Book Title	Encyclopedia of Machine Learning	
Chapter Number	00004	
Book CopyRight - Year	2010	
Title	Evolutionary Games	
Author	Particle	
	Given Name	**Moshe**
	Family Name	**Sipper**
	Suffix	**Prof.**
	Email	sipper@cs.bgu.ac.il
Affiliation	Division	Department of Computer Science
	Organization	Ben-Gurion University
	Street	P.O. Box 653
	Postcode	84105
	City	Beer-Sheva
	Country	Israel

E

Evolutionary Games

Moshe Sipper
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel
sipper@cs.bgu.ac.il

Definition

Evolutionary algorithms are a family of algorithms inspired by the workings of evolution by natural selection, whose basic structure is to

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*

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 1950s 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 cataloged 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 (Epstein, 1999).

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 (Laird & van Lent, 2000).

Recently, evolutionary algorithms have proven a powerful tool that can automatically "design" successful game-playing strategies for complex games (Azaria & Sipper, 2005a,b; Hauptman & Sipper, 2005b, 2007a,b; Shichel et al., 2005; Sipper et al., 2007).

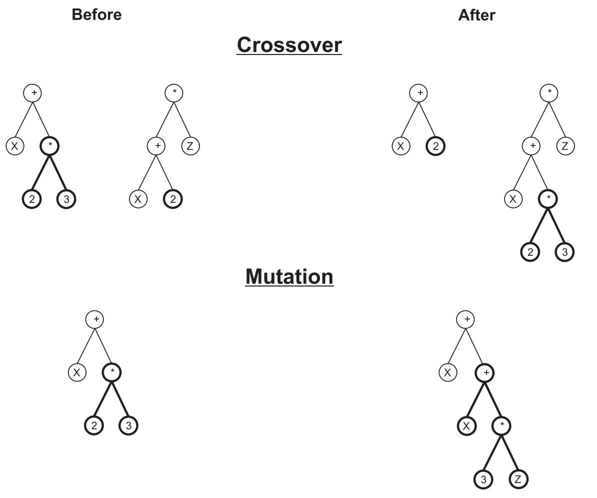
Structure of the Learning System

Genetic Programming

Genetic Programming is a subclass of evolutionary algorithms, wherein a *population* of individual LISP programs is evolved, each program 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.

The main mechanism behind genetic programming is precisely that of a generic evolutionary algorithm (Sipper, 2002; Tettamanzi & Tomassini, 2001), 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, that is, (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 the chosen genetic operator is used to construct the next generation. The most common operators are

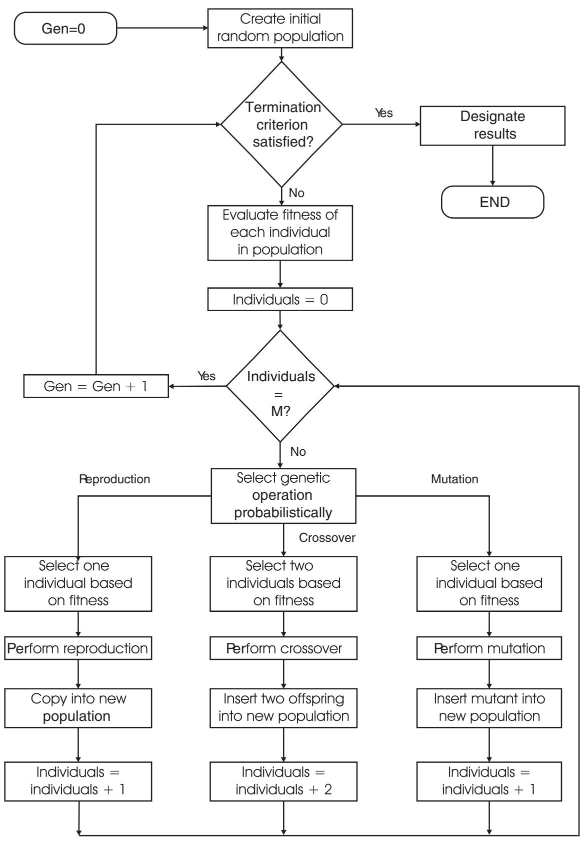


Evolutionary Games. Figure 1. **Genetic operators in genetic programming. LISP programs are depicted as trees. Crossover (*top*): Two subtrees (marked in *bold*) are selected from the parents and swapped. Mutation (*bottom*): A subtree (marked in *bold*) is selected from the parent individual and removed. A new subtree is grown instead**

- 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 subtrees rooted at these nodes. An example is shown in Fig. 1.
- Mutation (unary): Randomly select a node from the tree, delete the subtree rooted at that node, and then "grow" a new subtree in its stead. An example is shown in Fig. 1 (the growth operator as well as crossover and mutation are described in detail in Koza, 1992).

