M^2*ICAL* : A TECHNIQUE FOR ANALYZING IMPERFECT COMPARISON ALGORITHMS USING MARKOV CHAINS

OON WEE CHONG

(M.Sc.(Comp. Sc.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2007

To Bernice

Acknowledgements

This research has been a long and sometimes arduous journey, and I would never have made it to the finish line without the help of a number of wonderful people.

My sincerest gratitude goes to my supervisor Martin Henz, who took a naive PhD student low on confidence under his wing and taught him how to reach for the stars. I didn't think that I was capable of making a significant contribution to the field; I do now.

A lavish pesco-vegetarian dinner awaits my partner-in-crime Lim Yew Jin, seemingly the only other person interested in intellectual games research in the whole of Singapore. My initial thesis idea stemmed from him, and even though it eventually metamorphosed beyond recognition, Yew Jin was always available as a sounding board for ideas and a sympathetic ear as a fellow PhD student. Almost makes me forgive him for being 6 years younger and 6 times smarter.

A special thank you to Bruce Maxwell Brown, whose insights as a mathematician are invaluable to this mathematically-deficient computer science student. The number of cups of Spinelli coffee I owe him for his help tends to infinity.

Several are the professors who took time out to speak to a confused student about his half-formed, hare-brained ideas. These include Frank Stephan; Lee Wee Sun; Huang Zhiyong; Chionh Eng Wee; Rudy Setiono; Gary Tan; and Tay Yong Chiang. Your help

does not go unappreciated.

Finally, there is no-one I am more grateful to (and for) than my wife Bernice Ho, who was with me every step of the way. As the final chapter of this thesis ends, I look forward to embarking on the next chapter of my life with her love and companionship.

Contents

Summary

Practical optimization problems are often not well-defined, in the sense that the quality of a solution cannot be easily calculated. Many algorithms that attempt to solve such problems are comparison-based, i.e., they have a comparison function that compares two (or more) solutions and returns the superior one. Due to the difficulty in calculating the quality of a solution, the comparison function employed is imperfect (i.e., it may make an error). Machine learning algorithms in intellectual games (like chess or checkers) often fall into this class of imperfect comparison algorithms, since two solutions (players) are compared by playing them against each other, and there is generally a non-zero probability that the weaker player will defeat the stronger.

This thesis describes a 4-step process to model imperfect comparison algorithms into Markov Chains, using Monte Carlo simulations to handle the effects of comparison errors. We call this process the **M**onte Carlo **M**arkov Chain for **I**mperfect **C**omparison **AL**gorithms method, or the M^2ICAL method for short. Once the algorithm is modeled as a Markov Chain, it can then be analyzed using existing Markov Chain theory. Information that can be extracted from the Markov Chain include the estimated solution quality after t iterations; the standard deviation of the solutions' quality; and the time to convergence.

The *M*^{2}*ICAL* method is a 4-step process. In step 1, a representative sample population of the algorithm's search space is generated and allocated into the various classes in the Markov Chain; this sample population is used to find the probability distributions of the algorithm. Both the comparison error of the algorithm's comparison function and the neighbourhood distribution of the algorithm are estimated using several Monte Carlo simulations in steps 2 and 3 respectively. Finally, these distributions are used to devise the transition matrix of the Markov Chain model of the algorithm in step 4.

We take problems from the field of intellectual games as the running examples throughout the dissertation. In particular, we describe two case studies that show the capabilities of the *M*^{2}*ICAL* method. The first is the Simple Comparison Search Algorithm (SCSA) on the game of Modulo Nim (ModNim). This is a very simple hill-climbing algorithm on a strongly solved but computationally non-trivial game. The predictions made by the *M*^{2}*ICAL* method were very accurate in this case, which shows the potential capabilities of the technique in an ideal setting.

The second case study is the HC-Gammon generation program described in a 1998 publication by Pollack and Blair. It is a hill-climbing algorithm that generated players of the game of backgammon. The best solution was able to display a decent playing strength despite the simplicity of the approach, and these results were used by Pollack and Blair to question the importance of the temporal-difference learning technique used in another backgammon program TD-Gammon. We describe two different setups of the HC-Gammon experiments, one with a randomly generated artificial neural network as the initial player, and the other with a neural network with all weights and biases set to zero as the initial player. The results revealed several interesting aspects of the HC-Gammon experiments and enabled us to refute part of Pollack and Blair's claims.

As an analysis tool for practical algorithms, the various parameters involved in the *M*^{2}*ICAL* method must be carefully considered in order to make the modeling of the algorithm worthwhile. The effects of these parameters are discussed, along with some

other important issues that this pioneering work has failed to address. In any case, the M^2ICAL method is the first technique that can objectively analyze imperfect comparison algorithms, which will be useful for algorithm designers and practitioners.

List of Figures

List of Algorithms

Part I

Foundations

This dissertation presents a technique for analyzing imperfect comparison algorithms on optimization problems using a Markov Chain model that utilizes Monte Carlo simulations in its construction, using game-playing problems as our application domain. Our research therefore straddles several disciplines in computer science, including algorithm analysis, optimization algorithms, Markov Chain theory, Monte Carlo simulations and intellectual games research.

In this part of the thesis, we provide some foundational knowledge that is required for better understanding of the material that follows. Chapter 1 introduces the problem that we are trying to solve, which is to analyze the efficiency and effectiveness of imperfect comparison algorithms on optimization problems. We give an overview of such algorithms with an emphasis on the game-playing problem, and also detail the motivations behind our research. Chapter 2 provides background knowledge on existing research in the fields relevant to our work, and Chapter 3 gives the definitions and notations used throughout the dissertation.

Chapter 1

Introduction

1.1 Imperfect Comparison Algorithms

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. This could be due to the sheer complexity or size of the problem, or there could be other external factors that prevent a solution from being correctly evaluated. In these types of problems, it can be difficult or impossible to objectively evaluate the quality of a solution. Examples of such problems include:

- **Inaccurate or noisy data.** The accuracy of any algorithm can only be as good as the accuracy of the input data. If the input data to an algorithm is erroneous, then so will the output of the algorithm based on it. This phenomenon is sometimes called *"garbage in, garbage out"*.

- **Predictive algorithms.** Such algorithms usually only have information from the present time t, and have to provide a solution that is valid at some future time $T > t$. However, without knowing precisely how the problem will change in the future, predictive algorithms cannot guarantee an optimal solution at time T.

- **Imprecise problem definitions.** Many real-life problems are highly complex, e.g., large timetabling problems with several constraints. With so many different factors to consider, it is difficult for human users to accurately define their user requirements. While there may be algorithms that can solve such a problem precisely, their solutions are limited by the ability of the users to define their problem.

- **Problem mapping.** Sometimes, it is preferable to map a difficult problem to another problem that is not completely identical but better understood. For example, it has been shown that the paper spread problem in examination timetabling (i.e., increasing the amount of time between successive examination papers for each student) can be mapped to the vehicle routing problem, and can then be solved using a standard vehicle routing algorithm [LHO01]. However, since the target problem and the mapped problem are not identical, the final solution is not generally optimal.

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. Comparison-based algorithms range from simple greedy neighbourhood searches to genetic and evolutionary algorithms. However, since real-life optimization problems may not be well-defined, comparison-based algorithms must often employ a comparison function that is not 100% accurate as a result. We call algorithms that rely on such imperfect comparison functions *imperfect comparison algorithms*.

In real-world applications, the algorithmic solutions to such problems can be exceedingly complex, involving several different interacting components. Due to the complexity of such algorithms, it is difficult to evaluate their effectiveness; the efficacy of genetic algorithms, for example, is dependent on several factors including the data representation, mutation and crossover rates and methods, population size and number of

generations. While it is possible to establish upper, lower or average bounds of performance in some instances, a more precise estimation of algorithm performance is usually not performed. The difficulty is exacerbated if the target problem is not well-defined, because the inability to correctly evaluate produced solutions reduces the accuracy of any such analysis.

This research proposes a technique for the analysis of imperfect comparison algorithms. The technique 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 our technique **M**onte Carlo **M**arkov Chain for **I**mperfect **C**omparison **AL**gorithms, or M^2ICAL[1] for short; the models produced using this technique are similarly called M^2ICAL models. As far as we know, there have been no previous attempts to analyze the performance of complex imperfect comparison algorithms in practical settings.

1.2 The Game-Playing Problem

The running examples throughout this research will be the game-playing problem, which is an archetypal imperfect comparison problem. The game-playing problem has a long history in computer science. The aim of this problem is to create a program that can play an intellectual game such as chess or checkers well, as measured by its results against other (human or computer) players. The greatest practical successes have been achieved by using very deep brute force searches in order to find the move leading to the position that is evaluated as being of the highest quality; the evaluation function being maximized is often a hand-tuned weighted sum of features of a position. For example, the IBM supercomputer *Deep Blue* was able to defeat then World Chess Champion Gary

[1] pronounced *Michael*.

Kasparov 3.5-2.5 in an exhibition match using this approach [BN97].

Recently, more emphasis has been placed on using intellectual games as a test bed for machine learning techniques. The aim of such research is to use machine learning methods to generate a game player capable of playing an intellectual game at a high level of proficiency, usually while avoiding the use of very deep searches that may obscure the effect of the machine learning technique itself. Notable successes in this field include the backgammon program *TD-Gammon* [Tes95] that was based on a technique called temporal difference learning, and the checkers program *Anaconda* [CF01, Fog02] based on co-evolution of neural networks. Unfortunately, the results-oriented mentality of intellectual games research in computer science seems to have been passed on to this field as well. Research in this area involves applying a particular machine learning technique on a game, and then evaluating the final player produced by playing it against existing players (human or computer); while this approach to research can showcase the ability of a technique to create a strong game player, it provides little insight on how the technique achieves this feat, nor does it provide any bounds on algorithm performance. Such an approach would most likely be insufficient to prove an algorithm's capabilities if it was applied to algorithms in other fields in computer science.

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, for a given definition of player strength. This phenomenon has been dubbed the "Buster Douglas Effect" [PB98], named after the 45-1 underdog heavyweight boxer who defeated overwhelming favourite Mike Tyson to become World Heavyweight Champion. Even though algorithms like competitive co-evolution have been found to converge to optimality when this phenomenon does not occur [RB96], in all practical games the Buster Douglas Effect is present. Since the relative strength of two players is usually found

by playing them against each other, the Buster Douglas Effect essentially results in an imperfect comparison.

In this thesis, we describe how we can use Markov Chains to examine imperfect comparison algorithms that are applied to the game-playing problem. Markov Chains have been used to model algorithms in computer science for several decades. As long as the next state of an algorithm is related only to the current state, it can be modeled as a Markov Chain and analyzed using a variety of mathematical tools. Examples of such research include the analysis of Genetic Algorithms [WZ99], Simulated Annealing [Sor91] and Local Search algorithms [And02], which used Markov Chains to show important algorithm properties (e.g., proof of algorithm convergence). We show how the performance of imperfect comparison algorithms on the game-playing problem can be translated into Markov Chain form by using several Monte Carlo simulations to estimate various behaviourial aspects of the algorithm. This then allows us to take advantage of existing knowledge on Markov Chains to better evaluate the strengths and weaknesses of algorithms that attempt to generate game-playing programs.

1.3 Motivations

Most of existing computer science research in intellectual games is almost entirely results-oriented, in the sense that the objective of the researchers was to create a game-playing program capable of playing a particular game well in tournament conditions. The greatest successes were achieved using a large variety of methods, including very deep searches, human-defined evaluation functions, time management schemes and even dedicated hardware; all of these methods are usually heavily customized for maximum effectiveness for the particular game and/or tournament format. Since the domain of intellectual games is competitive by its very nature, this approach to research is perfectly understandable and sound. Furthermore, as the field is now very well-researched, it becomes exceedingly difficult to find new ways to improve the strength of a game-playing

program, so any new scheme that is able to do so will be of interest to practitioners in the field.

Academically speaking, there are disadvantages to this heavily results-oriented outlook. As long as a program succeeds in playing a strong game, there is interest in publications revealing the technique used to produce the program. When the technique used involves the combination of several different methods, it is often left to the interested practitioner to decide for themselves the relative worth of each of the methods when applied to their own game. Also, many of the techniques that are applied to a particular game cannot be used for another game, e.g., the evaluation functions used by the IBM chess supercomputer *Deep Blue* [BN97], which involves some 8000 components, cannot be easily applied to any other game. Many approaches also have some random component, which has a significant effect on the strength of the generated player, further obscuring the relative worth of the overall technique itself.

More recently, there has been work on using intellectual games as simply a test bed for machine learning techniques, where the aim of the research is not necessarily to create the strongest possible game-playing program, but to showcase the ability of the machine learning technique in the games domain. However, there are two significant problems with current research in this area. Firstly, existing publications usually focus on the results achieved by the best player generated by their approach, even when several different trials were required; while this gives an indication of the upper-bound potential of the approach, it does not provide much information on its average or worst-case performance. Secondly, it is difficult to judge the strength of the produced player in a fair and convenient manner.

One of the aims of this research is to analyze the performance of certain algorithms that search for strong game-playing programs. To do so, we model these algorithms into **M**arkov Chains using a number of **M**onte Carlo simulations to estimate the pertinent properties of the **I**mperfect **C**omparison **AL**gorithm; we have taken the liberty of naming

the models that are derived in this way *M*^{2}*ICAL* models.

The basic idea behind the *M*^{2}*ICAL* method is simple. Suppose we wish to analyze an imperfect comparison algorithm *A* that searches for a strong player for the game of chess. This is accomplished in 4 steps:

1. Using algorithm *A*'s player-generation function, we generate several "intermediate" players. Each of these players are evaluated by playing them against a number of randomly generated opponents to estimate their strengths, and these players are allocated to classes according to their estimated strengths. The aim of this step is to produce a population of players that are representative of the variety of players that algorithm *A* produces in the long run.

2. Using the representative population, we estimate the probability that a player from a weaker class will beat a player from a stronger class, for all pairs of classes (called the *error distribution*). This is done by randomly selecting players from each pair of classes and playing them against each other.

3. We estimate the probability that a player from class i will produce a player from class j for all pairs of classes i and j in one iteration of algorithm *A* (called the *neighbourhood distribution*). This is done by repeated applications of algorithm *A*'s neighbourhood function on the members of the representative population.

4. The transition matrix for the Markov Chain representation of algorithm *A* can now be derived using the error and neighbourhood distributions, which can then be analyzed using existing Markov Chain theory.

Even though the *M*^{2}*ICAL* model involves some approximations and therefore cannot guarantee complete accuracy, it allows us to estimate the performance of algorithms that are applied to problems where the quality of solutions is difficult to measure such as intellectual games. This helps us to evaluate the suitability of an algorithm to such

problems. In particular, when using Markov Chains to model the performance of an imperfect-comparison algorithm on an optimization problem, we pursue the following aims:

- To find the expected quality of the solution generated by the algorithm after a sufficiently large number of iterations such that the effect of the initial state is negated. This provides an upper bound on the expected quality of an algorithm.
- To find the number of iterations of the algorithm required for the expected solution quality to converge to its upper bound value.
- To find the expected quality of the solution after a fixed amount of time as measured by the number of iterations. This is important for time-crucial applications.
- To find the spread of the quality of the solutions generated by the algorithm, as measured by its standard deviation.
- To determine the importance of the various factors affecting algorithm performance. This allows the algorithm developer to concentrate his efforts on the aspects of the algorithm that have the greatest effect.

Ideally, the model should capture all the pertinent aspects of both the problem and the algorithm, and be able to predict the algorithm's performance to a reasonable degree of accuracy. Furthermore, the analysis method should provide a performance advantage over simply running Monte Carlo simulations on the solutions produced by running the algorithm itself.

This research examines the game-playing problem by implementing the M^2ICAL model on two games. The first game is called *Modulo Nim* (*ModNim* for short), which is a simple game that can be easily customised to suit our needs. The purpose of modeling ModNim in our thesis is to show the accuracy of our model on a small game using a simple algorithm, which represents a setting that is close to ideal. The second game is

backgammon, which is one of the most popular games in the world and the subject of much current research. We model the hill-climbing algorithms used by Pollack and Blair when generating their backgammon player, which we refer to as *HC-Gammon* [PB98]. Despite constraints on time and computational resources, we were still able to discover some interesting results in a reasonable amount of time by making some compromises on the granularity of the results.

1.4 Thesis Organization

This thesis is divided into 4 parts. **Part I: Foundations** gives an introduction to the problem, and provides the necessary background knowledge for the proper understanding of the rest of the thesis. Chapter 2 gives a brief description of the existing work that is relevant to our discussion, while Chapter 3 presents the definitions and terminology that will be used throughout this dissertation.

Part II: The *M*^{2}*ICAL* Method provides the basic formulation of the *M*^{2}*ICAL* method. In Chapter 4, we give the step-by-step description of the entire process, which results in the derivation of a Markov Chain model of the target imperfect comparison algorithm. We then explain how this Markov Chain model can be used to discover important properties of the modeled algorithm in Chapter 5. This part concludes with a description of the implementation of the *M*^{2}*ICAL* method on the simple, idealized problem of the Simple Comparison Search Algorithm (SCSA) on the game of Modulo Nim, which shows the capabilities of the *M*^{2}*ICAL* method in a close to optimal setting.

Part III: Case Study: HC-Gammon details the primary case study of this research, namely the HC-Gammon backgammon program by Pollack and Blair [PB98]. Chapter 7 first introduces the game of backgammon, and then describes the HC-Gammon experiments in some detail; it concludes with a description of the implementation of the *M*^{2}*ICAL* method on SCSA on backgammon. Chapter 8 describes how the *M*^{2}*ICAL* method can be implemented in the particular case of the HC-Gammon experiments to

produce the Markov Chain representation of the algorithm. We performed two sets of experiments based on the HC-Gammon setup, the first using a randomly generated artificial neural network as the initial player (Chapter 9), and the other using a neural network with all weights and biases set to zero as the initial player (Chapter 10). The results of these experiments are summarized in Chapter 11, along with some discussions on their implications.

Part IV: Further Discussions gives our thoughts on some theoretical issues relevant to the M^2ICAL method. In Chapter 12, we provide some analysis on the significance of the various parameters of M^2ICAL method in terms of their effects on the predictive accuracy of the model and the computation time. We then discuss how the M^2ICAL model can be adapted to various practical algorithms in Chapter 13. Finally, the thesis concludes in Chapter 14, which summarizes the academic contributions made in this dissertation, and also presents some possible avenues for future work.

Chapter 2

Background

2.1 Markov Chains

A Markov Chain is a sequence of random variables with the Markov property (i.e., the conditional probability of the next state is dependent only on the current state; the formal definition is given in Section 3.3). Since the next state is only dependent on the present state, Markov Chains are useful in modeling systems that are memoryless. The first results were produced as early as 1906 by Andrey Markov [Mar06, Mar71], and the process bears his name.

The key task in modeling a system as a Markov Chain is to determine the transition probabilities, which are the conditional probabilities for the system to go to a particular new state given a particular current state, for all states. If these values are known, then the Markov Chain is able to compute the probability that the system will be in any particular state at a future time. Markov Chains have been used in many diverse fields including computer science, mathematics, engineering, operations research, biology and economics. Markov Chains can be used to calculate the mean time to failure of components in important systems like air traffic control systems; to analyze economic models such as population forecasting and financial planning; to locate bottlenecks in

communication networks; and many other applications.

There is much existing work involving the modeling of algorithms using Markov Chains. Examples of such research include Genetic Algorithms [NV92, WZ99], Simulated Annealing [Sor91] and Local Search algorithms [And02]. The information that is most often sought when modeling a system into a Markov Chain is the probability of being in each state after a certain amount of time has passed when the system becomes operational. Often, the desired information is the probabilities of being in each state after a sufficient amount of time has passed such that the influence of the starting state is erased, which are called the *stationary probabilities* or the *stationary distribution*; when the system modelled is an algorithm, the stationary probabilities give the states to which the algorithm *converges*.

Generally all existing work has focused on systems that are precisely defined, in the sense that it is possible to instantiate a Markov Chain that models the system with complete accuracy. In this research, we examine a class of problems where this is not possible, namely the *game-playing problem*, where the task is to create a player that can play an intellectual game (like chess or checkers) well. Furthermore, much of the existing work with Markov Chains has been theoretical, where systems and algorithms are represented using Markov Chains, and then theoretical bounds for properties like time to convergence are proven and stationary distribution vectors are calculated. In contrast, we will go beyond theoretical considerations and provide a procedure with which particular instances of imperfect-comparison algorithms can be analyzed using Markov Chains.

2.2 Monte Carlo Simulations

A Monte Carlo simulation describes the process of determining the values of a probability distribution by repeatedly and randomly selecting instances of it, and then examining the proportion of each instance. A common analogy to describe a Monte Carlo

simulation is that of finding the proportions of colours on a multi-coloured dartboard by throwing several darts at it, and then counting the number of darts that strike each colour. It was given its exotic name (which refers to a famous casino in Monaco) by Polish-American mathematician Stanislaw Ulam in 1946.

Monte Carlo simulations have applications in several fields, including the study of systems where there is a large amount of uncertainty in the inputs like risk calculation in business; the calculation of multidimensional definite integrals with complicated boundary conditions; simulated annealing for protein structure prediction; and in semiconductor device research to model the transport of current carriers. In general, Monte Carlo simulations are useful in determining probability distributions that cannot be easily calculated in a precise way.

A crucial result in Monte Carlo simulations is that the standard deviation of the result from the actual value is inversely proportional to the square root of the number of samples. In applications where the result must be accurate to several decimal places, this slow rate of convergence of the accuracy of Monte Carlos simulations is considered a major failing. On the other hand, if a standard deviation of about 10% on the result is acceptable, then only about 100 trials are required. For our analysis of algorithm performance, we have found that the square root convergence rate for the accuracy of Monte Carlo simulations is sufficient.

Markov Chains and Monte Carlo methods have been used together in a class of algorithms called *Markov Chain Monte Carlo* (MCMC) methods. The aim of MCMC methods is to discover probability distributions by constructing a Markov Chain with stationary probabilities equal to the desired (but unknown) distribution, and then finding the distribution by numerically determining the Markov Chain's stationary distribution. Despite the superficial similarity in the names, our technique has little to do with MCMC methods of this sort. In this research, we make use of Monte Carlo simulations in a more direct way, by discovering the relevant probability distributions of algorithms in order to

construct the corresponding Markov Chain.

2.3 Intellectual Games

The game-playing problem is a well-established problem in the field of artificial intelligence, with roots going as far back as Arthur Samuel's seminal work on machine learning using the game of checkers [Sam59, Sam67]. It has always been a highly results-oriented field, and the paramount achievement would be to create a program that could defeat the best human player in any intellectual game. When the supercomputer *Deep Blue* defeated the then-reigning world chess champion Gary Kasparov in a 6-game exhibition match (by a score of 3.5-2.5) in 1997, a major milestone in the history of intellectual games research in computer science was reached [BN97]. By using a combination of massively parallel processing, specialized hardware for chess-related operations like move generation, and various game-related improvements like optimized alpha-beta search, a comprehensive opening book and endgame database, and a laboriously constructed evaluation function involving some 8000 components, the IBM team was able to create a program that could search 200 million positions a second.

Much of the playing strength of *Deep Blue* can be attributed to the extraordinary number of positions searched by the program. Even though intellectual games research is classified under the field of artificial intelligence, and while there is no doubt that search is an integral part of human intellegence, it is of relatively less importance in our efforts to simulate human cognitive processes. However, a well-known result in computer games research is that the playing strength of a program increases almost linearly with minimax search depth [Tho82]; consequently, much of the early research efforts in the field prior to the *Deep Blue*-Kasparov result focused on finding ways to extend or speed up the minimax search. This led to several game-related improvements like transposition tables and forward pruning heuristics, but results in other aspects of artificial intelligence and machine learning stemming from games research has been

limited.

Recently, more emphasis has been placed on using intellectual games as a test bed for machine learning techniques. The aim of such research is to use machine learning methods to generate a game player capable of playing an intellectual game at a high level of proficiency. Notable successes in this field include the backgammon program *TD-Gammon* [Tes95] that was based on temporal difference learning, and the checkers program *Anaconda* [CF01, Fog02] based on co-evolution of neural networks. In general, research in this area involves applying a particular machine learning technique on a game, and then evaluating the final player produced.

In the field of computer science research on intellectual games, it is difficult to objectively and easily evaluate the strength of players. The main problem lies in the fact that in a match between two players of different strengths, there is in general a non-zero probability that the weaker player will defeat the stronger. In particular, if the comparison function used in an algorithm to evaluate the relative strengths of two players involves playing them in a match, then this non-zero probability is a comparison error. This difficulty is one of the major motivating factors in our research, and we take the possibility of a comparison error into account when evaluating the (expected) strength of a player that is produced by an algorithm.

