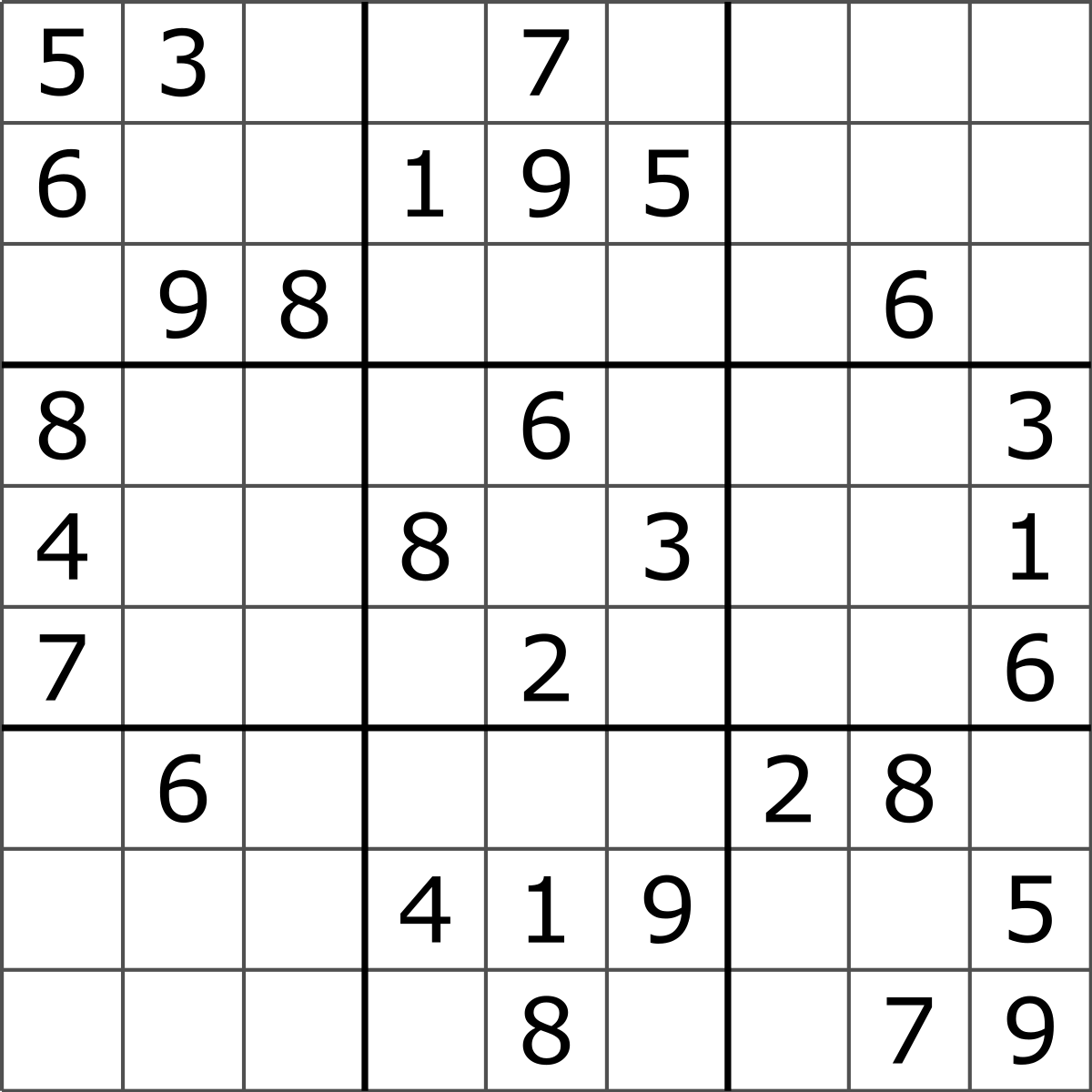
Independent Sudoku Solver Using Various Constraint Satisfaction Methods and Heuristics

Nicholas Montaquila

Abstract

The work described in this essay attempts to solve the problem of solving a Sudoku puzzle without any human assistance. The solution utilizes a recursive backtracking algorithm, which implements various constraint satisfaction techniques such as forward checking and arc consistency. The solution also attempts to leverage certain heuristics to better the algorithm’s performance. The result of this work is a program capable of reading in an incomplete Sudoku board, completing it successfully, and printing out the result.

