GPQuick Article

This is copied from Byte Magazine, February, 1994.

Mutating computer programs can evolve superior methods for solving loosely defined problems

Andy Singleton

Could a computer create better software than you could? Could 100,000 computers do the job? Can any purely mechanical process match the creative power of the human mind? Consider natural evolution. You may design a better mousetrap, but evolution, by the simple mechanical process of blind trial and error, can design a cat.

This is the challenge of genetic programming (GP for short)--to evolve a solution that is as elegant and sophisticated as a cat on the hunt. With an improved understanding of evolution and with the power of massively parallel computers, this goal is in sight. In this article I describe GPQUICK, a simple GP system in C++.

GP is one application of an important new technology known as genetic algorithms (or GAs). GAs can evolve a variety of computer-based objects and solutions. They are used for optimization--producing a better route map, a more efficient manufacturing schedule, or a more cost-effective alloy mixture. They can be used for interactive evolution: teamwork between a person and a machine to design pictures, buildings, or machines. And they can provide a superior method of computer learning for speech, vision, financial trading, fault detection, and so on.

How Does a GA Work?

A GA is a simple tactic for computer learning that is inspired by natural evolution. The computer produces a whole population. It then picks the best "individuals" and changes them, producing a new population of variations on the best individuals, with the hope that some of the new individuals will be even better. If this simple process is carried on for many generations, the results can be impressive.

A GA is simple; it is just a form of trial and error, with two elements making it more effective. The first is that it tries out variations of only the best individuals, making it more likely to be trying something good. The second is that it builds a good solution piecewise. It uses crossover (or mating) to combine the good parts of one individual with the good parts of another.

A genetic algorithm consists of three operations: evaluation, selection, and reproduction. Evaluation is the process of assigning a fitness score so you can find the best individuals. You need to make a fitness function that scores the output from each individual. Next, you select the individuals to be used as parents for the next generation. An automated GA usually picks the individuals with the highest fitness scores. Last, you reproduce those parents with genetic operations to generate new and possibly better individuals in the next generation.

The two main classes of genetic operations are mutation and crossover. Mutation operates on a single parent by randomly changi ng some part of it. For instance, a 1 might be changed to a 0, a number might be changed to another number, or a function might be changed to a different function. This operation is analogous to mutation of genes, in which the code for one amino acid changes to the code for a different amino acid. Crossover acts by combining parts of two parents. It is analogous to mating in biological organisms.

How Can a GA Be Used for Programming?

If you can define selection, mutation, and crossover operators, you can use a GA to generate almost anything. GP is simply the application of a GA to generate computer programs. Numerous computer languages or representations have been tried, with varying degrees of success.

In 1975, John Holland, the inventor of GAs, proposed a form of GP known as a classifier system. Classifiers are IF...THEN rules, such as "If you fall down, then cover your face." These rules can be represented as strings of 1s and 0s. For instance, a bit could represent whether you (1) are or (0) are not falling down, and another bit could represent whether or not you should cover your face. A GA can evolve these strings by testing them for effectiveness under a variety of circumstances and then reproducing the most effective ones with mutation and crossover.

In 1985, Nicheal Kramer proposed evolving computer program code directly and demonstrated the evolution of simple arithmetic expressions. He also proposed a tree representation with the special form of crossover used in GPQUICK.

John Koza demonstrated the evolution of Lisp expressions that are actually computer programs in 1987. He started using the term genetic programming to describe this process. In 1989 he applied for a patent on the use of GP for evolving a wide variety of applications. Like many software patents, it is controversial. Koza has written a book called Genetic Programming: On the Programming of Computers by Means of Natural Selection & Genetics (MIT Press, 1992) that details numerous GP experiments and provides practical advice.

S-expressions, more commonly known as Lisp expressions, have become the mainstream GP representation. Researchers are investigating other GP representations, including object structures, direct evolution of machine code, and the design of processors and machine languages that would be amenable to GP.

S-Expressions Make You Lisp

The form of GP that Koza described and GPQUICK uses generates S-expressions. An S-expression consists of a function followed by zero or more arguments. Each argument is in turn also an S-expression. You can represent "2+2" as (ADD 2 2), and "2+3*5/2" as (ADD 2 (MULTIPLY 3 (DIVIDE 5 2))).

