CSCI446 Project 1 - Search, Constraint Satisfaction, and Sudoku

Nicholas Kasten NICHOLAS.KASTEN@STUDENT.MONTANA.EDU

Connor Munson CONNOR.MUNSON@STUDENT.MONTANA.EDU

Dayton Wickerd DAYTON.WICKERD@STUDENT.MONTANA.EDU

Abstract

This report documents the results of implementing five different algorithms with the intent of solving Sudoku puzzles of varying difficulty. We implemented simple backtracking, backtracking with forward checking, arc consistency with backtracking, simulated annealing with a minimum conflict heuristic, and a genetic algorithm. Overall, the backtracking with forward checking algorithm performed the best, and was able to solve every puzzle every time. Simple backtracking and arc consistency performed worse than forward checking, but were still able to solve every puzzle every time. Simulated annealing was not able to solve every puzzle every time, but solved puzzles at a higher rate than the genetic algorithm, which only solved two puzzles.

1 Problem Statement and Hypothesis

The task was to solve Sudoku puzzles of 4 different difficulty levels (easy, medium, hard, and extreme) using the five algorithms: Simple Backtracking, Forward Checking, Arc Consistency, Simulated Annealing, and Genetic Algorithm.

Hypothesis: Our initial thoughts going into the project are that all the algorithms will guarantee a solution. We think that arc consistency will perform the best of the five algorithms since it contains a preprocessing step that eliminates possible values from each cell, which can solve certain puzzles without needing to backtrack. Simple backtracking will perform the worst because it is a brute force algorithm, and brute force algorithms are typically computationally intensive. We think simulated annealing will perform better than simple backtracking, due to the tuning factor, but not as well as the genetic algorithm, because the genetic algorithm has the potential for highly diverse populations, which can enforce solvability and limit search plateaus. We think that forward checking will perform slightly better than simple backtracking since it is still essentially simple backtracking, but it has small checks that can speed up the process. We expect the algorithms to perform best to worst in the following order: arc consistency, genetic algorithm, simulated annealing, forward checking, and backtracking. Overall, we expect each algorithm to solve each puzzle with some degree of variation in terms of runtime.

2 Algorithm Descriptions

We used five different algorithms to solve multiple types of Sudoku boards. Three of the algorithms use recursive backtracking, and the other two are local search algorithms.

2.1 Simple Backtracking

Simple backtracking is a brute-force search algorithm that uses depth-first search to find a solution. In the context of Sudoku, the algorithm will proceed by representing each puzzle state as a vertex in a graph. The edges between the vertices represent transitions from one state to another, or, in the case of Sudoku, placing a number in an open square.

The algorithm will then conduct a depth-first search of the graph of possible combinations of board states until a solution is reached. If one vertex (puzzle state) of the search violates a constraint (e.g, more than one of the same number in a row, column, or box), the algorithm will backtrack to the prior valid vertex and proceed to the next child of the vertex.

Backtracking is a complete algorithm, meaning it will always find a solution if the solution space is accessible. One potential issue with this algorithm is the runtime and complexity. The worst-case big-O runtime of depth-first search with backtracking is exponential.

2.2 Backtracking with Forward Checking

Forward checking is a more optimal version of simple backtracking. It builds on Backtracking by reducing the number of invalid spaces explored. In the context of Sudoku, the algorithm still represents each puzzle state as a vertex in a graph, and edges represent transitions between states, such as placing a number in an open square. However, unlike simple backtracking, forward checking immediately looks ahead whenever a value is assigned. When a number is placed in a square, forward checking updates the possible choices for the unfilled squares (their domains). If placing a number causes an unfilled square to lose all valid options (for example, a row, column, or box is left with no possible number), the algorithm will backtrack immediately, since this partial assignment cannot lead to a valid solution. It is a complete algorithm with an exponential run time at worst.

2.3 Arc Consistency

The arc consistency with backtracking algorithm is a variation of backtracking where a preprocessing step is included to prune the domains of each cell in the Sudoku puzzle. The way this works is that the arc consistency algorithm maintains a queue of constraint relationships (arcs) between cells and will loop over all the values, process them, and limit the domain (i.e, the set of values a cell can contain within the context of the constraints), and then run the backtracking algorithm on the puzzle with the reduced cell domains.

In practice, each cell will have a domain (set of integers [1,2,...9]) that is limited by the ac3 algorithm. The algorithm will use the constraints to limit the domains. For example, if one row in the initial puzzle already has the number “1”, the number “1” will be removed from all of the other domains in that row. If a cell’s domain is limited to one number, the cell can use that number later if the resulting puzzle state is valid. This will limit the

backtracking algorithm to only using the limited domains of the empty cells, instead of the entire possible domain ([1,2,...,9]) for every empty cell.

This decreases the runtime of backtracking, and for easier puzzles, the ac3 algorithm can potentially solve the puzzle entirely without needing to run the backtracking algorithm. The checking consistency of an arc can be done in $O(d^2)$ time, with the worst case time being $O(cd^3)$.

2.4 Simulated Annealing

Simulated annealing is a probabilistic search algorithm inspired by the physical process of annealing in metallurgy. In the context of Sudoku, the algorithm begins by filling the Sudoku board completely so that all the three-by-three boxes have no conflicts. Then the process of swapping values within the three-by-three boxes begins.

Swapping the values is in hopes of fixing the row and column conflicts. After each swap, the algorithm will determine a score for the board, called cost. This cost is calculated by counting the number of conflicts the board has. (for Sudoku, this might be the number of duplicate digits in rows, columns, and boxes).

If the change reduces the cost of the board, we accept the change and move forward. If the cost is worse, we will sometimes accept the new board based on a certain probability. This is to avoid being stuck at a local minima. This probability depends on a parameter called the "temperature," which starts high and decreases over time. At high temperatures, worse solutions are more likely to be accepted. As the "temperature" decreases the algorithm will start to favor better solutions. This gradually moves towards an optimal solution. Simulated annealing is not a complete algorithm and is not guaranteed to solve the puzzle as it is constrained to its predefined number of iterations.

2.5 Genetic Algorithm

This particular system demands a genetic algorithm with a penalty function and tournament selection. In terms of Sudoku, the algorithm will start by using the initial puzzle to create a population of randomly-filled puzzles without changing the initial provided values. Then, the population will simulate a biological reproduction, or a survival of the fittest. Each individual of the population will be evaluated for its fitness. In the context of Sudoku, the fitness (or penalty) function will evaluate the extent of the errors within each individual. An individual with a higher fitness score has more errors than an individual with a lower fitness score. The overall goal of this algorithm is to simulate reproduction on the population and, by doing so, reduce the fitness score of the population to find an individual with a score of zero. Simulating reproduction means repeating the following steps until an ideal individual is found (the problem is solved) or the maximum number of generations is reached:

- Use tournament selection to choose parents, where the tournament function will randomly select a sample of individuals, and compare their fitness to choose the best parent from that sample. This is repeated twice to get two parents.
- Once the two parents are selected, they will undergo a recombination, where they "reproduce" to create a new individual. This recombination can be done in a few

different ways. In terms of Sudoku, the recombination function will be tuned to favor certain aspects of each parent. For example, the top three rows of parent A are substituted for the top three rows of parent B. The recombination can be done in several ways, and there will be parameters that control the intensity of which parent's attributes are used more frequently.

