Development of an Algorithm to Create and Solve Killer Sudokus

Presented to:

Prof. Gilles Caporossi

Authors:

Yuchen Wang (11273123)

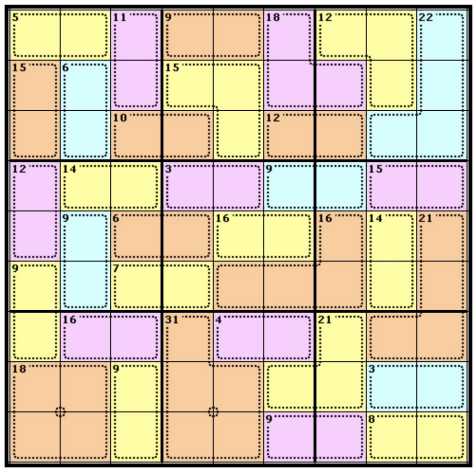
Sarah Dupuis (11274506)

Ziang Guo (11274095)

Jingyi Xiao (11273254)

Introduction

In the era of Covid-19, confinement has led us to try new and different activities in order to stay busy in these times of solitude. Games are a great way to keep the mind busy, and this one game in particular is not only fun, but also a great brain teaser! This game is called Killer Sudoku, a variation of the infamous Sudoku puzzle. The Killer Sudoku consists of a grid of 9x9 cells, which consists of nine 3x3 nonets. Each cell of the grid is an element of a cage, where each cage is associated with a value that indicates the sum of the cells within the cage. In the following example [1], the cages are delimited by different colors.



The rules of the game are as follows:

- Values within the cells vary from 1 to 9.
- No number can appear more than once in a row, in a column, in a nonet and in a cage.
- The sum of all values in a cage needs to be equal to the value on the upper left corner of the cage.

This study consists of developing a series of algorithms to create solvable Killer Sudokus with unique solutions as well as an algorithm to solve any Killer Sudoku. This paper will cover the process and difficulties encountered during the development of these algorithms.

Development of Algorithms

The development process of the algorithms to generate and solve Killer Sudokus consisted of several steps. We begin by generating a filled Sudoku without any cages (i.e. a normal Sudoku). We then start removing values to create a partially filled Sudoku (i.e. a Sudoku game). The Sudoku game is solved at every step of the generation process to ensure that it has a unique solution. We then use the filled Sudoku to generate cages from it such that no value is repeated within a cage. The cages are then combined with the generated Sudoku game to form a Killer Sudoku game. Lastly, we developed an algorithm that is capable of solving the generated Killer

Sudoku games. In addition to the algorithms, we also created a GUI to visualise the generated Killer Sudokus as it was difficult to understand the game in array and dictionary form. The following section will explain in detail each step of this process.

Creation of a Sudoku Game with a Unique Solution

The first step to consider before developing any algorithm to generate Sudoku games is to decide on the data structure in which the Sudoku grid will be stored. While most online tutorials stored the values in a list of lists, we decided to use NumPy 2D arrays as this data structure is significantly more compact and efficient than a list of lists[2].

Having decided on the data structure, we now turn to the second step which involves designing an algorithm to generate a fully filled Sudoku grid. The problem to be solved is defined as follows:

- Given an empty 9x9 grid as shown in Fig 1.1, we wish to find an arrangement of digits {1,2,3, …, 8, 9} such that every digit appears exactly once in each row, column, and nonet (3x3 region in the grid).

In the Sudoku generation literature, there exist two main approaches to generate a filled Sudoku grid, the variation algorithm, and the backtracking algorithm.

