The Effect of Guess Choices on the Efficiency of a Backtracking Algorithm in a Sudoku Solver

A Comparison of the Number of Backtracks Performed
by a Backtracking Algorithm in a Sudoku Solver

Moriel Schottlender
School of Engineering and Computing Sciences
New York Institute of Technology
New York, New York, USA
mschottl@nyit.edu

***Abstract*— There are several possible algorithms to automatically solve Sudoku boards; the most notable is the backtracking algorithm, that takes a brute-force approach to finding solutions for each board configuration. The performance of the backtracking algorithm is usually said to depend mainly on two implementation aspects: finding the next available empty cell in the board and finding the options of available legal number that are relevant to the given cell.**

While these pieces of the backtracking algorithm can vary in efficiency based on their implementation, tests show that the algorithm itself also relies on the statistical distribution of the guesses that it attempts to "plug in" to the board in every given cell. The backtracking algorithm uses an array of the legal numbers in the cell to attempt a solution before it moves on to the next cell. If a solution cannot be found, it backtracks and attempts to solve the board again with a different guess choice. The more errors the solver makes, the more backtracks it must perform, which decreases its overall efficiency and increases its effective runtime.

Tests of the solving algorithm were performed using 195 base solutions with multiple initial board configurations were performed to analyze the difference in the algorithm performance by comparing the number of recursive backtracks between sequential and randomly distributed guesses. Analysis show that using values that are given in a shuffled array significantly reduces the number of backtracks done by the solver and, as a result, improve the total effective efficiency of the algorithm as a whole. *(Abstract)*

Keywords-backtracking; sudoku; deterministic; random; stochastic (key words)

I. Introduction

Sudoku is a popular game that is played by millions of people every day. Sudoku puzzle solving has been shown to belong to the category of NP-complete problems [8]. However, this is true for the general $n^2 \times n^2$ Sudoku boards. For the case of the popular Sudoku game of 9x9 matrix, the problem is reduced to have finite amount of solutions. These boards can be solved with brute-force algorithms.

There have been several papers that delved into the subject of creating algorithmic methods of solving 9x9 Sudoku boards, and several known algorithms have been developed. The most straightforward algorithm is the backtracking algorithm, a brute-force approach to guess solutions recursively.

While the backtracking algorithm is relatively simple, its runtime optimization is usually said to depend on the implementation of its mechanics; that is, on the efficiency of finding new empty squares and on that of finding available numerical solutions per empty square. This paper will concentrate on a different element that affects the optimization of the backtracking algorithm in solving Sudoku games: the manner in which the algorithm chooses which of the available guess to plug into the board.

The most noticeable difference between the choices of guesses is the type of solutions that are produced. When the algorithm chooses its guesses in sequential order, the solutions given are deterministic; the same solution – with the same backtracks and recursions – will appear for the same initial board configurations. If the choice of guesses is shuffled, the

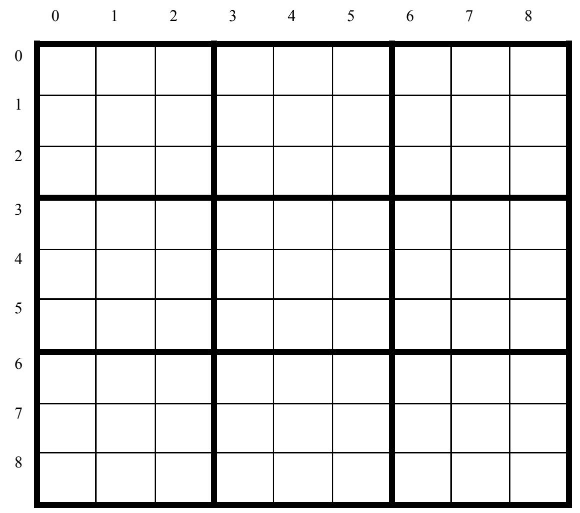

Figure 1. The classic Sudoku board: a 9x9 matrix with 3x3 subsections

solutions are varied, and so is the number of recursions and backtracks that the backtracking algorithm performs.

