

Guided Genetic Programming

Russ Abbott (rabbott@calstatela.edu), **Jiang Guo**, **Behzad Parviz**
Department of Computer Science (323-343-6690, fax 323-343-6672)
California State University, Los Angeles,
Los Angeles, Ca. 90032

Abstract. This paper argues that genetic programming has not made good on its promise to generate computer programs automatically. It then describes an approach that would allow that promise to be fulfilled by running a genetic programming engine under human guidance. **Key-words:** guided genetic programming.

1. Introduction

Genetic programming (GP) (Koza 1990) is the use of a genetic algorithm (Holland 1975) to generate computer programs.

In his www.genetic-programming.org web page, Koza writes,

Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem.

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

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.

Typically, the problem to be solved is characterized to the genetic programming system in terms of a fitness function, which itself is expressed as a relatively low-level computer program.

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 by now a broad range of computational problems would be subject to solution by a genetic programming engine. But that is not the case.

Just 5 years ago in his enthusiastically entitled book **Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!**, Langdon wrote,

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

The book promises to be a step in that direction. Yet the goal of computers that program themselves seems to be as far beyond our grasp as ever.

In a recent paper, Kirshenbaum [Kirshenbaum] wrote,

One of the main limitations of traditional genetic programming (Koza 1990, Banzhaf, et al. 1998) 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 is a bit of hyperbole as well. Langdon claims to evolve list, stack, and queue data structures and programs that manipulate them. And as we shall discuss below, Kinneer 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,

When attacking a problem known to require [programs of] more [than $O(1)$] computational complexity, the typical approach is for the experimenter to handcraft a “harness” which, for example, evaluates the candidate programs once for each element of an input data sequence and averages the result. ... In other work, the higher complexity is encapsulated in special purpose operators [that] perform specific operations such as summing or finding the mean of a vector of numbers.

Kirshenbaum's paper then goes on to describe a collection of iterative operator schemata. He writes,

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 answer seems to be both *yes* and *no*.

The remainder of this paper explores this question by examining the brief history of attempts to generate a sort program. We then describe how a genetic programming system with user guidance may be the best approach to generating more difficult programs.

2. Generating a sort program

Sort is of complexity $O(n \log(n))$. Simple 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 two most prominent are quite old.

O'Reilly and Oppacher

In [O'Reilly 1992], O'Reilly and Oppacher report a failed attempt. They attempt to generate a program that transforms an unsorted input array into a sorted array by moving element around in the array.

