M^2*ICAL*: A Tool for Analyzing Imperfect Comparison Algorithms

Wee-Chong Oon and Martin Henz
National University of Singapore
School of Computing
Computing 1, Law Link, Singapore 117590, Singapore
{oonwc,henz}@comp.nus.edu.sg

Abstract

Practical optimization problems often have objective functions that cannot be easily calculated. As a result, comparison-based algorithms that solve such problems use comparison functions that are imperfect (i.e. they may make errors). Machine learning algorithms that search for game-playing programs are typically imperfect comparison algorithms. This paper presents M^{2}ICAL*, an algorithm analysis tool that uses* **M***onte Carlo simulations to derive a* **M***arkov Chain model for* **I***mperfect* **C***omparison* **AL***gorithms. Once an algorithm designer has modeled an algorithm using* M^{2}ICAL *as a Markov chain, it can be analyzed using existing Markov chain theory. Information that can be extracted from the Markov chain include the estimated solution quality after a given number of iterations; the standard deviation of the solutions' quality; and the time to convergence.*

1. Introduction

Many algorithms that solve optimization problems are comparison-based, i.e. they have as their primary operation the comparison of two (or more) solutions in order to determine their relative superiority. However, real-world optimization problems are often not well-defined in the sense that the quality of a solution may not be satisfactorily expressed in terms of an easily calculable equation. Therefore, comparison-based algorithms must often employ a comparison function that is not 100% accurate. We call algorithms that rely on such imperfect comparison functions *imperfect comparison algorithms*.

The game-playing problem is an archetypal imperfect comparison problem. The aim of this problem is to create a program that can play an intellectual game such as chess or checkers well. Recently, intellectual games have been used as a test bed for machine learning techniques. Notable successes in this field include the backgammon program *TD-Gammon* [7] that was based on temporal difference learning, and the checkers program *Anaconda* [1, 2] based on co-evolution of neural networks.

This research proposes a tool for the analysis of imperfect comparison algorithms domain. The tool is based on the idea of modelling the algorithms as a discrete Markov chain with the help of Monte Carlo simulations, and then discovering important attrributes such as the expected solution quality; solution spread; and rate of convergence of the algorithm using numerical analysis. We call the tool **M**onte Carlo **M**arkov Chain for **I**mperfect **C**omparison **AL**gorithms, or *M*^{2}*ICAL*[1] for short; the models produced using the tool are similarly called *M*^{2}*ICAL* models. As far as we know, there have been no previous tools to analyze the performance of complex imperfect comparison algorithms in practical settings.

The application of the *M*^{2}*ICAL* tool to HC-Gammon, an imperfect comparison algorithm for generating backgammon players, is the subject of recent work in [5]. While that work concentrates on analyzing a particular known algorithm, the present paper focusses on the *M*^{2}*ICAL* tool itself (Section 3), and its usefulness (Section 4). Section 5 illustrates the tool by applying it to an imperfect-comparison algorithm for generating players of a simple game called Modulo Nim. The use of *M*^{2}*ICAL* for analyzing HC-Gammon is briefly reviewed in Section 6.

2. 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* $F : S \to \mathbb{R}$, 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* $Q : S \times S \to S$ as a function that takes two solutions s_i and s_j and returns

[1] pronounced *Michael*.

the superior one. If the comparison function does not always return the correct solution according to F, 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, 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*

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

3. The *M²ICAL* Tool

The *M²ICAL* tool proscribes a 4-phase analysis process for imperfect comparison algorithms:

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 paper, we explain how *M²ICAL* can be used to derive a Markov chain model of an algorithm that searches for strong game-playing programs. However, it should be reasonably simple to adapt the approach to other imperfect comparison problems.