To our knowledge, there is no existing research that directly addresses the effect of imperfect comparisons on the game-playing problem. For example, an existing framework on the competitive co-evolution algorithm [RB96] delivers its strongest results under the assumptions of full transitivity of player strengths (i.e., zero comparison error) or infinite memory. This project aims to establish a technique for modeling instances of algorithms on imperfect comparison problems such as intellectual games into Markov Chains using Monte Carlo simulations. We can then make use of Markov Chain theory to analyze and evaluate these algorithms.

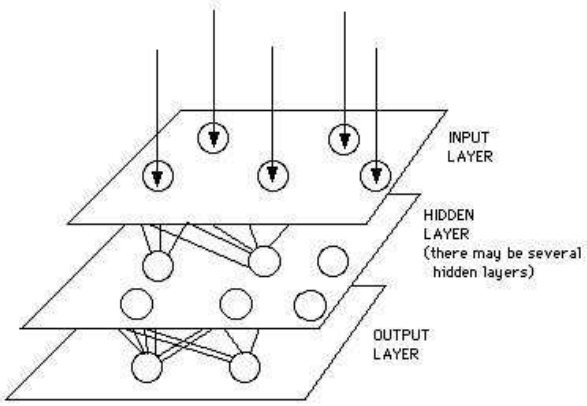

Figure 2.1: Basic structure of an artificial neural network

2.4 Evolutionary Artificial Neural Networks

Artificial neural networks (ANN) [Hay99, AM92], or simply neural networks, model the workings of the brain (and in particular the human brain). A brain consists of a huge number of interconnected cells called neurons. Basically, a real neuron "fires" when a stimulus excites it beyond a certain threshold, and sends this output into several other neurons. Therefore, depending on the input stimuli, this complex network of neurons produces different outputs according to the makeup of the neurons. While biological neurons can have as many as 200,000 different inputs, artificial neural networks simulate this structure in a simplified form to create a computational device that takes a certain input to produce the desired output.

The basic structure of an artificial neural network is shown in Figure 2.1. An ANN is divided into 3 layers, namely the input layer, the hidden layer(s) and the output layer. Each layer comprises a number of artificial neurons, which are simply functions that receive an input and produces an output. These neurons are connected by weighted connections that multiply the values produced by each neuron. The data to be computed is entered into the input layer; this data is passed through the one or more hidden layers; finally, an output is produced via the output layer.

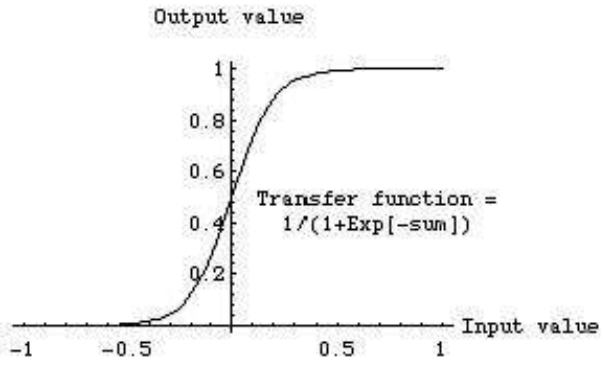

Figure 2.2: Sigmoid transfer function

The function (known as the transfer function) that each artificial neuron computes transforms the input into a real number output. Usually, the output is restricted to some desired range of values, for instance within [0, 1] or [-1, 1]. Such functions include sine, hyperbolic tangent and sigmoid. For example, the sigmoid function (shown in Figure 2.2) converts the input value into a value between 0 and 1. These transfer functions approximate the threshold function (where values are returned when the input is beyond a certain threshold), but smoothens the function.

The most important part of artificial neural networks is the connection weights. These weights determine the result of the neural network computation. The key to the accuracy of a neural network solution to a problem is the correct assignment of connection weights. In order to find the best assignment of connection weights, neural networks have to be "trained". One method of training neural networks uses *evolutionary computing*, which is a branch of computer science that makes use of some of the principles of Darwin's Theory of Evolution to solve problems. Concepts like the survival of the fittest, mutation of species and the inheritance of parental traits by offspring are employed on a population of solutions such that, at the end of several "generations", a solution would be evolved to solve the problem at hand.

In general, evolutionary computing techniques begin with a set of randomized solutions, which form the individuals of the initial population. The fittest of these individuals (the definition of fittest being dependent on the problem) are selected for survival, and offspring are produced. These offspring differ from their parents via mutation and/or the inheritance of traits from both parents (termed "crossover"). The offspring, combined with the original parents, form the initial population of the next generation. Techniques that fall under the category of evolutionary computing include genetic algorithms, evolutionary programming and artificial life.

The combination of the evolutionary process with artificial neural networks is a natural idea, since it parallels the evolutionary process that gave rise to brains in nature. Termed *evolutionary artificial neural networks (EANN)*, this technique makes use of the evolutionary computing mechanism to train artificial neural networks [Yao99]. The most straightforward implementation of EANNs involves the evolution of connection weights. The process begins with an initial population of neural networks of fixed architecture (usually fully connected) with randomized weights. Each network is then evaluated using a fitness function. The fittest individuals are retained, and offspring is produced using mutation and/or crossover operations, which affect the connection weights.

In this dissertation, our primary case study is the backgammon program HC-Gammon [PB98], which uses a simplified EANN; it uses mutation to generate offspring, and has a population with only a single member. While the lack of a multi-member population of solutions transforms the algorithm into a hill-climbing algorithm, it can still be classified as an evolutionary algorithm due to its use of mutation in its search for solutions.

Chapter 3

Definitions & Notations

This chapter describes some of the definitions and notations employed for the rest of the thesis. The basic concepts are explained here, while certain other symbols and notations are explained in the main body of the thesis as they are encountered. For a summary of the symbols used, refer to the Table of Symbols in Appendix A.

3.1 General Concepts

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.

An important class of algorithms that solves such problems is based on the comparison of two solutions. Thus, we define a *comparison function* $Q : S \times S \rightarrow S$ as a function that takes two solutions s_i and s_j and returns one of them. Conceptually, comparison functions compare two solutions to a problem and return the one that it considers superior. A *perfect comparison function (PCF)* for a problem **P** would be

$$PCF(s_i, s_j) = \begin{cases} s_i & F(s_i) \geq F(s_j) \\ s_j & \text{otherwise.} \end{cases} \quad \text{for all} \quad s_i, s_j \in S \tag{3.1}$$

Equation 3.1 is perfect in the sense that it exactly equates the relative desirability of two solutions to the problem. However, it is not always possible to formulate a PCF for a given problem, in particular when the objective function F of the problem cannot be calculated practically. Therefore, alternative *imperfect* comparison functions are used that approximate the requirements of the problem.

3.2 Games

The application domain in this dissertation are instances of the game-playing problem. The task of the game-playing problem in computer science is to create a program that can play a game well (so the solution space S of the game-playing problem is the set of all possible game-playing programs). The "better" a player is, the greater its "playing strength". However, despite the seemingly intuitive notion of playing strength that exists in common usage, there is no single commonly accepted method of measuring the strength of a player. In this section, we attempt to define the game-playing problem.

Games can be expressed in terms of a directed graph $G = (V, E)$, where each vertex represents a valid position in the game, and $E = \{(v_i, v_j) |$ there is a legal move from v_i to $v_j\}$. All terminal vertices have a *value* from an ordered set that denotes its desirability, e.g., {win, draw, loss}. We can use retrograde analysis to assign all non-terminal vertices with a corresponding value [Tho86]; hence, all positions v have a value $val(v)$. For example, Figure 3.1 shows the graph representation for the game of tic-tac-toe from a particular starting position. For the sake of simplicity, we only examine 2-player, turn-taking games that take their values from the set of {win, loss} where a win is preferable to a loss. The *game-theoretic value* of the game is the value of the

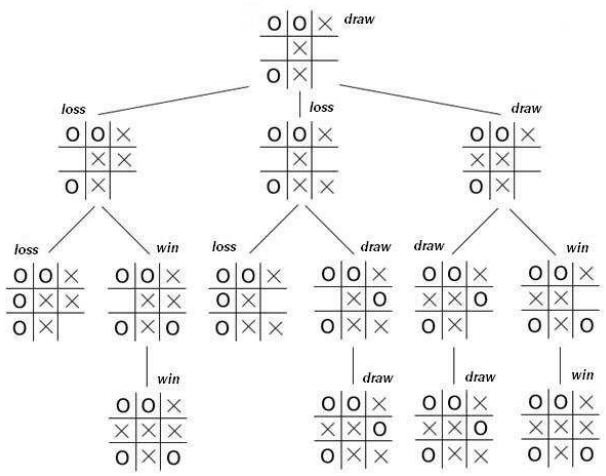

Figure 3.1: Game tree for a tic-tac-toe game from the given starting position (X to play)

initial position, so the result of a game will be its game-theoretic value given optimal play. Since we only examine win/loss games, it follows that the initial position is a win for either the first or second player.

For a given game, we also define a function $M : V \rightarrow \bar{E}$ that takes as its argument a position and returns the set of all legal moves from that position. Hence, $M(v_i) = \{(v_i, v_j)|(v_i, v_j) \in E\}$.

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 as output 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. If the only input is the position and the function is deterministic, then this function is also called a *strategy*. However, practical game-playing programs are often not deterministic, and also sometimes take information other than the current game position into consideration when making a move (e.g., opponent's previous history, time remaining, etc.); all such functions are also considered players.

The most natural comparison function to use when comparing two players is to simply play them against each other in a single game, and select the player that wins. Formally, this *beats* comparison function (BCF) is defined as follows:

$$BCF(PL_i, PL_j) = \begin{cases} PL_i & PL_i \text{ beats } PL_j \\ PL_j & \text{otherwise.} \end{cases} \quad \text{for all} \quad PL_i, PL_j \in S \tag{3.2}$$

For turn-taking games, the first argument is the first player and the second argument is the second player. Note that in general, $BCF(PL_i, PL_j) \neq BCF(PL_j, PL_i)$. 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*. However, even though there is an instinctive layman's notion of player strength, there is at present no universally accepted definition of the concept. In this thesis, we make use of the following definition of player strength. We make use of the notation 1_f to represent 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*

$$PS(PL_i) = \sum_{1 \leq j \leq |S|} 1_{PL_i \succ PL_j} + \sum_{1 \leq j \leq |S|} 1_{PL_j \prec PL_i} \tag{3.3}$$

The strength of player PL_i is found by counting all the players that it beats as both the first and the second player from the initial position. This definition most closely approximates the layman's notion of player strength, since the performance of a player in a tournament is determined by its results when playing on both sides. However, from a purely theoretical standpoint, this objective function contradicts the accepted notion of an "optimal" player, which is basically defined in existing computer science research as a player that plays best from the superior side, assuming that the game in question is a game-theoretic win for one side; such an optimal player may have a lower strength than a sub-optimal player under our definition.

3.3 Markov Chains

Our research relies heavily on existing Markov Chain theory. A sequence $\{X_t\}_{t\geq 0}$ of random variables with values from a set I is a *discrete-time stochastic process* with *state space* I. We assume in this thesis that I is finite; we define N to be the number of elements in the state space, and the elements in I are denoted by $i, j, k, ...$ Traditionally, the states in a Markov Chain are labelled $1..N$, which is the convention we employ in this thesis. If $X_t = i$ for some $i \in I$, we say that the process is in state i at time t.

Definition 3 (Markov Chain) *A* **Markov Chain** *is a discrete-time stochastic process where for all integers* $t \geq 0$ *and for all states* $i_0, i_1, \cdots, i_{t+1}$*,*

$$P(X_{t+1} = i_{t+1}|X_t = i_t, X_{t-1} = i_{t-1}, \cdots, X_0 = i_0) = P(X_{t+1} = i_{t+1}|X_t = i_t) \quad (3.4)$$

This equation is known as the *Markov property*. A Markov Chain is *homogenous* if the right hand side is independent of t.

Definition 4 (Transition Matrix) *The matrix* $P = \{p_{ij}\}_{i,j\in I}$ *is the* **transition matrix** *of a homogenous Markov Chain if, for all* t*,*

$$p_{ij} \quad = \quad P(X_{t+1} = j|X_t = i) \quad (3.5)$$

Since the values for each row i of a transition matrix give the probability of moving from state i to all states, each row in a transition matrix sums to 1.

Markov Chains are completely defined by their transition matrices. Another way to represent a Markov Chain is by using a weighted directed graph, where each node is a state in I, and the weight of each edge (i, j) is equal to p_{ij} in the transition matrix. By convention, the edge representing p_{ii} is often omitted; its weight is $1 - \sum_{j\neq i} p_{ij}$. This graphical representation is often useful for visualizing the Markov Chain.

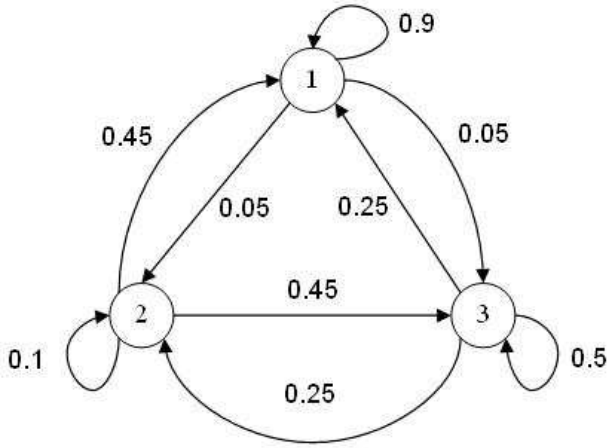

Figure 3.2: Graphical representation of Markov Chain of coin-guessing algorithm on 90% biased coin

For example, suppose an algorithm chooses a strategy for the guessing of coin flips, and it only considers 3 strategies. Strategy 1 always chooses “heads”, strategy 2 always chooses “tails”, and strategy 3 randomly chooses “heads” or “tails” with equal probability. If the current strategy selected by the algorithm incorrectly guesses the coin flip result, it randomly chooses one of the two other strategies to employ for the next guess. Otherwise, the algorithm retains its existing strategy.

The Markov Chain representing the above algorithm would have 3 states, corresponding to strategies 1, 2 and 3 respectively. Suppose this algorithm is implemented on a biased coin that lands on “heads” 90% of the time. If the algorithm is in state 1, it will guess correctly with probability 0.9 and hence remain in the same state, otherwise it will go to state 2 or state 3 with equal probability of 0.05. If the algorithm is in state 2, it will guess correctly with probability 0.1, but will guess incorrectly and therefore go to state 1 or state 3 with equal probability of 0.45. Finally, if it is in state 3, it will guess correctly with probability 0.5, otherwise it will move to state 1 or state 2 with equal probability of 0.25.

Figure 3.2 gives the graphical representation of the Markov Chain corresponding to the above algorithm. As stated previously, the self-loops in this diagram can be omitted. The transition matrix for this Markov Chain is as follows:

$$\begin{pmatrix} 0.9 & 0.05 & 0.05 \\ 0.45 & 0.1 & 0.45 \\ 0.25 & 0.25 & 0.5 \end{pmatrix}$$

Part II

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

Markov Chain theory has a wealth of literature dating back several decades. As a result, if we can accurately model imperfect comparison-based algorithms on optimization problems using Markov Chains, we can make use of existing tools to analyze their performance. In this part of the thesis, we show how an imperfect-comparison algorithm that is applied to the task of finding a strong player for an intellectual game can be modelled with a Markov Chain using a number of Monte Carlo simulations, and explain what can be learnt about the algorithm by doing so.

The model construction process is divided into 4 phases. The first phase involves generating a sample population of players that is representative of the algorithm's search space. The second phase makes use of these players to discover the probability that a weaker player defeats a stronger player given their relative player strengths. The aim of the third phase is to estimate the distribution of player strengths found by the algorithm's search function. Finally, the fourth phase combines the results found in the second and third phases to construct the transition matrix of the Markov Chain model of the algorithm. The purpose of this process is to make use of the characteristics of the sample population and the algorithm's search function to predict future *trends* in the algorithm. By mining the relevant probability distributions of the sample population, we can use the Markov Chain model to estimate how the algorithm will perform.

Chapter 4 presents an overview of the M^2ICAL method, which gives the steps involved in deriving a Markov Chain representation of an imperfect comparison algorithm. In Chapter 5, we show how the derived M^2ICAL model can be used to discover several useful properties of the algorithm. This part of the thesis concludes in Chapter 6 with an example of the M^2ICAL method at work on the case of the Simple Comparison Search Algorithm on the game of Modulo Nim.

Chapter 4

Method Overview

The task of modeling an algorithm as a Markov Chain begins with dividing it into distinct states, where the set of all states encompasses the set of all possible solutions to the problem. For example, a simple way to do so is to have a number of states equal to the number of possible solutions to the problem. However, in our research, we are primarily concerned with the performance of the algorithm in terms of the quality of the final solution as measured by the objective function F. Therefore, each state corresponds to a unique measure of quality of solution, which we call a *class*; how this is done will be detailed in the relevant sections of this thesis. We place a directed edge from state i to state j if it is possible to move from i to j in one iteration of the algorithm; the edge (i, j) is assigned a weight equal to the probability of the transition.

Assuming that we are able to divide the algorithm into a finite number of states, the main difficulty in modeling it lies in determining the weights for each edge. To do so, we perform 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.

To derive the Markov Chain model, the main technique is to use Monte Carlo simulations to estimate the various values involved. Hence, we have named our technique **M**onte Carlo **M**arkov Chain for **I**mperfect **C**omparison **AL**gorithms, or *M²ICAL* for short. Similarly, the Markov Chain models produced using this method are called *M²ICAL* models.

In this chapter, we explain how the *M²ICAL* method can be used to estimate the values of the transition matrix for the Markov Chain model of an algorithm that searches for strong game-playing programs. However, it should be reasonably simple to adapt our approach to imperfect comparison problems other than the game-playing problem.

4.1 Estimating Player Strength

Evaluating the strength of the generated players is a major difficulty in games research. A layman's notion of a player's strength in an intellectual game corresponds to its ability to beat other players. When analyzing a trivial or small game, it may be possible to find a player's strength by fully enumerating all players, but this is impossible for practical games since the number of possible players in such games is astronomically large.

In this research, we make use of Monte Carlo simulations 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 uniformly 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 $F'(PL_i)$:

$$F'(PL_i) = \min\left(N, \left\lfloor \sum_{j=1}^{M_{opp}} (1_{PL_i \succ PL_{ij}} + 1_{PL_{ij} \prec PL_i})/(g \cdot M_{opp}/N) \right\rfloor + 1\right) \quad (4.1)$$

This equation simply calculates the proportion of games won by the player and translates this value into the appropriate class. Suppose we wish to produce a Markov Chain with $N = 100$ states. Hence all players are divided into 100 sets corresponding to 100 states in the Markov Chain, whereby the members of each set have a similar estimated strength. 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 M_{opp} is set to 1000, then all players that win between 0 and 9 games are in state 1, those that win between 10 and 19 games are in state 2, and so on.

Conceptually, the space of all possible players is divided into N classes where all players with the same estimated player strength belong to the same class. We use the term *class* to denote the set of players of a given player strength, and the term *state* in the usual way to denote a state in a Markov Chain. Since there is a one-to-one correspondence between classes and states in our formulation, the two terms are often used interchangeably throughout the rest of this dissertation.

Using this technique, we can find the estimated strength of a player by playing it against M_{opp} randomly generated opponents. If the random generation of opponents is assumed to take $O(1)$ time, then the evaluation of each player takes $O(M_{opp})$ time. It turns out that the process of evaluating players takes up the bulk of the computation time for the *M²ICAL* method.

An intrinsic assumption of the *M²ICAL* method is that all players in the same class are identical in every way. This assumption is only necessarily true if the number of classes N is equal to the number of all possible players (i.e., each class consists of only one player), which is an infeasibly large number in all practical applications. However, the *M²ICAL* method is still able to produce a reasonable analysis of algorithm performance despite this simplifying assumption. Furthermore, the effect of this assumption can be reduced by increasing the number of classes N at the expense of added computation time; we discuss the implications of this in Section 12.1.

4.2 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. Ideally, we wish to have at least one representative from each class. 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. For instance, the initial player in an artificial neural network representation is often a network with its weights and biases uniformly randomly determined within a range of values (as is the case for our first set of experiments on the HC-Gammon backgammon program - see Section 8.2). 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 thesis, 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 5 (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 function of γ at i, i.e.,*

$$\Gamma_i = \sum_{j=1}^{i} \gamma_j \tag{4.2}$$

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

Let N be the number of classes in the Markov Chain. Due to memory constraints, 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 4.1, which assumes that the target algorithm randomly selects its initial player.

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

Initialize $\vec{s}$[1..N] = NULL;

for $i = 1$ ***to*** M_{sample} **do**
  Uniformly randomly generate player PL;
  STR = eval(PL);
  if *size(*$\vec{s}[STR]) < \hat{\gamma}$ **then**
    $\vec{s}[STR] \leftarrow PL$;
  else
    Randomly replace player in $\vec{s}[STR]$ with PL with probability $\frac{\hat{\gamma}}{\hat{\gamma}+1}$;
  end
end

for $i = 1$ ***to*** N **do**
  Choose an unmarked $\vec{s}$[j] s.t. $size(\vec{s}[j]) \geq size(\vec{s}[k])$ for all unmarked $\vec{s}$[k];
  COUNT = 0; FOUND = false;
  while *COUNT* $< M_{pop}$ **do**
    Randomly select player PL from $\vec{s}$[j];
    Generate neighbourhood player PL' from PL;
    STR = eval(PL');
    if *size(*$\vec{s}[STR]) < \hat{\gamma}$ **then**
      $\vec{s}[STR] \leftarrow PL'$;
      FOUND = true;
    else
      Randomly replace player in $\vec{s}[STR]$ with PL' with probability $\frac{\hat{\gamma}}{\hat{\gamma}+1}$;
    end
    COUNT++;
    if *COUNT ==* M_{pop} *&& FOUND == true* **then**
      COUNT = 0; FOUND = false;
    end
  end
  Mark $\vec{s}$[j];
end

Algorithm 4.1: Populating the Classes

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.

There are two objectives to be met when populating the classes of the Markov Chain. First, we want the representative players to have some variety. This is why we begin with an initial population from M_{sample} randomly generated players rather than starting with only a single starting player. Second, we wish to fill up as many classes with as many players (up to $\hat{\gamma}$) as possible. In particular, we wish to have representatives from classes that the target algorithm is likely to generate. Hence, it is logical to make use of the neighbourhood function employed by our target algorithm to systematically generate players to populate the various classes. If the neighbourhood function is unable to generate players of a particular strength, we assume that the algorithm itself is unlikely to generate such players over the course of running the actual algorithm.

4.3 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. Obviously, the greater the number of games involved in the match, the more likely it is that the stronger player will triumph over the weaker one. However, the tradeoff for this added accuracy in the comparison function is an increase in computation time required for each game to be played.

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. This is done, as usual, using Monte Carlo simulations.

Using the population of generated players as given in section 4.2, 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 4.2.

```
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 ≻ PL'};
        end
        w_ij = WINS/M_wpm;
    end
end
```

Algorithm 4.2: Computing the Win Probability Matrix W

This method of computation is most suitable for non-deterministic games and/or players, where the result of different games between the same players may have different results. For deterministic players on deterministic games, there is a possibility that the random selection of opponents for a particular pair of classes may choose the same players multiple times. However, if the selection process is truly random, then such occurences should not affect the accuracy of the win probability matrix.

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, $\bar{w}_{ij} = 1 - w_{ji}$. This gives us considerable flexibility in devising the comparison function Q based on combinations of the results of several games. 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,

$$\begin{aligned} W_{ij}^{\geq 3(2/2)} &= ((1-\bar{w}_{ij}) \cdot \bar{w}_{ij} \cdot w_{ij}^2) + \\ &\quad (\bar{w}_{ij}^2 \cdot w_{ij} \cdot (1-w_{ij})) + \\ &\quad (\bar{w}_{ij}^2 \cdot w_{ij}^2) \end{aligned} \tag{4.3}$$

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. For a given comparison function Q with fixed values for x, y_1 and y_2, we denote the *comparison error of Q at states i and j* by

$$\delta_{ij} = \begin{cases} W_{ij}^{\geq x(y_1/y_2)} & i < j \\ W_{ji}^{\geq x(y_1/y_2)} & i > j \\ 0 & i = j \end{cases} \tag{4.4}$$

This value gives the probability that the comparison function makes an error by selecting the weaker player (i.e., the weaker player defeats the stronger player in their match). If $i < j$, then an error occurs if the player from class i beats the player from class j; similarly, if $i > j$, and error occurs if the player from class j is victorious. The comparison error is zero if both players i and j belong to the same class.

4.4 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 $i = 1$ ***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
  $\vec{\lambda_i}$[1..N] = $\vec{\lambda_i}$[1..N]/M_{nei};
end

Algorithm 4.3: Finding the Neighbourhood Distribution λ_i

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.

4.5 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 neighbourhood distribution provides the probabilities

that a prospective next state is chosen given the current state, while the win probability matrix allows the calculation of the probability that this prospective next state is retained. For many algorithms, this information is all that is necessary to derive its Markov Chain model.

