CSCI 446 — Artificial Intelligence Project 1 Search, Constraint Satisfaction, and Sudoku

Diego Moylan DIEGO.MOYLAN@MSU.MONTANA.EDU
Department of Computer Science
Montana State University
Bozeman, MT 59715, USA

Jakob Kominsky JAKOB.KOMINSKY@MSU.MONTANA.EDU
Department of Computer Science
Montana State University
Bozeman, MT 59715, USA

Editor: Diego Moylan

Abstract

Our team implemented and compared five different algorithms for solving 9x9 sudoku boards of varying difficulty. We implemented basic backtracking, backtracking with forward checking, arc consistency, simulated annealing, and a genetic algorithm. Our experiment measured the number of value assignments that each algorithm made when solving identical puzzles for basic backtracking, backtracking with forward checking, and arc consistency. We compared the stochastic algorithms by using the minimum number of conflicts and fitness after a set of given number of iterations, while keeping generation and neighborhood sizes consistent. Our results showed that arc consistency was able to solve the sudoku puzzle with the least amount of value assignments. Although the stochastic algorithms showed significant improvement over iterations, they were not able to find solutions to the puzzles. This may have been due to our fitness functions only considering the amount of conflicts, which likely was not sufficient for solving the puzzle.

1 Problem Statement and Hypothesis

Sudoku can be modeled as a constraint satisfaction problem, where each empty cell in the 9x9 board is a variable with possible values from 1 to 9. The rules ensure that no row, column, or 3x3 box has repeated numbers. Constraint satisfaction problems involve numerous variables and constraints. They are useful in real-world applications because they provide a standardized and efficient way to represent problems. Based on our planned implementation of our algorithms, the backtracking algorithm is expected to solve all puzzles. However it may struggle to scale efficiently as the puzzle difficulty increases. Back tracking with forward checking and arc consistency will build off backtracking and be much faster with solving the medium and hard puzzles but these

algorithms will be harder to implement. Simulated annealing may find solutions to sudoku puzzles but may find it difficult to escape local maximums. The genetic algorithm is expected to have difficulty consistently finding solutions and be less reliable for more challenging puzzles, often getting stuck at local optima.

2 Description of Algorithms Implemented

In this project we implemented five main algorithms to try to solve a sudoku puzzle. Three of these algorithms relied on recursive backtracking as the main solving force, while the other two implemented stochastic elements to solve the puzzle. The three that used backtracking were simple backtracking, backtracking with forward checking, and backtracking with arc consistency. The stochastic algorithms were simulated annealing and a genetic algorithm with tournament selection.

Simple backtracking places a digit in the sudoku puzzle and checks if there are multiple of the same digit in the modified row, column, or 3x3 box. If there are no conflicts, it continues placing digits from the entire set of values in empty variables (cells) until a conflict occurs. When there is a conflict, the algorithm empties the cell it just modified and backtracks to try the next number in the set of values. If no numbers in the set of values are free of conflicts, it backtracks again to the previous cell. This recursion happens until the puzzle is filled with digits and there are no conflicts (solved state) or no digit in the complete set of values placed in the first empty cell leads to a solved sudoku.

Backtracking with forward checking first initializes the domains of each empty cell by eliminating values from the set of digits if those values are in neighboring cells. It then implements the backtracking logic, only choosing from values in the pared-down domain of a cell. This algorithm will place a digit, update the affected domains of empty cells, and find a conflict if one of those domains becomes empty. To stay consistent with recursion, our version keeps track of removed values in order to refill the domains when we need to backtrack.

Backtracking with arc consistency initializes the domains of each empty cell the same way as forward checking, with the caveat that given digits also have a domain—the only value of which being the given digit. However, the way it determines if a conflict has occurred is by checking the conflicts of the domains themselves. Essentially, the domain of a cell is reduced if one of its neighbors prohibits a certain value from existing in that domain. This algorithm will place a digit in an empty cell based on its domain and run the arc consistency algorithm to reduce domains. If no domains are length zero, a digit is placed in the next empty square. If one of those domains is empty, the square is emptied and we re-initialize the domains based on the updated state of the sudoku. Using this technique, we don't need to keep track of removed values, which significantly simplifies the implementation of the algorithm.

