GP-Gammon: Using Genetic Programming to Evolve Backgammon Players

Yaniv Azaria and Moshe Sipper

Department of Computer Science, Ben-Gurion University, Israel
{azariaya,sipper}@cs.bgu.ac.il, www.moshesipper.com

Abstract. We apply genetic programming to the evolution of strategies for playing the game of backgammon. Pitted in a 1000-game tournament against a standard benchmark player—*Pubeval*—our best evolved program wins 58% of the games, the highest verifiable result to date. Moreover, several other evolved programs attain win percentages not far behind the champion, evidencing the repeatability of our approach.

1 Introduction

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 next best 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 [1]).

The learning technique we have chosen to apply is *Genetic Programming* (GP), by which computer programs can be evolved [2]. 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 and loop 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. In Section 3 we present our algorithm for evolving backgammon-playing strategies using genetic programming. Section 4 presents results, followed by Section 5, wherein we conclude and describe future work.

2 Previous Work

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 [4]. Being commercial, with their innards unavailable for any scrutiny, we shall remain herein in the academic arena. Our benchmark competitor will thus be the freely available Pubeval—which has become a standard yardstick used by those applying AI techniques to backgammon. Pubeval is quite a strong machine player, trained on a database of expert preferences using comparison training [5].

Tesauro's approach is based on the Temporal Difference method, used to train a neural network through a self-playing model—i.e., learning is accomplished by programs playing against themselves and thus improving [4].

In 1997, Pollack, Blair, and Land [5] 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.* [6], whose approach is based on cognition (specifically, on the ACT-R theory of cognition [7]). 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 maintain a database of feature neighborhoods, recording the statistics of winning and losing for each such neighborhood. All possible moves are encoded as sets of the above features; then, the move with the highest win probability (according to the record obtained so far) is selected.

Finally, Qi and Sun [8] presented a genetic algorithm-based multi-agent 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.

3 Evolving Backgammon-Playing Strategies using Genetic Programming

We use Koza-style GP [2] 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 LISP program constructed from *functions* and *terminals*. The functions are usually arithmetic and logical 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) [9], 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.

Board evaluation. Tesauro [4] noted that due to the presence of stochasticity in the form of dice, backgammon has a high branching factor, 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.

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 function set of the contact tree is richer and contains various general and specific board query functions. The function set of the race tree is much smaller and contains only functions that examine the checkers' positions. This is because at the race phase, the moves of each player are independent of the opponent's status, and thus are much simpler.

Functions and terminals. Keeping in mind our use of STGP, we need to describe not only the functions and terminals 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.

The function set contains no domain-specific operators, but only arithmetic and logical ones, so we use the same function set for both contact and race trees. The function set is given in Table 1.

Table 1. 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

With terminals we use the ERC (*Ephemeral Random Constant*) mechanism, as described in Koza [2]. An ERC is a node that—when first initialized—is 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 functions:

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 contender whose turn it is, while 'Enemy' refers to the opponent. After completing the move, the contenders are swapped. 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. The last type of terminal is a function that provides general information about the board as a whole.

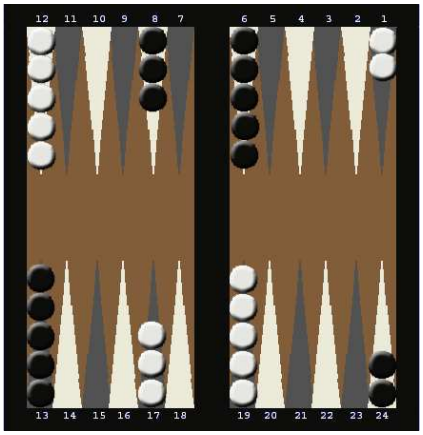

Fig. 1. Initial backgammon configuration. The White player's home positions are labeled 19-24, and the Black player's home positions are labeled 1-6.