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

1. Program architecture
2. Set of terminals
3. Set of functions
4. Fitness measure


Gen=0
Create initial random population
Termination criterion satisfied?
Yes
Designate results
END
No
Evaluate fitness of each individual in population
Individuals = 0
Gen = Gen + 1
Yes
Individuals = M?
No
Reproduction
Select genetic operation probabilistically
Mutation
Crossover
Select one individual based on fitness
Select two individuals based on fitness
Select one individual based on fitness
Perform reproduction
Perform crossover
Perform mutation
Copy into new population
Insert two offspring into new population
Insert mutant into new population
Individuals = individuals + 1
Individuals = individuals + 2
Individuals = individuals + 1

Evolutionary Games. Figure 2. Generic genetic programming flowchart (based on Koza, 1992). 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

5. Control parameters
6. Manner of designating result and terminating run

Evolving Game-Playing Strategies

Recently, we have shown that complex and successful game-playing strategies can be attained via genetic programming. We focused on three games (Azaria & Sipper, 2005a,b; Hauptman & Sipper, 2005b, 2007a,b; Shichel et al., 2005; Sipper et al., 2007):

1. *Backgammon.* Evolves a full-fledged player for the non-doubling-cube version of the game (Azaria & Sipper, 2005a,b; Sipper et al., 2007).
2. *Chess* (endgames). Evolves a player able to play endgames (Hauptman & Sipper, 2005b, 2007a,b; Sipper et al., 2007). 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 (Shichel et al., 2005; Sipper et al., 2007).

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 (Koza, 1992). 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.

In a two-player game, such as chess or backgammon, players move in turn, each trying to win against the opponent according to specific rules (Hong, Huang, & Lin, 2001). The course of the game may be modeled using a structure known as an adversarial game tree (or simply game tree), in which nodes are the positions in the game and edges are the moves. By convention, the two players are denoted as MAX and MIN, where MAX is the player who moves first. Thus, all nodes at odd-numbered tree levels are game positions where MAX moves next (labeled MAX nodes). Similarly, nodes on even levels are called MIN nodes, and represent positions in which MIN (opponent) moves next.

The complete game tree for a given game is the tree starting at the initial position (the root) and containing all possible moves (edges) from each position. *Terminal nodes* represent positions where the rules of the game determine whether the result is a win, a draw, or a loss. Although the game tree for the initial position is an explicit representation of all possible paths of the game, therefore theoretically containing all the information needed to play perfectly, for most (nontrivial) games it is extremely large, and constructing it is not feasible. For example, the complete chess game tree consists of roughly 10^{43} nodes (Shannon, 1950).

When the game tree is too large to be generated completely, only a partial tree (called a search tree) is generated instead. This is accomplished by invoking a *search algorithm*, deciding which nodes are to be developed at any given time and when to terminate the search (typically at nonterminal nodes due to time constraints). During the search, some nodes are evaluated by means of an *evaluation function* according to given heuristics. This is done mostly at the leaves of the tree. Furthermore, search can start from any position and not just at the beginning of the game.

Because we are searching for a winning strategy, we need to find a good next move for the current player, such that no matter what the opponent does thereafter, the player's chances of winning the game are as high as possible. A well-known method called the *minimax* search (Campbell & Marsland, 1983; Kaindl, 1988) has traditionally been used, and it forms the basis for most methods still in use today. This algorithm performs a depth-first search (the depth is usually predetermined), applying the evaluation function to the leaves of the tree, and propagating these values upward according to the minimax principal: at MAX nodes, select the maximal value, and at MIN nodes – the minimal value. The value is ultimately propagated to the position from which the search had started.

With games such as backgammon and chess one can couple a current-state evaluator (e.g., board evaluator) with a next-move generator. One can then 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 created 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 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. Due to lack of space we shall elaborate upon one game – Robocode – and only summarize the major results for backgammon and chess.

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 subprogram. 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 (Fig. 3).