3.1. Estimating Player Strength

Monte Carlo simulations are used to estimate the strength of a player over the space of all possible players. Let N denote the number of states in the Markov chain. For a target player PL_i, we randomly generate M_{opp} opponents $PL_{ij}, 1 \leq j \leq M_{opp}$. Player PL_i then plays a match of g games against each of these opponents. 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 $F'(PL_i)$:

$$F'(PL_i) = \left\lfloor \frac{\sum_{j=1}^{M_{opp}} (1_{PL_i \succ PL_{ij}} + 1_{PL_{ij} \prec PL_i})}{(g \cdot M_{opp}/N)} \right\rfloor + 1 \qquad (3)$$

Let $F(i)$ be the quality measure of state i. Therefore, the state space $I = \{i | \exists_{PL \in S}, F'(PL) = F(i)\}$. If the random generation of opponents is assumed to take $O(1)$ time, then the evaluation of each player takes $O(M_{opp})$ time. This process of evaluating players takes up the bulk of the computation time for *M²ICAL*.

3.2. Populating the Classes

In the first phase, the task is to populate the classes of the Markov chain (which represent different strength levels) with as many players as possible, with the given time and space constraints. The aim of this phase is to find a representative subset of the sample space that the algorithm will be searching, which will form the initial basis for the remaining steps in our technique.

Many algorithms that attempt to find strong game-playing programs begin with a randomly-generated player. From this initial player, other players are generated in some manner, e.g. by using a mutation function that changes a given player's values slightly. For the rest of this paper, we will refer to such functions by the generic term of *neighbourhood function*.

In order to get a representative subset of the algorithm's neighbourhood, we populate the classes in two separate

steps. In the first step, we randomly generate a number of players to provide a starting population for the model; this simulates the running of the algorithm several times using a randomly chosen initial player. In the second step, we make use of the algorithm's neighbourhood function for each of the classes in turn to generate more players in an attempt to fill up the remaining classes; this generates players that will be produced over the course of the target algorithm for inclusion into the sample population.

We define the *size* of a state i as follows:

Definition 3 (State Size) *Let $\bar{S}$ be a sample population of players, $\bar{S} \subseteq S$. The* **size of** *i, γ_i is the number of players $PL \in S$ where $F'(PL) = F(i)$. The* **cumulative size at** *i, Γ_i is the value of the cumulative distribution function of γ at i, i.e.*

$$\Gamma_i = \sum_{j=1}^{i} \gamma_j \quad (4)$$

By convention, we define $\gamma_j = 0$ when $j \leq 0$ or $j > N$. Note that $\gamma_j = \Gamma_j - \Gamma_{j-1}$. The total number of distinct players in the problem is Γ_N.

We set a *maximum class size* value of $\hat{\gamma}$, so that we only retain a maximum of $\hat{\gamma}$ players per class. We begin by generating M_{sample} players using the method employed by the target algorithm to select the initial player. For each player, we evaluate its strength by playing it against M_{opp} uniformly randomly generated opponents. We randomly retain up to $\hat{\gamma}$ players from each class generated this way and discard the rest.

After the initial M_{sample} players have been generated, we consider each class in turn. For each player PL in an unchecked class i with maximal size γ_i, we generate another player PL' using the algorithm's neighbourhood function and evaluate its strength. If PL' belongs to a class with fewer than $\hat{\gamma}$ players, then it is retained; otherwise it is retained with a probability of $\frac{\hat{\gamma}}{\hat{\gamma}+1}$, replacing a random existing player in that class (i.e. all players from the same class have an equal probability of being retained). We repeat this process until M_{pop} 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. The pseudocode for this phase is given in Algorithm 1.

In the worst case, the initial M_{sample} players all belong to the same class, and then the subsequent M_{pop} players generated using the neighbourhood function always generates only one new player in every instance. The algorithm would then take $O((M_{sample}+(N-1)\hat{\gamma}M_{pop})M_{opp})$ time. Assuming that $M_{sample} = M_{pop} = M_{opp} = \hat{\gamma} = O(N)$, then this process takes $O(N^4)$ in this very unlikely worst-case scenario. The storage of the generated players requires $O(N \cdot \hat{\gamma})$ space.

