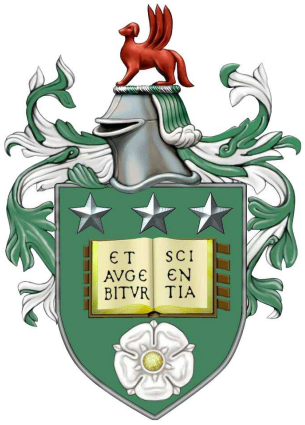
MATH3001: Project in Mathematics



Automatic Puzzle Solving

Supervisor: Dr. Philip Walker
Ryan Chan
200850644

March 3, 2017

Abstract

Sudoku is a logic puzzle, which is played by many people around the world; 9×9 Sudoku puzzles can be found in most newspaper puzzle pages and can be found easily by a quick Internet search. According to a paper published in 2006, the earliest known examples of Sudoku puzzles were published in 1979 in *Dell Pencil Puzzles & Word Games*. In this project, I will be investigating the logic of Sudoku puzzles and will develop an algorithm to solve Sudoku puzzles of arbitrary size. This will involve some logical techniques to remove numbers as possible candidates and also a backtracking algorithm to complete more advanced puzzles. We will only consider Sudoku puzzles of size n, where n is a square number. There do exist Sudokus of other sizes, such as 7×7 and 10×10, but the rules are different to a general Sudoku of size n, where n is a square number. In addition to this, I will be developing and coding a Sudoku grid generator.

All code in this project will be written using Python 3 and can be found in the appendix.

Contents

1 Introduction

A Sudoku puzzle is generally played on a 9×9 board and is then subdivided into 3×3 blocks, as shown in figure 1.

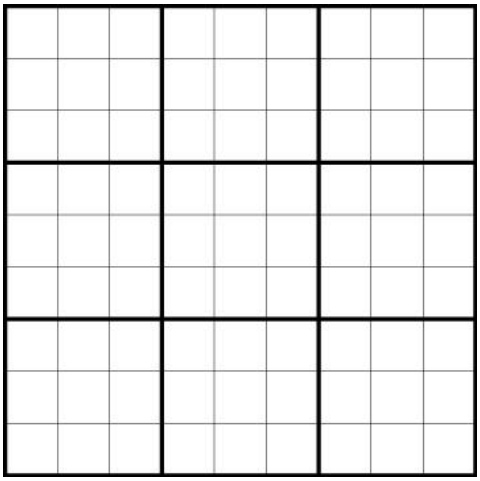


Figure 1: A 9×9 Sudoku grid

Felgenhauer and Jarvis (2006, p.15) state that "the aim of the solver is to complete the grid by filling in a digit in every box in such a way that each row, each column, and each 3×3 box contains each of the digits 1 - 9 exactly once". The definition of a Sudoku can be extended to Sudokus of size n, which is an $n \times n$ board with $\sqrt{n} \times \sqrt{n}$ sub-grids.

1.1 Number of different 9×9 Sudoku grids

The following calculation is based on the work by Felgenhauer and Jarvis (2006). The number of possible 9×9 Sudoku grids has been calculated to be the order of 10^{21}. They calculated this number by firstly labelling the blocks B1 - B9 as shown in figure 2.

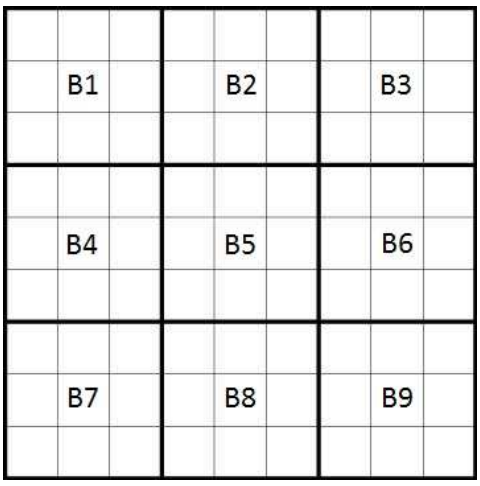


Figure 2: A labeled 9×9 Sudoku grid

By relabelling B1 into the block shown in figure 3, we reduce the number of grids by 9!. This is because it is only required to find the number of Sudoku grids with this form and multiply by 9!. We refer to a grid with this block as B1 to be in *standard form*.

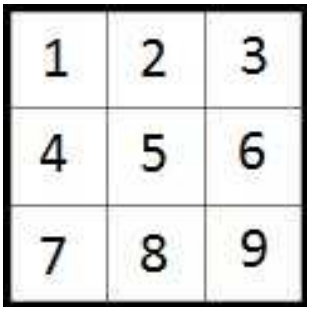


Figure 3: A block in standard form

Felgenhauer and Jarvis (2006, p.16) note that "given blocks B2 and B3, there are other possibilities B2′, B3′ say, such that the number of ways of completing blocks B1, B2 and B3 to full grids is the same as the number of ways of completing blocks B1, B2′ and B3′." Their method of counting the number of 9×9 Sudoku grids is to find the number of possibilities for blocks B2 and B3, and then for these possibilities, they find the number of ways of completing a grid from B1, B2 and B3 by brute force counting.

If B1 is in standard form, then the top row consists of $\{4,5,6\}$ and $\{7,8,9\}$ or a mixture of them. Hence, there are 20 possibilities for the top rows of B2 and B3:

- $\{4,5,6\}$—$\{7,8,9\}$
- $\{4,5,7\}$—$\{6,8,9\}$
- $\{4,5,8\}$—$\{6,7,9\}$
- $\{4,5,9\}$—$\{6,7,8\}$
- $\{4,6,7\}$—$\{5,8,9\}$
- $\{4,6,8\}$—$\{5,7,9\}$
- $\{4,6,9\}$—$\{5,7,8\}$
- $\{5,6,7\}$—$\{4,8,9\}$
- $\{5,6,8\}$—$\{4,7,9\}$
- $\{5,6,9\}$—$\{4,7,8\}$
- $\{7,8,9\}$—$\{4,5,6\}$
- $\{6,8,9\}$—$\{4,5,7\}$
- $\{6,7,9\}$—$\{4,5,8\}$
- $\{6,7,8\}$—$\{4,5,9\}$
- $\{5,8,9\}$—$\{4,6,7\}$
- $\{5,7,9\}$—$\{4,6,8\}$
- $\{5,7,8\}$—$\{4,6,9\}$
- $\{4,8,9\}$—$\{5,6,7\}$
- $\{4,7,9\}$—$\{5,6,8\}$
- $\{4,7,8\}$—$\{5,6,9\}$

The top row $\{4,5,6\}$—$\{7,8,9\}$ can be completed in $(3!)^6$ number of ways, as the other row must be $\{1,2,3\}$, and each set of three numbers can be written in $3!\,(=6)$ ways. This is also true for $\{7,8,9\}$—$\{4,5,6\}$. However, for the other 18 possibilities, as the top row of B2 is a mixture of the sets $\{4,5,6\}$ and $\{7,8,9\}$, the next following rows will also contain some permutation of $\{1,2,3\}$. Hence there are $3 \times (3!)^6$ possibilities.

Therefore, in total there are $(2 \times (3!)^6) + (18 \times 3 \times (3!)^6) = 2612736$ possible arrangements, given that the left-hand block is in standard form. Hence, the total number of possibilities for the top three rows of a 9×9 Sudoku grids is $9! \times 2612736 = 948109639680$.

Then by brute force counting, which is counting the number of ways that we can complete the grid given B1, B2 and B3, Felgenhauer and Jarvis (2006, p.22) computed that in total, there are $6670903752021072936960 \approx 6.671 \times 10^{21}$ valid Sudoku grids.

1.2 Number of essentially different 9×9 Sudoku grids

In the calculation above, solutions that could be transformed into another were counted as different answers. However, the number of essentially different Sudoku grids can be calculated, where "essentially different" means that we allow various possible symmetries. Russell and Jarvis (2006, pp.54-55) note that the list of operations that we can perform to still get a valid Sudoku grid is:

1. Relabel (i.e. permute) the 9 digits;
2. Permute the three vertical bands;
3. Permute the three horizontal bands;
4. Permute the three columns within a vertical band;
5. Permute the three rows within a horizontal band;
6. Any reflection or rotation.

A *vertical band* is defined as three blocks in a vertical 9×3 arrangement and a *horizontal band* is defined as three blocks in a horizontal 3×9 arrangement. By using group theory, Russell and Jarvis (2006) found that the number of essentially different Sudoku solutions is $2297902829591040 \approx 2.297 \times 10^{15}$.

1.3 Uniqueness of 9×9 Sudoku puzzles

Gordon Royle from The University of Western Australia (2017) has a collection of over 49151 examples of Sudokus with 17 clues with a unique solution, but has not found any fewer than 17. For several years, many have tried to find a 16 clue Sudoku puzzle with a unique solution, however, McGuire *et al.* (2011, p.190) have shown by performing an exhaustive computer search that there are no 16 clue puzzles with a unique solution.

2 Notation and Definitions

When referring to a cell on the board, (i, j) will be used to denote the cell at the intersection of row i and column j. The *mark-up* of a cell is the list of possible numbers that cell can be.

We call the *peers* of a cell (i, j) the squares that share the same row i, or the same column j, or the same $\sqrt{n} \times \sqrt{n}$ sub-grid. For example, in a 9×9 Sudoku puzzle, a cell has 20 peers. The shaded region in figure 4 shows the peers for cell $(1, 1)$.

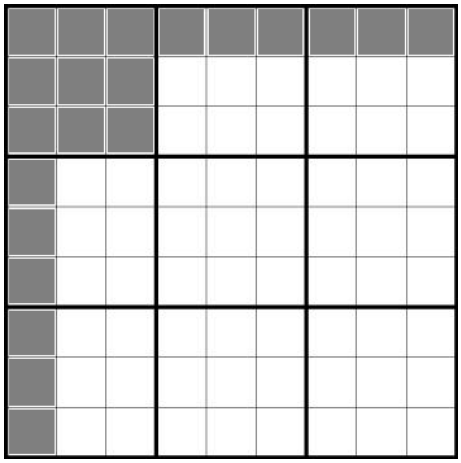


Figure 4: Peers of the cell (1,1)

In Python, a list of lists will be the data structure used to code a Sudoku puzzle and zeros will be used to indicate empty cells. For example, figure 5 shows how a possible Sudoku puzzle will be coded in Python.

