Finding analytic solutions to equations using genetic programming and predator-prey dynamics

Daniel Rausch and Dr. Jeff McGough
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701
Dan.Rausch@gmail.com
Jeff.McGough@sdsmt.edu

March 16, 2004

Abstract

We present a genetic programming approach to finding analytic solutions to nonlinear algebraic equations. Having solved the general quadratic equation by evolving the quadratic formula, we will show results from that equation, as well as other algebraic equations containing exponential or logarithmic operators. Each potential solution equation is expressed as an S-expression consisting of operators, identifiers, and constants. This lends itself to storage in binary tree form. Reproduction involves crossover and mutation. Crossover is done by swapping a randomly selected subtree from both parents, and mutation involves changing operators, identifiers, or constants.

In any genetic programming approach, a key issue is the selection of an effective fitness function. A good fitness function increases the convergence to a solution. However, fitness functions often utilize some prior knowledge of the solution. Many genetic programming examples will explicitly define a fitness function in this manner, but for our approach the fitness of a potential solution equation is determined endogenously through the predator-prey model.

In each epoch, our potential solution equations interact with the algebraic equations. The solution equation will generate a value based on the parameters of the algebraic equation. By plugging this potential root back into the algebraic equation, the error is determined.

The higher the residual value is, the more it hurts the solution equation. If the residual value exceeds a certain threshold, that solution equation is killed off. The more successful a solution equation is, the more offspring it is able to produce.

There are many different areas that affect the convergence of this system. After reproduction, part of the equation may become useless (i.e. dividing by one, subtracting zero, etc), so we looked into the effects of pruning those parts from the equations have on the overall convergence. The tuning of the threshold value and other parameters will affect the system. Also, the set of operators available to the system will change the dynamics of the convergence.

1 Introduction

This paper concerns itself with the general solution of algebraic equations using an approach known as Genetic Programming. Genetic Programming (GP) is a popular form of evolutionary computing. There are several similar approaches such as genetic algorithms, evolutionary strategies, which with genetic programming, uses the principles and ideas from biological evolution to guide the machine to a desired solution. The differences in the approaches lie more in the details than in the overall view, that of imitating nature; and combined with simulated annealing and other biologically motivated algorithms have enjoyed recent popularity. Genetic programming specifically has seen a whole host of successes in solving very complicated problems. Koza [2] has many examples including circuit design, program design, symbolic regression, pattern recognition, and robotic control.

We normally have some fixed problem to solve or to optimize. The solution (or optimizer) lives in some set of potential solutions. We must search it out. In real examples, the search space is much too large to attempt a brute force search, so some method must be utilized to reduce the number of examined solutions. In analogy to nature, we see our potential solution as an individual in some collection or population of potential solutions. The individuals who are stronger, meaning higher ranked according to some cost/fitness function, will be used to determine the makeup of the next collection of potential solutions. By employing analogs of sexual reproduction (recombination) and mutation a newer and hopefully more fit population will arise each generation. After a number of generations, the most fit member then becomes a candidate for the solution to the problem at hand. [1]

Of course, the difficulty is in the details. Representing the potential solution in some manner that accommodates recombination and mutation is the first step. Identifying a fitness (cost) function and details of reproduction is a second, possibly more difficult, step. Then, tuning the parameters so that the population has members which solve the problem can be an art in itself. Overall, a great deal of tuning of parameters and cost function components must occur for GP to be successful.

Our goal is two fold here. First, we intend to solve nonlinear algebraic equations. It is a great problem to test out the effectiveness of GP. Second, we intend to do so without significant effort in expert systems. In other words, we would like to determine if we can produce a system capable of solving mathematical problems without a great deal of heuristics and expertise. In fact, can we solve these mathematical problems with no built in expertise at all? Goal one is then to produce a system that can solve a quadratic equation (thus we start with a well understood problem). We are not looking for a solution to a single equation - a root finder would suffice, but find the formula for the general solution. This can be solved by a computer algebra system like Maple, but these require significant rules sets (and apriori knowledge).

