1-1-1983

An Artificial Intelligence Approach to the Game of Backgammon

Chang-Chung Yang

Rochester Institute of Technology
College of Applied Science and Technology

AN ARTIFICIAL INTELLIGENCE APPROACH TO THE GAME OF BACKGAMMON

A thesis submitted in partial fulfillment of the
Master of Science in Computer Science Degree Program

by

CHANG-CHUNG YANG

Approved by:

John A. Biles - Committee Chairman

Peter Lutz - Committee Member

Warren R. Carithers - Committee Member

January, 1983

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Mr. Al Biles, for the directing nd leading me on this project. I woild also like to thank Dr. Peter Lutz or the time he has spent as a rember of my committee. I would like to th- nk Mr. Warren Carithers for the suggestion and encouragement. Finally, I ould like tc thank my wife, Joy, for her help and support during the evolu- ion of this project.

I Chang-Chung Yang prefer to be contacted each time a request for reproduction is made. I can be reached at the following address. 260 Wistown Tr. Beavercreek Dayton OH 45430

e May 11, 1983

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1. OVERVIEW

We have designed a backgammon program to play intelligent games. It can make good decisions of the moves of its checkers. It also can learn from its past experience. This thesis describes how this program works.

This thesis is divided into seven chapters. Chapter one gives some background in Artificial Intelligence and the backgammon game. Chapter two summarizes the history of the last 30 years of computer game playing by some well-known researchers in A.I. field. Chapter three briefly describes the overview of this backgammon program, including the basic playing, input, move verification, output, game starting and terminating, and move generation routines. Chapter four describes implementation & performance of the multiple linear polynomials decision-making algorithm that was used in this project. This chapter also discusses different strategies and the scoring terms that were used in the strategy polynomials. Chapter five describes the fast learning algorithm that was used, and how it improved the performance of

the program. Chapter six gives sample games that the program played with a few human competitors and also analyzes the program's performance. Chapter seven discusses some future developments, extensions of the program, and the conclusions that we have from the result the project.

1.2. INTRODUCTION

One question that has arisen since the advent of computers is, "does a computer do exactly what it is told to do and no more ?" [SIMO 60]. In an attempt to address this question, Herbert Simon has stated:

"This statement is intuitively obvious, indubitably true, and supports none of the implications that are commonly drawn from it. A human being can think, learn, and create because the program his biological endowment gives him, together with the changes in that program produced by interaction with his environment after birth, enables him to think, learn, and create. If a computer thinks, learns, and creates, it will be by virtue of a program that endows it with these capacities. Clearly this will not be a program - any more than the human's is - that calls for highly stereotyped and repetitive behavior independent of the stimuli coming from the environment and the task to be completed. It will be a program that makes the system's behavior

highly conditional on the task environment - on the task goals and on the clues extracted from the environment that indicate whether progress is being made toward these goals. It will be a program that analyzes, by some means, its own performance, diagnoses its failure, and makes changes that enable its future effectiveness"[SIMO 60].

It is also wrong to conclude that the "intelligence" of a computer program cannot exceed that of its human programmer. The potential of Artificial Intelligence in computer field is the reason that this young field has advanced so rapidly in the last 30 years. The dream of A.I. people is that in the near future, machine intelligence will be able to solve problems which are currently impossible for human intelligence [FELD 63].

A favorite area of research in Artificial Intelligence is computer programs that play games. Why should one be interested in game playing? Here are some reasons:

Real life problems usually are perceived in such a vague manner that they are hard to express in a way acceptable to a computer. Because many of the factors involved are nonnumeric in nature, they cannot be stated precisely by those techniques of mathematics which

depend on actual numbers for their exposition. Also, the definitions of terms used in describing real problems are not always understood precisely enough.

Early A.I. researchers (Newell, Shaw & Simon, 1958) recognized that games were precise and well formulated and had some of the important characteristics of real problems. They were mostly nonnumeric in nature, and their behavior was sufficiently unpredictable to be not immediately amenable to known mathematical or data processing techniques. Therefore, it was hoped that as programs were built could learn how to solve games, the knowledge gained could be transferred to real problems. It was also hoped that in the process one would gain insight into the problem solving process [BANE 80].

Besides the first two purposes for studying games, another reason is that it provides a direct contest between man's wit and machine's wit. In short, game environments are very useful task environments for studying the nature and structure of complex problem-solving processes.

The purpose of this thesis is to investigate and implement several theories of Artificial Intelligence using a backgammon game program. This program is able to learn from past experience, make reasonable moves,

detect all illegal moves made by either players, etc. Its performance can be put on the expert level among backgammon players.

1.2. DESCRIPTION OF THE GAME OF BACKGAMMON

Backgammon's history dates back to ancient Sumer, making it one of the oldest games in existence. It was played in one form or another in ancient Egypt, Greece, Rome, Persia, India, China, Japan, Mexico and North America. More recently, it has been played throughout the Eastern and Western worlds, making it one of man's favorite intellectual diversions. It is an ancient and fascinating game, a gambling game which requires both luck and skill. With a single roll of the dice, a winning position can crumble or a seemingly hopeless position can be salvaged. Luck keeps the game interesting, but skillful play always will be rewarded. Backgammon is actually a game of great strategic richness and subtlety which must be studied to be fully appreciated.

