Version 2 – Submitted August 18, 1997 for *Encyclopedia of Computer Science and Technology* to be edited by Allen Kent and James G. Williams. **7,734 words**.

# Genetic Programming

**John R. Koza**
Computer Science Department
Stanford University
258 Gates Building
Stanford, California 94305 USA
PHONE: 650-941-0336
FAX: 650-941-9430
E-MAIL: Koza@CS.Stanford.Edu
WWW: http://www-cs-faculty.stanford.edu/~koza/

## 1. Introduction

Genetic programming is a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems. Genetic programming is based on the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as *crossover* (*sexual recombination*) and *mutation*.

John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the evolutionary process can be applied to solving mathematical problems and engineering optimization problems using what is now called the *genetic algorithm* (GA). The genetic algorithm attempts to find a good (or best) solution to the problem by genetically breeding a population of individuals over a series of generations. In the genetic algorithm, each *individual* in the population represents a candidate solution to the given problem. The *genetic algorithm* (GA) transforms a *population* (set) of individuals, each with an associated *fitness* value, into a new *generation* of the population using reproduction, crossover, and mutation.

Books on genetic algorithms that include those that survey the entire field, such as Goldberg (1989), Michalewicz (1992), and Mitchell (1996) as well as others that specialize in particular areas, such as the application of genetic algorithms to robotics (Davidor (1990), financial

applications (Bauer 1994), image segmentation (Bhanu and Lee 1994), pattern recognition (Pal and Wang 1996), parallelization (Stender 1993), and simulation an modeling (Stender, Hillebrand, and Kingdon 1994), control and signal processing (Man, Tang, Kwong, and Halang 1997), and engineering design (Gen and Cheng 1997).

Edited collection of papers on genetic algorithms include Davis (1987, 1991), Chambers (1995), Biethahn and Nissen (1995), Dasgupta and Michalewicz (1997), and Back, Fogel, and Michalewicz (1997).

Recent work on genetic algorithms can often be found in conference proceedings, such as the International Conference on Genetic Algorithms (Back 1997), ICEC – International Conference on Evolutionary Computation (IEEE 1997), the annual Genetic Programming Conference (Koza et al. 1997), Parallel Problem Solving from Nature (Voigt, Ebeling, Rechenberg, and Schwefel 1996), Artificial Evolution (Alliot et al. 1995), Genetic Algorithms in Engineering Systems: Innovations and Applications (IEE 1995), Evolutionary Computing (Fogarty 1995), Evolutionary Computation and its Applications (Goodman 1996), Frontiers of Evolutionary Algorithms (Wang 1997), Simulated Evolution And Learning (Yao, Kim, and Furuhashi 1997), International Conference on Evolvable Systems (Higuchi, Iwata, and Lui 1997), International Conference on Artificial Neural Nets and Genetic Algorithms (Pearson, Steele, and Albrecht 1995), and the Evolutionary Programming Conference (Angeline, Reynolds, McDonnell, and Eberhart 1997).

Genetic programming addresses one of the central goals of computer science, namely automatic programming. The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem. Paraphrasing Arthur Samuel (1959), the goal of automatic programming concerns,

> How can computers be made to do what needs to be done, without being told exactly how to do it?

In genetic programming, the genetic algorithm operates on a population of computer programs of varying sizes and shapes (Koza 1992). Genetic programming starts with a primordial ooze of

thousands or millions of randomly generated computer programs composed of the available programmatic ingredients and then applies the principles of animal husbandry to breed a new (and often improved) population of programs. The breeding is done in a domain-independent way using the Darwinian principle of survival of the fittest, an analog of the naturally-occurring genetic operation of crossover (sexual recombination), and occasional mutation. The crossover operation is designed to create syntactically valid offspring programs (given closure amongst the set of programmatic ingredients). Genetic programming combines the expressive high-level symbolic representations of computer programs with the near-optimal efficiency of learning of Holland's genetic algorithm. A computer program that solves (or approximately solves) a given problem often emerges from this process. See also Koza and Rice 1992.

Genetic programming breeds computer programs to solve problems by executing the following three steps:

(1) Generate an initial population of random compositions of the functions and terminals of the problem (i.e., computer programs).

(2) Iteratively perform the following substeps until the termination criterion has been satisfied:

  (A) Execute each program in the population and assign it a fitness value using the fitness measure.

  (B) Create a new population of computer programs by applying the following operations. The operations are applied to computer program(s) chosen from the population with a probability based on fitness.

