1. Introduction

Sudoku is a number-placement game, the objective of which is to place the numbers from 1 to 9 into a 9*9 grid. The 9*9 grid contains 81 cells, 9 rows, 9 columns and nine 3*3 blocks, so the requirement to successfully fill up a Sudoku grid is that each row, each column and each block must contain all the numbers from 1 to 9, and there are no repetitive numbers. The objective of this program is to construct a sudoku solver and a sudoku generator. The generator can generate a sudoku according to the difficulty given by the argument, and the solver can solve a sudoku puzzle either given by the generator or by an input file.

2. Basic functions

2.1 isfull(sudoku) function

This function is used to check if the Sudoku has been filled up and is valid to satisfy the requirement.

2.2 rowNum(p,sudoku), colNum(p,sudoku) and blockNum(p,sudoku) function

These functions are used to get he already existed numbers on the same raw, colunm and block of the given cell, and return a set.

2.3 initPoint(sudoku) function

the information of each cell is saved into a class, which contains the raw number, column number, the value and the available numbers of this position. And then the sudoku was read by a function called initPoint which returns a list of class contains information from all cells. After collecting all the information, next step is to solve the Sudoku.

```
6 class point:
7     def __init__(self,x,y):
8         self.x=x
9         self.y=y
10        self.available=[]
11        self.value=0
12
13 def initPoint(sudoku):
14     '''read a sudoku, save the information of each cell into a class, and return a list of class'''
15     pointList=[]
16     for i in range(0,len(sudoku)):
17         if sudoku[i] == 0:
18             p=point(i//9,i%9)
19             p.value = 0
20             for j in range(1,10):
21                 if j not in rowNum(p,sudoku) and j not in colNum(p,sudoku) and j not in blockNum(p,sudoku):
22                     p.available.append(j)
23             pointList.append(p)
24         if sudoku[i] != 0:
25             p=point(i//9,i%9)
26             p.value = sudoku[i]
27             p.available.append(sudoku[i])
28             pointList.append(p)
29     return pointList
```

3. The sudoku solver

The solver can be concluded into two steps.

3.1 solve_part(sudoku) function

In the first step, the Sudoku is solved by simply checking the length of available number list and comparing the available numbers. By repeating the comparing process in a ‘while’ loop, sudoku is updated until it cannot be changed anymore. This solving process can solve most easy sudokus. If one Sudoku cannot be solved by this way, it was move forward to next step.

```
273 def solve_part(sudoku):
274     if isfull(sudoku):
275         #print('success')
276         #showSudoku(sudoku)
277         return
278     else:
279        while True:
280            sudoku_old = sudoku[::]
281            check_unique_avialNum(sudoku)
282            update_1_availNum_sudoku(sudoku)
283            pointList = initPoint(sudoku)
284            if  sudoku_old == sudoku:
285                break
286     return
```

3.1.1 check_availNum_1(sudoku) function

This function is used to check the length of all the available number lists. If the length of available number list in a class is 1, this number is filled into the sudoku.

3.1.2 check_unique_avialNum(sudoku) function

This function is to compare each available number of a cell to available numbers of other cells on the same row. If one of the available numbers of one cell is unique among all other available numbers from the row, this number is put into this position. And then compare among the columns, blocks.

3.1.3 update_1_availNum_sudoku(sudoku) function

This function is the repeated operation of check_availNum_1(sudoku) function, which is, after sudoku was updated by previous update_1_availNum_sudoku(sudoku) function, to check the length of all the new available number lists, and update sudoku until it cannot be changed anymore.

3.2 Solve the sudoku by backtracking method

In this step, if the sudoku cannot be solve by previous comparing available numbers step, the sudoku would be solved by the backtracking method to guess the solution.

3.2.1 solve_sudoku(sudoku, pointList) function

Backtracking method in this case is, from the first cell where ‘0’ shows, to try to put the available numbers in this cell one by one. If one available number can fit in this cell, then move forward to the next cell and fill it with the available numbers until sudoku is filled up. If there is no available number can fit in the cell, then go back to the previous cell to try another possibility. When the sudoku is filled up, jump out of the loop.

```
def solve_sudoku(sudoku,pointList):
    if isfull(sudoku):
        #print('success')
        #showSudoku(sudoku)
        return
    if isfull(sudoku) is not True:
        first0 = sudoku.index(0)
        p = pointList[first0]
        candidates = p.available
        for value in candidates:
            if value not in rowNum(p,sudoku) and value not in colNum(p,sudoku) and value not in blockNum(p,sudoku):
                sudoku[first0] = value
                solve_sudoku(sudoku,pointList)
            if isfull(sudoku):
                break
            sudoku[first0] = 0
```

