Introduction to Genetic Programming

Matthew Walker

October 7, 2001

1 The Basic Idea

Genetic Programming (GP) is a method to *evolve computer programs*. And the reason we would want to try this is because, as anyone who's done even half a programming course would know, computer programming is hard.

Automatic programming has been the goal of computer scientists for a number of decades. Scientists would like to be able to give the computer a problem and ask the computer to build a program to solve it. Genetic programming, it has been said, shows the most potential as a way to automatically write computer programs.

So there is an amount of hope for GP's role in future computing, but what use is it now? Most research focuses on how we could improve the GP process, however there are commercially available genetic programming kernels that allow people to apply the technique.

This paper will look at the basics of genetic programming: theory and examples.

2 A More Detailed Look

Genetic Programming, one of a number of evolutionary algorithms, follows Darwin's theory of evolution—often paraphrased as "survival of the fittest". There is a population of computer programs (individuals) that reproduce with each other. Over time, the best individuals will survive and eventually evolve to do well in the given environment.

Before we can apply genetic programming we need a problem to solve. For this discussion we'll consider a variation of robotic soccer[1].

Normally, robotic soccer is played with three to five robots per team on a field about the size of half a table-tennis table. The robots are quite small (many fit within a 75mm-side cube) and they chase around an orange golf ball by bunting it with the front of their chasis. It would be quite possible to use genetic programming to "design" a strategy for an entire team to play soccer.[2]

[1] See `www.fira.net` and `www.robocup.org` for more information.

[2] This was successfully executed by Luke, S., Hohn, C., Farris, J., Jackson, G. and Hendler, J. 1998. Co-evolving Soccer Softbot Team Coordination with Genetic Programming in Kitano

However, before a soccer-playing team could evolve, it would be important for the robots to “know” how to follow the ball. It will be this skill that will be used as a problem domain for this discussion: how can genetic programming be used to evolve code so that a robot will follow the ball.

Now that we are armed with a problem we need the ability to rate an individual’s performance. Good ball-following somehow needs to be measured. This measure is called a *fitness test* and produces a value called *raw fitness*.

If we have the ability to create a random program that has some amount of control over the robot, then we’re in a position to start the evolutionary process.

We first need a population of (say, 5000) individuals that are random programs. We could now evaulate every individual (using the fitness test) and give it a rank. Good individuals, that follow the ball closely, would rank highly. Next, these individuals would reproduce with one another to produce a new population. If reproduction is biased so that highly ranked individuals reproduce more frequently, then the average performance of the new population will, almost certainly, be better than the average of the previous population. It could also be expected that the best individual in the new population would be better than the best individual of the previous population.

We could now evaulate every individual in the new population and rank each one. Again we could produce a new population by reproduction that was biased towards individuals that were more highly ranked. Over time the population would improve. Finally, after sufficient generations, the best individual in the last population would be an excellent solution to the problem.[3]

It is this process that is termed genetic programming. It could be used to evolve computer program solutions for any problem domain so long as individual solutions can be compared and ranked. The massive disadvantage of GP is the phenomenal computing resources required before any real-world problem can be tackled.

3 The Nitty Gritty

3.1 Creating an Individual

The fundamental elements of an individual are its genes, which come together to form code. An individual’s program is a tree-like structure and as such there are two types of genes: *functions* and *terminals*. Terminals, in tree terminology, are leaves (nodes[4] without branches) while functions are nodes with children. The function’s children provide the arguments for the function. In the example (see Figure 1) there are two functions (+ and $\times$) and three terminals (3, x and 6). The $\times$ (multiplication) function requires two arguments: the 3 and the return value of the subtree whose root is +. The + (addition) function also

H. (ed.) *RoboCup-97: Robot Soccer World Cup I.* Springer. Pages 398-411.

[3]Due to the random nature of GP, this is not guaranteed. In fact, even the evolution of an acceptable solution cannot be taken for granted.

[4]*Nodes* in genetic programming terms are often referred to as *points* on a tree.

