Solving Sudoku Using Depth-First and Local Search Based Algorithms

Anthony (AJ) Mann ANTHONYSTEVENMANN@GMAIL.COM
Artificial Intelligence Student
Montana State University
Bozeman, MT 59718, USA

Jackson Thorn JTHORN2448@GMAIL.COM
Artificial Intelligence Student
Montana State University
Bozeman, MT 59718, USA

Abstract

This paper aims to explore multiple constrained search methods with the goal of solving a Sudoku. Backtracking is a depth-first search that systematically searches for a solution by building a candidate solution and abandons partial solutions once it is determined that it cannot build a complete solution. Forward checking extends upon the brute-force nature of simple backtracking and provides the algorithm with domain updates to reduce the search space. Arc consistency builds on this further by providing persistent domain updates and creating assignments for cells with singular domain values. Simulated annealing and the genetic algorithm take stochastic approaches, starting with randomly initializing a complete candidate solution(s). Both simulated annealing and the genetic algorithm are stochastic by nature, and finding a solution to the Sudoku is not guaranteed. Each variation of backtracking proved to be able to solve a Sudoku and when provided with domain updates, the number of backtracks decreased. The stochastic algorithms came close to solving the Sudoku, though failed to reach a complete solution a majority of the time.

1 Introduction

Sudoku is a class of puzzles taking place on a 9x9 grid. In standard games, this is further subdivided into separate 3x3 sections. The start of the puzzle consists of a subset of numbers already filled into spaces on the grid. The goal of each puzzle is to finish filling the grid with numbers. However, there are three basic rules of number placement that constrain the puzzle. First, each row must contain all numbers, one through nine, with no repeats. Second, each column must also contain all numbers, one through nine, with no repeats. Finally, each 3x3 grid is also subject to the same constraints and must have all numbers, one through nine.

A Sudoku can have four difficulty levels: easy, medium, hard, and evil. As difficulty increases, the initial state of the puzzle contains fewer integers, giving each null space a larger domain of initial possibilities. The puzzles are provided as comma-separated.txt files. No data cleaning is required; they can be im- ported directly into Numpy arrays.

The Sudoku is completed with a single constraint solver that can switch between a set of given heuristics while using constraint satisfaction to navigate through the search. There are five different variations of this solver: backtracking (simple, forward checking, arc consistency) and local search (simulated annealing, genetic algorithm).

This experiment aims to compare the efficiency with which each algorithm solves a Sudoku. Backtracking is evaluated on the basis of the number of assignments, and subsequent number of backtracks, taken to reach a solution. The stochastic algorithms are evaluated by the best fitness across the number of iterations/generations.

We hypothesize that backtracking with arc consistency (AC3) will perform the best out of all algorithms. By nature, human intelligence employs this method to solve Sudoku. Additionally, backtracking with arc consistency should be able to solve lower difficulty Sudoku. Backtracking with forward checking should provide similar performance, but since assignments are not created during domain updates, it will take longer to reach a solution. Simple backtracking, though it will be able to solve the Sudoku, has the limitation of exploring every possible branch that the Sudoku could take which will severely limit its performance. The stochastic algorithms could outperform backtracking, but due to the vast variety in design choices, it is difficult to estimate how they will perform.

2 Descriptions of Implemented Algorithms

2.1 Simple Backtracking

Backtracking is a depth-first search based algorithm that systematically searches for a solution by building up a candidate solution. Partial solutions are abandoned once it has been determined that this path cannot lead to a complete solution. The starting point of the backtracking process is the first empty cell. The algorithm visits that empty cell, places a number $n \in [1, 9]$ and checks for any conflicts based on the normal Sudoku constraints. If a conflict is found, the algorithm backtracks and tries the next valid number. Once no conflicts are found, the value is considered valid and the algorithm moves on. This process repeats until the Sudoku is complete. Simple backtracking will only be used on Sudoku problems of easy and medium difficulty.

2.2 Backtracking with Forward Checking

Backtracking with forward checking is virtually the same as regular backtracking; however, the domains of any unassigned cells are pruned immediately after a cell is updated. When a number $n \in [1, 9]$ is assigned to an empty cell, all unassigned neighboring cells in the same row, column, and 3x3 grid as the initial number will be iterated over. For each neighboring cell, the number n is removed from the domain of possible assignment values. If this removal causes any unassigned cells to have an empty domain, then there is an inconsistency, and the algorithm immediately backtracks. This process repeats until the Sudoku is complete. Unlike with simple backtracking, backtracking with forward checking has the ability to look

forward and stop early. This slight spin on backtracking will also drastically condense the search space as each assignment is limited by its' neighboring cells.

