Genetic Algorithm Term Project

Casey Descoteaux

**Table of Contents**

[**Abstract** 3](#_Toc7021650)

[**Problem Description** 4](#_Toc7021651)

[**Introduction** 4](#_Toc7021652)

[**What is an NP Complete Problem?** 4](#_Toc7021653)

[**What is a Genetic Algorithm?** 4](#_Toc7021654)

[**Problem Statement and Expected Goal** 5](#_Toc7021655)

[**Summary** 6](#_Toc7021656)

[**Methodology** 7](#_Toc7021657)

[**Introduction** 7](#_Toc7021658)

[**Detailed Approach** 7](#_Toc7021659)

[**Data Structures and Algorithms** 8](#_Toc7021660)

[Child and Poly Instances 8](#_Toc7021661)

[Crossover and Mutation 9](#_Toc7021662)

[**Summary** 10](#_Toc7021663)

[**Results** 11](#_Toc7021664)

[**Introduction** 11](#_Toc7021665)

[**Experimental Problems** 11](#_Toc7021666)

[Problem 1: Required Precision 11](#_Toc7021667)

[Problem 2: Child Object Design 12](#_Toc7021668)

[Problem 3: Charting Results 13](#_Toc7021669)

[**Results** 13](#_Toc7021670)

[**Runtime Analysis** 16](#_Toc7021671)

[**Summary** 16](#_Toc7021672)

[**Conclusion** 17](#_Toc7021673)

[**References** 18](#_Toc7021674)

# **Abstract**

This paper is comprised of three sections, a problem description, a methodology, and a results section. The problem description will review the definitions of an NP-Complete problem and a genetic algorithm along with their relation and application to this project. This section will also discuss the problem that is going to be attempted and how it is suitable for a genetic algorithm. The methodology section will discuss some of the core functions in pseudocode to describe how the genetic algorithm elements will be implemented. The data structures that are going to be used in the program and their responsibilities will be discussed in some detail as well. Lastly, the results section will discuss some of the challenges that were faced in this project in some detail along with their solutions. Some specifics on the results that the algorithm is capable of producing along with some runtime analytics will be reviewed as well. The paper ends with an overall conclusion of the state of the algorithm and if it is fit for use.

# **Problem Description**

## **Introduction**

Through the advancements of artificial intelligence, it is now possible to find solutions to otherwise unsolvable problems in terms of both time and complexity. These problems are referred to as NP-Complete problems. This paper will discuss what an NP-Complete problem is and the challenges they present. This paper will also talk about intelligent systems as it pertains to genetic algorithms. Solutions to these NP-Complete problems can be found using these algorithms. Lastly, this paper will discuss an example of an NP-Complete problem that could be solved with the use of a genetic algorithm.

## **What is an NP Complete Problem?**

NP-Complete problems are problems that, when large enough, cannot be feasibly solved by a human. NP stand for nondeterministic polynomial, meaning that if a solution to the given problem can be found it can be found in polynomial time (Encyclopedia Britannica, n.d.). Note that the goal is to find a good enough solution to the problem, not the optimal one. NP-Complete problems that are limited in size can feasibly be solved through intuition however it is when these problems become extremely large where computers are introduced to assist.

## **What is a Genetic Algorithm?**

Computers, using a genetic algorithm, can be used to find solutions to otherwise unsolvable NP-Complete problems. Genetic algorithms use a binary string and a fitness value to represent the different population members in the current generation and to determine how close they are to finding a solution. The goal of these algorithms is to simulate evolution by taking a random initial population and applying crossover and mutation to create the next generation. Crossover is the process of taking the same section of the binary string from two members in the population and swapping them. This process creates the children for the next generation (MathWorks, n.d.). There are many methods to determining which members to cross such as biased random selection. Biased random selection increases the likelihood of members with higher fitness values being crossed together with the hope that continued increase in fitness will occur. Mutation is the process of flipping a bit in the binary string of a population member. This process increases the variability of the algorithm and can potentially lead to improvements in fitness. Once the given number of generations have taken place, the fitness values are evaluated to determine if a satisfactory solution was found. Genetic algorithms are not implemented with the intention of consistently finding an optimal solution to a problem, although the optimal solution may come up occasionally, the primary goal is to find a satisfactory solution.

