ECEN2703 Project: Sudoku Generator
Spring 2021
Kyle Shi

The basic principle behind the game, Sudoku, is to fill out a grid with information while following a certain set of rules. There are many types of sudoku games that one could play including sudoku involving filling the grid with pictures, sudoku boards that are bigger than 9x9, smaller than 9x9, rectangular sudoku grids, etc. The main game of sudoku, however, is centered around filling out a 9x9 board with numbers from 1 to 9. The rules are that a number may not appear in a column, row, or its subgrid more than once. If a board is filled out following these rules then the board is solved. I personally have not played many sudokus in the past so I decided that this would be an interesting project topic to take on. Additionally, sudoku has always been an iconic game in my mind whenever I think of puzzle games. Thus, for my ECEN 2703 project, I decided on coding a sudoku generator.

For the goal of this project, I laid out several requirements for my sudoku generator:

- Level 1: My code is able to generate a random 9x9 sudoku board which is already solved.
- Level 2: My code is able to remove numbers from the already solved board and create a partially filled board which has only one solution.
- Extra Credit: Can create boards with varying difficulties

For this project, my entire code is centered around the concept of a backtracking algorithm. Essentially, while solving a partially filled sudoku board, the algorithm will end up at empty squares where numbers must be filled in; this is a natural consequence of playing sudoku. When upon these empty squares, the algorithm will place a number in the empty square and test to see if any of the rules of sudoku are violated. For example, the algorithm is iterating through the first row of numbers [1, 4, 9, 8, 2, 0 …]. Right now, the algorithm is evaluating the square 0. It will first try 1. 1 clearly doesn't work because it's already in the row, then the algorithm will empty the square and try 2. 2 clearly doesn't work either because it's in the row already as well. The algorithm empties the square yet again and tries 3. This time, 3 works, so it is kept in the square and the algorithm moves onto evaluating the next empty square. As the algorithm moves on, if it finds that no gusses work at a certain square, it will backtrack as many times as needed

(revisit previous squares and empty them) and try different guesses until all of the guesses agree with the rules of sudoku. If this process doesn't work and the algorithm has exhausted all of its guesses, then the grid is unsolvable. To satisfy this backtracking algorithm, I split up level 1 of the project requirements into several helper functions and used these to create a solve_sudoku function which recursively calls on itself to implement the concept of backtracking.

- Level 1: Generate a random 9x9 sudoku board which is already solved.
    - Backtracking Algorithm
        - find_next_empty(grid)
        - valid_guess(grid, guess, row, col)
        - solve_sudoku(grid) -> fill_sudoku(grid)

To code the solve_sudoku function, the 2 helper functions find_next_empty and valid_guess were included to make things easier.

```
# Function that finds the next empty square in the grid, returns its position (row, col)
def find_next_empty(grid):
    # finds the next empty square,  row, col in the grid
    # return row, col
    for r in range(9):
        for c in range(9): # range(9) is 0, 1, 2, ... 8
            if grid[r][c] == 0:
                return r, c

    return None, None  # if no spaces in the puzzle are empty
```

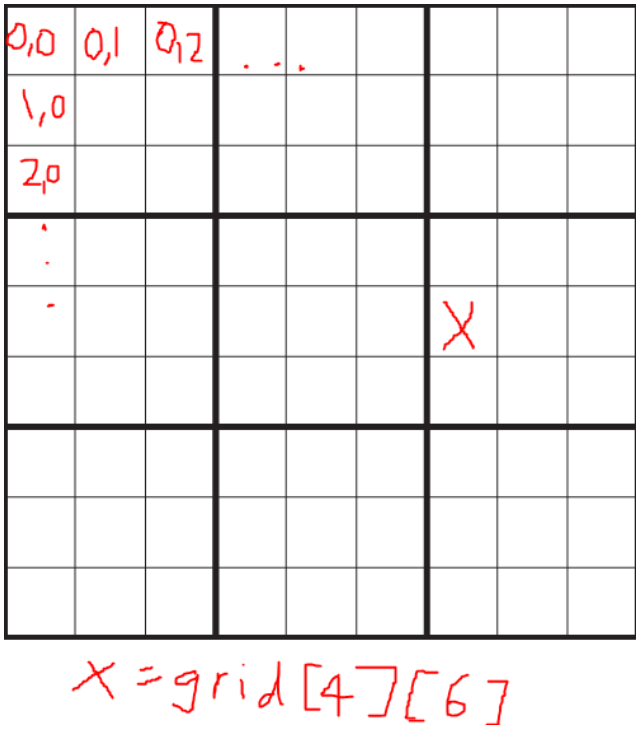
Essentially, the find_next_empty function returns the next empty square in the grid, with its row and column position. I nested 2 for loops so that the function runs through the grid from the top row, left to right, down to the bottom row. It first checks through row 0 and all the columns 0-8 (here the list starts at position 0) in row 0. Then, it moves onto row 1 and checks all the columns in row 1. It does this until it finds an empty square, then it returns that square. If no such square exists, the function returns None, None.

