UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Game-Learning Machine

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Michael Gherrity

Committee in charge:

Professor Paul R. Kube, Chairperson
Professor Richard K. Belew
Professor Walter J. Savitch
Professor Anthony V. Sebald
Professor Maxwell B. Stinchcombe

1993

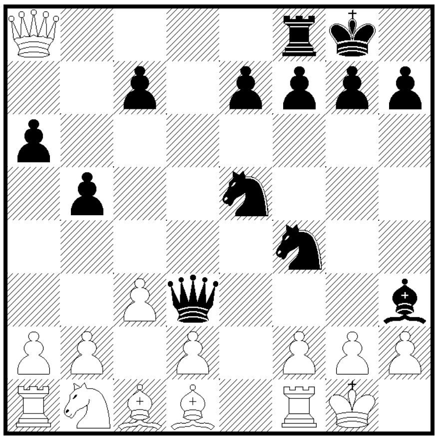
The dissertation of Michael Gherrity is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1993

To all those who have played me in a friendly game of chess. Especially, to my parents, for teaching me the game, my brother Pat and my sister Kathleen, my high school physics teacher Clinton Owen, my childhood neighbors John Wood and William Connell, my wife Ellen, and to many more whose names or games I have forgotten.



	Frank Poole	**HAL9000**
14.	Q×P	B×P
15.	R–K1	...

HAL: I'm sorry Frank, I think you missed it: Queen to Bishop three [six], Bishop takes Queen, Knight takes Bishop. Mate.

15.	...	Q–B6
16.	B×Q	N×B

Frank: Uh, huh. Yeah, looks like you're right. I resign.

HAL: Thank you for a very enjoyable game.

Frank: Yeah. Thank you.

– from the movie 2001: A Space Odyssey

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am especially grateful to my advisor, Professor Paul Kube. For years we would regularly meet and discuss ideas and my progress. I remember once during one of these meetings he proclaimed: "The landscape is strewn with the burning hulks of ideas." He has made a major contribution to this dissertation, both directly and indirectly. I would also like to thank all the members of my committee for assisting me in my efforts to complete this work.

The initial idea for this dissertation arose during an E-mail discussion that occurred in the summer of 1989 between the members of the Cognitive Computer Science Research Group (CCSRG) at the University of California, San Diego. The subject of the discussion was: "What would a computer have to do to convince you that it was smarter than you?" The CCSRG was always an inspiration to me in the early years of my graduate studies, and I would like to thank Professor Rik Belew for organizing and running this group.

This document would never have existed without the love and support of my wife, Ellen Caprio. She helped edit the manuscript, and often provided an objective opinion concerning the direction of the research. She is fond of saying that SAL was the "other woman." I'll take this opportunity to set the record straight. Ellen is my Dulcinea, there is no other.

A special thanks goes to Professor Gary Cottrell. Several times throughout the years he has shown an interest in my research and future, and has reviewed several papers I have written. I would also like to thank Ms. Patricia Naughton. More than once she has saved me from the administration and its heartless rules. It's nice to know that people care.

The results reported in this dissertation were obtained by using many hours of computer time. During its development, the SAL program was run on Apple Macintoshes, SUN computers, Silicon Graphics machines, HP-9000's, a Convex C-240, and a Cray-EL. The availability of all these machines provided relatively quick feedback to determine which ideas worked and which didn't. Such feedback was critical to the progress of this research. Most of this computer time was obtained at the Naval Command, Control, and Ocean Surveillance Center, RDT&E Division (NRaD) under a high performance computing fellowship. I would like to thank the NRaD Center for Advanced Computing for awarding me this fellowship. I would also like to thank Richard Freund at NRaD for allowing me to use some of these machines. In addition, I would like to thank my management at NRaD; Sunny Conwell, Rik Pierson, John Salzmann, and Bob Kolb, for permitting me to have a flexible schedule so that I could perform this work.

VITA

October 21, 1958	Born, Santa Ana, California
1981	B.S. in Physics, Massachusetts Institute of Technology
1981–1986	Lieutenant, Naval Nuclear Propulsion Program, U.S. Navy
1987–1988	Teaching Assistant, Department of Computer Science, University of California, San Diego
1988	M.S. in Computer Science, University of California, San Diego
1988–1993	Computer Scientist, Naval Command, Control and Ocean Surveillance Center, RTD&E Division
1993	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Gherrity, M., A Learning Algorithm for Analog, Fully Recurrent Neural Networks. *Proceedings of the International Joint Conference on Neural Networks*, Vol. I, pp. 643–644, Washington D.C., June 1989.

Belew, R. K. and M. Gherrity, Back Propagation for the Classifier System. *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 275–281, George Mason University, June 1989.

Gherrity, M., Simulating Non-local Systems on Hypercube Machines. *Proceedings of the 1991 Simulation Technology Conference*, pp. 340–344, Orlando, Florida, October, 1991.

Gherrity, M., and P. Kube, Quiescent Search is Beneficial. Technical Report CS93-289, University of California, San Diego, 1993.

Wang, M., W. G. Nation, J. B. Armstrong, H. J. Siegel, S. Kim, M. A. Nichols, and M. Gherrity, Multiple Quadratic Forms: A Case Study in the Design of Data-Parallel Algorithms, accepted for publication in the *Journal of Parallel and Distributed Computing*.

ABSTRACT OF THE DISSERTATION

A Game-Learning Machine

by

Michael Gherrity
Doctor of Philosophy in Computer Science
University of California, San Diego, 1993
Professor Paul R. Kube, Chair

This disertation describes a program which learns good strategies for two-person, deterministic, zero-sum board games of perfect information. The program learns by simply playing the game against either a human or computer opponent. The results of the program's learning the games of tic-tac-toe, connect-four, and chess are reported.

The program consists of a game-independent kernel and a game-specific move generator module. Only the move generator is modified to reflect the rules of the game to be played. The kernel remains unchanged for different games. The kernal uses a temporal difference procedure combined with a backpropagation neural network to learn good evaluation functions for the game being played.

Central to the performance of the program is the consistency search procedure. This is a game-independent generalization of the capture tree search used in most successful chess playing programs. It is based on the idea of using search to correct errors in evaluations of positions. This procedure is described, analyzed, tested, and implemented in the game-learning program. Both the test results and the performance of the program confirm the results of the analysis which indicate that consistency search improves game playing performance for sufficiently accurate evaluation functions.

Chapter I

Introduction

The relationships between machines, games, and thinking has been a topic of many discussions. In 1948, Norbert Wiener [69] considered whether chess playing ability "represents an essential difference between the potentialities of the machine and the mind [page 193]." In 1950, Alan Turing [65] suggested the "imitation game" as a test for whether a machine could think. He even suggested using chess problems as part of this test. Also in 1950, Claude Shannon [59] proposed using chess playing ability as a means of answering the question; "Could a machine be designed that would be capable of 'thinking'?"

After the first electronic digital computer became operational in 1951 [5], discussions concerning machines, games, and thinking were no longer purely academic. Computer programs for playing games became a major focus of research in Artificial Intelligence (AI). The first programs for playing chess played at a novice level. However today, many chess programs play extremely well. The current computer chess champion, DEEP THOUGHT [23], has a U.S. Chess Federation rating of about 2600 (the world champion currently has a rating of 2900 and an average tournament player has a rating of 1500).

Some people contend that a program like DEEP THOUGHT will never be able to become the world chess champion. Nevertheless, it seems clear that this machine plays a good game of chess. This would prove, according to Shannon, that machines are capable of thinking. However, many people remain unconvinced. Referring to a future match between the world chess champion and their next generation program, the authors of DEEP THOUGHT state in [23],

> ... the ingenuity of one supremely talented individual will be pitted against the work of generations of mathematicians, computer scientists and engineers. We believe the result will not reveal whether machines can think but rather whether collective human effort can outshine the best achievements of the ablest human beings.

So after over 40 years of research, it seems that good chess play is not a good test for whether a machine can think. Is there some other test which could be used to answer Shannon's question "Could a machine be designed that would be capable of 'thinking'?"

In this dissertation it is suggested, rather than base the answer on a machine's performance in a single two-player board game (such as chess), that the answer be based on a machine's performance on all such games. Specifically, the machine must be able to learn how to play any two-player board game well by simply playing the game. This idea of a game-learning machine was first suggested by Paul Richards [48] and Marvin Weinberg [68] in 1951 . Richards states in [49],

> Can one conceive of a machine that has absolutely no initial built-in knowledge but does have an "intelligent" ability to learn almost any game through experience alone?

I.A Why Play Games?

> Newton couldn't have discovered his laws of motion if he had concentrated on trying to understand the laws governing waterfalls or hurricanes. Instead, he boiled the problem of motion down to the most pristine case he could imagine – planets coasting through a vacuum.
>
> – Douglas Hofstadter [22, p. 566]

Typical board games require a player to make a sequence of decisions in a limited amount of time. There is usually insufficient time to investigate all the possible consequences of these decisions. Once made, the decisions cannot be revoked. The outcome is uncertain since the actions of the opponent are not always predictable.

This characterization of games would also apply to a number of decision problems many consider to require intelligence. Examples of decision problems can be found in natural language understanding, scene analysis, mathematical theorem proving, and information retrieval. Trying to understand such problems using these example domains is analogous to trying to understand the laws of motion using waterfalls or hurricanes. Board games are pristine in that they are simple to implement on a computer, have a clear-cut criteria for success or failure, and do not require a large database of facts. Defending his work on developing a chess-playing program, Shannon [58] states

> The chess machine is an ideal one to start with for several reasons. The problem is sharply defined, both in the allowed operations (the moves of chess) and in the ultimate goal (checkmate). It is neither so simple as to be trivial nor too difficult for satisfactory solution. And such a machine could be pitted against a human opponent, giving a clear measure of the machine's ability in this type of reasoning.

Perhaps the most distinguishing feature of decision problems is their combinatorial nature. Specifically, even though it is theoretically possible to exhaustively search through all sequences of actions for a solution, such a search is not feasible since there are exponentially many sequences. For example, in [59] Shannon estimates there are about 10^{120} different games of chess. Since it is not possible to exhaustively search for a solution, some other method is required for finding adequate answers to these problems. A two-player board game such as chess presents an experimental domain where methods for dealing with these combinatorial problems can be studied in isolation from the noise of other issues.

Developments toward solving the chess-playing problem have proven to be applicable towards the solution of many more practical problems. The technique of searching a state space was first proposed by Shannon, Turing, and Wiener for the game of chess. In their series *The Handbook of Artificial Intelligence* [6], Barr and

Feigenbaum consider such search methods to constitute some of the core ideas of AI [volume 1, p. 21]. Applications which have used search methods include robot planning, visual scene analysis, mathematical theorem proving, symbolic integration, and puzzle solving. In [59] Shannon listed a number of such applications, stating,

> Machines of this general type are an extension over the ordinary use of numerical computers in several ways. First, the entities dealt with are not primarily numbers, but rather chess positions, circuits, mathematical expressions, words, etc. Second, the proper procedure involves general principles, something of the nature of judgment, and considerable trial and error, rather than a strict, unalterable computing process. Finally, the solutions of these problems are not merely right or wrong but have a continuous range of "quality" from the best down to the worst.

Although advances in computer chess playing may have no immediate practical use, this problem domain has provided an excellent laboratory for developing many useful techniques and ideas. It is reasonable to expect that the game-learning domain will also prove fruitful.

I.B Selective Search and Brute Force Search

> Chess is the Drosophila of Artificial Intelligence.
>
> – Alexander Kronrod [33]

Typical chess-playing programs select a move by searching through thousands of moves, countermoves, counter-countermoves, etc., forming a vast game tree of possible board positions. For example, the DEEP THOUGHT program examines over 500,000 positions per second [4].

To ensure that key moves are not missed, every legal move from each position in the tree must be considered. The number of moves in any sequence of moves starting from the current board position (the current board position is called the *root* of the game tree) is referred to as the *depth* of the game tree. The search proceeds uniformly and incrementally in depth, where all the board positions that result from sequences of two moves are examined, then all the three move sequences, etc. Each additional move added to the depth of the search tree is called a *ply*. The DEEP THOUGHT program generally searches over 10 ply before making a move [23].

Several efforts have been made to grow the search tree in a more selective fashion [4, 50]. A *selective search* would determine which sequences of moves should be considered based on characteristics of the board positions examined in the game tree. Promising lines of play would be continued to great depth, whereas bad lines would be terminated early to avoid wasting time. An effective selective search procedure could be applied to problems more complicated than the game of chess. From each chess position there is usually about 35 legal moves. This is called the *branching factor*. A selective search procedure could conceivably be applied to problems with a much larger branching factor.

Contrary to a selective search procedure, the search procedures of the most successful chess-playing programs can be characterized by a uniform, incremental increase in the depth of the search tree. Such a search is conducted virtually independent of the characteristics of the board positions examined, and includes a large number of board positions. These search methods have been referred to as *brute force*.

When chess programs were first being developed, the speed of the available computers would achieve depths of less than four ply using a brute force search under tournament conditions [44]. It was found that a four ply search resulted in poor chess play. It became clear that some method of selective search was required to achieve better play. Since brute force search was ineffective in chess, researchers at the time came to the conclusion that a successful method of selective search would have to be discovered before a program would be able to play a good game of chess. The game of chess became the test domain of choice for new selective search algorithms.

Since the early 1950's, computer speeds have increased by over 4 orders of magnitude. A brute force search can now extend to over 10 ply in the game of chess [23]. A program searching to such a depth can beat all but the best chess players. In 1977, a program performing a brute force search became the best chess-playing program in the world [60]. Since that time, brute force search has continued to perform much better than any known selective search technique.

The assumption that a successful method of selective search must be found before a program would be able to play a good game of chess has been proven to be incorrect. The tremendous increase in computer speeds was apparently unforeseen when this assumption was made. A brute force search is now able to achieve enough depth to challenge even the best human chess players. However, for problems more complicated than chess (i.e., problems with a branching factor larger than about 50) a brute force search would once again be ineffective. Research in selective search algorithms may prove to be more fruitful than attempts to significantly increase computer speeds. The game-learning domain, where games with very large branching factors are included, would be a good domain in which to test new selective search algorithms.

I.C Game Learning

A common criticism of chess-playing machines is that they can only play chess, and have limited abilities to modify their playing style. It is argued that a thinking machine would not be so limited. In his book *Pattern Recognition, Learning, and Thought* [67], published in 1973, Leonard Uhr addressed the game-playing problem.

> What I am arguing is that, very simply, games are far harder than they may look on the surface, and that, further, we are not yet ready to handle them properly. The fact that we have achieved some success attests more to the determination and the cleverness of the programmer-analyst who hit upon ingenious methods for representing his problem and powerful characterizers with which his program could assess situations appropriately. Because the problem of an interesting game is much larger, *if* it is attacked without an appropriate set of tools (that is, appropriate representations and characterizers and methods for building appropriate characterizers), it is possible to handle it only by ingeniously designing a very special-purpose structure, one appropriate to that game. But this is just what we *don't* want; for we want to be developing general principles

> and theories, moving toward more and more powerful, rather than more and more specific, mechanisms. [p. 227]

This tendency of researchers to develop special-purpose mechanisms for playing chess seems reasonable since the stated problem has been to develop a chess-playing machine. However, the game-learning problem would not encourage this approach, since any special-purpose mechanism would probably not work for a large number of different games.

I.D Two-Person, Deterministic, Zero-Sum Board Games of Perfect Information

The domain of all games is exceedingly large. Aside from the common parlor games such as chess and checkers, it extends to esoteric games such as Nomic [22, pp. 70–86], where the rules of the game are constantly changed by the players. There are games where players are better off cooperating, such as in the prisoner's dilemma; games where each player is presented with different information about the state of the game, such as in card games; and games that have a random element, like backgammon.

To limit the scope of this dissertation, only a sub-domain of the domain of all games was considered. To minimize the amount of user-interface code that had to be written, only board games were addressed. Further simplification was achieved by limiting the domain to two-player, deterministic games of perfect information. Finally, only zero-sum games were included. This choice of domain was also influenced by the availability of computer programs which could be used as opponent players for training.

A significant concern was that such a drastic simplification of the domain, from learning any game to learning two-player, deterministic, zero-sum games of perfect information, would result in a domain so small that methods that are developed for its solution would not scale to larger domains. This has been an insidious problem in AI research. A method is shown to work well in a small domain, but when tried in the larger domain requires excessive computational resources to yield a practical solution. Such domains have been termed *micro-worlds* [15].

It is difficult to defend any choice of domain from the micro-worlds criticism. However, the chosen domain includes games like checkers, chess, othello, and go. These are difficult games for humans to master, and a program capable of learning to play well in all such games would seem to require rather general learning and reasoning abilities.

I.E Rule Learning Versus Strategy Learning

Some consideration was given to whether the rules of a game should be learned in addition to good strategy. Since the program is only permitted to learn from actually playing the game, the rules could be learned by playing the game and being told when a move was illegal. Information regarding legal moves could also be obtained by observing the opponent's moves (assuming the same rules applied to both players). In addition, criteria for winning the game would be available from the final board position of each game played.

A game-learning program was developed which learned the rules of a game using simple linear discriminant functions. The Perceptron Convergence Theorem [16] can be used to argue that given sufficient information about the board position and move, the rules of any game could be learned after a finite number of illegal moves. Information about a board position and move is sufficient if legal moves can be linearly separated from illegal moves. A program using the perceptron learning algorithm was able to learn the rules of tic-tac-toe after several thousand games.

The rules of most games are fairly simple. Hence a learning procedure that requires a finite number of illegal moves may actually be practical. However, a good strategy can be quite complex, even for games with simple rules. For example, the game of go has very simple rules and a simple criterion for winning. However, this game is thought by many to be more difficult than chess. Since good strategies for games tend to be much more complex than the rules of the game, it is doubtful that a linear approach, such as the perceptron, could achieve good play in a reasonable number of games.

An important difference between learning the rules of a game and learning good strategy is that no amount of computation can ever determine whether a novel move is legal. However, searching the game tree can determine whether a novel move is strategically beneficial. A program which can only learn from actually playing the game would either have to avoid making any move it has not already observed as legal in a previous game, or make novel moves and risk them being illegal. However, a good strategy could conceivably be learned by searching the game tree before a single move has been made.

It may be possible for a learning procedure to generalize across numerous games and become very good at determining the rules of a new game. However, this type of learning seems very different than what would be required to learn a good strategy for a game in which the rules are known. Alternatively, a program could become quite good at a game by making only moves that have previously been observed to be legal in past games, even though some legal moves were not being considered. However, it would be difficult to quantify the performance of such a program. The programmer could say: "Sure it lost the game, but it didn't know that castling to the queen side was legal!" To avoid these difficulties, the game-learning domain considered in this dissertation does not include learning the rules of the game.

I.F Overview

This dissertation describes one attempt at designing a program capable of learning to play two-player board games well by simply playing the game. Chapter II describes the reasoning behind many of the decisions that were made in the design of the program. Chapter III presents the search procedure used by the program and gives an analysis of the procedure showing that it improves game playing performance under fairly general conditions. Chapter IV describes the game-learning program SAL. The performance of SAL playing the games of tic-tac-toe, connect-four, and chess is reported in Chapter V. Finally, Chapter VI presents some concluding remarks.

Chapter II

Search and Learning

There have been many programs designed to play board games that do not incorporate any learning. Virtually all of these programs choose a move to make based on a search through a tree of possible moves and countermoves. The nodes of such a tree correspond to board positions, and the branches to legal moves. Since there is rarely time during a game to extend this search all the way to terminal positions (positions known to be either won, lost, or drawn), these programs use an *evaluation function* on positions at the leaves of the search tree. The value returned by the evaluation function is an estimate of the outcome of the game if that position was actually reached during play.

Some of the machines designed to play the game of chess, for example, are capable of searching through millions of nodes in several minutes. It is common to measure the searching ability of these programs in terms of *ply*. A ply is one level of the game tree, and corresponds to a move by one player. If all possible legal moves are considered, then the search is *full-width*. Hence a full-width, one-ply search from a given board position would generate all the possible board positions that result from a legal move from the given position, and evaluate each of these positions. The move that leads to the position which has the best evaluation would be the move chosen. A full-width, two-ply search would generate all the positions that result from a legal move from the given position, as well as all the positions that result from these generated positions.

There were several early attempts to learn simple games using rote-learning methods [49, 31], but these were not extended to more complex games. Rote-learning has been used in a competition chess program [56], but the learning was used only to prevent the program from repeating the same moves in human tournaments.

More recently, a game-learning program named HOYLE was developed by Susan Epstein [17]. HOYLE is provided with a variety of general game playing methods and uses various rote-learning techniques to improve the performance of these methods for specific games. In general, the program performs only a one-ply search. HOYLE has demonstrated an ability to learn a number of simple games, but its ability to learn more complex games such as chess is questionable.

Bruce Abramson [1] proposed a method for training evaluation functions which could be used for arbitrary games, however the functions were only used in one-ply searches during actual play, and the training was not done during play. All other programs can also be characterized as learning an evaluation function, but have only been designed to learn a specific game. For example, Arthur Samuel [53] and Arnold Griffith [20] designed programs that learned the game of checkers. Other board games in which learning programs have been designed include othello (Lee and Mahajan [27, 28]), backgammon (Tesauro and Sejnowski [62, 63]), go-moku (Yakowitz [70]), and chess (Zobrist and Carlson [71], Levinson and Snyder [30], and Nitsche [45]). Most of these programs only perform a one-ply search.

This chapter will describe some of the difficulties with using only a one-ply search, as well as some of the difficulties with using a deeper search. In addition, the problems with relying on learning methods will be described. Much of the design of the SAL program, discussed in later chapters, is based on the reasoning described in this chapter.

II.A The Need for Search

But for any problem worthy of the name, the search through all possi-

> bilities will be too inefficient for practical use. And on the other hand, systems like chess, or nontrivial parts of mathematics, are too complicated for complete analysis. Without complete analysis, there must always remain some core of search, or "trial and error."
>
> – Marvin Minsky [35]

> It appears that the clue to intelligent behavior, whether of men or machines, is *highly selective search*, the drastic pruning of the tree of possibilities explored. *For a computer program to behave intelligently, it must search problem mazes in a highly selective way, exploring paths relatively fertile with solutions and ignoring paths relatively sterile.*
>
> – Feigenbaum and Feldman [18]

In board games of perfect information, the current board position provides all the information necessary for the player whose turn it is to more to choose the correct next move. Theoretically then, a sufficiently accurate evaluation function could determine whether any board position will lead to a win, lose, or draw based only on the positions of the pieces on the board. The program would simply evaluate the board positions that result from each legal move, and then choose the move that leads to a won position. No positions other than the ones generated by this full-width, one-ply search would have to be evaluated.

Although it might be possible to develop, or even learn, such an accurate evaluation function in some games, there are numerous examples in the chess literature that illustrate how difficult this would be in the case of chess. Donald Michie, in [34], shows two board positions, reproduced here as Figures II.1 and II.2, that are identical except for the position of one pawn. In Figure II.1, d3h7 is a winning move, whereas in Figure II.2 this move loses. It seems unreasonable to expect an evaluation function, using only the locations of the pieces and a one-ply search, to be accurate enough to correctly evaluate these two positions.

These kinds of examples strongly suggest that an evaluation function alone will probably not be accurate enough for good play. Some form of a search beyond one-ply seems necessary. Such a search is described by the algorithm developed for the SAL program (found in Chapter III).