Introduction *[[1]](#footnote-1)*

Sudoku is a number-placement puzzle game. The objective of the game is to fill a 9x9 grid with a number between 1 and 9 such that each row, column, and each of the nine 3x3 sub-grids (hence forth referred to as regions) contains all of the digits from 1 to 9 exactly once. Figure 1 shows an example Sudoku board.

*Figure 1*

While this problem is certainly not impossible for a human to solve, the large number of variables does make the process quite tedious and time consuming. Since computers can perform calculations at such a higher rate than the human brain, the size of the domain is not an issue, making this problem ideal for a computer to solve.

This incredible speed with which a computer can perform actions is what makes a backtracking algorithm solution viable. Even when the algorithm chooses a path that will eventually lead to failure, it is able to detect this failure and move on to the next path so quickly that making a few incorrect decisions is hardly an issue. As long as the algorithm is built so that it avoids attempting the same path twice, a backtracking solution will always find a solution. However, this particular solution attempts to leverage various constraint satisfaction techniques and heuristics to improve the algorithm’s performance and limit the amount of incorrect decisions the algorithm makes along the way to finding the solution.

**Background**

Before exploring the methods used to improve the performance of the standard backtracking algorithm, it is important to fully understand the core logic behind it. In its simplest form, a backtracking solution behaves very similarly to depth-first search. The algorithm ultimately attempts to reach the bottom of the search tree as quickly as possible. When faced with multiple choices, the algorithm chooses one trivially and continues on. Once the algorithm reaches the bottom of the branch of the tree it is exploring and finds it to be an invalid solution, it works its way back up the tree until it reaches a point where it has not yet explored every possible choice. By following this systematic method and remembering which paths it has already taken, the algorithm will eventually find a valid solution. However, this rather naïve approach leaves open the possibility that the algorithm could potentially explore every single node in the state space before finding a valid solution. By utilizing the optimization methods leveraged in the solution explored in this essay, the algorithm will avoid making trivial decisions, and instead actively attempt to avoid making incorrect decisions, while still trying to reach the bottom of the search tree as quickly as possible.

**Solution Description**

The general objective of the solution is as follows: optimize the basic backtracking algorithm as to reduce the number of incorrect decisions the algorithm makes, and in instances when an incorrect decision is made, attempt to detect failure as early on as possible. In order to take advantage of these optimization techniques, it is important to encode the necessary information within the data structures we use to store the problem.

**Data Structures and Constraint Encoding**

The program starts by reading in the unsolved board and storing the value at each cell in a two-dimensional array. Each cell in the array contains the value of whatever value is currently occupying a specific square on the gameboard. For instance, arr[0][0] contains the value currently occupying the top-left square of the board. Every element in the array is either a digit between 1 and 9, or a period which resembles the square is blank. After populating this initial array, the program then creates a second two-dimensional array of the same size. This array stores the domain of remaining valid values for each square in the board. This is array is filled by going through every square on the board and checking which values are in the same row, column and region, and then removing those values from that cell’s remaining domain. Maintaining individual domains for each cell on the board is essential in order to use many of the optimization methods.

**Forward Checking**

Forward checking is a fairly simple method for detecting failure. Every time a value is assigned to one of the cells, the domain for every remaining unassigned cell is updated based on this new board. If the domain for one of the cells becomes empty, and the current node in the state-space is not a valid solution state, then this state is a failure and the algorithm should start backtracking. Forward checking essentially allows the algorithm to detect failure one level higher on the search tree than it would have if it had just continued naively. However, updating the domains of every unassigned variable in the state space can be extremely costly, especially in problems with many variables, such as Sudoku. Therefore, forward checking alone would not be a worthwhile approach to optimizing the basic backtracking solution.

**Arc Consistency**

The main detriment to forward checking is that it involves updating the domains of every unassigned variable in the state space, which can be extremely costly, particularly at higher levels of the search tree. Arc consistency is a way to mitigate how many domains must be updated at each step, while also enabling the algorithm to detect failure earlier.

Every time the domain of a variable is changed, be it by the assignment of a value to a variable, or by the removal of a variable from the domain as a result of arc consistency, the algorithm checks the domains of every adjacent variable, ensuring they contain a consistent value for every value remaining in the current variable’s domain. For every value it finds in the other variables’ domains that is not consistent, it removes that value from that variables domain.

