Solving sudoku using a backtracking algorithm

Dlamini Christopher(1965919). Hlomuka Siyabonga(). Musara Walter(1830229)

October 16, 2020

0.1 Aim

Given our implementation of a sudoku solving algorithm , that makes use of backtracking, we want to empirically analyse the perfomance of the said algorithm.Next we will compare the empirical analysis's results to the theory and see how our algorithm fares against a bruteforce approach. We assume all input boards are 9x9.

0.2 Summary of theory

0.2.1 sudoku

1 defines Sudoku as a logic-based, combinatorial number-placement puzzle. In classic sudoku, the objective is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

0.2.2 Backtracking algorithm

The obvious way for a computer to solve it is using a bruteforce method. This ,of course, is impractical so smarter algorithms are needed. One such algorithm is one that uses backtracking.

A backtracking algorithm for soduku recursively attempts to solve a given soduku puzzle by testing all possible configurations towards a solution until a solution is found.Unlike a bruteforce approach, each time a configuration is tested, if a solution is not found, the algorithm backtracks to test another possible configuration. This goes on till a solution is found or all configurations have been exhausted.

0.2.3 Perfomance of backtracking algorithm

For every unassigned index, there are 9 possible options so the time complexity is

$$O(9^{n^2}) \tag{1}$$

The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.

0.3 Experimental Methodology

The algorithm is implemented in java.To measure the perfomance we record the current time right before the algorithm starts solving a puzzle and then right after. This will indicate the approximate time it took to solve the sudoku puzzle. There will be different sudoku puzzle inputs.These puzzles will differ in the level of difficulty and all their solve-times will be recorded. We will also count the number of operations on the different puzzles. All of these results will be aid us in coming to a decisive conclusion after analysis.

0.4 Results

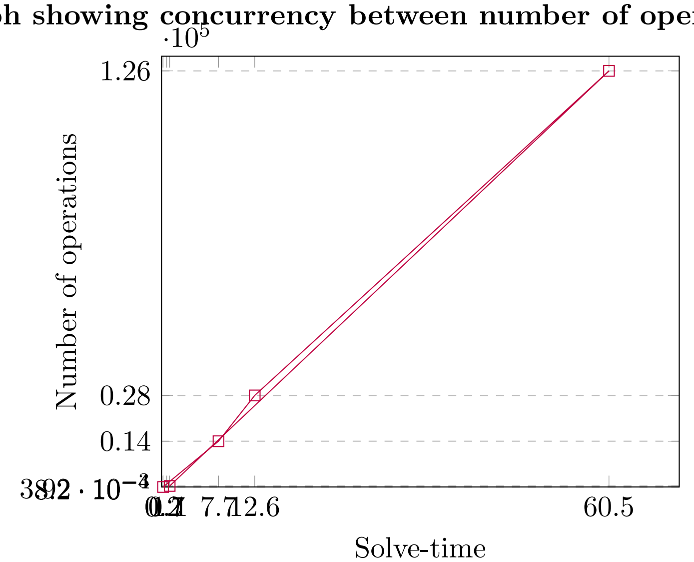
0.4.1 Some sample inputs with their solve-times

Table 1: Sample inputs and their results

input	**time(ns)**	**operations**
easy 0	712796	82
easy 1	644607	82
medium 0	834085	178
medium 1	1100161	503
hard 0	6623904	9741
hard 1	12554631	27694
very hard 0	13851059	27040
very hard 1	7422293	11355
solved puzzle	169714	82
empty puzzle	1087386	392
missing 1 value	187842	82

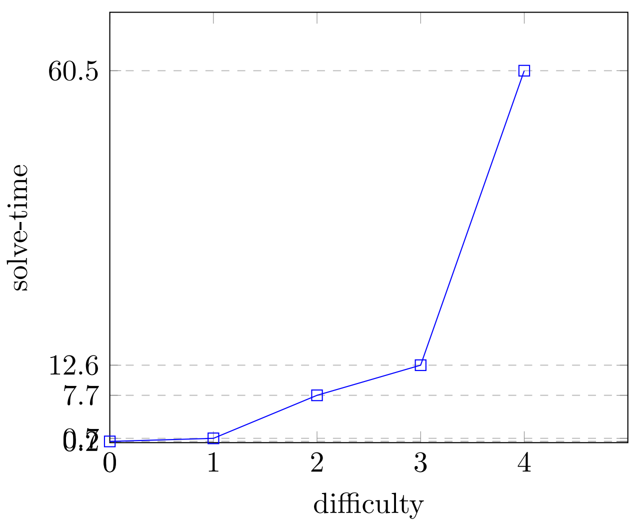
Taking the puzzle inputs with the longest durations from each difficulty level produces the following graph checking concurrency:

Plot0: Graph showing concurrency between number of operations and solve-time

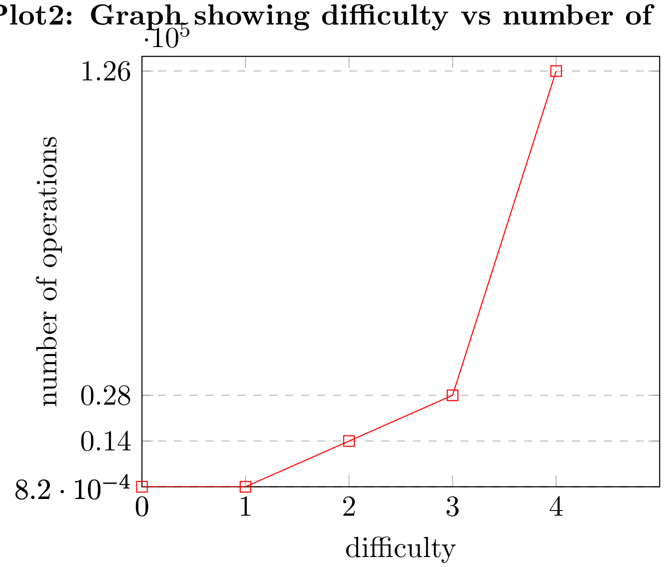


Now we check the perfomances with respect to the level of difficulty:

Plot1: Graph showing difficulty vs solve-time



Plot2: Graph showing difficulty vs number of operations



0.5 Interpretation of results

The table and the 3 plots above show some reassuring trends:

- There is a strong positive correlation between the solve-time and the number of operations as seen in Plot0.
- The solve time is directly proportional to the level of difficulty. Plot1 shows this.
- As expected given the first 2 statements, the number of operations the algorithm does is also directly proportional to the level of difficulty. This is confirmed in Plot2.

0.6 Relating the results to the theory

The theoratical complexity is given by $O(9^{n^2})$, in this case being $1.97 * 10^{77}$.This is confirmed by the results as no number of operations to solve single sudoku puzzle reached $1.97*10^{77}$, thus the equivalent time of $1.97*10^{77}$ operations will not be reached due to the strong positive correlation between operations and time to solve puzzle.

0.7 conclusion

Solving Sudoku using a backtracking algorithm method, like our algorithm, means trying each available number across all empty cells. Such an algorithm has a runtime complexity of $O(9^{n^2})$, where n is size of the Sudoku puzzle. The algorithm performs $1.97 * 10^{77}$ operations to find a solution. That is impractical, however in practice the runtime varies according to the difficulty of the puzzle itself and the number of options for each empty cell ,as our results have shown. Therefore the time complexity of $O(9^{n^2})$ is valid for the backtracking algorithm as it is just the upper bound. Our algorithm will have a better perfomance than the bruteforce approach.

0.8 Sample inputs

0.8.1 easy 0

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

0.8.2 easy 1

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

0.8.3 medium 0

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

0.8.4 medium 1

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

0.8.5 hard 0

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

0.8.6 hard 1

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

0.8.7 very hard 0

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

0.8.8 very hard 1

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

0.8.9 missing 1 value

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

0.9 References

[1] https://www.wikiwand.com/en/Sudoku
https://www.101computing.net/backtracking-algorithm-sudoku-solver/
https://en.wikipedia.org/wiki/Sudoku
https://www.geeksforgeeks.org/sudoku-backtracking-7/
https://medium.com/optima-blog/solving-sudoku-fast-702912c13307