M^2*ICAL* Analyses HC-Gammon

Wee-Chong Oon and **Martin Henz**
School of Computing
National University of Singapore
3 Science Drive 2
Singapore 117543

Abstract

We analyse Pollack and Blair's *HC-Gammon* backgammon program using a new technique that performs **M**onte Carlo simulations to derive a **M**arkov Chain model for **I**mperfect **C**omparison **AL**gorithms, called the *M*^{2}*ICAL* method, which models the behavior of the algorithm using a Markov chain, each of whose states represents a class of players of similar strength. The Markov chain transition matrix is populated using Monte Carlo simulations. Once generated, the matrix allows fairly accurate predictions of the expected solution quality, standard deviation and time to convergence of the algorithm. This allows us to make some observations on the validity of Pollack and Blair's conclusions, and also shows the application of the *M*^{2}*ICAL* method on a previously published work.

Introduction

One of the main difficulties in current research on algorithms that generate game-playing programs is how to evaluate the final generated player in a fair and accurate way. The cause of this difficulty is the fact that there is in general a non-zero probability that a "weaker" player defeats "stronger" player in a head-to-head match. This phenomenon has been dubbed the "Buster Douglas Effect" (Pollack & Blair 1998). Even though algorithms like competitive co-evolution have been found to converge to optimality when this phenomenon does not occur (Rosin & Belew 1996), in all practical games the Buster Douglas Effect is present. As a result, comparison-based algorithms that determine the relative strength of two players by playing them against each other are using an "imperfect" comparison function.

In this paper, we describe how **M**onte Carlo simulations can be used to derive a **M**arkov Chain model for an **I**mperfect **C**omparison **AL**gorithm, using a technique called the *M*^{2}*ICAL* method. We use this technique to model the HC-Gammon backgammon algorithm into a Markov Chain, which allows us to take advantage of existing Markov Chain theory to evaluate and analyse the algorithm.

Copyright © 2007, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Definitions and Notations

Let **P** be an optimization problem, and S the set of solutions to this problem. In general, any optimization problem **P** can be expressed in terms of a corresponding *objective function* $F : S \rightarrow \mathbb{R}$, which takes as input a solution $s \in S$ and returns a real value that gives the desirability of s. Then, the problem becomes finding a solution that maximizes F. We further define a *comparison function* $Q : S \times S \rightarrow S$ as a function that takes two solutions s_i and s_j and returns the superior one. If the comparison function does not always return the correct solution, it is called *imperfect*.

The aim of the game-playing problem is to create a program that can play a game well (so the solution space S of the problem is the set of all possible game-playing programs). Games can be represented by a directed graph $G = (V, E)$, where each vertex represents a valid position, and $E = \{(v_i, v_j)|$ there is a legal move from v_i to $v_j\}$. For simplicity, we only examine 2-player games, turn-taking, win-loss games.

Definition 1 (Player) *A* **player** *of a game $G = (V, E)$ is a function $PL : V \rightarrow E$ that takes as (one of its) input(s) a valid position $v \in V$ and returns a valid move $(v, v') \in E$.*

Our definition of a player is a function that takes as one of its inputs a legal position and returns its move. This includes non-deterministic functions as well as functions that take information other than the current game position into consideration when making a move.

One way to compare two players is to play them against each other and select the winner. Formally, this *beats* comparison function (BCF) is defined as follows:

$$BCF(PL_i, PL_j) = \begin{cases} PL_i & PL_i \text{ beats } PL_j \\ PL_j & \text{otherwise.} \end{cases} \quad \text{for all} \quad PL_i, PL_j \in S \qquad (1)$$

For turn-taking games, the first argument is the first player and the second argument is the second player. We use the shorthand notation $PL_i \succ PL_j$ to represent the case where $BCF(PL_i, PL_j) = PL_i$; and $PL_i \prec PL_j$ to represent $BCF(PL_i, PL_j) = PL_j$.

