Sudoku Solver

Benjamin Longbons and **Eric Klinginsmith** and **Sebastian Sánchez**
Washington State University Vancouver
14204 NE Salmon Creek Ave.
Vancouver, WA 98686

Abstract

This document explores the performance of two separate implementations of Sudoku puzzle agent and the resulting evaluation of the implementations. Each implementation is a distinct search method to solve any Sudoku puzzle. They will be evaluated based on how fast each solves a given set of puzzles. It was our goal to explore these methods to ultimately see how well they worked.

Introduction

A Sudoku puzzle like the one in Figure 1 is any n^2 by n^2 grid subdivided into n^2, n by n squares. In the typical case n is equal to 3 which makes the puzzle a 9 by 9 grid with 9 squares, each square being its own 3 by 3 grid. In any case, to solve the puzzle, distinct numbers 1 through n^2 must be filled into every row and column of the grid, so that no two numbers are in the same row, column, or square. Our implementations solve any Sudoku puzzle with an $n \leq 7$. Since the beginning, Sudoku puzzles have challenged many people with there simple concepts, yet complex possibilities. It was our vision to come up with a partial state solution to the common Sudoku puzzle that not only solved puzzles, but solved them in a reasonable amount of time.

Terminology

- *Order*: The most basic measure of a puzzle's size, labeled n.
- *Grid*: A square 2-dimensional array of Cells. Both dimensions of the array are n^2.
- *Cell*: A single place that can store a number. In a partial state, it may also be blank.
- *Row*: A horizontal slice of a Grid. It must contain all the numbers 1 to n^2 exactly once.
- *Column*: A vertical slice of a Grid. It must contain all the numbers 1 to n^2 exactly once.
- *Square*: An aligned $n \times n$ slice of a Grid. It must contain all the numbers 1 to n^2 exactly once.

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

Figure 1: Sudoku Puzzle

- *Neighbor* of a Grid: a Grid that differs by adding a value to a single Cell that previously had no values. The core function in all solvers is to iterate over all the Neighbors of a Grid at a certain Cell. For convenience, the Neighbors of a Grid at a Cell that already has a value is sometimes defined as the same Grid.

Implementations

BrutForce, BrutForce2.0, Brains

BrutForce The BrutForce implementation was chosen as a baseline, and simply iterates over every cell, and recursively checks all Neighbors at that Cell. It performs terribly when the first Cells visited are relatively unconstrained and it guesses wrong, but this implementation *was* useful to demonstrate the completeness of the general recursive approach.

BrutForce2.0 The BrutForce2.0 Implementation was a deliberately naive attempt to improve on the efficiency of the BrutForce, particularly for the worst cases, by statically ordering the Cells according to their initial neighbor count. Surprisingly, it actually performed worse in the few tests given to it, though as both implementations are very slow, it cannot be ruled out that the difference is just noise. This im-

plementation also changed the method slightly to skip over already filled cells.

Brains The *Brains* Implementation was a serious attempt to continue doing the best thing as the puzzle iteratively got closer to solved. It works by calculating, at each step, the number of neighbors at each yet-unfilled cell, and choosing the one with the least number of possibilities. Using this, it achieves pretty good performance for most puzzles. However, there is still a lot of variation, so it is not yet fully optimized.

Possible Improvements The *Brains* approach still suffers from two major flaws:

- It recalculates the list of best nodes every step. It would be better to remember a list of most-constrained Cells and update it.
- It always calculates possible neighbors from 1 to 9 in 3×3 puzzle. This will behave badly if larger numbers are more constrained.

Combined, these indicate the reason that certain puzzles take much more time to solve with this solver compared to the next.

SearchNxN Implementation

The *SearchNxN* Implementation searches for the solution by filling in one square at a time. To do this, *SearchNxN* first selects one of the squares to fill in. Once selected, *SearchNxN* will generate all possible solutions for the given square. These solutions are added to a stack. Then for every element on the stack *SearchNxN* selects the next square to fill in. This process is repeated until either all squares are filled in, or the stack is empty. If all squares are filled in, then the solution is printed to stdout. Otherwise, there is no solution and *None* is returned. The inspiration behind this method was taken from the common computer science problem of a three coloring. A three coloring is a problem in which when given a map with n regions to color an agent finds a way to color the entire map with only three colors so that no two regions have the same two colors. If we think of the Sudoku puzzle as being a map where no two numbers can be in the same row, column, or square and each square is a region, then a so called three coloring of a Sudoku puzzle would be to "color" each square so that there are no violations where a color would be one of the possible solutions to that square.

