Investigating the Application of Genetic Programming to Function Approximation

Jeremy E. Emch

Computer Science Dept.
Penn State University
University Park, PA 16802

Abstract

When analyzing a data set it is often useful and necessary to determine, or compute, a predictably reliable function approximation to aid in future and / or prior behavioral analysis. This work investigates the application of Genetic Programming (GP) to the problem domain of function approximation. We start by introducing GP and its basic operations and procedures, followed by a discussion of results obtained from developing an original application and various test data sets. Solutions to function approximation problems have previously been obtained by using classical techniques that require knowledge of the size and shape of the solution (regression, splines, etc.). Preliminary results support the notion that GP can be used to effectively solve function approximation problems where the size and shape of the solution space is unknown; a distinct advantage over classical techniques.

1 INTRODUCTION

As a branch of Genetic Algorithms (GA's) and Evolutionary Strategies (ES's), Genetic Programming (GP) has been applied successfully in a wide range of sciences and engineering areas. Unlike GA's or ES's, GP seeks to produce an executable program to solve the problem as the solution, rather than a solution of the problem. GA's are generally used as an optimization technique to search for the global optima of a function. While GA's typically operate on fixed-length coded strings, they are not suitable for problems where the size and shape of the solution are unknown (Koza 1989). Similar to GA's, GP works by emulating natural evolution, "survival of the fittest", to generate a model structure that maximizes (or minimizes) the objective function involving an appropriate measure of the level of agreement between the model and system response (Koza 1992). Drawing basic theories from GA's and ES's, GP seeks to produce executable source code to solve the problem described by the state of the application.

2 GENETIC PROGRAMMING

Genetic Programming is a technique that allows computers to solve problems without being explicitly programmed. Fundamentally, Genetic Programming applies Genetic Algorithms to a population of programs. As such, there are five primary steps in a simple GP algorithm:

1. Generate an initial population;
   This is a random process which depends on the specific type of representation scheme being used.
2. Calculate a score for each individual based on some fitness criteria;
   This is analogous to an individual's lifetime.
3. Create next generation;
   This step involves selection, mutation, and recombination.
4. Repeat steps 2 and 3 until some stopping criteria has been met;
   i.e.: Convergence, time, tolerance, generation limit, etc.
5. Repeat steps 1 through 4 for N runs.
   GA's are rarely run once.

2.1 REPRESENTATION OF STRUCTURES

In GA's, solutions are represented by binary coded strings or real numbers. However, solutions in GP are computer programs consisting of terminals and functions, selected from some set of enumerated functions and terminals.

Essentially, GP is programming a computer to program itself. Initially, a simple approach of breeding machine code and FORTRAN code, ignoring syntax was used. However, as noted in (De Jong 1987), this approach fails

because programs are not likely to compile and run, much less produce correct results. When GP was applied using the LISP language some success was reported. Later it would be realized that the natural structure of LISP objects (that is, everything is represented as a pre-order list) emulates a compiler parse tree. Additionally, LISP is not a compiled language; the development environment contains a run-time emulator that requires no compilation to execute code. Therefore, run-time execution of genetically evolved source code was natural.

Both linear and tree structures have been used. However, the normal terminology of genetic programming suggests that the data structures being evolved be able to be executed. As such, generally in GP, a computer program is depicted as a parse tree: Functions are denoted as roots or inner nodes of parse trees, and terminals are denoted as leaves of parse trees (see Figure 1). Additionally, other abstract data types could be used; such as Grids and neural networks.