The objective of the game-playing problem is to find a player with maximum *player strength*. In this research, we make use of the following definition of player strength. The notation 1_f is the indicator function for a boolean function f, i.e. 1_f returns 1 if f is true and 0 if f is false.

Definition 2 (Player Strength) *The strength of player PL_i, denoted by $PS(PL_i)$, is*

$$PS(PL_i) = \sum_{1 \leq j \leq |S|} 1_{PL_i \succ PL_j} + \sum_{1 \leq j \leq |S|} 1_{PL_j \prec PL_i} \quad (2)$$

HC-Gammon

In computer science, the greatest success in backgammon is undoubtedly Gerald Tesauro's *TD-Gammon* program (Tesauro 1995). Using a straightforward version of Temporal Difference learning called TD(λ) on a neural network, TD-Gammon achieved Master-level play. Pollack and Blair (Pollack & Blair 1998) implemented a simple hill-climbing method of training a backgammon player using the same neural network structure employed by Tesauro (which we will call *HC-Gammon*). Although HC-Gammon did not perform as well as TD-Gammon, it produced sufficiently good results for the authors to conclude that even a relatively naive algorithm like hill-climbing exhibits significant learning behaviour in backgammon. This supported their claim that the success of TD-Gammon may not be due to the TD(λ) algorithm, but is rather a function of backgammon itself.

Experimental Setup

Pollack and Blair used the same neural network architecture with 3980 weights employed by Tesauro to represent their backgammon players. The player evaluates the resulting positions from all possible moves for the given dice roll, and chooses the move that leads to the position with the highest evaluation. The initial player had all weights set to 0.0, which we call the *all-zero neural network* (AZNN).

Pollack and Blair implemented and tested 3 different hill-climbing algorithms. In Experiment 1, the challenger is derived via a *mutation function*, where gaussian noise is added to the neural network weights of the current player. In their paper, the only description provided of this function is the phrase "*the noise was set so each step would have a 0.05 RMS distance (which is the euclidean distance divided by $\sqrt{3980}$).*" Without further clarification available, we assume that the mutation function is as follows.

Let w_i, $1 \leq i \leq 3980$ be the weights of the current player, and w'_i be the corresponding weights of the challenger derived from the current player. For each weight w_i we randomly introduce gaussian noise to the magnitude of x_i, which will be normalized with a multiplier k, i.e. $w'_i = w_i + kx_i$. The value of k is computed as follows:

$$k = 0.05 \cdot \sqrt{3980 / \sum_i (w'_i - w_i)^2} \quad (3)$$

If the new player (the "challenger") defeats the original player (the "incumbent") in 3 out of 4 games, which comprises two pairs of games using two different sequences of dice rolls (called *dice streams*), then the challenger is deemed to be victorious. When this occurs, the incumbent is replaced using the *descendent function*:

$$\text{incumbent} = 0.95 \cdot \text{incumbent} + 0.05 \cdot \text{challenger} \quad (4)$$

Experiment 2 increased the challenger's requirements from having to win 3 out of 4 games to 5 out of 6 games after 10,000 iterations, and then to 7 out of 8 games after 70,000 iterations; the values 10,000 and 70,000 were chosen after inspecting the progress of their best player from Experiment 1. The final evolved player from Experiment 2 was the strongest player created, which was able to win 40% of the time against a reasonably strong public domain backgammon program called *PUBEVAL*. Finally, Experiment 3 implemented a dynamic annealing schedule by increasing the challenger's victory requirements when over 15% of the the challengers were successful over the last 1000 iterations.

The *M*^{2}*ICAL* Method

The *M*^{2}*ICAL* method is a process that is divided into 4 phases:

1. Populate the classes of the Markov Chain.
2. Generate the *win probability matrix W*.
3. Generate the *neighbourhood distribution* λ_i for each class i.
4. Calculate the transition matrix P using W and λ.

In this section, we describe how we applied the *M*^{2}*ICAL* method to HC-Gammon to derive a Markov Chain model with $N = 10$ states. Further details can be found in (Author 2007).