2.3 Backtracking with Arc Consistency (AC3)

Backtracking with arc consistency (ac3) significantly reduces the search space. If two cells are in the same row, column, or 3x3 grid, they have an arc between them. Before the backtracking process begins, ac3 checks the domains of each empty cell based on the initial state of the Sudoku by systematically removing inconsistent values from the empty cell's domain. If any empty cells have single domains, an assignment is created. These assignments update the Sudoku state, and ac3 is run again to find if any more empty cells now have single domains. If, at any point, the domains of the empty cells contain 2 or more values, the solver creates an assignment from one of those values. If any contradictions are found, the solver backtracks. The process repeats until the Sudoku is complete.

2.4 Simulated Annealing

Local search differs significantly from backtracking. Rather than constructing a solution iteratively, the algorithm starts with a complete assignment and attempts to fix any conflicts. The initial state of the Sudoku is complete but incorrect: all empty cells are filled randomly from $n \in [1, 9]$ such that the Sudoku will have nine of each number. Choosing which cells to swap is completely random. Cells that are fixed and given as hints in the beginning are not explored. A cell has conflicts if there are any duplicate values within its row, column, and 3x3 grid. The Sudoku's state can be represented by the total number of such conflicts. The two randomly chosen cell values are swapped and checked again for any updates in conflicts. If the new Sudoku board has less conflicts than the original board, the new board is kept. There is also a stochastic element to chosen the "better" algorithm. Sometimes the "worse" puzzle is kept based on random chance. This random chance works on the basis of a tuned hyperparameter: temperature. A higher temperature T allows the search space to be effectively explored, even if the Sudoku's state worsens. If the newly updated Sudoku board results in fewer conflicts ($\triangle E > 0$) it is always accepted. If not, the current temperature probabilistically decides if the value should be kept using a term from the Boltzmann Distribution: $e^{\frac{\triangle E}{T}}$. As the process iterates, temperature decreases until convergence based on a cooling schedule. This cooling schedule is another tuned hyperparameter. Due to the stochastic nature of the problem, a solution is not guaranteed to be found.

2.5 Genetic Algorithm

The local search process can also be performed through a genetic algorithm. Rather than evolving one solution over time, a population of candidate solutions evolves toward better solutions. Each individual in the population is a complete but incorrect Sudoku, where all empty cells are filled such that the puzzle contains exactly nine of each digit (1-9), but these digits are randomly distributed without regard to row, column, or 3×3 grid constraints. During evolution, only these randomly assigned values in non-fixed cells can change.

Each candidate has fitness $F = P - (V_{row} + V_{col} + V_{grid})$ where V_{row} is the count of conflicts

in all rows, V_{col} is the count of conflicts in all columns, V_{grid} is the count of conflicts in all 3x3 grids, and P is the maximum penalty, set arbitrarily to 1,000. A violation is defined as each occurrence of a duplicate beyond the first.

Based on fitness, parents are chosen for reproduction using tournament selection with tournament size t. A random sample of t individuals is selected from the population, and the individual with the highest fitness becomes a parent. This process is repeated to select a second parent. These two parents undergo grid-wise crossover, where each of the nine 3×3 grids is independently inherited from either parent with equal probability, creating a child that is a combination of these selected grids.

The children then undergo an adaptive mutation process with two phases: First, a count-correction phase repairs any number imbalances caused by crossover to restore exactly nine of each digit. Second, if counts are already balanced, fitness-adaptive mutations are applied. For lower fitness individuals (less than 990), conservative swap mutations exchange 3-8 pairs of non-fixed cells. For higher fitness individuals (greater than or equal to 990), increasingly aggressive mutations are applied, including up to 25 random swaps and scrambling of multiple 3×3 blocks.

When fitness equals the maximum penalty, a valid Sudoku solution has been found. Due to the stochastic nature of genetic search and the complex constraint inter-dependencies in high-fitness regions, a solution is not guaranteed to be found within any given number of generations.

3 Experimental Approach

3.1 Environment and Data

All algorithms were implemented in Python, using a Jupyter Notebook. Each Sudoku is provided as a comma-separated .txt file, with the '?' character representing empty cells and integers representing filled cells. The Sudoku is loaded into a Numpy array, with the '?' characters transformed into zeroes and the data types transformed into integers. Each algorithm outlined above is to be tested on four difficulty levels: easy, medium, hard, and evil. As difficulty increases, the number of empty cells increases, increasing the search space. Due to this, simple backtracking was only tested on easy and medium difficulty. Both the simulated annealing and genetic algorithms were limited on the number of iterations, as they could theoretically run forever searching for a solution. Performance metrics tracked include total assignments, backtracks (where applicable), and iterations.