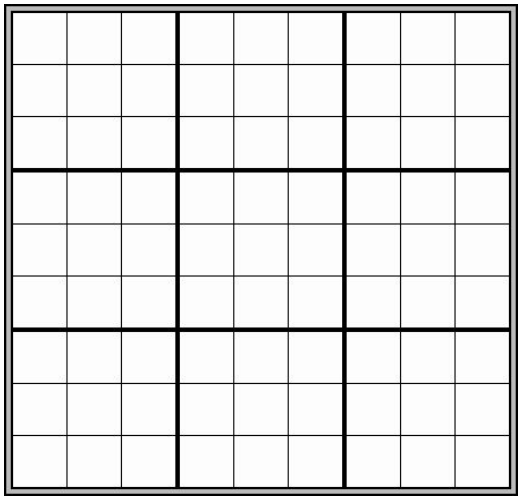


Fig 1.1 Empty Sudoku Grid

The variation algorithm is significantly faster than the backtracking algorithm but generates Sudoku grids that are related to each other. We start with a filled Sudoku grid and perform operations on the grid that preserves its validity, examples include ¼ , ½ and ¾ rotations, vertical and horizontal reflections, and swapping all instances of two arbitrary digits i, j ($i \neq j$) in the grid. For every "seed Sudoku", this algorithm generates 4*4*9! = 5,806,080 variations and is constant time as it is composed of simple operations.

The backtracking algorithm involves filling the grid with random digits at every step and checking to ensure that the digit filled complies with the rules of Sudoku. As the size of the Sudoku is constant, this algorithm has constant time complexity. The worst-case scenario is having to check 9^{9^2} cases. Although it does not make much sense to discuss time complexity when the input size does not change, we will provide a more in-depth analysis of the two algorithms in a more general scenario.

Algorithm 1 Backtracking Algorithm for Sudoku generation

```
Require: G, an empty 9x9 Sudoku grid
Require: check_full_grid(), checks if Sudoku grid is full
Require: check_valid(), checks if Sudoku digit entry is valid
GRID_SIZE ← 9
INT_LIST ← List of digits from 1 to 9.
function POPULATE_GRID(grid)
    for each cell in grid do
        row ← cell row
        column ← cell column
        if grid[row, column] = 0 then
            random_digits ← randomShuffle(INT_LIST)
            for digit in random_digits do
                if check_valid(grid, row, column, digit) then
                    grid[row, column] ← digit
                    if check_full_grid(grid) then
                        return True
                    else if populate_grid(grid) then
                        return True
                    end if
                end if
            end for
            break
        end if
    end for
    grid[row, column] ← 0
end function
```

We now briefly digress from the sudoku generation process to discuss the time complexity of the mentioned algorithms. While we only considered the 9*9 Sudoku in this project, the algorithms developed can in fact be used to generate any Sudoku of size $N^2 \times N^2$ (the grid length needs to be a square number). When we consider the time complexity in this situation, we see that the variation algorithm is still constant time while the backtracking algorithm is O (n^{n^2}) (with n being the range of values in the Sudoku).

For large scale Sudoku generation, it would be a good idea to combine the two algorithms together: using the backtracking algorithms to generate "seed Sudokus" and then use the variation algorithm to create variations. As the aim of the project was not necessarily developing algorithms for large-scale Sudoku generation, we have decided to only implement the backtracking method as the variation algorithm creates a chicken-and-egg problem of requiring a "seed Sudoku" to begin with.

The next step in generating a Sudoku game involves taking the filled Sudoku grid and removing numbers to create a playable Sudoku game. The biggest difficulty encountered in this part was that it was difficult to guarantee that the generated Sudoku game has a unique solution. Indeed, the first version of our generator randomly removed values to generate a playable game. However, after running our Killer Sudoku solver on the grid, we discovered that it had resulted in multiple solutions and most importantly resulted in a different filled Killer Sudoku grid than the original Sudoku grid that the cages were based off of.

Our approach to solving this problem was to create a Sudoku solver that finds all possible solutions to a Sudoku. In the game generating step, every time we remove a number we run this solver to check if the resulting grid has a unique solution and when the resulting grid has multiple solutions we go back to the previous Sudoku and attempt to remove another number. We end this process when the pre-specified number of attempts has been used up. Delahaye, J. P. [3] mentioned that the minimum number of remaining cells that a 9x9 Sudoku can start with and

still return a unique solution is prone to be 17. It has not been proved or rejected that a 9x9 Sudoku with 16 clues has a unique solution due to the overwhelming computation searching all the grids. In our algorithm, in order to create a Sudoku with a unique solution, we limit our normal Sudoku to have only one solution. Killer Sudoku is generated on the top of a normal Sudoku by adding cages as constraints, therefore as long as there is a unique solution in normal Sudoku, the solution in the associated Killer Sudoku will be unique as well. According to Delahaye, J. P., this would result in having at least 17 clues in the Killer Sudoku game.

