Sudoku Solving Genetic Algorithm

Bush Nghiem

December 2022

Abstract

Sudoku is a game popularized in 1984 in Japan where the word Sudoku comes from, meaning "the numbers must remain single". Over time there are have been many techniques and shortcuts developed that help players optimize and solve the puzzle faster. Evolution is a process that is similar in that it also slowly optimizes things over times to be better and thus evolution can form the basis for a good solution seeking program. So the major goal of this project is to make a genetic algorithm that correctly solves Sudoku puzzles when given to it. The algorithm will be tested with three different variable to properly assess how good the program actually is and give insight to how the program can possibly be better.

1 Introduction

The project was entirely done by just one person, Bush Nghiem. The problem that this project will intend to solve is based on the iconic puzzle Sudoku. While seeming simple at first, Sudoku when looked at a deeper lens will unravel many hidden patterns and intricate solution methods [2]. So the main objective of this project will be to write a program that will solve any normal Sudoku puzzle, given that the puzzle has a solution of course. There should be some emphasis on the "normal" aspect because Sudoku at this point has so many different variations now that solving more than one type in the same program would be exhausting as there are around $6.671 * 10^{21}$ normal Sudoku grids possible [3]. The reason for working on this project is because Sudoku is an entertaining puzzle which so many people have spent their free time on and the thought of figuring out how to make a program that solves them would be entertaining in its own right. This topic also happens to be related to this class at the same time, so all things just aligned together for this to happen. An important thing to mention it that the program will be written exclusively on Python.

The method that will be used for solving this puzzle is a local search algorithm since it is a puzzle and that means that it is environment is fully-observable, deterministic, etc, and the solution state for a Sudoku puzzle is easily testable as well. More specifically the selected algorithm for the project is an evolutionary/genetic algorithm, this is not particularly novel idea as it has been done before [9], but it would be a good choice since Sudoku can be represented as just strings which works well with genetic algorithms.

Before going any further with the state representation and the finding how "fit" a state is, prior knowledge is needed as one need to know what Sudoku is and how to solve a Sudoku puzzle. It consists of a 81 by 81 board with nine 3 by 3 subgrids/quadrants build into it, the objective of the game is to completely fill the board with digits where every column, row, and 3 by 3 quadrant must have all digits 1-9 with no duplicates. So the initial plan is to have the state be represented by nine strings that represent one row each and all of them together will represent the whole Sudoku grid. The way to determine how good a certain state is will be a heuristic that checks each row, column, and 3 by 3 quadrant, it will somehow find out how "good" the state is and sum it all up in a number that should represent how "good" the state is. The algorithm will then select the "fittest" states and allow them to "reproduce" and make a new state that will be a mix of the two states with a chance of a mutation altering the state, and this process will go on for many generations.

In the end, the goal is that the program will find a solution to the puzzle. But it is also important to find out what makes the algorithm good and what are ways to make the algorithm even better.

2 Background

The theory of evolution has fundamentally shaped humanities understanding of the world ever since it was introduced by Charles Darwin back in 1859. It is not hard to see why it has had such an impact, it gives an explanation for how and why every living organism functions the way it does. The natural process in which generations upon generations of organisms slowly adapt to their environment guarantees that every living being is optimized for survival. So when it comes to humanity and the problems that they face, it should not be surprising people tend to mimic nature and solve problems the way that nature had solved them through evolution. The field of biomimetics embodies this thought process fully which has lead to many breakthroughs and innovations. One of these breakthroughs appears in the field of computer science in the form of genetic/evolutionary algorithms.

Taking ideas from nature is hardly a novel concept, in fact it has occurred throughout all of human history. What started initially as humans based their rudimentary tools off of animal teeth has turned into humans designing their planes based on birds [6]. While biomimetics can simply be explained as just "copying" nature, it is really more akin to taking inspiration from natural sources and building upon those sources. While airplanes were originally based on birds, the actual designs ended up being extremely distinct from its avian progenitors. The results of biomimetics nowadays can be seen virtually everywhere, from drills being based on wood wasps to hydrophobic surfaces based on lotus leaves [11], the roots of biomimetics reach world wide with commercial results also being very nice as biomimetic products continue to generate billions of dollars [1]. Biomimetics is a practice that will continue to grow and intertwine itself with many other fields of study like engineering, medicine, chemistry, etc.