Algorithm 1 Populating the Classes

```
Initialize s⃗[1..N] = NULL
for i = 1 to M_sample do
  Uniformly randomly generate player PL
  STR = eval(PL)
  if size(s⃗[STR]) < γ̂ then
    s⃗[STR] ← PL
  else
    Randomly replace player in s⃗[STR] with PL with
    probability γ̂/(γ̂+1)
  end if
end for
for i = 1 to N do
  Choose an unmarked s⃗[j] s.t. WPM s⃗[j] ≥ WPM s⃗[k]
  for all unmarked s⃗[k]
  COUNT = 0; FOUND = false
  while COUNT < M_pop do
    Randomly select player PL from s⃗[j]
    Generate neighbourhood player PL' from PL
    STR = eval(PL')
    if WPM s⃗[STR]) < γ̂ then
      s⃗[STR] ← PL'; FOUND = true
    else
      Randomly replace player in s⃗[STR] with PL'
      with probability γ̂/(γ̂+1)
    end if
    COUNT++
    if COUNT == M_pop && FOUND == true then
      COUNT = 0; FOUND = false
    end if
  end while
  Mark s⃗[j]
end for
```

3.3. Comparison Function Generalization

When comparing the relative strengths of two players, the comparison function Q employed by the target algorithm usually involves playing them against each other in a match consisting of one or more games. Note that as long as we know the probability that a player PL beats another player PL' as first player and also as 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.

Algorithm 2 Computing the Win Probability Matrix W

for $i = 1$ **to** N **do**
  for $j = 1$ **to** N **do**
    $WINS = 0$
    for $k = 1$ **to** M_{wpm} **do**
      Randomly select a player PL from class i
      Randomly select a player PL' from class j
      $WINS += 1_{PL \succ PL'}$
    end for
    $w_{ij} = WINS/M_{wpm}$
  end for
end for

Algorithm 3 Finding the Neighbourhood Distribution λ_i

for $i = 1$ **to** N **do**
  Initialize $\vec{\lambda_i}[1..N]$ = 0.0
  for $j = 1$ **to** M_{nei} **do**
    Randomly select player PL from $\vec{s}[i]$
    Generate neighbourhood player PL' from PL
    $STR = eval(PL')$
    $\vec{\lambda_i}[STR]$++
  end for
  $\vec{\lambda_i}$[1..N] = $\vec{\lambda_i}$[1..N]/M_{cha}
end for

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} 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 pseudocode is given in Algorithm 2.

For each pair of classes, M_{wpm} games are played. Assuming that $M_{wpm} = O(N)$, then this algorithm takes $O(N^3)$ time.

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,

$$\begin{aligned} W_{ij}^{\geq 3(2/2)} &= ((1 - \bar{w}_{ij}) \cdot \bar{w}_{ij} \cdot w_{ij}^2) + \\ &\quad (\bar{w}_{ij}^2 \cdot w_{ij} \cdot (1 - w_{ij})) + \\ &\quad (\bar{w}_{ij}^2 \cdot w_{ij}^2) \end{aligned} \quad (5)$$

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.

3.4. Neighbourhood Distribution

Algorithms that attempt to produce a strong game-playing program search the domain of all possible players starting from the initial player or population of players. The set of players that the algorithm can potentially search is called the algorithm's *neighbourhood*. In this phase, we once again use Monte Carlo simulations to estimate the distribution of player strengths in the neighbourhood of the algorithm. To do so, we apply the neighbourhood function employed by the algorithm M_{nei} times for each class in our representative population of players, and then evaluate the strengths of the resultant players.

For each class, M_{nei} neighbourhood players are generated and evaluated. If the generation of neighbourhood players is assumed to take O(1) time, then up to a total of $O(M_{nei}M_{opp})$ operations are performed per class. Assuming that $M_{nei} = M_{opp} = O(N)$, then this part of the process runs in $O(N^3)$ time.

3.5. Transition Matrix

In the final phase, we combine the win probability matrix W with the neighbourhood distribution functions λ_i for each state i to find the transition matrix for the Markov chain model of this system.

