Comparison of Non-Deterministic Sudoku Algorithms

Kenneth Browder
BrowderKD20@gcc.edu

Caleb Frey
FreyCP20@gcc.edu

Keegan Woodburn
WoodburnKB20@gcc.edu

I. Introduction

In 1979, Howard Garns, a puzzle maker, created one of the most well-known games on earth: Sudoku. Though it was originally titled "Number Place," it was rebranded by a Japanese puzzle company, Nikoli. After publishing the game under the name "Sudoku", it quickly gained global attention, and has even been referred to as a mental virus [11]. Because of its renown and simple rules, Sudoku has become a staple computational problem, often showing up in demonstrations of various search algorithms and heuristics.

II. Terminology

Sudoku features a nine-by-nine grid, called a board, which is split into nine sub-grids. The game requires that every distinct row, column, and sub-grid contains every digit from one to nine. The rows are indexed by the letters A through I, and the columns are indexed by the numbers one through nine as illustrated in [Fig. 1].

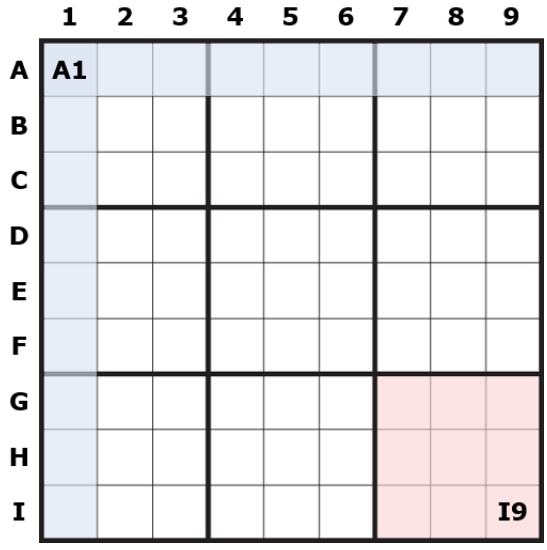

Fig. 1. A standard Sudoku board. The row and column units of A1 are highlighted in blue, and the sub-grid unit of I9 is highlighted in red

Each row, column, and sub-grid is called a unit. Additionally, each cell has a set of corresponding cells constraining its value, called peers. For example, cell A1 would have the peers A2 through A9, B1 through I1, and any cell that resides in A1's sub-grid. In other words, A1's peers are the units that A1 resides in—excluding duplicate cells and A1 itself. A player is given an initial board with only a few cells filled in. These pre-filled cells are called clues. Though commonly visualized as a grid with some filled cells and all others empty, for computational purposes we define what is called a group board. This is simply the same Sudoku board as normally visualized, but each empty cell is replaced with its possible values in the context of the constraints set by its peers. These possible values are called groups. For example, if A1 contained the value one and was the only filled cell on the board, A2's group would be the numbers two through nine, since A1 already has already been assigned the value one. Though many algorithms are used to solve Sudoku boards, we will focus on two: a standard backtracking algorithm and a novel genetic algorithm.

III. Backtracking and Heuristics

A. Constraint Satisfaction

The first way of solving a Sudoku puzzle is backtracking. For as long as Sudoku has been considered as a computational problem, backtracking has been the most common method for solving any given board. It is a brute-force algorithm that assigns a number for the given empty cell and backtracks if that number is found to be invalid in the context of its peers [7].

Generally, when trying to solve Sudoku using backtracking, the puzzle is defined as a constraint satisfaction problem. Constraint satisfaction problems have three main components: variables, domains, and constraints. Each variable maps to a domain and must satisfy one or more constraints [7] [8]. By turning Sudoku into a constraint satisfaction problem, we get a

set of variables—the cell indexes; a set of domains—-the group of each given cell; and a set of constraints—a list of inequalities, each stating that the left variable x cannot equal the right variable y. There is a constraint for every possible cell index and its peers. For example, A1 would have the constraints $(A1 \neq A2)$, $(A1 \neq A3)$, $\ldots$, $(A1 \neq A9)$, $(A1 \neq B1)$, $(A1 \neq C1)$, $\ldots$, $(A1 \neq I1)$, etc. This essentially restates the rule that every row, column, and sub-grid must contain every number from one to nine in a way that is more easily computed. Converting Sudoku into a constraint satisfaction problem allows for the backtracking algorithm to neatly assign a variable and check for correctness [2].