The valid_guess function is the crux behind verifying that the rules of sudoku are being followed every time a guess is placed in an empty square.

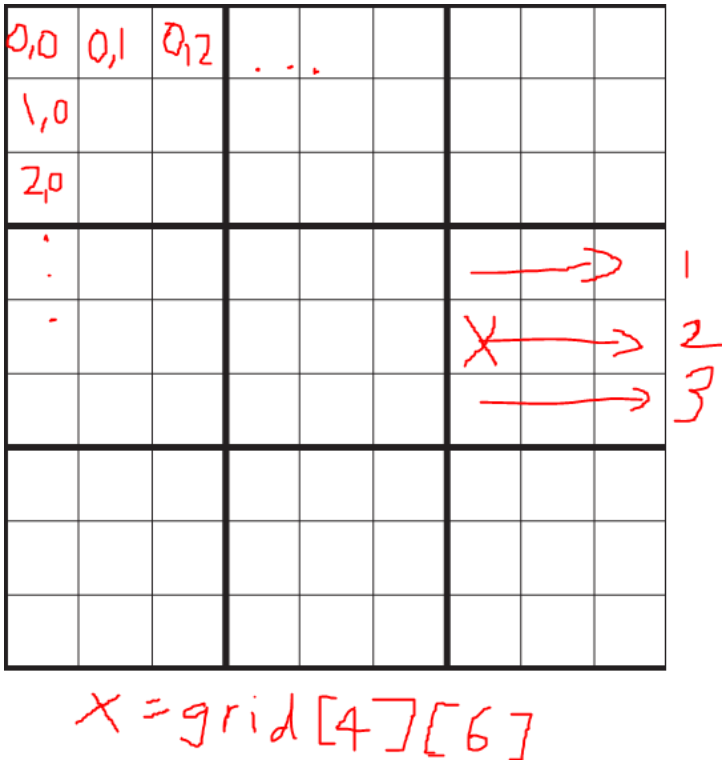
```python
# Function that checks if a guess is valid, returns True or False.
def valid_guess(grid, guess, row, col):
    # returns true or false for a guess
    # if a number guessed isn't in the row/column/box, returns true. else returns false

    # checking the row first, creates a list from the current row being checked
    row_vals = grid[row]
    if guess in row_vals:
        return False # if the guess is already in the row, it's not valid

    # col_vals = []
    # for i in range(9):
    #     col_vals.append(puzzle[i][col])
    # creates a list of the values in the column being checked
    col_vals = [grid[i][col] for i in range(9)]
    if guess in col_vals:
        return False # if the guess is in the column... it's not valid

    # checking to make sure the value isn't in the square
    row_start = (row // 3) * 3 # 10 // 3 = 3, 5 // 3 = 1, 1 // 3 = 0
    col_start = (col // 3) * 3
    # this for loop runs through each element in the square,
    # for example, row is 2, column is 1, then row_start = 0 * 3 = 0, col_start = 0 * 3 = 0.
    # and the loop checks these positions in the square (row position from 0 to 2, col position from 0 to 2)
    #           (0,0) -> (0,1) -> (0,2)
    #           (1,0) -> (1,1) -> (1,2)
    #           (2,0) -> (2,1) -> (2,2)

    for r in range(row_start, row_start + 3):
        for c in range(col_start, col_start + 3):
            if grid[r][c] == guess:
                return False

    return True
```