Backgammon is a dice-and-board game for two players. Each player begins the game with fifteen checkers of a different color from his opponent's, a pair of dice, a dice cup, and a doubling cube. Players move their checkers around the board according to the roll of the dice. The first player to get all of his checkers

around and finally off the board is the winner. In this thesis, the program will be referred to as X, and the human competitor will be referred to as O (opponent). The starting board position is as follows:

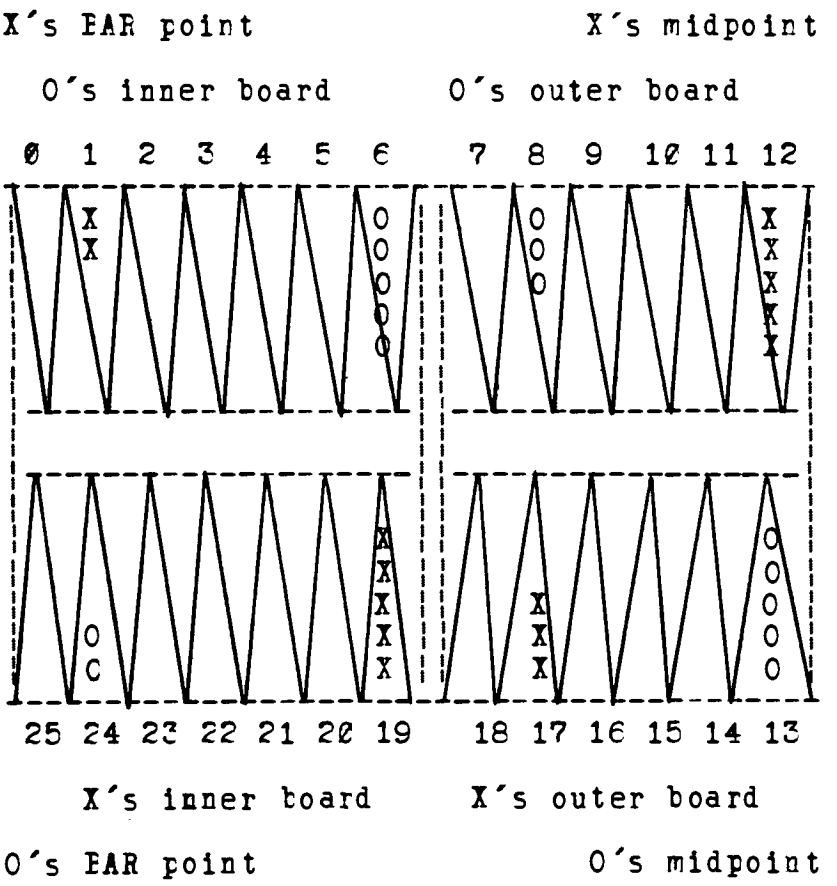

FIG 1.1
THE INITIAL BOARD CHECKERS' DISTRIBUTION

The board consists of 24 triangles called points and is divided into four quadrants, inner and outer

board for each player (figure 1.1). X moves in a clockwise direction, from position 1 to position 24. The opponent (O) moves in a counterclockwise direction, from position 24 to position 1.

After rolling a pair of dice, if the numbers on the dice are not the same, the player either uses each number to move one checker or uses the total of two numbers to move one checker. Each number is considered individually so that when the player uses both numbers to move one checker, he makes two separate moves in succession. If the numbers rolled on the dice are the same, the player uses that number four times. The dice number that needed to move a checker around the board are called "pip count". For example, if X want to move a chcker from board position 12 to 20, then he needs 8 pip counts.

When at least two checkers of one side occupy a point, the point is said to be "owned", and none of the opponent's checker can touch down or land there. On any roll, if the player uses two dice numbers together to move one of his checkers, then we say this checker uses one dice number to touch down a board position and another dice number to keep moving, eventually landing at the destination position.

A lone checker is called a "blot". Should a player's checker land on an opponent's blot, the opponent's checker has been "hit" and is placed on the "bar'. The bar is the board positions Ø and 25 on the board as described in figure 1.1. When a checker is hit, it is placed on the bar, where it is temporarily out of play. If a player has one or more checkers on the bar, he cannot move any other checker until all of his checkers on the bar have been reentered.

Once a player has brought all his checkers into his inner board, he can begin to remove them. This is called "bearing off". If a checker is hit by his opponent during the bearing off process, no more checkers can be born off until that checker has been brought back to his inner board. The first player to bear off all of his checkers wins the game.

There are three kinds of victory: a normal one for an agreed stake if the loser has born off at least one of his checkers. A gammon for twice the stake if the loser has not born off any of his checkers and a backgammon for three times the stake if the loser has not born off a checker and has at least one checker in his opponent's inner board or on the bar.

CHAPTER 2

THE HISTORY OF COMPUTER GAME DEVELOPMENT

2.1. GENERAL DESCRIPTION

The history of computer games development started in the 1950s. Because game situations provide problem environments which are relatively regular and well defined, but which afford sufficient complexity in solution generation so that intelligence and symbolic reasoning skills play a crucial role, computer game developing always is a favorite area of A.I. researchers. This chapter will briefly describe some researchers' efforts on computer games developing in the last 30 years.

2.2. 1950 - 1959

One of the earliest papers regarding computer games was written by the famous English mathematician and logician, A. M. Turing [TURI 50]. He proposed an Imitation Game that used a computer to simulate human behavior. The objective of his paper was to discuss the general question "Can a machine think ?". The question was to be decided by an unprejudiced comparison of the alleged "thinking behavior" of the machine with normal "thinking

behavior" in human beings.

Turing also described a program of chess that was sufficiently simple to be simulated by hand, without the aid of a digital computer. Turing's program considered all legal moves. In order to limit computation, however, he was very careful about the "tinuations". The program considered "continuations" were sucess moves to the current board position.

