Designing an Evolutionary Strategizing Machine for Game Playing and Beyond

Moshe Sipper, Yaniv Azaria, Ami Hauptman, and Yehonatan Shichel

***Abstract*— We have recently shown that genetically programming game players, after having imbued the evolutionary process with human intelligence, produces human-competitive strategies for three games: backgammon, chess endgames, and robocode (tank-fight simulation). Evolved game players are able to hold their own—and often win—against human or human-based competitors. This paper has a twofold objective: first, to review our recent results of applying genetic programming in the domain of games; second, to formulate the merits of genetic programming in acting as a tool for developing strategies in general, and to discuss the possible design of a strategizing machine.**

***Index Terms*— Evolutionary algorithms, Genetic programming, Backgammon, Chess, Robocode, Evolving game strategies, Strategizing.**

I. Introduction

Ever since the dawn of artificial intelligence in the 1950s, games have been part and parcel of this lively field. In 1957, a year after the Dartmouth Conference that marked the official birth of AI, Alex Bernstein designed a program for the IBM 704 that played two amateur games of chess. In 1958 Allen Newell, J. C. Shaw, and Herbert Simon reported a more sophisticated chess program (beaten in thirty-five moves by a ten-year-old beginner in its last official game played in 1960). Arthur L. Samuel of IBM spent much of the fifties working on game-playing AI programs, and by 1961 he had a checkers program that could play at the master's level. In 1961 and 1963 Donald Michie described a simple trial-and-error learning system for learning how to play Tic-Tac-Toe (Noughts and Crosses) called MENACE (for Matchbox Educable Noughts and Crosses Engine). These are but examples of highly popular games that have been treated by AI researchers since the field's inception.

Why study games? This question was answered by Susan L. Epstein, who wrote:

> There are two principal reasons to continue to do research on games... First, human fascination with game playing is long-standing and pervasive. Anthropologists have catalogued popular games in almost every culture... Games intrigue us because they address important cognitive functions... The second reason to continue game-playing research is that some difficult games remain to be won, games that people play very well but computers do not. These games clarify what our current approach lacks. They set challenges for us to meet, and they promise ample rewards. [1]

Studying games may thus advance our knowledge in both cognition and artificial intelligence, and, last but not least, games possess a competitive angle which coincides with our human nature, thus motivating both researcher and student alike.

Even more strongly, Laird and van Lent proclaimed that,

> ... interactive computer games are the killer application for human-level AI. They are the application that will soon need human-level AI, and they can provide the environments for research on the right kinds of problems that lead to the type of the incremental and integrative research needed to achieve human-level AI. [2]

In this paper we apply an evolutionary algorithm to the study and evolution of game-playing strategies. The idea of applying the biological principle of natural evolution to artificial systems, introduced more than four decades ago, has seen impressive growth in the past few years [3]. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming [3–10]. Evolutionary algorithms are common nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, circuit design, machine learning, economics, immune systems, ecology, and population genetics, to mention but a few. Herein, we concentrate on the evolutionary methodology of genetic programming [8], wherein a population of computer programs is evolved.

This paper has a twofold objective. First, to review our recent results on the evolution of winning strategies for three games: backgammon, chess (endgames), and robocode (tank-fight simulation). This review is written with our second objective in mind: to formulate the merits of genetic programming in acting as a tool for developing strategies in general, and to discuss the possible design of a strategizing machine. The first objective is attained in Section III, while the second is addressed in Section IV. First, however, we provide a brief introduction to genetic programming in the next section.

II. Genetic Programming

Genetic Programming is a sub-class of evolutionary algorithms, introduced by Cramer [11], and transformed into a field in its own right in large part due to the efforts of John Koza. In genetic programming we evolve a *population* of individual

The authors are with the Department of Computer Science, Ben-Gurion University, P.O. Box 653, Beer-Sheva 84105, Israel. E-mails: {sipper, azariaya, amiha, shichel}@cs.bgu.ac.il.

LISP expressions[1], each comprising *functions* and *terminals*. The functions are usually arithmetic and logic operators that receive a number of arguments as input and compute a result as output; the terminals are zero-argument functions that serve both as constants and as sensors, the latter being a special type of function that queries the domain environment.

With terminals we often use the ERC (*Ephemeral Random Constant*) mechanism, as described in Koza [8]. When first initialized, an ERC node is randomly assigned a constant value from a given range; this value does not change during evolution, unless a mutation operator is applied.

The main mechanism behind genetic programming is precisely that of a generic evolutionary algorithm [3], [4], namely, the repeated cycling through four operations applied to the entire population: evaluate-select-crossover-mutate. Starting with an initial population of randomly generated LISP programs, each individual is evaluated in the domain environment and assigned a *fitness* value representing how well the individual solves the problem at hand. Being randomly generated, the first-generation individuals usually exhibit poor performance. However, some individuals are better than others, i.e., (as in nature) variability exists, and through the mechanism of natural (or, in our case, artificial) selection, these have a higher probability of being selected to parent the next generation. The size of the population is finite and usually constant.

Specifically, first a genetic operator is chosen at random; then, depending on the operator, one or two individuals are selected from the current population using a *selection operator*, one example of which is *tournament selection*: Randomly choose a small subset of individuals, and then select the one with the best fitness. After the probabilistic selection of better individuals *genetic operators* are used to construct the next generation. The most common operators are:

- Reproduction (unary): Copy one individual to the next generation with no modifications. The main purpose of this operator is to preserve a small number of good individuals.
- Crossover (binary): Randomly select an internal node in each of the two individuals and swap the sub-trees rooted at these nodes. An example is shown in Figure 1.
- Mutation (unary): Randomly select a node from the tree, delete the subtree rooted at that node, and then "grow" a new sub-tree in its stead. An example is shown in Figure 1 (the growth operator as well as crossover and mutation are described in detail in Koza [8]).

The generic genetic programming flowchart is shown in Figure 2. When one wishes to employ genetic programming one need define the following six desiderata:

1) program architecture,
2) set of terminals,
3) set of functions,
4) fitness measure,
5) control parameters,
6) manner of designating result and terminating run.

[1]Languages other than LISP have been used, although LISP is still by far the most popular within the genetic programming domain.

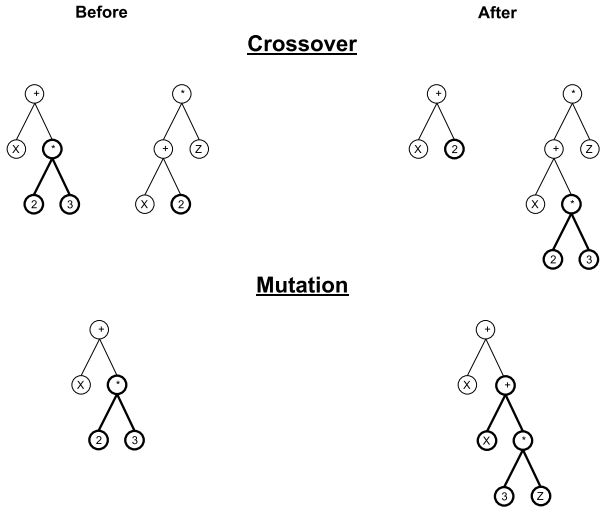

Fig. 1. Genetic operators in genetic programming. LISP programs are depicted as trees. Crossover (top): Two sub-trees (marked in bold) are selected from the parents and swapped. Mutation (bottom): A sub-tree (marked in bold) is selected from the parent individual and removed. A new sub-tree is grown instead.