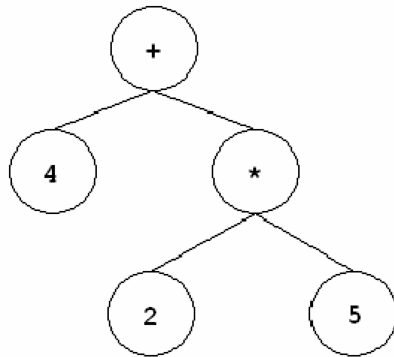

Figure 1: Simple Tree Structure

2.2 FUNCTIONS AND TERMINALS

A function set could include basic arithmetic operations, conditional operators or user-defined operators. A terminal set includes arguments for functions. (Banzhaf 1998) gives formal definitions for these nodes:

- Def. 1.1 "*The* ***terminal set*** *is comprised of the inputs to the GP program, the constants supplied to the GP program, and the zero-argument functions with side-effects executed by the GP program*"

- Def. 1.2 "*The* ***function set*** *is composed of the statements, operators, and functions available to the GP system*"

An illuminating and intimidating property of genetic programming should now be crystallizing; that is, in GP, the solution space is the enormous space comprised of all possible, syntactically correct, executable programs derived from the enumerated function and terminal sets. Thus, the decision for what functions and terminals will be available in your GP application must be taken with great care.

2.3 POPULATION GENERATION

For GP, the initial population generation is highly dependent upon the representation structure. As such, we will discuss this task by considering it in the context of structure.

Tree structures feature three popular techniques for initial population generation:

- Grow;
- Full; and
- Ramped Half-and-Half.

In the Grow technique, nodes are chosen randomly from the function and terminal sets. Once a terminal has been chosen for a branch, that branch has ended; even if the maximum depth has not been reached. Therefore, this technique produces trees of, possibly highly, irregular shape. This fact may have consequences in terms of memory and search speed later on. The Full technique, on the other hand, produces perfect trees. That is, trees which are "fully resident" and perfectly balanced. This method proceeds by only choosing functions until the maximum depth is reached, whereby it then chooses terminals. This technique relieves much in terms of search because the balanced nature of the structure will guarantee maximum efficiency. However, your memory woes may be just beginning. Recognizing that diversity is valuable, the Ramped Half-and-Half method proceeds as follows:

- First, a maximum depth K is specified;
- Then, the population is divided equally among individuals to be initialized with trees having depths 2 to K.
- For each depth group, half of the population is initialized with the Grow method, and half of the population is initialized with the Full method.

For linear structures the procedure is surprisingly simpler:

- Randomly choose length between 2 and the maximum length;
- Copy predefined headers;

- Initialize and add instructions until the chosen length is reached;
- Copy predefined footers;
- Copy predefined return instructions.

See Figure 2 for examples of these structures. It should be noted, however, that these techniques are not exhaustive and that other techniques may be applied.

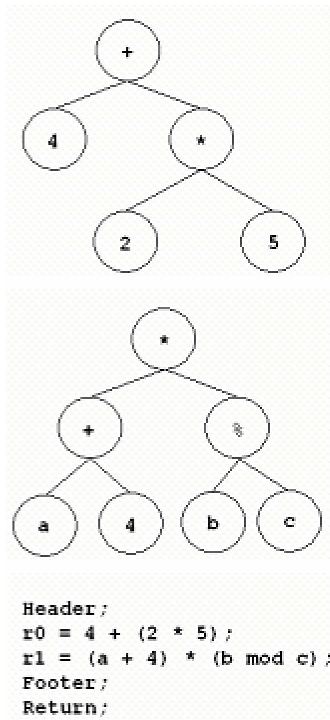

Figure 2: (from top to bottom) Grown Structure; Full Structure; Linear Structure.

2.4 ASSESSING FITNESS

The fitness function is that part of the GP that determines how well an individual performs related to the population given the same input. There are three primary types of fitness functions:

- Continuous: any function which exhibits a correspondence between improvements in how well the individual assimilates the training data and improvements in fitness.
- Standardized: zero is the value assigned to the fittest individual
- Normalized: when fitness is always between one and zero

The actual implementation of a fitness function, however, may be fuzzy and is application specific. For example, you may want the number of matching pixels in an image matching application, or deviation in fitting, or an error fitness: $\sum|p_i - o_i|$, or a squared error fitness: $\sum(p_i - o_i)^2$ (Banzhaf 1998).

2.5 OPERATIONS

In GP, the basic genetic operations of GA's and ES's are available: selection, mutation, and recombination. We will now explore each of these in turn.