3.2 Implementation

3.2.1 Backtracking

Simple backtracking was developed first in order to understand how to perform array operations on a Sudoku. Implemented by moving row-by-row, left-to-right, with sequential

value assignments from the domain $n \in [1, 9]$. Forward checking maintained pruned domain sets for each unassigned cell, with arc consistency triggering domain updates after assignment. Arc consistency applied constraint propagation with automatic assignment when cell domains reduced to singleton sets. Backtracking engaged only when propagation failed to make progress and selected cells based on the minimum remaining values heuristic.

3.2.2 Simulated Annealing

Initial temperature set to 35 with exponential cooling schedule ($\alpha = 0.998$). Candidate solutions maintained exactly nine instances of each digit 1-9, with random swaps between non-fixed cells as the neighborhood operator. The cost function calculated total constraint violations across rows, columns, and 3×3 grids.

3.2.3 Genetic Algorithm

Population size of 600 individuals with tournament selection size 4. Each individual in the population represented a complete Sudoku with balanced digit counts placed randomly in empty cells. Grid-wise crossover operated at the 3x3 grid level, in which each Sudoku contained 9. Adaptive mutation implements count correction due to crossover, followed by fitness-proportional swap and grid scramble. This mutation scaled with fitness: individuals with fitness greater than 990 underwent aggressive perturbations (up to 45 random swaps and 9 block scrambles) to escape local optima.

3.3 Termination Criteria

Backtracking, forward checking and arc consistency terminated upon finding the first valid solution, or exhausting the search space. Simulated annealing terminated after 20,000 iterations, or reaching zero conflicts. The genetic algorithm ran for a maximum of 5,000 generations, or until achieving fitness equal to the maximum penalty (1,000), indicating zero constraint violations.

4 Performance

The following tables and graphs provide insight into how each algorithm performed in solving a Sudoku. A discussion of these results is provided below.

4.1 Backtracking Performance

Table 1: Performance of Backtracking Algorithms on Easy Difficulty

Puzzle Group	**Simple Backtracking**		**Forward Checking**		**Arc Consistency (AC3)**	
	Assignments	Backtracks	Assignments	Backtracks	Assignments	Backtracks
Easy-P1	150	102	59	11	48	0
Easy-P2	678	633	523	478	45	0
Easy-P3	407	360	484	437	47	0
Easy-P4	132	87	149	104	45	0

Note: Simple Backtracking was only ran on puzzles of Easy or Medium difficulty.

Table 2: Performance of Backtracking Algorithms on Medium Difficulty

Puzzle Group	**Simple Backtracking**		**Forward Checking**		**Arc Consistency (AC3)**	
	Assignments	Backtracks	Assignments	Backtracks	Assignments	Backtracks
Med-P1	677	626	150	99	68	1
Med-P2	6888	6838	1301	1251	95	5
Med-P3	1511	1458	661	608	72	1
Med-P4	1057	1005	807	755	52	0

Note: Simple Backtracking was only ran on puzzles of Easy or Medium difficulty.

Table 3: Performance of Backtracking Algorithms on Hard Difficulty

Puzzle Group	**Forward Checking**		**Arc Consistency (AC3)**	
	Assignments	Backtracks	Assignments	Backtracks
Hard-P1	2606	2553	149	8
Hard-P2	8381	8328	359	34
Hard-P3	1703	1650	129	8
Hard-P4	1735	1682	125	8

Note: Simple Backtracking was only ran on puzzles of Easy or Medium difficulty.

Table 4: Performance of Backtracking Algorithms on Evil Difficulty

Puzzle Group	**Forward Checking**		**Arc Consistency (AC3)**	
	Assignments	Backtracks	Assignments	Backtracks
Evil-P1	788	733	55	0
Evil-P2	329	273	715	68
Evil-P3	1685	1630	84	1
Evil-P4	50	5	45	0

Note: Simple Backtracking was only ran on puzzles of Easy or Medium difficulty.