The transition matrix for several algorithms follow a discernable structure. For example, consider *strict hill-climbing algorithms*. Strict hill-climbing algorithms do not change their current state i if the next state j is not superior. However, since the relative quality of two solutions is determined by the imperfect comparison function Q, there is a chance of an error denoted by δ_{ij}. Let λ_{ij} be the probability that state j is chosen as the potential next state when the current state is i. Then the transition matrix for strict hill-climbing algorithms can be expressed as:

$$\begin{pmatrix} p_{11} & \cdots & \lambda_{1j}(1-\delta_{1j}) & \lambda_{1N}(1-\delta_{1N}) \\ \vdots & p_{ii} & \lambda_{ij}(1-\delta_{ij}) & \lambda_{iN}(1-\delta_{iN}) \\ \lambda_{i1}\delta_{i1} & \lambda_{ij}\delta_{ij} & p_{jj} & \vdots \\ \lambda_{N1}\delta_{N1} & \lambda_{Nj}\delta_{Nj} & \cdots & p_{NN} \end{pmatrix} \quad (6)$$

where $p_{kk} = 1 - \sum_{j=1}^{k-1}(\lambda_{kj}\delta_{kj}) - \sum_{j=k+1}^{N}(\lambda_{kj}(1-\delta_{kj}))$.

Once the transition matrix for the Markov chain is determined, we can use existing Markov chain theory to discover several important properties of the algorithm in question.

Algorithm 4 Expected player strength after t iterations

```
Initialize v[1..N]; EXPPS = 0
for i = 1 to t do
    v[1..N] = v^T P
end for
for i = 1 to N do
    EXPPS += v[i].F(i)
end for
return EXPPS
```

Algorithm 5 Convergence to Stationary

```
Initialize v[1..N]; COUNT = 0; DIFF = 1
while DIFF != 0 do
    v'[1..N] = v^T P
    for i = 1 to N do
        DIFF = abs(v[i].10^k) − abs(v'[i].10^k)
        if DIFF != 0 then
            break
        end if
    end for
    if DIFF == 0 then
        return COUNT
    end if
    v[1..N] = v'[1..N]; COUNT++
end while
```

4. Usefulness of Model

4.1. Expected Player Strength

The first property to discover about an algorithm is its expected solution quality after t iterations for a given value of t. For the game-playing problem, this is equivalent to the *expected player strength* of the current player after t iterations. We begin with a probability vector $v_{(0)}^{\rightarrow}$ of size N, $v_{(0)}^{\rightarrow} = \{v_1, v_2, \cdots, v_N\}$ that contains in each element v_i the probability that the initial player will belong to class i, i.e. the probability that it will be of estimated strength $F(i)$. The values of $v_{(0)}^{\rightarrow}$ depends on how the algorithm chooses its initial state, and can usually be easily determined.

Let $v_{(t)}^{\rightarrow}$ 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 $v_{(t)}^{\rightarrow}$ by performing a matrix multiplication of ${v_{(0)}^{\rightarrow}}^T$ and P t times, i.e. ${v_{(t)}^{\rightarrow}}^T = {v_{(0)}^{\rightarrow}}^T \cdot P^{(t)}$. The estimated strength of the player produced by the algorithm after t iterations, denoted by $PL^{(t)}$ is then given by

$$E(PS(PL^{(t)})) \quad = \quad \sum_{i=1}^{N} v_{(t)}^{\rightarrow}[i] \cdot F(i) \qquad (7)$$

Algorithm 4 shows this process in pseudocode form. The computation requires $t \cdot N^2$ floating point multiplications, which takes very little actual computation time. In general, once the transition matrix for 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 *M²ICAL* to analyze imperfect comparison algorithms.

4.2. Time to Convergence

Another property that would be useful to discover is the expected number of iterations required for the given imperfect comparison algorithm to converge to the values given in the stationary vector to a specified degree of accuracy. We could then terminate the algorithm once this number of iterations has been reached because further iterations will not improve the expected solution quality significantly.

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 [3], the practitioner is often 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; 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. We first instantiate a probability vector $\vec{v}$ of size N, $\vec{v} = \{v_1, v_2, \cdots, v_N\}$. We then perform successive vector multiplications of $\vec{v}^T P$, terminating when all the values of $\vec{v}$ in two successive iterations are equal to a degree of accuracy of k decimal places, as shown in Algorithm 5.

