Sudoku as a Constraint Satisfaction Problem: The Effect of Heuristics on Execution Time

Andy Donato and John Dukewich

The Pennsylvania State University
DS 497 Final Project

https://github.com/jdukewich/DS497Project/

May 1st, 2019

Contents

1 Introduction **2**

2 Background Research **3**

3 Experimentation **3**
3.1 Datasets 3
3.2 Problem Representation 4
3.3 Testing Methodology 5

4 Results **5**
4.1 Variable Selection Heuristics 5
4.2 Value Selection Heuristics 5
4.3 Execution Time Tables 6
4.4 Analysis 6

5 Acknowledgements **7**

6 References **7**

1 Introduction

Sudoku is a single-player logical puzzle game enjoyed by many as a leisurely brain exercise. Its formulation is quite simple - a player must complete a partially-filled grid of numbers from a subset of natural numbers such that each row, column, and sub-square contains a complete set of the choices. A sudoku board is of the dimensions $n^2 \times n^2, n \in Z^+$ that is populated with the positive integers $\{1, 2, ..., n^2\}$. The the most popular formulation of the game is a 9×9 grid filled with the numbers 1-9, though larger variants like hexadoku (16×16 choosing from hexadecimal numbers $\{1, 2, ..., F\}$) and alphadoku (25×25 choosing from English alphabetical characters $\{A, B, ..., Y\}$) exist for those looking for an extra challenge.

Sudoku players utilize various logical techniques of varying complexity to derive solutions to a puzzle. These techniques leverage the problem constraints to deduce a square's true value. For instance, one technique involves examining values a square *cannot* take on in the hopes of isolating its true value, while another examines recurrences of a value across the entire board in the hopes of deducing a square that *must* take on that value (Figure 1). Human sudoku-solving is entirely reliant on the player's ability to spot and apply these logical reduction techniques; the most difficult puzzles require increasingly obscure methods.

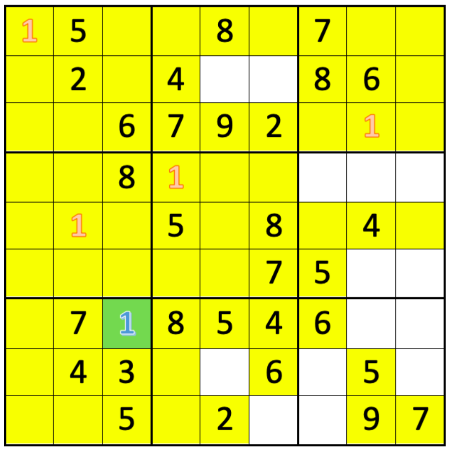


Figure 1: Example of a logical sudoku-solving technique. Squares highlighted in yellow cannot take the value of 1 (either by constraint violation or, more trivially, by pre-existing assignment). The green square (originally unassigned) must therefore be 1 because it is the only position left in it's sub-square that can take on that value.

A human player is limited by his own mind and the patterns he/she is able to keep track of and pick out. Computers, on the other hand, can bypass human logic and be programmed to solve a fair sudoku every time (that is, a puzzle with exactly one solution). There are numerous ways to algorithmically solve a sudoku puzzle. On the low end is the inglorious and time-consuming brute force method, guaranteed to find a a solution *eventually.* This technique is unfeasible for a sane human player due to the staggering combinatorics involved, and its time complexity overwhelms even the beefiest computers for larger puzzles ($n >> 3$). Elegant solutions involve Artificial Intelligence techniques to improve the efficiency of finding a solution. One such approach that this paper focuses on is modeling sudoku as a Constraint Satisfaction Problem (CSP). This class of problems is an attractive choice because the CSP has a standardized representation that can be fed into universal solvers that have been subjected to years of development and refinement.

Algorithmic solutions to the sudoku problem have been around for a long time, and it certainly does not challenge limits on the cutting edge of AI. So why study it? Sudoku can be characterized as a "toy" problem - though implementing it as a CSP is relatively straightforward, it can be used as a control when finding ways to improve existing techniques. This paper aims to find various tweaks to the CSP solver algorithm to more efficiently handle the sudoku problem.

