Sudoku Solver Using Constraint Satisfaction and Local Search

Michael Downs MICHAELD20312@GMAIL.COM
Division of Computer Science
Montana State University
Bozeman, MT 59715, USA

Sadman Sakib SADMAN.SAKIB@STUDENT.MONTANA.EDU
Department of Electrical and Computer Engineering
Montana State University
Bozeman, MT 59715, USA

Chance Larson CHANCET2004@ICLOUD.COM
Division of Computer Science
Montana State University
Bozeman, MT 59715, USA

Abstract

We built a Sudoku solver that can switch between several AI search methods. The search algorithms we utilized were plain backtracking (BT), backtracking with forward checking (FC), backtracking with arc consistency (AC3), simulated annealing (SA), and a genetic algorithm (GA). We ran each of these on puzzles across four difficulty levels: easy, medium, hard, and extreme. Instead of timing how long they ran, which depends too much on the computer used, we measured performance by counting constraint checks, assignments, backtracks, and inferences. Our expectation was that backtracking alone would perform well on the easier puzzles but would struggle with computation time as the puzzles got harder, while forward checking and AC3 would reduce the search effort and would more efficiently solve difficult puzzles. We hypothesize that the local search algorithms will be able to efficiently find solutions to complex puzzles, but may struggle when puzzles are simple. We discovered that our hypothesis was mostly correct; the FC and AC3 solved the puzzles the most efficiently, while BT became slow when puzzle complexity increased. However, the local search algorithms could not consistently find solutions to complex or simple problems; they got stuck a significant portion of the time.

1 Problem Statement and Hypothesis

Our goal was to complete Sudoku puzzles with varying degrees of difficulty using different AI search strategies. We wanted to find out which search strategy was the best at efficiently solving Sudoku puzzles. A Sudoku puzzle requires each row, column, and 3x3 box to contain the digits 1 through 9 without any repeated numbers. The rules are simple, but the search space that algorithms have to find a solution for can become quite large, especially as the puzzles difficulty increased. Our hypothesis was that plain backtracking would solve the easy and some medium puzzles without too much trouble, but that it would slow down significantly on the harder ones. Adding forward checking and AC3 should help by cutting off impossible options earlier, which we expected will reduce the search time. For the hardest puzzles, we believe that local search methods like simulated annealing and genetic algorithms may be well-suited; they can explore large search spaces in a different way than systematic backtracking. However, we are concerned about their performance on simple problems as the level of complexity they provide may bloat solve time in these cases. So, we think that the backtracking with forward checking and arc consistency will perform the best on the vast majority of puzzles as they are efficient and guaranteed to find a solution.

2 Algorithms Implemented

We worked with five algorithms in our solver. Three of them are based on backtracking but use different heuristics, and the other two are local search methods. Each one approaches the puzzle differently, which gave us a chance to compare their strengths and weaknesses.

2.1 Backtracking

Backtracking is the most direct approach. It models Sudoku as a constraint satisfaction problem in which

$$X = \{X_1, \dots, X_{81}\}, \quad D_i \subseteq \{1, \dots, 9\}, \quad \forall (i,j) \in E : \ C_{ij}(x_i, x_j) : \ x_i \neq x_j$$

where X is the set of cells, D_i is a cell's domain, i and j are Sudoku cells, E is the edge set of the constraint graph, C_{ij} is the binary constraint, and x_i, x_j are values for the Sudoku cells. With these variables, domains, and constraints in place, backtracking performs a depth-first search over partial assignments. This means that it selects an unassigned variable and checks whether it is locally consistent with its already-assigned neighbors:

$$\text{consistent}(i, v \mid A) \iff \forall j \in N(i) \cap \text{dom}(A) : \ v \neq A(j)$$

where i is the Sudoku cell, v is the candidate value, A is the current partial assignment, j is the already-assigned neighbor, $N(i)$ is i's neighbor set (row, column, and 3x3 grid). In less technical terms, the solver picks an empty element, tries a number, checks if it breaks any rules, and continues. If no valid number can be placed, it backtracks to the previous step and tries a different option. This guarantees a solution if one exists, but as puzzle difficulty increases the amount of trial and error can grow quickly, and therefore runtime.

