DRAFT

GP-Gammon: Genetically Programming Backgammon Players

Yaniv Azaria, Moshe Sipper
Dept. of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel.
E-mails: {azariaya,sipper}@cs.bgu.ac.il. Web: www.moshesipper.com.

June 12, 2005

Abstract. We apply genetic programming to the evolution of strategies for playing the game of backgammon. We explore two different strategies of learning: using a fixed external opponent as teacher, and letting the individuals play against each other. We conclude that the second approach is better and leads to excellent results: Pitted in a 1000-game tournament against a standard benchmark player—*Pubeval*—our best evolved program wins 62.4% of the games, the highest result to date. Moreover, several other evolved programs attain win percentages not far behind the champion, evidencing the repeatability of our approach.

Keywords: genetic programming, backgammon, self-learning

1. Introduction

Games, long considered epitomic of human intelligence, have attracted many a researcher in artificial intelligence, ever since the field's prehistoric times (namely, the 1950s). Tic-tac-toe, checkers, chess, robotic soccer, and multifarious other games have been targeted by those wishing to study (and possibly enhance) machine intelligence. After all, what better proof of the latter than a machine beating us (literally) at our own game?

Specifically, board games such as checkers, Othello, and backgammon have all yielded to machine-learning techniques in the past decades. The basic rules are few and relatively easy to learn, however, excelling at the game is an altogether different matter. An ideal strategy—one that *always* wins—is usually impossible to obtain (either through human or computer design), but heuristics that perform well against human or machine opponents can be found (albeit with much effort). Commercial interests are also at stake since developing an efficient game strategy can readily be turned into a winning product (as evidenced by the multi-billion dollar computer-game industry).

Our research herein focuses on the game of backgammon,[1] which falls into the category of board games that do not yield to exhaustive analysis (and solution), but which yield to heuristic solving, that is, a heuristic strategy that performs very well against human and machine players can be obtained. The probabilistic nature of the game makes it suitable for learning [20]. The application of machine-learning techniques to obtain strong backgammon players has been done both in academia and industry. The best commercial products to date are Jellyfish [3] and TD-Gammon [20]. For these, suitable interfaces for benchmarking are unavailable, and there are no published results concerning their performance against other programs. Our benchmark competitor will thus be the freely available Pubeval (described below)—which has become a standard yardstick used by those applying AI techniques to backgammon.

The majority of learning software for backgammon is based on artificial neural networks, which usually receive as input the board configuration and produce as output the suggested best next move. The main problem lies with the network's fixed topology: The designer must usually decide upon this *a priori*, whereupon only the internal synaptic weights change. (Nowadays, one sometimes uses evolutionary techniques to evolve the topology [21]).

The learning technique we have chosen to apply is *Genetic Programming* (GP), by which computer programs can be evolved [7]. A prime advantage of GP over artificial neural networks is the automatic development of structure, *i.e.*, the program's "topology" need not be fixed in advance. In GP we start with an initial set of general- and domain-specific features, and then let evolution determine (evolve) the structure of the calculation (in our case, a backgammon-playing strategy). In addition, GP readily affords the easy addition of control structures such as conditional statements, which may also evolve automatically.

This paper details the evolution of highly successful backgammon players via genetic programming. In the next section we present previous work on machine-learning approaches to backgammon, along with a few examples of applications of GP to other games. In Section 3 we present our algorithm for evolving backgammon-playing strategies using genetic programming, with the presence of an external opponent as "teacher." Section 4 presents the self-learning approach to the problem, and in Section 5 we compare the two approaches. This is followed by Section 6 that discusses the evolved strategies. Finally, we present concluding remarks and future work in Section 7.

[1] Readers unfamiliar with the game may consult the appendix.

2. Previous Work

In 1989, Tesauro presented Neurogammon [18], a neural-network player evolved using supervised learning and several hand-crafted input features of the backgammon game. This work eventually led to TD-Gammon, one of the top two commercial products to date [20] (Section 1). This work is based on the Temporal Difference (TD) method, used to train a neural network through a self-playing model—*i.e.*, learning is accomplished by neural networks playing against themselves and thus improving.

In 1997, Pollack, Blair and Land [11] presented HC-Gammon, a much simpler Hill-Climbing algorithm that also uses neural networks. Under their model the current network is declared 'Champion,' and by adding Gaussian noise to the biases of this champion network a 'Challenger' is created. The Champion and the Challenger then engage in a short tournament of backgammon; if the Challenger outperforms the Champion, small changes are made to the Champion biases in the direction of the Challenger biases.

Another interesting work is that of Sanner *et al.* [15], whose approach is based on cognition (specifically, on the ACT-R theory of cognition [1]). Rather than trying to analyze the exact board state, they defined a representational abstraction of the domain, consisting of general backgammon features such as blocking, exposing, and attacking. They maintained a database of feature neighborhoods, recording the statistics of winning and losing for each such neighborhood. All possible moves were encoded as sets of the above features; then, the move with the highest win probability (according to the record obtained so far) was selected.