3.2.2 count_solve(sudoku, pointList) function

This function called count_solve is used to control the solutions when generating Sudoku. Firstly, try to solve the given sudoku using the faster way described in the first step. If the Sudoku can be solved, then the solution number of the cell where the first '0' shows was deleted from the available number list. Without this solution number at the first '0' cell, try to solve the sudoku again to check if there are other solutions. The principle of this control is effective is because if there are more than one solution of one sudoku, the solutions are using different numbers within the same cell. So this method can effectively detect another solution. If the sudoku can be solved again, return 'False' to the function, which means there is another solution, otherwise return 'True'.

```
def count_solve(sudoku,pointList):
    '''control the solution of a sudoku is nomore than 1'''
    sudoku_try = sudoku[::]
    pointList_old = pointList[::]

    #solve_sudoku(sudoku_try,pointList)    #first solution
    solve_part(sudoku_try)  #faster
    if isfull(sudoku_try): #remove the first0 candidates from the available numbers
        first0 = sudoku.index(0)
        p = pointList[first0]
        p.available.remove(sudoku_try[first0])
    sudoku_try = sudoku[::]

    solve_sudoku(sudoku_try,pointList) #try the solve the sudoku again whiout the
    if isfull(sudoku_try): #another solution exsits
        return False
    if isfull(sudoku_try) is not True:
        return True
```

4. The sudoku generator

The principle of this generator is first generate a complete sudoku, and then chose a random cell, whose value is then changed into '0', after which the sudoku was tried to be solved. If the sudoku can be solved and there is only 1 solution, continue to change another random cell's value into '0' until reach the required empty cells.

4.1 Generate a complete sudoku

4.1.1 get_random_num(p, sudoku) function

This function is used to get a random number for a given cell from the available numbers.

4.1.2 sudoku_generate_backtracking () function

This function is used to generate a complete sudoku by using the get_random_num(p, sudoku) function to put a random valid number into each cell until it is filled up.

```
def sudoku_generate_backtracking():
    sudoku = [0] * 81
    pointList = initPoint(sudoku)
    i = 0
    while i < 81:
        random_num = get_random_num(pointList[i],sudoku)
        if random_num == -1:
            sudoku = [0] * 81
            i = 0
        else:
            sudoku[i] = random_num
            i += 1
    return sudoku
```

4.2 Generate a sudoku puzzle

sudoku_puzzle_dibble(sudoku, n) function

This function is used to start changing a random cell's value into '0', and then try to solve this sudoku and control the numbers of the solution by using the function count_solve (sudoku, pointList). If the return is 'True', it means the sudoku can be solved and only has one solution. Then continue to change another cell until reach the required empty cells, which is given by the function parameters according to the difficulty given by the input. Finally, return a sudoku puzzle.

```
def sudoku_puzzle_dibble(sudoku,n):
    '''change the random cell's value into '''
    sudoku_dibble = sudoku[:]
    enable_choice = list()
    for i in range(0, 81):
       enable_choice.append(i)
    i = 0
    j = 0
    while i < n:
        random.shuffle(enable_choice)
        random_index = enable_choice[j]
        if sudoku_dibble[random_index] != 0:
            sudoku_dibble[random_index] = 0
            sudoku_tmp = sudoku_dibble[::]
            pointList = initPoint(sudoku_tmp)
            flag = count_solve(sudoku_tmp,pointList)
            if flag:  #one sulution
                i += 1
                #print(sudoku_dibble)
                #print(i)
                enable_choice.remove(random_index)
                random.shuffle(enable_choice)
                j = 0
            else:
                sudoku_dibble[random_index] = sudoku[random_index]
                j += 1
                if j >= len(enable_choice):
                    break
            random_index = enable_choice[j]
        else:
            random_index = random.choice(enable_choice)
    return sudoku_dibble
```

5. Input and the output

5.1 input_sudoku() function

This function is to get the input way of a sudoku from the command line. If the argument is a file name, then the program can read the given file and save the 9*9 grid matrix in to a one-mention list, of which the length is 81.

Otherwise the program would ask a difficulty degree from the command line to generate a sudoku. The difficulty degree of a sudoku in this program is decided by the numbers of the empty cells: easy ones are with 19 cell, medium ones are with 39 cells, hard ones are with 59 cells. It is reported that for a sudoku with only 1 solution, there are at least 17 numbers should be given. So there are at most 64 empty cells in one sudoku to have a unique solution. But for in this program, if the 64 was set to be the requirement of the hard sudokus, the generating process was too slow. If change the requirement into 59 cells, the generating process would be much faster.

5.2 showSudoku(sudoku) function

Finally, this function is used to print out the sudoku.

6. Main program

The main program first gets a sudoku using the input_sudoku() function, and get the 'solve' command from input. The sudoku is fisrt solved by the faster method using solve_part(sudoku_try) function. If the sudoku cannot be solved by this method, it is then solved by the backtracking method using solve_sudoku(sudoku,pointList). If sudoku is solved successfully, print out sudoku using showSudoku(sudoku) function.

7. Conclusion

This program has successfully constructed a sudoku solver and a generator. The solver can solve a sudoku by using 2 methods, which can be given by reading a file or the generator. The generator can generate sudoku puzzles with different difficulty degrees, which is decided by the empty cells. But when generating a hard sudokus, it still a bit slow. In some case, the generating process still takes 1-2 minutes. Generally, when generating a hard sudoku, the generating time varies form 15 seconds to 2 minutes. Further improvement can be made on reducing the generating time.