

Genetic Programming Reconsidered

Russ Abbott

Department of Computer Science
California State University, Los Angeles,
Los Angeles, Ca. 90032
Email: RAbbott@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.

Teleological evolution (typical of genetic programming) differs from adaptive evolution (typical of agent-based systems). The former is goal oriented; the latter facilitates adaptation to a changing environment. It is possible to reformulate the former in terms of the latter. Within that framework, evolutionary pathways are more acceptable.

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 least not to the genetic programming system and typically not in any formal language.

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. After approximately a decade and a half, one might have hoped that a broad range of computational problems would by now be subject to solution by genetic programming. But that is not the case. Five years ago Langdon [2] wrote,

Computers that ‘program themselves’ [have] long been an aim of computer scientists.

Langdon’s book promises to be a step in that direction. Yet the goal of computers that program themselves seems as far beyond our grasp as ever. No programmer has ever lost his or her job to a GP system.

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. This is an especially severe limitation when the problem naturally presents its inputs in terms of complex data structures such as lists, sets, vectors, or arrays.

This too is a bit of hyperbole. Langdon claims to evolve list, stack, and queue data structures and programs that manipulate them. As we discuss below, Kinnear generated an $O(n^2)$ sort program even earlier. In fact, Kirshenbaum’s claim isn’t quite as strong as the above extract suggests. He continues,

In this paper, we present *bounded iteration* operators that allow the discovery of programs with arbitrary polynomial-time complexity.

So what’s the story? Are GP systems able to generate programs of greater than $O(1)$ complexity or not?

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: the fitness function necessarily provides information about what the solution parse tree must look like.

Finally, we distinguish between teleological evolution and adaptive evolution and discuss a class of problems for which Genetic Programming makes more sense.

