Sudoku Solver: Local Search, Genetic Algorithm, and Backtracking

Analysis of different search algorithms

Beomjun Bae, Sakshi Agarwal, Evrim Ozel, and Mihnea Chirila

CS271 - Artificial Intelligence
Final Project Report
December 7th, 2019

1 Problem Description

1.1 General Overview

Sudoku is a numerical combinatorial puzzle game. Given a 2 dimensional square grid, the solver is asked to find find an arrangement of numbers that satisfy a specific set of constraints. Here are the set of constraints defined below:

1. Each cell must contain only one number.
2. Each row of cells may not have a repeating number.
3. Each column of cells may not have a repeating number.
4. Each sub-grid is may not have a repeating number.

5	3			7				
6			1	0	5			
	0	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	0			5
				8			7	0

Figure 1: An Example Sudoku Board

In the last constraint, a sub-grid is referring to the smaller sized grid of length $\sqrt{N}$ given that the total length of the board is N. For example, a board of size $N = 9$ will have a sub-grid of size 3, as evident in Figure 1 above. The goal is to develop a solver that could solve a board of size $N = 25$. Consequently, each cell will contain a single integer ranging between 0 and 24 unlike the example in Figure 1.

1.2 Sudoku Board Definition

5	3	-1	-1	7	-1	-1	-1	-1
6	-1	-1	1	0	5	-1	-1	-1
-1	0	8	-1	-1	-1	-1	6	-1
8	-1	-1	-1	6	-1	-1	-1	3
4	-1	-1	8	-1	3	-1	-1	1
7	-1	-1	-1	2	-1	-1	-1	6
-1	6	-1	-1	-1	-1	2	8	-1
-1	-1	-1	4	1	0	-1	-1	5
-1	-1	-1	-1	8	-1	-1	7	0

Figure 2: Board that the Algorithm Sees

To solve this problem, we needed a data structure that could hold all of the information given in the initial board. Therefore, we created a 2 dimensional integer grid with size N by N with every cell filled with an integer from the range $[-1, N)$. Note that -1 is included and N is excluded from the possible values. This is because -1 represents an empty space on the board, and all the other values represent the numbers that the cell should have. Therefore, instead of the figure above, the algorithm would be seeing a board that look like Figure 2.

A nice property we get from this definition of a Sudoku board is that we get to coordinate each cell with two integers. For

example, if we were to get the Figure 1 Sudoku puzzle as an initial board, we can define the first cell with the pair of integers $(1, 0)$ which would have the value 6.

2 Approach

Our first observation of the Sudoku problem was that it can be boiled down into constraint satisfaction problem with search. The constraints are the 4 mentioned above, and the searching algorithm is needed because we need to find the correct arrangement of the numbers within a big search space that is the Sudoku puzzle.

For example, take a look another look at Figure 1. The cell $(2, 0)$ has the value -1, so it needs a value assignment. With all 4 of the constraint applied to the cell, we can see that the only possible values that can take place are 1 and 2. With same logic, cell $(1, 2)$ can take the values 2, 4, and 7. With all these different possibilities, there are different combinations of the number assignment that the board can have. We essentially have narrowed down Sudoku into a problem of finding the specific combination of all the possible combinations of board number assignment.

However, this search problem is no easy task. We estimate that there are N possible values that could be assigned to any given cell (x, y), and there are N^2 different cells that we need to assign values. Form these assumptions, we can see that the entire search space would be $N^{(N^2)}$. To give it a perspective of a worst case scenario, a mere 9 by 9 board would have 9^{81} different possibility, which a 77 digit number.

3 Local Search with Simulated Annealing

The first algorithm that we developed to solve Sudoku is a local search method with simulated annealing to introduce randomness.

3.1 Strength and Weaknesses of Local Search

A local search method takes a given state, S, and modifies is in small increments to find a close neighbor of the given state, S'. Then, we can either accept or reject new state S' dependent on whether we find it closer to the goal state. This algorithm is powerful in that it does not require a large space, because at most it needs to remember two states at any given moment. However, a big drawback is that it has the possibility of being stuck in a local extrema.

