# Evolving a Backgammon Player at Home

Arieh Bibliowicz
vainolo@vipe.technion.ac.il

Eitan Joshua
eitanj@galileo.co.il

16th March 2005

## 1 Introduction

A main goal in the field of Artificial Intelligence is to create machines (i.e. computer programs) that can reason as humans, or even better. Reasoning means many things, but can be simplified as a series of events, starting from the captation of the environment, understanding it, deciding on an action and acting on this decision. It is difficult to draw a line between each of these events, but even so, each one of then is a field of research on it's own. In this paper we concentrate on the decision making, although our work also goes into understanding the environment.

A widely accepted way of simulate reasoning is through game playing. Ever since Shannon's proposal for a chess-playing algorithm [1] and Samuel's development of a checkers learning program [2] the domain of complex games like Chess, Go, Backgammon and others has been widely used as a testing ground for different ideas in artificial intelligence and machine learning. The reason for this is that these games have a high degree of complexity and sofistication while at the same time their rules (the environment) are well defined, it is easy to simulate the game play and it's easy to determine when

the game has terminated.

# 2 Background

The effort to develop computer game players has given fruit in some types of games, while in others it hasn't. One field where a computer has shown a level of play matching that of a human grandmaster in chess, where Deep Blue II, developed by IBM, defeated the world champion Kasparov. Many researches have pointed out that the work done on chess is not relevant since most (if not all) of it's performance comes from the fact that it can search to a depth of 14 levels, examining millions of boards using a parallel custom-designed hardware. To quote Noam Chomsky [3], the research is "about as interesting as the fact that a bulldozer can lift more than some weight lifter". The game of Go is the common example of a problem that has not been solved, since to date we have not seen a Go playing program that can match the ability of a simple amateur.

## 2.1 Backgammon

Backgammon has drawn special interest for its special properties. Unlike Chess or Checkers, the number of possible alternatives (the "branching factor") in backgammon is too large to do a brute-force lookahead search, because there are so many possible board positions: two dice give 21 possible rolls, each which offers an average of 20 moves, resulting in a branching factor of several hundreds per ply [4]. Another interesting quality of backgammon is it's included randomness create by the dice rolled, which complicates the decision making because we can never know the dice that will appear in the future and they may change the value of our evaluation substantially.

Therefore it must have been a great surprise when in 1992 and later in 1995, Tesauro created a player that is able to compete with the best backgammon players in the world [4]. Tesauro's player is based on a Neural Network organized in a standard multilayer perception architecture. To bring the player to it's current level of play, TD-Gammon was trained using Tempo-

ral Difference (form here the TD). This algorithm is based on self play, in which the player plays against itself to learn the game and at the end of the game the player's weights are updated to learn from the mistakes and good evaluations done in the game. This player achieved a very high level of play by running hundreds of thousands of games against itself, and by many hand-crafted features added by the designers of the players. Tesauro also another player using TD, called Pubeval, that is widely used as a benchmarking player in backgammon research since it's internal data is public.

Another approach taken is Co-Evolution of the players. Pollack et. al. [5] developed a strong player by playing games in pairs between a champion player and a challenger created from the champion adding random noise to it's weights. Although they achieved good results after tens of thousands of generations, their player did not match even the level of play achieved by Pubeval. Darwen [6] created a better player by using an evolutionary algorithm. With an initial population of 200 players, who player all against all to achieve a ranking, after which players were selected for the next round, mutated and crossed-over. His best player achieved more than 50% wins against Pubeval after more than 400 generations of evolution, while using huge amounts of computer power.

## 2.2 Learning Games with Co-Evolution

A common way to create computer player is for human experts to hand write an *evaluation function* to judge the quality of a board game. This approach is unsuited for many tasks since the experts may not be available or even if they are, the way they think may be hard to translate into an algorithm.

For example (taken from [6]): imagine trying to evolve art with an evaluation function programmed by by Picasso. If nice pictures result, is that due to machine learning or to Picasso intelligence's human skill? Is it artificial intelligence or just a glorified editing tool. (remember Deep Blue?) The real problem stands in creating a Picasso quality paintings without Picasso.

In co-evolution the hypothesis (in our case, game players) are ranked on the basis of how they perform against the other hypothesis in the same

generation, or perhaps another population evolving in parallel [7]. Thus co-evolution is a way to discover solutions to problems **without** the help of external expertise. Co-evolution has worked well on the game of Checkers [8], simple chasing games [9]and other non-game tasks such as scheduling [10], Neural Network design [11] and Pattern Matching [12], to name a few.