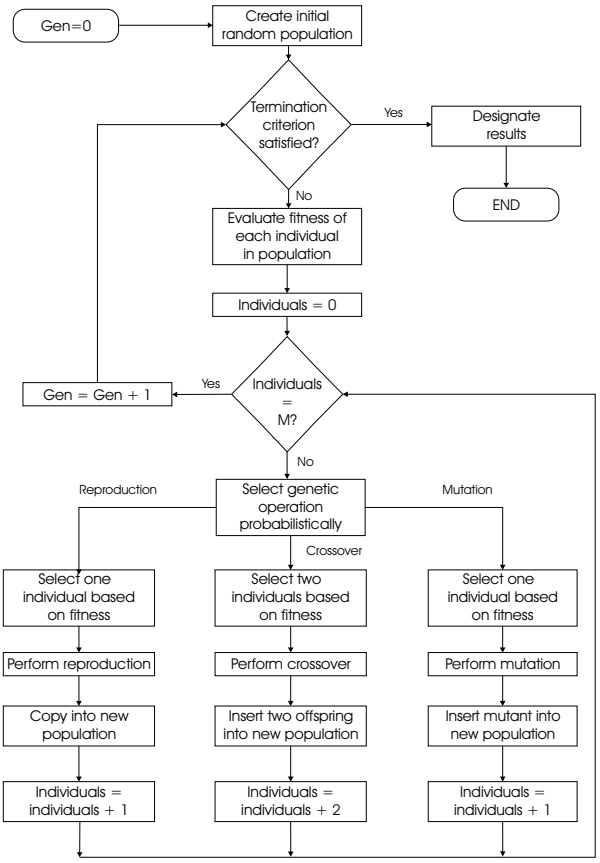

Fig. 2. Generic genetic programming flowchart (based on Koza [8]). M is the population size and Gen is the generation counter. The termination criterion can be the completion of a fixed number of generations or the discovery of a good-enough individual.

III. BACKGAMMON, CHESS, ROBOCODE: A WHOLLY TRINITY

In this section we present a summary of our recent work on attaining (limited) machine intelligence in the domain of games via the evolutionary methodology of genetic programming. We focused on three games [12–14]:

1) **Backgammon.** Evolve a full-fledged player for the non-doubling-cube version of the game [12].
2) **Chess** (endgames). Evolve a player able to play endgames [13]. While endgames typically contain but a few pieces, the problem of evaluation is still hard, as the pieces are usually free to move all over the board, resulting in complex game trees—both deep and with high branching factors. Indeed, in the chess lore much has been said and written about endgames.
3) **Robocode.** A simulation-based game in which robotic tanks fight to destruction in a closed arena (`robocode.alphaworks.ibm.com`). The programmers implement their robots in the Java programming language, and can test their creations either by using a graphical environment in which battles are held, or by submitting them to a central web site where online tournaments regularly take place. Our goal here has been to evolve robocode players able to rank high in the international league [14].

A strategy for a given player in a game is a way of specifying which choice the player is to make at every point in the game from the set of allowable choices at that point, given all the information that is available to the player at that point [8]. The problem of discovering a strategy for playing a game can be viewed as one of seeking a computer program. Depending on the game, the program might take as input the entire history of past moves or just the current state of the game. The desired program then produces the next move as output. For some games one might evolve a complete strategy that addresses every situation tackled. This proved to work well with robocode, which is a dynamic game, with relatively few parameters, and little need for past history.

Another approach (which can probably be traced back to Samuel [15]) is to couple a current-state evaluator (e.g., board evaluator) with a next-move generator. One can go on to create a minimax tree, which consists of all possible moves, counter moves, counter counter-moves, and so on; for real-life games, such a tree's size quickly becomes prohibitive. Deep Blue, the famous machine chess player, and its offspring Deeper Blue, rely mainly on brute-force methods to gain an advantage over the opponent, by traversing as deeply as possible the game tree [16]. Although these programs have achieved amazing performance levels, Chomsky [17] has criticized this aspect of game-playing research as being "about as interesting as the fact that a bulldozer can lift more than some weight lifter." The approach we used herein with backgammon and chess is to derive a very shallow, one-level tree, and evolve "smart" evaluation functions. Our artificial player is thus had by combining an evolved board evaluator with a (relatively simple) program that generates all next-move boards (such programs can easily be written for backgammon and chess).

In what follows we describe the definition of the six items necessary in order to employ genetic programming, as delineated in the previous section: program architecture, set of terminals, set of functions, fitness measure, control parameters, and manner of designating result and terminating run.

A. Program Architecture

Backgammon. The game of backgammon can be observed to consist of two main stages: the 'contact' stage, where the two players can hit each other, and the 'race' stage, where there is no contact between the two players. During the contact stage we expect a good strategy to block the opponent's progress and minimize the probability of getting hit. On the other hand, during the race stage, blocks and blots are of no import, rather, one aims to select moves that lead to the removal of a maximum number of pieces off the board.

This observation directed us in designing the genomic structure of individuals in the population. Each individual contains a contact tree and a race tree. When a board is evaluated, the program checks whether there is any contact between the players and then evaluates the tree that is applicable to the current board state. The terminal set of the contact tree is richer and contains various general and specific board query functions. The terminal set of the race tree is much smaller and contains only terminals that examine the checkers' positions. This is because at the race phase, the moves of each player are mostly independent of the opponent's status, and thus are much simpler.

Chess. As most chess players would agree, playing a winning position (e.g., with material advantage) is quite different than playing a losing position, or an even one. For this reason each individual contains three trees: an advantage tree, an even tree, and a disadvantage tree. These trees are used according to the current status of the board. The disadvantage tree is smaller because achieving a stalemate and avoiding exchanges requires less complicated reasoning.

Robocode. A robocode player is written as an event-driven Java program. A main loop controls the tank activities, which can be interrupted on various occasions, called *events*. The program is limited to four lines of code, as we were aiming for the HaikuBot category, one of the divisions of the international league with a four-line code limit.[2] The main loop contains one line of code that directs the robot to start turning the gun (and the mounted radar) to the right. This insures that within the first gun cycle, an enemy tank will be spotted by the radar, triggering a *ScannedRobotEvent*. Within the code for this event, three additional lines of code were added, each controlling a single actuator, and using a single numerical input that was supplied by a genetic programming-evolved sub-program. The first line instructs the tank to move to a distance specified by the first evolved argument. The second line instructs the tank to turn to an azimuth specified by the second evolved argument. The third line instructs the gun (and

[2]Other divisions are: NanoBots—limited to 250 bytes, MicroBots—limited to 750 bytes, MiniBots—limited to 1500 bytes, Sonnets—limited to 14 lines of code, and FemtoBots—with code size as small as effectively possible; see `http://robocode.yajags.com/divisions.php`.