- Once the recombination is done, there will be a chance that the new individual is mutated. In the context of Sudoku, this means that a cell, row, column, or box can be modified at random to ensure genetic diversity.

Each of the individuals in the new generation will replace a member of the previous generation. The final generation will either have an individual with a penalty (fitness) score of 0 (a solution to the puzzle) or the maximum number of generations will have been reached. In the case where the maximum number of generations is reached, parameters will be tuned accordingly. The runtime of this algorithm depends on the size of the population and the halting conditions. Apart from the algorithms, the system also requires some input processing and output management. The system will be able to take a puzzle in the form of a .txt file and use the data encoded in the file so that the algorithms can access the puzzles from the path provided. The system will also have a puzzle output function, which will convert the completed puzzle back into a .txt file

3 Experimental Approach

Each algorithm required a different experimental approach. Below each Algorithms experimental approach is outlined.

3.1 Simple Backtracking

For Simple Backtracking, we tracked 3 main variables: decisions, checks, and backtracks.Decisions tracked the number of attempted assignments or numbers placed. A check is any time the board is analyzed to see if the board does not violate the constraints. The final variable we measured was backtracks. A backtrack is anytime the algorithm had to return to an earlier state to attempt a new assignment. the lower these variable counts are, the faster the algorithm solves. These are good metrics to compare to each other between algorithms.

3.2 Forward Checking

For Forward Checking, we tracked 4 main variables: decisions, checks, backtracks, and prunes. Decisions, checks, and backtracks were implemented the same way as in Simple Backtracking. Prunes was a new variable added with Forward Checking. Prunes measure the number of domain values removed in the forward checking process. Decisions, checks, and backtracks are implemented so we can compare to Simple Backtracking. While Prunes is implemented so we can compare to Arc Consistency.

3.3 Arc Consistency

For Arc Consistency, we tracked 4 main variables: decisions, checks, backtracks, and prunes. These variables are implemented the same way as Forward Checking. Our main goal with

these variables is to be able to compare results to both Forward Checking and Simple Backtracking.

3.4 Simulated Annealing

For simulated Annealing, we had to take a bit of a different approach. We tracked 3 main variables: Move attempts, Iterations, and final cost. Move attempts allow us to compare to the previous 3 algorithms' decisions variable. While final cost and iterations are going to be used to compare to the genetic algorithm.

We tuned this algorithm using a grid search, where we selected a range of values for each hyperparameter and monitored the performance of the algorithm based on each combination of hyperparameters. The details of our approach can be found in table **??**.

3.5 Genetic Algorithm

For the Genetic algorithm we took a similar approach to Simulated Annealing. Offspring evaluated, Generations, and Best Fitness. Offspring evaluated is used to compare it to the attempted assignments for Backtracking, Forward Checking, and Arc Consistency. We use Generations to compare to iterations of Simulated annealing, and we use the best fitness to compare to the final cost.

We tuned this algorithm using a grid search, where we selected a range of values for each hyperparameter and monitored the performance of the algorithm based on each combination of hyperparameters. The details of our approach can be found in table **??**.

4 Results

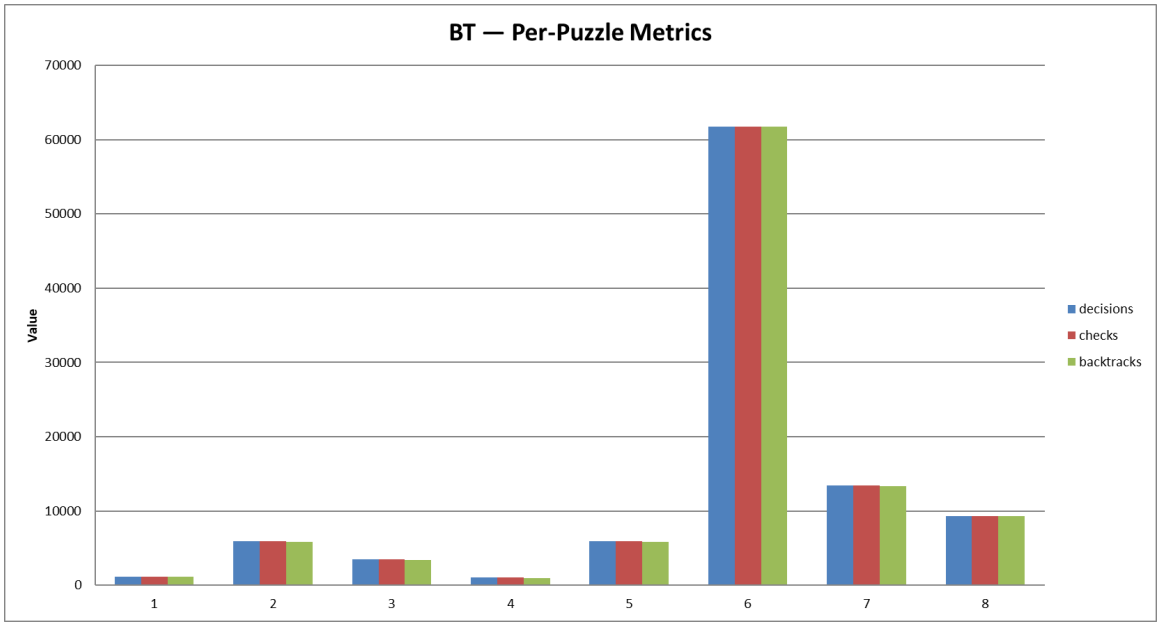

Figure 1: Simple backtracking metrics on all puzzles of easy and medium difficulty