2.5.1 Selection

There are two scenarios for selection: a GA scenario, where individuals are selected for variation after evaluation; and an ES scenario, where individuals are varied then selected. The underlying algorithms, however, from GA's and ES's remain. Primarily, (μ, λ), $(\mu + \lambda)$, Rank based, Tournament based, Fitness-Proportional, etc.

2.5.2 Mutation

Mutation proceeds based on structure. Tree structures are mutated by randomly selecting a node, other than the root, and replacing the subtree with a new randomly generated subtree. Linear structures, however, proceed by randomly choosing an instruction, then randomly choosing the type of change. This change is usually an instruction replacement, constant replacement, or register (location) change. Although (Koza 1992) suggested that in general, mutation only plays a minor role in GP and can be omitted in most cases; other researchers (Luke and Spector 1997) found that mutation is useful and crossover does not have considerable advantage over mutation in the case of small populations and large generations.

2.5.3 Recombination / Mating

Recombination is another operator that proceeds based on structure. Tree structures first choose the parents by some predefined mating policy. Then randomly selected subtrees in each parent are swapped with one another (see figures 3 and 4). For linear structures, the parents are selected in the same manner. Then, however, randomly selected linear segments of code are swapped between parents. It should be noted here, in the linear case, that this procedure leads to a breakdown of our GP assumption; that is, all generated structures need be executable. This could possibly be avoided by extending

the GP engine towards compilation, but this is an unreasonable task that would require and add massive increases in complexity.

2.6 CONTROL AND TERMINATION

Control over, and termination of, the GP application is basically the same as in other GA's or ES's. Control is exerted through the operators and variables available, and termination is forced through conditional operators (i.e.: maximum number of generations, clock time, tolerance, etc.). However, one must be aware of their computing environment. Functions and terminals must be chosen carefully to minimize the search space. GP structures are unfathomably powerful, yet are harder to search through and more costly in memory. Not only do they consume more space, but also consider search time. Blocks of memory are not necessarily contiguous, and may need to be loaded from secondary memory to primary memory to be operated on. This adds considerable computation time.

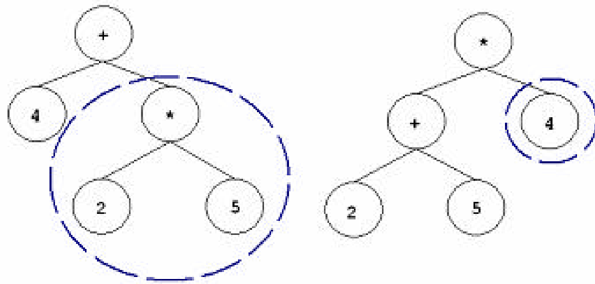

Figure 3: Two Parents Selected For Mating; Chosen Subtrees Are Highlighted.

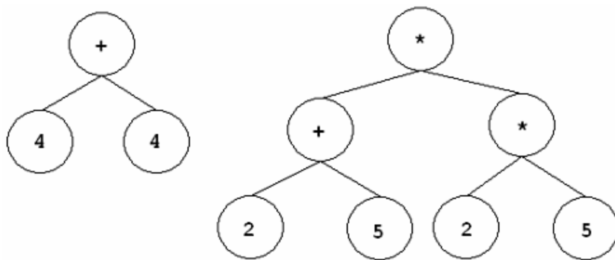

Figure 4: Newly Created Individuals Resulting From Recombination of Figure 3.

3 FUNCTION APPROXIMATION

When analyzing a data set it is often useful and necessary to determine, or compute, a predictably reliable function approximation to aid in future and / or prior behavioral analysis. Previously, we have used techniques such as regression, divided differences, polynomial, Lagrange or spline interpolation. These techniques have been proven to be accurate within some error defined by the chosen function representation and whether the function is known at many or few points. Unfortunately, some notion of the size and shape of the solution is necessary for determining which technique to employ for that particular problem; as the problem may not be suitable, or possibly completely intractable for the method. It is the purpose of this paper to investigate the application of Genetic Programming (GP) to the problem domain of function approximation. As such, a GP application was developed using a highly object-oriented approach utilizing all the features of GP discussed thus far (i.e.: selection, mutation, recombination, etc).

