International Journal of Modelling and Simulation

ISSN: 0228-6203 (Print) 1925-7082 (Online) Journal homepage: https://www.tandfonline.com/loi/tjms20

Game Learning Using A Genetic Algorithm

Heng-Ming Tai & David Robison

To cite this article: Heng-Ming Tai & David Robison (1997) Game Learning Using A Genetic Algorithm, International Journal of Modelling and Simulation, 17:4, 270-275, DOI: 10.1080/02286203.1997.11760340

To link to this article: https://doi.org/10.1080/02286203.1997.11760340

Published online: 01 Sep 2016.

Submit your article to this journal

Article views: 22

View related articles

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=tjms20

International Journal of Modelling and Simulation, Vol. 17, No. 4, 1997

GAME LEARNING USING A GENETIC ALGORITHM

Heng-Ming Tai* and David Robison

Abstract

This paper explores the use of a genetic algorithm for game learning. Genetic algorithms are search-and-optimization procedures inspired by natural selection and evolutionary genetics. They require little knowledge of the problem and no problem-specific information such as derivatives or continuity properties. The algorithm learns how to play the game by maximizing a fitness function that relates to the performance.

Key Words

Genetic algorithms, evolutionary algorithms, learning, optimization, machine intelligence

1. Introduction

The hope of artificial intelligence is to create a learning capability similar to that of the human brain. Recently, computational intelligence [1], which includes neural networks, fuzzy logic, and genetic algorithms, has emerged as a novel approach to learning. Learning can be considered as an adaptive and optimization process that mandates a trade-off between behavioural requirements in any dynamic environment. As almost all intelligent systems are evolutionary, it is essential to simulate evolutionary processes in order to create machine intelligence.

The fundamental approach to optimization is to formulate a performance measure—a cost function—that assesses the performance and iteratively improves this performance. Most classical methods of optimization generate a deterministic sequence of trial solutions based on the gradient or higher-order statistics of the cost function [2–3]. These techniques generate sequences that asymptotically converge to *locally* optimal solutions. It is widely known that these calculus-based optimization techniques often fail to perform adequately when a complete model of the problem is difficult to obtain, or cost functions have many local optima. Many enumerative schemes [4] have been proposed to overcome the flaws in calculus-based methods. However, these schemes lack efficiency because many search spaces are too large to search one at a time.

Genetic algorithms [5–8] and evolutionary algorithms [9–12] have emerged as useful search-and-optimization techniques that are applicable to difficult real-world problems. These methods combine a Darwinian survival-of-the-fittest strategy to eliminate unfit characteristics. These simulated evolutionary search algorithms share many similarities. They are population-oriented: successive populations of candidate solutions are generated in a probabilistic way following laws similar to that of natural evolution. They abstract the basic principles of selection, crossover, mutation, and replication from evolutionary theory. Each string (chromosome) of the subsequent generation is selected from the current one with a probability according to its fitness relative to others in the population. After selection, crossover is applied to two strings (parents) and creates two new strings (offspring). Mutation then performs independently on each string by probabilistically flipping each bit. The process is repeated until some stopping criteria apply. However, these methods also have important differences. Genetic algorithms focus on genetic representation of the problem and apply genetic operators, such as crossover and mutation, to abstracted chromosomes. Evolutionary algorithms, on the other hand, emphasize mutational transformations that maintain behavioural linkage between each parent and its offspring at the level of the individual or the species.

Genetic algorithms (GAs) have been successfully applied in a variety of areas, such as machine learning, neural networks, pattern recognition, combinatorial optimization, scheduling, control and engineering design. A GA possesses several important characteristics that are suited to engineering optimization problems. It is an efficient search technique—use of a chromosome population enables the search to proceed in parallel. The genetic operators bring about information exchange randomly among strings. This reduces the likelihood of termination at a local optimum. A GA is also extremely flexible, as it uses only candidate solution quality to guide its search. Auxiliary problem-specific information, such as derivatives or continuity properties, is not required.

