GENETIC PROGRAMMING, INDEXED MEMORY, THE HALTING PROBLEM, AND OTHER CURIOSITIES

Astro Teller
4019 Winterburn Ave
Pittsburgh, PA 15207

Carnegie Mellon University
Dept. of Computer Science
astro@cs.cmu.edu

Abstract

The genetic programming (GP) paradigm was designed to evolve functions that are progressively better approximations to some target function. The introduction of memory into GP has opened the Pandora's box which is algorithms. It has been shown that the combination of GP and Indexed Memory can be used to evolve any target algorithm. What has not been shown is the practicality of doing so. This paper addresses some of the fundamental issues in the process of evolving algorithms and proposes a variety of partial solutions, in general and for GP in particular.

I INTRODUCTION

Traditional GP evolves functions, not algorithms. Functions are mappings from inputs to outputs. They are reactive; their stimulus-response makes them the amebas of the computational world. Algorithms are procedures that incorporate current input and past inputs, into an iterative or recursive process that may eventually produce an output. Algorithms are the humans (or dolphins) of the computational world.

It is possible to update the GP paradigm to evolve algorithms. But what effects does this have on the efficacy of GP? Whenever one tool is taken from one task to be used for another, that tool brings biases and limitations. In the same way that back-propagation does not provide the same properties of convergence for recurrent neural networks that it does for feed-forward neural networks, the GP paradigm is not powerful enough to handle the complexities of algorithms.

After introducing GP, Indexed Memory, and their extension to general algorithms, this paper will outline some of the fundamental problems with evolving algorithms. Then four proposals for improving the GP paradigm to surmount these problems will be discussed. Techniques for implementing these solutions in GP will be included.

These proposals, while grounded in the theories of computation, are no substitute for experimentation. Evolutionary computation has been and will continue to be a field with a strong empirical basis. Time after time our intuitions have failed us about what will in fact work well. Only when serious experiments and controls are done and reported will we begin to know what is and what is not tractable. I hope, however, that these suggestions inspire fruitful investigations in this area.

2 GENETIC PROGRAMMING

Genetic Programming is a strategy for evolving functions that perform well on assigned tasks.[Koza 92] These evolved functions are represented in GP as Lisp-like expressions consisting of non-terminals (atomic functions) and terminals (e.g. variables and constants). Simple GP functions might look like:

- **(* 2 (IF (= 4 x) ELSE (x/(- 4 x))
  THEN (cos (exp (+ x (* (+ 5 6) x))))))**
- **(exp (/ 9 x) (- (+ (+ 9 8) x) 12))**

Given a sufficiently expressive set of mathematical functions, variables, and constants, this type of "language" can represent many desired functions, such as $x^3*\cos(2*x)$. This function might evolve in the following form:

- **(* (* x (* x x)) (cos (* x 2)))**

The process of finding a GP function that is a good or perfect approximation to the target function can be summarized as follows. The population is initialized with a set of randomly generated individuals. Each member from the current pool of functions is tested to determine its error on some task. A new pool is created in which the functions with lower error have higher representation. The new pool is then subjected to various kinds of recombination. Two popular varieties are mutation and crossover. Over time the most successful individuals in the population become increasingly accurate approximations of the correct solution to the task.

Some small percentage of the population is chosen at random from the new population and some non-terminal or terminal part of the function is changed. This is called "mutation". If the first function above had its third **x** changed to a 1, it would be removed from the population and the following function would be added.

(* 2 (IF-THEN-ELSE (= 4 x) (x/(- 4 1))
(cos (exp (+ x (* (+ 5 6) x))))))

In crossover, two functions are chosen at random from the new population and they exchange randomly selected parenthesized sub-expressions. The first two examples shown above might look like this after undergoing crossover:

- **(* 2 (IF-THEN-ELSE (= 4 x) (x/(- 4 x))**
  (- (+ (+ 9 8) x) 12))))
- **(exp (/ 9 x) (cos (exp (+ x (* (+ 5 6) x))))**

This language is not powerful enough to express many algorithms. For example, there is a simple procedure for checking whether a string is of the form $\mathbf{0^n1^n}$, but there is no way to make such a procedure in this kind of Lisp-like structure. There is no mechanism for arbitrary length strings to be shown to the function (tricks like Godel numbering just shuffle the problem to a different place) and no way for the function to iterate an arbitrary number of times.