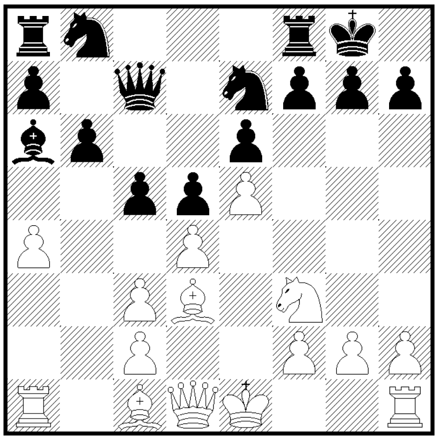


Figure II.1: White to move. The bishop sacrifice is sound.

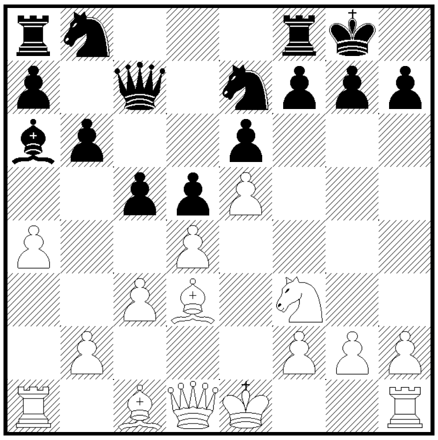


Figure II.2: White to move. The bishop sacrifice is unsound.

II.B Search Pathology

A common assumption, and one implicit in the previous section, is that a deep search of the game tree leads to a better choice of move than only performing a one-ply search. This assumption is based partly on the notion that a deep search reveals winning combinations and traps, and leads to positions that can be more accurately evaluated. This assumption was made by Samuel for his checkers-learning program [53], since the results of a deeper search were used as training values for the evaluations of the positions generated at the first ply.

Mathematical analysis by both Dana Nau [36, 37] and Don Beal [10] indicate that the assumption that a deeper search yields a better choice of move is not correct for many games. In fact, their analyses showed the opposite is true, that a deeper search results in worse play. The results are based on analyzing the propagation of evaluation function errors up the game tree using the standard minimax procedure. It is found that these errors increase as the depth of the search increases.

Contrary to these mathematical results, experience with chess playing programs has shown that the deeper the program searches, the better it plays [23]. Since this seems to be the case for most common games, games where deep search is not beneficial have been called *pathological*. The implication is that typical games like chess are not pathological, whereas games for which it can be shown that standard game tree search is ineffective are in some way contrived and unusual [24].

Many researchers [47, 14, 10, 57] have modified the mathematical assumptions used by Nau and Beal in an attempt to explain why games like chess are not pathological. None of these modifications seem compelling. It was assumed for this dissertation that the original mathematical assumptions were correct for many of the positions that occur in a typical game like chess, and that the standard game tree search procedure may be doing more harm than good when selecting a move from these positions. As a result, a modification of a search procedure proposed by Don Beal [9], called consistency search, was designed and shown to be beneficial even for pathological games. The description and analysis of the consistency search procedure is given in Chapter III.

II.C Learning an Evaluation Function

There have been two main approaches to learning an evaluation function: learning from positions that have occurred in expert games [53, 20, 28, 62], and learning by playing the game [53, 63, 30]. Learning from positions that occur in expert games is an off-line training procedure. The evaluation function is isolated from the game-playing portion of the program and is trained to mimic the decisions made by expert human players. Since the moves made by the human experts from each position in the game are known, these moves are considered to be the best moves and are used to train the evaluation function.

The domain chosen for this dissertation, specified in Chapter I, excludes training the evaluation function from positions that have occurred in expert play, and requires the program to learn from actually playing the game. There are several reasons for this:

- The technique of learning from expert games assumes the availability of a large number of documented expert games. Although such a database exists for games like chess, they do not exist for more obscure games.

- This technique assumes that human experts make the best moves. This is not at all clear, even for games as old as chess. For example, in 1986, a computer retrograde analysis of chess endgames discovered that most KQKBB and KQKNN endgame positions can be won, whereas it was previously thought that such positions were draws [64].

- A program which only learns from the games of better players, and cannot learn from its own games, raises questions about how the better players learned to play so well. This could indicate that the program is doing something else besides what most people would consider "thinking."

Chapter V presents the results of the SAL program playing the games of tic-tac-toe, connect-four, and chess against imperfect computer opponents.

II.C.1 The Credit Assignment Problem

> It is extremely doubtful whether there is enough information in "win, lose, or draw" when referred to the whole play of the game to permit any learning at all over available time scales... For learning to take place, each play of the game must yield much more information.
>
> – Alan Newell [43]

In a game of chess, a player might make over one hundred moves before the outcome of the game is known. Even though the player might have won the game, some of the moves may have been poor. Perhaps the opponent was simply unable to capitalize on the mistakes. To learn from the game, the player must somehow divide the credit for the win among the moves made, ideally giving more credit to the good moves, and less to the bad. Unfortunately, there can be no teacher to identify which moves are which.

A program which only learns from playing games must solve this credit assignment problem. Samuel [53] developed an algorithm for his checkers-learning program that provided a training value for each position that occurred in the game. This training value was given by the evaluation of the position after the next move. A modified version of this procedure, called the temporal difference method, was formalized and analyzed by Richard Sutton [61]. He proved convergence and optimality for this procedure in the linear case.

The temporal difference method was used by Gerald Tesauro in his backgammon-learning program [63]. He found that the program was able to learn to play backgammon at a "fairly strong intermediate level of performance" which surpassed the performance of the program when trained on human expert games. Chapter IV describes the use of the temporal difference method for the SAL program.

II.C.2 The Feature-Discovery Problem

In the game of chess it has been observed that the player with the highest material value tends to win the game. One of the first things beginning players are taught is the material values of the pieces; 1 for a pawn, 3 for a knight and bishop, 5 for a rook, and 9 for a queen. The total material value of each player in a given board position is an example of a *feature*. Other features in the game of chess include piece mobility, center control, and piece development [21]. Chess players have observed a strong correlation between the eventual outcome of a chess game and the values of these features for positions that occur in the game.

In computer programs designed to either play or learn a specific game, the programmer usually decides what features are to be computed from the board position. In a program designed to learn arbitrary games, however, the programmer is not able to choose the features since it is not known what game the program will be playing, and each game is different. For example, in the game of go a feature might be the number of empty squares (liberties) a player controls. However, the number of empty squares is not generally considered an important feature in chess since it is not a good predictor of the outcome of the game. Since it is not possible to compute all possible features of all possible games, a game-learning program must discover its own features relevant to the game currently being played.

The feature-discovery problem was recognized by Samuel during the design of his checkers-learning program [53]. For this program, Samuel choose a group of 38 features that might be relevant to the game of checkers, and designed a learning algorithm that would select the 16 best features from among this group. The evaluation function would use these 16 features. Samuel stated

> It might be argued that this procedure of having the program select terms for the evaluation polynomial from a supplied list is much too simple and that the program should generate the terms for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised.

Neural network learning algorithms have been shown to discover features in many simple problem domains [51]. Since it is assumed that only games of perfect information will be played, the current board position must contain sufficient information to allow the best move to be selected. Assuming that the learning algorithm for the evaluation function can discover its own features, the algorithm would only need the *raw* board position as input. A raw board position is some simple encoding of the positions of each piece on the board into a set of numbers to be given to the evaluation function. The backgammon program developed by Tesauro and Sejnowski [62, 63] uses only the raw board position as inputs to a neural network evaluation function.

A problem with only using the raw board position as inputs to the evaluation function is that it requires the learning algorithm to solve the very difficult feature-discovery problem. Not only is it questionable whether existing learning algorithms are capable of discovering good features to complex problems, but such discovery could require a very large number of training examples. The approach used for the SAL program was to generate a set of features using the rules of the game being played. Provided these features can be computed based on the rules of any board game, such features would not be specific to any game. In this way, the feature-discovery problem becomes a little more like the easier feature-selection problem considered by Samuel. Chapter IV describes how these features are computed in SAL.

II.D Combining Search with Learning

This section presents a qualitative discussion of the potential benefits of using an effective game tree search procedure to enhance an evaluation function learning algorithm. The discussion compares the use of the consistency search procedure, to be presented in Chapter III, with a one-ply search. The figures presented in this section are qualitative and are only intended to motivate the analyses and experimental results presented in subsequent chapters. It will be shown in Chapters III and V that the analytical and experimental results agree with this qualitative argument.

Procedures used for training an evaluation function gradually improve the accuracy of the function as more training examples are presented. One measure of the number of training examples presented to a game-learning program is the number of games played. Suppose the evaluation function error for a given learning algorithm were to decrease as shown in the top graph of Figure II.3. Note the diminishing decrease in the error as more training examples are presented.

Of interest is the probability of making a correct move from some position given an imperfect evaluation function. The middle graph of Figure II.3 shows one possible curve for this probability as a function of the evaluation function error. The curve shown results if only a one-ply search is performed. The specific equation for this probability is derived in Chapter III.

The bottom graph of Figure II.3 combines the first two graphs to give the probability of making a correct move as a function of the number of games played. Note that the diminishing decrease in the error as more games are played, shown in the top graph, leads to a diminishing increase in the probability of making a correct move. Assuming that the probability of making a correct move is proportional to playing strength, the improvement in the playing strength of the game-learning program will also diminish as more and more games are played. Hence if only a one-ply search is performed, a diminishing decrease in the evaluation function error would result in a diminishing improvement in the playing strength of the program.

Now suppose that a game tree search procedure is used in addition to an evaluation function learning procedure. Figure II.4 presents the same three graphs shown in Figure II.3 except that a consistency search is performed. Since it is assumed that the same learning algorithm is used, the top graph of Figure II.4 is identical to the top graph of Figure II.3. However, the middle graph of Figure II.4 shows the probability of making a correct move as a function of the evaluation function error after a consistency search is performed. The bottom graph shows the combination of the first two, and indicates the improvement in the strength of the program as the number of games increases.

If the evaluation function can be trained to make perfect evaluations, then the program will play perfectly whether or not a consistency search is performed. However, it was argued at the beginning of this chapter that a perfect evaluation function is an unreasonable expectation for complex games like chess. A comparison of Figure II.3 with Figure II.4 indicates that for an erroneous evaluation function, a game-learning program using a consistency search would eventually play better than a program using only a one-ply search.

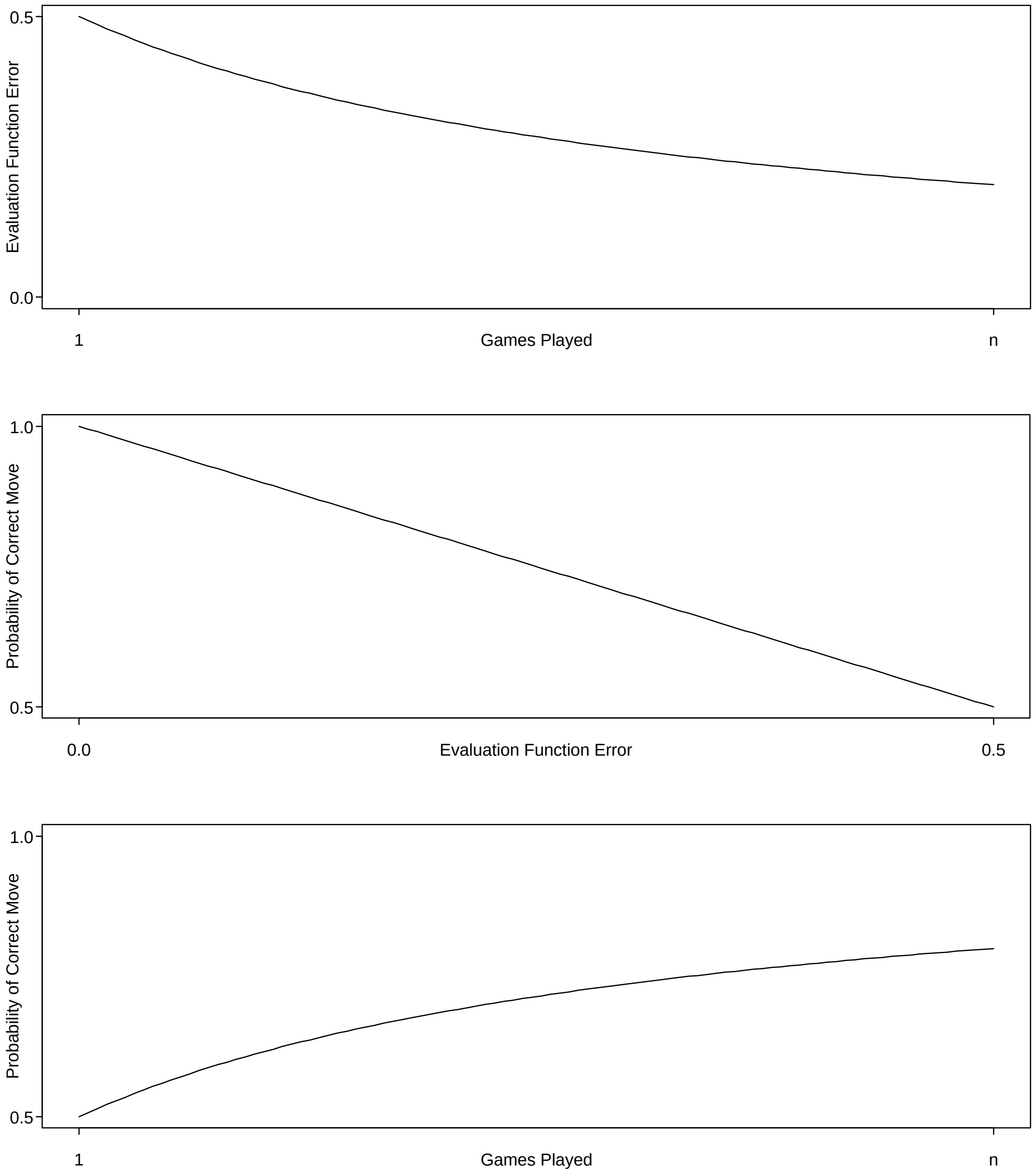


Figure II.3: Qualitative relationships between the probability of making a correct move, the evaluation function error, and the number of games played, when only a one-ply search is performed.

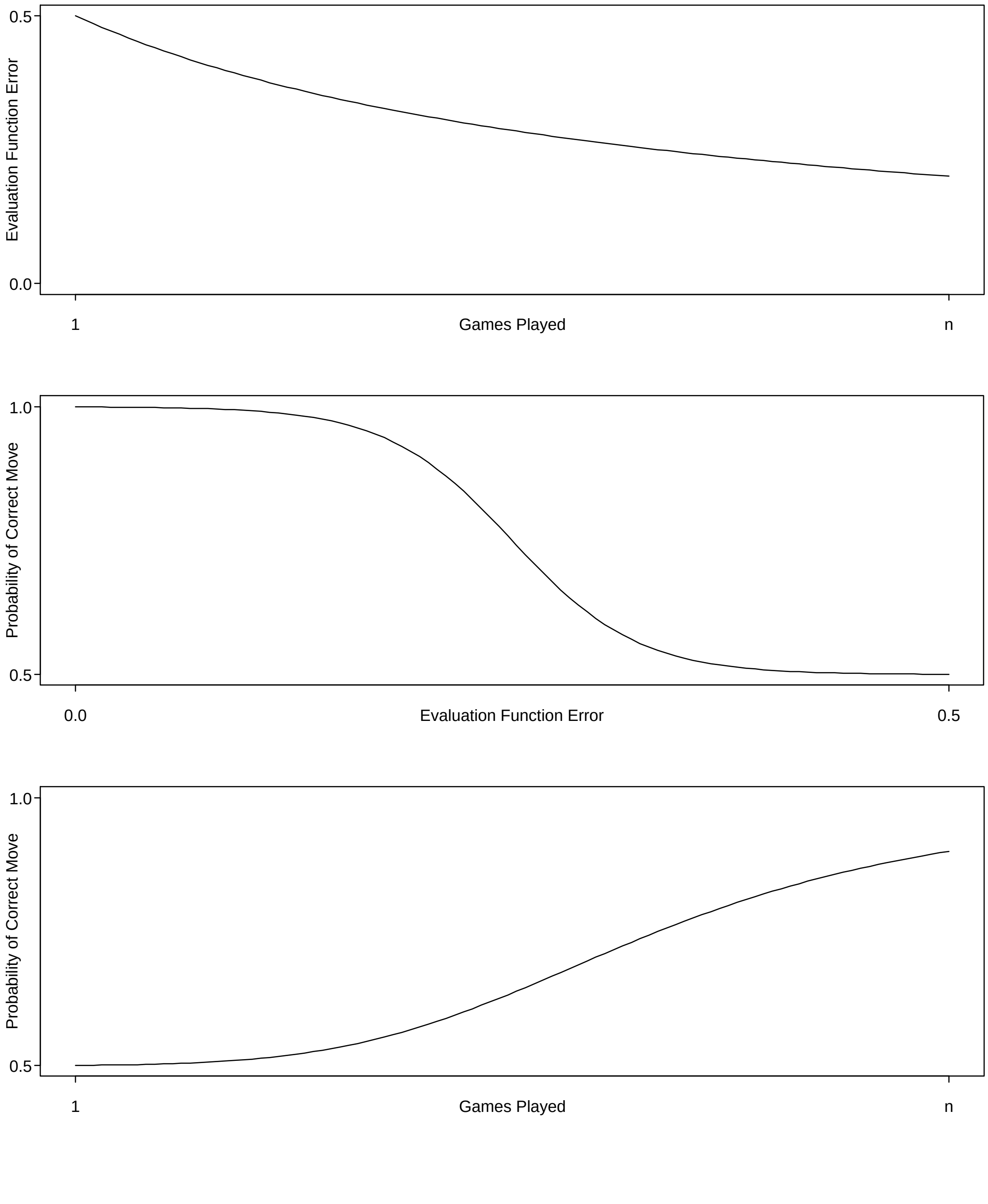


Figure II.4: Qualitative relationships between the probability of making a correct move, the evaluation function error, and the number of games played, when a consistency search is performed.

Chapter III

Consistency Search

This chapter describes and analyzes the search procedure used by the SAL program. This procedure is called *consistency search* since it shares the basic principle behind the consistency search procedure proposed by Don Beal [9, 11]. There are several important differences between the procedure described here and Beal's procedure however, and these will be highlighted.

Since the SAL program is designed to gradually learn a good evaluation function for the game being played, it must be expected that the evaluation function will make mistakes in the evaluation of some board positions. It would be desirable that in the event of an evaluation error, the search procedure would be able to choose a good move anyway. However, analytical results of traditional search procedures indicate that under some conditions these search procedures may actually degrade the ability of the program to choose a correct move when the evaluation function is erroneous. The consistency search procedure was designed in an attempt to ensure that the search would be beneficial even for an imperfect evaluation function.

This chapter describes the consistency search procedure and the analysis of its performance. After some background discussion, a description of the procedure is given for the case of a bi-valued evaluation function in Section III.D. This is extended to the case of a continuous evaluation function in Section III.E. An outline of an analysis of the procedure is given in Section III.F. Finally, Section III.G compares the analytical results with the qualitative results of Chapter II and describes the implementation of the procedure used in the SAL program. Section III.H contains the details of the analysis.

III.A Analyses of Traditional Game Tree Search

Computer chess programs using a full-width, fixed-depth, minimax search procedure show improved performance as the depth of the search is increased [23]. However, analyses of this procedure performed by Beal [9], Nau [36, 37, 39], and Pearl [47] show that the performance should actually decrease with increasing search depth for many types of games. Such behavior has been termed "pathological."

The original analyses by Beal, Nau, and Pearl assume a uniform game tree with random, independently assigned terminal outcomes. Beal [10], Nau [37, 41, 40], and Bratko and Gams [14] show that various dependencies between terminal outcomes can prevent pathology. Similarly, Pearl [47] showed that game trees which contain a number of short move sequences that lead to terminal positions, called "traps," are not pathological. Schrüfer [57] showed that trees where every won node has at least two lost children are not pathological.

Although there are certainly chess positions with traps or dependencies, it is not certain that every chess position has these features. In light of the analytical results, it is questionable whether a full-width, fixed-depth, minimax search is beneficial from all chess positions. For many positions, the quality of play observed in chess programs may well be in spite of the search they perform, rather than because of the search.

In addition, a game-learning program must be designed to learn arbitrary games. Suppose it was given the task of learning a good strategy for a game that was pathological. If the program used a full-width, fixed-depth, minimax search, then its performance on this task would be degraded due to the search. The consistency search procedure was designed to be beneficial even for pathological games.

III.B Alternative Search Procedures

There have been many different search procedures that have been proposed as an improvement over the traditional full-width, fixed-depth, minimax search. Only those procedures that are game independent are potentially useful for a game-learning program. For example, the quiescent search procedure proposed by Don Beal [12] assumes that it is illegal for a player to pass. However, for a game such as go passing is a legal move, hence a game-learning program using quiescent search would perform poorly on the game of go.

There have been several search procedures which attempt to achieve better play by selectively growing the search tree based on the values of the nodes already evaluated. These procedures include SSS* [47], conspiracy number search [32, 55], min/max approximation [50], singular extension [23], and $\alpha\beta$ conspiracy number search [2]. Such procedures are not directly applicable to the game-learning task since they assume an error-free evaluation function.

Palay [46] proposed a search procedure where the evaluation function returns a probability distribution as the value of a position rather than a single number. This idea was extended by both Russell and Wefald [52] and Baum and Smith [8] to include information about the costs and potential benefits of further node expansions. Although these methods claim various types of optimality, they are not usable for the game-learning task for several reasons:

1. The probability distributions are computed using a small number of features chosen by the programmer. Such features are not available for the design of a game-learning program (see Chapter II). Even interpreting the components of the raw board position as features would result in a very large number of features (a raw board position for the game of chess would have several hundred components).

2. Vast numbers of games are required to compile sufficient statistics to compute good probability distributions. Various off-line procedures are suggested for

collecting this data. Such methods are not practical for a game-learning program that can only learn from actually playing the game.

The incremental negamax algorithm described by Ingo Althöfer [3] is proven to exponentially decrease the effects of the evaluation function errors on the value of the root node as the depth of the search is increased. Although this algorithm is effective for erroneous evaluation functions, it does not perform a selective search. The game tree is grown using the full-width, incremental depth procedure used in traditional game-playing programs. Even if the alpha-beta search procedure is used, the number of nodes in the search tree grows exponentially with the depth of the tree. It is not known whether the exponential decrease in the error of the root value will exceed the exponential increase in the number of nodes in the search tree, particularly for arbitrary games.

III.C Quiescence

In chess playing programs, an important part of the search procedure is the capture tree. After the search reaches some fixed depth, and before the evaluation function is applied, an additional search of all captures is performed. The evaluation function is only applied to positions at the leaves of this capture tree. This ensures that only positions that are tactically quiet, or *quiescent*, are evaluated by the evaluation function.

The prototypical example of a non-quiescent position is a position that occurs in the middle of a queen exchange. Even though the result of the exchange is materially neutral, an evaluation function which considers material value would erroneously evaluate the position as being a queen down for the player whose turn it is to move. A capture tree search would not evaluate this position, but would extend the search to the position after the second queen is taken before applying the evaluation function.