2.2 Forward Checking

Forward checking improves on backtracking by updating possibilities as soon as a number is placed. Whenever the solver assigns a value, it removes that option from the domains of neighboring cells:

$$\forall j \in N(i) \setminus \text{dom}(A) : \ D_j \leftarrow \{ \, b \in D_j \ : \ C_{ij}(v, b) \, \}$$

If a neighbor loses all of its options, the solver knows immediately that the path won't work and backtracks right away:

$$\exists j \in N(i) \setminus \text{dom}(A) \ \text{ such that } \ D_j = \varnothing \implies \text{backtrack.}$$

This prevents it from going too far down paths that are already doomed.

2.3 Arc Consistency (AC3)

AC3 tries to keep the puzzle consistent as it searches. It repeatedly checks pairs of related cells for consistency

$$\forall a \in D_i \ \exists b \in D_j : \ C_{ij}(a, b).$$

and removes values that no longer have support in their neighbors:

$$D_i \leftarrow \{ \, a \in D_i \ : \ \exists b \in D_j, \ C_{ij}(a, b) \, \}.$$

Then, if D_i shrinks, (k, i) is re-enqueued for all $k \in N(i) \setminus \{j\}$. By narrowing down the possible values before backtracking, AC3 can reduce the total search effort. The tradeoff is that running AC3 takes extra work up front, so its effectiveness can vary depending on the puzzle.

2.4 Simulated Annealing

Simulated annealing takes a different approach. It begins with a complete puzzle that may not be valid and then makes random changes to reduce conflicts. Moves that improve the board are kept, and sometimes worse moves are accepted too, especially early on, which helps the solver avoid getting stuck. As the process continues, it becomes more selective, and the puzzle can eventually settle into a valid solution.

The algorithm first creates a random board and evaluates its fitness. Then a single alteration is performed and the fitness is again measured. If the new fitness is better than the previous step, the new state is taken as the base state and the process continues. If the new fitness is lower than the previous one than depending on the temperature the new board state is either accepted or rejected. If temperature is high, then the model explores more diverse options and when temperature is low, the model prefers more fit options.

Probability of acceptance is decided using Boltzmann Distribution. To solve the problem of the model sometimes being stuck close to the solution, we added a temperature control system where after a certain number of iterations the system would be increased to a higher temperature, allowing the model to again explore more options and find a better solution.

Hyperparameters:

- Initial Temperature - 1
- Cooling Rate - 0.95
- Cooling Schedule - 200
- Temperature Restart - 15000

2.5 Genetic Algorithm

The genetic algorithm uses a population of possible solutions instead of just one. Each candidate is scored by a fitness function that measures how close it is to a valid Sudoku. The better candidates are more likely to be combined and mutated to form new ones, while the weaker candidates are less likely to survive. Over many generations, the population can improve and may eventually produce a valid solution.

Hyperparameters:

- Initial Population - Boards are randomly filled using number 1-9 to create a gene. Initially, 100 genes are created and act as the first generation of genes.
- Selection - Tournament selection is used where a randomly selected group of genes are choosen and then the fittest of the group is used as parents for the crossover process. Tournament size is taken as 3/5.
- Crossover - Random one point crossover is employed where a random point on the grid is chosen and all non-initial slot is exchanged between the offspring.

- Mutation - After each crossover mutation takes place where one slot on the board has its value randomized. Mutation rate is a key parameter that we tuned for the model.

- Replacement - Generational replacement is used where the offspring completely replaces the older generation. In this process the total number of genes in each generation stays the same.

3 Experimental Setup

Our test strategy was different for the systematic search algorithms and the local search algorithms. In the case of systematic algorithms, the result can be deterministic and the model is guaranteed to solve the puzzle given sufficient time. However, in the case of local search algorithms, the approach is not deterministic and the solution largely depends on the heuristics of the initial starting grid. As such, these algorithms would be tested on how they performed rather than if they solved the puzzle or not.

For the local search algorithms, we used the following parameters to determine how they performed: assignments, backtracks, inferences, and constraint checks. We ran the models for each puzzle of each difficulty and then took the average value of the parameters. We omitted backtracking from the hard and extreme puzzles as it would take longer for this algorithm to solve the higher difficulty ones.

For the local search algorithms, we used graphs to demonstrate their performance. In the case of simulated annealing, we created graphs for fitness vs. iterations. And for Genetic Algorithm, the fitness vs. Generations graph was created.

