Genetic Programming: Computers That Program Themselves

Robin Stewart
CSCI 373: Artificial Intelligence

May 18, 2004

1 Introduction

One goal of artificial intelligence research is to create computers that can program themselves. *Genetic Programming* is a method of automatically generating computer programs through a process analogous to biological natural selection. Through a series of generations, programs are "evolved" to solve a given problem, increasingly with human-competitive results [KBIAM96]. In this paper we start with a discussion of how genetic programming works and what the major issues are. We will then analyze several areas of current research aimed at improving the results of genetic programming. Finally, we will consider the potential of this technique in generating new knowledge.

2 Elements of Genetic Programming

The idea of immitating evolution in computers (more broadly referred to as *Genetic Algorithms*) was first concretely proposed in the 1970's by Holland as a type of search algorithm [Hol76]. The algorithm essentially works by maintaining a population of potential solutions to the problem at hand. In each iteration or *generation*, every potential solution is evaluated to determine how well it at solves the problem. The best individuals are picked out and are either "sexually recombined" by swapping parts of the strings, or taken unaltered to form the population of the next generation. The algorithm stops after a predetermined number of generations or when a suitable solution has been found [KKS+03].

Koza launched the field of genetic programming by extending genetic algorithms to handle functional computer programs represented as tree structures [Koz92]. Each function call forms an internal node of the tree, while numeric values and variables are represented as leaf nodes. For example, Figure 1 depicts a functional program that calculates $x^2 + 5$. The multiplication and addition operators form internal nodes, and the variable x and constant 5 are leaves.

In genetic programming, the solutions being evolved and evaluated are functional program trees. Consequently, in order to pose a problem in suitable form one must determine the following problem-specific elements [KKS+03]:

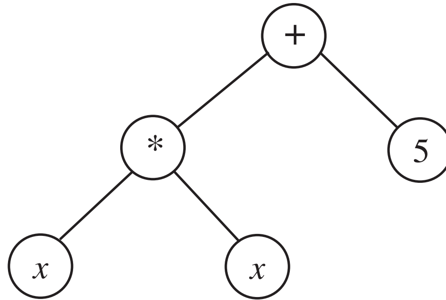

Figure 1: A program tree representing the calculation $x^2 + 5$.

1. *Terminal Set.* The set of possible leaf nodes for the problem. In the x^2+5 problem, the terminal nodes obviously end up being x and 5. But before the problem was solved, we may not have known that the solution would need 5 as opposed to some other constant, so our starting terminal set might be the set of integers between 1 and 10.

2. *Function Set.* The set of all possible functions that can be used. For instance, we might allow the mathematical operators +, -, *, and / (once again, we don't necessarily know which ones will be useful for the solution).

3. *Fitness Measure.* Evaluation begins by running each program and comparing the output with the desired output. In our example above, one way of calculating fitness is by averaging the error between the program's output and the desired output for many values of the variable x. Evaluating the fitness of a program can be a computationally intensive step, so finding heuristics that can speed up evaluation is always beneficial. Unfortunately, that is not always possible.

4. *Control Parameters.* These are a collection of parameters that affect the basic operation of the algorithm. They include the population size, the maximum number of generations, the number of "survivors" selected for the next generation, and so on. These parameters don't affect the solution population directly, but their values can affect global properties like the number of generations needed before a solution is found.

5. *Termination Criterion.* Finally, we need a way to signal for the algorithm to stop. This could be after a certain number of generations as mentioned above, or when a program surpasses a threshold fitness level.

The algorithm itself uses those elements to proceed as follows [PSV04]:

1. *Initialize* the space by randomly creating a population of program trees.

2. *Select* a subset of programs in the population, probabilistically favoring those with higher fitness. There are several ways to go about this:

   (a) Perhaps the simplest way is to just take a certain percentage off the top of the most-fit individuals. The serious drawback of this strategy is that some programs

which initially would not be in this top group (and would therefore be eliminated) might have the potential to do much better in later generations. Thus, a probabilistic method is desirable.

(b) In fitness-proportional selection, a program i with fitness f_i has a probability of being selected based on its fitness relative to the rest of the population of size n:

$$Prob(i) = \frac{f_i}{\sum_{j=1}^{n} f_j}.$$

(c) The problem with any fitness-proportional selection technique is that they are affected by every quirk of the fitness evaluation function – for instance, programs that score unreasonably well have a disproportionate chance of getting selected. To remedy this situation, *rank-based* selection ranks the individuals in order of fitness and bases the probability of selection on that rank, not the raw fitness score. Thus it uses the same formula as above, using modified fitness f_i':

$$f_i' = Max - (Max - Min)\frac{i-1}{n-1}.$$

where Max and Min define some arbitrary ranking scale.