# 3 Experimental Setup

We decided to create a Backgammon player using a feed-forward neural network with a fixed structure and evolutionary algorithms. Each player was seen as a *strategy* for the game of backgammon. A population of strategies play against each other for a fixed number of games and the fitness of an individual is taken to be the number of games won out of the number of games played.

Each strategy in the population is a *move evaluation* function. Each time the player moves, a pair of pseudo-random dice are rolled and a legal move generator lists all the possible game positions that can be reached with those dice. This gives us a one-move look-ahead search. The move evaluation function (a player) returns a value for each of the possible moves and the move with the highest value is taken.

Our move evaluator has 136 inputs taken directly from the board as follows, without any hand-written feature detectors or special encoding:

- 5 inputs for each point in the board, where 4 inputs are used to encode the number of checkers on the point (to a maximum of 15) in binary format, plus one input for the color of the checkers, that receives 0 for white checkers and 1 for black checkers.

- 4 inputs for each bar and home point, since they are owned by the player an only his checkers can populate it. The number of checkers in these points is also encoded in binary format.

This encoding is the raw encoding of the board, therefore the player must not only evaluate the board, it must also learn to interpret it's input to achieve better results.

The network consists of 20 hidden node each feeded the output of all input nodes, and one output node connected to them all, that returns the value of the board. The number of hidden nodes was chosen ad-hoc, but derives from the results achieved by a similar architecture in [4]. All weights are represented as real numbers and all non-input nodes have a variable bias. The initial weights are set to zero, with a random mutation for each of the players in the initial population.

Evolution proceeded as follows, after the ranking of the players has been calculated:

1. Players pass to the next generation with a probability directly proportional to it's ranking. No elitism was used.

2. Players are merged to fill the empty space left by the players removed. Merging is done on the neural network level. A pair of players is chosen (the order on which they were chosen depended directly on their ranking) and merged with a probability of the sum of their rankings divided by 2. The merging is done as follows: A random *merging factor* $\mathbf{f}$ between 0 and 1 is calculated, and a new player is created with a neural network who's weights $w_i$ are derived from the two parents. Formally, if players $p_1$ and $p_2$ merge with a merging factor $f$, then the resulting weight of the new network is $w_i^{p_1} \cdot f + w_i^{p_2} \cdot (1 - f)$. This mutation was chosen because it kept the player's *heavy* features (weights with high value).

3. Players are mutated with a 10% probability. A network weight is chosen at random and random Gaussian noise with mean 0 and standard deviation 0.1 is added to the weight, giving small changes to reduce major disruptions.

Players may suffer both merging and mutation.

Formally, evolution proceeds as follows:

As you can see, the basic evolutionary algorithm is very simple. The first procedure is straightforward so we won't expand into it, but one important

**Algorithm 1** Player Evolution

**Input:**

n - number of players.
x - number of neurons in hidden layer of NN.
k - number of generations to evolve.
g - number of pairwise games played.

**Algorithm**

Create $n$ initial players with x hidden neurons
Mutate every player once
While numOfGenerations < k Do
    Play all players against all players g times.
    Kill Players.
    Merge Players.
    Mutate Players.
Loop

**Algorithm 2** Kill Players

For i = 1 to n Do
    p = Normal Random $\in [0, 1]$
    If p > player[i].fitness Then
        Remove player[i] from Players
Loop

thing in it is the saving of each players **fitness**. The three remaining procedures are explained below. We follow the array notation for a player in the group of players: *player[i]* is the $i^{th}$ player in the Players vector, and the dot notation for properties of elements: *player[i].fitness* is the fitness of the $i^{th}$ player. We assume the procedures have all inputs available to the basic algorithm, plus the player vector and ranking array.

First of all, Kill Players. This procedure removes a player form the population with probability *fitness[i]*.

Merge Players is a more complicated function since we had to think of a way for good players to merge more than bad players. We use to helper functions: *Get Player With Max Fitness()* - returns the player with the maximum fitness, *Get Player With Next Fitness( fitness )* - returns the player with the next highest fitness after the given fitness, *Merge Players( Player1, Player2, x )* - merge player1 and player2 with merging factor x, as described

**Algorithm 3** Merge Players

```
While true Do
    P1 = Get Player With Max Fitness()
    While Players.size < n Do
        P2 = Get Player With Next Fitness( P1.fitness )
        mutualFitness = (P1.fitness+P2.fitness)/2
        p = Normal Random ∈ [0,1]
        If p < mutualFitness Then
            q = = Normal Random ∈ [0,1]
            Players.add( Merge Players( P1, P2, q ) )
        P2 = Get Player With Next Fitness( P2.fitness )
        If P2 is null Then
            P1 = Get Player With Next Fitness( P1.fitness )
            P2 = Get Player With Next Fitness( P1.fitness )
    Loop
Loop
```