One field that might not seem to have any practical use of biomimetics is computer science, where would nature even help in a field that consists of nothing but computers is

what one might think. But biomemetics actually makes a sizeable impact in computer science through the idea of genetic/evolutionary algorithms. Evolutionary algorithms are a strange piece of biomimetics because rather than taking a product of evolution and generating the new idea from it, evolutionary algorithms actually draws inspiration from evolution itself (if the name didn't make it obvious). Evolutionary algorithms originated in 1975 by J.H. Holland [4] and as previously mentioned, are based upon the principle of Darwinian evolution. The major concepts from evolution like are natural selection, survival of the fittest, mutation, etc, are represented in genetic algorithms for the purpose of solving which ever problem is required from it.

Genetic algorithms can be described in a multi-step procedure, starting with creating a initial population which are represented as string, evaluating the fitness of the population, create a new population, replace the original population, test if the population is a solution, repeat [5]. The most complicated part of the process mentioned is the creation of a new population, this step alone consists of three separate steps called selection, crossover, and mutation. The selection process has had many techniques developed over the years to refine it with some examples like simple roulette wheel selection, tournament selection, Boltzmann selection, etc [7]. But to explain the process in simplest terms, the selection process is the step that decides which string will be able reproduce and make a child for the new population. The crossover is the next step when creating a new population and it is the reproducing part of the step. Crossover is the step that takes the individuals selected from the selection process and combines them to make a new string for the new population. Like the selection process, there are many different ways of approaching this method with some being one-point crossover, uniform crossover, heuristic crossover, etc [8]. The final step to creating a new population is mutation which is the process to randomly alter parts of the new string generated by the parents. While this step might not seem necessary since selection and crossover should technically create new populations that look more and more like the solution, mutations create new information for the algorithm to take in which may make the algorithm less stagnant.

All these many techniques and processes all developed for genetic algorithms were created with the sole purpose of making the algorithm more efficient in its attempt to solve the problem. Ironically genetic algorithms face a variety of its own issues. For example, one extremely common issue is premature convergence, when the program stops too early at a non-optimal solution [7]. There are many more issues pertaining to genetic algorithms but the pros tend to out weight the cons enough for genetic algorithms to still be widely used. In fact genetic algorithms are still used for problems that arise in many different fields of study. It has obviously been used for computer science applications like machine learning and automatic programming, but its has also been used to create economic models, model the various parts of the immune system, generate social systems, and much more [10]. So despite some underlying issues with genetic algorithms, there still is high demand for the application of it.

With genetic algorithms still in such high usage to this day despite some limitations overall, it remains as a testament to how practical interpretations of natural design can

truly be. Genetic algorithms prove that biomimetics can be used to make natural solutions for problems that may seem unnatural at first glance. Humanity will continue to face new issues and problems, but the simplest solutions are the ones that have already been created. Using nature as prototype for new ideas will never stop being intuitive and genetic algorithms are just one example of that.

3 Problem Solving Approach

So now with all this prior knowledge on Sudoku and genetic algorithms, the question that now remains is how to put it all together. Initially, the plan was to have the state be represented by nine strings made up of nine numbers where each string represents one row of the puzzle and the way to determine how good a certain state is will be a fitness that checks each row, column, and 3 by 3 quadrant. This test will check how correct those qualities are and it will then add it all up as a single number that should represent how "fit" the state is. The algorithm then selects the "fittest" states and allows them to "reproduce" and make a new state that will be a mix of the two states with a chance of a mutation altering the state.

While many of these starting points did remain in the final product, there were many adjustments to make the process easier and more efficient overall. While the idea of having nine strings representing the Sudoku grid seemed good at first as it was easily understandable as each string could represent a row which is quite intuitive, but implementing it ended up being a nightmare as it would become necessary have to refer to nine strings every time to access the whole grid and the whole thing was just inefficient in general. So for state representation, a class was created and a string property was attached to represent the grid. The string has a length of 81 with nine digit intervals representing each row, so the [0:9] interval would represent the first row of the grid and [72:81] would represent the last row of the grid. This method of representation is much better than the original plan, not only is it easier to implement and work with, but it is also much more efficient as it only takes one string as opposed to nine. The string itself is comprised of nothing but numbers ranging from zero to nine with zero representing an empty space and the other numbers representing the actual number being filled in the grid.