Analytical evaluations of search procedures have not explicitly modeled the capture tree search. Rather, it has been assumed that this search is part of the node evaluation procedure. This procedure, which includes the capture tree search, acts as an evaluation function that can be applied to every node in the tree, not just quiescent nodes. This allows the analysis to assume a fixed-depth search, rather than the variable depth that results from the capture tree.

The importance of quiescence was realized by Shannon [58] and Turing [66] and experimentally verified by Gillogly [19] for the game of chess. Unfortunately, this procedure has only been shown to be beneficial in games like chess. There are many games in which a capture is not even a legal move, and it is difficult to identify a quiescent position.

III.D The Consistency Search Procedure

The consistency search procedure proposed by Beal [9] was intended to reduce the error in the value of a node by selective search. A consistent position is defined as one where the evaluation value of the position is equal to its minimax value after a one-ply search. The search proceeds depth-first, stopping only at consistent positions. Beal observed that this procedure, when applied to the game of chess using an evaluation function based on material, would generate a capture tree. Hence the consistency search procedure is a game-independent generalization of the capture tree.

Beal's algorithm is modified in this dissertation by considering the search as a means of identifying and correcting errors in the evaluation values of positions. Specifically, it is assumed that there is a single erroneous evaluation that causes an inconsistency, and the search is continued for only those children whose evaluation values may be erroneous. The search is used to correct the evaluation function error. The consistency search procedure described in this section is a modified version of Beal's procedure to enable the search to correct errors in the evaluation values of positions.

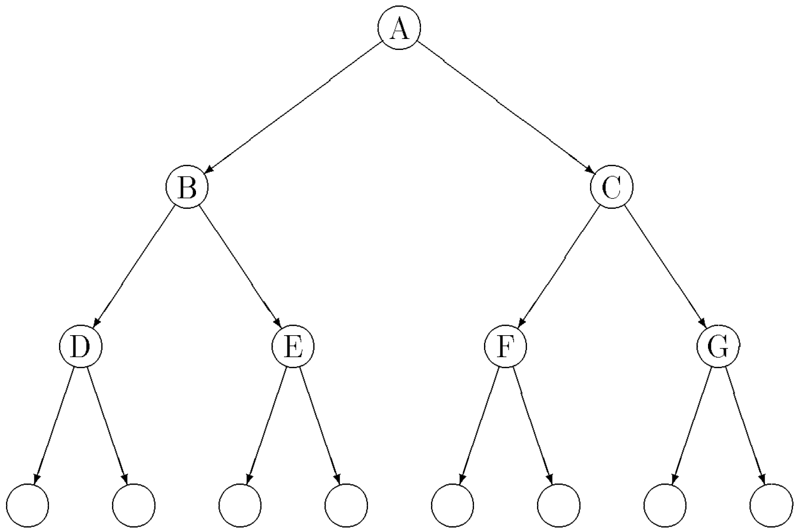


Figure III.1: A Binary Game Tree — The nodes indicate different positions and the links indicate possible legal moves.

Consider a two-player game where players alternate moves. Suppose the state of the game can be described by a "position," and there are a set of rules for moving from one position to another. Assume the game ends after a finite number of moves and one of the two players is the winner, the other the loser. All the possible moves of such a game can be characterized by a game tree [42]. A portion of such a tree for a game with only two legal moves from each position is shown in Figure III.1.

The consistency search procedure assumes there is an evaluation function f, which is a function of a position. This evaluation function returns a 1 if a position is estimated to be a won position for the player whose turn it is to move, and a 0 if it is estimated to be a losing position. If this evaluation function were always correct, then the decision of which move to make could be easily made by applying f to all the positions which are legally possible and moving to a position which returns a 0 (if none of the positions return a 0, then any move could be chosen at random). Since f is assumed to be imperfect however, a consistency search of the game tree is performed.

From the root position (node A in Figure III.1), all the legal positions (nodes B and C) are generated and evaluated by the evaluation function f. The search procedure determines whether these positions are consistent by generating and evaluating all the legal moves from these two positions (nodes D, E, F, and G). The evaluation values of nodes D and E are propagated up the tree to node B using the negamax procedure [26]. If the negamax value of node B is equal to its evaluation value, node B is considered consistent, and no further searching from this position is performed. Similarly, the evaluation values of nodes F and G are used to determine whether node C is consistent.

Suppose node B is not consistent. Then the evaluation value of node B does not agree with its negamax value after evaluating nodes D and E. Hence, at least one of the evaluations must be incorrect. The consistency search procedure assumes there is only one incorrect evaluation and attempts to correct the error by *expanding* the nodes that might be in error. A node is expanded by generating and evaluating some of its children. There are two possibilities depending upon the evaluation value of node B.

If the evaluation value of node B is 1, then all of its children must have an evaluation value of 1 (otherwise node B would be consistent). All of the children (nodes D and E) are expanded by generating and evaluating all their children in an attempt to determine which evaluation (either B, D, or E) is incorrect.

However, if the evaluation value of node B is 0, then at least one child must have an evaluation value of 0 (otherwise node B would be consistent). Since it is assumed that there is only one evaluation error, the error must occur at one of the nodes with a 0 evaluation value. If only one child has a 0 evaluation value, then that child is expanded. However, suppose both nodes D and E have 0 values. Then either node B is evaluated incorrectly, or *both* nodes D and E are evaluated incorrectly. The single error assumption implies that in this case, node B is evaluated incorrectly. The search is terminated and node B will be called a "coerced" node and its value taken as 1.

Technically, the number of children with 0 evaluation values necessary to coerce the parent is two. Let q denote this number. The value of q effects the depth of the consistency search. The following analysis of this procedure is performed assuming arbitrary values of q. If a node with an evaluation value of 0 has fewer than q children with 0 evaluation values, then only the children with 0 evaluation values are expanded.

If the consistency search procedure were being used in the game of chess, it would generate a capture tree. Chess evaluation functions are very sensitive to material differences, and a capture changes the material balance in favor of the player making the move, and against the player whose turn it is to move after the capture. From the perspective of the evaluation function then, a capture would typically lead to a bad position for the player whose turn it is to move. Such a position would have an evaluation value of 0. After a capture is made, the consistency search procedure would only expand nodes whose evaluation value is 0, which are additional capture moves. These are precisely the moves expanded in a capture tree.

Pseudo-code for the consistency search procedure is given in Figure III.2. The procedure begins by expanding all the children of the root node. The search continues until all the nodes on the search frontier (nodes which are parents of leaf nodes) are either consistent or coerced. The evaluation function values of the leaf nodes are now propagated up the tree using the negamax technique. The negamax values of the root child nodes are used to select a move. Adding the alpha-beta tree pruning technique to the consistency search procedure is straightforward and an example is given by Beal in [9].

III.E Extending the Procedure to Multivalued Evaluation Functions

The consistency search procedure described in the previous section applies to a be-valued evaluation function. Most evaluation functions can return a large

1. **Evaluate** the children of the root node $p_i, i = 1, 2, \ldots, b$
2. **Evaluate** the grandchildren of the root node
3. **Move** to the child of the root node whose $C(p_i)$ is least

```
procedure C(position p)
    Let f(p) ≡ evaluation function value of position p
    Let e(p) ≡ negamax value of position p
    if f(p) = e(p)
        return e(p)     p is consistent
    if f(p) = 1
        v = 1
        foreach child p_i, i = 1, 2, ..., b
                Evaluate the children of p_i
                v = min(v, −C(p_i))
        return v
    else
        if f(p_i) = 0 for q or more children of p
            return e(p)     p is coerced
        foreach child p_i, i = 1, 2, ..., b
                if f(p_i) = 0
                    Evaluate the children of p_i
                    v = min(v, −C(p_i))
        return v
```

Figure III.2: Pseudo-code for the consistency search procedure.

number of different values. Applying a simple threshold to these values to generate the values 0 or 1 could lose a significant amount of information.

Beal [11] suggested defining a consistent node for a multivalued evaluation function as one whose evaluation value differed from its one-ply negamax value by no more than some small constant. The question is then what to choose for the small constant. The approach used in the SAL program was to let the evaluation function value of the root node be the threshold at all the nodes where it is the root player's turn to move. Similarly, the negative of this value is the threshold at all the nodes where it is the opponent's move.

The reasoning behind this choice for multivalued evaluation functions is two-fold:

1. The evaluation value of the root position is the current estimate of the root player's chances for a win. The search is intended to find a move which improves these chances. Using the evaluation value of the root node as the threshold causes the consistency search procedure to be most sensitive to values better than this threshold.

2. Analyses of the case of multivalued evaluation functions performed by both Pearl [47] and Baum [7] showed that if pathology occurs for the bi-valued case, then it also occurs for the multivalued case. This indicates an insensitivity in the analyses between the use of bi-valued or multivalued evaluation functions.

For simplicity, only the bi-valued case will be analyzed here.

III.F Analysis of the Consistency Search Procedure

In this section, the effectiveness of the consistency search procedure is quantified. The probability of making an incorrect move using the search is compared with the probability of making an incorrect move if only a one-ply search is performed. This comparison is done for various different uniform branching factors in very deep game trees with independent, randomly assigned terminal outcomes. Recall that for these trees a full-width, fixed-depth, minimax search is pathological [9, 36], so a one-ply search gives a lower probability of making an incorrect move than a deeper search. The analysis shows that for sufficiently accurate evaluation functions, the consistency search significantly decreases the probability of making an incorrect move when compared to a one-ply search.

III.F.1 The Status of a Node

Assume a two-player game with b legal moves from each position where a fixed number of moves must be made before the outcome of the game is known. The board splitting game described by Pearl [47, pp. 270–273] is such a game. The game tree for the board splitting game is a uniform b-ary tree of depth d. Although games like chess do not have a constant branching factor, this assumption is reasonable since, except for late in the endgame, most chess positions have between 30 and 50 legal moves. The analysis is insensitive to the assumption of a fixed number of moves since all that is necessary is that terminal positions be several ply deeper than the depth of the consistency search.

Define the "status" of a node as the true value of the node for the player whose turn it is to move assuming perfect play by both players. Let $S \in \{W, L\}$ indicate whether a node has a WIN status or a LOSE status. Given an assignment of statuses to the leaves of a game tree, the statuses of all the internal nodes can be determined using the negamax procedure. Referring to Figure III.3, the status of the root node 0 is:

$$S_0 = \begin{cases} L & \text{if } S_1 = S_2 = \cdots = S_b = W \\ W & \text{otherwise.} \end{cases}$$

In the following analysis it is assumed that the status assignment of a terminal node is independent of the assignment of other terminal nodes. The case of

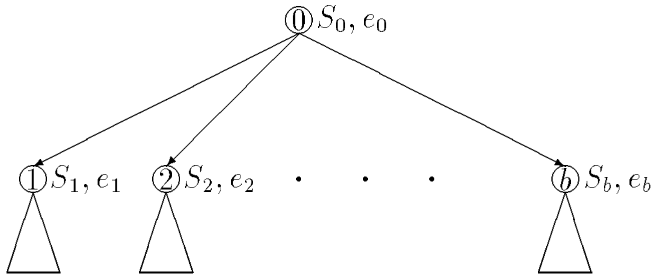


Figure III.3: The notation used for the analysis of a b-ary game tree. S_i is the status of node i assuming perfect play by both players, and e_i is the search value of node i.

independence is assumed in order to demonstrate that the effectiveness of the consistency search procedure is not due to any dependencies between the statuses of terminal nodes. Nau [37] has shown that such dependencies can eliminate pathology in full-width searches. Accordingly, suppose the terminal nodes of the game tree are assigned a WIN with probability p, and a LOSE with probability $1-p$.

III.F.2 A Hazardous, Practically-infinite Game Tree

The analysis assumes the game tree is deep enough that the consistency search procedure will not encounter a terminal node. Such a tree will be called *practically-infinite*. This assumption is intuitively appealing since it represents what would normally be considered a very complex position. Pearl [47] has shown that if a fixed-depth, full-width search can reach terminal nodes, then pathology is eliminated. By restricting the analysis to practically-infinite game trees, it is shown that the effectiveness of the consistency search procedure is not due to encountering terminal nodes.

The analysis also assumes that it is possible to make an incorrect move from the root position. Hence the status of the root node must not be LOSE since every child of a LOSE root node would have to have a WIN status, and no move error could be made. Likewise, at least one child of a WIN root node must have a WIN status. A game tree that has a possible incorrect move from the root will be called *hazardous*.

III.F.3 The Search Value of a Node

Since the game tree is assumed to be deep enough so that the search will not reach any terminal nodes, and the status of non-terminal nodes is not available to a player, the status of a node must be estimated. This analysis assumes that the status of a node is estimated using an evaluation function and the consistency search procedure. The application of the search procedure generates a search tree, which corresponds to only a small portion of the game tree.

Let f be an evaluation function which returns a 1 if the status of a node is estimated to be WIN, and a 0 if the status is estimated to be LOSE. The "search value" e of a leaf node in the search tree will be defined as the evaluation function value of the node. The search value of a non-leaf node is given by the usual negamax procedure:

$$e_0 = \begin{cases} 0 & \text{if } e_1 = e_2 = \cdots = e_b = 1 \\ 1 & \text{otherwise.} \end{cases}$$

This analysis only considers bi-valued evaluation functions. The case of continuous evaluation functions was considered by Pearl [47]. For a full-width, fixed-depth search, the results for the continuous case were similar to the bi-valued results. For simplicity, only the bi-valued case will be considered for the consistency search procedure to be described.

III.F.4 Error Probabilities

A notation similar to Pearl's [47] is adopted. The accuracy of the evaluation function is defined by the two error measures:

$$\begin{aligned} \alpha &\equiv P(f = 1 | S = L) \\ \beta &\equiv P(f = 0 | S = W). \end{aligned}$$

Given that a position is actually lost for the player whose turn it is to move, α is the probability that the evaluation function will incorrectly evaluate the position as won. Similarly, β is the probability that the evaluation function will incorrectly evaluate a won position as lost. It is assumed that α and β are independent of the depth of the position in the game tree.

The accuracy of the consistency search procedure can be characterized in a similar fashion. Specifically, the search value assigned to a node is an estimate of the status of the node, and may be incorrect. The probability of an erroneous assignment at a node of depth d is defined as

$$\begin{aligned}\alpha_d &\equiv P(e = 1 | S = L, D = d) \\ \beta_d &\equiv P(e = 0 | S = W, D = d)\end{aligned}$$

where the root node is at depth 0.

III.F.5 The Probability of Making an Incorrect Move

If the search value of one of the children of the root node is a 0, then it might be selected as the move to make. However, if the status of this node is a WIN, then it would be incorrect to make the move. The following analysis compares the probability of making an incorrect move after a consistency search is performed:

$$P(\text{error at root}|\text{Consistency Search}),$$

with the probability of making an incorrect move after a full-width, one-ply search is performed:

$$P(\text{error at root}|\text{One-Ply Search}).$$

Note that since a hazardous, practically-infinite game tree is pathological for a full-width, fixed-depth minimax search, a one-ply search results in a lower probability of making an incorrect move than a deeper search.

One-Ply Search

Let $N_{S=L} = i$ be the condition that the root has i children with a LOSE status and $b - i$ children with a WIN status. If a one-ply search is performed, the probability of making an incorrect move in a hazardous, practically-infinite game tree is given by:

$$\begin{aligned}
&P(\text{error at root}|\text{No Search}) \\
&= \sum_{i=1}^{b-1} P(\text{error at root}|N_{S=L}=i)\frac{P(N_{S=L}=i)}{\sum_{j=1}^{b-1} P(N_{S=L}=j)} \\
&= \sum_{i=1}^{b-1} \binom{b}{i} \frac{(1-p_1)^i p_1^{b-i} \pi(\alpha,\beta,i)}{1-p_1^b-(1-p_1)^b}, \qquad \text{(III.1)}
\end{aligned}$$

where p_1 is the probability of a node at depth 1 having a WIN status, and $\pi(\alpha,\beta,i)$ is the probability of making an error given a set of i LOSEs and $b-i$ WINs for the child node statuses. A formula for $\pi(\alpha,\beta,i)$ is derived in Section III.H.

If $p_1 \to 0$, then both the numerator and denominator of the terms in Equation III.1 vanish. To determine the probability of making an incorrect move in this case, let $p_1 = \epsilon$ as $\epsilon \to 0$. Using a series expansion in ϵ gives:

$$\frac{1}{1-(1-p_1)^b - p_1^b} = \frac{1}{b\epsilon}[1 + \frac{1}{2}(b-1)\epsilon + \mathcal{O}(\epsilon^2)].$$

The only $\mathcal{O}(\epsilon)$ term in the numerator of Equation III.1 occurs when $i = b-1$. Hence for a hazardous, practically-infinite game tree where $p_1 \to 0$, the probability of making an incorrect move approaches:

$$P(\text{error at root}|\text{No Search}) \to \pi(\alpha,\beta,b-1).$$

Consistency Search

If a consistency search is performed, the probability of making an incorrect move in a hazardous, practically-infinite game tree is identical to Equation III.1, except that the error probabilities of the children of the root node are those resulting from consistency search.

$$\begin{aligned}
&P(\text{error at root}|\text{Consistency Search}) \\
&= \sum_{i=1}^{b-1} \binom{b}{i} \frac{(1-p_1)^i p_1^{b-i} \pi(\alpha_1,\beta_1,i)}{1-p_1^b-(1-p_1)^b} \qquad \text{(III.2)}
\end{aligned}$$

If $p_1 \to 0$, then:

$$P(\text{error at root}|\text{Consistency Search}) \to \pi(\alpha_1,\beta_1,b-1).$$

III.F.6 Determining α_d and β_d

The equations for α_d and β_d in Equation III.2 can be determined by summing over all possible heights of the search tree:

$$\begin{aligned}
\alpha_d &= \sum_{h=1}^{n-d} \alpha_{d,h} A_{d,h} \\
\beta_d &= \sum_{h=1}^{n-d} \beta_{d,h} B_{d,h},
\end{aligned}$$

where the probability of a subtree rooted at a node of depth d having height h is:

$$\begin{aligned}
A_{d,h} &\equiv P(H = h | S = L, D = d) \\
B_{d,h} &\equiv P(H = h | S = W, D = d),
\end{aligned}$$

and the probability of an error in the search value at a node of height h is:

$$\begin{aligned}
\alpha_{d,h} &\equiv P(e = 1 | S = L, D = d, H = h) \\
\beta_{d,h} &\equiv P(e = 0 | S = W, D = d, H = h).
\end{aligned}$$

A consistent or coerced node is at a height of 1, and there are n moves in the game. The summation begins at $h = 1$ since the case of $h = 0$ corresponds to a node with no children. However, the consistency search procedure only stops at consistent or coerced nodes, which are determined by evaluating the children.

Consider the equation for $\alpha_{d,h}$. The α error at a node can be obtained by combining the error if the node has an evaluation value of 1 with the error if the node has an evaluation value of 0, given by:

$$\begin{aligned}
\alpha_{d,h} &= \alpha'_{d,h} P(f = 1 | S = L, D = d, H = h) \\
&\quad + \alpha''_{d,h} P(f = 0 | S = L, D = d, H = h),
\end{aligned}$$

where

$$\begin{aligned}
\alpha'_{d,h} &\equiv P(e = 1 | S = L, D = d, H = h, f = 1) \\
\alpha''_{d,h} &\equiv P(e = 1 | S = L, D = d, H = h, f = 0).
\end{aligned}$$

A single prime mark indicates the evaluation value at the node is 1, and a double prime mark indicates the evaluation value at the node is 0. Using Bayes' rule:

$$\begin{aligned}
&P(f=1|S=L,D=d,H=h) \\
&= P(H=h|S=L,D=d,f=1)\frac{P(f=1|S=L,D=d)}{P(H=h|S=L,D=d)} \\
&= A'_{d,h}\frac{\alpha}{A_{d,h}}
\end{aligned}$$

and

$$\begin{aligned}
&P(f=0|S=L,D=d,H=h) \\
&= P(H=h|S=L,D=d,f=0)\frac{P(f=0|S=L,D=d)}{P(H=h|S=L,D=d)} \\
&= A''_{d,h}\frac{(1-\alpha)}{A_{d,h}},
\end{aligned}$$

where

$$\begin{aligned}
A'_{d,h} &\equiv P(H=h|S=L,D=d,f=1) \\
A''_{d,h} &\equiv P(H=h|S=L,D=d,f=0).
\end{aligned}$$

Here, as throughout, it is assumed that the probability of incorrectly evaluating a node is independent of the depth of the node.

Similarly, β_d can be derived to give the final result:

$$\begin{aligned}
\alpha_d &= \sum_{h=1}^{n-d}[\alpha\alpha'_{d,h}A'_{d,h} + (1-\alpha)\alpha''_{d,h}A''_{d,h}] \\
\beta_d &= \sum_{h=1}^{n-d}[(1-\beta)\beta'_{d,h}B'_{d,h} + \beta\beta''_{d,h}B''_{d,h}].
\end{aligned}$$

Equations for each of the quantities just defined are derived in Section III.H.

III.F.7 Solutions to the Probabilistic Equations

For a given value of α and β, the system given by Equations III.3 through III.18 was numerically solved to give α_1 and β_1. From these, Equation III.2 then gives the probability of making an incorrect move if a consistency search is performed. This probability was compared to the probability given by Equation III.1, which applies to the case where a one-ply search is performed. Since the assumed game tree is pathological for a full-width, fixed-depth, minimax search, the search depth that results in the lowest probability of making an incorrect move is one-ply.

Figures III.4, III.5, III.6, III.7, and III.8 are contour plots as a function of the accuracy of the evaluation function. This accuracy is given in terms of α and β. Each figure applies to a hazardous, practically-infinite game tree with a branching factor of 10. For the figures applicable to a consistency search, a q-level of two was used. This means that at nodes with an evaluation value of 0 the search was terminated, and the node coerced, if two or more children also had evaluation values of 0.

For hazardous, practically-infinite game trees, two types of trees are much more probable than any others [38]: either all but one of the children of the root node have LOSE statuses, or all but one have WIN statuses. If all but one of the children have LOSE statuses, then there is only one possible incorrect move. This is an "easy win" for the root node player. However, if all but one of the children have WIN statuses, then there are $b-1$ possible incorrect moves. This is a "difficult win" for the root node player. Results for both types of game trees are shown in the figures.

Figure III.4 is a plot of the probability of making an incorrect move if only a one-ply search is performed. The evaluation function is applied to the 10 legal positions that are possible and the move is then selected. Note that the more accurate the evaluation function, the lower the probability of making an incorrect move.

Figure III.5 is a plot of the probability of making an incorrect move after a consistency search is performed. This plot applies to the case of an easy win. The contours are computed assuming that the consistency search terminates before it exceeds a depth of 10-ply. Figure III.6 is a plot of the probability of the consistency search exceeding a depth of 10 ply for the case of an easy win.

Figure III.7 is a contour plot of the ratio of the probability of making an incorrect move when a one-ply search is performed (shown in Figure III.4), over the probability of making an incorrect move when a consistency search is performed (shown in Figure III.5). The solid contour lines indicate regions where this ratio is greater than 1.0, and hence the consistency search is beneficial. The dashed contour lines indicate regions where the ratio is less than 1.0, and hence the consistency search is not beneficial. For clarity, contour lines for ratios greater than 3.0 are not drawn. For easy reference, Figure III.6 is duplicated as the bottom plot of Figure III.7. Similar plots are shown for the case of a difficult win in Figure III.8.