In 2001, Darwen [4] studied the coevolution of backgammon players using single- and multi-node neural networks, focusing on whether non-linear functions could be discovered. He concluded that with coevolution, there is no advantage in using multi-node networks, and that coevolution is not capable of evolving non-linear solutions.

Finally, Qi and Sun [13] presented a genetic algorithm-based multiagent reinforcement learning bidding approach (GMARLB). The system comprises several evolving teams, each team composed of a number of agents. The agents learn through reinforcement using the Q-learning algorithm. Each agent has two modules, Q and CQ. At any given moment only one member of the team is in control—and chooses the next action for the whole team. The Q module selects the actions to be performed at each step, while the CQ module determines whether the agent should continue to be in or relinquish control. Once an agent relinquishes control, a new agent is selected through a bidding process,

whereby the member who bids highest becomes the new member-in-control.

GP has been applied to games other than backgammon. In 2002, Gross *et al.* [5] applied GP to improve the heuristics for the existing scaffolding chess algorithm. One of us (MS) has successfully applied GP to two other games: Robocode and chess endgames. Robocode is a tank-fight simulator where (human) users submit Java-written tank programs, the object being to destroy your (tank) opponents. GP-Robocode was able to rank second in an international league, out of 27 contestants, with all other 26 being human written [16]. For chess, the GP-evolved GP-EndChess was able to draw or win against a Master-based strategy and against CRAFTY, which finished second in the 2004 Computer Chess Championship [6].

3. Evolving Backgammon-Playing Strategies using GP

We use Koza-style GP [7] to evolve backgammon strategies. In GP, a population of individuals evolves, where an individual is composed of LISP sub-expressions, each sub-expression being a program constructed from *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. Sensors are a special type of function that query the domain environment (in our case, backgammon board configurations).

In order to improve the performance of the GP system, we used *Strongly Typed Genetic Programming* (STGP) [10], which allows to add data types and data-type constraints to the LISP programs, thereby affording the evolution of more powerful and useful programs.

In STGP, each function has a *return type* and *argument types* (if there are any arguments). In our implementation, a type can be either an *atomic type*, which is a symbol, or a *set type*, which is a group of atomic types. A node n_1 can have a child node n_2 if and only if the return type of n_2 is compatible with the appropriate argument type of n_1. An atomic type is compatible with another atomic type if they are both identical, and a set type is compatible with another set type if they share at least one identical atomic type.

Note that the types are mere symbols and not real data types; their purpose is to force structural constraints on the LISP programs. The data passed between nodes consists only of real numbers.

3.1. Board evaluation

Tesauro [20] noted that due to the presence of stochasticity in the form of dice, backgammon has a high branching factor (about 20 moves on average for each of the 21 dice rolls), therefore rendering deep search strategies impractical. Thus, we opted for the use of a flat evaluator: after rolling the dice, generate all possible next-move boards, evaluate each one of them, and finally select the board with the highest score.

This approach has been used widely by neural network-based players and—as shown below—it can be used successfully with genetic programming. In our model, each individual is a LISP program that—using the sensors—receives a backgammon board configuration as input and returns a real number that represents the board score.

An artificial player is had by combining an (evolved) board evaluator with a program that generates all next-moves given the dice values.

3.2. Major preparatory steps

Koza [7] defined five major steps in preparing to use GP for problem solving:

1. Determining the set of terminals.
2. Determining the set of functions.
3. Determining the fitness measure.
4. Determining the parameters and variables of controlling the run.
5. Determining the method of designating a result and the criterion for terminating a run.

These steps are suitable for the case of evolving LISP programs whose architectures contain a single tree (one subroutine). As explained below, and as is often the case with non-trivial problems [7], LISP programs can consist of more than one tree. Therefore, we need to add a preliminary step to the list: Determining the program architecture.

3.2.1. *Program architecture*

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 probably 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 has 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.

One can argue that since the strategies of the two stages of the game are independent, it would be better to train contact and race individuals independently. However, the final 'product' of the evolutionary process is a complete individual that needs to win complete games, and not only one of the game stages. For example, to train a race individual would require generating unnatural board race configurations that would not represent the complete wide range of starting race configurations a backgammon game can produce. Therefore, it seems more natural to train the individuals for both stages together.

3.2.2. *Terminal set*

Keeping in mind our use of STGP, we need describe not only the terminals (and later the functions) but also their type constraints. We use two atomic types: *Float* and *Boolean*. We also use one set type—*Query*—that includes both atomic types.

With terminals, we use the ERC (*Ephemeral Random Constant*) mechanism, as described in Koza [7]. An ERC is a node that—when first initialized—is randomly assigned a constant value from a given range; this value does not change during evolution, unless a mutation operator is applied.

The terminal set is specific to our domain (backgammon), and contains three types of terminals:

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

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 1). The term 'Player' refers to the

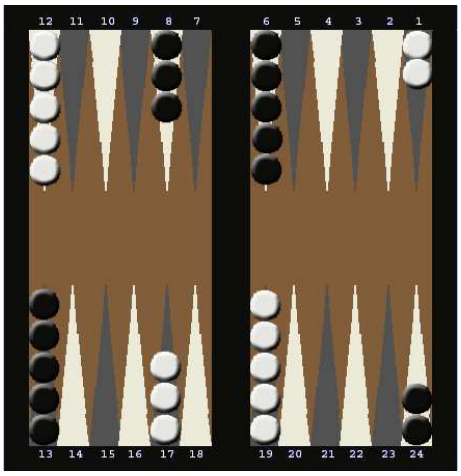

Figure 1. Initial backgammon configuration. The White player's home positions are labeled 19-24, and the Black player's home positions are labeled 1-6.