**Algorithm 4** Mutate Players

```
For i = 1 to n Do
    p = Normal Random ∈ [0,1]
    If p < 0.1 Then
        Mutate( player[i] )
Loop
```

before, by merging their NN. If there are two players with the same fitness, they are return in the same order they were entered in the Players array. We came at last to the following algorithm:

Last but not least, Mutate Players adds mutation to the evolution. Mutation is done by adding random Gaussian noise to a random weight in the player's NN:

# 4 Platform Description

We programmed the system with Java using an Object Oriented methodology. It consists of four parts: the playing environment, the game playing system, the players and the tests.

## 4.1 Playing Environment

This part of the program provides all the functionality necessary to play backgammon. It includes:

**Board:** Where the game is played, keeps the consistency of the board, and provides functions to expand possible board moves for given dice.

**Dice:** Random (hopefully) dice.

## 4.2 Game Playing Environment

Provides functionality to play backgammon games. It includes:

**Game:** Connects between players and the board environment to play a game of backgammon

**GroupEvolution:** Component is in charge of the evolution algorithm.

## 4.3 Players

Provides different types of players. Since the Neural Network is used only on the players, we decided to include it in this part of the system and not as a stand alone component. In this part we have:

**NeuralPlayer:** A player whose evaluation function is based on a Neural Network

**PubEvalPlayer:** Player based on Thesauro's Pubeval player

**InteractivePlayer:** Player used for interactive playing.

## 4.4 Tests

We provided one test:

**PubEvalTest:** Test a player against pubeval.

# 5 Experiments

On the first program run, we evolved 20000 generations of players using the algorithm defined above, saving all the players every 10 generations. To evaluate the performance of the players, we used an external player, in this case Pubeval, a player created by Tesauro which is ranked as being a good player. All players every 100 generation played 50 games against PubEval.

After the results of the first program run, we decided to change the way that players merged. In the second run, the merging factor was set to 0.5, which means that half of the weight comes from each of the players. We think that this would make this operator less disruptive. We evolved 10000 generations with this algorithm and tested them the same way as before.

# 6 Results

To our greatest delight, the results achieved by our players were very satisfying. In out initial tests, we found that our players did evolve, in a way, achieving a good level of play, by all standards.

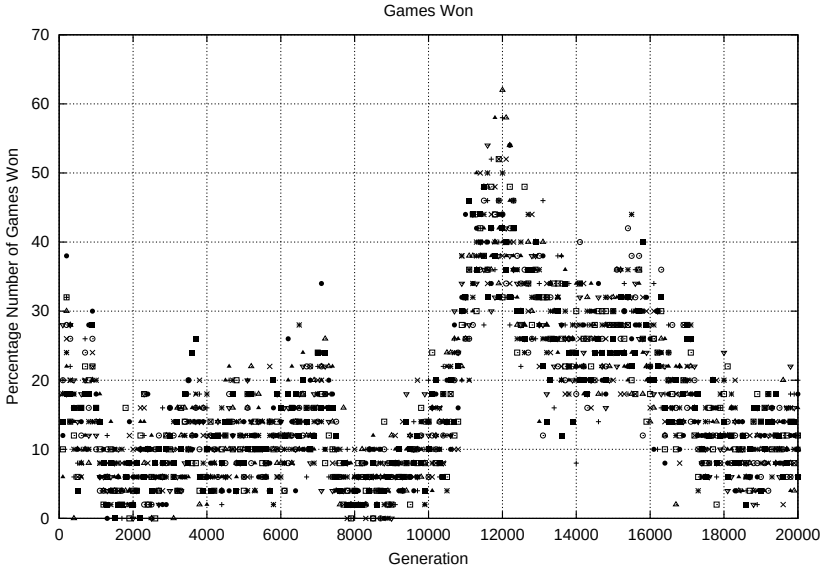
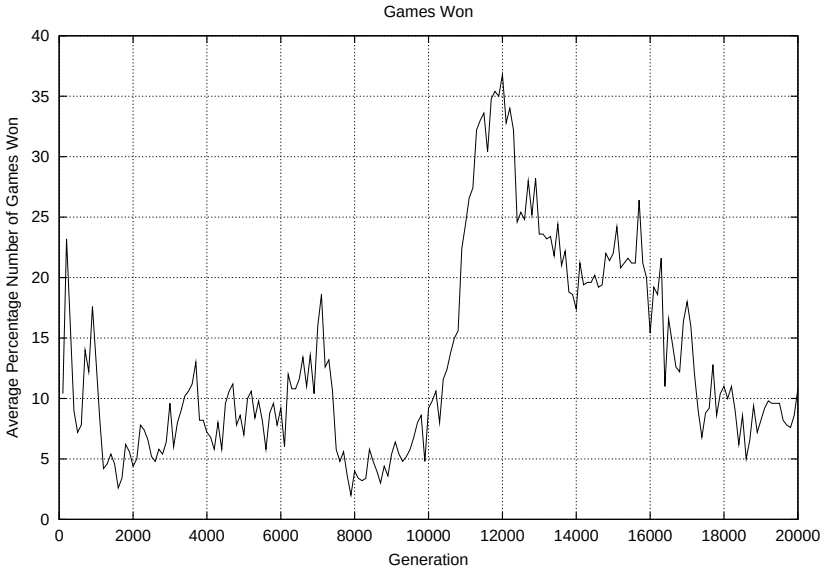



