Genetic and Evolutionary Algorithms and Programming

Michael Orlov, Moshe Sipper and Ami Hauptman
Department of Computer Science, Ben-Gurion University
PO Box 653, Beer-Sheva 84105, Israel
{orlovm,sipper,amiha}@cs.bgu.ac.il

July 2, 2007

Article Outline

Glossary

Evolutionary algorithms / Evolutionary computation

A family of algorithms inspired by the workings of evolution by natural selection whose basic structure is:

1. produce an initial **population** of individuals, these latter being candidate solutions to the problem at hand
2. evaluate the **fitness** of each individual in accordance with the problem whose solution is sought
3. *while* termination condition not met *do*
   (a) **select** fitter individuals for reproduction
   (b) **recombine (crossover)** individuals
   (c) **mutate** individuals
   (d) **evaluate** fitness of modified individuals
4. *end while*

Genome / Chromosome

An individual's makeup in the population of an evolutionary algorithm is known as a genome, or chromosome. It can take on many forms, including bitstrings, real-valued vectors, character-based encodings, and computer programs. The representation issue—namely, defining an individual's genome (well)—is critical to the success of an evolutionary algorithm.

Fitness

A measure of the quality of a candidate solution in the population. Also known as *fitness function.* Defining this function well is critical to the success of an evolutionary algorithm.

Selection

The operator by which an evolutionary algorithm selects (usually probabilistically) higher-fitness individuals to contribute "genetic" material to the next generation.

Crossover

One of the two main "genetic" operators applied by an evolutionary algorithm, wherein two (or more) candidate solutions ("parents") are combined in some pre-defined manner to form "offspring."

Mutation

One of the two main "genetic" operators applied by an evolutionary algorithm, wherein one candidate solution is randomly altered.

I Definition of the Subject and its Importance

Evolutionary algorithms are a family of search algorithms inspired by the process of (Darwinian) evolution in Nature. Common to all the different family members is the notion of solving problems by evolving an initially random population of candidate solutions, through the application of operators inspired by natural genetics and natural selection, such that in time "fitter" (i.e., better) solutions emerge. The field, whose origins can be traced back to the 1950s and 1960s, has come into its own over the past two decades, proving successful in solving multitudinous problems from highly diverse domains including (to mention but a few): optimization, automatic programming, electronic-circuit design, telecommunications, networks, finance, economics, image analysis, signal processing, music, and art.

II Introduction

The first approach to artificial intelligence, the field which encompasses evolutionary computation, is arguably due to Turing [31]. Turing asked the famous question: "Can machines think?" Evolutionary computation, as a subfield of AI, may be the most straightforward answer to such a question. In principle, it might be possible to evolve

an algorithm possessing the functionality of the human brain (this has already happened at least once: in Nature).

In a sense, nature is greatly inventive. One often wonders how so many magnificent solutions to the problem of existence came to be. From the intricate mechanisms of cellular biology, to the sandy camouflage of flatfish; from the social behavior of ants to the diving speed of the peregrine falcon—nature created versatile solutions, at varying levels, to the problem of survival. Many ingenious solutions were "invented" (and still are), without any obvious intelligence directly creating them. This is perhaps the main motivation behind evolutionary algorithms: creating the settings for a dynamic environment, in which solutions can be created and improved in the course of time, advancing in new directions, with minimal direct intervention. The gain to problem solving is obvious.

III Evolutionary Algorithms

In the 1950s and the 1960s several researchers independently studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. Central to all the different methodologies is the notion of solving problems by evolving an initially random population of candidate solutions, through the application of operators inspired by natural genetics and natural selection, such that in time "fitter" (i.e., better) solutions emerge [9, 16, 19, 29]. This thriving field goes by the name of *evolutionary algorithms* or *evolutionary computation*, and today it encompasses two main branches—genetic algorithms [9] and genetic programming [19]—in addition to less prominent (though important) offshoots, such as evolutionary programming [10] and evolution strategies [26].

A *genetic algorithm* (GA) is an iterative procedure that consists of a constant-size *population* of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces which are too large to be exhaustively searched. The symbol alphabet used is often binary, but may also be character-based, real-valued, or any other representation most suitable to the problem at hand.