Turing introduced the notation of a "dead" position: One that in some sense was stable (to the point where the material is not going to change with the next move), and hence could be evaluated. Turing's program evaluated material at dead positions only. He made the value of material dominant in his static evaluation, so that a decision problem remained only if "minimaxing" revealed several alternatives that were require in material. The "minimax" algorithm searches the game tree and follows the branch that will offer the opponent minimun opportunities while giving maximum benefits to the player searching for a move.

He applied a supplementary additive evaluation to the positions reached by making the alternative moves. Although Turing's program was not a very good chess player, it reached the bottom rung of the human intelli-

gence ladder.

In 1956 a group at Los Alamos programmed MANIAC I to play chess [KIST 57]. In the Los Alamos program, all alternatives were considered; all continuation was explored to a depth of two moves; the static evaluation function (fixed depth of the search continuation) consisted of a sum of material and mobility measures; the values were integrated by a minmax procedure, and the best alternative in terms of the effective value was chosen for the move.

Because of computation time limits, a major concession was required. Instead of the normal chess board of 8x8, they used a reduced board, 6x6. For every move, it took 12 minutes to make a move. Although the resulting program was a weak player, it could beat a weak human player. It also was the first example of actual play on a computer.

In 1957, Alex Bernstein developed a chess-playing program for the IBM 704 for the full 8x8 board [BERN 58]. In Bernstein's program, only a fraction of the legal alternatives and continuations were considered. There were a series of subroutines, called plausible move generators, that proposed the moves to be considered. The program considered at most seven alterna-

tives, which were obtained by operating the generators in priority order. The program explored continuations two moves ahead and used the plausible move generators at each stage, so that, at most, seven direct continuations were considered from any given position. Bernstein's program gave us the first information about radical selectivity. in move generation and analysis. This program took 8 minutes to make a move. Its performance was almost equal to Los Alamos program, but its computation time was much less, showing selectivity to be a very powerful device.

In 1958, Newell, Shaw, and Simon [NEWE 58] developed a chess program called NSS (their initials) chess program. One new characteristic of this program that was the use of numerical additive evaluation functions to compare alternatives. This program was organized in terms of a set of goals, which were conceptual units of chess - King safety, passed Pawns, and so on. Each goal had several routines associated with it: (1) A routine that specified the goal. (2) A move generator that found moves positively related to carrying out the goal. (3) A procedure for making a static evaluation of any position with respect to the goal. (4) An analysis move generator that found the continuations required to resolve a situation into dead positions.

The alternative moves came from move generators, considered in the order of priority of their respective goals. Each move, when it was generated, was subjected to an analysis. Which generated an exploration of the continuations following from the move until dead positions were reached, Static evaluations then were computed for them. The performance of this program was superior to its predecessors.

2.3. 1960 - 1969

In 1963, A. L. Samuel developed a checker program that was capable of learning. Basically, this program played by looking ahead a few moves and evaluating the resulting board positions much as a human player might do. "Looking ahead" was implemented by computing all possible next moves, starting with a given board position. The method of scoring the resulting board positions was in terms of a linear scoring polynomial. One way of looking at the various terms in the scoring polynomial was that those terms with numerically small coefficients should measure criteria related to intermediate goals (piece ratio etc). Terms with large coefficients, then would measure long term goals (winning or losing). The achivement of these intermediate goals indicates that the machine was going in the right direction, such that the large terms eventually will increase.

Samuel conducted a series of studies on how to get a computer to learn to play checkers. He experimented with three different learning methods - rote learning, polynomial evaluation functions, and signature tables - and showed that significant improvement in playing checkers could be obtained. The learning procedure in the current thesis is similar to Samuel's polynomial evaluation functions. Samuel's program was trained in several ways, by playing against itself, by playing against people, and by following published games between master players. As the program learned more, it improved slowly but steadily, becoming, in Samuel's words, a "rather better-than-average novice, but definitely not ... an expert".

In 1968, Donald Waterman developed a computer program that learned to play draw poker. Draw poker is a game of imperfect information in which psychological factors become important. Waterman developed a production system to encode a set of heuristics for poker, and he sought to have his program discover these production rules through experience. He tried three different learning methods, automatic training (trained by some book example), an advice-based method (trained by some advise), and an analytic method (trained by analysis the result of games). Among them, automatic training pro-

vided the best performance improvement.

2.4. 1970 - PRESENT

In 1972, James J. Gillogly published a paper, "The Technology Chess Program" [GILL 72]. He used a classic exhaustive search strategy, showing what can be done by brute force search in computer games.

In 1974, Arnold K. Griffith submitted his paper, "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to [ARNO 74]. In this paper he artfully contested the idea of search and fancy static evaluations using the game of checkers. He found out that different learning procedure will give dramaticially different result of performance, he also found out the best learning procedure is automatic learning.

In 1975, Hans Berliner's PhD thesis [BERL 75], "Chess as Problem Solving; The Development of a Tactics Analyzer." dealt extensively with chess. In his thesis, Berliner advocated sophisticated, goal-directed plausible move generation, drastically trimmed search trees, and dynamically determined search depth.

In 1979, Berliner developed a backgammon program called BKG 9.8. In July, 1979, this program defeated the world backgammon champion, Luigi Villa, by the

impressive score 7-1 in a $5,000 winner-take-all match. It was the first time a computer program had beaten a world champion at any board or card game. In Berliner's BKG 9.8 program, he used a so called SNAC (Smoothness, Nonlinearity and Application Coefficients) approach for his move evaluation. Such an approach strongly depends upon solid backgammon knowledge. Unlike the best programs for playing chess, BKG 9.8 uses more by positional judgment than brute calculation. This means that it plays backgammon much as human experts do.