3. The last step is to *transform* the chosen individuals to produce "offspring" programs for the next generation. The basic possibilities are:

   (a) *Reproduction*, which leaves the candidate program unchanged.

   (b) *Crossover*, which swaps a subtree in one program with a subtree in another. In some cases care must be taken that the swapped functions return the same type of value (e.g., boolean or continuous).

   (c) *Mutation*, which randomly alters part of a program (discussed further below).

4. *Repeat* with the new, transformed population until a termination state is reached.

3 Architecture-Altering Operations

A severe limitation of the basic genetic programming discussed so far is its inability to create structures like subroutines and loops which are crucial components of human-created programs. Fortunately, functional programs (programs representable as a tree structure) can handle such components, as demonstrated by LISP constructs like `defun` and `do`. The set of transformations that can create, modify and delete these structures during a genetic programming run are called *architecture-altering operations* [KIAK99]. They are all variants of mutation because the "decision" to use one is random; they differ from basic mutations in that they alter the underlying architecture of the program instead of simply modifying function and argument calls.

Subroutine creation allows a programmer to reuse the same code for different purposes instead of having to re-create it in each instance (a particularly arduous process when the

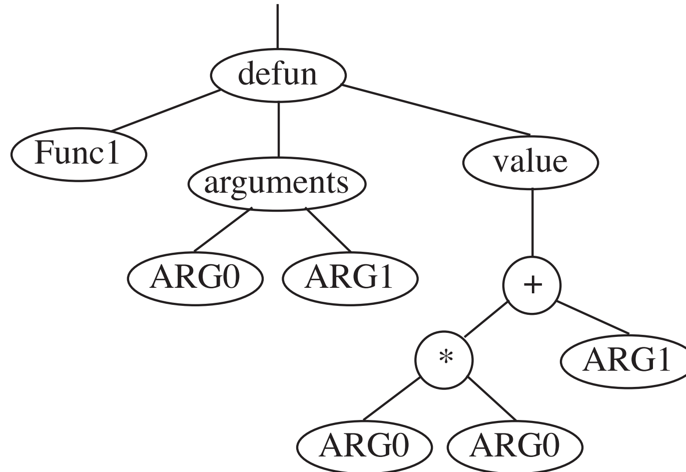

Figure 2: The old x^2+5 calculation can be generalized into a function node using architecture-altering operations.

duplicate code has to be evolved through natural selection). In genetic programming this is done with function-defining nodes (Figure 2). Each one has three branches: a function name, a list of arguments (which can be empty), and a subtree determining what calculation the function should perform. There are six architecture-altering operations that correspond to subroutines: creating, duplicating, and deleting subroutines; and creating, duplicating, and deleting arguments to a subroutine. Duplication provides the essential feature by allowing a subroutine to be used more than once [KIAK99].

It is also possible to implement loops and memory storage/access in a similarly automatic fashion. The details of designing architecture-altering operations to implement these constructs are so complicated that at first glance it hardly seems possible. In particular, these structures bring with them the unfortunate possibility of infinite (or simply time-consuming) loops, so the genetic programming system must be capable of filtering them out. The simplest way of doing so is to limit the amount of computer time or memory space allocated to evaluating each program in a population. If the program is still working on the problem when time runs out, it will be penalized accordingly when the next generation is selected. Despite the seemingly messy details, Koza et al. [KIAK99, KKS+03] and others have been remarkably successful in designing these operations.

4 Controlling Bloat

One problem that crops up in genetic programming runs is the tendency of evolved code to continually increase in size. This not only makes program evaluation more computationally intense but also produces problem solutions that are much longer than necessary. This phenomenon of *bloat* occurs for at least three reasons [Pol03]:

1. *Replication accuracy.* Having large chunks of effectively inactive code (code that does not affect the program's actual output) increases the probability that a program will re-

main functional even when modified by crossovers and mutations. Thus such programs tend to survive well in the population, especially in later generations.

2. *Removal bias.* On average, inactive code resides in lower parts of a program tree than active code, and thus inactive subtrees tend to be smaller than active ones. In the crossover operation, programs that have an inactive subtree switched out usually do better than programs with an active subtree switched out. Because of the average size difference between these subtrees, the successful programs tend to get bigger.

3. *Program search space.* Towards the end of genetic programming runs, program size does not correlate well with program fitness; that is, programs of many different sizes perform equally well. Since there are combinatorially more bigger programs than smaller programs, the probability is that more bigger programs will be selected.

Several ways of addressing this issue have been proposed, many of which are analogous to the issues of limiting the potentially infinite running time and memory requirements of candidate programs. The simplest technique is to set a program length (or program tree depth) cutoff: any program that is too long is not allowed to proceed to the next generation. A practical objection to this method is that the size of the cutoff must be determined in advance by the human question poser. Even more worrisome is the fact that programs have a better chance of moving to the next generation if they are very close to the cutoff because their offspring are likely to be discarded due to length (forcing the algorithm to use the parent instead) [Pol03].