Robocode Player's Code Layout

```
while (true)
    TurnGunRight(INFINITY); //main code loop
...
OnScannedRobot() {
    MoveTank(<GP#1>);
    TurnTankRight(<GP#2>);
    TurnGunRight(<GP#3>);
}
```

Fig. 3. Robocode player's code layout (HaikuBot division).

Float-ERC()	ERC – random real constant in range [0,5]
Player-Exposed(*n*)	If player has exactly one checker at location *n*, return 1, else return 0
Player-Blocked(*n*)	If player has two or more checkers at location *n*, return 1, else return 0
Player-Tower(*n*)	If player has h or more checkers at location *n* (where $h \geq 3$), return $h - 2$, else return 0
Enemy-Exposed(*n*)	If enemy has exactly one checker at location *n*, return 1, else return 0
Enemy-Blocked(*n*)	If enemy has two or more checkers at location *n*, return 1, else return 0
Player-Pip()	Return player *pip-count* divided by 167 (initial pip count)
Enemy-Pip()	Return enemy *pip-count* divided by 167 (initial pip count)
Total-Hit-Prob()	Return sum of hit probability over all exposed player checkers
Player-Escape()	Measure the effectiveness of the enemy's barrier over its home positions
Enemy-Escape()	Measure the effectiveness of the player's barrier over its home positions

(a)

Float-ERC()	ERC – random real constant in range [0,5]
Player-Position(*n*)	Return number of checkers at location *n*

(b)

Add(F, F)	Add two real numbers
Sub(F, F)	Subtract two real numbers
Mul(F, F)	Multiply two real numbers
If(B, F, F)	If first argument evaluates to a non-zero value, return value of second argument, else return value of third argument
Greater(F, F)	If first argument is greater than second, return 1, else return 0
Smaller(F, F)	If first argument is smaller than second, return 1, else return 0
And(B, B)	If both arguments evaluate to a non-zero value, return 1, else return 0
Or(B, B)	If at least one of the arguments evaluates to a non-zero value, return 1, else return 0
Not(B)	If argument evaluates to zero, return 1, else return 0

(c)

Fig. 4. Backgammon representation. a) Terminal set of the contact tree. Note that zero-argument functions—which serve both as constants and as sensors—are considered as terminals. b) Terminal set of the race tree. c) Function set of both trees (B: Boolean, F: Float).

radar) to turn to an azimuth specified by the third evolved argument (Figure 3).

B. Terminal and Function Sets

The terminal and function sets for the three games are given in Figures 4 (backgammon), 6 (chess), and 7 (robocode); each terminal and function is accompanied by a short explanation. The functions mostly perform simple arithmetic and logic operations, while the terminals embody the bulk of the imbued human intelligence.

Backgammon. The terminal set contains three types of terminals [12] (Figure 4):

1) The Float-ERC function calls upon ERC (Section II) directly. When created, the terminal is assigned a constant, real-number value, which becomes the return value of the terminal.

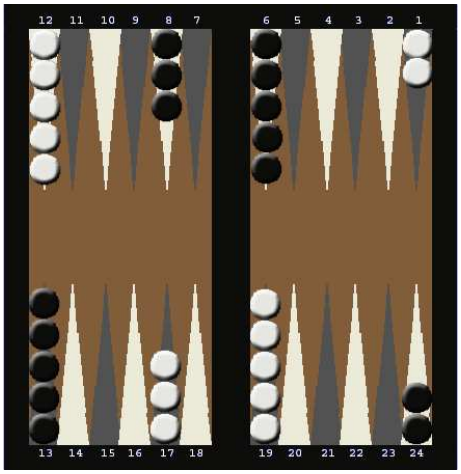

Fig. 5. Initial backgammon configuration. The White player's home positions are labeled 19-24, and the Black player's home positions are labeled 1-6.

2) The board-position query terminals use the ERC mechanism to query a specific location on the board. When initialized, a value between 0 and 25 is randomly chosen, where 0 specifies the bar location, 1-24 specify the inner board locations, and 25 specifies the off-board location (Figure 5). The term 'Player' refers to the contender whose turn it is, while 'Enemy' refers to the opponent. When a board query terminal is evaluated, it refers to the board location that is associated with the terminal, from the player's point of view.
3) For the last type of terminal we took advantage of one of genetic programming's most powerful attributes: The ability to easily add non-trivial functions that provide useful information about the domain environment. In our case, these terminals are functions that provide general information about the board as a whole, *e.g.*, how far is the player from winning, and an estimation of the risk of being hit by the enemy.

Chess. We developed most of our chess terminals by consulting several high-ranking chess players.[3] The terminal set examines various aspects of the chessboard, and may be divided into three groups [13] (Figure 6):

1) Float values, created using the ERC mechanism (Section II). An ERC is chosen at random to be one of the following six values $\pm 1 \cdot \{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\} \cdot MAX$ (MAX was empirically set to 1000), and the inverses of these numbers. This guarantees that when a value is returned after some group of features has been identified, it will be distinct enough to engender the outcome.
2) Simple terminals, which analyze relatively simple aspects of the board, such as the number of possible moves for each king, and the number of attacked pieces for each player. These terminals were derived by breaking relatively complex aspects of the board into simpler notions. More complex terminals belong to the next group (see below). For example, a player should capture

[3]The highest-ranking player we consulted was Boris Gutkin, ELO 2400, International Master, and fully qualified chess teacher.

NotMyKingInCheck()	Is the player's king not being checked?
IsOppKingInCheck()	Is the opponent's king being checked?
MyKingDistEdges()	The player's king's distance form the edges of the board
OppKingProximityToEdges()	The player's king's proximity to the edges of the board
NumMyPiecesNotAttacked()	The number of the player's pieces that are not attacked
NumOppPiecesAttacked()	The number of the opponent's attacked pieces
ValueMyPiecesAttacking()	The material value of the player's pieces which are attacking
ValueOppPiecesAttacking()	The material value of the opponent's pieces which are attacking
IsMyQueenNotAttacked()	Is the player's queen not attacked?
IsOppQueenAttacked()	Is the opponent's queen attacked?
IsMyFork()	Is the player creating a fork?
IsOppNotFork()	Is the opponent not creating a fork?
NumMovesMyKing()	The number of legal moves for the player's king
NumNotMovesOppKing()	The number of illegal moves for the opponent's king
MyKingProxRook()	Proximity of my king and rook(s)
OppKingDistRook()	Distance between opponent's king and rook(s)
MyPiecesSameLine()	Are two or more of the player's pieces protecting each other?
OppPiecesNotSameLine()	Are two or more of the opponent's pieces protecting each other?
IsOppKingProtectingPiece()	Is the opponent's king protecting one of his pieces?
IsMyKingProtectingPiece()	Is the player's king protecting one of his pieces?