4.2 Stochastic Performance

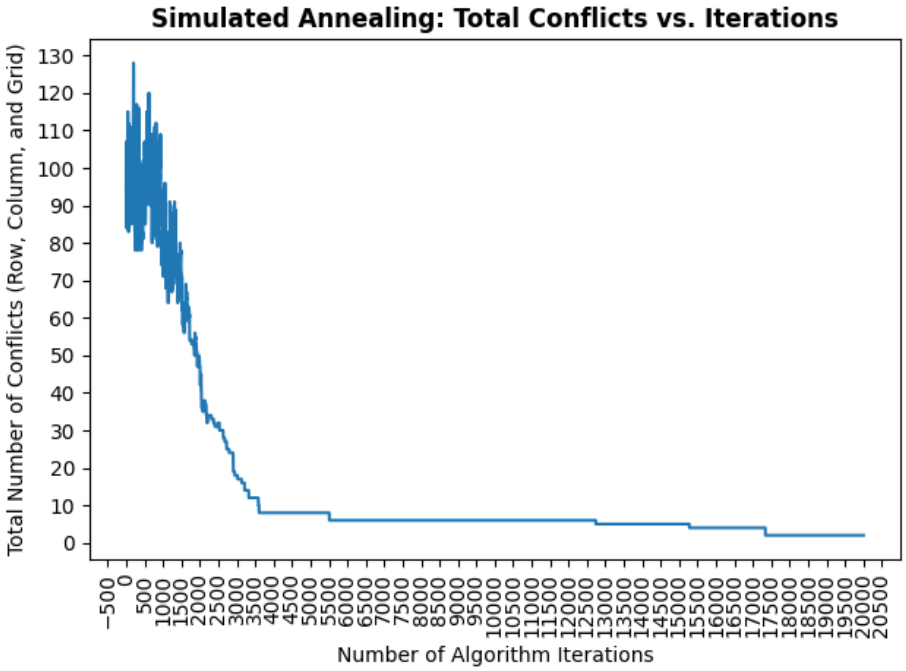

Figure 1: SA being run on "Easy-P1." Algorithm stopped with a conflict score of 2 at 20,000 iterations with 40,000 assignments being made.

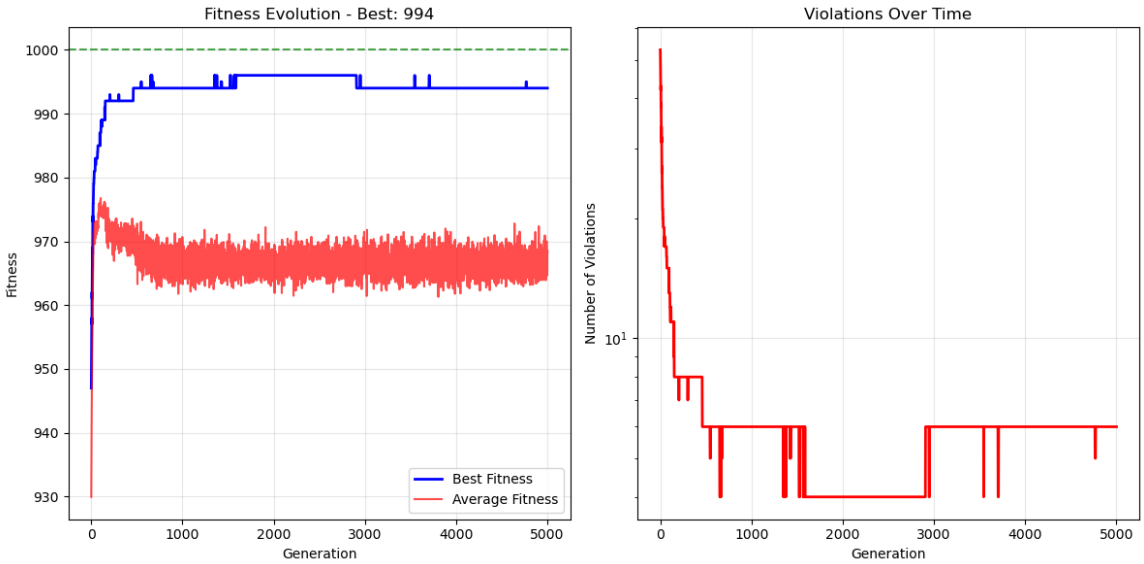

Figure 2: Genetic algorithm being run on "Easy-P1."