A function in an expression is called a node. Functions with no arguments, such as numeric constants, are called terminals. All functions return some value. Functions can also perform actions, otherwise known as side effects. You could have functions like (TURN_RIGHT), (TURN_RIGHT number_of_degrees), and so on, which give S-expressions procedural abilities.

The PROG function can string actions together into a procedural program, such as (PROG GO_FORWARD GO_FORWARD TURN_ RIGHT). If you add conditionals such as (IF condition do_this otherwise_do_that), looping constructs such as (WHILE condition do_this) or (FOR count do_this), and a memory array with (SET element value) and (GET element), you have a Turing complete programming language that can represent any structured program.

Achieving Closure

To do GP, you chop a piece off one program and stick it together with a piece of another program, on the off chance that something good may result. If you tried this with one of your C++ programs, you'd have approximately a 0 percent chance of creating a working program. With S-expressions, though, you can rig the game so that you have a much higher probability of success.

You design your functions so that they all return the same argument type, usually a floating-point number, and they all accept arguments of this type. This situation is called clos ure. Now you can use any expression as an argument for any other expression. You can swap a subexpression from one program for a subexpression in another program and always end up with a syntactically valid program. It might be completely useless, but it will never send smoke billowing out from under the CPU.

Crossover

GPQUICK uses subtree crossover. First you select two parents. Then you pick a node, any node, from the first parent. This node is, by definition, the beginning of some complete subexpression. If you wrote the expression out as a parse tree, the subexpression would be a branch or subtree. You extract this, leaving a hole in the program. You then pick a node in the second parent, extract the following subexpression, and swap it with the first subexpression.

For example, I could cross (ADD 2 (MULTIPLY 3 (DIVIDE 5 2))) with (ADD (MULTIPLY 7 11) 23). In the first parent, I pick the fourth node, 3. In the second parent, I pick the second node, or (MULTIPLY 7 11). Swapping, I end up w ith (ADD 2 (MULTIPLY (MULTIPLY 7 11) (DIVIDE 5 2))).

Mutation

Any random change to a program qualifies as a mutation. There are many types of changes that you can make to an S-expression and still end up with a syntactically valid program. For simplicity, I have included in GPQUICK one type of mutation that picks a random function and changes it to a different function with the same number of arguments.

Applying this mutation to (ADD 2 (MULTIPLY 3 (DIVIDE 5 2))), I select the third node, MULTIPLY, for mutation. I can change it to any two-argument function. I select ADD at random, ending up with (ADD 2 (ADD 3 (DIVIDE 5 2))). To mutate a terminal, you would substitute another terminal.

The GP Code and the Incredible Shrinking Interpreter

GPQUICK uses a representation for the GP code called linear prefix jump table, which was suggested to me by Mike Keith. Programs coded this way are small and evaluate quickly. When you evaluate populations of several thousand programs, size and speed are very important.

The GP code is an array of 2-byte structures of type node, defined by the following:

```
typedef struct {

unsigned char op;

unsigned char idx;

} node; // node type
```

Structure member op (for op code) is a function number. The idx is an immediate operand that can represent a table or a constant index. The GP code is stored in an object called a Chrome, short for chromosome.

In GPQUICK, functions are represented by objects of class Function. All the functions are stored in the array funcarray and given an array index. You could install the functions NUMBER (to return numeric constants), ADD, SUBTRACT, MULTIPLY, and DIVIDE as functions 0, 1, 2, 3, and 4. Function NUMBER uses the idx to determine which number to return. Other functions do not use idx and leave a zero value.

The expression (ADD 2 (MULTIPLY 3 (DIVIDE 5 2))) would be represented in our GP machine language as {(1,0) (0,2) (3,0) (0,3) (4,0) (0,5) (0,2)}, where the first element (1,0) is an ADD (op code 1) node, and the second (0,2) is a NUMBER (op code 0) node returning the value 2.

Our Chrome method to evaluate the function at position ip in the code stack looks like this:

```
ip++;return funclist[expr[ip].op] -> eval(this);
```

This calls the appropriate Function eval method. The Func-tion eval method calls the Chrome eval method recursively to get its arguments. The eval method for function ADD is as follows:

```
return chrome->eval() + chrome->eval();
```

The eval method for function NUMBER is

```
return chrome->expr[ip].idx;
```

That's all there is to the interpreter. Add functions to suit your taste.