(a)

EvaluateMaterial()	The material value of the board
IsMaterialIncrease()	Did the player capture a piece?
IsMate()	Is this a mate position?
IsMateInOne()	Can the opponent mate the player after this move?
OppPieceCanBeCaptured()	Is it possible to capture one of the opponent's pieces without retaliation?
MyPieceCannotBeCaptured()	Is it not possible to capture one of the player's pieces without retaliation?
IsOppKingStuck()	Do all legal moves for the opponent's king advance it closer to the edges?
IsMyKingNotStuck()	Is there a legal move for the player's king that advances it away from the edges?
IsOppKingBehindPiece()	Is the opponent's king two or more squares behind one of his pieces?
IsMyKingNotBehindPiece()	Is the player's king not two or more squares behind one of my pieces?
IsOppPiecePinned()	Is one or more of the opponent's pieces pinned?
IsMyPieceNotPinned()	Are all the player's pieces not pinned?

(b)

If(B, F, F)	If first argument evaluates to a non-zero value, return value of second argument, else return value of third argument
Smaller(F, F)	If first argument is smaller than second, return 1, else return 0
And(B, B)	If both arguments evaluate to a non-zero value, return 1, else return 0
And3(B, B, B)	If all arguments evaluate to a non-zero value, return 1, else return 0
Or(B, B)	If at least one of the arguments evaluates to a non-zero value, return 1, else return 0
Or3(B, B, B)	If at least one of the arguments evaluates to a non-zero value, return 1, else return 0
Not(B)	If argument evaluates to zero, return 1, else return 0

(c)

Fig. 6. Chess representation. Opp: opponent, My: player. a) Simple terminals, which analyze relatively simple aspects of the board. b) Complex terminals, which check upon aspects a human player would. c) Function set (B: Boolean, F: Float).

his opponent's piece if it is not sufficiently protected, meaning that the number of attacking pieces the player controls is greater than the number of pieces protecting the opponent's piece, and the material value of the defending pieces is equal to or greater than the player's. Adjudicating these considerations is not simple, and therefore a terminal that performs this entire computational feat by itself belongs to the next group of complex terminals.

The simple terminals comprising this second group are derived by refining the logical resolution of the previous paragraphs' reasoning: Is an opponent's piece attacked? How many of the player's pieces are attacking that piece? How many pieces are protecting a given opponent's piece? What is the material value of pieces attacking and defending a given opponent's piece? All these questions are embodied as terminals within the second group. The ability to easily embody such reasoning within the genetic programming setup, as functions and terminals, is a major asset of genetic programming. Other terminals were also derived in a similar manner (Figure 6). Note that some of the terminals are inverted—we would like terminals to always return positive (or true) values, since these values represent a favorable position. This is why we used, for example, a terminal evaluating the player's king's *distance* from the edges of the board (generally a favorable feature for endgames), while using a terminal evaluating the *proximity* of the opponent's king to the edges (again, a positive feature).

3) Complex terminals. These are terminals that check the same aspects of the board a human player would. Some prominent examples include: OppPieceCanBeCaptured, considering the capture of a piece; checking if the current position is a draw, a mate, or a stalemate (especially important for non-even boards); checking if there is a mate in one or two moves (this is the most complex terminal); the material value of the position; comparing the material value of the position to the original board—this is important since it is easier to consider change than to evaluate the board in an absolute manner.

   Since some of these terminals are hard to compute, and most appear more than once in the individual's trees, we used a memoization scheme to save time [18]: After the first calculation of each terminal, the result is stored, so that further calls to the same terminal (on the same board) do not repeat the calculation. Memoization greatly reduced the evolutionary run-time.

Robocode. We divided the terminals into (again...) three groups according to their functionality [14] (Figure 7):

1) Game-status indicators: A set of terminals that provide real-time information on the game status, such as last enemy azimuth, current tank position, and energy levels.
2) Numerical constants: Two terminals, one providing the constant 0, the other being an ERC (Ephemeral Random Constant). This latter terminal is initialized to a random real numerical value in the range [-1, 1], and does not

Energy()	Returns the remaining energy of the player
Heading()	Returns the current heading of the player
X()	Returns the current horizontal position of the player
Y()	Returns the current vertical position of the player
MaxX()	Returns the horizontal battlefield dimension
MaxY()	Returns the vertical battlefield dimension
EnemyBearing()	Returns the current enemy bearing, relative to the current player's heading
EnemyDistance()	Returns the current distance to the enemy
EnemyVelocity()	Returns the current enemy's velocity
EnemyHeading()	Returns the current enemy heading, relative to the current player's heading
EnemyEnergy()	Returns the remaining energy of the enemy
Constant()	An ERC in the range [-1, 1]
Random()	Returns a random real number in the range [-1, 1]
Zero()	Returns the constant 0

(a)

Add(F, F)	Add two real numbers
Sub(F, F)	Subtract two real numbers
Mul(F, F)	Multiply two real numbers
Div(F, F)	Divide first argument by second, if denominator non-zero, otherwise return zero
Abs(F)	Absolute value
Neg(F)	Negative value
Sin(F)	Sine function
Cos(F)	Cosine function
ArcSin(F)	Arcsine function
ArcCos(F)	Arccosine function
IfGreater(F, F, F, F)	If first argument greater than second, return value of third argument, else return value of fourth argument
IfPositive(F, F, F)	If first argument is positive, return value of second argument, else return value of third argument
Fire(F)	If argument is positive, execute fire command with argument as firepower and return 1; otherwise, do nothing and return 0

(b)

Fig. 7. Robocode representation. a) Terminal set. b) Function set (F: Float).

change during evolution.

3) Fire command: This special function is used to curtail one line of code by not implementing the fire actuator in a dedicated line.

C. Fitness Measure

Backgammon. We explored two different modes of learning: using a fixed external opponent as teacher, and coevolution—letting the individuals play against each other. As external opponent (and later for benchmark purposes as well) we used *Pubeval*, a free, public-domain board evaluation function written by Tesauro [19]. The program—which plays well—has become the *de facto* yardstick used by the growing community of backgammon-playing program developers. Coevolution refers to the simultaneous evolution of individuals—from different species (populations) or from the same species (population)—wherein fitness is coupled. Such coupled evolution favors the discovery of complex solutions whenever complex solutions are required [20]. Simplistically speaking, one can say that coevolving species can either compete (e.g., to obtain exclusivity on a limited resource, or in a gaming scenario) [21], [22] or cooperate (e.g., to gain access to some hard-to-attain resource) [23], [24]. We used single-species competitive coevolution, which proved to be better than the external-opponent approach.