With the grid being represented, the rules of Sudoku would have to be implemented as well. Many functions were defined to check the grids and see if they were correct. There were three functions, checkrow(), checkcol(), and checkquad(). These functions would look at the desired row, column, or quadrant, and it would check for any duplicate numbers in the interval. If any duplicate numbers were detected, it would then return false and true otherwise. They are simple to use as each had only one argument required. The argument needed was just a single integer ranging between 1-9 and it refers to the desired row, column, or quadrant. Rows were easy to implement as the grid was designed to have nine digit intervals represent the rows and thus the integer argument could be taken and just multiplied by 9 to get the desired interval, but columns and quadrants were not so simple. The way that columns were created was that the function took the integer argument as an index and then grab that index from every row and compile to a single string representing

the column. Quadrants were the most complicated to check as they required specific three digit intervals all from different rows, so the way to obtain the required quadrant is to first differentiate the quadrants into three sections 1-3, 4-6, and 7-9, each section required three digit intervals from different rows. Once the sections were defined, getting the quadrant that was needed was similar to columns but instead of one digit intervals from all nine rows, it is instead three digit intervals from three rows. The three check functions would all be manifested into one isWin() function. This function ran one loop to use all three check functions nine times for each row, column, and quadrant, and only one check function returning false would result in isWin returning false. If none of the check functions returned false at all, then isWin() would return True.

The final thing needed for the representation of Sudoku is a way to fill in the empty squares. Previously mentioned was how the empty spaces were represented as zeros, so two functions were created to deal with these, isComp() and complete(). The function isComp would detect any zeros in the string and return false if any were found. The complete function is one that would find all zeros and replace each of them with a random number from one to nine.

Even with state representation finished, a major piece of the puzzle was still unfinished as the algorithm had yet to be designed. Luckily a majority of the steps required for genetic algorithm was already described in high detail [7]. So implementing the algorithm would simply be treading on grounds that have already been paved. The class named geneAlg would be created to neatly organize all the needed functions and properties in once place. Representing a population was something that was not initially planned out, but it was decided that the best way to represent the population is a list that contained nothing but the objects representing the grid described in detail earlier. This list would be filled with the same starting grid but the complete function would then randomly fill each grid with different numbers which would make up the initial population. This made each individual of the population easy to access and also allowed for sorting which would be essential.

The next required function would be a way to somehow evaluate how fit an individual of the population is. The first idea that came to mind for determining the "goodness" of an a Sudoku grid is using the already built check functions to see how many rows, columns, and quadrants are correct and then add points for that. This idea would fall apart quickly as all of the initial population would have the same fitness as every single individual had no correct rows, columns, or quadrants at the start. Another method that was thought up was to count all the duplicate numbers in the grid and then subtract points equal to that. This idea also fell apart quickly when implemented because all grids end up having 81 duplicates in total. While the counting duplicate method failed, it was not totally scrapped. The method actually utilized still counted duplicates, but it was more precise in the end. It would be made up of three functions that each looked through the rows, columns, and quadrants like the check functions before. But instead of just looking for a single duplicate, it would count the amount of duplicate numbers in the desired row or column or quadrant. So the final function fitness() would use all three functions to count the amount of duplicates in each row, column, and quadrant separately and add them up to a single number.

Now with a way of determining how fit an individual is, all that is required is the process of making an offspring from them. This process is split into crossover and mutation, both being essential to the overall goal. The crossover function was created first, it worked by taking two grids as arguments and then randomly selecting an interval of the string from the first grid and splicing it into the second grids string to make a new string. The mutation function first takes one grid as an argument, it randomly selects a digit in the string, and then it replaces that digit with a random number from one to nine.

All of these things accumulated together finally result in a full representation of a Sudoku grid and all the required parts to make a functioning genetic algorithm that solves Sudoku grids.