GP was selected since it seems to fit both goals. One problem immediately pops up. Don't we essentially code in significant expertise when we write the cost function? Maybe so.

To address this, we have the cost function implicitly defined and evolve so that the system creates it. Thus it builds the expertise. This is the motivation for the predator-prey co-evolutionary approach. It allows the system to figure out the cost function. It also opens the door to much slower convergence or no convergence at all (meaning convergence to a desired solution).

The basic idea is that the equations to be solved are considered to be "prey" and the formulas to solve then are considered predators. The process of a formula solving an equation is likened to a predator consuming a prey. The more successful a predator is the more prey it will eat. This equates to solving more equations. Prey are generated but the predators must evolve.

We had hoped to find that the GP produced a formula which was algebraically equivalent to the quadratic formula. This was not the case. We generated approximations which for long enough runtimes were reasonable accurate. Some speculation early on focused on Taylor approximations as being the expected result, which was not the case. The approximations have more in common with continued fractions as can be seen in the last section.

Section 2 presents the details on how we represent the objects in the machine and defines the basic problem. Section 3 presents the extension of the fitness function into the co-evolutionary approach. Section 4 give the details on how new generations are formed and how certain parameters are selected. Section 5 describes the results and what the results mean.

2 Data Storage

In order to simplify the arithmetic with all the calculations, we limited ourselves to real-valued variables and real-valued arithmetic. By construction, our polynomials are guaranteed to have real-valued solutions. Therefore, by not dealing with complex numbers, we avoid the issue of the multi-valued square-root operation.

2.1 Predators

The first step in storing the equations is to define a grammar to describe them. An S-expression grammar is used, not unlike those used in LISP. [2] A valid S-expression with our grammar is an identifier, a constant, or an operation, each enclosed in parenthesis. An identifier is a single letter. A constant is a positive or negative floating-point number. An operation consists of an operand followed by two expressions. An expression is an identifier, a constant, or another operation. The operands consist of, at minimum, the $+ - * /$ binary operators, which correspond to addition, subtraction, multiplication, and division,

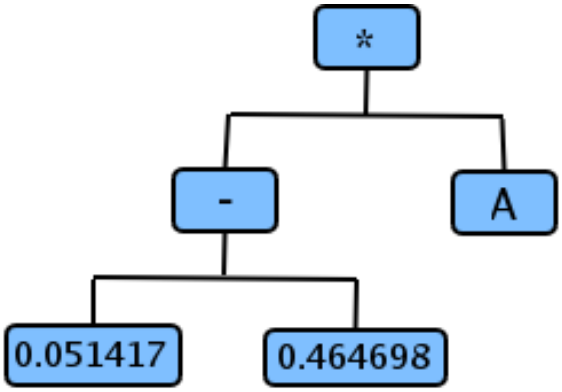

Figure 1: Equation in tree format

respectively. Other operators have been added, such as square-root and the square operators. In this case, the operator used for square root is &, and the square function is represented by $\wedge$.

This grammar lends itself to storage in binary tree form. Each leaf node correspond to either an identifier or a constant, and the other nodes are binary operators. When evaluating the tree, an array storing the numerical values of the identifiers is passed in. Any calculations done using an identifier will use those values. The evaluation function uses a depth-first algorithm to calculate the value of the equation. If a node is a constant or an identifier, the value is returned. However, if the node is an operation, it recursively evaluates the left subtree and the right subtree, and then applies the operand to those values.