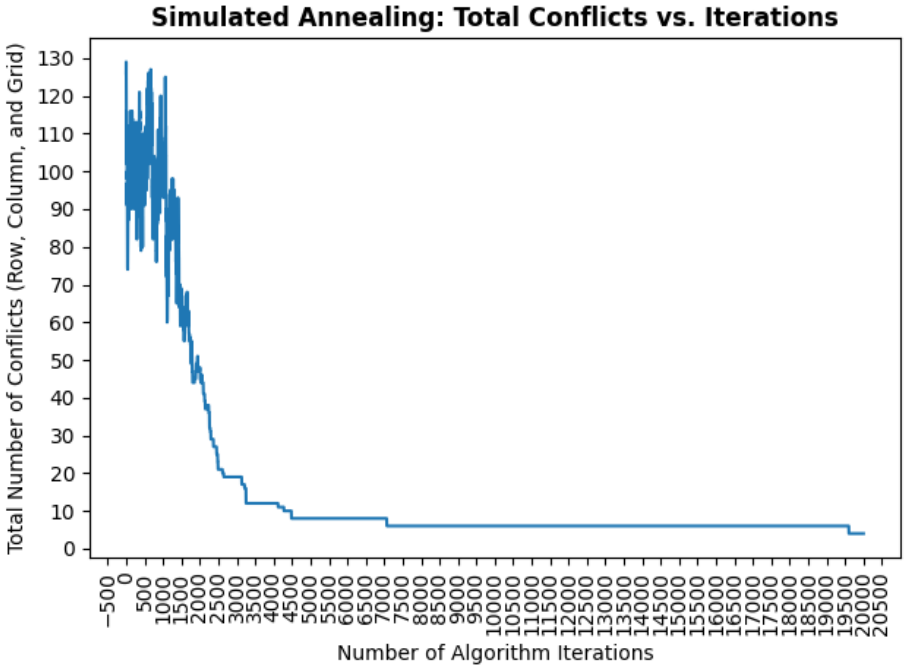

Figure 3: SA being run on "Med-P1." Algorithm stopped with a conflict score of 4 at 20,000 iterations with 40,000 assignments being made.

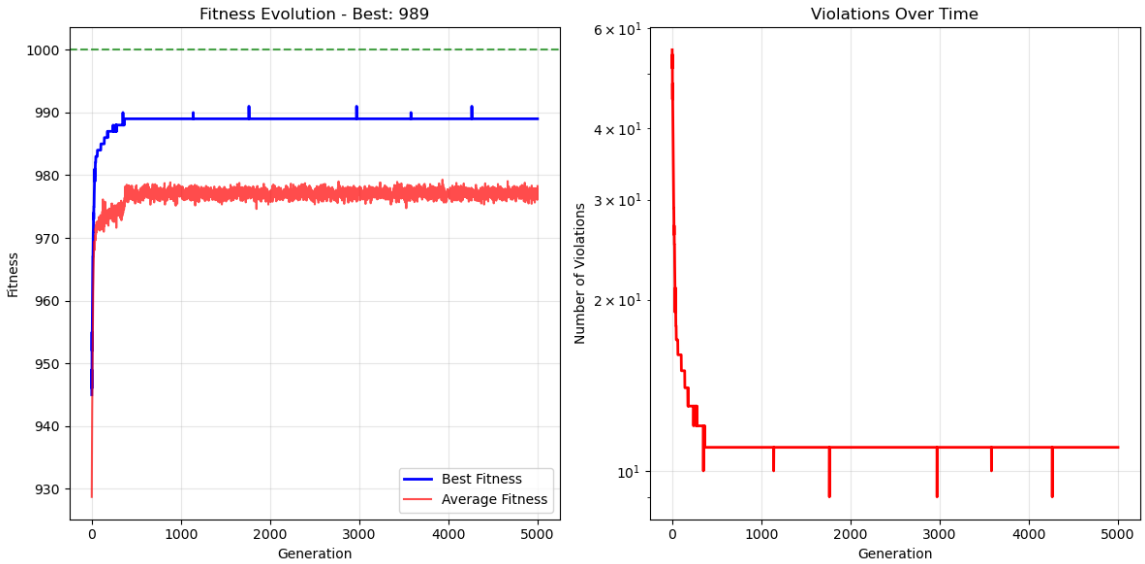

Figure 4: Genetic algorithm being run on "Med-P1."

