PSEUDO-KU
Java Sudoku Solver using Backtracking

Dino Anastasopoulos: 1900661
Philani Mpofu: 1848751
Reece James Peters: 1924514

07 October 2020

Contents

1 Aims

The main aim of this lab report is to review the performance of our group's implementation of the Backtracking Algorithm on solving sudoku puzzles with varying amounts of zero spaces in the question puzzle. Conversely, one could also interpret it as reviewing the algorithm's performance on solving sudoku puzzles with varying amounts of filled spaces or clues in the initial puzzle. In this report we hypothesize that the more open spaces (or the fewer number of clues) there are in the initial sudoku puzzle, then the more computation will be required to solve such a puzzle.

It is thus our goal to verify our theoretical assumptions (which will be expanded upon in subsequent sections) of the algorithm's performance with the empirical evidence we obtain from running our program on many different sudoku puzzles. Simply put, we want to check that the algorithm has the same space and time complexities that we believe it to have theoretically. If this is not the case, then it is also our goal to determine why it is not.

2 Summary of Theory

Our initial assumption is that the Backtracking Algorithm will have the following space and time complexities when solving 9x9 sudoku puzzles:

2.1 Space Complexity : O(n^2)

The initial and solved sudoku puzzle would need to be stored in a 2D matrix made up arrays of size n. Thus, in any case the space complexity would be O(n^2). [2]

2.2 Time complexity: O(9^{n^2})

There exists 9 possible options for every open space in the initial sudoku puzzle. In the worst case, the algorithm would need to visit every open space in every iteration of the algorithm and change all of the values 9 times. Thus, the time complexity in this case would be O(9^{n^2}) where n is the number of open spaces. In the best case, the algorithm would need to visit every open space once and change the value within that open space once. Thus, the time complexity in the best case would be O(n^2). [2]

3 Experimental Methodology

As stated in our aims, our initial assumption is that more open spaces (or fewer clues) in the initial sudoku puzzle will result in more computation needed in order to find the unique solution of a given puzzle. We have utilized 4 metrics to measure the performance of the algorithm, namely number of comparisons, number of changes, number of recursive calls, and the amount of time taken to return a unique solution to a given puzzle. Please see below discussions on our chosen metrics:

3.1 Number of comparisons

This metric measures the number of times the program compares the current value of the puzzle it is on with the rest of the incomplete puzzle. At each iteration, the program checks that the current value does not break the puzzle (ie: it checks to see that the current value of the puzzle is not repeated within that value's row, column or sub-grid. It is our assumption that puzzles with more empty spaces will need more comparisons than puzzles with less empty spaces. [3]

3.2 Number of changes

This metric measures the number of times the program changes a value it has just compared to a value that will not break the board (since it has just found that the value at the current position has broken the board). It also measures the number of times the program changes an empty space to a filled space (a space with any positive integer less than 10). Again, our group assumes that the program will make more changes to puzzles with more empty spaces than it will to puzzles with less empty spaces. [3]

3.3 Number of Recursive Calls

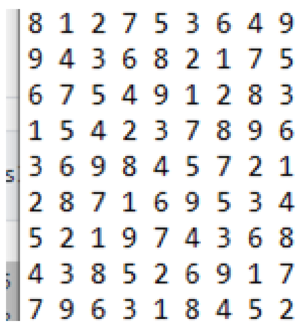
This metric measures the number of times the program is recursively called to find a unique solution to the given puzzle. Whenever the program finds a value that breaks the puzzle the Backtracking Algorithm gets called at that point and the program starts again from the beginning. The value that broke the puzzle will not be considered at this function call. This process is done to find a value for that position of the puzzle that doesn't break the rest of the puzzle. Again, we believe that more empty spaces will lead to more recursive calls.

3.4 Run Time

This metric measures the length of time (in milliseconds) it takes for the program to output a unique solution to a given puzzle. It is our assumption that puzzles with more empty spaces will take much longer to complete than puzzles with less empty spaces. [3]

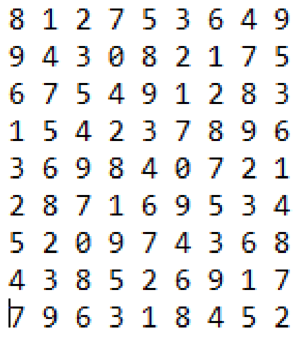
Initially, we had decided to use a puzzle's difficulty, instead of the number of open spaces it has, as the independent variable to measure the performance of our program. We had graphed the difficulty of the graph against the number of comparisons, number of changes, number of recursive calls and run time to solve the puzzle. However, this metric was abandoned for a number of reasons. Firstly, the graphs that were generated using this variable gave very little insight to the performance of our program (if any). Secondly, we found that in some cases puzzles of differing difficulties gave the same results. This is obviously incorrect, because an easy puzzle should not be getting the same results as a medium puzzle for example. We discovered that a 'hard' easy puzzle would yield the same results as an 'easy' medium puzzle. Thirdly, it is very difficult to define a puzzle's difficulty. Difficulty is a concept that can be understood by a human, but it would be very abstract to a computer. One could look at the number of clues to determine that puzzle's difficulty, but this is approach is a bit simplistic, as the positions of those clues could be a factor to consider when defining difficulty.