In this paper, the learning capability of a genetic algorithm is examined via the playing of a simple video game. The video game situation provides an effective as well as an approachable environment for testing the learning ability of GAs and other artificial (and computational) intelligence techniques. An introduction to the workings of a genetic algorithm is presented in section 2. In the remainder of this paper, a GA technique is applied to the learning of the falling ball game.

* Department of Electrical Engineering, University of Tulsa, 600 S. College Ave., Tulsa, OK 74104 USA; e-mail: tai@ohm.ee.utulsa.edu.

(paper no. 1994-87)

2. Genetic Algorithms

Genetic algorithms [5–7] are search-and-optimization procedures inspired by natural selection and evolutionary genetics, which enable biological specie to adapt over many generations and become better suited to the environment. They have been demonstrated as effective global optimization tools, especially for multimodal and non-continuous functions [8, 9].

The genetic algorithm evolves a set of elements, called a *population*. Each element $x_i (i = 1, \ldots, N)$, termed a *chromosome*, in the population represents a possible solution of the problem and is assigned a fitness score, $f(x_i)$ according to the objective. The most common representation of x_i is a binary string, which is obtained by concatenating encoded parameters of the solution. Each bit of x_i is called a *gene*, and its value is called an *allelic value*. The desired degree of precision and numerical accuracy determines the appropriate length of the binary coding. Note that other strings, such as character strings or real-valued strings, have also been used for representing solutions.

A GA is capable of maximizing a given fitness function that plays the role of the environment. It works by manipulating the initial population and subsequent ones so that the fitness of the entire population and that of the best chromosome increase from generation to generation. A simple genetic algorithm [7, 8] is typically implemented as follows:

1. The problem to be addressed is defined and captured in a fitness function. A representation for possible solutions must be coded. An initial population of candidate solutions $[x_1, \ldots, x_N]$ is generated.
2. Each string, x_i, in the population is decoded into the actual parameters for evaluation and is assigned a fitness value.
3. The likelihood of each string's being selected for reproduction is proportional to its fitness relative to other strings in the population. Using the *roulette wheel selection (proportional selection)* [7], the probability that string x_i is selected is given by:

$$p_i = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$$

provided that the fitness of each string is a strictly positive number. According to this assigned probability of reproduction, p_i, a mating pool (an intermediate population) is generated by probabilistically selecting strings from the current population.
4. The selected strings generate offspring by use of genetic operators such as crossover and mutation. The standard one-point crossover is performed by selecting a random position within the string and a random pair of strings from the mating pool. The section that appears before the selected position in the first string splices with the section that appears after the selected position in the second string, and vice versa. Other, more sophisticated, crossover operators have been proposed. The last step in creating a new generation is the mutation, which modifies each bit of each new string in the population. The standard mutation is simply the toggling of a single bit in the binary coding.
5. If the available computing time has not expired or the number of generations has not completed, the process proceeds to step 2 and the cycle is repeated. Otherwise the process is terminated.

The above genetic search procedure can be summarized in the flowchart shown in fig. 1.

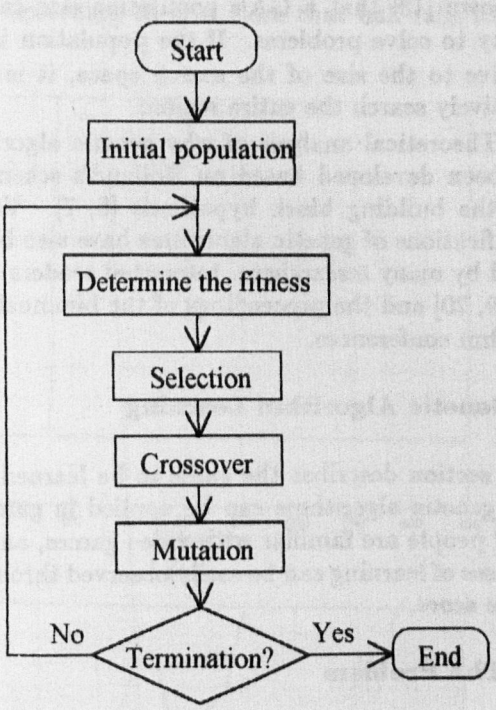

Figure 1. Genetic algorithm flowchart.