Another possibility, the *parsimony pressure* method, decreases the fitness score of a program as its length increases. This way, an arbitrary cutoff is avoided but longer programs are less likely to get selected for the next generation. There are many variations on this general idea. One promising approach called the *Tarpeian* method chooses some subset of programs, probabilistically favoring longer programs over shorter ones. This subset is then discarded before creating the next generation. It can be proven in many cases that if the average fitness of the discarded set is better than that of the full set, the average program size in the next generation will increase; and otherwise it will decrease. Thus, the method discourages formation of bigger programs if and only if they are no more fit than shorter ones [Pol03].

5 Maintaining Diversity

Another aspect of the genetic programming population that can significantly affect results is the diversity of the various programs in the population. Once again, the reasoning is parallel to biological evolution – if the individuals in a population are highly similar, no particularly exciting new offspring will be produced. Formulated as a search issue, a highly similar population is in more danger of getting stuck at a local maximum. On the other hand, too much diversity can also be detrimental because important, functioning features will have a greater tendency to get lost.

The basic idea of diversity is easy to grasp, but looking deeper we find that there is not even a consensus on how best to define the term. One way is to compare just the *structure*

of program trees while ignoring node labels. Thus, a program that computes $x/5 - y$ would be considered the same as the program $x^2 + 5$ (Figure 1) for the purposes of the diversity measure (because the programs have the same structure). Another, completely different, possibility is to define a "distance" measure based on the number of exact (including labels) subtrees that two programs share. Still another possibility is to compare the output from running two programs rather than looking at the programs themselves [BGK04].

Globally applicable methods of enhancing diversity should apply to all of these definitions. The easiest solution is to simply increase the population size maintained in each generation. The main problem with that is the sheer computational burden – every new individual will have to be evaluated and transformed. However, the approach is extremely effective and has continued to be possible due to the exponential growth in hardware speed over the past 30 years. Another compelling possibility is inspired by biological *geographic diversity* where the separation between groups of individuals allows intra-group crossover to occur only rarely. This option has the added benefit of working very well with distributed computing: a different population can be maintained on each processor, while sharing and mixing populations periodically [BGK04].

The amount of diversity that is desirable depends on the stage of evolution. When populations are making a lot of progress towards a solution, that progress should be encouraged and captured by letting it take over significant chunks of the population. Alternately, when little progress is being made, it's better to encourage diversity in the hopes that the population is resting at a local maximum. How can we accomplish this without introducing complicated diversity calculations? Geographic diversity handles it nicely by allowing convergence within separated groups but also periodically enforcing change by mixing groups. Many other strategies are also being researched.

6 Conclusions

The use of architecture-altering operations in particular allows genetic programming to evolve solutions to a wide variety of problems with a minimum amount of specification by the human posing the question. As computational capacity increases, programs can be evolved to perform increasingly complex tasks. Genetic programming clearly does *not* guarantee that the solution it returns is optimal, but advances in controlling bloat and maintaining diversity help to insure that the solutions found are as optimal as possible. There exist logic-based automatic programming systems (e.g., [Ols95]) which in theory can return an optimal solution, but in practice such a guarantee takes far too much computational time. As a result, heuristics are added that guide the transformations of possible programs away from dead ends.

But what makes genetic programming particularly exciting is its ability to constantly make "leaps of faith" in generating new programs, rather than the incremental transformations used in logic-based systems). Crossover operations can generate new programs unlike anything seen previously in the population. It can be argued that this is analogous to the process of invention, which is usually based not just on logical reasoning but by making connections that no one has made before [KKS+03].

Indeed, already genetic programming has achieved 36 results which Koza et al. [KKS+03]

define as "human-competitive," meaning that the computer has either discovered a solution that has been previously patented, a solution that performs similarly to a previous patent, or a solution that qualifies as a newly patentable invention. The majority of these are programs which are comprised of instructions for making analog electrical circuits. Many circuit designs have been found that match the functionality of already-patented designs, and Koza et al. [KKS+03] have now *filed* patents for two new circuits invented by their supercomputer. With inevitable future increases in computing power, the breadth and ability of genetic programming will only continue to increase.

References

[BGK04] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[Hol76] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1976.

[KBIAM96] J.R. Koza, F. H. Bennet II, D. Andre, and Keane M.A. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 1–10, 1996.

[KIAK99] John R. Koza, Forrest H. Bennett II, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[KKS+03] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, 2003.

[Koz92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[Ols95] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, 1995.

[Pol03] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming, Proceedings of the 6th European Conference*, pages 204–217, 2003.

[PSV04] J. Potvin, P. Soriano, and M. Vallee. Generating trading rules on the stock markets with genetic programming. *Computers and Operations Research*, 31(1):1033–1047, 2004.