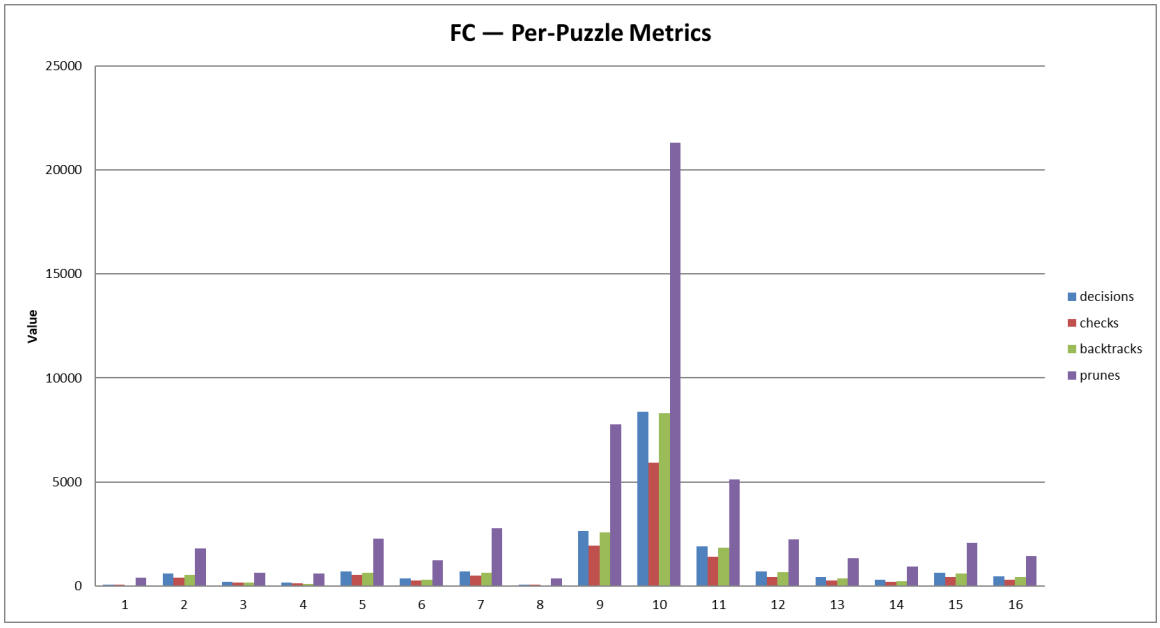

Figure 2: Forward Checking metrics of every puzzle provided

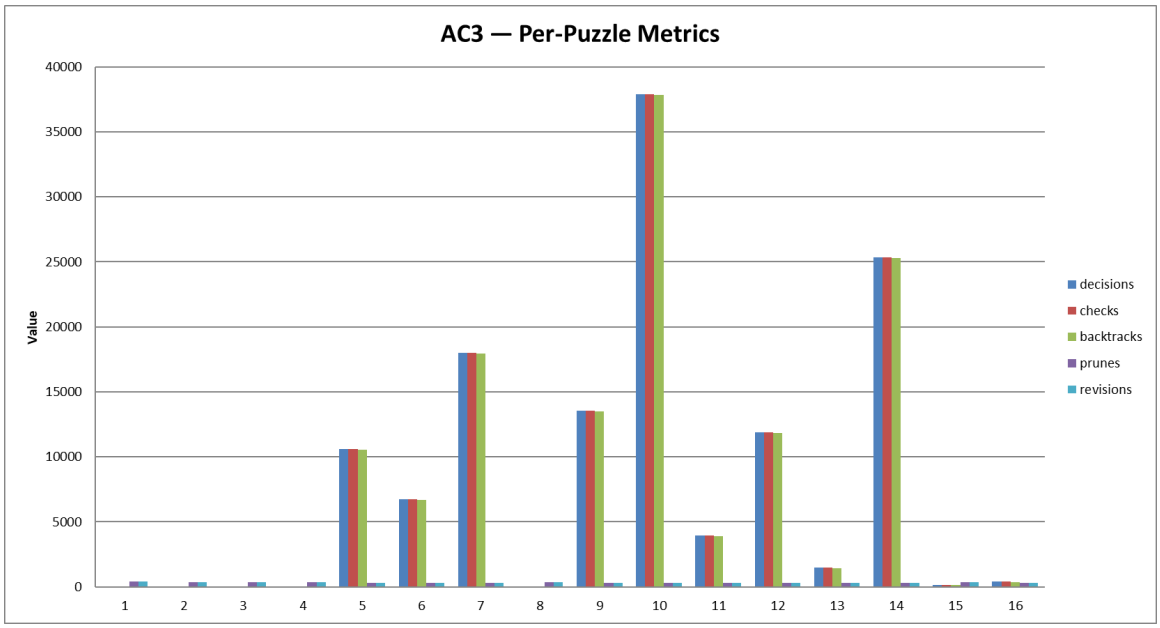

Figure 3: Arc Consistency metrics for every puzzle provided

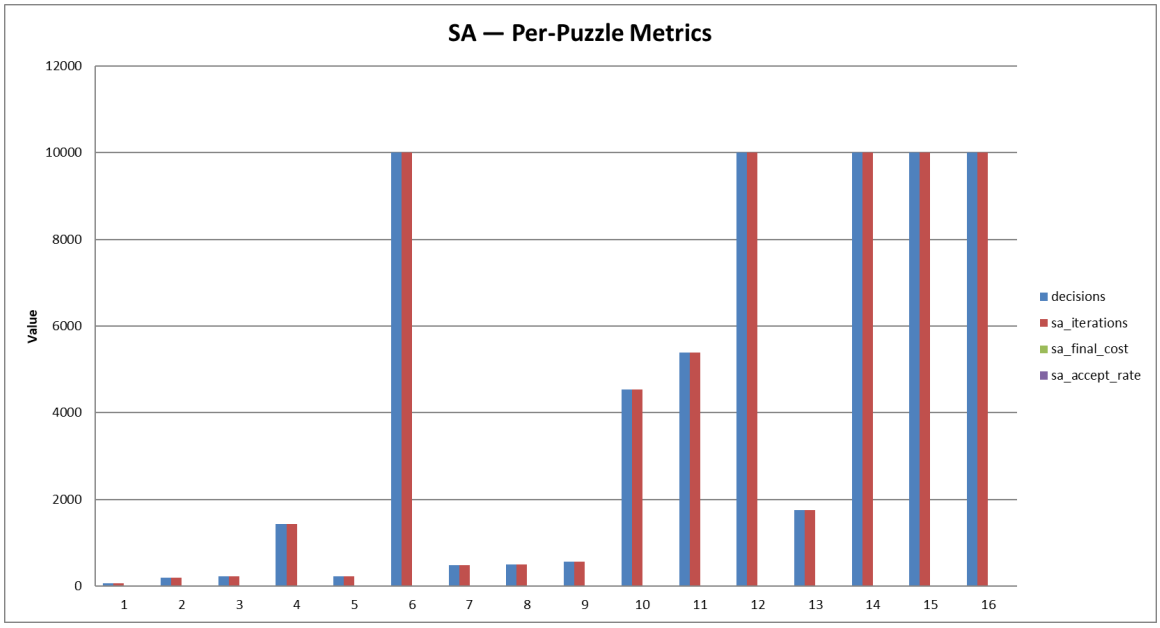

Figure 4: Simulated Annealing metrics for every puzzle provided

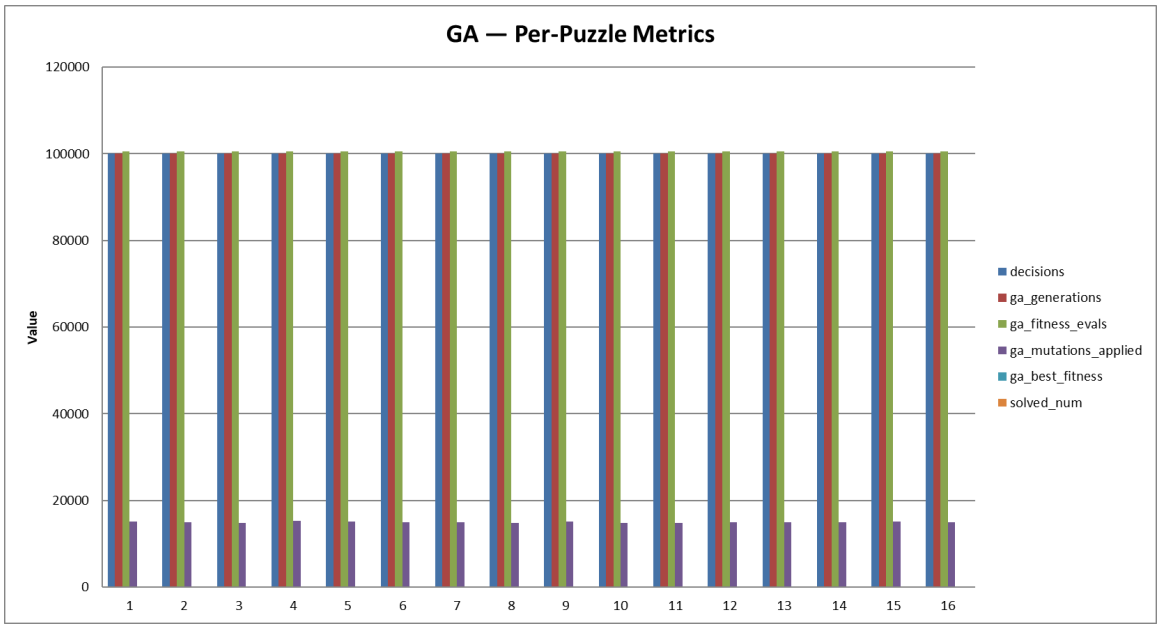

Figure 5: Genetic Algorithm metrics for every puzzle provided

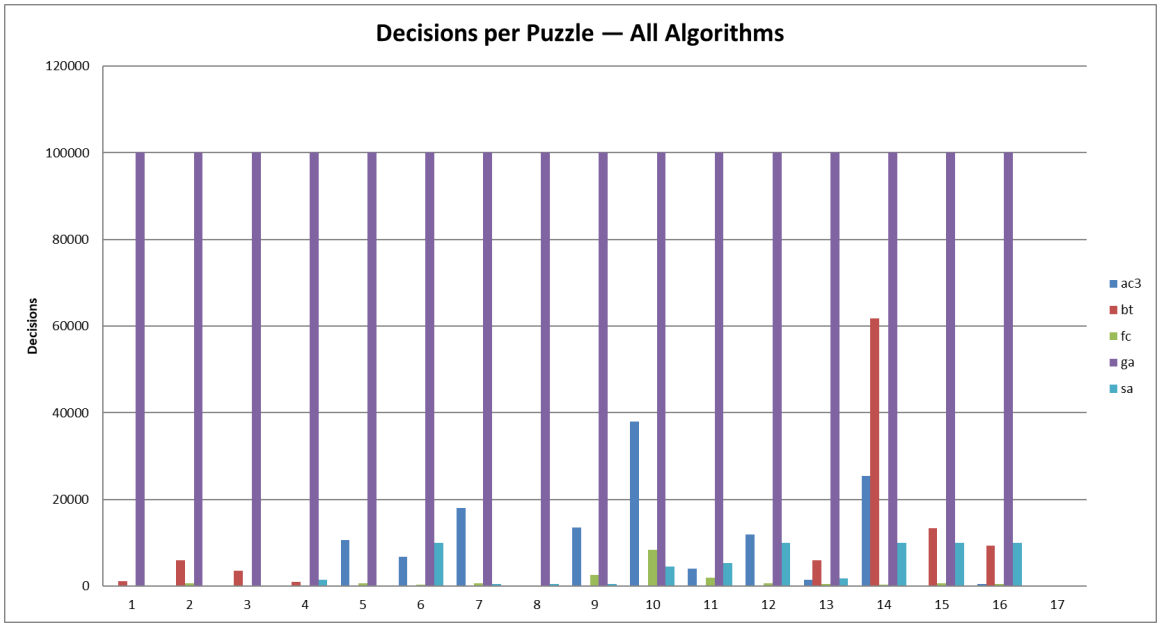

Figure 6: Decisions of every algorithm on every puzzle provided

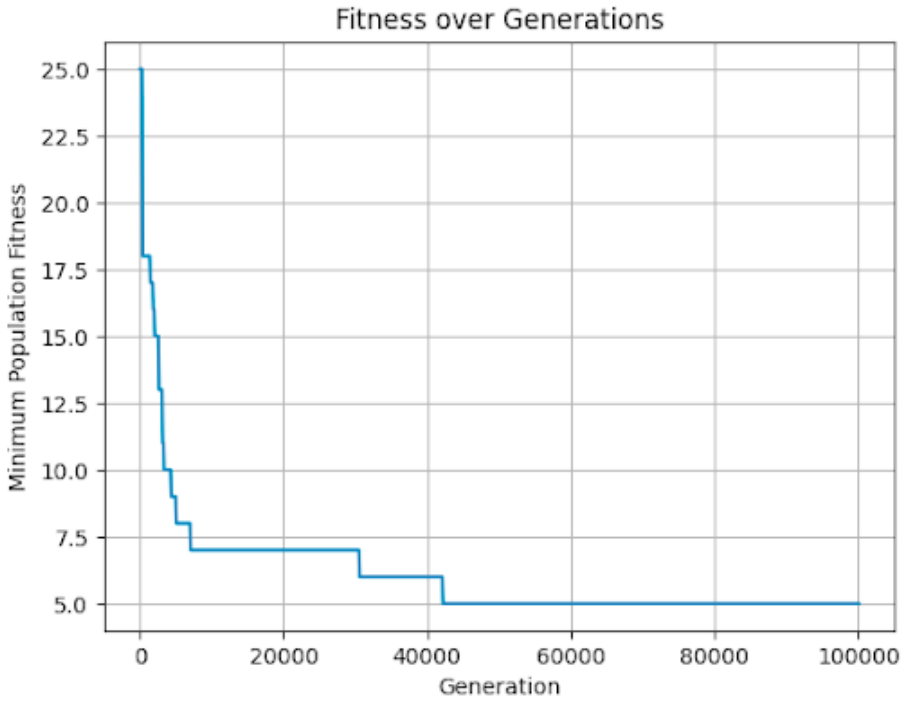

Figure 7: GA Fitness over generations for Puzzle Easy 01

