

PSEUDO-KU  
Java Sudoku Solver using Backtracking

Dino Anastasopoulos: 1900661

Philani Mpofu: 1848751

Reece James Peters: 1924514

07 October 2020

# Contents

<b>1</b>	<b>Aims</b>	<b>2</b>
<b>2</b>	<b>Summary of Theory</b>	<b>2</b>
2.1	Space Complexity : $O(n^2)$ . . . . .	2
2.2	Time complexity: $O(9^{n^2})$ . . . . .	2
<b>3</b>	<b>Experimental Methodology</b>	<b>3</b>
3.1	Number of comparisons . . . . .	3
3.2	Number of changes . . . . .	3
3.3	Number of Recursive Calls . . . . .	3
3.4	Run Time . . . . .	3
<b>4</b>	<b>Presentation of results</b>	<b>6</b>
<b>5</b>	<b>Interpretation of results</b>	<b>8</b>
<b>6</b>	<b>Theory relationship</b>	<b>9</b>
<b>7</b>	<b>Conclusion</b>	<b>9</b>

# 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 incomplete cells (having value 0). Conversely, one could also interpret it as reviewing the algorithm's performance on solving sudoku puzzles with varying amounts of completed cells or clues (having non zero values). In this report, we hypothesize that puzzles with fewer clues (more cells with 0 values) will require more computation in order to be solved.

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 testing our algorithm 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 incomplete cell. In the worst case, the algorithm would need to visit every incomplete cell in every iteration 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 incomplete cells. In the best case, the algorithm would need to visit every incomplete cell once and change the value once only. 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 puzzles with more incomplete cells (or fewer clues) will result in more computation needed to find the unique solution. We have utilized 4 metrics to measure the performance of the algorithm, namely the 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 value of the current cell in the puzzle with the rest of the incomplete puzzle. At each iteration, the program checks that the current value does not break the puzzle (ie: it ensures that the value of the current cell is not repeated within that row, column or sub-grid).

### 3.2 Number of changes

This metric measures the number of times the program determines that a specific value is safe to enter in the current cell and so changes its value from a zero (incomplete cell), to the determined number.

### 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.

### 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 incomplete cells will need more comparisons, more changes, more recursive calls and a longer runtime than puzzles with less incomplete cells(more clues).[3]

Initially, we had decided to use a puzzle's difficulty (easy, medium, difficult, expert) as the independent variable to measure the performance of our algorithm. We had plotted the difficulty of the graph against the number of comparisons, changes, 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). This was due to the fact that easy puzzles generally have 38 clues, and expert puzzles have up to 21 clues at a minimum, which is not a big enough range to accurately measure the performance. Secondly, we found that in some cases puzzles of varying 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 "difficult" easy puzzle would yield the same results as an "easy" medium puzzle. Thirdly, it is very difficult to define a puzzle's difficulty by a word rather than a scale of values as it is a concept that can be understood by a human, but it would be very abstract to a computer.

We then realized it would be a better option to analyse our algorithm based on number of clues as a way to determine that puzzle's difficulty. This approach proved to have flaws, as the positions of those clues could also 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 incomplete cells). This process made up our empirical experimentation. Below are some illustrations of the first three iterations of the algorithm performed on puzzle taken from Sudoku.com [1]

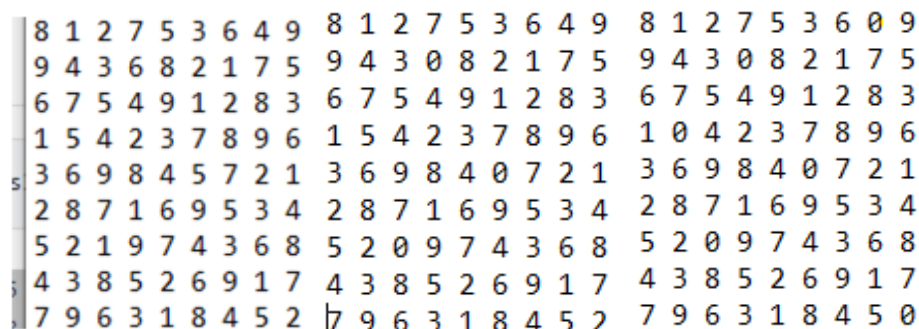


Figure 1: Iteration 1

Figure 2: Iteration 2

Figure 3: Iteration 3

As you can see, the puzzle in the first iteration has no white spaces since it 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