4.3. Variance and Standard Deviation

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:

$$\sigma^2 \quad = \quad \sum_{i=1}^{N} (\vec{v}[i] - \mu)^2 \cdot F(i) \qquad (8)$$

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

square root of the variance, which 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 and the expected solution quality of an algorithm helps the practitioner decide if re-running 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 re-running 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).

The standard deviation also helps to determine if the results of a particular run are anomalous. This may be particularly pertinent to algorithms that generate game-playing programs, since the current methodology is to present primarily the results obtained by the best run. If the best run is indeed an anomaly (e.g. it is far superior to the predicted expected solution quality even after the standard deviation is taken into account), then the results achieved would overstate the actual ability of the algorithm.

5. SCSA on Modulo Nim

This section shows how the *M²ICAL* tool can be used to examine the performance of a simple algorithm SCSA on a simple game-playing problem called *Modulo Nim*, or *ModNim* for short. The purpose behind performing this case study is twofold. Firstly, the simplicity of both the algorithm and problem allows us to explain the implementation of the *M²ICAL* model without having to handle extraneous factors that may be present in more complex instances. Secondly, this experiment represents a close to ideal setup for *M²ICAL*. The algorithm is simple enough that the neighbourhood function is easily and precisely captured, and the short duration of each game of ModNim allows us to increase the sample sizes of the Monte Carlo simulations, thereby increasing the accuracy of the estimations. Therefore, this case study can serve as a "proof of concept".

5.1. Modulo Nim

The rules of ModNim are simple. The initial position of ModNim contains K sticks. On a player's turn, he can remove no fewer than 1 stick and at most M sticks. The player who removes the last stick loses (and his opponent wins). We use the notation ModNim(K,M) to denote the game of ModNim with K sticks in the initial position and at most M sticks removed per move; we have chosen the game of ModNim(100, 3) for our case study. The winning strategy can be expressed mathematically as follows: to win, remove a number of sticks to leave n sticks so that n satisfies the equation $n \bmod (M + 1) = 1$.

In our experiments, we represent a deterministic ModNim(K,M) player using a vector $\{m_1, m_2, \cdots, m_K\}$ of K integers. The range of the first $M - 1$ elements m_i is $[1..i]$, and the range of the remaining elements is $[1..M]$; the element m_i represents the number of sticks that the player removes when there are i sticks left. To randomly generate a player, we simply randomly determine the value of each element in the vector within the presribed ranges.

Algorithm 6 Simple Comparison Search Algorithm (SCSA)

```
Choose s_(0) uniformly randomly from S; t := 1
while t < MAXGEN do
  Let s_(t) = Q(s_(t-1), s)
  if t = MAXGEN then
    return s_(t)
  else
    increment t
  end if
end while
```

5.2. Simple Comparison Search

The simplest comparison-based algorithm is one that successively improves a current solution by uniformly randomly finding a better solution and replacing it. We call this the *Simple Comparison Search Algorithm (SCSA)*, as given in Algorithm 6.

When SCSA is applied to the game-playing problem, then each solution s is in fact a player of the game PL. For our experiments, the comparison function $Q(PL, PL')$ plays a single game of ModNim(100,3) where PL is the first player and PL' is the second player, and returns the winner. Hence, the incumbent always plays as the first player.

5.3. Model Construction

The parameters were chosen semi-arbitrarily such that an acceptable degree of accuracy could be achieved within a reasonable amount of computation time. We set the number of states in the Markov chain N to 100. The number of randomly generated opponents M_{opp} used to estimate player strength was set at the value of $M_{opp} = 1000$.