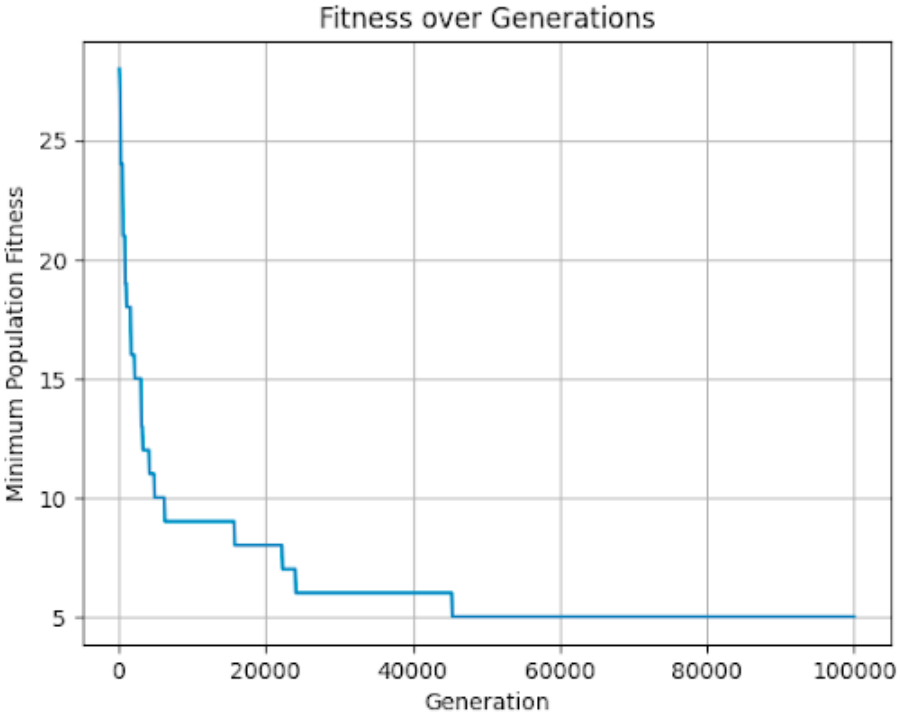

Figure 8: GA Fitness over generations for Puzzle Medium 01

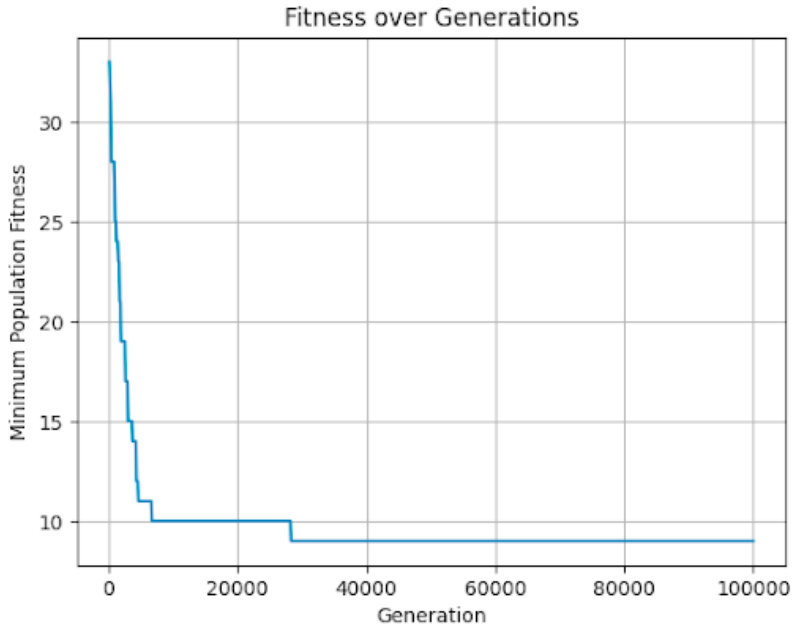

Figure 9: GA Fitness over generations for Puzzle Hard 01

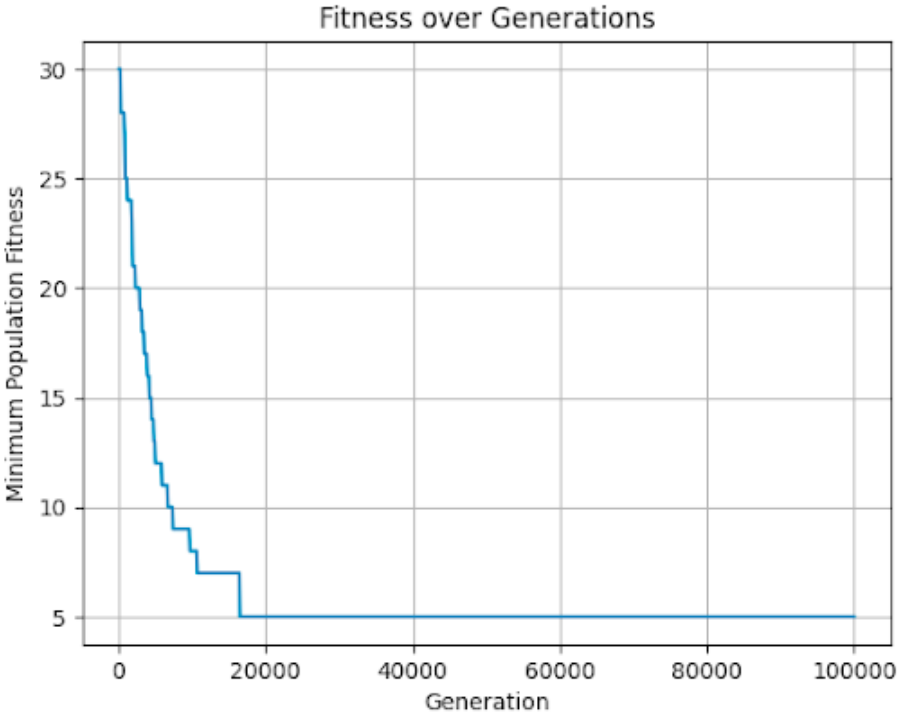

Figure 10: GA Fitness over generations for Puzzle Extreme 03

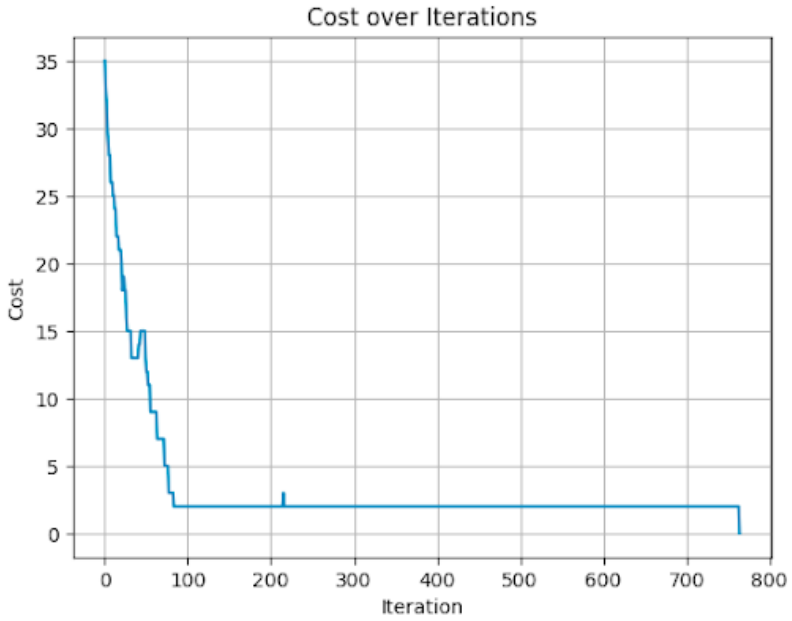

Figure 11: SA cost over iterations for Puzzle Easy 01

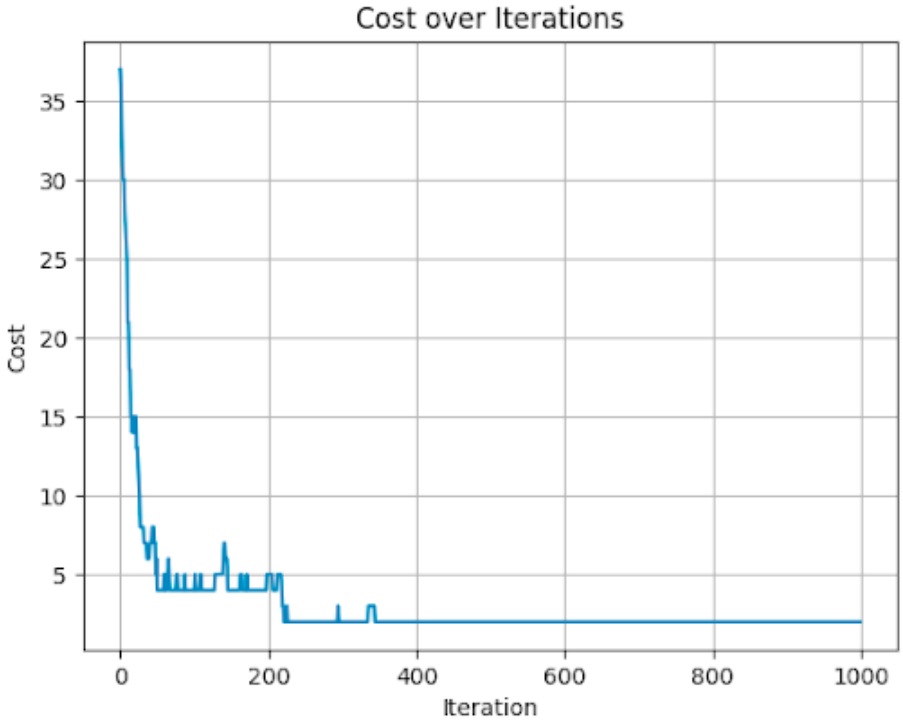

Figure 12: SA cost over iterations for Puzzle Medium 01

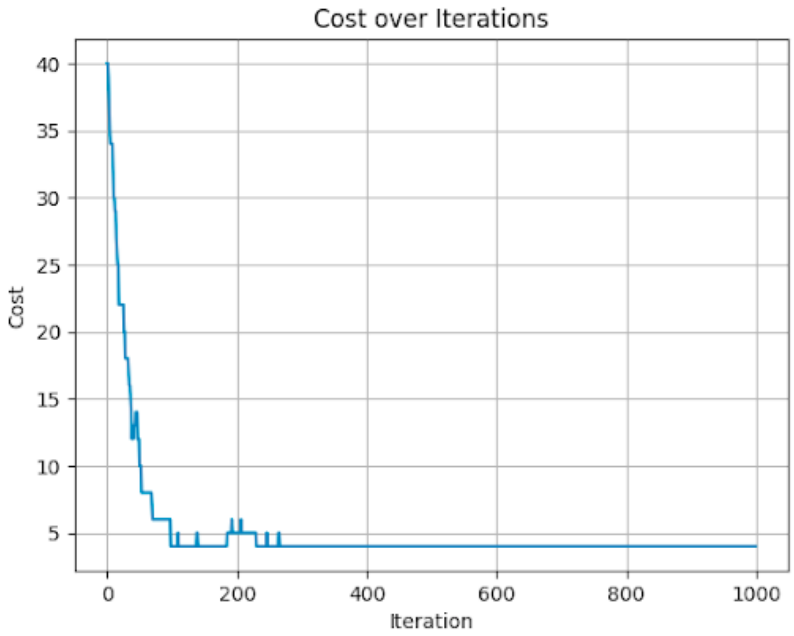

Figure 13: SA cost over iterations for Puzzle Hard 01

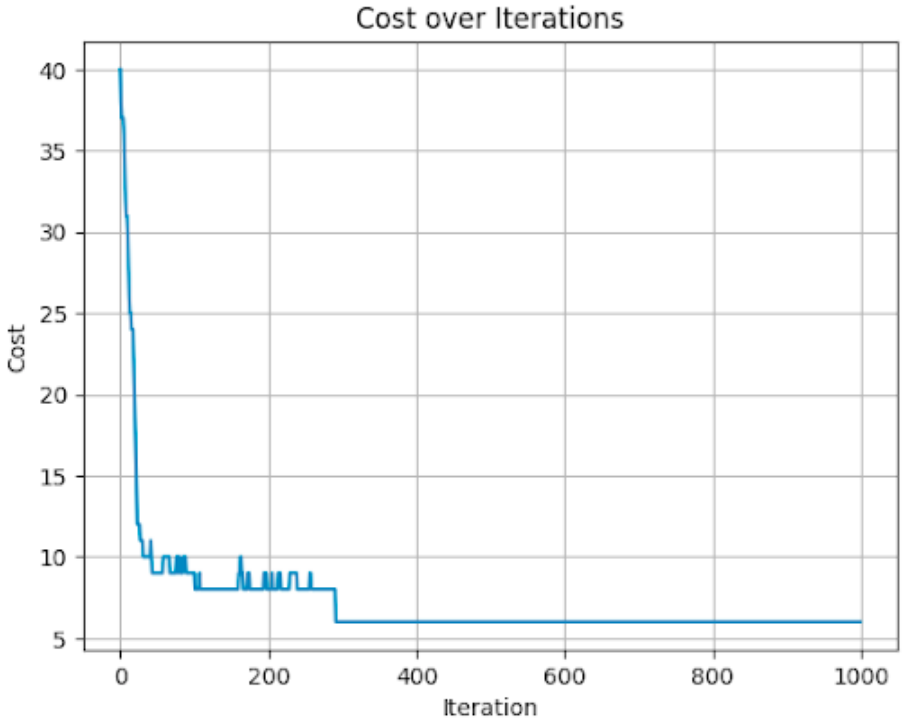