Figure 1 shows that co-evolution managed to evolve a player that performs very well against Pubeval. We see that the evolution is slow at start, with

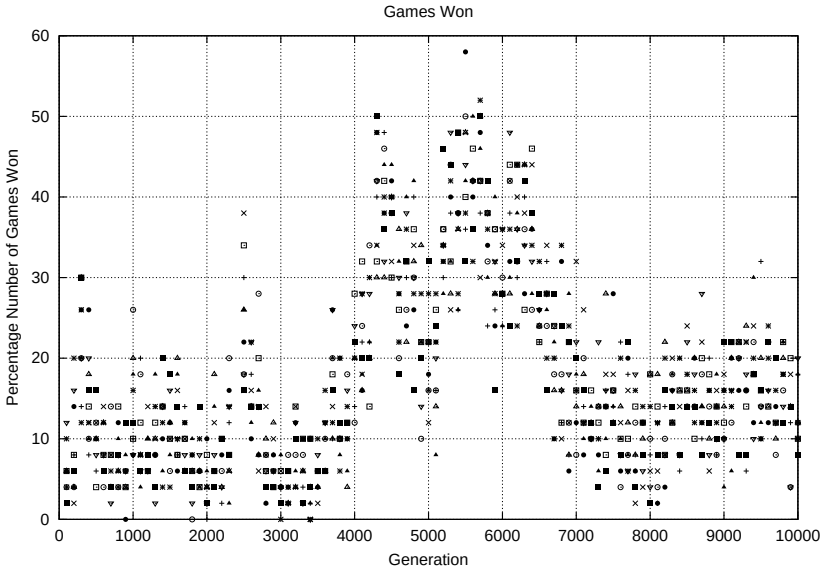
many ups and downs, but suddenly (at about generation 10000) a steep climb brings us to a high peak over 50% of the games (the good results of early generations are probably the result of the small number of games played and the "lucky beginner" rule). After this, the players devolve to a very bad grade of playing. This can be seen a lot better in the following graph:



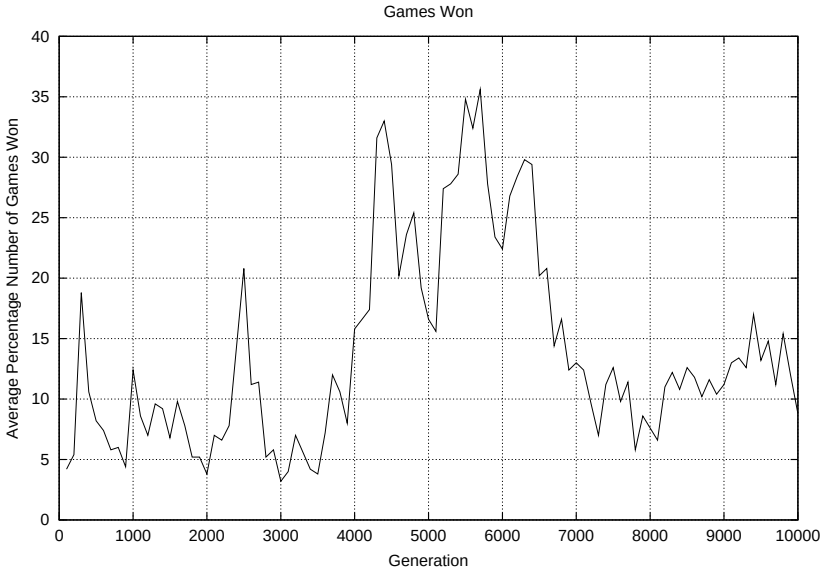

It is easy to see that the evolution is not smooth. On the contrary, it behaves almost randomly most of the time. There are two factors that may produce these results: First, the evolved population and the number of games played is very small, allowing an easy death of a good player caused by a group of lucky beginners. Second, the evolutionary operators are disruptive. Since mutation is done in small amounts and with low probability, we must conclude that the merge operator is not working properly.