Similar plots have been generated for a variety of branching factors and q values from 2 to 50. They all have had the same general form, shown in figures III.7 and III.8. Of particular interest is the region in the lower left corner of the plots, representing the case of a fairly accurate evaluation function. In this case the consistency search is always beneficial, provided q is greater than one. The magnitude of improvement increases as q is increased, but the probability of the search not terminating also increases as q is increased. However, for a sufficiently accurate evaluation function the search always terminates since the consistency search only expands nodes that may have been erroneously evaluated, and an accurate evaluation function would rarely make an error.

III.F.8 Monte Carlo Studies

To verify the probabilistic equations, a large number of game trees were randomly constructed and then searched using the consistency search procedure just described. A comparison between three such tests with the mathematical results are presented in this section.

The Monte Carlo program was designed to generate a large number of full-width, fixed-depth game trees with randomly selected terminal values (either WIN or LOSE). These terminal values were propagated up the tree to determine the true status of each internal node.

For the case in which a one-ply search is performed, each child of the root node was "evaluated" by choosing a random number and deciding, based on the given

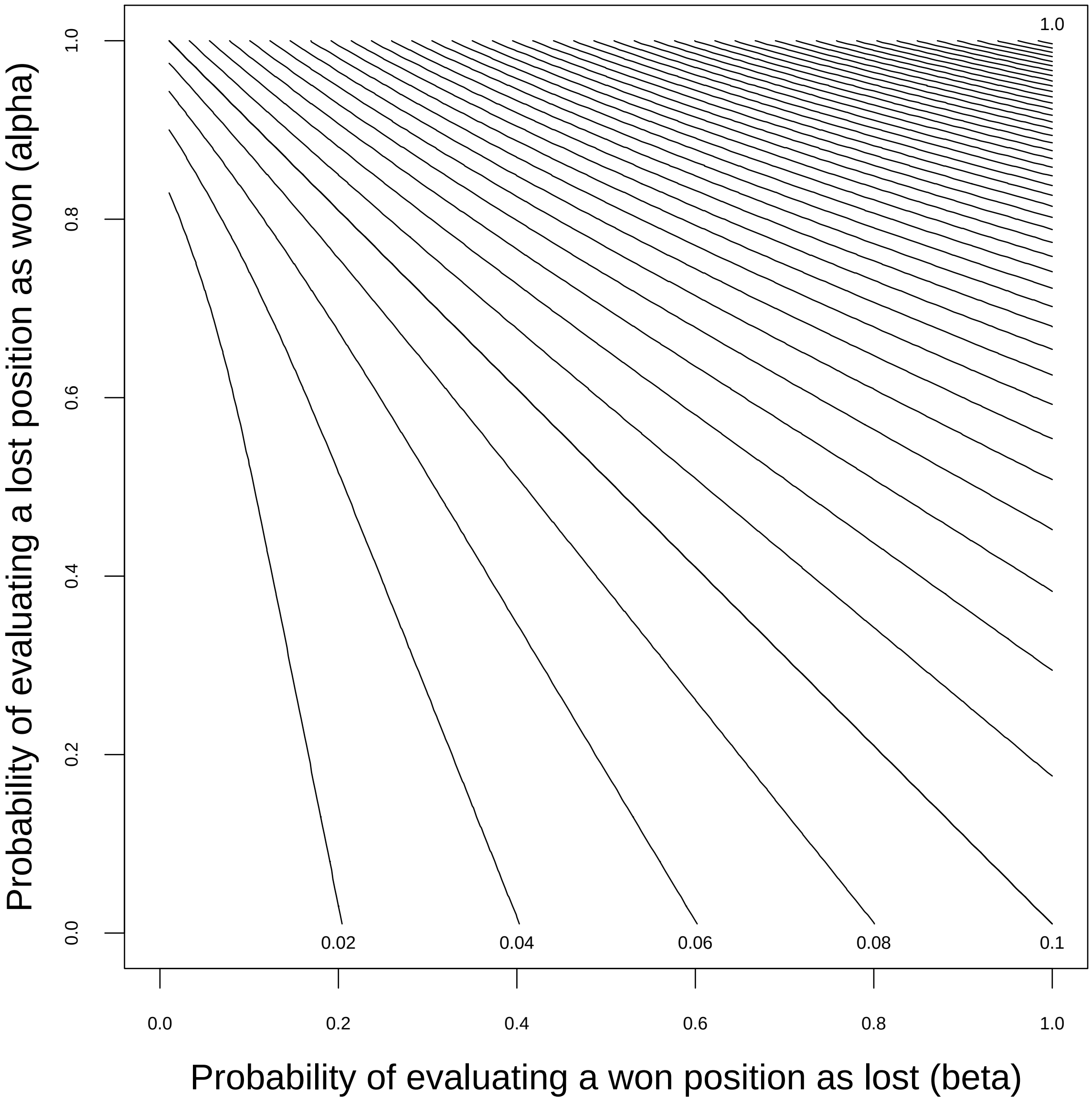


Figure III.4: The probability of making an incorrect move after a one-ply search for an easy win. The values of some of the contour lines are given in the margins of the plot within the outline of the plot region.

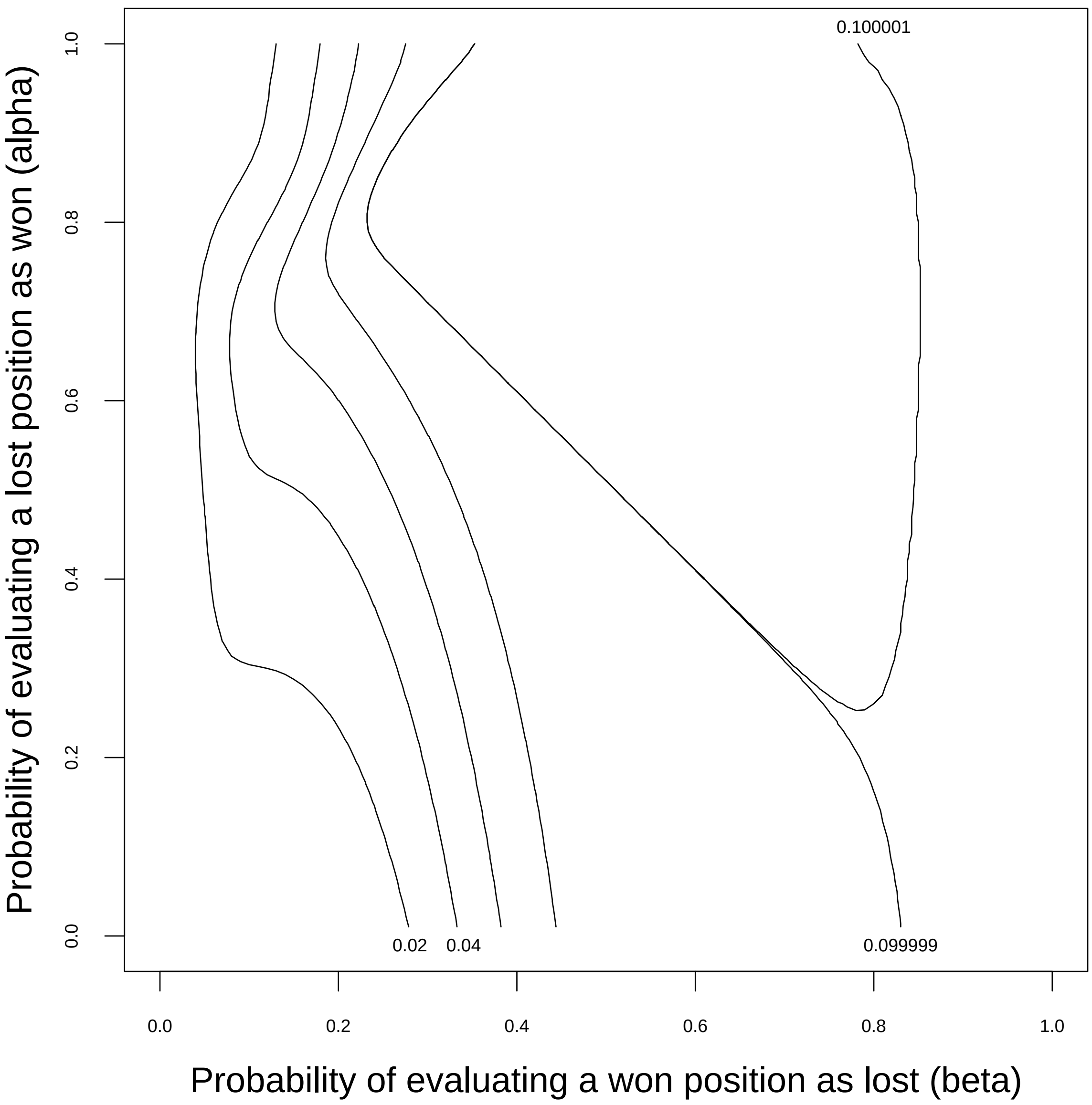


Figure III.5: The probability of making an incorrect move after a consistency search for an easy win.

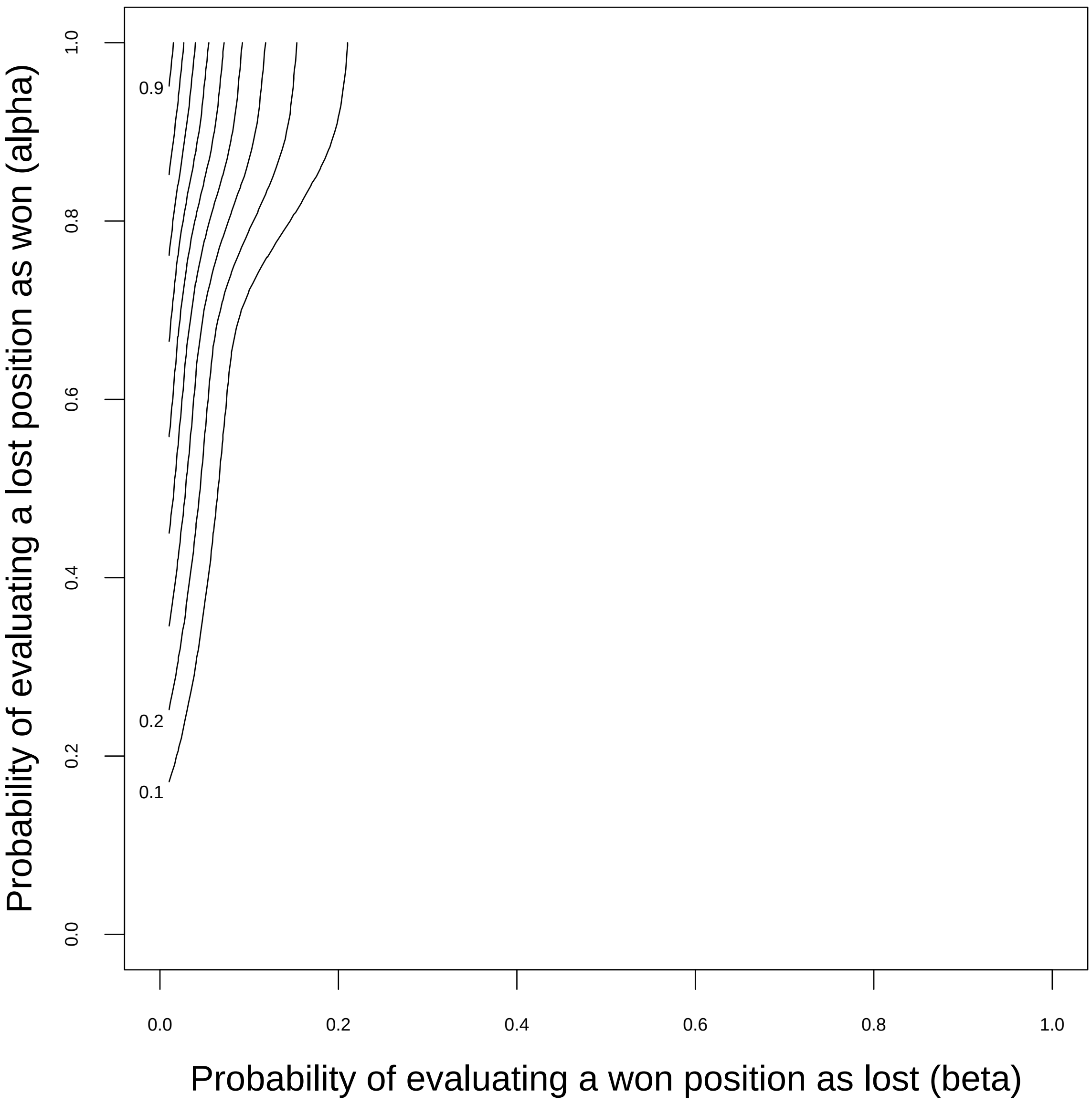


Figure III.6: The probability of a consistency search exceeding 10 ply for an easy win.

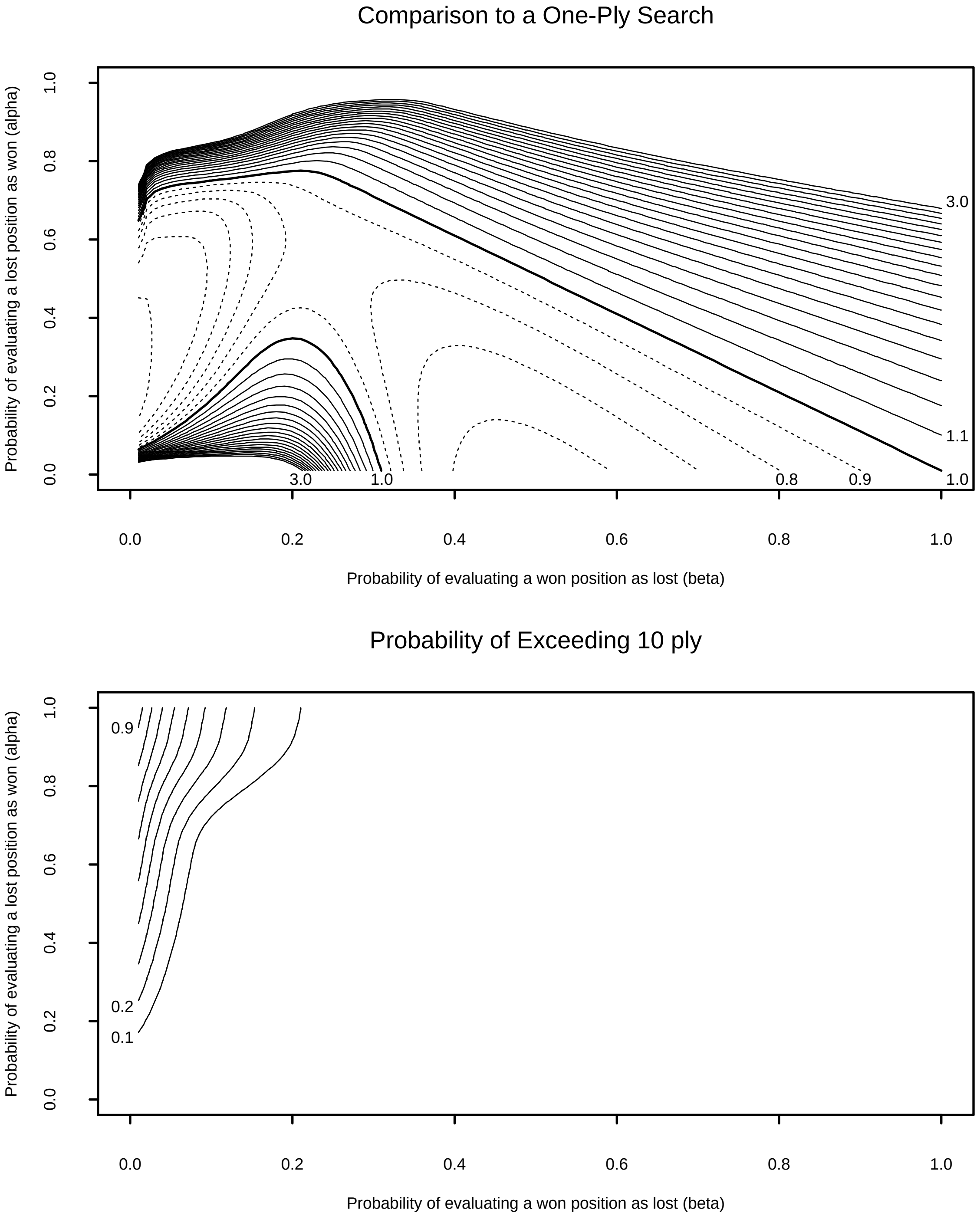
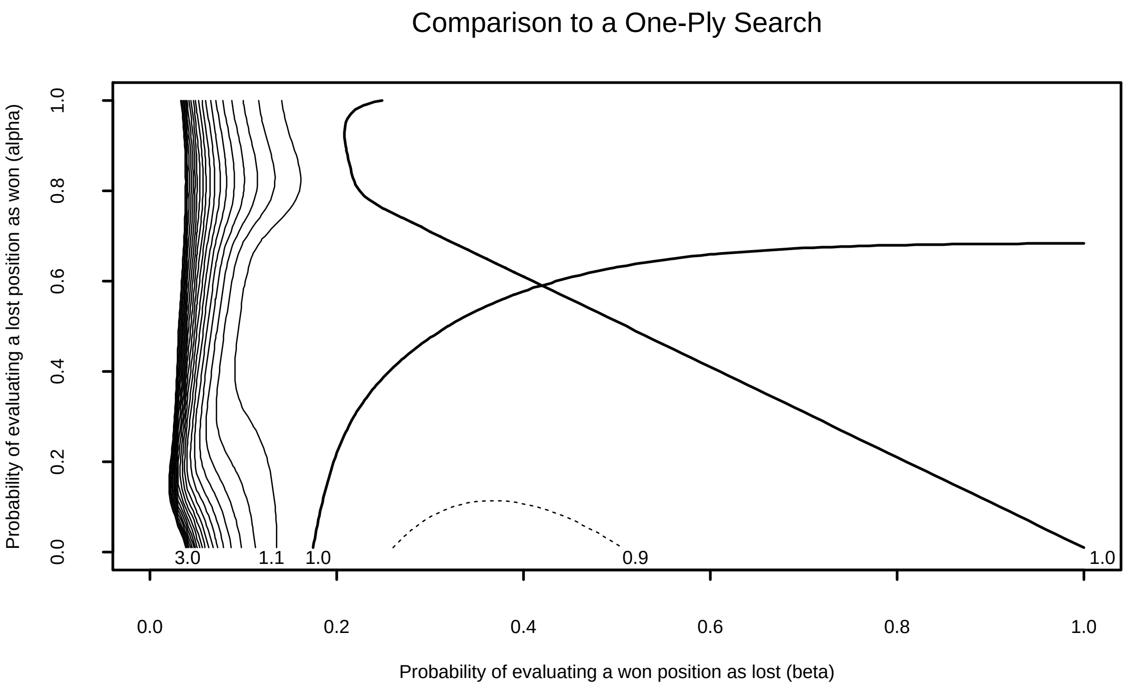


Figure III.7: The characteristics of the consistency search for an easy win. Dashed lines indicate areas where a one-ply search is better than a consistency search, and solid lines indicate areas where the consistency search is better than a one-ply search.



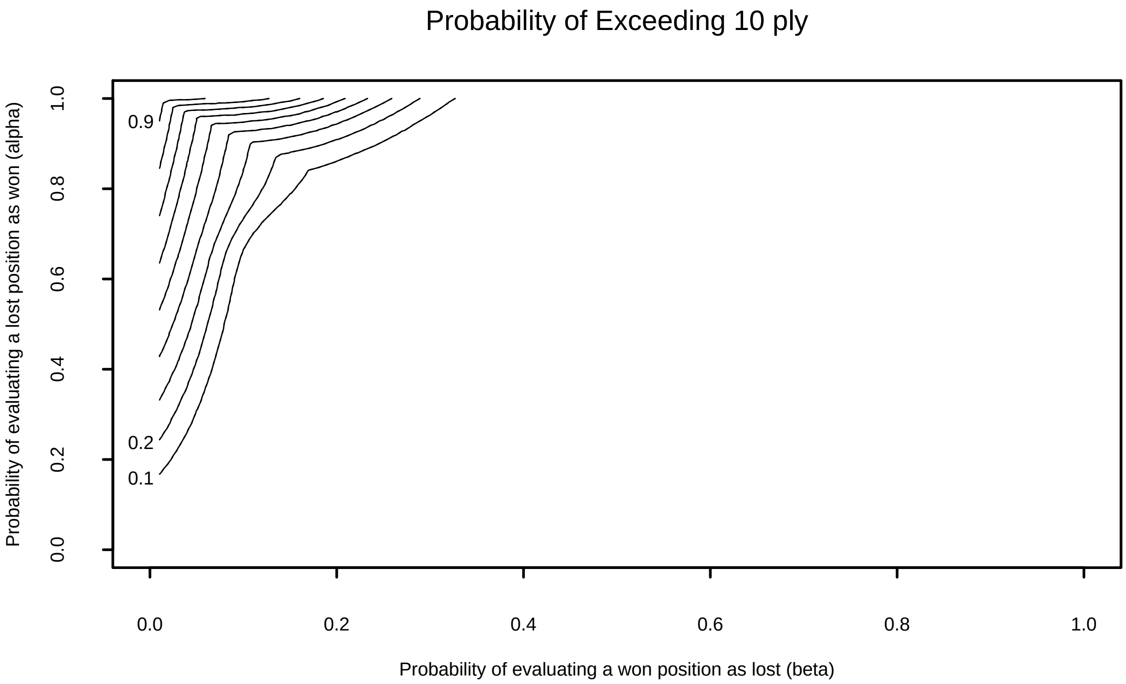


Figure III.8: The characteristics of the consistency search for a difficult win. Dashed lines indicate areas where a one-ply search is better than a consistency search, and solid lines indicate areas where the consistency search is better than a one-ply search.

α and β values, whether the evaluation would be incorrect for that node. Since only a one-ply search is performed in this case, one of the nodes evaluated to be a lost position would be randomly chosen as the move to make. Since the actual status of each node is known, it is known whether the selected move is correct or incorrect.

The case in which a search is performed on the game tree is considered in a similar fashion. A consistency search is performed and a move is selected. Again, a count is kept of the number of game trees in which the consistency search selected an incorrect move. The validity of the mathematical results could then be verified by comparing the number of incorrect moves selected in both cases with the predicted number generated from the mathematical equations. Counts were also kept of all the intermediate values corresponding to Equations III.3 through III.18. The results showed that these counts were in agreement with the computed values.

Table III.1 gives the parameter values and some of the overall results for three Monte Carlo tests of the consistency search procedure. Table III.2 compares the number of incorrect moves selected with the predicted number. Over a dozen such tests have been performed, providing similar verification of the predicted values. Of all the Monte Carlo test runs, the percentage difference between the predicted and actual number of incorrect moves was greatest for test 1. Some error is to be expected from random fluctuations and from the fairly shallow Monte Carlo game trees necessary due to computational limitations.

III.G Using Consistency Search

The analysis of consistency search indicates that the search is beneficial if the evaluation function is suitably accurate. If the evaluation function is not accurate enough however, the search could result in degraded performance. For a game-learning program this could correspond to worse play than a one-ply search until the accuracy of the evaluation function improves. This is observed in the SAL program.