GPQUICK Architecture

The object-oriented architecture of GPQUICK is divided into three pieces: the Chrome and Function classes, which are the underlying program representation; the Problem class, which holds the objectives, functions, and data necessary to define and solve a particular GP task; and the Pop (short for population) class, which holds a populatio n of Chromes and runs a GA on them, calling the fitness function in the Problem to do an evaluation. This three-part architecture has worked well for me in a variety of applications.

Obviously, there is a tremendous variety of potential Problems. Also, many different kinds of GAs can be associated with subclasses of Pop, each with different methods of selection, a different mix of genetic operations, and a different topology for the population (e.g., single, parallel, or distributed processors). As long as they share the same Chrome representation, most Problems can run with most Pops and GAs.

The Chrome class is the carrier of a genetic program. A Function object carries information that the Chrome needs for initializing, displaying, and evaluating expressions that contain that function. Many functions, such as the arithmetic functions, are problem-independent; however, some functions, such as data terminals, require the data structures and evaluation environment of a particular problem. The ta ble "Chrome and Function Classes" lists important data structures and member functions.

Pop carries an array of Chromes, evaluates them with the fitness function provided in the current Problem, and performs a GA to generate new and better Chromes. Pop uses a steady-state GA, generating one new Chrome and replacing one old one at a time, as opposed to making a whole new batch as a generation.

The generate method is the agent that selects one or more parents and applies mutation or crossover. This virtual method could be replaced by code with different strategies for selection, reproduction, or interaction with parallel populations. (The important data structures and member functions are listed in the table "Pop and Problem Classes.")

To do GP, you must have a fitness function that defines the problem and a set of primitive operations sufficient for solving it. Any implementation of GPQUICK must have a subclass of Problem that includes the appropriate fitness function and primitive functio ns.

Most programs need to be evaluated in a particular context. If you're trying to evolve a program that can navigate a maze, you need to define the maze. Setting up the environment should be done in the Problem constructor and at the beginning of the fitness function.

Is SSRProb Useful?

The default problem for GPQUICK attempts to evolve an arithmetic formula matching John*(George-Paul)+2.5* Ringo. This sort of problem is known as symbolic regression. Using symbolic regression to discover a function that you already know is not useful, but you can modify SSRProb to learn prediction and classification from real-world examples. This is called genetic induction, and it has several advantages compared to competing statistical or neural techniques.

To do genetic induction, set up a training database that includes the value you want to predict and fields for each of the input variables in your model. Load the training database into a memory array. Add a static variable for the current recor d number. Add terminals that return the input field values for the current record. In the fitness function, evaluate for a set of randomly chosen records, and calculate the error compared to the desired value. The resulting expression will hopefully return accurate values for new input examples. I use a souped-up version of genetic induction to evolve traders and list selectors.

The Future of GP

GPQUICK can produce arithmetic expressions, but it will never produce an operating system. Its successors may.

The trend to use more and faster processors will increase GP capabilities. GP consumes huge amounts of processor time evaluating millions of candidate programs. As the programs' behavior becomes more sophisticated, time spent on evaluation rises.

Fortunately, GP is ideally suited to massively parallel computers of almost any design. Just as evolution can proceed among a population that is spread out over a continent, so GP can proceed with populations that are distributed among a multi tude of processors, with good individuals traveling occasionally from one processing neighborhood to the next for interbreeding. The Creation Mechanics GP toolkit supports desktop supercomputing, which links a network of PCs into a population for parallel processing. GP may well become a core application for the next generation of parallel computers.

As understanding of evolution and the mechanics of creation improves, so will GA effectiveness. We are working to understand specific problems; a basic problem concerns finding the right fitness function and environment for rapid evolution. We are also investigating the evolution of evolvability--finding the types of structures that respond well to mutation and crossover. Perhaps the most important question concerns modularity and hierarchy: how to build up the organs and cells, subroutines and data structures, of a genetic program.

Knowledge of the design of genetic algorithms is improving rapidly. Ultimately, we will be able to deploy an even more effective weapon--continuous, evolutionary improvement in the creative mechanism itself. We will build GP systems that can evolve a better version of themselves. This will quickly and dramatically extend GP capacity in ways that we cannot predict. We will have to call this thing by another name: artificial life.