4 Design of Experiments and Results

With all the necessary prerequisites in place for a genetic algorithm, the experimentation of how effective it can be can take place now. Even though the way the algorithm works is entirely reliant on randomness, there were some adjustable variables encoded into the algorithm that could be controlled and tested. The three variables that could be changed are the selection rates, mutation rates, and the size of the population. For further explanation, the selection rate in this program the size of the population that gets to reproduce and make new offspring. Since the fittest individuals are desired, the fittest of the population are the ones to be included in the selected population. For example, if the selection rate was 25%, the top 25% of the current population each have an equally random chance of reproducing. Selecting a certain top percentage of the population was easy since the population was put in a list, a sorting algorithm was made that ordered the population from fittest to least fit. The mutation rate is simply the chance that each offspring has to randomly mutate and have the mutation function applied to it. Finally the population variable is self-explanatory, it is a variable that determines the size of the population.

So testing how effective the algorithm will need to be done in three separate steps, testing how much population size will have an effect the algorithm, testing how much selection rate will have an effect on the algorithm, and testing how much mutation rate will have an effect on the algorithm. In order to effectively do this, there must be tests done on the algorithm where only one variable at a time will actually vary and the rest will be constants for that test. There must also be a measurable value that can properly assess how well the algorithm has done. This all has to be done with the same Sudoku grid to maintain consistency among all the trials.

So three separate tests will be happening, one where selection rate is a variable, population size is a variable, and mutation rate is a variable. The value that will be measured is the fitness() value of the best individual of the population at the thirtieth generation, this value was picked since it does show how close the algorithm is to the solution and thirty generations is a good cutoff time because if the algorithm has not found an answer by then, it will never find the answer at all and is probably stuck. One thing to note is that the value is better when it is lower with zero being the best possible value, this is because the fitness

function counts the amount of duplicate values which is undesirable. There will be five trials done for each variable with 75 trials total being done for all variables. This means for each variable, there will be five variations of the variable to test five times each. This is done to see the approximate average for the trials and view the effects of each variable clearly.

4.1 Population Size

With selection rate being a constant of 10 percent, mutation rate being a constant of 25 percent, and population size being a variable

Trial	Size 100	Size 200	Size 300	Size 400	Size 500
1	36	16	6	12	8
2	38	18	16	4	4
3	24	24	18	4	8
4	35	16	4	19	6
5	22	24	14	0	10
Avg	31	19.6	11.6	7.8	7.2

4.2 Selection Rate

With the population being a constant of 200, mutation rate being a constant of 25 percent, and selection rate being a variable

Trial	S.Rate 10%	S.Rate 20%	S.Rate 30%	S.Rate 40%	S.Rate 50%
1	12	16	48	70	86
2	20	23	41	64	86
3	22	18	37	71	72
4	23	24	52	66	73
5	18	8	787	66	82
Avg	19	19.8	45.4	67.4	79.8

4.3 Mutation Rate

With selection rate being a constant of 10 percent, population size being a constant of 200, and mutation rate being a variable

Trial	M.Rate 10%	M.Rate 20%	M.Rate 30%	M.Rate 40%	M.Rate 50%
1	37	14	25	31	22
2	28	20	20	18	22
3	34	18	12	22	14
4	36	21	27	18	24
5	28	22	8	25	24
Avg	32.6	19	18.4	22.8	21.2

5 Analysis

After testing all the trials and recording the data, some very clear patterns can be seen and many different conclusions may be drawn. But it is important to remember that understanding the data and trying not to get the wrong idea is just as important as the data itself.

A very obvious pattern appears in the population size data, the higher the population size, the better the algorithm does in general. This makes sense intuitively as a larger population will have a larger variety of information to pull from, and more information to pull from will increase the chances that the correct answer will be found and less chance for the algorithm to get stuck. So having a large population size increases the chances that the algorithm will find the correct answer sounds great at first, but it does not take account of the time taken for each trial. The a single trial with a population size of 500 took about five minutes while trial with a population size of 100 took only 12 seconds. So the algorithm would most likely find the correct answer every time if you gave it a population size of 1000000, but the problem is that this would take a colossal amount of time.

For the selection rate data, an inverse relationship seems to be found. The larger percentage of the population to pull from actually makes the algorithm worse. This makes complete sense of course, since the algorithm automatically pulls from the fittest individuals anyways. Increasing the pool of the individuals to pick from would just increase the amount of undesirable genes which would be bad.

For the mutation rate data, the pattern might not be as obvious. The data seems to suggest that having too high of a mutation rate makes the algorithm worse, but having mutation rate that is too low is also a bad thing as well. The best mutation rate seems to lie somewhere between 20 percent and 30 percent. So this intuitively makes sense because if the mutation rate is too high, then there is a higher chance of a mutation lowering the fitness of an individual. But if the mutation rate is too low, then the algorithm will stagnate and never reach the correct answer. So there is a Goldilocks's situation where the mutation rate has to be not too low, and not too high, but just right.