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

```
[
[9,0,0,2,3,7,6,8,0],
[0,2,0,8,4,0,0,7,3],
[8,0,7,1,0,5,0,2,9],
[0,0,4,5,9,8,3,0,0],
[2,0,0,0,0,1,0,0,6],
[5,1,0,0,0,0,0,4,7],
[4,0,1,3,0,6,2,9,5],
[0,5,0,9,1,0,7,3,8],
[3,0,8,0,5,0,0,0,0]]
```

Figure 5: An example of a 9×9 Sudoku puzzle coded in Python

3 An Algorithm to solve Sudoku puzzles logically

3.1 Simple Sudoku Puzzles

In this section, we develop an algorithm, which solves a set of Sudoku puzzles using logic rules. Agnes M. Herzberg and M. Ram Murty (2009, p.708) state that every Sudoku square is a Latin square, but not conversely because of the condition on the 3×3 sub-grids. The Concise Oxford Dictionary of Mathematics (2014) gives the definition of a Latin square as: "A square array of symbols arranged in rows and columns in such a way that each symbol occurs exactly once in each row and once in each column." This gives a starting point to develop an algorithm to solve elementary Sudoku puzzles by using this Latin square property of Sudokus and the mark-up of the puzzle. Recall, the definition for the *mark-up* of a cell is the set of possible numbers that the cell can contain. In Python, this can be coded using the function *possibilities_puzzle*, which can be found on pages 27 - 28 in the appendix. This loops through each cell in the puzzle and marks up the cell if it is an unknown entry. This kind of mark-up only checks the values within its row, column and square and eliminates them values as the possible candidates for a cell.

The cell order which the algorithm searches through the entries is shown in figure 6.

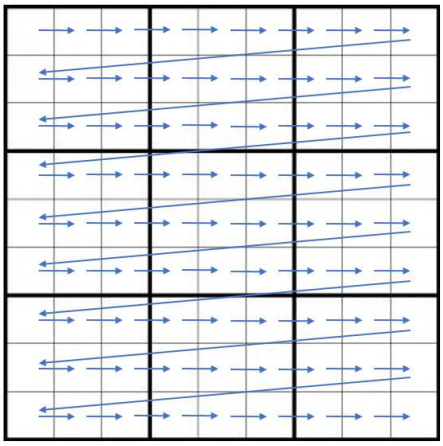


Figure 6: The cell order used to search for entries

To illustrate how this function works, the result of calling *possibilities_puzzle* on the Sudoku in figure 5 is shown in figure 7 (Python result on the right).

After marking up every cell in the puzzle, if the mark-up of a cell has only one possibility, then that must be the solution for that particular cell. In terms of coding this into Python, the algorithm must look for lists of length one, and if it finds these, then it will lock that value into the cell. For example, in the Sudoku in figure 7, in cell $(1, 2)$, the only possibility is 4, hence that must be the answer in that cell. This was achieved by the *definite_answers* function, which is on page 28 on line 118 in the appendix. This loops through the entries of the puzzle to find lists of length 1, and then locks in those answers.

9	[4]	[5]	2	3	7	6	8	[1,4]
[1,6]	2	[5,6]	8	4	[9]	[1,5]	7	3
8	[3,4,6]	7	1	[6]	5	[4]	2	9
[6,7]	[6,7]	4	5	9	8	3	[1]	[1,2]
2	[3,7, 8,9]	[3,9]	[4,7]	[7]	1	[5,8,9]	[5]	6
5	1	[3,6,9]	[6]	[2,6]	[2,3]	[8,9]	4	7
4	[7]	1	3	[7,8]	6	2	9	5
[6]	5	[2,6]	9	1	[2,4]	7	3	8
3	[6,7,9]	8	[4,7]	5	[2,4]	[1,4]	[1,6]	[1,4]

```
[
[9,[4],[5],2,3,7,6,8,[1,4]],
[[1,6],2,[5,6],8,4,[9],[1,5],7,3],
[8,[3,4,6],7,1,[6],5,[4],2,9],
[[6,7],[6,7],4,5,9,8,3,[1],[1,2]],
[2,[3,7,8,9],[3,9],[4,7],[7],1,[5,8,9],[5],6],
[5,1,[3,6,9],[6],[2,6],[2,3],[8,9],4,7],
[4,[7],1,3,[7,8],6,2,9,5],
[[6],5,[2,6],9,1,[2,4],7,3,8],
[3,[6,7,9],8,[4,7],5,[2,4],[1,4],[1,6],[1,4]]]
```

Figure 7: The mark-up of the Sudoku in figure 5

For 'easy' Sudoku puzzles, using this technique and locking in the values at each step is sufficient to find the solution. In order to do this, it is necessary to keep updating the Sudoku grid, so that after an answer is locked in, it would then delete itself from the possibilities of its peers. Although at this stage, it is possible to keep looping the function *possibilities_puzzle* and keep marking up the cell, more advanced techniques that will be developed later require eliminating possibilities in different ways, and hence it is better to find the numbers that have been locked in and remove it from its peers. This is coded in the *update_possibilities* function, which is shown on page 28 in the appendix. For example, in section 3.2.3, we implement the 'naked pairs' technique, which is a way to narrow down the candidates for a cell. If the algorithm kept marking up the puzzle as it does with *possibilities_puzzle*, then the candidates that were eliminated previously would reappear again as potential solutions. This is because this function only checks whether or not a possible candidate violates the rule that the digits 1 - 9 can only appear once in a row, column or square. Hence, it is required that the *update_possibilities* function is used instead to update the puzzle and *possibilities_puzzle* is only called at the start to mark-up the puzzle, so that the algorithm can begin solving it.

Figure 8 shows the step-by-step process of how this algorithm works so far.

By looking at each step, the algorithm first marks up the puzzle and then it looks for entries with lists of length one. The algorithm then locks in the answers, and the updates the puzzle, so it can loop again. For example, looking at step from 1 to 2, the cells $(1, 2)$ and $(1, 3)$ are [4] and [5] respectively, and so the algorithm locks these answers in. Then at the next step, cell $(1, 9)$ is updated to [1], as 4 can no longer be a possibility in that cell, by the condition that a row must only contain digits 1 - 9 exactly once. This occurs for all cells on the board and keeps looping until a solution is found.

In order to stop the loop, the function *is_full* is called, which is shown on page 26 and starts on line 6. This checks whether the puzzle is full or not, and if it is, then the loop is over.

9	[4]	[5]	2	3	7	6	8	[1,4]
[1,6]	2	[5,6]	8	4	[9]	[1,5]	7	3
8	[3,4,6]	7	1	[6]	5	[4]	2	9
[6,7]	[6,7]	4	5	9	8	3	[1]	[1,2]
2	[3,7, 8,9]	[3,9]	[4,7]	[7]	1	[5,8,9]	[5]	6
5	1	[3,6,9]	[6]	[2,6]	[2,3]	[8,9]	4	7
4	[7]	1	3	[7,8]	6	2	9	5
[6]	5	[2,6]	9	1	[2,4]	7	3	8
3	[6,7,9]	8	[4,7]	5	[2,4]	[1,4]	[1,6]	[1,4]

9	4	5	2	3	7	6	8	[1]
[1]	2	[6]	8	4	9	[1,5]	7	3
8	[3]	7	1	6	5	4	2	9
[7]	[6]	4	5	9	8	3	1	[2]
2	[3,8,9]	[3,9]	[4]	7	1	[8,9]	5	6
5	1	[3,9]	6	[2]	[2,3]	[8,9]	4	7
4	7	1	3	[8]	6	2	9	5
6	5	[2]	9	1	[2,4]	7	3	8
3	[9]	8	[4,7]	5	[2,4]	[1]	[6]	[1,4]

9	4	5	2	3	7	6	8	1
1	2	6	8	4	9	[5]	7	3
8	3	7	1	6	5	4	2	9
7	6	4	5	9	8	3	1	2
2	[8]	[3,9]	4	7	1	[8,9]	5	6
5	1	[3,9]	6	2	[3]	[8,9]	4	7
4	7	1	3	8	6	2	9	5
6	5	2	9	1	[4]	7	3	8
3	9	8	[7]	5	[2,4]	1	6	[4]

9	4	5	2	3	7	6	8	1
1	2	6	8	4	9	5	7	3
8	3	7	1	6	5	4	2	9
7	6	4	5	9	8	3	1	2
2	8	[3,9]	4	7	1	[9]	5	6
5	1	[9]	6	2	3	[8,9]	4	7
4	7	1	3	8	6	2	9	5
6	5	2	9	1	4	7	3	8
3	9	8	7	5	[2]	1	6	4

9	4	5	2	3	7	6	8	1
1	2	6	8	4	9	5	7	3
8	3	7	1	6	5	4	2	9
7	6	4	5	9	8	3	1	2
2	8	[3]	4	7	1	9	5	6
5	1	9	6	2	3	[8]	4	7
4	7	1	3	8	6	2	9	5
6	5	2	9	1	4	7	3	8
3	9	8	7	5	2	1	6	4

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

Figure 8: Step-by-step process of how the algorithm works so far

3.2 Medium Sudoku Puzzles

The algorithm that has been developed thus far can be summarised as follows:
After initially marking-up the puzzle,

1. Search for entries with only one possible answer.
2. For entries with only one possibility, lock in this number.
3. Update the possibilities in the cells.
4. If unsolved, start at 1. again.

This algorithm so far is sufficient to solve 'easy' Sudoku puzzles, where at each step there are entries with only one possibility. However, the algorithm fails at more advanced Sudokus, as it will reach a point in which there are no entries with only one possibility. For example, the algorithm would fail for the uncompleted Sudoku puzzle in figure 9, as there is no entry with just one possible candidate. This is because at this point, the algorithm only checks the Latin square property (checking the row and column values) and the values in its $\sqrt{n} \times \sqrt{n}$ sub-grid to eliminate possible values. Hence, more subtle ways to eliminate possibilities are needed.