Consider the following S-expression

$$(* (- \ \ 0.051417 \ \ \ 0.464698) \ A)$$

This equation is stored in tree form as shown in Figure 1. Given the condition $A = -1.236548$, this equation evaluates, in infix notation, to

$$(0.051417 - 0.464698) * (-1.236548) = 0.511042$$

This is an example of what the real quadratic formula would look like using the grammar previously defined.

$$(\ / \ (\ +(\ - \ 0 \ B \ )(\ \& \ (\ - \ (\ * \ B \ B \ ) \ (\ * \ (\ * \ 4 \ A \ ) \ C \ ) \ ) \ ) \ ) \ (\ * \ 2 \ A \ ) \ )$$

In standard notation, this evaluates to

$$\frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

The structure used to store the information for each predator consists of a double precision value to store the fitness and a pointer to the root node of the S-expression tree.

```
typedef struct
{
    Node * sexpr;       /* pointer to root node of equation tree */
    double hitpoints;   /* stores remaining hitpoints */
} Predator;
```

The Node structure used to store the S-expression was defined as:

```
/* BinaryOperator structure */
typedef struct {
    char op;
} BinaryOperator;

/* Terminal structure */
typedef struct {
    enum TermType type;
    union {
        char *Identifier;
        double Constant;
    } x;
} Terminal;

/* Function structure */
typedef struct
{
    double power;
} PowFunction;

struct NodeStruct {
    enum NodeType type;
    int number; /* the node's number. */
    int largestchild;
            /* the node's largest child's number.
             * Only meaningful at the root.
             */
    union {
        BinaryOperator BinOp;
        Terminal Term;
        PowFunction Pow;
    } x;
    struct NodeStruct* parent;      /* Pointer to parent */
    struct NodeStruct* operandL;    /* Left Operand */
    struct NodeStruct* operandR;    /* Right Operand */
};
typedef struct NodeStruct Node;
```

2.2 Prey

The prey population consists of the algebraic equations that our solution equations, the predators, are trying to solve. For this population, the coefficients and possibly some roots are stored. Each set of algebraic equations has a limited domain of coefficients in which solutions can be found. For example, linear equations are of the form $Ax + B$, where A and B are floating point values. Clearly, this has the solution $x = -B/A$. This solution breaks down when $A = 0$. Likewise, quadratic equations, which were a major part of this study, are even more complicated, for not all quadratic equations have solutions that live in $\Re$. So those equations were created by randomly selecting two real-valued roots (α and β), and then calculating B_1 and C_1 coefficients by multiplying out the factors. This gives an equation of the form

$$(x - \alpha)(x - \beta) = x^2 + B_1 x + C_1$$

Then by multiplying the entire equation by some other random number A, an equation with three coefficients and two real roots has been found.

$$Ax^2 + Bx + C$$

These equations are not static throughout the entire run of the system. If during an evaluation a potential solution equation solves within some $\epsilon > 0$ value, that algebraic equation is considered "solved", and a new equation is created to take its place. That way, gradually the set of algebraic equations will get more and more challenging. It is more likely that the simplest prey relative to the predators will be the first to be solved, so by removing them from the population and replacing them by a new algebraic equation, the overall set of equations to be solved will have increased difficulty. This helps drive potential solutions toward better approximations and prevents the solutions from solving a static set. The quadratic equations were stored in the following data structure.

```
typedef struct
{
    double root1;   /* roots of this quadratic */
    double root2;
    double A;       /* corresponding coefficients */
    double B;
    double C;
    int solved;     /* flag show if equation was solved */
} quadraticPrey;
```

3 Fitness

When using a genetic program to solve a problem, one of the most important issues to address is determining the fitness of the individuals. An ideal fitness function not only

guides the system in the correct direction, but also utilizes known information about the problem to speed up convergence while not incurring much computation. Because we wanted to restrict the artificial addition of new information as much as possible, the primary fitness function utilized contains very little information outside that within the system. Though this slowed down convergence, this project focused more on limiting information rather than optimal convergence.