To measure the performance of our program we started off with the initial solved puzzle, ran our program on it and recorded and plotted the results. Then we randomly removed 3 clues from that puzzle and repeated that same process. This process was repeated until there were no more clues left in that puzzle (ie: the puzzle was left only with white spaces). This process made up our empirical experimentation. Here are some illustrations of the first three iterations of the algorithm of a puzzle taken from sudoku.com (2020) [1]



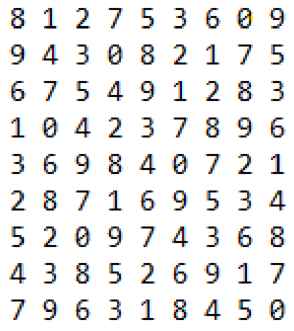
```
8 1 2 7 5 3 6 4 9
9 4 3 6 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 5 4 2 3 7 8 9 6
3 6 9 8 4 5 7 2 1
2 8 7 1 6 9 5 3 4
5 2 1 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2
```

Figure 1: Iteration 1



```
8 1 2 7 5 3 6 4 9
9 4 3 0 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 0 7 2 1
2 8 7 1 6 9 5 3 4
5 2 0 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2
```

Figure 2: Iteration 2



```
8 1 2 7 5 3 6 0 9
9 4 3 0 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 0 4 2 3 7 8 9 6
3 6 9 8 4 0 7 2 1
2 8 7 1 6 9 5 3 4
5 2 0 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 0
```

Figure 3: Iteration 3

As you can see, the puzzle in the first iteration has no white spaces as that is the solved puzzle. The puzzle in the second iteration has white spaces in positions (1,3), (4, 5) and (6,3). The puzzle in the third iteration has white spaces in positions (1,3), (4,5), (6,3), (0,7), (3,1) and (8,8). Results of the performance of this implementation of the algorithm will be presented, interpreted and reviewed in subsequent sections.