B. Backtracking Algorithm

The algorithm we use is a common depth-first search backtracking algorithm written by Dhanya Job and Varghese Paul [4].

Algorithm 1 Backtracking

Input: A group board S
Output: A board S' which is the solution to S or **false** if S cannot be solved

```
1:  if all group sizes = 1 then
2:      return S
3:  end if
4:  for all groups G in S do
5:      for all values v in G do
6:          assign v to index of G in S
7:          if Solve(S') then
8:              return S'
9:          end if
10:         undo assignment to index of G
11:     end for
12: end for
13: return false
```

As seen in Algorithm 1, this algorithm employs the use of recursion to emulate the backtracking necessary to solve a Sudoku board. This technique will take the next empty cell, assign a value within its domain (as seen on line seven), recursively call the board with this new assignment, and repeat the whole process until either an invalid choice is made, in which case the algorithm will backtrack, or there are no possible choices left. If there are no possible choices left, then the solution has been found.

C. AC-3 and Heuristics

Though backtracking is sufficient to solve a Sudoku board, it has an enormous run-time of $O(9^c)$ where c is the number of empty cells. Because of this, backtracking for Sudoku also often employs other algorithms and heuristics. For this paper, we implemented a minimum domain value heuristic described by Crawford et al. [2]. This greedy heuristic simply states that when choosing what cell to assign a value, pick the cell with the smallest domain. By doing this, the domains of other peers are diminished faster than randomly selecting a cell to assign a value to. We also implemented Alan Mackworth's AC-3, which is an arc consistency algorithm for maintaining consistency between variables in a constraint satisfaction problem [1]. AC-3 is split into two parts: the main algorithm and the Revise algorithm.

Algorithm 2 AC-3

Input: A CSP, $P = (X,\ D,\ R)$
Output: **true** and P' (which is arc-consistent) or **false** and P' (which is arc-inconsistent because some domain remains empty

```
1:  for every arc R_ij ∈ R do
2:      Append (Q, (R_ij) and Append (Q, (R_ji))
3:  end for
4:  while Q ≠ ∅ do
5:      select and delete R_ij from queue Q
6:      if Revise(R_ij) = true then
7:          if D_i ≠ ∅ then
8:              Append(Q, (R_ki)) with k ≠ i, k ≠ j
9:          else
10:             return false /* empty domain */
11:         end if
12:     end if
13: end while
```

In the main algorithm a set of arcs is defined by the given constraints and their inverse values. For example, if a constraint states that $x = y$, then the arc set would be $\{(x = y),\ (y = x)\}$. This can be seen in lines one and two. This is done to maintain two-way consistency, since the arc $(x = y)$ is handled differently than the arc $(y = x)$ throughout the algorithm. These arcs are then put into a queue, usually called an agenda. This queue is then iterated over. For each iteration, the next arc in the queue is removed and considered by the Revise algorithm.

The Revise algorithm attempts to prune values from the domain of the first variable x with respect to the domain of the second variable y. For example, suppose the variables x and y have the domains D_x and D_y respectively, and that both these domains are the set $\{1,\ 2,\ 3\}$. If the Revise Algorithm is given the arc $(x > y)$, it will cycle through each value

Algorithm 3 Revise

Input: A CSP P' defined by two variables $X = (X_i,\ X_j)$, domains D_i and D_j, and constraint R_{ij}.
Output: D_i, such that X_i is arc-consistent relative X_j and the Boolean variable *change*

1: *change* ← **false**
2: **for** each $a \in D_i$ **do**
3: **if** $\nexists\, b \in D_j$ such that $(X_i = a,\ X_j = b) \in R_{ij}$ **then**
4: remove a from D_i
5: *change* ← **true**
6: **end if**
7: **end for**
8: **return** *change*

of D_x and verify that the elements in D_y satisfy the constraint as defined by the arc. This is seen in lines 2 and 3. Should the constraint be violated, the value in D_x that the algorithm is currently comparing to D_y will be removed as seen in line 4. In this case, when considering the first element of D_x (which is 1), there is no element in D_y that satisfies the constraint $(x > y)$, since 1, 2, and 3 are all greater than or equal to 1. Thus, the algorithm will prune 1 from D_x. This process is repeated for the remaining elements in D_x—though none will be pruned as they all satisfy the constraint.