Consider a learning algorithm that improves both the alpha and beta errors

	Test 1	Test 2	Test 3
Branching Factor	4	2	5
Quiescent Level	3	2	1
Alpha	0.01	0.1	0.03
Beta	0.02	0.2	0.06
Game Tree Depth	5	5	5
Leaf Win Probability	0.7	0.5	0.7
Game Trees Generated	10^6	10^4	3.0×10^4
Hazardous Trees where the search terminates	842,328	4,388	26,001

Table III.1: Characteristics of three Monte Carlo tests of the consistency search procedure.

	Test 1		Test 2		Test 3	
	Pred.	Actual	Pred.	Actual	Pred.	Actual
Move Errors without a Search	17,654	17,329	658	655	900	875
Move Errors with a Search	692	809	323	340	669	653

Table III.2: A comparison of three Monte Carlo simulation results with the mathematically predicted results of the evaluation function equally. Figure III.9 illustrates the performance path such a learning algorithm would take in terms of the evaluation function error probabilities. Given this performance path, Figures III.10 and III.11 are graphs of both the consistency search and a one-ply search comparing the probability of making a *correct* move as a function of evaluation function error. These figures were generated using the results of the analysis shown in Figures III.7 and III.8. Figures III.10 and III.11 correspond to the middle plots of the qualitative Figures II.3 and II.4.

The magnitude of the benefit from consistency search increases with increasing q-level. Hence an incremental approach was used for the SAL program. The search was first performed with $q = 1$. If the search terminated and there was still time, a search using $q = 2$ was performed. The value of q was increased in this fashion until the available time was exhausted. The results of the completed search with the highest q value was used to choose the move.

III.H Derivations

In this section, formulas for some quantities of interest in the analysis of consistency search are derived. The symbols e, f, S, H, and D refer to random variables representing the search value, evaluation function value, status, height, and depth of a search tree node, respectively. Subscripts on e, f, S, H, or D indicate node number, as indicated in Figure III.3. The symbols α, β, A, and B, and others represent probabilities of interest. The first subscript on these symbols indexes depth, the value of the random variable D; the second subscript, if it exists, indexes height, the value of the random variable H. Further, a prime "$'$" on these symbols indicates that the random variable f is understood to have value 1, and a double prime "$''$" indicates that the random variable f has value 0. For some functions, only the derivation of the $f = 1$ case will be shown.

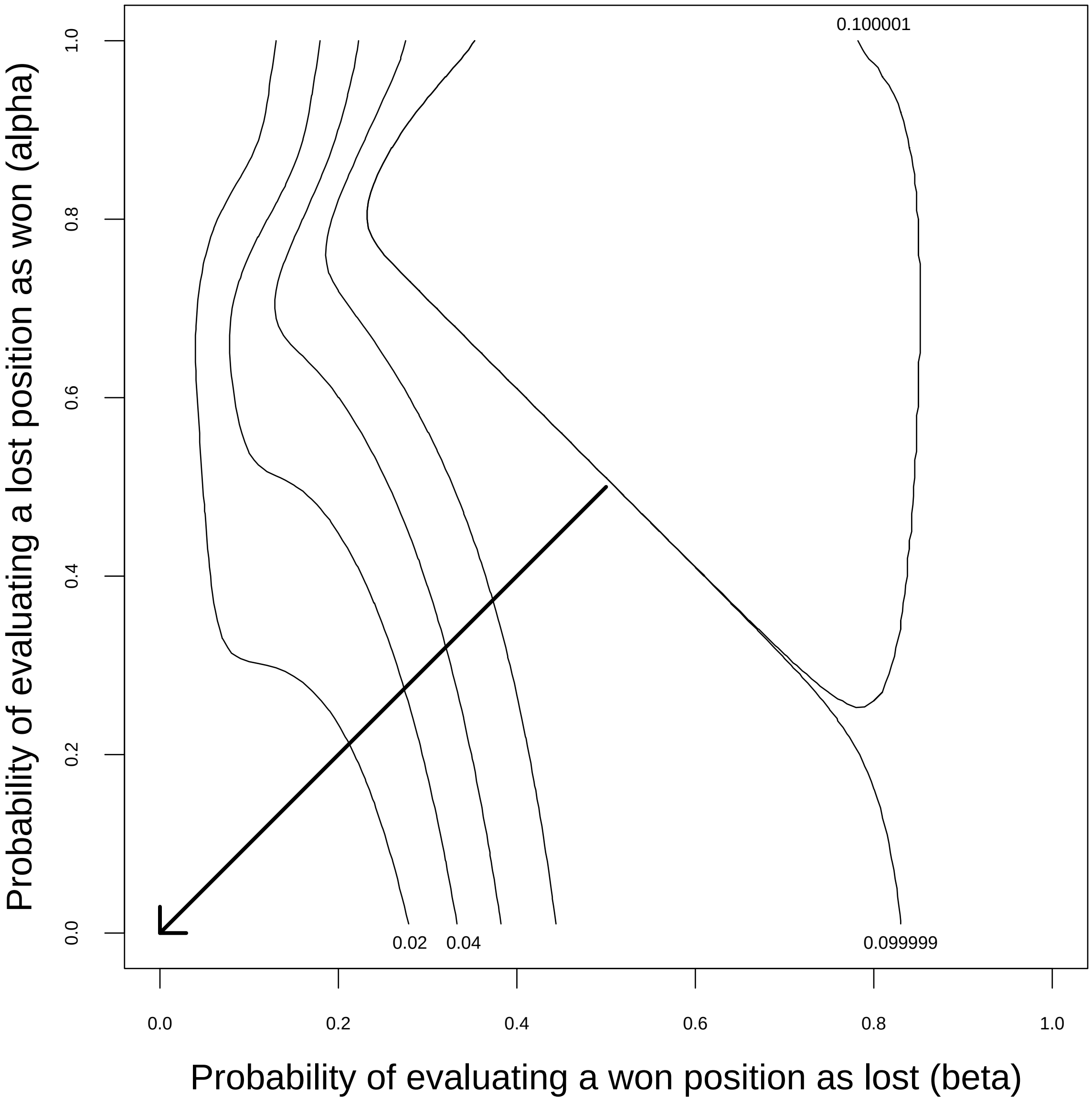


Figure III.9: The performance path a learning algorithm would take if it improved both the alpha and beta errors equally and was started at a $\alpha = \beta = 0.5$. The arrow traces the path.

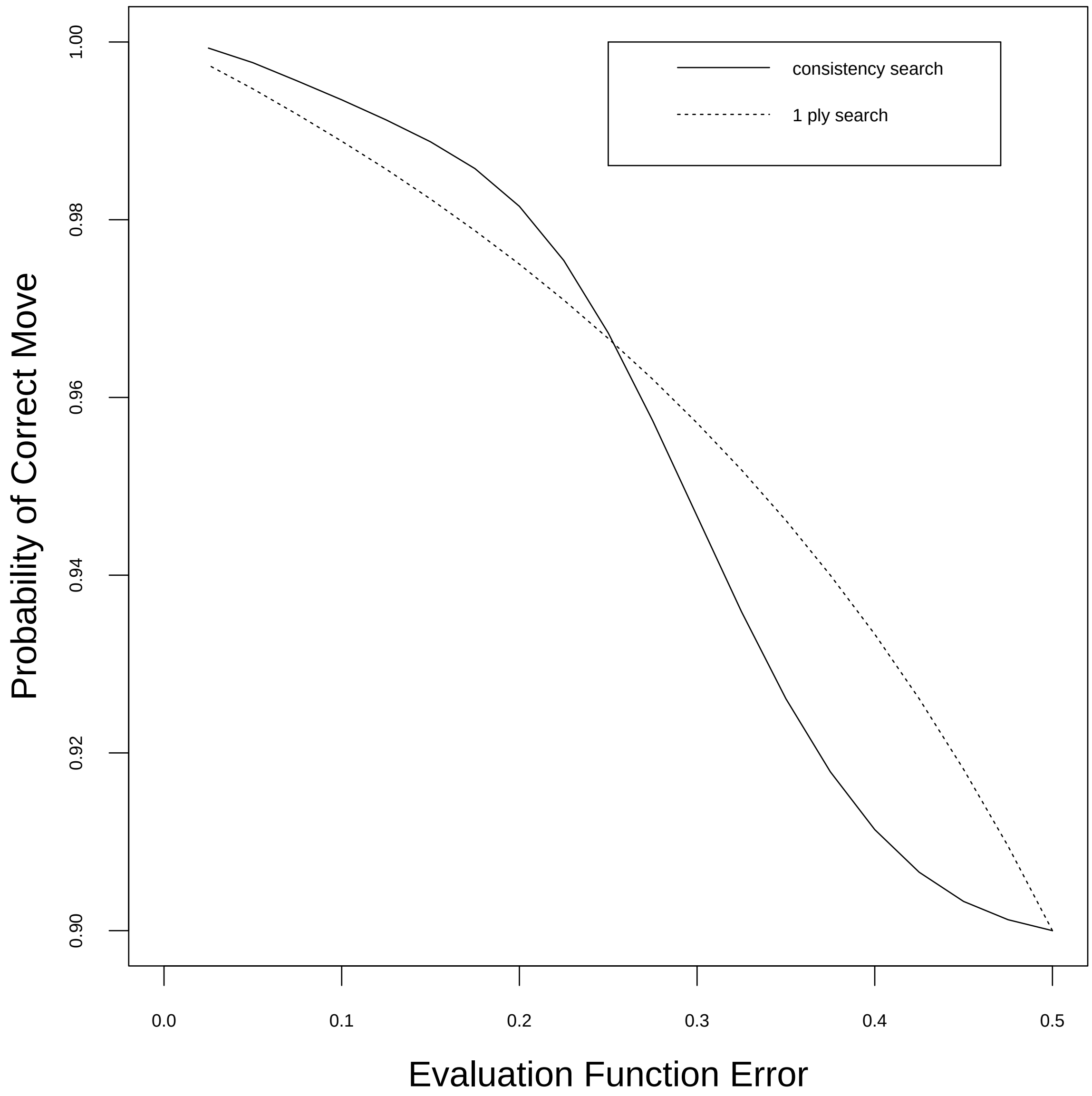


Figure III.10: A comparison of the probability of making a correct move for an easy win.

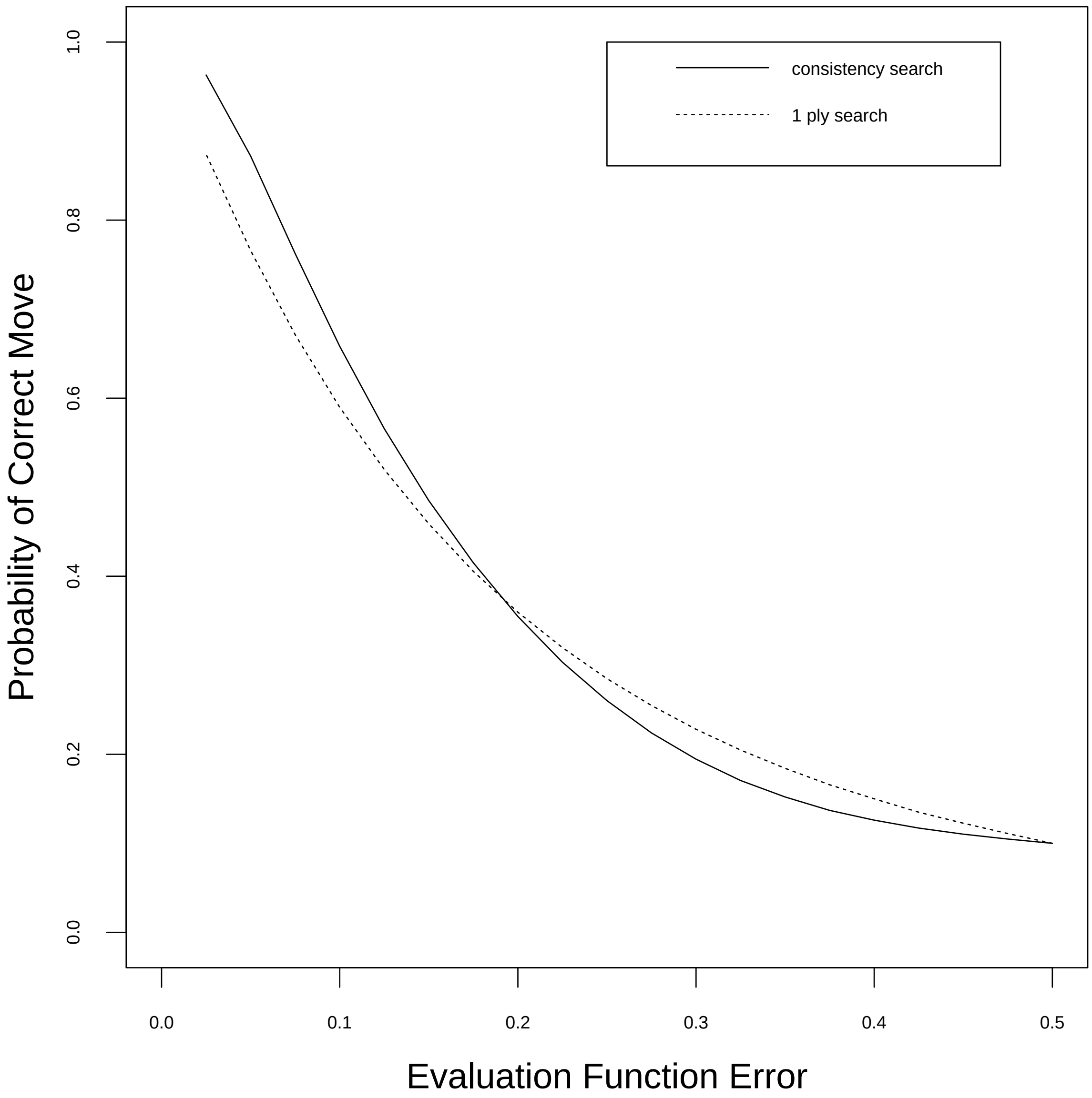


Figure III.11: A comparison of the probability of making a correct move for a difficult win.

III.H.1 Determining $A'_{d,h}$ and $A''_{d,h}$

The function $A_{d,h}$ is the probability that a search tree of height h will be generated by a consistency search starting from a node of depth d which has a LOSE status. $A'_{d,h}$ applies to the case where the evaluation function value is 1, and $A''_{d,h}$ applies to the case where the evaluation value is 0:

$$\begin{aligned} A'_{d,h} &\equiv P(H=h|S=L,D=d,f=1) \\ A''_{d,h} &\equiv P(H=h|S=L,D=d,f=0). \end{aligned}$$

The cases where $h=1$ and $h>1$ will be determined separately.

Case 1: A consistent or coerced node ($h=1$).

Every child of a LOSE node must have a WIN status, independent of the depth of the node. If the node has an evaluation function value of 1, then it is consistent if and only if at least one of its children has a value of 0.

$$A'_{d,1} = 1-(1-\beta)^b \tag{III.3}$$

If the node has an evaluation function value of 0, then it is consistent if all of its children have a value of 1, and it is coerced if q or more of its children have a value of 0.

$$A''_{d,1} = (1-\beta)^b + \sum_{i=q}^{b} \binom{b}{i} \beta^i (1-\beta)^{b-i} \tag{III.4}$$

Case 2: An inconsistent and non-coerced node ($h>1$).

Referring to Figure III.3, let node 0 be a node of height h, depth d, and evaluation function value of 1 with a LOSE status. By definition, the height of a non-terminal, non-root node will be greater than 1 if and only if it is neither consistent nor coerced. For $h>1$,

$$\begin{aligned} A'_{d,h} &= P(H_0=h, H_0>1|S_0=L, D_0=d, f_0=1) \\ &= P(H_0>1|S_0=L, D_0=d, f_0=1) \end{aligned}$$

$$\times P(H_0 = h | H_0 > 1, S_0 = L, D_0 = d, f_0 = 1)$$
$$= \quad (1 - A'_{d,1}) P(H_0 = h | H_0 > 1, S_0 = L, D_0 = d, f_0 = 1).$$

Define $N_{f=0} = i$ as the event that i children of node 0 have an evaluation value of 0, and $N_{S=L} = i$ as the event that i children of node 0 have a LOSE status. Then since, by definition, the height of a non-terminal, non-root node will be greater than 1 if and only if it is neither consistent nor coerced, the conditioning event:

$$H_0 > 1, S_0 = L, D_0 = d, f_0 = 1$$

is identical to the event:

$$N_{f=0} = 0, N_{S=L} = 0, D_0 = d, f_0 = 1.$$

The event $H_0 = h$ in this case is the event that the maximum height of consistency searches begun at b independent nodes with a WIN status evaluated as 1 at depth $d+1$ is $h-1$. The probability of one such search having height i is just $B'_{d+1,i}$, as defined in the next section. The probability of one such search having height less than h is then:

$$\sum_{i=1}^{h-1} B'_{d+1,i}$$

and so the probability of b independent such searches each having heights less than h is:

$$\left(\sum_{i=1}^{h-1} B'_{d+1,i} \right)^b .$$

Similarly, the probability of b independent such searches each having heights less than $h-1$ is:

$$\left(\sum_{i=1}^{h-2} B'_{d+1,i} \right)^b .$$

The event of b independent such searches each having height less than h is the union of the mutually exclusive events that b independent such searches each have height less than $h-1$, and that b independent such searches have maximum height $h-1$:

$$P(H_0 = h | H_0 > 1, S_0 = L, D_0 = d, f_0 = 1)$$

$$
\begin{aligned}
&= \left(\sum_{i=1}^{h-1} B'_{d+1,i}\right)^b - \left(\sum_{i=1}^{h-2} B'_{d+1,i}\right)^b \\
&\equiv \Psi'_{d+1,h}(b).
\end{aligned}
$$

Finally,

$$
A'_{d,h} = (1 - A'_{d,1})\Psi'_{d+1,h}(b). \tag{III.5}
$$

In the case $f_0 = 0$, for $h > 1$,

$$
\begin{aligned}
A''_{d,h} &= P(H_0 = h, H_0 > 1 | S_0 = L, D_0 = d, f_0 = 0) \\
&= (1 - A''_{d,1})P(H_0 = h | H_0 > 1, S_0 = L, D_0 = d, f_0 = 0).
\end{aligned}
$$

Here, the conditioning event,

$$
H_0 > 1, S_0 = L, D_0 = d, f_0 = 0
$$

is identical to the event that all b children of node 0 have depth $d+1$, WIN status, and at least one and fewer than q of them are evaluated as 0 (in this case, only children evaluated as 0 are expanded). The conditioning event is just the union of the disjoint events:

$$
N_{f=0} = r, S_0 = L, D_0 = d, f_0 = 0
$$

for $r = 1, \ldots, q-1$. Thus

$$
\begin{aligned}
&P(H_0 = h | H_0 > 1, S_0 = L, D_0 = d, f_0 = 0) \\
&\quad = \sum_{r=1}^{q-1} P(H_0 = h | N_{f=0} = r, S_0 = L, D_0 = d, f_0 = 0) \\
&\qquad \times \frac{P(N_{f=0} = r | S_0 = L, D_0 = d, f_0 = 0)}{\sum_{j=1}^{q-1} P(N_{f=0} = r | S_0 = L, D_0 = d, f_0 = 0)}.
\end{aligned}
$$

Similar to before,

$$
\begin{aligned}
&P(H_0 = h | N_{f=0} = r, S_0 = L, D_0 = d, f_0 = 0) \\
&\quad = \left(\sum_{i=1}^{h-1} B''_{d+1,i}\right)^r - \left(\sum_{i=1}^{h-2} B''_{d+1,i}\right)^r \\
&\quad \equiv \Psi''_{d+1,h}(r)
\end{aligned}
$$

where $B''_{d,h}$ is as defined in the next section. Also,

$$P(N_{f=0} = r | S_0 = L, D_0 = d, f_0 = 0) = \binom{b}{r} \beta^r (1-\beta)^{b-r}.$$

Finally, noting that:

$$(1 - A''_{d,1}) = \sum_{i=1}^{q-1} \binom{b}{i} \beta^i (1-\beta)^{b-i}$$

and renaming subscripts, the value of $A''_{h,d}$ for $h > 1$ is:

$$A''_{d,h} = \sum_{i=1}^{q-1} \binom{b}{i} \beta^i (1-\beta)^{b-i} \Psi''_{d+1,h}(i). \tag{III.6}$$

III.H.2 Determining $B'_{d,h}$ and $B''_{d,h}$

The function $B_{d,h}$ is the probability that a search tree of height h will be generated by a consistency search starting from a node of depth d which has a WON status. $B'_{d,h}$ applies to the case where the evaluation function value is 1, and $B''_{d,h}$ applies to the case where the evaluation function value is 0:

$$\begin{aligned} B'_{d,h} &\equiv P(H = h | S = W, D = d, f = 1) \\ B''_{d,h} &\equiv P(H = h | S = W, D = d, f = 0). \end{aligned}$$

The cases where $h = 1$ and $h > 1$ will be determined separately.

Case 1: A consistent or coerced node ($h = 1$).

If the node has an evaluation function value of 1, then it is consistent if at least one of its children has a value of 0.

$$\begin{aligned} B'_{d,1} &= \frac{\sum_{i=1}^{b} P(H = 1 | N_{S=L} = i, D = d) P(N_{S=L} = i | D = d)}{\sum_{i=1}^{b} P(N_{S=L} = i | D = d)} \\ &= \sum_{i=1}^{b} S(d+1, i)(1 - \alpha^i (1-\beta)^{b-i}) \end{aligned} \tag{III.7}$$

where $S(d, i)$ is the probability of having i children with a LOSE status at depth d given that at least one child has a WIN status.

$$\begin{aligned} S(d, i) &= \frac{P(N_{S=L} = i | D = d)}{\sum_{j=1}^{b} P(N_{S=L} = i | D = d)} \\ &= \binom{b}{i} \frac{p^{b-i}(d)(1-p_d)^i}{1 - p^b(d)} \end{aligned}$$

If the node has an evaluation function value of 0, then it is consistent if all of its children have a value of 1, and it is coerced if q or more of its children have a value of 0. Again, incorporating the probability of having i children with LOSE statuses gives:

$$B''_{d,1} = \sum_{i=1}^{b} S(d+1,i)\left(\alpha^i(1-\beta)^{b-i} + \sum_{j=q}^{b}\sum_{k=\max(0,j-b+i)}^{\min(i,j)} G(i,j,k)\right) \tag{III.8}$$

where $G(i,j,k)$ is the probability of having k children with LOSE statuses evaluated as 0 given that there are j children with evaluation values of 0 and i children with LOSE statuses.

$$G(i,j,k) \equiv \binom{j}{k}\binom{b-i}{j-k}(1-\alpha)^k\alpha^{i-k}\beta^{j-k}(1-\beta)^{b-i-j+k}$$

As $p(d+1) \to 1$, both the numerator and denominator of equations III.7 and III.8 approach 0. To determine the value of $B'_{d,1}$ and $B''_{d,1}$ in this case, let $p(d+1) = 1-\epsilon$ as $\epsilon \to 0$. Keeping only terms of order ϵ gives:

$$\begin{aligned} B'_{d,1} &\to 1-\alpha(1-\beta)^{b-1} \\ B''_{d,1} &\to \alpha(1-\beta)^{b-1} + \sum_{j=q}^{b}\sum_{k=\max(0,j-b+1)}^{1} G(1,j,k). \end{aligned}$$

Case 2: An inconsistent and non-coerced node ($h > 1$).