In 1980, David Wilkins [WILK 80] investigated the extent to which knowledge can replace and support search in selecting a chess move. He developed a program called PARADISE (PAttern Recognition Applied to DIrecting SEarch), which finds the best move in tactically sharp middle game positions from the games of chess masters. It encodes a large body of knowledge in the form of production rules. The actions of the rules post concepts in a data base of chess position and their scoring value while the conditions match patterns in the board positions in the data base. The program uses the knowledge base to discover plans during static analysis and to guide a small tree search which confirms that a particular plan is best. The search is "small" in the sense that the size of the search tree is of the same order of

magnitude as a human master's search tree.

Once a plan is formulated, it guides the tree search for several plys and expensive static analyses are done infrequently. PARADISE avoids placing a depth limit on the search, by using a global view of the search tree, information gathered during the search, and the analysis provided by the knowledge base. PARADISE exhibits expert performance on any position it has the knowledge to understand. This program has 3 possible failures. (1). The best plan is never suggested. (2). The search becomes unbounded. (3). A mistake is made in the analysis.

The above summaries only biiefly describe a few papers of game develop in the last 30 years, but it shows the trend and progress of the game development history. We can see that the more recent game program shows more new techniques and their performance of game are also much tetter.

CHAPTER 3

GAME MANAGEMENT PROGRAM

3.1. GENERAL DESCRIPTION

This chapter describe the backgammon game set up and management program, because the rule of backgammon is very complicate and have a lot exception, this part took us quite a while to program it. This is a prolong but necessary step to let the computer knows the rule of backgammon.

The literature on game playing programs suggests that, for most games, building positional judgement of the current board situation into a program is extremely difficult. Hence an enormous amount of information goes into such judgments. Backgammon, however, is not such a case; it has a domain where it is possible to compare two situations and make a judgment about which one is the better without having to worry about an exhaustive analysis.

The program was developed in three stages: (1) Set up the basic game playing program. (2) Implement decision-making process. (3) Implement learning and improvement process.

3.2. DATA STRUCTURES

The basic data structure in the backgammon program represents the game board, illustrated in FIG 1.1. For the convenience of positional judgment and board evaluation used in the decision-making and learning stages, We chose a simple data structure to describe the board - a one dimensional integer array of 26 elements (indexed from Ø to 25). Each element in this array stands for one position on the board and contains the number of checkers at this position. Element Ø is X's BAR point, element 25 is the opponent's BAR point. If one element in this array is positive, then the corresponding board position is occupied by X's checkers. If it is negative, then the corresponding board position is occupied by the opponent's checker(s). If it is zero, then the corresponding board position is empty.

At the beginning of each game, the program initializes the board's array position 1 to +2, position 6 to -5, position 8 to -3, position 12 to +5, position 13 to -5, position 17 to +3, position 19 to +5, position 24 to -2, and the rest of the positions to zero as illustrate in FIG 1.1.

After a move is made by either side, the board is updated. When it is X's move, he makes moves according

to the dice numbers he rolls. The move is from some starting position (must be occupied by his checker(s)) to a destination positions (must not have more than one of opponent's checkers). The starting position will be decreased by the number of checker(s) that leave this position. If the content of the destination position is greater than -1, then it will be increased by the number of checker(s) that moved to it. If the content of the destination position is -1, then its content will become the checkers' number that moved to it, and the opponent's BAR point will be increased by -1.

When it is the opponent's move, a similar board-updating routine is performed, the only difference being that the content of the starting position will be increased by the number of checkers that leave it, and the destination position's content will be decreased by the number of checkers moved to it. If X's blot has been hit, then X's BAR point will be increased by 1.

There are two other memory locations called Xbear-off and Obearoff. Xbearoff is the number of checkers that X has born off. Obearoff is the number of checkers his opponent has born off. When either player bears one of his checkers off the board, his bear off location will increased by 1. Whichever player's bear off location reaches 15 checkers first, is the winner.

GAME MANAGEMENT PROGRAM

Because the data structure of the board is so simple, it enables the development of very complicated position judgment and board evaluation algorithms in the decision-making and learning stage. This is one reason that backgammon is one game that is very suitable for computer game playing.

Another data structure is the move structure which contains 3 elements.