Additionally, when a value is pruned from D_x, the Boolean variable that Revise ultimately returns is set to true. Once this procedure is done over a given arc, if Revise returns true, the main algorithm confirms that D_x is not empty as seen on lines six and seven. If D_x is not empty, then all possible arcs with a second variable equal to x are added to the agenda. Continuing with the previous example, this would mean that the arc $(y < x)$ would be added to the agenda. This is done because as D_x changes, the variables that may be constrained by D_x need to be reconsidered. If D_x previously satisfied the constraint $(y < x)$, a change to D_x could affect the satisfaction of that constraint.

In terms of a Sudoku board, this essentially checks the peers of each cell and their constraints, and prunes values accordingly. As described in a previous example, if A1 is given the value 1, then A2's domain would be changed due to the constraints set. Once this process is done over all the arcs, if no arc has produced an empty domain, then the problem is considered arc consistent.

When applied to an unsolved Sudoku board, AC-3 can often significantly reduce the search space, and in some cases even find a solution as we will discuss in our results. Despite this, its usefulness depends on the position of the given clues and their relation to each other, meaning one board with seventeen clues could take thousands of times longer than another board with the same number of clues.

D. Results

As we discuss our results, the complexity of a board and how that is defined is important. We define five levels of difficulty for Sudoku boards: very easy, easy, medium, hard, and very hard. Each level of difficulty encompasses a number of clues that an initial board contains. Very easy boards contain more than 50 clues. Easy boards contain 30 to 50 clues. Medium boards contain 25 to 30 clues. Hard boards contain 20 to 25 clues. Very hard boards contain 17 to 20 clues. With these distinct types of boards, we can more easily identify the capabilities of each algorithm.

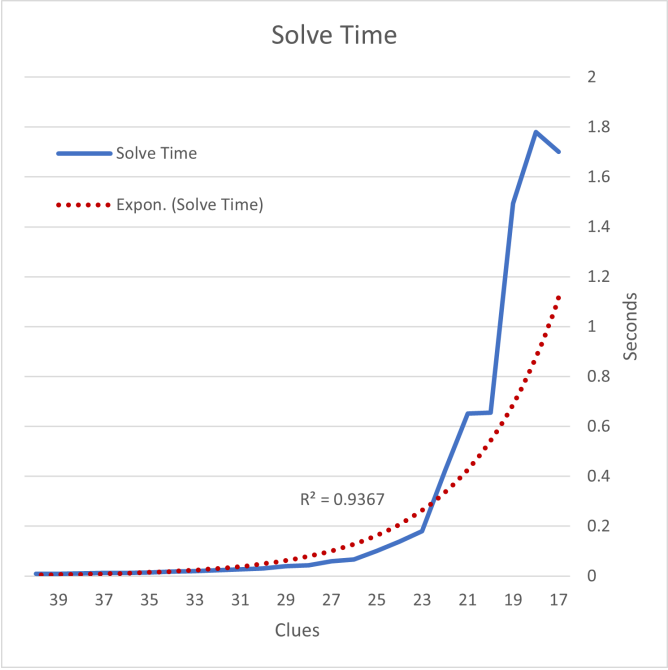

Fig. 2. Average solve times of boards with clues ranging from 17 to 40. Solve times are indicated by the blue line and the exponential trend-line is indicated by the dotted red line.

We collected four million initial Sudoku boards ranging from seventeen to eighty clues, with 62,500 boards mapped to each number of clues. Because of the quantity of boards and how they were generated, some may have multiple solutions. However, because we will be running both algorithms on the same boards, this should not affect the analysis of either algorithm.

The backtracking algorithm in tandem with AC-3 performs as expected: as the number of clues goes down, the run-time required to solve a given board goes up. This inverse relationship can be seen in [Fig. 2], where we show the average solve times for 17 to 40 clues. Each point is the average of 1,000 boards. After 26 clues, the time to solve the board consistently takes less than 100 milliseconds and after 40 clues, the algorithm takes less than one millisecond to solve a given board. As indicated by the R-squared value of 0.937 in [Fig. 2], the correlation between clues and run-time is strong. This almost exponential curve is to be expected. As we are given less clues, the algorithm must rely more on backtracking than on the heuristics we chose—which only work well with enough clues. In the worst possible scenario, a board could have a run-time very close to $O(9^c)$ where c is the number of empty cells.