For example, in the context of the Sudoku board, we might reach a certain point where making any changes will make the board violate more constraints while the current state is not the goal state we are looking for. To combat this problem, we must introduce some randomness to make sure we get away from this local extrema. This is where we use simulated annealing.

3.2 Description of Simulated Annealing

Simulated Annealing is an optimization technique, inspired by the annealing process used to find a strengthened chemical structure given an initial metal. This is done by carefully and slowly cooling a metal from an immensely high temperature. The 2 properties of this annealing that must be highlighted are the fact that initially the metal is heated and that it is cooled slowly.

This idea of temperature management is precisely how we are going to introduce randomness to our local search. At first, we want to introduce the maximum amount of randomness, but as we start to find neighbors that we evaluate to be better, we slowly decrease the randomness causing the problem to zeroing in on a specific solution. The idea is that because we had the freedom to observe many different neighbors of the initial state, we should be able to select the best neighbor.

3.2.1 General Outline of the Algorithm

Here is the general outline of the local search with simulated annealing algorithm:

1. Initialize the Sudoku board and the temperature
2. Loop until temperature is at minimum.
    (a) Loop until maximum number of iterations has been reached.
        i. Determine neighboring state via the neighborhood function.
        ii. Determine the fitness of the current and neighboring state.
        iii. If the neighboring state has a better fitness score than the current, then change the current state to the neighboring state.
        iv. Else, if the randomness defined by the temperature permits, change the current state to the neighboring state anyways.
        v. Else stick to the current state.
        vi. Keep track of state with lowest energy
    (b) decrease the temperature

3.2.2 Initialization

The initialization is done by figuring out the frequency of all the values that appear on the initial board and assigning an appropriate amount of each values into all empty cells of the board. For example, in a 9 by 9 board, we want all the values from 0 to 8 to appear exactly 9 times on the board. Ensuring this correctness in frequency of the values is important, because our switch-value neighborhood function requires it, which is further described below.

3.2.3 Neighborhood Function

A neighbor of a given state is determined by randomly choosing two cells that are not initially filled in, and switching the values. Whether a cell was initially filled in or not is evaluated in the initialization step. Because we are changing the values of two cells, we are not introducing a new value to the board in the entirety of the local search process.

Therefore, in the initialization step, ensuring the correct frequency of all the values is essential to finding the correct solution.

3.2.4 Fitness Function

The fitness function implemented here involves determining whether an integer is repeated or is not present in a particular row, column, or the sub-grid. A fitness value is assigned to a possible solution based on the number of repeated or non-present integers. The more repeated or non-present integers there are in a solution's rows, columns, and sub-grid, the higher the fitness value assigned to that solution. Ultimately, the goal state would have a fitness score of 0, because it will not have any repeated or non-present integers.

3.3 Evaluation of Local Search with Simulated Annealing

In order to evaluate this algorithm, we measure the success rate and total number of iterations given the number of empty cells in the initial Sudoku board. We measure these statistics according to the number of empty cells instead of the actual size of the board, because we hypothesize that the search-space complexity is only dependent on the number of empty cells.

For example, a 9 by 9 board has only 81 cells, whereas a 16 by 16 board has 256 cells. However, our neighborhood function only exchanges the values between empty cells, not the entire board. Therefore, if the 9 by 9 board had 50 empty cells and the 16 by 16 board had only 10 empty cells, we should expect that the 16 by 16 board would be solved faster. Therefore, we decided to measure the success rate and the total number of iterations by the total number of empty cells. Here are the results:

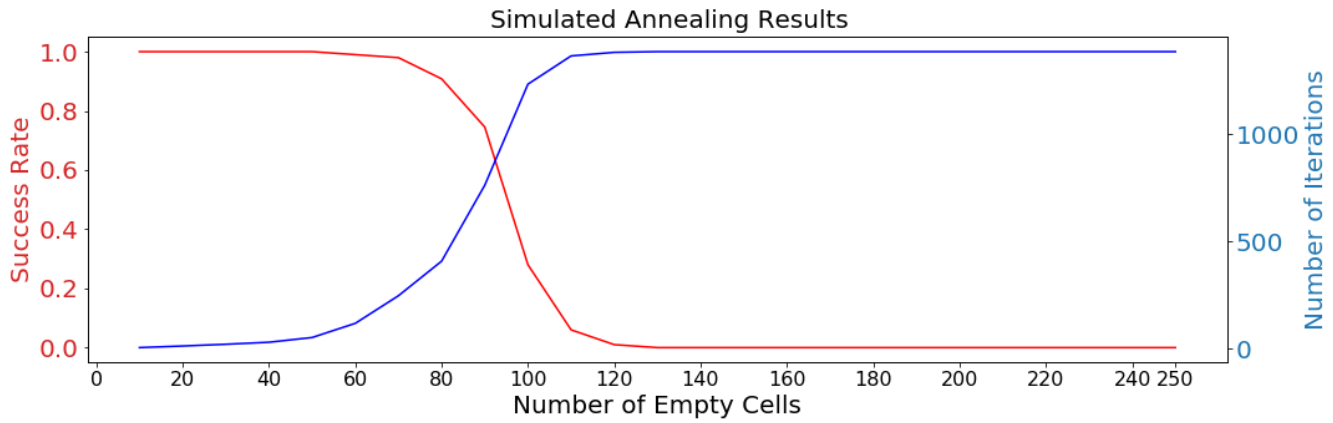

Figure 3: Success Rate and Number of Iterations for Local Search

It is evident that when there are fewer number of empty cells on the board, which is equivalent to "easy" difficulty, the success rate of the algorithm is nearly 100% percent. However, as we increase the number of empty cells, the success rate decreases significantly. At Number of Empty Cells being 120, we nearly have 0% success rate.

An explanation for this can be attributed to the fact that our fitness function could be improved. To understand, we analyze a single run's fitness score:

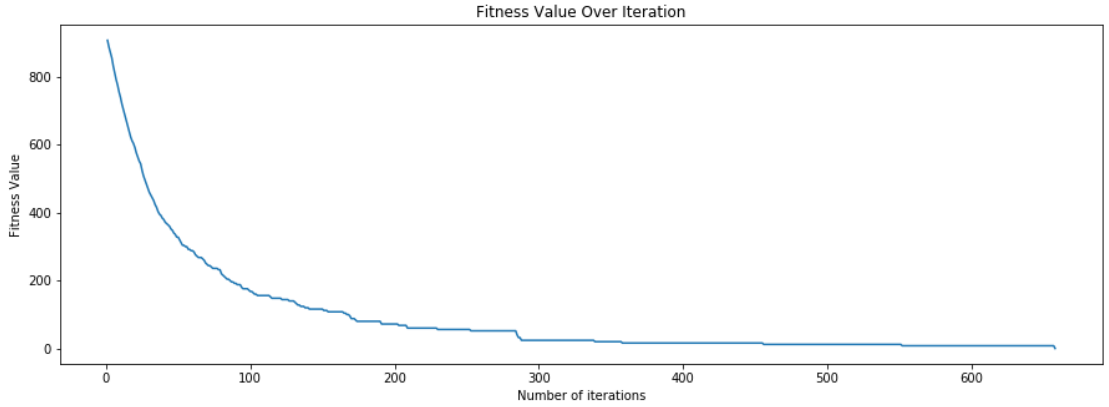

Figure 4: Change in Fitness Value over Iterations

Above is a plot of the fitness function for a successful run of a 16 by 16 board with 90 cells empty. We notice a sharp decrease in the early stages of the run, whereas the change in fitness value from 200^{th} iteration on-wards is imperceptible. This is where the fitness function fails to quickly identify which neighbors are actually close to the goal state. Once in this phase where the fitness score cannot guide the local search, the search becomes almost random, resulting in failure most of the time.

We conclude here that to achieve better results in a scaled up environment with higher difficulty, we need to be able to develop a better fitness function that can distinguish the final few changes that remain at the end of the runs.