## **Problem Statement and Expected Goal**

An example of an NP-Complete problem that could be solved with the use of a genetic algorithm is the polynomial satisfiability problem. This project initially started out with the Boolean satisfiability problem which states that given a Boolean formula, is there an assignment of variables that makes the formula true (WolfranMathWorld, 2019). However, after some initial testing it was determined that a genetic algorithm is not the correct approach for this problem due to the limited variable values, only true and false. In order to maintain a similar theme with increased variability the problem was changed to satisfy a random polynomial with floats as the variable values multiplied by an exponent. These variables, the number of which is determined by the user, and their exponents will be generated at random and will be tested for fitness based on how close the outcome falls to the user supplied target. This is an NP-Complete problem since the population of variables that make up the formula could become too large to be feasibly handled by humans. One challenge that could arise is ensuring that the algorithm is performing the correct order of operations which can feasibly be resolved by testing future operations to ensure that expressions are being tested in the correct order.

## **Summary**

Using genetic algorithms, it is possible to find solutions to problems that would otherwise not be feasibly solvable my humans. These problems are referred to as NP-Complete problems, problems that can be solved in polynomial time by a computer. One method for finding a solution to one of these problems is a genetic algorithm. These algorithms simulate evolution by crossing segments of binary strings and mutating bits in order to create new generations. This continues for a given number of iterations with fitness values being adjusted every generation. These adjusted fitness values factor into the biased random selection for crossover when the next generation is created. When the given number of generations have been created, the binary strings are evaluated using their fitness values one last time to determine the most satisfactory solution. It is not feasible to expect the optimal solution for the problem as this process would require testing every combination of variables which is not feasible for extremely large populations. An example of such an NP-Complete problem that can be put through a genetic algorithm is the polynomial satisfiability problem. This problem deals with the orientation of variables in a polynomial equation to find a set of variable values to meet a given target.

# **Methodology**

## **Introduction**

This paper will describe in more detail the specific actions that are going to be taken to create the polynomial genetic algorithm. The first section will discuss the implementation of the genetic algorithm fundamentals such as what the “genome” will look like, how crossover and mutation will be implemented, how the fitness function will be calculated, and the population size. Additionally, more details about the data structures, classes, and algorithms that will be used to accomplish these feats will be discussed in some detail. These elements were heavily influenced by the Boolean satisfiability problem trial that was attempted before since the equation layout and expression solving in both scenarios share some similarities.

## **Detailed Approach**

The detailed approach will describe in some detail what the genome will look like as well as how crossover and mutation will be implemented. Other fundamentals such as calculating fitness and determining population size will also be discussed. The population size is set to one-hundred and is not alterable due to it being used to define array lengths. In C++, the language used for this algorithm, arrays can only have their size defined by a constant variable. The major aspects that are held in each genome are the current state, the fitness value, and the variables used. The lists holding the current state and variables used are made up of another class which holds all of the information required for an individual polynomial variable including the variable label (X, Y, Z, etc.), the base value, the exponent, and the operation to be performed between that variable and the next variable. The only operations used are addition and subtraction, not only did they prove to be satisfactory, they also provide greater performance than adding the options of multiplication and division as this would require additional if testing and increased calculation logic when dealing with order of operations. The resulting value after all variables have been analyzed will be tested against the user defined target value to determine the fitness for the genome with a value of zero being a solution. Lastly, the genomes will have to undergo crossover and mutation between each generation.

Crossover will be performed by swapping the values of the back half of the variables used in the equation of the selected parents and inserting them into the other to create two children for the next generation. Since the current state is made up of a custom class that holds all variable information the risk of selecting an index in the middle of a variable’s data is mitigated. Mutation is another function that has to be addressed as well, since the goal is to find any equation that provides a satisfactory solution, this would suggest that each Child could consist of a different equation. However, this presents too many random elements for the algorithm to function effectively. With the equations being the same, the subject of mutation is the value of each variable as this is the core factor that differentiates each Child. Although the variable objects are made up of two values, the base and the exponent, the base is the only value that is changed. This is again to avoid implementing too many random factors that negate the effects of the algorithm.