The standard genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness. Many selection procedures are available, one of the simplest being *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Thus, high-fitness ("good") individuals stand a better chance of "reproducing," while low-fitness ones are more likely to disappear.

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space; these are generated by genetically-inspired operators, of which the most well known are *crossover* and *mutation*. Crossover is performed with probability p_{cross} (the "crossover probability" or "crossover rate") between two selected individuals, called *parents*, by exchanging parts of their genomes (i.e., en-

codings) to form two new individuals, called *offspring*. In its simplest form, substrings are exchanged after a randomly-selected crossover point. This operator tends to enable the evolutionary process to move toward "promising" regions of the search space. The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some (small) probability p_{mut}. Genetic algorithms are stochastic iterative processes that are not guaranteed to converge. The termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level. Figure 1 presents the standard genetic algorithm in pseudo-code format.

```
begin GA
    g:=0    { generation counter }
    Initialize population P(g)
    Evaluate population P(g)    { i.e., compute fitness values }
    while not done do
        g:=g+1
        Select P(g) from P(g − 1)
        Crossover P(g)
        Mutate P(g)
        Evaluate P(g)
    end while
end GA
```

Figure 1: Pseudo-code of the standard genetic algorithm.

Let us consider the following simple example, demonstrating the GA's workings. The population consists of 4 individuals, which are binary-encoded strings (genomes) of length 10. The fitness value equals the number of ones in the bit string, with $p_{cross} = 0.7$ and $p_{mut} = 0.05$. More typical values of the population size and the genome length are in the range 50-1000. Note that fitness computation in this case is extremely simple, since no complex decoding or evaluation is necessary. The initial (randomly generated) population might look as shown in Table 1.

Label	Genome	Fitness
p_1	0000011011	4
p_2	1110111101	8
p_3	0010000010	2
p_4	0011010000	3

Table 1: The initial population.

Using fitness-proportionate selection we must choose 4 individuals (two sets of parents), with probabilities proportional to their relative fitness values. In our example, suppose that the two parent pairs are $\{p_2, p_4\}$ and $\{p_1, p_2\}$ (note that individual p_3 did not get selected as our procedure is probabilistic). Once a pair of parents is selected, crossover is effected between them with probability p_{cross}, resulting in two offspring. If

no crossover is effected (with probability $1 - p_{cross}$), then the offspring are exact copies of each parent. Suppose, in our example, that crossover takes place between parents p_2 and p_4 at the (randomly chosen) third bit position:

111|0111101
001|1010000

This results in offspring $p'_1 = 1111010000$ and $p'_2 = 0010111101$. Suppose no crossover is effected between parents p_1 and p_2, forming offspring that are exact copies of p_1 and p_2. Our interim population (after crossover) is thus as depicted in Table 2:

Label	Genome	Fitness
p'_1	1111010000	5
p'_2	0010111101	6
p'_3	0000011011	4
p'_4	1110111101	8

Table 2: The interim population.

Next, each of these four individuals is subject to mutation with probability p_{mut} per bit. For example, suppose offspring p'_2 is mutated at the sixth position and offspring p'_4 is mutated at the ninth bit position. Table 3 describes the resulting population.

Label	Genome	Fitness
p''_1	1111010000	5
p''_2	0010101101	5
p''_3	0000011011	4
p''_4	1110111111	9

Table 3: The resulting population.

The resulting population is that of the next generation (i.e., p''_i equals p_i of the next generation). As can be seen, the transition from one generation to the next is through application of selection, crossover, and mutation. Moreover, note that the best individual's fitness has gone up from 8 to 9, and that the average fitness (computed over all individuals in the population) has gone up from 4.25 to 5.75. Iterating this procedure, the GA will eventually find a perfect string, i.e., with maximal fitness value of 10.

Another prominent branch of the Evolutionary Computation tree is that of *genetic programming*, introduced by Cramer [7], and transformed into a field in its own right in large part due to the efforts of Koza [19]. Basically, genetic programming (GP) is a GA (genetic algorithm) with individuals in the population being programs instead of bit strings.