While AC-3 may speed up the solve time, it is nondeterministic as it is dependent on the number of cells that we can use for propagation. Because of this uncertainty, we collected and generated boards that would render propagation much less effective, forcing the longest possible solve times. Of these, there were a handful of notable boards, each of which took longer than 1000 seconds. The longest board took over a day to solve. This solve time is likely due to a combination of the value ordering for our variable domain and the restricted propagation techniques used. Each variable has an initial domain range of the numbers one through nine unless a clue has already been assigned to that variable. This could cause problems when a cell's true value is 9, but the algorithm must cycle through all values up to 9 first. Additionally, AC-3 is a general arc consistency algorithm, meaning propagation techniques specific to Sudoku were ignored. Methods such as naked and hidden pair reduction, x-wing location, and sword fishing are all heuristics that would likely improve the solve times of the difficult boards we found [6].

Despite the enormous run-time among a small subset of boards, backtracking still seems to be a good algorithm for solving Sudoku boards due to the sheer quantity that it can solve in less than a second as seen in [Fig. 3], where the majority of solved boards lie under the half-second mark.

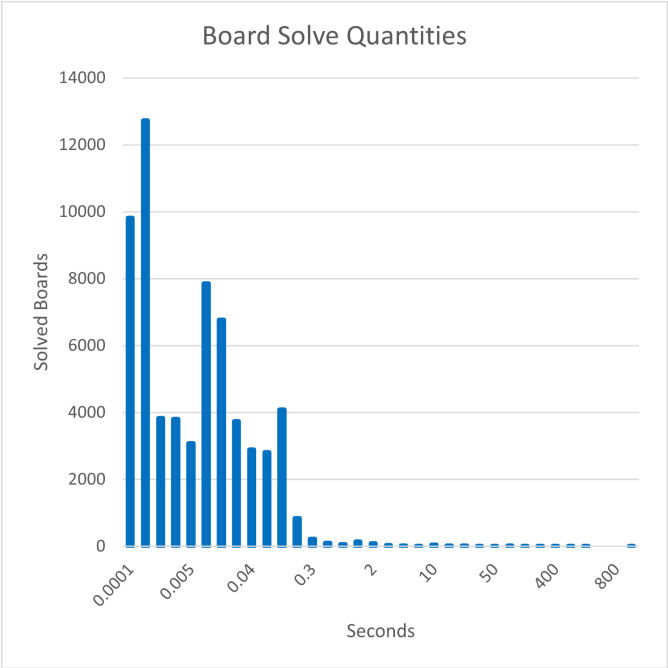

Fig. 3. Number of boards solved per time interval.

IV. Multistage Genetic Algorithm

A. Solving Sudoku With Genetic Algorithms

The second approach to solving Sudoku is to use a genetic algorithm. Genetic algorithms take inspiration from biological concepts such as population fitness, crossover, and mutation. In the traditional sense, genetic algorithms model the evolution of a population to favor better candidates with a probability, and in the context of a computational model it is similar. The algorithm represents possible solutions to a problem as a population, and given a fitness function, continually evolves that population to converge to the optimal solution [1]. This understanding of genetic algorithms is partially sufficient for the context of Sudoku solving. The complication of applying this understanding of a genetic algorithm is due to the constraints of the clues given. If a traditional genetic algorithm was applied to a Sudoku board, during the crossover and mutation operations, the constraints of the preset board would not be maintained, and the population values would diverge from the original required clues. Because of this understanding of the difficulties of applying a traditional genetic algorithm, other adaptations of this algorithm have surfaced, addressing these issues, and providing a novel solution.