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:

$$\begin{pmatrix} p_{11} & \lambda_{12}(1-\delta_{12}) & \lambda_{13}(1-\delta_{13}) & \cdots & \lambda_{1j}(1-\delta_{1j}) & \cdots & \lambda_{1N}(1-\delta_{1M}) \\ \lambda_{21}\delta_{21} & p_{22} & \lambda_{23}(1-\delta_{23}) & \cdots & \lambda_{2j}(1-\delta_{2j}) & \cdots & \lambda_{2N}(1-\delta_{2N}) \\ \lambda_{31}\delta_{31} & \lambda_{32}\delta_{32} & p_{33} & \cdots & \lambda_{3j}(1-\delta_{3j}) & \cdots & \lambda_{3N}(1-\delta_{3N}) \\ \vdots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ \lambda_{i1}\delta_{i1} & \lambda_{i2}\delta_{i2} & \lambda_{i3}\delta_{i3} & \cdots & p_{ii} & \cdots & \lambda_{iN}(1-\delta_{iN}) \\ \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ \lambda_{N1}\delta_{N1} & \lambda_{N2}\delta_{N2} & \lambda_{N3}\delta_{N3} & \cdots & \lambda_{Nj}\delta_{Nj} & \cdots & p_{NN} \end{pmatrix} \tag{4.5}$$

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

For algorithms of this type, the WPM and neighbourhood distribution functions are sufficient to derive its Markov Chain model. However, depending on the details of the algorithm, additional information may be required that may in turn require additional Monte Carlo simulations. For example, traditional annealing methods usually result in a non-homogenous Markov Chain, which may require an additional distribution to reflect the changes in the transition matrix in each iteration. We discuss this further in Section

13.4.

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. For certain special cases, some properties can be computed analytically rather than numerically. For example, Appendix B describes how the stationary distribution (which is the probability of system being in each state after it converges) can be computed analytically when the comparison error δ_{ij} is equal for all i and j. However, in the general case, a numerical analysis involving several matrix multiplications on P is required. The derivation of these properties and their utility is covered in the next chapter.

Chapter 5

Usefulness of Model

There are two basic underlying assumptions in the *M*^{2}*ICAL* method. The first is that all members of a particular class in the Markov Chain model have similiar neighbourhoods; this is affected by the number of classes N in the Markov Chain (and therefore the number of players represented by each class), which we discuss further in Section 12.1. The second assumption is that the various distributions constructed using the Monte Carlo simulations are representative of the workings of the algorithm in question, the accuracy of which can be improved by increasing the sample sizes involved. Although neither of these assumptions are generally true in the strictest sense, if we accept that the resultant model is a close enough approximation of the system that we wish to analyze, then we can estimate several useful properties of the algorithm by analyzing these distributions, using existing Markov Chain theory.

This chapter explains how certain interesting properties can be extracted from the Markov Chain representation of the algorithm, which gives an approximate measure of both its efficiency and its effectiveness. This information can help the practitioner to pinpoint possible weaknesses in the algorithm, and direct him towards possible avenues of improvement in the algorithm design.

5.1 Expected Player Strength

The first and arguably the most important property to discover about an algorithm is its expected solution quality after t iterations for a given value of t. After all, one of the aims of any algorithm must be to produce the highest quality solution possible. For the game-playing problem, this is equivalent to the *expected player strength* of the current player after t iterations.

Since the N states in the Markov Chain are ordered by ascending (estimated) player strength, the current state of the target algorithm at time t corresponds to the quality of the solution that the algorithm has produced at that time. Hence, assuming that the Markov Chain is an accurate representation of the target algorithm, its performance over time can be described by its expected state (which is equivalent to its expected player strength) after each iteration. To find the expected player strength of the algorithm, we begin with a probability vector $v_{\vec{(0)}}$ of size N, $v_{\vec{(0)}} = \{v_1, v_2, \cdots, v_N\}$ that contains in each element v_i the probability that the initial player will belong to class i, i.e., the probability that it will be of estimated strength $F(i)$. The values of $v_{\vec{(0)}}$ depends on how the algorithm chooses its initial state, and can usually be easily determined.

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

$$E(PS(PL^{(t)})) \;=\; \sum_{i=1}^{N} v_{\vec{(t)}}[i] \cdot F(i) \tag{5.1}$$

Algorithm 5.1 shows this process in pseudocode form, where the vector $\vec{v}$ stores the estimated player strength probabilities after every iteration.

Initialize $\vec{v}$[1..N]; EXPPS = 0;

for $i = 1$ ***to*** t **do**
  $\vec{v}[1..N] = \vec{v}^T P$;
end

for $i = 1$ ***to*** N **do**
  EXPPS += $\vec{v}[i].F(i)$;
end

return EXPPS;

Algorithm 5.1: Expected player strength after t iterations

The computation of the expected player strength after t iterations requires $t \cdot N^2$ floating point multiplications, which takes very little actual computation time. Therefore, it is feasible to compute the expected player strength for all values from 1 to t in order to observe the change in expected player strength over 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 the *M²ICAL* method to analyze imperfect comparison algorithms.

5.2 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.

Existing 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 [MT93]. These various concepts describe how a system depicted by a Markov Chain converges to a stationary distribution to varying degrees of strictness. However, the practitioner is often less interested in the theoretical definitions of system convergence, but is more concerned with the practical performance of the algorithm. For instance, an algorithm whose expected solution quality always increases by infinitesimal amounts does not by definition "converge", but the practitioner would be interested in knowing at what point in time does the algorithm's increase in solution quality become so small that further iterations of the algorithm will be of limited effect.

Hence, 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. Basically, we wish to detect the point in the algorithm where all the elements in the expected player strength vector in two successive iterations are identical up to k decimal places. To do so, we once again employ a straightforward numerical method by first instantiating 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. 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.

The level of strictness of the convergence is determined by the accuracy value k. For example, when $k = 3$, then Algorithm 5.2 gives the number of iterations required for all elements in two successive iterations of the probability vector $\vec{v}$ to be equal to a degree of 3 decimal places. Hence, additional iterations of the target algorithm past this point will change probability of the current solution being in any class by no more than 0.001%; consequently the expected solution quality should change by no more than 0.001% for each additional iteration beyond this point (and in fact, the actual change in solution quality should be considerably less).

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

Algorithm 5.2: Convergence to Stationary

Note that not all Markov Chains have steady state convergent behaviour. It is possible for an algorithm to exhibit periodic behaviour, and hence the Markov Chain representing it never converges to any particular state. For example, consider a game where the best strategy defeats all other strategies except the weakest one (which loses to all other strategies except the strongest one); a simple hill-climbing algorithm that searches for strategies in such a game would never converge. Such cases would cause Algorithm 5.2 above to run infinitely, although this is easily rectified by limiting the number of iterations. However, knowing that an algorithm never converges will also be useful to the practitioner since he can then make an informed decision on when to terminate the algorithm, especially when combined with a careful examination of the expected player strength to find the point where the player strength begins to decrease.

5.3 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:

$$\sigma^2 = \sum_{i=1}^{N} (\vec{v}[i] - \mu)^2 \cdot F(i) \tag{5.2}$$

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.

5.4 Summary

By modeling the algorithm using the M^2ICAL method, we can estimate three important aspects of the algorithm in question. Firstly, the expected solution quality after any number of iterations can be measured, which will give us an idea of the capabilities of the algorithm when implemented on the problem. Secondly, if the algorithm converges, then the number of iterations required for the system to converge to a given degree of accuracy can be found. This provides the practitioner with an indication of the number of iterations to run the algorithm, beyond which the expected amount of improvement to solution quality will be limited. Thirdly, the standard deviation of the algorithm can also be estimated, which shows the spread of the solutions. This gives a measure of the brittleness of the solutions generated by the algorithm, and tells us whether further runs using the same setup is likely to result in vastly stronger (or weaker) players.

Although the steps required to create this Markov Chain may seem involved and

complex, the underlying principle is quite simple. The purpose of the *M*^{2}*ICAL* method is to use Monte Carlo simulations to estimate the workings of the algorithm, and then to use Markov Chain theory to predict certain aspects of the algorithm given these estimations. In effect, this model is a type of trend predictor, which gives a projection of the algorithm's processes based on these estimations.

Chapter 6

Example: Modulo Nim

This chapter shows how the *M²ICAL* method can be used to examine the performance of a simple algorithm SCSA on a simple game-playing problem called Modulo Nim (ModNim). The purpose behind performing this case study on such a simple algorithm and game is twofold. Firstly, the simplicity of both the target algorithm and the target game problem allows us to explain the implementation of the *M²ICAL* model without having to handle possible extraneous factors that may be present in more complex instances. Secondly, this experiment represents a close to ideal setup for the *M²ICAL* method. 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".

It is worth reiterating that this case study is not meant to be a realistic example of a practical approach. We acknowledge that SCSA is not an algorithm that will be employed in many (if any) practical applications, and also that ModNim is a trivial strongly-solved game that elicits no more than passing interest to the intellectual games research community. However, we believe that the results that we have produced in this case study shows the potential of the *M²ICAL* method, and the accuracy of the resultant

model's predictions is an indication of the underlying soundness of our approach.

6.1 Modulo Nim

Modulo Nim, or *ModNim* for short, is a simple version of the game of Nim [BCG82]; it is a strongly solved win/loss game that has exactly one winning move in each position.

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. 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 $n \bmod (M+1) = 1$. Note that all games of ModNim(K,M) where $K \bmod (M+1) = 1$ are second player wins, and all other configurations are first player wins.

ModNim possesses several desirable traits for the purposes of this project. Firstly, the number of all possible ModNim players is determined by the values of K and M, which we can define to suit our needs. In particular, K defines the maximum length of a game and M determines the maximum branching factor in any position. There are exactly K unique positions and exactly $(K-M+1)M+(M-1)!$ unique players in ModNim(K,M). For our experiments, we use the game of ModNim(100,3) which has over 1.1×10^{47} unique players[1] and is a game-theoretic win for the first player. While even this game is considered small when compared to more popular games like chess and checkers, it is a large enough problem to show the *M²ICAL* model in a practical setting. Secondly, games of ModNim(K,M) have a maximum length of K moves and M choices per move, so they are fast to compute. Thirdly, since ModNim players can be

[1] 114,528,337,940,446,962,452,546,917,725,693,616,156,023,893,778 to be exact.

represented simply by a vector of K integers in the range of $[1..M]$, it is easy to generate players uniformly randomly.

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.

6.2 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)*, defined as follows.

Choose $s_{(0)}$ uniformly randomly from S;
for $t = 1$ ***to*** $MAXGEN$ **do**
  Choose s uniformly randomly from S;
  Let $s_{(t)}$ = $Q(s_{(t-1)}, s)$;
end
return $s_{(t)}$;

Algorithm 6.1: Simple Comparison Search Algorithm (SCSA)

SCSA begins by uniformly randomly choosing a solution $s_{(0)}$ from the solution space S. In each iteration t, a solution s is uniformly randomly selected from the entire solution space. SCSA then compares s with the solution found in the previous iteration using the comparison function Q; the solution favored by Q is retained and the other solution discarded. This continues until a number of generations equal to $MAXGEN$ is reached.

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.

We chose to examine SCSA for this study because despite its simplicity, it serves as the basis for more complex algorithms, and variants of SCSA have been used in practical games research, e.g., Pollack and Blair's hill-climbing backgammon player [PB98]. Furthermore, the simplicity of SCSA allows us to explain the Markov Chain model without having to handle the specifics of more complex algorithms.

6.3 Model Construction

For our ModNim experiments, the parameters chosen were set 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. Hence, the ModNim players will be divided into 100 different classes, where each class represents the set of players of a particular strength. The number of randomly generated opponents M_{opp} used to estimate the strength of each player was set at the value of $M_{opp} = 1000$.

Admittedly, these input parameters were not chosen based on any scientific analysis beyond trial and error. This set of experiments using SCSA on ModNim were performed to see if our initial concept of the *M²ICAL* method was at all viable, and due to the simplicity of both the algorithm and the game, we were able to set these parameters to reasonably high values while still maintaining an acceptable computation time. In fact, it is very likely that the parameter values used for this experiment were much higher than was required for an accurate model. For more complex algorithms and/or problems, a more systematic determination of appropriate input parameters may be necessary.

Since our ModNim(100,3) players can be conveniently represented using a 100-element vector, memory limitations are not as important in this experiment. Furthermore, since the mutation function for SCSA involves simply choosing a random player, the neighbourhood of SCSA is already captured by our initial step of populating the classes with M_{sample} randomly generated players. For these reasons, we decided to simply populate the classes with $M_{sample} = 10,000$ randomly generated players and retain all of them for the subsequent steps of the framework. In terms of Algorithm 4.1, the method used is equivalent to setting the parameters as $M_{sample} = 10,000$, $\hat{\gamma} = \infty$ and $M_{pop} = 0$.

As previously stated, this case study was our initial experiments to determine the viability of the *M*^{2}*ICAL* method. Given the specifications of SCSA and ModNim(100,3), it was not necessary at the time to populate the classes in a more complex way. It was only after we started working on analyzing a more complex algorithm (with a more complex neighbourhood function and memory constraints) that the systematic method of populating the classes given in Algorithm 4.1 was devised.

Since SCSA is a strict hill-climbing algorithm, we can make use of Equation (4.5) directly to represent the Markov Chain model of the system. In our version of SCSA, the comparison function Q returns the winner of one game of ModNim(100,3) where the incumbent is the first player. Therefore, the the error (δ_{ij}) distribution for all pairs of states i and j is identical to the WPM W. Hence we can find the δ_{ij} distribution via the direct application of Algorithm 4.2, such that $\delta_{ij} = W_{ij}$. The number of games played between every pair of classes M_{wpm} was arbitrarily set to a value of 1000.

Recall from Equation (4.5) that λ_{ij} is the probability that state j is chosen as the potential next state when the current state is i. For SCSA, since the challenger is uniformly randomly generated, the probability that the challenger is from class j corresponds to the proportion of players that belong to class j out of all possible players, i.e., $\lambda_{ij} = \frac{\gamma_j}{\Gamma_N}$.

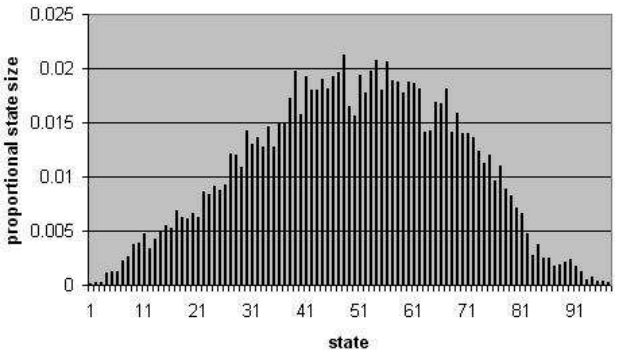

Figure 6.1: Sample state size ($\frac{\gamma_i}{\Gamma_N}$) distribution

Therefore, it is not necessary to employ Algorithm 4.3, which determines the neighbourhood distribution for each state in turn, although it will produce similar results.

To estimate the λ_{ij}-distribution from our population of M_{sample} of ModNim(100,3) players, we simply order the states in ascending estimated player strength, and then count the number of players in each state. These values are then divided by M_{sample} to give an estimated value for $\frac{\gamma_i}{\Gamma_N}$. Figure 6.1 shows the sample state size distribution of our experiment.

Because the distribution is derived from a sample of size M_{sample}, it may not be able to capture state size information if the number of players with that player strength is small ($< 1/M_{sample}$ of the entire solution space). For our sample, the 10,000 random players failed to produce any players from states 1, 99 and 100. We simply omit these states from our model, and apply it to the remaining 97 states instead.

Having determined both the δ_{ij}-distribution and the λ_{ij}-distribution, we simply substitute these values into Equation 4.5 to produce the transition matrix that represents the Markov Chain model of the system. This final step completes the application of the *M*^{2}*ICAL* method to SCSA on ModNim(100,3).

These experiments were performed on a Pentium-IV 1.6 GHz PC with 512MB RAM. It took approximately 24 hours for the entire process.

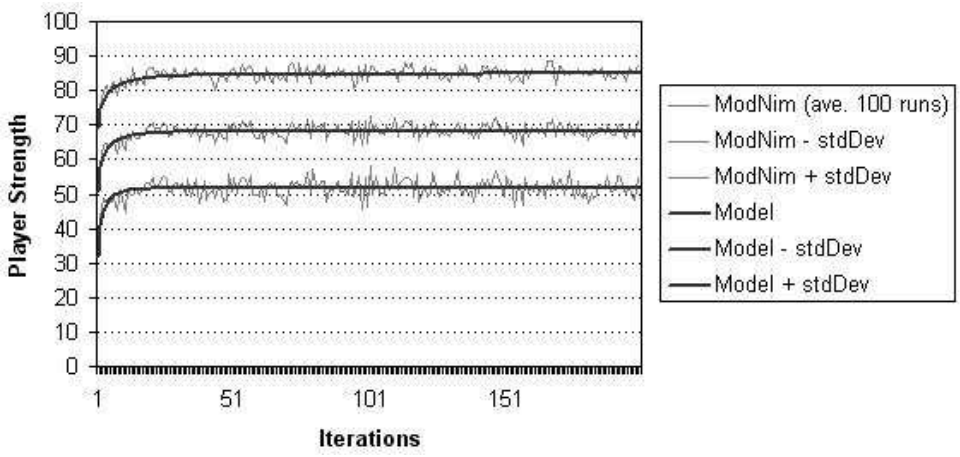

Figure 6.2: Model and experimental results for ModNim(100,3) using SCSA

6.4 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 6.2 gives the results averaged over the 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 5.2 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%. Considering the fact that several existing papers on intellectual games perform experiments for 1000 iterations or more (albeit with different techniques) [PB98, CKL+03], if SCSA is representative of typical algorithms in this problem domain, then this small time-to-convergence value suggests that such experiments could actually be terminated much sooner.

The simplicity of the game and algorithm allowed us to use a large sample population (10,000 players) as well as a large number of states (100), which reduced the inherent inaccuracy of the Monte Carlo simulations involved in the process. Furthermore, the neighbourhood function for SCSA is uniformly random, which is also easy to estimate using Monte Carlo simulations. These factors contributed to the high degree of accuracy of the resultant M^2ICAL model.

6.5 Summary

Our experiments with SCSA on ModNim(100,3) primarily serves as an example of how the M^2ICAL method functions. SCSA is the simplest possible imperfect comparison algorithm, and is unlikely to be a particularly effective algorithm for any practical problem. Similarly, ModNim is a fully solved game with a mathematically calculable optimal strategy, and our chosen explicit representation of ModNim strategy using a multi-element array is generally infeasible for practical games. Nonetheless, these experiments show that the Markov Chain model derived using the M^2ICAL method is able to provide highly accurate predictions on the performance of SCSA on ModNim(100,3).

Part III

Case Study: HC-Gammon

So far, we have been working with the very naive SCSA on the very simple game of ModNim. While SCSA is useful to show how the Markov Chain model works, it is almost never used in practical applications. This is because its selection criterion is completely random, making use of no expert knowledge other than that which is encompassed by the comparison function. Also, while ModNim is certainly useful for explaining how the model works, it is a strongly solved "toy" problem and not a practical game that is currently the target of computer games research.

In this part of the dissertation, we apply our model to the practical game of backgammon. Unlike ModNim, backgammon is a game that is very much the focus of current research. In particular, we will model an algorithm very close to the hill-climbing approach used by Pollack and Blair [PB98], which was used to critique the effectiveness of the temporal difference learning approach in the Master-level *TD-Gammon* backgammon program [Tes95]. For convenience, we will refer to the program generated by Pollack and Blair's technique as *HC-Gammon,* even though the authors themselves left the program nameless.

Chapter 7 introduces both the game of backgammon and the HC-Gammon experiment done by Pollack and Blair, which is the target problem for the M^2*ICAL* method in this part of the thesis. We show how the M^2*ICAL* method is adapted to HC-Gammon in Chapter 8. Two sets of experiments based on Pollack and Blair's work are performed; the first uses a random starting player and is described in Chapter 9, while the second uses an all-zero neural network starting player, and is described in Chapter 10. Finally, Chapter 11 summarizes both the results and the findings obtained from both sets of experiments.

Chapter 7

Introduction

7.1 Backgammon

Backgammon is believed to have originated in Mesopotamia in the Persian empire, and is the oldest known recorded game. It was derived from the game of *Senat*; gaming boards for Senat have been found in Egypt that date back to 3000-1788 BC. In terms of popularity, it is held in similar standing to chess, checkers and go. There is an annual World Backgammon Championship, which was last held in Monte Carlo, Monaco in July 2007. Backgammon is also one of the games involved in the annual Computer Olympiad, last held in Amsterdam, The Netherlands in June 2007, which allows the best game-playing programs in the world to compete in a single arena. The relevant rules for backgammon are reproduced here from [Kei96] in order to fix our terminology.

Backgammon is a game for two players, played on a board consisting of twenty-four narrow triangles called *points*. The triangles alternate in color and are grouped into four quadrants of six triangles each. The quadrants are referred to as a player's *home board* and *outer board*, and the opponent's home board and outer board respectively. The home and outer boards are separated from each other by a ridge down the center of the board called the *bar*.

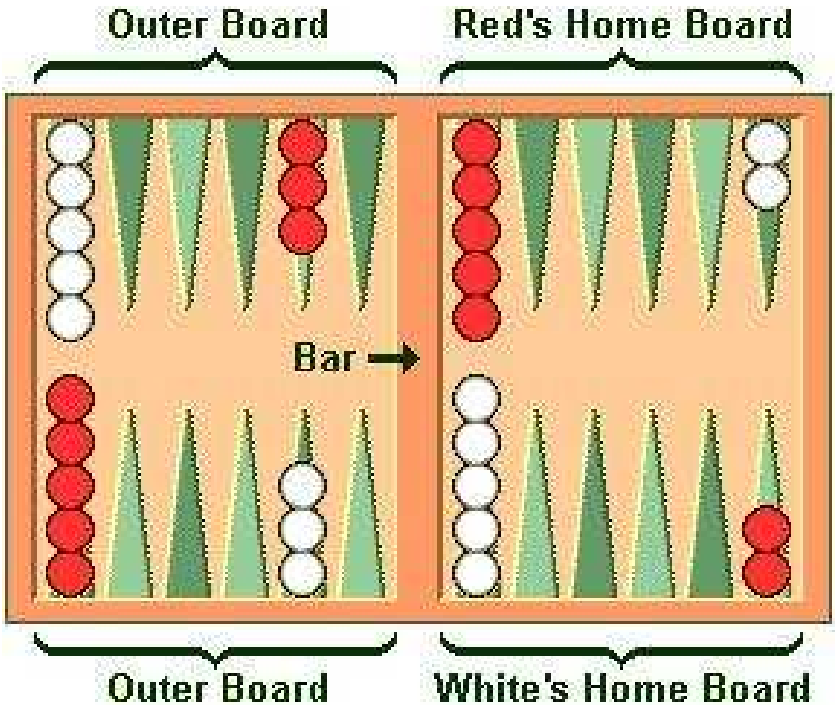

Figure 7.1: Starting position for backgammon

The points are numbered for either player starting in that player's home board. The outermost point is the twenty-four point, which is also the opponent's one point. Each player has fifteen checkers of his own color. The initial arrangement of checkers is: two on each player's twenty-four point, five on each player's thirteen point, three on each player's eight point, and five on each player's six point. See Figure 7.1.

The object of the game is move all your checkers into your own home board and then bear them off. The first player to bear off all of their checkers wins the game. To start the game, the first player throws two dice and moves his checkers according to the numbers showing on both dice; the players alternate turns. The roll of the dice indicates how many points, or *pips*, the player has to move his checkers. The checkers are always moved forward, to a lower-numbered point. Figure 7.2 shows the direction of movement for the player of the white checkers; red checkers move in the opposite direction.

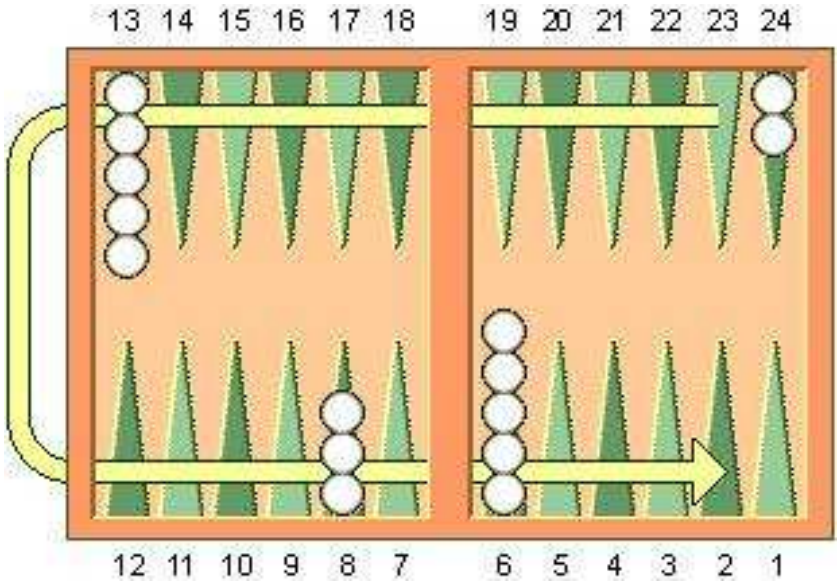

Figure 7.2: Direction of movement for white pieces

The following movement rules apply:

1. A checker may be moved only to an *open point*, one that is not occupied by two or more opposing checkers.

2. The numbers on the two dice constitute separate moves. For example, if a player rolls 5 and 3, he may move one checker five spaces to an open point and another checker three spaces to an open point, or he may move the one checker a total of eight spaces to an open point, but only if the intermediate point (either three or five spaces from the starting point) is also open. See Figure 7.3 for an example of how a roll of 5/3 can be played from the starting position.

