Student Projects

Genetic Programming

Todd Chaffins

Todd Chaffins is currently a Ball State senior with a major in Computer Science and minors in Mathematics and Physics. As a junior he participated in the Honors Undergraduate Fellowship program which enabled him to research genetic programming. Dr. Rich Stankewitz was his advisor for the fellowship.

As personal computers become more powerful and ubiquitous, they are called upon to perform and solve an ever increasing number of tasks and problems. In order to solve these problems or perform tasks a computer must execute a computer program. The act of writing such a computer program can be very complex and difficult. Under the pressure of an ever-growing need for computer programs, genetic programming/evolutionary computing emerged as a promising method in which the computer itself performs this often daunting task of writing software.

Genetic programming [2] is an attempt to create either computer programs or solutions to a given problem by a process which mimics evolution in nature. The fundamental principle of the theory of evolution is "survival of the fittest." This principle essentially explains why certain species survive and others do not. The organisms of a generation whose traits are most conducive to survival and reproduction typically survive and reproduce. During reproduction, the next generation of organisms is created from the parent organisms by combining different portions of each parent's DNA. The offspring's genetic material is formed in a process called recombination or crossover. During the creation of the offspring's DNA, the DNA of one parent is copied for use in the offspring. At a certain point in the genetic material there is a crossover where the source for the offspring's DNA switches to the other parent. Frequently crossover occurs multiple times in the creation of the offspring's DNA.

Another less common way the genetic material is changed in the evolution of the generations is mutation. Mutation is a random change of the DNA, which in nature is usually caused by an error in copying the genetic material. Usually mutation is harmless and has little effect on the resulting organism.

Sometimes, however, it can give an organism an advantage or disadvantage. For example, the result of an extreme mutation in a fish could be the growth of a new appendage which gives off light. This light could draw prey near allowing the fish to feed more easily or it could draw predators near decreasing the fish's chances of survival. If the mutation makes the organism more fit with respect to the environment, then it survives and the mutation survives as well. The impact of the mutation is determined by the environment.

To illustrate crossover and mutation in the context of language, imagine the DNA of an organism as a sentence. Crossover of the text would involve swapping of text between the sentences. For example:

> Given the following sentences as the parent DNA:
>
> Parent 1: The car is fast.
>
> Parent 2: My bike is slow.

The following children could result from a single crossover at the eighth character:

> Child 1: The car is slow.
>
> Child 2: My bike is fast.

It is easy to see the drastic effect crossover can have. Mutation can have an equally impressive effect on the DNA. For Example:

> Given the following sentence as the parent DNA:
>
> Parent: The car is fast.
>
> The following sentence could result from a single mutation at the seventh character:
>
> Child: The cat is fast.

In genetic programming one can think of a computer program as an organism. Consider, for simplicity, the task of programming a computer to play flawless tic-tac-toe. A flawless tic-tac-toe program is not very challenging to write and can be done fairly simply since a perfect playing strategy is known. For the sake of this example, we will pretend that we do not know how to program flawless tic-tac-toe. Instead of writing the program to perform the given task by coding it directly, we illustrate how one might use genetic programming to have a computer write the program for us.

First we must understand what a tic-tac-toe playing program is. A tic-tac-toe playing program allows a computer to play tic-tac-toe and as such, given any possible configuration of the board, this program must produce a move which places an X or O on the board. Our goal is to use genetic programming to ultimately create a program which behaves in an intelligent manner.

The first step is to randomly create a first generation of programs for playing tic-tac-toe. Let us consider how to create a single program. For every input (C, M) where C is the configuration of the board and M is the "mark" (X or O) of the computer, the program must select an output, i.e., a placement of M into a vacant spot on the board. Let us denote the output by the function notation

$Y(C, M)$. A randomly generated program then means that for a given input (C, M) we randomly assign an output (from the vacant spots on the board). Once we do this for all possible inputs we have a tic-tac-toe playing program. Note, however, that this program is deterministic, i.e., a function, and thus, given the specific input (C, M), the program will *always* produce the same output $Y(C, M)$. The choice of output was determined at random, but once this is set, this program must always produce the same output. So, this is not a program that plays at random, but rather a program that was created by random means, and, once created, does not involve any randomness in its choice of moves.

The first step in the genetic programming process is to produce a generation of, say 50, such randomly created programs. One will note that it is the human programmer's job to determine the sample space from which the first generation is extracted. In the tic-tac-toe example the sample space was determined to be the space of all tic-tac-toe playing programs. In more general situations the human programmer may impose more or less structure on the sample space in an effort to better determine the end result.

Each program in this initial generation is then evaluated by what is called a fitness function, which gauges how well each program performs the given task. For a tic-tac-toe playing program the fitness would be determined by how well it plays tic-tac-toe, taking into account not just whether or not the program won or lost, but also how many moves it took to win or lose. For example we might let the program play 500 games, and then produce a fitness value by means of a weighted average of the total wins, losses, and number of moves. Just as an organism's survival is determined by its fitness in the environment, a program's survival is determined by its fitness as evaluated by the fitness function. If a solution has a high fitness, that solution, or at least many of its significant attributes, will survive to the next generation.