It is simple to check that the guess isn't in the row. I simply created a variable to store the values in the row of the grid and Python matches the value of guess to each of the row values and returns false if the guess is in the row. Checking that guess wasn't repeated in the column was practically identical except for the fact that I had to create a for loop to grab all the row values corresponding to a single column. E.g, if checking that guess at grid[2][5] isn't repeated in the column, the for loop grabs values from the grid like this: grid[2][0], grid[2][1], grid[2][2], grid[2][3] . . . grid[2][8] and adds them to the list cols_vals. Then guess is compared with the values in cols_vals. Finally, to check that the value isn't in the 3x3 subgrid was fairly more difficult than the above two rules. To do so, the guess value must be compared with the other

values in its subgrid. For example, the guess X would have to be compared with only the values



in its own subgrid. To do this, because X is in position (4,6), it must be compared to the values in squares (4,7), (4,8), (3,6), (3,7), (3,8), (5,6), (5,7), and (5,8). Thus, I start comparing X to the other values in its subgrid by iterating from position (3, 6 -> 8), then moving down to (4, 6->8), and finally (5, 6->8).



To find the starting row and column positions, I simply take the row position of (Xrow//3) * 3 and (Xcol//3)*3. In this case, the starting row is (4//3)*3 = 1 * 3 = 3. Then the starting column is (6//3)*3 = 2*3 = 6. Then I simply created 2 for loops iterating through the rows on the outside from row_start to row_start + 3 and iterated through the columns on the inside loop, checking each value against X. Calculating the starting row/column and ending row/column this way ensures that the for loops I wrote compare the guess X to every value in its subgrid.

With the valid_guess and find_next_empty functions created, the sudoku_solve function becomes much easier to deal with.

```python
# returns True or False depending on whether or not a grid is solveable.
# also alters the grid that's being checked (the inputted grid will be solved if it's solvable)
def solve_sudoku(grid):
  global solution_counter
  # main solver function
  # the input is a list of lists
  # returns true if a solution exists as well as solves the grid
  # don't input an empty grid, it will generate a filled grid but it will be the same each time..

  # choosing position on the puzzle to guess from
  row, col = find_next_empty(grid)

  # checking to see if there are no empty spaces left
  if row is None and col is None:  # find_next_empty returns None if there is no position in the grid that's empty
    solution_counter += 1
    return True  # if this is true, then the puzzle is solved

  # if row isn't None, make a guess between 1 and 9 and place that number
  for guess in range(1, 10): # range(1, 10) is 1, 2, 3, ... 9
    # check if guess valid
    if valid_guess(grid, guess, row, col):
      # if valid, place guess
      grid[row][col] = guess

      if solve_sudoku(grid): # recursive call to solver function
        return True

    #if nothing is valid, try a new number (set that square back to empty)
    grid[row][col] = 0

  # grid is not solvable if none of the tried numbers work
  return False
```

To solve a sudoku grid, first I chose an empty position on the board with the find_next_empty(grid) function and set the values for row, col, equal to the position found by find_next_empty(grid). If, however, there's no empty position in the board, that means the board is already solved so solution_counter increments and True is returned. However, if there still exists an empty square on the board, then the function moves onto the for loop and guesses numbers from 1 to 9. Here the valid_guess() function is ran to see if the guess works for that square, and if it does, then that square is filled with guess. The function is then recursively called and ran again. If however, the guess isn't valid, the grid is set to empty (value of 0) and a new guess is tried. If none of those work, then the grid is not solvable and there is no solution. On the other hand, if the board is solvable, eventually find_next_empty() won't be able to find any empty squares, and the function will increment solution_counter and return True (because there are no empty squares left and all the guesses worked). As you can see, the backtracking recursive

algorithm behind the solve_sudoku function is fairly intuitive and makes good sense when used together with the helper functions I've written. In the end, the function solves an input board as well as returns True or False depending on whether or not a given board is solvable or not.

Now, using the same logic used to write the sudoku_solve function, a sudoku_fill function is implemented.

```
# THIS FUNCTION CREATES A RANDOMIZED FULLY SOLVED SUDOKU GRID USING AN EMPTY GRID, this is NOT meant to be a solver function, it will not tell you whether or not a board is solveable
def fill_sudoku(grid):
  # choosing position on the puzzle to guess from
  row, col = find_next_empty(grid)

  # checking to see if there are no empty spaces left
  if row is None and col is None:  # find_next_empty returns None if there is no position in the grid that's empty
      return grid  # if this is true, then the puzzle is solved

  if grid[row][col] == 0:
    shuffle(number_list) # this shuffles the list everytime there is an empty grid, this makes the new guesses randomized so each filled grid is random
  # if row isn't None, make a guess between 1 and 9 and place that number
    for guess in number_list: # range(1, 10) is 1, 2, 3, ... 9
    # check if guess valid
     if valid_guess(grid, guess, row, col):
        # if valid, place guess
        grid[row][col] = guess

        # recursive call to solver function
        if fill_sudoku(grid):
            return grid

     # if nothing is valid, try a new number (set that square back to empty)
     grid[row][col] = 0

  # grid is not solvable if none of the tried numbers work
  return False
```

