CS21120 Programming Assignment: Sudoku Solver

Matthew Hibbin
mjh32@aber.ac.uk
17/11/20

Contents

Task 1: The Grid Class

This task was the most challenging of the assignment as it required a creative approach for the isValid() method. Efficiency was a top design priority, which was achieved by setting out the two following rules. The first rule was, if a duplicate value is found, the program should return false immediately instead of checking after a row, column or sub-grid has been traversed through. The second rule ensured there can be no more than two for loops nested within each other.

The isValid() method makes use of Hash Sets in order to check for duplicate values in rows, columns and sub-grids as Hash Sets do not allow duplicate values. For example, when a row is traversed, each value is attempted to be added to the Hash Set. If it is successful then it will carry on until the end of the row is reached and will clear the Hash Set for the next row. If it is not successful, then the method will return false. A continuously decreasing negative number was also added to the set if an empty cell was detected.

It was not necessary to use separate nested for loops to traverse the rows and columns. Instead, both could be traversed at the same time by making smart use of the x and y value calculated by the two for loops. The rows are traversed normally, whilst the columns will be traversed by swapping the y and x values when indexing the grid. As for the sub-grids, a separate pair of nested for loops are used in order to reuse the Hash Sets as too many Hash Sets would increase the runtime. In these for loops, the two previous Hash Sets are used along with a third one to iterate through three sub-grids simultaneously. The upper three sub-grids of the Sudoku grid are checked first, then the central and lower ones. The Hash-Sets are cleared each time the program traverses to the next set of sub-grids.

Task 2: The Solver Class

This task was easier than the last as implementing the solving algorithm was a very smooth process. Instead, the solve() method revealed problems with the isValid() method from the Grid class that weren't previously seen. The main problem was that the original design for the isValid() method was to also return false when it detected an empty cell in the grid. This meant the solve() method would always return false as the isValid() method would return false for every digit tried in the first empty cell, if the Sudoku Grid had more than one empty cell. When this was patched, the solve() method worked as expected and, along with the isValid() method, passed all the automated tests provided in tests package.

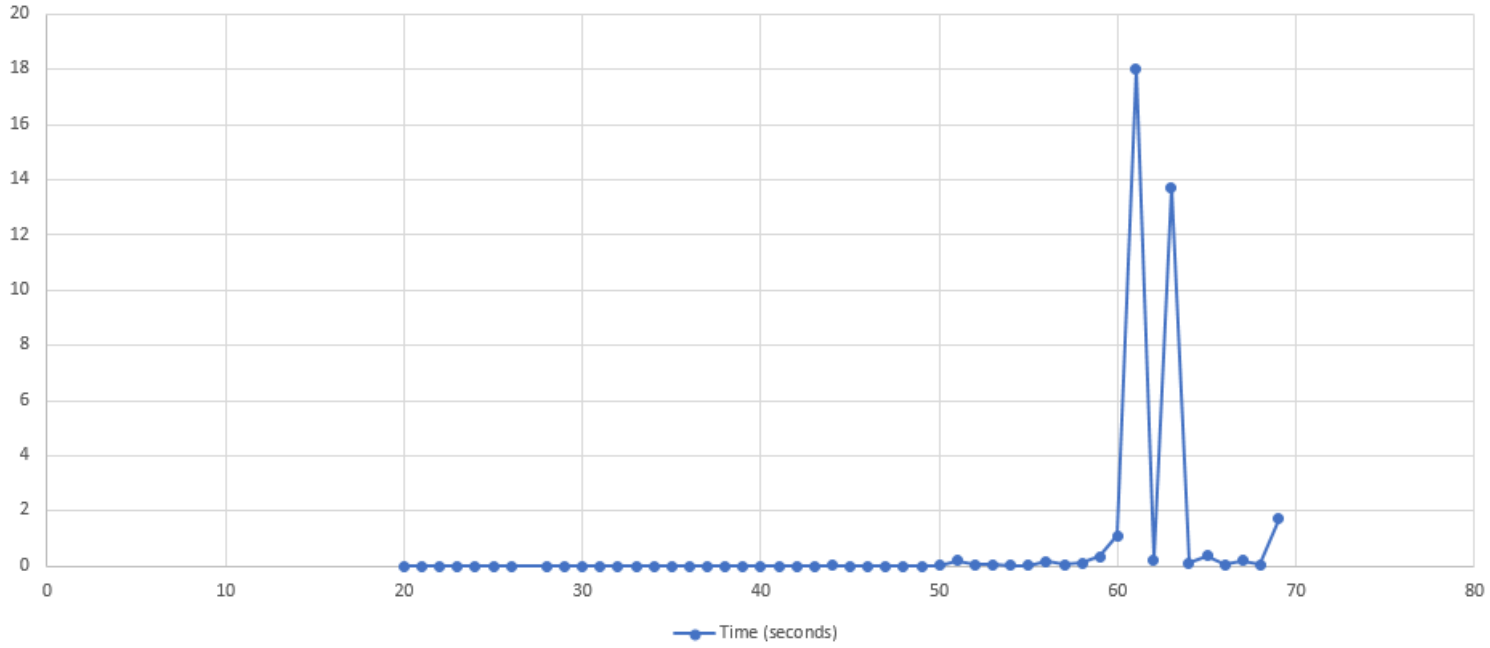
The worst case Big-O complexity of the solve() method is $O(n^2)$. This program uses a brute-force approach to solving the problem thus this is the worst case runtime. Generally, brute-force algorithms should not be used due to their inefficiency and can take a long time to execute. However, in this case it is appropriate, as the Sudoku Grid is of a small set size which in turn limits the size of n where n is the number of empty cells. This allows for any modern computer to run this program in a reasonable amount of time.