4 Genetic Algorithm

The second algorithm that we implemented is a genetic algorithm.

4.1 Strength and Weaknesses of Genetic Algorithm

Genetic algorithm is similar to the above local search with simulated annealing algorithm in that they are both stochastic search methods. Therefore, we can expect this algorithm to have relatively small space requirements and able to prevent being stuck in a local minima. However, the genetic algorithm differs from the previous algorithm in 2 major ways:

1. Population of candidates instead of a single neighbor
2. Mutation function instead of neighboring function

As you can tell from the word choices, this algorithm is inspired by the process of natural selection. The details of the implementation are explained further below, but overall, this algorithm attempts to introduce an increased amount of variability per iteration by having a group of board configuration instead of having a single individual.

4.2 Description of a Population and its Mating Process

The goal of this algorithm is to maintain a fixed size of a population of Sudoku board configurations in every generation, while continuously improving the overall "fitness" score. Because every iteration consists of this population, we will refer to them as a generation and a member of a generation as an individual.

First, given a generation, we proceed to score every individual in the generation via the fitness function mentioned in the previous section. Secondly, we randomly choose pairs of individuals from the generation to mate and produce the next generation. However, we bias this selection process so that it is more likely to select the individuals with better fitness score more often.

In order to avoid getting stuck in local maxima, restart heuristics, with best individuals tracking has been added to the genetic algorithm.

4.2.1 General Outline of the Algorithm

Here is the general outline of the local search with Genetic Algorithm:

1. Phase 1:
    (a) Fill in obvious board values.
2. Phase 2:
    (a) Repeat.
        i. Generate initial population.
        ii. Repeat.
            A. Rank solutions in terms of fitness and only retain the top percentage specified by the selection rate.
            B. Repeat.
            C. Randomly select two individuals from the population .
            D. Randomly choose a crossover point.
            E. Recombine the solutions to produce two new solution strings.
            F. Apply the mutation operator to the solutions.
            G. Until a new population has been generated.
        iii. Until a solution has been found, or the restart threshold has been reached.
    (b) Until a solution has been found, or the maximum number of generations has been reached.

4.2.2 Initialization

The Algorithm's initialization is split between the 2 phases of the algorithm and consists of filling in "obvious" board cells, and in the second phase, generating random board configurations to create the population.

An "obvious" cell is described as an empty cell that has only one possible solution (cell

value that does not violate Sudoku rules), or if a row/column/subset of the sudoku board has only one possible value in that respective cell, then that cell is filled with the value. These are also known as "Norvig's basic Sudoku rules", and have been integrated to constrain the search space to fewer values, where possible.

Afterwards, the board is split into N subset, each subset represents one of the board's square regions. For each of these subsets, a random permutation is generated. This permutation does not change the board's fixed values.

4.2.3 Mate Selection

For each member of the population a "fitness" score is given, to determine how close it is to the solution, using a fitness function. This fitness function computes the individual's score, by counting the number of "genes"(cells) in the individual's configuration that violate the Sudoku rules. A lower fitness score denotes a better individual

Once calculated, the population is sorted in increasing order of fitness, and based on the selection rate parameter, the top $< selection\_rate > \%$ of individuals are selected for mating. The members of selected part of the population, have a probabilistic chance of being chosen, such that even less ideal candidates have a chance to mate.

The Algorithm chooses the elite (top $< elite > \%$ of the population) and passes them on to the next generation. For the rest of next generation's population, the algorithm generates offspring by crossing randomly chosen pairs of distinct individuals from the selection pool.

4.2.4 Crossover Function

Once a pair is selected for mating, the program chooses a crossover point at random. The crossover takes place at the region level, thus for a NxN sudoku board, with N regions, there are 8 possible crossover points. The resulting offspring, as seen in the figure below, is then added to the next generation's population.