4 Results

We ran each algorithm on puzzles from all four difficulty levels: easy, medium, hard, and extreme. Instead of reporting wall clock time, we relied on decision-based metrics to judge performance. The two most important were the number of assignments each algorithm had to make and the number of constraint checks it performed. Backtracks and inferences were also recorded, but our main comparisons focus on how assignments and constraint checks changed with puzzle difficulty. For local search methods, which do not operate in a strict step by step fashion, we examined their fitness curves to see how they improved (or failed to improve) over time.

4.1 Systematic Search Methods

The backtracking family of solvers gave us a clear sense of how quickly raw search can become taxing on the system. On easy puzzles, plain backtracking averaged about 342 assignments, while forward checking ended up with nearly the same assignment count. AC3, in contrast, solved the same puzzles with only 47 assignments, although it needed far more constraint checks than either BT or FC. Similar results were seen across all the puzzles we ran, where AC3 spent more time checking possibilities up front, but could then drastically reduce the number of guesses it needed. almost to zero.

Table 1: Average decision metrics for Easy puzzles

Algorithm	Assignments	Backtracks	Inferences	Constraint Checks
BT	342	296	0	18,513
FC	342	296	1,994	16,422
AC3	47	1	550	137,834

On the hard puzzles, we compared forward checking and AC3 directly. Forward checking required nearly 8,000 assignments and more than 40,000 inferences, while AC3 solved the same puzzles with only 74 assignments and just over 1,000 inferences. Although AC3 did need about 10 times as many constraint checks as FC , it used that overhead to solve the puzzles in significantly less guesses. Essentially, FC spent its effort making repeated assignments and pruning domains, while AC3 focused on consistency checks up front and avoided most of the search entirely. That difference in where the work was done made AC3 the more reliable solver.

Table 2: Average decision metrics for Hard puzzles (BT omitted)

Algorithm	Assignments	Backtracks	Inferences	Constraint Checks
FC	7,817	7,764	41,826	360,336
AC3	74	21	1,206	407,294

4.2 Local Search Methods

The story was very different for the local search algorithms. Simulated annealing often looked promising at the start; on easy puzzles, its fitness dropped steadily and sometimes converged to a perfect solution (Figure 1). On hard puzzles, though, it tended to flatten out after the initial improvements, leaving the solver stuck with conflicts it could not work past (Figure 2). These plateaus were common, and while SA occasionally succeeded, most runs failed to complete a puzzle within the within the step and restart limits we used.

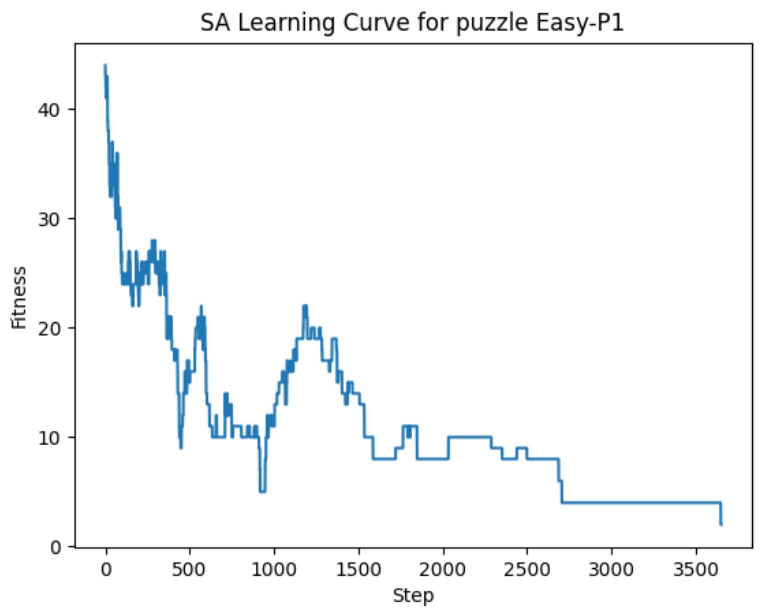

Figure 1: Simulated Annealing learning curve for an Easy puzzle.