## **Data Structures and Algorithms**

### Child and Poly Instances

In this section, some pseudocode on how the above concepts will be performed will be discussed. Due to the level of variability from user input, lists are used in many cases instead of arrays. In the genome, called Child, the list of variables, called Poly, are held in a list called currentState. When the program is first run the user is asked how long the equation should be, how many unique Poly variables the Child objects have to choose from when creating the equation, how much mutation should occur, and the target value. When the Child objects are initially created their Poly objects that make up the equation have their base value and exponent randomly assigned between values -10 - 10 and 0 – 10 respectively. Once the equations are set they are solved and the resulting value is stored in a runningTotal variable held by Child. Due to the restricted use of addition and subtraction no logic regarding order of operation is required which improves performance. Once all Poly objects in currentState have been evaluated the runningTotal is compared to the target value, using absolute value, to set the fitness for the Child. The main then calls a method that sorts the Child objects in the population based on how close their fitness values are to the target leading to a solution being a fitness of zero.

### Crossover and Mutation

Between generations this sorted list is put through a biased random selection to select the parents that will have their equations crossed to create the next generation. Crossover takes the mid-point of the used variables from the two parents and swaps the back halves. In order to accomplish this an iterator set to the vector of used variables objects is incremented until it is at the midpoint of the vector, represented as usedVars.size() / 2). This variable subset is sent to a function of the other parent so that the values of the selected variables can be altered. Mutation will also in this area of the program by taking a percentage of the population, based on the user determined mutation level, and select variables at random to have their base values changed, with the number of variables effected determined by the original variable pool entered by the user. Although the number of variables effected is based on the original pool, only the used variables are altered. Mutation is carried out in this way, as opposed to using the used variable pool count, to ensure that instances of low used variables will still receive sufficient mutation. After crossover and mutation occur the equations have to be re-evaluated to ensure that all instances of the Poly objects under the effected variable names are changed to their new values.

## **Summary**

By using two custom classes, Child and Poly, a population can be created with each member of the population having an identical equation with a variable pool based on user input. With each generation the members of the population are evaluated and are tested by how far their result falls from the target value. Once this is complete the next generation can be formed by using a biased random selected to select two parents to have their used variable values crossed and potentially mutated. Once the members of the new generation have had their equations re-evaluated the process loops a given number of generations. Given the increased variability between this problem and the initially attempted Boolean problem, it is significantly less likely that a solution will be found in early generations.

# **Results**

## **Introduction**

This paper will provide a conclusion and overview of the development cycle for this genetic algorithm project. This paper will discuss some of the substantial challenges that had to be overcome in order to get the program to its current state. Results of the program including the effectiveness, speed, and output charts will also be discussed and presented. Since one of the objectives of the algorithm is to find a satisfactory solution in a reasonable amount of time, runtime analytics will be discussed as well. Lastly, a summary will be presented detailing the level of success compared to the expectations of the algorithm.

## **Experimental Problems**

Throughout the course of this project there have been three major challenges that had to be overcome. First was the problem of required precision from the algorithm. In many common NP-Complete problems put through a genetic algorithm, the outcome is in question while the pieces to obtain a solution are known. Solving a polynomial is the opposite, in order to determine if the equation is satisfactory, a target value must be used as a comparison while all of the variables in the equation are not known. Second was the initial attempt of the polynomial satisfiability problem that allowed each Child object to have a unique equation. This challenge required various changes in different areas of the program to properly correct. Lastly, the third challenge that will be discussed in this section will be the challenge of plotting the results.

### Problem 1: Required Precision

Many NP-Complete problems put through a genetic algorithm have the goal of orienting the known objects to obtain a satisfactory solution. Due to this project being a math equation, this approach cannot be used because without a target value there is no way to gauge if the equation is satisfactory. This results in an approach that uses the solution value, the target value of the equation, as the known value while the variables that make up the equation are the unknown values. This means that there is exact precision required from the algorithm in order to find a solution to the problem. This problem was never overcome, however, the algorithm can come within 5% of the target value on a fairly consistent basis given the user input allows for enough mutation and provides a sufficient number for the variable pool. The random nature of a genetic algorithm becomes increasingly evident as the equation approaches the target value when specific values are required to improve the fitness. This should still prove to be satisfactory as this value can easily be accounted for by the user by adding a value to a variable to make up the difference. This result satisfies the goal of a genetic algorithm in that it takes a problem not feasibly solved by a human and turns it into one that can.