We could encode the $\mathbf{0^n1^n}$ problem by giving the GP function a series of tests. The variable **x** could be used to give the current character (**x** = {0,1,2}) where 2 means "end of string". The fitness (average success) of the function could be based on whether it returned a positive or negative number when it received the 2. Here the GP environment might interpret positive numbers as "in the form $\mathbf{0^n1^n}$" and negative numbers as "not in the form $\mathbf{0^n1^n}$". But since each function has no access to the previous inputs and the previous inputs (where x was equal to 0 or 1) determine the correct response when the end of string is reached, this problem is unsolvable for traditional GP. In general, solutions to these kind of problems requires something extra: **memory**.

3 INDEXED MEMORY

Indexed memory is a simple addition to the GP paradigm.[Teller "Evolution" 94] **Read** and **Write** are added as new non-terminals in the language. Each GP function is given access to its own array of integers, indexed over the integers. The expression (Read 5) returns an integer in just the same way that (* 5 17) does. In general, (Read X) returns the integer stored in memory position X, where X is an integer. And (Write -19 101) returns the old value of memory position 101 and has the side effect of changing the value of memory position 101 to be -19. A simple GP function that utilized indexed memory might be:

(IF (< (Read 4) (Read (Read (+ x 4))))
THEN (Sqrt (+ (* x 8) 1)) ELSE (* x x))

Using indexed memory, a GP function could save past inputs and use them in a process like the $\mathbf{0^n1^n}$ problem. For each 0 or 1 that the function received, it could store the value in a memory location. Then, when **x** = 2, it could decide whether to return a positive or negative number based on the values stored in its memory. However, since the series of 0's and 1's can be arbitrarily long and the GP function is of fixed size, even if the memory size is infinite, there is a length of strings above which the function cannot always classify the input sets correctly. So memory is necessary, but there is another element missing from traditional GP: **iteration**.

4 TURING COMPLETENESS OF GP+IM

GP plus indexed memory (GP+IM) has a key ingredient that was missing from GP. But, for the reason just given, the procedure for checking $\mathbf{0^n1^n}$ can not be represented with any GP function, even if it contains Read's and Write's. But writing such an algorithm in the following language is possible.

Repeat
Evaluate < GP+IM function>
Until
< Some particular state of the memory >

In fact, all algorithms can be expressed in this language.[Teller "Turing" 94] The formal description is that this language is Turing complete. For the remainder of this paper, GP will be used to refer to traditional GP and GP+IM will be used to abbreviate this language of **Repeat GP+IM Until StateX**. In this language the GP+IM paradigm works as follows: There is some task to perform. Input is placed in the memory array. Then the GP+IM function is repeatedly evaluated until its memory moves into a specific state.(e.g. Memory[1] == 0) When this happens the memory is examined and one or more memory values are extracted and interpreted as a response. (e.g. Memory[25] < 0 means "unbalanced", Memory[25] >= 0 means "balanced") This new paradigm is not evolving functions. It is evolving algorithms.

5 THE HALTING PROBLEM

The most obvious difficulty with evolving algorithms is the property of algorithms that they may run on forever. It has been shown that, in general, it is not always possible to tell if an algorithm will ever halt on a particular input. This property of algorithms is known as the halting problem.[Hopcroft 79]

The halting problem presents a real stumbling block for the effective evolution of algorithms. Suppose that there are N algorithms in the population and the GP+IM procedure is to test each algorithm on some input and use the results to approximate its real fitness. Its real fitness is its average performance on all possible inputs, but that is impractical or impossible to calculate exactly since there are often a huge or infinite number of possible inputs to an algorithm. If the procedure waits until all N algorithms have returned an answer, there may never be a next generation because some of the algorithms may never return an answer.