This paper will discuss the effect of choosing ordered versus shuffled guess choices behavior on the number of recursions and backtracks.

II. The Mechanics of Sudoku puzzles

A. The Rules and Structure of the Game

Before we analyze the behavior of the backtracking algorithm in a Sudoku solver, we should understand how the Sudoku game works. The game is comprised of a 9x9 matrix of cells and is subdivided logically to nine 3x3 segments, as seen in Figure 1. Each cell is allowed to hold an integer between 1 and 9. Numbers must only appear once in each row, in each column and in each 3x3 section.

When we consider the possible values a certain cell can hold, we must evaluate first the cells in the same row, column and segment. If any of those cells contain a number, that number is removed from consideration as a valid choice for the current empty cell. This results in an array of legal numbers that we are allowed to consider for the current empty cell. When solved by hand, players usually mark the list of available numbers at the edge of each cell and then remove these as more and more values are filled in, resulting in smaller available numbers to choose for each cell. The game is won when all cells are filled in with legal values.

III. The Backtracking Algorithm

A. General Purpose

The backtracking algorithm is a fairly straight-forward, brute-force approach to solving puzzles through applying possible guesses and, if needed, retracing its steps to replace guesses with new ones.

When applied to Sudoku puzzles, the algorithm travels across the 9x9 grid and finds relevant valid numbers for each given empty cell. It proceeds to pick a number from the available options and apply it as a guess to the cell, then continue to the next. When it encounters a problem, such as a case where no more valid numbers are available, it backtracks to the previous examined cell and changes its value to the next possible number.

Generally, the efficiency of the backtracking algorithm depends on the implementation of finding the next empty cell and finding the array of valid numbers for the given cell.

B. Implementation

Since this paper focuses on the effect of guess choices on the performance of the backtracking algorithm, the algorithm itself was implemented with a simple and straight-forward approach following the basic principle of backtracking, and without any special concern for optimization. The pseudo code of the implementation is in Figure 3, based on an explanation in [1] and [2].

The algorithm simply searches for the nearest empty cell in each operation by looping through the Sudoku board matrix.

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

Figure 2. Deterministic solution for an empty Sudoku board

```
solve(game):
  // Count recursions
  counter_recursion++
  if no empty cells found return SUCCESS
  else
    // Find the next empty cell in the matrix
    current_cell = getNextEmptySquare()
    // Find the array of legal numbers for that cell
    legal_numbers_array = getLegalNumbersForCell()
    for each number in legal_numbers_array
      // Insert guess
      Set value of current_cell to current legal number
      if ( solve(game) )
        return SUCCESS
      else
        // Backtrack
        counter_backtracks++
        Remove value from current_cell
    // If all guesses in the array failed, return failure
    return FAILURE
```

Figure 3. Psedo-code of the backtracking algorithm used in this paper.

Once found, the given cell will be examined to find an array of possible legal numbers.

Finding the legal numbers is also straight-forward. We begin with an array full of all numbers 1-9. Then, we go over the cells in the row, column and 3x3 segment to find the cells with existing values; each values that is found in any of those is removed from the initial array, so that at the end we are left only with numbers that are not present in either the row, column or segment, and are valid and legal numbers the given cell. The algorithm then picks a number from the given array for its guess choice.

C. Guess Choices: Ordered vs. Random

The backtracking algorithm checks for the legal numbers of each new empty cell is focused on in an attempt to solve the board. These numbers are given in some array, and the algorithm picks the guesses in order. The array itself, however, can be sorted or shuffled.

For example, assuming the board is empty, the array of legal numbers for the first empty cell would be all numbers from 1 to 9 and the backtracking algorithm will always go over this array in order. The difference, however, would be between serving the backtracking algorithm a simply ordered array: [1,2,3,4,5,6,7,8,9] or a shuffled one, such as [9,3,5,8,1,2,7,6,4]. The simply ordered array can also be ordered in reverse; tests show this doesn't change any of our results.

Since the ordered array is always the same for each cell in every initial condition, it produces deterministic solutions that have the same backtracks and recursions. For a solution of an empty board, the first empty cell, using the legal number options mentioned above, would receive the value 1. The second cell, now having the legal number array [2,3,4,5,6,7,8,9] will receive the number 2, the third array 3, etc., the result of this operation can be seen in Figure 2.