Algorithm 2 Algorithm for Sudoku game generation

Require: G_f, an filled 9x9 Sudoku grid
Require: solve_sudoku, finds number of solutions to Sudoku grid
Require: attempts, number of retries when the generated game has multiple solution

```
function GENERATE_GAME_GRID(filled_grid, num_attempts)
    game_grid ← filled_grid
    attempts ← num_attempts
    while attempts > 0 do
        cell ← random non-empty cell from game_grid
        row ← cell row
        column ← cell column

        backup ← game_grid[row, column]
        game_grid[row, column] ← 0

        num_solutions ← solve_sudoku(game_grid)
        if num_solutions ≠ 1 then
            game_grid[row, column] ← backup
            attempts = attempts − 1
        end if
    end while
    return game_grid
end function
```

The Sudoku solver algorithm is almost identical to the backtracking Sudoku generation algorithm. The difference being that a counter variable that counts the number of full grid solutions is added and the stopping criterion was amended to stopping only when all combinations have been tried. The solver algorithm has the same worst-case time complexity as the backtracking Sudoku generation algorithm and the solver is run by the global game generation algorithm for a maximum of n^2 times (number of cells in the Sudoku).

Generation of the Cages

In order to generate cages, we return to the previously generated filled Sudoku grid as our canvas. Two different approaches were considered to generate cages. The first one consisted of going through the cells of the full Sudoku in a left to right manner, where at each cell we decided if a new cage would be created or if the cell would be appended to either cage of its adjacent cells. The second approach consisted of following a "random walk" approach within the cells of the full Sudoku in order to form the cages.

We originally planned on only doing the first approach, however, we were worried that there would possibly be repeated patterns. Therefore, we decided to also consider the random walk approach, as the added random element would make a pattern of cages almost impossible. After trying both methods, it was found that they both generate adequate cages.

For both approaches, the structure of the output of the functions remains the same. Each cell of the grid will be assigned to a cage ID. The functions will output a dictionary where the keys are the cage IDs and the dictionary values are the positions of cells per cage ID, as well as the sum of the values of the cells within a cage.

Ordered Cages Approach

The first approach consisted of going through the grid from left to right, first row to last row. There are four possible scenarios to consider:

1. The cell is at the starting point (0,0), where the first value is the row and the second is the column of the grid.
2. The cell is in the first row, but not at the starting point.
3. The cell is in the first column, but not at the starting point.
4. The cell is not in the first row, nor the first column.

Since the first scenario consists of the starting point, we will set this cell to be within cage ID: 0. Since the function goes through the grid from left to right, we then fall into scenario 2.

In scenario 2, a cell can either be appended to the cage of the cell that is to its left or create a new cage. To determine whether we create a new cage or not, we generate a random value that is uniformly distributed between 0 and 1. In addition, we want to limit the size of cages to a maximum of five cells. Therefore, if the size of the cage of the cell to the left is below 5 and if the random value is greater or equal to a fixed threshold, we then decide to append the cell to the cage to the left. If these conditions are not satisfied, we then create a new cage.

In scenario 3, we can either append the cell to the cage of the above cell or we can create a new cage. If the size of the cage of the cell above is below 5, if the random value is greater or equal to a fixed threshold and if the value of the cell is not already in the cage above, then we decide to append the cell to the cage above. If these conditions are not satisfied, we then create a new cage.

Finally, scenario 4 is a combination of scenario 2 and 3. The only difference is that the threshold of the random value for appending above is two times larger than the threshold to append to the left. The reason for this is that we wanted to split the probability between the two possibilities.

After trying different values of fixed thresholds, it was found that a threshold of 0.3 generated good cages. What we mean by “good” is that we wanted a good variety of cage sizes and as well we wanted to limit the number of cages of size one. To do so, we set the probability of creating a new cage lower than the probability of appending to a new cage.

The pseudocode of this algorithm can be found in the following figure.