    (i) *Darwinian Reproduction*: Reproduce an existing program by copying it into the new population.

    (ii) *Crossover*: Create two new computer programs from two existing programs by genetically recombining randomly chosen parts of two existing programs using the crossover operation (described below) applied at a randomly chosen crossover point within each program.

    (iii) *Mutation*: Create one new computer program from one existing program by mutating a randomly chosen part of the program.

(3) The program that is identified by the method of result designation is designated as the result for the run (e.g., the best-so-far individual). This result may be a solution (or an approximate solution) to the problem.

Multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms (called *automatically defined functions*) may also be evolved (Koza 1994a, 1994b). An *automatically defined function* (*ADF*) is a function (i.e., subroutine,

subprogram, DEFUN, procedure) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (or subprogram) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) *reusable* function-defining branches (i.e., automatically defined functions) along with a main result-producing branch. Typically, the automatically defined functions possess one or more dummy arguments (formal parameters) and are reused with different instantiations of these dummy arguments. During a run, genetic programming evolves different subprograms in the function-defining branches of the overall program, different main programs in the result-producing branch, different instantiations of the dummy arguments of the automatically defined functions in the function-defining branches, and different hierarchical references between the branches.

Architecture-altering operations enhance genetic programming with automatically defined functions by providing a way to automatically determine the number of such subprograms, the number of arguments that each subprogram possesses, and the nature of the hierarchical references, if any, among such subprograms (Koza 1995). These operations include branch duplication, argument duplication, branch creation, argument creation, branch deletion, and argument deletion. The architecture-altering operations are motivated by the naturally occurring mechanism of gene duplication that creates new proteins (and hence new structures and new behaviors in living things) (Ohno 1970).

Recent research on genetic programming is described in Banzhaf, Nordin, Keller, and Francone (1997), the proceedings of the annual Genetic Programming Conferences (Koza et al. 1997), and in most of the conferences cited earlier on evolutionary computation. Edited collection of papers on genetic programming include Kinnear (1994) and Angeline and Kinnear (1996).

Before applying genetic programming to a problem, the user must perform five major preparatory steps. These five steps involve determining

(1) the set of terminals,

(2) the set of primitive functions,

(3) the fitness measure,

(4) the parameters for controlling the run, and

(5) the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is to identify the set of terminals. The terminals can be viewed as the inputs to the as-yet-undiscovered computer program. The set of terminals (along with the set of functions) are the ingredients from which genetic programming attempts to construct a computer program to solve, or approximately solve, the problem.

The second major step in preparing to use genetic programming is to identify the set of functions that are to be used to generate the mathematical expression that attempts to fit the given finite sample of data. Each computer program (i.e., parse tree, mathematical expression, LISP S-expression) is a composition of functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. Each of the functions in the function set should be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, the function set and terminal set selected should have the closure property so that any possible composition of functions and terminals produces a valid executable computer program. For example, a run of genetic programming will typically employ a protected version of division (returning an arbitrary value such as zero when division by zero is attempted).

The evolutionary process is driven by the *fitness measure*. Each individual computer program in the population is executed and then evaluated, using the fitness measure, to determine how well it performs in the particular problem environment. The nature of the fitness measure varies with the problem. For many problems, fitness is naturally measured by the discrepancy between the result produced by an individual candidate program and the desired result. The closer this error is to zero, the better the program. In a problem of optimal control, the fitness of a computer program may be the amount of time (or fuel, or money, etc.) it takes to bring the system to a

desired target state. The smaller the amount, the better. If one is trying to recognize patterns or classify objects into categories, the fitness of a particular program may be measured by accuracy or correlation. For electronic circuit design problems, the fitness measure may involve how closely the circuit's performance (say, in the frequency or time domain) satisfies user-specified design requirements. If one is trying to evolve a good randomizer, the fitness might be measured by means of entropy, satisfaction of the gap test, satisfaction of the run test, or some combination of these factors. For some problems, it may be appropriate to use a multiobjective fitness measure incorporating a combination of factors such as correctness, parsimony (smallness of the evolved program), efficiency (of execution), power consumption (for an electrical circuit), or manufacturing cost (for an electrical circuit).

The primary parameters for controlling a run of genetic programming are the population size, *M*, and the maximum number of generations to be run, *G*.