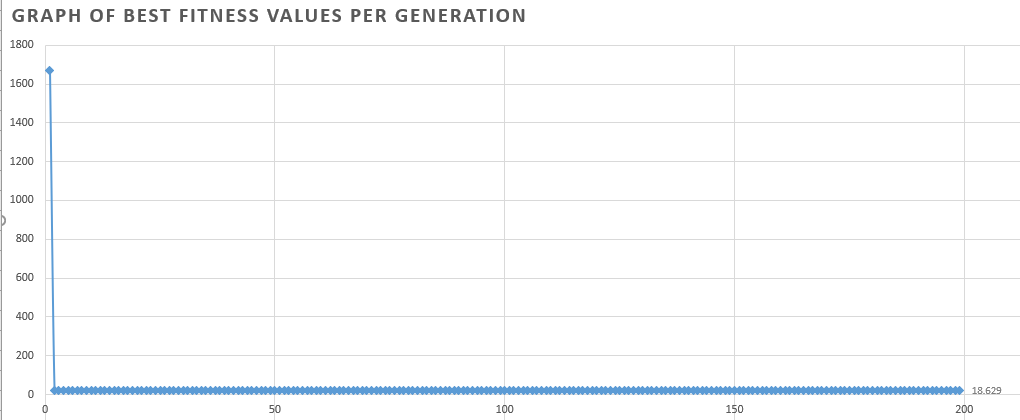
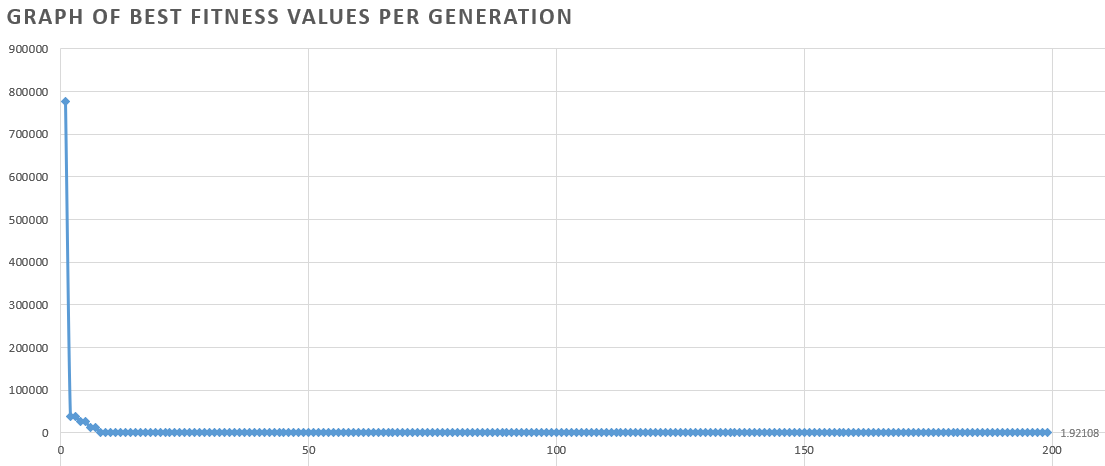
### Problem 2: Child Object Design

The initial design of the Child class allowed for each instance to have a unique equation since the primary goal is to find any equation that results in the user defined target value. This structure resulted in slow and/or erratic progress toward the target value. Due to each object having its own equation coupled with crossover that swapped the equation structure of the parents, there was no sense of progress from generation to generation. Since the equation layout was different from generation to generation it was like creating an initial population over and over. The only progress came from the mutation of the variables that would slowly turn to desirable values. This approach was far too random for a genetic algorithm. In order to resolve this, all Child objects use the same equation, maintaining the variable value as the subject of mutation. This change also required that the crossover method be changed since swapping a portion of the equations would not result in any changes to the Child equations. Crossover now occurs by swapping half of the variable values of the two parent objects. Now that the equation maintains the same for all objects, a sense of progression can be seen from generation to generation.