The terminal set for contact trees is given in Table 2 and that for race trees in Table 3.

Fitness measure. 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 rest $\frac{n}{2}$ 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.

Breeding strategy. 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).

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

- The unary reproduction 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.

Table 2. Terminal set of the contact tree. Note that zero-argument functions—which serve both as constants and as sensors—are considered as terminals.

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 his 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 his home positions using the same method as above

Table 3. 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

- The binary 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 mutation operator randomly selects one node from one of the trees, deletes the subtree that is rooted at that node and grows a new subtree instead. (Crossover and mutation are described in detail in Koza [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 on good individuals. The MutateERC operation is described in [10].

We chose a selection method that supports relative fitness—*tournament selection*, as described in Koza [2]: randomly choose a small subset of individuals, and then select the one with the best fitness. This method is simple, respects the relative fitness scale, and also affords a fair chance of selecting low-fitness individuals in order to prevent early convergence.

4 Results

For benchmark purposes we used *Pubeval*—a free, public-domain board evaluation function written by Tesauro [11]. The program—which plays very well—seems to have become the *de facto* yardstick used by the growing community of backgammon-playing program developers. Several researchers in the field have pitted their own creations against Pubeval.

Our population consisted of 128 individuals, which evolved for 500 generations. We used the ECJ GP System of Luke [12]. We repeated the experiment 20 times and calculated the average, minimum, and maximum benchmark values every five generations. Figure 2 shows the benchmark curve of our individuals. Table 4 shows how our best evolved players fared against Pubeval, alongside the performance of the other approaches described in Section 2.

To get an idea of the human-competitiveness of our players we referred to the HC-gammon statistics (`demo.cs.brandeis.edu/hcg/stats1.html`), according to which HC-Gammon wins 58% of the games when counting abounded games as wins, and 38% when not counting them. Considering that HC-Gammon wins 40% of the games versus Pubeval, we expect—by transitivity—that GP-gammon (with win percentage of 56% vs. Pubeval) is a very strong player in human terms.

On a standard workstation our system plays about 700–1,000 games a minute. As can be seen in Figure 2, 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

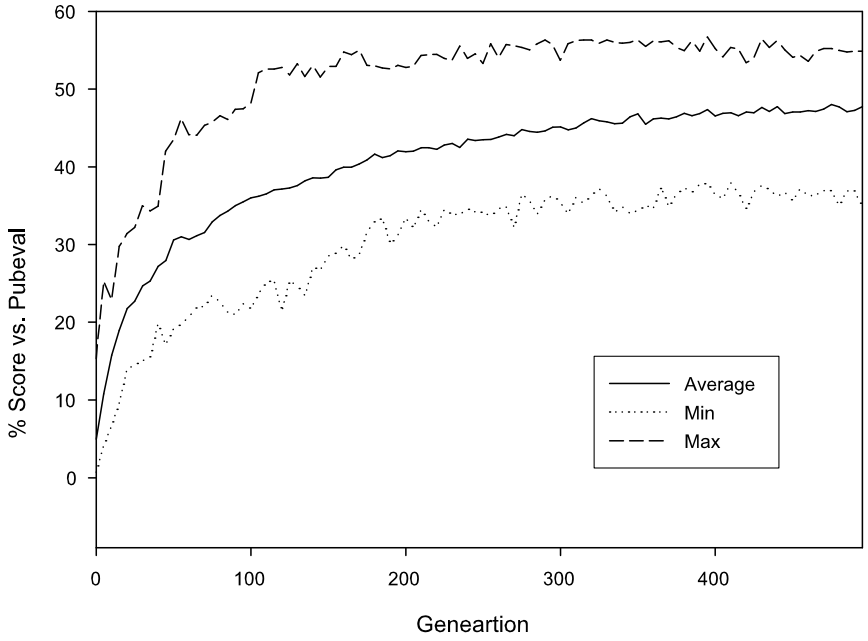

Fig. 2. Benchmark curve. The benchmark score of an individual equals the score it attained in a 1000-game tournament against Pubeval, divided by the sum of scores gained by both the individual and Pubeval.