Each run of genetic programming requires specification of a *termination criterion* for deciding when to terminate a run and a method of *result designation*. One frequently used method of result designation for a run is to designate the best individual obtained in any generation of the population during the run (i.e., the *best-so-far individual*) as the result of the run.

In genetic programming, populations of thousands or millions of computer programs are genetically bred for dozens, hundreds, or thousands of generations. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic crossover operation appropriate for mating computer programs. A computer program that solves (or approximately solves) a given problem often emerges from this combination of Darwinian natural selection and genetic operations.

Genetic programming starts with an initial population (generation 0) of randomly generated computer programs composed of the given primitive functions and terminals. Typically, the size of each program is limited, for practical reasons, to a certain maximum number of points (i.e. total number of functions and terminals) or a maximum depth (of the program tree). The

creation of this initial random population is, in effect, a blind random parallel search of the search space of the problem represented as computer programs.

Typically, each computer program in the population is run over a number of different *fitness cases* so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable or a sampling of different initial conditions of a system. For example, the fitness of an individual computer program in the population may be measured in terms of the sum of the absolute value of the differences between the output produced by the program and the correct answer to the problem (i.e., the Minkowski distance) or the square root of the sum of the squares (i.e., Euclidean distance). These sums are taken over a sampling of different inputs (fitness cases) to the program. The fitness cases may be chosen at random or may be chosen in some structured way (e.g., at regular intervals or over a regular grid). It is also common for fitness cases to represent initial conditions of a system (as in a control problem). In economic forecasting problems, the fitness cases may be the daily closing price of some financial instrument.

The computer programs in generation 0 of a run of genetic programming will almost always have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat more fit than others. These differences in performance are then exploited.

The Darwinian principle of reproduction and survival of the fittest and the genetic operation of crossover are used to create a new offspring population of individual computer programs from the current population of programs.

The reproduction operation involves selecting a computer program from the current population of programs based on fitness (i.e., the better the fitness, the more likely the individual is to be selected) and allowing it to survive by copying it into the new population.

The crossover operation creates new offspring computer programs from two parental programs selected based on fitness. The parental programs in genetic programming are typically of different sizes and shapes. The offspring programs are composed of subexpressions (subtrees,

subprograms, subroutines, building blocks) from their parents. These offspring programs are typically of different sizes and shapes than their parents.

For example, consider the following computer program (presented here as a LISP S-expression):

```
(+ (* 0.234 Z) (- X 0.789)),
```

which we would ordinarily write as

$0.234\,Z + X - 0.789$.

This program takes two inputs (X and Z) and produces a floating point output.

Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))).
```

One crossover point is randomly and independently chosen in each parent. Suppose that the crossover points are the * in the first parent and the + in the second parent. These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs.

The two offspring resulting from crossover are as follows:

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