To evaluate fitness under coevolution we used the Single Elimination Tournament method [25]: Start with a population of n individuals, n being a power of two. Then, divide the individuals into $\frac{n}{2}$ arbitrary pairs, and let each pair engage in a relatively short tournament of 50 games. Finally, set the fitness of the $\frac{n}{2}$ losers to $\frac{1}{n}$. The remaining $\frac{n}{2}$ winners are divided into pairs again, engage in tournaments as before, and the losers are assigned fitness values of $\frac{1}{n/2}$. This process continues until one champion individual remains. Thus, the more tournaments an individual "survives," the higher its fitness.

TABLE I

CONTROL PARAMETERS.

	Backgammon	Chess	Robocode
Population size	128[a]	80	256
Generation count	500	150 – 250	100-200[b]
Selection method	tournament	tournament	tournament
Reproduction probability	0.1	0.35	0
Crossover probability	0.65	0.5	0.95
Mutation probability	0.25	0.15	0.05

[a]We first used a single population, later increased to 50 populations when more extensive computing resources were placed at our disposal.

[b]We manually stopped a run when fitness was observed to level off.

Chess. We used coevolution, with the fitness of an individual determined by its success against its peers. Specifically, we employed the random-2-ways method [26], in which each individual plays against a fixed number of randomly selected peers (typically 5). The scoring method was based on the one used in chess tournaments: victory—1 point, draw—$\frac{1}{2}$ point, loss—0 points. In order to better differentiate our players, we rewarded $\frac{3}{4}$ points for a material advantage (without mating the opponent). Strategies were first evolved to play one type of endgame, and then to play multiple endgames. The former means that the same pieces (one endgame type) were used as starting board, with their positions changing randomly, while the latter means that several combinations of pieces (several endgame types) were used, their placement also being random. Since random starting positions can sometimes be uneven (for example, allowing the starting player to attain a capture position), every starting position was played twice, each player playing both Black and White. This way a better starting position could benefit both players and the tournament was less biased (this stratagem was adopted for both fitness evaluation and post-evolutionary benchmarking).

Robocode. Again, we experimented with both external opponent and coevolution, with—as opposed to backgammon—the former proving better. However, not one external opponent was used to measure performance but three, these adversaries downloaded from the HaikuBot league (`robocode.yajags.com`). The fitness value of an individual equals its average fractional score (over three battles).

D. Control Parameters

The control parameters for all three games are summarized in Table I.

E. Result Designation and Run Termination

Backgammon. Run terminates when generation count is reached. Every five generations we pitted the four individuals with the highest fitness in a 1000-game tournament against

TABLE II
COMPARISON OF BACKGAMMON PLAYERS: WIN PERCENTAGE IN TOURNAMENT AGAINST PUBEVAL.

Player	**%Wins**
GP-Gammon	62.4
Darwen [27]	52.7
GMARLB-Gammon [28]	51.2
ACT-R-Gammon [29]	45.94
HC-Gammon [30]	40.00

TABLE III
PERCENT OF WINS, ADVANTAGES, AND DRAWS FOR BEST GP-ENDCHESS PLAYER IN TOURNAMENT AGAINST TWO TOP COMPETITORS.

	%Wins	%Advs	%Draws
Master	6.00	2.00	68.00
CRAFTY	2.00	4.00	72.00

Pubeval, and the individual with the highest score in these tournaments, over the entire evolutionary run, was declared best-of-run.

Chess. Run terminates when generation count is reached. Every 10 generations the best individual was extracted and pitted in a 150-game tournament against two very strong external opponents: 1) A program we wrote ('Master') based upon consultation with several high-ranking chess players (the highest being Boris Gutkin, ELO 2400, International Master); 2) CRAFTY—a world-class chess program, which finished second in the 2004 World Computer Speed Chess Championship (`www.cs.biu.ac.il/games/`). Speed chess ("blitz") involves a time-limit per move, which we imposed both on CRAFTY and on our players. Not only did we thus seek to evolve good players, but ones that play well *and fast*.

Robocode. Run terminates when fitness is observed to level off. Since the game is highly nondeterministic a "lucky" individual might attain a higher fitness value than better overall individuals. In order to obtain a more accurate measure for the evolved players we let each of them do battle for 100 rounds against 12 different adversaries (one at a time). The results were used to extract the top player—to be submitted to the international league.

F. Results

Backgammon. Our top evolved player was able to attain a win percentage of 62.4% in a tournament against Pubeval, about 10% higher than the previous top method. Moreover, several evolved strategies were able to surpass the 60% mark, and most of them outdid all previous works (see Table II for comparison).

Chess. Results for our best multiple-endgame evolved strategy are shown in Table III.

Robocode. We submitted our top player to the HaikuBot division of the international league. At its first tournament it came in third, with all other programs being human-written. Our GP-Robocode later came in second of 28 in two subsequent contests (one is held every week; see Table IV).

IV. DESIGNING A STRATEGIZING MACHINE

In their book, Koza *et al.* [31] delineate 16 attributes a system for automatically creating computer programs might beneficially possess:

1) Starts with problem requirements.
2) Produces tractable and viable solution to problem.
3) Produces an executable computer program.
4) Automatic determination of program size.
5) Code reuse.
6) Parameterized reuse.
7) Internal storage.
8) Iterations, loops, and recursions.
9) The ability to organize chunks of code into hierarchies.
10) Automatic determination of program architecture.
11) Ability to implement a wide range of programming constructs.
12) Operates in a well-defined manner.
13) Problem-independent, i.e., possesses some degree of generalization capabilities.
14) Applicable to a wide variety of problems from different domains.
15) Able to scale well to larger instances of a given problem.
16) Competitive with human-produced results.

Our own work has prompted us to suggest an additional attribute to this list:

17) Cooperative with humans.

We believe that a major reason for our success in evolving winning game strategies is genetic programming's ability to readily accommodate human expertise in the *language of design.* Ronald, Sipper, and Capcarrère defined this latter term within the framework of their proposed *emergence test* [32]. The test involves two separate languages—one used to *design* a system, the other used to describe *observations* of its (putative) emergent behavior. The effect of surprise arising from the gap between design and observation is at the heart of the emergence test (for details see [32]). Our languages of design for the three games possess several functions and terminals that attest to the presence of a (self-proclaimed) intelligent designer. These design languages, which give rise to powerful languages of observation in the form of successful players, were designed not instantaneously—like Athena springing from Zeus's head fully grown—but rather through an incremental, interactive process, whereby man (represented by the humble authors of this paper) and machine (represented by man's university's computers) worked hand-in-keyboard. To wit, we began our experimentation with small sets of functions and terminals, which were revised and added upon through our examination of evolved players and their performance. Figure 8 describes three major steps in our hand-in-keyboard development of the evolutionary chess setup.