So is all lost ? That remains an open question, but there are some possible solutions. For example, a better GP+IM procedure takes this same population of N algorithms and the same task for the algorithms to be tested on. But this procedure runs the N fitness tests in parallel and stops when the time since the previous algorithm completed passes some threshold. This termination criteria is very much like making popcorn. When there start to be fewer and fewer pops per second you listen and when the frequency drops below a threshold, you stop the process and give up on the still unpopped kernels. In the case of the population however, you don't want to assign zero fitness to those algorithms that had not completed yet when the fitness test was terminated. A more graceful assignment of blame for these non-responses is to keep a running average over multiple tests or even generations for each individual and to give that individual 0 fitness for that test, but to calculate its fitness for reproduction based on all the tests it has performed since it was created. This way algorithms that run on forever for a few particular cases may still thrive in a population because their average success at a task is still above average. The crucial parameter in this popcorn method is the threshold after which the remaining algorithms are abandoned. This is exactly the stopping rule or the "secretaries problem" in the field of operations research. Given functions which describe the costs and values of executing for another unit of time, optimal thresholds can be determined.[Silverman 89]

There is another possible solution to the halting problem in the evolution of algorithms. GP+IM has access to all of memory at every transition (evaluation of the GP+IM function) and can, in general, change many memory elements on every transition. Instead of waiting for each algorithm in a population to finish, the process of testing the algorithms can work by demanding an immediate response after the algorithms have had some time to run. This technique in fields like planning is called **Anytime** algorithms. Suppose for each individual in the population the input is placed in memory and the individual is given a short amount of time to run. After this short period of time the values of certain memory positions are extracted and interpreted as the answer. Instead of evolving algorithms that eventually or even quickly halt, there is now no issue of halting. We have traded the halting problem for the question "Is it practical to evolve algorithms which keep their 'best guess' on the right answer up-to-date with the computation they are doing ?" There have been no rigorous test of this particular technique, but initial results, described in the applications section of this paper, show that this strategy may be viable.

These proposals suppose certain features of the space of algorithms that are not necessarily true. One proposal assumes that fitness proportionate reproduction clears out most of the algorithms that always run on forever and both assume they can use successful algorithms to breed other algorithms that have higher than random fitness. It will be shown below that these are not safe assumptions. The Popcorn method and the Anytime procedure are both useful additions to the GP+IM paradigm, but there are more hurdles to be cleared.

6 THE SPACE OF ALGORITHMS

To understand better why the problem is not so easily solved, we need to understand the topological natures of the space of functions and the space of algorithms. The space of functions is the set of all finite non-recursively defined mappings from inputs to outputs in a particular domain. The topology of the space of functions is determined by the syntax in which these mappings are written. Two functions are close to each other in the space of functions, not if they do similar things, but if they are syntactically similar. In the space of functions it happens that, while not the same, these two concepts are related. For example, $\mathbf{x^2 * 3x - 1}$ is syntactically similar to $\mathbf{x^2 * 3x + 1}$ and in fact the graphs of $\mathbf{y = x^2 * 3x - 1}$ and $\mathbf{y = x^2 * 3x + 1}$ are quite close. This fact is essential in the success of GP. GP implicitly relies heavily on the notion that if a function is successful then other functions "near" it in the space of functions will have a higher chance of being successful than functions chosen at random from the space. The word "functionality" will be used now to denote what a function **or** algorithm does, as opposed to the syntax in which it is written.

The space of algorithms is not so well behaved. This space is the set of all procedures that can recognize recursively enumerable sets. As before, the topology of the space is determined by the syntax with which these procedure are described. As with functions, two algorithms are close to each other, not if they accept the same set of input, but if they are syntactically similar. Unlike functions however, there is no real correspondence between syntactically similar algorithms and the inputs they accept. If a large mathematical expression is taken and a single **+** changed to a **-**, the expression will often be very similar. As any programmer knows, a one character change in a program usually causes complete chaos and a totally different response to many or all inputs. For example the two algorithm segments:

- **While x<10 {y=y*y; x=x+1}**
- **While x<10 {y=y*y; x=x-1}**

are very similar in their form, but very dissimilar in their function.

This claim that changing algorithms is different from changing functions needs further substantiation. It is clear that $\mathbf{x^2 + 9x + 1}$ and $\mathbf{-x^2 + 9x + 1}$ are syntactically similar and yet their graphs are close for only a tiny fraction of their domain. However, there is a fundamental difference between the change in functionality a small syntactic perturbation causes to a function and the change in functionality a small syntactic perturbation causes to an algorithm. This difference is related to the notion of chaos.