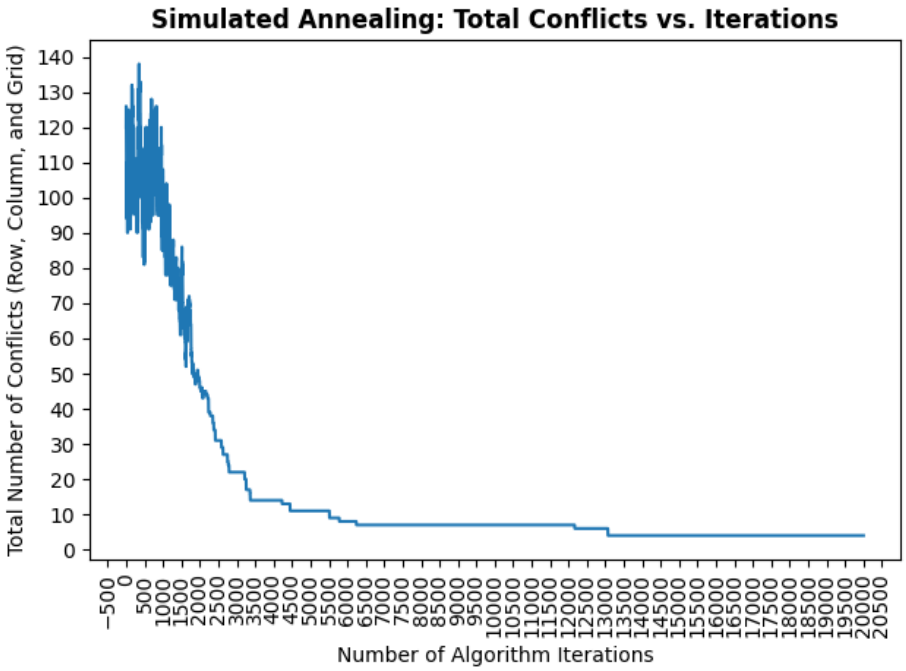

Figure 5: SA being run on "Hard-P1." Algorithm stopped with a conflict score of 4 at 20,000 iterations with 40,000 assignments being made.

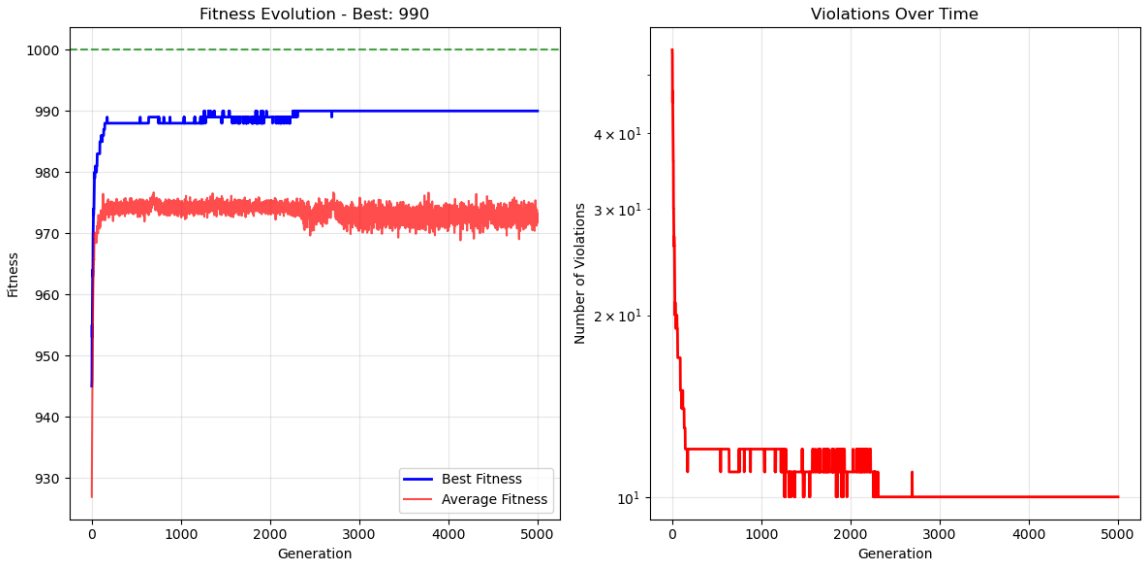

Figure 6: Genetic algorithm being run on "Hard-P1."