We believe that genetic programming represents a viable means to automatic programming, and perhaps more generally to machine intelligence, in no small part due to attribute 17: more than many other adaptive-search techniques (e.g., genetic algorithms, artificial neural networks, and ant algorithms), genetic programming's representational affluence and openness lend it to the ready imbuing of the designer's own

TABLE IV

BEST GP-ROBOCODE TAKES SECOND PLACE AT HAIKUBOT LEAGUE ON NEW YEAR'S DAY, 2005

(`robocode.yajags.com/20050101/haiku-1v1.html`). THE TABLE'S COLUMNS REFLECT VARIOUS ASPECTS OF ROBOTIC BEHAVIOR. *Survival*: EACH LIVE ROBOT IS GIVEN 50 POINTS WHENEVER ANOTHER ROBOT DIES (THIS IS TRUE WHERE ROBOTIC TEAMS ARE CONCERNED—IN THE ONE-ON-ONE SCENARIO WE STUDIED THE WINNER GETS 50 POINTS). *Last Survivor Bonus*: THE LAST ROBOT ALIVE IS GIVEN AN ADDITIONAL REWARD OF 10 POINTS FOR EACH DEAD ROBOT (FOR THE ONE-ON-ONE CASE THE WINNER GETS 10 POINTS). *Bullet Damage*: TOTAL DAMAGE INFLICTED TO OTHER ROBOTS BY HITTING THEM WITH BULLETS. *Bonus (bullet damage)*: 20% OF THE DAMAGE INFLICTED BY BULLETS, FOR EACH ROBOT KILLED BY BULLETS. *Ram Damage*: THE AMOUNT OF DAMAGE INFLICTED TO OTHER ROBOTS BY RAMMING THEM, MULTIPLIED BY TWO. *Bonus (ram damage)*: 30% OF THE DAMAGE INFLICTED BY RAMMING, FOR EACH ROBOT KILLED BY RAMMING. *Total Score*: SUM OF ALL OF THE ABOVE . THE FINAL RANK IS DETERMINED BY THE *Rating* MEASURE, WHICH REFLECTS THE PERFORMANCE OF THE ROBOT IN COMBAT WITH RANDOMLY CHOSEN ADVERSARIES.

Rank	Rating	Robot	Total Score	Survival	Last surv.	Bullet dmg.	Bonus	Ram dmg.	Bonus	1sts	2nds	3rds
1	543.42	ms.AresHaiku 0.4	33386	14450	2890	13359	2543	137	0	292	30	0
2	219.73	geep.haiku.GPBotC 1.0	34859	9950	1990	17931	2892	1783	304	206	115	0
3	192.13	cx.haiku.Xaxa 1.1	34981	12200	2440	17132	2907	150	143	246	76	0
4	177.08	kawigi.haiku.HaikuTrogdor 1.1	34938	11300	2260	18232	2753	360	26	228	93	0
5	172.38	ms.ChaosHaiku 0.1	33919	9700	1940	18201	2770	1109	190	199	125	0
6	164.79	mz.HaikuGod 1.01	35120	11150	2230	16968	2340	2122	300	268	97	0
7	117.36	pez.femto.WallsPoetHaiku 0.1	33634	9800	1960	17610	2731	1430	92	198	124	0
8	70.48	pez.femto.HaikuPoet 0.2	35972	11650	2330	18424	2994	514	51	240	86	0
9	70.34	cx.haiku.Escape 1.0	31837	10600	2120	16110	2445	510	41	214	108	0
10	45.12	kawigi.femto.FemtoTrogdor 1.0	25986	6900	1380	13907	1805	1769	215	140	182	0
11	41.09	kawigi.haiku.HaikuLinearAimer 1.0	34415	10450	2090	18744	2985	139	0	213	108	0
12	40.40	kawigi.haiku.HaikuCircleBot 1.0	30217	10000	2000	15772	2382	56	0	206	119	0
13	28.63	cx.haiku.MeleeXaxa 1.0	38853	9850	1970	21680	2411	1864	1069	201	123	0
14	17.25	soup.haiku.RammerHK 1.0	46774	7950	1590	22746	2557	10289	1630	168	157	0
15	8.28	kawigi.haiku.HaikuSillyBot 1.2	20264	6500	1300	11133	1282	44	0	138	186	0
16	0.15	cr.OneOnOneHaiku 1.1	37955	7650	1530	20105	2856	5341	464	154	166	0
17	-52.97	cx.haiku.Smoku 1.1	28672	9250	1850	15321	2161	57	22	187	134	0
18	-54.54	ahf.HaikuAndrew .1	19153	7150	1430	9186	1148	178	50	146	177	0
19	-56.70	dummy.haiku.Disoriented 1.0	23957	6500	1300	14560	1419	133	36	135	188	0
20	-63.77	soup.haiku.CutoffHK 1.0	29878	4550	910	17264	1559	5040	546	99	224	0
21	-115.04	kawigi.haiku.HaikuChicken 1.0	24091	6400	1280	14560	1758	86	0	133	190	0
22	-120.51	soup.haiku.MirrorHK 1.0	27971	5700	1140	18524	2071	501	26	125	200	0
23	-154.28	davidalves.net.PhoenixHaiku 1.0	19902	5450	1090	11890	1394	72	0	117	206	0
24	-174.56	klo.haikuBounC 1.0	25039	6050	1210	15618	1591	503	58	123	199	0
25	-193.80	soup.haiku.RandomHK 1.0	19045	2800	560	14477	1059	141	0	59	264	0
26	-208.10	tango.haiku.HaikuTango 1.0	16682	4600	920	9982	875	292	4	96	227	0
27	-303.69	soup.haiku.DodgeHK 1.0	13440	2250	450	9829	482	424	0	46	274	0
28	-410.69	soup.haiku.WallDroidHK 1.0	7959	1950	390	4905	191	490	21	44	277	0

intelligence within the language of design. While artificial-intelligence (AI) purists may wrinkle their noses at this, taking the AI-should-emerge-from-scratch stance, we argue that a more practical path to AI involves man-machine cooperation. Genetic programming, as evidenced herein, is a forerunning candidate for the 'machine' part.

This brings up a related issue, derived from Koza *et al.*'s affirmation that, "[g]enetic programming now routinely delivers high-return human-competitive machine intelligence" [33]:

- Human-competitive: Getting machines to produce human-like results, e.g., a patentable invention, a result publishable in the scientific literature, or a game strategy that can hold its own against humans.
- High-return: Defined by Koza *et al.* as a high "artificial-to-intelligence ratio" (**A/I**), namely, the ratio of that which is delivered by the automated operation of the artificial method to the amount of intelligence that is supplied by the human applying the method to a particular system.
- Routine: The successful handling of new problems once the method has been "jump-started."
- Machine intelligence: To quote Arthur Samuel, getting "machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence."