The Fisher—Yates shuffle was used to randomly shuffle the choice array. The algorithm was performed each time an empty cell was first analyzed to retrieve its initial guess choices array.

Unlike the ordered array, using the shuffled array results in each cell receiving a random value out of the legal available numbers which results with varied amount of backtracks and recursions even for the same board configuration. It also results in non-deterministic solutions for each board configuration.

Beyond the issues of deterministic and non-deterministic solutions, choosing guess choices from a shuffled array for an initially empty board produces significantly fewer recursions and backtracks on average, serving as further optimization for the backtracking algorithm.

This behavior raised the question as to whether this is a consistent trend that exposes the guess choices as another optimization factor to be considered. To test this further, the algorithm was run on multiple base solutions with different configurations of prefilled numbers.

IV. Testing Methodology

Two groups of tests were performed to gather information about the effect of the two different types of guess choices. The first test was built around the deterministic solution for an empty board as seen in figure 2. The second test involved gathering information about the amount of backtracks in multiple non deterministic solutions as bases. The second test was then repeated for the case of a guess choice array that is ordered in reverse; the results were the same for both ordered arrays, so only one is represented in the results and analysis.

The counting of recursions and backtracks were done in the algorithm itself, and are shown in the pseudo-code in Figure 3. Recursions are counted on every iteration of the *solve()* method, which also includes a backtrack; the pure recursion count would be the recursion count that results from the operation minus the backtracks. However, in the best case scenario for an empty board, there will be as many recursions as there are empty cells, and any additional recursions will occur only because of backtracks. Each backtrack will result in at least one recursion. The two counters are related, and the performance is directly affect by the number of backtracks in each case.

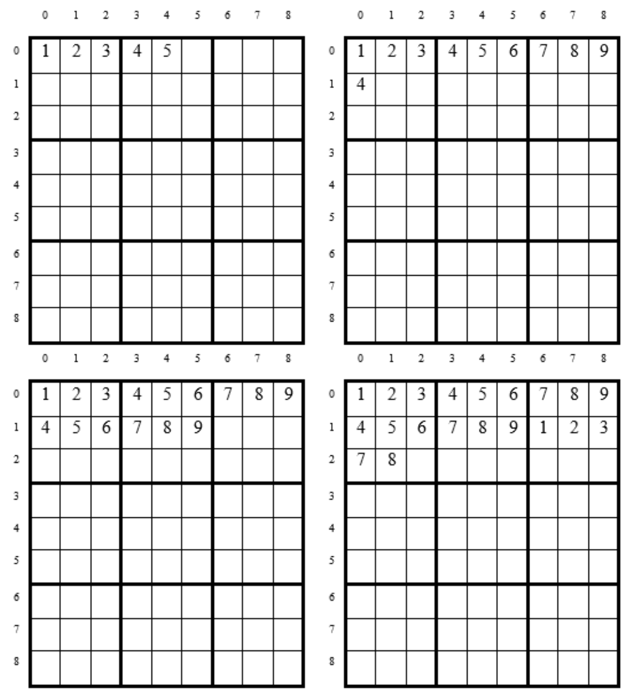

Figure 4. Initial board configuration using the deterministic solution as a base, with 5, 10, 15 and 20 prefilled numbers.

For that reason, while both numbers were counted and collected, the graphs and data focused on the number of backtracks alone, as a focus to the behavior in general.

A. *Testing the Deterministic Solution as Basis*

The first examination of the effect of guess choices was performed using the deterministic solution to an empty board as a basis. The full solution is seen in Figure 2.

Both guess choices were tested with boards that varied in the amount of prefilled numbers. The boards started empty, then were tested against boards with 5 prefilled numbers taken from the base solution and filled into sequential empty cells on the board. Each configuration was then solved the data about recursions and backtracks saved for analysis.