After thorough research we decided to use research from the paper "A novel multistage genetic algorithm approach for solving Sudoku puzzle" by Chel, Haradhan, et al. This paper outlines an approach to solving Sudoku using an adaptation of the genetic algorithm framework, designed to address the previously stated issues. The research that was done indicated to us that

Algorithm 4 Multistage Genetic Algorithm

Input: Parent population size μ, Offspring generation factor k, number of cycles C, Number of Iterations over each cycle N, crossover probability p_c, mutation probability p_m, group table updating parameters. Group table GT

Output: Desired solution

```
GT ← Create group table
while fitness ≥ 0 do
    while cycle ≤ C do
        if cycle ≥ 1 then
            update group table
        end if
        pop_size ← μ/cycle
        offspring_size ← pop_size · k
        generate population of size pop_size
        for each iteration i ≤ N do
            perform crossover
            evaluate population
            choose best μ solutions
        end for
        perform mutation over a few good solutions
        evaluate mutated solutions
        find best solution to updating group table
        cycle ← cycle + 1
    end while
end while
```

this was the best candidate paper for laying out an algorithm, though after diving deeper into the material within the paper, we realized several problems with the paper, from grammatical errors to simply nonsensical content. These issues will be later discussed in detail. As a result of these issues, certain liberties had to be taken throughout the implementation and replication of this algorithm, all of which will also be described, and all of which we feel as though are reasonable modifications that follow the intent of the authors.

B. Population Generation

The population is generated from the group table of the most recent cycle. For each cell that is not fixed, a random value from the group of possible values at that cell is chosen. This creates a population of random children, with varying fitness.

C. Selection and Fitness Function

A fundamental aspect to any genetic algorithm is the fitness function [10]. The fitness function, as prior mentioned, is used to evaluate how "good" a particular member in a generation is, for whatever metric good is defined to be. In this paper, fitness is defined as the number of repetitions of each number $j = 1 \cdots 9$ over each of the rows, columns, and sub-grids. In the traditional understanding of genetic algorithms, fitness is maximized over the iterations of the algorithm, but in this paper, fitness is minimized to reward less repetitions. The optimal fitness of a Sudoku board is 0, with no repetitions in any of the three categories.

The fitness function of this paper is shown below and is described in the paper as follows: "In the formulation, first, second and third summation captures repetition over rows, columns, and sub-grids respectively. The fourth summation captures the amount how far that number deviates from 9. N(j) stands for number of a particular integer $j : 1 \leq j \leq 9$ over the whole 9 x 9 Sudoku" [3].

$$F = \sum_{j=1}^{9}\sum_{i=1}^{9} n_r^i(j) + \sum_{j=1}^{9}\sum_{i=1}^{9} n_c^i(j) + \sum_{j=1}^{9}\sum_{i=1}^{9} n_b^i(j) + \sum_{j=1}^{9} abs(N\,(j) - 9) - 27$$

We, however, do not believe that this fitness function accurately behaves as the authors described. We believed that a reasonable assumption to make was that the first three terms of the proposed fitness function should sum to 9 (when there are no repetitions), so that the last term would cancel those three out, allowing the fitness to be 0 as proposed in the algorithm's pseudo-code. The last term shows how far each number deviates from the expected total count of each. After attempting to implement this fitness function, it became clear that the first three terms did not perform as described, and did not meaningfully capture repetitions in rows, columns, or sub-grids. Either this value should be 0, if we assume that the n_r^i counts repetition per row, or should be 81, if counting the number of occurrences per row. This applies to the second and third terms as well. Neither of these interpretations follow the description of their fitness function to "capture repetition" [3], so we propose an alternate fitness function inspired by the original. The goal of the fitness function is to capture repetition (and additionally, the lack of any value), and to do this, we define the fitness function:

$$F = \sum_{j=1}^{9}\sum_{i=1}^{9}\left(1 - n_r^i(j)\right)^2 + \sum_{j=1}^{9}\sum_{i=1}^{9}\left(1 - n_c^i(j)\right)^2$$

$$+ \sum_{j=1}^{9}\sum_{i=1}^{9}\left(1 - n_b^i(j)\right)^2 + \sum_{j=1}^{9} abs(N\,(j) - 9)$$

where the only difference of each term is finding the distance of $n_r^i(j)$, the count of number j in row i, from 1, instead of just the count $n_r^i(j)$. Again, this change applies to all three of the first terms, but the fourth term is left as original. Lastly, the constant 27 was removed from the end, as the updated terms should sum to 0, and therefore do not need a counterbalance. This change captures what the authors described, and penalizes solutions with more repetition in each row, column, and block. This updated fitness function was then used in the rest of the implementation of this algorithm, and provided comprehensible results, alongside being an intuitive implementation of this concept.