Estimating Player Strength

We make use of Monte Carlo simulations to estimate the strength of a player over the space of all possible players. The target player PL_i plays a match of g games against each of $M_{opp} = 100$ randomly generated opponents $PL_{ij}, 1 \leq j \leq M_{opp}$. To divide all players into N unique sets of players of similar strength, we group them by *estimated player strength of PL_i*, denoted by $F'(PL_i)$:

$$F'(PL_i) = \left\lfloor \frac{\sum_{j=1}^{M_{opp}} (1_{PL_i \succ PL_{ij}} + 1_{PL_{ij} \prec PL_i})}{(g \cdot M_{opp}/N)} \right\rfloor + 1 \quad (5)$$

We also define $F(i)$ be the quality measure of state i. Therefore, the state space $I = \{i | \exists_{PL \in S}, F'(PL) = F(i)\}$.

Populating the Classes

In the first phase, the task is to populate the classes of the Markov Chain with a representative subset of the algorithm's search space by making use of its neighbourhood function. However, HC-Gammon uses a fixed initial player (the AZNN player). If the model was derived using players in the neighbourhood of the AZNN player, then the results will reflect the properties of this neighbourhood (rather than the search space of the algorithm in the long term). To address this issue, we perform $M_{sample} = 200$ runs of the HC-Gammon algorithm using the AZNN player as the initial player, advancing each run one iteration at a time in parallel until at least 50% of the runs have experienced at least 10 replacements, i.e. the challenger has defeated the incumbent at least 10 times. In our experiment, this event occurred after 47 iterations. At this point, we use the current players of the 200 runs as the initial sample. We call this process of running the algorithm until a sufficient number of replacements has occurred *introducing a time-lag*.

We set a *maximum class size* value of $\hat{\gamma}$, so that we only retain a maximum of $\hat{\gamma}$ players per class. For each of the M_{sample} players, we evaluate its strength by playing it against M_{opp} randomly generated opponents. We randomly retain up to $\hat{\gamma} = 20$ players from each class generated this way and discard the rest. Then, for each player PL in an unchecked class i with maximal size, we generate a player PL' using the mutation function, and then a descendent player PL'' from PL and PL' using the descendent function. If PL'' belongs to a class with fewer than $\hat{\gamma}$ players, then it is retained; otherwise it is retained with a probability of $\frac{\hat{\gamma}}{\hat{\gamma}+1}$. We repeat this process until $M_{pop} = 100$ new players have been generated. If at least one of the M_{pop} players produced belongs to a class that initially had fewer than $\hat{\gamma}$ players, then we generate a further M_{pop} players from the same class, and repeat this process until no such players are produced out of the set of M_{pop} players.

Comparison Function Generalization

Knowing the probability that a player PL beats another player PL' as both first and second player, we can compute the probability that PL beats PL' in at least x out of y games (where y_1 games are as first player and y_2 are as second, $y = y_1 + y_2$). Hence, we wish to compute an $N \times N$ *win probability matrix (WPM)* W, such that its elements w_{ij} provides the probability that a player from class i beats a player from class j playing first.

For all pairs of classes i and j we randomly select a player PL from class i and a player PL' from class j and play a game between them with PL as first player and PL' as second, noting the result. We repeat this $M_{wpm} = 200$ times for each pair of classes i and j, and then compute the value of w_{ij} as $1_{s \succ s'}/M_{wpm}$.

The WPM W gives us the probabilities for winning as first player. Let $\bar{W}$ be the corresponding win probability matrix that provides the winning probabilities as second player. For a win-loss game, $\bar{w}_{ij} = 1 - w_{ji}$. We define a shorthand notation $W_{ij}^{\geq x(y_1/y_2)}$ to denote the probability that a player PL from class i would beat a player PL' from class j at least x times in a match where PL plays as first player y_1 times and as second player y_2 times. For example,

$$W_{ij}^{\geq 3(2/2)} = ((1 - \bar{w}_{ij}) \cdot \bar{w}_{ij} \cdot w_{ij}^2) + (\bar{w}_{ij}^2 \cdot w_{ij} \cdot (1 - w_{ij})) + (\bar{w}_{ij}^2 \cdot w_{ij}^2) \qquad (6)$$

The probabilities of other results based on multiple games can be computed in a similar manner. In this way, we avoid having to recompute our probability distributions for different comparison functions.