When it comes to determining the quality of a potential solution equation, a simple scoring method is used. At the beginning of each epoch, each solution equation is allotted a certain number of hit-points. Then the equation is evaluated against a certain number of randomly chosen sets of coefficients. During each of these competitions, a metric is taken of how well the equation did against that set of values. If the equation accurately calculated a root, few hit-points are deducted and that set of coefficients is removed from the population. If the equation did poorly, many hit-points will be subtracted away. If at any point during these competitions the equations runs out of hit-points, it is considered dead, and not allowed to reproduce. If an evaluation causes an error, by either trying to divide by zero or taking a negative square-root, the equation is also marked as dead. This method of determining fitness was used because it allowed simple, fast comparison between equations to rank them after each epoch. Also, it didn't require any knowledge of the actual solution formulas to determine a fitness for our potential solution equations.

The success of an evaluation can be determined in two different manners. The potential root can be compared to the actual root for the equation, and this difference can be subtracted from the hit-points that the solution equation has. The other method is to plug the value back into the equation it is using to determine the root, and see how close the output is to zero. Both options have their advantages and disadvantages. In the first case, if the actual roots are known, the algorithm tends to converge quicker. However, not all types of equations have simple root methods, and if an equation is known to determine the root, that often defeats the point of using a genetic algorithm to find a root equation. In this case, comparing to know roots adds information specific to the given set of algebraic equations in the system. This technique for determining fitness was only used as a comparison to the the second method.

To demonstrate the difference between the two techniques, consider the following quadratic.

$$x^2 - 3x + 2 = 0 \tag{1}$$

This equation has solutions at $x = 1$ and $x = 2$. In this example, $A = 1$, $B = -3$, and $C = 2$. Now consider this potential solution equation.

$$4A + \frac{B}{C} \tag{2}$$

Equation (2) estimates the root to Equation (1) to be

$$4(1) + \frac{-3}{2} = 2.5 \tag{3}$$

To calculate fitness with the first method, this potential root is compared to the actual roots for this equation, and the actual error is determined.

$$\begin{aligned} |1 - 2.5| &= |-1.5| = 1.5 \\ |2 - 2.5| &= |-0.5| = 0.5 \end{aligned} \tag{4}$$

Taking the minimum error value, that value would then be subtracted from the hit points of Equation (2).

Using the second method of determining fitness, this value is plugged back into the original quadratic (Equation (1)).

$$(2.5)^2 - 3(2.5) + 2 = 0.75 \tag{5}$$

The absolute value of Equation (5) would then be subtracted from the hit points of the potential solution equation.

By using the second method, much of the information of the problem is removed. All different sets of equations can be solved using the same fitness function. Unfortunately, this fitness function especially struggles with equations that have multiple roots. For example, many quadratics have two roots, so the potential solution equation can map to either root with a given input and score equally well. This leads to contention when one potential equation is converging to the larger roots, and another equation is going toward the smaller roots. If these two potential equations attempt to recombine to create a child, there is high probability that the child won't be very successful. In fact, there were several runs using this method of fitness on quadratics that converged to an average of the positive and negative solutions to the quadratic equation.

4 Recombination and Mutation

4.1 Recombination

Recombination is rather straight-forward considering the structure of the solution equations. Before recombination begins, the surviving equations are sorted based upon their remaining hit-points. Then the equations are separated into two separate groups, highest in one group, and the lowest in another. To create a child, two random parents are chosen, with one parent from the high group and the low group being chosen. Once the two parents are selected, one is randomly chosen to be the primary parent. This parent is completely copied. Then a random node is selected as the crossover point. The other parent becomes the secondary parent. A random node is selected from this second parent, and the subtree from that node of the second parent is copied. Once a copy of this subtree has been made, it replaces the subtree chosen from the first tree. See Figure 2 for an example of crossover.