The sole difference between this function and the solve_sudoku function is that the guesses being used to fill the board are randomized each time that the function is called and this function takes an empty grid as input. This ensures that when filling out the grid, the algorithm doesn't check guesses incrementally from 1 to 9 but checks the guesses in the order of the random numbers in the numbers list. This way, this function can always take an empty grid (filled with only 0s) as input and generate a fully solved sudoku grid.

- Level 2: Create partially filled board
    - grid_remove_17(grid)
        - get_non_empty()

With the generation of random fully solved sudoku grids, I created a function grid_remove_17 to remove elements from the fully solved grid while ensuring that after removing each element there still existed a unique solution. To do this, I once again employed a

backtracking recursive algorithm.

```
# returns a list of the number of non empty squares in the grid
def get_non_empty(grid):
  non_empty = []
  for rows in range(len(grid)):
    for cols in range(len(grid)):
      if grid[rows][cols] != 0:
        non_empty.append((rows,cols)) # appends the non empty element to the non_empty list
  return non_empty
```

Here, the get_non_empty() function is employed to store all the non 0 (or empty) elements of the board in a list. As similar functions have been explained above in fair detail, the explanation of the logic behind this function is trivial and is omitted here.

```
# This function returns a sudoku board with only 17 clues, takes in the input of a fully solved board
def grid_remove_17(grid):
  global solution_counter

  # this is the number of iterations that are going to be ran through
  iterations = 5
  solution_counter = 1 # before we remove from the grid, there is exactly 1 solution
  non_empty = get_non_empty(grid)
  non_empty_count = len(non_empty)
  while iterations > 0 and non_empty_count > 17:
  # selecting random cell that's not already empty
    row = randint(0,8)
    col = randint(0,8)
    while grid[row][col]==0:
      row = randint(0,8)
      col = randint(0,8)
    # removing a value from the value of non_empty (since we are removing a value from the grid)
    non_empty_count -= 1
    # store the value where the number was replaced
    store_replaced = grid[row][col]
    grid[row][col] = 0

    # copying the entire current grid
    grid_copy = []
    for rows in range(0,9):
      grid_copy.append([])
      for cols in range(0,9): # filling out the copy of the grid from left to right, up to down. (row 1 (left to
        grid_copy[rows].append(grid[rows][cols])

    # counting number of solutions this grid has
    solution_counter = 0
    solve_sudoku(grid_copy) # this will produce a value for the global variable "solution_counter". it will also
    # if number of solutions is != 1, put back in original number (store)
    if solution_counter != 1:
      grid[row][col] = store_replaced
      iterations -= 1
  return grid
```

The grid_remove_17 function takes a fully solved board as input and runs a while loop in which a number on the grid is removed, the grid is copied, then the grid is passed to the solve_sudoku()