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 GP'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.

The terminal set for contact trees is given in Table I and that for race trees in Table II.

3.2.3. *Function set*
The function set contains no domain-specific operators, but only arithmetic and logic ones, so we use the same function set for both contact and race trees. The function set is given in Table III.

3.2.4. *Fitness measure*
Our first approach to measuring fitness is based on an external opponent in the role of a "teacher." 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—seems to have become the *de facto* yardstick used by the growing community of backgammon-playing program developers.

Table I. Terminal set of the contact tree. Note that zero-argument functions—which serve both as constants and as sensors—are considered as terminals. The double horizontal lines distinguish between the three types of terminals (see text).

F=Float-ERC	ERC – random real constant in range [0,5]
Q=Player-Exposed(n)	If player has exactly one checker at location n, return 1, else return 0
Q=Player-Blocked(n)	If player has two or more checkers at location n, return 1, else return 0
Q=Player-Tower(n)	If player has h or more checkers at location n (where $h \geq 3$), return $h - 2$, else return 0
Q=Enemy-Exposed(n)	If enemy has exactly one checker at location n, return 1, else return 0
Q=Enemy-Blocked(n)	If enemy has two or more checkers at location n, return 1, else return 0
F=Player-Pip	Return player *pip-count* divided by 167 (*pip-count* is the number of steps a player needs to move in order to win the game. This value is normalized through division by 167—the *pip-count* at the beginning of the game.)
F=Enemy-Pip	Return enemy *pip-count* divided by 167
F=Total-Hit-Prob	Return sum of hit probability over all exposed player checkers
F=Player-Escape	Measure the effectiveness of the enemy's barrier over its home positions. For each enemy home position that does not contain an enemy block, count the number of dice rolls that could potentially lead to the player's escape. This value is normalized through division by 131—the number of ways a player can escape when the enemy has no blocks
F=Enemy-Escape	Measure the effectiveness of the player's barrier over its home positions using the same method as above

Table II. Terminal set of the race tree.

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

Table III. Function set of the contact and race trees.

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

Several researchers in the field have pitted their own creations against Pubeval, some using it as teacher (external opponent) as well.

To evaluate fitness, we let each individual (backgammon strategy) play a 100-game tournament against Pubeval. Fitness is then the individual's score divided by the sum of scores of both players (individual and Pubeval).

3.2.5. *Control parameters*

The major parameters that control a run are: population size M (set to 128), and number of generations G (set to 500). GP has a few additional minor control parameters, which will be mentioned below.

When generating the initial random population, the method of creating each tree is Koza's [7] *Full-Builder*: A random integer d be-

tween *min-depth* and *max-depth* is chosen, and then a full tree of depth d is grown. After completing the creation of the first generation, the individuals are evaluated.

After the evaluation stage, we need to create the next generation of individuals from the current generation. This process involves two primary operators: *breeding* and *selection*. Of a finite set of breeding operators (described below), one is chosen probabilistically; then, one or two individuals (depending on the breeding operator) are selected from the current generation. Finally, the breeding operator is applied to the selected individual(s). This process continues until the population size has been reached—and the new generation thus created.

We use four breeding operators in our model, either unary (operating on one individual) or binary (operating on two individuals): *identity*, *sub-tree crossover*, *point mutation*, and *MutateERC*:

- The unary identity operator is the simplest one: 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.

- The binary sub-tree crossover operator randomly selects an internal node in each of the two individuals (belonging to corresponding trees—either race or contact) and then swaps the sub-trees rooted at these nodes.

- The unary point mutation operator randomly selects one node from one of the trees, deletes the subtree that is rooted at that node and grows a new sub-tree instead.[2]

- The unary MutateERC operator selects one random node and then mutates every ERC within the sub-tree that is rooted at that node. The mutation operation we used is the addition of a small Gaussian noise to the ERC. We used this breeding operator to achieve two goals: first, this is a convenient way to generate new constants as evolution progresses; and, second, it helps to balance the constants in good individuals. The MutateERC operation is described in [2].

As for selection, we chose *tournament selection*, as described in Koza [7]: randomly choose a small subset of individuals, and then select the one with the best fitness. This method is simple and affords a fair chance of selecting low-fitness individuals in order to prevent

[2] The details of crossing over sub-trees and growing new sub-trees due to mutation are fully described in Koza [7]. Bloat control is afforded by the software used, through the simple placement of upper bounds.

