STUDIES IN MACHINE LEARNING
USING GAME PLAYING

By

MICHAEL WAYNE SEALE

Bachelor of Science in

Electrical Engineering

University of Arkansas

Fayetteville, Arkansas

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1990

STUDIES IN MACHINE LEARNING

USING GAME PLAYING

Thesis Approved:

J P Chandler
Thesis Advisor

Huizhu Lu

Blayne E. Mayfield

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGMENTS

I extend special thanks to Dr. John Chandler, my principal advisor, for his advice, assistance, and confidence that this project would be completed. I also want to express my sincere thanks to the other members of my committee, Dr. Huizhu Lu and Dr. George Hedrick, the department head. Their suggestions and support were very important to this project. My thanks is also extended to Dr. K. M. George for his much needed assistance with the C++ Programming Language.

I am very grateful to the Air Force for giving me the opportunity to further my education and professional qualities. The understanding and support I have received from the Air Force Institute of Technology has been outstanding. Additionally, I extend my appreciation to Colonel John Barton, commander of the 1842 Electronics Engineering Group for his belief in my talents and his influence on me.

Finally, and most importantly, sincere gratitude is extended to my wife, Jeanie, and my children, Amy, Michael Jr., and Joey for their patience and encouragement through the difficult times. I could not have accomplished anything without their support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The field of machine learning is described by Carbonell and Langley [6] as a study of "computational methods for acquiring new knowledge, new skills, and new ways to organize existing knowledge." Games can provide a convenient vehicle for a study in machine learning if they meet certain conditions, such as having no practical algorithms for guaranteeing a win, and having clearly definable goals and rules of activity.

One of the first attempts to apply machine learning to game playing was A. L. Samuel ([17], [18]). He investigated several methods of machine learning using the game of checkers in the late 1950s and 1960s. The basic premise behind his first approach was to program a computer to improve its move selection by adjusting the parameters used to determine the relative value of a particular board position.

This project has attempted to generalize Samuel's parameter adjustment technique to two games, checkers and halma. Halma is a game with the same rules as Chinese checkers but is played on a square board similiar to a checkerboard [8]. A computer program was written to accomplish this task using an object oriented language and

was designed to be highly modular. The learning section of the program is shared between the two games.

CHAPTER II

BACKGROUND

Early Work on Computer Games

Some of the early work on computer game playing theory was done by John Von Neumann and Oscar Morgenstern [23]. In their monumental work "The Theory of Games and Economic Behavior" they presented the minimax algorithm and discussed its application to the game of chess. They theorized that if a player could look far enough ahead, he would be able to decide whether his present position is win, loss, or draw. Using this information, he could then always make the most informed move. However, they concluded that there is "no practically usable method to determine the best move. This . . . difficulty necessitates the use of those incomplete, heuristic methods of playing, which constitute good chess."

C. E. Shannon published a paper in 1950 describing a procedure for programming a computer to play chess [19]. Shannon argued that a chess player can look at a chess board and conclude whether the position is good or bad for one side or the other. The better chess player probably considers more factors and breaks each factor down into subcategories. Shannon suggested that a computer program score board positions in the same way using material, pawn

structure, and mobility as principal scoring factors. He went on to describe two strategies for implementing a look-ahead tree of moves. The type A-strategy searches to a fixed depth while the type B-strategy searches to a variable depth. In his opinion the type B-strategy could be further improved by using forward pruning to eliminate undesirable branches from the tree. The work done by Shannon was independently paralleled by A. M Turing [4]. Although many other variations on the Shannon/Turing method have been formulated, their basic technique is still used by most game playing programs today.

Deficiencies in Chess Playing Programs

D. Michie [13] identified the primary deficiencies of current chess playing programs and went on to discuss the outlook for improvement. The first defect is the horizon effect which renders a program oblivious to all events which may occur beyond its look-ahead search tree. The second defect is a lack of long-range ideas. A human Master chess player generally executes long-range plans that may include many intermediate goals along the way. Chess programs can flounder aimlessly with unrelated intermediate goals. The brute-force method of examining millions of possibilities in look-ahead analysis before selecting a move can stand up to human Grandmasters only in purely tactical play [13]. The human Grandmasters are superior in strategic, or positional

play because they have built up "associative stores of conceptualized chess knowledge." According to Michie, future prospects for computers to achieve Grandmaster status will necessitate the large-scale transfer of knowledge from humans or books to the computer. He describes advances that must be made under three areas to facilitate this process:

> (1) The design of data structures in forms suitable for representing conceptualized knowledge (descriptions, patterns, and theories) which are also convenient for the human user to modify and increment interactively.
>
> (2) Improved facilities for inductive inference, so that programs can acquire new knowledge both from illustrative examples supplied by human tutors, and also from the results of their own internal generation of examples for self-administration.
>
> (3) The engineering of conceptual interfaces between program and human expert, making it easier for the latter to 'teach' the machine.

Machine Learning

Michalski ([12], ch 1) postulates that the development of learning machines is important to the continued progress in artificial intelligence and related fields. The basic premise behind this point is that more and more knowledge must be imparted to AI systems. "Such knowledge must encompass domain-specific facts and rules, commonsense

heuristics and constraints, and general concepts and theories about the world." With this backdrop, a brief introduction to machine learning is presented.

Michalski ([12], ch 1]) classifies learning into several "learning strategies" that are briefly identified here. Rote learning is described as a process in which "the information from the teacher is more or less directly accepted and memorized by the learner." Learning by instruction places the burden of learning primarily on the teacher, with the learner responsible for "selection and reformulation." Deductive learning allows the learner to draw "deductive, truth preserving inferences from the knowledge given and store useful conclusions." Learning by induction is defined as follows: "If the transformation process involves generalization of input information and selection of the most plausible or desirable result, that is, the inductive inference, then we have inductive learning." Finally, learning by analogy is identified as using both deductive and inductive processes.

Forsyth ([7], ch 1]) discusses a "framework for learning" which includes the following components: The Critic, the Learner, the Rules, and finally the Performer. All of these components are necessary for learning to take place. The Critic can be described as the component that "compares the actual with the desired output." The desired output is also termed as an "ideal system." The Learner is "the heart of the system ... and has responsibility for

amending the knowledge base to correct erroneous performance." Forsyth defines the Rules as "the data structures that encode the system's current level of expertise ... and are used to guide the activity of the performance module." The last component, the Performer, "is the part of the system that carries out the task. The performer uses the rules in some way to guide its activity. Thus when the rules are updated, the performance of the system as a whole changes.

Samuel's Machine Learning Techniques

The work done by A. L. Samuel ([17], [18]) was one of the early pioneering successes in the field of machine learning. In his first article, he discussed his checker playing program that incorporated two separate learning procedures, rote learning and a generalized learning procedure. Rote learning involved saving all of the board positions encountered during play, together with their computed scores. References were then made to this memory record in order to save computing time and also to allow a farther look-ahead. Rote learning was found to be minimally successful for the opening game and to a lesser extent in the end game. However, rote learning was found to be somewhat ineffective in the middle game. Samuel's generalization learning procedure consisted of having the computer continually re-evaluate the coefficients of the

linear polynomial. The polynomial is used to evaluate the terminating board positions of the look-ahead tree search. The generalization procedure was found to be relatively effective in the middle game. In his second article Samuel discusses several improvements to the checker playing program and also used a different learning procedure. One improvement was the implementation of alpha-beta pruning of the look-ahead tree. This process allowed for much deeper and generally more effective look-ahead tree searches. The second improvement, called the signature-table technique, was implemented in order to overcome the inter-parameter effects and their interactions upon the linear polynomial. This method involved grouping related parameters together into subsets called signature types. From these subsets, for a particular board position, a value is calculated that serves as an address into a signature table where tabulated values are retrieved that reflect the relative worth for these particular combinations. These improvements were used with an improved book learning procedure. This technique involves presenting the program with a particular board position and allowing the program to select its best move. The move selected is then compared with the book-recommended move. The book-recommended move can be described as the move that is considered best by expert human checker players. The program then uses the difference between its move and the book-recommended move to adjust its evaluation procedure.

Of Samuel's different methods of machine learning, the current author finds the generalized method to be the most interesting. The rote learning method is basically a procedure for the storage and retrieval of information that is gradually accumulated as the program is faced with more and more game playing decisions. Rote learning was basically used to speed up the heuristic decision making process and to allow for deeper searches using the time saved [17]. The book learning method is interesting in that if given enough input data, that is book-recommended moves, the program can gradually improve its level of play [18]. The primary drawback to this method, in this author's opinion, is the fact that an outside source of information, or database, must be in existence and made available to the program in order for it to learn. Samuel's generalized learning method allows the computer to improve its level of play by playing against itself or a human opponent and continually adjusting its evaluation function based on perceived needed adjustments [17]. No outside instruction is needed. The method is advantageous because for many games, and real life situations for that matter, there simply is no large base of information from which to draw.

Samuel's learning by generalization technique as applied to the game of checkers is described as follows [17]. The score for a board position is found by computing the scoring polynomial. The terms of the polynomial consist of measurements of the board position such as center

control, threat of fork, etc. Each term is multiplied by a coefficient which assigns a weight to the particular parameter in relation to all the other terms. The sum of these terms gives the score for the board position. The coefficients of the terms of the scoring polynomial are modified during the machine learning process. When the checkers program is presented a board position from which to pick the best move, it first invokes the objective function and assigns a value to the initial board position. Then it creates the look-ahead tree of moves and searches for the best move based upon each side taking his best move at each turn. Eventually, the best look-ahead board position is found and the objective function is invoked to assign a value to it. The fundamental assumption for the learning process is that the score calculated for the initial board position should look like the score calculated for the terminating board position of the look-ahead search [17]. If there is a difference in scores, then the evaluation of the initial board position is assumed to be incorrect. The coefficients of the scoring polynomial are modified so that they cause the score for the initial board position to more closely resemble the score for the look-ahead board position.

The coefficients of the terms in the scoring polynomial are modified indirectly by a complicated process [17]. The difference between the initial board score and look-ahead board score is called delta. If delta is positive, then the

coefficients of the terms of the scoring polynomial have been given too much weight. Conversely, if delta is negative, then the coefficients have been given too little weight. The coefficient terms are only modified if delta is larger than some set value and this value is adjusted throughout the game. The goal during the coefficient modification procedure is to assign the optimum weights to the polynomial terms in relation to each other. Polynomial terms that are more important in determining if a board position is potentially good should tend to increase relative to lesser important terms, or terms that are disadvantageous (i.e., coefficients with a negative sign). Instead of adjusting the coefficients directly, correlations between the signs of the polynomial coefficients and the sign of delta are calculated and used to modify the coefficients of the scoring polynomial. The correlations take into consideration the number of times that each polynomial term has been used and has had a nonzero value. The coefficient term with the largest correlation value is then set at a prescribed maximum value with proportionate values determined for all of the remaining coefficients. The scoring polynomial retains 16 terms out of a possible 38 terms at any one time. Once a particular term has been given the minimum coefficient value over some set number of moves, that term is dropped out of the polynomial and the next term in the queue of waiting terms is reinserted to the polynomial. Polynomial terms were dropped out and then

reintroduced later, probably as a way of speeding up the program. In the current author's opinion, the time necessary to calculate 38 terms and/or combinations of terms on the machine that Samuel used would have been excessive. The speed and power of modern day machines might allow the polynomial to keep and adjust all of the terms during the game.

Discussion

Samuel demonstrated that his generalized method of learning does tend to improve the accuracy of the scoring polynomial and thereby improve the level of the program's move selection [17]. Intuitively speaking, why does this occur? The current author believes that the answer lies in the fact that the program is given a sense of direction that is kept separate from the board scoring polynomial. In the case of checkers, this sense of direction is the objective of gaining material. Material credit is given for jumping the opponent pieces (thereby removing them from the board) and reaching the opposite end of the board so that regular pieces can be promoted to kings. If an objective function uses material solely to determine its moves, then it fails to recognize tactical situations and arrangements of pieces that may be more important than a particular gain or loss in material. This sense of direction that the program is given should be dominant over the other board scoring parameters,

but not so dominant that it fails to allow the other parameters to affect the final move decision. In this way, the dominant sense of direction causes the coefficients of the scoring polynomial terms to be corrected in the right direction. If programmed correctly, the proper weighting of the coefficients may result as well.

The primary drawback to the scoring polynomial is its linear nature ([1], [17]). One method that Samuel used in order to overcome this problem was to divide the game into six phases and to use a different scoring polynomial for each phase. For example, in the opening game the measurement called advancement is an important parameter but in the end game shouldn't be a factor at all. Perhaps some method of machine learning similiar to the coefficient modification scheme can be found that will allow the program to decide which polynomial terms to use in the various phases of the game. Many variations on Samuel's generalization method might be practical and advantageous.

CHAPTER III

PROJECT OBJECTIVES

This project has been limited to two-person, zero-sum games with perfect information. A zero-sum game means that a gain in material or position by one side results in an identical loss to the other side. A game with perfect information means that players are informed at any move of the choices of all the previous moves in the play ([16], ch 2).