For consistency, the initial conditions included prefilled numbers from the base solution, filling the board sequentially from the top-left of the board. Figure 4 shows four initial configurations based on the deterministic solution with 5, 10, 15 and 20 prefilled numbers as an example. Each board was then solved first with ordered and then with shuffled guess choices.

When ordered guess choices were used, only one solution was obtained to collect the number of recursions and backtracks performed; there was no need to test more than once, since that method in that case is deterministic, and produces the same result for every initial case.

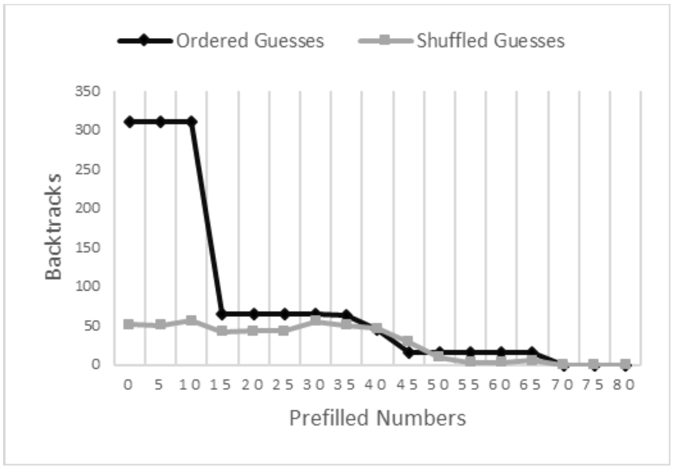

Figure 5. Amount of backtracks per prefilled values using the deterministic solution as base (first test)

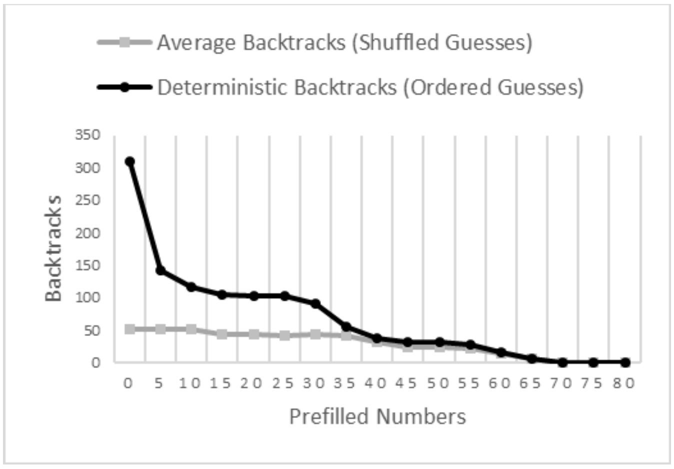

Figure 7. Amount of backtracks per prefilled values averaged over 195 base solutions (second test)

For the case of shuffled guesses, each board configuration was solved 10,000 times, and the average recursions and backtracks was obtained for each configuration. The number of attempts was purposefully large to make sure the standard error is in acceptable levels.

B. Gathering Data over Multiple Solution Bases

In order to verify that the observed results weren't only valid for the deterministic solution, another test was performed to include various random solutions.

First, the solving algorithm was run with shuffled guess choices on an empty board 195 times, each time producing random base solutions. These solutions were stored in an indexed object and were referred to directly for each iteration of the test.

Then, the method from the first test was repeated with the same configuration of prefilled numbers, this time taken from each of the 195 boards. For each base solution and each configuration the deterministic solver ran once to gather backtrack and recursions data, and the non-deterministic solver ran 1,000 times to get the average backtrack and recursion data.

While the number of attempts for each non deterministic solver is 10 times fewer than in the first test, that number was sufficient to produce results that could be shown to represent an overall trend, as verified with calculations of standard deviation and standard error.

V. Results

The first test results showed a clear difference in performance as represented by the amount of backtracks required between ordered and shuffled guess choices. Figure 5 shows the results of the number of backtracks required to solve a board with different configuration of prefilled numbers for ordered and shuffled guesses. The difference is especially notable for boards with up to 15 prefilled numbers, and becomes roughly equal in both methods for boards with 30 prefilled numbers and above.

