Algorithms for Solving the Game of Sudoku

Caitlyn Kloeckl

May 10th, 2021

Abstract

This paper addresses solving various games of Sudoku of the standard 9x9 size Sudoku grid. The algorithms focused on in this paper include framing Sudoku as a constraint satisfaction problem to use recursive backtracking along with using the genetic algorithm. Solving Sudoku as a constraint satisfaction problem was shown to be greatly outperform the genetic algorithm when comparing solving time.

1 Introduction

The Sudoku game consists of a single 9x9 grid. The 9x9 grid also consists of 9 3x3 sub-grids that have bolded outlines. The rules consist of the following: each row, each column, and each subgrid can only use each number 1-9 once. Each game is begins with some given initial numbers in their respective squares–as can be seen on the left side of Figure 1.

The goal of Sudoku is to fill out the entire grid while abiding by these rules. A completed game is depicted on the right side of Figure 1 with the numbers that had to be found in grey. The rules of the game and the sheer number of ways a Sudoku grid can be filled in makes this a difficult computing problem. With around 10^{21} Sudoku solutions, using any basic search algorithm including depth first search or breadth first search is ineffective and untimely. This paper explores alternative algorithms to solve Sudoku grids efficiently.

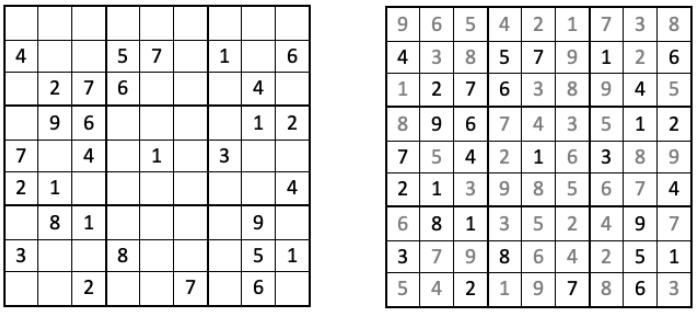

Figure 1: An example game of Sudoku with a starting grid (left) and a completed grid (right). The game's goal is to successfully fill in the grey numbers. Figure made in Microsoft Word.

2 Literature Review

2.1 The Game of Sudoku

Sudoku is a game played on a 9x9 size grid and Sudoku puzzles start with around 20 given values filled in. Sudoku is a challenging and unique puzzle to solve using computer algorithms. Although Sudoku is static, the unique rules about where the numbers can be placed gives the game its complexity. The puzzle becomes increasingly difficult when fewer initial values are given.

The main algorithms to be explored are the genetic algorithm and the constraint satisfaction algorithm. Other previously applied algorithms are also discussed as follows.

2.2 Alternative Algorithms

A few algorithms have been applied to this problem. Some have been successfully applied and others have not. Dhanya Job and Varghese Paul applied recursive backtracking to the Sudoku game in 2016 [2]. The backtracking algorithm works by trying actions until reaching a dead end or solution. If it reaches

a dead end it will go back an action and try the other options available at that point. Dhanya and Varghese found that the recursive backtracking algorithm did solve the puzzles with 17 to 22 initial values given. The time it took to solve these puzzles ranged from 18-23 seconds [2].

Another algorithm explored was the simulated annealing algorithm. The researchers found this algorithm to only occasionally find a solution and most of the time the algorithm would reach t-max and terminate early. When this algorithm found a solution, it would still take up to 500 seconds [5]. Since the algorithm could not find solutions most of the time, this algorithm was found to be very ineffective.

2.3 The Genetic Algorithm

A main focus of this paper, the genetic algorithm, is inspired by evolution and includes crossover, mutation, and selection in order to 'evolve' and find a solution [3]. While keeping the initial values in the original places, the genetic algorithm applies swapping of rows/columns (crossover), random number changes (mutation), then choosing the best boards after some of those actions were applied (selection). Yuji Sato and Hazuki Inoue found this algorithm to be very fruitful and would find a solution every time [7]. This paper ([7]) did not include the solving time, only the number of generations created (up to 100,000) until the solution was found.

Xiu Qin Deng and Yong Da Li took a creative approach to the genetic algorithm and applied small searches and careful evaluation of each board before applying crossovers, mutations or selections. By doing so, they greatly decreased the solving time. This algorithm found a solution in an average of 1.75 seconds [1].