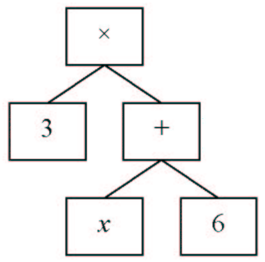

Figure 1: A typical, albeit simple, individual that returns $3(x + 6)$.

takes two arguments which are provided by the x and the 6. The example may be interpreted as $3 \times (x + 6)$.

The genes that will be available to the GP system must be selected or created by the user. This is an important decision as poor selection may well render the system incapable of evolving a solution.

3.2 Creating a Random Population

Assuming the genes have been chosen, the first step is to create the random population. Koza[5] specifies three techniques, namely *grow*, *full* and *ramped-half-and-half*. He also specifies that the initial population will not have duplicate individuals.

3.2.1 Grow

With this first technique the entire population is created by using the grow method which creates one individual at a time. An individual created with this method may be a tree of any depth up to a specified maximum, m.

1. Starting from the root of the tree every node is randomly chosen as either a function or terminal.
2. If the node is a terminal, a random terminal is chosen.
3. If the node is a function, a random function is chosen, and that node is given a number of children equal to the arity (number of arguments) of the function. For every one of the function's children the algorithm starts again, unless the child is at depth m, in which case the child is made a randomly selected terminal.

This method does not guarantee individuals of a certain depth (although they will be no deeper than m). Instead it provides a range of structures throughout the population.

[5] John Koza, the person who coined the term "genetic programming", is considerably well renowned in this field. His publications almost define canonical GP.

The method may produce individuals containing only one (terminal) node. Such individuals are quickly bred out if the problem is non-trivial, and so are not really much value. The example individual (Figure 1) could have been produced by the grow method.

3.2.2 Full

The full method is very similar to the grow method except the terminals are guaranteed to be a certain depth. This guarantee does not specify the number of nodes in an individual. This method requires a final depth, d.

1. Every node, starting from the root, with a depth less than d, is made a randomly selected function. If the node has a depth equal to d, the node is made a randomly selected terminal.

2. All functions have a number (equal to the arity of the function) of child nodes appended, and the algorithm starts again. Thus, only if d is specified as one, could this method produce a one-node tree.

3.2.3 Ramped-half-and-half

To increase the variation in structure both grow and full methods can be used in creating the population—this technique, ramped-half-and-half, is the sole method used by Koza. Only a maximum depth, md, is specified but the method generates a population with a good range of randomly sized and randomly structured individuals.

1. The population is evenly divided into parts: a total of $md - 1$.

2. Half of each part of the population is produced by the grow method. The other half is produced using the full method. For the first part, the argument for the grow method, m, and the argument for the full method, d, is 2. For the second part 3 is used. This continues to part $md - 1$, where the number md is used. Thus a population is created with good variation, utilising both grow and full methods.

3.3 Fitness Test

Once the initial random population has been created, the individuals need to be assessed for their fitness. This is a problem specific issue that has to answer the question “how good (or bad) is this individual?” In the robotic soccer domain, the question could be replaced with an algorithm like: let the individual play soccer for five minutes; the fitness value is the same as the number of goals scored. This could be an acceptable test as it can distinguish between different levels of ability—a good player may score six goals, while an excellent player may score ten. It is important that the fitness test gives as detailed an answer a possible, thus a test that answered “yes, at least one goal was scored” or “no, zero goals were scored” would not be appropriate. This problem specific fitness value is given the term raw fitness—in this case, the greater the value the better the individual.

For a ball-follower a good fitness test could be measured by the sum of the distances between the robot and the ball for a number of discrete times. In this case a smaller value indicates a better individual.

3.4 The Genetic Operations

Having applied the fitness test to all the individuals in the initial random population, the evolutionary process starts. Individuals in the new population are formed by two main methods: *reproduction* and *crossover*. Once the new population is complete (i.e. the same size as the old) the old population is destroyed.