Two games have been selected for this project, checkers and halma. Halma, as mentioned in the introduction, is a game with the same rules as Chinese checkers [8]. Three halma board sizes were programmed: 6 by 6 squares, 8 by 8 squares, and 12 by 12 squares. Checkers and halma were selected because they have relatively simple rules of play but still contain all of the basic characteristics of an intellectual activity. The goal of this project was to write an Object Oriented Program (OOP) that improves its level of game playing when given the rules of the game, an inherent drive to win the game, and a set of parameters for evaluating play. The set of parameters may be incomplete and the individual parameters are not orthogonal.

Samuel's generalization learning procedure was applied to two games but his methods were modified in some important

ways. Samuel's program was written in assembly language on an IBM 704 (a slow machine by modern standards) that had a limited memory and used magnetic tape for secondary storage ([17], [18]). The program produced by this project was developed using a more modern machine and was written using the C++ high-level language. Checkers and halma were programmed with a secondary goal of designing a highly modular program. The rote learning technique used by Samuel was not implemented because of its limited value as an instrument of true machine learning. The signature-table technique was not considered in this project because of the amount of time that would be involved in order to replicate the signature subsets and tables and because a large set of book recommended moves was not available for the game of halma.

CHAPTER IV

MINIMAXING GAME TREE SEARCH TECHNIQUES

Computer programs of games typically search very large trees of hypothetical moves in order to determine the best move. For example, examine the game of chess. A board position, contains 64 squares and an indication of what piece occupies each square for each side. The nodes in the search tree represent board positions. The branches in the search tree represent the moves that would be taken from a certain board position. Chess has an average branching factor of 35 [11]. The branching factor is defined as the number of branches leaving a node. The difficulty with the brute force approach is that the game trees grow exponentially and the time to search every branch to a reasonable depth becomes excessive. To examine every move in an average chess tree to a depth of five (assuming the root is level 1) would require the evaluation of 1,071,875 board positions. Deep searches are desirable because they usually result in more informed move selection. However, deep searches alone will not guarantee that the best move will be selected. Nau [14] demonstrated that if the evaluation function is in error, "searching deeper does not increase the probability of making a correct decision."

The look-ahead tree search for a game is typically

described as a minimaxing process because at alternate levels of the search tree the moves that would be made by the opposing side must be considered ([20], ch 2). The assumption is that the active side (the side whose turn it is to move) will choose the best move and thereby seek to obtain the maximum score from a beginning board position. On the other hand, the passive side (the side whose turn it is not to move) would select his best move which would be the worst move for the active side. The passive side seeks to obtain the minimum score from a beginning board position. Thus at odd depths of the tree, the moves leading to maximum scores for board positions are sought, and at even depths of the tree, moves leading to minimum scores are sought. The active side is called MAX and the passive side is called MIN ([20], ch 2).

A depth first search is conducted such that the branches of immediate successors of the current node are evaluated from the left. The successors of each node are expanded until some criteria are used to end the search. The current author defines a leaf node as a board position in which the game has been won or lost by the active side. A search down a particular branch may end with a terminating board position when some arbitrary conditions have been met. These conditions might be defined as reaching a quiescent state at or below some minimum depth, also called horizon of the search. The definition of quiescence depends upon the game and the programmer. For example a chess program might

take a quiescent condition, or "dead state" [11], to be a board position in which neither side can capture an opponent's piece. Generally speaking, chess programs use more complex definitions of quiescence than this. The score for a terminal board position, or leaf as the case may be, is backed up the tree to the root node. A score is obtained for a board position by using an objective function to evaluate the relative worth of the board. For example a chess program's evaluation function might consist of the material balance (the difference in value of pieces held by each side) and the strategic balance (a composite measure of such things as mobility, square control, pawn formation structure, and king safety [11]). After the final branch of the root node has been examined, the score is backed up to the root. The branch from the root that led to the terminating or leaf node that produced the score is assumed to be the best move to take for the active side.

The Alpha-Beta Algorithm

Virtually all programs of complex games like chess incorporate some method for pruning, or eliminating useless branches from the look-ahead search tree. The method with the longest history [10] and the method still commonly used today is the Alpha-beta search algorithm. In order to illustrate the value of alpha-beta pruning, consider the game tree of Figure 1 which is created by a minimax search

procedure that does not use pruning [2]. Board positions for a look-ahead move by the first player are shown by squares, while board positions for the second player are shown by circles. To simplify the drawing, all nodes are assumed to have a branching factor of two. Nodes are created in the order that they are labeled (a-b-c-d ... and so on). Since e is a terminal board position, the evaluation function returns a value of -2 to node d. Then node f is created and a value of +3 is returned to node d. Since the value is being returned to a circle node the score is minimized. That is +3 is not less than -2, so the -2 score remains at node d. After all branches from node d have been explored, the score -2 is returned to node c. Next, node g is created and the minimum score from nodes h and i returned to node g is -5. The -5 is returned to node c, but since the score is maximized to node c, the -2 is greater and remains there. The -2 at node c is returned to node b and then nodes j through p are created and scores backed up in a similar fashion. The -2 at node b is backed up to node a and then the search continues down the right branch from node a. The final score at node a ends with a -2 and came from the branch that led to node b. This branch then represents the best move from board position a.

Alpha-beta pruning can be explained simply as a technique for not exploring those branches of a search tree which the active player would be wise enough not to choose, or that the passive player would not have chosen because it

would have been unproductive for him. The alpha value is a lower bound that the active player must exceed before deciding on the move as being better than the previously selected move. The beta value is an upper bound that the passive player must undercut before deciding on the move as being better than the previously selected move. A formalism for evaluating the alpha-beta algorithm called "negamax" was introduced by Knuth and Moore [10]. This approach eliminates the need to alternately maximize and minimize backed up scores. Instead, scores for terminating board positions are always considered from the active player's point of view. This view was also used by Campbell and Marsland [5]. Their recursive procedure, written using a "C/PASCAL-like language" for the alpha-beta algorithm, is reproduced in Figure 2. The functions that are called are not described but are assumed to exist and perform as indicated.

To illustrate the application of this algorithm, it is applied to the search tree of Figure 1. The game search tree of Figure 1 is altered by the alpha-beta algorithm and is presented in Figure 3. The branches with dashed lines can be left unexplored without influencing the final move choice. The final alpha and beta values are shown next to each nodes. Since the variable m is set to alpha prior to the loop in the above algorithm, we can assume that the changes to m are, in effect, changes to alpha. It is important to note that the initial values of alpha and beta

at node a are -infinity and +infinity. Nodes b, c, and d are created with alpha and beta values the carried down from node a. Node e is terminal and scored at -2. The -2 is returned and the sign is changed by the algorithm. Since +2 is greater than -infinity, the alpha value at node d is changed to +2. Nodes f results in no change to alpha at node d. The +2 at node d is returned to node c as -2. Since -2 is greater than -infinity, the new alpha value at node c is -2. Node g is created with alpha and beta values of -infinity and +2. Node h is created and the score of -4 is returned to node g as +4. The alpha value at node g is now 4, which is greater that the beta value of 2. This represents a cutoff, or rather a node that will not yield any better moves than those already discovered. Node i does not need to be created or evaluated because eliminating it has no effect on the final outcome of the search. The process continues with the final move selection at node a being identical to the selection using the regular minimaxing technique. The solid nodes in Figure 3 represent board positions that do not need to be evaluated. Note that thirteen fewer nodes have been created and that nine fewer terminating board positions have been evaluated. This represents a very significant decrease in time complexity.

Reducing the Search Further

One very important observation can be made from examining the tree of Figure 3. The order of evaluation of the moves may affect how many cutoffs are found by the algorithm. In other words, if the best move happens to occur down the leftmost branch from the root node, then more cutoffs may be found than if the best move does not occur down that branch. Samuel [18] tried several methods for increasing the probability that the better paths are explored first. The best method he found was to conduct a preliminary plausibility survey for any given board situation by looking ahead a fixed amount, and then rearranging the available moves into "their apparent order of goodness on the basis of this information and to specify this as the order to be followed in the subsequent analysis." The difficulty with this technique is to determine how deep to perform the preliminary search. If the search is not performed deep enough, then the new order of the available moves may not in fact result in a shorter search. On the other hand, too deep a search takes away time from the actual search to be performed subsequently. There is also a question as to whether or not this plausibility analysis should be applied at all levels during the main look-ahead or only the first few levels. Knuth and Moore [10] demonstrated that reordering successor positions of some nodes makes no difference in the number of nodes

evaluated in the search to follow. They concluded that as much as 50 percent of the time taken for reordering successor branches may in fact be wasted.

Other methods of reducing the tree search have been proposed that do incur a risk with them. They fall under the category of forward pruning algorithms. These algorithms eliminate branches from the look-ahead search tree in the hopes of not eliminating a branch that contains the best move. The interval enclosed by the alpha and beta bounds is referred to as the alpha-beta window [5]. In the normal alpha-beta algorithm, alpha is initialized to -infinity and beta is initialized to +infinity for the root. This guarantees that the score backed up to the root lies within the initial window. However, the narrower the initial window, the smaller the tree that is grown out of the root node and therefore the better an algorithm will perform. Of course the danger here is that the window will not include the best score. No attempts at forward pruning methods were attempted in this project.

CHAPTER V

PROGRAM DESCRIPTION

Game Representation Approach

The checker game was programmed using Samuel's techniques for game board representation and move generation [17]. The program produced by this project was written on a machine using 32-bit integers, the same number of usable squares on a checkerboard. A board position is represented by four unsigned integers. The first integer contains 1's in bit positions which correspond to squares which contain pieces for one side. The second integer contains 1's in bit positions which correspond to pieces for the same side which are kings. The other two integers are used in a similiar fashion to represent pieces for the other side. Possible moves are represented by five unsigned integers. One integer simply contains a 0 if no moves are possible, a 1 if the only moves available are not jumps (also called slides), and a 2 if jump moves are available. The other four integers are bit vectors that represent the pieces of the side about to move that can initiate moves in the four directions allowed in checkers. The four directions are right-forward, left-forward, right-backward, and left-backward (see Figure 4).

This method of board representation has several advantages. To begin with, possible moves for all pieces in a certain direction can be computed simultaneously [22]. For example, right forward slides are computed by first performing an operation to place 1's into bit positions for all squares that do not contain a piece from either side. Then, this integer is shifted to the left an appropriate number of bit positions to place the 1's where they would have started from for a right-forward slide. Finally, an AND operation is performed between this integer and the integer representing pieces for the side about to move. The resulting integer contains all pieces for the side about to move that can initiate a right-forward slide. A second advantage for this board representation is its minimum storage requirements. An array of 16-bit integers used to represent a checkerboard would require four times as many bytes. The machine used by Samuel had 36-bit integers, which was an advantage because by ignoring certain bit positions in the integer, all bits could be shifted by equal amounts [22]. Using 32-bit integers, additional masking operations and staggered bit-shifting techniques were required.

Board representation and move generation for the game of halma was handled quite differently. Halma was required to be played on boards of three different sizes: 6 by 6, 8 by 8, and 12 by 12. Some of the methods used to encode the halma game were taken from an existing program written by

David M. Smith in Fortran [21]. The technique of representing pieces by bits in an unsigned integer accomplished for checkers was not possible for halma because of the different sized boards required and the need for up to 144 bit positions. The halma board is represented using a one-dimensional array of size 145 (actual number of useable elements is 144). Thus only 32 elements are used for a 6 by 6 board, 64 elements are used for an 8 by 8 board, and all 144 elements are used for a 12 by 12 board. Possible moves are generated for one piece at a time for the side about to move and stored in another array. In halma, because all the squares on the board are used, slides and jumps are allowed in all eight directions (see Figure 5).

The most general choice for board representation would have been a two-dimensional array, sized large enough to represent any board needed. The checker game could be represented by an array using the first 8 by 8 elements and ensuring that every other element must remain empty. This board representation could have possibly allowed slides and jumps from both games to share the same code. However, programming the games this way might not result is much savings in code and would have resulted in a slower program because of an increased number of subprocedure calls. This would happen because of the differences in legal moves between the two games. For example, a checker piece can only jump opposing pieces in restricted directions. On the other hand, a halma piece can jump its own pieces and

opposing pieces in any direction.

Auxillary Files

Several auxiliary files are associated with each game and are discussed briefly below. There are four game initialization files: ch_game, ch_open, ha_game, and ha_open. The "**_game" files must be created prior to the start of a game and are used to set up the initial board positions and other parameters. The "**_open" files are used to name the execution profiles and select a first move for the checker game. There are two coefficient files: ch_coeff and ha_coeff. These files do not have to be present at the start of a game. However, they should not be deleted after their creation unless the learning process is started from a new initial condition. A file called "seed_sav" is created by the learning mechanism to assist with the initialization of the polynomial coefficients. A file called "ln_init" is used by the learning mechanism to initialize parameters for the polynomial modification procedure. If this file is not present, it is created and default values are assigned. There are four execution profiles that are created during the execution of any game or series of games. These files end with a number from 0 to 99. The prochgm.## and prohagm.## files record the starting board position and all moves that take place during execution of the program. The prochln.## and prohaln.##

files record data about the learning mechanism and the polynomial coefficient modifications that take place during execution of the program. Finally, there are two coefficient profiles created: prochcoeff and prohacoeff. These files are appended at the end of each game with the final alpha coefficients.