2 Background Research

The methods for modeling and solving a CSP can be more easily understood through Russell and Norvig's descriptions in *Artificial Intelligence: A Modern Approach* [1]. A CSP consists of three components: a set of variables $\{X_1, ..., X_n\}$, a set of domains $\{D_1, ..., D_n\}$ for each variable, and a set of constraints that specify the allowable combinations of values for the variables. A sudoku is a simple example of a CSP, as the variables, domains, and constraints are easily seen and understood. For an $n^2 \times n^2$ sudoku, each cell on a sudoku board would be a variable, totaling n^4 variables. The domain for each variable will be the set $\{1, ..., n^2\}$. There will be $3n^2$ constraints, which are the *AllDiff* constraints for each row, column, and $n \times n$ sub-square, enforcing that no two variables in that constraint can have the same value. Therefore, a classic 9×9 sudoku would contain 81 variables, the domain for each cell would be the numbers {1, 2, ..., 9}, and there would be 27 *AllDiff* constraints.

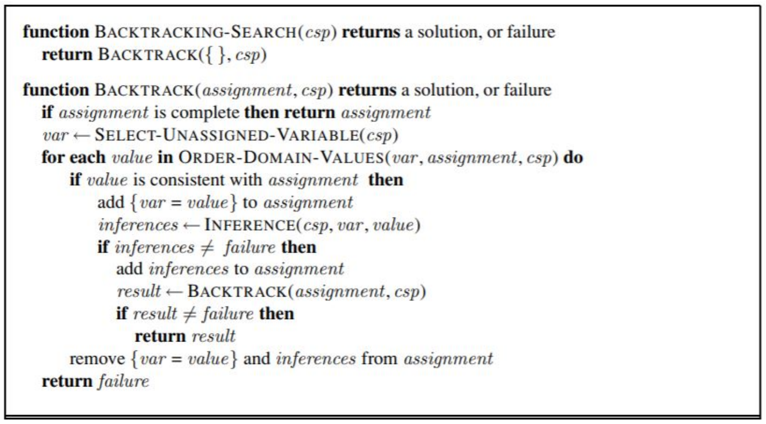
As opposed to a normal state-space search algorithm, CSPs allow for a choice between search and a type of inference known as constraint propagation. In constraint propagation, assigning one variable to a value reduces the allowable values for another variable, which reduces the values for another, and so on. CSP solvers often mix constraint propagation along with search, or even as a preprocessing step. Constraint propagation can be carried out through local consistency, which is often implemented through algorithms such as node or arc consistency, or even general k-consistency. However, the focus of this investigation is not on the well-established constraint propagation techniques, but instead the search methods used to solve the CSP.

There are numerous methods of search applied to solving CSPs, and many search methods make use of heuristics to improve their efficiency. This broad range of algorithms and heuristics warrants more research and investigation to see if some combination would be best for solving a sudoku. A paper by Machado and Chaimowicz [2] investigates the method of using a *simulated annealing* search combined with specifically designed heuristics. Another interesting method is laid out by Weber [3], modeling the sudoku as a satisfiability problem. This research experiment will investigate specifically the use of the recursive backtracking search to solve the sudoku, as seen in Figure 2.