3.1 APPLICATION PARAMETERS

The application employs a steady-state methodology, following the ES scenario previously described. Ramped Half-and-Half initialization is used for the following examples, though no performance metrics were taken with respect to the available initialization procedures. Tournament selection is the method of choice; where four members of the population are randomly selected to take place in the tournament. The individual with the best fitness is retained for recombination (with another randomly selected individual from the base population) and mutation. After the genetic operators are employed on the selected individuals another fitness evaluation is performed on the offspring. If the fitness of the offspring is better than the two losers of the tournament (the two individuals with lowest fitness), the offspring replace them in the base population. The probability of mutation was 0.05, and the probability of recombination was 0.75. No analysis of these probabilities was performed due to time constraints; however, these values are generally regarded as acceptable within the GP community. It is completely feasible that better results may be obtained with further analysis of these parameters. The termination criteria was a combination of maximum function evaluations (0.5 million) and tolerance (0.01). Finally, the function set was comprised of all the mathematical functions available in the standard C header <math.h>, and the terminal set was comprised of the variable 'x' and randomly generated constants.

3.2 EXAMPLE 1

Table 1: Training Set For Example 1.

X	0	1	2	3	5
Y	0	1	4	9	25

Table 1 shows the training data input to the GP application. The GP was executed for 100 unique seeds with a population size of 40 individuals. The application successfully evolved an optimal solution (that is, fitness = 0, or the evolved function passed through all points in the training data set) in all 100 cases, with an average of 10,072 function evaluations and an average time of 0.3 seconds (CPU time).

This example was included to illustrate an important drawback of GP; that is, there is no way to guarantee the simplicity of the evolved solution. For example, in one run the GP evolved the following equation: (((x * x) - ((x - x) + (atan((10369 * x)) ^ (x - x)))) + (((10369 * x) + x) ^ (x - x))) (see figure 5), which simplifies to the optimal function x^2.

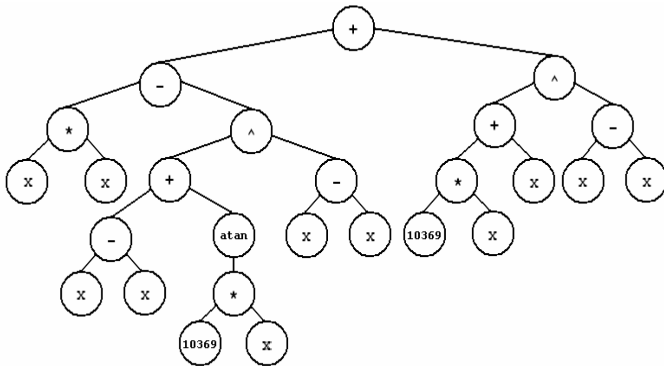

Figure 5: A Complicated Structure Which Reduces To The Much Simpler Function x^2.

3.3 EXAMPLE 2

Table 2: Training Set For Example 2

X	0.1	0.4	0.5	0.8
Y	2.08153	1.87932	1.90439	1.7758
X	1	1.1	1.4	1.7
Y	1.33804	0.7934	0.27794	-0.0297
X	1.8	2	2.1	2.6
Y	0.65020	-0.4715	-0.4321	0

Table 2 shows the training data input to the GP application. The GP was executed for 100 unique seeds with a population size of 40 individuals. The application successfully evolved an optimal solution (that is, fitness = 0, or the evolved function passed through all points in the training data set) in only 8% of the cases, with an average of 125,556 function evaluations and an average time of 25.699 seconds (CPU time). However, 98% of seeds found an approximate solution within a 0.01 error tolerance.