Program Modules

A principal advantage of the C++ programming language is the class. A class is a data type that leads to modular design and object-oriented programming ([3], ch 3). In regular C, a structure contains only the variable portion of a data structure. The functions that are to be used with the structure must be declared separately. In C++, a class contains the variables, or storage locations for the data structure, as well as the functions that manipulate the variables ([3], ch 3). Access to the variables and functions of a class can be given to or restricted from other classes as desired. The program produced by this project was organized into the following classes that will be discussed:

```
driver
ch_base
    checkers (derived from ch_base)
    ch_search (derived from ch_base)
ha_base
    halma (derived from ha_base)
    ha_search (derived from ha_base)
learn
```

Driver Class

The driver class drives the program and is the only class object created in the main function. Two types of game play are available. "Opp_play" is a game played between the computer and a human opponent. If "opp_play" is desired, the program is executed without passing any arguments to the main function. During execution, the main function prompts the human opponent for the type of game to play, checkers or halma. "AB_play," which stands for alpha-beta play, is a game played by the computer against itself. One side called alpha uses a dynamic set of scoring parameters to score a board position. The other side, beta, uses a static set of scoring parameters that do not change during the game. To invoke "AB_play", execution of the program is initiated with one argument, a string that must be either "ch" or "ha." During "AB_play," all input and output is between files. This allows "AB_play" to be executed as a background process in a UNIX environment. Once execution begins and the appropriate game has been selected, the main function calls the appropriate publicly accessed functions in the driver class to play the game. If a move selected wins a game, this is reported to the driver function and a winner is announced. The game classes, halma and checkers, are nested within the private section of the driver class. Figure 6 contains a function calls flow chart for the main function and driver class.

Checkers is driven using three functions: ch_start, ch_opp, and ch_AB. The ch_start function is invoked by either "ch_opp" or "ch_AB" in order to read the appropriate files and perform initializations prior to the start of the checker game. The ch_opp function prompts the human player for input moves and displays the moves and updated game boards. Both the ch_opp and ch_AB functions access the appropriate checker class functions and variables in order to drive the game. They also record every move during a game to a game execution profile.

Halma is driven using four functions: ha_start, ha_store, ha_opp, and ha_AB. The ha_start function is invoked by either "ha_opp" or "ha_AB" in order to read the appropriate files and perform initializations prior to the start of the halma game. Both the ha_opp and ha_AB functions access the appropriate halma class functions and variables in order to drive the game. They also call the ha_store function to record the move and resultant board position after every move.

Checker Game

The checker game is represented using three classes: ch_base, checkers, and ch_search. "Ch_base" is a base class from which the checkers and ch_search classes are derived. The ch_search class is nested within the private section of

the checkers class. The functions within each class will be discussed briefly below. A function call flow chart for the ch_search class is presented in Figure 7.

The ch_base class contains eleven functions: initialize, generate_mvs, post_mv, revers_bd, shift_rf, shift_lf, shift_rb, shift_lb, revers_rf, revers_lf, revers_rb, and revers_lb. The learn class is nested within the private section of the ch_base class. The initialize function is called from the driver class and is used to initialize variables in preparation for the start of the game. "Initialize" attempts to open a file containing the polynomial coefficients. If the file is not found, "initialize" creates the file and writes to it a number containing the quantity of coefficients needed for the polynomial. "Initialize" then calls the data_in function of the learn class in order to input the initial alpha and beta polynomial coefficients. The generate_mvs function generates all possible moves from the board position as found in the array of unsigned integers "bd" and places the moves into an array of unsigned integers called "gen." "Generate_mvs" calls the four reverse functions necessary to perform the shifting and masking of bits to generate moves for a board position. A jump move placed into "mv_data" is never more than a single jump. "Generate_mvs" determines if the jump move can be continued and if so, places a flag into an element of "mv_data" for the calling routine. "Generate_mvs" also determines if the continued jump is one

in which a branch exists. That is, a choice of jumps must be made. If the continued jump does face a branch, the right-forward jump is stored into two elements of "mv_data" for the calling routine. The post_mv function receives a move as found in "mv_data" and posts it to the board position as found in "bd." "Post_mv" calls the four shift functions necessary to perform the shifting and masking of bits to post a move to a board position. The revers_bd function reverses the board position so that all functions in the program can analyze a board position from the forward direction.

The checkers class contains six functions: oppon_mv, verify_mv, open_select, comptr_mv, close_gm, and print_bd. The oppon_mv function is called by the driver class to post a move selected by the human opponent to the board position as found in "bd." The move is passed to "oppon_mv" as an argument in the character string "mv_str." The verify_mv function is called by "oppon_mv" to verify as a legal move a single step of the move selected by the human opponent. The open_select function is called by the driver class if the first move selected by the computer is to be a varied move selection. A varied first move means that the first move is selected from one of the four following moves and and is selected the given percentages: 11-16 (60%), 9-14 (25%), 11-16 (10%), and 10-14 (5%). The comptr_mv function is called by the driver class in order for the computer to select and post a move. "Comptr_mv" calls the search

function from the ch_search class if there is a choice of moves to be made. A jump with no choices is posted to the board position directly. The actual move taken is passed back to the driver function in the argument "mv_str." The function close_gm is called by the driver class in order to terminate the game. It determines which side won the game or declares a draw. Then it calls the coeff_exch function in the learn class to update and save the polynomial coefficients. Finally, it closes out the learning execution profile. The print_bd function is called by the driver class to display the board position for the human opponent.

The ch_search class contains five functions: search, recur, expand_mvs, order_mvs, and score_bd. The search function is called by the checkers class in order to search for and select the best move for the computer. "Search" transfers the actual board position to its own board position array "bd." It records some data for the learning execution profile and sets up other variables prior to a call to the recur function. After the call to the recur function the move selected is passed back to the calling routine in the array "gm_mv_data." After the look-ahead tree search is performed, "search" calls the poly_mod function of the learn class to modify the alpha coefficients. The recur function is an encoding of the alpha-beta algorithm presented earlier. "Recur" builds a look-ahead search tree of moves in order to find the best move. "Recur" calls the three remaining functions in the

ch_search class which are discussed below. The expand_mvs function is called in order to transform the move masks as found in the array "gen" into separate and distinct moves. The moves are passed back to the calling routine in the unsigned char array "moves." The order_mvs function is called by the recur function to reorder the available moves from the root of the search tree. The reordering of the moves is designed to allow the alpha-beta algorithm to find the maximum number of cutoffs [10]. The score_bd function is called in order to assign a relative value to a board position as found in the array "bd." It uses the alpha or beta coefficients depending on the value of the short integer "turn." Score_bd returns the score for the board position relative to the side whose turn it is to move. This is the "nega-max" technique first formalized by Knuth and Moore [10].

Halma Game

The halma game is represented using three classes: ha_base, halma, and ha_search. Ha_base is a base class from which the halma and ha_search classes are derived. The ha_search class is nested within the private section of the halma class. The functions within each class will be discussed briefly below. A function call flow chart for the ha_search class is presented in Figure 8.

The ha_base class contains eight functions:

initialize, revers_bd, generate_mvs, next_square, winning_mv, jump, reorder_mvs, and recovr_mv. The actual game playing board is kept in an array of short integers called "gm_bd." The learn class is nested within the private section of the ha_base class. The initialize function is called from the driver class and is used to initialize variables in preparation for the start of the game. "Initialize" attempts to open a file containing the polynomial coefficients. If the file is not found, initialize creates the file and writes to it a number containing the quantity of coefficients needed for the polynomial. "Initialize" then calls the data_in function of the learn class in order to input the initial alpha and beta polynomial coefficients. The revers_bd function reverses the board position so that all functions in the program can analyze a board position from the forward direction. The generate_mvs function generates all possible moves from the board position passed as an argument in the array of short integers "bd." The best "wide" number of moves is found and returned in an array of short integers called "moves." Jump moves are given greater priority than slide moves. A short integer is passed to generate_mvs as an argument called "save_mv." "Save_mv" determines whether "generate_mvs" searches for the best move, or recovers all the steps of a previously selected move. "Generate_mvs" calls the five remaining functions in the ha_base class which are discussed below. The next_square function is called by "generate_mvs"

to find the next square that contains a piece for the side whose turn it is to move. A flag is returned if no more squares are found. The winning_mv function is called by "generate_mvs" in order to see if the best move found by "generate_mvs" will win the game. If so, a flag is returned to announce that the game is won. The jump function is called by "generate_mvs" and updates the necessary variables prior to testing a square for the start of a new jump move. The reorder_mvs function takes the currently generated move and tests it against the best moves found so far by "generate_mvs." If the new move is found to be better than the worst move saved so far, the new move is placed into the "moves" array and the elements of the array are reordered. Thus the move reordering is implicitly performed at all levels of play. The recovr_mv function is called by generate_mvs if the "save_mv" flag is set. When "recovr_mv" finds a match between the current move generated and the move that was selected, it saves all steps of the move into an array of short integers called "mv_steps."

The halma class contains five functions: oppon_mv, comptr_mv, print_bd, stall_check, and close_gm. The oppon_mv function is called by the driver class to post a move selected by the human opponent to the board position as found in "gm_bd." The move is passed to "oppon_mv" as an argument in the character string "mv_str." If the move is invalid or if the move wins the game, a flag is returned to the calling routine. The comptr_mv function is called by

the driver class in order for the computer to select and post a move. "Comptr_mv" calls the search function of the ha_search class in order to select the best move. The actual move taken is passed back to the driver in the argument "mv_str." The print_bd function is called by the driver class to display the board position for the human opponent. The function stall_check is called by "comptr_mv" to determine if the opposing side is attempting to force the game to a draw by not moving a piece out of a starting square. If this condition is detected, the opponent is declared the looser of the game. The function close_gm is called by the driver class in order to terminate the game. It determines which side won the game or declares a draw. Then it calls the coeff_exch function in the learn class to update and save the polynomial coefficients. Finally it closes out the learning execution profile.

The ha_search class contains three functions: search, recur, and score_bd. The search function is called by the halma class in order to search for and select the best move for the computer. The board position is passed as an argument in the short integer array "bd." "Search" records some data for the learning execution profile and sets up other variables prior to a call to the recur function. "Search" places the actual move selection into two short integers called "start_mv" and "stop_mv." After the look-ahead tree search is performed, "search" calls the poly_mod function of the learn class to modify the alpha

coefficients. The recur function is an encoding of the alpha-beta algorithm presented earlier. "Recur" builds a look-ahead search tree of moves in order to find the best move. The score_bd function is called by "search" and also by "recur" in order to assign a relative value to a board position passed as the argument "bd." "Score_bd" uses the alpha or beta coefficients depending on the value of the short integer "turn." "Score_bd" returns the score for the board position relative to the side whose turn it is to move. This is the "nega-max" technique first formalized by Knuth and Moore [10].

Learn Class

The learn class contains four functions: data_in, initialize, data_out, coeff_exch, and poly_mod. The data_in function reads in the polynomial coefficients from a file and writes some data to a learning execution profile. The calling routine passes file pointers to the proper files as arguments to "data_in." If "data_in" detects that the coefficient file contains no coefficients, it calls the initialize function. "Initialize" creates the alpha and beta coefficients and randomly assigns values to them. This assures that the learning process starts from an initial condition with no particular bias. The coeff_exch function is called at the close of each game in order update the coefficent values. The poly_mod function is called

throughout the game when alpha is to move. It determines if the alpha coefficients should be modified and if so, it makes the modifications. The processes within the coeff_exch and poly_mod functions will be discussed in greater detail later.

Game Module Difficulties

The checker program was found to have virtually no drive for a win in the latter stages of a game. It would flounder around aimlessly and usually play to a draw. During a game played against a human opponent, the problem was not so acute because a human opponent usually drives for a win forcing the program into tactical situations in which the look-ahead tree search is highly effective. The problem was solved by putting two different tilts to the board evaluation function, separate from the evaluation polynomial. The first tilt, given to regular pieces, is a strong drive to the king row. Before this tilt was added, a look-ahead tree search would often fail realize that just over the horizon a regular man piece could become a king. The second tilt was given to kings, which were given a strong drive toward opposing pieces. By causing kings to advance toward the enemy, still considering loss of material to be highly detrimental, the look-ahead tree search and board scoring strategies were drawn more fully into play. These two tilts resulted in the checker program playing more

decidedly towards a win in the end game.

The halma program, on occasion, would attempt to play a game to a draw. This difficulty was totally unexpected because the halma game inherently contains a very strong drive for a win. However, situations were found to occur when a single piece was not moved out of the starting squares while the opponent had advanced to the point of winning the game. If the active side forsaw that it would lose the game by moving this piece, it would refuse. In order to overcome this difficulty, a function was added to detect this condition and announce a forfeit by the guilty side.