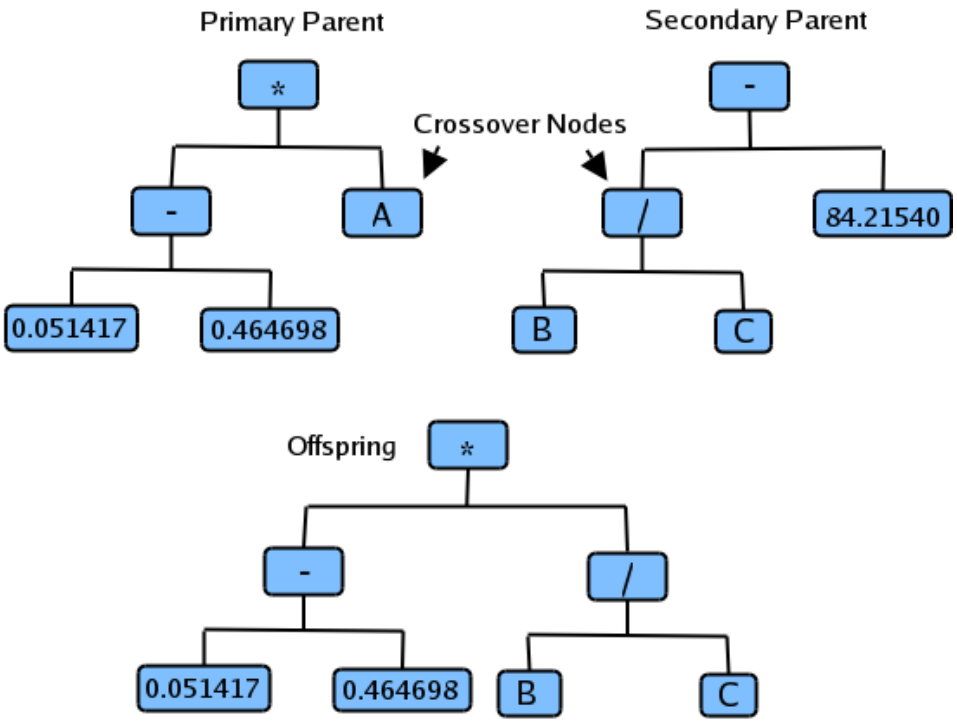

Figure 2: Example of crossover

During the tests, various cutoffs were experimented with to determine the high level and low level parents. If the cutoff level was too small, for example only the top 5% were considered 'high', then the information being passed on after each generation was too restricted and convergence was hindered. Most of the successful runs occurred with 10% or higher. Another issue that was considered was allowing the high level equations to survive the epoch. Runs were taken where the high level equations were destroyed, and other runs were done with them surviving. The success or failure of this variable was closely coupled with the mutation rate. Survival of high level parents worked best with high mutation rate, and destroying all the parents converged best with low mutation rate.

4.2 Mutation

There are three different states a node can be. It is either an operator, a constant, or a variable. If that node is selected to be mutated, each state has its own options. For an operator node, the operator is randomly switched to one of the other operators. For the constant node, the value of the constant can be changed to a new random constant, or the constant can be switched to a variable. Likewise, the variable can be switched to a different variable, or changed to a random constant value.

Mutation rate is a key issue in developing good convergence. Often, only a small percentage of the initial population of equations survives the first few epochs, which limits the amount

of genetic material to choose from. So a high mutation rate is essential in order to find new information to make the potential solution equations more accurate. Traditionally, some of the literature suggests a mutation rate of $1\% - 2\%$. [3] When using a mutation rate of this level, it was vital that the parents didn't survive the epoch. In other words, all the solution equations were new for each epoch. Using this rate, the system was very susceptible local maximums. An initial fitness value that was reasonably high would be reached quite quickly, often within 20-50 epochs. However, once this value was reached, the system would remain near this level, fluctuating slightly above or below, for thousands of epochs. This is a classic example of premature convergence. [2] To help fix this issue, a much higher mutation rate was tried. Rates of up to 20% were used, and although the higher rates helped progress, once the rate got too high, the fitness level fluctuated too violently to be useful. This is especially true when all the parents were killed after each epoch. In that case, the fitness of the system wouldn't converge at all. However, if the top portion of the parents were allowed to survive, the system showed convergence. Best convergence was shown with mutation rate of 10% and allowing the top 10% of the parent population to survive each epoch [2]. This consistently outperformed more traditional mutation rates coupled with no parent survival.