Several issues must be addressed when using a genetic algorithm. First, selection in proportion to fitness may cause problems. Using proportional selection, superior strings, as they merge, can guide the population quickly. However, if a superior string surfaces earlier and dominates the population, then that population may converge prematurely on a possibly suboptimal solution. This occurs when the population reaches a configuration such that crossover no longer generates offspring that can outperform their parents. Under such circumstances, crossover simply reproduces the current parents. Schraudolph and Below [14] recently proposed a dynamic parameter encoding (DPE) technique for alleviating the problems associated with premature convergence. The method dynamically resizes the available range of each parameter. Thus DPE can zoom in on solutions that are closer to the global optimum than is provided by the initial precision.

There are other heuristics that have been proposed to overcome the drawback of roulette wheel selection. For example, proportional selection can be based on ranking by fitness. Using this method, no one string can overpower the population in a single generation. But good solutions will be developed slowly.

One other problem with proportional selection is that this procedure cannot guarantee asymptotic convergence to the global minimum in that the best string in the population may be lost at any generation [15]. This can be

overcome by using *elitist selection* [16]. This procedure maintains the best string in the population and ensures asymptotic convergence.

Several researchers have investigated how to choose the best values for GA parameters. The result is generally problem-specific. For a binary-coded GA, typical values for the probabilities of crossover and mutation are in the ranges 0.6~0.9 and 0.001~0.01, respectively [16, 17]. It is known [18] that a GA's population size can affect its ability to solve problems. If the population is too small relative to the size of the search space, it is difficult to effectively search the entire region.

Theoretical analysis of why genetic algorithms work has been developed based on Holland's schema theorem and the building block hypothesis [5, 7]. Variants and modifications of genetic algorithms have also been investigated by many researchers. Interested readers are referred to [19, 20] and the proceedings of the biannual genetic algorithm conferences.

3. Genetic Algorithm Learning

This section describes the game to be learned and shows how genetic algorithms can be applied in game learning. Most people are familiar with video games, and the effectiveness of learning can be easily observed through changes in the score.

3.1 The Problem

The game to be learned is known as the *falling ball game* (see fig. 2). A random selection of green or red balls falls from the top of the computer screen. A pod is located at the bottom of the screen, and can be moved horizontally to the right or the left. Horizontal positions of the balls are completely random. One point is awarded for every green ball that is caught by the pod. However, one point is taken away for every red ball that is caught. Points are neither taken nor given for balls missed.

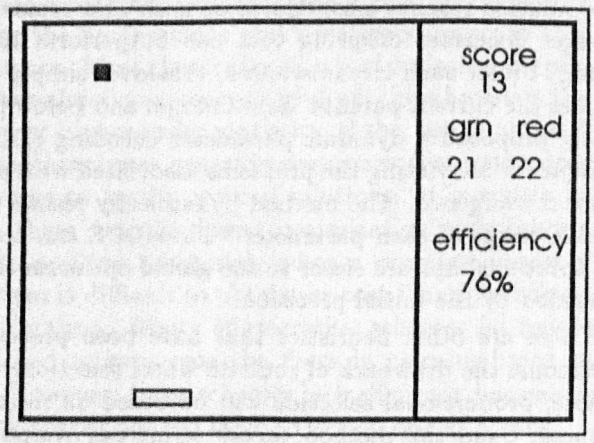

Figure 2. The falling ball game.

The falling ball game is a simple game with a distinct set of input and output data. The inputs to the player, whether human or algorithm, are the pod's distance from green balls, the pod's distance from red balls, and the score. Ideally, the only information supplied to the player prior to playing the game is that scoring is good. The player must link the distances from respective balls to maximize the score.