CHAPTER VI

MACHINE LEARNING

Program Learning Techniques

To initiate the learning process, the game playing module creats the ch_coeff or ha_coeff file and records two values: the number of polynomial terms in the evaluation function and the maximum value that a polynomial coefficient can obtain. The learning mechanism retrieves these two parameters and uses them to initialize the polynomial coefficients to random values within the proper range. After each move by alpha, the polynomial modification procedure determines whether or not to modify the alpha coefficients. The alpha coefficients are modified throughout a game while the beta coefficients remain static. At the end of each game, the game playing module determines which side won the game and passes this information to the learning mechanism. If alpha won "wins_needed" number of consecutive games, the alpha coefficients are transferred to the beta coefficients. If alpha lost "losses_needed" number of games in a row, the largest positive or negative alpha coefficient is reassigned a value of 0. No action is taken when a game is declared a draw. The assumption is that the modifications to the alpha coefficients will eventually lead

to a point at which alpha plays a better or worse game than beta. This process allows the learning process to backtrack if an obvious wrong direction has been taken and find new dominant alpha terms in the evaluation polynomial. This method is similar to the method used by Samuel [17]. Perhaps the learning mechanism should be allowed to backtrack on its own without resetting any coefficient to zero. Because of time constraints, however, no attempts at such an approach were taken in this project.

Prior to actually modifying the alpha coefficients, the game module performs some preliminary processing. Whenever the computer selects a move for alpha, the initial board position is scored and the individual scoring parameters are saved. Then the look-ahead board position is scored and compared to the initial board position score, with the difference assigned to the variable "delta." Then the scoring parameters and delta are passed to the coefficient modification procedure.

The learning process takes over and uses the parameter values passed to it to compute the individual terms of the scoring polynomial. A record of whether the parameter measurement was present or not in the initial board position is saved for the previous 30 initial board positions. The oldest record is overwritten by the new values. Next, the average of the absolute values of the last five delta values is found. The size of the current delta is compared with the size of the average delta. If the current delta is

larger than the average delta divided by "delta_cut", a decision is made to modify the alpha coefficients. A decision not to modify the alpha coefficients simply means that the error detected in the evaluation function is not great enough, based on the recent past, to warrant any modification of the alpha coefficients.

The polynomial modification procedure is accomplished by first calculating correlation values for each term of the polynomial and then using the correlation values to modify the alpha coefficients. Actually, the correlation values are not true correlations, as they are allowed to range beyond positive and negative one. The following equation is applied to each correlation term where i is an index, corr is the correlation term, and poly is a polynomial term:

```
corr[i] = (poly[i] - avg_poly) * (delta - avg_delta)
```

The average delta is found over the five previous delta values, while the average polynomial term is the average of all the terms from the current board evaluation.

The final step of the modification procedure is to actually modify the polynomial coefficients. Preceding the actual modifications, cutoff values are calculated. Cutoff values are used to determine the amount by which to modify the coefficients and are based on the average size of the correlation terms. Two cutoffs are used: the first being the correlation average multiplied by a value called "cut1"

and the second being the correlation average multiplied by a value called "cut2." The coefficients are incremented or decremented by either "mod_amt1" or "mod_amt2" based on the sign of the correlation and the size of the correlation in reference to the cutoff values.

Discussion

Some departures from Samuel's techniques were taken [17]. Samuel modified the coefficients by finding the ratio of the largest correlation value to the other correlation values and fixing the coefficient terms at integral powers of two based on this ratio. The current author abandoned this approach for two reasons. First of all, the integer size in the machine used contained only 32 bits as opposed to 36 bits for the machine used by Samuel [18]. Thus, fixing the coefficients at powers of two would result in a smaller range of coefficient values. Secondly, this author discovered that setting the coefficients based upon the largest correlation value resulted in an extreme amount of fluxuation and instability of the coefficient terms. The method incorporated into the final version of the program allows the coefficient values to range between a predetermined range of values called plus and minus "max_coeff." Coefficient terms are incremented or decremented a small amount from their previous values based on the size and sign of their correlation terms.

Samuel's machine learning technique [17] apparently modified all coefficient values whether or not the measured parameter for that term was present in the initial board evaluation. The current author limited modifications to coefficient terms that actually contained the measured parameter in the board evaluation. This slows down the rate of change for those parameters that occur infrequently. However, the current author believes that if the parameter was not present when the board was initially scored, then it could not have led to an error in the evaluation function. Therefore, the coefficient of that term should not be modified.

The current author attempted to apply a weighting factor to the correlation terms which would give more weight to those polynomial terms that occur more frequently. The intention was to allow those parameters that occur more frequently to be changed more rapidly. However, this method caused those terms to migrate towards the maximum or minimum coefficient values while the infrequently occuring terms tended to change very little. Therefore, no weighting factor was applied to the correlation values.

Results

Test data using the method described above is presented in Appendix B. Several different methods of actually adjusting the coefficients were attempted, most with

less-than-desirable results as discussed above. The test data accumulated indicates that the coefficients never really did stabilize, but rather continued to vary. This was to be expected and was experienced by Samuel as well ([17], [2]). From the data presented, no claim is made that the program markedly improves its level of play. However, some level of improvement in move selection probably does occur. Just precisely how to quantify such a statement is unclear. Perhaps this points to an area of further research.

One possible way to test the effectiveness of the learning mechanism was briefly attempted and is described below. The underlying assumption was that the program plays a relatively good game when the polynomial coefficients are preset to zero and not changed during the game. When this is done, the move selection is based totally on the program's inherent drive to win. For the checker game, the drive to win consists of material gain and the board tilts discussed earlier. For the halma game, the drive to win consists of the weighting mechanism applied to the board position. The procedure was tryed on
the checker game only. The beta coefficients were set to zero and the alpha coefficients were set to the values obtained after five games (see Appendix C). The learning mechanism was disabled, and "AB_play" was invoked. The procedure was repeated for the alpha coefficients after 10, 15, and 20 games. Using alpha coefficients from games 5 and

20, beta was declared the winner. Using alpha coefficients from games 10 and 15, the games were declared drawn. These results are not totally unexpected because the initialization process assigns random values to the polynomial coefficients at the start of the learning process. This means that alpha should initially lose games, but after some period of time, alpha should begin to win games. The fact that alpha lost using the game 20 coefficients may indicate that the learning process regressed somewhat after game 15.

Most of the design goals of the program were achieved. The final version is highly modular, and to the degree tested, has demonstrated itself to be reliable. The learning section of the program is shared between the two games and is general enough that it could be applied to any similar type of game. An attempt was made to write the driver class such that it could drive either game. However, no practical method could be found using the C++ programming language to implement such a procedure. The alpha-beta tree search methods implemented by the program were highly effective, although difficult to perfect.

CHAPTER VII

SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

The purpose of this study was to attempt to generalize Samuel's parameter adjustment learning procedure to two games, checkers and halma. Checkers and halma were selected because they are relatively simple games, but still contain all of the processes necessary for learning to occur. The primary goal of the project was to write an Object Oriented Program (OOP) that improves its level of game playing when given only the rules of the game, a sense of direction, and a set of parameters for evaluating play. The C++ programming language was chosen because it is a high-level, object-oriented language, and facilitates modular programming.

The machine learning mechanism was extremely difficult to program effectively. The learn class was one of the smaller classes in the program, and yet it took by far the longest to write. In spite of these remarks, one can safely conclude that the resultant learning mechanism was extremely crude. Much more work should be done in order to perfect the implementation of learning attempted by this project. Samuel's description of the exact parameter adjustment procedure [17] was somewhat vague, and by at least one other account [2], the results were far from optimal.

Machine learning is an important and active area of research today. This author concludes by encouraging more research in the field. In particular, this author encourages anyone interested, to attempt to perfect the learning method described by Samuel [17] and in this article. The program written in this report is available free of charge to anyone, for research purposes (only).

BIBLIOGRAPHY

1. Akl, S. G., "Checkers-Playing Programs," Encyclopedia of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

2. Banerji, R. B., "Game Playing," Encyclopedia of of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

3. Berry, J. T., C++ Programming, Howard W. Sams & Company, Berkeley, Cal., 1988.

4. Bowden, B. V. (ed.), Faster than Thought, Chapter 25, Pitman, London, 1953.

5. Campbell, M. S. and Marsland, T. A., "A Comparison of Minimax Tree Search Algorithms," Artificial Intelligence 20 (1983) 347-367.

6. Carbonell, J. and Langley, P., "Machine Learning," Encyclopedia of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

7. Forsyth, R. and Rada, R., Machine Learning: Applications in Expert Systems and Information Retrieval, Ellis Horwood Limited, Chichester, United Kingdom, 1986.

8. Gardner, M., "Mathematical Games," Scientific American, 225 (Oct 1971) 104-107.

9. Griffith, A. K., "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers," Artificial Intelligence, 5 (1974) 137-148.

10. Knuth, D. E. and Moore, R. W., "An Analysis of Alpha-Beta Pruning," Artificial Intelligence 6 (1975) 293-326.

11. Marsland, T. A., "Computer Chess Methods," Encyclopedia of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

12. Michalski, R., Carbonell, J., and Mitchell, T., Machine Learning-Volume 2, Morgan Kaufmann, Los Altos, Cal., 1986.

13. Michie, D., On Machine Intelligence, Second Edition, Ellis Horwood Limited, Chichester, United Kingdom, 1986.

14. Nau, D. S., "Decision Quality as a Function of Search Depth on Game Trees," J. Assos. Comp. Mach., 30, 687 (1983).

15. Newborn, M., Computer Chess, Academic Press, New York, 1975.

16. Parthasarathy, T. and Raghavan, T.E.S., Some Topics in Two-Person Games, American Elsevier, New York, 1971.

17. Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers," IBM Journal, 3, 210 (July 1959).

18. Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers II - Recent Progress," IBM Journal, 11, 601 (November 1967).

19. Shannon, C. E., "Programming a Computer for playing Chess," Philosophical Magazine 41, 256 (March 1950).

20. Slagle, J. R, Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill, New York, 1971.

21. Smith, D., "Unpublished Term Paper (COMSC 4343)," Oklahoma State University, 1973.

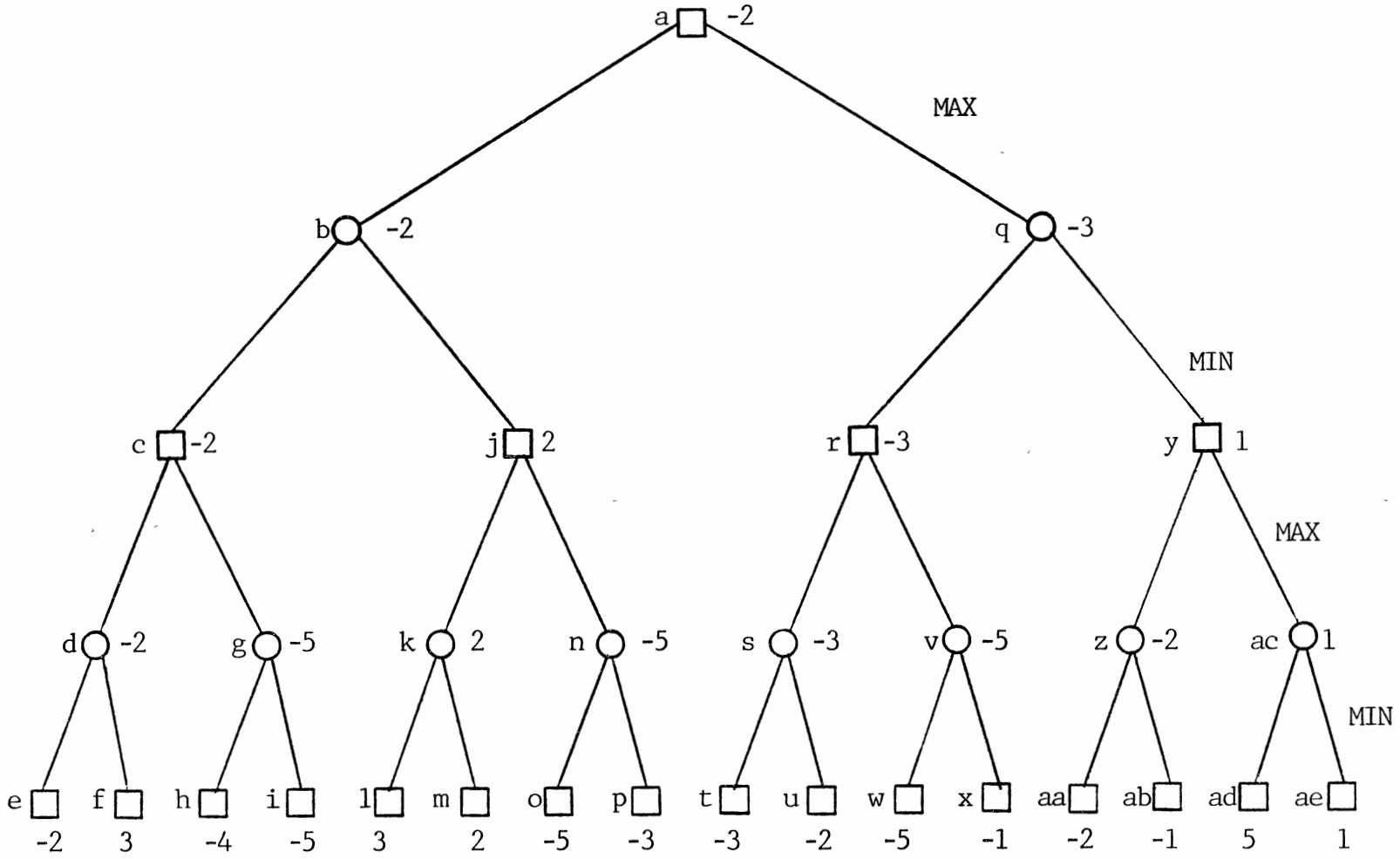
22. Struble, G., Assembler Language Programming-The IBM System/370 Family, Addison-Wesley, Reading, Mass., 1984.