Neighbourhood Distribution

For each class i, we first generate the *challenger probability distribution* C, which gives the probability in $c_i(j)$ that an incumbent player PL in class i will create a challenger PL' in class j using HC-Gammon's mutation function. This is done using a Monte Carlo simulation by creating $M_{cha} = 200$ challengers this way, evaluating each of them, and then estimating the probability distribution using this sample. We store all of the generated challengers in vectors $\vec{c}_1, \vec{c}_2 \cdots \vec{c}_N$, such that if $eval(PL') = STR$ then PL' will be stored in $\vec{c}_{STR}$, including a pointer from PL' to its parent PL.

Next, for every non-empty vector $\vec{c}_j$, we find the *descendent probability distribution* D_{ij} that gives the probability in $d_{ij}(k)$ that a descendent created from a crossover between a player from class i and class j will be of strength k. To do so, we randomly select a parent-challenger pair PL from class i and PL' from class j, create a descendent from the crossover of PL and PL' and evaluate it. This is repeated $M_{des} = 200$ times to provide the probability distribution. Note that for HC-Gammon, the descendent probability distribution D_{ij} is essentially its neighbourhood distribution.

Transition Matrix

Having estimated the values for the win probability matrix W, the challenger probability distribution C_i and the descendent probability distribution D_{ij}, we can now formulate the transition matrix P of the Markov Chain representing HC-Gammon for each of the 3 experiments.

For Experiment 1, the transition matrix P is given by:

$$p_{ik} = \sum_j c_i(j) \cdot W_{ji}^{\geq 3(2/2)} \cdot d_{ij}(k) \qquad (7)$$

where $W_{ji}^{\geq 3(2/2)}$ is computed using Equation (6).

For Experiment 2, we substitute $W_{ji}^{\geq 5(3/3)}$ into Equation (7) after 10,000 iterations, and $W_{ji}^{\geq 7(4/4)}$ after 70,000 iterations. We call these resultant transition matrices P' and P'' respectively.

The model for Experiment 3 also makes use of P, P' and P''. Let κ be the number of challengers that must replace the incumbent in the last Λ iterations before a change in comparison function is made; in this case, $\kappa = 150$ and $\Lambda = 1000$. Let $\alpha_{(t)}$, $\alpha'_{(t)}$ and $\alpha''_{(t)}$ be the probability that the algorithm is employing at iteration t the comparison function represented by P, P' and P'' respectively.

Let $a_{(t)}$ be the probability that the challenger wins in iteration t. To calculate this value, we require the probabilities that the incumbent player is in each class i, given by the probability distribution vector for the previous iteration $\vec{v}_{(t-1)}[i]$. Then, given the probability $c_i(j)$ that the incumbent produces a challenger from class j, we can find the probability that the challenger wins using the appropriate winning probability W_{ji} by summing these values over all combinations of i and j.

$$a_{(t)} = \sum_{i=1}^{N} \sum_{j=1}^{N} \vec{v}_{(t-1)}[i] \cdot c_i(j) \cdot (\alpha_{(t)} \cdot W_{ji}^{\geq 3(2/2)} + \alpha'_{(t)} \cdot W_{ji}^{\geq 5(3/3)} + \alpha''_{(t)} \cdot W_{ji}^{\geq 7(4/4)}) \qquad (8)$$

Let $b_{(t)}^{k/l}$ be the probability that exactly k challengers were victorious between iterations $t - l + 1$ and t inclusive. We can assume without loss of generality that $l \leq t$. Observe that $b_{(t)}^{0/1} = 1 - a_{(t)}$ and $b_{(t)}^{1/1} = a_{(t)}$. For $t - l + 1 < s \leq t$, we find that:

$$b_{(s)}^{k/l} = b_{(s-1)}^{k-1/l-1} \cdot (1 - a_{(s)}) + b_{(s-1)}^{k/l-1} \cdot a_{(s)} \quad (9)$$

In this way, we can recursively express $b_{(t)}^{k/l}$ in terms of b values for iteration $(t-1)$ and $a_{(t)}$.