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

1	5	6	8	[3,7]	[3,7]	9	4	2
9	8	3	2	4	5	1	7	6
7	2	4	9	1	6	8	3	5
2	6	9	7	[3,5,8]	4	[3,5]	[1,8]	[1,3]
[4,8]	[4,7]	[1,7,8]	[1,5]	[3,5,8]	[1,2, 3,8]	[2,3, 4,5]	6	9
[4,8]	3	5	[1,6]	[6,8]	9	7	[1,2,8]	[1,4]
[3,4, 5,8]	1	[7,8]	[4,5,6]	2	[7,8]	[3,4,6]	9	[3,4,7]
6	9	[2,7,8]	3	[5,7,8]	[1,7,8]	[2,4]	[1,2]	[1,4,7]
[3,4]	[4,7]	[2,7]	[1,4,6]	9	[1,7]	[2,3, 4,6]	5	8

Figure 9: An example of where the algorithm will fail

3.2.1 Peer-Checking

Another way to find solutions is to look at the possibilities of the peers of a cell. Recall, we define the *peers* of a cell as the squares that share either the same row, the same column or same $\sqrt{n} \times \sqrt{n}$ sub-grid. To use this to fill in more squares, we can look at the possibilities of a row, column or square and count how many times a number comes up as a possibility. If a number only comes up as a possibility once in a row, column or square, then that number must be the solution in the cell where it occurred. For example, looking at the first column of the Sudoku puzzle in figure 9, the number 5 only comes up as a possibility once at (7,1). By the condition that in each column, digits 1 - 9 must occur once, we can conclude that 5 must be in that cell.

To write this in Python, a dictionary will be used for the data structure that keeps count of the numbers. In Python, a dictionary is an unordered set of *key:value* pairs. Here, the key will denote a number between 1 and n, and the value will represent the number of times that candidate occurs in the row, column or square. For an $n \times n$ Sudoku, there will be $3n$ peer-checkers (n rows, n columns and n $\sqrt{n} \times \sqrt{n}$ sub-grids). Each peer-checker initialises the dictionary:

```
{1:0, 2:0, 3:0, ..., n:0}
```

The function then loops through the possibilities in each cell of the row, column or square and will count how many times a number occurs.

For example, applying this to the first column of the Sudoku in figure 9 would return the dictionary:

```
{1:0, 2:0, 3:2, 4:4, 5:1, 6:0, 7:0, 8:3, 9:0}
```

Next, we follow the rule that if a key has value 1, then we put that candidate in the square where it occurred as a possibility. So in this case, 5 has a value 1, and so the function will lock in the answer where that occurred, in cell $(7, 1)$. After this, the function calls *update_possibilities* to remove 5 as a candidate in the unknown entries that share the same row, column or square. This technique can be applied using the functions *peer_checker_row*, *peer_checker_col* and *peer_checker_square* and we loop this function for each row, column and square. The functions *peer_checker_row*, *peer_checker_col* and *peer_checker_square* can be found in the appendix on pages 29 - 30.

3.2.2 The 'Naked-Pairs' technique

The following method is given by Davis (2011, p.35). A *naked pair* (or conjugate pair) is a pair of sets containing the same two numbers that are in the same row, column or $\sqrt{n} \times \sqrt{n}$ sub-grid. If a naked pair can be found, then we can remove them as possibilities from the rest of the row, column or $\sqrt{n} \times \sqrt{n}$ sub-grid, depending where the pair occurred.

In the example in figure 10, there is a naked pair $[3, 7]$ in the first column, which are highlighted by being in the grey cells. Hence 3 and 7 can be removed from the possibilities of the cells that are in the same column, namely $(2, 1)$ and $(3, 1)$ in this case. Also note that in this example there are other naked pairs that could be used to eliminate possibilities, such as the naked pair $[4, 7]$ in column 5 in order to remove the 7 in cell $(2, 5)$.

Similarly, if there is a naked pair within a row or box, then the candidates in the naked pair can be removed from the possibilities in the same row or box. The implementation of this technique in Python can be found on pages 30 - 31 in the appendix. This method works because the numbers in the naked pair must be in those two cells, and so it can be removed from the possibilities of the other cells that share the same row, column or $\sqrt{n} \times \sqrt{n}$ sub-grid.

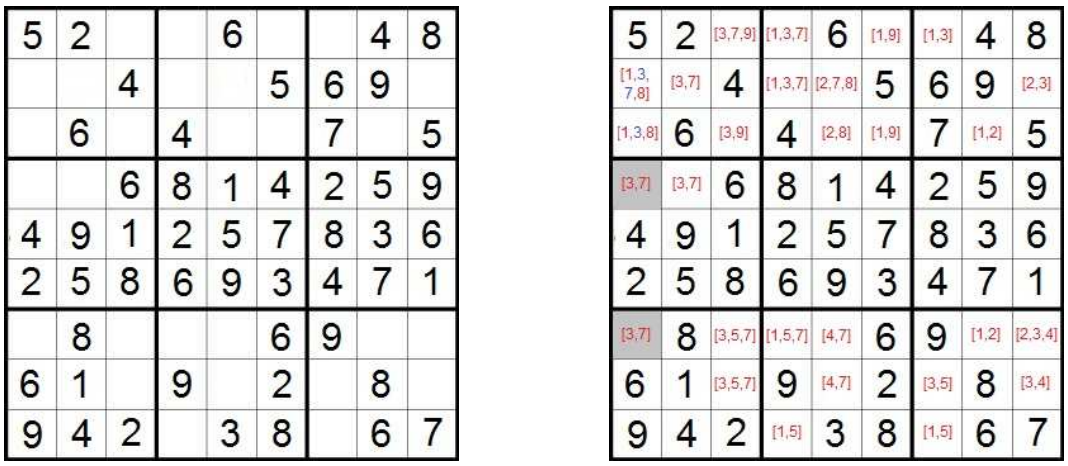


Figure 10: An example where the naked pairs technique can be used

3.2.3 Locked Candidates

If in a $n \times n$ Sudoku puzzle, a candidate can only appear in a row or column within a $\sqrt{n} \times \sqrt{n}$ sub-grid, then that candidate is said to be *locked*. For example, in figure 11, 7 is a locked candidate in the top left sub-grid (shown in the grey cells), as it can only occur in the top row inside its 3×3 sub-grid. As 7 has to be the top row and in the box, then it is possible to remove this as possibility from the other unknown entries that are also in the top row. Hence, 7 can be removed from cells $(1, 5)$ and $(1, 9)$.

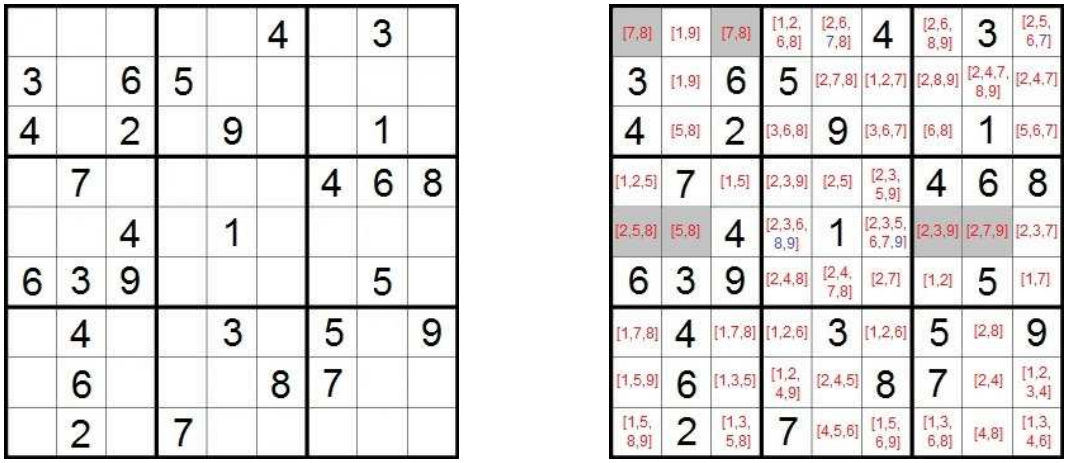


Figure 11: An example where the locked candidate technique can be used

Similarly, in the fifth row, 8 and 9 are locked candidates (shown in the grey cells), so they can be removed from the possibilities in cell $(5, 3)$ and $(5, 6)$.

This method works because if there is a locked candidate to be found in a row or column, then as it has to appear in that row or column and in the $\sqrt{n} \times \sqrt{n}$ sub-grid, it cannot appear in any of the other cells in that row or column.

The implementation of this method can be found on pages 31 - 33 in the appendix. To generalise this method to Sudoku puzzles of size n, the function first counts the number of occurrences of each candidate in an $\sqrt{n} \times \sqrt{n}$ sub-grid. If the number of occurrences for a candidate is between 2 and $\sqrt{n}$, then it is possible that the candidate could be *locked*. Next for each of these candidates, it checks if they only occur in the same row or column; this is done by calling the functions *same_col_check* and *same_row_check*, which are defined on page 31 in the appendix. If they only occur in the same row or column, then they are *locked* and hence the function will delete all other occurrences of the candidate from the row or column.

3.2.4 The Constraints Function

The *constraints* function is defined on page 33 in the appendix; it combines the techniques explained in this section. This loops these functions until the puzzle is either solved, or can no longer fix anymore answers. It checks this by using the *is_full* function defined on page 26 in the appendix and by seeing if a change was made after applying the techniques. The *constraints* function returns `True` if the puzzle is completed and `False` if no change was made and the puzzle was not solved. Note that the function initially tries to only use the technique explained in 3.1, and if it cannot lock any answers, then it will use the techniques in sections 3.2.1-3.2.3. This is because if a puzzle can be solved by only using *definite_answers* and *update_possibilities*, then the other functions will still be called and hence the algorithm will be less efficient. Thus, to improve the solution times, it tries to lock down answers repeatedly before using the more advanced techniques.

So far, the algorithm only consists of logical steps to lock answers down. In the next section, this will be combined with a backtracking algorithm to solve harder Sudoku puzzles.

4 An Algorithm to solve Sudoku puzzles using backtracking

In section 3, we developed an algorithm that solves a set of Sudoku puzzles using logical steps and some techniques. Many Sudoku puzzles can be solved in this way, however, this approach does not always guarantee a solution will be found. To tackle this problem, one could attempt to code more advanced techniques. Stuart (2015) gives a plethora of possible sophisticated techniques, such as the *X-Wing* and *Swordfish* techniques. However, a criticism of this is that this could take many more lines of code to implement. Moreover, we cannot be certain that by adding these techniques that the algorithm will be able to solve *every* Sudoku puzzle.

Instead of coding up more sophisticated techniques, a backtracking algorithm or a brute force search algorithm can be used to finish a puzzle. A brute force search would try all the possible combinations of the possible numbers until it finds a valid solution. By using the *count_brute_force* function in the code (defined on page 26 in the appendix), we are able to count up the number of possibilities that we could possibly try in order to brute force search for a solution. However, for complicated puzzles this number could be very large. For example, figure 12 shows the result after applying the *constraints* function to a Sudoku puzzle, with the numbers in blue being the numbers that have been locked in by the function. By multiplying the number of candidates that an unknown cell can have together (done by using the *count_brute_force* function), we can see there are $779,980,467,213,434,880,000,000$ number of possibilities we would need to go through to try and find the 1 unique solution of this puzzle. Hence, a backtracking algorithm is used instead to complete the Sudoku grid after initially imposing constraints.