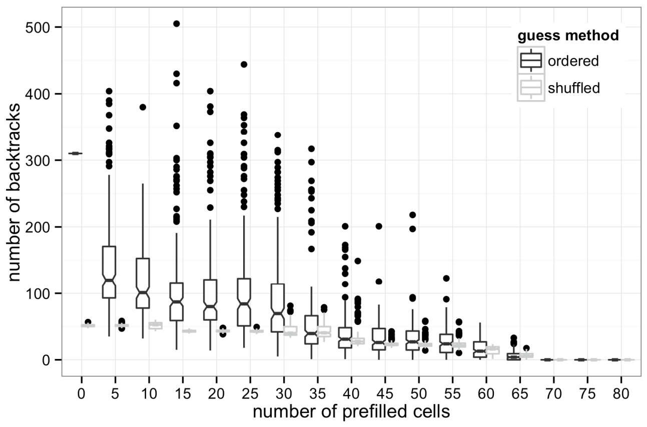

Figure 6. The distribution of backtracks across all 195 solutions and prefilled number configurations for both ordered and shuffled methods

This showed a sufficiently significant difference to perform further tests with more solutions as bases. The second test was performed over 195 base solutions and shows the trend continues to hold. Figure 6 shows the average number of backtracks over the 195 solutions for shuffled and ordered guesses, and demonstrates the same general effect; the algorithm requires significantly fewer backtracks on average for up to 35 prefilled numbers, after which the two methods are roughly equal.

VI. Analysis

When the initial idea of testing the behavior of the backtracking Sudoku solver came up the preliminary result seemed to be counter-intuitive. Shuffling the guess choice array is done for each empty cell test and each recursion. While the guess choice array is small (up to 9 total values) the meaning was still adding another loop to each recursion; even for the base case of solving an empty board this means potentially adding another loop for over 300 recursions. Intuitively and on the surface, shuffling the guess array seems like it should decrease the efficiency of the algorithm.

However, the results showed not only that the process runs with fewer recursions and requires fewer backtracks, but that the difference is significant. The question of whether this was only true for an empty board – or even if this was only true for a single type of solution – was handled with further tests, and the results show a clear trend of improved behavior using shuffled guess choices.

After closer examination of the results, however, this effect can be explained intuitively. The deterministic approach uses low numbers as it works forward on the board, which leaves the solver with a narrower range of available numbers as the solution progresses. This approach risks having no options for certain cells and means having to backtrack and replace values to increase the range of options on later cells.

In contrast, the non-deterministic shuffled approach means having wider distribution for available numbers in all cells in the board, which has the potential of obtaining better solutions with less need to backtrack and change guesses.

Further, the shuffled approach has a bigger chance of accidentally picking the most optimized solution for the entire board in each given cell. It may theoretically, by chance alone, solve certain percentage of the board in a manner that requires no corrections. Overall, then, using the shuffled approach has a much better potential of producing solutions with fewer backtracks.

This is best seen by examining the distribution of backtracks needed with ordered versus shuffled choices across all 195 base solutions, as graphed in Figure 7. It's fairly clear that ordered solutions produce a bigger distribution of backtracks, and could result with a relatively high number of backtracks. The shuffled solutions, however, perform much better in general, with their range of backtracks smaller and on a lower range than that of ordered guesses.

VII. Potential Issues and Future Work

The tests show a clear difference between using ordered and shuffled guess choices. However, these tests are not without limits that should be acknowledged.

In the tests performed for this paper the purpose was to make sure there is consistency when testing both methods on a simple implementation of the algorithm, which is why prefilled numbers were applied into consecutive cells on the board. However, further tests should probably be done on boards that include prefilled numbers that are randomly distributed on the board to test whether this has any effect on the difference in performance.

Another useful question to consider is whether 195 board configuration is enough to conclude the difference is valid for all Sudoku boards. This is especially important considering the fact there can be billions of possible Sudoku boards. However, it should be noted that the 195 base solutions were created using a properly shuffled choice array, which produces a randomized solution. Moreover, the difference in performance shown in the data is quite significant, and the calculation of standard deviation and standard error are within acceptable range. While it would likely be best to continue testing to more boards, the general behavior is likely to stay the same.