```
(* (* Z Y) (* 0.234 Z)).
```

Thus, crossover creates new computer programs using parts of existing parental programs. Because entire sub-trees are swapped, the crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability based

on fitness, crossover allocates future trials to regions of the search space whose programs contains parts from promising programs.

The mutation operation creates an offspring computer program from one parental programs selected based on fitness. One crossover point is randomly and independently chosen and the subtree occurring at that point is deleted. Then, a new subtree is grown at that point using the same growth procedure as was originally used to create the initial random population.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation). Each individual in the new population of programs is then measured for fitness, and the process is repeated over many generations.

The hierarchical character of the computer programs that are produced is an important feature of genetic programming. The results of genetic programming are inherently hierarchical. In many cases the results produced by genetic programming are default hierarchies, prioritized hierarchies of tasks, or hierarchies in which one behavior subsumes or suppresses another.

The dynamic variability of the computer programs that are developed along the way to a solution is also an important feature of genetic programming. It is often difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance. Moreover, advance specification or restriction of the size and shape of the solution to a problem narrows the window by which the system views the world and might well preclude finding the solution to the problem at all.

Another important feature of genetic programming is the absence or relatively minor role of preprocessing of inputs and postprocessing of outputs. The inputs, intermediate results, and outputs are typically expressed directly in terms of the natural terminology of the problem domain. The programs produced by genetic programming consist of functions that are natural for the problem domain. The postprocessing of the output of a program, if any, is done by a *wrapper* (*output interface*).

Finally, another important feature of genetic programming is that the structures undergoing adaptation in genetic programming are active. They are not passive encodings (i.e., chromosomes) of the solution to the problem. Instead, given a computer on which to run, the structures in genetic programming are active structures that are capable of being executed in their current form.

Automated programming requires some hierarchical mechanism to exploit, *by reuse* and *parameterization*, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines do this in ordinary computer programs.

Automatically defined functions can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs in the population. Each multi-part program in the population contains one (or more) function-defining branches and one (or more) main result-producing branches. The result-producing branch usually has the ability to call one or more of the automatically defined functions. A function-defining branch may have the ability to refer hierarchically to other already-defined automatically defined functions.

Genetic programming evolves a population of programs, each consisting of an automatically defined function in the function-defining branch and a result-producing branch. The structures of both the function-defining branches and the result-producing branch are determined by the combined effect, over many generations, of the selective pressure exerted by the fitness measure and by the effects of the operations of Darwinian fitness-based reproduction and crossover. The function defined by the function-defining branch is available for use by the result-producing branch. Whether or not the defined function will be actually called is not predetermined, but instead, determined by the evolutionary process.

Since each individual program in the population of this example consists of function-defining branch(es) and result-producing branch(es), the initial random generation must be created so that every individual program in the population has this particular constrained syntactic structure.

Since a constrained syntactic structure is involved, crossover must be performed so as to preserve this syntactic structure in all offspring.

Genetic programming with automatically defined functions has been shown to be capable of solving numerous problems (Koza 1994a). More importantly, the evidence so far indicates that, for many problems, genetic programming requires less computational effort (i.e., fewer fitness evaluations to yield a solution with, say, a 99% probability) with automatically defined functions than without them (provided the difficulty of the problem is above a certain relatively low break-even point).

Also, genetic programming usually yields solutions with smaller average overall size with automatically defined functions than without them (provided, again, that the problem is not too simple). That is, both learning efficiency and parsimony appear to be properties of genetic programming with automatically defined functions.

Moreover, there is evidence that genetic programming with automatically defined functions is scalable. For several problems for which a progression of scaled-up versions was studied, the computational effort increases as a function of problem size at a *slower rate* with automatically defined functions than without them. Also, the average size of solutions similarly increases as a function of problem size at a *slower rate* with automatically defined functions than without them. This observed scalability results from the profitable reuse of hierarchically-callable, parameterized subprograms within the overall program.

