SURVEY OF GENETIC ALGORITHMS AND GENETIC PROGRAMMING

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305
Koza@CS.Stanford.Edu 415-941-0336
http://www-cs-faculty.stanford.edu/~koza/

ABSTRACT

This paper provides an introduction to genetic algorithms and genetic programming and lists sources of additional information, including books and conferences as well as e-mail lists and software that is available over the Internet.

1. GENETIC ALGORITHMS

John Holland's pioneering book *Adaptation in Natural and Artificial Systems* (1975, 1992) showed how the evolutionary process can be applied to solve a wide variety of problems using a highly parallel technique that is now called the *genetic algorithm*.

The *genetic algorithm* (GA) transforms a *population* (set) of individual objects, each with an associated *fitness* value, into a new *generation* of the population using the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as *crossover* (*sexual recombination*) and *mutation*.

Each *individual* in the population represents a possible solution to a given problem. The genetic algorithm attempts to find a very good (or best) solution to the problem by genetically breeding the population of individuals over a series of generations.

Before applying the genetic algorithm to the problem, the user designs an artificial chromosome of a certain fixed size and then defines a mapping (encoding) between the points in the search space of the problem and instances of the artificial chromosome. For example, in applying the genetic algorithm to a multidimensional optimization problem (where the goal is to find the global optimum of an unknown multidimensional function), the artificial chromosome may be a linear character string (modeled directly after the linear string of information found in DNA). A specific location (a gene) along this artificial chromosome is associated with each of the variables of the problem. Character(s) appearing at a particular location along the chromosome denote the value of a particular variable (i.e., the gene value or allele). Each individual in the population has a fitness value (which, for a multidimensional optimization problem, is the value of the unknown function). The genetic algorithm then manipulates a population of such artificial chromosomes (usually starting from a randomly-created initial population of strings) using the operations of reproduction, crossover, and mutation. Individuals are probabilistically selected to participate in these genetic operations based on their fitness. The goal of the genetic algorithm in a multidimensional optimization problem is to find an artificial chromosome which, when decoded and mapped back into the search space of the problem, corresponds to a globally optimum (or near-optimum) point in the original search space of the problem.

In preparing to use the conventional genetic algorithm operating on fixed-length character strings to solve a problem, the user must

(1) determine the representation scheme,
(2) determine the fitness measure,
(3) determine the parameters and variables for controlling the algorithm, and
(4) determine a way of designating the result and a criterion for terminating a run.

In the conventional genetic algorithm, the individuals in the population are usually fixed-length character strings patterned after chromosome strings. Thus, specification of the *representation scheme* in the conventional genetic algorithm starts with a selection of the string length L and the alphabet size K. Often the alphabet is binary, so K equals 2. The most important part of the representation scheme is the mapping that expresses each possible point in the search space of the problem as a fixed-length character string (i.e., as a *chromosome*) and each chromosome as a point in the search space of the problem. Selecting a representation scheme that facilitates solution of the problem by the genetic algorithm often requires considerable insight into the problem and good judgment.

The evolutionary process is driven by the *fitness measure*. The fitness measure assigns a fitness value to each possible fixed-length character string in the population.

The primary parameters for controlling the genetic algorithm are the population size, M, and the maximum number of generations to be run, G. Populations can consist of hundreds, thousands, tens of thousands or more individuals. There can be dozens, hundreds, thousands, or more generations in a run of the genetic algorithm.

Each run of the genetic algorithm 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 of the genetic algorithm 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.

Once the four preparatory steps for setting up the genetic algorithm have been completed, the genetic algorithm can be run.

The evolutionary process described above indicates how a globally optimum combination of alleles (gene values) within a fixed-size chromosome can be evolved.

The three steps in executing the genetic algorithm operating on fixed-length character strings are as follows:

(1) Randomly create an initial population of individual fixed-length character strings.
(2) Iteratively perform the following substeps on the population of strings until the termination criterion has been satisfied:
(A) Assign a fitness value to each individual in the population using the fitness measure.
(C) Create a new population of strings by applying the following three genetic operations. The genetic operations are applied to individual string(s) in the population chosen with a probability based on fitness.
(i) Reproduce an existing individual string by copying it into the new population.
(ii) Create two new strings from two existing strings by genetically recombining substrings using the crossover operation (described below) at a randomly chosen crossover point.
(iii) Create a new string from an existing string by randomly mutating the character at one randomly chosen position in the string.
(3) The string that is identified by the method of result designation (e.g., the best-so-far individual) is designated as the result of the genetic algorithm for the run. This result may represent a solution (or an approximate solution) to the problem.