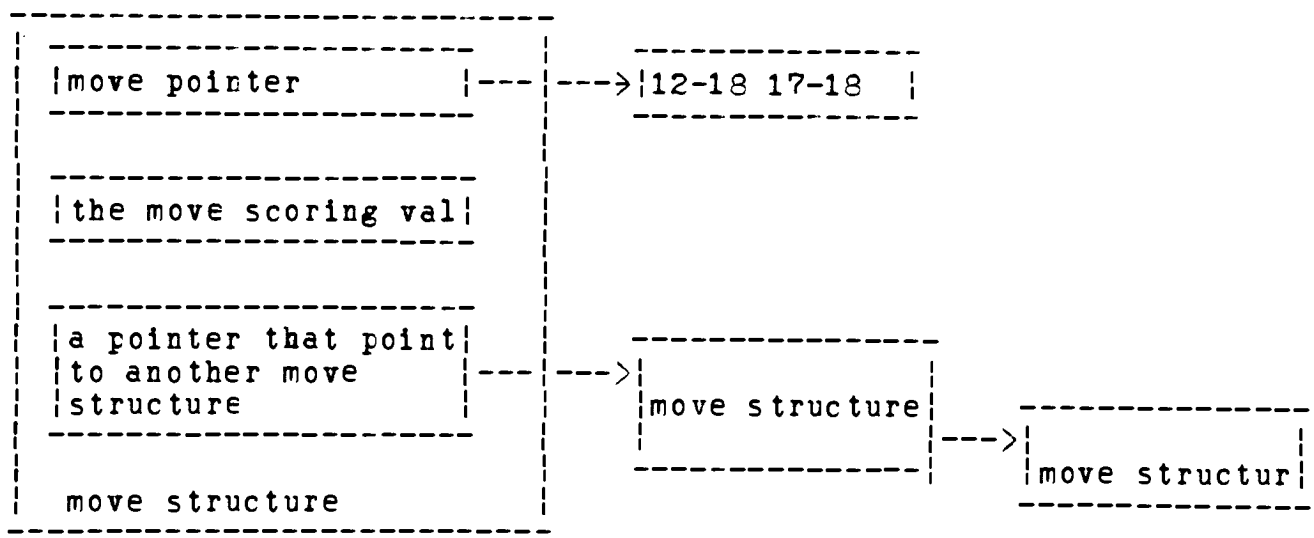

FIG 3.1
THE LINKED LIST ARRAY OF MOVES STRUCTURE

The first element is a pointer to an integer array that contains the checkers' move positions (starting positions and destination positions). The second element is a long integer value that is set by an evaluation function in relation to the board after the above move is made. This reflects the quality of the move in

this move structure. The third element is a pointer to anc ther move structure. A moves generating routine generates all the next legal moves to a moves linked list array. Each element in this array is a move structure as described above. The program will allocate approximate memory space to this moves linked list array depending on how many legal next moves it has (sometimes the legal next moves are more than one hundred; Other times it may be none).

3.3. BASIC ALGORITHM

The basic algorothm of the overall backgammon program is described in Figure 3.1. Not considering the detail algorithm (like decision-making and learning), the basic algorithm is simple and straight forward, two players just take turns to make their move, program will verify their moves, also find out who win the game.

GAME MANAGEMENT PROGRAM

```
Input two dice numbers
While two dice numbers are equal
   Input another two dice numbers
Decide whose turn it is

While game not over
   Select strategy
   Show the current board

   If it is opponent's turn
      Enter opponent's move
      While this move is illegal
         Enter another move
      Update the board according to a legal move
      If opponent's bear off location reaches 15 checkers
         Opponent win the game and game is over

   Else now is program's turn
      Pass the selected strategy polynomial to
       the evaluation polynomial
      Generate all legal next moves
      Score all legal next moves and choose the
       highest scoring move from them
      Teacher enters the move that he thinks is
       the best move
      Program learns by adjusting polynomial coefficients
       until program agrees with teacher's move
      Update the board according to this move
      If program's bear off location reaches 15 checkers
         Program wins the game and game is over

   Input the next two dice numbers

End of while
```

FIG 3.1.
BASIC ALGORITHM

3.4. USER INTERFACE

All input and output activities are made through a CRT. At the beginning of execution, the program prints

a listing of coefficients that are used in the different strategys' polynomials. It then requests the player to enter the dice numbers of the opening roll for each player. If the two dice numbers are the same, then the program continues to request another opening roll until two dice numbers are not equal. The higher dice number's player uses both dice numbers to make his first move after which all the moves are taken in turns. Before each player makes his move, the program prints out the current board position and whose turn it is.

A move is specified by entering the starting positions and destination positions on the board. For example, at the beginning of the game, if X's dice numbers are {6,1}, the move that is selected to input is expressed as:

12-18 17-18

The 12, 17, 18 stand for the positions on the board as shown in FIG 1.1. Each single movement is separated by a space (like space between 12-18 and 17-18), and the dash between two integers means "to' (12-18 means move one checker from position 12 to position 18). One rule for making moves is the movement must be completed. This means that even though a checker only touches down in one position (without landing there), this touch down

movement also must be specified. For example, the dice roll {6,1} also could make the move:

12-18 18-19

Actually there is only one checker moved from position 12 to position 19. It only touched down on position 18, but the total movement must be specified, because if location 18 is owned by the opponent's checkers, this move becomes illegal.

During the game both player should input the correct move input according to the rule as described above, after one player wins the game, the program prints out who wins the game and which kind of victory he has - A normal win, A gammon or A backgammon.

3.5. SYSTEM FILES

Two system files are used by the program. One file, called movefile, is used to keep a record of all the moves that are made during the game. Before the opponent makes his moves, he can request to see this file. After each game, the movefile constains all moves that were made during the game, and opponent could check this move file, figure out what kind errors he had make during the game.

The other system file, called coefile, is used to keep all coefficients that are used in the 5 strategy polynomials. Each polynomial has one line of coefficients in this file. There are also 3 long integers in this file for the low boundaries of RUNNING GAME, HOIDING GAME, and ATTACKING GAME. The purpose for the 3 lower boundaries is to decide when the current strategy is not suit for the current board situation any more. If the highest move scoring value is smaller than the lower bound, then program will give up the current strategy and choose another strategy. ALL coefficients are initialized to 50. During the learning process, they are adjusted up or down depending on the importance of their corresponding polynomial terms. At the end of each game, these new, adjusted weights are saved in coefile and used at the beginning of the next game. It functions as human's experience and after a few games, the weights reach relatively stable values.

3.6. MOVES VERIFICATION ROUTINES

There are four move verifying routines, two for X's move verification & two for the opponent's move verification. The multiple routines are required because backgammon has 4 different sets of rules for legal moves depending on the stage of the game. Because the rules of backgammon are very complicated and have many

exceptions, these four move verifying routines are rather complex.

A given move made by either player may have Ø to 4 individual movements, depending on the two dice numbers and the current board position. In every move, the player must make all possible individual movements. Four parameters are passed to each of the four verification routines, the current board array, the move to be attempted, and the dice numbers. These four routines return a Ø if they find that the move passed to them is legal A 1 is returned otherwise. Each player users two move verifying routines, one for the middle game and another one for the end game stage, because the rule of backgammon is different between middle game and end game, so the each player have to use two different verifying routine for different time during the game.