Our discussion regarding attribute 17 stands in contrast to the second property above—high-return—which we believe to be of little import in the domain of human-competitive machines, and indeed, in the attainment of machine intelligence in general. Rather than aiming to maximize **A/I** we believe the "correct" equation is:

$$A - I \geq M_\epsilon,$$

where M_ϵ stands for "meaningful epsilon." When wishing to attain machine competence in some real-life, hard-to-learn domain, then—by all means—imbue the machine with as much **I**(ntelligence) as possible! After all, if imbuing the **I** reduces the problem's complexity to triviality, then it was probably not hard to begin with. Conversely, if the problem is truly hard, then have man and machine work in concert to push the frontiers of **A** as far as possible. Thus, it is not $\max(A/I)$ that is of interest but the added value of the machine's output: granting the designer "permission" to imbue the machine with as much **I** as he can, will it then produce a $\Delta A = A - I$, namely, added intelligence, that is sufficiently meaningful? Even if this meaningful epsilon (M_ϵ) is small in (some) absolute terms, its relative value can be huge (e.g., a chip that can pack 1-2% more transistors, or a game player that is slightly better—and thus world champion).

One problem with the $\max(A/I)$ view is its ignoring the important distinction between two phases of intelligence (or knowledge) development: 1) from scratch to a mediocre level,

1. Initial runs.
Terminals: No distinction between "good" and "bad" (i.e., no negative terminals). Example: IsMyKingInCheck and is IsOppKingInCheck (later the former will become NotMyKingInCheck). Terminals included:

- Is[My/Opp]PieceAttacked()
- MaterialCount()
- NumMoves[My/Opp]King()
- Is[My/Opp]PieceAttacked()
- Is[My/Opp]PieceProtected()
- Is[My/Opp]QueenAttacked()
- IsMate()
- ERCs in range [-1000,+1000]

Functions:

- Arithmetic: *, +, -
- Logic: And2, And3, And4, Or2, Or3, Or4, Not
- Others: If, $<$, =, $>$

2. Later runs. We consulted a chess Master.
Terminals:

- Modified to distinguish between positive and negative, e.g., NotMyKingInCheck and MyKingDistEdges
- Added IsMaterialIncrease()
- Added Not[My/Opp]KingMovesDecrease()
- Added Num[My/Opp]PiecesNotAttacked()
- Added IsMyKingProtectingPiece()
- Added IsMyPieceAttackedUnprotected()
- Added IsOppKingBehingPiece()
- Added IsStalemate()

Functions:

- Removed arithmetic functions except for Negate (see [13] for reasons)
- Removed $>$ to simplify computation
- Used IfAdvantageThen[Left Subtree]Else[Right Subtree] to create separate calculations

3. Final runs. We further consulted a Master, adding complex and simple terminals.
Terminals:

- Added MateInOne()
- Added IsOppKingStuck()
- Added OppPieceCanBeCaptured()
- IsMaterialIncrease() changed to 100*IsMaterialIncrease()
- Added ValueOf[My/Opp][Attacking/Protecting]Pieces()
- Added Is[My/Opp][Not]Fork()
- Added [My/Opp]King[Dist/Prox]Rook()
- Added [My/Opp]Pieces[Not]SameLine()
- Num[My/Opp]Pieces[Not]Attacked()
- ERCs: Now only six values allowed: $\pm$ {0.25, 0.5, 1}*1000

Functions:

- Removed Negate
- Changed program topology to three trees

Fig. 8. Three major steps in developing the evolutionary chess setup.

and 2) from mediocre to expert level. Traditional AI is often better at handling the first phase. Genetic programming allows the AIer to focus his attention on the second phase, namely, the attainment of true expertise. When aiming to develop a winning strategy, be it in games or any other domain, the genetic-programming practitioner will set his sights at the mediocre-to-expert phase of development, with the scratch-to-mediocre handled automatically during the initial generations of the evolutionary process. Although the designer is "imposing" his own views on the machine, this affords the "pushing" of the **A** frontier further out. Note that, at the limit, if **I** tends to zero, you may get an extremely high A/I ratio, but with very little truly meaningful **A**. Focusing on $A-I$ underscores the need, or wish, for a high level of intelligence, where even a small M_ϵ becomes important.

Cognitive psychology recognizes the importance of *schemata*, a fundamental notion first defined by Bartlett in his influential book from 1932 [34]. Schemata are mental patterns or models that give rise to certain cognitive abilities—complex unconscious knowledge structures such as symmetry in vision, plans in a story, and rules. Much of our knowledge is encoded as schemata, to be neurally activated when their components are triggered in a certain way (only a certain configuration of face parts will activate the "face" schema). Genetic programming is able to go beyond low-level "bits-and-pieces" knowledge and handle what may well be schemata analogs. In our treatment of games we were able to encode meaningful patterns (schemata) as terminals, then to be combined through the use of functions. This adds a whole new dimension to the representations one can design.

As an example, a chess master's knowledge seems to comprise some 100,000 schemata [35], and his advantage over the machine is the ability to combine these schemata intelligently in response to a given situation. It is not impossible to imagine programming 100,000 chess features, in striving to grant your machine as much **I** as possible; but finding and applying the correct combinations is exponential in nature. Here genetic programming steps in, constantly trying new combinations and combinations of combinations, beyond that which is possible to accomplish by (traditional) AI (artificial neural networks, for example, also traverse the search space but they lack the ability to integrate deep knowledge in a natural manner).

Genetic programming is able to combine search with pattern recognition, as is true of humans, with the terminals acting as pattern recognizers (e.g., safety of king, mate in one). Chess players, for example, seem to make extensive use of patterns, or templates [36]. Patterns are a powerful addition to the toolbox of the machine's **A**, enabling it to make use of an important element of **I**.

Studying our three special-purpose strategizing machines in an attempt to generalize upon them, we first note the close similarity of the function sets, which contain mostly standard arithmetic and logic functions. A standard function set could thus be envisaged, including a minimal set of necessary functions. A straightforward proposal would be to define a Turing-machine equivalent set of operators. While this approach makes sense from a theoretical point of view, it does not quite promote the *practical* design of a strategizing machine. Turing-machine equivalence can readily be had by using a very small subset of operators (e.g., a W-machine [37], as used by [38] to build self-replicating structures, comprises an instruction set of size six; however, forming useful programs with such a limited repertoire is very arduous). A balance need be struck between a sufficiently large function set so as to be useful, though not too large so as to be unmanageable. Further research is required as to the exact makeup of this set.

The terminal sets of the three strategizing machines are quite different, embodying, as they were, most of the human **I**. The challenge here is quite different than for the function set since a standard terminal set is, *ipso facto*, not attainable. Rather, the goal here would be to create a proper interface, so that the designer (user) could easily incorporate his knowledge within the genetic programming framework.

To summarize the main points presented in this paper:

- Genetic programming has proven to be an excellent tool for automatically generating complex game strategies.
- A crucial advantage of genetic programming lies in its ability to incorporate human intelligence readily.
- As such, genetic programming is an excellent choice when complex strategies are needed, and human intelligence is there for the imbuing.

In short, $GP + I \Rightarrow HC$, i.e., Genetic Programming + (Human) Intelligence yields Human-Competitiveness.