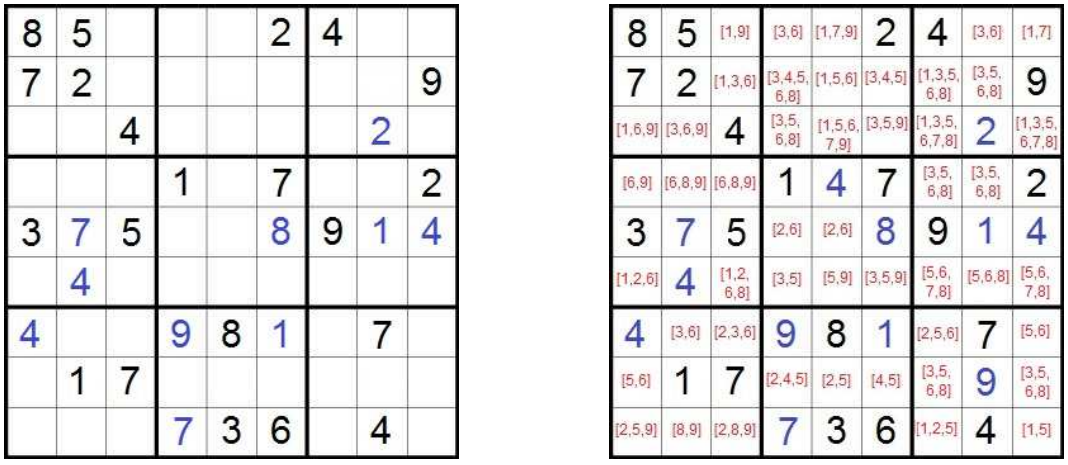


Figure 12: An example of a Sudoku puzzle that can't be solved by applying *constraints* alone

4.1 Backtracking Algorithm

A Dictionary of Computer Science (2016) defines *backtracking* to be a "property of an algorithm that implies some kind of tentative search for a goal, and the possibility that any search path may turn out to be a dead end; the algorithm then retreats back down the search path to try another path." In the context of using this to complete a Sudoku grid, the algorithm will search

for a solution by making a guess in a cell, but if the guess is wrong, then it will go back and try a different guess for that cell. The implementation of a backtracking algorithm is done by the *guess* function, which can be found on pages 33 - 34 in the appendix.

After making a guess in a cell by choosing a candidate from the list of possibilities it can be, the algorithm will apply the *constraints* function, defined on page 33 in the appendix, to see if it can solve the rest of the puzzle. However, if an error is found, then it will backtrack to guess a different possibility for that cell. Hence, this can be seen as following rules rather than just searching for an answer, as Hayes (2006, p.15) puts it "backtracking itself can be viewed as a logical operation; it supplies a proof by contradiction."

To code this in Python, a *stack* is used in order to keep track of the Sudoku grid before making guesses so we can retreat back if a wrong guess is made. The Dictionary of Computer Science (2016) states that a *stack* is a "linear list where all accesses, insertions, and removals are made at one end of the list, called the top. This implies access on a last in first out (LIFO) basis: the most recently inserted item on the list is the first to be removed." To make use of a stack, the algorithm makes a guess; say it is guessing x is the answer in cell (i, j), then the algorithm will remove x as a possible candidate in cell (i, j), and insert a copy of that Sudoku grid to the stack. If by choosing x to be in (i, j) leads to an error (i.e. by choosing this possibility, it reaches a point where a cell has no possible answers or a number occurs more than once in a row, column or square), then by contradiction, we can conclude that x is not in cell (i, j). Then, the algorithm will go back to the last item in the stack, which will be the grid before making the guess that x should be in cell (i, j) and with x no longer a possible candidate for that cell.

After making a guess and applying the *constraints* function, there are three possible outcomes:

1. The puzzle can be completed immediately.
2. The algorithm reaches a point where it can no longer lock any answers and needs to make another guess.
3. The algorithm reaches a contradiction, hence it goes back up the stack to guess a different number.

The algorithm now combines the *constraints* function and a backtracking algorithm so that it can be more efficient than brute force searching for a solution. Note that before a guess is made, a function *first_smallest_mark_up* is called, which is shown on page 26 in the appendix. The reason why this is useful is that if we chose to try a number in a cell with only 2 possibilities, then there is a probability of $\frac{1}{2}$ that the guess is correct. On the other hand, say if we chose to try to guess a number in a cell that has 6 possibilities, then we would expect to be correct $\frac{1}{6}$ of the time, as we have no idea in this case what number is better to choose. Hence by choosing to guess a cell with the least number of possible candidates, we maximise the probability of being correct.

To illustrate how the *guess* function works, if it was applied to the puzzle in figure 12, the *guess* function will look for lists of length 2 in the puzzle, as that is the smallest length of possibilities. It does this in the order defined in figure 6. The first guess that the algorithm makes is by choosing 1 in cell $(1, 3)$, as it will be the first entry that the algorithm finds with 2 possibilities. After choosing 1, it will then apply the *constraints* function to try and finish the puzzle. In this case, by choosing 1 and making more guesses after that, the algorithm reaches an error, so it fills in cells and reaches a point where a cell has no more possibilities. The result of choosing 1 in cell $(1, 3)$ is shown in figure 13. Then, as a contradiction is found, the algorithm goes back up the stack to the puzzle with only 9 as a possibility in that cell.

```
[[8, 5, 1, 6, 9, 2, 4, 3, 7],
[7, 2, 6, 4, 1, 3, 5, 8, 9],
[9, 3, 4, 8, 7, 3, 5, 2, 6],
[6, [8, 9], [8, 9], 1, 4, 7, 3, 5, 2],
[3, 7, 5, 2, 6, 8, 9, 1, 4],
[1, 4, 2, 3, 5, 9, 7, 6, 8],
[4, 6, 3, 9, 8, 1, 2, 7, 5],
[5, 1, 7, [], 2, 4, 8, 9, 3],
[2, [8, 9], [8, 9], 7, 3, 6, 1, 4, 1]]
```

Figure 13: An example mark up of a Sudoku grid with an error

Another error that could occur is that it chooses to guess a value that leads to two or more numbers appearing in a row, column or $\sqrt{n} \times \sqrt{n}$ sub-grid. This may occur if the algorithm keeps guessing for a solution and then two cells or more that share the same peer only have one possibility, which is the same number. This is not as a result of the *constraints* function, but because the choice of number that is being guessed is wrong. This error is also checked in the function *error_check* (defined on pages 26 - 27), where it counts the occurrence of a number in each row, column and $\sqrt{n} \times \sqrt{n}$ sub-grid; if a number occurs more than once, then an error is found.

4.2 The Solve Function

The *solve* function can be found on page 34 in the appendix. The purpose of this function is to combine the *constraints* function and the *guess* function. The function starts with calling *possibilities_puzzle* to initialise the puzzle to start solving it. Recall the *constraints* function returns `True` if the puzzle is completed. Hence by using *if/else* statements, the solution is printed and the puzzle is returned if the Sudoku is solved. However, if it is not completed, then it will need to start searching for a solution. It first creates the stack to keep track of the guesses, and *guess* will be called to complete the puzzle.

The next section looks at how this algorithm performs and how well it was implemented.

5 Results

5.1 Testing the Algorithm

The complete algorithm can be found in the appendix, in the program named *solver.py*. The *solve* function can be found on page 34, which combines the code explained in sections 3 and 4 to solve Sudoku puzzles. Sudoku puzzles can be found easily by an Internet search and we will apply the algorithm to see how it performs in solving these. The computer used to test the algorithm was the XPS 13 with a 2.30GHz Intel Core CPU, so it is possible to have significantly faster solution times with a more powerful computer.

As reported by Collins (2012), Finnish Mathematician Arto Inkala claimed to have created the worlds hardest Sudoku puzzle, which is shown in figure 14. The algorithm was able to solve this in 4.44 seconds. After initially trying to solve the puzzle using logic rules only and applying the *constraints* function, no numbers were locked in. Hence, a solution had to be found using the backtracking algorithm.

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

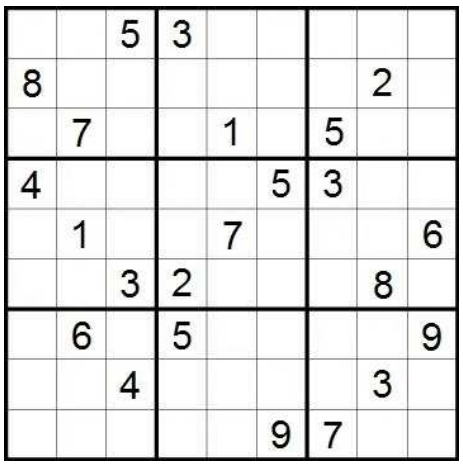
```
The solution:
[8, 1, 2, 7, 5, 3, 6, 4, 9]
[9, 4, 3, 6, 8, 2, 1, 7, 5]
[6, 7, 5, 4, 9, 1, 2, 8, 3]
[1, 5, 4, 2, 3, 7, 8, 9, 6]
[3, 6, 9, 8, 4, 5, 7, 2, 1]
[2, 8, 7, 1, 6, 9, 5, 3, 4]
[5, 2, 1, 9, 7, 4, 3, 6, 8]
[4, 3, 8, 5, 2, 6, 9, 1, 7]
[7, 9, 6, 3, 1, 8, 4, 5, 2]
Time taken:4.440655837307019
```

Figure 14: Sudoku puzzle created by Arto Inkala in 2012 and its solution

In addition, Inkala also created two other puzzles in 2006 and 2010, which he claimed to be the hardest puzzles he had created. Hutchinson (2010) reported that Arto Inkala created a puzzle in 2010, which he also described as "the worlds hardest Sudoku". The algorithm took 1.48 seconds to solve this. This puzzle and its solution is shown in figure 15.