The concept of chaos (non-linear dynamics) is that tiny changes to a system cause large changes over time. When a function is changed in some small way there is a corresponding change in its functionality. But the functional "distance" between the two functions stops there. When an expression in an algorithm is changed in some small way, the effect of that expression is changed as though it were a function. But on the next iteration of the algorithm, its actions are based on its previous results. The changes in the previous results due to the small change cause further changes to the current results and these changes to the functionality of the algorithm compound on each iteration. Though the analogy is not perfect, think of a random walk in a high dimensional space that starts at the origin. Making a change to a function is like taking one random step. Making a change to an algorithm is like taking an unknown and potentially infinite number of random steps. The more steps taken, the farther the expected distance the last step of the random walk is from the origin. The conclusion is that of the previous paragraph. The space of algorithms is extremely discontinuous with respect to functionality and therefore fitness surfaces over this space will be extremely discontinuous.

GP works by searching syntactically nearby functions and this works well because these functions are often functionally similar too. GP+IM, in its present incarnation as a paradigm for evolving algorithms, is probably not powerful enough to search the space of algorithms effectively. Fundamentally, this is because the correspondence between syntactic and functional similarity is so weak in algorithms. So the evolution of algorithms that solve non-reactive problems demands new genetic operators. Mutation must give way to some new operator that changes how the algorithm acts in some special case, like an edge condition. Crossover must give way to a new genetic operator which can take algorithms and recombine them in non random ways. One useful operator would be to take two algorithms and create children algorithms that accept the union, intersection, union of the inverses, etc. of the sets of inputs accepted by the parents. These children are related to their parents functionality and so hill climbing might proceed. Such "operators" have been made for other syntactic representations of the complete space of algorithms. [Hopcroft 1979]

7 CHANGING THE ALGORITHM SPACE

Because the space of algorithms is so discontinuous, the mutation operator might almost as well erase the old individual and make a new one from scratch. A better alternative is to use an algorithm-mutation operator which takes into account how algorithms work and what it is mutating. For example, this new operator would be careful to leave unchanged any variables, constants or memory locations that are tested in IF-THEN-ELSE's and then affected in the GP+IM subtree of that IF-THEN-ELSE. The algorithm-mutation operator would have a more positive effect by only changing constants by small percentages and only changing non-terminals to closely related non-terminals. (i.e. **<** <--> **<=** <--> **=** <--> **>=** <--> **>**)

That is a specific technique in GP+IM for avoiding some of the discontinuities of the space of algorithms. In making new operators we leave the topology of the space alone but redefine our evolution topology by making the notion of "near" more constrained and therefore more likely to have similar functional properties. But this does not change the fundamental topology of the space. Another technique to get around the discontinuity of the space of algorithms is to change the actual shape of the space by finding some representation that is more continuous (and therefore easier to hill-climb). We can introduce a new non-terminal that takes 3 arguments: (EITHER X Y Z). This function chooses a random number between the MINVALUE and MAXVALUE. If the number chosen is less than X then Y is evaluated and returned. Otherwise Z is evaluated and returned. This introduces a kind of fuzzy logic into GP+IM. It should be clear that this does not reduce the expressiveness of GP+IM because (EITHER MAXVALUE Y Z) is the same as just Y. Nor does EITHER increase the expressiveness because GP+IM is already Turing complete.

What we do get from EITHER is a smoothing out of the topology of the space of GP+IM algorithms. Now when we change (EITHER 11 Y Z) to (EITHER 12 Y Z) the entire algorithm will usually do what it did before. This means that now two algorithms that are syntactically similar are more likely to have similar functionality, especially if they differ only in values in the first position of EITHER non-terminals. This again make it easier to hill climb in the fitness space.

These attempts to find better operators or better representations suffer from a basic flaw. The very nature of non-linear dynamics is that changes **no matter how small**, lead to unpredictable end results. Does that mean that we should abandon all hope of hill- climbing in this space ? No. There is nothing we can do to insure that all or even any of the algorithms in a neighborhood will have high fitness. But we can increase the **likelihood** that an algorithm in a well chosen neighborhood will have high fitness. For example, if we could find flawless operators, we would have solved the halting problem, which is unsolvable. However, operators may exist that produce algorithms that halt with higher frequency then algorithms chosen at random from the space. New representations, like the EITHER operator, still allow changes to the system and so any representation will be chaotic. But empirical studies may show that the EITHER operator has some positive effect on hill-climbing in the algorithm space.

8 APPLICATION

I have begun work at CMU with Dr. Manuela Veloso on a system for the synthesis of computer vision and machine learning. The ultimate goal is a general signal to symbol vision translation system. An expandable set of objects are to be recognized in natural settings (held by a person, sitting on a cluttered work bench, etc.).