Algorithm Ordered Cage Algorithm

Require: G, a full Sudoku grid
Ensure: Cages, a dictionary with sums and positions of cages

```
randValue ← Array of 100 random values uniformly distributed between 0 and 1
cageID ← 0
trans ← 0.3
maxSize ← 5
Cages ← Empty dictionary with key: cageID, values: cage sum and cage positions

for each row do
    for each column do
        if upper_left_corner == True then
            Append(Cages, Key: cageID, Sum: G[row,col], Pos: (row,col))
        end if

        if first_column == False then
            left ← getCageID(Cages, (row, col-1))
            leftSize ← getSize(Cages, left)
            leftVals ← getValues(G, left)
        end if

        if first_row == False then
            above ← getCageID(Cages, (row, col-1))
            aboveSize ← getSize(Cages, above)
            aboveVals ← getValues(G, above)
        end if

        if first_row == True and first_column == False then
            rand ← random value from randValue
            if leftSize < maxSize and rand ≥ trans then
                    Append(Cages, Key: cageID, Sum: G[row,col], Pos: (row,col))
            else
                    cageID += 1
                    Append(Cages, Key: cageID, Sum: G[row,col], Pos: (row,col))
            end if
        end if

        if first_row == False and first_column == True then
            rand ← random value from randValue
            if aboveSize < maxSize and rand ≥ trans and G[row,col] not in aboveVals then
                    Append(Cages, Key: above, Sum: G[row,col], Pos: (row,col))
            else
```

```
                    cageID += 1
                    Append(Cages, Key: cageID, Sum: G[row,col], Pos: (row,col))
                end if
            end if

            if first_row == False and first_column == False then
                rand ← random value from randValue
                if aboveSize < maxSize and rand ≥ trans*2 and G[row,col] not in aboveVals then
                    Append(Cages, Key: above, Sum: G[row,col], Pos: (row,col))
                end if
            else
                if  leftSize < maxSize and rand ≥ trans and G[row,col] not in leftVals then
                    Append(Cages, Key: left, Sum: G[row,col], Pos: (row,col))
                else
                    cageID += 1
                    Append(Cages, Key: cageID, Sum: G[row,col], Pos: (row,col))
                end if
            end if
        end for
    end for
    return Cages
```

Random Walk Approach

We call the second approach "random walk" since the cages are arranged like a growing snake with random direction, very similar to the video game *Snake*. We start off with a list of all the cells that have not been assigned to a cage. We then randomly select a cell from the list, assign it to a cage and remove it from the non-assigned list. From the current cell, we can randomly select one direction from up, down, left or right and move one step to select the next cell to append to the same cage, as long as the next cell meets the requirements of the game. If the next cell is valid, we append it to the cage and remove it from the list of non assigned cells. If the selected next cell does not meet the requirements of validity, we remove that direction from the possible transition options and randomly pick another direction again. Once there are no available directions to go, or the current cage has reached its predefined limit of size, we will move to a different cell from the non assigned list, and repeat the whole "random walk" process until there are no more cells in the non assigned list.

A very important part of this algorithm is to define the requirements of validity when selecting the next cell by "walking". We set the following three requirements:

1. The next cell must be in the grid, which means the position coordinate must be between 0 and 8 (include 0 and 8). Since here we did not discuss the scenario of "current cell" as we did in approach 1, there is a possibility that the next cell will not be located inside the grid. For example, if the current cell is at position [8, 8], the next cell can only be left or above of the current cell. If it "walks" outside the grid, this next cell will not meet the requirements.
2. The next cell must not be previously assigned to a cage. We use a non assigned list to check whether a cell has been assigned to a cage or not. If the next cell is not in that list, the requirement will not be met, and we will have to step back and randomly pick a different direction.
3. The third requirement is set by the rules of Killer Sudoku. A cage cannot include the same value twice. For example, if the cage contains already the number 5, we cannot "walk" to the next cell with a value 5.

Theoretically, the maximum size of a cage is 9, which includes numbers 1 to 9. However, we can also predefine a maximum size for cages, such as 5. This will significantly change the difficulty of the game. The minimum of this predefined value is 2, otherwise, the answer for all cells is given.