Figure 14: SA cost over iterations for Puzzle Extreme 03

Table 1: BT summary statistics

Metric	Median	Mean	Std
decisions	5896.00	12739.38	20243.17
checks	5896.00	12739.38	20243.17
backtracks	5848.50	12690.50	20242.24

Table 2: FC summary statistics

Metric	Median	Mean	Std
decisions	529.50	1139.31	2045.59
checks	352.00	809.25	1455.88
backtracks	481.00	1088.44	2044.53
prunes	1614.50	3269.44	5183.66

Table 3: AC-3 summary statistics

Metric	Median	Mean	Std
decisions	2716.00	8126.88	11097.62
checks	2716.00	8126.88	11097.62
backtracks	2670.00	8092.69	11084.05
prunes	314.50	323.63	34.57
revisions	314.50	323.63	34.57

Table 4: SA summary statistics

Metric	Median	Mean	Std
decisions	1593.00	4084.94	4386.18
sa_iterations	1593.00	4084.94	4386.18
sa_final_cost	0.00	0.75	1.24
sa_accept_rate	0.89	0.87	0.09

Table 5: GA summary statistics

Metric	Median	Mean	Std
decisions	100000.00	100000.00	0.00
ga_generations	100000.00	100000.00	0.00
ga_fitness_evals	100500.00	100500.00	0.00
ga_mutations_applied	14973.00	14991.75	123.41
ga_best_fitness	5.50	5.50	2.31

4.1 Tuning Results

We used a grid-search approach to tune the hyperparameters for both simulated annealing and the genetic algorithm. We picked out a set of values that we thought covered a reasonable range for each algorithm.