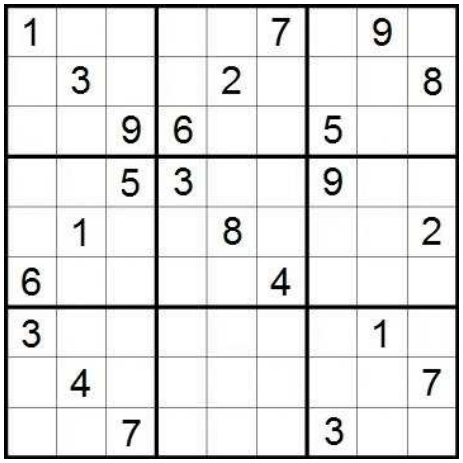
Inkala named the puzzle that he created in 2006 *AI Escargot*, as Stuart (2008) states. This puzzle and its solution is shown in figure 16. For this puzzle, the algorithm took 4.84 seconds to solve.

To test the algorithm further, Project Euler (2015) provides 50 9×9 Sudoku puzzles. The algorithm was able to solve all of these; the longest time to solve one of them was 0.39 seconds, whereas the shortest time to solve one of them was 0.015 seconds. The mean average of the time to find a solution to a puzzle was 0.088 seconds.



```
The solution:
[1, 4, 5, 3, 2, 7, 6, 9, 8]
[8, 3, 9, 6, 5, 4, 1, 2, 7]
[6, 7, 2, 9, 1, 8, 5, 4, 3]
[4, 9, 6, 1, 8, 5, 3, 7, 2]
[2, 1, 8, 4, 7, 3, 9, 5, 6]
[7, 5, 3, 2, 9, 6, 4, 8, 1]
[3, 6, 7, 5, 4, 2, 8, 1, 9]
[9, 8, 4, 7, 6, 1, 2, 3, 5]
[5, 2, 1, 8, 3, 9, 7, 6, 4]
Time taken:1.4836864139209274
```

Figure 15: Sudoku puzzle created by Arto Inkala in 2010 and its solution



```
The solution:
[1, 6, 2, 8, 5, 7, 4, 9, 3]
[5, 3, 4, 1, 2, 9, 6, 7, 8]
[7, 8, 9, 6, 4, 3, 5, 2, 1]
[4, 7, 5, 3, 1, 2, 9, 8, 6]
[9, 1, 3, 5, 8, 6, 7, 4, 2]
[6, 2, 8, 7, 9, 4, 1, 3, 5]
[3, 5, 6, 4, 7, 8, 2, 1, 9]
[2, 4, 1, 9, 3, 5, 8, 6, 7]
[8, 9, 7, 2, 6, 1, 3, 5, 4]
Time taken:4.836783499278081
```

Figure 16: Inkala's *AI Escargot* Sudoku puzzle and its solution

In the puzzles provided by Project Euler (2015), only 4 puzzles required the backtracking algorithm to be used, as the rest could be solved using logic techniques alone. Hence the solutions were found quicker than the puzzles created by Finnish Mathematician, Arto Inkala, where all required a guess to be made.

The algorithm was written so that it can solve Sudoku puzzles of arbitrary size. Maack (2017) provides 16×16 and 25×25 Sudoku puzzles on his website, by taking 10 random puzzles out of the 24 'medium' 16×16 Sudoku puzzles, the algorithm was able to solve all of them within 0.097-1.72 seconds. Further, it was able to solve the 25×25 puzzles that were available on the website.

5.2 Evaluation

Overall, the algorithm is accurate in solving arbitrary instances of Sudoku puzzles. However, the implementation and efficiency of the algorithm can be improved.

A weakness of the backtracking algorithm is that the data structure used to code a Sudoku puzzle is a list of lists. If the algorithm needed to search for a solution, at each step, a function *deepcopy* was used to add a copy of the puzzle to the stack. This function can be imported from the *copy* package. This is less efficient than making a copy of a string, say. Hence a possible way to improve the efficiency of the algorithm would be to use strings to represent Sudoku puzzles, rather than a list. This can also apply to other areas in the algorithm, for example searching through the elements in a list is slower than searching through characters in a string. Hence this is a possible way to reduce the time needed to find a solution.

Further, when the algorithm is making a guess, it finds the list with the least number of possibilities, then chooses the first number in the list to guess. However, there could be better ways to choose which number to guess. For example, the number could be chosen by random. This can be done by shuffling the list before choosing the first element, which can be achieved by importing the *random* package in Python and using `random.shuffle(puzzle[i][j])`. This can be added on line 394 in the *guess* function on page 33 in the appendix. (Note that this method makes it possible to generate valid Sudoku grids by calling the *solve* function on an empty Sudoku grid). However, using this did not always reduce the time required to find a solution and was not always reliable. For example, in one test using this method, the algorithm solved *AI Escargot* in 0.55 seconds, compared to 4.84 seconds before. However, in another test, it took 10.29 seconds to solve. In a small sample of 10 tests, the solution times (in seconds) to complete *AI Escargot* were:

```
0.55560, 0.82070, 4.70421, 8.37626, 5.03782,
4.57218, 8.5901, 8.58921, 10.29317, 4.5212
```

Hence, despite having the chance to reduce the time significantly, there is also a chance to increase the solution time if a wrong path is taken. Although this is a small sample, it shows that randomly guessing a value is not a reliable method to reduce the time needed to find a solution.

This can also be applied to how it chooses which cell to guess in. The cell order for which the algorithm finds the cell with the least number of possibilities is shown in figure 6. But there could be alternative ways to define the order in which the algorithm chooses a cell to guess in. However, improving these choices and ordering of guesses could take many more lines of code to implement, whereas optimising the code in other ways and writing cleaner code could increase the efficiency more.

6 Sudoku Generator

6.1 Generating 9×9 Sudoku grids

The following section is based on the work by Evans *et al.* (2011) and presents a method for generating 9×9 Sudoku puzzles.

1. Bona (2004, p.1) gives the following definition for a *permutation* of a set: "a linear ordering of the elements of the set $[n] = \{1, 2, 3, ..., n\}$ is called a *permutation*." The method starts off with the top row, and fills these cells in with a random permutation of $\{1, 2, ..., 9\}$.

   At this step, there are 9! choices.

2. Next, lets denote the numbers in the top row as $\{a_{11}, a_{12}, ..., a_{19}\}$ and also let $\{a_{11}\prime, a_{12}\prime, a_{13}\prime\}$ and $\{a_{11}\prime\prime, a_{12}\prime\prime, a_{13}\prime\prime\}$ be the two permutations of the set $\{a_{11}, a_{12}, a_{13}\}$, not necessarily different. And then similarly for $\{a_{14}, a_{15}, a_{16}\}$ and $\{a_{17}, a_{18}, a_{19}\}$ in the same way. Then the second and third row can be either:

   $$\{a_{14}\prime, a_{15}\prime, a_{16}\prime, a_{17}\prime, a_{18}\prime, a_{19}\prime, a_{11}\prime, a_{12}\prime, a_{13}\prime\}$$
   $$\{a_{17}\prime\prime, a_{18}\prime\prime, a_{19}\prime\prime, a_{11}\prime\prime, a_{12}\prime\prime, a_{13}\prime\prime, a_{14}\prime\prime, a_{15}\prime\prime, a_{16}\prime\prime\}$$

   or

   $$\{a_{17}\prime, a_{18}\prime, a_{19}\prime, a_{11}\prime, a_{12}\prime, a_{13}\prime, a_{14}\prime, a_{15}\prime, a_{16}\prime\}$$
   $$\{a_{14}\prime\prime, a_{15}\prime\prime, a_{16}\prime\prime, a_{17}\prime\prime, a_{18}\prime\prime, a_{19}\prime\prime, a_{11}\prime\prime, a_{12}\prime\prime, a_{13}\prime\prime\}$$

   Thus, these choices will make the first three rows of the Sudoku. At this step, there are $2(3!)^6$ number of choices.

3. At the next step, the method looks to fill in the column entries to complete the grid. Evans *et al.* (2011) use C_i to denote the first three elements for the column i. Hence, we have

   $$C_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix}, C_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}, ..., C_9 = \begin{bmatrix} a_{19} \\ a_{29} \\ a_{39} \end{bmatrix}.$$

   Then as $[C_1, C_2, C_3]$ forms the first 3×3 sub-grid, the first three columns of the Sudoku grid can be either

   $$\begin{bmatrix} C_1 & C_2 & C_3 \\ C_2 & C_3 & C_1 \\ C_3 & C_1 & C_2 \end{bmatrix} \textit{ or } \begin{bmatrix} C_1 & C_2 & C_3 \\ C_3 & C_1 & C_2 \\ C_2 & C_3 & C_1 \end{bmatrix}$$

   And similarly for the columns $[C_4, C_5, C_6]$ and $[C_7, C_8, C_9]$. Hence there are $2^3 = 8$ ways to permute these.

4. After completing the 9×9 grid, it is possible to get even more Sudoku grids by permuting the rows. Let R_4, R_5 and R_6 be the fourth, fifth and sixth row of the grid, and similarly

for R_7, R_8 and R_9. Then $MR = \{R_4, R_5, R_6\}$ forms the middle horizontal band and $BR = \{R_6, R_7, R_8\}$ forms the bottom horizontal band.

Evans *et al.* (2011) note that any permutation of the rows in MR and BR will still make the grid a valid Sudoku grid. Hence this step gives $(3!)^2$ more choices.

This method can generate $(9!) \times 2(3!)^6 \times 2^3 \times (3!)^2 = 9,751,984,865,280 \approx 10^{13}$ different 9×9 Sudoku grids.

6.2 Generating $n \times n$ Sudoku grids

In extension to the work done by Evans *et al.* (2011), this method can be generalised to create Sudoku grids of size n and which can be coded in Python. The function *create_sudoku* can be found in the appendix on page 35. The function takes an argument, m, which is equal to $\sqrt{n}$.

1. For the first step, the method starts off with filling the squares in the top row with a random permutation of $\{1, 2, ..., n\}$.

   At this step, there are $n!$ number of choices. The code to implement this in Python can be found on lines 40 - 44 in the *generator.py* program, which is on page 35 in the appendix.

2. At the next step, in the method given by Evans *et al.* (2011), the top row is split into chunks of size 3 and then fills in the next 2 rows. To generalise this to an $n \times n$ Sudoku puzzle, the top row is split into chunks of size $\sqrt{n}$. Now lets denote the numbers in the top row as $\{a_{11}, a_{12}, ..., a_{1n}\}$, then the top row is made by combining the elements of the set $\{\{a_{11}, ..., a_{1\sqrt{n}}\}, ..., \{a_{1(n-\sqrt{n}+1)}, ..., a_{1n}\}\}$. Then to fill in the next $(\sqrt{n} - 1)$ rows, by the rule that the digits 1 - n must appear only once in a $\sqrt{n} \times \sqrt{n}$ sub-grid, we must find the possible derangements of the set $\{\{a_{11}, ..., a_{1\sqrt{n}}\}, ..., \{a_{1(n-\sqrt{n}+1)}, ..., a_{1n}\}\}$.