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>;
```

The elements within angle brackets are program variables that serve as parameters to the construct that the GP engine generates.

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, but the differences made significant differences in the outcomes.

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.

When Kinnear reinstated the simple (unintelligent) swap but added an operation that returns the index of the smaller of two array elements, it turned out to be somewhat less easy but still not too difficult to generate a sort.

Without either of these two problem-specific operations, it turned out to be fairly difficult to generate a 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 framework—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.

Further, O'Reilly and Oppacher did not have intelligent swap as a primitive operation. Even had their system generated it, it would not have been highly valued since it would only rarely have put elements into their correct final positions.

Once a fragment such as the above appears in a population and is rated useful, it is a trivial additional (random) step to wrap another iteration construct around it to get full bubble sort. Kinnear's system did exactly that.

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.

III. EVOLUTIONARY PATHWAYS

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?

In answering this question, it is important not to be fooled by when in a run a solution appears. There is evidence that GP often functions just as

well when using mutation instead of crossover. (See Luke [7] Chapter 7 for a brief discussion and additional references.) When operating without crossover, GP becomes a parallel, iterative hill-climbing algorithm, where the fitness function defines the path up the hill and a large population increases the chances of finding the right hill. Within such a framework, both a good fitness function and a large population are critical.

An alternative approach is to attempt to generate building blocks dynamically—as we did in [8]. In Koza's automatically defined function (ADF) architecture [1] the (building block) functions are not specified in advance. The GP system is allowed to define whatever functions seem most effective. Although Koza reports the generation of GP-generated programs that make use of ADFs he does not report that any of those ADFs would be considered useful building blocks on their own, i.e., functions that one would expect to find in a program library.

IV. 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. (Perhaps this is one of the reasons Kirshenbaum suggested that GP never went beyond complexity $O(1)$.)

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—although we will not attempt to characterize how efficient a system must be to be considered evolutionarily efficient.

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

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

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

Clearly, a GP engine when running the fitness function \mathbf{Fit} will generate the parse tree \mathbf{P} , which, when executed, will compute \mathbf{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?

V. 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.

VI. PARSE TREES FOR PROBLEMS THAT HAVE MULTIPLE SOLUTIONS

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.

Let F be some computable function. Since F is computable, there must be one or more programs that compute it; hence there must be one or more parse trees for such programs. Let

SetOfPs be the set of parse trees for all the programs that compute F .

Let the fitness function Fit rank GP-generated parse trees, i.e., population elements, on the basis of how closely they resemble elements in SetOfPs . In particular, Fit takes as its fitness measure the maximum match between the candidate population element and the elements of SetOfPs .

Clearly, a GP system when running the fitness function Fit will generate a parse tree in SetOfPs , which, when executed, will compute F .

Thus any warmer/colder fitness function that has a positive correlation with a similarity function for solution parse trees is still necessarily a cheat. A fitness function with a positive correlation to Fit defines, simply as a consequence of that correlation, an evolutionary pathway based on parse-tree similarity. Since by definition, we want to generate some element in SetOfPs , a fitness function that is positively correlated to Fit is a cheat, and a fitness function that is not correlated to Fit will provide no useful guidance to a GP engine. The higher the correlation the easier it will be for the GP engine to find a solution.

The preceding discussion implicitly assumed that SetOfPs was finite. What of problems for which there are an infinite number of solutions? Almost every GP function set is general enough so that one can construct a *nop* subtree, i.e., a subtree that when plugged into another tree will not change the result produced by the parent tree. In general, such *nop* trees can be expanded to be arbitrarily large. Hence, most problems have an infinite number of possible solution trees.

In practice, most GP systems restrict the size of generated results. Without such restrictions, GP runs tend to bog down under the weight of parse-tree bloat. Thus infinite sets of solutions are not generally at issue. In addition, as one matches a candidate parse tree against larger and larger solutions parse trees, the matches will be less and less good the greater the difference in size between the candidate and solution trees. Thus even if one wanted to consider the complete (infinite) set of parse trees when computing the fitness function of a candidate parse tree, one would have to match against only the finite subset of that infinite set that consists of parse trees that are comparable in size to the candidate tree.

Furthermore, most GP systems have fairly powerful simplification systems available to them. *Nop* subtrees can be eliminated easily. Although it is undecidable in general whether a subtree is a *nop* subtree, since most GP systems limit execution time, the undecidability issue is mooted as well.

VII. USE OF GP FOR AGENTS IN AGENT-BASED SYSTEMS

The results in the previous sections presumed the use of a genetic programming system to generate a fixed target function. In that case it seemed that it was not possible not to cheat.

In a dynamic environment the situation is somewhat different. Imagine a GP system used by rule-driven agents in an agent-based environment in which each agent has its own set of objectives—objectives, which may or may not be compatible with those of the other agents. Assume also that each agent has its own internal GP system, which it uses to help it revise its internal rule set. (Clearly this implies both sophisticated agents and lots of computing power.)