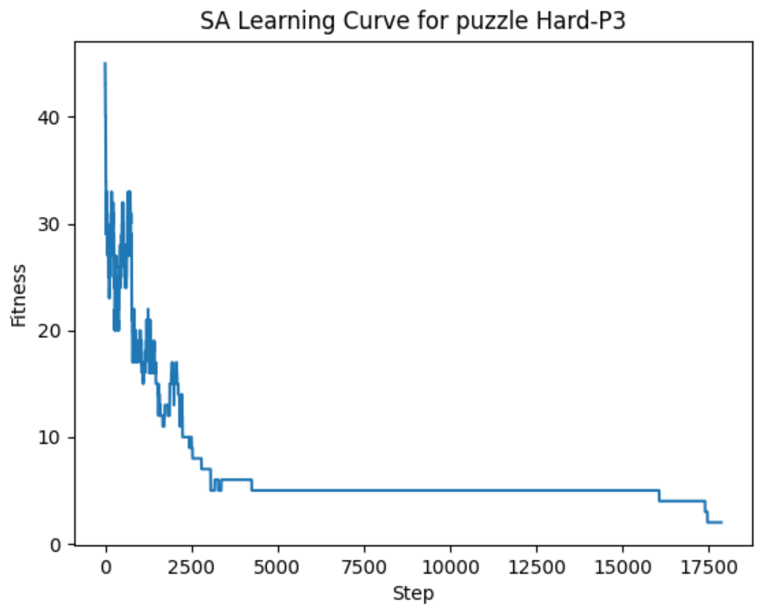

Figure 2: Simulated Annealing learning curve for a Hard puzzle.

Genetic algorithms showed nearly the same pattern. On easy puzzles, the population improved quickly in the first few generations, sometimes giving the impression that a solution was within reach (Figure 3). But on harder puzzles, progress slowed down to a snails pace and very rarely made it all the way to a valid board (Figure 4). Like SA, GA had early gains followed by long plateaus, and while it demonstrated the ability to reduce conflicts quickly, it almost always fell short of actually solving the Sudoku puzzle.

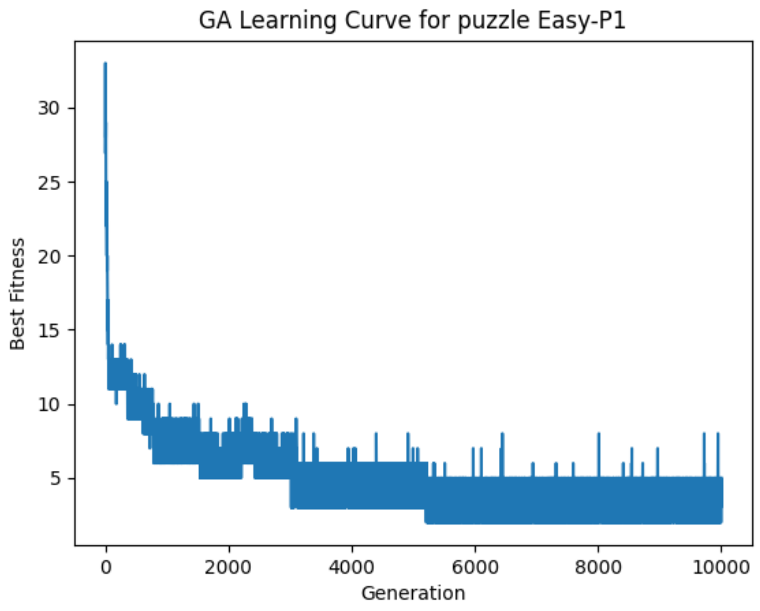

Figure 3: Genetic Algorithm learning curve for an Easy puzzle.