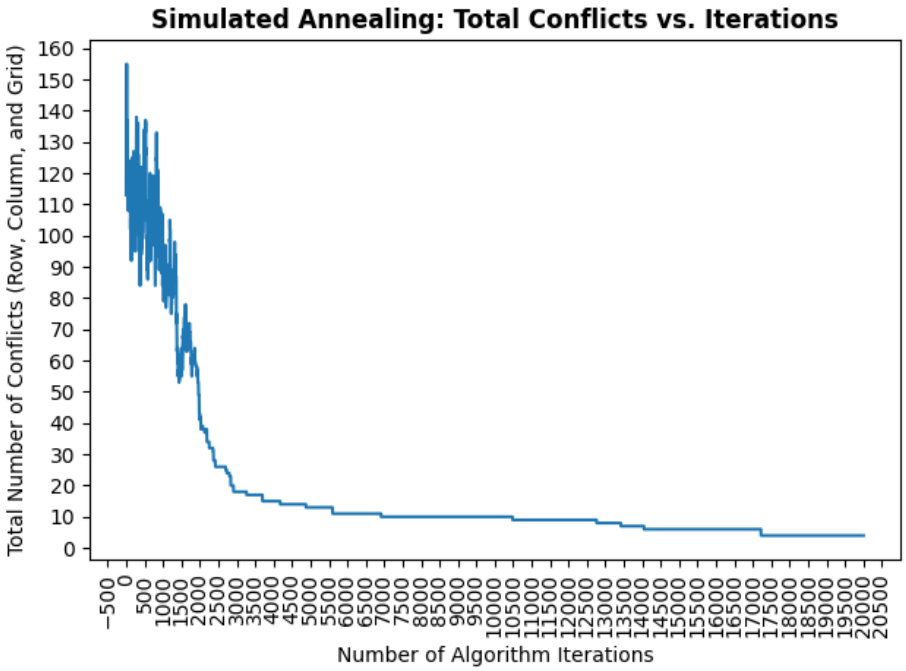

Figure 7: SA being run on "Evil-P1." Algorithm stopped with a conflict score of 2 at 20,000 iterations with 40,000 assignments being made.

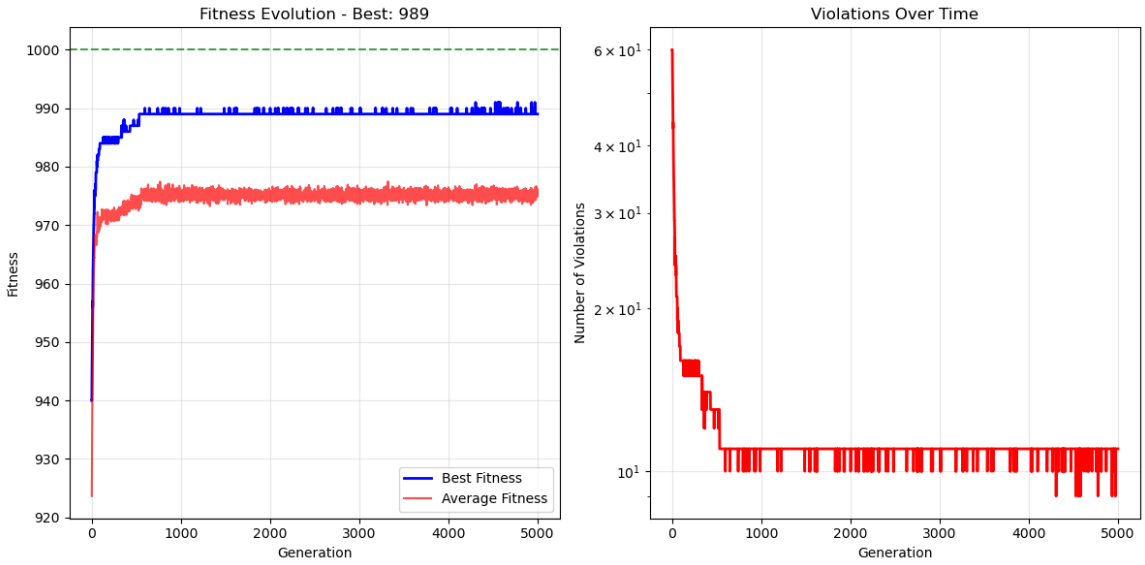

Figure 8: Genetic algorithm being run on "Evil-P1."

5 Results

5.1 Backtracking Results

The results shown for the 3 backtracking algorithms directly proves our hypothesis: AC3 will perform the best. Across all 4 tests with varying difficulty, AC3 was the clear winner. AC3 consistently achieved the fewest number of assignments and backtracks. As expected, easier puzzles were able to be solved purely through AC3 domain updates. As puzzle difficulty increased so did the total number of assignments, due to the expansion of the search space. Forward checking performed better than AC3 on the Evil-P3 puzzle, but that was it. Unexpectedly, AC3 was able to solve two of the evil puzzles without backtracking once. This specific Sudoku set alludes to hard puzzles being more difficult for AC3 to solve than evil puzzles, however these variations likely boil down to a small sample size. Overall, each algorithm was able to produce a solved Sudoku.

Backtracking with forward checking was the second best algorithm of the bunch. Forward checking was only beat outright by simple backtracking for the "EasyP3" puzzle. This vast improvement from simple backtracking was much more noticeable during the testing with puzzles of medium difficulty. This improved performance is greatly attributed to its ability to stop at dead ends sooner than simple backtracking would.

As expected, simple backtracking performed the worst out of all 3 backtracking algorithms. It effectively takes a 'brute-force' approach at solving the puzzle. Though it works, it is not efficient and there is not much more to say.