When single-part programs are involved, genetic programming automatically determines the size and shape of the solution (i.e., the size and shape of the program tree) as well as the sequence of work-performing primitive functions that can solve the problem. However, when multi-part programs and automatically defined functions are being used, the question arises as to how to determine the architecture of the programs that are being evolved. The *architecture* of a multi-part program consists of the number of function-defining branches (automatically defined functions) and the number of arguments (if any) possessed by each function-defining branch. The architecture may be specified by the user, may be evolved using evolutionary selection of

the architecture (Koza 1994a), or may be evolved using architecture-altering operations (Koza 1995).

## 2. The Threshold of Practicality

Genetic programming has been used to produce results that are competitive with human performance on certain non-trivial problems. In fields as diverse as cellular automata, space satellite control, molecular biology, and design of electrical circuits, genetic programming has evolved a computer program whose results were, under some reasonable interpretation, competitive with human performance on the specific problem. For example, genetic programming with automatically defined functions has evolved a rule for the majority classification task for one-dimensional two-state cellular automata with an accuracy that exceeds that of the original human-written Gacs-Kurdyumov-Levin (GKL) rule, all other known subsequent human-written rules, and all other known rules produced by automated approaches for this problem (Andre, Bennett, and Koza 1996). Another example involves the near-minimum-time control of a spacecraft's attitude maneuvers using genetic programming (Howley 1996). A third example involves the discovery by genetic programming of a computer program to classify a given protein segment as being a transmembrane domain without using biochemical knowledge concerning hydrophobicity (Koza 1994a; Koza and Andre 1996a). A fourth example illustrated how automated methods may prove to be useful in discovering biologically meaningful information hidden in the rapidly growing databases of DNA sequences and protein sequences. Genetic programming successfully evolved motifs for detecting the D-E-A-D box family of proteins and for detecting the manganese superoxide dismutase family that detected the two families either as well as, or slightly better than, the comparable human-written motifs found in the database created by an international committee of experts on molecular biology (Koza and Andre 1996b). A fifth example is recent work on facility layouts (Garces-Perez, Schoenefeld, and Wainwright 1996).

An additional group of examples is provided by work in which genetic programming has been used to evolve both the topology and numerical component values for electrical circuits,

including lowpass filters, crossover (woofer and tweeter) filters, asymmetric bandpass filters, amplifiers, computational circuits, a time-optimal controller circuit, a temperature-sensing circuit, and a voltage reference circuit (Koza, Bennett, Andre, Keane, and Dunlap 1997).

### 3. Operations on Complex Data Structures

Ordinary computer programs use numerous well-known techniques for handling vectors of data, arrays, and more complex data structures. One important area for work on technique extensions for genetic programming involves developing workable and efficient ways to handle vectors, arrays, trees, graphs, and more complex data structures. Such new techniques would have immediate application to a number of problems in such fields as computer vision, biological sequence analysis, economic time series analysis, and pattern recognition where a solution to the problem involves analyzing the character of an entire data structure. Recent work in this area includes that of Langdon (1996) in handling more complex data structures such as stacks, queues, rings, and lists, the work of Teller (1996) in understanding images represented by large arrays of pixels, and the work of Handley (1996) in applying statistical computing zones and iteration to biological sequence data and other problems.

### 4. Evolution of Mental Models

Complex adaptive systems usually possess a mechanism for modeling their environment. A mental model of the environment enables a system to contemplate the effects of future actions and to choose an action that best fulfills its goal. Brave (1996b) has developed a special form of memory that is capable of creating relations among objects and then using these relations to guide the decisions of a system.

### 5. Automatically Defined Functions, Automatically Defined Macros, and Modules

Computer programs gain leverage in solving complex problems by means of reusable and parameterizable subprograms. Automated machine learning can become scalable (and truly useful) only if there are techniques for creating large and complex problem-solving programs from smaller building blocks. Spector (1996) has developed the notion of automatically defined

macros (ADMs) for use in evolving control structures. Rosca (1995) has analyzed the workings of hierarchical arrangements of subprograms in genetic programming. Angeline (1994) has studied modules that are made available to all programs in the population through a genetic library.

Automatically defined functions and architecture-altering operations for creating useful electrical subcircuits (Koza, Andre, Bennett, and Keane 1996).

## 6. Cellular Encoding

Gruau (1994) described an innovative technique, called *cellular encoding* or *developmental genetic programming* in which genetic programming is used to concurrently evolve the architecture of a neural network, along with the weights, thresholds, and biases of the individual neurons in the neural network. In this technique, each individual program tree in the population is a specification for developing a complete neural network from a starting point consisting of a very simple embryonic neural network containing a single neuron. Genetic programming is applied to populations of these network-constructing program trees in order to evolve a neural network to solve various problems.