   Wallis (2013, p.151) defines a *derangement* of an ordered set of objects as "a way of re-arranging the objects so that none appears in its original position." By finding possible derangements, the next row will not violate the constraints of a valid Sudoku puzzle. A function named *derangement* is used to find the different derangements of elements in a list and is defined on page 35 in the appendix.

   The permutations of a set can be found using the *permutations* function, which can be imported from the *itertools* package in Python. Then the *derangement* function works by taking in a list of permutations and then loops through a replicate of the list. The first item in the list is used to compare with other permutations. If it finds a permutation that has an object in the same position then it is deleted from the list. Then the first item in the list will be deleted, so only the derangements of the list are returned. For example, the possible permutations of the set $\{1, 2, 3\}$ are:

   ```
   [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
   ```

Then by calling the *derangement* function on this list, it returns the list:

```
[(2, 3, 1), (3, 1, 2)]
```

Now after finding the different derangements of $\{\{a_{11}, ..., a_{1\sqrt{n}}\}, ..., \{a_{1(n-\sqrt{n}+1)}, ..., a_{1n}\}\}$, the next step is choose an order for the next $(\sqrt{n}-1)$ rows. This is done by the *order* function, which is shown on page 35 in the appendix. This is called until the next $(\sqrt{n}-1)$ rows are determined. This function works by choosing a random element in the list for the first object by shuffling the list, and then calling the *derangement* function to delete the elements that can not be the next element in the list. This keeps on calling itself until the list of derangements is empty, meaning the next $(\sqrt{n}-1)$ rows are determined.

To put this together, this is implemented in Python by using the code on lines 46 - 61 on pages 35 - 36 in the appendix.

3. At the third step, the method fills in the column entries. Now let C_i denote the first $\sqrt{n}$ elements of column i. Then for an $n \times n$ Sudoku,

$$C_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ ... \\ a_{\sqrt{n}1} \end{bmatrix}, C_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ ... \\ a_{\sqrt{n}2} \end{bmatrix}, ..., C_n = \begin{bmatrix} a_{1n} \\ a_{2n} \\ ... \\ a_{\sqrt{n}n} \end{bmatrix}.$$

Then $[C_1, C_2, ..., C_{\sqrt{n}}]$ forms the first $\sqrt{n} \times \sqrt{n}$ sub-grid. To fill in the rest of the vertical band, we need to find the different derangements of this list, and find a new order in the same way as in step 2. This is done by the code on lines 63 - 94 on page 36 in the appendix. Comparing this to step 3 in the method by Evans *et al.* (2011), the derangements of $[C_1, C_2, C_3]$ are $[C_2, C_3, C_1]$ and $[C_3, C_1, C_2]$, hence the first three columns of the Sudoku grid can be either

$$\begin{bmatrix} C_1 & C_2 & C_3 \\ C_2 & C_3 & C_1 \\ C_3 & C_1 & C_2 \end{bmatrix} \text{ or } \begin{bmatrix} C_1 & C_2 & C_3 \\ C_3 & C_1 & C_2 \\ C_2 & C_3 & C_1 \end{bmatrix}.$$

Then we do the same for columns $[C_{\sqrt{n}+1}, ..., C_{2\sqrt{n}}]$,...,$[C_{n-\sqrt{n}+1}, ..., C_n]$.

4. Finally, after completing the $n \times n$ Sudoku grid, for each horizontal band, any permutation of the rows will still give a valid Sudoku grid. This can be done in Python using code on lines 96 - 105 on pages 36 - 37 in the appendix. In the method given by Evans *et al.* (2011), only the rows in the middle and bottom bands are permuted, however by permuting the top $\sqrt{n}$ rows as well, we are able to generate more Sudoku grids.

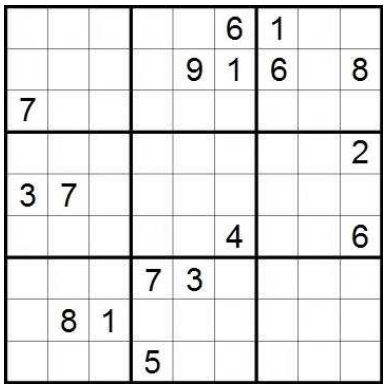
The function *create_sudoku* can be found on pages 35 - 37 in the appendix within the *generator.py* program. This combines the steps described above to generate valid Sudoku grids of size n, where n is a square number.

6.3 Generating puzzles using the Solver algorithm

The method given in sections 6.1 and 6.2 describes a methodical approach to generate a Sudoku grid. However, as briefly stated in section 5.2, it is possible to generate a valid Sudoku grid using *solve* function in the *solver.py* program. To do this, when the solver algorithm searches guesses a number in a cell, it must select a random number from the list rather than the first one in the list. This can be done by adding `random.shuffle(puzzle[i][j])` in the *guess* function on line 394, which can be found on page 33 in the appendix. By adding this line, the list is shuffled before selecting the first element. Hence by trying to solve an empty grid, it will return a valid Sudoku grid.

6.4 Constructing puzzles and difficulty of puzzles

Davis (2011, p.57) states that the difficulty of a Sudoku puzzle has little to do with the number of clues given initially and that difficulty ratings are given to indicate how hard it would be for a human to solve. A first thought to create a harder puzzle is to give less clues. However, having less clues given does not always mean the puzzle is harder. For example, the solver algorithm was able to solve the puzzle in figure 17 with the logic rules explained in section 3 alone in 0.14 seconds. However, the puzzle created by Finnish mathematician Arto Inkaka shown in figure 14 has 21 clues, but required a guess and took 4.44 seconds to solve.



```
The solution:
[9, 2, 8, 4, 7, 6, 1, 5, 3]
[4, 5, 3, 2, 9, 1, 6, 7, 8]
[7, 1, 6, 8, 5, 3, 2, 4, 9]
[1, 6, 4, 3, 8, 7, 5, 9, 2]
[3, 7, 2, 9, 6, 5, 4, 8, 1]
[8, 9, 5, 1, 2, 4, 7, 3, 6]
[6, 4, 9, 7, 3, 2, 8, 1, 5]
[5, 8, 1, 6, 4, 9, 3, 2, 7]
[2, 3, 7, 5, 1, 8, 9, 6, 4]
Time taken:0.1432105444603354
```

Figure 17: A 17 clue puzzle and its solution

To generate a puzzle from a grid, the following method described by Davis (2011, p.58) can be used:

1. Remove some numbers from the grid randomly.
2. Try to solve the resulting puzzle.
   (a) If the puzzle has a unique solution (i.e. the solved solution is the same as the original), then go back to 1.
   (b) If the puzzle has a unique solution and the puzzle has the desired numbers of unknowns, end.

(c) If the puzzle does not have a unique solution, refill the cells that had numbers removed and go back to 1.

This method starts to remove numbers from the grid a few at a time. This is because if all the numbers were removed at once, then it could take a long time until a puzzle is found that gives a unique solution. Hence, to reduce the time needed to find a unique puzzle, the number of values removed will be done incrementally. If the solved version of the puzzle is different than the solution then it will go back and choose different values to remove from the puzzle.

7 Conclusion

In this report, I have coded a pair of algorithms: one that can solve Sudoku puzzles of arbitrary size and one that can generate Sudoku grids. The algorithm to solve Sudoku puzzles was a combination of applying logical rules to find solutions to cells and a backtracking algorithm if it was needed. Although the efficiency of the algorithm to solve Sudoku puzzles can be improved, it is accurate in completing this task. With more time permitted, it would have been possible to optimise this and to write the code in a cleaner way. Moreover, to extend this project, it is possible to also code a function to generate puzzles, which would take a Sudoku grid and remove cells to return a puzzle. It would have been easy to randomly take entries out of the grid, but to ensure that there is a unique solution, entries must be taken out with care. A method to generate Sudoku puzzles was briefly explained in section 6.4.

References

[1] *The Concise Oxford Dictionary of Mathematics*. [Online]. 5th ed. 2005. s.v. Latin square. [Accessed 14 November 2016]. Available from:
`http://www.oxfordreference.com/view/10.1093/acref/9780199235940.001.0001/acref-9780199235940-e-1622?rskey=DWBex3&result=1651`

[2] *A Dictionary of Computer Science*. [Online]. 7th ed. 2016. s.v. Backtracking. [Accessed 5 February 2017]. Available from:
`http://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-304?rskey=RDYvfX&result=366`

[3] *A Dictionary of Computer Science*. [Online]. 7th ed. 2016. s.v. Stack. [Accessed 5 February 2017]. Available from:
`http://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-5006?rskey=Rk8Vtu&result=5481`

[4] Anon. 2015. *Project Euler*. [Online]. [Accessed 1 March]. Available from:
`https://projecteuler.net/problem=96`

[5] Bona, M. 2012. *Combinatorics of Permutations*. 2nd ed. Florida: Taylor & Francis Group.

[6] Collins, N. 2012. World's hardest sudoku: can you crack it?. *The Telegraph*. [Online]. 28 June. [Accessed 1 March]. Available from:
`http://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html`

[7] Crook, J.F. 2009. A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles. *Notices of the American Mathematical Society*. **56**(4), pp.460-468.

[8] Davis, T. 2011. The Mathematics of Sudoku. *Expeditions in Mathematics*. pp.31-59.

[9] Evans, R., Lindner, B. and Shi, Y. 2011. Generating Sudoku puzzles and its applications in teaching mathematics. *International Journal of Mathematical Education in Science and Technology*. **42**(5), pp.697-704.

[10] Felgenhauer, B. and Jarvis, F. 2006. Mathematics of Sudoku I. *Mathematical Spectrum*. [Online]. **39**, pp.15-22. [Accessed 16 November 2016]. Available from:
`http://afjarvis.staff.shef.ac.uk/maths/felgenhauer_jarvis_spec1.pdf`

[11] Greenspan, G. and Lee, R. 2005. *Web Sudoku*. [Online]. [Accessed 1 March 2017]. Available from:
`http://www.websudoku.com`

[12] Hayes, B. 2006. Computer Science: Unwed Numbers. *American Scientist*. **94**(1), pp.12-15.

[13] Herzberg, A.M. and Murty, M.R. 2007. Sudoku Squares and Chromatic Polynomials. *Notices of the American Mathematical Society*. **54**(6), pp.708-717.