In GP we evolve a population of individual LISP expressions[1], each comprising

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

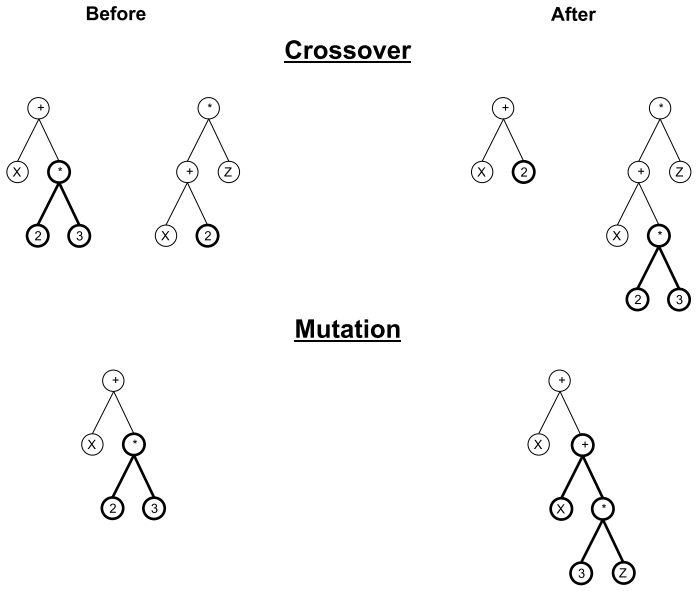

Figure 2: Genetic operators in genetic programming. LISP programs are depicted as trees. Crossover (top): Two sub-trees (marked in bold) are selected from the parents and swapped. Mutation (bottom): A sub-tree (marked in bold) is selected from the parent individual and removed. A new sub-tree is grown instead.

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.

The main mechanism behind GP is precisely that of a GA, namely, the repeated cycling through four operations applied to the entire population: evaluate-select-crossover-mutate. However, the evaluation of a single individual in GP is usually more complex than with a GA since it involves running a program. Moreover, crossover and mutation need to be made to work on trees (rather than simple bit strings), as shown in Figure 2.

IV A Touch of Theory

Evolutionary computation is mostly an experimental field. However, over the years there have been some notable theoretical treatments of the field, gaining valuable insights into the properties of evolving populations.

Holland [17] introduced the notion of *schemata*, which are abstract properties of binary-encoded individuals, and analyzed the growth of different schemas when fitness-proportionate selection, point mutation and one-point crossover are employed. Holland's approach has since been enhanced and more rigorous analysis performed; however, there were not many practical consequences on the existing evolutionary techniques, since most of the successful methods are usually much more complex in many aspects. Moreover, the schematic analysis suffers from an important approximation of infinite population size, while in reality schemata can vanish.

Note that the "No Free Lunch" theorem states that "...for any [optimization] al-

gorithm, any elevated performance over one class of problems is exactly paid for in performance over another class." [32]

V Extensions of the Basic Methodology

We have reviewed the basic evolutionary computation methods. More advanced techniques are used to tackle complex problems, where an approach of a single population with homogeneous individuals does not suffice. One such advanced approach is coevolution [24].

Coevolution refers to the simultaneous evolution of two or more species with coupled fitness. Such coupled evolution favors the discovery of complex solutions whenever complex solutions are required. Simplistically speaking, one can say that coevolving species can either compete (e.g., to obtain exclusivity on a limited resource) or cooperate (e.g., to gain access to some hard-to-attain resource). In a competitive coevolutionary algorithm the fitness of an individual is based on direct competition with individuals of other species, which in turn evolve separately in their own populations. Increased fitness of one of the species implies a diminution in the fitness of the other species. This evolutionary pressure tends to produce new strategies in the populations involved so as to maintain their chances of survival. This "arms race" ideally increases the capabilities of each species until they reach an optimum.

Cooperative (also called symbiotic) coevolutionary algorithms involve a number of independently evolving species which together form complex structures, well suited to solve a problem. The fitness of an individual depends on its ability to collaborate with individuals from other species. In this way, the evolutionary pressure stemming from the difficulty of the problem favors the development of cooperative strategies and individuals.

Single-population evolutionary algorithms often perform poorly—manifesting stagnation, convergence to local optima, and computational costliness—when confronted with problems presenting one or more of the following features: 1) the sought-after solution is complex, 2) the problem or its solution is clearly decomposable, 3) the genome encodes different types of values, 4) strong interdependencies among the components of the solution, and 5) components-ordering drastically affects fitness [24]. Cooperative coevolution addresses effectively these issues, consequently widening the range of applications of evolutionary computation.