The basic inspiration for the work is the idea that the signal to symbol problem can be thought of as a large number of algorithms being applied in parallel to a signal. The signal in this case is a two dimensional array of pixels. Each algorithm casts a "vote" for the likelihood that the symbol shown in the picture is the symbol that this particular algorithm is specialized to find. Given a large number of such algorithms, one for each symbol to be recognized, and an algorithm for taking all these votes and returning some final decision and confidence, a real breakthrough in computer vision object recognition may be possible. This new machine learning technique is called **PADO**: **P**arallel **A**lgorithm **Discovery** and **O**rchestration.

Each algorithm is similar to a genetically programmed function with a fairly standard set of non-terminals. In addition, the non-terminal set includes a few primitive functions that act on areas of a picture (average pixel value, variance in pixel value, etc.). And each algorithm has access to an array of memory through the indexed memory non-terminals. This sounds like a function with memory, not an algorithm, because there is no loop construct and no recursion. So this "algorithm" is repeatedly applied until it signals its completion by some particular state of its memory (e.g. Memory[1]==1) . This repeated application of a GP function with indexed memory is Turing complete. [Teller "Turing" 94]

Each algorithm is run by repeated application of the GP+IM function until it signals its completion through some memory state or some maximum amount of time has elapsed. The "answer" the algorithm returns is extracted from the memory (e.g. Answer = Memory[17]+Memory[99]). This is the Anytime method mentioned in section 5. It is in this way that we avoid problems related to halting. Though the research is too young to have reliable results about object recognition, this Anytime approach has already shown promise. These algorithms have already demonstrated the ability to respond in this way to simple tasks like the noting the passage of time or the average intensity of a picture.

The second kind of problem mentioned in this paper is the discontinuity of the space of algorithms. One of the proposed solutions to this problem was to find better operators. In this context a better operator is one that has a high chance of producing as output algorithms with equal or higher fitness then those algorithms the operator took as input. Though we have not perfected the process yet, we are working on a method for evolving the operators themselves. In the end we hope, not only to find useful operators, but by examining these operators to learn something about the topology of the algorithm space, at least for this domain.

9 CONCLUSIONS

GP was designed to work on functions. The introduction of memory and the possibility to iterate or recurse has expanded the problems that the new GP+IM system can solve. But now we are faced with the reality that hill climbing in the space of algorithms is a much harder task, and one that GP was not designed for. The halting problem and the highly discontinuous nature of functionality in the space of algorithms were presented as two main stumbling blocks to the successful evolution of algorithms.

There are recourses. GP was not designed to tackle these problems but it can be expanded to deal well with these added complexities. Four improvements were suggested in this paper about how to improve GP+IM to handle this problems. The Popcorn method and the Anytime procedure were suggested as ways to avoid issues related to halting. It was discussed that "nearby" algorithms were not likely to be functionally similar and this violated an important implicit assumption of the traditional GP paradigm. The first proposed technique for solving this problem was to change the genetic operators used to find new algorithms so that new algorithms had a better than random change of being successful. An example was constraining mutation to change terminals and non-terminals only to closely related values or functions. The second proposal was to change the actual topology of the space so that more "nearby" algorithms had similar functionality. This was done by the introduction of the EITHER operator. This creates a type of fuzzy logic in the GP+IM system which has the effect of smoothing out the topology of the space of algorithm which, in turn, makes hill-climbing more effective.

As was mentioned in the introduction, this paper is filled with suggestions, not answers. Only good science on real problems is likely to produce even reliable rules of thumb. I feel safe in predicting, however, that the surprises that await us in the evolution of algorithms dwarf by comparison all surprises that the evolution of functions has provided so far.

REFERENCES

Hopcroft, John et al. **Introduction to Automata Theory, Languages, and Computation**. Addison-Wesley Publishing Co. 1979

Koza, John R. **Genetic Programming: On the Programming of computers by means of natural selection**. The MIT Press. 1992.

Silverman, Stephen **A Note on the Secretary Problem** IBM Research Report, IBM T.J. Watson Research Center. 1989

Teller, Astro "The Evolution of Mental Models" Chapter 9. **Advances in Genetic Programming.** editor Kinnear, Kim. MIT Press. 1994

Teller, Astro "The Turing Completeness of Genetic Programming with Indexed Memory" **IEEE World Congress on Computational Intelligence**. June 1994. (under review for publication)