23. Von Neumann, J., and Morgenstern, O., Theory of Games and Economic Behavior, Princeton Univ. Press, Princeton, New Jersey, 1944.

APPENDIXES

APPENDIX A

FIGURES



Reference: Banerji, R. B., "Game Playing," Encyclopedia of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

Figure 1. Minimax Move Tree

NOTATION

> greater than
>= greater than or equal to
< less than

FUNCTIONS

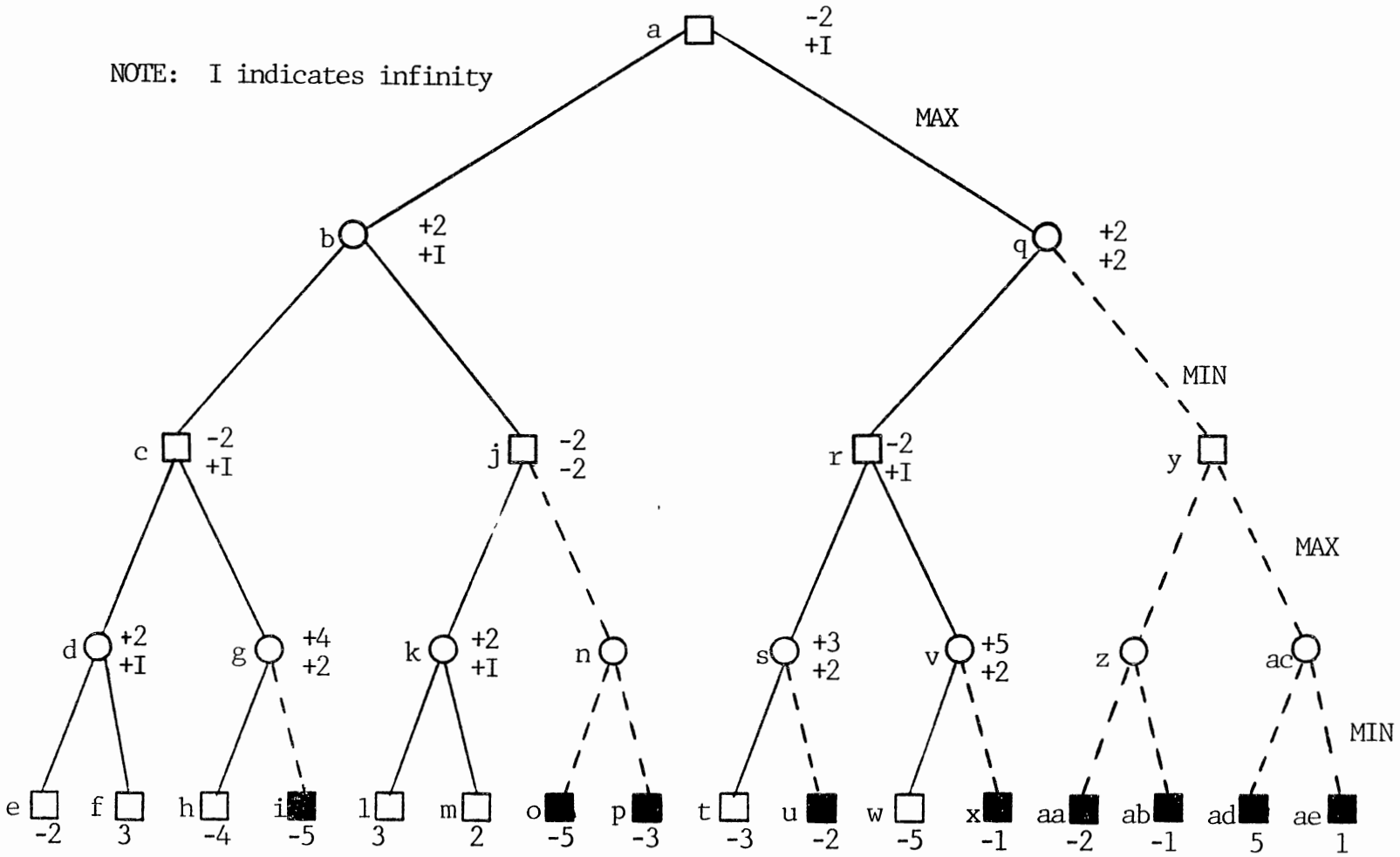
(1) terminal - determine if node p is terminal
(2) staticvalue - evaluate board position and assign a value
(3) generate - determine successor board positions p(1)...p(w)

C/PASCAL-LIKE PSEUDOCODE

```
 1.  alphabeta(p: position; alpha, beta: integer)
 2.  {
 3.    m, i, t, w: integer;
 4.    if(terminal(p))
 5.       return(staticvalue(p));
 6.    w = generate(p);
 7.    m = alpha;
 8.    for i = 1 to w do
 9.    {
10.      t = -alphabeta(p(i), -beta, -m);
11.      if(t > m)
12.        m = t;
13.      if(m >= beta)
14.        return(m);
15.    }
16.    return(m);
17.  }
```

REFERENCE: Campbell, M. S. and Marsland, T. A., "A Comparison of Minimax Tree Search Algorithms," Artificial Intelligence 20 (1983) 347-367.

Figure 2. Alpha-Beta Algorithm



Reference: Banerji, R. B., "Game Playing," Encyclopedia of Artificial Intelligence-Volume 1, Wiley & Sons, New York, 1987.