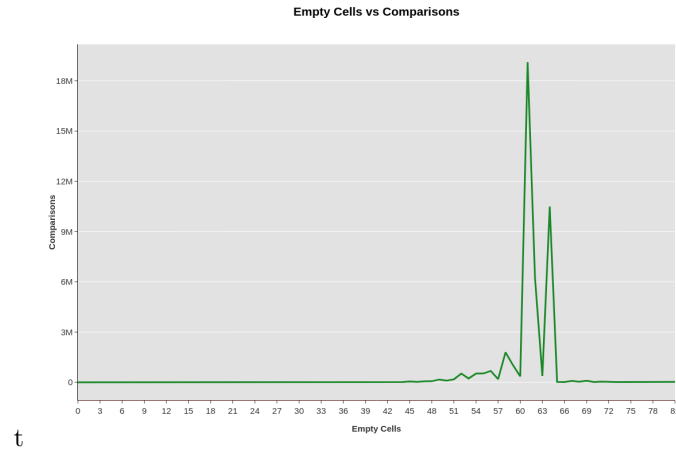


Figure 4: Number of Comparisons

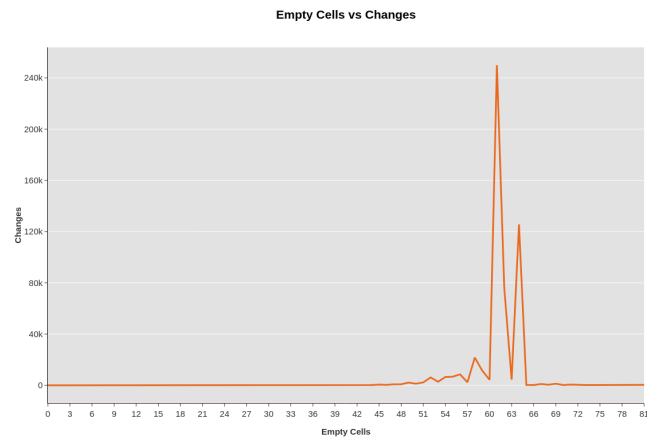


Figure 5: Number of Changes

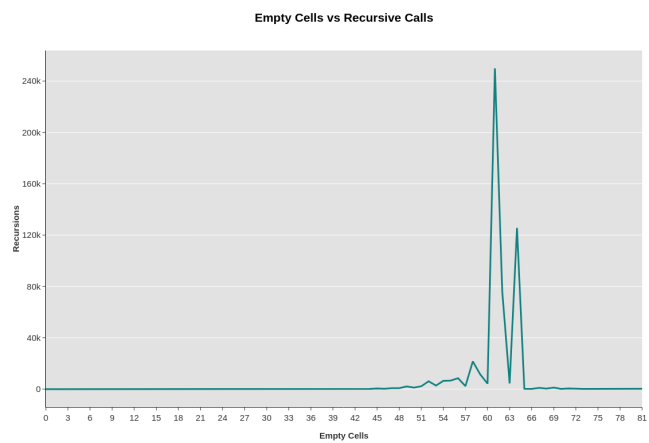


Figure 6: Number of Recursive Calls

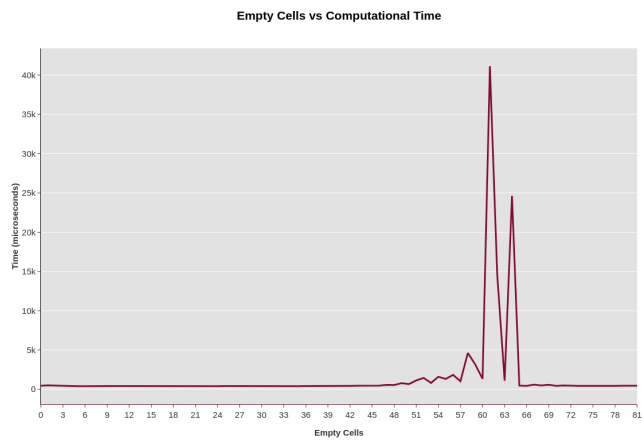


Figure 7: Run Time



## 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 since changes to a cell can only be made once it has been compared to other cells, 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 incomplete cells) the metrics were quite similar, in that they were fairly low and did not seem to grow a significant amount as clues were removed. This is most probably due to the fact that the algorithm was finding the correct number for each incomplete cell on its first or second try, as there was abundance of clues on the board deterring the algorithm from placing the wrong number in each cell.

As the number of incomplete cells grew beyond this point (20-40 incomplete 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 be attributed to the fact that more cells were incomplete at each iteration, so while the algorithm was still finding the solution for each cell rather quickly, there were simply more cells 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 incomplete cells). The increase in all metrics was exponential and usually maxed out between 55 and 65 incomplete cells). We assume this is due to the fact that at this point the board starts having more incomplete cells than clues. This means the algorithm has more cells 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 cell, as it is unlikely to find them early on. Thus, more comparisons will be made for each cell as well. These factors compound to give us the exponential growth observed. It is also, unsurprisingly, in this interval that the time complexity starts increasing.

A very unexpected discovery is what happens during the last interval (60-81 incomplete 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 which value it is allowed to place in each cell, 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 incomplete cells. The rest of the 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 incomplete cells), medium puzzles about 30 clues (51 incomplete cells) and difficult ones about 22 clues (59 incomplete cells). 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  $O(9^{n^2})$  but only when the data lies within a certain interval. The best case runs in  $O(n^2)$  and occurs when all incomplete cells are the only incomplete cell in their row, column and sub grid which results in only 1 "legal" value that can be chosen. The worst case runs in  $O(9^{n^2})$  and occurs when there are about twice as many incomplete cells 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).