The pseudocode of this algorithm can be found in the following figure.

Algorithm Random walk cage generator algorithm

Require: A 9×9 matrix satisfying normal Sudoku rule; the maximum cage size, an integer between 2 and 9(included)

Ensure: Every cell is assigned to cages. No cell is assigned to two cages. One cage does not contain cells with same value.

$cages \leftarrow \{\}$
$cageSums \leftarrow \{\}$
function RANDOM WALK GENERAT CAGES($grid$, $maxCageSize$)
    for all $cell$ in $grid$ **do** $notAssigned.append(cell)$
    end for
    randomly shuffle the order of the $notAssigned$
    while $notAssigned \notin \emptyset$ **do**
        $currentCell \leftarrow notAssigned[0]$
        $cages[cageId].append(CurrentCell)$
        $cageSums[cageId] \leftarrow valueofCurrentCell$
        $walkDirectionList \leftarrow [up, down, left, right]$
        while $walkDirectionList \notin \emptyset$ **and** $len(cage[cageId]) < maxCageSize$ **do**
            $direction \leftarrow$randomly pick a direction from $walkDirectionList$
            $nextCell \leftarrow currentCell$ move one step in $direction$
            if $nextCell \notin board$ **or** $nextCell \notin notAssigned$ **or**
                value of $nextCell \in cages[cageId]$ **then**
                $walkDirectionList.remove(direction)$
                Continue
            end if
            $cages[cageId].append(nextCell)$
            $cageSums[cageId] \leftarrow$sum(value Of $cageValues[cageId]$)
            $notAssigned.remove(currentCell)$
            $currentCell \leftarrow nextCell$
            Reset $walkDirectionList$
        end while
        $notAssigned.remove(currentCell)$
        $cageId++$
    end while
end function

Creation of a Killer Sudoku Solver

When thinking of how one can solve a Killer Sudoku, usually a player will start off with cages that have few combinations of possible values. One would start off by filling in the single cages and then will take note of the possible values of cells within cages that have few possible combinations, for example, a cage of size two with sum 17 can only consist of values 9 and 8. Therefore, following this logic, we considered developing a solver based off of possible combinations of values for each cage. Of course, if we were to use this logic, the complexity of the algorithm would be extremely high, as in the worst case there are 81 cells, 9 possible values for each cell, needless to say, the number of possible combinations is incredibly high and would result in a significantly long computation time.

After doing some research, it was found that backtracking is often used to solve similar games. Backtracking consists of going through the grid and trying out values until we obtain a solution that is valid. More precisely, the algorithm will start at the first empty cell it finds, tries different values from 1 to 9 until it finds a value that respects the rules of the game, and then moves on to the next empty cell and so on and so forth, until no values from 1 to 9 are valid. When this happens, the algorithm backtracks to the previous cell and tries different values until it's valid. This backtracking is a form of recursion, where we try to solve the game every time that we add a new value. This method is much faster than considering all combinations and has a worst-case scenario complexity O(n^{n^2}).

In order for a value to be considered valid, it must not already be in the same row, column, nonet, or cage. As well, if the cell for which we are trying to find a value is the only empty cell within a cage, its value must be equal to the designated sum of the cage. If there are other empty cells within the cage, the sum of the value of the cell we are testing and the possible other cells that are not empty within the cage must be smaller than the designated sum of the cage.

In order to make the solver algorithm even more efficient, we can reduce the number of possibilities by already filling the cells that are cages of size one or of only one empty cell, as

they can only be equal to the sum of the cage minus the value of given cells (0 in the case of a cage having size one). Once that is complete, the solver will go through the rows and columns of the game (left to right, top to bottom), in order to find an empty cell. Once an empty cell is found, it will test values 1 to 9 until it finds a value that is valid. If there are no possible values, the cell is not updated, and the algorithm goes back to the previously updated cell and tries different values until the cell is updated with a valid value. If the value is valid, the cell will be set to that value, and the process restarts with the updated grid (recursion of the solver). Once there are no more empty cells, the game is considered to be solved.

Since the original Sudoku that was generated to create the Killer Sudoku was made to have a unique solution, we do not need to worry about obtaining varying solutions with this solver.