Terminal and Function Sets We divided the terminals into three groups according to their functionality (Shichel et al., 2005) (Fig. 4):

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.

```
Robocode Player's Code Layout

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

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

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)

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 fire-power and return 1; otherwise, do nothing and return 0

(b)

Evolutionary Games. Figure 4. Robocode representation. (a) Terminal set (b) Function set (F: Float)

2. Numerical constants: Two terminals, one providing the constant 0 and the other being an ephemeral random constant (ERC). 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.

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 just one but three external opponents were used to measure performance; these adversaries were 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 (Koza, 1992) 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 Shichel et al., (2005) and Azaria, Hauptman, and Shichel (2007).

Backgammon and Chess: Major Results

Backgammon 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 Azaria and Sipper (2005a) and Azaria et al. (2007).

Chess (endgames) 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 who play well *and fast*. Results are shown in Table 1. 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 Azaria et al., (2007) and Hauptman and Sipper (2005b).

Evolutionary Games. Table 1 **Percent of wins, advantages, and draws for the best GP-EndChess player in the tournament against two top competitors**

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

Deeper analysis of the strategies developed (Hauptman & Sipper, 2005a) 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 Hauptman and Sipper (2007b) *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 (Hauptman & Sipper, 2007b).

Finally, in Hauptman and Sipper (2007a), 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 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.

Cross References

►Evolutionary Computation
►Genetic Algorithms
►Genetic Programming

Recommended Reading

Azaria, Y., & Sipper, M. (2005a). GP-Gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines, 6*(3), 283–300.

Azaria, Y., & Sipper, M. (2005b). GP-Gammon: Using genetic programming to evolve backgammon players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, & M. Tomassini (Eds.), *Proceedings of 8th European conference on genetic programming (EuroGP2005), LNCS* (Vol. 3447, pp. 132–142). Heidelberg: Springer.

Campbell, M. S., & Marsland, T. A. (1983). A comparison of minimax tree search algorithms. *Artificial Intelligence, 20*, 347–367.

Epstein, S. L. (1999). Game playing: The next moves. In *Proceedings of the sixteenth National conference on artificial intelligence* (pp. 987–993). Menlo Park, CA: AAAI Press.

Hauptman, A., & Sipper, M. (2005a). Analyzing the intelligence of a genetically programmed chess player. In *Late breaking papers at the 2005 genetic and evolutionary computation conference*, GECCO 2005.

Hauptman, A., & Sipper, M. (2005b). GP-EndChess: Using genetic programming to evolve chess endgame players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, & M. Tomassini (Eds.), *Proceedings of 8th European conference on genetic programming (EuroGP2005), LNCS* (Vol. 3447, pp. 120–131). Heidelberg: Springer.

Hauptman, A., & Sipper, M. (2007a). Emergence of complex strategies in the evolution of chess endgame players. *Advances in Complex Systems, 10*(Suppl. 1), 35–59.

Hauptman, A., & Sipper, M. (2007b). Evolution of an efficient search algorithm for the mate-in-N problem in chess. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, & A. I. Esparcia-Alcázar (Eds.), *Proceedings of 10th European conference on genetic programming (EuroGP2007), LNCS* (Vol. 4445, pp. 78–89). Heidelberg: Springer.

Hong, T.-P., Huang, K.-Y., & Lin, W.-Y. (2001). Adversarial search by evolutionary computation. *Evolutionary Computation, 9*(3), 371–385.

Kaindl, H. (1988). Minimaxing: Theory and practice. *AI-Magazine, 9*(3), 69–76.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

Laird, J. E., & van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *AAAI-00: Proceedings of the 17th National conference on artificial intelligence* (pp. 1171–1178). Cambridge, MA: MIT Press.

Shannon, C. E. (1950). Automatic chess player. *Scientific American, 48*, 182.

Shichel, Y., Ziserman, E., & Sipper, M. (2005). GP-Robocode: Using genetic programming to evolve robocode players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, & M. Tomassini (Eds.), *Proceedings of 8th European conference on genetic programming (EuroGP2005), LNCS* (Vol. 3447, pp. 143–154). Heidelberg: Springer.

Sipper, M. (2002). *Machine nature: The coming age of bio-inspired computing*. New York: McGraw-Hill.

Sipper, M., Azaria, Y., Hauptman, A., & Shichel, Y. (2007). Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 37*(4), 583–593.

Tettamanzi, A., & Tomassini, M. (2001). *Soft computing: Integrating evolutionary, neural, and fuzzy systems*. Berlin: Springer.