The next generation is created in several ways in genetic programming. A solution that has a very high fitness may carry over through reproduction completely unchanged to the next generation, while others with high fitness values will be combined through analogous processes of crossover and mutation to create new programs. For the tic-tac-toe problem a mutation would result in a program (function) which is identical to the original except it would change the function output corresponding to a specific input (C_0, M_0). Crossover could produce a much greater change in the tic-tac-toe problem and would result in programs which have swapped entire sections of their game-playing strategy. For instance, programs Y and Y' could swap all outputs when the board has three open spots. Thus given a generation of programs $Y_1, Y_2, \ldots, Y_{50}$, the next generation would be produced by reproduction, mutation and crossover, which occur at rates that the human programmer sets as parameters at the beginning of the genetic programming process.

The next generation created by this process of reproduction, mutation and crossover is then likely to be composed of individuals which on average are more fit to the environment or, in this example, are programs better at playing tic-tac-toe. This new generation then undergoes the same process as the first generation with the end goal being to create more fit solutions. The evolution-

ary process continues on for many generations until it is terminated manually after a preset number of generations, or it is terminated automatically when a solution is found reaching a certain preset fitness level.

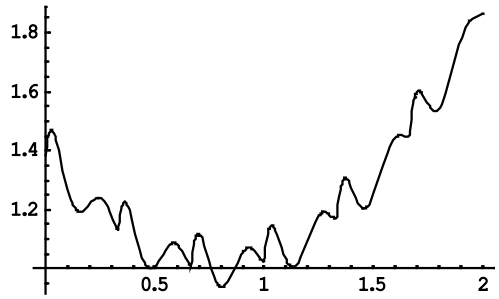
The important facet of this approach is that the human programmer does not need to devise the program's algorithm. He/she only needs to perform the much simpler tasks of devising both an appropriate fitness function and a sample space of programs that can be created by the genetic programming algorithm. The process of genetic programming will then create the program for him/her. While the evolutionary approach seems a bit overly complex in our example of tic-tac-toe, the very same process could be used to approach such formidable and unsolved problems as finding a strategy for flawless chess. It is this distinction between devising the solution directly and creating a fitness function and sample space, that makes evolutionary computing so attractive. Even the most complex of problems can be distilled into relatively simple fitness functions with the evolutionary approach. With genetic programming, the computer essentially learns how to solve the problem for the programmer.

Genetic programming has already been used by researchers to produce many nontrivial results. Examples of these results include a soccer-playing program and the design of several important electrical circuits whose designs are competitive and in some cases better than human-created solutions. The aforementioned soccer-playing program took part in a competition called Robo Cup. The other soccer-playing programs in the Robo Cup competition were written directly by humans. The soccer-playing program created by evolutionary computing placed ahead of about half of the human created algorithms [1]. This soccer-playing program and many other programs like it have shown that genetic programming can create viable solutions to problems.

Working with Dr. Stankewitz, I was able to research genetic programming as a participant in Ball State University's Honors College Undergraduate Research Fellows program which provided funding for one semester of research. The goal for my research in genetic programming was to implement a genetic programming library (a collection of functions which allow populations of programs to be created, evaluated, and a new generation of programs to be generated) and to be able to use that library to solve some problem. It frees the programmer from the details of the genetic programming algorithm and thus allows the programmer to concentrate only on creating a fitness function, deciding on an initial pool of operators and terminals (which determine the sample space), and setting the genetic programming control parameters. Using the library one can exploit the power of the genetic programming paradigm using a very small amount of C^{++} code.

Once the library was created and properly tested, it was used to solve a problem of function minimization. Figure 1 shows the graph of a function which we want to minimize. The program created was able to repeatedly find the global minimum of the function, provided with an accuracy of six decimal places in less than one hundred generations. It is important to note that the program does not use extensive math knowledge or techniques and as such is applicable to a wider class of functions (e.g., functions are not required to be differentiable). The individuals, which in this case consisted of an x-value,

Figure 1: A graph of the function for which the genetic program found the global minimum



were judged in the fitness function based on how low their corresponding y-value was in comparison to the other individuals' y-values. The individuals were represented as binary strings for which crossover was the swapping of sub-strings and mutation (of a bit in the string) was a "bit flip" from zero to one, or one to zero. The minimization program was created using a genetic programming library with less than fifty lines of code and is applicable to any function which has a global minimum. While the program is applicable to a wide range of functions and produced repeatable results, there is no guarantee (due to the probabilistic nature of the genetic programming algorithm) that the program will find a minimum for every function. It will, however, "eventually" find a minimum if given enough time, and it generally does so efficiently as compared to other algorithms.

With the library created through this research I hope to apply genetic programming to a wide variety of problems. One of the advantages of genetic programming is its ability to be applied with little or no modification to completely different types of problems. One such problem I plan to explore is to find an algorithm for ray-triangle intersection. While this is a problem that has been solved before, the existing algorithms are still too slow for certain possible uses in computer graphics and simulations. With the flexibility of genetic programming it might be possible to implement a genetic program to produce a new and more efficient algorithm. I believe that the ease of use and widespread applicability will make genetic programming an important technique for problem solving in the future.

References

[1] D. Andre and A. Teller, Evolving Team Darwin United. In Asada, Minoru and Kitano, Hiroaki (editors), RoboCup-98: Robot Soccer World Cup II. Lecture Notes in Computer Science **1604** Springer-Verlag (1999) 346–352.

[2] J. Koza, Genetic Programming: On the programming of computers by means of natural selection (complex adaptive systems), MIT Press (1992).