Other potential tests would involve changing the methodology of solving the boards. For instance, it would be interesting to check if the effect of guess choices remains evident after optimizing the backtracking algorithm with methods such as using intelligent backtracking [6] that picks certain empty cells intelligently rather than sequentially. This test can show if the difference between guess choices hold for backtracking methods that are more complex.

Finally, the test shows that randomness clearly affects the performance of the backtracking algorithm as it applies to Sudoku, but it may imply that the effect expands to more applications of the algorithm itself. More work in testing this algorithm under different uses could show whether this is a general optimization consideration or simply one related to specific puzzles.

VIII. Conclusion

While the usual candidates for optimization, when referring to the backtracking algorithm, are elements of its implementation as a brute-force search method, there seem to be relevant and significant influence to the choice of ordered versus random guesses. Using the shuffled guess array produces a significantly better performance for the algorithm than using an ordered array for guess choices; this effect should be taken into consideration when discussing optimization to the Sudoku backtracking algorithm.

It should be noted that this paper is not the first to consider the effect of randomness on solving Sudoku puzzles. Papers like [4], [7] and [9] examine the effect of stochastic approaches when solving Sudoku games. The explanation offered in [4] can also help explain the results in this paper, namely that using randomness as a factor, difficult puzzles can be solved as efficiently as easy ones, because the methodology works by going over random choices – either choice of empty cells, or guesses, or general approach to the brute-force algorithm.

While the explanation of the phenomenon should be expanded, the effect is fairly clear: randomized guess choices are much more efficient, on average, in solving Sudoku puzzles than ordered ones, and should be considered as further optimization factor.

References

[1] M. Schottlender. "Designing a Javascript Sudoku puzzle: an adventure in algorithms." *SmarterThatnThat.com*. 1 Feb 2014. Web. 5 Mar 2014. <http://moriel.smarterthanthat.com/tips/javascript-sudoku-backtracking-algorithm/ >.

[2] The111. "Answer 1: Recursively solving a Sudoku puzzle using backtracking theoretically." *StackOverflow*. 11 Aug 2013. Web. 5 Mar 2014. <http://stackoverflow.com/questions/18168503/recursively-solving-a-sudoku-puzzle-using-backtracking-theoretically>

[3] R Fisher, F. Yates. "*Statistical tables for biological, agricultural and medical research*" (3rd ed.). London: Oliver & Boyd. pp. 26–27. OCLC 14222135

[4] M. Perez and T. Marwala, "Stochastic optimization approaches for solving Sudoku," *School of Electrical & Information Engineering,*

University of the Witwatersrand. 13 pages; 6 May 2008. doi:10.1016/j.eswa.2012.04.019. <http://arxiv.org/abs/0805.0697>

[5] B. Felgenhauer, F. Jarvis, "Enumerating possible Sudoku grids," unpublished. <http://members.home.nl/jfhm-bours/Dutch/Projecten/Sudoku/sudoku.pdf>

[6] R. Korf, “Artificial intelligence search algorithms,” *Algorithms and theory of computation handbook* Page 22-17 Chapman & Hall/CRC 2010, ISBN 978-1-58488-820-8

[7] R. Lewis, “On the combination of constraint programming and stochastic search: the Sudoku case,” Hybrid Metaheuristics, Lecture Notes in Computer Science, Vol 4771, 2007, 96-107 <http://link.springer.com/chapter/10.1007/978-3-540-75514-2_8>

[8] T. Yato, T. Seta, “Complexity and completeness of finding another solution and its application to puzzles,” IEICE Trans. Fundamentals, Vol E86-A, No 5, p 1052-1060. 2008 http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf

[9] R. Lewis, “*Metaheuristics can solve sudoku puzzles,*” Journal of Heuristics Archive, 13(4), 387-401

[10] A. Majumder, A. Kumar, N. Das and N. Chakraborty, "The game of Sudoku-advanced backtrack approach," *IJCSNS International Journal of Computer Science and Network Security*, VOL.10 No.8, August 2010 <http://paper.ijcsns.org/07_book/201008/20100839.pdf>