[14] Hutchinson, P. 2010. The world's hardest sudoku?. *The Guardian*. [Online]. 22 August. [Accessed 1 March]. Available from:

`https://www.theguardian.com/media/2010/aug/22/worlds-hardest-sudoku`

[15] McGuire, G., Tugemann, B. and Gilles Civario. 2014. There Is No 16- Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. *Experimental Mathematics*. **23**(2), pp.190-217.

[16] Royle, G. 2017. *The University of Western Australia*. [Online]. [Accessed 12 February 2017]. Available from:

`http://staffhome.ecm.uwa.edu.au/~00013890/sudokumin.php`

[17] Russell, E. and Jarvis, F. 2006. Mathematics of Sudoku II. *Mathematical Spectrum*. [Online]. **39**, pp.54-58. [Accessed 16 November 2016]. Available from:

`http://afjarvis.staff.shef.ac.uk/maths/russell_jarvis_spec2.pdf`

[18] Stuart, A. 2015. *SudokuWiki*. [Online]. [Accessed 22 February 2017]. Available from:

`http://www.sudokuwiki.org/`

[19] Wallis, W.D. 2013. *A Beginner's Guide to Discrete Mathematics*. New York: Springer Science+Business Media.

A Sudoku Solver: solver.py

```
from math import sqrt
from copy import deepcopy
from functools import reduce
import random

def is_full(puzzle):
  #checks if the grid is full yet or not; returns True if puzzle is full
  n=len(puzzle)
  for i in range(n):
    for j in range(n):
      if (puzzle[i][j]==0) or (type(puzzle[i][j])==list):
        return False

  return True

def count_brute_force(puzzle):
  #counts the number of solutions we'd need to try if we went brute force route
  n = len(puzzle)
  mark_ups = []
  for i in range(n):
    for j in range(n):
      if (type(puzzle[i][j])==list):
        mark_ups.append(len(puzzle[i][j]))

  solutions = reduce(lambda x, y: x*y, mark_ups)

  return solutions

def first_smallest_mark_up(puzzle):
  #finds the first position with the least number of possibilities
  n=len(puzzle)
  smallest_len = n
  for i in range(n):
    for j in range(n):
      if (type(puzzle[i][j])==list) and (len(puzzle[i][j])<smallest_len):
        smallest_len=len(puzzle[i][j])

  (i,j)=(0,0)
  while not(type(puzzle[i][j])==list and len(puzzle[i][j])==smallest_len):
    while not(type(puzzle[i][j])==list and len(puzzle[i][j])==smallest_len):
      if j<(n-1):
        j+=1
      else:
        i+=1
        j=0

  return (i,j)

def error_check(puzzle):
```

```
  #checks for any errors in the puzzle; returns True if an error is found
  n=len(puzzle)

  #checking for any lists of length zero
  for i in range(n):
    for j in range(n):
      if (type(puzzle[i][j])==list) and (len(puzzle[i][j])==0):
        return True

  #checking if any number occurs more than once within a peer
  for i in range(n):
    for j in range(n):
      #checks the rows
      check = dict([(i, 0) for i in range(1, n+1)])
      for y in range(n):
        if (type(puzzle[i][y])==int) and puzzle[i][y]!=0:
          check[puzzle[i][y]]=check[puzzle[i][y]]+1
      for key in list(check.keys()):
        if (check[key]>1):
          return True
      #checks the columns
      check = dict([(i, 0) for i in range(1, n+1)])
      for x in range(n):
        if (type(puzzle[x][j])==int) and puzzle[x][j]!=0:
          check[puzzle[x][j]]=check[puzzle[x][j]]+1
      for key in list(check.keys()):
        if (check[key]>1):
          return True
      #checks the sqrt(n)xsqrt(n) sub-grids
      check = dict([(i, 0) for i in range(1, n+1)])
      x=int(int(i/sqrt(n)) * sqrt(n))
      y=int(int(j/sqrt(n)) * sqrt(n))
      for u in range(x, int(x+sqrt(n))):
        for v in range(y, int(y+sqrt(n))):
          if (type(puzzle[u][v])==int) and puzzle[u][v]!=0:
            check[puzzle[u][v]]=check[puzzle[u][v]]+1
      for key in list(check.keys()):
        if (check[key]>1):
          return True

  return False

def possibilities_puzzle(puzzle):
  #replaces the unknown values with a set of possible solutions
  n=len(puzzle)
  for i in range(n):
    for j in range(n):
      if (puzzle[i][j]==0) or (type(puzzle[i][j])==list):
        puzzle[i][j] = list(range(1,n+1))

        #checking the column values
```

```
        for x in range(n):
          if (puzzle[x][j] in puzzle[i][j]):
            puzzle[i][j].remove(puzzle[x][j])
        #checking the row values
        for y in range(n):
          if (puzzle[i][y] in puzzle[i][j]):
            puzzle[i][j].remove(puzzle[i][y])
        #checking the sqrt(n)xsqrt(n) square values
        x=int(int(i/sqrt(n)) * sqrt(n))
        y=int(int(j/sqrt(n)) * sqrt(n))
        for u in range(x, int(x+sqrt(n))):
          for v in range(y, int(y+sqrt(n))):
            if (puzzle[u][v] in puzzle[i][j]):
              puzzle[i][j].remove(puzzle[u][v])

  return puzzle

def definite_answers(puzzle):
  #takes a puzzle with unknown entries as list of possibilities
  #locks in any cells that are lists of length 1
  n=len(puzzle)
  for i in range(n):
    for j in range(n):
      if (type(puzzle[i][j])==list) and (len(puzzle[i][j])==1):
        puzzle[i][j]=puzzle[i][j][0]

  return puzzle

def update_possibilities(puzzle):
  #takes a puzzle with unknown entries as list of possibilities
  #updates the unknown entries to remove any possibilities that may have been
  #locked in by a step
  n=len(puzzle)

  for i in range(n):
    for j in range(n):
      if type(puzzle[i][j])==int:
        #checking the column values
        for x in range(n):
          if type(puzzle[x][j])==list:
            puzzle[x][j]=[item for item in puzzle[x][j] if item!=puzzle[i][j]]
        #checking the row values
        for y in range(n):
          if type(puzzle[i][y])==list:
            puzzle[i][y]=[item for item in puzzle[i][y] if item!=puzzle[i][j]]
        #checking the sqrt(n) x sqrt(n) square values
        x=int(int(i/sqrt(n)) * sqrt(n))
        y=int(int(j/sqrt(n)) * sqrt(n))
        for u in range(x, int(x+sqrt(n))):
          for v in range(y, int(y+sqrt(n))):
            if type(puzzle[u][v])==list:
```

```
152                 puzzle[u][v]=[item for item in puzzle[u][v] if item!=puzzle[i][j]]
153
154   return puzzle
155
156 def peer_checker_col(puzzle, j):
157   #implements the 'peer-checker' technique for a column
158   #takes a puzzle with unknown entries as the set of possibilities
159   #checks how many times a number occurs as a possibility in the column it's in
160   n=len(puzzle)
161   peer_column = dict([(i, 0) for i in range(1, n+1)])
162   definite_answers = []
163
164   #looping through the column to count how many times it occurs
165   for x in range(n):
166     if (type(puzzle[x][j])==list):
167       for item in puzzle[x][j]:
168         peer_column[item]=(peer_column[item]+1)
169   #checking if any dictionary values are 1
170   for key in list(peer_column.keys()):
171     if (peer_column[key]==1):
172       definite_answers.append(key)
173   #values in definite_answers only occur in the column once, so can be locked in
174   for x in range(n):
175     if (type(puzzle[x][j])==list):
176       for item in puzzle[x][j]:
177         for answer in definite_answers:
178           if item==answer:
179             puzzle[x][j]=answer
180
181   update_possibilities(puzzle)
182   return puzzle
183
184 def peer_checker_row(puzzle, i):
185   #implements the 'peer-checker' technique for a row
186   #takes a puzzle with unknown entries as the set of possibilities
187   #checks how many times a number occurs as a possibility in the row it's in
188   n=len(puzzle)
189   peer_row = dict([(i, 0) for i in range(1, n+1)])
190   definite_answers = []
191
192   #looping through the row to count how many times it occurs
193   for y in range(n):
194     if (type(puzzle[i][y])==list):
195       for item in puzzle[i][y]:
196         peer_row[item]=(peer_row[item]+1)
197   #checking if any dictionary values are 1
198   for key in list(peer_row.keys()):
199     if (peer_row[key]==1):
200       definite_answers.append(key)
201   #values in definite_answers only occur in the row once, so can be locked in
202   for y in range(n):
```

```
203     if (type(puzzle[i][y])==list):
204       for item in puzzle[i][y]:
205         for answer in definite_answers:
206           if item==answer:
207             puzzle[i][y]=answer
208 
209   update_possibilities(puzzle)
210   return puzzle
211 
212 def peer_checker_square(puzzle, i, j):
213   #implements the 'peer-checker' technique for a sub-grid
214   #takes a puzzle with unknown entries as the set of possibilities
215   #checks how any times a number occurs as a possibility inthe square it's in
216   n=len(puzzle)
217   peer_square = dict([(i, 0) for i in range(1, n+1)])
218   definite_answers = []
219   x=int(int(i/sqrt(n)) * sqrt(n))
220   y=int(int(j/sqrt(n)) * sqrt(n))
221 
222   #loops through the possible values of each cell and counts them
223   for u in range(x, int(x+sqrt(n))):
224     for v in range(y, int(y+sqrt(n))):
225       if (type(puzzle[u][v])==list):
226         for item in puzzle[u][v]:
227           peer_square[item]=(peer_square[item]+1)
228   #checking if any dictionary values are 1
229   for key in list(peer_square.keys()):
230     if (peer_square[key]==1):
231       definite_answers.append(key)
232   #values in definite_answers only occur in the sub-grid once, so can be locked
      in
233   for u in range(x, int(x+sqrt(n))):
234     for v in range(y, int(y+sqrt(n))):
235       if (type(puzzle[u][v])==list):
236         for item in puzzle[u][v]:
237           for answer in definite_answers:
238             if item==answer:
239               puzzle[u][v]=answer
240 
241   update_possibilities(puzzle)
242   return puzzle
243 
244 def naked_pairs(puzzle):
245   #implements the 'naked pairs' technique
246   #takes a puzzle with unknown entries as the set of possibilities
247   #looks for lists of length 2 that are in the share the same peer
248   n=len(puzzle)
249 
250   for i in range(n):
251     for j in range(n):
252       if (type(puzzle[i][j])==list) and len(puzzle[i][j])==2:
```

```
253         #checks the columns
254         for x in range(n):
255           if x!=i and puzzle[i][j]==puzzle[x][j]:
256             #execute if found a naked pair in its column
257             for w in range(n):
258               if w!=i and w!=x and type(puzzle[w][j])==list:
259                 puzzle[w][j]=[item for item in puzzle[w][j] if
260                 item not in puzzle[i][j]]
261         #checks the rows
262         for y in range(n):
263           if y!=j and puzzle[i][j]==puzzle[i][y]:
264             #execute if found a naked pair in its row
265             for w in range(n):
266               if w!=j and w!=y and type(puzzle[i][w])==list:
267                 puzzle[i][w]=[item for item in puzzle[i][w] if
268                 item not in puzzle[i][j]]
269         #checks the sqrt(n) x sqrt(n) boxes
270         x=int(int(i/sqrt(n)) * sqrt(n))
271         y=int(int(j/sqrt(n)) * sqrt(n))
272         for u in range(x, int(x+sqrt(n))):
273           for v in range(y, int(y+sqrt(n))):
274             if (i,j)!=(u,v) and puzzle[i][j]==puzzle[u][v]:
275               for s in range(x, int(x+sqrt(n))):
276                 for t in range(y, int(y+sqrt(n))):
277                   if puzzle[s][t]!=puzzle[i][j] and type(puzzle[s][t])==list:
278                     puzzle[s][t]=[item for item in puzzle[s][t]
279                     if item not in puzzle[i][j]]
280
281   definite_answers(puzzle)
282   update_possibilities(puzzle)
283   return puzzle
284
285 def same_row_check(coordinates):
286   #if all i coordinates are the same, then they're in same row; return True
287   n=len(coordinates)
288   for coord in range(1,n):
289     if coordinates[coord][0]!=coordinates[0][0]:
290       return False
291
292   return True
293
294 def same_col_check(coordinates):
295   #if all j coordinates are the same, then they're in same column; return True
296   n=len(coordinates)
297   for coord in range(1,n):
298     if coordinates[coord][1]!=coordinates[0][1]:
299       return False
300
301   return True
302
303 def locked_candidates(puzzle, i, j):
```

```
#implements the 'locked candidates' technique
#takes a puzzle with unknown entries as the set of possibilities
#looks for candidates that only occur in a row or column in its sub-grid
n=len(puzzle)
candidates = dict([(i, 0) for i in range(1, n+1)])
x=int(int(i/sqrt(n)) * sqrt(n))
y=int(int(j/sqrt(n)) * sqrt(n))

#loops through the possible values of each cell and counts them
for u in range(x, int(x+sqrt(n))):
  for v in range(y, int(y+sqrt(n))):
    if (type(puzzle[u][v])==list):
      for item in puzzle[u][v]:
        candidates[item]=(candidates[item]+1)

#checks if any occur between 2 and sqrt(n) times, then its possible they're
  locked
for key in list(candidates.keys()):
  if candidates[key]<2 or candidates[key]>sqrt(n):
    candidates.pop(key)
  else:
    candidates[key]=[]

#checking if the remaining candidates are locked; checks if appear in same row
  /column
#seeing where they occur in the grid
for u in range(x, int(x+sqrt(n))):
  for v in range(y, int(y+sqrt(n))):
    if (type(puzzle[u][v])==list):
      for item in puzzle[u][v]:
        for key in list(candidates.keys()):
          if item==key:
            candidates[key].append((u,v))

#checking if they occur all in the same column or row
for key in list(candidates.keys()):
  #checking columns
  if same_col_check(candidates[key]):
    for i in [i for i in range(n) if i not in range(x, int(x+sqrt(n)))]:
      j=candidates[key][0][1]
      if type(puzzle[i][j])==list and (key in puzzle[i][j]):
        puzzle[i][j].remove(key)
  #checking rows
  if same_row_check(candidates[key]):
    for j in [j for j in range(n) if j not in range(y, int(y+sqrt(n)))]:
      i=candidates[key][0][0]
      if type(puzzle[i][j])==list and (key in puzzle[i][j]):
        puzzle[i][j].remove(key)

definite_answers(puzzle)
update_possibilities(puzzle)
```

```
353   return puzzle
354
355 def constraints(puzzle):
356   #applys the logic rules and techniques to the Sudoku puzzle
357   n=len(puzzle)
358
359   change_checker = []
360   while change_checker!=puzzle:
361     change_checker=deepcopy(puzzle)
362     definite_answers(puzzle)
363     update_possibilities(puzzle)
364
365   if not(is_full(puzzle)):
366     #more advanced techniques are needed
367     naked_pairs(puzzle)
368     for i in range(0,n,int(sqrt(n))):
369       for j in range(0,n,int(sqrt(n))):
370         locked_candidates(puzzle,i,j)
371     for i in range(n):
372       peer_checker_row(puzzle,i)
373     for j in range(n):
374       peer_checker_col(puzzle,j)
375     for i in range(0,n,int(sqrt(n))):
376       for j in range(0,n,int(sqrt(n))):
377         peer_checker_square(puzzle,i,j)
378
379   #calling the function until the sudoku puzzle is solved or makes no changes
380   if is_full(puzzle):
381     return True
382   elif puzzle==change_checker:
383     return False
384   else:
385     constraints(puzzle)
386
387 def guess(stack, puzzle):
388   #implements a backtracking algorithm to complete a Sudoku grid
389   n=len(puzzle)
390
391   while not(is_full(puzzle)) or error_check(puzzle):
392     smallest_index=first_smallest_mark_up(puzzle)
393     (i,j)=smallest_index
394     #random.shuffle(puzzle[i][j])
395
396     if len(puzzle[i][j])!=0:
397       #if there's still possibilities, it will make a guess
398       item=deepcopy(puzzle[i][j][0])
399       puzzle[i][j].remove(puzzle[i][j][0])
400       stack.append(deepcopy(puzzle))
401       puzzle[i][j]=item
402       constraints(puzzle)
403
```

```
404     if error_check(puzzle):
405       #if there's an error, go back to last item in stack
406       if len(stack)!=1:
407         puzzle=deepcopy(stack[-1])
408         stack.pop()
409       elif len(stack)==0:
410         return False
411 
412   return solve(puzzle)
413 
414 def solve(puzzle):
415   n=len(puzzle)
416   possibilities_puzzle(puzzle)
417   constraints(puzzle)
418 
419   if constraints(puzzle):
420     print("The solution:")
421     for i in range(n):
422       print(puzzle[i])
423   elif not(constraints(puzzle)) and error_check(puzzle):
424     #didn't solve it and has an error
425     print("There has been an error in solving this puzzle")
426   elif not(constraints(puzzle)):
427     #didn't solve it but needs to search for a solution now
428     stack=[deepcopy(puzzle)]
429     return guess(stack,puzzle)
430 
431   return puzzle
```