Figure 3. Alpha-Beta Move Tree

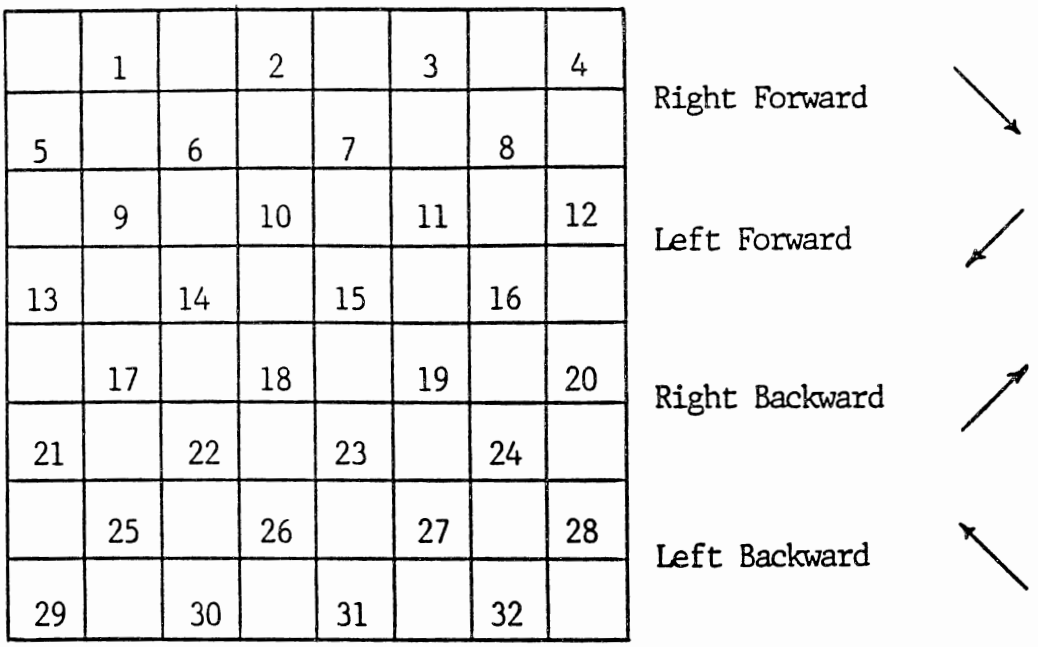


Figure 4. Labeled Checkerboard and Directions of Movement

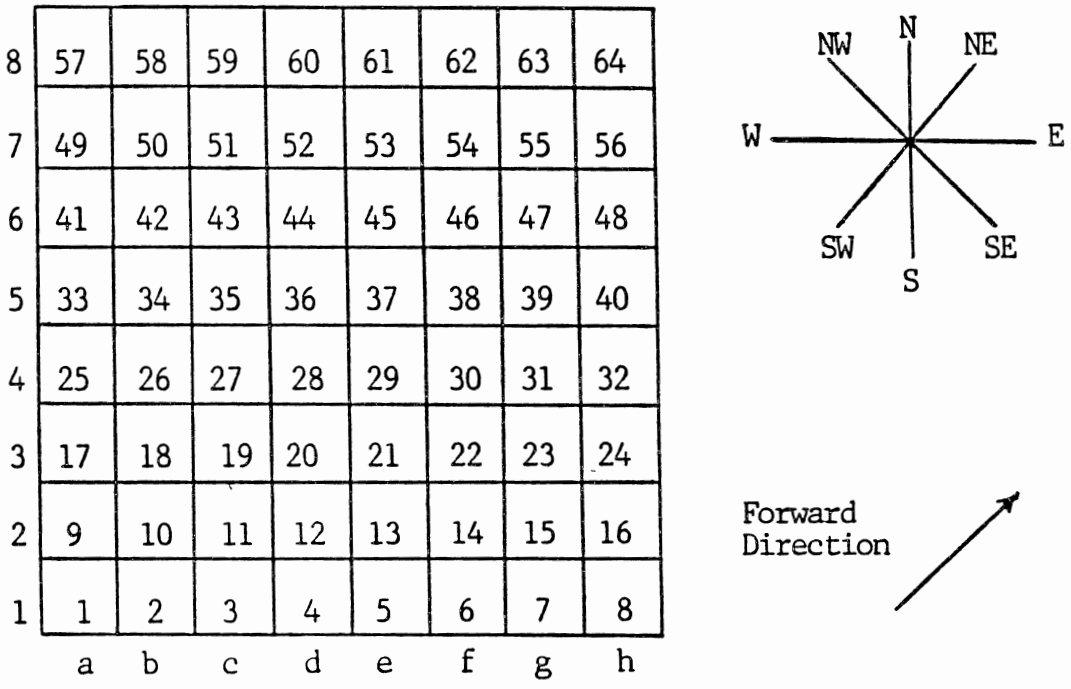


Figure 5. Labeled Halmaboard and Directions of Movement

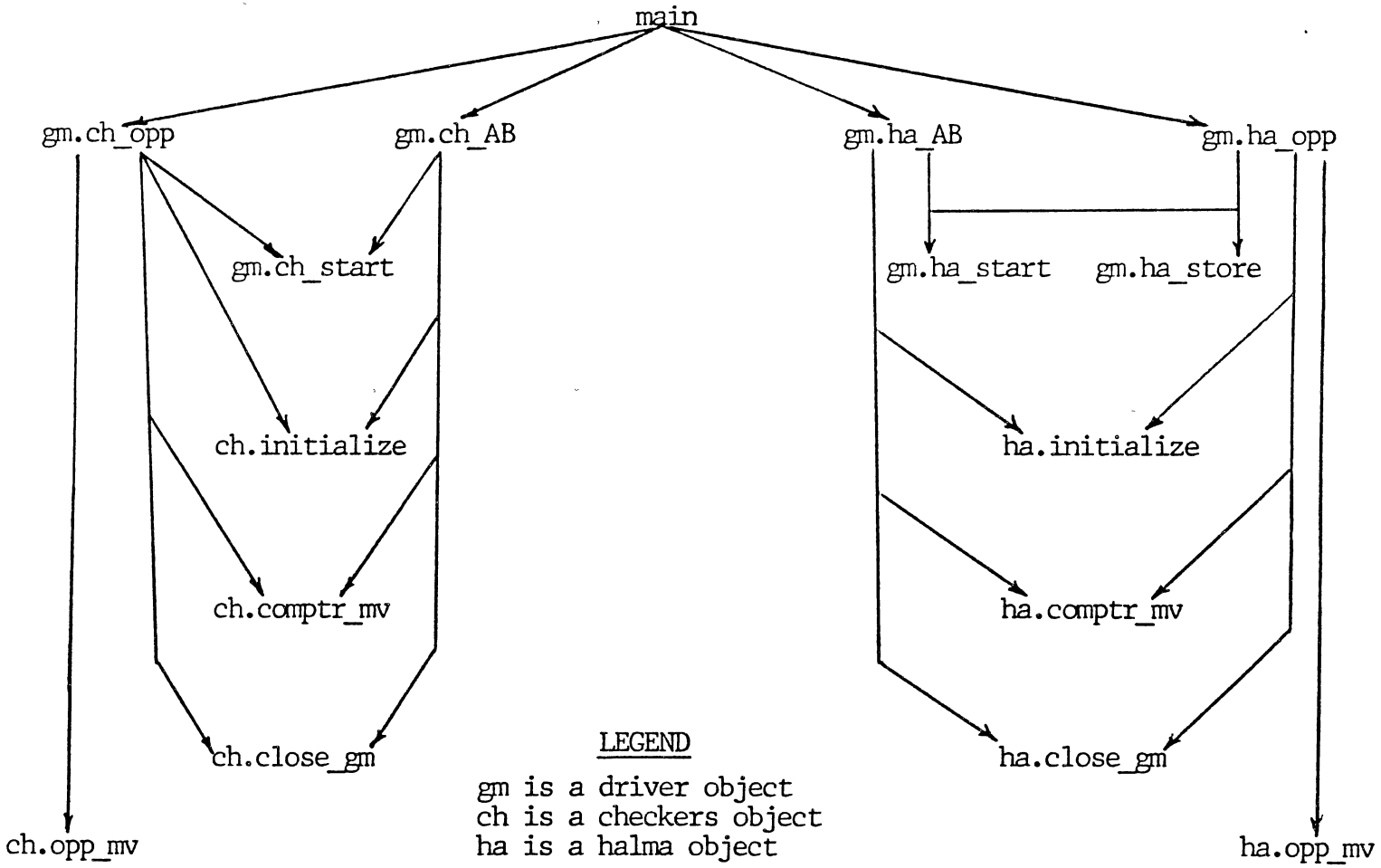
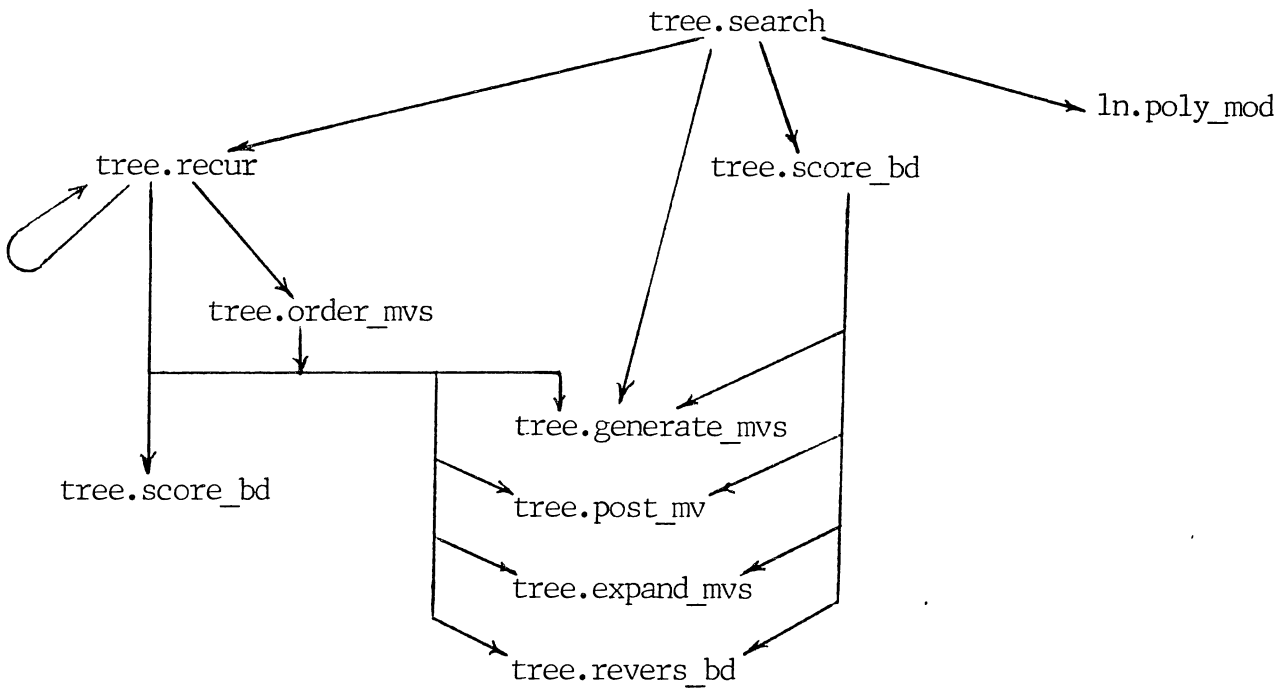


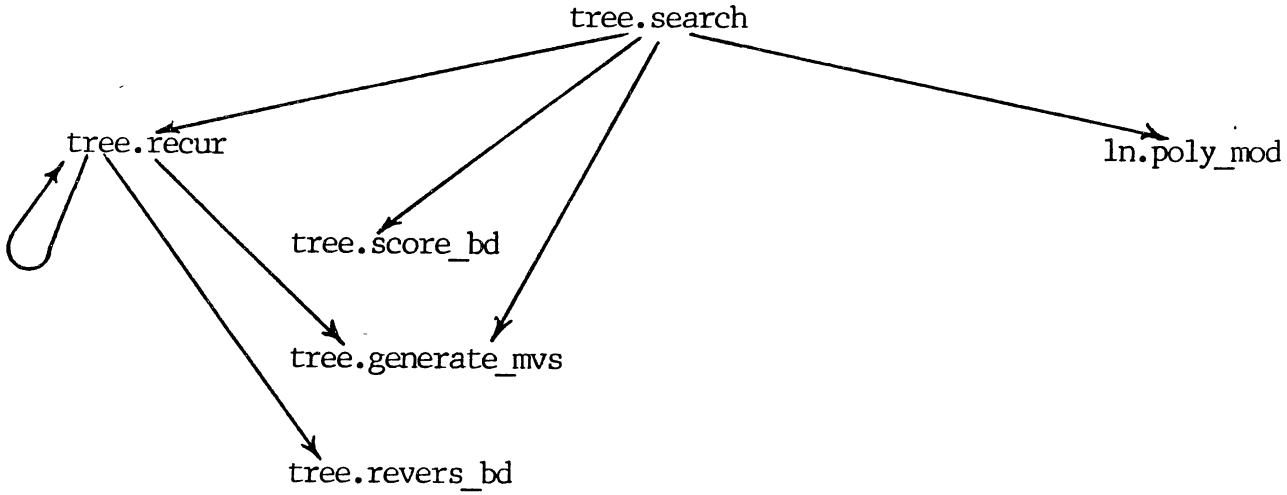
Figure 6. Function Call Flow Chart for Main Function and Driver Class



LEGEND

tree is a ch_search object
ln is a learn object

Figure 7. Function Call Flow Chart for Ch_Search Class



LEGEND

tree is a ha_search object
ln is a learn object

Figure 8. Function Call Flow Chart
for Ha_Search Class

APPENDIX B

EVALUATION FUNCTION PARAMETERS

Checker Game Parameters

The definitions for the checkergame parameters are taken from Samuel ([17], Appendix C).

1. Advancement: The parameter is credited with 1 for each passive man in the 5th and 6th rows (counting in passive's direction). In the current author's program, advancement is turned off after the first "early_game" moves.

2. Apex: The parameter is debited with 1 if there are no kings on the board, if either square 7 or 26 is occupied by an active man, and if neither of these squares is occupied by a passive man.

3. Back Row Bridge: The parameter is credited with 1 if there are no active kings on the board and if the two bridge squares (1 and 3, or 30 and 32) in the back row are occupied by passive pieces.

4. Center Control I: The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24, and 25 which is occupied by a passive man.

5. Center Control II: The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24, and 25 that is either currently occupied by an active piece or to which an active piece can move.

6. Double-Corner Credit: The parameter is credited with 1 if the material credit value for the active side is 3 (1 for men and 2 for kings) or less, if the passive side is ahead in material credit, and if the active side can move into one of the double-corner squares.

7. Cramp: The parameter is credited with 2 if the passive side occupies the cramping square (13 for Black, and 20 for White) and at least one other nearby square (9 or 14 for Black, and 19 or 24 for White), while certain squares (17, 21, 22, and 25 for Black, and 6, 11, 12 and 16 for White) are all occupied by the active side.

8. Diagonal Moment Value: The parameter is credited with "diag_mom_1" for each passive piece located on squares 2 removed from the double corner diagonal files, with 1 for each passive piece located on squares 1 removed from the double-corner files and with "diag_mom_2" for each passive piece in the double-corner files.

9. Dyke: The parameter is credited with 1 for each string of passive pieces that occupy three adjacent diagonal squares.

10. Exposure: The parameter is credited with 1 for each passive piece that is flanked along one or the other diagonal by two empty squares.

11. Pole: The parameter is credited with 1 for each passive man that is completely surrounded by empty squares.

12. King Center Control: The parameter is credited with 1 for each of the following squares: 11, 12, 15, 16, 20, 21, 24, and 25 which is occupied by a passive king.

13. Back Row Control: The parameter is credited with 1 if there are no active kings and if either the Bridge or the Triangle of Oreo is occupied by passive pieces.

14. Triangle of Oreo: The parameter is credited with 1 if there are no passive kings and if the Triangle of Oreo (squares 2, 3, and 7 for Black, and squares 26, 30 and 31 for White) is occupied by passive pieces.

15. Node: The parameter is credited with 1 for each passive piece that is surrounded by at least three empty squares.

16. Gap: The parameter is credited with 1 for each single empty square that separates two passive pieces along a diagonal, or that separates a passive piece from the edge of the board.

17. Hole: The parameter is credited with 1 for each empty square that is surrounded by three or more passive pieces.

18. Threat: The parameter is credited with 1 for each square to which an active piece may be moved and in so doing threaten the capture of a passive piece on a subsequent move.

19. Double Diagonal File: The parameter is credited with 1 for each passive piece located in the diagonal files terminating in the double-corner squares.

20. Total Mobility: The parameter is credited with 1 for each square to which the active side could move one or more pieces in the normal fashion, disregarding the fact that jump moves may or may not be available.

21. Deny: The parameter is credited with 1 for each square defined in Total Mobility if on the next move a piece

occupying this square could be captured without an exchange.

22. Undenied Mobility: The parameter is credited with the difference between Total Mobility and Deny.

23. Exchange: The parameter is credited with 1 for each square to which the active side may advance a piece and, in so doing, force an exchange.

24. Move: The parameter is credited with 1 if pieces are even with a total piece count (1 for men and 2 for kings) of less than 12, and if an odd number of pieces are in the move system, defined as those vertical files starting with squares 1, 2, 3, and 4.

25. Threat of Fork: The parameter is credited with 1 for each situation in which passive pieces occupy two adjacent squares in one row and in which there are three empty squares so disposed that the active side could, by occupying one of them threaten a sure capture of one or the other of the two pieces.

Halma Game Parameters

1. Stragglers: Pieces from the active side that have lagged behind the rest of the pieces of the active side considering movement in the forward direction. The greatest number of rows or columns of separation between the straggler and the nearest active piece is added for each straggler found.

2. Diagonal Pairs: The parameter is credited 1 for each pair of active pieces that are adjacent to each other and lined up in the forward direction.

3. Column Pairs: The parameter is credited 1 for each pair of active pieces that are adjacent to each other and lined up vertically.

4. Row Pairs: The parameter is credited 1 for each pair of active pieces that are adjacent to each other and lined up horizontally.

5. Difference: The parameter is credited 1 for the difference (if greater than one) between the number of active pieces above and below the main diagonal in the forward direction.

6. Southwest Jumps: The parameter is credited 1 for each active piece that could on the next turn be jumped from the Southwest direction.

7. South Jumps: The parameter is credited 1 for each active piece that could on the next turn be jumped from the South direction.

8. West Jumps: The parameter is credited 1 for each active piece that could on the next turn be jumped from the West direction.

9. Southeast Jumps: The parameter is credited 1 for each active piece that could on the next turn be jumped from the Southeast direction.

10. Northwest Jumps: The parameter is credited 1 for each active piece that could on the next turn be jumped from the Northwest direction.

11. Southwest Blockage: The parameter is credited 1 for each passive piece that is blocked by active pieces from initiating a move in the Southwest direction.

12. South Blockage: The parameter is credited 1 for each passive piece that is blocked by active pieces from initiating a move in the South direction.

13. West Blockage: The parameter is credited 1 for each passive piece that is blocked by active pieces from initiating a move in the West direction.

14. Northeast Blockage: The parameter is credited 1 for each passive piece that is blocked by active pieces from initiating a move in the Northeast direction.

15. Southeast Blockage: The parameter is credited 1 for each passive piece that is blocked by active pieces from initiating a move in the Southeast direction.