3 Experimentation

3.1 Datasets

A considerable portion of this project was obtaining sudoku puzzles themselves to solve in a machine-readable format. We used several techniques to obtain such puzzles to feed our program, the most



```
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 2: Backtracking Search Algorithm taken from *Artificial Intelligence: A Modern Approach* [1]

straightforward of which involved utilizing the website https://qqwing.com/generate.html to generate a CSV of 9×9 sudoku puzzles. We used a unique sudoku generator at https://github.com/ngbaanh/unique-solution-sudoku-generator to individually build easy 4×4 puzzles, but we did not find any pre-built mass-production services for $n > 3$. We ended up building a webscraping script that procedurally made requests to a printable hexadoku site (https://www.sudoku-puzzles-online.com/hexadoku/print-hexadoku.php), extracted the puzzle from the HTML, and converted it into a standard base-10 16×16 sudoku. By extracting large numbers of different size puzzles, we hoped to study how the problem scales as the state space becomes larger, requiring more variables, domains, and constraints.

3.2 Problem Representation

Our vision for this project did not include re-inventing the wheel by toiling to reproduce an industry-standard CSP solver; instead, we decided to use an available open-source CSP solver as a baseline to build from. This work investigates the efficiency of various baseline CSP solvers as well as the effectiveness of different heuristics used in the CSP algorithm.

We vetted several Python libraries for constraint satisfaction, but we ultimately settled on one called *python-constraint*. The documentation and explanation of this library can be found at https://labix.org/python-constraint. The Github repository for *python-constraint* can be found at https://github.com/python-constraint/python-constraint. This library provides a `Problem()` class which encompasses the whole CSP. The class has methods such as `addVariable(variable, domain)` and `addConstraint(constraint, variables)` to build the CSP according to the problem definition. The `constraint` parameter is a object of type `Constraint`, an abstract base class representing enforced relationships between variables. For example, our implementation heavily utilizes the subclass `AllDifferentConstraint()`, which, as the name suggests, enforces that all the variables associated with an instance of this constraint must have different values.

Our formulation and code for this project can be seen at https://github.com/jdukewich/DS497Project/. To formulate an $n^2 \times n^2$ sudoku as a CSP to be solved by the *python-constraint* library, we procedurally added variables for each of the cells with the call `addVariable(i, [1, ..., ` n^2`])`, where i is the index/identifier of each variable ranging from $(0...n^2 - 1)$. Cells with pre-populated values from the puzzle definition are simply initiated with a single-element domain `[k]`, where k is the cell's value. To add the constraints, we needed to add the `AllDifferentConstraint()` for each row, column, and sub-square.

Theoretically, we could have simply represented the sudoku board as a 2D list. Setting constraints, however, requires logic for pulling out certain slices of the board at a time (row, column sub-square). This led us to design a lightweight class representation around the array to house all the board-based logic in one place. The class contains helper functions for extracting rows, columns, and sub-squares, updating values by coordinate (row, col) or index $(0...n^2 - 1)$, and checking board consistency. Our class implementation can be found in the GitHub repository linked above. After the CSP is built in the `Problem()` class with its associated variables and constraints, the process of solving the CSP is abstracted away using the library's `getSolution()` method. The solving algorithm used can be one of either `BacktrackingSolver()`, `RecursiveBackgrackingSolver()`, or `MinConflictsSolver()` by default.

The experimental component of this work revolves around the inner workings of the *python-constraint* library. We altered the implementation of the library's recursive backtracking solver. As written, the algorithm is quite similar to the implementation found in *Artificial Intelligence: A Modern Approach* [3]. Heuristics are used in two areas in this solver. The first guides the order unassigned variables are selected in to test. The second guides the order of value selection from a variable's domain. By default, *python-constraint* uses a mixture of the Degree and Minimum Remaining Value heuristics to select unassigned variables, and it does not use any heuristic for ordering the domain values.

3.3 Testing Methodology

We developed a standardized testing procedure as a means of quantifying the effectiveness of each heuristic. Program execution time served as the primary metric for comparison. Admittedly, execution time is by no means universal; it can vary wildly between machines and can fluctuate with other processes running on a system. With this in mind, we put together a standard set of test conditions that minimized test-to-test variations and enabled us to draw relative comparisons.

Each test measures the execution time for solving a set of multiple puzzles. In our initial experimentation, we found that most 9×9 and 16×16 puzzles could be solved in well under a second, so we opted to repeat the process many times to better illustrate the differences between heuristics. We prepared two sets of puzzles to be tested with each heuristic: fifty 9×9 puzzles and fifty 16×16 puzzles. We built a modular testing script (*.py file) executed in shell on the command prompt. The exact testing conditions (i.e. square size and heuristics used) are passed through command line arguments. The tests were carried out on a laptop running 64-bit Linux Ubuntu 18.04 with an Intel i5 processor and 8 GB of RAM. All other programs and non-essential processes were killed prior to the execution of each test. Each combination of heuristics and square size was run three times, with the ultimate execution time value taken as the average of the three.

4 Results

4.1 Variable Selection Heuristics

Six different heuristics were used to determine the selection of an unassigned variable. The implementation details can be seen in the Github repository listed earlier.

