

Genetic Programming Reconsidered

Russ Abbott, Behzad Parviz, and Chengyu Sun

Department of Computer Science
California State University, Los Angeles
Los Angeles, California
{RAbbott, BParviz, CSun}@CalStateLA.edu

Abstract. *Even though the Genetic Programming (GP) mechanism is capable of evolving any computable function, the means through which it does so is inherently flawed: the user must provide the GP engine with an evolutionary pathway toward a solution. Hence Genetic Programming is problematic as a mechanism for generating creative solutions to specific problems.*

Keywords: *genetic programming, evolutionary pathway, fitness function, teleological evolution, adaptive evolution.*

I. INTRODUCTION

According to Koza [1],

Genetic programming starts from a high-level statement of what needs to be done and automatically creates a computer program to solve the problem.

This is a bit of hyperbole from a couple of perspectives.

1. In most genetic programming applications, no high-level problem statement is ever provided, at best, the problem may be expressed informally in English; the problem to be solved is characterized to the genetic programming (GP) system in terms of a fitness function, which itself is expressed as a relatively low-level computer program.
2. More significantly, the range of computer programs generated by GP systems has been disappointingly limited. Kirshenbaum [3] has written,

One of the main limitations of traditional genetic programming ... is that the solutions that are discoverable are limited to the class of algorithms known as *constant time* or $O(1)$ algorithms, those that make a single pass through the operators, with no loops or recursion.

The remainder of this paper explores this and related questions by examining the brief history of attempts to generate a sort program. We then describe how a genetic programming system may be used to generate *any* computable function. Although surprisingly strong, this result is disappointing in that the mechanism through which the generation occurs casts doubt upon the independent utility of the genetic programming evolutionary engine: the system must be supplied with an evolutionary path that leads to a problem solution.

We then show that any fitness function that provides *warmer/colder* information to a GP evolutionary engine is of necessity a cheat in the same way.

II. GENERATING A SORT PROGRAM

Sort is of complexity $O(n \times \log(n))$. Naïve sort programs are $O(n^2)$. Sort is a good test case for genetic programming because like most real-life computer programs, sort (especially naïve sort) combines some (but not too much) algorithmic sophistication with simple arithmetic and data structure operations.

A literature search revealed very few publicly reported attempts to generate sort programs. The most prominent are quite old.

A. O'Reilly and Oppacher

O'Reilly and Oppacher [4] report a failed attempt to generate a sort program. They attempt to generate a program that transforms an unsorted input array into a sorted array by moving elements around in the array.

The program is provided with (integer) variables, which may be used as indices into the array.

The allowed operations are: decrement a variable, read an array location indexed by a variable, and swap adjacent array locations.

The allowed control structures (expressed in C notation) are:

```
if (Array[<i>] < Array[<j>]) <body>;  
for (int i = <low>; i < <high>; i++) <body>;  
for (int i = <high>-1; i >= <low >; i--) <body>;
```

O'Reilly and Oppacher report that using what today would be considered relatively small populations, they were unable to generate a correct sort program. Recently O'Reilly [5] wrote that in her estimate a larger population size would have been unlikely to have changed the outcome.

It is worth noting that the constrained **for**-loop used by O'Reilly is quite similar to the Automatically Defined Iteration construct described by Koza [1]. The other operators and data structures are also quite similar to those reported in that book. Unfortunately Koza does not report whether attempts were made to generate a sort program using his version of those constructs.

B. Kinnear

Kinnear [6] also reports on experiments to generate a sort program. Like O'Reilly and Oppacher, Kinnear attempted to generate a program that, given an unsorted array terminates with the array sorted. Kinnear experimented with a number of function sets. Overall, the functions were similar to those used by O'Reilly and Oppacher. When Kinnear replaced the swap operation with what might be called an *intelligent swap*, one that swapped adjacent elements but only if they were out of order, it was relatively easy to generate a sort. Two nested loops with the intelligent swap as the embedded body produces a bubble sort.

C. Fitness functions

The real key to Kinnear's success, though, was his fitness function, which counted the number of swaps needed to convert the output of a program into a correctly sorted result. (The fewer the better.) In contrast, O'Reilly and Oppacher's fitness function counted the number of out-of-place elements. (Again, the fewer the better.) This difference appears to be the key to the difference in outcome.