Let $\beta_{(t)}^{k/l}$ be the probability that *at least* k challengers were victorious in the last l iterations at iteration t. We can easily compute this value using b values as follows:

$$\beta_{(t)}^{k/l} = \sum_{i=k}^{l} b_{(t)}^{i/l} = 1 - \sum_{i=0}^{k-1} b_{(t)}^{i/l} \quad (10)$$

We can now compute the values of $\alpha_{(t)}$, $\alpha'_{(t)}$ and $\alpha''_{(t)}$:

$$\alpha_{(t)} = \begin{cases} 1 & 0 < t < l \\ \alpha_{(t-1)} - \alpha_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} & t \geq l \end{cases} \quad (11)$$

$$\alpha'_{(t)} = \begin{cases} 1 - \alpha_{(t)} & 0 < t < l \\ \alpha'_{(t-1)} + \alpha_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} & \\ \quad -\alpha'_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} & t \geq l \end{cases} \quad (12)$$

$$\alpha''_{(t)} = 1 - \alpha_{(t)} - \alpha'_{(t))} \qquad t \geq 2l \quad (13)$$

$\alpha_{(t)} = \alpha'_{(t)} = \alpha''_{(t)} = 0$ otherwise. Obviously, this formulation can be generalized to the cases when the number of possible annealing steps is greater than three. Once the α values are computed for a particular iteration t, then the transition matrix for that iteration $P_{(t)}$ is simply:

$$P_{(t)} = (\alpha_{(t)} \cdot P) + (\alpha'_{(t)} \cdot P') + (\alpha''_{(t)} \cdot P'') \quad (14)$$

Usefulness of Model

To find the **expected player strength** of the current player after t iterations, we begin with a probability vector $\vec{v}_{(0)}$ of size N, $\vec{v}_{(0)} = \{v_1, v_2, \cdots, v_N\}$ that contains in each element v_i the probability that the initial player will belong to class i. The values of $\vec{v}_{(0)}$ depends on how the algorithm chooses its initial state.

Let $\vec{v}_{(t)}$ be the corresponding estimated player strength probability vector of the algorithm after t iterations. Given the transition matrix P of our Markov Chain, we can compute $\vec{v}_{(t)}$ by performing a matrix multiplication of $\vec{v}_{(0)}^T$ and P t times, i.e. $\vec{v}_{(t)}^T = \vec{v}_{(0)}^T \cdot P^{(t)}$. The estimated strength of the player produced by the algorithm after t iterations, denoted by $PL^{(t)}$ is then given by

$$E(PS(PL^{(t)})) = \sum_{i=1}^{N} \vec{v}_{(t)}[i] \cdot F(i) \quad (15)$$

The computation of the expected player strength requires $t \cdot N^2$ floating point multiplications, which takes very little actual computation time. In general, once the the Markov Chain has been determined, the computation of the expected solution quality using this method will be much faster than running the target algorithm itself, and then using Monte Carlo simulations to determine the estimated solution quality after every iteration. This is one of the main advantages of using the *M*^{2}*ICAL* method to analyze imperfect comparison algorithms.

While Markov Chain theory has several concise definitions on the convergence of a system, including concepts of ϕ-irreducibility, Harris recurrence and geometric ergodicity of Markov Chains (Meyn & Tweedie 1993), the practitioner is often less more concerned with the practical performance of the algorithm. Our notion of the **time to convergence** of an algorithm is admittedly not theoretically concise, but we believe that it is useful to the practitioner. Essentially, we detect the number of iterations required before all the elements in $\vec{v}$ are identical up to k decimal places in two successive iterations; this is done concurrently with the computation of the expected player strength given above. The number of iterations required for this to occur is the expected number of iterations for the algorithm to converge to the stationary values to a degree of accuracy of k decimal places.