The playing arena contains 512 horizontal pixels. Therefore, both the balls and the pod can have 512 individual locations. Given that there are two different balls to consider, the input space easily exceeds 1,000 binary-encoded possibilities. The output space contains a similar magnitude of possibilities and is a function of changing score. Once the input data and the output data are determined, an algorithm that will effectively and efficiently handle the situation must be chosen. An artificial neural network probably could do the job; however, this game situation easily fits the criteria for a genetic algorithm.

3.2 The GA Approach

In implementing the genetic algorithm, a string structure and a fitness function must be formulated. How the genetic operators (selection, crossover, and mutation) perform needs to be defined. GA control parameters such as population size, crossover probability, and mutation rate also should be determined.

The chromosomes are binary-encoded data strings. This problem requires two 9-bit substrings (each represents the pod's distance from the ball) to be concatenated into an 18-bit string. Because the objective of the game is to maximize scoring, we employ changes in score in the fitness function. The fitness function is formulated as:

$$f(x) = \frac{1}{1+e^{-\lambda(s/2-g)}} - \frac{1}{1+e^{-\lambda(s/2-r)}} + 1$$

where $x = (g : r)$, s denotes the size of the pod and λ is a positive constant determining the smoothness of the function. g and r denote the distance from the green ball and the red ball to the pod, respectively. One is added to f to ensure non-negative fitness values. It is obvious that the function f increases with increasing score. Hence f satisfies the necessary conditions pointed out in [7] as a fitness funciton.

A discrete representation for the above fitness function is described in table 1. Row 3, for example, indicates that the green is caught but not the red. Hence a fitness value (or score) of 2 is granted because one point is given to successfully achieve the goal. This is just the way a human player conceives the rule.

Following the GA procedures described in section 2, an initial population is generated by randomly selecting ones and zeros to fill N 18-bit strings. The fitness function f is applied to each string in the population to determine its fitness value. Each string is extracted from the previous population with probability p_i, which is proportional to its fitness as a ratio to the average fitness of all strings. To ensure asymptotic convergence, the elitist selection scheme is employed. Note that the best string not only is retained, but also is used to generate new chromosomes.

Table 1

Distance between Green Ball and Pod	Distance between Red Ball and Pod	Score
$g < s/2$	$r < s/2$	1
$g > s/2$	$r > s/2$	1
$g < s/2$	$r > s/2$	2
$g > s/2$	$r < s/2$	0

After the intermediate population is established, the crossover operator is applied. Crossover is performed (different from that described in section 2) by swapping the first section (bits 1 to p) of the first string with the last section (bits 18-p to 18) of the second of the second string (see fig. 3). The resulting pair of strings replaces the original pair in the population. Strings in the intermediate population then go through the standard bit mutation.

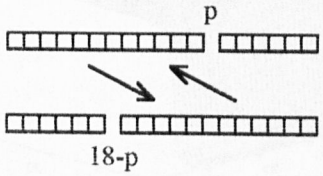

Figure 3. Crossover.

For all simulations of the genetic algorithm, the following parameters were used:

Population size $N = 20$
Crossover probability $= 0.25$
Mutation rate $= 0.01$

4. Results and Discussion

Evalution of the learning process is measured by averaging the fitness values of an entire population. Dividing this average by the highest fitness value yields an effectiveness ratio. The effectiveness of learning can also be observed via score changes during the game play.

Figs. 4(a), (b), and (c) show learning curves of average fitness values corresponding to λ equal to 5, 0.5, and 0.05, respectively. The number of balls fallen corresponds to the generation number evolved in the GA search. Each curve in the same graph represents a complete new game. One indicates that the learning starts early with only a few balls fallen. The other corresponds to the situation in which it takes longer to learn the rules. Once the learning starts, however, it evolves quickly because this game is not complicated. The parameter λ plays an important role in the learning performance. When $\lambda \gg 1$, the fitness function f is approximated by the discrete representation as in table 1. After the algorithm is allowed to play the game, a distinct characteristic is observed. For the discrete fitness function, the algorithm begins to learn after only a few iterations (generations) or after several thousand iterations. Nevertheless, once the learning starts, complete learnig is achieved very quickly, as can be seen in fig. 4(a). When continuous fitness functions (small λ) were used, a steady and consistent learning was obtained. In this application, however, employing continuous fitness functions corresponds to giving the algorithm prior knowledge of how to play the game. This occurs in that game points are only awarded according to whether or not the ball is caught, not according to how close that ball falls to the pod.