5 Results

In most runs, this system finds solution equations with high fitness levels very quickly, often within 100 epochs. However, these high fitness levels are often misleading, because each equation only competes against a small fraction of the entire search space. So the high fitness values are due to blind luck as much as the actual accuracy of the equation. As the epochs continue, the fitness levels of the top equations may not change much, but they are tested against an increasingly large number of algebraic equations with varying coefficients, the potential solution equations increase in overall accuracy.

Because of the issues with multiple roots, the first method works significantly better in the case of the quadratic. Solution equations were evolved with relative error less than or equal to 10^{-5} over limited ranges of the quadratic variables, such as $\{A, B, C | A, B, C \leq 10\}$. When using the second method, the error bound was several orders of magnitude higher, often between 10^{-1} and 10^{-2}.

5.1 Example Solutions

During a run of 5000 epochs, the Equation (6) below ended up having the highest fitness value. Its relative error was 10^{-5}, and the graph of this function is shown in Figure 3(b), along with the graph of the quadratic equation. Setting $A = 1$ allows plotting in three dimensions.

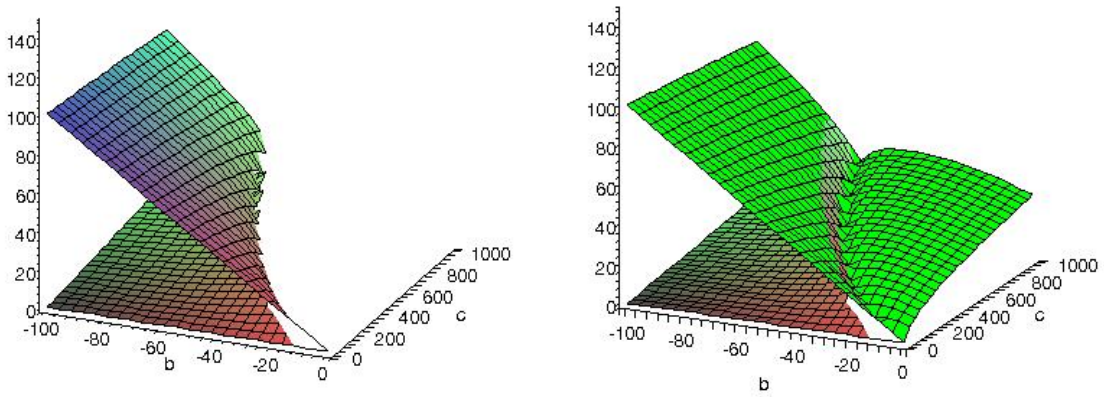

Figure 3: (a) Quadratic Equation mapped in 3 space (a=1). (b) Equation (6) (10^{-5} relative error)