3. A player who rolls doubles plays the numbers shown on the dice twice. For example, a roll of 6/6 means that the player has four sixes to use, and he may move any combination of checkers he feels appropriate to complete this requirement.

4. A player must use both numbers of a roll if this is legally possible (or all four

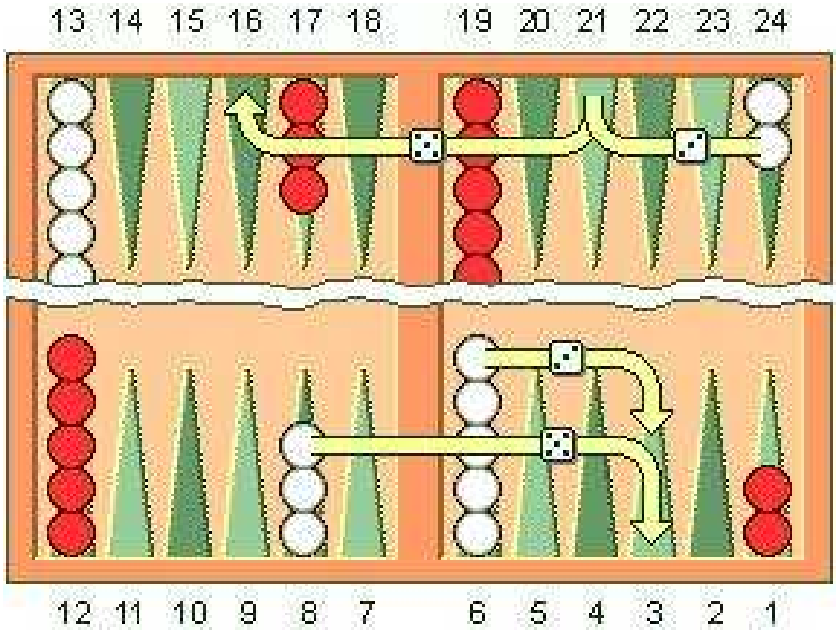

Figure 7.3: Two ways that White can play a roll of 5/3

numbers of a double). When only one number can be played, the player must play that number. If either number can be played but not both, the player must play the larger one. When neither number can be used, the player loses his turn. In the case of doubles, when not all four numbers can be played, the player must play as many numbers as he can.

A point occupied by a single checker of either color is called a *blot*. If an opposing checker lands on a blot, the blot is *hit* and placed on the *bar*. Any time a player has one or more checkers on the bar, his first obligation is to *enter* those checker(s) into the opposing home board. A checker is entered by moving it to an open point corresponding to one of the numbers on the rolled dice.

For example, if a player rolls 4/6, he may enter a checker onto either the opponent's four point or six point, so long as the prospective point is not occupied by two or more of the opponent's checkers. If neither of the points is open, the player loses his turn. If a

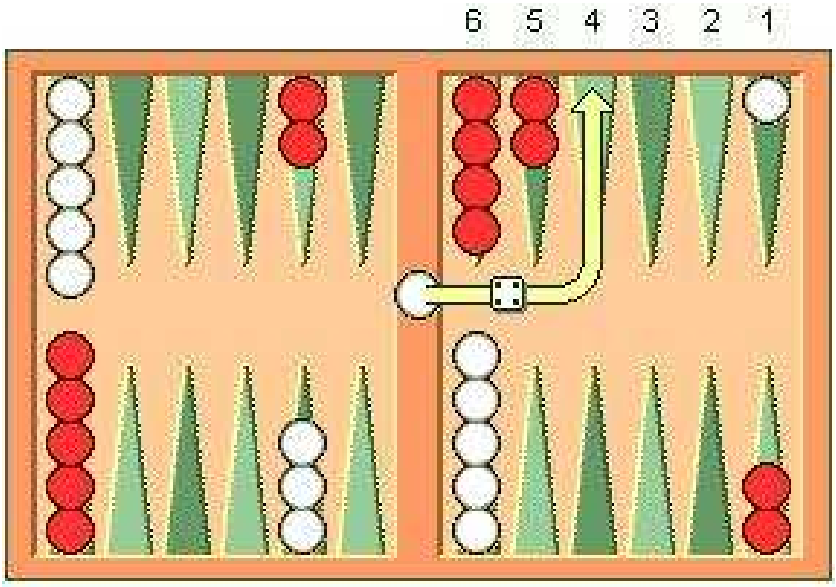

Figure 7.4: Entering from the bar

player is able to enter some but not all of his checkers, he must enter as many as he can and then forfeit the remainder of his turn. After the last of a player's checkers has been entered, any unused numbers on the dice must be played, by moving either the checker that was entered or a different checker. For example in Figure 7.4, if White rolls 4/6, he must enter the checker red's four points since Red's six point is not open.

Once a player has moved all of his fifteen checkers into his home board, he may commence *bearing off*. A player bears off a checker by rolling a number that corresponds to the point on which the checker resides, and then removing that checker from the board. Thus, rolling a 6 permits the player to remove a checker from the six point. If there is no checker on the point indicated by the roll, the player must make a legal move using a checker on a higher-numbered point. If there are no checkers on higher-numbered points, the player is permitted (and required) to remove a checker from the highest point on which one of his checkers resides. A player is under no obligation to bear off if he can make an otherwise legal move. Figure 7.5 show an instance where two

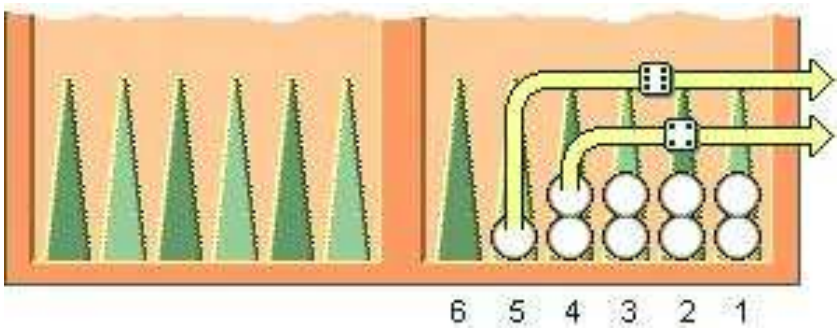

Figure 7.5: White bears off two checkers on a roll of 4/6

checkers bear off on a roll of 4/6.

A player must have all of his active checkers in his home board in order to bear off. If a checker is hit during the bear-off process, the player must bring that checker back to his home board before continuing to bear off. The first player to bear off all fifteen checkers wins the game.

7.2 HC-Gammon

In computer science, the greatest success in backgammon is undoubtedly Gerald Tesauro's *TD-Gammon* program [Tes95]. Using a straightforward version of Temporal Difference learning called TD(λ) on a neural network, TD-Gammon achieved Master-level play, which is far superior to any other backgammon program that had been created before. The latest version TD-Gammon 3.0 defeated Grandmaster Neil Kazaros in a match by +6 points in 20 games.

What makes this feat even more remarkable is the fact that TD(λ) is an unsupervised learning technique that learns through self-play, and the initial version of TD-Gammon was a strong player without the use of expert knowledge. Pollack and Blair [PB98] had the suspicion that the success of the temporal difference learning approach was not principally due to the power of the learning technique, but was also a function of the

mechanics of the game of backgammon. To test this hypothesis, they 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 supports 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.

In their paper, Pollack and Blair stated that they used several different experimental setups with varying levels of success, but they only reported the results for their best setup in detail. In this part of the thesis, we attempt to model what Pollack and Blair did in their various HC-Gammon experiments using the *M²ICAL* method, and analyze each of these setups to explain how they achieved their results. This would allow us to evaluate the validity of some of their claims.

7.3 Experimental Setup

Each backgammon player is represented as a standard fully-connected feedforward artificial neural network with one input layer, one hidden layer and a single output node using the sigmoid transition function. For each point on the backgammon board, 4 nodes represent the number of white checkers on that point. If there are 0, 1, 2 or 3 checkers on that point, then the first 0, 1, 2 or 3 nodes are given the value of 1, and the rest of the nodes are given the value 0. If there are three or more checkers on the point, then the first 3 nodes are given a value of 1, and the fourth node takes the value $(n-3)/2$ where n is the number of checkers on the point.

The four nodes for white and four nodes for red at each of the 24 points added up to 192 input nodes. In addition, there is one node for each side representing the number of checkers on the bar (taking the value of $n/2$ where n is the number of checkers on the

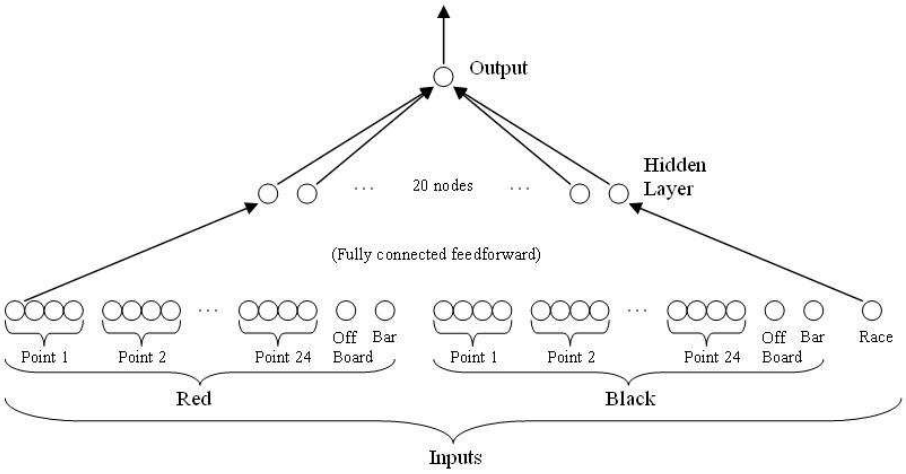

Figure 7.6: Artificial Neural Network architecture for HC-Gammon

bar), and another node for each side giving the number of checkers already successfully removed from the board (these took the value $n/15$ where n is the number of checkers already borne off). Finally, one additional unit indicated whether the game was in a *race* situation, i.e., all the checkers from both sides are past the opponent's checkers, and therefore hitting checkers is no longer possible. This gives a total of 197 input nodes.

The input nodes are fully connected to a 20-node hidden layer, which are then connected to a single output node. This makes a total of 3980 weights; Figure 7.6 shows this architecture. The value returned by the output node provides an evaluation of the desirability of the input position. The backgammon player chooses its move in a given current position by evaluating the resulting positions from all possible moves for the given dice roll, and then choosing the move that leads to the position with the highest evaluation. In effect, the players perform a one-move lookahead. The initial player had all weights set to 0.0.

Using this neural network architecture to represent backgammon players, Pollack and Blair implemented and tested 3 different hill-climbing algorithms, which we call Experiments 1 to 3. In Experiment 1, Pollack and Blair used an approach that is similar

to SCSA, except that instead of uniformly randomly generating the challenger in each iteration, the challenger is derived from the current player via a *mutation function*, where gaussian noise is added to the neural network weights of the current player to produce the challenger. In their paper [PB98], 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. To implement the mutation function, 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:

$$\frac{\sqrt{\sum_i (w_i' - w_i)^2}}{\sqrt{3980}} = 0.05$$

$$k = 0.05 \cdot \sqrt{\frac{3980}{\sum_i x_i^2}} \tag{7.1}$$

Since the game of backgammon uses dice rolls to determine the legal moves in each position, a common method of comparing two computer backgammon players is to allow them to play 2 games against each other, one as first player and one as second, using the same sequence of dice rolls (or *dice streams*). In Experiment 1, if the new player (the "challenger") defeats the original player (the "incumbent") in 3 out of 4 games, i.e., two pairs of games using two different dice streams, then the challenger is deemed to be victorious. However, instead of replacing the incumbent directly with the victorious challenger, Pollack and Blair altered the algorithm to retain most of the traits of the incumbent while adjusting the weights towards the challenger's using the *descendent function*:

$$\text{new incumbent} = 0.95 \cdot \text{old incumbent} + 0.05 \cdot \text{challenger} \tag{7.2}$$

We call this descendent function a *95% Inheritance* function, since the new champion inherits 95% of the incumbent's weights.

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, implementing a simple form of simulated annealing (see Section 13.4). 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.

7.4 SCSA on Backgammon

To gain some insight on the effect of analyzing a complex game like backgammon using the *M*^{2}*ICAL* method, we analyzed SCSA (Algorithm 6.2) on backgammon. For this experiment, the comparison function Q used by SCSA involves playing the incumbent player and the challenger player in a 2-game match using identical dice streams with different starting players, and the incumbent player is replaced only if the new player beats it in *both* games. We constrain the weights and biases of the neural network to a range of [-2.0, 2.0] to approximate the domain of all possible backgammon players, and the challenger is generated by uniformly randomly selecting values for all 3980 weights of the neural network within this range.

Since games of backgammon take much longer to complete than ModNim, we only generate 1000 random players to estimate our distributions (rather than the 10,000 that

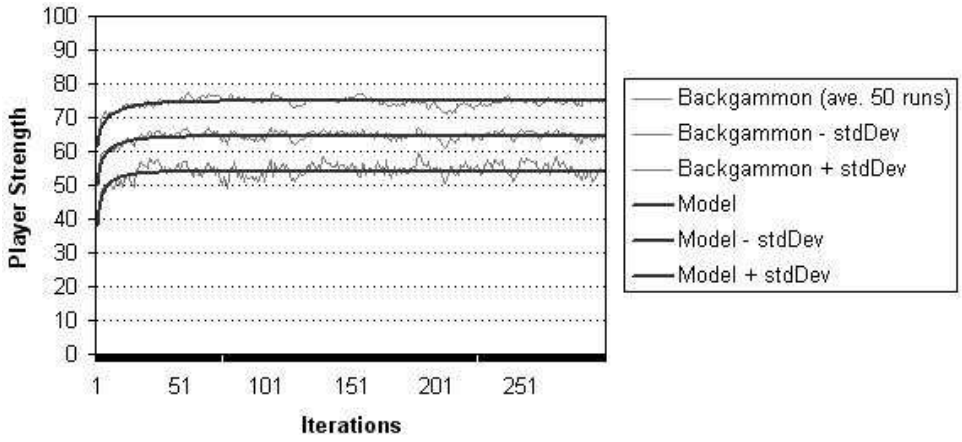

Figure 7.7: Model and experimental results for SCSA on backgammon

we generated for ModNim), and use only 100 players playing 2 games each to evaluate player strength (rather than 1000 for ModNim). Our 1000 random players only occupied 66 out of 100 classes, so the unrepresented 34 classes were discarded. The remainder of the experimental setup is identical to our ModNim(100,3) experiments.

Figure 7.7 shows the average results of 50 runs of SCSA on backgammon, and the expected solution quality and spread forecast by the M^2ICAL model. Once again, the values predicted by the model are given by the bold black lines, and the ones provided by 50 runs of SCSA on backgammon are given by the grey lines. Although we ran the algorithm up to 1000 iterations, only the values for the first 300 iterations are provided here for greater clarity since the remaining iterations show a similar trend.

In this instance, the model accurately predicts an expected solution quality of 64.531% compared to the average value of the actual runs that fluctuated within 61% and 67%. The standard deviation calculated by the model was $\pm$10.2510%, which was also borne out by the sample standard deviation achieved by our 50 runs of SCSA. The standard deviation for backgammon is significantly smaller than for ModNim; the fact that only 66 classes were represented may have a bearing on this, but since the players used in our sample population were generated using the method employed by the algorithm itself,

the prediction remained accurate. Hence, according to the model, SCSA on backgammon produces less volatile results than for ModNim(100,3); if the strengths of backgammon players follows a normal distribution, then the model shows that about 68% of all players will fall within a range of 20.5% of all player strengths when generated using SCSA.

Finally, Algorithm 5.2 reveals that SCSA on backgammon converges to a stationary solution to a degree of accuracy of 3 decimal places in 272 iterations (and to 2 decimal places in 203 iterations). Despite the added complexity of backgammon compared to ModNim, the number of iterations required before SCSA converges is still surprisingly low. This suggests that convergence rate is less a function of the target problem, and more a function of the algorithm.

Backgammon remains a very popular subject for current computer science research on games, and there is no disputing that backgammon is a much more complex game than ModNim. These results show that the M^2ICAL model also gives very accurate predictions when analysing SCSA on backgammon with players represented by artificial neural networks. Therefore, there is evidence to show that the M^2ICAL method is able cater to complex games and data representations without losing much (if any) accuracy in its predictive capabilities.

Chapter 8

Model Construction

Compared to our previous ModNim(100,3) experiments, HC-Gammon is significantly more complex in both the target problem and the imperfect comparison algorithm for analysis. Unlike ModNim, backgammon is a non-deterministic game since a player's possible moves are dictated by dice rolls. To cater for the effect of lucky dice rolls on the results of games between two players, the comparison function used by Pollack and Blair involves multiple pairs of games using identical dice streams. Furthermore, games of backgammon take much longer to complete than ModNim, so computation time becomes an important factor.

The HC-Gammon generation algorithm is a hill-climbing algorithm that is not strict. Even though the algorithm does not change the incumbent player unless it loses to its opponent in a match, the resultant player may not necessarily be stronger than the incumbent because it is the offspring of the two players (and not the winning player itself). Therefore, we cannot separate the values of the transition matrix for this system into neighbourhood (λ_{ij}) and error (δ_{ij}) components like we did for SCSA. Instead, the transition matrix must be found in a more complex way.

This chapter describes how we can use the *M²ICAL* method to analyze HC-Gammon despite the added complexity of both the problem and the algorithm.

8.1 Determining Input Parameters

Compared to ModNim, each game of backgammon requires significantly more time to complete. For ModNim(K,M), each game can only last for a maximum of $K-1$ moves, but backgammon games can potentially last for 200 moves or more, which is frequently the case early in the algorithm when the generated players are poor. Furthermore, the branching factor of backgammon is estimated to be about 20 for each possible dice roll, while ModNim(K,M)'s branching factor is M. Finally, it takes longer for the neural network implementation of the backgammon players to evaluate each position than for our ModNim implementation, which simply returns a move using what is essentially a table look-up.

In practical terms, the added computation time for backgammon games forces us to carefully determine the various input parameters for our Monte Carlo generation of the Markov Chain probability distributions. While it is clear that larger sample sizes (and therefore more sample games) will in general provide more accurate results, there is a definite trade-off in terms of computation time. Therefore, we must set the various input parameters in a way that minimizes computation time while still retaining an acceptable degree of accuracy in our results.

The first major task is to decide on the number of classes N in our Markov Chain. The more classes we use in our Markov Chain, the finer the granularity of the results achieved, but this is at the expense of added computation time. We were able to perform our ModNim experiments using 100 classes, but when we attempted to run the model for backgammon using the same parameters, we found that the estimated amount of time required to generate the values for the model might be as much as a year. Therefore, we decided to construct the Markov Chain using only $N = 10$ classes; while using only 10 classes certainly reduces the granularity of the results, we felt that it was sufficient for us to observe the general properties of HC-Gammon, and still provide interesting insights to the system.

One of the most common operations in our Markov Chain generation process involves evaluating the strength of a player by playing it against M_{opp} uniformly randomly generated players. This operation is required both while populating the classes and when deriving the neighbourhood distribution. Therefore, the value of M_{opp} is vital to the overall running time of the entire process. We eventually decided on a value of $M_{opp} = 100$, which provides an approximately 90% confidence interval if the results of a player against randomly generated opponents is normally distributed with a mean equal to its actual strength. Further details of how this value is derived is given in Section 12.2.

The values for the remaining parameters in the model generation process were influenced by the necessity of minimizing running time, but were ultimately decided arbitrarily. During the population of the classes, we set the maximum class size $\hat{\gamma}$ at 20, and started with an initial population of randomly generated players of $M_{sample} = N \cdot \hat{\gamma} = 200$. We set the number of additional sample population players generated in each iteration $M_{pop} = 100$.

Whenever we wish to find a distribution of values across the $N = 10$ classes, it is necessary to generate a sufficiently large number of instances in order to discover the distribution. In our experiments, we decided that 200 sample points would give an adequate reflection of the various distributions. Therefore, to generate the winning probability matrix W, the number of games between each pair of classes is set at $M_{wpm} = 200$. We also set the number of challengers generated M_{cha} to derive the challenger probability distribution C_i, and the number of descendents generated M_{des} to derive the descendent probability distribution D_{ij} to be both 200; these distributions were required to derive the neighbourhood distribution for HC-Gammon.

With these values, we were able to use the *M²ICAL* method to generate the Markov Chain representation of HC-Gammon on a Pentium-IV 1.6 GHz PC with 512MB RAM in about 3 weeks in total.

8.2 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 memory space constraints and within a reasonable amount of computation time. Ideally, we wish to have at least one representative from each class. Recall from our experiments on backgammon using SCSA in Section 7.4 that uniformly randomly selecting 1000 backgammon players only managed to generate players from 66 out of 100 classes. This is obviously less than ideal, but in our examination of SCSA, a better method of generating players was not available to us. However in the case of HC-Gammon we have another tool at our disposal, namely the neighbourhood function employed by Pollack and Blair; we can make use of this neighbourhood function to systematically generate players in order to populate the classes.

Using Algorithm 4.1, we populated the 10 classes of our Markov Chain. We began with $M_{sample} = 200$ ANNs with the weights and biases uniformly randomly determined from the range of [-2.0, 2.0] to approximate the domain of all possible backgammon players. For each class we made use of the mutation function to generate challenger players, and if the challenger player defeats the parent, then a descendent player is generated for the purpose of populating the classes (further details on the mutation and descendent functions are given in Section 9.1). In this way, we systematically generate additional players from each class until no further players from a class with fewer than $\hat{\gamma} = 20$ players is generated after $M_{pop} = 100$ attempts. Populating the classes took approximately 50 hours to complete.

8.3 Comparison Function Generalization

When comparing the relative strengths of two players, the HC-Gammon comparison function plays them against each other in a match consisting of multiple sets of 2 games.

Each 2-game set comprises one game as first player and one game as second player using the same dice stream for both games. Pollack and Blair tried several different comparison functions based on this principle at different stages of their algorithms (3 wins out of 4, 5 wins out of 6, etc.). We are able to handle all of these different comparison functions by generating the win probability matrix W as detailed in Section 4.3.

Using the population of generated players as given in Section 8.2, 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 $1_{s \succ s'}/M_{wpm}$, as given in Algorithm 4.2. This process was completed in under 24 hours.

8.4 Neighbourhood Distribution

The general matrix given in Equation (4.5) applies to strict hill-climbing algorithms, where the next state changes only if it is superior to the current state. While the HC-Gammon method is also hill-climbing, it is not strict since the player retained for the next state is not the winning player in the match but a separate player generated by combining the two players. In evolutionary computing parlance, this is a type of crossover operation that creates a descendent from two parents. Therefore, we cannot use Equation (4.5) directly. Instead, we must discover two separate distributions and combine them in order to find what is essentially the neighbourhood function for this system.

For each class i in turn, we first generate the *challenger probability distribution* C_i, 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 our usual Monte Carlo simulation method by creating $M_{cha} = 200$ challengers this way, evaluating each of them, and then estimating the overall probability distribution using this sample. We store all of the generated challengers (which are separate from our initial

population found in Section 4.2) 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. This part of the process is synonymous to Algorithm 4.3.

Next, for every non-empty vector $\vec{c}_j$ after the creation of M_{cha} challengers, 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 once again perform a Monte Carlo simulation by randomly selecting a parent-challenger pair PL from class i and PL' from class j, and then creating a descendent from the crossover of PL and PL' and evaluating 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. The pseudocode for the generation of the descendent probability distribution D_{ij} is given in Algorithm 8.1.

For each class, M_{cha} challengers are generated and evaluated, which takes $O(M_{cha}M_{opp})$ time. Then for each pair of classes, M_{des} descendents are generated and evaluated, which takes $O(M_{des}M_{opp})$ time; in total, the N classes will require $O((N^2M_{des} + N \cdot M_{cha})M_{opp})$ time to run. Assuming that $M_{des} = M_{cha} = M_{opp} = O(N)$, then the entire algorithm runs in $O(N^4)$ time. Since the classes are considered in turn, only one set of generated challengers needs to be retained at any one time. Therefore, the algorithm requires $O(M_{cha})$ space for the purpose of storing interim players. This was the most time-consuming step in the generation of our Markov Chain model, requiring about 15 days to complete.