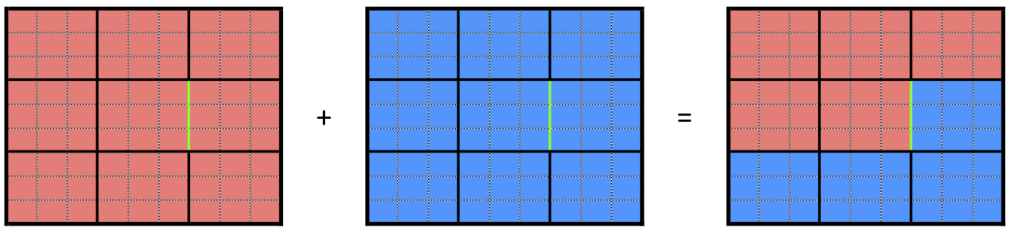

Figure 5: 9x9 Crossover Example

4.2.5 Mutation Function

The mutation algorithm performs as follows. Based on the mutation factor, there is a probabilistic chance that a mutation might occur in an offspring. If a mutation is to occur, a region is randomly selected for the mutation, and then two cells from that region are randomly selected to switch places, if either of the two cells is fixed, then another pair of cells is selected from this region until a mutation occurs.

4.2.6 Restart Heuristic

The program implements a restart heuristic in an attempt to escape local maxima. Upon reaching a certain number of generations without improving on the best fitness of the population, the algorithm will store the $best10\%$ of the individuals in a separate population. Once every 10 restarts the algorithm will this separate population and use it as the initial population for the algorithm.

4.3 Evaluation of Genetic Algorithm

To understand the performance of the algorithm, we analyze a single run of the algorithm on a board with 20 empty cells:

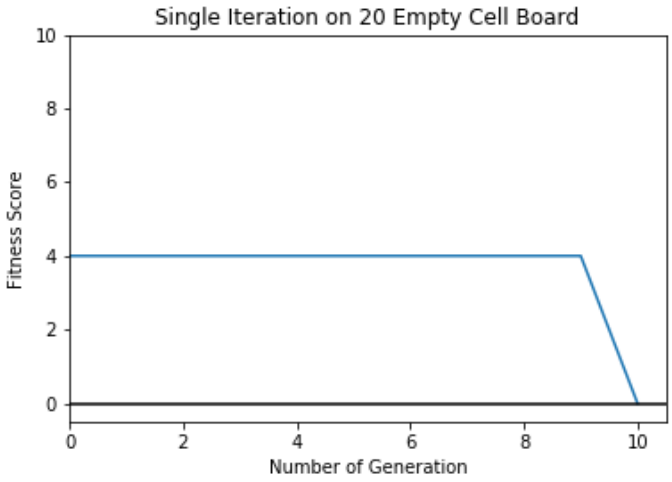

Figure 6: Genetic Algorithm's Minimum Fitness Score per Generation

We can see that Norvig's initialization algorithm allows the minimum fitness score of the very first generation to be pretty low at 4. However, we can see that the minimum does not change for the next 8 generations until it finally drops to 0 at 10th generation. This is because the mating function does not guarantee a better fit child, thus the chances of finding a lower fitness score every iteration is very low. This is understandable, because a random combination of two fit parents does not necessarily guarantee a fit child.

Lastly, we observed that the scalability of the genetic algorithm is poor. The reason why we ran the above test only on a 20 empty cell board is because a reproducible results only occur at around 20. At 50, the algorithm may solve it in 1 generation or in a 100 generations. And at 100+ boards, we observe that we never converge to a solution.

We note here that 1 possible improvement to this algorithm is to reduce the randomness by guaranteeing the child of a mating process to have a better fitness score. This will may result in running into a local minima, but the restart process should be able to prevent the algorithm from failing.

5 Backtracking Algorithm

Last algorithm that we implemented is a non-stochastic backtracking algorithm with Arc consistency, Minimum Remaining Value ordering (MRV), and Least Constraining Value (LCV) selection.

5.1 Strength and Weaknesses of Backtracking Algorithm

Backtracking is a way of applying depth-first search to a constraint satisfaction problem. The algorithm selects an empty cell, and tries the possible number assignments that does not violate the above 4 constraints. After making a valid assignment, it proceeds to make more assignments. If at any point the algorithm finds out that for a chosen cell, there are no possible values that preserves the 4 constraints, it will back track the most recent assignment, and it will try a different value in the removed cell. If the algorithm backtracks to the initial state of the board with no possible number assignments left, then the algorithm will return a "fail".