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

The allowed operations are: decrement a variable, read an array location by 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 = <low>; i < <high>; i--)  
  <body>;
```

The elements within angle brackets are 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 [O'Reilly email] 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 in [Koza 1999]. The other operators and data structures are also quite similar to those reported in that book. Unfortunately, [Koza 1999] does not report whether attempts were made to generate a sort program using his version of those constructs.

Kinnear

Kinnear [Kinnear] also reports on experiments to generate a sort program. Like O'Reilly and Oppacher, Kinnear attempted to generate a program that, when given an unsorted array will terminate 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.

Fitness functions

It is worth comparing Kinnear's fitness function with that of O'Reilly and Oppacher. Kinnear's fitness function counted the number of swaps needed to convert the output of a program run into a correctly sorted result. O'Reilly and Oppacher's counted the number of out-of-place elements.

This difference may have contributed to the difference in the results of their experiments. A program consisting solely of the inner loop of bubble sort would have reduced the number of inversions needed to completely sort the array. Such a program would have been rewarded by Kinnear's fitness function but not by O'Reilly and Oppacher's.

We suspect that a well-designed fitness function, one that can guide a GP system along a path to a solution can be a powerful aid in getting to a solution. See further discussion below.

The results of Kinnear and Kirshenbaum demonstrate that programs of complexity greater than $O(1)$ can be generated by GP systems—but perhaps only if powerful (and applicable) operations and/or control structures are provided as primitives.

3. Steps to a sort program

Oogp [Abbott] is an object-oriented genetic programming system that operates within a Java environment. In the next few sections we discuss how we used oogp to generate a sort program under user guidance.

The task given oogp was to generate a program that accepted as input an unsorted `List` (a subclass of Java's `ArrayList`, see [Sun]) of `Integer`¹ objects and returned as output a new `List` with the `Integer` objects sorted.

¹ As explained in [Abbott], the `List` elements were really `Int` objects, which, like `Integer`'s are wrapped primitive `int`'s. Unlike `Integer`'s, `Int`'s are mutable. For sort, mutability is not an issue.

As an object-oriented system, oogp works with methods rather than functions. The equivalent of the traditional GP *function set* is the collection of methods defined in the classes that oogp is allowed to use.

For the sort problem, oogp had available to it all the `ArrayList` methods, including `add()`, `get()`, `indexOf()`, `remove()`, etc.

The sort task given to oogp should be contrasted with the work cited in the previous section in which the input was an array and the job was to use a very restricted set of operations to move elements around in that array. Oogp was operating in a much more general environment than that used by either O'Reilly and Oppacher or Kinnear. As [Koza 1999] points out, the more flexibility available to a GP system, the harder it is to find a solution. [Koza 1999] reports that his Genetic Programming Problem Solver typically solved problems by using one to two *orders of magnitude* more evaluations than those used by a GP system tailored to the specific problem.

Besides the standard `ArrayList` methods, two additional `List` methods were provided. They are `iterate()`² and `insertAsc()`.

```
List iterate(Function start,  
              Function continue);
```

`Iterate()` is a method of `List` objects. Although it performs a loop-like function, it is not built into oogp as a primitive control structure. `Iterate()` takes two arguments, both of them executable functions, and it returns a `List` object. Its two arguments are as follows.

The `start()` function takes the `List` object to which `iterate()` is applied as an argument and returns a `List`. For the sort problem the correct answer is for `start()` to return the empty list. (An `emptyList()` method is available that does exactly that.)

² As explained in [Abbott] `iterate()` is more general than shown here. See the appendix for the `iterate()` code.

The `continue()` function also returns a `List`. It takes two arguments. The first is an element of the `List` to which `iterate()` is applied; the second is a `List` object. `Continue ()` is of the following form.

```
List
  continue(Int currentElement,
           List previousResult){
  <Compute and return a new List
  result based on the
  previousResult and the
  currentElement.>
}
```

`Iterate()` executes by examining in sequence the elements of the `List` on which it is called. It repeatedly calls `continue()`, passing to it the current `List` element (as `currentElement`) and the result of the previous call to `continue()` (as `previousResult`).

The first argument to `iterate()` is called once and provides a value for `previousResult` when `continue()` is called the first time.

```
void insertAsc(Int I);
```

The `insertAsc()` method is also a method of `List` objects. It takes one argument, which it inserts into the `List` object to which it is applied. In particular, it inserts its argument in front of the first element of the `List` that is greater than or equal to the argument. In other words, if the list on which `insertAsc()` is called is ordered, it inserts its argument into that list in the correct position.

Given `iterate()` and `insertAsc()` oogp produced the following sort method.