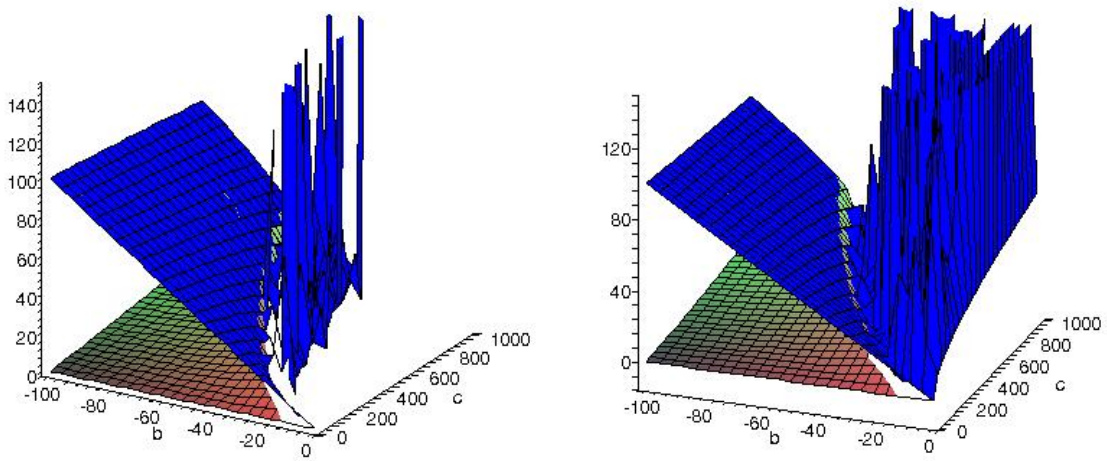

Figure 4: (a) Equation (7) (10^{-3} relative error). (b) Equation (8) (10^{-3} relative error)

$$((c - 0.249947\,(b(38.01378045 + (1.113362\,(19.375409 - c^{-1})^{-1} + (96.62821190 + 0.06372656122\,b)^{2.0})^{0.5})^{-2.0} + b)^{2.0})^{2.0})^{0.25} - 0.5000441566\,b \quad (6)$$

Some of the solutions evolved into very large equations, such as the following:

$$\begin{aligned}&((c((c((c((c((c(1.025417525\,b^2 + \\ &c(\tfrac{c}{(b - 348.4329700\,(1425.455203 + (c + 93.29497699)^{0.5})^{-2.0})^{2.0} + 1 - 2\,c} - 1)^{-1} - c)^{-0.5} + b)^{2.0} + \\ &1)^{-0.5} + b)^{2.0} + 0.003249486066\,b^{2.0})^{-0.5} + b)^{2.0} + \\ &0.002297505575\,b)^{-0.5} + b)^{2.0} + (c((c((c(0.02541752543\,b(-604.1633377 + b) + \\ &((\tfrac{c}{(b - 1339.198329\,(3.911086 + c)^{-0.5}(1425.455203 + (c + 60.23093608)^{0.5})^{-2.0})^{2.0} + (1 + c)^{0.5} - c})^{2.0} + \\ &c)^{2.0} - c)^{-0.5} + b)^{2.0} + 1)^{-0.5} + b)^{2.0} + \\ &0.05576893058\,\tfrac{(-0.3806695978 + b)^{2.0}(2.703704 - 21.649633\,b)^{-0.5}}{c})^{-0.5} + 13.729620 + c)^{-1})^{0.5}\end{aligned} \quad (7)$$

This equation, where $A = 1$, plots as Figure 4.

$$\begin{aligned}&((b + c((b + c((b + c((b + c((b + c((b + c((b + c((b + c((b + 1.414213562\,c^{0.5})^{2.0} + \\ &(b + ((c(17.149483 + c)^{0.5} + 1)^{0.5} + c)^{0.5})^{2.0})^{-0.5})^{2.0} + c^{-1})^{-0.5})^{2.0} + \\ &1.249643667)^{-0.5})^{2.0} + 0.01589486515)^{-0.5})^{2.0} + c^{-1})^{-0.5})^{2.0} + c^{-1})^{-0.5})^{2.0} + \\ &(2 + (1 + c)^{0.5})^{0.5})^{-0.5})^{2.0} + (21.131135 + c)^{-1})^{-0.5})^{2.0})^{0.5}\end{aligned} \quad (8)$$

This equation plots as Figure 4(b).

5.2 Quadratic Formula

With the previous two equations demonstrating the results of a typical run, the question that needs to be addressed is, "Where is the quadratic formula?" There are two major issues that hinder the appearance of the quadratic formula. First, the quadratic formula contains a square root. Any invalid operation, such as dividing by zero or taking the square root of a negative number, set the fitness of the equation to zero, so that equation will have no opportunity to reproduce. Because of this, any equations that may have the square root with a subtraction inside, like the quadratic equation requires, are prime candidates to be eliminated by an invalid operation.