2.4 The Constraint Satisfaction Algorithm

The constraint satisfaction algorithm is framed as a problem that has a a finite set of variables, which are to be assigned values, that must meet constraints [9]. Applying this to the game of Sudoku, the variables are the empty squares, the values are a range of 1-9, and the constraints are the rules of the columns, rows, and subsquares.

Helmut Simonis found this algorithm to be quite successful when applied to Sudoku. With varying levels of puzzle difficulty, the algorithm Helmut Simonis applied took from 40-741ms (0.04-0.741 sec) [8]. This greatly outperforms the previously mentioned algorithms.

Marlos C. Machado and Luiz Chaimowicz took a creative approach to the constraint satisfaction algorithm by combining it with metaheuristics and simulated annealing. They found the constraint satisfaction algorithm by itself does not always find a solution, but when combined with simulated annealing it finds more solutions and they argue take less solving time. For the standard 9*9 board size, this paper found it took up to 1.2 seconds to solve the board [4].

While [4] found a decrease in solution time based on their base case, [8] found faster solutions by only using constraints. This may be a discrepancy on how the algorithms were programmed or the device the algorithms were run on.

The following paper explores a direct comparison of the genetic algorithm and the constraint satisfaction algorithm in the same environment and similar coding styles to reduce discrepancies.

3 Experimental Design and Algorithms

As mentioned earlier, the main algorithms are analyzed include the genetic algorithm and the constraint satisfaction algorithm/recursive backtracking. The following sections describe each algorithm and how they are applied to the Sudoku game.

3.1 The Genetic Algorithm

The genetic algorithm was inspired by natural selection. The pseudocode is shown below from [6]:

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 2: The pseudo code from [6] depicting the genetic algorithm.

There are a few unique functions that the genetic algorithm implements. First, this algorithm has a population. For this implementation of Sudoku, a population consists of 100 boards. These boards are initially created by randomly filling out the empty spaces of the board for each of the 100 'candidates' in the population.

Each of these candidates then has a 'fitness' level. This just measures how close a candidate is to a solution. In this implementation, the fitness level ranges from 0 to 1.0 (0% to 100%) for each candidate. A fitness level is calculated by going through each row, column, and subgrid. The formula for a total *row* fitness is summarized as follows:

$$\text{fitness}_{rows} = \frac{\frac{\text{row1 unique \#'s}}{9} + \frac{\text{row2 unique \#'s}}{9} + \ldots + \frac{\text{row9 unique \#'s}}{9}}{9}$$

Where the equation above would also be applied to the columns and subgrids. Then a 1.0 *total* fitness level means the total fitness formula below equals 1:

$$\text{total fitness} = \text{fitness}_{rows} * \text{fitness}_{columns} * \text{fitness}_{subgrgids}$$

Otherwise, when not equal to 1, the worse the fitness level of the rows, columns, and subgrids, the worse overall fitness for the candidate.

The genetic algorithm then uses the candidates generated, along with their calculated fitness levels and performs a few actions. The first action is applying crossovers. In this implementation, the algorithm picks 2 candidates, and break up the board vertically, and swaps halves. This crossover is only applied to the 2 fittest candidates within a generation. From this swap the algorithm keeps both sides of the swap (these are called child 1 and child 2) and both move onto the next generation and 'survive'.

The genetic algorithm then mutates one of the children from the crossover. The mutation is implemented by randomly picking a square in the grid and then changing it to a new random number in the range 1-9. This whole process is repeated (after creating the initial population); locating the 2 fittest, crossover of the 2 fittest, mutation, creating population but including the previous 2 fittest.

3.2 The Constraint Satisfaction Algorithm

The constraint satisfaction algorithms consists of domains, variables, and constraints. In the application to Sudoku, the variables are the 81 squares each with a domain 1-9. The constraints are the rules of the game. When CSP's are applied to problems, many can be solved purely with inference. Inference is using the constraints (rules) to reduce the number of possible values for each variable. However, when applying this Sudoku, often there are still many possible values left for each variable. Because of this, recursive backtracking–a search algorithm–was also implemented. The backtracking algorithm pseudo code is shown below from [6]:

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 3: The pseudo code from [6] depicting the backtracking algorithm pseudocode.

The backtracking algorithm works by, after applying the constraints, picking one of the Sudoku squares with the smallest remaining domain and does a depth first search, randomly filling out the other squares. Between each variable being randomly assigned, the algorithm applies/checks the constraints ensuring that the selection is valid, further speeding up the solution time.