3.7. MOVES GENERATOR ROUTINES

There are two move generating routines, ALLMOVES and ALLENDMOVES. ALLMOVES generates all the legal next moves for X without bear off consideration. ALLENDMCVE generates all the possible moves that will bear off some checker(s). Both routines generates a linked list of potential moves. Most of the time the program will use only one of these two move generater, but in some cases,

both are used. For example, when X has only one checker left at his outer board and the rest of his checkers are in his inner board, he can move his checkers around the board without bearing off any of his checkers. He can also move his outer board checker to his inner board and bear off some of his checkers. In such a case, both moves generating routines will be used.

After the move generators are called, the moves link list of moves produced by "ALLENDMOVE" is linked to the end of "ALLMOVES" linked list to form one long linked list that includes all the legal next moves.

CHAPTER 4

DECISION MAKING

4.1. GENERAL DESCRIPTION

This chapter describes how the program chooses the best move from all the legal next moves. This decision-making process is very important because the better choice the program can make, the more "intelligence" it shows.

Berliner's backgammon program [BERL 80], in the central idea was that the evaluation space (the different time and different board situations during any one game) was warped in such a way that in certain parts of the space, a particular feature could be more important than it was in other parts. The transition in importance from one part of the space to another part was made smoothly, by slowly changing the coefficients of the move evaluation polynomial. Moreover the transition depends on the other features present. Which means that the importance of a particular feature is a non-linear function. The features that control the transitions are called application coefficients. They are special slowly changing variables that replace the normal constant coefficients in the linear polynomial

evaluation function.

Berliner tried using one giant nonlinear polynomial evaluation function that was applied to all different strategies at different times during the game with all the coefficients changing slowly during the game. Such a method depends on a master level backgammon player's knowledge in order to know when and how these coefficients should be changed, and it is not very flexible to switch among different strategies. The biggest disadvantage in Berliner's BKG 9.8 was that his program could not learn, because all the coefficients in his program depended on the programmer's knowledge of backgammon, and could not be adjusted by the program itself.

What we have done in the present thesis is this: Instead of building a single giant polynomial that covers all different strategys, we set up 5 different polynomials, each used in different board situations and different times during the game. In this way each polynomial concentrates on only one kind strategy, without being affected by other considerations that are very important in other strategies but have nothing to do with the current one. The program, then, will not be confused by some contradictory factors which were used in different strategies. In this a way, each polynomial will be specifically suitable for the environment to

which it applies.

All coefficients of these five polynomials can be adjusted by the program while it plays with its opponent, which shows rudimentary learning. Each strategy has a lower bound. Whenever a move's scoring value from the evaluation function is lower than its lower bound value, the program will abandon its currently strategy and check the current board situation to find a better strategy for the current board situation. It then starts using the new strategy from X's next roll. This approach, is called MSIP, for Multiple Switchable Linear Polynomial.

4.2. EVALUATION FUNCTION

The five different strategy polynomials are RUNNING GAME, HOLDING GAME, ATTACKING GAME, BEAR IN and BEAR OFF. Each of them has a different number of terms and different range of coefficient values. The evaluation function is only two lines of "C" code and can be applied to all five polynomials. It is written in "C" as:

```
for (i = Ø; i < l; i++)
   np->polyval += ((*termpt[i])(b,board) * coeffs[i]);
```

Where l is the length of the current strategy polynomial, and np->polyval is the strategy evaluation value

of the board after the move in question has been made on the board. termpt[i] is an array of function addresses. When using different strategies, different function addresses will be passed to this array. In (b,board), "b" is the board array after a move has been made, and "board" is the board array before that move was made on the board. (*termpt[i])(b,board) will return an integer value that was evaluated by a term function. coeffs[i] is an integer array, each element of which is the weight of a term.

The term value multiplied by its weight will give a scoring value of one consideration factor for the board after a move has been made. Each strategy polynomial can be expressed mathematically as:

$$S = \sum_{i=1}^{l} Wi * Ei$$

Where S is the strategy's scoring value of the board after a move has been made. Wi is the weight of one term in a polynomial and is always a positive integer value. Ei is the evaluation value of a term function.

The program will pass each term function two board arrays, the board array before a move was made and the board array after a move was made. Each term will return

an integer value which may be positive, zero or negative, depending on what factor this term is considering. If it is positive, this means that after X has made that move on the board, his future board development has been enhanced. If it is zero, this term has no relation to this move. If it is negative, then this move had a bad effect on X's future board development. How important each term is, will depend on its weight.

4.3. STRATEGIES AND THEIR CONSTITUENT TERMS

The five different strategies uses a total of forty-five terms. Some terms are used only in one strategy, while others are used in more than one. At the beginning of execution, the RUNNING GAME strategy (also called NORMAL GAME) will automatically be chosen to make the move selection. After a few moves, the program will check who is ahead in the game and how good or how bad the board situation is. It then will choose the strategy that it is best for the current board situation and use it for X's future move selection. The newly selected strategy may be switched during the game when it no longer suits the board situation, but for reasons of consistency, the program will not let the strategy switch happen too often.

RUNNING GAME, HOLDING GAME and ATTACKING GAME strategies are used in the middle game. When the end game stage is reached, the program will switch to BEAR IN and BEAR OFF strategies. When there are more than 11 and less than 15 of X's checkers in his inner board, and X needs less than 40 pip counts (the dice numbers needed to move X's checkers) to move all his outer board checkers to his inner board, the program will automatically choose the BEAR IN strategy for X's future moves selection. After all of X's checkers have moved to his inner board, the program will begin using the BEAR OFF strategy for X's future move selections.