It is also useful to know the spread of the solutions generated by the algorithm. This is measured by the **standard deviation** of the solutions, and can be calculated from the probability vector $\vec{v}$ after any number of t iterations. We first find the variance σ^2 of the vector:

$$\sigma^2 = \sum_{i=1}^{N} (\vec{v}[i] - \mu)^2 \cdot F(i) \quad (16)$$

where $\mu = \sum_{i=1}^{N} \vec{v}[i] \cdot F(i)$. We can then find a range of expected values given by $[\mu - \sigma, \mu + \sigma]$, where σ is the standard deviation. Assuming that the set of solutions generated by the algorithm can be approximated by a normal distribution, then about 68% of all solutions found by the algorithm will have a strength within this range (and about 95% will be within $[\mu - 2\sigma, \mu + 2\sigma]$).

The standard deviation helps the practitioner decide if rerunning the algorithm is worthwhile. For example, assume that the quality of the solution generated by one run of the algorithm is close to the predicted expected quality. If the standard deviation is small, then it is less likely that rerunning the algorithm will produce a superior result; conversely, if the standard deviation is large, then it may be worthwhile to re-run the algorithm in the hopes of generating a superior solution (although the probability of generating an inferior solution could be just as high).

Results and Analyses

Exp 1: Inheritance

Figure 1 shows the predictions given by the time-lag *M*^{2}*ICAL* model compared to 25 runs of HC-Gammon; the values begin at iteration 48. The model predicts that the expected player strength of HC-Gammon will rise steadily from 67.80% at iteration 48 before converging to a value of 86.99%, to an accuracy of 5 decimal places, after approximately 1050 iterations. This is reasonably close to the results obtained from the average of 25 runs of HC-Gammon,

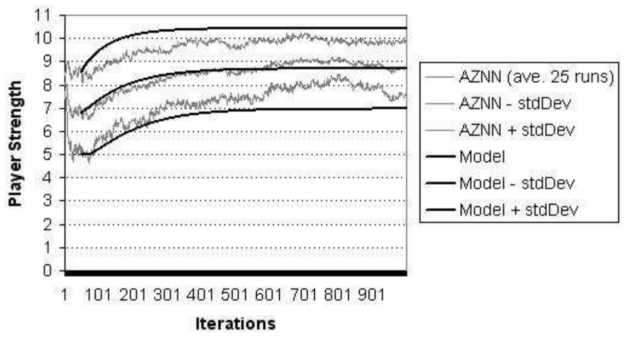

Figure 1: Time-Lag Model and experimental results for HC-Gammon using the AZNN initial player

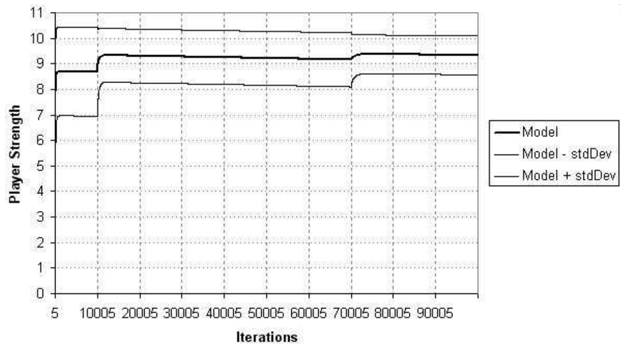

Figure 2: Time-Lag Model for fixed annealing schedule HC-Gammon using the AZNN initial player

which fluctuates between 85.5% and 90.5%. However, the model predicts that the standard deviation of the player strength will be about 1.8 classes, overestimating the standard deviation of the values obtained from the 25 runs of HC-Gammon, which fluctuate between 0.95 and 1.25 classes. Nonetheless, the values obtained from the actual runs fall well within the range of values predicted by the model.