Consider, for instance, the evolution of neural networks [33]. A neural network consists of simple units called neurons, each having several inputs and a single output. The inputs are assigned weights, and a weighted sum of the inputs exceeding a certain threshold causes the neuron to fire an output signal. Neurons are usually connected using a layered topology.

When we approach the task of evolving a neural network possessing some desired property naively, we will probably think of some linearized representation of a neural network, encoding both the neuron locations in the network, and their weights. However, evolving such a network with a simple evolutionary algorithm might prove quite a frustrating task, since much information is encoded in each individual, and it is not homogeneous, which presents us with the difficult target of evolving the individuals as single entities.

On the other hand, this task can be dealt with more sagely via evolving two independently encoded populations of neurons and network topologies. Stanley and Miikkulainen [30] evaluate the fitness of an individual in one of the populations using the

individuals of the other. In addition to the simplification of individuals in each population, the fitness is now dynamic, and an improvement in the evolution of topologies triggers a corresponding improvement in the population of neurons, and vice versa.

VI Lethal Applications

In this section we review a number of applications that—though possibly not killer (death being in the eye of the beholder. . .)—are most certainly lethal. These come from a sub-domain of evolutionary algorithms, which has been gaining momentum over the past few years: human-competitive machine intelligence. Koza et al. [20] recently affirmed that the field of evolutionary algorithms "now routinely delivers high-return human-competitive machine intelligence", meaning, according to [20]:

- 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."

Indeed, as of 2004 the major annual event in the field of evolutionary algorithms—GECCO (Genetic and Evolutionary Computation Conference; see `www.sigevo.org`)—boasts a prestigious competition that awards prizes to human-competitive results. As noted at `www.human-competitive.org`: "Techniques of genetic and evolutionary computation are being increasingly applied to difficult real-world problems—often yielding results that are not merely interesting, but competitive with the work of creative and inventive humans."

We now describe some winners from the HUMIES competition at `www.human-competitive.org`. Lohn et al. [22] won a Gold Medal in the 2004 competition for an evolved X-band antenna design and flight prototype to be deployed on NASA's Space Technology 5 (ST5) spacecraft:

> The ST5 antenna was evolved to meet a challenging set of mission requirements, most notably the combination of wide beamwidth for a circularly-polarized wave and wide bandwidth. Two evolutionary algorithms were used: one used a genetic algorithm style representation that did not allow branching in the antenna arms; the second used a genetic programming style tree-structured representation that allowed branching in the antenna arms. The highest performance antennas from both algorithms were fabricated and tested, and both yielded very similar performance. Both antennas were comparable in performance to a hand-designed antenna produced by the antenna contractor for the mission, and so we consider them examples of human-competitive performance by evolutionary algorithms. [22]

Preble et al. [25] won a Gold Medal in the 2005 competition for designing photonic crystal structures with large band gaps. Their result is "an improvement of 12.5% over the best human design using the same index contrast platform."

Recently, Kilinç et al. [18] was awarded the Gold Medal in the 2006 competition for designing oscillators using evolutionary algorithms, where the oscillators possess characteristics surpassing the existing human-designed analogs.

VII Evolutionary Games

Evolutionary games is the application of evolutionary algorithms to the evolution of game-playing strategies for various games, including chess, backgammon, and Robocode.

Motivation and Background

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 introduced 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 (or 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. [8]

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 [21] 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. [21]

Evolving game-playing strategies

Recently, evolutionary algorithms have proven a powerful tool that can automatically "design" successful game-playing strategies for complex games [2, 3, 13–15, 27, 28].

1. **Chess** (endgames). Evolve a player able to play endgames [13–15, 28]. 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.

2. **Backgammon.** Evolve a full-fledged player for the non-doubling-cube version of the game [2, 3, 28].

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 [27, 28].

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 [19]. 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 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. The approach we used with backgammon and chess is to derive a very shallow, single-level tree, and evolve "smart" evaluation functions. Our artificial player is thus had by combining an evolved board evaluator with a 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 six items necessary in order to employ genetic programming: program architecture, set of terminals, set of functions, fitness measure, control parameters, and manner of designating result and terminating run.

Example: Chess

As our purpose is to create a schema-based program that analyzes single nodes thoroughly, in a way reminiscent of human thinking, we did not perform deep lookahead.