3.4.1 Reproduction

An asexual method, reproduction is where a selected individual copies itself into the new population. It is effectively the same as one individual surviving into the next generation. Koza[6] allowed 10% of the population to reproduce. If the fitness test does not change, reproduction can have a significant effect on the total time required for GP because a reproduced individual will have an identical fitness score to that of its parent. Thus a reproduced individual does not need to be tested, as the result is already known. For Koza, this represented a 10% reduction in the required time to fitness test a population. However, a fitness test that has a random component, which is effectively a test that does not initialise to exactly the same starting scenario, would not apply for this increase in efficiency. The selection of an individual to undergo reproduction is the responsibility of the *selection function*.

3.4.2 Crossover

Organisms' sexual reproduction is the analogy for crossover. Crossover requires two individuals and produces two different individuals for the new population. In this technique genetic material from two individuals is mixed to form off-spring. Figure 2 describes the process. Koza uses crossover on 90% of the population—it is the more important of the two methods because it provides the source of new (and eventually better) individuals.

There are a few other evolutionary operations: *editing*, *mutation*, *permutation*, *encapsulation* and *decimation*, most of which were ignored by Koza in his earlier work but have recently been given more consideration.[7]

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

[7]Koza J. R., Bennett, H. B., Andre, D. and Keane, M. A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers.

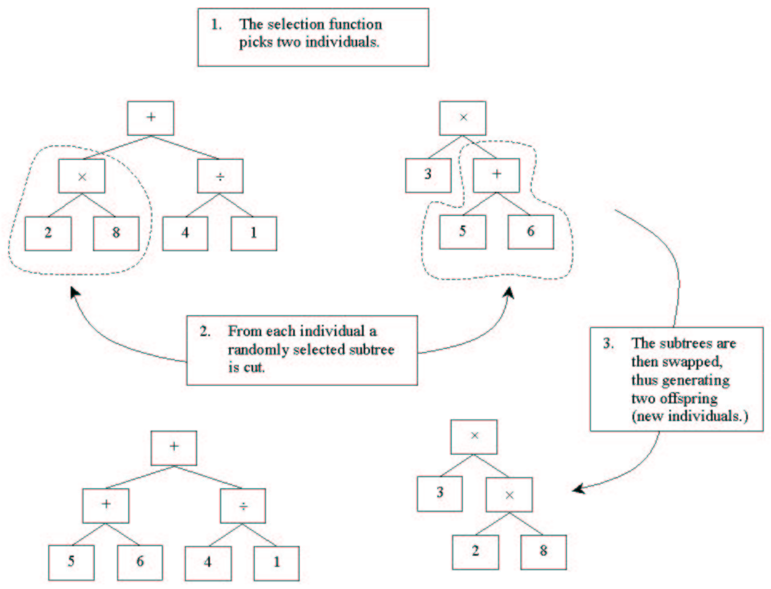
1. The selection function picks two individuals.
+
×
÷
2
8
4
1
×
3
+
5
6
2. From each individual a randomly selected subtree is cut.
3. The subtrees are then swapped, thus generating two offspring (new individuals.)
+
+
÷
5
6
4
1
×
3
×
2
8

Figure 2: This genetic operation is termed two-offspring crossover.

3.5 Selection Functions

These methods would result in only a random search if it were not for the selection function. Koza uses a number of selection functions-with fitness-proportionate selection, greedy over-selection, and tournament selection being the most common. The implementation for this report utilised only the first two methods.

3.5.1 Fitness-Proportionate Selection

With fitness-proportionate selection individuals are selected depending on their ability compared to the entire population, thus the best individual of a population is likely to be selected more frequently than the worst. The probability of selection is calculated with the following algorithm:

1. The raw fitness is restated in terms of *standardised fitness*. A lower standardised fitness value implies a better individual. If the raw fitness increases as an individual improves then an individual's standardised fitness

is the maximum raw fitness (i.e. the fitness of the best individual in the population) minus the individual's raw fitness. If the raw fitness decreases as an individual improves, standardised fitness for an individual is equal to the individual's raw fitness.

2. Standardised fitness is then restated as *adjusted fitness*, where a higher value implies better fitness. The formula used for this is:

$$adj(i) = \frac{1}{1 + std(i)}$$

   where $adj(i)$ is the adjusted fitness and $std(i)$ is the standardised fitness for individual i. The use of this adjustment is beneficial for separation of individuals with standardised fitness values that approach zero.