In this chapter, I'll will only explain a few term functions here, all the detail of the 45 terms that were used the five strategies polynomials will be explained in APPENDIX A : 5 STRATEGIES AND THEIR TERMS.

The RUNNING GAME strategy includes 25 terms. This strategy will be selected when X is far ahead in the game. Its goal is to move X's checkers around the board as quickly and safely as possible. Each of the 25 terms' evaluation functions will give a scoring value to the new board that assumes a move has been made on it.

One of the RUNNING GAME strategy's terms that can look 2 rolls ahead is the BEENHIT term. "BEENHIT" fig-

ures the probabilities that X's blots will be hit by the opponent. This term can look two moves ahead. It will check that after X makes a move, how many X's blots will left on the board, and the probabilities of all blots that will be hit by the opponent's direct shot (within 6 pips in front of X's blot) and indirect shot (more than 6 pips in front of X's blot). It also counts how much ground will he lose. For example, if the board position 20 has an X checker, after having been hit, it must return to X's BAR point - position 0, and X must rolls 20 more pip counts to let this checker back to position 20 after been hit, X lose 20 pips. This term can be expressed as:

$$\text{term value} = -\sum_{i=1}^{n} Pi * Li$$

where n is X's blot numbers. Pi is the probability that a given blot will be hit by the opponent. Li is the pip counts lost when a blot is hit by the opponent. "HITOPPONENT" is similar to the "BEENHIT" term. It can look 3 moves ahead and calculate the probabilities that different positions will hit the opponent's blot(s). It then finds the best position in front of the opponent's blot(s). If the opponent's next move cannot cover his blot(s), X's checkers will have the highest probability

of hitting his opponent's blot(s). BEEBHIT and HITCP-POINT enable the program to look a few moves ahead without doing the exhaustive search done by many game programs. This saves considerable memory and time.

"OBLOTOUT" will count that after X made a move, how many of the opponent's blots had been hit by X. Of course, X would try to hit as many the opponent's blots as possible. This term will return a positive value, if X has a prime (owned 5 or 6 continue points) between position 16 and 24, then hitting the opponent's blot(s) will return a very high positive value.

The other two middle game (before X starts bear in and bear off his checkers) strategies are called HOLDING GAME (also called PRIMING GAME) and ATTACKING GAME. The chief objective of HOLDING GAME isfor X to hold a point or points in the opponent's inner or outer board in order to prevent him from safely coming home, and to try to tc trap some of the opponent's checkers behind X's prime. This strategy will be used when X is behind in the game by one or two rolls because he cannot keep using the RUNNING GAME strategy, merely hoping for some lucky doubles to bring him ahead in the game. He has to do something more aggressively in order to change the board's situation.

The chief objective of ATTACKING GAME is: X try to hit and attempt to close out his opponent, usually hitting in X's inner board. This strategy will be used when X is hopelessly behind in the game (at least 3 rolls behind his opponent). The only hope that X has to win is to try to hit as many of his opponent's blots as he can, even though he will leave some blots, and to try to build X's inner points in order to establish an inner board prime to hopefully close out the opponent's BAR checker(s). Those terms which were used in HOLDING GAME and ATTACKING GAME are the same. Both of them have 26 terms, but the weights of the terms are very different. Most of their terms are the same as those terms which were used in RUNNING GAME. These terms include "POS20", "POS18", "POS5", "POS21", "POS4", "POS19", "POS22", "POS12", "TOTAIPT", "CONTINPT", "OVERLOAD", "BEENHIT", "HITOPPONENT", "OBLOTOUT", "XBLOTOUT", "INNERBLOT", "OINBAR", "DEADCHECKER", "BUILDERS", "DUPLICATION", "DIVERSE", "XBLOCKED", "OBLOCKED", "MIDDLEMEN", etc. There are two other terms also used in HOLDING GAME and ATTACKING GAME strategies. One is "CLOSEOUT", another one is "HITMEN". In RUNNING GAME, those X's checkers that in the opponent's inner board are considered as very bad factors to X. Because when they move toward X's inner board, they are very easy been hit by the opponent. But in HOLDING GAME and ATTACKING GAME, it is

not such case. These checkers become an asset for X, they are like guns thta point to the opponent's head when the opponent moves his checkers toward his inner board, so these X's back checkers should tries to stay at their position as long as possible, and in the right time to hit the opponent's blot(s).

There are two other strategies used in the end game (when X starts to bear in and bear off his checkers) - These are the BEAR IN and BEAR OFF strategies. BEAR IN strategy will be used when most of X's checkers are in his inner board with only a few left outside his inner board. This strategy deals with how to bear X's checkers into his inner board with or without opposition. It is also necessary to distinguish between wanting to save a gammon or win a race while bearing in. There are 9 terms were used in BEAR IN polynomial.

The last strategy used in the program is the BEAR OFF strategy. There are 13 terms used in BEAR OFF strategy polynomial, four of which already have been used in the BEAR IN strategy, "CLEARRIGHTMOST", "OBLOTOUT", "INNERPTS", "XINBIOT". The other nine terms include: "TAKEOFF", "SPREADOUT", "OFFBEENHIT", "SPAREMEN", "DANGERMEN", "INNERGAP", "FORCEGO", "EVENMEN".