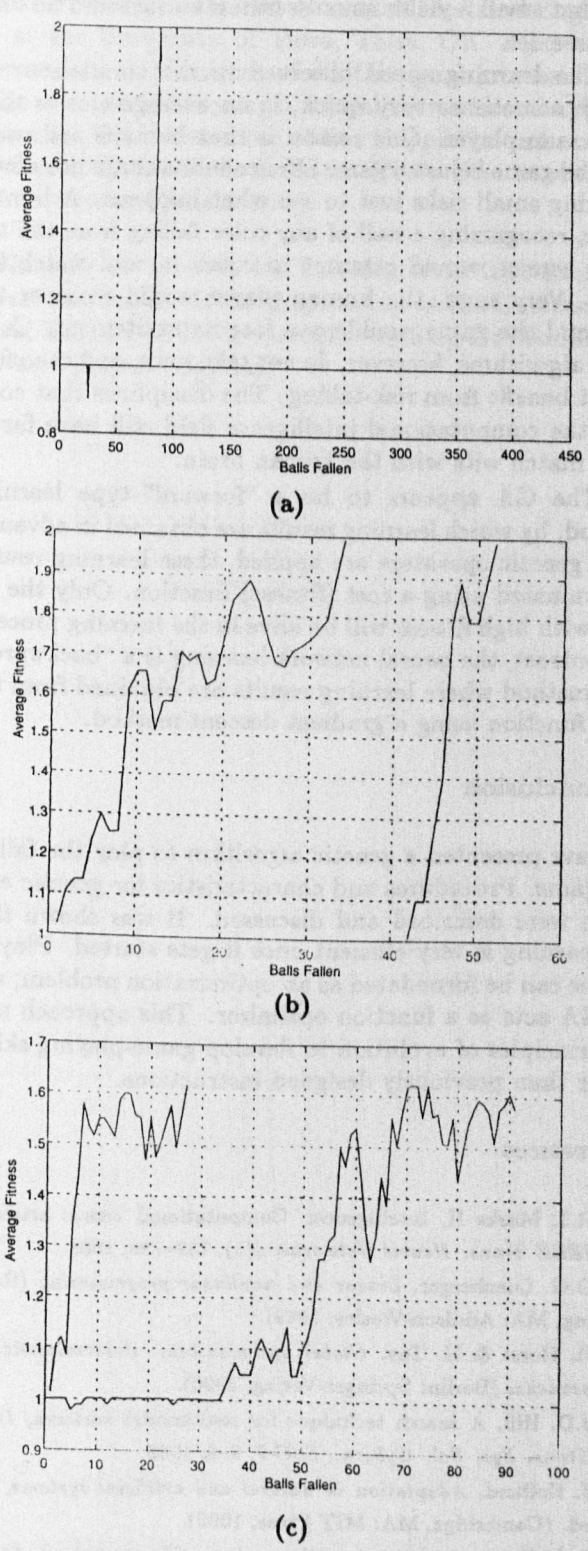

Figure 4. Average fitness values with λ equal to (a) 5, (b) 0.5, (c) 0.05.

The simplicity of the game allows for quick learning in general. To further assess the learning ability of genetic algorithms, we could make the game more complicated. For instance, the pod could be replaced by two side bars. Differential scoring would result depending whether the balls fall between the side bars, hit the side bars, or miss the bars completely. A game of this nature would have more than one local maximum in the fitness surface, in contrast to fig. 5. Figs. 5(a)–(c) are fitness surfaces as functions of g and r for different values of λ. It can be seen that small λ yields smooth fitness surface and no local optima exist.