About choosing a the next square We would like to now talk about how *SearchNxN* selects the next square to fill in. There were two types of selection methods within this implementation. These are:

1. inOrder
2. mostAdjacent

inOrder inOrder is the simplest implementation of the two. It simply fills in each square one at a time left to right top to bottom. Although this method is simple, the time complexity to get the next square is constant.

mostAdjacent

mostAdjacent is a little more complicated. It chooses the square that has the most unfilled squares in the same row or column as that square. For example, in Figure 2 if we assume square 1, 2, and 8 are filled in, then square 1 has 3 unfilled in squares in the same row and column has it, square 4 has 4 unfilled in squares, square 5 would have 2, and so on and so forth. This method was developed to try to eliminate the branching factor of future squares. The idea was that the square with the most unfilled squares on the same row and column of it would limit the most possibilities for future squares. This supposedly would make *SearchNxN* faster than the inOrder method. To find out if this was true, we ran some early tests and found that this was not true at all. It turns out that mostAdjacent took 30 minutes to solve v.s. inOrder which only took 2 minutes. This led us to scrap the mostAdjacent method and go with inOrder.

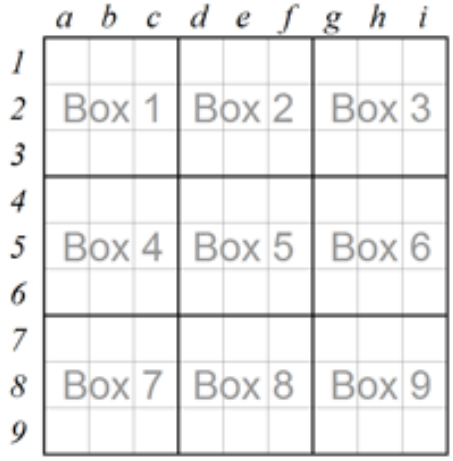

Figure 2: Numbered Sudoku Puzzle

About sole candidates We would like to mention one other part to our implementation. This is the idea of sole candidates. In a Sudoku puzzle, there are sometimes cells where only one number can be placed there. This means that any partially solved puzzle can have one or more sole candidates. If a sole candidate is found, then our implementation will try to find another. This continues until either all the sole candidates have been found, or it hits a cell where there are no possibilities. If the result is the latter, then that branch of the search is invalid and therefore the search must then try a different path to find the solution. If there are no other branches to explore and a solution has not been found, then there is no solution to the given puzzle.

Possible improvements There are several things that could be done to improve the *SearchNxN* implementation. These are:

1. Find a way to select a the next square to fill in that is faster than the inOrder method.
2. Implement an arc consistency check beyond the sole candidates.

3. Work on getting parts of the existing code to be more efficient by making critical functions less redundant.

The above ideas are all things that could make our implementation go from okay to great.

Strengths and Weaknesses of both Strategies

- Strengths
  - Compleate and optimal
  - For SearchNxN, sole candidates function makes strategy run faster by finding constraints
- Weaknesses
  - Recursive functions slow down computation.
  - The solvers are slow when solving Sudoku puzzles of order greater than 3

Experiments

Benchmarks were performed on only two strategies due to computational resource constraints. The strategies used to collect data were *SearchNxN* and *Brains*. The data collected was the time each strategy took to solve a Sudoku puzzle. The timer started before the search function in each strategy, and it stopped after the puzzle was solved. About 1300 3×3 Sudoku puzzles were solved using the strategies mentioned above, outliers were eliminated from the data by taking the top 90% of the data set sorted from lowest to highest and were run using a computer with the following specifications.

Intel(R) Core(TM) i7-2635QM CPU @ 2.00GHz
Memory Ram 8 GBs
Linux version 2.6.32-279.19.1.el6.x86_64

SearchNxN Performance

Now let's look at the performance of the *SearchNxN* implementation.There are several different ways we measured the performance of the *SearchNxN* implementation. The way we choose to look at it was the more puzzles that solved within a certain amount of time the better. Figure 3 shows what percentage of puzzles this method solved in 5 second time intervales.