We did three grid-search runs for simulated annealing. We ran the grid search on extreme puzzle 1, since we thought that tuning on a harder puzzle would improve performance as a whole. We tuned three hyperparameters: the maximum iterations for a single run of simulated annealing, the temperature, which is the main variable in the algorithm that calculates the probability of accepting a worse puzzle state, and the cooling rate is the rate at which the temperature changes every iteration. In the first run, we used the following values as our hyperparameter ranges:

- Maximum iterations: [1,000, 5,000, 10,000, 50,000]
- Temperatures: [0.5, 1.0, 1.5, 2, 2.5]
- Cooling rates: [0.99, 0.995, 0.999, 0.9995]

The results of our first grid search algorithm with a best cost of 0 were:

- Maximum iterations: 5000
- Temperature: 0.5
- Cooling rate: 0.995

The first grid search resulted in a large number of solved puzzles, many of which were on completely different sets of hyperparameters, so to further tune the hyperparameters, we limited the ranges of each hyperparameter as follows:

- Maximum iterations: [5,000, 10,000, 50,000]
- Temperatures: [0.5, 1.0, 1.5, 2]
- Cooling rates: [0.995, 0.999, 0.9995]

We ran grid search again, and the results are as follows:

- Maximum iterations: 5000
- Temperature: 0.5
- Cooling rate: 0.999

The second run did not solve any puzzles, as the best cost that we found was a score of 2. This was surprising, so we again expanded the hyperparameters as follows:

- Maximum iterations: [5,000, 7,000, 10,000, 50,000]
- Temperatures: [0.5, 1, 1.5, 2]
- Cooling rates: [0.995, 0.999, 0.9995, 0.9999]

This final run of grid search returned a set of hyperparameters with a solved puzzle as follows:

- Maximum iterations: 10,000
- Temperature: 1.5
- Cooling rate: 0.9999

We opted to use this set of hyperparameters, as after further testing, it performed relatively well on all puzzle difficulties. We also conducted a grid search for the genetic algorithm. Using a similar method, we ran the genetic algorithm on extreme puzzle 1 for the same reasons. We tuned four different hyperparameters for the genetic algorithm: population size, mutation rate, number of tournament selections, and maximum generations. We conducted three total runs, and the results of these runs are as follows:

Run 1 parameter ranges:

- Population sizes: [50, 100, 200, 500]
- Mutation rates: [0.1, 0.15, 0.2, 0.25]

- Tournament selections: [2, 3, 5, 7]
- Maximum generations: [50,000, 75,000, 100,000]

Run 1 parameter results with a best fitness of 2:

- Population size: 200
- Mutation rate: 0.1
- Tournament selections: 7
- Maximum generations: 75,000

This run of grid search had multiple results with a best fitness score of two, so we narrowed the parameters to the following values:

- Population sizes: [200, 500]
- Mutation rates: [0.1, 0.15, 0.2]
- Tournament selections: [3, 7]
- Maximum generations: [75,000, 100,000]

Run 2 results with a best fitness of 2:

- Population size: 500
- Mutation rate: 0.1
- Tournament selection: 3
- Maximum generations: 100,000

Again, this run had two different results with a best fitness of two, so we further narrowed the hyperparameters to the following values:

- Population sizes: [200, 500, 750]
- Mutation rates: [0.1, 0.15]
- Tournament selections: [3]
- Maximum generations: [75,000, 100,000]

This final grid search resulted in one of our only solved puzzles using the genetic algorithm. The hyperparameters that caused this solve are as follows:

- Population size: 500
- Mutation rate: 0.15
- Tournament selection: 3
- Maximum generations: 100,000

We decided to use these hyperparameters for all of our genetic algorithm runs since they resulted in a solved puzzle, which had not been done until this point.

5 Discussion and Conclusion

We found that only four of the five algorithms could reliably solve each puzzle. Simple backtracking, forward checking, arc consistency, and simulated annealing all solved each of the provided puzzles. Each of the four algorithms that could solve each puzzle performed well, with each algorithm solving the puzzle in a short time. The genetic algorithm only solved two of the hundreds of iterations we attempted.

5.1 Simple Backtracking, Forward Checking, and Arc Consistency

In general, simple backtracking and forward checking exceeded our expectations. Arc consistency met our expectations, but it underperformed compared to forward checking in some respects. These three algorithms solved every single puzzle faster than simulated annealing and the genetic algorithm. We attribute the success of these algorithms to the completeness of backtracking, forward checking, and arc consistency, meaning that if a solution exists, these algorithms will find it.

5.1.1 Backtracks

According to table 1, the median number of backtracks for all easy and medium puzzles of the simple backtracking algorithm is 5,848.5 backtracks. This indicates that the fiftieth percentile of puzzles solved by backtracking uses almost 6,000 backtracks. Compared to the other backtracking algorithms (forward checking, arc consistency) on the same puzzles (easy and medium), forward checking only used a median of 306 backtracks (Table 2), and arc consistency only used a median of 62 backtracks (Table 3). This shows that arc consistency and forward checking perform better than simple backtracking, with arc consistency performing better than forward checking across the easy and medium puzzles. This is consistent with our hypothesis since arc consistency can solve easy puzzles without needing to backtrack.

Including the remaining puzzle types (easy, medium, hard, extreme), forward checking outperformed arc consistency in the number of backtracks, which came as a surprise to us. Across all puzzles, forward checking had a median of 481 (Table 2) backtracks, while arc consistency had a median of 2,670 (Table 3) backtracks. This challenges our hypothesis. We thought that arc consistency would perform better than forward checking because all of the domain checks were done before the backtracking. We figured that doing domain checks before backtracking would put arc consistency at a better starting point in the solution space compared to forward checking, leading to fewer backtracks.

A possible explanation for forward checking outperforming arc consistency is that domain checking in arc consistency can clear easy puzzles without backtracking, but the domain checking of arc consistency can be limited in medium, hard, and extreme puzzles. Forward checking does all of the domain checks while backtracking, which could result in a more efficient use of those domain checks. Additionally, arc consistency performed remarkably poor on medium puzzle 2, which could skew the data. Arc consistency had a total of 25,298 backtracks on medium puzzle 2, which is inconsistent with the other medium puzzles by more than a factor of 10. The average number of backtracks for the other medium puzzles is only 641.67 backtracks. Without medium puzzle 2, arc consistency had a median of

1,429 backtracks, which is still worse than forward checking (Table 3). Overall, we did not expect forward checking to outperform arc consistency, but we did expect forward checking and arc consistency to outperform simple backtracking.

5.1.2 Validity Checks

According to table 1, the median number of checks for all easy and medium puzzles of the simple backtracking algorithm is 5896.5 checks. This indicated that the fiftieth percentile of puzzles solved by backtracking uses almost 6000 checks. Compared to the other backtracking algorithms (forward checking, arc consistency), on the same puzzles (easy and medium), forward checking only used a median of 232.5 checks(Table 2), while arc consistency had a median of 81. This shows that arc consistency and forward checking perform better than simple backtracking, with arc consistency performing better than forward checking across the easy and medium puzzles.