Simulated annealing initializes the puzzle by creating a boolean 'protection' matrix to keep track of given digits, then filling the empty cell with random digits with

equal probability. This will inevitably create a matrix with repeated digits in rows, columns, and 3x3 boxes. To count conflicts, the algorithm checks the difference of the size of the set of unique digits in each row, column, and box with that of a completed sudoku. The algorithm then gathers a selection of relatively nearby neighbors in state space by randomly mutating non-given digits, then counts the number of conflicts within each neighbor. The puzzle in this selection with the lowest number of conflicts is said to have the highest 'energy' and is chosen as the candidate to switch with the current puzzle. If it has higher energy than the current puzzle, the algorithm will switch to this puzzle. If it does not, the algorithm will switch to the new puzzle with a certain probability based on a 'temperature' value. The probability of switching to a lower energy candidate starts relatively high, but decreases quickly, as temperature decreases exponentially with the number of iterations. Every iteration, this process repeats, taking in the new state-space neighbors and determining whether to switch. The algorithm will stop when either the number of conflicts is zero (solved state) or the number of iterations has hit the maximum set by the program (failure). Hyperparameters in this algorithm included the initial temperature, the temperature reduction coefficient (alpha), the number of iterations, and the number of neighbors checked per iteration.

The genetic algorithm initializes the puzzle the same way as simulated annealing; a protection matrix is created to keep track of given digits, then empty cells are filled with random digits with equal probability. The algorithm also counts conflicts the same way as simulated annealing, by using the size of the set of unique digits in the rows, columns, and boxes. The number of conflicts is then passed into a fitness function that penalizes puzzles with a large amount of conflicts. As a result, puzzles with the fewest conflicts are considered the most fit. Parents are selected from the population via tournament selection, where two puzzles are drawn from the current generation without replacement, and the puzzle with the higher fitness becomes one of two parents. After selecting another parent the same way, the parents are recombined into two child puzzles by flattening out the matrices and picking an arbitrary recombination point. These children are resized to 9x9 matrices and mutated. The mutations do not affect the protected digits, but otherwise any cell can be replaced with a random digit, with probability based on the mutation rate hyperparameter. Children are created this way until the number of children is equal to the generation size. At this point, the algorithm checks all children for the number of conflicts and repeats until there is a puzzle with no conflicts (solved state) or the number of iterations has hit the maximum set by the program. Hyperparameters in this algorithm included the generation size, number of generations, the random mutation rate, and the row-swap mutation rate.

3 Description of Experimental Approach

To evaluate the performance of our algorithms we decided to count the number of value assignments that each algorithm made when solving identical puzzles. Because the stochastic algorithms did not converge to a solution, they were compared against each other by using the minimum number of conflicts each had after a given number of iterations with comparable generation/neighborhood sizes.

Counting the number of value assignments does not tell us the time complexity of these algorithms, but it elucidated a rough idea of the efficiency with which they could solve the puzzles. To count assignments, we simply had a global variable that would increment whenever a digit was placed in the sudoku puzzle. We were able to run an algorithm on a puzzle then reset the puzzle and count so the next algorithm could be run. This allowed us to easily make graphs of the performance of the backtracking algorithms to test how well they could solve different levels of sudoku puzzles.

The experimental approach was similar for the stochastic algorithms.We tuned each algorithm's hyperparameters to perform at its respective best. This led to both coming to a local minimum, although unfortunately neither were able to find a solution. Using this method we were able to track the minimum conflicts for each generation and also arrive at a final comparison for the two stochastic algorithms.

Hyperparameter selection is very important for both simulated annealing and genetic algorithms. The performance of simulated annealing is highly sensitive to its temperature tuning and cooling schedule; if these numbers are not tuned they can cause the algorithm to cool too quickly and fail to converge to a solution. For the genetic algorithm, its generation size, mutation rate, and number of generations were all impactful up to a certain point.

For simulated annealing we first tried an initial temp of 1 and an alpha value of 0.99. We found that these values cooled the algorithm down too much, causing reduced exploration of states for further iterations. We tested out a range of values for these hyperparameters and came to a conclusion of having our initial temp set to 10 and alpha set to 0.995. The neighborhood size and number of iterations determined when and whether the algorithm would come to a stopping point, usually occurring around 1500 iterations.

The genetic algorithm's performance depends heavily on its hyperparameters. Suboptimal parameter selection can lead to premature convergence to a suboptimal local maximum or insufficient selective pressure. Our population size was set to 100 to have a reasonably large population to choose parents from. The number of generations was set to 2000 as this allowed there to be sufficient generations with meaningful evolutionary progress. The mutation rate was set to 0.008 to improve genetic diversity and the algorithm's ability to escape local optima.