The pseudocode of this algorithm can be found in the following figure.

Algorithm Killer Sudoku Solver

Require: $Cages$, a dictionary with sums and positions of cages. $Game$, the grid with a few filled cells.
Ensure: $Game$, the grid with all cells filled

```
for cageID in Cages do
cageValues ← getValues(Cages, cageID, pos, Game)
count0 ← 0
    for val in cageValues do
        if val == 0 then
            count0 += 1
            pos0 ← getPos(Cages, cageID, val)
        end if
    end for
    if count0 == 1 then
            sum ← getValue(Cages, cageID, sum)
            val ← sum − sum(cageValues)
            Game[pos0] ← val
    end if
end for
def solve():
emptyCell ← getEmpty(Game)
if emptyCell == None then
            return True
else
            pos ← emptyCell
end if
for k in (1:9) do
    if valid(Game, Cages, k, pos) == True then
            Game[pos] ← k
    end if
    if solve(Game, Cages) == True then
            return True
    end if
            Game[pos] ← 0
end for
return False
```

Creation of a GUI to Visualize the Killer Sudoku

In this section, we develop a GUI to visualize the whole process of our algorithm. We divided the visualizing process into three parts: generating a Killer Sudoku, generating the game with empty cells and solving the game.

The main Python library utilized in the visualization is Turtle, which is a graphical module and provides convenient animation. It is suitable for visualizing a Killer Sudoku since this module uses commands to draw graphics on a x-y plane. Our Killer Sudoku is a 9 × 9 grid and each cell is a square of same size. Therefore we could simply pre-set the width of the cell size on the x-y plane and move the turtle pen according to the rows and columns in the grid. As the drawing functions and methods in each visualization part are very similar, we will first introduce the general functions being used, and then introduce each visualization process.

The main functions for drawing lines, painting colors, and writing texts are the following:

1. Drawing the grid and the lines: *turple.pensize(), turtle.penup(), turtle.pendown(),* and *turtle.goto(x-axis, y-axis)* "simulate" the selection of the pen size and the line-drawing process on the paper of a person. These are the main functions called for drawing the grid row and column lines. For drawing nine rows and columns respectively, ten solid lines are needed in both directions. Cell size is the element to determine the distance of each line. When it comes to drawing the nonet line (3 × 3 square), pen size was set thicker enough to be distinguishable from the normal lines.

2. Filling in colors for different cages: *turtle.fillcolor(), turtle.begin_fill(), turtle.end_fill(), turtle.forward(cellSize), turtle.right(90).* Specifically, *fillcolor()* and *begin_fill()* functions "inform" the turtle that we will now not only draw lines but also fill in the color in a cell. *Turtle.fillcolor()* can take an RGB index vector as input. In our algorithm we create three random integer numbers, ranging from 80 to 256 (starting from 80 since dark colors conflicting with the cell number color is not desirable) to get R, G, and B. As color is required to fill in the cell square, *turtle.forward(cellSize)* and *turtle.right(90)* helps us to mimic the process of drawing and

filling in the cell squares, in which we will use a for loop to turn our pen four times 90 degrees each to finish a square drawing.

3. Writing a number: *turtle.write()*. Firstly, we go to the x-y coordinate where the number is expected to be placed (either the cell number in the center of the cell, or the "sum" number on the top-left corner in each cage). Secondly, *turtle.write()* will input the corresponding value and write it with font attributes (fonts, font size, etc.).

After introducing the main functions, we will further discuss the detailed implementation of visualization within each part.

1. Generating a Killer Sudoku: beginning with drawing the entire empty grid, the visualization algorithm sequentially fetches different cage IDs and paints the same color for each cell belonging to the same cage ID. In order to avoid the lines being covered by the colors, thicker lines are drawn for nonets only after coloring all of the cells in the grid. The final step is to write down the numbers and the sums on the first cell of each cage.

2. Generating the game with empty cells: the visualization of the game being generated follows the same methods and functions as in 1). The only difference lies in the number stored within the grid variable. As we discussed in the previous section, we create a regular Sudoku with a unique solution, on top of which cages are randomly generated. Therefore, after generating the game with empty cells, we would obtain a new "game grid", where some numbers are removed but the cage colors remain the same as the filled grid. The drawing process is based on this game grid and the complete cage information.