### Problem 3: Charting Results

The last challenge that will be discussed is plotting the results of the algorithm to a graph. For this project, the results of the algorithm are saved to an Excel spreadsheet that are then plotted on a graph. The first challenge was determining how to create this in such a way that allowed for a graph to be persistent and automatically update. This challenge arose since, in order for the same spreadsheet to be used each time the algorithm is run, the entire spreadsheet had to be cleared. To accommodate for this, a functionality in Excel was used that allows for one Excel project to reference another Excel project. By housing the graph in its own Excel project, it can still reference the range of values that will be populated by the algorithm and automatically update when both files are opened. The second challenge in this area was that, despite the algorithm making progress, the line of the graph was very erratic. This was a result of some generations having a negative value that was best fit while others had positive values that were best fit. By inserting the absolute value of the best fitness in each generation instead of the raw value, the graph becomes neater and the progress of the algorithm is easier to follow.

## **Results**

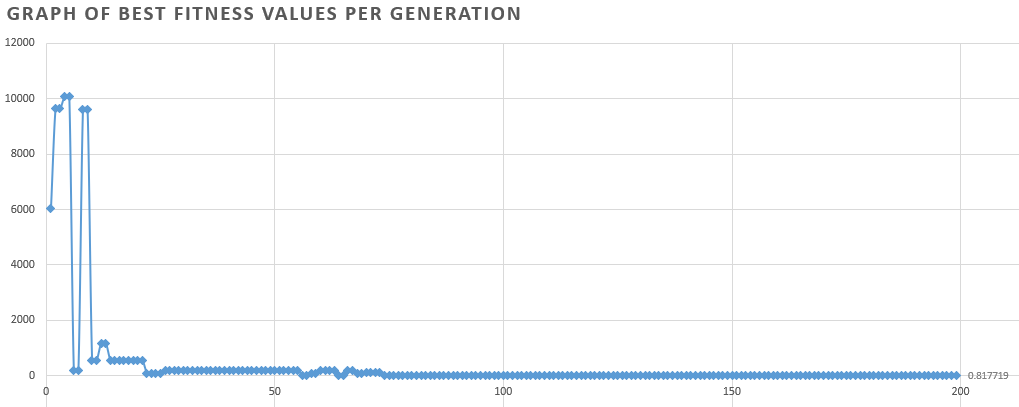
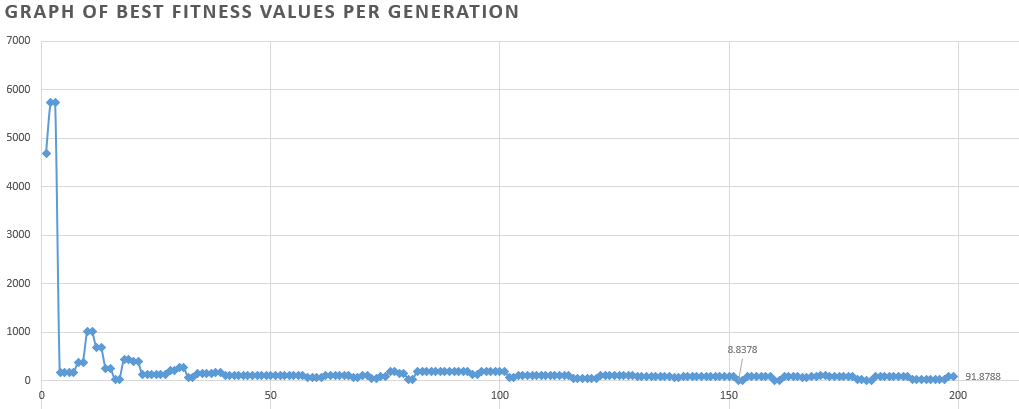
One of the personal goals for this project was to provide satisfactory solutions in 200 generations to make the algorithm as time efficient as possible, as is a goal of genetic algorithms. This goal resulted in higher mutation rate values than may be expected in other algorithms. The algorithm is able to find satisfactory solutions within the 200-generation window, however, sometimes the mutation rates result in objects with good fitness values being altered before they can pass their desirable values to a variety of other objects in the population. In the majority of cases this gets corrected over following generation but there are occasions where this is not the case. Since this algorithm was designed to find a satisfactory solution in just 200 generations, the algorithm can be run multiple times with good time efficiency if desired. The charts below are the results with a 30-expression long equation, variable pool of 10, target of 300, using the four different mutation levels (none, low, moderate, and high), over 200 generations. These graphs were taken from the first attempt of each mutation level with each test being performed in less than 1-minute.