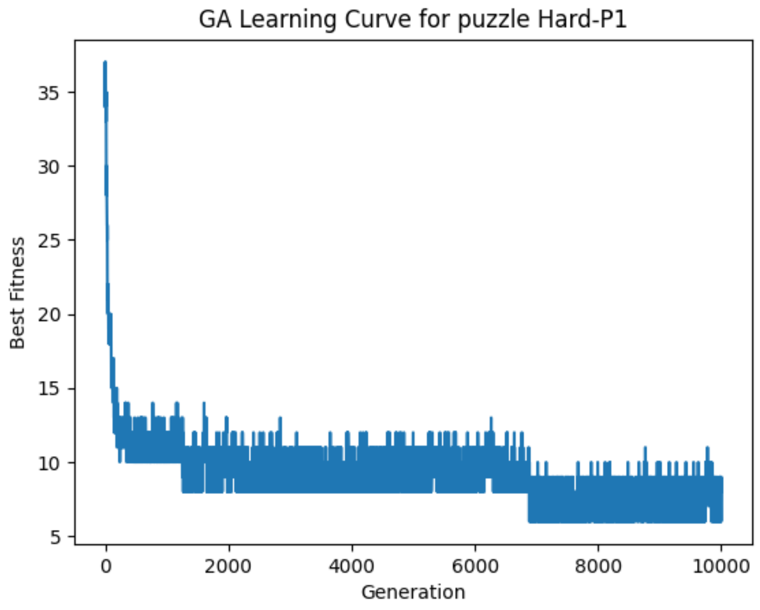

Figure 4: Genetic Algorithm learning curve for a Hard puzzle.

5 Discussion

The goal of our tests was to compare five different solvers on Sudoku and see how much work they needed to finish puzzles of different difficulty levels. Instead of timing them, we measured things like assignments and constraint checks so that the results were not tied to a specific computer or setup. This let us see which algorithms were efficient and which ones slowed down as the puzzles got harder.

The systematic solvers showed the biggest differences. Backtracking worked fine on the easy puzzles but the number of assignments went up as soon as we tried medium puzzles. Forward checking was a little better since it could cut off bad choices earlier, but it still ended up with about the same number of assignments as plain backtracking. AC3 stood out because it kept assignments very low, even when the puzzles were hard. The cost was more constraint checks, but the trade was worth it because AC3 almost always solved the puzzle without much guessing. This made AC3 the most reliable of the three to actually solve a puzzle.

The local search methods behaved differently. Simulated annealing usually improved the board a lot at the start and cut down conflicts quickly. After that progress slowed, and most runs got stuck and plateaued with a few conflicts left. The genetic algorithm worked by evolving a population of boards, and in our tests it showed the same kind of pattern of simulated annealing where there was noticeable improvement in the first set of generations, followed by long stretches where the population stopped getting better. Both SA and GA were good at reducing errors fast, but they were not dependable for finishing a puzzle.

Our results lined up with some portions of our hypothesis but not all of it. Backtracking and AC3 did behave as we had predicted, with AC3 being our most efficient algorithm overall. Forward checking helped but not quite as much as we thought it would. The local search methods were surprisingly weaker than we had initially predicted. While both SA and GA did make decent progress, they rarely finished the puzzles entirely. And were not as well suited for the hardest cases as we had hoped.

6 Summary

This project gave us the chance to build and test different AI search methods using the simple concept of a sudoku puzzle. We compared 5 different algorithms; backtracking, forward checking, arc consistency, simulated annealing, and a genetic algorithm. We used a range of puzzles that contained varying levels of initial numbers to start with and were able to compare the different algorithms in how they solved the puzzles.

From running these experiments we saw how quickly simple search methods can get bogged down, how adding constraint reasoning can make a significant difference, and how local search methods can improve quickly while solving but struggle to fully solve. Backtracking and forward checking behaved mostly how we expected them to, while AC3 performed as expected and was the best of the three systematic search methods. Simulated annealing and the genetic algorithm were especially interesting to develop, and to see how they handled solving the sudoku puzzles. While they were able to make large strides quickly, they would get "stuck" and not be able to fully complete the given puzzles.

This project allowed us to compare the trade offs between a systematic search and local search, and the benefits of each. Systematic solvers could provide reliable solutions most of the time, but still had vulnerabilities. While local search showed how different AI approaches may be able to initially explore a problem space but can not guarentee a solution.

7 Contributions

Code:

- Michael Downs - Backtracking, Forward Checking, Arc Consistency
- Sadman Sakib - Simulated Annealing, Genetic Algorithm
- Chance Larson - Metrics Definition, Data Collection, Graph Creation

Paper:

- Michael Downs - Algorithm Descriptions: Backtracking, Forward Checking, Arc Consistency
- Sadman Sakib - Algorithm Descriptions: Simulated Annealing, Genetic Algorithm
- Chance Larson - Abstract, Problem Statement and Hypothesis, Results, Graphs, Discussion, Summary