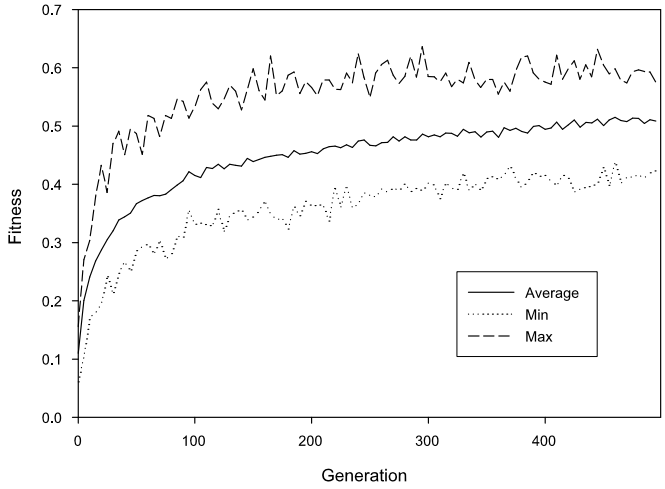

Figure 2. Fitness curve when using an external opponent. The fitness of an individual is the score it obtained in a 100-game tournament, divided by the sum of scores obtained by both players (the individual and Pubeval).

early convergence. GP has a few more minor parameters—*e.g.*, size of initial trees and probability of selecting each breeding operator—that are of less import.

3.2.6. *Termination criterion and result designation*

An ideal backgammon strategy, which wins whenever possible, does not exist, so our runs were terminated when reaching a fixed number of generations: 500. As for result designation, our goal is to find the best player possible, so every five generations we pitted the four individuals with the highest fitness in a 1000-game tournament against Pubeval, and the individual with the highest score in these tournaments, over the entire evolutionary run, was declared best-of-run.

3.3. Results: External opponent

Due to the stochasticity of our domain, we repeated each experiment 20 times. For each performance measure, we computed the average, minimum, and maximum values of the best-fitness individual every five generations over the 20 runs.

Our first measure of performance is the fitness (as defined in Subsection 3.2.4) curve of our experiments, shown in Figure 2.

A *prima facie* observation might lead to the conclusion that these results are remarkable; indeed, scoring over 60% in a backgammon tournament against Pubeval is an exceptional result that is far beyond

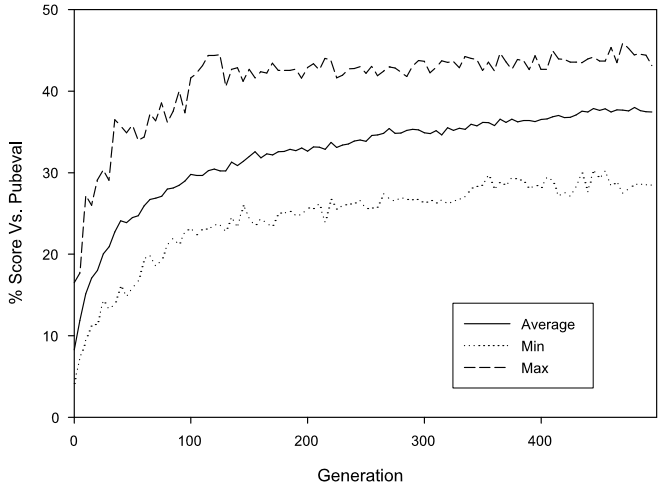

Figure 3. Benchmark curve when using an external opponent. The benchmark score of an individual is the score it obtained in a 1000-game tournament against Pubeval, divided by the sum of the scores obtained by both players (the individual and Pubeval).

the highest result ever published. Unfortunately, the fitness is computed using tournaments of 100 games, too short for a backgammon player benchmark.

In order to obtain a better indication of performance, we had the best-of-generation individual (according to fitness) play a 1000-game tournament against Pubeval. Figure 3 shows the results of this benchmark, where performance is seen to drop well below the 50% mark.

The results displayed in Figure 3, being more indicative of performance, raise the question of whether better players can be had. We answer in the affirmative in the next section.

4. Self Learning

The performance of our evolved strategies in Section 3 indicates that GP-based individuals are able to learn to play backgammon, but not necessarily to excel in it. One might think that when training against an external opponent, evolved individuals would be able to overpower this opponent (*i.e.*, win above 50% of the games)—a thought not borne out. Moreover, the evolved individuals are probably over-fitted to the strategy of Pubeval, casting doubt upon their generalization capabilities.

This observation led us to the next phase of experimentation—self-learning—described in this section: Instead of playing against an external opponent, individuals play against each other.

4.1. Major preparatory steps

To allow us to compare the performance of both learning methods as accurately as possible, most of the preparatory steps remain the same (as defined in Subsection 3.2): program architecture, sets of terminals and functions, and the control parameters.

In this experiment, the evolutionary process is internally driven, *i.e.*, *the evolving strategies play against each other* (and not against an external opponent). As such, the fitness of an individual is relative to its cohorts. To avoid overly lengthy evaluation times, methods such as Round Robin—where each individual is pitted against all others—were avoided. Through experimentation we concluded that a good evaluation method is the Single Elimination Tournament: 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.