This shows that about 62% of the puzzles were solved within 5 seconds, and about 30% of the puzzles were solved from 5 to 10 seconds. This graph also includes the outliers, which are on the far right of the graph. The most extreme outlier holds a time of around 240 seconds and takes up less than 1% of the total puzzles.

In addition this data was tabulated to show statically our results if Table 1.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	2.272	3.909	6.211	7.016	240.700

Table 1: Statistical data for SearchNxN

However, looking at the data with the outliers is not quite representing the normal case. Therefore, we also made a histogram for the top 90% of the data. This resulted in Figure 4.

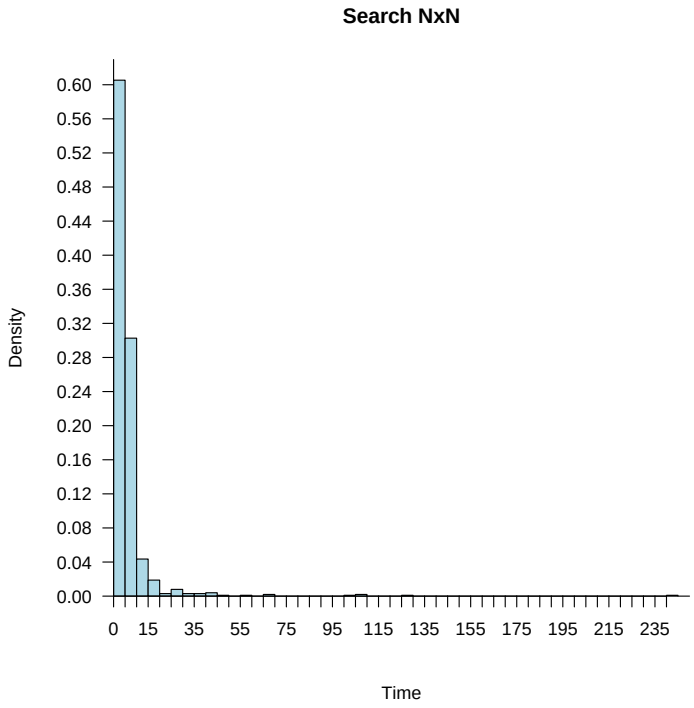

Figure 3: Histogram of SearchNxN method

As one can see about 20% of these puzzles were solved between 1 and 2 seconds, and there are no puzzles that took longer than 14 seconds. This data shows that when we ignore the outliers, most of the puzzles solve in a reasonable amount of time. In addition, statistical data has been provided in Table 2.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	2.298	3.870	4.705	6.747	13.830

Table 2: Statistical data of SearchNxN top 90%

As one can see the data supports the histogram by showing that the mean is right around 5 seconds and the maximum is right around 14.

Brains Performance

Brains is a recursive strategy where it iterates through the whole Sudoku puzzle each time the recursive function is called, which adds overhead. The fastest time this strategy solves a puzzle is one second because finding the number of possible combinations for each cell is a very expensive operation. The data distribution has an inverse exponential shape as you can see in Figure 6. In addition, about 20% of our sample was solved in 1 to 2 seconds.

Challenges

We experienced many challenges along our journey to solve Sudoku puzzle. The first was to come up with a reasonable