Because this algorithm does not have any randomness in its search, it has an amazing property of guaranteeing the absence of solution when it fails. However, the same non-stochastic aspect of the algorithm is its biggest downfall. The run time of the algorithm is slower than the first two algorithms, because in worst case scenario, it has to search through $O(N^{N^2})$ possibilities.

5.2 Description of Backtracking Algorithm

5.2.1 General Outline of the Algorithm

BacktrackAlgorithm()

1. Order the variable according to MRV
2. Order the possible number assignments according to LCV
3. Loop over the ordered number assignments:
   (a) check if the assignment meets the 4 constraints
   (b) If it does not meet the constraints, terminate loop
   (c) If it does meet the constraints, Assign the number to the cell
   (d) check if the board is Arc consistent
   (e) if the board is not Arc consistent, terminate loop
   (f) if the board is Arc consistent, call BacktrackAlgorithm()

(g) check result of BacktrackAlgorithm()

(h) if successful/ true, return true

(i) otherwise continue to next iteration of the loop

4. return failure/ false

5.2.2 Minimum Remaining Value Order

When we are choosing which cell to assign a number to, we choose the cell that has the least number of possible values. For example, the from the Figure 1 initial board, we get the Figure 3 as our domain size. Then, our MRV algorithm will choose one from the following set of cells to assign values:

$$(4,4),(6,5),(6,8),\text{ or }(7,7)$$

This because these 4 cells have the least number of possible values. Essentially, MRV algorithm reduces the size of the total space that needs to be searched.

		3	2		4	4	4	3
	3	3				4	3	4
2			2	2	2	5		3
	3	4	3		3	4	4	
	2	4		1		3	3	
	2	4	2		2	4	3	
3		6	3	2	1			1
2	3	3				2	1	
3	4	5	4		2	4		

Figure 7: Possible Domain Sizes

Furthermore, when we have ties from the Minimum Remaining Value ordering, we use the Maximum Degree Heuristic to break the ties. This is done by checking how many empty cells it has in its corresponding row, column, and the box. For example, we will choose $(6,5)$, because it has 14 different empty cells, whereas cells $(4,4),(6,8)$, and $(7,7)$ only have 10, 12, and 11 empty cells respectively.

5.2.3 Least Constraining Value Selection

After having chosen a cell, we must decide which value to assign. This is determined by the Least Constraining Value selection, which means that we select values that most likely to be correct. This heuristic allows us to arrive at the solution quicker than otherwise.

5.2.4 Arc Consistency

We implemented AC-3 algorithm outlined in the lecture slides, and it allows us to reduce the search space by eliminating some of the possible choices. In some simple 9 by 9 cases, the solution can be found without a single backtrack.

5.3 Evaluation of Backtracking Algorithm

In order to evaluate this algorithm, we measure the number of backtracks given the empty cells in a board, and here are the results:
First thing to notice is that the number of backtracks is nearly 0 up until 130 empty cells. This is because most of these cases the Arc consistency algorithm solves most of the puzzle, and there are no need for any backtracking. However, as soon as the number