```
void sort() {
  iterate(
    emptyList(),
    previousResult
    .insertAsc(currentElement);)
}
```

This sort program implements an insertion sort by calling `insertAsc()` to insert elements from the input list one at a time into an originally empty list. This simple insertion sort is of complexity $O(n^2)$. Like Kinnear's first solution, it is a relatively trivial solution to the problem of generating a sort program.

Perhaps a word should be added here about fitness functions and randomly generated programs. In our experiments we found that most results arose in later generations. But we also found that the pedigree of most correct programs was quite short.

Often the system discovered that the only way to return an output list of the appropriate length was to use `iterate()` starting with an `emptyList()`. Our fitness function rewarded programs for generating output lists that were as long as but different from (not the same object as) the input list.

From there it's a small (random) step to find that `insertAsc()` as configured above produces the correct result.

4. A less trivial solution

Rather than try to generate a less trivial sort by using lower-level methods as Kinnear did, we took an alternate approach. We asked whether oogp was capable of generating the (building block) methods that it used.

This approach suggests a remedy to O'Reilly and Oppacher's critique that GP is not organized to generate solutions in a hierarchical manner. They wrote,

GP's process of putting a solution together is not ... hierarchical. Hierarchical solution construction is a process of top-down and bottom-up, level by level, construction of increasingly more efficient pieces of a solution. ... In this way, a hierarchy of function specialization and generalization is built.

Proceeding hierarchically, one may ask whether oogp is capable of generating the `insertAsc()` method. The answer, of course, is that it is: as a method of the two methods: `add()` and `findPos()`.

The standard `ArrayList` method `add(Int i, Int element)` inserts element into the `List` to which it is applied at position `i`. The element previously at `i` and all elements at higher positions are pushed up by one position. This method is available to `List` objects since `List` is a subclass of `ArrayList`.

Define `findPos()` to be a `List` method that when given an element, returns the position within the `List` where that element would be inserted by `insertAsc()`.

If such a method existed, could oogp generate `insertAsc()`? The answer, of course, is that it can quite easily.

```
void insertAsc(Int int_0) {
    Int int_1;
    int_1 = findPos(int_0);
    add(int_1, int_0);
}
```

What about `findPos()`, can oogp generate that? Again, the answer is yes.