APPENDIX C

TEST DATA

Checkers Coefficients

Initial Coefficients:

-17	-34	15	25	11
11	3	-12	14	-23
-5	0	0	37	2
7	28	-20	-3	-18
-25	28	-27	-12	-15

Games 1 through 20:

1.	-16	-32	16	24	10
	11	2	-16	16	-22
	-7	0	1	37	-2
	5	26	-19	-6	-14
	-19	32	-28	-14	-13
2.	-15	-29	16	19	10
	11	5	-18	20	-21
	-1	3	1	36	2
	8	29	-15	-2	-24
	-17	26	-27	-16	-11
3.	-14	-27	18	18	13
	11	5	-25	21	-18
	5	2	3	36	3
	10	31	-12	3	-35
	-18	23	-28	-16	-11
4.	-14	-26	19	10	11
	8	6	-28	28	-15
	1	2	6	39	3
	11	31	-11	0	-24
	-20	25	-28	-17	-13
5.	-13	-24	22	16	14
	8	6	-39	29	-9
	7	2	9	40	6
	14	34	-9	6	-34
	-21	21	-30	-17	-13
6.	-15	-23	23	10	12
	5	7	-36	34	-13
	1	2	10	39	4
	18	34	-8	1	-24
	-23	24	-30	-18	-14
7.	-14	-23	30	17	16

	2	7	-34	0	-10
	4	2	17	40	6
	23	37	-5	5	-33
	-18	20	-29	-21	-20
8.	-15	-24	33	18	17
	2	8	-38	3	-13
	1	2	20	39	6
	27	38	-7	4	-29
	-16	19	-29	-21	-17
9.	-16	-20	36	15	22
	2	9	-40	11	-7
	7	4	23	39	10
	31	39	-3	12	-40
	-14	12	-27	-21	-22
10.	-18	-22	37	20	17
	2	10	-30	10	-12
	-5	4	24	39	1
	39	40	-5	2	-21
	-13	21	-29	-23	-17
11.	-17	-22	38	18	19
	2	9	-29	15	-10
	-4	4	30	0	0
	36	34	-3	3	-22
	-12	25	-27	-23	-26
12.	-16	-22	40	17	23
	1	10	-36	20	-10
	-4	4	35	5	0
	40	34	-5	5	-22
	-11	22	-27	-23	-29
13.	-15	-22	0	24	32
	0	10	-37	26	-3
	3	4	39	8	4
	39	38	-2	12	-36
	-4	14	-24	-24	-38
14.	-17	-27	2	27	27
	0	11	-34	24	-7
	-1	4	39	10	-2
	39	38	-5	7	-35
	-4	19	-26	-22	-32
15.	-19	-27	3	20	29
	-1	12	-33	26	-7
	-3	4	39	11	-4
	40	39	-6	6	-33
	-4	15	-25	-22	-32

16.	-21	-27	4	15	31
	-2	13	-35	29	-7
	-5	4	39	12	-6
	0	39	-7	5	-31
	-4	11	-24	-22	-32
17.	-23	-27	5	12	26
	-7	14	-32	32	-8
	-8	3	39	13	-11
	-1	40	-6	1	-26
	-2	10	-20	-22	-29
18.	-25	-26	8	13	25
	-10	15	-37	34	-8
	-8	2	39	16	-11
	-1	0	-5	2	-26
	1	13	-17	-22	-31
19.	-25	-26	12	16	23
	-10	18	-40	36	-2
	-2	2	40	21	-7
	6	6	-2	9	-38
	5	13	-10	-22	-39
20.	-27	-29	13	13	26
	-15	19	-31	36	-10
	-10	2	39	22	-12
	6	7	-2	2	-25
	6	18	-8	-21	-31

Halma Coefficients

New Coefficients:

-17	-34	15	25	11
11	3	-12	14	-23
-5	0	0	37	2

Games 1 through 30:

1.	-17	-32	12	27	10
	11	4	-13	16	-26
	-5	2	1	37	2
2.	-15	-29	9	30	11
	11	3	-14	16	-28
	-5	3	2	37	2
3.	-18	-23	5	36	9
	9	2	-14	16	-33
	-6	4	2	37	2
4.	-20	-19	2	38	6

	7	0	-13	15	-38
	-7	5	2	37	2
5.	-16	-16	0	36	4
	4	-2	-11	14	-36
	-9	2	3	37	2
6.	-17	-16	-5	37	2
	2	-5	-10	9	-37
	-12	0	7	37	2
7.	-16	-20	-9	37	0
	0	-9	-9	6	-36
	-18	-2	10	37	2
8.	-15	-15	-12	31	-2
	-2	-7	-7	2	-40
	-17	-1	10	37	2
9.	-14	-15	-9	30	-1
	-3	-5	-11	5	-40
	-20	1	9	37	2
10.	-12	-19	-13	35	1
	-4	-4	-14	8	0
	-26	0	7	37	0
11.	-18	-18	-16	40	-2
	-4	-5	-16	10	1
	-28	2	6	37	-1
12.	-23	-15	-16	0	-3
	-4	-6	-18	12	2
	-29	4	5	37	-2
13.	-23	-23	-22	3	-2
	-4	-9	-18	14	0
	-23	2	6	37	-2
14.	-21	-20	-20	0	-5
	-5	-13	-19	16	-4
	-19	-2	6	0	-2
15.	-21	-18	-19	-2	-9
	-6	-17	-19	19	-7
	-16	-6	8	0	-2
16.	-17	-12	0	1	-4
	-8	-16	-24	18	-9
	-19	-3	7	0	-2
17.	-17	-16	3	3	-1
	-10	-17	-25	18	-11

	-17	-1	6	0	-2
18.	-17	-18	5	2	-2
	-13	-18	0	17	-13
	-14	-2	7	0	-2
19.	-19	-15	3	-1	-1
	-13	-18	2	16	-14
	-15	-1	7	0	-2
20.	-20	-9	-1	-5	0
	-13	-18	3	16	-16
	-13	-1	8	0	-2
21.	-23	-3	-1	-3	2
	-14	-20	4	14	-19
	-11	-1	9	0	-2
22.	-15	-8	-6	-7	3
	-13	-19	2	15	-17
	-9	-1	6	0	-2
23.	-11	-5	-4	-10	1
	-15	-19	-2	18	-16
	-9	-2	2	0	-2
24.	-5	-4	0	-9	0
	-19	-21	-3	20	-14
	-8	-2	1	0	-2
25.	-2	-3	-2	-1	1
	-20	-22	-1	15	-11
	-14	0	2	0	-2
26.	-1	-4	-5	-5	1
	-18	-22	0	17	-12
	-11	2	2	0	-2
27.	-5	-4	-3	-3	-3
	-18	-22	-1	18	-12
	-10	1	1	0	-2
28.	-4	-13	-2	-2	-4
	-17	-23	-1	13	-12
	-8	6	0	0	-2
29.	-3	-12	2	-6	-2
	-17	-26	1	16	-13
	-12	9	-1	0	-2
30.	0	-12	4	-2	3
	-18	-25	3	10	-16
	-16	9	0	0	-2