The genetic operation of *reproduction* is based on the Darwinian principle of reproduction and survival of the fittest. In the reproduction operation, an individual is probabilistically selected from the population based on its fitness (with reselection allowed) and then the individual is copied, without change, into the next generation of the population. The selection is done in such a way that the better an individual's fitness, the more likely it is to be selected. An important aspect of this probabilistic selection is that every individual, however poor its fitness, has some probability of selection.

The genetic operation of *crossover* (sexual *recombination*) allows new individuals (i.e., new points in the search space) to be created and tested. The operation of crossover starts with two parents independently selected probabilistically from the population based on their fitness (with reselection allowed). As before, the selection is done in such a way that the better an individual's fitness, the more likely it is to be selected. The crossover operation produces two offspring. Each offspring contains some genetic material from each of its parents.

Suppose that the crossover operation is to be applied to the two parental strings 10110 and 01101 of length $L = 5$ over an alphabet of size $K = 2$. The crossover operation begins by randomly selecting a number between 1 and L–1 using a uniform probability distribution. Suppose that the third interstitial location is selected. This location becomes the *crossover point*. Each parent is then split at this crossover point into a crossover fragment and a remainder. The crossover operation then recombines remainder 1 (i.e., – – – 1 0) with crossover fragment 2 (i.e., 011 – –) to create offspring 2 (i.e., 01110). The crossover operation similarly recombines remainder 2 (i.e., – – – 01) with crossover fragment 1 (i.e., 101 – –) to create offspring 1 (i.e., 10101).

The operation of mutation allows new individuals to be created. It begins by selecting an individual from the population based on its fitness (with reselection allowed). A point along the string is selected at random and the character at that point is randomly changed. The altered individual is then copied into the next generation of the population. Mutation is used very sparingly in genetic algorithm work.

The genetic algorithm works in a domain-independent way on the fixed-length character strings in the population. The genetic algorithm searches the space of possible character strings in an attempt to find high-fitness strings. The fitness landscape may be very rugged and nonlinear. To guide this search, the genetic algorithm uses only the numerical fitness values associated with the explicitly tested strings in the population. Regardless of the particular problem domain, the genetic algorithm carries out its search by performing the same disarmingly simple operations of copying, recombining, and occasionally randomly mutating the strings.

In practice, the genetic algorithm is surprisingly rapid in effectively searching complex, highly nonlinear, multidimensional search spaces. This is all the more surprising because the genetic algorithm does not know anything about the problem domain or the internal workings of the fitness measure being used.

1.1 Sources of Additional Information

David Goldberg's *Genetic Algorithms in Search, Optimization, and Machine Learning* (1989) is the leading textbook and best single source of additional information about the field of genetic algorithms.

Additional information on genetic algorithms can be found in Davis (1987, 1991), Michalewicz (1992), and Buckles and Petry (1992). The proceedings of the International Conference on Genetic Algorithms provide an overview of research activity in the genetic algorithms field. See Eshelman (1995), Forrest (1993), Belew and Booker (1991), Schaffer (1989), and Grefenstette (1985, 1987).

Also see the proceedings of the IEEE International Conference on Evolutionary Computation {IEEE 1994, 1995). The proceedings of the Foundations of Genetic Algorithms workshops cover theoretical aspects of the field. See Whitley and Vose (1995), Whitley (1992), and Rawlins (1991).

Fogel and Atmar (1992, 1993), Sebald and Fogel (1994), and Sebald and Fogel (1995) emphasizes recent work on evolutionary programming (EP).

The proceedings of the Parallel Problem Solving from Nature conferences emphasize work on evolution strategies (ES). See Schwefel and Maenner (1991), Maenner and Manderick (1992), and Davidor, Schwefel, and Maenner (1994).

Stender (1993) describes parallelization of genetic algorithms. Also see Koza and Andre 1995. Davidor (1992) describes application of genetic algorithms to robotics. Schaffer and Whitley (1992) and Albrecht, Reeves, and Steele (1993) describe work on combinations of genetic algorithms and neural networks. Forrest (1991) describes application of genetic classifier systems to semantic nets.

Additional information about genetic algorithms may be obtained from the GA-LIST electronic mailing list to which you may subscribe, at no charge, by sending a subscription request to `GA-List-Request@AIC.NRL.NAVY.MIL`. Issues of the GA-LIST provide instructions for accessing the genetic algorithms archive, which contains software that may be obtained over the Internet. The archive may be accessed over the World Wide Web at `http://www.aic.nrl.navy.mil/galist/` or through anonymous ftp at `ftp.aic.nrl.navy.mil` (192.26.18.68) in `/pub/galist`.