One of the properties of Single Elimination Tournament is that half of the population is always assigned the same low fitness. Although there is a certain 'injustice' in having relatively good individuals receive the same fitness as others with poorer performance, this method has proven advantageous. Our preliminary experiments with 'fairer' methods, like round-robin, showed that they lead to premature convergence because bad individuals are rarely selected; preserving some amount of low-performance individuals allows the discovery of new strategies. On the other hand, an individual must exhibit a consistently good strategy in order to attain high fitness, and thus we are very likely to preserve good strategies.

4.2. Results: Self-learning

Figure 4 shows the benchmark curve vs. Pubeval of the individuals evolved through self-learning. Table IV shows how our evolved players fared against Pubeval, alongside the performance of the other approaches described in Section 2. The best player of each of our runs is

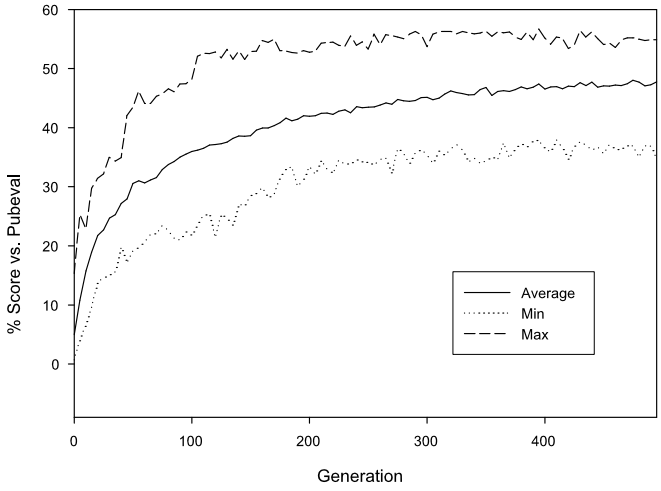

Figure 4. Benchmark curve when using self-learning. The benchmark score of an individual is the score it obtained in a 1000-game tournament against Pubeval, divided by the sum of the scores obtained by both players (the individual and Pubeval).

the individual that was designated according to the procedure described in Subsection 3.2.6.

4.2.1. *Computational resources*

On a standard workstation our system plays about 700–1,000 games a minute. As can be seen in Figure 4, to achieve good asymptotic performance our method requires on the order of 500,000–2,000,000 games (100–300 generations) per evolutionary run—about 2-3 days of computation. In comparison, GMARLB-Gammon required 400,000 games to learn, HC-Gammon – 100,000, and ACT-R-Gammon – 1000 games. The latter low figure is due to the explicit desire by ACT-R-Gammon's authors to model human cognition, their starting point being that a human can at best play 1,000 games a month (should he forego all other activities). Note that as opposed to the other individual-based methods herein discussed (e.g., employing one or a few neural networks), our approach is population based; the learning cost *per individual* is therefore on the order of a few thousand games.

Our primary goal herein has not been to reduce computational cost, but to attain the best machine player possible. As quipped by Milne Edwards (and quoted by Darwin in *Origin of Species*), "nature is prodigal in variety, but niggard in innovation." With this in mind, we did not mind having our processes run for a few days. After all, backgammon being a hard game to play expertly (our reason for choosing it), why

Table IV. Comparison of backgammon players. GP-Gammon-i designates the best GP strategy evolved at run i, which was tested in a tournament of 1000 games against Pubeval. (In comparison, GMARLB-Gammon used 50 games for evaluation, ACT-R-Gammon used 5000 games, Darwen used 10,000 games, and HC-Gammon used 200 games.) "Wins" refers to the percentage of wins against Pubeval.

Rank	**Player**	**Wins**	**Rank**	**Player**	**Wins**
1	GP-Gammon-1	56.8[a]	13	GP-Gammon-12	51.4
2	GP-Gammon-2	56.6	14	GMARLB-Gammon [13]	51.2[b]
3	GP-Gammon-3	56.4	15	GP-Gammon-13	51.2
4	GP-Gammon-4	55.7	16	GP-Gammon-14	49.9
5	GP-Gammon-5	54.6	17	GP-Gammon-15	49.9
6	GP-Gammon-6	54.5	18	GP-Gammon-16	49.0
7	GP-Gammon-7	54.2	19	GP-Gammon-17	48.1
8	GP-Gammon-8	54.2	20	GP-Gammon-18	47.8
9	GP-Gammon-9	53.4	21	ACT-R-Gammon [15]	45.94
10	GP-Gammon-10	53.3	22	GP-Gammon-19	45.2
11	GP-Gammon-11	52.9	23	GP-Gammon-20	45.1
12	Darwen [4]	52.7	24	HC-Gammon [11]	40.00

[a] Sanner *et al.* [15] quoted a paper by Galperin and Viola, who used TD(λ) training to purportedly obtain players with win percentage 59.25 against Pubeval. The reference for Galperin and Viola is of a now-obsolete URL, and all our efforts to obtain the paper by other means came to naught. Moreover, it seems to be but a short project summary and not a bona fide paper with full experimental details. Thus, the article does not meet two necessary criteria of a valid scientific publication: availability and repeatability. We have therefore not included their result herein.