Referring to Figure III.3, let node 0 be a node of height h, depth d, and evaluation function value 1 with a WIN status. The probability of this node being at a height greater than 1 is $1 - B'_{d,1}$. All the children of this node must have an evaluation value of 1, and all are expanded:

$$\begin{aligned} B'_{d,h} &= (1-B'_{d,1})\sum_{i=1}^{b} P(H_0 = h | N_{S=L} = i, D_0 = d) \\ &\quad \times \frac{P(N_{S=L} = i | D_0 = d)}{\sum_{i=1}^{b} P(N_{S=L} = i | D_0 = d)} \\ &= (1-B'_{d,1})\sum_{i=1}^{b}(\sum_{j=1}^{b} P(N_{h>1} = j | N_{S=L} = i, D_0 = d)) \end{aligned}$$

$$
\begin{aligned}
& \times \frac{P(N_{S=L} = i | D_0 = d)}{\sum_{k=1}^{b} P(N_{S=L} = k | D_0 = d)} \\
= \quad & (1 - B'_{d,1}) \frac{\Omega'_h(d+1, b) - p^b_{d+1} \Psi'_h(d+1, b)}{1 - p^b_{d+1}}
\end{aligned} \tag{III.9}
$$

where

$$
\begin{aligned}
\Omega'_h(d, x) \quad = \quad & \left(\sum_{i=1}^{h-1} (1 - p_d) A'_{d,i} + p_d B'_{d,i} \right)^x \\
& - \left(\sum_{i=1}^{h-2} (1 - p_d) A'_{d,i} + p_d B'_{d,i} \right)^x .
\end{aligned}
$$

If the evaluation function value of the node is 0, then there must be at least one and fewer than q children with an evaluation function value of 0. Only children with an evaluation function value of 0 are expanded. A similar derivation as given for $B''_{d,1}$ and $B'_{d,h}$ gives:

$$
\begin{aligned}
B''_{d,h} \quad = \quad & (1 - B''_{d,1}) \frac{\sum_{i=1}^{b} S(d+1, i) \sum_{j=1}^{q-1} \sum_k G(i, j, k) \Gamma''_h(d+1, j, k)}{\sum_{i=1}^{b} S(d+1, i) \sum_{j=1}^{q-1} \sum_k G(i, j, k)} \\
= \quad & \sum_{i=1}^{b} S(d+1, i) \sum_{j=1}^{q-1} \sum_k G(i, j, k) \Gamma''_h(d+1, j, k)
\end{aligned} \tag{III.10}
$$

where the summation over k starts at $\max(0, j - b + i)$ and ends at $\min(i, j)$. $\Gamma''_h(d+1, j, k)$ is the probability that node 0 has a height of h given that there are j children with 0 evaluation function values, of which k have LOSE statuses.

$$
\begin{aligned}
\Gamma''_h(d, j, k) \quad \equiv \quad & \left(\sum_{i=1}^{h-1} A''_{d,i} \right)^k \left(\sum_{i=1}^{h-1} B''_{d,i} \right)^{j-k} \\
& - \left(\sum_{i=1}^{h-2} A''_{d,i} \right)^k \left(\sum_{i=1}^{h-2} B''_{d,i} \right)^{j-k}
\end{aligned}
$$

As $p(d+1) \rightarrow 1$, both the numerator and denominator of equations III.9 and III.10 approach 0. To determine the value of $B'_{d,h}$ and $B''_{d,h}$ in this case, let $p(d+1) = 1 - \epsilon$ as $\epsilon \rightarrow 0$. Keeping only terms of order ϵ gives:

$$
\begin{aligned}
B'_{d,h} \quad \rightarrow \quad & (1 - B'_{d,1}) \Phi'_{d+1,h} \\
B''_{d,h} \quad \rightarrow \quad & (1 - B''_{d,1}) \sum_{j=1}^{q-1} \sum_{k=0}^{1} G(1, j, k) \Gamma''_h(d+1, j, k)
\end{aligned}
$$

where

$$\Phi'_{d,h} \equiv \left(\sum_{i=1}^{h-1} B'_{d,i}\right)^{b-1} \sum_{i=1}^{h-1} A'_{d,i} - \left(\sum_{i=1}^{h-2} B'_{d,i}\right)^{b-1} \sum_{i=1}^{h-2} A'_{d,i}.$$

III.H.3 Determining $\alpha'_{d,h}$ and $\alpha''_{d,h}$

The function $\alpha_{d,h}$ is the probability of the search value of a node at height h and depth d indicating a won position when actually the node is a lost position. $\alpha'_{d,h}$ applies to the case where the evaluation function value of the node is 1, and $\alpha''_{d,h}$ applies to the case where the evaluation function value is 0:

$$\begin{aligned} \alpha'_{d,h} &\equiv P(e=1|S=L, H=h, D=d, f=1) \\ \alpha''_{d,h} &\equiv P(e=1|S=L, H=h, D=d, f=0). \end{aligned}$$

The cases where $h=1$ and $h>1$ will be determined separately.

Case 1: A consistent or coerced node ($h=1$).

Referring to Figure III.3, let node 0 be a frontier node (height 1). If node 0 has an evaluation function value of 1, then

$$\begin{aligned} \alpha'_{d,1} &= 1 - P(e_0=0|S_0=L, H_0=1, D_0=d, f_0=1) \\ &= 1 - \frac{P(e_0=0, H_0=1|S_0=L, D_0=d, f_0=1)}{P(H_0=1|S_0=L, D_0=d, f_0=1)} \\ &= 1 - \frac{P(e_0=0, H_0=1|S_0=L, D_0=d, f_0=1)}{A'_{d,1}}. \end{aligned}$$

But since the evaluation function value of node 0 is 0, and this node is consistent, the search value of node 0 must be 1. Hence,

$$\alpha'_{d,1} = 1. \qquad \text{(III.11)}$$

If node 0 has an evaluation function value of 0, the derivation continues from the case for $\alpha'_{d,1}$.

$$\alpha''_{d,1} = 1 - \frac{P(N_{f=0}=0|N_{S=L}=0, D_0=d)}{A''_{d,1}}$$

$$
\begin{aligned}
&= 1-\frac{[P(f_1=1|S_1=W,D_1=d+1)]^b}{A''_{d,1}} \\
&= 1-\frac{(1-\beta)^b}{A''_{d,1}} \qquad \text{(III.12)}
\end{aligned}
$$

Case 2: An inconsistent and non-coerced node ($h > 1$).

Referring to Figure III.3, let node 0 be a node of height h, depth d, and evaluation function value of 1 with a LOSE status. All the children of this node must have an evaluation function value of 1, and they are all expanded.

If $N_{e=0} = i$ is the condition that i children have a search value of 0, then

$$
\begin{aligned}
\alpha'_{d,h} &= 1-P(e_0=0|S_0=L,H_0=h,D_0=d,f_0=1) \\
&= 1-P(N_{e=0}=0|N_{S=L}=0,H_0=h,D_0=d,N_{f=0}=0) \\
&= 1-\sum_{i=1}^{b} P(N_{e=0}=0|N_{S=L}=0,N_{h>1}=i,N_{f=0}=0) \\
&\quad \times \frac{P(N_{h>1}=i|N_{S=L}=0,D_0=d,N_{f=0}=0)}{\sum_{j=1}^{b} P(N_{h>1}=j|N_{S=L}=0,D_0=d,N_{f=0}=0)} \\
&= 1-\frac{\psi'_h(d+1,b)}{\Psi'_h(d+1,b)} \qquad \text{(III.13)}
\end{aligned}
$$

where

$$
\psi'_h(d,x) = \left(\sum_{i=1}^{h-1}(1-\beta'_{d,i})B'_{d,i}\right)^x - \left(\sum_{i=1}^{h-2}(1-\beta'_{d,i})B'_{d,i}\right)^x .
$$

If the evaluation function value of the node is 0, than there must be at least one and fewer then q children with an evaluation function value of 0. Only children with an evaluation function value of 0 are expanded.

$$
\begin{aligned}
\alpha''_{d,h} &= 1-P(N_{e=0}=0|N_{S=L}=0,H_0=h,D_0=d,f_0=0) \\
&= 1-\sum_{i=1}^{q-1} P(N_{e=0}=0|N_{S=L}=0,H_0=h,D_0=d,N_{f=0}=i) \\
&\quad \times \frac{P(N_{f=0}=i|N_{S=L}=0,H_0=h,D_0=d)}{\sum_{j=1}^{q-1} P(N_{f=0}=j|N_{S=L}=0,H_0=h,D_0=d)}
\end{aligned}
$$

Using Bayes' rule,

$$
P(N_{f=0}=i|N_{S=L}=0,H_0=h,D_0=d)
$$

$$\begin{aligned} &= P(H_0 = h | N_{S=L} = 0, D_0 = d, N_{f=0} = i) \\ &\quad \times \frac{P(N_{f=0} = i | N_{S=L} = 0, D_0 = d)}{P(H_0 = h | N_{S=L} = 0, D_0 = d)}. \end{aligned}$$

Combining these two equations gives:

$$\alpha''_{d,h} = 1 - \frac{\sum_{i=1}^{q-1} \binom{b}{i} \beta^i (1-\beta)^{b-i} \psi''_h(d+1, i)}{\sum_{i=1}^{q-1} \binom{b}{i} \beta^i (1-\beta)^{b-i} \Psi''_h(d+1, i)}. \tag{III.14}$$

III.H.4 Determining $\beta'_{d,h}$ and $\beta''_{d,h}$

The function $\beta_{d,h}$ is the probability of the search value of a node at height h and depth d, indicating a lost position when actually the node is won. $\beta'_{d,h}$ applies to the case where the evaluation function value of the node is 1, and $\beta''_{d,h}$ applies to the case where the evaluation function value is 0:

$$\begin{aligned} \beta'_{d,h} &\equiv P(e = 0 | S = W, H = h, D = d, f = 1) \\ \beta''_{d,h} &\equiv P(e = 0 | S = W, H = h, D = d, f = 0). \end{aligned}$$

The cases where $h = 1$ and $h > 1$ will be determined separately.

Case 1: A consistent or coerced node ($h = 1$).

Referring to Figure III.3, let node 0 be a frontier node (height 1). If node 0 has an evaluation function value of 1, then at least one of the children must be 0. Hence,

$$\beta'_{d,1} = 0. \tag{III.15}$$

If node 0 has an evaluation function value of 0, then the search value is 0 only if all the children have an evaluation function value of 1.

$$\begin{aligned} \beta''_{d,1} &= \frac{P(e_0 = 0, H_0 = 1 | S_0 = W, D_0 = d, f = 0)}{P(H_0 = 1 | S_0 = W, D_0 = d, f = 0)} \\ &= \frac{P(N_{f=0} = 0 | S_0 = W, D_0 = d)}{B''_{d,1}} \end{aligned}$$

$$
\begin{aligned}
&= \left(\frac{1}{B''_{d,1}}\right) \sum_{i=1}^{b} P(N_{f=0} = 0 | N_{S=L} = i, D_0 = d) \\
&\quad \times \frac{P(N_{S=L} = i | D_0 = d)}{\sum_{j=1}^{b} P(N_{S=L} = j | D_0 = d)} \\
&= \frac{[\alpha(1 - p(d+1)) + (1-\beta)p(d+1)]^b - [(1-\beta)p(d+1)]^b}{(1 - p_{d+1}^b) B''_{d,1}} \qquad \text{(III.16)}
\end{aligned}
$$

If $p(d+1) = 1$, both the numerator and denominator of Equation III.16 are 0. The value of $\beta''_{d,1}$ can be found in this case by letting $p(d+1) = 1 - \epsilon$ and considering the limit as $\epsilon \to 0$. Then,

$$
\beta''_{d,1} \to \frac{\alpha(1-\beta)^{b-1}}{B''_{d,1}}.
$$

Case 2: An inconsistent and non-coerced node ($h > 1$).

Referring to Figure III.3, let node 0 be a node of height h, depth d, and evaluation function value of 1 with a LOSE status. All the children of this node must have an evaluation function value of 1, and they are all expanded.

$$
\begin{aligned}
\beta'_{d,h} &= P(N_{e=0} = 0 | S_0 = W, H_0 = h, D_0 = d) \\
&= \sum_{i=1}^{b} P(N_{e=0} = 0 | N_{S=L} = i, H_0 = h, D_0 = d) \\
&\quad \times \frac{P(N_{S=L} = i | H_0 = h, D_0 = d)}{\sum_{i=1}^{b} P(N_{S=L} = i | H_0 = h, D_0 = d)}
\end{aligned}
$$

Using Bayes' rule,

$$
\begin{aligned}
P(N_{S=L} = i | H_0 = h, D_0 = d) &= P(H_0 = h | N_{S=L} = i, D_0 = d) \\
&\quad \times \frac{P(N_{S=L} = i | D_0 = d)}{P(H_0 = h | D_0 = d)}.
\end{aligned}
$$

Combining these results gives:

$$
\beta'_{d,h} = \frac{\theta'_{d+1,h} - p_{d+1}^b \psi'_{d+1,h}}{\Omega'_{d+1,h} - p_{d+1}^b \Psi'_{d+1,h}} \qquad \text{(III.17)}
$$

where

$$
\begin{aligned}
\theta'_{d,h} &= \left(\sum_{i=1}^{h-1} \alpha'_{d,i} A'_{d,i}(1 - p_d) + (1 - \beta'_{d,i}) B'_{d,i} p_d\right)^b \\
&\quad - \left(\sum_{i=1}^{h-2} \alpha'_{d,i} A'_{d,i}(1 - p_d) + (1 - \beta'_{d,i}) B'_{d,i} p_d\right)^b.
\end{aligned}
$$

If the evaluation function value of the node is 0, then there must be at least one and fewer than q children with an evaluation function value of 0. Only children with an evaluation function value of 0 are expanded.

$$\begin{aligned}\beta''_{d,h} &= \sum_{i=1}^{b} P(e_0 = 0 | N_{S=L} = i, H_0 = h, D_0 = d) \\ &\quad \times \frac{P(N_{S=L} = i | H_0 = h, D_0 = d)}{\sum_{i=1}^{b} P(N_{S=L} = i | H_0 = h, D_0 = d)}\end{aligned}$$

Using Bayes' rule,

$$\begin{aligned}P(N_{S=L} = i | H_0 = h, D_0 = d) &= P(H_0 = h | N_{S=L} = i, D_0 = d) \\ &\quad \times \frac{P(N_{S=L} = i | D_0 = d)}{P(H_0 = h | D_0 = d)}.\end{aligned}$$

Combining these results gives:

$$\beta''_{d,h} = \left(\frac{1}{B''_{d,h}}\right) \sum_{i=1}^{b} S(d+1, i) \sum_{j=1}^{q-1} \sum_{k} G(i, j, k) \lambda''_h(d+1, j, k) \tag{III.18}$$

where

$$\begin{aligned}\lambda''_h(d, j, k) &\equiv \left(\sum_{i=1}^{h-1} \alpha''_{d,h} A''_{d,i}\right)^k \left(\sum_{i=1}^{h-1} (1 - \beta''_{d,h}) B''_{d,i}\right)^{j-k} \\ &\quad - \left(\sum_{i=1}^{h-2} \alpha''_{d,h} A''_{d,i}\right)^k \left(\sum_{i=1}^{h-2} (1 - \beta''_{d,h}) B''_{d,i}\right)^{j-k}\end{aligned}$$

and the summation over k starts at $\max(0, j - b + i)$ and ends at $\min(i, j)$.

If $p(d+1) = 1$, both the numerator and denominator of Equation III.17 and III.18 are 0. To determine the correct value of $\beta'_{d,h}$ and $\beta''_{d,h}$ in this case, let $p(d+1) = 1 - \epsilon$ and let $\epsilon \to 0$. Then,

$$\beta'_{d,h} \to \left(\frac{1}{\Phi'_{d+1,h}}\right) \phi'_{d+1,h}$$

where

$$\begin{aligned}\phi'_{d,h} &= \left(\sum_{i=1}^{h-1} (1 - \beta'_{d,i}) B'_{d,i}\right)^{b-1} \sum_{i=1}^{h-1} \alpha'_{d,i} A'_{d,i} \\ &\quad - \left(\sum_{i=1}^{h-2} (1 - \beta'_{d,i}) B'_{d,i}\right)^{b-1} \sum_{i=1}^{h-2} \alpha'_{d,i} A'_{d,i}\end{aligned}$$

and

$$\beta''_{d,h} \rightarrow \left(\frac{1}{B''_{d,h}}\right) \sum_{j=1}^{q-1} \sum_{k=0}^{1} G(1,j,k)\lambda''_h(d+1,j,k).$$

III.H.5 Determining π

The function $\pi(x, y, i)$ is the probability of making an error given a specific sequence of child node statuses. Argument x is the probability of estimating a lost child as won, y is the probability of estimating a won child as lost, and i the number of children with a LOSE status. An error occurs if a move is made to a child whose status is WIN. This can happen if the evaluation function incorrectly estimates that such a position is lost. If there are n children whose evaluations indicate a lost position, but m of these nodes are actually won positions, the probability of randomly choosing an incorrect node is m/n. Let index j be the number of LOSE nodes which are correctly evaluated to be lost positions, and index k be the number of WIN nodes which are incorrectly evaluated to be lost positions. Then,

$$\begin{aligned} \pi(x,y,i) &= \left(\frac{b-i}{b}\right) x^i (1-y)^{b-i} \\ &\quad + \sum_{j=0}^{i} \sum_{k=1}^{b-i} \left(\frac{k}{j+k}\right) \binom{i}{j} \binom{b-i}{k} (1-x)^j x^{i-j} y^k (1-y)^{b-i-k}. \end{aligned}$$

The first term accounts for the possibility that every child is evaluated to be a won position.

Chapter IV

SAL

SAL (Search And Learning) is a game-learning program designed to test the ideas and theoretical results presented in previous chapters. The program was written in ANSI C and runs on a variety of machines, including the Macintosh SE-30 and the Cray Y-MP. The program was not designed to compete in computer game-playing tournaments, and hence does not include many techniques that have been shown to improve the playing strength of competition programs, such as transposition tables, aspiration windows, and various heuristics [54, 25]. Although these techniques would improve SAL's performance, they were not necessary to test the search methods and learning issues described in this dissertation.

This chapter provides a detailed description of the SAL program. Section IV.A gives a top–level description of the the major modules of the program. Specifics of the search algorithm are presented in Section IV.B. The learning portion of the program is described in Sections IV.C and IV.D.

IV.A Architecture

The SAL program consists of two major modules, shown in Figure IV.1; a game–independent kernel and a game–specific move generator. The SAL kernel includes both the search and learning routines, and is *not* modified for different games.

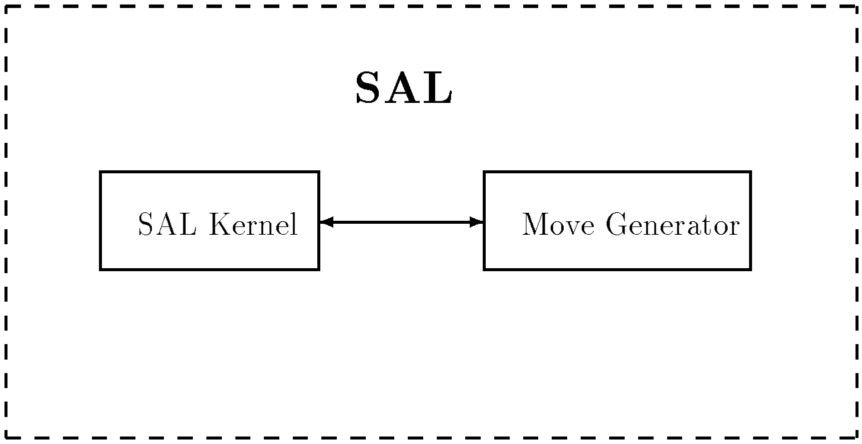


Figure IV.1: A top-level diagram of the SAL game-learning program.

The rules of the game to be played are given to the SAL program by providing a move generator, which is necessarily different for each different type of game SAL is to play. A move generator was written for the games of tic-tac-toe, connect-four, and chess to obtain the results described in Chapter V.

IV.A.1 Move Generator

The move generator incorporates the rules of the game to be played. These rules must be provided by the user. Rather than develop a sophisticated interface to allow a user to tell SAL the rules of arbitrary games, the user communicates the rules of a game to SAL by providing three subroutines.

MoveGenerator This routine is passed a board position and the player whose turn it is to move and must return all the legal moves that are possible from the given position.

MakeMove Given a move, this routine modifies the board position appropriately. This routine must also keep track of any non-positional information.

EndOfGame Given a board position, this routine returns whether the game is over and, if so, which player won the game or whether it was drawn.

In addition, the size of the board and the number and types of the pieces must also be provided. These constants, as well as the three subroutines, are compiled with the SAL kernel to make the SAL executable. The initial board configuration must be provided in a file that is read during execution.

IV.B Search Algorithm

SAL uses the consistency search algorithm described in Chapter III combined with the standard alpha-beta search procedure [26]. For each position at which it is SAL's turn to move, a full-width, two-ply search tree is generated. An incremental consistency search is performed to determine which nodes of this search tree to expand further. Nodes that are cutoff by the alpha-beta algorithm are not considered for expansion. The search tree is selectively expanded and the q–level of the search is increased until all the frontier nodes are consistent or time is exhausted. A move is chosen based on the results of the completed search with the highest q–level.

IV.C Evaluation Functions

SAL uses two evaluation functions; one for the player who moves first, the other for the player who moves second. Positions where it is the first player's turn to move are used to train the first player's evaluation function, and similarly for the second player's evaluation function. A single evaluation function would learn from moves made by both players, and hence would learn in fewer games. However, a single evaluation function would also limit the domain of possible games to only those which were *symmetric*, in that a good position for one player is also a good position for the other if the pieces were swapped. Since SAL has two evaluation functions, it can also learn non-symmetric games.

Each evaluation function is a backpropagation neural network with a single layer of hidden units. The network is fully connected between layers. The number of hidden units used for the tests reported in this dissertation was one for every ten input units. This choice, as with the choice of learning rate and the range of the initial weights was rather arbitrary, but was based somewhat on the results reported by Tesauro in [63].

Chapter II presented the argument that the evaluation function learning should not be relied on to achieve expert play in games such as chess. Search is also required. Consistent with this argument, a minimal amount of effort was spent tuning the neural networks. Although other learning algorithms were not tried, one of the benefits of adding a search procedure is to minimize the sensitivity of the program to weaknesses of the various learning methods.

IV.D Board Game Features

Chapter II described the feature-discovery problem which must be addressed by a game-learning program. It was noted that relying solely on the evaluation function learning algorithm to discover the relevant features of a game would probably require too many training examples to be practical in the game-learning domain. For this reason, a set of features applicable to all games was developed for SAL. These features are all computed from the moves already made, or from the "raw" board position using the rules of the game as provided by the move generator.

There is a binary input unit to the neural network corresponding to each feature. The activation value of the input unit is given by the value of the associated feature. The features can be divided into the following three categories:

Positional Features: The value of these features are computed from the current board position.

- **The raw board position:** There is a set of feature for each square on the board. The features in this set correspond to the different piece types. The feature corresponding to the piece type occupying the square, if any, is set to one. The rest are set to zero.