4 Presentation of results

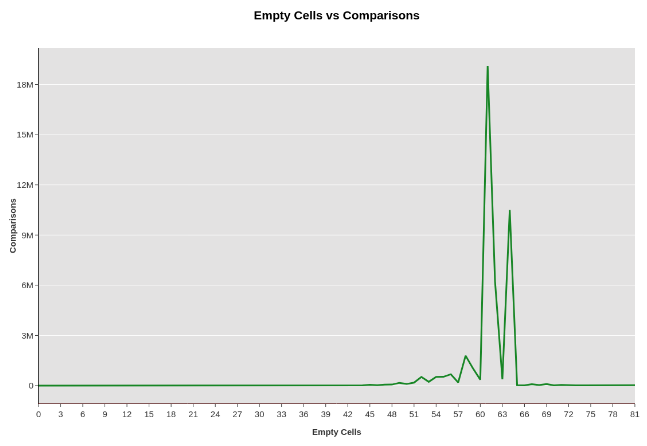


Figure 4: Number of Comparisons

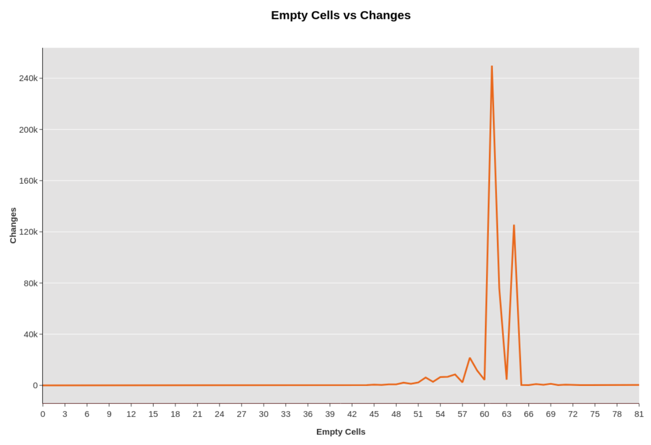


Figure 5: Number of Changes

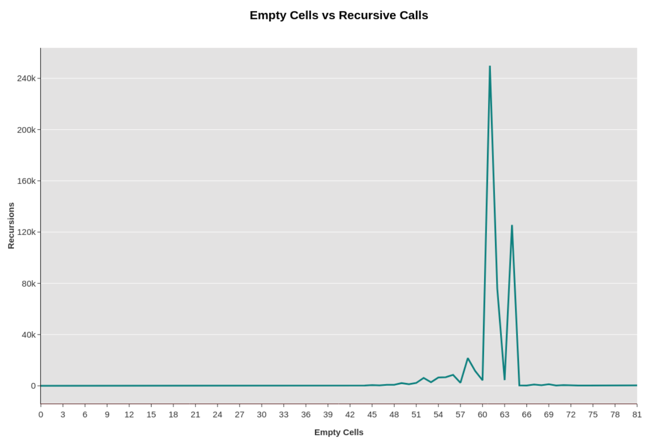


Figure 6: Number of Recursive Calls

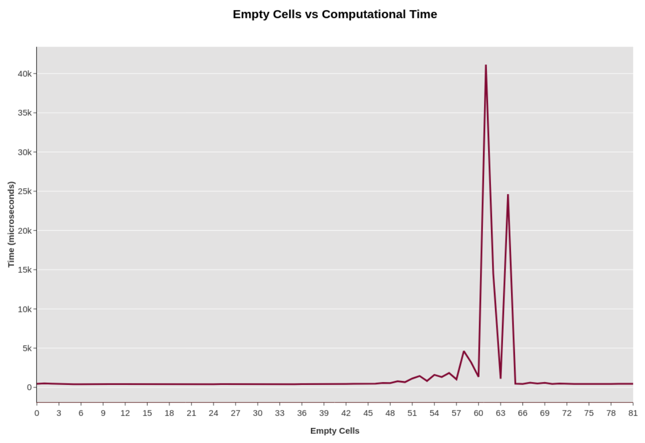


Figure 7: Run Time

5 Interpretation of results

The results obtained, while occasionally inconsistent, did lend to the discovery of trends and patterns within the performance of the algorithm. All four metrics grew, and shrunk, at incredibly similar rates. This makes sense as changes to a block can only be made once comparisons to other blocks have taken place, and time is directly tied to the number of processes that need to run (i.e. array accesses, integer comparisons, and variable changes).

During the very early iterations (0-20 empty cells) the metrics were quite similar, in that they were fairly low and did not seem to grow by much as clues were removed. This is most probably due to the fact that the algorithm was finding the correct number for each empty empty block on its first or second try, as there was abundance of clues on the board deterring the algorithm from placing the wrong number in said block.

As the number of empty blocks grew beyond this point (20-40 empty cells) the metrics began to grow slightly faster, but still quite linearly. We assume this is tied to the reasoning in the previous paragraph, and growth can attributed to the fact that more blocks were empty at each iteration, so while the algorithm was still finding the solution for each block rather quickly, there were simply more blocks to check and fill.

It should be noted that time was fairly constant for all aforementioned states. This is most likely due to the fact that the minuscule increases in comparisons and changes made little to no difference to the processing capacity of our 21st century computers.

The meat of the growth lies in this next interval (40-60 empty cells). The increase in all metrics was exponential and usually maxed out between 55 and 65 empty cells). We assume this is due to the fact that at this point the board starts having more empty cells than clues. This means the algorithm has more blocks to fill and less information to guide it to the correct answer. As such it will naturally go through more and more recursions to find the correct value for each block, as it is unlikely to find them early on. As a result more comparisons will have to be made for each block as well. These factors compound to give us the exponential growth observed. It is also, unsurprisingly, in this interval that time complexity starts increasing.

A very unexpected discovery is what happens during the last interval (60-81 empty cells). All metrics seem to drop rapidly and then stay fairly consistent until the end. We have deduced that this is due to the fact that there are so few clues left on the board that the algorithm has very few restrictions on what it is allowed to place in each block, and thus finds the result fairly quickly as there are numerous solutions that can be built from the very few number of clues. At this point the algorithm is basically making its own solution to the puzzle.

6 Theory relationship

The results have both confirmed and debunked our theories. We expected to see constant exponential growth due to the factors we had anticipated throughout the data, but this was only the case for the interval of 40-60 empty cells. The rest of the other datapoints were fairly similar and linear due to the nature of sudoku and the approach our algorithm takes to solve it.

It is interesting to note that that sudoku generally measures its difficulty on its number of clues, where easy puzzles tend to have about 38 clues (43 empty blocks), medium ones about 30 clues (51 empty blocks) and difficult ones about 22 clues (59 empty blocks). It is no coincidence that all of these lie within the interval where we noticed most of our expected trends. Anything less or more and the puzzle simply becomes too easy for both human and computer alike.

7 Conclusion

It has been determined that our algorithm does have a complexity of $\mathrm{O}(9^{n^2})$ but only when the data lies within a certain interval. The best case runs in $\mathrm{O}(n^2)$ and occurs when all empty cells are the only empty cell in their block, row and column. The worst case runs in $\mathrm{O}(9^{n^2})$ and occurs when there are about twice as many empty blocks as there are clues. Due to the strange nature of the data and sudoku as a whole, the 'average' case does not seem to be determinate.

References

[1] Easybrain. *Sudoku.com*, 2020 (accessed 19 October 2020).

[2] GeeksforGeeks. *Sudoku — Backtracking-7 - GeeksforGeeks*, 2020 (accessed 15 October 2020).

[3] Sian Jones, Paul Roach, and Stephanie Perkins. *Sudoku Puzzle Complexity*, 03 2011 (accessed 12 October 2020).