Including the remaining puzzle types (easy, medium, hard, extreme), forward checking outperformed arc consistency in the number of checks. This was a surprise to us. Across all puzzles, forward checking had a median of 352 (Table 2) checks, while arc consistency had a median of 2716 (Table 3) checks. This goes against our hypothesis. We thought that arc consistency would have to perform fewer validity checks than forward checking, as we thought the initial reduction of the domains would allow for quicker solves for all problem sets.

We believe this occurs because, as the problems become more difficult, the initial domain reductions from arc consistency have less impact, since fewer cells can be narrowed down to a single value.

5.1.3 Prunes

The prunes statistic only applies to the forward checking and arc consistency algorithms as defined in section 3. According to table 2, forward checking pruned a median of 1,614.5 values from cell domains. Arc consistency, in comparison, pruned a median of 314.5 values from cell domains (Table 3). It can be interpreted that arc consistency performs better than forward checking because arc consistency does not waste any time on unnecessary prunes. Forward checking is designed to naturally make more prunes since forward checking prunes domains while backtracking, and since backtracking can traverse paths that do not lead to a solution, forward checking will have more prunes than arc consistency. This is consistent with our conclusion, as we expected arc consistency to perform better than forward checking overall.

5.2 Simulated Annealing and Genetic Algorithm

5.2.1 Cost v.s Fitness

The cost and fitness statistics only apply to simulated annealing and the genetic algorithm. According to table 4, simulated annealing had a median cost of 0. However, the genetic algorithm had a median final fitness of 5.5 5. This goes against our original hypothesis in two ways. We thought that the genetic algorithm would perform better than simulated

annealing, and we thought that every algorithm would consistently solve. Not only did the genetic algorithm perform worse than simulated annealing, it also did not solve consistently.

5.2.2 Iterations v.s Generations

The iteration and generation statistics are only relevant for simulated annealing and the genetic algorithm. As shown in Table 4, simulated annealing required a median of 1,593 iterations to reach a solution. In contrast, the genetic algorithm had a median of 100,000 generations (Table 2), reflecting the fact that it often failed to consistently solve the puzzles. On the occasions when the genetic algorithm did succeed, it required 18,319 generations—still far less efficient than simulated annealing's median of 1,593 iterations. These results once again contradict our initial hypothesis, as simulated annealing outperformed the genetic algorithm, which struggled with consistency.

5.3 Conclusion

Hypothesis recap. We hypothesized that all five algorithms would be able to solve all given puzzles, with performance ordered best to worst: arc consistency, genetic algorithm, simulated annealing, forward checking, and backtracking.

What we built. We implemented five algorithms (BT, FC, AC3, SA, GA) and evaluated them on 4 easy, medium, hard, and extreme Sudoku puzzles. We counted **decisions** as a common metric across all five algorithms, which was either an attempted assignment or a move attempt. For backtracking variants (FC and AC3) we also tracked checks, backtracks, prunes, and revisions. For SA we tracked iterations, acceptance rate, and final cost. For GA we tracked generations, fitness evaluations, mutations, and best fitness. SA and GA were tuned using grid searches (as detailed earlier).

What happened. Four algorithms (BT, FC, AC3, SA) were able to reliably solve every puzzle. GA was only able to solve a puzzle twice over hundreds of runs under our hyperparameters. With the backtracking algorithms on all puzzles, FC had significantly less backtracks than BT and fewer than AC3 (Tables **??**, **??**, **??**). AC3 did the best on easy and medium puzzles, which might be contributed to forcing a value to be placed if the domain size was reduced to a single option. In our testing, SA had a median final cost of 0 and 1593 decisions (Table **??**), and the cost over iterations curve show a fast drop then plateau, usually due to being stuck in a local minimum. GA had a median of 100000 decisions and would drop down to a mean best fitness of around five. Using the hyperparameters discussed earlier, each run would get stuck in solving the puzzle before the max number of generations was reached.

Completeness and constraint propagation was much stronger for the puzzles. BT is guaranteed to find a solution, FC and AC3 are variants that only improve the performance with prunes that can potentially solve easier puzzles before ever needed to do backtracking. AC3's pruning performance when it comes to harder puzzles starts to decline potentially due to the initial domain reductions being weaker, while FC pruning during search appears to perform better. With a high acceptance rate early on for SA, we found that it was able to avoid getting stuck in a local minimum and routinely find a solution to the puzzle given. GA's performance relied heavily on mutation which if not tuned optimally, results in it getting stuck and unable to complete the puzzle.

For Sudoku, constraint based backtracking methods appear to be the most reliable under the metrics and puzzles tested in this project. SA worked well considering it does not guarantee completeness and often reached a final cost of zero. Given our hyperparameters for GA, we were never close to the other four algorithms in terms of actually solving the puzzle or performance. Future iterations could include additional diversity, repair, mutation rates and types to improve performance and reduce the chance of getting stuck in a local minimum.

6 Summary

We were tasked with creating five different algorithms that could solve Sudoku puzzles of varying difficulty (easy, medium, hard, extreme). The five algorithms we implemented were simple backtracking, backtracking with forward checking, arc consistency with backtracking, simulated annealing with a minimum conflict heuristic, and a genetic algorithm. We expected each algorithm to solve each puzzle with performance as follows:

1 Arc Consistency with Backtracking

2 Genetic Algorithm

3 Simulated Annealing

4 Backtracking with Forward Checking

5 Simple Backtracking

We found that not all of the algorithms could solve every puzzle reliably. The backtracking algorithms (simple backtracking, forward checking, and arc consistency) were able to solve every puzzle with differing levels of performance. Forward checking performed the best overall, followed by arc consistency and simple backtracking. The local search algorithms (simulated annealing and the genetic algorithm) suffered in terms of solving puzzles, with simulated annealing solving at a significantly higher rate than the genetic algorithm, but much worse than the backtracking algorithms. The genetic algorithm performed the worst overall, with only two documented solves. Overall, we can reject our hypothesis on the grounds that the genetic algorithm performed much worse than expected, and forward checking performed the best of all the algorithms.

7 References

D., David. "Solving Sudoku Puzzles with Genetic Algorithm." Road to ML, 9 Nov. 2019, `nidragedd.github.io/sudoku-genetics/`.

GNU.org. "Simulated Annealing." https://www.gnu.org, `www.gnu.org/software/gsl/doc/html/siman.html`.

Russell, Stuart, and Peter Norvig. Artificial Intelligence: A Modern Approach. 4th ed., Pearson Higher Ed, 2021.

Solving Sudoku As a Constraint Satisfaction Problem Using Constraint Propagation with Arc-Consistency Checking and then Backtracking with Minimum Remaining Value Heuristic and Forward Checking in Python. Sandipanweb, 18 Mar. 2017, sandipanweb.wordpress.com/2017/03/17/solving-sudoku-as-a-constraint-satisfaction-probl

Solving Sudoku with Consistency. ConSystLab, consystlab.unl.edu/Documents/Papers/IH-IJCAI18.pdf.

Sudoku Solver Using Contraint Propagation and Backtracking. Sudokupy, aurbano.github.io/sudoku_py/.

Why Are Integers Immutable in Python? Stack Overflow, stackoverflow.com/questions/37535694/why-are-integers-immutable-in-python.