- **The number of each piece type on the board:** There are a set of features used to encode the number of each type of piece on the board. The version of SAL that learned to play chess used three features for each piece type. Specifically, if there is one white pawn, the first feature of the set associated with the number of white pawns is set to one, and the other two features are set to zero. If there are two white pawns, the first two features are set to one. If there are three or more white pawns, the first two features are set to one and the third feature is set to the number of white pawns in excess of two on the board. This encoding scheme was chosen somewhat arbitrarily.

Non–positional Features: The value of these features are computed from the move that was made to arrive at the current board position.

- **The type of piece moved:** There is a feature for each of the possible types of pieces in the game. The feature corresponding to the type of the piece just moved is set to one.
- **The type of piece captured:** The feature corresponding to the type of the piece just captured is set to one.

Rule–based Features: The value of these features are obtained from the board positions that could occur after the next move, and from the board positions that could occur if the player who just moved were to move again. These board positions are all computed by a call to the move generator.

- **The pieces potentially lost:** There is a set of features used to encode the number of each type of piece that could be captured if the player who just moved were able to immediately move again. In chess, this would correspond to the pieces under attack.
- **The squares potentially lost:** There is a feature for each square of the board. If the player who just moved could occupy the square if allowed to

immediately move again, the feature is set to one.

- **The pieces potentially gained:** There is a set of features used to encode the number of each type of piece that could be captured this move. In chess, this would correspond to the pieces attacked.
- **The squares potentially gained:** There is a feature for each square of the board. If the player whose turn it is to move could occupy the square, the associated feature is set to one.
- **A possible win:** This feature is set to one if it is possible to win the game with the next move.
- **A possible loss:** This feature is set to one if the player who just moved could win the game if allowed to immediately move again. In chess this would correspond to a check.

IV.E Temporal Difference Learning

Each move of the game being played provided a training example for one of the neural network evaluation functions. A temporal difference algorithm [61] was used to provide the target value for each example. The value of λ was set to 0. The evaluation value of the next board position was used as the target value for the evaluation value of the current position. The backpropagation algorithm [51] was used to determine the amount each weight of the network should be changed. The weights of the network were actually changed after each game was over.

Chapter V

Performance of SAL

This chapter presents the results of testing the SAL program on the games of tic-tac-toe, connect-four[1], and chess. Since the optimum strategy in tic-tac-toe is known, it is a useful game to demonstrate the general behavior of SAL as well as to examine the performance of the search algorithm. The game of connect-four is significantly more complex than tic-tac-toe, and provides a successful demonstration of the SAL program compared with a traditional alpha-beta search program for this game. The results of SAL playing the game of chess demonstrate the effectiveness of the search and learning procedures in a considerably more difficult game.

V.A Tic-Tac-Toe

The game of tic-tac-toe, or noughts and crosses, is a simple game which was used to test some of the characteristics of the SAL program. The game is simple since generally only the first three half-moves allow for any variation, the remaining moves being either forced or irrelevant. A naive estimate of the size of the full game tree would be roughly $9! = 362,880$ nodes, although a more careful count including early termination and forced moves gives $75,482$ nodes [2]. A game tree of this size could easily be searched completely in a reasonable length of time, however SAL's search

[1]Connect-Four is a trademark of the Milton-Bradley company

tree was limited to about 100 nodes. This node limitation forced the search procedure to prune many lines of play without looking deeper along those lines, which is what must be done in more complex games such as chess.

V.A.1 SAL versus TTT

SAL was tested by playing against an imperfect, "expert" tic-tac-toe program called TTT. TTT would always complete a tic-tac-toe if possible, and would always block a potential tic-tac-toe by SAL. If neither of these conditions exist in the current board position, TTT would move to a randomly selected square.

For a 3×3 board with 2 piece types, there are 61 features computed by SAL. The neural network evaluation functions have $61 \times 7 \times 1$ units and 442 weights. The weights were initialized to random values between -0.05 and +0.05. The learning rate was set to 0.01. The size of the search tree was limited to about 100 nodes. Figure V.1 shows how frequently SAL loses as a function of the games played. The performance of a full-width, 1-ply search, and a full-width, 2-ply search are shown for comparison.[2]

Figure V.1 demonstrates that SAL is capable of learning a good strategy in the game of tic-tac-toe. The figure shows a comparison between the performance of the consistency search and both a one-ply and a two-ply search. The two-ply search does better than the consistency search until a sufficient number of games are played. This was predicted in Chapter III since the consistency search procedure is only beneficial for sufficiently accurate evaluation functions, and this accuracy is only learned after numerous games are played.

The comparison between the consistency search and a full-width, two-ply search is relevant since the consistency search performs a full-width, two-ply search prior to selectively expanding additional nodes. If the selective expansion performed by the consistency search was not beneficial, the full-width, two-ply search results

[2]The figure was made using the S program [13] and the curves drawn using the lowess scatterplot smoothing function.

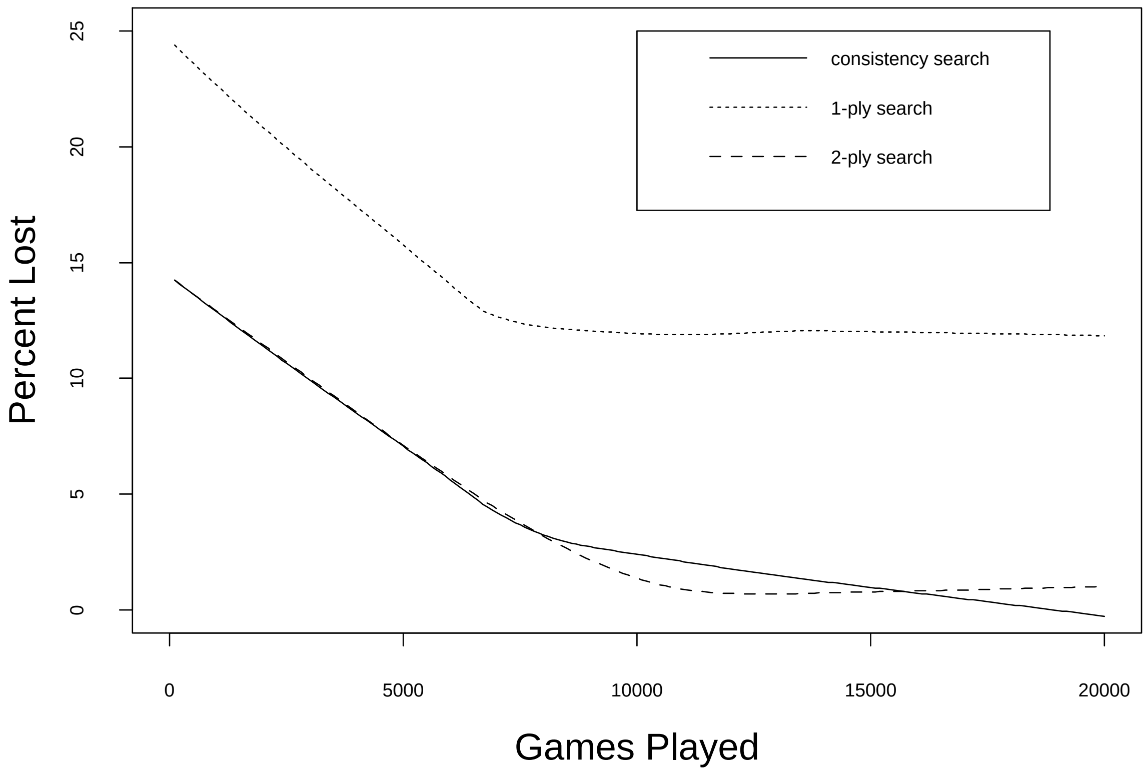


Figure V.1: SAL's performance against a tic-tac-toe program.

would be superior to the consistency search results. In pathological games, the one-ply search would perform better than the two-ply search. Hence the results of a one-ply search are also shown in figure V.1.

V.A.2 An Example Search Tree

An example search tree is shown in figure V.2. This search tree was generated after 10,000 games had been played against TTT. It was not possible to show the board position at each of the 93 nodes in the tree. Hence, positions are only given for several interesting nodes. The board position at each node can be determined since the nodes are generated in a left-to-right, top-to-bottom order. Specifically, the leftmost child node is the board position resulting from moving into the first empty square in a left-to-right, top-to-bottom order on the board. The board position for the next child corresponds to a move to the next empty square, and so on. The evaluation value of each position is shown in the figure.

Notice that the position labeled A was evaluated as -0.12, which is less than the value of -0.07 of the root position to player x. This indicates that position A is estimated to be a worse position for player x than the root position. However, player x can win the game by moving to the lower right-hand square. Thus the evaluation of position A is incorrect.

The consistency search procedure evaluates the children of node A and finds an inconsistency between the evaluation value of position A, and its negamax value. Specifically, the negamax value of A is 0.0, which is greater than the root position value of -0.07 to player x. This indicates that position A is a better position for player x, rather than a worse position as initially estimated. Since the negamax value of 0.0 results from the evaluation of position B, either position A or position B must be incorrectly evaluated.

At first, the q-level of the search is 1 and it is automatically assumed that the value of the deeper node, position B, is correct. Node A is said to be coerced. The value of 0.0 for position A is used for selecting a move for this iteration. However, the next iteration of the search is for a q-level of 2, which results in the expansion of node B to determine which position, either A or B, is evaluated incorrectly. The negamax value of position A after this expansion is 0.5, indicating that position A is better than the root position for player x. Thus the consistency search procedure was able to determine that the evaluation value of position A was incorrect.

The expansions of the children of nodes C, D, and E are similar to the case just described, except that all the children must be expanded to determine which child, or the parent, is incorrectly evaluated. In this case it is determined that positions D and E are incorrectly evaluated, although they all lead to draws.

The search terminated after all the frontier nodes were consistent. The move leading to position E was finally chosen. Notice that a full-width, one-ply search would have incorrectly chosen the move to position A due to the erroneous evaluation of this position. It should be noted that the last two children of node C were cutoff in the last iteration of the search due to the use of the alpha-beta algorithm. In this case, the cutoff didn't matter since both these children were consistent. However, even if the children were not consistent, there would be no further expansions along these lines of play due to the alpha-beta cutoff.

V.B Connect-Four

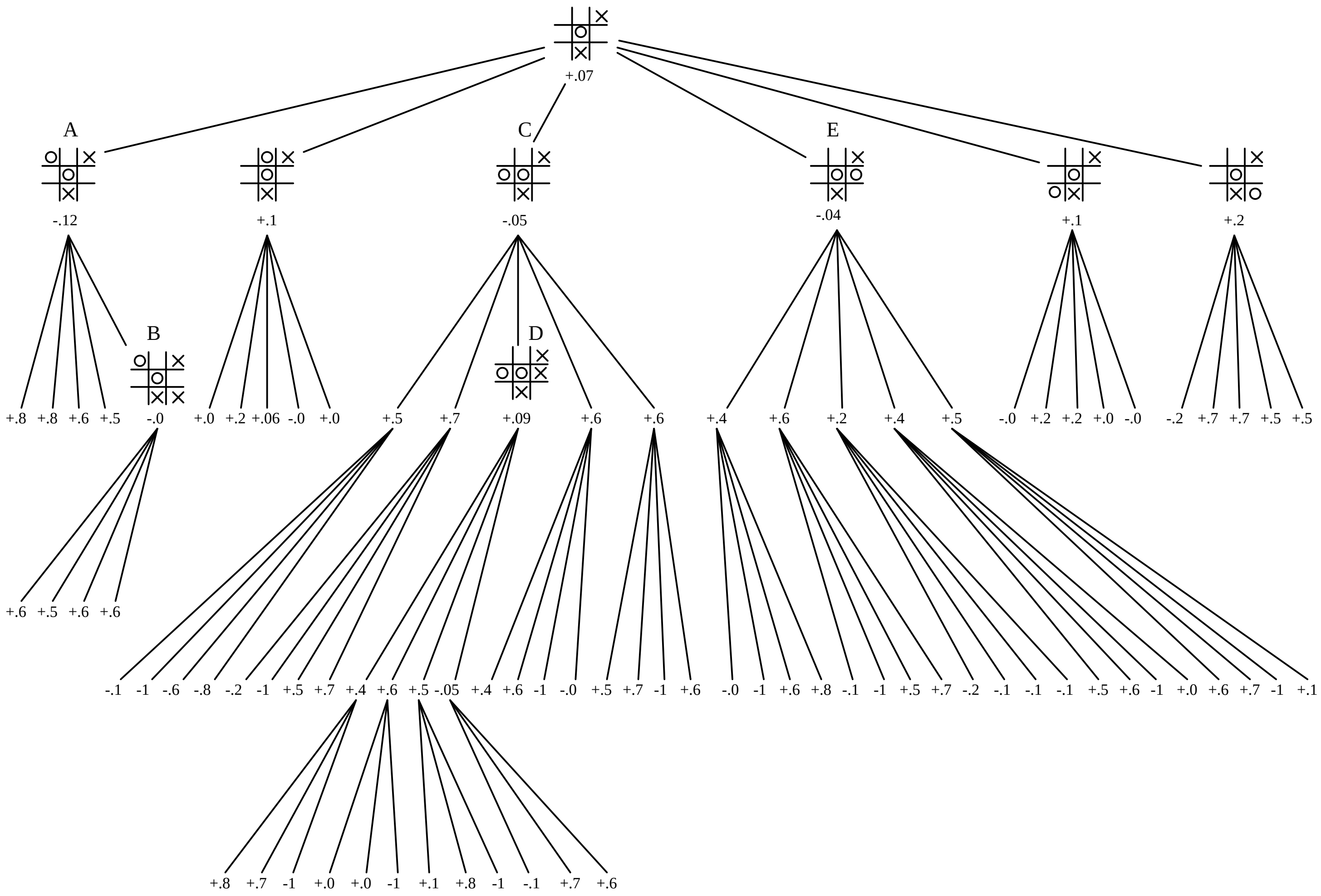
The game of connect-four was chosen as another test of the SAL program. The game is available commercially and consists of a 7 column frame that stands vertically with 6 slots per column. Each player has a number of identical pieces that can fit into the slots. Players alternate moves by dropping one of their pieces into one of the columns. The piece falls to the lowest unoccupied slot in the column. The winner is the player that gets four of their pieces in a line, either vertically, horizontally, or diagonally. A draw is possible if all the slots become filled without either player winning.

This game was selected for several reasons. It has a constant branching factor of 7 until near the end of the game, which is consistent with the assumptions used to analyze the consistency search procedure in Chapter III. It has simple rules which enabled the move generator for SAL to be written in a couple of hours. In addition, connect-four was used to test the min/max approximation procedure in [50] where a simple heuristic evaluation function was described. Lastly, it is a considerably more complex game than tic-tac-toe.

V.B.1 SAL versus C4

An opponent program, called C4, was written to play connect-four against SAL. C4 performs the standard alpha-beta, iterative deepening search used in most game playing programs. This is an alpha-beta minimax search to two ply, followed

Figure V.2:



An example of a search tree generated by the SAL program playing tic-tac-toe.

by a search to three ply, etc. until a specified number of nodes have been created in the search tree. The move is selected based on the completed search with the greatest depth. Unlike the program described in [50], no move ordering was performed. The C4 program was limited to search trees with less than 1000 nodes. The first move made by the C4 program was made at random.

The evaluation function described by Ronald Rivest in [50] was used. This evaluation function is based on the number of partially filled "segments" a move would create. A segment is a group of four slots in a line. The sum of scores from all the segments a move would create is the resulting position's evaluation value. A score of 1 is given to a segment consisting of only one piece, a score of 10 is given to a segment with two pieces of the same type, and a score of 50 is given to a segment consisting of three pieces of the same type. Segments with more than one type of piece are given a score of zero.

A 7×7 board with two piece types was used for the SAL program. Moves to the top row of the board were considered illegal by the move generator. The feature vector had 221 elements which resulted in $221 \times 23 \times 1$ unit neural network evaluation functions. The weights were initialized to random values between -0.05 and $+0.05$. The learning rate was set to 0.01. SAL was limited to search trees with less than about 500 nodes.

Figure V.3 shows how often SAL won against the C4 program as a function of the number of games played. Curves are given for the case where SAL performs a consistency search as well as for the cases where a two ply, and only a one ply search are performed. Several tests were performed with different random values. All gave similar results.

As with tic-tac-toe, figure‘V.3 shows that the consistency search is superior to both a one ply and a two ply full-width search. Based on the performance of the one ply case, it is apparent that the evaluation function starts out very inaccurate, but eventually becomes fairly accurate as more games are played. Hence the curve for the consistency search procedure is consistent with the qualitative curve shown in

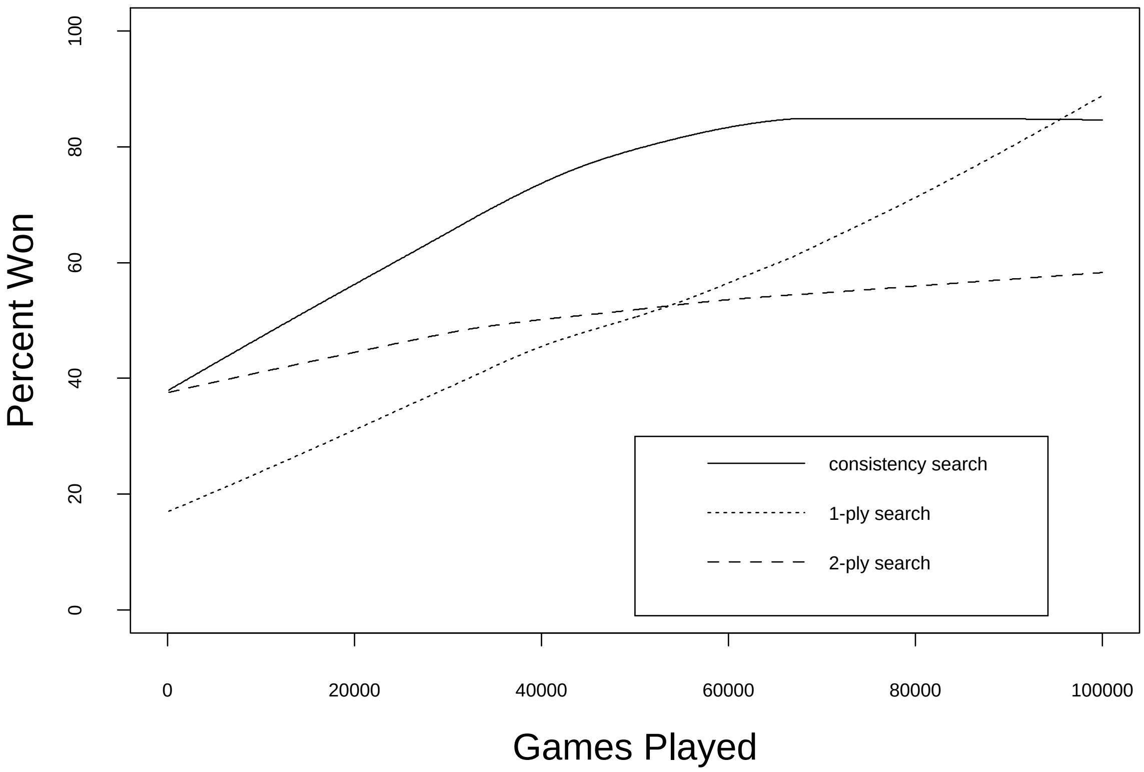


Figure V.3: SAL's performance against a connect-four program.

figure II.4 and the analytical results presented in Chapter III. It is interesting that the one ply search is eventually superior to the two ply search, which is an indication of a pathological game.

V.C Chess

The game of chess was also used as a test environment for the SAL program. Chess is a very complex game, with a branching factor of about 35 and usually over 50 moves per game. It was chosen as a test environment since chess playing programs are readily available and since most people are familiar with the game and its difficulty. There have also been numerous published reports on the strength of various different chess playing programs, and a well-established rating system can be used to measure the strength of a program relative to human players.

V.C.1 SAL versus GNUCHESS

GNUCHESS is a public-domain chess program distributed with the source code by the Free Software Foundation. The program is written in C and has been ported to a number of different machines. Under tournament conditions it plays a master-level game of chess. For testing the SAL program, GNUCHESS was set to make a move in about one second, which reduced its playing strength to a rating of about 1500–1600, which is the strength of the average tournament chess player.

A chess move generator was written for the SAL program to provide the program with the rules of the game. For a 9×9 board with 12 piece types, there are 1031 features computed by SAL. The neural networks have $1031 \times 104 \times 1$ units. As with tic-tac-toe, the weights were initialized to random values between -0.05 and +0.05, and the learning rate was set to 0.1. Players alternate who moves first. The size of the search tree was limited to about 3000 nodes.

Since the SAL program is given no information about chess except the rules, it starts off playing very badly. To quantify the program's improvement, several factors were measured. Figure V.4 is a plot of the number of moves SAL makes before the game is over. A game where SAL moves randomly averages about 15 moves before GNUCHESS wins. Of the 4200 games played, SAL drew 8 and lost the rest (SAL has actually won a couple of games against GNUCHESS, but they have all been due to a bug in the GNUCHESS program). Figure V.5 is a plot of the number of material points SAL captures before the end of the game. The points are: 1 for a pawn, 3 for a bishop and knight, 5 for a rook, and 9 for a queen. There are 39 possible points in a game if no pawns are promoted. On average, SAL is searching to a depth of 4 ply with a search tree of 1500 nodes.

An interesting feature of both Figures V.4 and V.5 is that the performance curve is continuing to rise somewhat linearly. This is notably different than what would be expected from a one-ply search, which would level off [29]. Although this

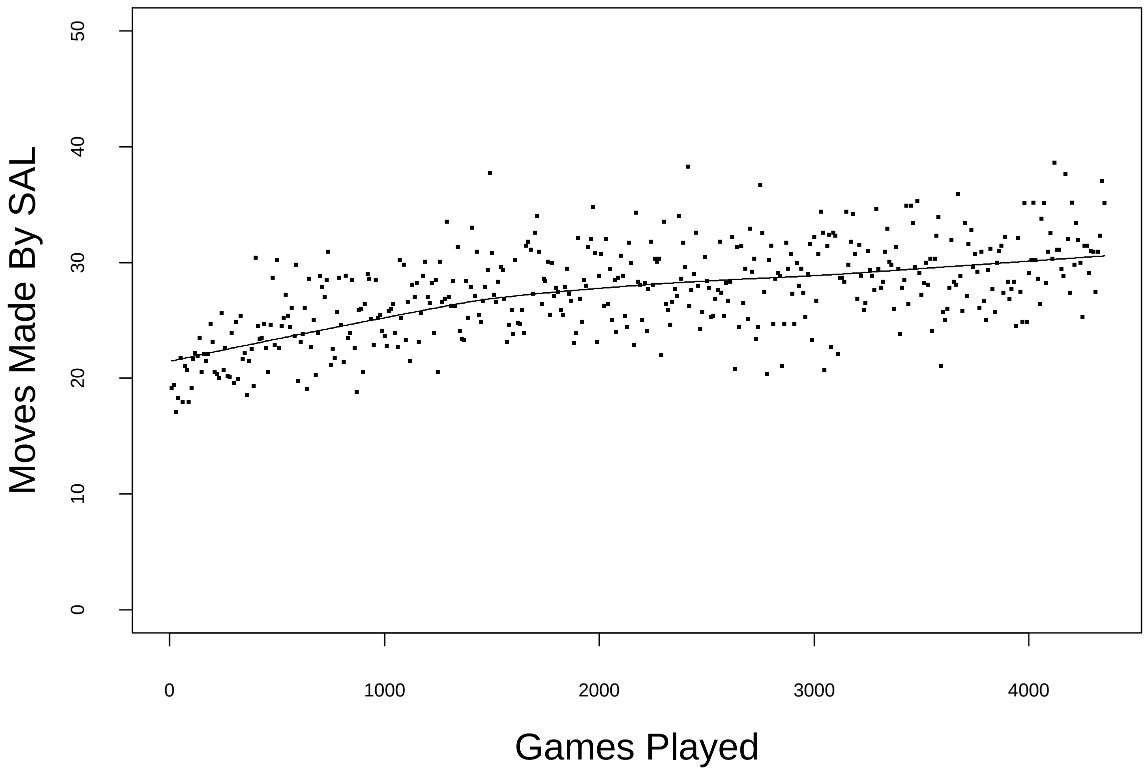


Figure V.4: SAL versus GNUCHESS — The number of moves SAL makes before the game is over. Each data point is the average of 10 games.