4 Presentation of Results

In our experiments we compared the count of the assignments made by each algorithm when running on identical puzzles. For the stochastic puzzles we decided to observe their minimum conflicts and fitness once progress stopped, as neither converged to a solution. Since fewer assignments indicate higher efficiency we will present raw numbers as well as the relative improvement of each algorithm. As seen in figure 1, we found that the number of assignments made by the simple backtracking algorithm exploded with the difficulty of the puzzle, as it went from an average of 2885 assignments on easy puzzles to 71439 on hard difficulty. Forward checking reduced assignments made by roughly 90% compared to backtracking on easy puzzles. Across all puzzle difficulties arc consistency required fewer than 100 assignments, whereas backtracking and forward checking required thousands to tens of thousands assignments. Arc consistency made an average of 95% fewer assignments than backtracking, and 90% fewer than forward checking across all difficulties, making it by far the superior algorithm in assignment efficiency.

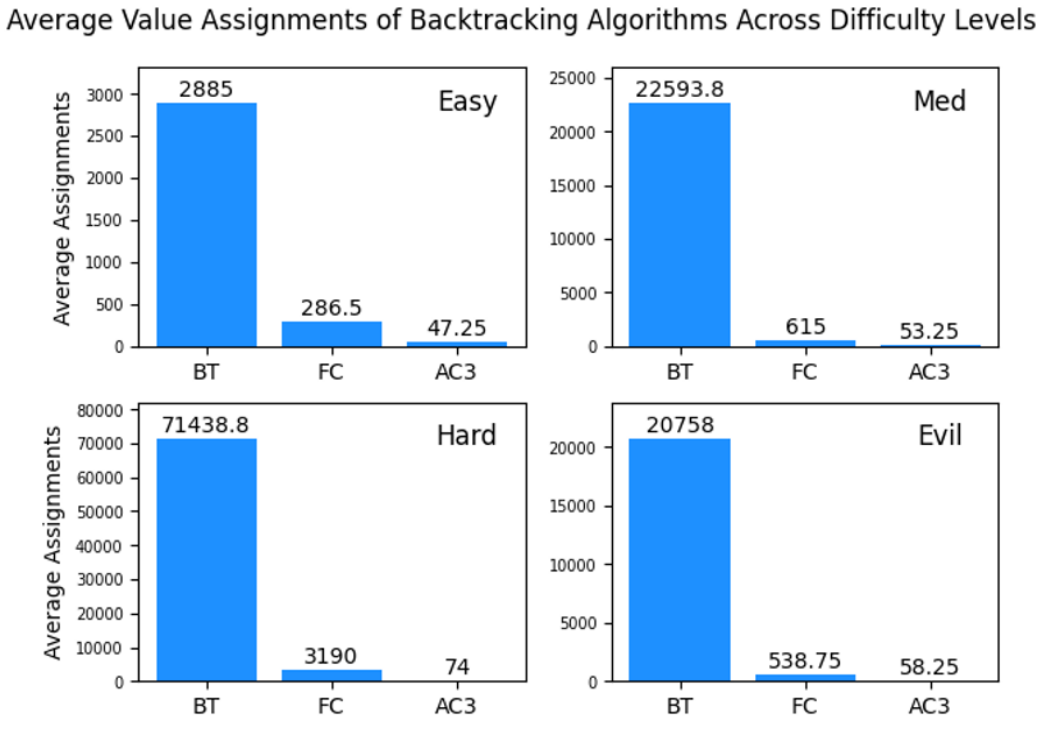

Figure 1: Bar charts of the average number of value assignments each backtracking algorithm made on the full set of puzzles in each difficulty category

The results illustrate that arc consistency scaled the best when it came to assignments made in the sudoku puzzle. Incorporating constraint propagation methods

leads to dramatic efficiency gains over basic backtracking, especially as difficulty increases. Forward checking is a middle ground as it offers substantial improvements at lowering assignments made compared to backtracking, which was by far the least efficient. For simulated annealing we compared the running best value for minimum conflicts across each difficulty of the puzzle. Simulated annealing has high temperatures in the early iterations, allowing the algorithm to explore more of the state space. As the temperature decreased the algorithm became more selective and moves chosen were fewer than the initial exploration iterations. Shown in figure 2, when iterations ran over 400 the low temperature caused the algorithm to make only positive energy (negative conflict) moves, reducing the fluctuations seen in early iterations. The plateaus at the later end of the iterations show that the algorithm may have found a local optimum.