8.5 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

for $i = 1$ ***to*** N **do**
  for $j = 1$ ***to*** M_{cha} **do**
    Randomly select player PL from $\vec{s}$[i];
    Generate challenger player PL' from PL;
    STR = eval(PL');
    $\vec{c}_{STR} \leftarrow PL'$;
  end
  for $j = 1$ ***to*** N **do**
    Initialize $\vec{d_{ij}}$[1..N] = 0.0;
    if $\vec{c}_j$ *is non-empty* **then**
      for $k = 1$ ***to*** M_{des} **do**
        Randomly select player PL' from $\vec{c}_j$;
        PL = $PL'.parent$;
        Generate mutant player PL'' from PL and PL';
        STR = eval(PL'');
        $\vec{d_{ij}}$[STR]++;
      end
    end
    for $k = 1$ ***to*** N **do**
      $\vec{d_{ij}}$[k] = $\vec{d_{ij}}$[k]/M_{des};
    end
  end
  for $j = 1$ ***to*** N **do**
    Clear $\vec{c}_j$;
  end
end

Algorithm 8.1: Finding the Descendent Probability Distributions D_{ij}

three experiments. Each value p_{ik} in the transition matrix is the sum of the product of three sets of values over all values of j from 1 to N: (1) the probability that a challenger from class j is generated from a class i incumbent, given by $c_i(j)$; (2) the probability that a descendent from class k is generated from parents of classes i and j, given by $d_{ij}(k)$; (3) and the relevant winning probability of a class i player over a class j player depending on the comparison function.

For instance, if the comparison function returns the challenger only if it beats the incumbent at least 3 out of 4 games, then

$$p_{ik} = \sum_j c_i(j) \cdot W_{ji}^{\geq 3(2/2)} \cdot d_{ij}(k) \tag{8.1}$$

where

$$\begin{aligned} W_{ji}^{\geq 3(2/2)} = & ((1-\bar{w}_{ji}) \cdot \bar{w}_{ji} \cdot w_{ji}^2) + \\ & (\bar{w}_{ji}^2 \cdot w_{ji} \cdot (1-w_{ji})) + \\ & (\bar{w}_{ji}^2 \cdot w_{ji}^2) \end{aligned} \tag{8.2}$$

In this manner, we are able to compute all the values of p_{ik} for the transition matrix P representing the Markov Chain model. This completes the implementation of the *M*^{2}*ICAL* method on HC-Gammon.

Chapter 9

Experiments A: Random Initial Player

This chapter presents our first attempts at analyzing the HC-Gammon experiments conducted by Pollack and Blair. The main difference between these experiments and HC-Gammon is our assumption that the initial player is a randomly-selected artificial neural network, while Pollack and Blair began with a neural network with all weights and biases set to zero. There are two reasons for this modification. Firstly, beginning with a single fixed player presents the M^2ICAL method with certain problems that require additional changes to the general formulation (see Chapter 10 for further details). Secondly, it was our original belief that the identity of the initial player should have minimal effect on the overall performance of the algorithm, especially since most existing research on unsupervised machine learning techniques on intellectual games begin with randomly generated players, e.g., [CF01, KW01, LM01].

In this chapter, we describe in detail how the M^2ICAL method can be used to generate a Markov Chain representation of the HC-Gammon generation algorithm that uses a randomly generated initial player, and present the results produced by the model. The modelling of the algorithm that begins with an all-zero neural network initial player, which was the scheme employed by Pollack and Blair for HC-Gammon, is described in Chapter 10.

9.1 Exp A1: Inheritance

The very first algorithm attempted by Pollack and Blair was identical to SCSA, except that it made use of the mutation function given in Equation (7.1) to generate challengers rather than uniformly randomly selecting them; the challenger replaces the incumbent if it wins at least 3 out of 4 games in their match (i.e., it is a strict hill-climbing algorithm[1]). Pollack and Blair observed that this setup *"...worked reasonably well. The networks so evolved improved rapidly at first, but then sank into mediocrity"* [PB98]. They believed that this effect may be due to the possibility of a much weaker challenger defeating a stronger incumbent in their once-off match, which they quaintly termed the "Buster Douglas Effect" (named after the 45-1 underdog heavyweight boxer who defeated overwhelming favourite Mike Tyson to become World Heavyweight Champion), which is synonymous with a comparison error in our terminology. To address this possibility, instead of replacing the incumbent with the challenger, they used the descendent function given in Equation (7.2) to generate the incumbent for the next iteration.

When populating the classes using the *M²ICAL* method to model this experiment, we managed to generate 20 players from classes 1 to 9 (for a total of 180 players), but were unable to generate any players from the highest class. We therefore constructed the Markov Chain with only 9 classes. Furthermore, by using the combination of a mutation and a descendent function to determine the next incumbent player, this algorithm is no longer a *strict* hill-climbing algorithm because the new player may not be superior to the previous incumbent when evaluated by the comparison function. Hence, we cannot make use of the transition matrix for strict hill-climbing algorithms given by Equation (4.5), but must instead construct the transition matrix for the Markov Chain in the way explained in Section 8.4.

Figure 9.1 shows the comparison between the results predicted by our model and the

[1]To model this experiment, we can directly apply the generic transition matrix for strict hill-climbing algorithms given in Equation (4.5), where $\lambda_{ij} = C_i(j)$ and $\delta_{ij} = W_{ij}^{\geq 3(2/2)}$.

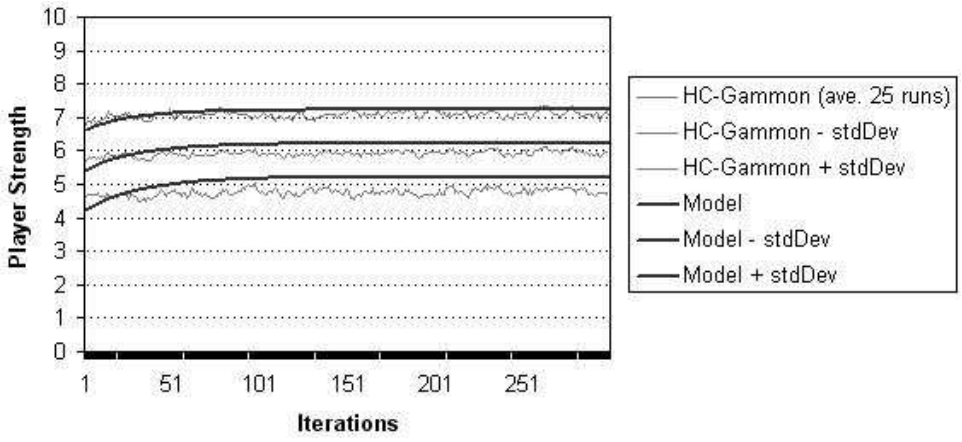

Figure 9.1: Model and experimental results for HC-Gammon 95% Inheritance on backgammon

values found by actually running the algorithm 25 times to 1000 generations (only the values for the first 300 iterations are shown here; the remaining 700 iterations follow the same trend). As usual, the estimated player strength (and $\pm$ the standard deviation) predicted by the model are given by the bold black line, and the corresponding values for the actual runs are given by the grey line.

A visual inspection of these results reveals that although the values predicted by the model are not as accurate as for our previous experiments on SCSA on both Mod-Nim(100,3) and backgammon, it still provides a reasonable estimate of the values achieved by the algorithm. 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%. This predicted value is higher than 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 a little smaller than the sample standard deviation of the 25 runs of between 10.2% and 12.5%.

Therefore, our model slightly overestimated the expected solution quality and slightly

underestimated the standard deviation of the algorithm. We believe that this slight inaccuracy is due to the fact that we only used $N = 10$ classes in our Markov Chain, and even so the discrepancy in the expected player strength was only about 1%. The small number of classes used in the model will also tend to underestimate the standard deviation since the players are divided into only a few classes, which allows less variation in player strengths.

It is interesting to note that the expected player strength for using the 95% Inheritance neighbourhood function is actually quite poor. The predicted value of 62.25% means that the generated player is not much better than average, and in fact is inferior to the player resulting from SCSA described in Section 7.4 (which has an expected player strength of 64.531%)! This is completely contradictory to the reported result that a player generated in this way is able to compete with the *PUBEVAL* program. Although Pollack and Blair ran this algorithm for 100,000 iterations, which is far in excess of the 288 iterations that the model states is required in order for the expected player strength to converge to an accuracy of 5 decimal places, our model suggests that the additional iterations of the algorithm cannot account for such a dramatically higher playing standard.

9.2 Exp A2: Fixed Annealing Schedule

Pollack and Blair noticed in Experiment 1 that counter-intuitively, even though networks in later generations are supposedly stronger, the number of challengers able to defeat the incumbent did not decrease. They surmised that this is because the challengers derived from the incumbent possess a similar strategy, and therefore due to the small number of games used to determine the superior player, there is a high possibility that the challenger will emerge victorious in a match. Therefore, the second algorithm they examined followed an "annealing schedule" such that after 10,000 generations the challenger must win 5 out of 6 games (rather than 3 out of 4); and after 70,000 the challenger must win

7 out of 8. The values 10,000 and 70,000 were chosen after observing the frequency of successful challengers in their best run of Experiment 1.

The player evolved using this second algorithm was the strongest player achieved. However, when this algorithm was re-run a further 9 times, all of the subsequent players generated were poor. Nonetheless, the strongest conclusions drawn by Pollack and Blair on the capabilities of temporal difference learning on the game of backgammon were based on the capabilities of the initial strongest player generated.

We can easily customize the Markov Chain model to handle this fixed annealing schedule. Essentially, the Markov Chain representing this fixed annealing schedule makes use of three separate transition matrices. From iterations 1 to 10,000, the model is identical to Exp A1, using the transition matrix P calculated using the comparison function $W_{ji}^{\geq 3(2/2)}$ as given in Equation (8.1); from iterations 10,001 to 70,000, the transition matrix used is calculated using $W_{ji}^{\geq 5(3/3)}$, which we call P'; finally, the transition matrix P'' calculated using $W_{ji}^{\geq 7(4/4)}$ is used for iterations 70,001 onwards. Explicitly, the transition matrices P' and P'' are:

$$p'_{ik} = \sum_j c_i(j) \cdot W_{ji}^{\geq 5(3/3)} \cdot d_{ij}(k) \tag{9.1}$$

where

$$\begin{aligned} W_{ji}^{\geq 5(3/3)} = & ((1 - \bar{w}_{ji}) \cdot \bar{w}_{ji}^2 \cdot w_{ji}^3) + \\ & (\bar{w}_{ji}^3 \cdot w_{ji}^2 \cdot (1 - w_{ji})) + \\ & (\bar{w}_{ji}^3 \cdot w_{ji}^3) \end{aligned} \tag{9.2}$$

and

$$p''_{ik} = \sum_j c_i(j) \cdot W_{ji}^{\geq 7(4/4)} \cdot d_{ij}(k) \tag{9.3}$$

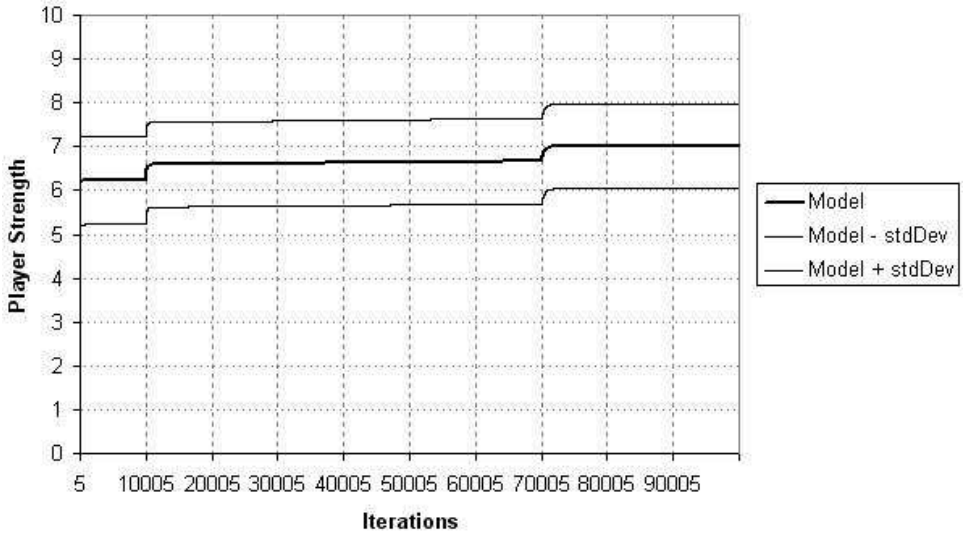

Figure 9.2: Model results for fixed annealing schedule HC-Gammon 95% Inheritance on backgammon

where

$$
\begin{aligned}
W_{ji}^{\geq 7(4/4)} &= ((1-\bar{w}_{ji}) \cdot \bar{w}_{ji}^{3} \cdot w_{ji}^{4}) + \\
&\quad (\bar{w}_{ji}^{4} \cdot w_{ji}^{3} \cdot (1-w_{ji})) + \\
&\quad (\bar{w}_{ji}^{4} \cdot w_{ji}^{4}) \qquad (9.4)
\end{aligned}
$$

Figure 9.2 shows the expected player strength, along with the addition and subtraction of the standard deviation, that was predicted by the model; we are unable to compare these predictions with the average results of several actual runs of the algorithm because using Monte Carlo simulations to evaluate the strength of the generated players up to 100,000 iterations would take too much computation time (multiple days for each run depending on the frequency of the Monte Carlo player strength evaluations).

The model produces the expected prediction: the expected player strength is the same as for Experiment A1 up to 10,000 iterations, which uses the transiton matrix P. At

this point, the algorithm switches to the best-5-out-of-6 comparison function captured by transition matrix P', causing the expected player strength to increase by approximately 0.3 classes within 250 iterations before rapidly stabilizing (although the expected player strength continues to increase at a very slow rate). At 70,000 iterations, the comparison function switches to the best-7-out-of-8 scheme, represented by the transition matrix P''. The expected solution quality once again increases, by about 0.25 classes, before showing signs of stabilization although it takes around 1000 iterations for this to happen.

If the model's prediction is accurate, then several interesting comments can be made about this algorithm. Firstly, recall that the expected strength of the produced player converged to 62.25% after about 288 iterations. After 100,000 iterations, the expected strength was about 70.01%, an increase of less that 8% (less that one player class). Considering the amount of time required to run the algorithm up to 100,000 iterations and the small increase in strength as a result, it can be argued that it would be preferable to spend the computation time elsewhere. Secondly, the increase in strength does not occur in a gradual manner, but is instead "stepped" at the points where the comparison functions anneal, i.e., at iterations 10,000 and 70,000. Since the model predicts that the expected strength starts to converge after about 288 iterations (and 853 iterations after the first change change in the comparison function), perhaps it would be more efficient to set the annealing points closer to these values. Thirdly, the standard deviation predicted by the model remains roughly around 10% to 12% throughout. This suggests that it might be wiser to run the algorithm multiple times for fewer iterations rather than to run it a few times for a seemingly excessive 100,000 iterations.

However, the model does not tell the whole story. In particular, the model does not take into account the computation time required to perform the stricter comparison functions since the results are given in terms of iterations. Therefore, although the results suggest that we could continuously increase the annealing by using ever stricter comparison functions (e.g., best-9-out-of-10 and so on), this would eventually become

infeasible due to the added computation time required to play the increased number of games. Nonetheless, the model does show that Pollack and Blair's arbitrarily determined value of 100,000 iterations for the experiment seems severely excessive.

9.3 Exp A3: Dynamic Annealing Schedule

When the algorithm using the fixed annealing schedule at 10,000 and 70,000 generations was re-run a further 9 times, all of the subsequent players generated were poor. The authors discovered that the annealing schedule that was chosen based on observing the traits of the first run did not transfer well to the other runs, which had different challenger success frequencies. Therefore, they implemented a third algorithm, which increased the number of games required for the challenger to win when the challenger success rate exceeded 15% over the last 1000 generations. This algorithm was run 10 times, and *"all ten players evolved under this regime were competitive"* [PB98], although none of them were superior to the one generated by the hand-tuned player in the first run of Experiment 2.

The Markov Chain model for this dynamic annealing schedule makes use of the same three transition matrices P, P' and P'' as in Exp A2, but in a more complex way. From iterations 1 to 1000, the model is identical to the model in Exp A1, which is represented by the transition matrix P. Beyond iteration 1000, the transition matrix at iteration t for the Markov Chain model, $P_{(t)}$, is a composite of P, P' and P'', weighted by the probability that the algorithm is using the 5-out-of-6 or the 7-out-of-8 comparison function.

Let κ be the number of challengers that must replace the incumbent in the last Λ iterations before the next, more stringent, comparison function is employed; 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 the transition matrices P, P' and P'' respectively. We know that when $1 \leq t \leq \Lambda$, $\alpha_{(t)} = 1.0$ while $\alpha'_{(t)}$ and

$\alpha''_{(t)}$ are both equal to zero. Similarly, between iterations $\Lambda + 1$ and 2Λ, $\alpha_{(t)}$ and $\alpha'_{(t)}$ are non-zero and sum up to 1, while $\alpha''_{(t)}$ remains at zero, and all three values should be non-zero after iteration 2Λ. Furthermore, $\alpha_{(t)} = \alpha'_{(t)} = \alpha''_{(t)} = 0$ when $t \leq 0$.

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, which is given by the probability distribution vector for the previous iteration $v_{(t-1)}[i]$. Then, knowing the probability that the incumbent produces a challenger from class j, which is given by the mutant or challeger function $c_i(j)$, we can find the probability that the challenger wins using the appropriate winning probability W_{ji} (given in Equations (8.2), (9.2) and (9.4)) by summing these values over all combinations of i and j.

$$\begin{aligned} a_{(t)} &= \alpha_{(t-1)} \cdot \left(\sum_{i=1}^{N} \sum_{j=1}^{N} v_{(t-1)}[i] \cdot c_i(j) \cdot W_{ji}^{\geq 3(2/2)} \right) + \\ &\quad \alpha'_{(t-1)} \cdot \left(\sum_{i=1}^{N} \sum_{j=1}^{N} v_{(t-1)}[i] \cdot c_i(j) \cdot W_{ji}^{\geq 5(3/3)} \right) + \\ &\quad \alpha''_{(t-1)} \cdot \left(\sum_{i=1}^{N} \sum_{j=1}^{N} v_{(t-1)}[i] \cdot c_i(j) \cdot W_{ji}^{\geq 7(4/4)} \right) \\ &= \sum_{i=1}^{N} \sum_{j=1}^{N} v_{(t-1)}[i] \cdot c_i(j) \cdot \\ &\quad (\alpha_{(t)} \cdot W_{ji}^{\geq 3(2/2)} + \alpha'_{(t)} \cdot W_{ji}^{\geq 5(3/3)} + \alpha''_{(t)} \cdot W_{ji}^{\geq 7(4/4)}) \end{aligned} \tag{9.5}$$

Note that $a_{(t)} = 0$ when $t \leq 0$.

Let $b_{(t)}^{k/l}$ be the probability that exactly k challengers were victorious between iterations $t - l + 1$ and t inclusive, i.e., in the last l iterations. For convenience, we define $b_{(t)}^{k/l} = 0$ when $t \leq 1$. Obviously, if $k > l$ then $b_{(t)}^{k/l} = 0$. Furthermore, if $l > t$ then $b_{(t)}^{k/l} = b_{(t)}^{k/t}$. Therefore, we can assume without loss of generality that $l \leq t$ in the following construction.

Observe that $b_{(t)}^{0/1} = 1 - a_{(t)}$, since the probability that no challengers win in the first iteration out of the last 1 iteration is the probability that the challenger did not win in

that iteration. By the same token, $b_{(t)}^{1/1} = a_{(t)}$.

For $t-l+1 < s \leq t$, we find that the probability of exactly k victorious challengers in iteration t is the probability that there were $k-1$ victorious challengers in the last $l-1$ iterations and the challenger is victorious in iteration t, plus the probability that there were k victorious challengers in the last $l-1$ iterations and the challenger fails to defeat the incumbent in iteration t. Hence,

$$b_{(s)}^{k/l} = b_{(s-1)}^{k-1/l-1} \cdot (1 - a_{(s)}) + b_{(s-1)}^{k/l-1} \cdot a_{(s)} \tag{9.6}$$

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

This suggests a method of finding $b_{(t)}^{k/l}$ algorithmically, assuming that the values for $a_{(s)}$, $t-l+1 \leq s \leq t$ are known. To do so, in each iteration we first compute the values of $b_{(t-l+1)}^{0/1}$ and $b_{(t-l+1)}^{1/1}$. These values allow us to compute $b_{(t-l+2)}^{0/2}$, $b_{(t-l+2)}^{1/2}$ and $b_{(t-l+2)}^{2/2}$, which in turn gives us sufficient information to compute the relevant b values for iteration $(t-l+3)$, and so on, until we reach iteration t, where all values from $b_{(t)}^{0/l}$ to $b_{(t)}^{k/l}$ is computed. Note that for $t-l+k \leq s \leq t$, we are only required to compute the values of $b_{(s)}^{0/l}$ to $b_{(s)}^{k/l}$ inclusive. By substituting the values of κ and Λ into k and l respectively into the above formulation, we can find the values of $b_{(t)}^{0/\Lambda}$ to $b_{(t)}^{\kappa/\Lambda}$ inclusive.

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:

$$\beta_{(t)}^{k/l} = \sum_{i=k}^{l} b_{(t)}^{i/l} \tag{9.7}$$

$$= 1 - \sum_{i=0}^{k-1} b_{(t)}^{i/l} \tag{9.8}$$

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

$$\alpha_{(t)} = \begin{cases} 0 & t \leq 0 \\ 1 & 0 < t < l \\ \alpha_{(t-1)} - \alpha_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} & t \geq l \end{cases} \tag{9.9}$$

$$\alpha'_{(t)} = \begin{cases} 0 & t < l \\ 1 - \alpha_{(t)} & 0 < t < l \\ \alpha'_{(t-1)} + \alpha_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} - \alpha'_{(t-1)} \cdot \beta_{(t)}^{\kappa/\Lambda} & t \geq l \end{cases} \tag{9.10}$$

$$\alpha''_{(t)} = \begin{cases} 0 & t < 2l \\ 1 - \alpha_{(t)} - \alpha'_{(t))} & 0 < t < l \end{cases} \tag{9.11}$$

$$\tag{9.12}$$

Obviously, this formulation can be generalized to the cases when the number of possible annealing steps is greater than 3. Once the α values are computed for a particular iteration t, then the transition matrix for that iteration $P_{(t)}$ is simply:

$$P_{(t)} = (\alpha_{(t)} \cdot P) + (\alpha'_{(t)} \cdot P') + (\alpha''_{(t)} \cdot P'') \tag{9.13}$$

The algorithm for numerically computing the transition matrix for each iteration t first computes the value of $a_{(t)}$, which is a function of $\alpha_{(t-1)}$, $\alpha'_{(t-1)}$ and $\alpha''_{(t-1)}$. It then computes the values of $b_{(t)}^{0/\Lambda}$ to $b_{(t)}^{\kappa/\Lambda}$, using the stored values of $a_{(t-l+1)}$ to $a_{(t)}$. This enables the determination of $\beta_{(t)}$, which in turn provides sufficient information for the computation of $\alpha_{(t)}$, $\alpha'_{(t)}$ and $\alpha''_{(t)}$. Since we know that $\alpha_{(1)} = 1.0$ and $\alpha'_{(1)} = \alpha''_{(1)} = 0.0$, we are able to compute $P_{(t)}$ for every iteration t. The pseudocode for this algorithm is given in Algorithm 9.1.

Unfortunately, the algorithm performance predicted by the model is entirely different from the behaviour reported by Pollack and Blair. According to the model, the probability that annealing occurs at iteration 1000, given by $\alpha'_{(1000)}$, is close to 1.0, i.e., it is almost certain that at least 150 out of the first 1000 challengers would be victorious. In

Initialise $\alpha = 1.0, \alpha' = 0.0, \alpha'' = 0.0$;
Initialize $\kappa = 150, \Lambda = 1000$;
for $t = 1$ ***to*** *MAX_ITERATIONS* **do**
  Compute a using Equation (9.5);
  for $l = 1$ ***to*** $min(t, \Lambda)$ **do**
    for $k = min(l, \kappa)$ ***downto*** 0 **do**
      Compute b_k corresponding to $b^{k/l}_{(t-\Lambda+l)}$ using Equation (9.6);
    end
  end
  Compute $\beta^{\kappa/\Lambda}_{(t)}$ using Equation (9.7);
  Compute α, α' and α'' using Equation (9.9);
  Compute $P_{(t)}$ using Equation (9.13);
end

Algorithm 9.1: Finding the transition matrix $P_{(t)}$

contrast, the probability that a second annealing occurs, given by $\alpha''_{(t)}$, remains close to 0.0 throughout that 100,000 iterations of the model. This produces the results shown in Figure 9.3, where a sharp increase in player strength is observed up to iteration 1100 or so, whereupon the increase in strength becomes very small over the remainder of the algorithm. Indeed, actual runs of the algorithm bears out this prediction; all 25 runs of the algorithm switched to the 5-out-of-6 comparison function at iteration 1000 and never employs the 7-out-of-8 comparison function.