After further testing of the players with more games, we have found that we have created a generation of players that averages more than 43% wins against Pubeval, with is not bad at all.

To test our hypothesis, we did a second evolution, this one were the merging factor between the players was set to 0.5, thinking that this operator would be less disruptive. The results are shown in the following graph:




From the results we can see that the change in the merging operator didn't substantially change the way it devolved the players after getting good results. As can be seen in the player's average graph, it also didn't help in smoothing the evolution. Therefore we think that the number of players or the number of games player between the players must be increased for the evolution to continue in a good way.



This players evolved more rapidly, but this is most probably by chance than a cause of the new merging operator. After further testing with more games, this evolution run created a group of players that averages 40% wins against pubeval.

Because of the huge amount of computer power needed to evolve larger amounts of players, we were unable to test these hypothesis. This would be a very interesting thing to investigate if the resources can be acquired.

# 7 Conclusions

The "No Free Lunch Theorem" [13] shows that solving a given problem requires an algorithm that is suited to the solution. But what algorithm is best suited for each solution, and in our case, for creating a backgammon move evaluator? There is currently no way to answer this question with certainty. We tried one solution to the problem. After evolving our player for a long time we created players that can match the average backgammon player, without any expert intervention. We think that an important part of artificial intelligence is self learning, and as we see by the work done, this can be achieved.

It is important to note that we achieved the current results using only a home-class computer (AMD K6-300Mhz and AMD Athlon 750Mhz), and a simulation programmed in Java, which tends to be slow. In spite of this, we achieved the results in a manner of days (it took more time to test the players than to evolve them!). One important feature of our approach is that it can be highly parallelized. In the current era of cheap desktops and the Internet, evolution can be done on computers around the world (much like the SETI@home project) using user's computers while they're idle. Even if the algorithm is not the best one suited for the purpose, it may produce better results in less time because of this property.

# References


[1] Shannon, C. E "Programming a Computer for Playing Chess". Philosophical Magazine 41, (1950), 265-275.

[2] Samuel, A. "Some Studies in Machine Learning Using the Game of Checkers". IBM Journal of Research and Development 3, (1959), 210-229

[3] Chomsky, N. "Language and Though". Wakefield, (1992)

[4] Tesauro, G. "Temporal Difference Learning and TD-Gammon". Communications of the ACM, (1995), 38(3):58-68.

[5] Pollack, J. B. and Blair A. D. "Co-Evolution in the Successful Learning of a Backgammon Strategy". (1997)

[6] Darwen, P. J. "Why Co-Evolution Beats Temporal Difference Learning at Backgammon for a Linear Architecture, but not a Non-Linear Architecture". Congress on Evolutionary Computation, 1009-1010. Seoul, Korea. (2001).

[7] Rosin, C. D. and Belew, R. K. "New Methods for Competitive Coevolution" Evolutionary Computation, 5(1):1-29. (1997)

[8] Chellapilla, K. and Fogel, D. B. "Evolution, Neural Netorks, Games and Intelligence". Proceedings of the IEEE, 87(9):1471-1496. (1999)

[9] Floreano, D., Nolfi, S. and Mondada, F. "Competitive Co-Evolutionary Robotics: From Theory to Practice". Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior. MIT Press-Bradford Books. Cambridbge, MA, USA. (1998).

[10] Husbands P. and Mill F. "Simulated Co-Evolution as the Mechanism for Emergent Planning and Scheduling". Proceedings of the Fouth International Conference on Genetic Algorithms, 264-279. Morgan Kaufmann. (1991).

[11] Yao, X. and Shi, Y. "A Preliminary Stury on Designing Artificial Neural Networks Using Co-Evolution". Proceedings of the IEEE SIngapore International Conference on Intelligent Control and Instrumentation, 149-154. IEEE Singapore Section. (1995)

[12] Zhao, Q. "Cooperative Co-Evolution of Patter Recognition Agents". (2000)

[13] Wolpert, D. H., and Macready, W. G. "No Free Lunch Theorems for Optimization". IEEE Trans. Evolutionary Computation, 1(1):67-82. (1997)

[14] Fogel D. B. "Evolutionary Computation: Toward a New Philosophy of Machine Intelligence". IEEE Press Marketing, 2nd Edition. Piscataway, NJ, USA. (2000)