Since the mutation function for SCSA involves choosing a random player, the neighbourhood of SCSA is already captured when populating the classes with M_{sample} randomly generated players. We decided to simply populate

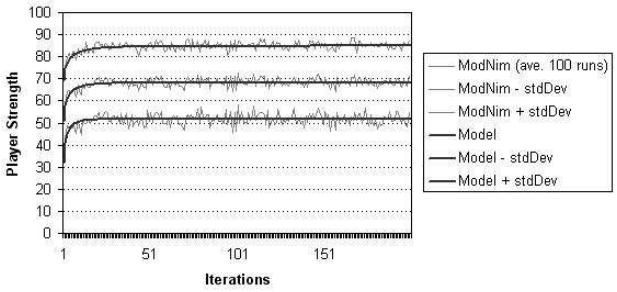

Figure 1. Model and experimental results for ModNim(100,3) using SCSA

the classes with $M_{sample} = 10,000$ randomly generated players and retain all of them for the subsequent steps of the process. In terms of Algorithm 1, this is equivalent to setting the parameters as $\hat{\gamma} = \infty$ and $M_{pop} = 0$.

Since SCSA is a strict hill-climbing algorithm, we can make use of Equation (6) directly to represent the Markov chain model of the system. The error (δ_{ij}) distribution for all pairs of states i and j is identical to the WPM W, which can be computed using Algorithm 2, such that $\delta_{ij} = W_{ij}$. The number of games played between every pair of classes M_{wpm} was set to a value of 1000.

For SCSA, $\lambda_{ij} = \frac{\gamma_j}{\Gamma_N}$. Therefore, we need not use Algorithm 3 although it will produce similar results. To estimate the λ_{ij}-distribution from our sample population, we simply count the number of players in each state and divide these values by M_{sample} to give an estimated value for $\frac{\gamma_i}{\Gamma_N}$.

Having determined both the δ_{ij}-distribution and the λ_{ij}-distribution, we substitute these values into Equation 6 to produce the transition matrix that represents the Markov chain model of the system. These experiments were performed on a Pentium-IV 1.6 GHz PC with 512MB RAM. It took approximately 24 hours for the entire process.

5.4. Experimental Results

By modeling the implementation of SCSA on ModNim(100,3) as a Markov chain as given in the previous chapter, we were able to discover several useful properties of the system. Figure 1 gives the results averaged over 100 runs of SCSA, along with the expected solution quality and spread forecast by the model. The bold black lines give the estimated player strength predicted by the model as well as the estimated player strength when the predicted standard deviation is added or subtracted; the grey line gives the corresponding values for the 100 runs of SCSA. We ran the algorithm to 1000 iterations each, but only the first 200 iterations are shown here for the purpose of clarity because the remaining iterations follow a similar trend.

The model predicts that the expected solution quality will eventually converge to a value of 68.3551%, closely matching the average solution quality achieved by the actual runs (which fluctuates within a range of 67% to 70%). Our model also shows that the solution quality of SCSA on ModNim(100,3) has a standard deviation of $\pm$16.4793%, and a visual inspection of the sample standard deviation of our 100 runs confirms that this prediction is also accurate. If the strength of ModNim(100,3) players produced by SCSA is normally distributed, then this indicates that different runs of SCSA on ModNim(100,3) could produce players of radically differing strengths, where about 68% of the players produced will have strengths over a range of over 32% of all player strengths.

Furthermore, using Algorithm 5 we find that SCSA converges to a stationary solution to a degree of accuracy of 3 decimal places in 207 iterations. This suggests that further iterations of the algorithm beyond 207 will not improve the expected solution quality found by SCSA by more than 0.001%.

6. HC-Gammon

In computer science, the greatest success in backgammon is undoubtedly Gerald Tesauro's *TD-Gammon* program [7]. Using a straightforward version of Temporal Difference learning called TD(λ) on a neural network, TD-Gammon achieved Master-level play. Pollack and Blair [6] implemented three versions of a simple hill-climbing method of training a backgammon player using the same neural network structure employed by Tesauro (known as *HC-Gammon*).

We were able to use the *M^{2}ICAL* tool to model the HC-Gammon algorithm, as reported in [5]. One issue that needed to be addressed was the fact that HC-Gammon used a neural network with all weights set to zero as the initial player, which we call the *all-zero neural network* (AZNN). 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 performed $M_{sample} = 200$ 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 used the current players of the 200 runs as the initial sample. We call this process *introducing a time-lag*.