Brave (1996a) has extended and applied this technique to the evolution of finite automata.

## 7. Automatic Programming of Multi-Agent Systems

The cooperative behavior of multiple independent agents can potentially be harnessed to solve a wide variety of practical problems. However, programming of multi-agent systems is particularly vexatious. Bennett's recent work (1996) in evolving the number of independent agents while concurrently evolving the specific behaviors of each agent and the recent work by Luke and Spector (1996) in evolving teamwork are opening this area to the application of genetic programming. See also Iba (1997).

## 8. Autoparallelization of Algorithms

The problem of mapping a given sequential algorithm onto a parallel machine is usually more difficult than writing a parallel algorithm from scratch. The recent work of Walsh and Ryan (1996) is advancing the autoparallelization of algorithms using genetic programming.

## 9. Co-Evolution

In nature, individuals do not evolve in a vacuum. Instead, there is co-evolution that involves interactions between agents and other agents as well as between agents and their physical environment (Angeline and Pollack 1994, Pollack and Blair 1996).

## 10. Complex Adaptive Systems

Genetic programming has proven useful in evolving complex systems, such as Lindenmayer systems (Jacob 1996) and cellular automata (Andre, Bennett, and Koza 1996) and can be expected to continue to be useful in this area.

## 11. Evolution of Structure

One of the most vexatious aspects of automated machine learning from the earliest times has been the requirement that the human user predetermine the size and shape of the ultimate solution to his problem (Samuel 1959). There can be expected to be continuing research on ways by which the size and shape of the solution can be made part of the *answer* provided by the automated machine learning technique, rather than part of the *question* supplied by the human user. For example, architecture-altering operations (Koza 1995) enable genetic programming to introduce (or delete) function-defining branches, to adjust the number of arguments of each function-defining branch, and to alter the hierarchical references among function-defining branches. Brave (1995) showed that recursion could be implemented within genetic programming. It is also possible to evolve iterations using genetic programming (Koza and Andre 1996b).

## 12. Foundations of Genetic Programming

Genetic programming inherits many of the mathematical and theoretical underpinnings from John Holland's pioneering work (1975) in the field, including the near-optimality of Darwinian search. However, the genetic algorithm is a dynamical system of extremely high dimensionality. Many of the most basic questions about the operation of the algorithm and the domain of its applicability are only partially understood. The transition from the fixed-length character strings of the genetic algorithm to the variable-sized Turing-complete program trees (Teller 1994) and even program graphs (Teller 1996) of genetic programming further compounds the difficulty of the theoretical issues involved. There is increasing work on the grammatical structure of genetic programming (Whigham 1996) and the theoretical basis for genetic programming (Poli and Langdon 1997).

## 13. Optimization

Recent examples of applications of genetic programming to problems of optimization include work (Soule, Foster, and Dickinson 1996) from the University of Idaho, the site of much early work on genetic programming techniques, and the work of Garces-Perez, Schoenefeld, and Wainwright (1996).

## 14. Evolution of Assembly Code

The innovative work by Nordin (1994) in developing a version of genetic programming in which the programs are composed of sequence of low-level machine code offers numerous possibilities for extending the techniques of genetic programming (especially for programs with loops) as well as enormous savings in computer time. These savings can then be used to increase the scale of problems being considered. See also Banzhaf, Nordin, Keller, and Francone (1997).

## 15. Techniques that Exploit Parallel Hardware

Evolutionary algorithms offer the ability of solve problems in a domain-independent way that requires little domain-specific knowledge. However, the price of this domain-independence and knowledge-independence is paid in execution time. Application of genetic programming to

realistic problems usually requires substantial computational resources. The long-term trend toward ever faster microprocessors is likely to continue to make ever-increasing amounts of computational power available at ever-decreasing cost. However, for those using algorithms that can beneficially exploit parallelization (such as genetic programming), parallelization is even more important than microprocessor speed in terms of delivering large amounts of computational power. In genetic programming, the vast majority of computer resources are used on the fitness evaluations. The calculation of fitness for one individual in the population is usually independent and decoupled from the calculation of fitness of all other individuals. Thus, parallel computing techniques can be applied to genetic programming (and genetic algorithms in general) with almost 100% efficiency (Andre and Koza 1996). In fact, the use of semi-isolated subpopulations often accelerates the finding of a solution to a problem using genetic programming and produces not just near-linear speed-up, but super-linear speed-up. Parallelization of genetic programming will be of central importance to the growth of the field.