2. GENETIC PROGRAMMING

Genetic programming is an attempt to deal with one of the central questions in computer science (posed by Arthur Samuel in 1959), namely

> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what needs to be done, without being told exactly how to do it?

All computer programs – whether they are written in FORTRAN, PASCAL, C, assembly code, or any other programming language – can be viewed as a sequence of applications of functions (operations) to arguments (values). Compilers use this fact by first internally translating a given program into a parse tree and then converting the parse tree into the more elementary assembly code instructions that actually run on the computer. However this important commonality underlying all computer programs is usually obscured by the large variety of different types of statements, operations, instructions, syntactic constructions, and grammatical restrictions found in most popular programming languages.

Any computer program can be graphically depicted as a rooted point-labeled tree with ordered branches.

Genetic programming is an extension of the conventional genetic algorithm in which each individual in the population is a computer program.

The search space in genetic programming is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain. The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions, or domain-specific functions.

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) demonstrated a result that many found surprising and counterintuitive, namely that an automatic, domain-independent method can genetically breed computer programs capable of solving, or approximately solving, a wide variety of problems from a wide variety of fields.

In applying genetic programming to a problem, there are 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., mathematical expression, LISP S-expression, parse tree) 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.

These first two major steps correspond to the step of specifying the representation scheme for the conventional genetic algorithm. The remaining three major steps for genetic programming correspond to the last three major preparatory steps for the conventional genetic algorithm.

In genetic programming, populations of hundreds, thousands, or millions of computer programs are genetically bred. 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 functions and terminals appropriate to the problem domain. The creation of this initial random population is, in effect, a blind random search of the search space of the problem represented as computer programs.

Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. This measure is called the *fitness measure*. The nature of the fitness measure varies with the problem.

For many problems, fitness is naturally measured by the error produced by the computer program. The closer this error is to zero, the better the computer 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 of time (or fuel, or money, etc.), the better. If one is trying to recognize patterns or classify examples, the fitness of a particular program may be measured by some combination of the number of instances handled correctly (i.e., true positive and true negatives) and the number of instances handled incorrectly (i.e., false positives and false negatives). Correlation is often used as a fitness measure. On the other hand, if one is trying to find a good randomizer, the fitness of a given computer program might be measured by means of entropy, satisfaction of the gap test, satisfaction of the run test, or some combination of these factors. For electronic circuit design problems, the fitness measure may involve a convolution. 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), or efficiency (of execution).

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 is used to create 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.

The mutation operation may also be used in genetic programming.

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.

At each stage of this highly parallel, locally controlled, decentralized process, the state of the process will consist only of the current population of individuals.

The force driving this process consists only of the observed fitness of the individuals in the current population in grappling with the problem environment.

As will be seen, this algorithm will produce populations of programs which, over many generations, tend to exhibit increasing average fitness in dealing with their environment. In addition, these populations of computer programs can rapidly and effectively adapt to changes in the environment.

The best individual appearing in any generation of a run (i.e., the best-so-far individual) is typically designated as the result produced by the run of genetic programming.

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.

The genetic crossover (sexual recombination) operation operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs consisting of parts of each parent.