[b] This is an average value over a number of runs. The authors cited a best value of 56%, apparently a fitness peak obtained during one evolutionary run, computed over **50 games.** This is too short a tournament and hence we cite their average value. Indeed, we were able to obtain win percentages of over 65% (!) for randomly selected strategies over 50-game tournaments, a result which dwindled to 40-45% when the tournament was extended to 1000 games.

Table V. Using the island model. I-GP-Gammon-i designates the best GP strategy evolved at distributed run i.

Rank	Player	Wins
1	I-GP-Gammon-1	62.4
2	I-GP-Gammon-2	62.2
3	I-GP-Gammon-3	62.1
4	I-GP-Gammon-4	62.0
5	I-GP-Gammon-5	61.4
6	I-GP-Gammon-6	61.2
7	I-GP-Gammon-7	59.1

should a machine learn rapidly? (see also [17]) Be that as it may, we do plan to tackle the optimization issue in the future.

4.3. Using additional resources

Wishing to improve our results yet further we employed a distributed asynchronous island model. In this experiment we used 50 islands, designated Island-0 thorough Island-49. Starting at generation 10, for each generation n, every Island i that satisfies $i \; mod \; 10 \; = \; n \; mod \; 10$, migrates 4 individuals to the 3 adjacent neighbors (a total of 12). Individuals are selected for migration based on fitness using tournament selection (Subsection 3.2.5) with repeats. The rest of the setup is identical to that of Section 4. Table V shows the improved results.

To get an idea of the human-competitiveness of our evolved players we referred to the HC-Gammon homepage [12], which presents statistics of games played by HC-Gammon [11] against human players. Accordingly, HC-Gammon wins 58% of the games when counting abounded games as wins, and 38% when disregarding them. Considering that HC-Gammon wins 40% of the games vs. Pubeval we expect, by transitivity, that our 62%-vs-Pubeval GP-Gammon is a very strong player in human terms.

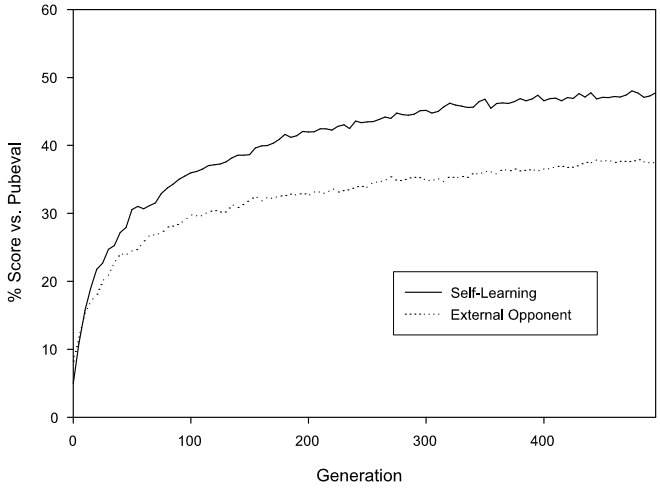

Figure 5. Comparing average benchmark performance of external-opponent and self-learning.

5. Comparing External-Opponent with Self-Learning

One would expect that strategies evolved using an external opponent and tested against the same program would perform much better (with respect to the benchmark program) than strategies that have been evolved without any prior knowledge of the benchmark strategy. Surprisingly, this is not the case here; it is clear that the performance of the self-learning approach is much better than the external approach: Figure 5 shows a comparison of average performance of the external-opponent and self-learning approaches and Figure 6 shows a comparison of maximum performance.

In order to explain these results, we should examine the learning model of both approaches. Population size and initialization method are the same for both; indeed, Figure 5 shows that the performance of the strategies of the first generation are the same for both approaches. Also, selection and breeding operators are identical, so that if, for instance, we assume the performances of the strategies at generation t for both approaches are equal, then their performances at generation $t+1$ should be equal, too.

This observation leads us to conclude that the key to the success of self-learning lies in the only difference between the two approaches: the fitness measure. With an external opponent, each individual is measured only by playing against Pubeval, which is known to be a

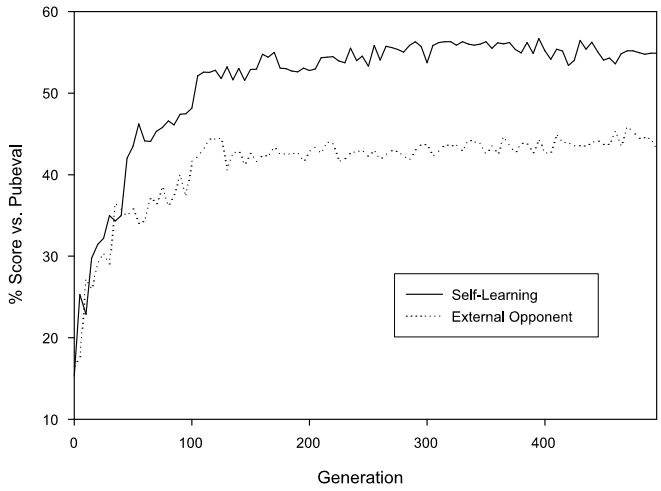

Figure 6. Comparing maximum benchmark performance of external-opponent and self-learning.

good player, but still far from perfect (as yet, no such player exists) and has its own advantages and weaknesses.