Low mutation

Best fitness: 1.92108

No mutation

Best fitness: 16.629



High mutation

Best fitness: 8.8378

Last fitness: 91.8788

Moderate mutation

Best fitness: 0.817719

As expected, no mutation quickly resulted in a local maximum and high mutation resulted in the best fitness value not being the final value due to parents with good fitness values being changed before they can pass their variable values. Low and moderate mutation are much better suited as they allow for enough variability while not being too aggressive as to allow for better fitness values to spread their desirable variable values to other member of the population.

## **Runtime Analysis**

As discussed above, a 30-expression equation with an initial variable pool of 10 can be tested over 200 generations a little under 1 minute. If the number of expressions is changed to 100, keeping all other inputs the same, the runtime is about 3 minutes. Since only the used variables are subject to crossover and mutation, adjusting the variable pool does not result in a significant change the runtime unless the initial variable pool is extremely large. In order to obtain a satisfactory solution within the goal of 200 generations, the algorithm performs best if the target value is around ten times the equation length, if the equation is comprised of 100 or fewer variables, and the variable pool is between 10-20. Although the reasons behind the suggested target value are not certain, observations suggest that values too low are difficult to achieve with large equations due to the wide range of possible values for each variable (-10000000000 through 10000000000). A value of ten times the equation length is the value most consistently approached in 200 generations. Additionally, the 200-generation goal is best applied to equations of approximately 100 variables or less with variable pools ranging from 10-20. This variable pool size allows for some variables to be selected more than once when the equation is initially created. This means that if a variable is altered via crossover or mutation to a more desirable value, that change takes effect in multiple location in the equation. Larger equations, equations with varying variable pools, and equations with various target values can still be run, however, more generations are more likely to be required to find a satisfactory solution.

## **Summary**

Using 200 generations as a baseline, this algorithm performs fairly reliably, however, some equation parameters, including those that fall within the suggested bounds, may require multiple attempts to obtain a satisfactory solution for the user. Due to the random nature of a genetic algorithm, finding an equation that results in an outcome that matches the exact target value of the user is unlikely. This problem is unique to other NP-Complete problems in that the goal is known while the objects to get there are not whereas many other NP-Complete problems are the reverse. This challenge could not be overcome during development of this project, however, the algorithm is capable of providing an equation close enough to the target that the user can manually determine how changing a certain variable/variables would result in a solution.

## **Conclusion**

This paper discussed some of the larger challenges that arose during the development of this project including one that was never fully realized, the precision required from the algorithm. Genetic algorithms are typically used in cases where the target value is not known which does not apply to this case. This algorithm is designed to alter the objects in order to achieve a known target. Although obtaining the exact target is not done, the algorithm significantly simplifies the problem by providing the best equation it can create along with the remaining value between the result of the equation and the target. At this point the problem is human solvable. The performance measurements and results of the algorithm are based on 200 generations with a population of 100, reliably providing a value within 5% of the target, with fast runtimes which allows for the algorithm to be run multiple times if desired. Given the unique factors that this algorithm faces compared to other genetic algorithms, the results it is capable of producing should prove satisfactory.

# **References**

Encyclopedia Britannica. (n.d.). *NP-complete problem*. Retrieved from

https://www.britannica.com/science/NP-complete-problem

MathWorks. (n.d.). *How the genetic algorithm works*. Retrieved form

https://www.mathworks.com/help/gads/how-the-genetic-algorithm-works.html

WolfranMathWorld. (2019). *Satisfiability problem*. Retrieved from

http://mathworld.wolfram.com/SatisfiabilityProblem.html