This process can be accomplished by keeping a queue of variables that must have arc consistency performed on them. As soon as a value is assigned to a variable, that variable is added to the queue. Then, as arc consistency is performed, and variables whose domain is affected are added to the queue. The algorithm proceeds to perform arc consistency on the next variable in the queue until the queue is empty. Throughout this process, forward checking is also being maintained. If any of the domains become empty as a result of performing arc consistency, and the current Sudoku board is not a valid solution, then a failure is detected and the algorithm begins backtracking.

By utilizing forward checking in conjunction with arc consistency, implied failures can be detected earlier on, which can help prevent the algorithm from going deeper into the search tree when it is on a path that will inevitably lead to failure.

**Heuristics**

The techniques discussed up to this point improve the algorithm’s ability to detect when it has made incorrect decisions as early as possible, however they do not help prevent the algorithm from making those incorrect decisions in the first place. Heuristics can be used when choosing which square on the Sudoku board to fill next, and which value to assign to it in order to attempt to prevent making incorrect choices and give the algorithm the best possible chance to get on a path to a successful solution as quickly as possible.

**Minimum Remaining Values**

The minimum remaining values (MRV) heuristic is used when deciding which square of the board to assign a value to next. MRV’s logic is that by assigning a value to the variable with the smallest remaining domain first, this limits the branching factor of the local space the algorithm must explore. As a result, the algorithm will either explore all of the options for that node quickly and move on to the next one, or find a solution quickly and complete the problem. By maintaining this systematic and consistent policy of choosing which square on the board to fill next, the performance of the algorithm improves in both speed, as well as predictability, when compared with the variant of simply choosing a square at random.

The process for finding the smallest remaining domain of the all of the remaining unassigned squares is straightforward. Simply iterate through every value in the array containing the values present on the game board. If that array has a value assigned to it already, then move on to the next one. When the algorithm reaches the first unassigned square, it should keep track of what the size of that squares domain is, as well as the index of that square in the two-dimensional array. As the algorithm continues iterating, it should update this variable if it comes across an unassigned domain that is smaller than the current minimum.

This process can be optimized even further by immediately exiting the iteration process when reaching an unassigned square with a domain size of 1, as it will be impossible to find a smaller unassigned domain throughout the rest of the board.

MRV is an effective heuristic in limiting the amount of states the algorithm explores before finding a valid solution. This helps improve the number of recursive calls that must be made on the algorithm, which improves the speed with which it finds a solution.

**Related Work**

Another heuristic that can be useful for improving the efficiency of a recursive backtracking solution is the least constraining value (LCV) heuristic. This particular heuristic was not implemented in this solution, mostly due to time constraints. However, the behavior of LCV will be explained as a reference for further work to be done on this project.

LCV is a heuristic that is used when deciding which value to assign to a currently unassigned square. The logic of the heuristic is to choose whichever value will constrict the domains of its neighboring squares the least. This should give the algorithm the best chance at finding a solution successfully if indeed a solution exists on its current leaf of the search tree.

Theoretically LCV can be used with MRV to choose to fill the square with the smallest remaining domain, and then choosing to assign the value to it that constricts the domain of its neighbors the least. In theory, this should limit the amount of states the algorithm needs to explore, while also giving the algorithm the best chance of finding a solution as it goes further down the search tree.

**Experiments**

The main experiment that was used while developing this algorithm was to print out a sample Sudoku board and fill it out by hand, but only making decisions that the algorithm would make based on its current state. If at any point during this process the tester filling out the board noticed they were making a poor decision, despite following the algorithm, it was noted why this decision was known to be poor. After the experiment was complete, the notes were reviewed, and the logic displayed in the notes was attempted to be implemented in the algorithm itself.

**Conclusion**

The result of this project was a program capable of reading in an incomplete Sudoku board, solving it using a recursive backtracking algorithm, and printing the solved board. This project provided an educational application to the various constraint satisfaction methods learned during the completion of CS4100 at Northeastern University. The concepts in the course were mostly explored on a theoretical level. Therefore, the completion of this project provided an opportunity to learn how to fully implement and leverage these concepts to improve the performance of what, at its core, is a fairly simple algorithm.

1. [↑](#footnote-ref-1)