The learning speed observed in the simulation, although sometimes very quick, is on average slower than the human player. One reason is that humans are aware that the game is just a game. Hence humans are not afraid of taking small risks just to see what happens. A human player, recognizing a ball of any color falling from the top of the screen, would attempt to catch it, and watch the score. Very soon, the human player would discover the rules and the game would soon lose its excitement. Computer algorithms, however, do not take risks, and therefore do not benefit from risk-taking. The disciplines that comprise the computational intelligence field still have far to go to match wits with the human brain.

The GA appears to be a "forward"-type learning method, by which learning results are obtained in advance. After genetic operators are applied, these learning results are evaluated using a cost (fitness) function. Only the results with high fitness will be alive in the learning process. By contrast, the neural network learning is a "backward"-type method where learning results are obtained from the error function using a gradient descent method.

5. Conclusion

We have presented a genetic algorithm to play the falling ball game. Procedures and characteristics for genetic evolution were described and discussed. It was shown that GA learning is very efficient once it gets started. Playing a game can be formulated as an optimization problem; and the GA acts as a function optimizer. This approach uses the principles of evolution to develop game-playing skills, rather than previously designed instructions.

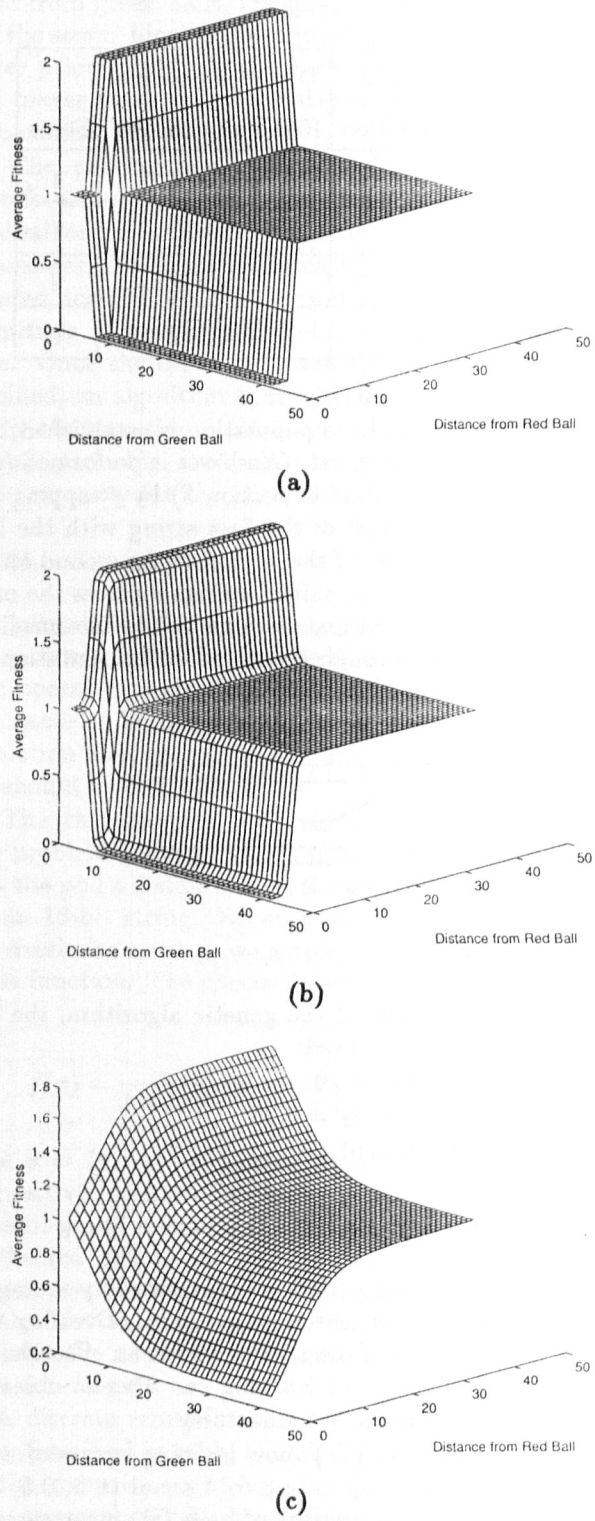

Figure 5. Fitness surfaces for game-learning with λ equal to (a) 5, (b) 0.5, (c) 0.05.