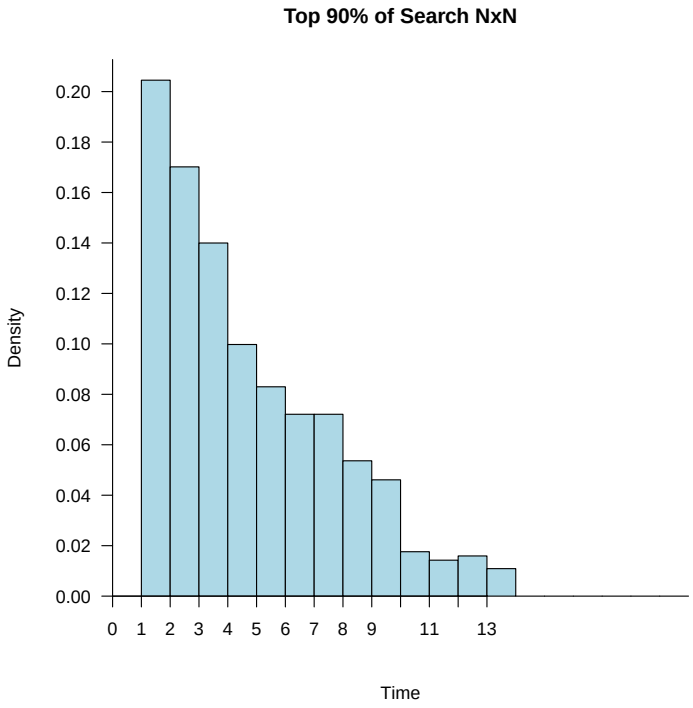

Figure 4: Histogram of SearchNxN method of the top 90%

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.001	2.251	4.334	5.125	7.112	16.380

Figure 5: Descriptive Statistics for Top 90% of Brains

way to represent the states of the board. We thought of a data structure that formed the board out of a set of cached cells. This data structure also included the ability to check what possibilities could be filled into any cell.

The next challenge was to come up with possible ways of solving the problem. The decision was made early on to do a semi-brut force recursive method and also to develop an original solution. These solutions where later named *Brains* and *SearchNxN* respectively. From here it was just a matter of working through the smaller challenges of improving the speed of each solution as they evolved.

Conclusions

Doing a partial state search is very expresive on performance because each time a cell is filled in, constraints have to be checked. In addition, performance is limited by Python because it is an interpreted language, and our brench marks are specific to the hardware used for this project. There are many ways to solve a Sudoku puzzle using search methods. Our approaches use depth-first search because we know the depth-limit for each strategy. For example, in a 3×3, *SearchNxN*'s depth-limit is 9, and *Brains*'s depth-limit is 81. Also, both strategies are optimal and complete.

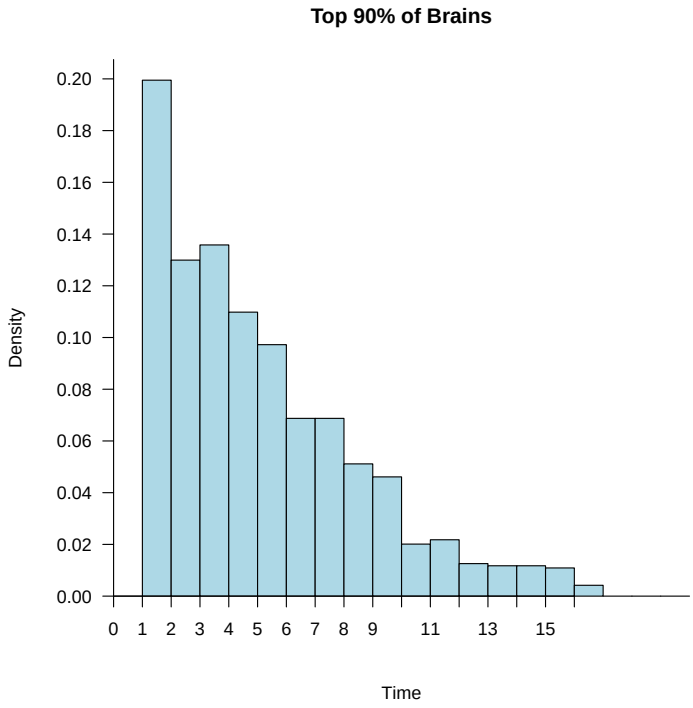

Figure 6: Top 90% of Brains

Work still remains to be done in both search strategies. *SearchNxN* needs a more efficient way to select the next square in the puzzle to fill in, and adding an arc consistency to this strategy will increase performance. Moreover, *Brains* needs to choose a better path to avoid recalculating the list of best nodes. This will reduce computation time. Both search strategies use recursion to fill in cells, and we believe this is why our data reveal that both strategies have about the same performance, *SearchNxN* being the best. In addition, We concluded there is not much statistical significance between the strategies, but *SearchNxN* has the biggest potential for improvement given it is not based upon a brut-force method.