D. Crossover Operator

One of the many shortcomings of the paper from which we sourced the genetic algorithm used to solve Sudoku was its crossover operator. In any genetic algorithm, the crossover operator is a method of combining two "parent" genes in order to generate new genes that are a combination of both of the parent genes. The paper we used cited a geometric crossover algorithm specifically designed for Sudoku, but the actual algorithm they described in the paper was much more similar to a uniform crossover algorithm citation. In a uniform crossover operation, two parent genes are selected from the gene pool, then the child genes are chosen such that they are each duplicated from the parent gene, except that for each character in the gene string, a random choice is made with probability p to swap between the children in the corresponding spot. For the crossover operator, we followed the general outline described in the paper, without differing much from the original implementation. On line 4 of Figure 1, the loop performs crossover on the population that was generated from the population generation step on line 3. Then, the population is evaluated and sorted using the redefined fitness function, and the best μ solutions are chosen, where μ is the size of the parent population. This iteration happens N = some arbitrary constant times, increasing the average of fitness of the population with each iteration.

E. Mutation Operator

We decided that the mutation operator defined by the authors was reasonable and intuitive, and implemented their mutation operator. This operator is similar to crossover, but instead of swapping from parent cells, picks a random other value from the initial board's group table. This introduces more randomness into the population, and partially helps to avoid local minimums.

F. Updating the Group Table

The last step of each iteration is to update the current group table. The proposed method of doing this was unclear and introduced concepts that were not relevant to Sudoku at all. One of the proposed methods of group table updating involved checking diagonals for repetition, which indicated to us that this section of the paper could likely be further improved and optimized in a way that provided our algorithm with better results. The broad idea that was defined was to use the best solution, remove any repetitions, and then to probabilistically fix cells of the group table. This was the general idea that we stuck to, yet we did not think that the described multi-directional validation was the approach that made sense to implement this. Instead, we used the best solution from the mutation step, and for each cell, checked if it was: 1) in the group table of the original board, and 2) was in an arbitrary percentage of the other best μ solutions (we found that 80% performed well for larger population sizes, and 90% performed well when the population size was smaller). If both were true, the cell in the group table was probabilistically fixed to the cell of the best solution with some probability U. Step 1 of our revised implementation prevents repetitions, given that any group in the initial group table already eliminates repetition.

G. Simulated Annealing

After implementing and testing the initial genetic algorithm previously discussed, we found that the genetic algorithm would often get stuck in local minima—configurations of the group table which had a lower fitness than most similar boards, but which would not result in a valid configuration of the Sudoku board. When initially researching genetic algorithms to implement, we came across a paper which proposed a Hybrid Genetic Algorithm and Simulated Annealing (HGASA) approach in order to avoid getting stuck in local minima [9].

Simulated Annealing is another stochastic search algorithm which attempts to imitate the cooling process of metals in order to efficiently walk through a large search space. The general approach of Simulated Annealing relies on a cost function and a neighbor generating function. An arbitrary choice in the solution space is generated, which we will call Candidate.

Algorithm 5 Simulated Annealing

Input: Initial Temperature t, decay d, neighbors n, cost c **Output:** Desired solution

choose candidate C
$temp \leftarrow t$
while $cost(C) > 0$ **do**
    $N \leftarrow Neighbor(C)$
    if cost of $N < cost(C)$ **then**
        $C \leftarrow N$
    else if probability $p = e^{(cost(N)-cost(C))/t}$ **then**
        $C \leftarrow N$
    end if
    $temp \leftarrow temp \cdot d$
end while

The key for Simulated Annealing is the proper choice for each of the parameters, which varies based on the problem. The paper from which we sourced the HGASA approach did not specify the optimal choices for the neighbor function, cost function, initial temperature, or decay schedule, so we chose what seemed like reasonable choices for each. The cost function we use for the hybrid approach is the same one used for the standalone genetic algorithm that we initially implemented. The neighbor function we use chooses an arbitrary number of spots to change on an existing solution, with a probability of actually making a change at each one.