B Sudoku Generator: generator.py

```
import random
from copy import deepcopy
from itertools import permutations

def derangement(list_of_permutations):
  #takes a list of possible permutations and returns a list of derangements

  replicate=deepcopy(list_of_permutations)
  #loops through the replicate list
  for i in range(1,len(replicate)):
    for item in range(len(replicate[i])):
      #if there's object in the same position, then remove from list
      if replicate[i][item]==replicate[0][item] and (replicate[i] in
    list_of_permutations):
        list_of_permutations.remove(replicate[i])

  if len(replicate)>0:
    list_of_permutations.remove(replicate[0])

  return list_of_permutations

def order(derangements,m):
  #takes a set of possible derangements and gives an order for the next (m-1)
    choices

  #first shuffling the derangements to get a random one to choose first
  random.shuffle(derangements)
  #chooses the next row
  order=[derangements[0]]
  #loops through the derangements to keep deleting the ones that are not
    possible
  for i in range(m):
    derangement(derangements)
    if len(derangements)>0:
      order.append(derangements[0])

  return(order)

def create_sudoku(m=3):
  n=m**2
  puzzle = [[0 for i in range(n)] for j in range(n)]

  ##### Step 1: Getting a permutation of n numbers (n! possibilities) #####

  numbers = list(range(1, n+1))
  random.shuffle(numbers)
  puzzle[0]=numbers

  ##### Step 2: Getting the first m rows #####
```

```
47
48  #splitting the list into lists of length m
49  chunks = [numbers[x:x+m] for x in range(0,n,m)]
50
51  #gets a possible derangements of the next (m-1) rows and gets a new order
52  derangements = derangement(list(permutations(chunks)))
53  random.shuffle(derangements)
54  new_order = order(derangements,m)
55
56  #expanding the chunks into just one list and making them the next (m-1) rows
57  for i in range(1,m):
58    for item in new_order:
59      for chunk in item:
60        random.shuffle(chunk)
61    puzzle[i]=[item for sublist in new_order[i-1] for item in sublist]
62
63  ##### Step 3: Getting the columns #####
64
65  #getting a list of lists which contain the columns of the first m rows
66  columns = [[] for i in range(n)]
67  for j in range(n):
68    for i in range(m):
69      columns[j].append(puzzle[i][j])
70
71  column_chunks = [columns[x:x+m] for x in range(0,n,m)]
72
73  #getting possible derangements for the columns and get a new order
74  column_derangements = []
75  for i in range(len(column_chunks)):
76    column_derangements.append(derangement(list(permutations(column_chunks[i])))
     )
77  new_order_col = []
78  for i in range(len(column_derangements)):
79    new_order_col.append(order(column_derangements[i], m))
80
81  #col_band is the columns for each band
82  col_band=dict((band,[]) for band in range(1,m))
83  for column in range(m):
84    for band in range(1,m):
85      col_band[band].append(new_order_col[column][band-1])
86
87  for band in range(1,m):
88    col_band[band]=[item for sublist in col_band[band] for item in sublist]
89
90  #now putting these into the right squares
91  for band in range(1,m):
92    for i in range((band*m), (band*m)+m):
93      for j in range(n):
94        puzzle[i][j]=col_band[band][j][i%m]
95
96  ##### Step 4: Permute the rows for the horizontal bands #####
```

```
97
98  #get possible permutations of each band
99  new_permutation=[]
100 for i in range(0,m):
101   possible_perms=list(permutations(poss for poss in range((i*m),(i*m)+m)))
102   permutation_choice=possible_perms[random.randint(0,len(possible_perms)-1)]
103   for row_choice in permutation_choice:
104     new_permutation.append(row_choice)
105 puzzle=[puzzle[i] for i in new_permutation]
106
107 return puzzle
```