Results

For the experiment, a board from https://www.7sudoku.com/difficult that was labeled as 'Very Difficult' was initially solved. Numbers from this solved board were then incrementally, and randomly, removed. This created boards with 80, 70, ..., 40, 30 initial known values. Each of these boards were then input into each the CSP and Genetic algorithms and run for 20 trials each. The solving times for these runs are shown below.

Number of Initial Known Values	CSP Solving Time (sec)	Genetic Solving Time (sec)
80	0.0100	0.0430
70	0.0101	0.0442
60	0.0098	0.0591
50	0.0104	0.2246
40	0.0133	4.3157
30	0.02840	N/A (timed out)

Table 1: Solution time for varying initial values for both the constraint satisfaction algorithm and the genetic algorithm.

4 Analysis

The results from Table 1 are plotted in Figure 4 below. As labeled, the constraint algorithm results are shown in blue and the genetic algorithm results are shown in red. From this, it is clear that as the board becomes more difficult (fewer inital values) the CSP algorithm clearly outperforms the genetic algorithm.

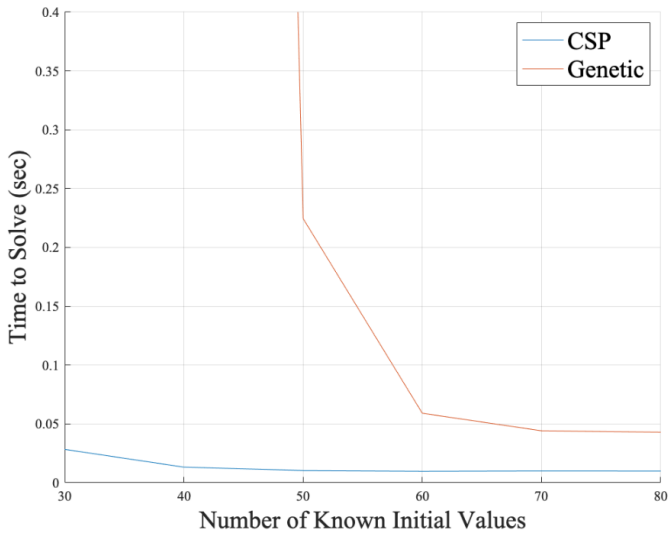

Figure 4: Results from Table 1 plotted in MATLAB.

5 Conclusion

In conclusion, both the genetic algorithm and the constraint satisfaction algorithm give unique perspectives on solving the game of Sudoku. While results show the constraint satisfaction algorithm solved Sudoku much quicker, the genetic algorithm still has a unique solving technique that was shown to be successful with 50+ known initial values.

The genetic algorithm also has room for improvement: setting the mutation rate, the 'fitness' level calculation/alternate formulas, and other population generation techniques. All of these improvements could greatly increase solution time but were out of the scope of this paper. The CSP algorithm has already shown to a have very fast solution time for very difficult problems. For other problem besides Sudoku, the CSP algorithm may also prove to be very successful.

Works Cited

References

[1] X. Q. Deng and Y. D. Li. A novel hybrid genetic algorithm for solving sudoku puzzles. *Optimization Letters*, 7(2):241–257, 2013.

[2] D. Job and V. Paul. Recursive backtracking for solving 9* 9 sudoku puzzle. *Bonfring International Journal of Data Mining*, 6(1):07–09, 2016.

[3] O. Kramer. *Genetic algorithm essentials*. Studies in computational intelligence ; v. 679. Springer, Cham, Switzerland, 2017.

[4] M. C. Machado and L. Chaimowicz. Combining metaheuristics and csp algorithms to solve sudoku. In *2011 Brazilian Symposium on Games and Digital Entertainment*, pages 124–131. IEEE, 2011.

[5] P. Malakonakis, M. Smerdis, E. Sotiriades, and A. Dollas. An fpga-based sudoku solver based on simulated annealing methods. In *2009 International Conference on Field-Programmable Technology*, pages 522–525, 2009.

[6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[7] Y. Sato and H. Inoue. Solving sudoku with genetic operations that preserve building blocks. pages 23–29, 2010.

[8] H. Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer, 2005.

[9] S. Zivny. *The complexity of valued constraint satisfaction problems*. Cognitive technologies. Springer, Dordrecht, 2012.