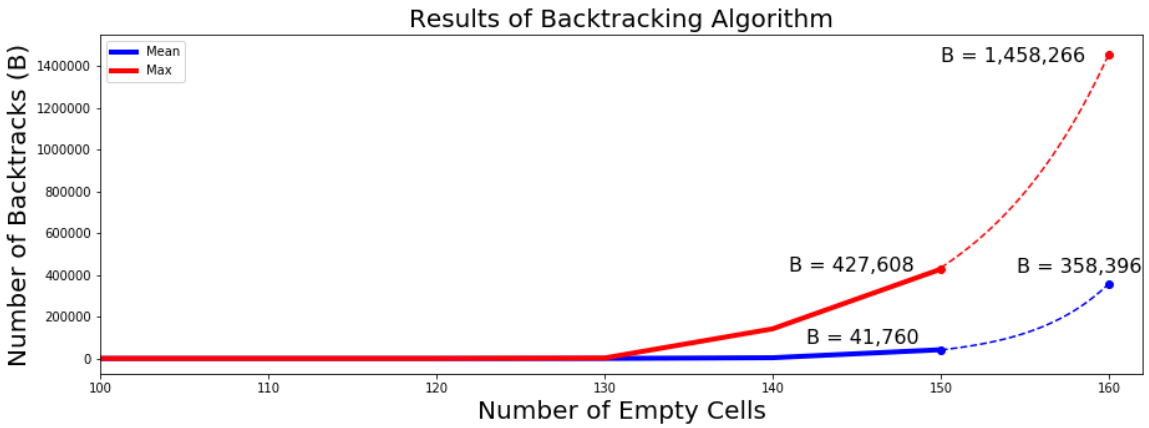

Figure 8: Number of Backtracks for Every Difficulty of the Sudoku Board

of empty cells grows beyond 140, the number of backtracking grows exponentially. At 150 empty cells, the backtracking algorithm nearly takes on average about 41, 000 backtracks. This is equivalent to 2 minutes depended on the clock cycle speed of the environment. At this rate, at 160 cells removed, the algorithm is expected to take over 300,000 backtracks which takes about 20 minutes. When considering the worst case scenario, the algorithm could take over a million backtracks, which would take about an hour and 15 minutes.

Although backtracking algorithm performs better than local search with simulated annealing, backtracking algorithm fails to scale up to the difficulty of the Sudoku puzzle we want to solve. Our goal is a 25 by 25 board which can have up to 625 cells empty. We expect that developing a better heuristic might be able to improve the algorithm slightly, but the ability to scale up to 600+ empty cells seems implausible for backtracking algorithm.

6 Conclusion

The Sudoku problem was a good way to take a better look at the trade-offs of different Constraint Satisfaction Problem, and to tackle it we implemented 3 algorithms:

1. local search with Simulated Annealing
2. genetic algorithm
3. backtracking algorithm

Among these algorithms, here are the key differences among these algorithms:

1. Stochastic vs Deterministic
2. Local vs Non-Local

The strength of the deterministic backtracking algorithm was the guarantee of a valid output. The algorithm either terminates with a solved board or identifies it to be unsolvable. However, stochastic algorithms may terminate inconclusive when it reaches a local

optima.

The weakness of the deterministic algorithm is that it has to search through an exponential amount of board configurations. Even though the heuristic functions like MRV sorted the search space, the algorithm still had difficulties when we scale the size of the problem up to 25x25 boards or with many empty cells.

On the other hand, the strength of a stochastic algorithms was the memory usage. Because we only needed to have a current state and a neighboring state at any given moment in the algorithm, the memory space was constant at $O(c)$. However, the memory complexity grows linearly or exponentially for deterministic search algorithm, because we need to remember the states that we need to go back to.

Also in this paper, the difference between local and non-local search scheme was explored. Because local search solves Sudoku a small increment at a time, we can observe a slow, but guaranteed improvement every iteration. However, non-local search scheme behaves very unpredictably because we cannot guarantee the improvement of the fitness score for every iteration (generation). These properties mean that the genetic algorithm is much more randomized compared to simulated annealing.

We note that a possible improved solution to Sudoku problem may be a hybrid of the backtracking and local search algorithms. We would begin the algorithm by running local search to get a nearly solved board, then the rest of board will be solved using backtracking. This utilizes the strengths of both algorithms. Local search's fast convergence in the early phase and backtracking's guaranteed solution in the later phase should improve the overall search time.

7 References

1. Russell, S. J., & Norvig, P. (2021). Artificial intelligence: a modern approach. Hoboken: Pearson.

8 Appendix

8.1 Running the code

The project is written in C++, and the project must be ran using a GNU compiler. We provide a makefile to compile the code. Run

$$makemain$$

and to run the project run

$$./main$$

Note that the current file runs tests. If you want to run specific board configurations, please reach out to our team.