**16. Evolvable Hardware**

One of the newest areas of evolutionary computation involves the use of evolvable hardware (Sanchez and Tomassini 1996; Higuchi, Iwata, and Lui 1997). Evolvable hardware includes devices such as field programmable gate arrays (FPGA) and field programmable analog arrays (FPAA). The idea of evolvable hardware is to embody each individual of the evolving population *into hardware* and thereby exploit the massive parallelism of the hardware to perform evolution "in silicon." These devices are reconfigurable with very short configuration times and download times. Thompson (1996) has pioneered the use of field-programmable gates arrays to evolve a frequency discriminator circuit and a robot controller using the recently developed Xilinix XC6216 chip. Considerable growth can be anticipated in the use of evolvable hardware to accelerate genetic programming runs and perform evolution.

## 17. Future Work

The presence of some or all of the following characteristics make an area especially suitable for the application of genetic programming:

- an area where conventional mathematical analysis does not, or cannot, provide analytic solutions,
- an area where the interrelationships among the relevant variables are poorly understood (or where it is suspected that the current understanding may well be wrong),
- an area where finding the size and shape of the ultimate solution to the problem is a major part of the problem,
- an area where an approximate solution is acceptable (or is the only result that is ever likely to be obtained),
- an area where there is a large amount of data, in computer readable form, that requires examination, classification, and integration, or
- an area where small improvements in performance are routinely measured (or easily measurable) and highly prized.

For example, problems in automated control are especially well suited for genetic programming because of the inability of conventional mathematical analysis to provide analytic solutions to many problems of practical interest, the willingness of control engineers to accept approximate solutions, and the high value placed on small incremental improvements in performance.

Problems in fields where large amounts of data are accumulating in machine readable form (e.g., biological sequence data, astronomical observations, geological and petroleum data, financial time series data, satellite observation data, weather data, news stories, marketing databases) also constitute especially interesting areas for potential practical applications of genetic programming.

## Bibliography

Alliot, J. M. Lutton, E., Ronald, E., Schoenauer, M., and Snyers, D. (editors). 1995. *Artificial Evolution: European Conference, AE 95, Brest, France, September 1995, Selected Papers*. Lecture Notes in Computer Science, Volume 1063. Berlin: Springer-Verlag.

Andre, David, Bennett III, Forrest H, and Koza, John R. 1996. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo,

Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press. Chapter 18.

Angeline, Peter J. 1994. Genetic programming and the emergence of intelligence. In Kinnear, K. E. Jr. (editor). *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Angeline, Peter J. and Pollack, Jordan B. Coevolving high-level representations. In Langton, Christopher G. (editor). *Artificial Life III, SFI Studies in the Sciences of Complexity*. Volume XVII Redwood City, CA: Addison-Wesley. Pages 55–71. 1994**.**

Angeline, Peter J., Reynolds, Robert G., McDonnell, John R., and Eberhart, Russ (editors). *Evolutionary Programming VI. 6th International Conference, EP97, Indianapolis, Indiana, USA, April 1997 Proceedings*. Lecture Notes in Computer Science, Volume 1213. Berlin: Springer-Verlag. 125–136.

Back, Thomas. (editor). 1997. *Genetic Algorithms: Proceedings of the Fifth International Conference*. San Francisco, CA: Morgan Kaufmann.

Back, Thomas, Fogel, David B., and Michalewicz, Zbigniew (editors). 1997. *Handbook of Evolutionary Computation*. Bristol, UK: Institute of Physics Publishing and New York: Oxford University Press.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1997. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

Bauer, R. J., Jr. 1994. *Genetic Algorithms and Investment Strategies*. John Wiley.

Bennett, Forrest H III. 1996. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Bhanu, Bir and Lee, Sungkee. 1994. *Genetic Learning for Adaptive Image Segmentation*. Boston: Kluwer Academic Publishers.

Biethahn, Jorg and Nissen, Volker (editors). 1995. *Evolutionary Algorithms in Management Applications*. Berlin: Springer-Verlag.

Brave, Scott. 1995. Using genetic programming to evolve recursive programs for tree search. *Proceedings of the Fourth Golden West Conference on intelligent System*s. Raleigh, NC: International Society for Computers and Their Applications. Pages 60 – 65.

Brave, Scott. 1996a. Evolving deterministic finite automata using cellular encoding. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Brave, Scott. 1996b. The evolution of memory and mental models using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Chambers, Lance (editor). 1995. *Practical Handbook of Genetic Algorithms: Applications: Volume I.* Boca Raton, FL: CRC Press.

Dasgupta, D. and Michalewicz, Z. (editors). 1997. *Evolutionary Algorithms in Engineering Applications*. Berlin: Springer-Verlag.

Davidor, Yuval. *Genetic Algorithms and Robotics*. Singapore: World Scientific 1991.

Davis, Lawrence (editor) 1987 *Genetic Algorithms and Simulated Annealing*. London: Pittman.

Davis, Lawrence 1991. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

Fogarty, Terence C. (editor). 1995. *Evolutionary Computing: AISB Workshop, Sheffield, U. K., April 1995, Selected Papers*. Lecture Notes in Computer Science, Volume 993. Berlin: Springer-Verlag.

Garces-Perez, Jaime, Schoenefeld, Dale A., and Wainwright, Roger L. 1996. Solving facility layout problems using genetic programming**.** In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Gen, Mitsuo and Cheng, Runwei. 1997. *Genetic Algorithms and Engineering Design*. New York: John Wiley and Sons.