Since the highest (10th) class falls within one standard deviation of the expected player strength, we can expect that about 13.6% of all players produced using this algorithm will be in the top 10% of all possible players if the strength of the players is normally distributed. Note however that since our model contains only 10 classes, we can only predict the expected strength of the generated players to within a 10% range, so this experiment does not show that the algorithm will be able to generate players that can beat very strong players like *PUBEVAL* or TD-Gammon, who are probably in the top 1% or better of all possible players.

Exp 2: Fixed Annealing Schedule

When the annealing schedule is at 10,000 and 70,000 iterations, the M^2*ICAL* model prediction of its expected player strength is given in Figure 2. As expected, the results show distinct "steps" in the expected player strengths after the annealing points.

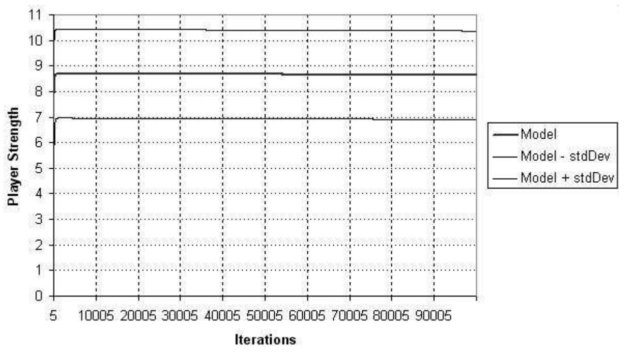

Figure 3: Time-Lag Model for dynamic annealing schedule HC-Gammon using the AZNN initial player

The most startling aspect of this model is the fact that after both the first and second annealing point, the expected player strength reaches a local maximum value and then starts to decline. This occured at about iteration 12,170 when the 5-out-of-6 comparison function was used (at a player strength of 93.30%); the algorithm reached its highest expected playing strength of approximately 94.76% of all possible players after about 74,900 iterations when 7-out-of-8 comparison function was in effect. Beyond this point, the expected player strength decreases, until it reached a value of 93.29% at iteration 100,000. Hence, this model shows the possibility that not only is running an algorithm for extremely large numbers of iterations not significantly beneficial to the algorithm's performance, it could be detrimental.

Exp 3: Dynamic Annealing Schedule

Even though the time-lag model begins at iteration 48, for simplicity we assume that the sample players that we obtained at this point were from iteration 0. Surprisingly, we find that the probability of 15% of the last 1000 challengers defeating the incumbent 3-out-of-4, α', is close to zero throughout the algorithm (needless to say, α'' is even smaller). These results are given in Figure 3, which is in effect almost exactly the same as the results for Experiment 1 (Figure 1), extended to 100,000 iterations.

Limited experiments using actual runs of the algorithm bear out these findings: none of the actual runs using the AZNN as the initial player ever managed to achieve the 15% challenger success rate to elicit an increase in the comparison function requirements. Once again, our experiments contradict the results reported by Pollack and Blair, who explicitly stated that the rate of challenger success increased as the number of iterations of their algorithm increased. Furthermore, none of the players generated using any of the configurations detailed in this chapeter were able to defeat *PUBEVAL* over 15% of the time. We are currently unable to definitively explain the discrepancies in our results, although there are two areas where our emulation of the HC-Gammon algorithm is most likely to be inaccurate. The first is in our interpretation of their mutation function, which represents our best guess given the description provided by the authors; the second is the move ordering function for our

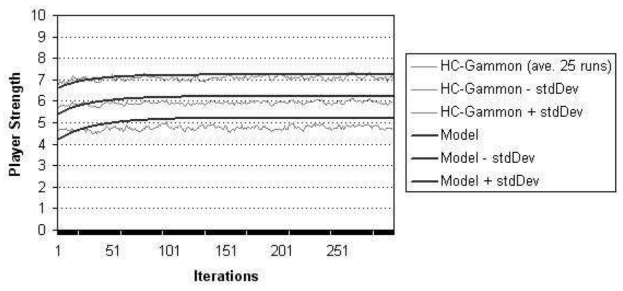

Figure 4: Model and experimental results for HC-Gammon with random initial player

backgammon implementation, which was not mentioned by the authors at all.