Table 4. 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. Only the top 5 runs are shown (out of 20). For ACT-R-Gammon and HC-Gammon, the values cited are the best values obtained. For GMARLB-gammon, 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.

Player	**% Wins vs. Pubeval**
GP-Gammon-1	56.8
GP-Gammon-2	56.6
GP-Gammon-3	56.4
GP-Gammon-4	55.7
GP-Gammon-5	54.6
GMARLB-Gammon [8]	51.2
ACT-R-Gammon [6]	45.94
HC-Gammon [5]	40.00

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 should a machine learn rapidly? (see also [13]) Be that as it may, we do plan to tackle the optimization issue in the future.

5 Concluding Remarks and Future Work

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 function 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.* [6].)

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) [14] and architecture-altering operations [15] will most likely come in quite handy here.[1]

GP is known to be computer-intensive, being both memory- and time-avaricious. Witness Koza's use of a 1,000-Pentium cluster[2] and populations of up to 10,000,000 individuals [16]. Unfortunately, our own resources were limited to but a few workstations. We believe quite firmly that upping the resources will lead to the evolution of much better players. Part of our belief stems from a few multi-cpu experiments, which we performed on a cluster of workstations that were made available to us for a short period of time. Jumping at the occasion, we were able to attain a win percentage of 58% against Pubeval—the best known result to date. Hopefully, we will gain access to more resources in the future, thereby attempting to improve our players yet further.

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, able to beat previous automatically obtained strategies.

Acknowledgements

We are grateful to Assaf Zaritsky for helpful comments. Special thanks to Diti Levy for helping us with the drawing in Figure 1. The research was partially supported by the Lynn and William Frankel Fund for Computer Science.

[1] 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.

[2] `www.genetic-programming.com/machine1000.html`

References

1. Yao, X.: Evolving artificial neural networks. Proceedings of the IEEE **87** (1999) 1423–1447
2. Koza, J.R.: Genetic programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA (1992)
3. Dahl, F.: JellyFish Backgammon. (1998-2004) http://www.jellyfish-backgammon.com.
4. Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM **38** (1995) 58–68
5. Pollack, J.B., Blair, A.D., Land, M.: Coevolution of a backgammon player. In Langton, C.G., Shimohara, K., eds.: Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems, Cambridge, MA, MIT Press (1997) 92–98
6. Sanner, S., Anderson, J.R., Lebiere, C., Lovett, M.: Achieving efficient and cognitively plausible learning in backgammon. In Langley, P., ed.: Proceedings of the 17th International Conference on Machine Learning (ICML-2000), Stanford, CA, Morgan Kaufmann (2000) 823–830
7. Anderson, J.R., Lebiere, C.: The Atomic Components of Thought. Lawrence Erlbaum Associates, Mahwah, NJ (1998)
8. Qi, D., Sun, R.: Integrating reinforcement learning, bidding and genetic algorithms. In: Proceedings of the International Conference on Intelligent Agent Technology (IAT-2003), IEEE Computer Society Press, Los Alamitos, CA (2003) 53–59
9. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3** (1995) 199–230
10. Chellapilla, K.: A preliminary investigation into evolving modular programs without subtree crossover. In Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R., eds.: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 23–31
11. Tesauro, G.: Software–Source Code Benchmark player "pubeval.c". (1993) http://www.bkgm.com/rgb/rgb.cgi?view+610.
12. Luke, S.: ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System. (2000) http://www.cs.umd.edu/projects/plus/ec/ecj/.
13. Sipper, M.: A success story or an old wives' tale? On judging experiments in evolutionary computation. Complexity **5** (2000) 31–33
14. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, Massachusetts (1994)
15. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, San Francisco, California (1999)
16. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA (2003)