5.2 Stochastic Results

Declaring a clear winner here is extremely difficult. Out of our small sample of runs with stochastic algorithms, we never solved even one of the puzzles. There could be many things to blame for this shortcoming: hyperparameters tuned incorrectly, algorithms not given enough time to solve, not choosing the best implementation, etc. Regardless, these results show a clear flaw with these algorithms, they are never 100% able to achieve a solution. Even though we never found a solution, we still got some interesting data.

In the simulated annealing graphs we see the same things happening: the number of conflicts initially "shoots" up and is sporadic, but mellows out as time continues. This is exactly the stochastic nature of this algorithm at work. When the temperature is high at the beginning, the algorithm is more likely to accept other solutions that may not necessarily be better than the current solution. This abandoning the "better" solution is all in the name of trying to explore the entire search space. As many iterations of the algorithm are processed, the cooling schedule dampens the temperature and its influence. After 10,000 or so iterations, the temperature is near 0, meaning that the algorithm is not as explorative as it once was. This manipulates the algorithm to focus in on a solution; the algorithm almost becomes more greedy as the temperature "cools."

The genetic algorithm failed to find a complete solution for any puzzle. It was found that, regardless of difficulty, the puzzle effectively converged with a fitness score of 990 after roughly 1,000 iterations - meaning only 10 total violations remained in the puzzle. These violations were likely rows, columns and grids that contained duplicate numbers - and the only way to break out of that maxima was to get lucky through random number generation. Unfortunately, there is not much more to discuss in terms of results: each algorithm run converged to a local maxima quickly and failed to break out.

6 Discussion

The backtracking algorithms performed in a predictable manner. Each algorithm improved upon the last - simple backtracking was effectively brute-force, forward checking pruned the domain for empty cells which reduced the search space and AC3 found empty cells with singleton domains to create assignments. Their implementations did not require significant design decisions and proved effective at solving Sudoku puzzles.

The stochastic algorithms performed in a stochastic manner. It was impossible to predict what would happen. The issue with these algorithms is balancing exploration and exploitation. If there is too much exploration, the puzzle stays stuck in a stochastic state. If there is too much exploitation, the puzzle is stuck in a local maxima. The genetic algorithm attempted to balance this by using grid-wise crossover, since this (theoretically) preserved the puzzle's state better than floating point, row-wise or column-wise crossover. The mutation strategy attempted to first fix the counts on the child, almost always broken by crossover, to at the least provide the individual with a complete but incorrect solution. Once these counts were corrected, adaptive mutation randomly pushed values around the grid in hopes of generating a complete solution. Due to the randomness of this, our algo-

rithm could theoretically run forever. One such run that was not included in the results did converge to a complete solution - 16,000 iterations later. Because of this, we decided to cap iterations and discuss that while our algorithm may be able to eventually provide a complete solution, it would not be an efficient one. The genetic algorithm can be improved, particularly through cross-validation, which ours did not implement due to the perceived computational complexity and time constraints. Other, more complex crossover and mutation strategies could be deployed along with population replacement after stagnation. Overall, we decided to keep it fairly simple and learned about the complexity the genetic algorithm could have.

As mentioned in the results above, the simulated annealing algorithm worked as we had envisioned it theoretically. The amount of conflicts started high and sporadic, but then worked its way to a much more stable number. When the puzzle had ≤ 10 conflicts, the algorithm slowed down drastically. Due to random swapping being the way the algorithm was implemented, these last few conflicts were painfully slow. The algorithm had much less "breathing room" and correct choices to choose from. It could be stuck at 4 or even 2 conflicts for thousands upon thousands of iterations. This could be our own downfall as the temperature or cooling schedule parameters could not have been perfectly tuned for our problem. Given more time, we would be interested to see if properly tuning the parameters would yield a much more consistent and quicker puzzle completion. From our short tuning adventure, we found that a $T = 35$ and cooling rate $\alpha = 0.998$ worked the best for most puzzles.

7 Summary

Implementing each backtracking algorithm was very satisfying. They were straightforward, performed as expected, and produced a solved Sudoku. Implementing the stochastic algorithms proved frustrating. AJ developed simulated annealing, Jackson developed the genetic algorithm. Due to the sheer number of design choices for both, actually finding solutions proved extremely frustrating and we did not achieve the results we expected. They could be adapted with better hyperparameter tuning or more advanced design choices not covered in this class to potentially become very efficient methods for solving a Sudoku.

8 Appendix

Jackson wrote the following:

- Simple Backtracking
- AC3
- Genetic Algorithm
- Paper

AJ wrote the following:

- Puzzle initialization and Algorithm selection logic
- Forward Checking
- Simulated Annealing
- Paper