An early, well-known AI program—Newell's and Simon's General Problem Solver (GPS) [39]—ultimately proved to be far from general and quite limited in scope. As opposed to their GPS we do not advocate complete generality and—more importantly—neither do we promote total machine autonomy. We believe our approach represents a more practical means of attaining machine expertise—at least at the current state of AI—and suggest the replacing of the original GPS with a more humble one: Genetic Programming Solver.

ACKNOWLEDGEMENTS

We are grateful to Assaf Zaritsky, John Koza, and the anonymous reviewers for their many helpful comments.

REFERENCES

[1] S. L. Epstein, "Game playing: The next moves," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence*. 1999, pp. 987–993, AAAI Press, Menlo Park, California USA.

[2] J. E. Laird and M. van Lent, "Human-level AI's killer application: Interactive computer games," in *AAAI-00: Proceedings of the 17th National Conference on Artificial Intelligence*. 2000, pp. 1171–1178, The MIT Press, Cambridge, Massachusetts.

[3] M. Sipper, *Machine Nature: The Coming Age of Bio-Inspired Computing*, McGraw-Hill, New York, 2002.

[4] A. Tettamanzi and M. Tomassini, *Soft Computing: Integrating Evolutionary, Neural, and Fuzzy Systems*, Springer, Berlin, 2001.

[5] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, Piscataway, NJ, second edition, 1999.

[6] M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*, The MIT Press, Cambridge, MA, 1999.

[7] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Heidelberg, third edition, 1996.

[8] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, Cambridge, Massachusetts, 1992.

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press, Ann Arbor, Michigan, 1975, (Second edition, Cambridge, MA: MIT Press, 1992).

[10] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence Through Simulated Evolution,*, John Wiley, New York NY, 1966.

[11] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Proceedings of the 1st International Conference on Genetic Algorithms*, J. J. Grefenstette, Ed. 1985, pp. 183–187, Lawrence Erlbaum Associates, Inc. Mahwah, NJ, USA.

[12] Y. Azaria and M. Sipper, "GP-Gammon: Using genetic programming to evolve backgammon players," in *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds. 2005, vol. 3447 of *Lecture Notes in Computer Science*, pp. 132–142, Springer-Verlag, Heidelberg.

[13] A. Hauptman and M. Sipper, "GP-EndChess: Using genetic programming to evolve chess endgame players," in *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds. 2005, vol. 3447 of *Lecture Notes in Computer Science*, pp. 120–131, Springer-Verlag, Heidelberg.

[14] Y. Shichel, E. Ziserman, and M. Sipper, "GP-Robocode: Using genetic programming to evolve robocode players," in *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds. 2005, vol. 3447 of *Lecture Notes in Computer Science*, pp. 143–154, Springer-Verlag, Heidelberg.

[15] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, July 1959.

[16] G. Kendall and G. Whitwell, "An evolutionary approach for the tuning of a chess evaluation function using population dynamics," in *Proceedings of the 2001 Congress on Evolutionary Computation (CEC2001)*. 2001, pp. 995–1002, IEEE Press.

[17] N. Chomsky, *Language and Thought*, Moyer Bell, Wakefield, RI, 1993.

[18] H. Abelson and G. J. Sussman with J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Massachusetts, second edition, 1996.

[19] G. Tesauro, *Software Source Code Benchmark player "pubeval.c"*, 1993, http://www.bkgm.com/rgb/rgb.cgi?view+610.

[20] J. Paredis, "Coevolutionary computation," *Artificial Life*, vol. 2, no. 4, pp. 355–375, 1995.

[21] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evolutionary Computation*, vol. 5, no. 1, pp. 1–29, 1997.

[22] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D*, vol. 42, pp. 228–234, 1990.

[23] M. A. Potter and K. A. De Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *Evolutionary Computation*, vol. 8, no. 1, pp. 1–29, Spring 2000.

[24] C.-A. Peña-Reyes and M. Sipper, "Fuzzy CoCo: A cooperative-coevolutionary approach to fuzzy modeling," *IEEE Transactions on Fuzzy Systems*, vol. 9, no. 5, pp. 727–737, October 2001.

[25] P. J. Angeline and J. B. Pollack, "Competitive environments evolve better solutions for complex tasks," in *Proceedings of the 5th International Conference on Genetic Algorithms (GA93)*, S. Forrest, Ed., 1993, pp. 264–270.

[26] L. A. Panait and S. Luke, "A comparison of two competitive fitness functions," in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds. 2002, pp. 503–511, Morgan Kaufmann Publishers, San Francisco, CA.

[27] P. Darwen, "Why co-evolution beats temporal-difference learning at backgammon for a linear architecture, but not a non-linear architecture," in *Proceedings of the 2001 Congress on Evolutionary Computation (CEC-01)*, Seoul Korea, 2001, pp. 1003–1010.

[28] D. Qi and R. Sun, "Integrating reinforcement learning, bidding and genetic algorithms," in *Proceedings of the International Conference on Intelligent Agent Technology (IAT-2003)*. 2003, pp. 53–59, IEEE Computer Society Press, Los Alamitos, CA.

[29] S. Sanner, J. R. Anderson, C. Lebiere, and M. Lovett, "Achieving efficient and cognitively plausible learning in backgammon," in *Proceedings of the 17th International Conference on Machine Learning (ICML-2000)*, P. Langley, Ed., Stanford, CA, 2000, pp. 823–830, Morgan Kaufmann.

[30] J. B. Pollack, A. D. Blair, and M. Land, "Coevolution of a backgammon player," in *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, C. G. Langton and K. Shimohara, Eds., Cambridge, MA, 1997, pp. 92–98, MIT Press.

[31] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, California, 1999.

[32] E. M. A. Ronald, M. Sipper, and M. S. Capcarrère, "Design, observation, surprise! A test of emergence," *Artificial Life*, vol. 5, no. 3, pp. 225–239, Summer 1999.

[33] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, Norwell, MA, 2003.

[34] F. C. Bartlett, *Remembering: An Experimental and Social Study*, Cambridge University Press, Cambridge, UK, 1932.

[35] H. A. Simon and K. Gilmartin, "A simulation of memory for chess positions," *Cognitive Psychology*, vol. 5, no. 1, pp. 29–46, July 1973.

[36] F. Gobet and H. A. Simon, "Templates in chess memory: A mechanism for recalling several boards," *Cognitive Psychology*, vol. 31, pp. 1–40, 1996.

[37] H. Wang, "A variant to Turing's theory of computing machines," *Journal of the ACM*, vol. IV, pp. 63–92, 1957.
[38] J.-Y. Perrier, M. Sipper, and J. Zahnd, "Toward a viable, self-reproducing universal computer," *Physica D*, vol. 97, pp. 335–352, 1996.
[39] A. Newell and H. A. Simon, "GPS, a program that simulates human thought," in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, Eds., pp. 279–296. McGraw-Hill, New York, 1963.