```
Int findPos(Int int_0) {
    iterate(0,
        {if (int_0 > currentElement)
            previousResult.incr();
            previousResult;
        }
    );
}
```

These results show that starting with `iterate()`, `emptyList()`, and `add()` as methods in the `List` class, oogp (and presumably any other GP system) can generate `findPos()`, `insertAsc()`, and `sort()` fairly easily—

- if asked to do so one method at a time,
- in that order, and
- if once generated, the building-block functions are made available as `List` methods.

5. Guided GP

Is it possible to generate `findPos()`, `insertAsc()`, and `sort()` simultaneously?

In oogp we can generate all three methods at the same time by running the three evolutions in parallel.

To do so we create a separate population (and separate threads) for each of the three methods. In addition, we define `findPos()` and `insertAsc()` as method stubs in the `List` class.

In general, when oogp generates a candidate program, the function set available to it consists of the methods in the classes that it can see. In this case, we allow oogp to use `findPos()` when attempting to generate `insertAsc()` and to use both `findPos()`

and `insertAsc()` when attempting to generate `sort()`.

When executing a program that contains `findPos()` or `insertAsc()`, oogp uses the current best element that had been generated to that point for that method.

The result is a collaborative effort that combines a genetic programming engine (running three evolutions in parallel) with human intuition.

- The goal is to generate a sort program.
- The user contributes what in this case turns out to be (although need not be) a top-down analysis, offering the suggestion that `findPos()` and `insertAsc()` might be useful functions to have on hand when generating a sort.
- Oogp then sets to work generating a sort using the originally available `List` operations along with increasingly good implementations³ of `findPos()` and `insertAsc()`.

Clearly such an approach is not as desirable as having oogp generate building blocks all by itself. But to date, no GP system seems capable of doing that. Allowing a user to suggest building blocks may be a reasonable compromise.

We did *not* attempt to have oogp generate the `iterate()` method. The computational intricacies required for `iterate()` seem to be beyond the capabilities of current genetic programming systems.

The fact that we feel obliged to make that statement suggests that GP is still gravely limited in what it can do. After all, our hand-made implementation of `iterate()` is only 10 lines of Java code. (See the appendix for the actual code.) Yet it is difficult to imagine *an evolutionary path* that would produce this code.

We emphasize *evolutionary path* because that is what GP claims to provide. Clearly if one runs a random program generator long enough, any program will eventually appear.

³ Note that running three generations in parallel can be quite inefficient. The fitness of higher-level elements changes as the fitness of lower-level elements improves.

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 significant evidence that GP often functions just as well when using mutation instead of crossover. (See Luke Chapter 7 for a brief discussion and additional references.) This makes GP 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. From that perspective, a good fitness function, one that identifies useful stepping-stones along the way, is critical.

6. Comparison with other work

The notion of generating multiple functions simultaneously was employed by Langdon [Langdon] in his generation of `List` operations. In that work, a predefined suite of `List` primitives was evolved simultaneously.

Langdon's intent was to evolve the entire suite in one genetic programming run, even though the separate operations were not defined in terms of each other. It isn't clear why it was important to Langdon's work to evolve the functions in the same run. His primary objective seemed to be simply to demonstrate that these functions could be evolved. But the fact is, he did evolve them together. Langdon's work can be seen as a precursor to the evolution of new `Class` definitions that we are anticipating in extensions to oogp.

In Koza's [Koza 1999] automatically defined function (ADF) architecture 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 ADF's it is not clear whether any functions emerged that would be considered useful building blocks on their own, i.e., functions of the general utility of `findPos()` or `insertAsc()`.

7. Conclusions and comments

Through a user-computer collaborative effort, a genetic programming system is able to generate a sort program from the operations available in a standard `List` class.

Heretofore, the human contributions to the success or failure of a genetic programming attempt to solve a problem have been somewhat disguised: (a) in the functions and terminals provided to the system and (b) in the fitness function used to evaluate generated programs.

The work reported here suggests that instead of hiding the human contribution we might profit by encouraging it. In particular, it seems worthwhile for the user explicitly to invite the GP system to generate building-block functions that the user thinks might be useful.

We also believe that a user can profitably provide guidance by defining fitness functions that map a path through the evolutionary landscape toward problem solutions.

References

- Abbott, R. J., J. Guo and B. Parviz, "Object-Oriented Genetic Programming: An Initial Implementation," submitted to *The 2003 International Conference on Machine Learning; Models, Technologies and Applications*, 2003.
- Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert E.; and Francone, Frank D. 1998. *Genetic Programming: An Introduction*. San Francisco, CA: Morgan Kaufmann.
- Holland J.H., *Adaptation in natural and artificial system*, Ann Arbor, The University of Michigan Press, 1975
- 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>.
- Kirshenbaum, Evan, *Iteration Over Vectors in Genetic Programming*. HP Laboratories Technical Report HPL-2001-327, December 17, 2001.
- Koza, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University Computer Science De-

- partment Technical Report STAN-CS-90-1314. June, 1990.
- Koza, John R., Bennett, Forrest H, III, Andre, David, and Keane, Martin A. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann. 1999.
- Langdon, W B., *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Kluwer Academic Publishers, 1998.
- 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>.
- O'Reilly, U-M and Franz 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>.
- O'Reilly, U-M, Email communication, 2/03
- Sun Microsystems,
<http://java.sun.com/j2se/1.4.1/docs/api/java/util/ArrayList.html>

Appendix. The Java code for `iterate()`.

```

public Object iterate(Function start, Function continue) throws Exception {
    Iterator it = underlyingIterator();
    Object[] args = new Object[]{null, start.applyTo(new Object[]{this})};
    while (it.hasNext()) {
        args[0] = it.next();
        args[1] = continue.applyTo(args);
    }
    return args[1];
}

```

Oogp uses `java.lang.reflect.Method.invoke()` to execute its code. `invoke()` requires that arguments be passed as `Object` arrays. Our `applyTo()` method eventually calls `invoke()`. The `Object[] args` contains `currentElement` and `previousResult` as `args[0]` and `args[1]` respectively. `iterate()` is declared to throw an exception because its function arguments may throw an exception. If a program throws an exception, execution is terminated and the program is given the worst possible fitness value. The `underlyingIterator()` function returns an iterator for the `List` object to which `iterate()` is applied. This allows `iterate()` to be used for other than `List` objects.