3. Solving the game: since we expect to visualize the entire solving process, we will render the visualization interface at the very beginning (which is different from the other two parts, where the visualization interface only appears after we finish generating the Killer Sudoku game). Similarly, the visualization first returns the same interface as 2), a game grid with empty cells. The most important thing in the visualization of the solver is that it displays the real-time solving process. This is accomplished by consistently overriding backtracking trial numbers as long as it satisfies the validation criteria. At some points, when backtracking finds that there is no number that can continue the game, it will go back to update the previous cell. When this happens, the

visualization tool will go to the position of this previous cell and repaint the cell with the same color of its cage. This will "remove" the number we filled in before and then override a new potential trial number.

The difficulty in GUI mainly lies in visualizing the game solving process. As we just mentioned, *turtle* cannot remove the number that has already been filled in. That is the reason we decided to use the same color to repaint this cell.

Conclusion

In this project, we have coded and implemented an algorithm to generate and solve a Killer Sudoku from scratch. Based on the rules of Sudoku, Killer Sudoku further introduces cages, where the sum of the numbers belonging to the same cage should be the same as indicated in the grid. The series of algorithms developed in this project generate a Sudoku with backtracking. Although one can apply the variation algorithm to 5,806,080 variations from a single "seed Sudoku" in a constant time, we focus on the backtracking algorithm with a 9x9 grid. The complexity of the backtracking algorithm is O (n^{n^2}), where n is the size of the grid (in our project, n remains at 9). The first step of this algorithm is to generate a filled grid with the backtracking algorithm. Secondly, we generate cages by applying either the random walk or the ordered cage approach. Thirdly, we will generate a game grid based on the normal Sudoku with a unique solution. Since a Killer Sudoku has one more constraint (i.e. cage) than a normal Sudoku, a unique solution normal Sudoku will guarantee a unique solution Killer Sudoku. Fourthly, we implement the a solver which will initially fill in the single cages and then use a similar backtracking algorithm to solve the game. Finally, we provide a GUI for visualizing the entire process of generation and solving a Killer Sudoku.

For future work, it would be interesting to develop on the following points:

1. It is noticeable that some adjacent cages are assigned to very similar colors. This makes it difficult for players with visual deficiency to distinguish different cages. It could be interesting to try limiting the possible colors and ensure that no adjacent cages have similar colors.

2. As mentioned in the section “Creation of a Sudoku game with a Unique Solution”, our algorithm generates a Killer Sudoku with cage constraints based on a normal Sudoku with a unique solution. This means that the algorithm will return a Sudoku game with at least 17 clues in Killer Sudoku. Intuitively, Killer Sudoku could have less than 17 clues because there are more constraints applied on the Sudoku. Using a unique-solution normal Sudoku ensures us to generate a valid unique-solution Killer Sudoku, but at the same time it limits us to further reduce the number of clues in Killer Sudoku. In the future improvement, we could design a game generating algorithm directly based on Killer Sudoku cages so that we could remove more clues than our current algorithm. Even if the normal Sudoku derived from this Killer Sudoku may have more than one solution, by adding the cage constraints it will eventually yield a unique solution.

3. Another possible improvement is to return the level of difficulty of a Killer Sudoku game. Currently, we can use the number of attempts to proxy this difficulty level (the higher the attempts, the less number of clues remains in a Killer Sudoku game). As well, we could base the level of difficulty according to the solver running time. In that case, we will need to first solve the Killer Sudoku game, return the solving time, and assign a difficulty level before we return the visualized Sudoku game to the players.

References

[1] killerSudoku.com. (2019). *How To Play Killer Sudoku*. Retrieved 11/2020, from killerSudoku.com: https://killersudoku.com/pages/how-to-play-killer-sudoku

[2] docs.python.org (2020). *Efficient arrays of numerical values*. Retrieved 11/2020, from docs.python.org : https://docs.python.org/3/library/array.html

[3] Delahaye, J. P. (2006). The science behind Sudoku. Scientific American, 294(6), 80-87.