The high value of $\alpha'_{(1000)}$ is easily explained. Early in the algorithm, the randomly chosen initial player is not a particularly strong backgammon player. As a result, there is a high probability that it would produce a descendent that can beat it in a 3-out-of-4 match. In fact, if the incumbent and the challenger are of similar strength, where the probability of one beating the other in any given game is 50%, then the probability of the challenger beating the incumbent 3 out of 4 games is 5/16 = 31.25%, more than twice

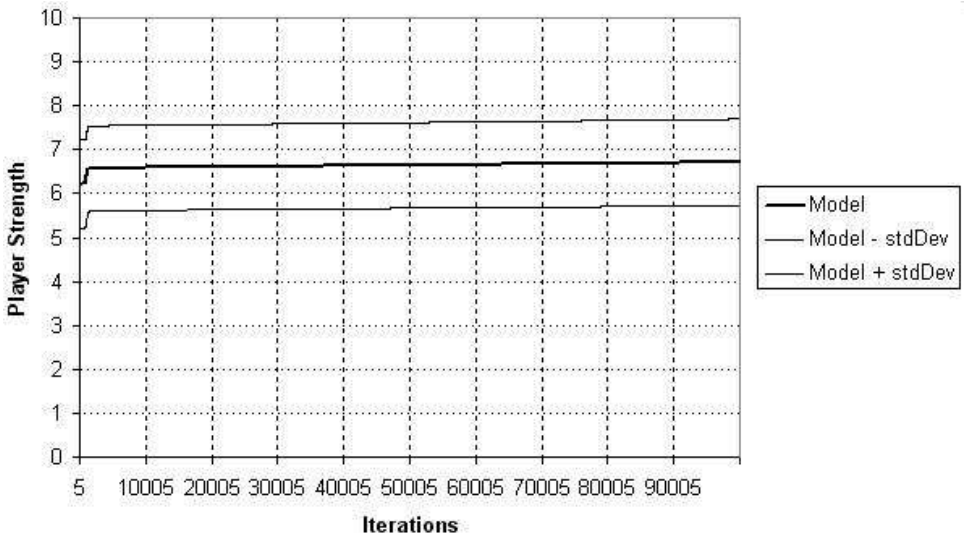

Figure 9.3: Model results for dynamic annealing schedule HC-Gammon 95% Inheritance on backgammon

the required probability of 15% for a annealing to occur. The low value of $\alpha''_{(t)}$ can also be explained in this manner: if the incumbent and the challenger are of similar strength, the probability that the challenger is victorious in at least 5 out of 6 games is 7/64 $\approx$ 10.9%, significantly lower than the required 15% probability. Of course, this analysis is not exact since the descendent produced may be of greater or lesser strength than the incumbent, but it illustrates the probabilities involved in this process.

Although it was not explicitly stated by the authors, it can be surmised that this behaviour was not observed in HC-Gammon. Certainly, we can assume that when the authors implemented this dynamic annealing scheme, the 7-out-of-8 comparison function was employed in at least some of the runs of their algorithm. Furthermore, none of the players generated using any of our configurations were able to defeat *PUBEVAL* over 10% of the time. It is apparent that there is some element in the experimental setup of HC-Gammon that has not been implemented in the same way in our experiments.

Chapter 10

Experiments B: All-zero Initial Player

10.1 Single Initial Player

The experimental results given in the previous chapter represented the expected performance of the HC-Gammon algorithm when the initial player is a neural network with randomly determined neuron and synapse weights. Both the results predicted by the M^2ICAL model and the values achieved by actual runs of the algorithm were rather poor, and the generated players were significantly weaker than reported by Pollack and Blair. Furthermore, we did not encounter the counterintuitive phenomenon of an increase in challenger success when the player strength increases as reported by Pollack and Blair. The most likely factor causing these disparities in the results is the fact that while we used a randomly generated neural network as our initial player, Pollack and Blair used as the initial player a neural network with all its initial neuron and synapse weights set to zero.

When such an *all-zero neural network* (AZNN) is used to evaluate backgammon positions, it would return an identical value for all positions (namely zero). The move chosen by a player using the AZNN as its evaluation function therefore depends on how the implementation of the player chooses between positions of equal value; this was not

discussed by Pollack and Blair in their paper. In our experiments, we order the moves in descending order of the origin point for the first die, and then the second die. e.g., on a roll of 4/1 by White, we first consider if it is legal to move a checker from point 24 to 20, then another checker from 24 to 23; then we consider moving from point 24 to 20 followed by 23 to 22, and so on. The ordering of the moves is similar for Red, except that it is in ascending order. Hence, an AZNN player will play the first legal move found in this manner.

While we do not know if Pollack and Blair implemented their move ordering function in the same way, we found that the estimated player strength of our AZNN player is between 7 and 8 (out of 10 classes). This is significantly higher than the average estimated player strength of randomly generated neural network players, which was between 5 and 6. It transpired that using the AZNN player as the starting point for the algorithm accounts for at least some of the disparity in the strength of the player generated by our experiments and HC-Gammon, since the AZNN player is generally of a superior strength than a randomly generated neural network player.

This presented a problem for our approach. By using Monte Carlo simulations to estimate the various performance aspects of an algorithm, the Markov Chain model is able to provide a prediction on the performance of the algorithm in the *average case*. However, by starting the experiment using a single fixed initial player (the AZNN player), the algorithm always begins its search in a very local neighbourhood. One straightforward way to implement our approach is to populate the classes using an initial population of $M_{sample} = 1$, namely the AZNN player. Figure 10.1 shows the comparison between the average results achieved by 25 runs of HC-Gammon using the AZNN player as the initial solution, and the predictions given by the Markov Chain model implemented in this way.

A visual inspection of the graph shows that the Markov Chain model provides a very poor prediction of the algorithm's performance. The expected player strength predicted

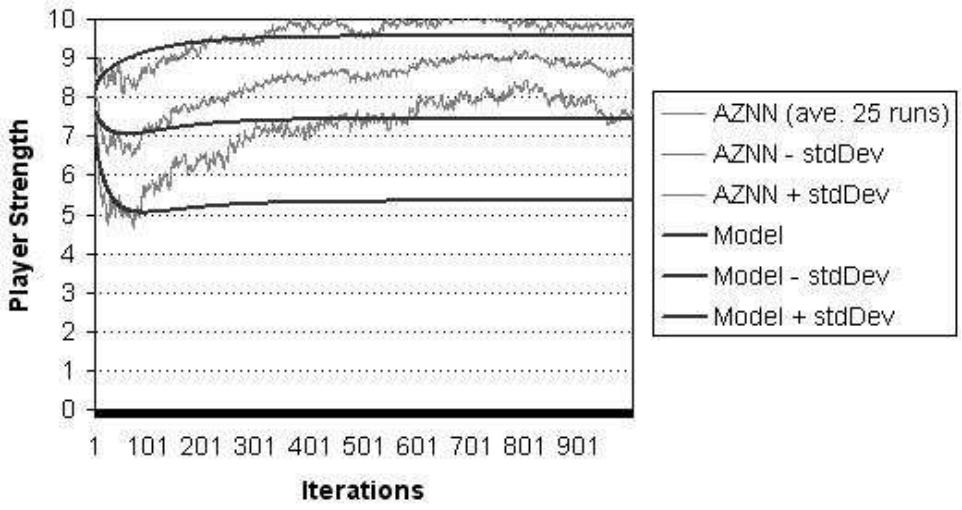

Figure 10.1: Model and experimental results for HC-Gammon using the AZNN initial player

by the model begins at a value between 7 and 8, which corresponds to the estimated strength of the AZNN player, then falls to just above 7 before increasing again to a consistent value just below 7.5. In contrast, the results achieved by the 25 actual runs of HC-Gammon also first experiences a drop in average player strength (to values between 6.5 and 7) for about 100 iterations; then the results improve until around iteration 400 when the average player strength fluctuates between 8.5 and 9.

Although the model was able to capture the algorithm's initial drop in player strength, its prediction drastically underestimates the performance of the algorithm in the long term. This is not an unexpected result: by populating the classes using players that stem from the single player (i.e., the AZNN player), the representative sample of players upon which the Markov Chain model is based is composed largely of players from the immediate neighbourhood of a single initial player. Therefore, the probability distributions derived by the M^2ICAL method from this population will reflect the properties of the immediate neighbourhood of the AZNN player. As the actual runs of HC-Gammon showed, the algorithm at the neighbourhood of the AZNN player first experiences a

drop in player strength before rising again (when the algorithm moves away from the immediate neighbourhood of the AZNN player). However, since a large proportion of the players in the sample belongs to the local neighbourhood of the AZNN player, the *M*^{2}*ICAL* model can only predict the initial fall in player strength, but not the subsequent rise.

10.2 Exp B1: Inheritance

One way to address this issue is to populate the classes with players that are sufficiently far removed from the initial player so that the effect of a particular local neighbourhood is alleviated. To do so, 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 of the algorithm. At this point, we use the current players of the 200 runs as the initial sample for populating the classes. We call this process of running the algorithm until a sufficient number of replacements has occurred *introducing a time-lag*. Interestingly, we were able to populate all 10 classes with $\hat{\gamma} = 20$ players for a full sample population of 200 players using this initial sample.

Figure 10.2 shows the predictions given by the time-lag *M*^{2}*ICAL* model, compared to the same 25 runs of HC-Gammon; the values for the model begin at iteration 48. With this modification, the model is able to provide a much more accurate prediction of the expected player strength of HC-Gammon. 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%. However, the model predicts that the standard deviation of the player strength will be about 1.8 classes, overestimating

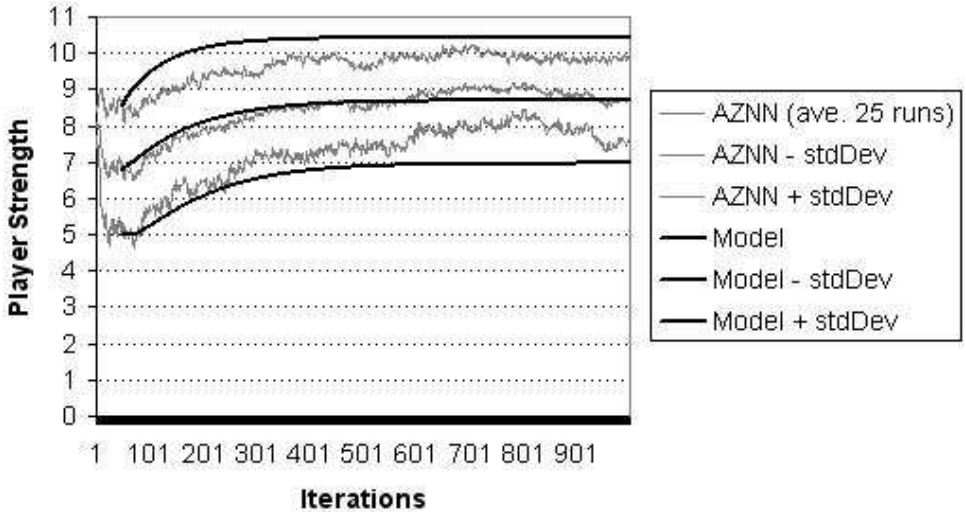

Figure 10.2: Time-Lag Model and experimental results for HC-Gammon using the AZNN initial player

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.

Several interesting observations can be made. Firstly, both the model's prediction and the average results of the actual runs confirm that the players generated by beginning with the AZNN player as the initial player are much stronger than if the initial player was randomly generated. In fact, the strength of the player is expected to be almost in the top 87% of all possible players. Secondly, since the highest (10th) class falls within one standard deviation of the expected player strength, then 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

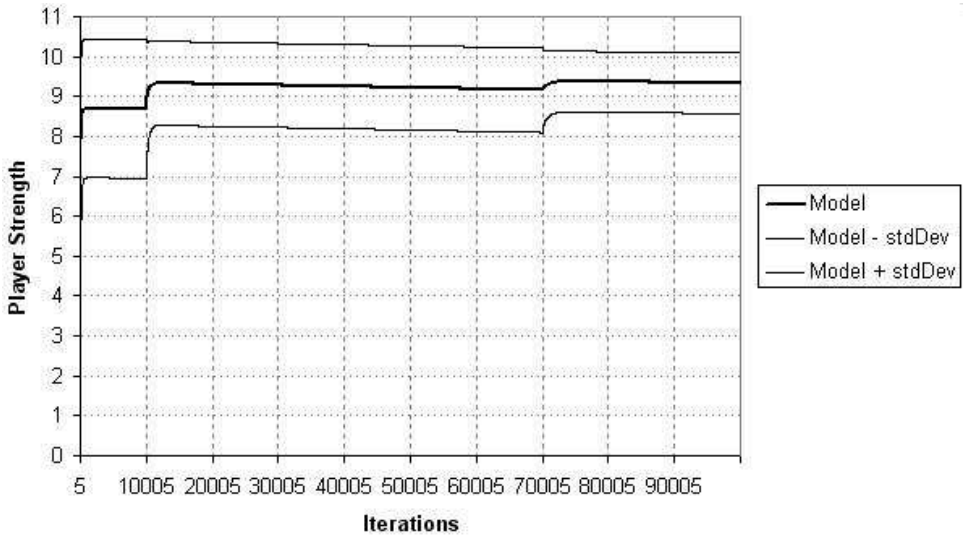

Figure 10.3: Time-Lag Model for fixed annealing schedule HC-Gammon using the AZNN initial player

PUBEVAL or TD-Gammon, who are probably in the top 1% or better of all possible players.

10.3 Exp B2: Fixed Annealing Schedule

When the annealing schedule is fixed such that the challenger is required to beat the incumbent in 5 out of 6 matches after iteration 10,000 and 7 out of 8 matches after iteration 70,000, the time-lag M^2*ICAL* model prediction of its expected player strength is given in Figure 10.3. As expected, the results are similar to the corresponding experiment with a randomly chosen initial player, with distinct "steps" in the expected player strengths after the annealing points (see Section 9.2). Although the expected player strength of the algorithm as predicted by the model changes throughout, it converged to a degree of accuracy of 5 decimal places at iteration 1,050, it also converged to 5 decimal places 2,302 iterations after the 5-out-of-6 comparison function is used.

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.

Obviously, this model is not completely accurate, and we cannot claim that further iterations of the algorithm beyond a certain optimal point will definitely result in reduced playing strength based on this experiment alone. This observation may be due to the inherent inaccuracies involved in estimating distributions using Monte Carlo simulations, combined with the small number of classes we used in this model. However, this model does highlight the fact that such a danger is present, and it is possible that not only is running an algorithm for extremely large numbers of iterations not significantly beneficial to the generated program's performance, it could be detrimental to it. Such a phenomenon occurs when the probability of generating a superior descendent is offset by the probability that an inferior descendent can defeat (and therefore replace) its superior parent.

10.4 Exp B3: 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, and then perform the same procedure as given in Experiment A3 to generate the Markov Chain model for this algorithm. 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 10.4, which is in

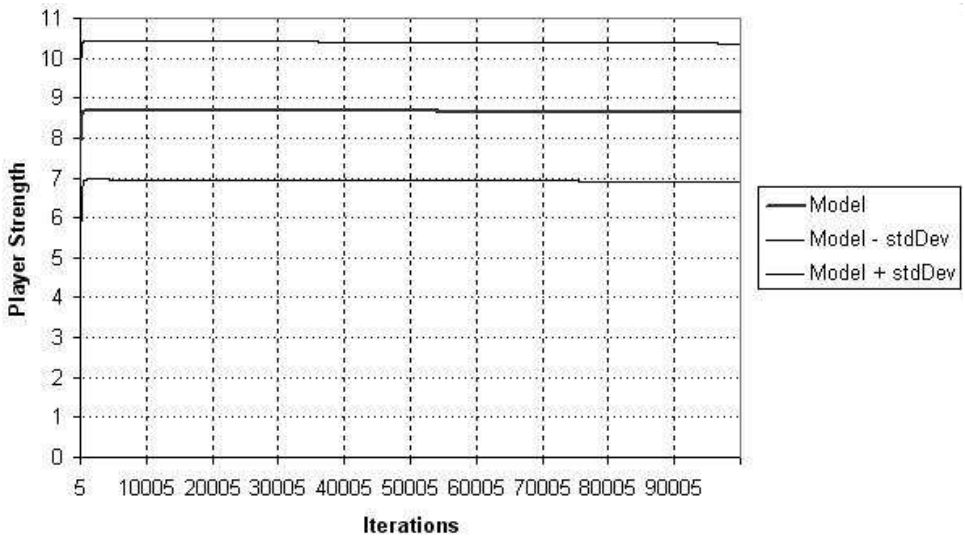

Figure 10.4: Time-Lag Model for dynamic annealing schedule HC-Gammon using the AZNN initial player

effect almost exactly the same as the results for Experiment A1 (Figure 10.2), extended to 100,000 iterations.

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 chapter 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 different from the original HC-Gammon implementation. 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

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

Chapter 11

Summary

This part of the thesis described two sets of experiments, both of which were meant to emulate the algorithm used by Pollack and Blair in their generation of the HC-Gammon backgammon player. In this section, we summarize our findings from these experiments, and comment on how they affect the conclusions drawn by Pollack and Blair in their paper.

11.1 Usefulness of model

Although we were unable to reproduce HC-Gammon's results due to our inability to duplicate their experimental setup precisely, we can use the algorithm that we *have* implemented as an example of how the M^2ICAL method can be useful for the evaluation and refinement of algorithms. We have shown that the expected player strength of the algorithm starting from the AZNN as the initial player should be in the 90th percentile of all possible players; while this seems impressive, it is likely that the set of "interesting" backgammon players (i.e., players that are able to play backgammon with some measure of "intelligence") could belong to the top 99th percentile, 99.9th percentile or even higher. In any case, the standard deviation of the results given by the model shows

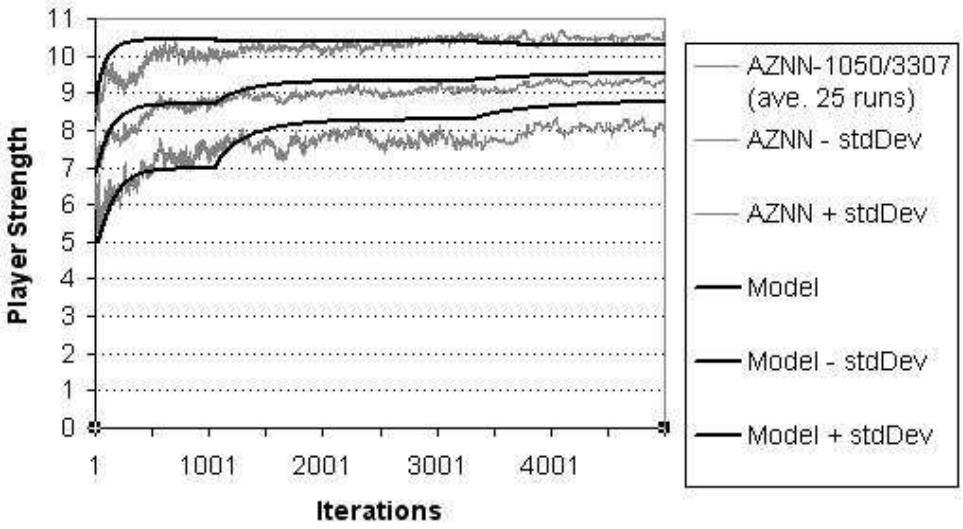

Figure 11.1: Time-Lag Model for annealing at 1050 and 3307 iterations using the AZNN initial player

that this algorithm is at least theoretically capable of producing players of this level. Unfortunately, we are unable to provide more precise results using our 10-class model.

The most obvious improvement to the algorithm is the number of iterations run. The M^2ICAL model suggests that running the algorithm to 100,000 iterations is needlessly excessive, and in fact the annealing points, where the requirements for the challenger to displace the incumbent are increased, should be placed closer to the points of convergence. Figure 11.1 shows the comparison between the model's prediction of the expected player strength and the average of 25 actual runs of a fixed annealing setup using the AZNN initial player, where the annealing points are set at iterations 1,050 and 3,307 respectively; these are the points of convergence to 5 decimal places predicted by the model in Experiment A2. We find that the expected player strength after 5,000 iterations is at 95.31%, which is higher than the 93.29% achieved after 100,000 iterations in the original fixed annealing setup (with annealing points at 10,000 and 70,000 iterations). These values are once again borne out by the results obtained from the average

of 25 runs of the actual algorithm. Hence, we are able to produce stronger players on average in 1/20 of the time, simply by choosing the annealing points more carefully.

11.2 Comments on HC-Gammon

Most of the conclusions made by Pollack and Blair on the favourable characteristics of backgammon to self-learning and its effect on the evaluation of TD-Gammon's temporal difference learning approach are 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.

Our first set of experiments assumed that the initial player was a randomly generated ANN. While the M^2ICAL method was able to capture the workings of this algorithm, the results obtained were much poorer than those reported by Pollack and Blair. The main difference between our experiment and the original HC-Gammon was the fact that the authors used an all-zero neural network as their initial player, a difference that turns out to be crucial to the performance of the algorithm. This presented a problem for the M^2ICAL method since Monte Carlo simulations are poor at capturing the long-term effects of an algorithm that stems from a very localised neighbourhood. We resolved this problem by introducing a time-lag from the start of the algorithm before commencing our Monte Carlo simulations, which allowed the algorithm to move sufficiently far enough away from its initial local neighbourhood to generate an effective estimation. This algorithm produced significantly stronger players than the first set of experiments, although it was still not as strong as the ones reported by Pollack and Blair. We can only assume that the discrepancy in player strength is due to some differences in our experimental setups, possibly in the mutation function and/or the move ordering function.

The drastic difference between the strengths of the players generated using a randomly generated player rather than the AZNN as the initial player is somewhat startling.

This seemingly minor change resulted players that were almost 3 classes (or 30% of all possible players) weaker. The fact that the AZNN is itself in the 80% percentile of all players helps the algorithm to find better solutions, since the starting neighbourhood of the algorithm is already of a reasonably high quality. Nonetheless, this fact reveals that although Pollack and Blair managed to produce a strong player using a simple 95% Inheritance 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, at least for *our* implementation of the hill-climbing approach. 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"* [PB98], 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 models showed that this is not the case.

11.3 Conclusions

In this part of the thesis, we have shown how a simple but non-trivial algorithm can be modelled as a Markov Chain by using the M^2ICAL method, namely the hill-climbing approach used to create the backgammon player HC-Gammon. 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. In doing so, we discovered some interesting aspects of the algorithm that could have ramifications for other similar algorithms, and we were also able to improve the algorithm so that it could produce a superior player in a shorter amount of computation time.

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, although the accuracy can be increased at the cost of added computation time. 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.

Part IV

Further Discussions

In this final part of the thesis, we delve deeper into the intricacies of the *M*^{2}*ICAL* method. In general, the more Monte Carlo simulations we perform, the more accurate will be the output of our model. Our experiments on SCSA have shown that when the algorithm being analyzed is simple enough such that its neighbourhood function can be closely estimated using Monte Carlo simulations, then the model can predict the algorithm's performance almost exactly. However, for most practical algorithms it would be prohibitively time-consuming to perform the very large number of Monte Carlo simulations in order to model the algorithm to such a high degree of accuracy. Therefore, we must reduce the number of Monte Carlo simulations used at the expense of some of the accuracy of the model. Nonetheless, even a somewhat less accurate model can be useful, as shown by our experiments on HC-Gammon.

For the remainder of this dissertation, we will discuss the issues that arise when employing our basic technique on practical problems. In Chapter 12, we examine the trade-off between the accuracy of the model and computation time of the various parameters, including the number of classes N; the player evaluation size M_{opp}; the class population parameters; and the neighborhood distribution parameters. The factors involved in adapting the model for practical algorithms that possess traits that are different from the algorithms that we have already examined is discussed in Chapter 13. Finally, we conclude this dissertation in Chapter 14 with a summary of the results and contributions made, along with some possible directions for further research.

Chapter 12

Significance of Parameters

One of the greatest criticisms of the M^2ICAL method is the amount of time required to run the multitude of Monte Carlo simulations in order to produce the Markov Chain model of the system. While the ability to estimate important properties like the expected player strength and solution spread is useful, if it requires an excessive amount of time to create the model, its usefulness becomes severely limited. In particular, it would be impractical to make use of the M^2ICAL method if you can run the algorithm itself several times in the same amount of time, thereby deriving the same properties in that manner.