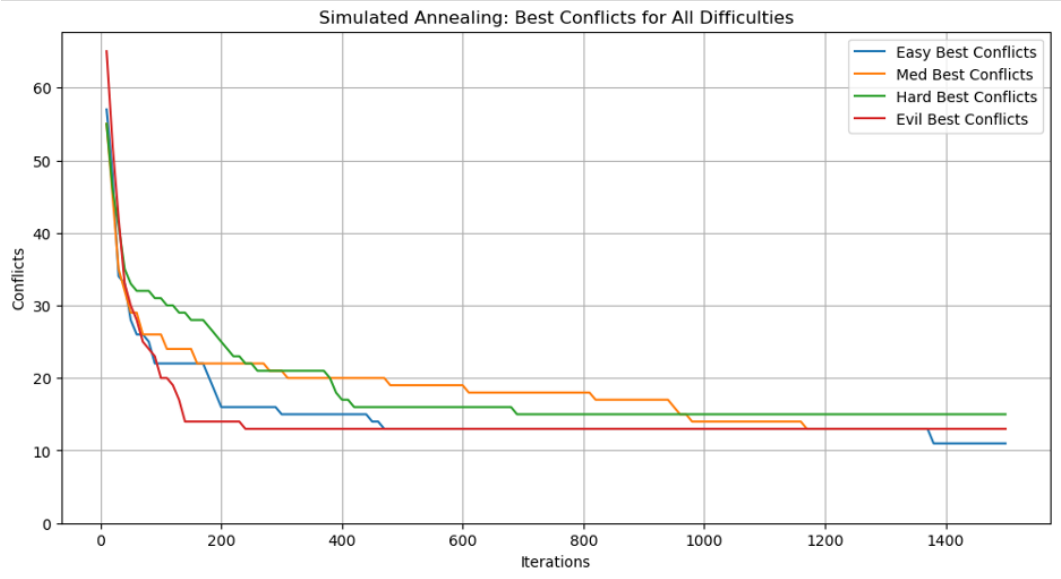

Figure 2: Graph of running best minimum conflicts vs. iterations using simulated annealing

The genetic algorithm compared the minimum and maximum conflicts that each generation had over a certain amount of generations. The graph in Figure 3 shows the rise and fall of the algorithm trying different states and randomly mutating for better or worse conflicts.

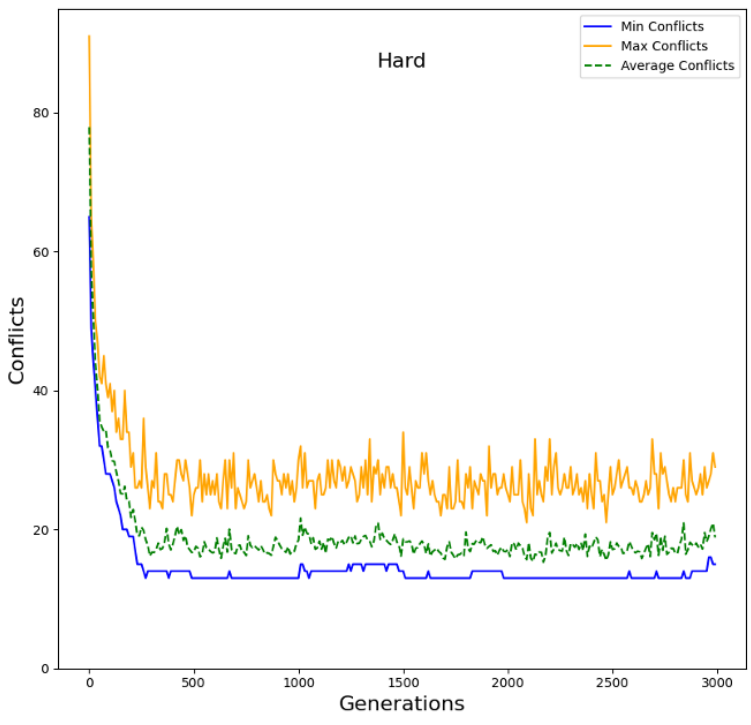

Figure 3: Graph tracking the number of conflicts in the members of each generation when running the genetic algorithm on puzzle Hard-P1