3. *Normalised fitness* is the form used by both selection methods. It is calculated from adjusted fitness in the following manner:

$$norm(i) = \frac{adj(i)}{\sum_{k=1}^{M} adj(k)}$$

   where $norm(i)$ is the normalised fitness for individual i, and M is the number of individuals in the population.

4. The probability of selection (sp) is:

$$sp(i) = \frac{norm(i)}{\sum_{k=1}^{M} norm(k)}$$

   This can be implemented by:

   (a) Order the individuals in a population by their normalised fitness (best at the top of the list)

   (b) Chose a random number, r, from zero to one.

   (c) From the top of the list, loop through every individual keeping a total of their normalised fitness values. As soon as this total exceeds r stop the loop and select the current individual.

3.5.2 Greedy Over-Selection

To reduce the number of generations required for a GP run, Koza made use of greedy over-selection. Again, individuals are selected based on their performance but this method biases selection towards the highest performers.

Every individual is assessed, and their normalised fitness value calculated.

1. Using the normalised fitness values, the population is divided into two groups. Group I includes the top 20% of individuals while Group II contains the remaining 80%.

2. Individuals are selected from Group I 50% of the time. The selection method inside a group is fitness-proportionate.

3.6 User Decisions

The user must make a number of decisions before the GP system may begin. Firstly, the available genes need selecting and creating. Secondly, the user must specify a number of control parameters. The decisions are critically important as they have a limiting effect on the search space of possible programs. Too great a limit may remove all chance of evolving an acceptable individual. However, not restricting the search-space sufficiently has its own issues.

The selection of genes to perform a task is important. If it is not possible to solve the problem given the available genes, then what hope has the system got to evolve a solution? It is important to include every necessary function and terminal. However, selecting more than the minimum required set of genes poses little problem as the genes that are not useful will be bred out of the population.

The control parameters that need to be set are:

1. Population size: A larger population allows for a greater exploration of the problem space at each generation and increases the chance of evolving a solution. In general, the more complex a problem the greater the population size needed.

2. Maximum number of generations: The evolutionary process needs to be given time; the greater the maximum number of generations the greater the chance of evolving a solution. However, further evolution of a population does not guarantee a solution will be found—it may be better to start again with a different initial population. So if, after a user-defined number of generations, a sufficiently successful individual has not evolved then the process should halt.

3. Probability of crossover: What proportion of the population will undergo crossover before entering the new population? [Koza, 1992] does not change this value from 0.90—90% of the population undergoes crossover.

4. Probability of reproduction: The proportion of individuals in a population that will undergo reproduction. Throughout Koza's work this value stays constant, at 0.10—10% of the population undergoes reproduction.

Koza lists a number of other minor control parameters that have not been discussed here.

3.7 Computing resources

Genetic programming uses a phenomenal amount of processing time even for apparantly simple problem domains. On a Pentium 133 Mhz, the ball-following problem took almost 40 hours hours of computation—and the evolved solution was anything but ideal.

The major consumers of processing power are the fitness tests. Over 50 generations for a population of 5000, 250,000 fitness tests must be executed. But a population size of 5000 is *very* small: In 1999 Koza's work commonly used populations greater than 600,000 individuals! And it was not uncommon for runs to take more than 100 generations.

Given these figures, it is drastically important that the fitness testing of individuals be as efficient as possible. However for complex problem domains quick fitness tests are frequently implausible. With this in mind, parallelisation of the genetic programming paradigm has received significant effort.

4 Further Reading

If you would like to know more about genetic programming, I would recommend the following resources:

- Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press.

- Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press.

- Koza J. R., Bennett, H. B., Andre, D. and Keane, M. A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers.

There is also considerable information on the Internet. An excellent starting point would be *EvoWeb* at:
`http://gargantia.cs.uni-dortmund.de/index.html`.
A list of Internet-based resources can also be found at:
`http://www.massey.ac.nz/∼mgwalker/gp/index.html`.