Backgammon players that gain experience by playing only with one other player, who does not improve and has only one fixed reply for each game configuration, are likely to form a strategy adapted to this particular environment, *i.e.*, to the external opponent's specific strategy, achieving a moderate score against it. However, in order to gain a significant and consistent advantage over the external opponent, a new strategy needs to be "discovered." As it turns out, the individuals were unable to discover such a novel strategy by playing only against Pubeval, and therefore they converged to a moderate level of performance.

On the other hand, with self-learning, individuals exhibiting good performance are likely to play against two or more different opponents at each generation. Moreover, the term "good performance" is relative to the performance of other individuals in the population and not to those of an external opponent, which performs much better at the beginning of evolution.

A human playing against many different opponents would probably fare better than one who has learned only from a single teacher, due to the fact that the former is exposed to many strategies and thus must develop responses to a wide range of game conditions. In terms of evolution, considering our domain, the fitness of an individual measured by playing backgammon against a variety of other individuals is

likely to be more reliable than fitness measured by playing only against Pubeval.

6. Playing Backgammon the GP Way

As is often the case with genetic programming, evolved individuals are highly complex, especially when the problem is a hard one—e.g., backgammon. Much like a biologist examining naturally evolved genomes, one cannot divine the workings of the program at a glance. Thus, we have been unable—despite intense study—to derive a rigorous formulation concerning the structure and contribution of specific functions and terminals to the success of evolved individuals (this we leave for future work). Rigorousness aside, though, our examination of many evolved individuals has revealed a number of interesting behaviors and regularities, hereafter delineated.

Recall that our terminal set contains two types of board-query functions: those that perform specific board-position queries (e.g., Player-Exposed(n) and Player-Blocked(n)), and those that perform general board queries (e.g., Enemy-Escape and Total-Hit-Prob). These latter are more powerful, and, in fact, some of them can be used as stand-alone heuristics (albeit very weak) for playing backgammon.

We have observed that general query functions are more common than position-specific functions. Furthermore, GP-evolved strategies seem to "ignore" some board positions. This should come as no surprise: the general functions provide useful information during most of the game, thus inducing GP to make use of them often. In contrast, information pertaining to a specific board position has less effect on overall performance, and is relevant only at a few specific moves during the game.

We surmise that the general functions form the lion's share of an evolved backgammon strategy, with specific functions used to balance the strategy by catering for (infrequently encountered) situations. In some sense GP strategies are reminiscent of human game-playing: humans rely on general heuristics (e.g., avoid hits, build effective barriers), whereas local decisions are made only in specific cases. (As noted above, the issue of human cognition in backgammon was central to the paper by Sanner *et al.* [15].)

7. Concluding Remarks and Future Work

7.1. Attribute 17

In their book, Koza *et al.* [9] 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. 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 current 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 backgammon strategies is GP'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* [14]. 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 [14]). Our language of design possesses several functions and terminals that attest to the presence of a (self-proclaimed) intelligent designer (Tables I, II, and III). This design language, which gives rise to a powerful language of observation in the form of successful players, was 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.

We believe that GP 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, ant algorithms), the GPer, owing to GP's representational affluence and openness, is better positioned to imbue the language of design with his own intelligence. 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. GP, as evidenced herein, is a forerunning candidate for the 'machine' part.

7.2. Future work

Our model divides the backgammon game into two main stages, thus entailing two types of trees. A natural question arising is that of refining this two-fold division into more sub-stages. The game dynamics may indeed call for such a refined division, with added functions and terminals specific to each game stage.

However, it is unclear how this refining is to be had: Any (human) suggestion beyond the obvious two-stage division is far from being obvious—or correct. One possible avenue of future research is simply to let GP handle this question altogether and evolve the stages themselves. For example, we can use a main tree to inspect the current board configuration and decide which tree should be used for the current move selection. These 'specific' trees would have their own separately evolving function and terminal sets. Automatically defined functions

(ADFs) [8] and architecture-altering operations [9] will most likely come in quite handy here.[3]

Our application of an adaptive—so-called "intelligent"—search technique in the arena of games is epitomic of an ever-growing movement. Our evolved backgammon players are highly successful, boding well for the future of GP-evolved strategies.

Appendix

Rules of Backgammon

The game of backgammon starts with the board configuration shown in Figure 1. The object of the game is to remove all of one's checkers (pieces) outside the board. In the figure, White moves along the positive direction (ascending board positions) while Black moves along the negative direction (descending board positions). Each player has *home* positions, as shown in Figure 1.

Each player casts in turn a pair of dice and moves as follows: If the dice show different values, the player moves two checkers according to the dice values (movement of one board position per one die point). If the two dice show identical values, the player moves four times according to the dice values (e.g., if the dice values are both 2, the player moves four times 2 positions). A player is not allowed to move onto board positions where two or more of the opponent's checkers are placed.