APPENDIX D

PARTIAL PROGRAM LISTING

```
//              -----  File:  LEARN.C  -----
#include <stream.h>
#include <string.h>
#define delta_terms 5
#define items_per_line 5
#define past_counts 30
#define max_num_pars 25
#define arbitr_neg 12000
#define delta_cut 2
#define mod_amt1 1
#define mod_amt2 2
#define cut1 .2
#define cut2 2
/*  CONSTANT DEFINITIONS:
      delta_terms - number of past delta terms saved
      items_per_line - number of items to save to file
                       before a newline
      past_counts - number of past sets of parameter counts
                    to save
      max_num_pars - maximum number of parameters
      arbitr_neg - value returned from rand used to make a
                   number negative
      delta_cut - divisor of delta average to establish a
                  cutoff for the decision to modify or not
                  modify the alpha coeffs
      mod_amt1 - increment/decrement alpha coefficient
      mod_amt2 - increment/decrement alpha coefficient
      cut1 - cutoff multiplier for modifying alpha coeffs
      cut2 - cutoff multiplier for modifying alpha coeffs
*/

/*************************************************************
*************************************************************
Class:                    L E A R N

The learn class implements the machine learning mechanism.
It is used by both the checker and halma games.  It consists
of five functions: data_in, data_out, initialize, poly_mod,
and coeff_exch.  The alpha coefficients are contained in the
integer array a[].  The beta coefficients are contained in
the integer array b[].  The five last delta terms are saved
in the integer array d[].  Counts of the occurances of board
evaluation parameters are saved in the integer array c[][].
*************************************************************
*************************************************************/
class learn
{
    static int num_pars;  //size of array of coeff terms
    static int c[past_counts][max_num_pars];  //past
                                  // occurances of parameter
    static int d[delta_terms];  //delta values
    static int c_indx;  //index for next c term to overwrite
    static int d_indx;  //indx for next d term to overwrite
```

```
    static int wins;  //consecutive wins in recent games
    static int losses;  //consecutive losses
    static int wins_needed;  //wins needed to assign new
                             //  beta coeffs
    static int losses_needed;  //losses needed to assign
                               //  an alpha coeff to 0
    static int max_coeff;  //largest size of a poly coeff
    void initialize();  //function to initiate learn process
  public:
    int a[max_num_pars];  //alpha polynomial coefficients
    int b[max_num_pars];  //beta polynomial coefficients
    FILE *poly;  //file pointer for the coefficient file
    FILE *pro_ln;  //file pointer for the learn profile
    FILE *pro_coeff;  //file pointer for the coeff profile
    int ln_switch;  //switch to turn learning on or off
    int ratio;  //ratio of drive to win to polynomial
    void poly_mod(int, short*);  //coeff modification
    void coeff_exch(short);  //overwrite beta coeffs,
                             //  or reset alpha
    void data_in();  //read data in
    void data_out();  //store data
};

/*************************************************************
Function:       L E A R N :: P O L Y _ M O D

The poly_mod function performs the modification of the alpha
coefficients.  It uses the parameters passed to it to
calculate the individual polynomial terms.  From each term,
using delta passed to it, the correlation between each
polynomial term and delta is calculated.  These correlations
are used to actually perform the coefficient modification
and are stored in the integer array corr[].
*************************************************************/
   void learn::poly_mod(int delta, short par[])
   {
      short i, j = 0;  //indexing variables
      int sum = 0;  //temporary for calculating averages
      int avg;  //temporary for calculating averages
      short count = 1;  //count of c[][] items
      int abs_delta;  //for decision to modify
      int p[max_num_pars];  //polynomial terms
      int corr[max_num_pars];  //correlations
      int p_avg;  //polynomial average
      int d_avg;  //delta average
      int coeff_cut1, coeff_cut2;  //modify coefficients
      short mod_flag = 0;  //flag to output to profile
      int abs(int);  //function declaration

      for(i = 0; i < num_pars; ++i)
      {  p[i] = abs(par[i] * a[i]);  //calculate poly term
         if(p[i] > 0)  //poly term nonzero
         {  sum += p[i];
```

```
      ++count;
   }
   if(par[i] > 0)  //if measured par present
      c[c_indx][i] = 1;  //overwrite count term
   else c[c_indx][i] = 0;  //    " "
}
p_avg = sum / count;  //poly term average
fprintf(pro_ln, "p_avg: %d\n", p_avg);
sum = 0;
for(i = 0; i < delta_terms; ++i)
   sum += abs(d[i]);
avg = sum / delta_terms;  //average abs of delta terms
abs_delta = abs(delta);
fprintf(pro_ln, "abs_delta: %d   abs_delta_avg: %d\n",
   abs_delta, avg);
if(abs_delta >= (avg / delta_cut))  //modify coeffs
{  mod_flag = 1;  //set flag for profile
   sum = 0;
   for(i = 0; i < delta_terms; ++i)
      sum += d[i];  //sum up delta terms
   d_avg = sum / delta_terms;  //avg of delta terms
   fprintf(pro_ln, "delta: %d   delta_avg: %d
      d_term: %d\n", delta, d_avg, (delta - d_avg));
   sum = 0;
   count = 0;
   for(i = 0; i < num_pars; ++i)
   {  if(par[i] > 0)  //find correlation
      {  corr[i] = (p[i] - p_avg) * (delta - d_avg);
         sum += abs(corr[i]);  //sum of correlations
         ++count;
      }
      else corr[i] = 0;
   }
   if(count > 0)
      avg = sum / count;  //average abs of correls
   //modify the coefficients:
   coeff_cut1 = (int)(cut1 * (float)avg);  //    " "
   coeff_cut2 =  cut2 * avg;  //size of modification
   fprintf(pro_ln, "avg_corr: %d     cuts: %d  %d\n",
      avg, coeff_cut2, coeff_cut1);
   for(i = 0; i < num_pars; ++i)
      if(par[i] > 0)  //measured parameter was present
      {  if(corr[i] >= coeff_cut2)
            a[i] += mod_amt2;  //increment alpha
         else if(corr[i] >= coeff_cut1)
            a[i] += mod_amt1;  //increment alpha
         else if(corr[i] <= -coeff_cut2)
            a[i] -= mod_amt2;  //decrement alpha
         else if(corr[i] <= -coeff_cut1)
            a[i] -= mod_amt1;  //decrement alpha
         if(a[i] > max_coeff)
            a[i] = max_coeff;  //limit of +max_coeff
         else if(a[i] < -max_coeff)
```

```
                    a[i] = -max_coeff;  //limit of -max_coeff
                }
            }
            d[d_indx] = delta;  //overwrite new delta term
            d_indx = (d_indx + 1) % delta_terms;  //next d term
            c_indx = (c_indx + 1) % past_counts;  //next count
            //write to the learn profile:
            if(mod_flag == 1)  //coeffs were modified
            {  fprintf(pro_ln, "Correlations: \n");
               for(i = 0; i < num_pars; ++i)
               {  fprintf(pro_ln, "  %12d", corr[i]);
                  if(((i + 1) % items_per_line) == 0)
                     fprintf(pro_ln, "\n");
               }
               fprintf(pro_ln, "Coefficients: \n");
               for(i = 0; i < num_pars; ++i)
               {  fprintf(pro_ln, "  %12d", a[i]);
                  if(((i + 1) % items_per_line) == 0)
                     fprintf(pro_ln, "\n");
               }
               fprintf(pro_ln, "Parameters: \n");
               for(i = 0; i < num_pars; ++i)
               {  fprintf(pro_ln, "  %12d", par[i]);
                  if(((i + 1) % items_per_line) == 0)
                     fprintf(pro_ln, "\n");
               }
            }
         }

/*************************************************************
Function:      L E A R N : : C O E F F _ E X C H

The coeff_exch function is called at the conclusion of each
game.  If alpha wins two games in a row, the beta
coefficients are overwritten with the alpha coefficients.
If alpha looses two games in a row, the largest alpha
coefficient is reset to zero.  Finally, the alpha
coefficients are written to the coefficient profile.
*************************************************************/
   void learn::coeff_exch(short win_side)
   {
      short i, j;  //indexing variables
      short sum, sum1;  //sums of counts of parameters
      short largest = 0;  //largest abs of any coefficient
      short lar_indx = -1;  //index into a[]

      if(win_side == 0)  //a draw recorded
      {  wins = 0;  //reset counts
         losses = 0;  //    " "
      }
      else if(win_side == 1)  //a win recorded
      {  ++wins;
         if(wins == wins_needed)  //consecutive wins
```

```
         {  for(i = 0; i < num_pars; ++i)
               b[i] = a[i];  //transfer a to b coeffs
            fprintf(pro_ln, "New beta coefficients\n");
            wins = 0;  //reset count of consecutive wins
         }
         losses = 0;  //reset count of consecutive losses
      }
      else  //a loss recorded
      {  ++losses;
         if(losses == losses_needed)  //enough losses
         {  for(i = 0; i < num_pars; ++i)  //locate largest
            {  if(abs(a[i]) > largest)  //new largest coeff
               {  largest = abs(a[i]);
                  lar_indx = i;
               }
               else if(abs(a[i]) == largest)  //tiebreaker
               {  sum = 0;
                  sum1 = 0;
                  for(j = 0; j < past_counts; ++j)
                  {  sum += c[j][i];  //sum of new counts
                     sum1 += c[j][lar_indx];  //sum of cnts
                  }
                  if(sum > sum1)  //new counts larger
                  {  largest = abs(a[i]);
                     lar_indx = i;
                  }
               }
            }
            a[lar_indx] = 0;  //set largest coeff to 0
            losses = 0;  //reset count of losses to 0
            fprintf(pro_ln, "Coefficient %d goes to 0\n",
               lar_indx);
         }
         wins = 0;  //reset count of consecutive wins
      }
      for(i = 0; i < num_pars; ++i)  //store coeffs
      {  fprintf(pro_coeff, "%12d", a[i]);
         if(((i + 1) % items_per_line) == 0)
            fprintf(pro_coeff, "\n");
      }
      fprintf(pro_coeff, "\n");
   }

/*************************************************************
Function:         L E A R N ::  D A T A _ I N

The data_in function is called at the start of every game.
It determines if the coefficients are present or not.  If
the coefficients are present, the alpha, beta, delta, and
count terms are all read in.  If the coefficients are not
present, the initialize function is called to initialize
these variables.
*************************************************************/
```

```
void learn::data_in(void)
{
   int i, j;  //indexing variables
   FILE *ln_init;  //file pointer for learn variables

   ln_init = fopen("ln_init", "r");  //open file to read
   if(ln_init == 0)  //file not found
   {  ln_init = fopen("ln_init", "w");  //create file
      ln_switch = 1;  //set defaults
      wins_needed = 2;
      losses_needed = 2;
      fprintf(ln_init, "%d  %d  %d\n", ln_switch,
         wins_needed, losses_needed);  //default values
   }
   else  //file was found
      fscanf(ln_init, "%d %d %d", &ln_switch,
         &wins_needed, &losses_needed);
   fclose(ln_init);
   fscanf(poly, "%d %d", &num_pars, &max_coeff);
   j = fscanf(poly, "%d", &d_indx);  // necessary values
   if(j == -1)  //new coeff file detected
      initialize();  //randomly assign values
   else  //read in needed values
   {  fscanf(poly, "%d", &c_indx);
      fscanf(poly, "%d %d", &wins, &losses);
      for(i = 0; i < num_pars; ++i)
         fscanf(poly, "%d", &a[i]);  //read in alpha
      for(i = 0; i < num_pars; ++i)
         fscanf(poly, "%d", &b[i]);  //read in beta
      for(i = 0; i < delta_terms; ++i)
         fscanf(poly, "%d", &d[i]);  //read in delta
      for(i = 0; i < past_counts; ++i)
         for(j = 0; j < num_pars; ++j)
            fscanf(poly, "%d", &c[i][j]);  //counts
   }
   fprintf(pro_ln, "Initial Coefficients:\n");
   for(i = 0; i < num_pars; ++i)
   {  fprintf(pro_ln, "%12d", a[i]);
      if(((i + 1) % items_per_line) == 0)
         fprintf(pro_ln, "\n");
   }
   fprintf(pro_ln, "\n");
}

/*************************************************************
Function:        L E A R N :: D A T A _ O U T

The data_out function writes the data to the coefficient
file.  The alpha and beta coefficients, past delta values,
and counts of previous parameters are saved.
*************************************************************/
  void learn::data_out()
   {
```

```
      short i, j;  //indexing variables

      fseek(poly, 0, 0);  //reset read, write pointer
      fprintf(poly, "%d %d\n", num_pars, max_coeff);
      fprintf(poly, "%d %d\n", d_indx, c_indx);
      fprintf(poly, "%d %d\n", wins, losses);
      for(i = 0; i < num_pars; ++i)
      {  fprintf(poly, "%12d", a[i]);  //write alpha values
         if(((i + 1) % items_per_line) == 0)
            fprintf(poly, "\n");
      }
      fprintf(poly, "\n");
      for(i = 0; i < num_pars; ++i)
      {  fprintf(poly, "%12d", b[i]);  //write beta values
         if(((i + 1) % items_per_line) == 0)
            fprintf(poly, "\n");
      }
      fprintf(poly, "\n");
      for(i = 0; i < delta_terms; ++i)
         fprintf(poly, "%12d", d[i]);  //write delta values
      fprintf(poly, "\n");
      for(i = 0; i < past_counts; ++i)
      {  fprintf(poly, "\n");
         for(j = 0; j < num_pars; ++j)
         {  fprintf(poly, "%12d", c[i][j]);  //write counts
            if(((j + 1) % items_per_line) == 0)
               fprintf(poly, "\n");
         }
      }
      fclose(poly);  //close all files
      fclose(pro_ln);
      fclose(pro_coeff);
   }

/*************************************************************
Function:       L E A R N :: I N I T I A L I Z E

The initialize function randomly assigns values to the
alpha, beta, and delta terms.  It uses a file called
"seed_sav" in order to seed the rand function.  If the
seed_sav file is not present, it is created.  All other
terms are initialized to zero.  Finally, the coefficient
profile is written to.
*************************************************************/
   void learn::initialize(void)
   {
      int i, j, x, y, z;  //indexing and temporary variables
      unsigned int seed;  //seed for rand function
      FILE *start;  //file pointer for seed file
      int rand();  //randomizing function
      void srand(unsigned int);  //seeding function

      start = fopen("seed_sav", "r+");  //open seed file
```

```
      if(start == 0)  //file not found
      {  start = fopen("seed_sav", "w");  //open seed file
         seed = 2;  //initialize seed
      }
      else  //read in seed value
      {  fscanf(start, "%d", &seed);
         fseek(start, 0, 0);
      }
      fprintf(start, "%d\n", seed+1);  //write new seed
      fclose(start);  //close file
      srand(seed);  //seed random function
      d_indx = 0;
      c_indx = 0;
      losses = 0;
      wins = 0;
      for(i = 0; i <= 2; ++i)
         for(j = 0; j < num_pars; ++j)
         {  x = rand();  //call random function
            if(x < arbitr_neg)  //make value negative
               y = -1;
            else y = 1;
            z = (y * rand()) % max_coeff;  //make coeff
            if(i == 0)
               a[j] = z;  //save alpha value
            else b[j] = z;  //save beta value
         }
      for(i = 0; i < delta_terms; ++i)
         d[i] = a[i*2];  //initialize delta terms
      for(i = 0; i < past_counts; ++i)
         for(j = 0; j < num_pars; ++j)
            c[i][j] = 0;  //initialize count terms
      fprintf(pro_coeff, "NEW COEFFICIENTS:\n");  //profile
      for(i = 0; i < num_pars; ++i)
      {  fprintf(pro_coeff, "%12d", a[i]);
         if(((i + 1) % items_per_line) == 0)
            fprintf(pro_coeff, "\n");
      }
      fprintf(pro_coeff, "\n");
   }

//          -----  File:  CH_SEARCH.C  -----
/*************************************************************
Function:       C H _ S E A R C H :: R E C U R

The recur function performs the recursive look-ahead tree
search.  This function is essentially an implementation of
the alpha-beta algorithm presented by Knuth, D. E. and More,
R. W., "An Analysis of Alpha-Beta Pruning," ARTIFICIAL
INTELLIGENCE, 6 (1975) 137-148.
*************************************************************/
   int ch_search::recur(int alpha, int beta)
   {
      short i;  //indexing variable
```

```
   unsigned char moves[20][2];  //used to expand moves
   int static_value;  //score of board
   int new_alpha;  //better alpha value
   short br_count;  //number of move branches
   short br_indx;  //current move branch index
   unsigned int bd_sav[4];  //save original bd position

   generate_mvs(0);  //generate moves or jumps
   //quiescence decision:
   if(gen[4] == 0 || (gen[4] == 1 && ply >= depth))
   {  static_value = score_bd();  //score the board
      return static_value;  //return the score
   }
   for(i = 0; i < 4; ++i)
      bd_sav[i] = bd[i];  //save the original board
   if(ply == 1 && gen[4] == 1)  //root node, no jumps
   {  br_count = order_mvs(moves);  //arrange moves
      for(i = 0; i < 4; ++i)  //  into best order
         bd[i] = bd_sav[i];  //restore the original bd
   }
   else br_count = expand_mvs(moves);
   //basic recursive loop:
   for(br_indx = 0; br_indx < br_count; ++br_indx)
   {  for(i = 0; i < 4; ++i)
         bd[i] = bd_sav[i];  //restore the original bd
      mv_data[0] = m1[moves[br_indx][0]];  //retrieve mv
      mv_data[1] = moves[br_indx][1];  //   " "
      post_mv();  //post the move to bd
      while(mv_data[3] > 0)  //post multiple jumps
         post_mv();
      ++ply;  //prepare for recursive call
      revers_bd();  //   " "
      new_alpha = -recur(-beta, -alpha);  //recursve call
      --ply;  //return from recursive call, reset ply
      if(new_alpha > alpha)  //a better alpha was found
      {  alpha = new_alpha;  //reassign alpha
         if(ply == 1)  //initial branch that lead to
         {  mv_sav1 = m1[moves[br_indx][0]];  //  best
            mv_sav2 = moves[br_indx][1];  //  move
         }
      }
      if(alpha >= beta)  //a cutoff was found
         return(alpha);
   }
   return(alpha);  //no cutoff was found
}
```

VITA

MICHAEL WAYNE SEALE

Candidate for the Degree of

Master of Science

Thesis: STUDIES IN MACHINE LEARNING USING GAME PLAYING

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Clarksville, Arkansas, August 28, 1952, the son of James J. and Bess E. Seale.

Education: Graduated from Hall High School, Little Rock, Arkansas in May 1971; received Bachelor of Science Degree in Electrical Engineering from the University of Arkansas at Fayetteville, in May 1985; completed requirements for the Master of Science degree at Oklahoma State University in May, 1990.

Professional Experience: U.S. Air Force, January, 1975 to Present. Microwave Systems Engineer for the 1842 Electronics Engineering Group at Scott Air Force Base, Illinois, from October, 1985 to July, 1988. Current rank: Captain.