Java Sudoku Solver using Backtracking

Pseudo-Ku

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 initial 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 best, worst and average case 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

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 3 metrics to measure the performance of the algorithm, namely number of comparisons, number of changes, 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. [1]

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 found that the value at the position is currently on 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 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.

Add in : 2 implementations – one random and one done linearly (adding in white spaces) - also add in that we initially tried doing difficulty, but results were not as comprehensible as the ones we get with the three chosen metrics. And also graphs look the same for difficulty (ie: the hardest easy graphs will look similar to the easiest medium graphs) - also add in that it's difficult to define difficulty, because it's not only number of clues that matter, but also the position of those clues that will matter as well.

4 Presentation of results

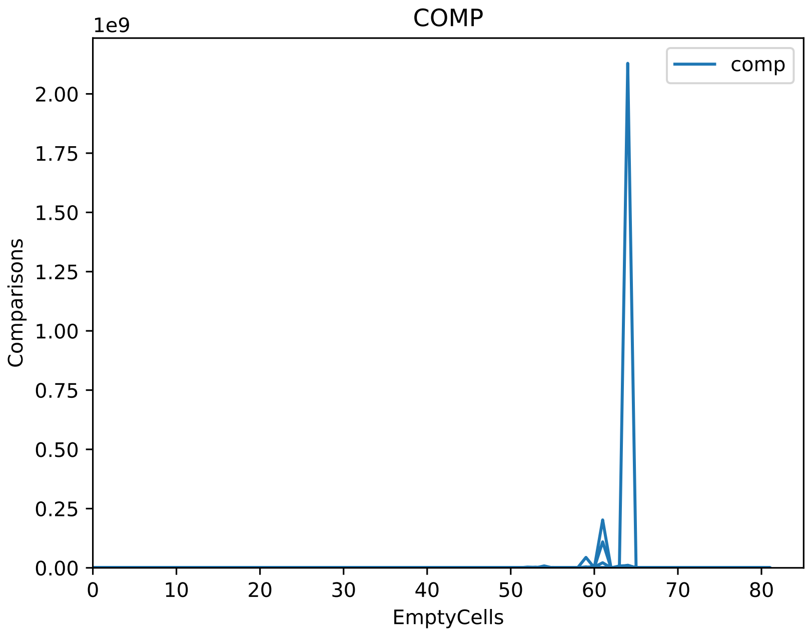

Figure 1: Comparisons Plot

5 Interpretation of results

6 Theory relationship

7 Conclusion

8 References

References

[1] Sian Jones, Paul Roach, and Stephanie Perkins. *Sudoku Puzzle Complexity*, 03 2011.

9 Acknowledgements