The first heuristic is simply no heuristic at all. The solver will just loop through the unordered list of variables and choose the first one that has not been assigned yet to test.

The second heuristic is the degree heuristic, which will order the list of variables in descending order of the number of constraints the variable is an element of.

The third heuristic is the minimum remaining values (MRV) heuristic which orders the list of variables in increasing number of legal remaining values in its domain. For example, a variable that can only be $\{2, 3\}$ will be tested before one that can be $\{1, 2, 4, 5\}$.

The fourth heuristic tested is randomization. The order of the variables is pseudo-randomized and the solver will iterate through that random order to pick the next unassigned variable to test.

The fifth heuristic is a combination of degree and MRV. This was the default heuristic used by the *python-constraint* library.

The sixth heuristic is a combination of MRV and randomization. This heuristic first sorts based upon MRV. Then, let x be the smallest remaining domain size other than 1. This heuristic will apply randomness to all variables with x remaining legal values. This will keep the idealogy of MRV but add in a bit of pseudo-randomness.

4.2 Value Selection Heuristics

Four different heuristics were used to determine the ordering of values to be tested for each variable. The implementation details can be seen in the Github repository listed earlier.

The first heuristic is once again no heuristic at all. The values are simply tested in the order $\{1, 2, ..., n^2\}$.

The second heuristic is randomization. The order of the values is pseudo-randomized and the solver will iterate through that random order to pick the next unassigned value to test.

The third heuristic is the least-constraining value (LCV). This heuristic gives preference to values that will rules out the fewest possible choices for other variables.

The fourth heuristic gives preference to the least used values. For example, if the value 1 has not been assigned to a variable yet, this heuristic would give preference to testing 1 before other values that have been used in assignments.

4.3 Execution Time Tables

Table 1: Average Execution Time (seconds) of 9x9 Sudoku Trials

	Variable Selection Heuristic					
Value Selection Heuristic	None	Degree	Random	MRV	Degree+MRV	MRV+Random
None	–	–	–	0.826	0.998	0.921
Random	–	–	–	0.963	1.145	1.124
Least Constraining Value	–	–	–	1.482	1.642	1.428
Least Used Value	–	–	–	1.100	1.222	1.019

Note: trials marked with "–" did not terminate within a reasonable amount of time (60 seconds).

Table 2: Average Execution Time (seconds) of 16x16 Sudoku Trials

	Variable Selection Heuristic					
Value Selection Heuristic	None	Degree	Random	MRV	Degree+MRV	MRV+Random
None	–	–	–	2.286	2.673	2.960
Random	–	–	–	2.322	2.638	2.961
Least Constraining Value	–	–	–	3.484	3.817	4.208
Least Used Value	–	–	–	2.546	2.860	3.219

Note: trials marked with "–" did not terminate within a reasonable amount of time (60 seconds).

4.4 Analysis

Our trials reveal just how important a heuristic is to the success of finding a CSP solution in a reasonable amount of time. Certain heuristics, like degree and random (variable) proved entirely ineffective at solving even a single problem in reasonable time, while others enabled the CSP to solve many puzzles in just a few seconds.

The sudoku problem proved to be much more sensitive to the variable heuristic used than the value heuristic. This observation aligned with our expectations - for an $n^2 \times n^2$ sudoku puzzle with n^4 variables and n^2 domains, there will always be n^2 times as many variables than domains. An effective variable heuristic will thus always have a greater impact than an effective value one since it cuts down on a relatively greater number of choices. That being said, we postulate that the effect of a value heuristic would begin to emerge for $n >> 4$ as execution times of testing every value in a domain become significant.

Problem-specific heuristics can provide a considerable advantage in execution time as opposed to the generalized heuristics many CSP solvers implement by default. As seen through our experimentation, the *python-constraint* library used a mixture of the degree and minimum remaining value heuristics for variable selection. However, for the sudoku, the inherent symmetry of the problem renders the degree heuristic unnecessary since each variable is subject to the same number of constraints. Therefore, by forgoing the degree heuristic and only using MRV, the execution time had a noticeable improvement due to eliminating the extraneous sorting by degree.

5 Acknowledgements

Being a group project, we ensured that both members of the group contributed in a meaningful manner.

John and Andy both contributed greatly to the completion of this report. The work on the report was approximately split equally, with both of us working through it together. John implemented the heuristics in code as well as altered the usage of the *python-constraint* library to allow for experimental investigation. John also helped in generating data to be used for the 9×9 and 16×16 puzzles. Andy wrote out the class representation of a sudoku board in a clean and efficient manner to allow for both easy manipulation and quick utilization. He laid out the original implementation of the CSP via *python-constraint* such that it worked with the class representation he built. He also designed the test framework and built the test scripts.

This statement serves as an electronic signature for myself, **John Dukewich**, to acknowledge that I agree with all contents held within this report, including the Acknowledgements section detailing my contributions to this project.

This statement serves as an electronic signature for myself, **Andy Donato**, to acknowledge that I agree with all contents held within this report, including the Acknowledgements section detailing my contributions to this project.

6 References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, Dec. 2009.

[2] M. Machado and L. Chaimowicz. (2011). "Combining Metaheuristics and CSP Algorithms to Solve Sudoku." *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*. 124-131. 10.1109/SBGAMES.2011.18.

[3] T. Weber, "A SAT-based Sudoku Solver," in *Proceedings of 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings,* G. Sutcliffe and A. Voronkov, Eds., Dec. 2005, pp. 11–15.