We evolved individuals represented as LISP programs. Each such program receives a chess endgame position as input, and, according to its sensors (terminals) and functions, returns an evaluation of the board, in the form of a real number.

Our chess endgame players consist of an evolved LISP program, together with a piece of software that generates all possible (legal) next-moves and feeds them to the program. The next-move with the highest score is selected (ties are broken stochastically). The player also identifies when the game is over (either by a draw or a win).

Program architecture. As most chess players would agree, playing a winning position (e.g., with material advantage) is very different than playing a losing position, or an even one. For this reason, each individual contains not one but three separate 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, since achieving a stalemate and avoiding exchanges requires less complicated reasoning. Most terminals and functions were used for all trees.

The structure of three trees per individual was preserved mainly for simplicity reasons. It is actually possible to co-evolve three separate populations of trees, without binding them to form a single individual before the end of the experiment. This would require a different experimental setting, and is one of our future-work ideas.

Terminals and functions. While evaluating a position, an expert chess player considers various aspects of the board. Some are simple, while others require a deep understanding of the game. Chase and Simon found that experts recalled meaningful chess formations better than novices [6]. This led them to hypothesize that chess skill depends on a large knowledge base, indexed through thousands of familiar chess patterns.

We assumed that complex aspects of the game board are comprised of simpler units, which require less game knowledge, and are to be combined in some way. Our chess programs use terminals, which represent those relatively simple aspects, and functions, which incorporate no game knowledge, but supply methods of combining those aspects. As we used strongly typed GP [23], all functions and terminals were assigned one or more of two data types: *Float* and *Boolean*. We also included a third data type, named *Query*, which could be used as any of the former two. We also used ephemeral random constants (ERCs).

The terminal set. We developed most of our terminals by consulting several high-ranking chess players. [2] The terminal set examined various aspects of the chessboard, and may be divided into 3 groups:

Float values, created using the ERC mechanism. ERCs were 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 guaranteed that when a value was returned after some group of features has been identified, it was distinct enough to engender the outcome.

Simple terminals, which analyzed 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 belonged to the next group (see below). For example, a player should capture 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

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

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 were 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 were embodied as terminals within the second group. The ability to easily embody such reasoning within the GP setup, as functions and terminals, is a major asset of GP.

Other terminals were also derived in a similar manner. See Table 4 for a complete list of simple terminals. 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).

Complex terminals: these are terminals that check the same aspects of the board a human player would. Some prominent examples include: the terminal 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. See Table 5 for a full list of complex terminals.

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 [1]: 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.

The function set. The function set used included the If function, and simple Boolean functions. Although our tree returns a real number, we omitted arithmetic functions, for several reasons. First, a large part of contemporary research in the field of machine learning and game theory (in particular for perfect-information games) revolves around inducing logical rules for learning games (for example, see [4, 5, 11]). Second, according to the players we consulted, while evaluating positions involves considering various aspects of the board, some more important than others, performing logical operations on these aspects seems natural, while mathematical operations does not. Third, we observed that numeric functions sometimes returned extremely large values, which interfered with subtle calculations. Therefore the scheme we used was a (carefully ordered) series of Boolean queries, each returning a fixed value (either an ERC or a numeric terminal, see below). See Table 6 for the complete list of functions.

Fitness evaluation. As we used a competitive evaluation scheme, the fitness of an individual was determined by its success against its peers. We used the random-2-ways method, in which each individual plays against a fixed number of randomly selected peers. Each of these encounters entailed a fixed number of games, each starting from a randomly generated position in which no piece was attacked.

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

Table 4: Simple terminals for evolving chess endgame players. *Opp:* opponent, *My:* player.

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

Table 5: Complex terminals for evolving chess endgame players. *Opp:* opponent, *My:* player. Some of these terminals perform lookahead, while others compare with the original board.

Function	Description
F=If3(B_1, F_1, F_2)	If B_1 is non-zero, return F_1, else return F_2
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise
B=Or3(B_1, B_2, B_3)	Return 1 if at least one of B_1, B_2, B_3 is non-zero, 0 otherwise
B=And2(B_1, B_2)	Return 1 only if B_1 and B_2 are non-zero, 0 otherwise
B=And3(B_1, B_2, B_3)	Return 1 only if B_1, B_2, and B_3 are non-zero, 0 otherwise
B=Smaller(B_1, B_2)	Return 1 if B_1 is smaller than B_2, 0 otherwise
B=Not(B_1)	Return 0 if B_1 is non-zero, 1 otherwise

Table 6: Function set of GP chess player individual. *B:* Boolean, *F:* Float.