The second problem involves the various pieces that make up the quadratic formula. By itself, $\frac{-b}{2}$, and to a lesser extent $\sqrt{b^2 - 4ac}$, does not provide a very close approximation to the quadratic formula. Tests were run where the initial random population was augmented with several copies of the pieces that comprise the quadratic equation. With these artificial initial conditions, results were mixed. Some runs successfully found the quadratic equation within 20 epochs. Other runs went for hundreds and thousands of epochs, with no sight of the quadratic. An analysis of the population after less than 100 epochs of an unsuccessful run revealed minimal data remaining from the pieces inserted into the population.

As was mentioned before, it seems that the during the first epochs, it is better to be lucky than good. The success or failure of an equation relies heavily on the set of coefficients that it must compete against. Because of the way reproduction is set up in this system, the data from the initially successful equations is disseminated throughout the population rather quickly, which forces out much of the other genetic data. And with the dominant factor being crossover rather than mutation in reproduction, it is difficult for the system to recover that information.

5.3 Continued Fractions

Even though this system didn't converge to the solution that we were hoping, it did converge to a solution. Consider the following example:

$$\begin{aligned} Ax^2 + Bx + C &= 0 \\ Ax^2 &= -Bx - C \end{aligned} \tag{9}$$

Divide through by Ax.

$$x = \frac{-B}{A} - \frac{C}{Ax} \tag{10}$$

Let $A = 1$.

$$x = -B - \frac{C}{x} \tag{11}$$

Substitute the right side for x on the right side.

$$x = -B - \frac{C}{-B - \frac{C}{x}} \tag{12}$$

Continue these substitutions.

$$x = -B - \cfrac{C}{-B - \cfrac{C}{-B - \cfrac{C}{-B - \cdots}}} \tag{13}$$

This is an example of the solution x expressed as a continued fraction. This method is often used to represent irrational numbers in terms of integers. In this case, if the sequence converges, the solution approximates a root based on the coefficients. Because all the equations in this system are guaranteed roots, this equation does represent a root approximation based on the coefficients of the equation. Runs with a variety of parameter settings converged to solutions of this form. For example, Equation (8) has several sections that are raised to negative powers. If these were to be written as fractions, it would be very similar to (13). This is also the case with Equation (7) as well. Many of the solutions this system converged toward mimicked the form of the continued fraction representation of the solution. In some ways, this is an even better solution to the actual quadratic formula, because solutions of this form can be expressed with only the four major operations of addition, subtraction, multiplication, and division. There is no need to complicate things with square roots and squaring operations.

6 Conclusion

GP is a very versatile tool for problem solving and automated design. We have been able to generate a solution to a general quadratic equation using GP. We have also succeeded in having the system discover the solution rather than have an algorithm for discovery provided. It generated forms which were unexpected and very interesting.

Future directions for this project include casting a wider net to attack higher order equations or other nonlinearities. These include equations involving exponentials and logarithms. One would expect that having a GP attack these kinds of equations would lead to solutions involving Taylor expansions, but as this test showed, we may be surprised. Another possible addition is converting to complex arithmetic. This may enhance convergence, relax domain restrictions, and open the possibility of closed form solutions.

We are also interested in changing the dynamics of the predator/prey model. Right now, we are only evolving the predator population, but the prey could be set to evolve as well. Also, both the predator and prey population sizes are fixed. Allowing for dynamic population sizes may cause some interesting behavior between the two populations. Another possibility is to parallelize the code and create an island model. This would create relatively isolated populations on each of the nodes, allowing members of each population to occasionally jump to a new island. Adding this separation may help prevent the premature convergence that that we fought with in our system.

Work on this project was greatly aided by Erin Nichols, who graciously allowed us to use her S-expression parsing code, binary tree manipulations functions, and other code and information from previous projects. Also, we found Melanie Mitchell's book invaluable as an introduction on Genetic Algorithms [3].

References

[1] David E. Goldberg, *Genetic Algorithms in Search, Optimization, & Machine Learning*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[2] John Koza, *Genetic Programming II*, MIT Press, Cambridge, MA, 1994.

[3] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1999.