References

[1] R.J. Marks II, Intelligence: Computational versus artificial, *IEEE Trans. Neural Networks*, *4*(5), 737–739, 1993.

[2] D.G. Luenberger, *Linear and nonlinear programming* (Reading, MA: Addison-Wesley, 1989).

[3] R. Horst & H. Tuy, *Global optimization: Deterministic approaches* (Berlin: Springer-Verlag, 1990).

[4] J.D. Hill, A search technique for multimodal surfaces, *IEEE Trans. Sys. Sci. Cybern.*, *SSC-3*, 2–8, 1969.

[5] J. Holland, *Adaptation in natural and artificial systems*, 2nd ed. (Cambridge, MA: MIT Press, 1992).

[6] J. Holland, Genetic algorithms, *Scientific American*, 66–72, July 1992.

[7] D.E. Goldberg, *Genetic algorithms in search, optimization, and machine learning* (Reading, MA: Addison-Wesley, 1989).

[8] D.B. Fogel, An introduction to simulated evolutionary optimization, *IEEE Trans. on Neural Networks*, *5*(1), 3–14, 1994.

[9] I. Rechenberg, *Evolutionsstrategie: Optimierung Technischer Systeme nachPrinzipien der Biologischen Evolution* (Stuttgart: Frommann-Holzboog, 1973).

[10] H.-P. Schwefel, *Numerical optimization of computer models* (Chichester, UK: John Wiley, 1981).

[11] L.J. Fogel, A. Owens, & M. Walsh, *Artificial intelligence through simulated evolution* (New York: John Wiley, 1966).

[12] D.B. Fogel, *System identification through simulated evolution: A machine learning approach to modeling* (Needham Heights, MA: Ginn Press, 1991).

[13] H. Muhlenbein, Parallel genetic algorithms, population genetics and combinatorial optimization, *Proc. 3rd Int. Conf. on Genetic Algorithms*, San Mateo, CA, 1989, 416–421.

[14] N.N. Schraudolph & R.K. Below, Dynamic parameter encoding for genetic algorithms, *Machine Learning, 9*, 9–21, 1992.

[15] G. Rudolph, Convergence properties of canonical genetic algorithms, *IEEE Trans. on Neural Networks, 5*(1), 96–101, 1994.

[16] J.J. Grefenstette, Optimization of control parameters for genetic algorithms, *IEEE Trans. Sys., Man and Cybern., 16*, 122–128, 1986.

[17] K.A. DeJong, *The analysis of the behavior of a class of genetic adaptive systems*, doctoral diss., University of Michigan, Ann Arbor, 1975.

[18] D.E. Goldberg, Sizing populations for serial and parallel genetic algorithms, *Proc. 3rd Int. Conf. on Genetic Algorithms*, San Mateo, CA, 1989, 70–77.

[19] G. Rawlins, ed., *Foundations of genetic algorithms 1* (San Mateo, CA: Morgan Kaufmann, 1991).

[20] D. Whitley, ed., *Foundations of genetic algorithms 2* (San Mateo, CA: Morgan Kaufmann, 1993).

Biographies

Heng-Ming Tai received his B.S. degree from National Tsing-Hua University, Hsinchu, Taiwan, and his M.S. and Ph.D. degrees from Texas Technical University, Lubbock, Texas, all in electrical engineering. He is currently an Associate Professor in the Department of Electrical Engineering at the University of Tulsa, Tulsa, OK. His research interests are in the areas of digital signal processing, intelligent control, fuzzy logic, neural networks and applications. Dr. Tai is a member of Eta Kappa Nu and a senior member of IEEE. He is listed in *Who's Who in America and Science and Engineering.*

David Robison received his B.S. degree in 1994 from the Department of Electrical Engineering at the University of Tulsa, Tulsa, OK. He is currently working for WorldCom, Inc. as a project engineer.