Goodman, Erik D. (editor). 1996. *Proceedings of the First International Conference on Evolutionary Computation and Its Applications*. Moscow: Presidium of the Russian Academy of Sciences.

Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Gruau, Frederic. 1994. Genetic micro programming of neural networks. In Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press. Pages 495–518.

Handley, Simon. 1996. A new class of function sets for solving sequence problems. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Haynes, Thomas and Sen, Sandip. 1997. Crossover operators for evolving a team. In Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University*. San Francisco, CA: Morgan Kaufmann. Pages 162 – 167.

Higuchi, Tetsuya, Iwata, Masaya, and Lui, Weixin (editors). 1997. *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science, Volume 1259. Berlin: Springer-Verlag.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press. The 1992 second edition was published by The MIT Press.

Howley, Brian. 1996. Genetic programming of near-minimum-time spacecraft attitude maneuvers. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Iba, Hitoshi. 1997. Multiple-agent learning for a robot navigation task by genetic programming. In Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University*. San Francisco, CA: Morgan Kaufmann. Pages 195 – 200.

IEE. 1995. *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA)*. London: Institution of Electrical Engineers.

IEEE. 1997. *Proceedings of the Fourth IEEE Conference on Evolutionary Computation.* IEEE Press.

Jacob, Christian. 1996. Evolving evolution programs: Genetic programming and L-Systems. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press. Pages 107–115.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: MIT Press.

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

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R. 1995. Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. *Proceedings of 14th International Joint Conference on Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Koza, John R. and Andre, David. 1996a. Evolution of iteration in genetic programming. In *Evolutionary Programming V*: *Proceedings of the Fifth Annual Conference on Evolutionary Programming*. Cambridge, MA: MIT Press.

Koza, John R. and Andre, David. 1996b. Automatic discovery of protein motifs using genetic programming. In Yao, Xin (editor). 1996. *Evolutionary Computation: Theory and Applications*. Singapore: World Scientific.

Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A, and Dunlap, Frank. 1997. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*. 1(2).

Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13–16, 1997, Stanford University*. San Francisco, CA: Morgan Kaufmann.

Langdon, W. B. 1996. Using data structures within genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic*

*Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Luke, Sean and Spector, Lee. 1996. Evolving teamwork and coordination with genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. Pages 150–156.

Man, K. F., Tang, K. S., Kwong, S., and Halang, W. A. 1997. *Genetic Algorithms for Control and Signal Processing*. London: Springer-Verlag.

Michalewicz, Z. 1992. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer-Verlag.

Mitchell, Melanie. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Nordin, Peter. 1994. A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Ohno, Susumu. *Evolution by Gene Duplication*. New York: Springer-Verlag 1970.

Pal, Sankar K. and Wang, Paul P. 1996. *Genetic Algorithms and Pattern Recognition*. Boca Raton, FL: CRC Press.

Pearson, D. W., Steele, N. C., and Albrecht, R. F. 1995. *Artificial Neural Nets and Genetic Algorithms*. Vienna: Springer-Verlag.

Poli, Riccardo and Langdon, W. B. 1997. A new schema theory for genetic programming with one-point crossover and point mutation. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. Pages 278 – 285.

Pollack, Jordan B. and Blair, Alan D. 1996. Coevolution of a backgammon player. In *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Rosca, Justinian P. 1995. Genetic programming exploratory power and the discovery of functions. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). 1995. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press.

Samuel, Arthur L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3(3): 210–229.

Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag.

Spector, Lee. 1996. Simultaneous evolution of programs and their control structures. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Soule, Terence, Foster, James A., and Dickinson, John. 1996. Code growth in genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Stender, Joachim (editor). 1993. *Parallel Genetic Algorithms*. Amsterdam: IOS Publishing.

Stender, Joachim, Hillebrand, and Kingdon, J. (editors). 1994. *Genetic Algorithms in Optimization, Simulation, and Modeling*. Amsterdam: IOS Publishing.

Teller, A. 1994. Turing completeness in the language of genetic programming with indexed memory. *Proceedings of The First IEEE Conference on Evolutionary Computation.* IEEE Press. Volume I. Pages 136-141.

Teller, Astro and Veloso Manuela. 1996. PADO: A new learning architecture for object recognition. In Ikeuchi, Katsushi and Veloso, Manuela (editors). *Symbolic Visual Learning*. Oxford University Press.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Voigt, Hans-Michael, Ebeling, Werner, Rechenberg, Ingo, and Schwefel, Hans-Paul (editors). 1996. *Parallel Problem Solving from Nature – PPSN IV*. Berlin: Springer-Verlag.

Walsh, Paul and Ryan, Conor. 1996. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Wang, Paul P. (editor). 1997. *Proceedings of Joint Conference of Information Sciences*.

Whigham, Peter A. Search bias, language bias, and genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Yao, Xin, Kim, J.-H. and Furuhashi, T. (editors). 1997. *Simulated Evolution and Learning*. Lecture Notes in Artificial Intelligence, Volume 1285. Heidelberg: Springer-Verlag.