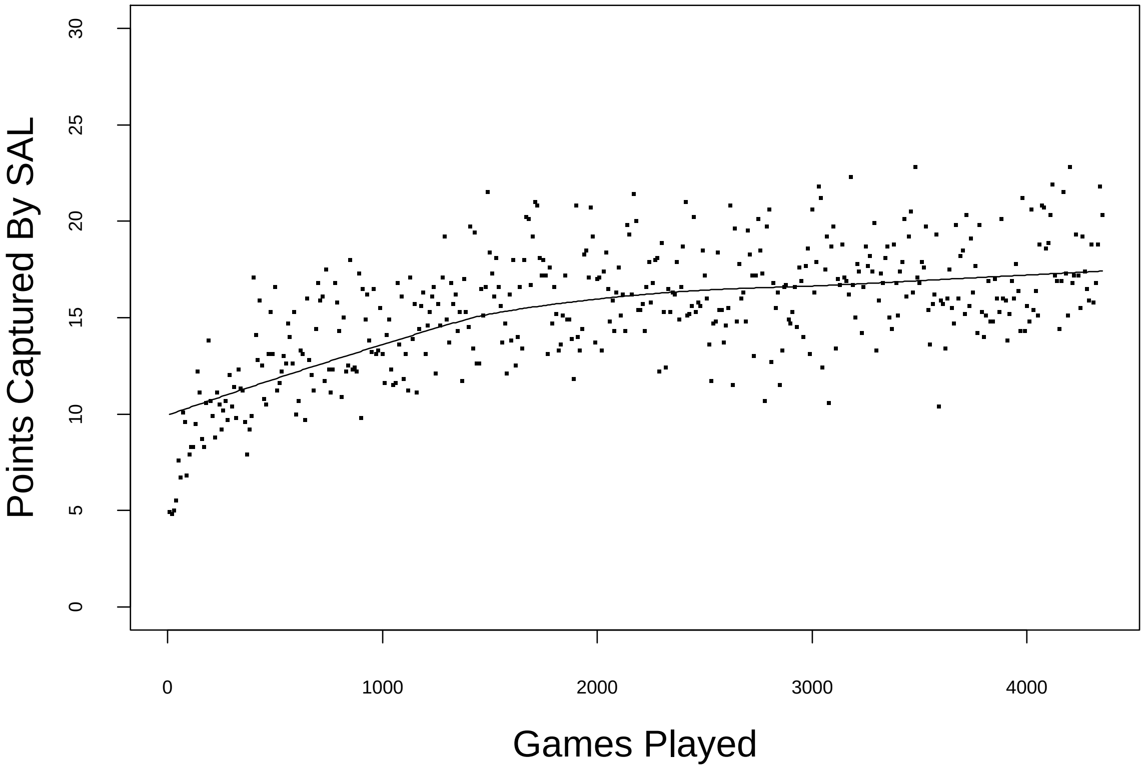


Figure V.5: SAL versus GNUCHESS — The number of points SAL captures before the game is over. Each data point is the average of 10 games.

appears to be a positive result, more chess games must be played to make more definitive conclusions. The SAL program plays about 150 games a day against GNUCHESS when running on a machine such as a Convex C-240 or a Cray EL.

V.C.2 Sample Games

In this section several games between SAL and GNUCHESS will be given. Many sequences of games were played, each sequence consisting of from one to several thousand individual games. Every sequence was started with a random assignment of weights to the evaluation functions. The GNUCHESS program chooses its opening moves randomly from a book. The selection of games chosen to be included in this section were in some sense typical of the many different games observed.

The First Game

Actually, there were many first games, one for each sequence of games that was played. The shortest game played was fool's mate, which has two moves. The first game in the sequence of games shown in Figures V.4 and V.5 lasted 14 moves. These are shown below with the number of ply searched and the number of nodes in the search tree given in parentheses.

	SAL		**GNUCHESS**
1.	c4	(2 — 421)	e5
2.	Qc2	(2 — 680)	Nc6
3.	Qd3	(3 — 1917)	Nf6
4.	Kd1	(3 — 1111)	d5
5.	Nh3	(2 — 1321)	BxN
6.	e3	(2 — 1357)	...

The SAL program is started with no knowledge of the value of the pieces, since this is not in the rules of chess.

	SAL		**GNUCHESS**
6.	...		Bg4 check

The consistency search procedure starts from a full-width, two-ply search tree. Any move that leaves SAL's king in check will be evaluated as a losing move since SAL's king will be captured on the second ply of the search.

	SAL		**GNUCHESS**
7.	Be2	(3 — 1340)	BxB
8.	KxB	(3 — 1369)	Bb4
9.	Nc3	(2 — 1339)	PxP
10.	Nd1	(2 — 1613)	...

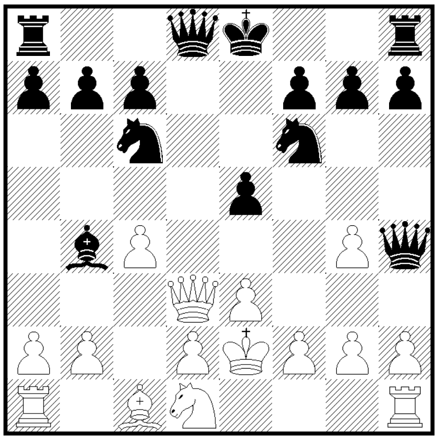


Figure V.6: First Game — Position after 10. Nd1 ...

	SAL		**GNUCHESS**
10.	...		QxQ check
11.	Kf3	(3 — 1069)	Nd4 check
12.	Kg3	(3 — 1089)	Qg6 check
13.	Kh4	(3 — 1074)	Qg4 mate

Although move 13 indicates a mate, SAL actually makes one final move since the end of the game has been defined to be when a king is captured.

	SAL		**GNUCHESS**
14.	Kh5	(3 — 1193)	NxK

A Drawn Game

In the sequence of games shown in Figures V.4 and V.5, 8 games ended in a draw. All were drawn due to perpetual check. A description of the 3100th game follows.

	GNUCHESS	**SAL**	
1.	Nf3	e6	(3 — 1117)
2.	Nc3	c5	(3 — 1734)
3.	e3	Nc6	(8 — 1908)
4.	Bb5	a6	(3 — 2400)
5.	BxN	Pd7xB	(3 — 2051)
6.	0–0	Bd6	(3 — 1241)
7.	Ne4	c4	(6 — 2702)
8.	NxB	QxN	(2 — 923)

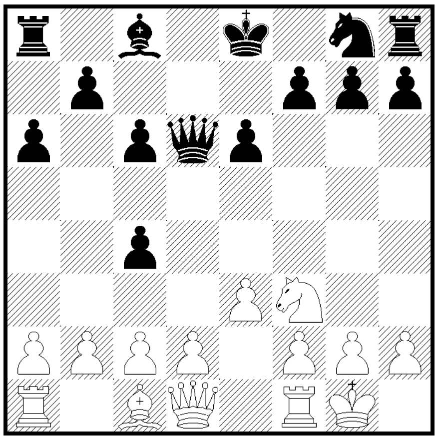


Figure V.7: Drawn Game — Position after 8. ... QxN

9.	Qe2	b5	(2 — 1016)
10.	b3	Nf6	(7 — 2679)
11.	Bb2	Ne4	(8 — 2601)
12.	BxP	Rg8	(3 — 1789)
13.	Be5	RxP check	(2 — 1761)
14.	KxR	Qc5	(2 — 1614)
15.	PxP	f6	(5 — 2479)
16.	d3	PxB	(2 — 1389)
17.	PxN	PxP	(2 — 813)
18.	Ng5	Ra7	(2 — 904)
19.	Qh5 check	Kf8	(10 — 2269)
20.	NxPh7 check	RxN	(9 — 2286)
21.	QxR	Qe7	(4 — 959)
22.	Qh8 check	Kf7	(7 — 2963)
23.	QxB	Qh4	(2 — 808)
24.	Qd7 check	Kf8	(3 — 1021)
25.	QxPc6	Qg4 check	(2 — 830)
26.	Kh1	Qf3 check	(2 — 779)
27.	Kg1	Qg4 check	(2 — 713)
28.	Kh1	Qf3 check	(d — 779)
	draw		

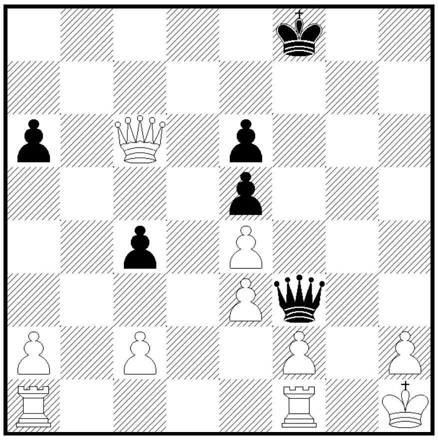


Figure V.8: Drawn Game — Position after 28. ... Qf3 check

GNUCHESS was clearly winning prior to move 25. Since all the drawn games SAL has played have been due to perpetual check, it seems that there may be a bug in the GNUCHESS program that doesn't allow for this possibility. The software was not examined to determine if this were the case. In any event, SAL's performance is much improved over its earlier games.

Chapter VI

Conclusion

As described in Chapter I, the SAL program was designed as a step toward an answer to Shannon's question [59]: "Could a machine be designed that would be capable of 'thinking'?" Specifically, it was proposed that a "thinking" machine would need to be able to learn to play games well by simply playing the game. The SAL program was developed to test some ideas about how such a game-learning program might be designed.

This Chapter reflects on the results of the previous chapters, and attempts to assess their significance. Section VI.A summarizes the ideas and results of this dissertation that are thought to be of significant value. Section VI.B describes some of the areas that seem to have been imprecisely addressed, though no better approach is apparent. Section VI.C lists some of the improvements that the next version of the SAL program might contain. Finally, Section VI.D presents some aspects of the full game-learning domain that are not present in the domain selected for the SAL program but which would make a successful program more useful in practical applications.

VI.A Summary of Important Ideas and Results

- In 1950, Shannon [59] proposed the chess-playing problem as a means of answering the question: "Could a machine be designed that would be capable of 'thinking'?" Research to develop a good chess-playing program has helped to improve computer performance in numerous more practical applications. Chess playing programs are now capable of beating most human players, and yet few believe that the programs are "thinking". This dissertation proposes that the game-learning problem replace the chess-playing problem as a good test-bed for AI research.

- An argument was presented for the need for using some form of game tree search to achieve good play in complex games such as chess. A modified version of Beal's consistency search procedure was developed and analytically shown to be beneficial under reasonable conditions. This procedure was implemented in the SAL game-learning program.

- A set of evaluation function features was described applicable to a large class of different games. The features allow a learning algorithm to concentrate more on the feature-selection task rather than the much more difficult feature-discovery task. The features are computable from the raw board position, the move that was just made, the possible board positions after the opponents next move, and the possible board positions if the player who just moved could immediately move again. These features were tested in the SAL program.

- SAL, a game-learning program, was developed and tested using the games of tic-tac-toe, connect-four, and chess. The results of these tests showed that performance on tic-tac-toe and connect-four were as predicted; the results using chess were positive though inconclusive due to computational limitations. The results of SAL learning the game of connect-four clearly show the benefit of using the consistency search procedure.

VI.B Open Issues

This section highlights some of the areas which were not thoroughly addressed in the previous chapters due to either a lack of a better idea about how to handle them or some intuitive notions that they would not have a significant effect on the results. Additional comments are made for areas that may be of some concern to the reader.

VI.B.1 Consistency Thresholds

The evaluation functions used by the SAL program returned a continuous value. This value was converted into either a 0 or a 1 based on a threshold set by the evaluation value of the root position. This bi-valued evaluation was propagated by the consistency search procedure.

Test results from the SAL program indicated that the root node value of the initial board position varied widely with each new game. This implies that a significant amount of error is being introduced into the consistency search due to this choice of threshold. Some of this variability may be due to the fact that there is a random element to the opponent program's first move. A detailed investigation of the variance of the root node value in mid-game positions was not done.

The method of setting the threshold using the value of the root node was chosen because it was believed that the value of positions that actually occur in games would be more reliable, and because it simplified the mathematical analysis. The method is intuitively unsatisfying however, since it is based on the evaluation value of a single position, which is assumed in the consistency search procedure to be erroneous. A better method is probably required.

VI.B.2 Independent Evaluations

The analysis of consistency search assumed that the error in the evaluation value of the parent and the child positions are independent. This would seem to be a bad assumption since the features of the two positions would be nearly identical. This problem has been noted by many researchers, but none have suggested a better assumption. Since the SAL program uses two evaluation functions, one for each player, it may more closely approximate this assumption than would a program using a single evaluation function.

VI.C The Next Generation SAL

Some ideas for improving the SAL program are:

- SAL must play many thousands of games before it achieves expert play, even for simple games. One method of improving this learning rate would be to learn from some of the information in the search tree, similar to Samuel's checker program [53]. This was not done in the current version of SAL. The alpha-beta search procedure used by SAL reduces the size of the search tree, but leaves some nodes with values different from their negamax values. This could introduce errors in a learning procedure which used the evaluation values of the nodes as target values. Also, there could be unusual feedback effects when learning from sequences of moves whose final outcome will remain unknown.

- Some of the features used by SAL were generated from board positions that might occur after the opponents next move, or if the player who just moved could immediately move again. For the game of chess this resulted in roughly 70 different board positions being used to determine the value of only a few features. It was necessary to limit the number of features generated from these possible board positions due to computational limitations. Using more of the information from each of these possible positions might yield better results.

- The search procedure was designed assuming there is only one evaluation error among a parent and its children. The procedure attempts to correct such an error. A similar procedure could be designed which attempts to correct up to two evaluation errors. This procedure would consider the evaluation values of the parent, its children, and its grandchildren. It is not known whether the additional benefit obtained from such a procedure is worth the added complexity.

VI.D Future Research

In addition to improving SAL's performance in the domain described in this dissertation, a clear direction of future research is to increase the size of this domain. In particular, including non-deterministic games such as backgammon would be interesting. Perhaps the most useful in terms of real applications would be to include games of imperfect information. Many complex decisions are made by humans without complete information about the state of the environment. A program capable of learning to make good decisions for games of imperfect information would have wide application.

Bibliography

[1] B. Abramson. *The Expected–outcome model of two–player games.* Pitman Publishing, London, 1991.

[2] L. V. Allis, M. van der Meulen, and H. J. van den Herik. $\alpha\beta$ conspiracy–number search. In D. F. Beal, editor, *Advances in Computer Chess 6*, pages 73–95. Ellis Horwood Limited, New York, 1991.

[3] I. Althöfer. An incremental negamax algorithm. *Artificial Intelligence*, 43:57–65, 1990.

[4] T. Anantharaman, M. S. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43:99–109, 1990.

[5] W. Aspray and A. Burks, editors. *Papers of John Von Neumann on Computing and Computer Theory.* The MIT Press, 1987.

[6] A. Barr and E. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume 1. William Kaufmann, Inc., Los Altos, 1981.

[7] E. B. Baum. Minimax is not optimal for imperfect game players. 1993.

[8] E. B. Baum and W. D. Smith. Best play for imperfect players and game tree search. April 1993.

[9] D. F. Beal. An analysis of minimax. In M. R. B. Clarke, editor, *Advances in computer chess 2*, pages 103–109. Edinburgh University Press, Edinburgh, 1980.

[10] D. F. Beal. Benefits of minimax search. In M. R. B. Clarke, editor, *Advances in computer chess 3*, pages 17–24. Pergamon Press, Oxford, 1982.

[11] D. F. Beal. Recent progress in understanding minimax search. In *Proceedings of the Association for Computing Machinery annual conference*, pages 164–169, New York, 1983. Association for Computing Machinery.

[12] D. F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[13] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language.* Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California, 1988.

[14] I. Bratko and M. Gams. Error analysis of the minimax principle. In M. R. B. Clarke, editor, *Advances in computer chess 3*, pages 1–15. Pergamon Press, Oxford, 1982.

[15] H. L. Dreyfus. *What Computers Can't Do: The Limits of Artificial Intelligence.* Harper and Row, New York, second edition, 1979.

[16] R. O. Duda and P. E. Hart. *Pattern classification and scene analysis.* John Wiley and Sons, 1973.

[17] S. Epstein. Prior knowledge strengthens learning to control search in weak theory domains. *International Journal of Intelligent Systems*, 7:547–586, 1992.

[18] E. A. Feigenbaum and J. Feldman, editors. *Computers and thought.* McGraw-Hill Book Company, Inc., 1963.

[19] J. J. Gillogly. The technology chess program. *Artificial Intelligence*, 3:145–163, 1972.

[20] A. K. Griffith. A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence*, 5:137–148, 1974.

[21] D. Hartmann. Notions of evaluation functions tested against grandmaster games. In D. F. Beal, editor, *Advances in Computer Chess 5*, pages 91–143. Elsevier Science Publishers B.V., Amsterdam, 1989.

[22] D. R. Hofstadter. *Metamagical themas: Questing for the essence of mind and pattern.* Basic Books, Inc., New York, 1985.

[23] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific American*, 263(4):44–50, October 1990.

[24] H. Kaindl. Minimaxing theory and practice. *AI Magazine*, pages 69–76, Fall 1988.

[25] H. Kaindl, R. Shams, and H. Horacek. Minimax search algorithms with and withould aspiration windows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(12):1225–1235, December 1991.

[26] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[27] K. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[28] K. Lee and S. Mahajan. The development of a world class Othello program. *Artificial Intelligence*, 43:21–36, 1990.

[29] R. Levinson. Experience-based creativity. 1991.

[30] R. Levinson and R. Snyder. Adaptive pattern-oriented chess. In *Proceedings of AAAI-91*, pages 601–606. Morgan-Kaufman, 1991.

[31] H. H. Martens. Two notes on machine 'learning'. *Information and Control*, 2:364–379, 1959.

[32] D. A. McAllester. Conspiracy numbers for min–max search. *Artificial Intelligence*, 35:287–310, 1988.

[33] J. McCarthy. Chess as the drosophila of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 197–216. Springer–Verlag, New York, 1990.

[34] D. Michie. King and rook against king: Historical background and a problem on the infinite board. In M. R. B. Clarke, editor, *Advances in Computer Chess 1*, pages 30–58. Edinburgh University Press, Edinburgh, 1977.

[35] M. Minsky. Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8–30, January 1961. also in Computers and Thought.

[36] D. S. Nau. Pathology on game trees: a summary of results. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 102–104, Stanford, CA, 1980. Stanford University.

[37] D. S. Nau. An investigation of the causes of pathology in games. *Artificial Intelligence*, 19:257–278, 1982.

[38] D. S. Nau. The last player theorem. *Artificial Intelligence*, 18:53–65, 1982.

[39] D. S. Nau. Decision quality as a function of search depth on game trees. *Journal of the Association for Computing Machinery*, 30(4):687–708, October 1983.

[40] D. S. Nau. On game graph structure and its influence on pathology. *International Journal of Computer and Information Sciences*, 12(6):367–383, 1983.

[41] D. S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, 21:221–224, 1983.

[42] J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, 1947.

[43] A. Newell. The chess machine: An example of dealing with a complex task by adaptation. In *1955 Western Joint Computer Conference*, pages 101–108, 1955.

[44] A. Newell, J. C. Shaw, and H. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–335, October 1958. also in Computers and Thought.

[45] T. Nitsche. A learning chess program. In M. R. B. Clarke, editor, *Advances in Computer Chess 3*, pages 113–120. Pergamon Press, Oxford, 1982.

[46] A. J. Palay. *Searching with probabilities*. PhD thesis, Carnegie-Mellon University, 1985.

[47] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[48] P. I. Richards. Machines which can learn. *American Scientist*, 39:711–716, 1951.

[49] P. I. Richards. On game-learning machines. *The Scientific Monthly*, pages 201–205, April 1952.

[50] R. L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34:77–96, 1988.

[51] D. E. Rumelhart, J. L. McClelland, and et.al. *Parallel distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. The MIT Press, Cambridge, Massachusetts, 1986.

[52] S. Russell and E. Wefald. *Do the right thing: Studies in limited rationality*. The MIT Press, Cambridge, Massachusetts, 1991.

[53] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, July 1959. also in Computers and Thought.

[54] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, November 1989.

[55] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43:67–84, 1990.

[56] T. Scherzer, L. Wcherzer, and D. Tjaden. Learning in bebe. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 197–216. Springer–Verlag, New York, 1990.

[57] G. Schrüfer. Presence and absence of pathology on game trees. In D. F. Beal, editor, *Advances in computer chess 4*, pages 101–112. Pergamon Press, Oxford, 1986.

[58] C. E. Shannon. A Chess-playing machine. *Scientific American*, February 1950.

[59] C. E. Shannon. Programming a computer for playing Chess. *Philosophical Magazine*, 41(7):256–275, March 1950.

[60] D. J. Slate and L. R. Atkin. CHESS 4.5 – The Northwestern University chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, chapter 4, pages 82–118. Springer-Verlag, 2 edition, 1983.

[61] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[62] G. Tesauro and T. J. Sejnowski. A parallel network that learns to play Backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[63] G. J. Tesauro. Practical issues in temporal difference learning. Technical Report RC 17223 (#76307), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, September 1991.

[64] K. Thompson. Retrograde analysis of certain endgames. *Journal of the International Computer Chess Association*, 9(3):131–139, 1986.

[65] A. M. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, October 1950. also in Computers and Thought.

[66] A. M. Turing. Digital computers applied to games. In B. V. Bowden, editor, *Faster than Thought: A Symposium on Digital Computing Machines*, pages 286–310. Sir Isaac Pitman and Sons, Ltd., London, 1953.

[67] L. Uhr. *Pattern recognition, learning, and thought: Computer–programmed models of higher mental processes*. Prentice-Hall, Inc., 1973.

[68] M. Weinberg. Mechanism in neurosis. *American Scientist*, 39:74–98, January 1951.

[69] N. Wiener. *Cybernetics: Or control and communication in the animal and the machine*. John Wiley and Sons, Inc., 1948.

[70] S. Yakowitz. A statistical foundation for machine learning, with application to Go-Moku. *Computers Math. Applic.*, 17(7):1095–1102, 1989.

[71] A. L. Zobrist and Jr. F. R. Carlson. An advice–taking chess computer. *Scientific American*, 228:92–105, June 1973.