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))).
```

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 videotape *Genetic Programming: The Movie* (Koza and Rice 1992) provides a visualization of the genetic programming process and of solutions to various problems.

2.2 Automatically Defined Functions

I believe that no approach to automated programming is likely to be successful on non-trivial problems unless it provides 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.

Accordingly, *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994) describes how to evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms (called *automatically defined functions* or *ADF*s). A visualization of the solution to numerous example problems using automatically defined functions can be found in the videotape *Genetic Programming II Videotape: The Next Generation* (Koza 1994).

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.

2.3 Evolutionary Selection of Architecture

One technique for creating the architecture of the overall program for solving a problem during the course of a run of genetic programming is to *evolutionarily select* the architecture dynamically during a run of genetic programming. This technique is described in chapters 21 – 25 of *Genetic Programming II* : *Automatic Discovery of Reusable Programs* (Koza 1994a). The technique of evolutionary selection starts with an architecturally diverse initial random population. As the evolutionary process proceeds, individuals with certain architectures may prove to be more fit than others at solving the problem. The more fit architectures will tend to prosper, while the less fit architectures will tend to wither away.

The architecturally diverse populations used with the technique of evolutionary selection require a modification of both the method of creating the initial random population and the two-offspring subtree-swapping crossover operation previously used in genetic programming. Specifically, the architecturally diverse population is created at generation 0 so as to contain randomly-created representatives of a broad range of different architectures. Structure-preserving crossover with *point typing* is a *one-offspring* crossover operation that permits robust recombination while guaranteeing that any pair of architecturally different parents will produce syntactically and semantically valid offspring.

2.4 Architecture-Altering Operations

A second technique for creating the architecture of the overall program for solving a problem during the course of a run of genetic programming is to evolve the architecture using architecture-altering (Koza 1995).

2.8 Sources of Additional Information

In addition to the author's books (Koza 1992, 1994) and accompanying videotapes (Koza and Rice 1992, Koza 1994), the first *Advances in Genetic Programming* book (Kinnear 1994) and the upcoming second book in this series (Angeline and Kinnear 1996) contain about two dozen articles each on various applications and aspects of genetic programming.

In addition to the conferences mentioned in the earlier section on genetic algorithms, the conferences of artificial life {Brooks and Maes 1994) and simulation of adaptive behavior (Cliff et al. 1994) and have articles on genetic programming.

Additional information about genetic programming may be obtained from the GP-LIST electronic mailing list to which you may subscribe, at no charge, by sending a subscription request to `genetic-programming-request@cs.stanford.edu`.

Information about obtaining software in C, C++, LISP, and other programming languages for genetic programming, information about upcoming conferences, and links to various

researchers in the genetic programming field may be accessed over the World Wide Web at `http://www-cs-faculty.stanford.edu/~koza/`. There will be a first conference on genetic programming at Stanford on July 28-31, 1996.

3. REFERENCES

Albrecht, R. F., C. R. Reeves, and N. C. Steele. 1993. *Artificial Neural Nets and Genetic Algorithms*. Springer-Verlag.

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

Belew, R. and L. Booker (editors) 1991. *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Brooks, Rodney and Maes, Pattie (editors). 1994. *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Buckles, B. P. and F. E. Petry. 1992. *Genetic Algorithms*. Los Alamitos, CA: The IEEE Computer Society Press.

Cliff, Dave, Husbands, Philip, Meyer, Jean-Arcady, and Wilson, Stewart W. (editors). 1994. *From Animals to Animats 3 Proceedings of the Third International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: The MIT Press.

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

Davidor, Yuval, Schwefel, Hans-Paul, and Maenner, Reinhard (editors). 1994. *Parallel Problem Solving from Nature - PPSN III*. Berlin: Springer-Verlag.

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

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

Eshelman, Larry J. (editor). 1995. *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann Publishers.

Fogel, David B. and W. Atmar (editors). 1992. *Proceedings of the First Annual Conference on Evolutionary Programming*. San Diego, CA: Evolutionary Programming Society .

Fogel, David B. 1993. and W. Atmar, Eds., *Proceedings of the Second Annual Conference on Evolutionary Programming*. San Diego, CA: Evolutionary Programming Society.

Forrest, Stephanie. 1991. *Parallelism and Programming in Classifier Systems*. London: Pittman.

Forrest, Stephanie. 1993. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc.

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

Grefenstette, John J. (editor). 1985. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Grefenstette, John J. (editor). 1987. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press. Second Edition available as Cambridge, MA: The MIT Press 1992.

IEEE. 1994. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press.

IEEE. 1995. *Proceedings of the Second IEEE Conference on Evolutionary Computation.* IEEE Press.

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 the 14th International Joint Conference on Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann.

Koza, John R. and Andre, David. 1995. *Parallel Genetic Programming on a Network of Transputers*. Stanford University Computer Science Department technical report STAN-CS-TR-95-1542. January 30, 1995.

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

Maenner, R. , and B. Manderick (editors). 1992. Proceedings of the Second International Conference on Parallel Problem Solving from Nature. North Holland.

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

Rawlins, G. (editor) 1991. *Foundations of Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc.

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

Schaffer, J. D. and D. Whitley (editors). 1992. *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks 1992*. Los Alamitos, CA: The IEEE Computer Society Press.

Schwefel, Hans-Paul, and Maenner, Reinhard (editors). 1991. *Parallel Problem Solving from Nature.* Berlin: Springer-Verlag.

Sebald, A. V. and Fogel, L. J. (editors). 1994. *Proceedings of the Third Annual Conference on Evolutionary Programming*. River Edge, NJ: World Scientific.

Sebald, A. V. and Fogel, L. J. (editors). 1995. *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press.

Stender, J., Ed. *Parallel Genetic Algorithms*. IOS Publishing.

Whitley, Darrell (editor) 1992. *Foundations of Genetic Algorithms and Classifier Systems 2*, Vail, Colorado 1992. San Mateo, CA: Morgan Kaufmann Publishers Inc.

Whitley, Darrell and Vose, Michael (editors). 1995. *Proceedings of Third Workshop on the Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishers Inc.