After examining the solution set of the 100 seeds it was evident that the best solutions of a majority of the test cases included only base trigonometric functions. Subsequently, the function set was adjusted to include only the basic trigonometric functions that appeared in the solution set (sin, cos, tan), and the GP application was again run for 100 random seeds. The application then obtained an optimal solution for all 100 cases, with an average run-time of 10.2 seconds (CPU time). This analysis provided significant insight into the GP application. Principally, that the GP is very sensitive to the function and terminal sets available. This suggests that an adaptive procedure for reducing the complexity (size) of said sets during the runs of the GP application would significantly enhance the application and its results. Figure 6 shows a plot of the optimal solution, and figure 7 shows the simplest equation evolved for this example. The figures correspond to the function:

$$\frac{4 \sin\left(x + x^{x^4 - x^2 + 3x}\right)}{e^x}.$$

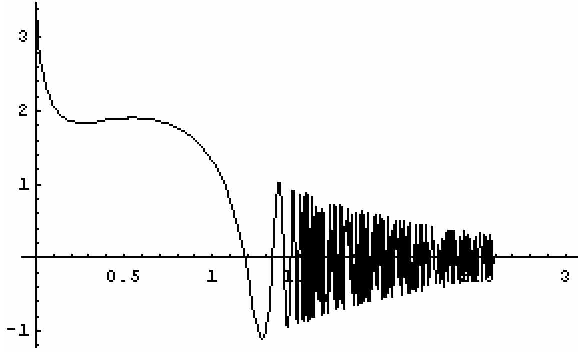

Figure 6: Optimal Solution For Example 2.

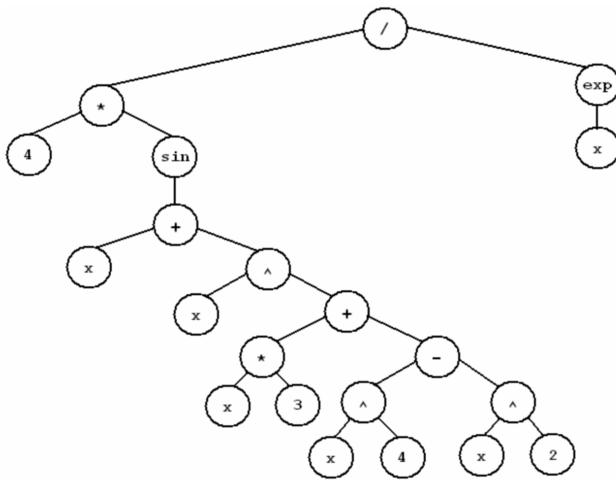

Figure 7: Simplest Optimal Solution Evolved For Example 2.

3.4 EXAMPLE 3

As a final example, an error versus generation plot (figure 8) has been included with multiple functions, varying in degree and difficulty. This plot illustrates the applicability of GP to function approximation, as well as the underlying principle of GP – evolution. The plot shows the evolution of the solution over time, with eras of little solution progression and generations that significantly improve the solution quality. This data was obtained without altering the application parameters previously discussed, thus showing the generality allowed by the GP implementation.

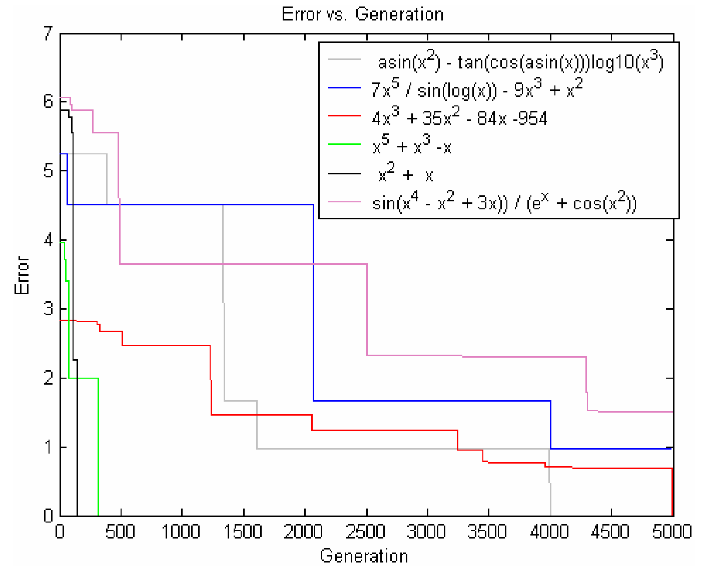

Figure 8: Error vs. Generation for a small suite of functions.