Algorithm 6 Neighbor

Input: Solution S, Neighborhood N, Mutation Probability P_m **Output:** Neighbor

$indices \leftarrow$ unique indices on on a Sudoku board
$neighbor \leftarrow S$
for $index \in indices$ **do**
    if $probability = P_m$ **then**
        $neighbor[index] \leftarrow$ random GT value
    end if
end for
return $neighbor$

In the spirit of the paper, we decided to only use Simulated Annealing as an alternative to mutation when the genetic algorithm got stuck for a certain number of cycles, which means that Simulated Annealing only needed to function as a more intelligent random solution generator. Because of this, the value we use for the neighborhood is 81, which means that any value on the board is allowed to change, although the actual probability of any value on the board changing is 0.2. Also, the initial temperature for the algorithm is set to 1.5, so that the probability of accepting a worse solution is be very low, therefore Simulated Annealing should not greatly hinder the progress of the genetic algorithm. However, this would also make it so that the Simulated Annealing portion of the hybrid algorithm would not be as effective at getting out of local minima, but since Simulated Annealing is being performed on the whole population this set of parameters allows for solutions to more easily be found. The decay factor is set to 0.95, which was the example decay factor in the paper.

H. Results

As both the Simulated Annealing and Genetic Algorithms are non-deterministic algorithms, the most feasible way to analyze the run-time of HGASA at our level of study is to observe the actual run-time of the approach over actual problems. The difficulty of boards is defined as previously for backtracking, but the graphs use the number of empty spaces on the

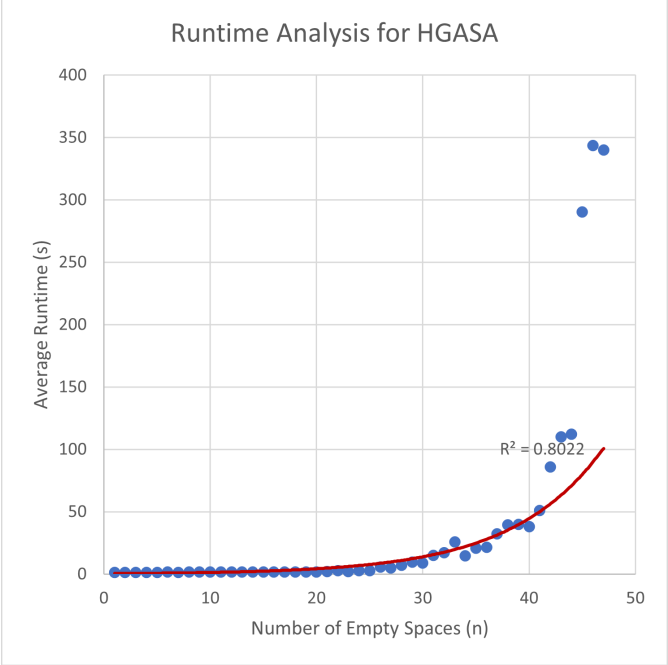

Fig. 4. A scatter-plot of the average hybrid algorithm solve times. The red line indicates the exponential trend-line.

board rather than the number of clues. Just as with backtracking, with an increase in the number of empty spaces provided for HGASA to solve on, the run-time increases in what appears to be an exponential fashion. An initial exponential curve fit, although it has a r^2 value (0.8022), does not visually match the data very well because all the boards up until 20 empty cells have about the same run-time [Fig. 4].

Eliminating these data points results in a much better curve fit, with an r^2 value of 0.9091 [Fig. 5].