Random Initial Player

Evolutionary machine-learning algorithms usually begin with a randomly generated initial player, so Pollack and Blair's decision to begin their experiments with the AZNN initial player seemed somewhat counter-intuitive. It is interesting to note that according to our Monte Carlo evaluations, the AZNN player belongs to the 80th percentile of all players. In order to gauge the effect of starting with a superior player, we implemented the M^2ICAL method on the algorithm that starts with a neural network with weights uniformly randomly selected from the range of [-0.2, 0.2].

Figure 4 shows the the results for Experiment 1 using a randomly determined initial player (only the values for the first 300 iterations are shown here; the remaining 700 iterations follow the same trend). In this case, the expected player strength for the model converges to an accuracy of 5 decimal places after 288 iterations, to a value of about 62.25%, compared to the average of the 25 actual runs, which fluctuates between 59% and 61%. Furthermore, the standard deviation given by the model after a large number of iterations is around 10.05%, which is close to the sample standard deviation of the 25 runs of between 10.2% and 12.5%.

Note that the predicted value of 62.25% means that the generated player is not much better than average. The drastic difference between the strengths of the players generated using a randomly generated player rather than the AZNN as the initial player was almost 3 classes (or 30% of all possible players), which reveals that although Pollack and Blair managed to produce a strong player using a simple hill-climbing algorithm, the hill-climbing approach *in general* is not fully responsible for the success of the algorithm; the initial starting player is crucial. In particular, this observation casts doubt on Pollack and Blair's hypothesis that certain qualities of backgammon *"operates against the formation of mediocre stable states"* (Pollack & Blair 1998), where the algorithm is trapped in a local optimal. If their hypothesis is correct, then the identity of the initial player should have no long-term effect on the quality of produced players. Our experiments showed that this is not the case.

Conclusions

In this paper, we have shown how the HC-Gammon algorithm can be modelled as a Markov Chain by using the M^2ICAL method. Even though we were unable to duplicate the reported results, the model was able to predict the performance of our experimental setups reasonably well despite having only 10 classes in the Markov Chain.

Most of Pollack and Blair's conclusions on the favourable characteristics of backgammon to self-learning and its effect on the evaluation of TD-Gammon's temporal difference learning approach is based on how they managed to produce a player that could defeat *PUBEVAL* 40% of the time using a simple hill-climbing algorithm. Even though we were unable to reproduce their result, our experiments do cast doubt on some of their conclusions and suppositions. In particular, the dramatic effect on the generated player's strength depending on whether the initial player was the AZNN or a randomly generated ANN showed that getting trapped in local optima is still an issue for hill-climbing algorithms on backgammon.

We do not claim that the M^2ICAL method is able to produce Markov Chain models that reflect the performance of algorithms with anywhere close to 100% accuracy; this is impossible for practical problems due to the inherent inaccuracies involved in doing Monte Carlo simulations. However, we hope that by implementing the model on an actual, published algorithm, we have shown the possible benefits of having a technique that can evaluate algorithm performance in objective terms.

References

Author, H. 2007. M^2ICAL*: A Technique for Analyzing Imperfect Comparison Algorithms using Markov Chains*. Ph.D. Dissertation, National University of Singapore, Singapore. Manuscript.

Meyn, S. P., and Tweedie, R. L. 1993. *Markov Chains and Stochastic Stability*. London: Springer.

Pollack, J. B., and Blair, A. D. 1998. Coevolution in the successful learning of backgammon strategy. *Machine Learning* 32:225–9240.

Rosin, C. D., and Belew, R. K. 1996. A competitive approach to game learning. In *Proceedings of the 9th Annual ACM Conference on Computational Learning Theory (COLT-96)*, 292–302.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–68.