4 CONCLUSIONS

The results of this investigation confirm that Genetic Programming is indeed a powerful tool; more so than GA's or ES's in that GP seeks to evolve an executable program as the solution of the problem, rather than simply a solution to the problem. Additionally, unlike GA's and ES's, the GP operation of recombination can produce different offspring from identical parents. In fact, when depth is greater than two with at least one full parent, the probability that random recombination will produce new offspring is greater than not. Unfortunately, GP is also considerably more complex than GA's or ES's. The test suite applied to the GP application shows that GP can be used as a tool for function approximation and that it is especially effective when the size and shape of the solution space is unknown. However, the test suite also showed that GP has considerable difficulty finding a simple program that also produces high precision. The complexity of the best solution is also affected by the pre-selected function and terminal sets. The complexity (size) of the function and terminal sets are directly related to the generality and search speed of the GP application. When increasing said sets to improve generality, and thus

the search space of the application, complexity (in time and space) is also increased. Additionally, it must be noted that the GP application is tool that is subject to significant constraints that must be accounted for when used. Among them is the fact that the output is only as good as its training set. If used in predictive analysis one should expect that the more training data employed, the more accurate the predictions. Although, increasing the training data set size also adds computation time.

The examples contained within this paper were initialized with a tree depth of 4, and were allowed to evolve unconstrained. The premise was that code bloat would be easily distinguished among the runs and analysis could be performed. However, this did not occur. The unconfirmed reason for this is probably due to the nature of the recombination procedure. Recombination of tree data structures can be viewed as not only additive, but also as destructive (contributing to the confinement of code bloat). Also, since the application was written in C++ (a compiled language) it contains a function parser as part of the fitness evaluation module. Recalling the aforementioned issues related to memory, processing time, and tree structures; the GP application seems particularly well suited for parallelization. Specifically, a hierarchical approach, combining the master-slave and multi-population approaches, would increase the search space and diversity of the application.

5 FUTURE WORK

A simplification procedure (to effectively collapse the tree representation) would be an interesting exercise extending this work. This would not only be beneficial in reducing complexity but also as a way of facilitating diversity within the population. Also, an adaptive procedure to reduce the complexity of the function and terminal sets at run-time would enhance the performance and accuracy of the application across unknown function training sets. Additionally, a procedure that would offer the user several optimal functions, with varying degrees of complexity, could be useful in the analysis of large training sets. Also, further analysis of the application needs to be performed. This would include behavioral analysis of the application with respect to code bloat, diversity loss, and initialization requirements. Also, the performance of the application with respect to tree depth needs to be assessed.

References

Box, G. E. and Jenkins, G. M., (1970). *Time Series Analysis: Forecasting and Control*, Holden-Day, San Francisco, CA.

Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D. (1998). *Genetic Programming: An Introduction*. Morgan Kaufmann Publ., San Francisco, CA.

De Jong, K, (1987). On Using Genetic Algorithms to Search Program Spaces. In *Proceedings of the Second International Conference on Genetic Algorithms and Their Application*, 210-216, Lawrence Erlbaum Associates, Inc. Mahwah, NJ.

Cheney, W. and Kincaid, David, (2002). *Numerical Analysis: Mathematics of Scientific Computing*, Brooks/Cole, Pacific Grove, CA.

Koza, John R. (1989). Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence.* San Mateo: Morgan Kaufman.

Koza, J. R., (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge MA, 162-169.

Luke, S. and Spector, L., (1997). A Comparison of Crossover and Mutation in Genetic Programming. *In Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97)*. J. Koza et al, eds. San Francisco: Morgan Kaufmann, 240–248.