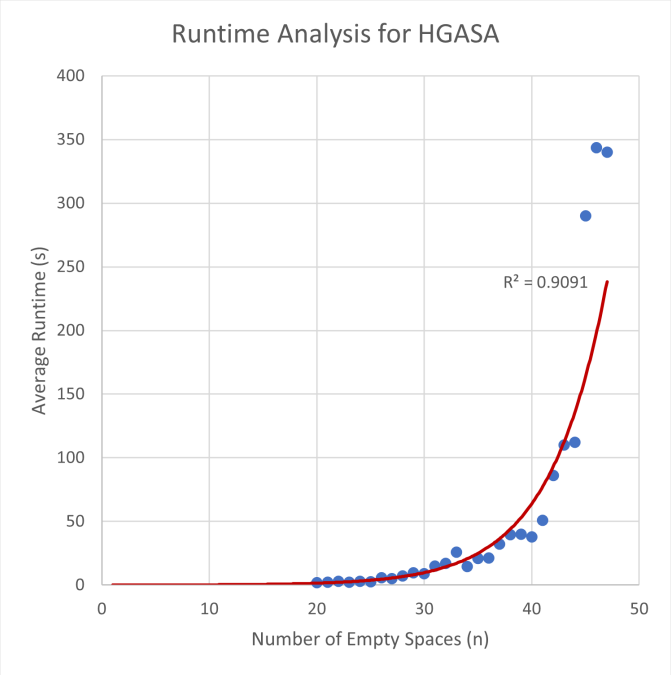

Fig. 5. The same scatter plot as Fig. 4, but without the first 20 data-points, highlighting the exponential curve.

There are a number of ways both the experimental analysis of run-time and our implementation of HGASA could have been improved that could make the run-time much faster. Due to the time consuming, non-deterministic nature of HGASA, it was necessary to run many trials of each test simultaneously. This meant that although the timing code was, in general, true to how long it would take the algorithm to solve each board, the overhead on running on Python's thread system made it so that we could not tell the difference in run-time between the easier boards. Thus, the exponential fit had to be done on the more difficult boards. Also, we could have used a more intelligent mutation operator both in the neighbor function and in the mutation stage of the genetic algorithm that would greatly reduce the search space for our heuristic search algorithm—which would not cause the run-time to be any better than exponential but would likely decrease the constant factor on the run-time.

V. Conclusion

Both backtracking and the HGASA seem to have exponential run-times according to our experimental analysis. However, there is a clear difference in both consistency and scale of the run-time, for which the backtracking approach comes on top in both cases. Very often, backtracking is able to solve boards in sub-millisecond times, while our implementation of HGASA seems to take around a second even on the easiest boards, although we suspect that this may have to do with the overhead of testing rather than the approach itself. Nevertheless, HGASA is often unable to solve boards more difficult than 30 clues in any reasonable amount of time, and even when it is able to solve these boards it could never compete with the solve times of backtracking, which are on the scale of a second. In general, the clear choice is to use backtracking to efficiently solve sudoku. Even if our use of HGASA was competitive with the backtracking we used in terms of scale and consistency, the extra development time needed to tune parameters and test different configurations seems to make stochastic search methods a poor choice for this problem. That said, there is a strength to HGASA: because of the random nature of stochastic search, certain boards which take days for our implementation of backtracking to solve are able to be solved in minutes by HGASA.

References

[1] A. K. Mackworth, "Consistency in networks of relations," Readings in Artificial Intelligence, pp. 69–78, 1981.

[2] B. Crawford et al., "Solving sudoku with constraint programming," Communications in Computer and Information Science, pp. 345–348, 2009.

[3] Chel, Haradhan and Mylavarapu, Deepak and Sharma, Deepak. (2016). A novel multistage genetic algorithm approach for solving Sudoku puzzle. 808-813. 10.1109/ICEEOT.2016.7754798.

[4] D. Job and V. Paul, "Recursive backtracking for solving 9*9 sudoku puzzles," Bonfring International Journal of Data Mining, vol. 6, no. 1, pp. 07–09, 2016.

[5] E. C. Chi, K. L. Lange, 'Techniques for Solving Sudoku Puzzles', ArXiv, t. abs/1203.2295, 2012.

[6] M. Becker, "Solving Sudoku," Sudoku – solving techniques, 2013. Available: https://www.stolaf.edu/people/hansonr/sudoku/explain.htm.

[7] M. Ercsey-Ravasz and Z. Toroczkai, "The chaos within Sudoku," Scientific Reports, vol. 2, no. 1, 2012.

[8] S. C. Brailsford, C. N. Potts, B. M. Smith, 'Constraint satisfaction problems: Algorithms and applications', European Journal of Operational Research,119. 3, 557–581, 1999.

[9] M. Perez and T. Marwala, “Stochastic optimization approaches for solving sudoku,” arXiv.org, 06-May-2008. [Online]. Available: https://arxiv.org/abs/0805.0697.
[10] S. Forrest, “Genetic algorithms,” ACM Computing Surveys, vol. 28, no. 1, pp. 77–80, 1996.
[11] “The history of Sudoku,” classic. Available: https://sudoku.com/how-to-play/the-history-of-sudoku