Something that one may notice is that the algorithm does not find the answer that often. Throughout all the trials, the true answer was found only once, but this was purposefully done. If the tests were done in a way which the answer was found too often, then many data points will be zero and nothing of note would be found in a table filled with zeros. So if the correct answer was truly desired in this experiment, the base population size would be set

higher to a number like 1500 instead of 200 (this would take an ridiculously huge amount of time).

6 Conclusion

After reading all the data and analyzing it, it has become extremely apparent on what makes variables a genetic algorithm good and what can be done to make this program much better. Starting with what variables make the algorithm good, generally the larger the population is, the more likely the program is to obtain the correct solution. But this does have a downside as it also exponentially increases the amount of time the program takes to run. This brings up one potential way of making this algorithm better and that is to make the code more time efficient, the code as it is clearly not perfect and it does take way too much time for even just a population size of 500. There are some loops in the code and the sorting algorithm is not perfect, so those are ways to further optimize the algorithm and make it more time efficient.

The selection rate is the variable that when lowered makes the algorithm better, but this was only because the way selection worked in this program was that it randomly selected an individual to be a parent from that top percentage. If parents were selected in a more systematic approach, then it could be possible that having a higher selection rate would make the algorithm more accurate as it is pulling more information from a larger pool rather than a small percentage. So another way to potentially make this program work better is write the code in a way that does not just randomly selects the parent from a "good" pool, but selects parents that would better suit each other according to their needs. For example, if one grid fulfilled many of the the correct conditions except some rows, then the algorithm would look for suitable individuals that did have those rows correct. This would make the algorithm more accurate but at the same time it might make it less time efficient, so there is a potential drawback to everything.

The mutation rate was an odd variable to deal with because there was not an obvious "perfect" mutation rate to strive towards, but even then there are still possible ways to improve upon the algorithm dealing with it. The major reason for not wanting too high a mutation rate is that it makes the population too unstable and that makes it hard to find the solution if it changes too much. But like the selection rate, if the algorithm was written in a way which mutations would not occur on parts of the string that are desirable and only on incorrect places, then a higher mutation rate would be better as it only targets the spots that could potentially make the fitness better rather than worse.

Perhaps the thing take away from all of this is that while the algorithm is functional, there is still a massive list of things to further improve upon the program. The thing to remember is that all of this work is done to solve a simple number puzzle, and it still does quite a subpar job at solving the puzzle efficiently. But perhaps with enough time and effort, this program will be improved to the point of doing nearly perfect job much like how evolution slowly turned simple single celled beings into the creatures seen around the world today.

References

[1] Bharat Bhushan. Biomimetics: lessons from nature–an overview. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1893):1445–1486, 2009.

[2] Tom Davis. The mathematics of sudoku, 2006.

[3] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible sudoku grids. *Preprint available at http://www. afjarvis. staff. shef. ac. uk/sudoku/sudoku. pdf*, 2005.

[4] Stephanie Forrest. Genetic algorithms. *ACM Computing Surveys (CSUR)*, 28(1):77–80, 1996.

[5] L Haldurai, T Madhubala, and R Rajalakshmi. A study on genetic algorithm and its applications. *International Journal of Computer Sciences and Engineering*, 4(10):139, 2016.

[6] Jangsun Hwang, Yoon Jeong, Jeong Min Park, Kwan Hong Lee, Jong Wook Hong, and Jonghoon Choi. Biomimetics: forecasting the future of science, engineering, and medicine. *International journal of nanomedicine*, 10:5701, 2015.

[7] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, 2021.

[8] Ashima Malik. A study of genetic algorithm and crossover techniques. *International Journal of Computer Science and Mobile Computing*, 8(3):335–344, 2019.

[9] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. In *2007 IEEE congress on evolutionary computation*, pages 1382–1389. IEEE, 2007.

[10] Tom V Mathew. Genetic algorithm. *Report submitted at IIT Bombay*, 2012.

[11] Amitava Mukherjee. *Biomimetics: Learning from Nature*. BoD–Books on Demand, 2010.