A *blot* occurs when a player has a single checker at some board position. In this case the opponent can "hit" this point, sending the player back to the position just before the beginning of the board (in Figure 1, position 0 for White, and position 25 for black; also known as the "bar"). Before moving another checker the player must re-enter the board at the opponent home positions, determined by the cast of dice. The player cannot move while any of his checkers remain on the bar.

When a player has all his checkers at his home positions (Figure 1), he can begin removing checkers outside the board. The player can remove a checker at the position corresponding to a die value. If, for example, one die shows 4, then White can remove the checker at position 21 in Figure 1. If this position is empty, the player has two

[3] Early experiments with ADFs in our current work produced lower results and—as non-ADF runs worked quite nicely—we decided to concentrate our efforts there. This does not preclude, however, the beneficial use of ADFs in the refinement of our methodology described in the paragraph.

options: 1) if the other home positions behind the current position (19 and 20, in our example) are empty, then the player can remove a checker from the position closest (22, 23, and 24, in our example) to that corresponding to the die value (21, in our example); 2) otherwise, the player cannot remove any checkers, and must resort to other possible non-checker-removing moves.

The game ends when one of the players has removed all his checkers off the board. If—as is often the case—the game in question is part of a series, then the winner is awarded one point. In case the loser still has all his checkers on the board, the winner is said to have won a *gammon*—and is awarded two points. A special case of gammon—called *backgammon*—occurs when a player wins and the opponent still has checkers in the player's home; the winner is then awarded three points (this latter case is extremely rare).

Acknowledgements

We are grateful to Assaf Zaritsky for helpful comments. Special thanks to Diti Levy for helping us with the drawing in Figure 1. We are grateful to the anonymous reviewers. Special thanks to Pierre Collet for his very detailed and constructive comments.

References

1. Anderson, J. R. and C. Lebiere: 1998, *The Atomic Components of Thought*. Mahwah, NJ: Lawrence Erlbaum Associates.
2. Chellapilla, K.: 1998, 'A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover'. In: J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (eds.): *Genetic Programming 1998: Proceedings of the Third Annual Conference*. University of Wisconsin, Madison, Wisconsin, USA, pp. 23–31.
3. Dahl, F.: 1998 -2004, 'JellyFish Backgammon'. http://www.jellyfish-backgammon.com.
4. Darwen, P.: 2001, '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, pp. 1003–1010.
5. Gross, R., K. Albrecht, W. Kantschik, and W. Banzhaf: 2002, 'Evolving Chess Playing Programs'. In: 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.): *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. New York, pp. 740–747.
6. Hauptman, A. and M. Sipper: 2005, 'GP-EndChess: Using genetic programming to evolve chess endgame players'. In: *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*. (to appear).
7. Koza, J. R.: 1992, *Genetic programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.
8. Koza, J. R.: 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Massachusetts: MIT Press.
9. Koza, J. R., F. H. Bennett III, D. Andre, and M. A. Keane: 1999, *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, California: Morgan Kaufmann.
10. Montana, D. J.: 1995, 'Strongly Typed Genetic Programming'. *Evolutionary Computation* **3**(2), 199–230.
11. Pollack, J. B., A. D. Blair, and M. Land: 1997a, 'Coevolution of a Backgammon Player'. In: C. G. Langton and K. Shimohara (eds.): *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA, pp. 92–98.
12. Pollack, J. B., A. D. Blair, and M. Land: 1997b, 'DEMO Lab's HC-Gammon'. http://demo.cs.brandeis.edu/bkg.html.
13. Qi, D. and R. Sun: 2003, 'Integrating reinforcement learning, bidding and genetic algorithms'. In: *Proceedings of the International Conference on Intelligent Agent Technology (IAT-2003)*. pp. 53–59.
14. Ronald, E. M. A., M. Sipper, and M. S. Capcarrère: 1999, 'Design, Observation, Surprise! A Test of Emergence'. *Artificial Life* **5**(3), 225–239.
15. Sanner, S., J. R. Anderson, C. Lebiere, and M. Lovett: 2000, 'Achieving Efficient and Cognitively Plausible Learning in Backgammon'. In: P. Langley (ed.): *Proceedings of the 17th International Conference on Machine Learning (ICML-2000)*. Stanford, CA, pp. 823–830.
16. Shichel, Y., E. Ziserman, and M. Sipper: 2005, 'GP-Robocode: Using genetic programming to evolve robocode players'. In: *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*. (to appear).

17. Sipper, M.: 2000, 'A Success Story or an Old Wives' Tale? On Judging Experiments in Evolutionary Computation'. *Complexity* **5**(4), 31–33.
18. Tesauro, G.: 1989, 'NEUROGAMMON: A Neural-Network Backgammon Learning Program'. *Heuristic Programming in Artificial Intelligence* **1**(7), 78–80.
19. Tesauro, G.: 1993, 'Software–Source Code Benchmark player "pubeval.c"'. http://www.bkgm.com/rgb/rgb.cgi?view+610.
20. Tesauro, G.: 1995, 'Temporal Difference Learning and TD-Gammon'. *Communications of the ACM* **38**(3), 58–68.
21. Yao, X.: 1999, 'Evolving artificial neural networks'. *Proceedings of the IEEE* **87**(9), 1423–1447.