Consider the following program fragment:

```
for (int i = <low>; i < <high>; i++)  
    intelligentSwap(i, i + 1);
```

This fragment, the inner loop of bubble sort, makes one pass through the array, swapping adjacent out-of-order elements. Such a fragment is trivially generated at random in Kinnear's frame-

work—and with only a bit more work in O'Reilly and Oppacher's.

Since such a fragment would have reduced the number of swaps required to put the array in order, Kinnear's system would have valued it significantly more highly than a random program fragment. Since for most inputs such a fragment would not put elements into their correct positions, O'Reilly and Oppacher's system would not have valued it much more highly than a random program fragment.

Moral: a well-designed fitness function can guide a GP system *along an evolutionary pathway* to a problem solution that it might not otherwise find.

We emphasize *evolutionary pathway* because that is what GP claims to find. Clearly if one runs a random program generator long enough, any program will eventually appear. The question for GP is: what are the evolutionary pathways whose discoveries it facilitates?

III. ANY COMPUTABLE FUNCTION CAN BE GENERATED BY A GP SYSTEM

It is important to acknowledge that the genetic programming framework is theoretically capable of generating all computable functions. On an intuitive level, this may seem surprising since traditionally, most of the functions that have been generated within a GP framework have been finite state machines. The amount of memory available for computation is generally fixed in advance. GP generated programs are typically limited to a fixed number of fixed size variables, and they have no means to write to an external store.

But as Teller showed early on [9], the genetic programming framework need not be so constrained. The GP framework can be generalized so that unlimited memory is available, allowing such a system to generate any computable function.

Even so, the heading for this section doesn't really say much. After all, if a function is computable, a system (like GP), if allowed to generate random computer programs and if run long enough will, like the proverbial monkey at a keyboard, eventually generate it.

Here we wish to make a stronger claim. A GP system can generate any computable function in a manner consistent with the sort of evolutionary

efficiency typically associated with genetic programming.

The following argument shows that a GP system can generate any computable function by following an evolutionarily efficient pathway.

Let F be some computable function. Since F is computable, there must be a program that computes it; hence there must be a parse tree for such a program. Let P be a parse tree for a program that computes F .

Let the fitness function Fit rank GP-generated parse trees, i.e., candidate population elements, on the basis of how closely they resemble P . (See, for example, Shasha [10] for a discussion of tree matching.)

Clearly, a GP engine when running the fitness function Fit will generate the parse tree P , which, when executed, will compute F .

This, of course, is a disappointing result. For one thing, if one already has a parse tree for program that computes a desired function, there is no point in using a GP system to generate that program.

Secondly, even if one wants to use GP to generate a function that one already knows—perhaps just to show that GP is capable of generating such a function—using a fitness function that compares the GP generated candidates to the desired *parse tree* seems like cheating. In traditional GP, one typically compares the *output* produced by executing candidate parse trees to the *output* produced by the desired parse tree when it is executed.

But if a direct comparison of parse trees is cheating, what of the sorts of implicit comparisons we have traditionally allowed? Consider the fitness functions in the sort generations discussed above.

The Kinnear fitness function (the number of swaps needed to put the elements in order, lower is better) was more helpful, and more importantly a better guide to the eventual parse tree, than the O'Reilly and Oppacher fitness function (the number of elements in their incorrect places, again, lower is better). Was that cheating? It doesn't seem like cheating until one realizes that Kinnear's fitness function, unlike O'Reilly and Oppacher's, provided Kinnear's GP system with information about an evolutionary pathway that leads to a sort, namely that the inner loop of bubble sort is a good building block.

Intentionally or not, Kinnear's fitness function helped his GP system find its way to a solution. Is there a GP experimenter alive who has not constructed a fitness function that was intended to help his or her GP system find an answer?

IV. IS IT POSSIBLE NOT TO CHEAT?

Consider the fitness function in a traditional run of a GP system. As long as the fitness function is not binary (correct vs. not correct), it will rank the generated candidate functions on a scale that indicates how close the function is to being correct. Let's call any such non-binary fitness function a *warmer/colder* fitness function.

Now consider a second function that provides a similarity ranking between the parse tree of candidate functions and the parse tree of a program that computes the target function correctly—and assume for the time being that there is only one such program; we deal with the more general case in the next section.

Let's ask how well these two functions correlate with each other. Clearly, if there is a perfect correlation, the GP system will quickly find a parse tree that generates the target function. In effect, we are in the same situation as we were when generating the parse tree for the arbitrary computable function in the previous section by matching parse trees.