In such an environment, even if the agents have fixed goals, since the environment is constantly changing (as other agents change their ways of interacting), there are no fixed target parse trees for any agent. Even if an agent achieves a maximally effective set of rules for some state of the environment, it is likely that as other agents adapt to that agent's behavior and change their behaviors, those rules will cease to be maximally effective.

In a sufficiently complex environment in which no Nash equilibrium is ever established (either because the agents exist in a constantly shifting fitness landscape composed of the behaviors of the other agents or because the environment itself changes for external reasons), each agent will be required to update its rule set continually.

In an environment in which there is no permanently optimum program, and hence no permanently optimal parse tree, use of a GP system to continually explore the space of possible parse trees seems to be a good use of the GP mechanism without the sense that that mechanism is being led to a predestined answer.

VII. TELEOLOGICAL VS. ADAPTIVE EVOLUTION

Let's call the process followed by traditional Genetic Programming (in which there is a fixed goal functionality) *teleological evolution*, i.e., evolution toward a predefined goal; and let's call the sort of evolution that occurs in constantly changing environments *adaptive evolution*.

A. Adaptive Evolution in Nature

Evolution in nature is adaptive: there is no predefined set of specific, functionally-defined behaviors toward which species evolve—reproductive success is an outcome result rather than an explicit goal. Yet (or perhaps as a consequence) natural evolution has produced amazingly complex functionality.

A technique commonly seen in natural evolution is the re-use of functionality. Structures that are useful for one purpose in one environment are appropriated to achieve another goal in a changed environment. To take just one example, precursors to hemoglobin had the ability to sequester oxygen. Originally, these proteins served to protect early life in an oxygen-rich environment: oxygen is toxic to many cell components. The ability of these proteins to remove oxygen from the immediate environment and store it away was “re-purposed,” and hemoglobin now serves as an oxygen transportation mechanism.

In a Genetic Programming framework such re-purposing would be similar to (a) the extraction of a subtree from a function that served a useful purpose in one environment and (b) its implantation as a subtree into another function that served a different purpose in another environment. When the parse tree for the first function evolved, the more general second function was presumably not needed. But (as it just happened to turn out) a subtree of that original parse tree was useful for producing the second function when it was needed. This is adaptive evolution.

B. Teleological Evolution as a Sequence of Adaptive Evolutionary Steps

One can reconceptualize Genetic Programming-style teleological evolution as an exploitation of adaptive evolution that telescopes a sequence of adaptive evolutionary environments.

Consider a teleological evolutionary system that is governed by a fitness function that is normalized to range from 0 to 10, higher is better.

One can imagine a sequence of evolutionary processes in which the n^{th} step (n ranging from 0 to 10) takes as its (local) fitness function the production of programs with (global) fitness n . In such a sequence, one would use as the initial population at the n^{th} step the terminal population of the $(n-1)^{\text{st}}$ step.

Although not identical, such a sequence of evolutionary steps is sufficiently similar to that of traditional Genetic Programming to be a useful analog. The difference, of course, is that in standard Genetic Programming the multiple evolutionary environments all exist concurrently. One might call such a staged strategy *teleological evolution through controlled sequential adaptation*.

Such an approach might very well serve as a practical way to maintain desired levels of population diversity in day-to-day use. This sort of staged approach combines the population dynamics of genetic algorithms with the strategy, used in simulated annealing, of controlling the rate of convergence. The criterion for when to move from one step to the next would be a runtime parameter.

When used in a practical setting, one would also want each step, i.e., step n , to generate as broad a range as possible of elements with fitness level n . To accomplish that, one could rank population elements not only on how closely they achieved fitness value n , but also on how dissimilar they are to other population elements.

B. Sequential Adaptive Evolution Clarifies the Role of Evolutionary Pathways

Teleological evolution based on a sequence of adaptive evolutionary steps makes the notion of an evolutionary pathway both more explicit and more acceptable. In nature, evolutionary pathways are always after-the-fact: one doesn't know that one is following a path to a result until one gets there. In controlled sequential adaptive evolution, the pathway is at least hidden. Since the sequence of environments is defined in advance, the goal is pre-defined. But the environment at each step reflects only an allegiance to a specific sub-goal.

Although there is no guarantee, a successful final result of such a process i.e., a program with fitness value 10, will most likely be generated by combining components from elements that had been well adapted to environments that existed in earlier steps.

Such a step-wise approach still defines an evolutionary pathway, but it is a pathway that seems less contrived. To be sure, the (global) fitness function must be positively correlated to a parse tree matching function for the (set of) successful final parse tree(s). That is, successful elements at step n must have parse trees that are better matches to the final result than successful elements earlier in the sequence. Otherwise, taking those intermediate steps would be counterproductive.

As we saw, this is what happens in nature; it should be no surprise that it happens in an evolutionary process modeled on natural evolution. Controlled sequential evolutionary adaptation makes it clear that evolutionary processes are most successful when there is an evolutionary path (or a collection of evolutionary paths) that an evolutionary engine can follow. A good fitness function illuminates the path.

VIII. FITNESS FUNCTIONS CAN HIDE AS WELL AS GUIDE

A corollary of these observations is that fitness functions can hide as well as guide. Genetic programming suggests the possibility of creative solutions to unsolved problems. But a fitness function that constrains the evolutionary pathway along a particular route will make it much more difficult to find solutions whose parse trees require pathways that do not correlate with that of the selected fitness function.

IX. 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 author thanks 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.