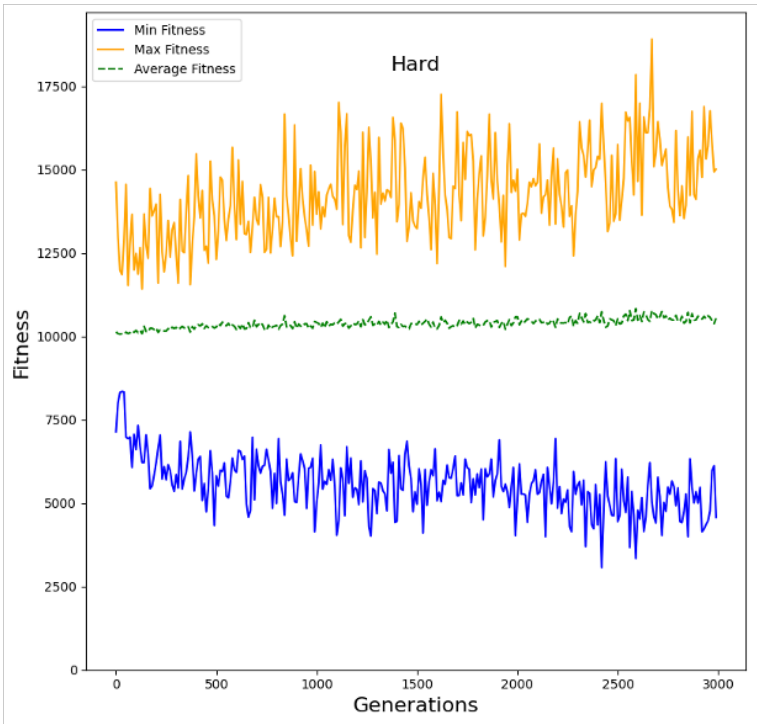

Figure 4: Graph tracking the fitness values in the members of each generation when running the genetic algorithm on puzzle Hard-P1

Early in the runs the genetic algorithm shows significant improvement that signifies strong selection and effective recombinations producing better offspring with less conflicts. Figure 4 doesn't show a significant increase in fitness with generations because our fitness function for the genetic algorithm calculated relative fitness among the generation, so the average value stayed relatively steady.

5 Behavior of Algorithms

The way the algorithms solve these puzzles was implemented differently but each algorithm was made for faster assignments and solutions. Bactracking explores the entire puzzle by assigning values to empty cells and revisiting them if any conflicts in the constraints are found. This is an exhaustive search but generates a solution and often requires many unnecessary assignments due to the large search space. As the puzzle difficulty increases, backtracking still solves the puzzle reliably but can become slower because of the broadness of exploration.

Forward checking builds on backtracking by looking ahead and when it assigns a value to a cell, it will prune the domains of other unassigned cells to prevent conflicts later. This cuts down wasted assignments and eliminates deeper inconsistencies

throughout the puzzle. Forward checking is much more efficient in assigning values to cells when compared to backtracking.

Arc consistency enforces consistency across all constraints before any assignment to a cell is made. By maintaining consistent domains throughout the puzzle, it dramatically reduces the number of assignments required. This makes arc consistency highly effective for constraint satisfaction like sudoku.

Simulated annealing uses probabilistic and random exploration to navigate the solution space, this allows it to escape local minima. While it cannot get to a solution, the conflicts over iterations go down and show that the algorithm is getting to a local minima. When the puzzle becomes more constrained with puzzle difficulty it can fail or take longer to find a solution to the puzzle.

The genetic algorithm maintains a population of candidate solutions and uses tournament selection with mutations to improve over generations. Like simulated annealing, it does not guarantee a solution. Its performance depends heavily on getting conflicts, tournament selection, population size, mutation rates, and number of generations. This makes it less predictable than constraint based algorithms and makes it harder to find solutions.

The results key in on the trade-offs between constraint based algorithms and stochastic algorithms. Stochastic algorithms can explore solutions through probabilistic optimization and mimicking biological evolution. These can be less reliable for finding solutions in problems like sudoku. Constraint based algorithms are quite reliable and efficient in solving constraint based problems like sudoku. The trade-offs of these puzzles have their advantages and disadvantages but both have their strengths and weaknesses depending on problem context.

6 Summary

In the process of creating this paper, we implemented five different algorithms to solve sudoku puzzles. We found that stochastic algorithms struggled with converging to a solution because we only had one metric contributing to our fitness function. However, backtracking algorithms were quite effective at finding solutions, especially arc consistency, which vastly outperformed the other algorithms in terms of value assignments.

7 Division of Work

After assigning the coding responsibilities we found that the original separation of work was not equal, so we modified it so we could get everything done on time and with a reasonably equal amount of effort expended between us. Diego handled I/O, simple backtracking, and the genetic algorithm. Jakob was responsible for forward checking and simulated annealing, and we both worked on arc consistency. Testing and experimentation was split equally.

References

Russell, S. J., & Norvig, P. (2021). Artificial Intelligence: A modern approach. Pearson.