The basis of the model is the use of Monte Carlo simulations to estimate the workings of the algorithm in question. In general, the more data points are used in a Monte Carlo simulation, the more accurate the result will be. Hence, there is always a trade-off between the number of data points used (and the resultant increase in computation time) and the accuracy of the result. However, different parameters in the model affect different aspects of the accuracy of the model. In this chapter, we discuss the effects of varying the various parameters in the model, which would help the practitioner who wishes to implement the M^2ICAL method to decide how to customize the model to suit his needs.

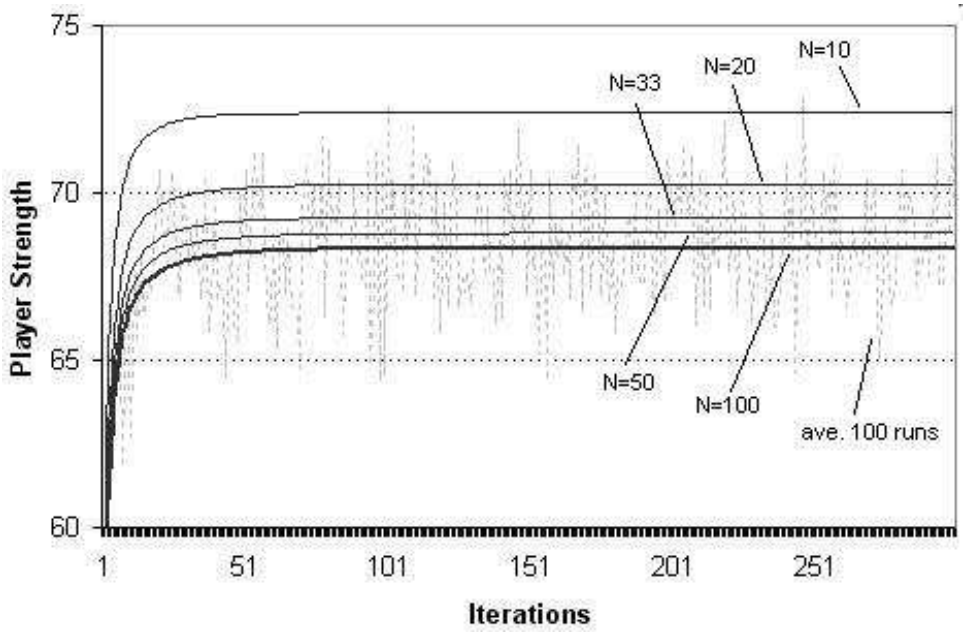

Figure 12.1: Expected player strength using N=100, 50, 33, 20 and 10 for ModNim(100,3) using SCSA

12.1 Number of Classes N

The first and perhaps the most important parameter to decide when employing this technique is the number of classes or states N in the Markov Chain. In the model, all players that belong to a particular class are grouped together and are essentially indistinguishable from each other. The set of all possible players is divided into N separate classes, so the results returned by the Markov Chain will only be accurate up to $1/N$ of the entire search space. Hence, the value of N affects the *granularity* of the results.

To examine the impact of having a different number of classes on the results obtained by the Markov Chain model, we return to our experiment on SCSA on ModNim(100,3). Using the neighbourhood distribution λ_{ij} and the error distribution δ_{ij} that we have already derived, we restructure the model so that it contains fewer than our original $N = 100$ (therefore, all other parameters in the model remain the same). Figure 12.1

gives the expected solution quality predicted by the Markov Chain model when N=100, 50, 33, 20 and 10 respectively. Our original (and most accurate) results obtained with a model using $N = 100$ classes is provided by the bold line. The results given by the smaller models are given in grey and labelled accordingly, while the results found by averaging 100 runs of SCSA is given by the dashed grey line.

In this instance, we see that as the number of classes in the model N decreases, the expected player strength predicted by the model is overestimated by increasing amounts. In particular, notice that when $N = 10$, the predicted expected player strength converges to a value of 72.38%, which is certainly higher than the values obtained from the actual runs of the algorithm. Compared to the expected strength of 68.36% predicted when $N = 100$, we see that there is only a slightly greater than a 4% disparity in the predictions even though only 10% of the classes were used.

This example illustrates that there is a tradeoff between the accuracy of the result and the number of classes involved in the model (and hence its computation time). While in this case it turns out that the player strength is overestimated, it is possible that the results will be underestimated instead for other problems. In any case, for a given percentage decrease $p\%$ in the number of classes, we can expect a disparity of *up to* $p\%$ between the two models, but this example shows that the disparity is more likely to be significantly less.

Figure 12.2 shows the corresponding results for the predicted standard deviation of the Markov Chain model when $N = 100$, 50, 33, 20 and 10. Once again, the original results when $N = 100$ are given by the bold line, the results of the smaller models are given by grey lines, and the sample standard deviations of 100 runs of SCSA on ModNim are given by the dashed grey line. The standard deviation for $N = 50$ is labelled as such, while the standard deviation for $N = 33$, 20 and 10 are so close that they are practically indistinguishable in Figure 12.2.

In this case, we see that the standard deviation is underestimated as the number of

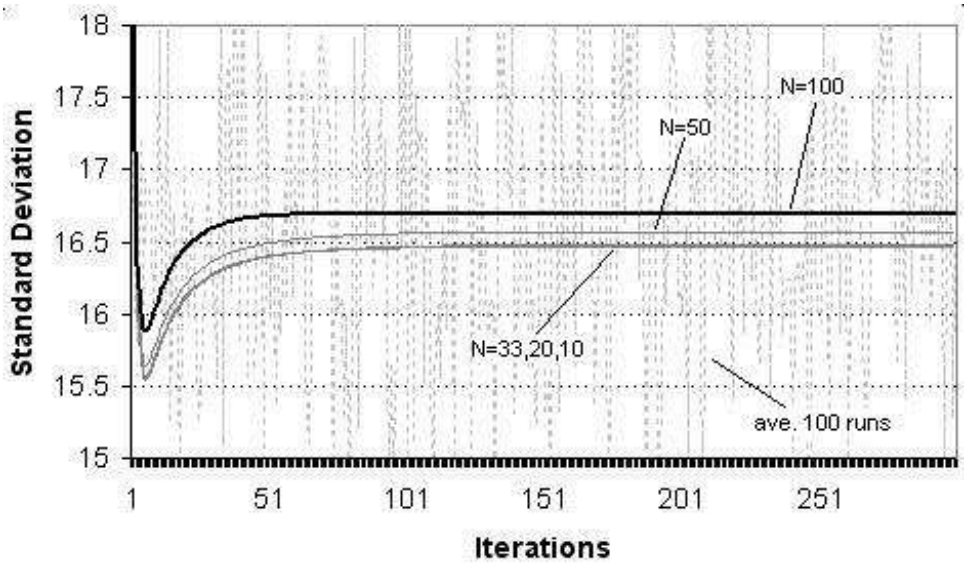

Figure 12.2: Standard Deviation using N=100, 50, 33, 20 and 10 for ModNim using SCSA

classes decreases. It is logical to expect the predicted standard deviation to be underestimated as the number of classes in the model decreases since players of similar strengths are divided into fewer classes. Hence there would be a smaller spread of players with distinct strengths, resulting in a lower standard deviation. However, in this example the disparity is very small and still well within the results from the actual runs of the algorithm. In fact, the difference between $N = 100$ and $N = 10$ after convergence is less than 0.22% of all classes.

The number of classes in the Markov Chain model has a significant effect on the running time of the model, since it affects several other parameters. However, it appears that even when reducing the number of classes (e.g., from $N = 100$ down to $N = 10$), the decrease in accuracy is still small enough to provide a decent estimation of estimated player strength and standard deviation. Of course, the nature of the problem largely determines the minimum number of classes that the model must contain in order for the information to be useful, but these observations are encouraging since there appears to

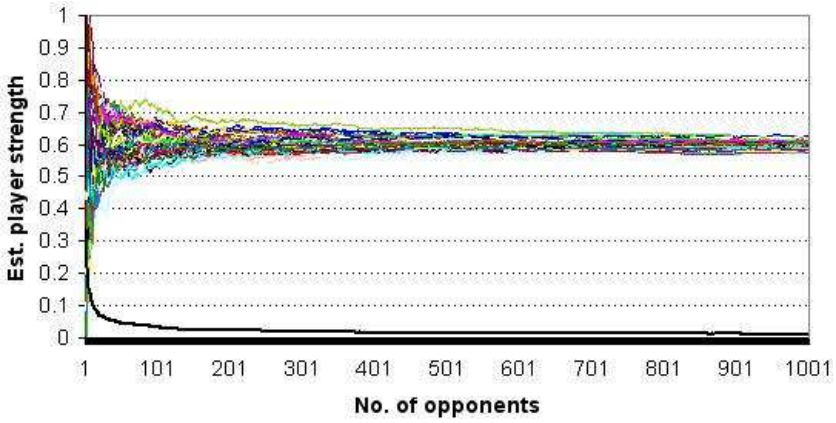

Figure 12.3: Estimated player strength (and std. dev.) with M_{opp}

be considerable leeway in deciding N when the various parameters are being decided in order to make the application of this model feasible on a difficult problem.

12.2 Player Evaluation Size M_{opp}

While the number of classes N determines the granularity of the results obtained, it is the player evaluation size M_{opp} that has the greatest effect on the amount of computation time required to generate the *M²ICAL* model. This is because whenever a player is generated, its strength must be estimated by playing it against M_{opp} randomly generated players. Hence, it is advisable to determine the smallest value of M_{opp} that provides an acceptable degree of accuracy when evaluating player strength.

While it is a well-known result that the accuracy of a Monte Carlo simulation is proportional to the square root of the number of sample points, this is not particularly helpful since we do not know the accuracy of the comparison function. Therefore, in order to determine an appropriate value for M_{opp} in our backgammon experiments, we randomly generated a backgammon player, and then measured the value of its estimated strength for various values of M_{opp}. Since we play two games against each opponent

using the same dice stream, once as first and once as second player, the number of games is played is $2 \times M_{opp}$. Figure 12.3 shows the progression of the estimated playing strength for 100 sets of M_{opp} randomly generated opponents, which clearly shows that as M_{opp} increases, the estimated playing strength for the player converges towards a similar value (around 60% in this case). Repeated trials with other randomly generated players produced similar results.

The bold black line shows the standard deviation σ of the estimated playing strength values; at $M_{opp} = 100$, the standard deviation for this player was $\sigma = 0.031716$. Assuming a normal distribution of values with the mean value equal to the actual strength of the player, the 90% confidence interval at this point is $\pm 1.64485\sigma = \pm 0.052168$, which encompasses slightly more than a 10% strength range (or 1 class in our Markov chain of 10 classes). For our experiments, we decided that an approximately 90% confidence interval of a 1-class range is sufficient for our investigation, and therefore we used the value of $M_{opp} = 100$ in our experiments[1]. Using this value, each player usually takes 30 to 45 seconds to evaluate on a Pentium-IV 1.6 GHz PC with 512MB RAM.

It is crucial to the feasibility of the *M²ICAL* method that we are able to attain a confidence interval of as high as 90% in our player strength evaluation even with a complex game like backgammon, using a value of M_{opp} small enough to ensure a reasonable running time. For the analysis of more complex algorithms or problems that may require more player evaluations, further computation time can be saved by sacrificing some degree of accuracy in the evaluations.

[1] For a 90% confidence interval of $\leq \pm 0.05$, we require an M_{opp} value of about 110. While we could have used $M_{opp} = 110$ for our experiments, we felt that $M_{opp} = 100$ for a 10-class Markov Chain simply 'fits' better.

12.3 Class Population Parameters

The *M*^{2}*ICAL* method involves first generating and evaluating a set of players, and then using these players to estimate the various distributions required to derive a Markov Chain model. Therefore, it is important that the sample of players used is both large enough and varied enough to be a representative sample of the players that are produced by the algorithm in question. Once again, there is a definite tradeoff between the accuracy of the final model and time and memory constraints. For example, since ModNim players can be represented by a simple vector of integers, memory space is not a significant constraint in our ModNim experiments. Combined with the fast computation time for each game of ModNim, we were able to simply generate 10,000 ModNim players and use them as our representative sample. However, the neural networks used to represent backgammon players take up significantly more memory space, and backgammon games take much more time to complete. Hence, we had to restrict the number of players in our representative sample both to reduce the number of Monte Carlo evaluations required and to limit the amount of memory used to store these players. In this section, we discuss cases where both time and space are significant factors in our decisions.

There are three parameters in our approach that affect the nature of the representative population. The first is M_{sample}, which is the initial number of players that are generated. If we consider all algorithms starting from different initial players as distinct parallel runs, then the value of M_{sample} can be thought of as the number of distinct parallel runs from which we produce the rest of the representative population. The value of M_{sample} must be carefully chosen with respect to the total number of classes N in the Markov Chain model. If M_{sample} is too large, then it is likely that a large number of players retained by the model will come from the first few iterations of the algorithm, and therefore players from later in the algorithm may be under-represented. On the other hand, if M_{sample} is too small, then there may be too many players from later in the algorithm in the representative population, when local optimals may have been reached.

Furthermore, we require a value of M_{sample} that is large enough so that we can estimate the distribution of the initial population for $\vec{v}[1..N]$. As a rule of thumb, we find that a value of $M_{sample} = N \cdot \hat{\gamma}$ gives sufficient variety to the representative population.

The second parameter is $\hat{\gamma}$, which is the maximum number of players from each class that is retained in the representative population. In our current formulation of the *M²ICAL* method, we limit the sample size of each class to the same value of $\hat{\gamma}$ for simplicity; it may be worthwhile to implement a separate maximum class size $\hat{\gamma}_i$ for each class i. In any case, a limit to the sample size for each class is required since otherwise players of strengths similar to the initial population will be over-represented. The appropriate value of $\hat{\gamma}$ is determined by both memory and time constraints. There must be sufficient memory to store up to $N \cdot \hat{\gamma}$ players; furthermore, as $N \cdot \hat{\gamma}$ increases, so does the time required to generate the players of these player strengths. We decided on the value of $\hat{\gamma} = 20$ for our backgammon experiments because we could store 200 neural network backgammon players in memory without requiring disk I/O, and the amount of time required to fill the classes was reasonable.

The third parameter is M_{pop}, which is the number of descendents that are generated when attempting to produce players from non-full classes. This value must also be carefully chosen. If M_{pop} is too high, then too many of the players in the sample population will be direct descendents of players from certain classes, and most of the players in the population will come from early in the algorithm. If M_{pop} is too low, then the process of populating the classes may be unable to produce players of sufficiently varied strengths to produce a meaningful Markov Chain. In our backgammon experiments, we have found that a value of $M_{pop} = 100 = \hat{\gamma}/2$ is able to produce a sufficiently varied sample population in a reasonable amount of computation time.

Our analyses of the effects of these three parameters M_{sample}, $\hat{\gamma}$ and M_{pop} are unavoidably general at this time, since if any of these parameters are changed, then the entire experiment must be re-run in order to perceive the effects. However, as long as

a little trial-and-error and common sense is applied, we believe that there should not be undue difficulty in finding appropriate values for these parameters for any given problem and algorithm.

12.4 Error and Neighbourhood Distribution Parameters

After populating the classes, the *M²ICAL* model involves discovering the error and neighbourhood distributions of the algorithm by performing several Monte Carlo simulations on the sample population. To find the error distribution across all pairs of classes i and j, we simply play M_{wpm} games between randomly selected players from class i and class j, and then record the proportion of wins achieved. It is reasonable to expect that an appropriate value for M_{opp} would also be suitable for M_{wpm}; M_{opp} is the number of games required before the strength of a player can be determined to an acceptable level of accuracy, while M_{wpm} is the number of games required before the proportion of wins between two classes players can be ascertained to a similarly acceptable level of accuracy. This is why we chose $M_{wpm} = 200 = 2 \cdot M_{opp}$ (because we play 2 games for every opponent) for our backgammon experiments.

Note that the value of M_{wpm} does not greatly affect the running time of the model, since generating the win probability matrix W only involves playing O($M_{wpm} \cdot N^2$) games. Compared to other parameters like M_{pop} and $\hat{\gamma}$, which affects the number of players generated (each requiring M_{opp} games to evaluate), the amount of time required to generate the win probability matrix is one order of magnitude lower. Hence, we can set M_{wpm} to a high value without severely increasing the running time. However, our backgammon experiments did not require a value of M_{wpm} greater than 200.

The process required to discover the neighbourhood distribution λ_{ij} for the algorithm obviously depends on the algorithm itself. For SCSA on ModNim, the neighbourhood distribution is $\frac{\gamma_j}{\Gamma_N}$ and requires no additional computation. For HC-Gammon, however, the neighbourhood distribution requires the combination of the challenger distribution

C_i, estimated using M_{cha} applications of the mutation function for each class i; and the descendent distribution D_{ij}, estimated using M_{des} applications of the descendent function for each pair of classes i and j.

Ostensibly, for more complex algorithms, even more distributions are required before the neighbourhood distribution can be formulated and included into the Markov Chain model. However, one generalization we can make is that the distributions required must be in terms of the number of classes N, since we wish to discover how the next state in the algorithm is affected by the current state in terms of its probability of belonging to the various classes. We believe that a good value for parameters like M_{cha} and M_{des} would once again be $2 \cdot M_{opp}$, which is the number of games required for the Monte Carlo evaluation of a player to have approximately 90% confidence. For much the same reason, this number of trials (200 in the case of our HC-Gammon experiments) should also produce an estimated distribution with close to 90% confidence.

Chapter 13

Adapting the Model

The aim of this research is to devise a technique that can analyze imperfect comparison algorithms in general (and algorithms on the game-playing problem in particular). In choosing our target problems, our aim was to show the capabilities of the *M²ICAL* method in a simple yet non-trivial setting. Therefore, neither of the two case applications that we have chosen, namely SCSA on ModNim and HC-Gammon on backgammon, were particularly complex algorithms.

Depending on the nature of the problem, much work may be required in order for our technique to be useful for the analysis of practical algorithms. While some aspects of algorithms can be easily and trivially handled by the model, others require possibly significant changes to the basic *M²ICAL* method before an accurate analysis can be achieved. This chapter discusses some of the possible issues that may arise when employing the model on practical algorithms.

13.1 Simple Adaptations

Certain types of algorithms are easily handled by the basic *M²ICAL* method presented in this thesis. For example, a common type of strict hill-climbing algorithm is called

a *neighbourhood search algorithm*, where the potential next state is selected from the neighbourhood of the current state. Algorithms of this type can be modeled as a Markov Chain using the transition matrix given in Equation (4.5), by using Monte Carlo simulations to estimate λ_{ij} based on the neighbourhood function employed.

Furthermore, any technique that does not affect the algorithm directly has no effect on the implementation of the model whatsoever. For instance, in the field of game-playing programs, there are several techniques that are employed in order to improve the overall playing strength of the generated player. These include:

- **Opening books.** A set of pre-computed moves from the start of the game that are considered desirable, which saves the program from having to compute positions early in the game.
- **Endgame databases.** A set of pre-computed evaluations of all positions with some trait (e.g., the set of all chess positions with 2 bishops and a king vs. a king), such that the program can end the search once a position in the database is reached.
- **Transposition tables.** A technique that reduces the number of re-evaluations of the same position of the course of a game.
- **Move ordering methods.** Methods that decide the order of moves to search, which may significantly reduce the search time, e.g., the history heuristic.

All of these techniques affect the speed and depth of search that the game-playing program can achieve. However, these techniques do not affect our model because they are implemented within the individual game-playing programs themselves. While the results predicted by the model may indeed be affected by these techniques, there is no difference in the implementation of the M^2ICAL method.

13.2 Player Strength Evaluations

Evaluating the strength of the generated players is a major difficulty in games research. A layman's notion of a player's strength in an intellectual game corresponds to its ability to beat other players. When analyzing a trivial or small game, it may be possible to find a player's strength by fully enumerating all possibilities, but this is impossible for practical games since the number of possible players in such games is astronomically large. In existing research on intellectual games, two methods of evaluating player strength have become accepted practice, but both of these methods have their weaknesses.

The first way to evaluate player strength that has been employed in existing research is to compete the generated player against a fixed benchmark player: Pollack and Blair's backgammon program was measured against the open-source program *PUBEVAL* [PB98]; Kendall and Whitwell's chess program played against the commercial software *Chessmaster 2100* and *Chessmaster 8000* [KW01]; and the checkers program *Anaconda* was evaluated against both a commercial program *Hoyle's Classic Games* [CF00] and also against the world champion checkers program *Chinook* [Fog02]. However, the usefulness of this approach depends heavily on the ability of the benchmark player. If the benchmark player is too weak, results against that player may overstate the ability of the generated player. On the other hand, an overly strong benchmark player that is far superior to the generated player will be victorious over the generated player by a very wide margin, which would severely obscure the abilities of the generated player when compared to average players.

The second is to compete the player against human opposition: *Anaconda* played on the Internet checkers community *www.zone.net* [CF01, Fog02], and Moriarty and Miikkulainen's Othello program was evaluated by 1993 World Othello Champion David Shaman [MM95]. Unfortunately, this approach is usually very time-consuming and dependent on the availability and ability of human opposition. In fact, checkers was famously and mistakenly considered solved when Samuel's program beat a self-proclaimed

blind checkers Master [Sam59], who was in reality much weaker than he claimed, causing researchers to ignore checkers in computer science for decades.

In the *M²ICAL* method, we introduce a third way to evaluate players, which is to use *randomly generated players* when performing Monte Carlo evaluations of player strength. Depending on the data representation and method of random generation, the set of players from which the opponents is drawn can be uniformly selected from the space of all possible players. This technique has certain advantages over existing methods: since a variety of opponents is generated, the strength of the player is tested against a wider spectrum of player strengths than when only a single fixed benchmark player is employed; and the ready availability and ease of implementation allows many more games to be played than when human opposition is used.

Our experiments have provided examples of cases where this technique works. However, for other practical problems, using randomly generated opponents for Monte Carlo evaluation of players may be insufficient. The critical observation is that for most games, the number of "strong" players is a very small percentage of the total number of all possible players. The comparison error between these "strong" players and the overwhelming majority of randomly generated players would be essentially zero. For example, you would never expect Gary Kasparov to be beaten by a randomly generated chess player no matter how many games were played. Hence, if the algorithm that we were analyzing was able to create a "strong" player, beyond a certain number of iterations in the algorithm the Monte Carlo evaluations based on random opponents would return a 100% success rate for the incumbent. At this point, the model loses its ability to differentiate between the strengths of the generated players.

When Pollack and Blair looked to evaluate the strength of HC-Gammon, they used the strong public domain backgammon program *PUBEVAL* as a benchmark player. In our experiments, we could not make use of *PUBEVAL* for the evaluation of players because experiments showed that *PUBEVAL* loses less than 20 out of 1000 games against

randomly generated opponents (which means that the strength of *PUBEVAL* is within the top 2 percent of all possible players). Hence, if we measured the strength of the generated players using their results against *PUBEVAL*, practically all of these players would be placed into the lowest class. Once again, the model would have lost its ability to differentiate between the strengths of the generated players.

How the strengths of the generated players should be evaluated depends on the capabilities of the algorithm itself. In our experiments using SCSA and HC-Gammon, it turns out that these algorithms do not generally produce players that are able to compete with "strong" players, and therefore we could use randomly generated players for evaluation purposes. However, for algorithms that are able to eventually generate "strong" players, it may be possible to make use of existing "strong" players for the Monte Carlo evaluations at some point in order to provide a better picture of the algorithm's performance.

The idea of introducing a time-lag in Exp A3 and B3 can be used for this purpose. Early in the algorithm, when the current player is relatively weak, we could make use of randomly generated players for the Monte Carlo evaluations. This model might predict that the expected solution quality of the algorithm would reach the highest class after t iterations; we could then create a second model with a time-lag of length t, but make use of existing "strong" players (such as *PUBEVAL*) for the Monte Carlo evaluations instead. The main aim of this framework is to provide information on the capabilities of the algorithm in question. Depending on the algorithm, the most useful information that the model can provide could be relative to the space of all possible players (i.e., using randomly generated players), or relative to particular benchmark "strong" players. By changing the evaluating opponents at various stages of the algorithm and re-generating the model, the information provided by the *M*^{2}*ICAL* method can maintain relevance to the practitioner.

13.3 Populations

In this dissertation, we examined two algorithms where the state of the algorithm consists of only a single player. In practical algorithms, it is often the case that a *population* of p players is retained, and the potential next state is derived from this set of p players. For example, a genetic algorithm would produce "offspring" from the current population of players using operations like mutation and crossover, and the current state after each iteration of the algorithm would be a population of players. If there are $|S|$ distinct players in the game-playing problem, then there are $\binom{|S|}{p}$ possible populations of p players. This is an increase in the search space by O($|S|^p$).

Theoretically, a population can be treated in a similar way as any solution by evaluating the quality of the population using Monte Carlo simulations in some way. In an algorithm employing a population of p players, one obvious way to evaluate the quality of the population is to evaluate each member of the population by playing them against M_{opp} randomly generated opponents, and then adding the estimated strengths of each player together. However, this would increase the running time by a multiplicative factor of p, which may be infeasible for practical problems.