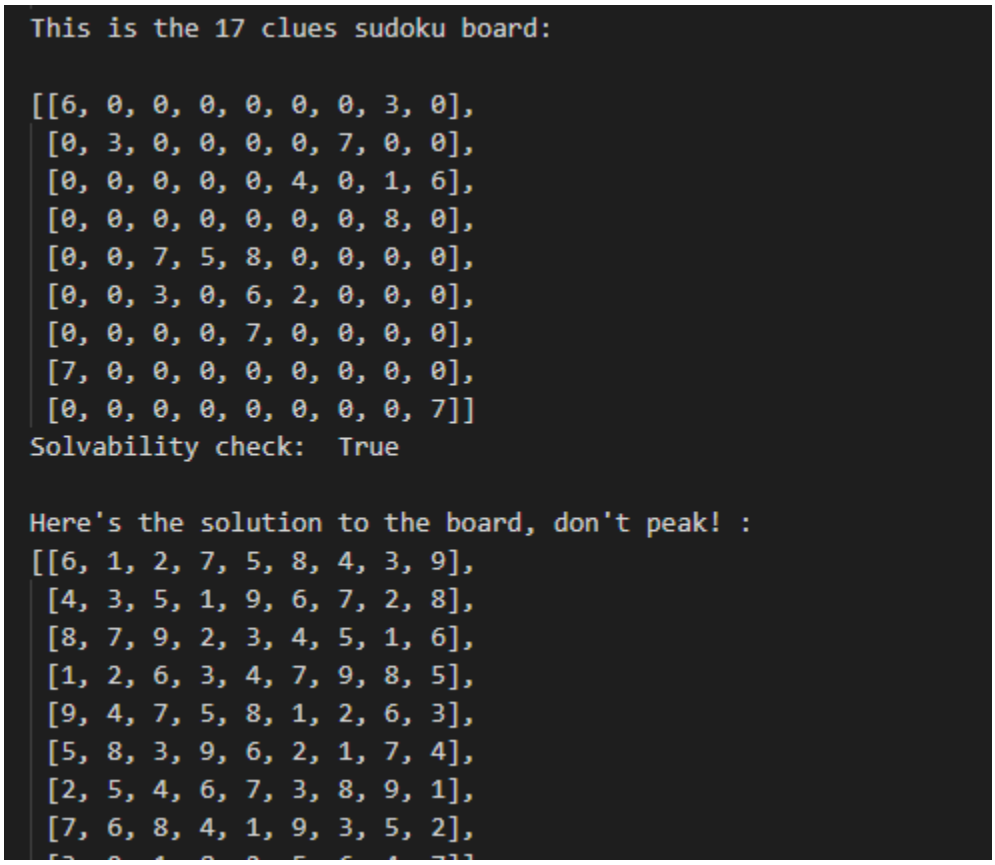
function (once again, employing backtracking recursive logic). Additionally, each time a number is emptied in the grid, the number of nonempty elements in the grid is subtracted by 1. If the grid isn't able to be solved after a number has been removed, and the solution_counter isn't equal to 1, then the number that was removed is placed back and the loop iterates again. The if statement "if solution counter != 1" ensures that in the end, if the current board has more than one solution, the removed number is put back into the grid and the loop is repeated. This ensures that the board always has only one unique solution. The while loop runs until the number of nonempty elements of the grid is equal to 17 (the loop stops at 17 because that is the minimum number of clues needed in a sudoku board for a unique solution).

- Extra Credit: Create board with different difficulties
    - grid_remove_diff(grid, clues: int)

Inherently, the logic behind creating different difficulty boards has already been displayed in my grid_remove_17 function because 17 is simply the number of nonempty numbers that I chose for the board to have. The difference between grid_remove_diff and grid_remove_17 is that grid_remove_diff takes in user input for how many clues they want to leave in the board. Additionally, the number of clues may not be less than 17 because that would result in a board with not a single unique solution and the number of clues may not be greater than 80 because then the board would be completely solved already… This way, the grid_remove_diff function lets the user choose the difficulty of board they'd like created while the boards still maintain unique solutions. For reference, boards with 17-26 clues are generally under the hard category, boards with 27-36 clues are under the medium category, and boards with 36+ clues are considered easy. You get to choose what difficulty of board you'd like!

In the end, I tested my sudoku_solve function on several boards from the puzzles.ca website. It passed all of the tests and returned the same solutions as the ones provided on puzzles.ca. In my code, I've left 2 example test boards from puzzle.ca as well as included the links to these boards and their solutions to demonstrate this. To demonstrate that the solver returns false on an unsolvable board, I purposely created 3 boards that each violate one of the different rules of sudoku. They all returned false, which can be checked in my code file. In my code, I've also default generated 2 boards, one with 17 clues using grid_remove_17 and one with 30 clues using grid_remove_diff (you can change the number of clues to what you'd like as long

as its not smaller than 17 and not larger than 80!). The solutions of each of the boards are also printed underneath them. Finally, all the example cases are printed at the bottom. To run my code file, all you have to do is run it in your editor and check the outputs. In my code, I've commented more specific details regarding each function and the boards that I've printed. This is an example of what a board with 17 clues looks like after printing:



Overall, the project was very successful. I learned a lot regarding sudoku and got to practice implementing recursive functions and the backtracking algorithm. All the criteria which I initially set out for my project including the extra credit were also met. (To run a successful demo of the code, all you have to do is run the code file! I have more instructions as well included in the comments of my code.)