainly for debugging purposes.

    Returns:
        str : String representation of the board.
    """
    output = ""
    for line in self.board:
        for num in line:
            output += str(num) + " "
        output += "\n"

    return output


def mrv(self) -> list[tuple[int]]:
    """ Calculates the minimum remaining values heuristic.

    Returns:
        int : The minimum remaining values heuristic.
    """
    minValue = 10
```

```
        returnList = []
        # Goes over the 3D boxes structure and appends the
        # cells that have the smallest possible values
        for row in range(9):
            for col in range(9):
                if self.boxes[row][col] == None:
                    continue

                if len(self.boxes[row][col]) < minValue:
                    minValue = len(self.boxes[row][col])
                    returnList = [(row, col)]

                elif len(self.boxes[row][col]) == minValue:
                    returnList.append((row, col))

                else:
                    continue

        return returnList


    def degree(self, cells) -> list[tuple[int]]:
        """ Calculates the degree heuristic.

        Args:
            cells (list[tuple[int]]): List of tuples representing
                the cells with equal minimum values.

        Returns:
            list[tuple[int]] : Returns the cells with the highest
                amount of unassigned neighbors.
        """
        maxNeighborsCells = []
        maxNeighbors = 0 #initialize to 0 since it is the least amount of neighbors that
        #This loop goes over the cells that were passed and checks which ones have the m
        for cell in cells:
            # Each cell can only have a maximum of 20 neighbors
            # We calculate the amount of empty neighbors that each cell has
            # by subtracting the assigned neibouts from 20
            numNeighbors = 20 - len(self.getNeighbors(cell)) #number of unassigned neigh

            # Replace max if true
            if numNeighbors > maxNeighbors:
                maxNeighbors = numNeighbors
                # Reset the list
                maxNeighborsCells = [cell]

            # If the same amount of neighbors, add to the list
            elif numNeighbors == maxNeighbors:
                maxNeighborsCells.append(cell)
```

```
            else:
                continue

        return maxNeighborsCells


    def assign(self, cell, assignment) -> bool:
        """ Puts the corresponding assignment in the given cell
        after checking if it is consistent with sudoku rules

        Args:
            cell (tuple[int]) : tuple with the row and col where
                you want to put the assignment
            assignment: int with value that we want to assign to
                a corresponding cell

        Returns:
            Bool : Indicates wether or not the assignment was
                consistent with sudoku rules.
        """
        neighbors = self.getNeighbors(cell)

        #If a neighbors already has the same assignment, then assignment is not consiste
        if (assignment in neighbors):
            return False

        #Put the assignment in the cell and purge all neighbors
        else:
            self.board[cell[0]][cell[1]] = assignment
            self.boxes[cell[0]][cell[1]] = None
            self.purge(assignment, cell)
            return True


    def purge(self, assignment, cell) -> None:
        """ Deletes the current assignment in all the neighbors of cell.
        Args:
            cell (tuple[int]): The row and column of the cell.
            assignment (int): Number that we want to delete from the possible
                values of cell's neighbors.

        Returns:
            None.
            Just purges the possible values for all neighbours of a cell.
        """
        row = cell[0]
        col = cell[1]

        for i in range(9):
            #If cell is already assigned, no need to purge
```

```
            #if cell is already assigned, no need to purge
            if self.boxes[i][col] is None:
                continue
            #if assignment is in neighbors, remove it
            if assignment in self.boxes[i][col]:
                self.boxes[i][col].remove(assignment)
            #If cell is already assigned, no need to purge
            if self.boxes[row][i] is None:
                continue
            # if assignment is in neighbors, remove it
            if assignment in self.boxes[row][i]:
                self.boxes[row][i].remove(assignment)

        box_row, box_col = row // 3 * 3, col // 3 * 3
        #Purging the neighbors from the block
        for i in range(box_row, box_row + 3):
            for j in range(box_col, box_col + 3):
                if self.boxes[i][j] is None:
                    continue
                if assignment in self.boxes[i][j]:
                    self.boxes[i][j].remove(assignment)
```

sudoku.py

```
#!/usr/bin/python3

""" Driver code for the program.
    This is the file that you will invoke from
    the command line like so:
    $ python3 sudoku.py <filename>

Authors:
    Juan Jose Castano Moreno
    jc10536@nyu.edu

    Rishyak Panchal
    rishyak@nyu.edu
"""

import argparse
from copy import deepcopy

from board import Board


def makeFile(name : str, board : Board) -> None:
    """ Creates a file with the solution.
```

```
    Args:
        name (str): The filepath of the file.
        board (Board): The solved board.

    Returns:
        None.
        Just creates a file with the solution.
    """
    # Get output filename
    filename = name.lower().replace('input', 'output')

    # Create and write to output file
    with open(filename, 'w') as f:
        f.write(board.__str__())


def backTrack(board : Board) -> Board:
    """ Backtracking search. Loosely follows
    the algorithm outlined in the assignment document.

    Args:
        board (Board): The board to solve.

    Returns:
        Board : The solved board.
    """
    # Get MRV
    heuristic = board.mrv()

    # If MRV is empty, board is solved
    if not heuristic:
        return board

    # If MRV is more than one cell, call degree
    if len(heuristic) > 1:
        heuristic = board.degree(heuristic)

    # Choose the first from the heuristic
    cell = heuristic[0]

    row = cell[0]
    col = cell[1]

    # We iterate over all legal values
    for value in board.boxes[row][col]:
        # Make a deepcopy of the board for backtracking
        # purposes since python lists and objects are
        # always passed by reference.
        newBoard = deepcopy(board)

        # Set the value, if successful, recurse
```

```
        # Set the value, if successful, recurse
        # Otherwise, go to next value
        if(newBoard.assign(cell, value)):
            result = backTrack(newBoard)

            # If board is complete, return it
            if result is not False:
                return result

    # Failure represented by False
    return False


def main() -> None:
    """ Main function.

    Returns:
        None.
    """
    # Parse arguments
    parser = argparse.ArgumentParser(description='Solve a sudoku problem using backtrack
    parser.add_argument('filename', help='The txt file containing the initial state.')
    cmdline = parser.parse_args()

    # Read input file
    with open(cmdline.filename, 'r') as f:
        input = []
        for line in f:
            line = list(map(int, line.strip().split()))
            input.append(line)

    # Create board
    initial = Board(input)

    # Solve and get final board
    final = backTrack(initial)

    # Make output file
    makeFile(cmdline.filename, final)


if __name__ == "__main__":
    main()
```