Alternatively, we may decide to estimate the quality of a population simply by evaluating its "best" member, since such algorithms usually return this member as the solution at the end of the algorithm. This would result in a running time asymptotically identical to our original approach. Unfortunately, such a scheme may not work well in practice because the intrinsic assumption in the model is that the neighbourhood function of two solutions of similar quality will be similar. In algorithms that use populations, the composition of the next state is generally dependent on all the p members of the current state. By considering the strength of only the "best" member of the population when evaluating the entire population, we fail to take into account the effect of the other members of the population when determining the neighbourhood function. While such a scheme may not be completely inaccurate (after all, if the "best" player is of a particular strength,

it is reasonable to assume that the strengths of the other players in the population will not be wildly different), it may not be sufficient in practical cases.

Perhaps the best way to handle populations is to compromise between the two extremes, by classifying the population based on the strengths of $p' < p$ of its members, where p' depends on the specifics of the problem. This would increase the running time by a factor of p', and is yet another parameter of the model where the tradeoff is between running time and accuracy. The appropriate value for p' is problem-dependent, and deserves further research.

13.4 Annealing

Another common technique used in algorithms that generate game-playing programs is annealing. An annealing algorithm begins with a neighbourhood function that is relatively broad, in the sense that there is a relatively large number of possible challengers for any given incumbent. As the algorithm progresses, the neighbourhood function employed is made narrower and narrower, thereby increasing the probability of generating a player from the immediate neighbourhood of the incumbent. This process simulates the annealing process in metallurgy, where the atoms of a metal are agitated by heating, and then slowly cooled, resulting in a harder material when the atoms re-settle into a tighter configuration.

Experiments A2, A3, B2 and B3 (see Sections 9.2, 9.3, 10.3 and 10.4 respectively) are annealing techniques in the sense that when the requirements of the comparison function are increased, the probability of generating a weaker challenger that is able to defeat the incumbent is reduced, and hence the frequency of incumbent replacement is also reduced. In practical problems, annealing is not usually implemented in this stepwise manner. Instead, it is usually implemented as some kind of adaptive parameter that acts on the neighbourhood function by multiplying the amount of mutation from the

parent by a factor, which is reduced over time. This means that the Markov Chain is non-homogenous, and the transition matrix changes after every iteration depending on the effect of the annealing scheme on the neighbourhood function. In our current approach, a direct application of the *M*^{2}*ICAL* method would be to re-compute the transition matrix after every iteration. This is of course impractical, since doing so will take a much longer time than simply running the algorithm itself.

Instead, we need to estimate the effect of the annealing scheme on the neighbourhood function so that this effect can be included in our computation of the transition matrix over time. One possible way to approach this task is to observe that the purpose of annealing is to restrict the newly generated challengers to players that are more similar to the incumbent over time. In effect, this would cause the challenger distribution function C_{ij} to be modified by a factor that is inversely dependent on the magnitude of $|i - j|$ (i.e., an incumbent from class i is increasingly likely to generate a challenger from class j if i and j are close together). In order to discover the nature of this factor, we can once again make use of Monte Carlo simulations to discover how the neighbourhood distributions are affected when the annealing scheme is performed over the iterations, and then perform linear regression to translate this effect into a computable function. However, this would add another layer of estimation into the technique, and is likely to reduce the accuracy of the model. Further research is required to ascertain if the *M*^{2}*ICAL* method remains feasible using this technique.

Chapter 14

Conclusions

14.1 Academic Contributions

In this dissertation, we described a method of analyzing the performance of imperfect comparison algorithms by modeling them as Markov Chains using Monte Carlo Simulations to estimate the relevant distributions. We call this process the M^2ICAL method, and it allows the practitioner to predict the performance of the algorithm in terms of metrics like expected solution quality; standard deviation; and time to convergence. To the best of our knowledge, this is the first technique proposed that is capable of analyzing imperfect comparison algorithms objectively.

Machine learning approaches to finding strong players in the game-playing problem is an archetypal example of imperfect comparison algorithms. We have used two instances of algorithms in this field to illustrate the capabilities of the M^2ICAL model. The first example is SCSA on ModNim(100,3), which is a simple algorithm on a solved but computationally non-trivial game. This example, analysed in Chapter 6, showed the high degree of accuracy that the M^2ICAL model's predictions can obtain in an idealized situation where the neighbourhood distributions of the algorithm can be accurately estimated using Monte Carlo simulations.

The second example analysed in Part III of this thesis examines HC-Gammon, a hill-climbing algorithm on the game of backgammon utilized by Pollack and Blair in their 1998 publication. This is a more complex algorithm and problem than SCSA on ModNim, but the M^2ICAL method was still able to derive a Markov Chain depiction of the algorithm that could produce reasonably accurate predictions of its performance. By implementing the M^2ICAL method on the HC-Gammon generation algorithm, we were able to discover certain properties that cast doubt on some of the assertions made by the authors in their original work.

Ultimately, the concept behind the M^2ICAL method is simple. The comparison errors that occur in imperfect comparison algorithms make it difficult to accurately and objectively evaluate their performance using existing methodologies. The M^2ICAL method handles these comparison errors by performing Monte Carlo simulations to estimate their probability distributions, and translates these distributions into a Markov Chain model. The algorithm can then be analyzed using existing Markov Chain theory.

14.2 Why use M^2ICAL ?

The M^2ICAL method 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[1]. As an analysis tool, the M^2ICAL method is only useful if it presents a performance advantage over running the target algorithm itself if it is to justify the overhead involved in generating the distributions required to derive the model.

Theoretically, all of the properties modelled by the M^2ICAL method can be obtained by running the target algorithm multiple times and performing Monte Carlo simulations

[1] in particular, our definition of "time to convergence" is not in accordance with traditional definitions of Markov Chain convergence, but may be of greater use to the practitioner

on these runs. However, while the initial derivation of the M^2ICAL model using several Monte Carlo simulations is time-consuming, once the model is derived it can predict the expected performance of the algorithm 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 straightforward application of the M^2ICAL method on existing algorithms to re-check the veracity of the analyses could be the source of much useful information. Since there have been no techniques available for the analysis of imperfect comparison algorithms in practical settings prior to the M^2ICAL method, it is likely that some of the previous analyses of such algorithms may be erroneous or unconfirmed; for example, our analysis of HC-Gammon has refuted the supposition that backgammon tends not to cause hill-climbing approaches to enter mediocre local optimal states. We could therefore implement the M^2ICAL method to either correct or confirm (or at least bolster) the analysis of existing research.

Beyond the re-examination of existing work, the M^2ICAL method can be helpful in the design of new algorithms. By modeling the algorithm into a M^2ICAL model, 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. Hence, suitable adaptations of the M^2ICAL method can potentially be of great use to the design of practical algorithms.

There may also be more useful information already present in the produced M^2ICAL

model than is detailed in this dissertation. The Markov Chain representation is fully defined by the transition matrix, and this matrix is derived by combining several probability distributions. Therefore, careful analysis of these distributions may uncover *reasons* behind the behaviour of the model itself. For example, it may be discovered that the winning probability W_{ij} of a player from class i beating a player from class j, $i < j$, compared to the corresponding neighbourhood probability λ_{ij}, may be the determining factor in whether the expected solution quality increases or decreases over time. If so, then an increase in the strictness of the comparison function may be required to reduce the probability of such comparison errors.

The *M*^{2}*ICAL* method presents pioneering work on the analysis of imperfect comparison algorithms. In a field like computer science research on intellectual games, it provides an objective alternative to the existing method of gauging the performance of algorithms using the results of the best player produced against benchmark players of possibly inaccurately determined strength. Other fields with similar difficulties in judging solution quality may also benefit from using the *M*^{2}*ICAL* method for algorithm analysis.

14.3 Future Work

The most important weakness of the *M*^{2}*ICAL* method is the amount of computation time required to derive the model, which must be significantly less than running the algorithm itself multiple times in order for the modeling to be worthwhile. Initial investigations on the effects of the input parameters show that certain important parameters like the number of classes N and the number of samples taken for the Monte Carlo evaluation of solutions M_{opp} can be reduced for a significant reduction in computation time but with a less than proportionate reduction in prediction accuracy. However, it is difficult to generalize the effects of "cutting corners" on these parameters, and they should be determined on a case-by-case basis.

Many of the parameters that we employed were arbitrarily determined, and further experiments should be performed in order to discover good heuristics for the parameters. For example, it is not known precisely how the accuracy of the prediction is related to factors like the number of classes in the Markov Chain; population sample size and diversity; the complexity of the problem, etc. It may be worthwhile to implement the M^2ICAL method on an easily customisable problem in order to judge the effects of such factors - ModNim may be a suitable test case for this purpose.

There are also certain types of algorithms that have not been tested in this research. These include algorithms employing populations or annealing, which are two common devices of practical machine learning algorithms in use today. While we have proposed possible ways to handle these types of algorithms, it remains to be seen if these proposed solutions can maintain the accuracy level of the M^2ICAL method using a feasible amount of computation time.

It is worth restating that although this dissertation focused primarily on the game-playing problem, the M^2ICAL method can potentially be applied to any imperfect comparison algorithm, or any comparison-based algorithm on an optimization problem that is not well-defined. As long as there is a way to use Monte Carlo simulations to estimate the quality of a solution to an acceptable degree of accuracy, the algorithm can be modeled and analyzed in this way. Researchers from other fields may therefore find the M^2ICAL method a useful analytical tool.

Bibliography

[AM92] Simon Anderson and George MacNeil. Artificial neural networks technology. Technical Report F30602-89-C-0082, Data & Analysis Center for Software, 1992.

[And02] Edward James Anderson. Markov Chain modeling of the solution surface in local search. *Journal of the Operational Research Society,* 53(6):630–636, 2002.

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, New York, 1982.

[BN97] Yngvi Björnsson and Monty Newborn. Kasparov versus Deep Blue: Computer chess comes of age. *International Computer Games Association (ICGA) Journal*, 20(2):92, 1997.

[CF00] Kumar Chellapilla and David B. Fogel. Anaconda defeats Hoyle 6-0: A case study competing an evolved checkers program against commercially available software. *Proceedings of Congress on Evolutionary Computation (CEC '00),* pages 857–863, 2000.

[CF01] Kumar Chellapilla and David B Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. on Evolutionary Computation*, 5(5):422–428, 2001.

[CKL+03] S. Y. Chong, D. C. Ku, H. S. Lim, M. K. Tan, and J. D. White. Evolved neural networks learning Othello strategies. In *Proceedings of Congress on Evolutionary Computation (CEC'03)*, pages 2222–2229, 2003.

[Fog02] David B Fogel. *Blondie24: Playing at the Edge of AI*. Academic Press, London, UK, 2002.

[Hay99] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice-Hall Inc., 2nd edition, 1999.

[Kei96] Tom Keith. Standard rules of Backgammon. Website, 1996. `http://www.bkgm.com/rules.html`.

[KW01] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 995–1002, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 2001. IEEE Press.

[LHO01] Andrew Lim, Wee-Kit Ho, and Wee-Chong Oon. Maximizing paper spread in examination timetabling using a vehicle routing method. In *Proceedings of the 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '01)*, pages 359–366, 11 2001.

[LM01] Alex Lubberts and Risto Miikkulainen. Co-evolving a Go-playing neural network. In Richard K. Belew and Hugues Juillè, editors, *Coevolution: Turning Adaptive Algorithms upon Themselves*, pages 14–19, San Francisco, California, USA, 7 2001.

[Mar06] Andrey Andreyevich Markov. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete, 2-ya seriya*, 15:135–156, 1906.

[Mar71] Andrey Andreyevich Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. In R. Howard, editor, *Dynamic Probabilistic Systems, volume 1: Markov Chains*. John Wiley and Sons, 1971.

[MM95] David Moriarty and Risto Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3–4):195–209, 1995.

[MT93] S. P. Meyn and R. L. Tweedie. *Markov Chains and Stochastic Stability*. Springer, London, 1993.

[NV92] A. Nix and M. D. Vose. Modeling Genetic Algorithms with Markov Chains. *Annals of Mathematics and Artificial Intelligence*, 5:79–88, 1992.

[PB98] Jordan B. Pollack and Alan D. Blair. Coevolution in the successful learning of Backgammon strategy. *Machine Learning*, 32:225–9240, 1998.

[RB96] Christopher D. Rosin and Richard K. Belew. A competitive approach to game learning. In *Proceedings of the 9th Annual ACM Conference on Computational Learning Theory (COLT-96)*, pages 292–302, 1996.

[Sam59] Arthur L. Samuel. Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[Sam67] Arthur L. Samuel. Some studies in machine learning using the game of Checkers ii - recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[Sor91] G. Sorkin. Efficiency of simulated annealing: Analysis by rapidly-mixing Markov Chains and results for fractal landscapes. Technical Report UCB/ERL M91/12, EECS Department, University of California, Berkeley, 1991.

[Tes95] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[Tho82] Ken Thompson. Computer Chess strength. *Advances in Computer Chess*, 3:55–56, 1982.

[Tho86] Ken Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.

[WZ99] Alden H. Wright and Yong Zhao. Markov Chain models of genetic algorithms. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 734–741, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[Yao99] Xin Yao. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE*, 87(9):1423–1447, 1999.

Appendix A

Table of Symbols

Symbol	Brief Description	Page
$\succ$, $\prec$	Shorthand notation for BCF	24
$\alpha_{(t)}$, $\alpha'_{(t)}$, $\alpha''_{(t)}$	The probability that the comparison function of P, P' or P'', respectively, is being used at iteration t	87
$\beta_{(t)}^{k/l}$	The probability that at least k challengers won in the last l iterations at iteration t	89
δ_{ij}	The comparison error of Q at states i and j	38
γ_i	The size of state i	33
$\hat{\gamma}$	The maximum number of players retained per class	34
Γ_i	The cumulative distribution function of γ at i	33
Γ_N	The total number of distinct players in the problem	34
κ	The number of challengers that must win in the last Λ iterations to change the comparison function	87
Λ	The number of most recent iterations considered when deciding to change the comparison function	87
λ_i	The neighbourhood distribution for class i	39

Table of Symbols (continued)

Symbol	Brief Description	Page
λ_{ij}	The probability that state j is chosen as the potential next state when the current state is i	40
μ	The mean of a probability vector $\vec{v}$	47
σ^2	The variance of a probability vector $\vec{v}$	47
1_f	A boolean indicator, returns 1 if f is true and 0 if f is false	24
$a_{(t)}$	The probability that the challenger wins in iteration t	88
$b_{(t)}^{k/l}$	The probability that exactly k challengers won between iterations $t-l+1$ and t inclusive	88
BCF	The *beats* comparison function	24
$c_i(j)$	The probability that a player in class i will create a challenger in class j	76
C_i	The challenger probability distribution for a class i	76
$d_{ij}(k)$	The probability that a descendent created from a parent of class i and challenger of class j will be of strength k	77
D_{ij}	The descendent probability distribution using parent state i and challenger state j	77
E	The set of edges in G representing all legal game moves	22
$E(PS(PL))$	The estimated strength of player PL	43
F	The objective function to an optimization problem **P**, $F: S \rightarrow \mathbb{R}$	21
$F(s)$	The quality of a solution s	21
$F(i)$	The quality measure of state i	32
$F'(PL_i)$	The estimated strength of player PL_i	31
g	The number of matches a player plays against each of M_{opp} opponents to estimate its strength	31

Table of Symbols (continued)

Symbol	Brief Description	Page
G	The directed graph representing a game	22
I	The finite state space of a Markov Chain	25
$M(v)$	The set of all legal moves from game position v, $M : V \rightarrow \bar{E}$	23
M_{cha}	The number of challengers generated to derive the challenger probability distribution C_i	76
M_{des}	The number of descendents generated to derive the descendent probability distribution D_{ij}	77
M_{nei}	The number of players generated to evaluate the neighbourhood distribution λ_i	39
M_{opp}	The number of randomly generated opponents used to estimate a player's strength	31
M_{pop}	The minimum number of new players generated from an unchecked class	34
M_{sample}	The number of initial players generated	34
M_{wpm}	The number of matches played between each pair of classes i and j to compute w_{ij}	37
ModNim(K,M)	The game of ModNim with K sticks initially and at most M sticks removed per move	51
N	The size of state space I (i.e., number of states) of a Markov Chain	25
p_{ij}	The element at row i, column j of transition matrix P	25
P	The transition matrix of a homogenous Markov Chain	25
P, P', P''	The transition matrices calculated using different comparison functions (to handle annealing schedules)	84

Table of Symbols (continued)

Symbol	Brief Description	Page
P	An optimization problem	21
PCF	A perfect comparison function	21
PL	A player of a game G, $PL : V \rightarrow E$	23
$PL^{(t)}$	A player produced by an algorithm after t iterations	43
$PS(PL_i)$	The strength of a player PL_i	24
Q	A comparison function, $Q : S \times S \rightarrow S$	21
s	A solution to an optimization problem **P**, $s \in S$	21
S	The set of solutions to an optimization problem **P**	21
	(the set of all players in a game-playing problem)	22
$\bar{S}$	A sample population of players	33
v	A vertex in G representing a valid game position	22
V	The set of vertices in G, representing all valid game positions	22
$v_{(t)}^{\rightarrow}$	The player strength probability vector of the algorithm after t iterations. Each element v_i ($i = 1$ to N) of $v_{(t)}^{\rightarrow}$ is the probability that a player belongs to class i after t iterations	43
$val(v)$	The value of a game position v	22
w_i	The weights of the current player for HC-Gammon	68
w_i'	The weights of the challenger derived from the current player for HC-Gammon	68
w_{ij}	The probability that a player from class i beats a player from class j playing first	36
$\bar{w}_{ij}$	The probability that a player from class i beats a player from class j playing second	37
W	The win probability matrix as first player	36

Table of Symbols (continued)

Symbol	Brief Description	Page
$\bar{W}$	The win probability matrix as second player	37
$W_{ij}^{\geq x(y_1/y_2)}$	The probability that a player from class i beats a player from class j at least x times out of y_1 games as first player and y_2 games as second player	38
X_t	The variable at time t of a Markov Chain	25

Appendix B

Analytical Determination of Stationary Vector

In this section, we make two simplifying assumptions on the nature of the imperfect comparison problem using SCSA. These two assumptions allow us to analytically calculate the final stationary vector for the resultant Markov Chain, without having to compute it numerically.

The first assumption is that the objective function F for the problem is strictly increasing, i.e., there exists an ordering $\{s_1, s_2, \ldots, s_{|S|}\}$ of all solutions $s_i \in S$ such that $F(s_i) = F(s_j)$ implies $i = j$, and $F(s_i) > F(s_j)$ implies $i > j$. Hence, no two solutions have equal quality, and there is one unique optimal solution. This can be modeled using a discrete Markov Chain with state space I equal to S. Such a case occurs when we consider the *ranking function* $R : S \rightarrow \mathbb{Z}^+$ that uniquely ranks all solutions in S according to F, and returns an integer value equal to the rank of the solution. We can define a ranking function for any problem with a countable number of solutions even when there are multiple solutions of equal quality, since the ranking function would (possibly arbitrarily) rank solutions of equal quality by unique rankings. Note that the ranking function is both strictly increasing as well as uniform, in that $R(s_j) - R(s_{j-1}) = R(s_k) - R(s_{k-1})$

for all $2 \leq j, k \leq |S|$. For convenience, we assume that the objective function is a ranking function. We number each state by its rank, such that $R(s_i) = i$.

The second assumption is that the comparison error is equal over all solutions, i.e., $\delta_{ij} = \delta$ for all $s_i \in S$. This means that the comparison error between two classes i and j are independent of the identities of i and j, i.e., if s_i is superior to s_j, then the comparison function returns s_i with probability $1 - \delta$ and s_j with probability δ. While this assumption is generally untrue for any particular problem, there could be instances where it applies. For example, in an application where the comparison is perfect but performed remotely, there could be an error probability of δ in the transmission of the result of the comparison function to the computational part of the system.

Given the above Markov Chain model of the algorithm, we can now construct its transition matrix. Let the total number of states $N = |S|$. To construct the $N \times N$ transition matrix P for this Markov Chain, we consider $p_j, 1 < j \leq N$. The value of $p_{jk}, 1 \leq k < j$ is the probability that s_k is chosen and Q makes an error, which is $\frac{\delta}{N}$. The value of $p_{jk}, j < k \leq N$ is the probability that s_k is chosen and Q makes a correct decision, which is $\frac{1-\delta}{N}$. Since $\sum_k p_{jk} = 1$,

$$\begin{aligned} p_{jj} &= \frac{N - (j-1)\delta - (N-j)(1-\delta)}{N} \\ &= \frac{j + (N - 2j + 1)\delta}{N} \end{aligned} \tag{B.1}$$

Hence the transition matrix P is as follows:

$$\begin{pmatrix} \frac{1+(N-1)\delta}{N} & \frac{1-\delta}{N} & \frac{1-\delta}{N} & \frac{1-\delta}{N} & \cdots & \frac{1-\delta}{N} \\ \frac{\delta}{N} & \frac{2+(N-3)\delta}{N} & \frac{1-\delta}{N} & \frac{1-\delta}{N} & \cdots & \frac{1-\delta}{N} \\ \frac{\delta}{N} & \frac{\delta}{N} & \frac{3+(N-5)\delta}{N} & \frac{1-\delta}{N} & \cdots & \frac{1-\delta}{N} \\ \frac{\delta}{N} & \frac{\delta}{N} & \frac{\delta}{N} & \frac{4+(N-7)\delta}{N} & \cdots & \frac{1-\delta}{N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\delta}{N} & \frac{\delta}{N} & \frac{\delta}{N} & \frac{\delta}{N} & \cdots & \frac{N+(1-N)\delta}{N} \end{pmatrix} \tag{B.2}$$

Note that this is identical to the matrix given in (4.5) when $\lambda_{ij} = 1/N$ and $\delta_{ij} = \delta$.

Having constructed the transition matrix for the system, we can now find the expected quality of the solution after a sufficiently large number of generations, such that the influence of the starting state has been eliminated. This can be found by computing the *stationary distribution vector* π of the Markov Chain, defined by the stationary equations as $\pi^T = \pi^T P$. To compute this vector, we first find the expressions for two successive elements of π from the stationary equations.

$$\begin{aligned}
\pi_j &= \frac{1-\delta}{N}(\pi_1 + \pi_2 + \cdots + \pi_{j-1}) + \frac{\delta}{N}(\pi_{j+1} + \pi_{j+2} + \cdots + \pi_N) \\
&\quad + \left[\frac{j + (N - 2j + 1)\delta}{N}\right] \pi_j \\
\pi_{j-1} &= \frac{1-\delta}{N}(\pi_1 + \pi_2 + \cdots + \pi_{j-2}) + \frac{\delta}{N}(\pi_j + \pi_{j+1} + \cdots + \pi_N) \\
&\quad + \left[\frac{j - 1 + (N - 2j + 3)\delta}{N}\right] \pi_{j-1}
\end{aligned}$$

Taking the difference,

$$\begin{aligned}
\pi_j - \pi_{j-1} &= \frac{1-\delta}{N}\pi_{j-1} - \frac{\delta}{N}\pi_j + \left[\frac{j + (N - 2j + 1)\delta}{N}\right] \pi_j \\
&\quad - \left[\frac{j - 1 + (N - 2j + 3)\delta}{N}\right] \pi_j \\
\pi_j &= \left[\frac{(1-\delta)N + (2\delta - 1)(j-2)}{(1-\delta)N + (2\delta - 1)j}\right] \pi_{j-1} \qquad \text{(B.3)}
\end{aligned}$$

Let $\omega_j = (1-\delta)N + (2\delta - 1)j$, i.e., $\pi_j = \frac{\omega_{j-2}}{\omega_j}\pi_{j-1}$ as given in Equation B.3 above. Since $\pi_2 = \frac{\omega_0}{\omega_2}\pi_1$, we find that

$$\pi_j = \frac{\omega_{j-2}}{\omega_j}\frac{\omega_{j-3}}{\omega_{j-1}}\frac{\omega_{j-4}}{\omega_{j-2}} \cdots \frac{\omega_0}{\omega_2}\pi_1 = \frac{\omega_1\omega_0}{\omega_j\omega_{j-1}}\pi_1 \qquad \text{(B.4)}$$

Since π is a distribution vector, by definition $\sum \pi_i = 1$. We can therefore calculate π_1 given N and δ using Equation B.4 above by summing all the elements together and

solving for π_1. This allows us to calculate all the elements of π. We can then calculate the expected quality of the solution generated by SCSA after a sufficiently large number of generations T as:

$$E(F(s_{(T)}) = \sum_{i=1}^{N} F(s_i) \cdot \pi_i \tag{B.5}$$

In the case where the objective function is a ranking function, Equation B.5 is equivalent to $E(R(s_{(T)}) = \sum_{i=1}^{N} i \cdot \pi_i$. This value gives the expected quality of the solution generated by SCSA in the best case, i.e., SCSA cannot expect to achieve a superior result for a given uniform comparison error δ no matter how many iterations of the algorithm is performed.