Figure 2 shows the predictions given by the time-lag *M^{2}ICAL* model, compared to 25 runs of HC-Gammon for the first experiments conducted by Pollack and Blair; the

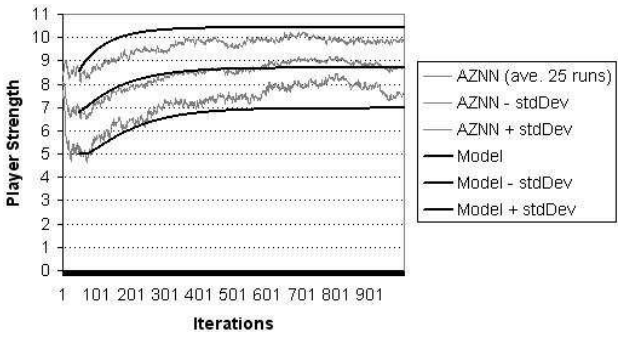

Figure 2. Time-lag model and experimental results for HC-Gammon using the AZNN initial player (figure reproduced from [5] to illustrate the capacity of M^2ICAL to analyze HC-Gammon).

values for the model begin at iteration 48. It 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, which fluctuates between 85.5% and 90.5%. Further details on the use of M^2ICAL for HC-Gammon are given in [5] and [4].

7. Conclusions

M^2ICAL is designed to be a practical analysis tool for complex real-world imperfect comparison algorithms. The formulation is kept general to maintain applicability to the maximum number of problems, and some of the design decisions were made with practical considerations in mind. Theoretically, all of the properties modelled by M^2ICAL can be obtained by running the target algorithm multiple times and performing Monte Carlo simulations on these runs. However, while the initial derivation of the M^2ICAL model is time-consuming, once the model is derived it can produce the results more quickly than running the algorithm itself several times. The model can also handle certain changes to the algorithm without requiring a re-run of the entire process, e.g. different victory conditions of the comparison function in HC-Gammon, game-playing improvements like opening books and endgame databases, etc. Therefore, the model is useful in predicting the effects of "what if" scenarios by modeling such changes, which aids the algorithm designer in making effective changes to the algorithm.

The M^2ICAL tool can be used to re-check the veracity of the analysis of existing algorithms. Since there have been no techniques available for the analysis of imperfect comparison algorithms in practical settings prior to M^2ICAL, it is likely that some of the previous analyses of such algorithms may be erroneous or unconfirmed; we could therefore use the M^2ICAL tool to either correct or confirm (or at least bolster) the analysis of existing research.

The method can also be helpful in the design of new algorithms since the practitioner can compare the effects of changes to certain parameters in the algorithm without having to re-implement and re-run the algorithms. The predicted standard deviation can help to determine if a re-run of the algorithm in hopes of producing a superior solution is justified, and the time to convergence provides a good ending point for the algorithm.

The M^2ICAL tool presents pioneering work on the analysis of imperfect comparison algorithms. In a field like intellectual games, it provides an objective alternative to the existing method of evaluating algorithms based on the results of the best player produced against benchmark players of possibly inaccurately determined strength. Other fields with similar difficulties may also benefit from using M^2ICAL.

References

[1] K. Chellapilla and D. B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. on Evolutionary Computation*, 5(5):422–428, 2001.

[2] D. B. Fogel. *Blondie24: Playing at the Edge of AI*. Academic Press, London, UK, 2002.

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

[4] W.-C. Oon. M^{2}ICAL*: A Technique for Analyzing Imperfect Comparison Algorithms using Markov Chains*. PhD thesis, School, Address, 2007. Submitted.

[5] W.-C. Oon and M. Henz. M^2ICAL analyses HC-Gammon. In R. C. Holte and A. Howe, editors, *Proceedings of the Twenty-Second Conference on Artificial Intelligence, AAAI-07*, pages 621–626, Vancouver, Canada, July 2007. AAAI Press.

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

[7] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.