If it were possible, merging the isValid() and solve() method would increase the efficiency of the program. Since the solve() method traverses the grid the same way the isValid() method does, the grid can be checked within the solve() method whilst the digits of empty cells are changed. This would allow only two nested for loops to be used throughout the two classes. In this case, it would be advisable to put some of the tasks of this large solve() method into their own private class methods, to better fit an Object Oriented Design.

Task 3: Solving Puzzles

The Main class contains the main method which solves all four hundred puzzles present in the Examples class. Each puzzles completion is timed in milliseconds which is then calculated into seconds. The overall time for all 400 puzzles to be solved is also timed in milliseconds and then calculated into minutes. The Solve class contains a method to find the number of empty cells in the current Sudoku Grid and is used to print that value before solving the puzzle.

The results from timing the puzzles confirmed the Big-O runtime calculated in Task 2. Compared to the other puzzles, those that contained more empty cells took significantly longer to complete . Here is a graph to show the average time taken to solve a puzzle with a certain number of empty cells.



Graph 1; Time Taken (seconds) against Number of Empty Cells

The results in Graph 1 show that on average, the puzzles with more empty cells will significantly increase the amount of time the program takes to complete a puzzle. However, at the end of Graph 1 the time taken to solve the last few puzzles dramatically falls, which was an unexpected result. Upon further investigation, those puzzles in particular favoured a brute-force approach, as the first few attempts at inserting numbers into the grid the program made, would be acceptable.

Task 4: Generating Puzzles

The isValid() method would be very useful in aiding the generation of puzzles as it can check whether the puzzle is actually solvable as it is being generated. I created a pseudocode algorithm which would generate random puzzles. The algorithm is as follows:

```
generator {
        limit ← 0
        for each grid cell {
                value ← randomNum(0, 1)
                // 'randomNum()' gives a random number in the range of the two parameters
                if value = 1 AND limit <=6 {
                        limit++
                        for each digit 1-9 random order {
                                cell ← digit
                                 if grid.isValid() {
                                        break
                                }
                                if 9th time running {
                                        cell ← 0
                                        limit ← 7
                                }
                        }
                } else {
                        cell ← value
                }
                if at the end of a row {
                        limit ← 0
                }
        }
}
```

The easiest way to implement this would be to insert a random digit from zero to nine in each cell, to see whether it's valid or not. In an attempt to achieve a more balanced grid regarding empty and filled cells, the algorithm above is more complicated than if it were to achieve basic functionality. There is a 50% chance an empty cell and if it is not empty, then a random number, from one to nine, will be placed instead. The algorithm could be improved by allowing it to recursively modify the grid in order to change previously placed values. This would prevent an uneven spread of numbers in the grid. However, the implementation above is simple and efficient at creating valid Sudoku Puzzles.

Self-evaluation

I believe the most challenging part of this project involved the isValid() method and its efficiency. I made multiple attempts at developing different implementations that would allow the program to solve all four hundred puzzles in a shorter amount of time. One of these attempts included making use of integer arrays to store the values currently being checked and checking for duplicates within that array. This was not faster, as it required the array to have all nine values within it first, before the program could check for duplicates. Hash Sets were able to check as soon as a value was placed, making it a more efficient solution.

I think I did very well at implementing the Solve class and the Main class as both work flawlessly. The Solve class perfectly implements the algorithm given for the solve() method and also has an extra method to check how many empty cells are currently present in the grid. The main method uses the other classes to solve all four hundred puzzles, while calculating and providing other useful information about each puzzle. It also writes an average of the times taken to solve puzzles, with a certain amount of empty cells, to a text file. This can be useful when you want to export the data, for example I used this functionality to aid in the creation of Graph 1.

Overall, I am confident that my attempt at the assignment has successfully completed all the tasks that were specified. I have also added extra functionality, such as the empty cell counter method and file writer method. These features amalgamate well with the rest of the program and could prove useful to the user. I expect to get a mark around 60 to 70% for this assignment.