Clearly also, any fitness function that correlates positively with a parse tree similarity function defines, perhaps unintentionally, a similarity measure for parse trees. In doing so, it thereby provides the GP system with a clearly defined evolutionary path to the correct result.

On the other hand, imagine that there is no positive correlation between a functionally defined (i.e., results-oriented) fitness function and a parse tree similarity function. Could any GP system generate a correct result if the fitness function it is using does not help it retain in its population increasingly good approximations of a parse tree for a correct function? Clearly not—except by generating parse trees at random. Since it is a particular parse tree that we want to generate, a fitness function that is not correlated to a similarity function for that parse tree will provide no useful guidance to a GP engine.

The key point is the following. Although the implied intent of the fitness function in GP is to measure how close the output of a candidate function is to the output of a target function, the GP engine, *simply by virtue of how it is defined,*

actually uses the fitness function as a measure of how close a candidate parse tree is to an intended parse tree: the better the fitness, the higher the parse tree is ranked. *Thus the degree to which a GP fitness function correlates with a parse tree similarity function will determine the success of a GP run.*

A direct correlation of this observation is that any time one provides a warmer/colder-style fitness function, one is either (a) implicitly providing an evolutionary path for the GP system, if the fitness function correlates positively with a parse tree similarity function, or (b) misleading the GP system, if the fitness function does not correlate positively with a parse tree similarity function.

Population and algorithmic parameters (such as population size, mutation rate, etc.) that constrain or encourage diversity cannot alter that fact.

In an earlier paper [8], we suggested an approach to genetic programming called *Guided Genetic Programming* in which the user suggests intermediate goals for the GP system. It is now clear that intended or not, every GP run that includes a positively correlated warmer/colder fitness function is a guided GP run. Since it is impossible for such a fitness function not to provide guidance toward an evolutionary path, one might as well explicitly design fitness functions that are useful guides.

The previous discussion was expressed in terms of problems for which there were unique solutions. The same argument holds for problems that can be solved in multiple ways, i.e., problems for which there are multiple parse trees that produce correct results.

V. CONCLUDING THOUGHTS: FUNCTION MUST MIRROR FORM

In his most recent book, Koza [11] describes results produced through the use of genetic programming. Even though the results are not computer programs as traditionally understood, it seems clear that every evolutionary engine is subject to the evolutionary pathway constraint: to be evolutionarily effective, a fitness function that measures the *functionality* of a result must correlate positively with a similarity measure comparing the generated *form* with a fully effective *form*. Otherwise the fitness function will not be a useful guide for traversing an evolutionary pathway.

ACKNOWLEDGMENT

The authors thank Debora Shuger for insightful discussions.

REFERENCES

- [1] Koza, J. R., F. H Bennett III, D. Andre, and M. A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann, 1999.
- [2] Langdon, W B., *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Kluwer Academic Publishers, 1998.
- [3] Kirshenbaum, E., *Iteration Over Vectors in Genetic Programming*. HP Laboratories Technical Report HPL-2001-327, December 17, 2001.
- [4] O'Reilly, U-M and F. Oppacher, "An Experimental Perspective on Genetic Programming," *Proceedings of Parallel Problem Solving from Nature II*, 1992. [<http://www.ai.mit/people/unamay/papers/ppsn92.ps>].
- [5] O'Reilly, U-M, Email communication, 2/03.
- [6] Kinnear, K, "Generality and Difficulty in Genetic Programming: Evolving a Sort", *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, Morgan Kaufman, 1993. [<ftp://cs.ucl.ac.uk/genetic/ftp.io.com/papers/kinnear.icga93.ps.Z>].
- [7] Luke, S., *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*, PhD Dissertation. U. Maryland, 2000. [<http://cs.gmu.edu/~sean/papers/thesis2p.pdf>].
- [8] Abbott, R. J., J. Guo, and B. Parviz, "Guided Genetic Programming," *The 2003 International Conference on Machine Learning; Models, Technologies and Applications*, 2003.
- [9] Teller, A., "Turing Completeness in the Language of Genetic Programming with Indexed Memory", *Proceedings of the 1994 {IEEE} World Congress on Computational Intelligence*, IEEE Press, 1994
- [10] Shasha, D and K. Zhang, "Approximate Tree Pattern Matching," in *Pattern Matching in Strings, Trees, and Arrays* A. Apostolico and Z. Galil (eds.). Oxford University Press. 1997, 341-371.
- [11] Koza, J. R., M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, 2003.