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

Table 7: Percent of wins, advantages, and draws for best GP-EndChess player in tournament against two top competitors.

The score for each game was derived from the outcome of the game. Players that managed to mate their opponents received more points than those that achieved only a material advantage. Draws were rewarded by a score of low value and losses entailed no points at all.

The final fitness for each player was the sum of all points earned in the entire tournament for that generation.

Control parameters and run termination. We used the standard reproduction, crossover, and mutation operators. The major parameters were: population size – 80, generation count – between 150 and 250, reproduction probability – 0.35, crossover probability – 0.5, and mutation probability – 0.15 (including ERC).

Results. We pitted our top evolved chess-endgame players 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*. Results are shown in Table 7. As can be seen, GP-EndChess manages to hold its own, and even win, against these top players. For more details on GP-EndChess see [13, 28].

Deeper analysis of the strategies developed [12] revealed several important shortcomings, most of which stemmed from the fact that they used deep knowledge and little search (typically, they developed only *one* level of the search tree). Simply increasing the search depth would not solve the problem, since the evolved programs examine each board very thoroughly, and scanning many boards would increase time requirements prohibitively. And so we turned to evolution to find an optimal way to overcome this problem: How to add more search at the expense of less knowledgeable (and thus less time-consuming) node evaluators, while attaining better performance. In [15] *we evolved the search algorithm itself*, focusing on the *Mate-In-N* problem: find a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N. We showed that our evolved search algorithms successfully solve several instances of the Mate-In-N problem, for the hardest ones developing 47% less game-tree nodes than CRAFTY. Improvement is thus not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements [15].

Finally, in [14], we examined a strong evolved chess-endgame player, focusing on the player's emergent capabilities and tactics in the context of a chess match. Using a number of methods we analyzed the evolved player's building blocks and their effect

Robocode Player

```
while (true)
    TurnGunRight(INFINITY); //main code loop

...

OnScannedRobot() {
    MoveTank(<GP#1>);
    TurnTankRight(<GP#2>);
    TurnGunRight(<GP#3>);
}
```

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

on play level. We concluded that evolution has found combinations of building blocks that are far from trivial and cannot be explained through simple combination—thereby indicating the possible emergence of complex strategies.

Example: Robocode

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

Terminal and function sets. We divided the terminals into three groups according to their functionality [27], as shown in Table 8:

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 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.

Terminal	Description
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 (Ephemeral Random Constant) in the range $[-1, 1]$
Random()	Returns a random real number in the range $[-1, 1]$
Zero()	Returns the constant 0

(a) Terminal set.

Function	Description
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) Function set (*F:* Float).

Table 8: Robocode representation.

Fitness measure. We explored two different modes of learning: using a fixed external opponent as teacher, and coevolution—letting the individuals play against each other; the former proved 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).

Control parameters and run termination. The major evolutionary parameters [19] were: population size – 256, generation count – between 100 and 200, selection method – tournament, reproduction probability – 0, crossover probability – 0.95, and mutation probability – 0.05. An evolutionary 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.

Results. We submitted our top player to the HaikuBot division of the international league. At its very first tournament it came in third, later climbing to first place of 28 (`robocode.yajags.com/20050625/haiku-1v1.html`). All other 27 programs, defeated by our evolved strategy, were written by humans. For more details on GP-Robocode see [27, 28].

Backgammon: Major results

We pitted our top evolved backgammon players against *Pubeval*, a free, public-domain board evaluation function written by Tesauro. The program—which plays well—has become the *de facto* yardstick used by the growing community of backgammon-playing program developers. 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. For more details on GP-Gammon see [2, 3, 28]

VIII The Future

Evolutionary computation is a fastly growing field. As shown above, difficult, real-world problems are being tackled on a daily basis, both in academia and in industry. In the future we expect major developments in the underlying theory. Partly spurred by this we also expect major new application areas to succumb to evolutionary algorithms, and many more human-competitive results. Expecting such pivotal breakthroughs may seem perhaps a bit of overreaching, but one must always keep in mind Evolutionary Computation's success in Nature.

References

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

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

[3] Yaniv Azaria and Moshe Sipper. GP-Gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, September 2005. doi: 10.1007/s10710-005-2990-0.

[4] M. Bain. *Learning Logical Exceptions in Chess*. PhD thesis, University of Strathclyde, Glasgow, Scotland, 1994. URL `citeseer.ist.psu.edu/bain94learning.html`.

[5] G. Bonanno. The logic of rational play in games of perfect information. Papers 347, California Davis - Institute of Governmental Affairs, 1989. available at http://ideas.repec.org/p/fth/caldav/347.html.

[6] N. Charness. Expertise in chess: The balance between knowledge and search. In K. A. Ericsson and J. Smith, editors, *Toward a general theory of Expertise: Prospects and limits*. Cambridge University Press, Cambridge, 1991.

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

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

[9] David B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Wiley-IEEE Press, third edition, February 2006. ISBN 0-471-66951-2.

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

[11] Johannes Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–161, September 1996. URL `citeseer.ist.psu.edu/furnkranz96machine.html`.

[12] A. Hauptman and M. Sipper. Analyzing the intelligence of a genetically programmed chess player. In *Late Breaking Papers at the 2005 Genetic and Evolutionary Computation Conference*, 2005.

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

[14] A. Hauptman and M. Sipper. Emergence of complex strategies in the evolution of chess endgame players. *Advances in Complex Systems*, 2007. (in press).

[15] A. Hauptman and M. Sipper. Evolution of an efficient search algorithm for the mate-in-n problem in chess. In *Proceedings of 10th European Conference on Genetic Programming (EuroGP2007)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2007. (in press).

[16] 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).

[17] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, Cambridge, Massachusetts, second edition, 1992.

[18] Selçuk Kilinç, Varun Jain, Varun Aggarwal, and Ugur Cam. Catalogue of variable frequency and single-resistance-controlled oscillators employing a single differential difference complementary current conveyor. *Frequenz: Journal of RF-Engineering and Telecommunications*, 60(7–8):142–146, July–August 2006.

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

[20] 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.

[21] 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*, pages 1171–1178. The MIT Press, Cambridge, Massachusetts, 2000.

[22] Jason D. Lohn, Gregory S. Hornby, and Derek S. Linden. An evolved antenna for deployment on NASA's Space Technology 5 mission. In Una-May O'Reilly, Tina Yu, Rick Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, volume 8 of *Genetic Programming*, chapter 18, pages 301–315. Springer, 2005. ISBN 978-0-387-23253-9. doi: 10.1007/0-387-23254-0_18.

[23] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2): 199–230, Summer 1995.

[24] C.-A. Peña-Reyes and M. Sipper. Fuzzy CoCo: A cooperative-coevolutionary approach to fuzzy modeling. *IEEE Transactions on Fuzzy Systems*, 9(5):727–737, October 2001.

[25] Stefan Preble, Michal Lipson, and Hod Lipson. Two-dimensional photonic crystals designed by evolutionary algorithms. *Applied Physics Letters*, 86(6):061111-1–3, February 2005. doi: 10.1063/1.1862783.

[26] H.-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, New York, 1995.

[27] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. GP-Robocode: Using genetic programming to evolve robocode players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano van Hemert, and Marco Tomassini, editors, *Genetic Programming: 8th European Conference, EuroGP 2005, Lausanne, Switzerland, March 30–April 1, 2005*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154. Springer, August 2005. ISBN 978-3-540-25436-2. doi: 10.1007/b107383.

[28] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel. Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 2007. (in press).

[29] Moshe Sipper. *Machine Nature: The Coming Age of Bio-Inspired Computing*. McGraw-Hill, July 2002. ISBN 0-071-38704-8.

[30] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, Summer 2002. doi: 10.1162/106365602320169811.

[31] Alan Mathison Turing. Computing machinery and intelligence. *Mind*, 59 (236):433–460, October 1950. URL http://links.jstor.org/sici?sici=0026-4423(195010)2:59:236<433:CMAI>2.0.CO;2-5.

[32] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. doi: 10.1109/4235.585893.

[33] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, September 1999.