4.4. APPROXIMATE DECISION MAKING TIME

The times needed for the program to select a move will depending on the number of all the next legal moves. Normally, rolling doubles will need more time than rolling two different dice numbers and small dice numbers will need more time than large dice numbers. From my experience, decision making never took more than 5 seconds and usually only about 1 or 2 seconds. Compare this with human competitors' speed which is usually about 20 seconds. We must admit, then, that the machine's thinking speed is considerably faster than the human's thinking speed.

CHAPTER 5

LEARNING AND IMPROVING

5.1. GENERAL DESCRIPTION

One thing that Berliner's BKG 9.8 program did not do was learn. The program discussed in this thesis has this ability. The way it learns is similar on the surface to a human's learning behavior. However, the actual learning mechanism in the program is probably not much like the mechanisms in the human brain.

The learning procedure is as follows: When it is X's turn to move, the program will select the highest scoring move from all the legal next move and print this move out through the CRT. Then it will ask the teacher to enter a move which will be considered as the best move. After the teacher has entered his move, the program begins adjusting the polynomial's weights (as described below) until the teacher's move becomes the highest scoring among all legal next moves. The program then will make this move on the board. If the teacher's move is the same as the program's initial selection in the beginning, then all weights of the polynomial will not be changed. In this way, the teacher can teach the program to play different levels of games. He can use

world champion game examples to teach the program to play expert level games. He can also use some beginner's game examples to teach the program to play very poor games. How good the program plays will depend on the weights in the coefficients file which depend on which kind of games the teacher was taught the program.

5.2. FAST LEARNING

Many programs described in the liturature were capable of improving their performance by their past experience, but most of them learned only after a complete game had been played. By the result of winning or losing the game adjustments to the strategies are made. This made of learning requires many games and only can do some general adjustments to the strategies that were used during the games. Obviously, with the gradualarity of change so large, becomes slow and inefficient.

The learning algorithm that was used in the MSLP program is called the Fast Learning Algorithm (FLA). It learns after every move and needs only 3 to 4 game examples from some backgammon instructional books to become a "good" player. What the teacher needs to do is enter all the moves according to the book's game examples. After learning from the sample games, the program can play almost as well as the player who played the sample

games. Such learning algorithms save a lot time of learning and the teacher doesn't need to have a vast amount knowledge of the games. In fact, anyone can use some backgammon books' game samples to teach this program to play expert level games. The ability of the game playing program then will not be limited by the teacher's ability.

5.3. LEARNING ALGORITHM

The learning routine used in this program is called "LEARNING". Its algorithm is as follows:

b1, b2 are two temporary board arrays, both initialized to the current board array. The program's best move will made on b1, the teacher's move will made on b2. FIGURE 5.1 shows the learning algorithm.

```
Copy original board array to b2
Make teacher's move on b2

While1 teacher's move still not the best move
   Calculate teacher move's evaluation value
   Copy original board array to b1
   Make current best move on b1
   If current best move is teacher's move
      Learning finish, exit while1 loop
   Else
      For each term, figure difference in term values between
       teachers move and currnt best move

      While2 current best move's evaluation value greater
       than teacher move's evaluation value
         For each term value
            If teacher move's term value is smaller than
             current best move's term value
               Then decrease the corresponding weights.
            If teacher move's term value is greater than
             current best move's term value
               Then increase the corresponding weights
            Use the new weight's polynomial to evaluate
             both teacher move's total scoring value and
             best move's total scoring value
      End of while2

      Select the highest scoring move from all next moves

End of while1
Copy the new polynomial's coefficients array to the
 coefficient file
```

FIG 5.1.
LEARNING AIGORITHM

The idea of the learning algorithm is to keep comparing two moves, the teacher's move and the current best move. At the beginning of the learning procedure, the teacher's move may not have the highest value among

all legal next moves. If this is so, then the program will find the current highest scoring move and compare every one of its term's values to the corresponding teacher move's term values. If the term value for the teacher's move is lower than that for the best move, the corresponding weight will be decreased. If the term value for the teacher's move is higher than that for the best move, the corresponding weight will be increased.

Using this weight adjust mechanism, the teacher's move eventually will score higher than the current best move. The program then will use the new adjusted weight to reevaluate the legal moves checking to see whether the teacher move is now the highest scoring among all legal moves. If it still is not, then the coefficient adjustment cycle is repeated until the teacher's move finally outscores all other legal moves.

In the learning algorithm, we are not always increasing or decreasing the weights by the same value. The magnitute of the change will depend on each weight's past history of change. In the weight adjustment cycle, if a weight usually increases, it means this weight's value should be much higher. The more times it is increased, then, the higher will be the value added to it. For example, the first time a weight is increased by 3. If it should be increased again, then it will be

increased by 6. The third time, it will be increased by 9, etc. Similarly, decrementing the weights also is done the same way, except when a weight reaches 1, it will not be decreased any more. An integer array called x1is used to stcre the terms "record". The program compares the teacher's term value to the corresponding current best move's term value. If the teacher's term scores higher than the current best move's, then the correspon-ing x1 array's element will be increased by 1. If the two terms areal, then the corresponding x1 array's element will not changed. If the current best move term value outscores the teacher's move, the corresponding x1 array's element will be decreased by 1. In this way, the x1 array will record all terms' past performance. There are four different situations in changing the weights of the polynomial: for every element in the x1 array: (1) If the element should be decreased and its current value is already negative, also if its corresponding weight subtracts by 3 times the x1 array corresponding element's value is greater than zero, then the corresponding weight will be decreased by 3 times this element's value. (2) If this element should be decreased, and its current value is positive, also if the corresponding weight subtracts by 2 is greater than zero, then the corresponding weight will be subtracted by 2. (3) If this element should be increased, and its

current value is greater than zero, then the corresponding weight will be added by 3 times the x1 array corresponding element's value. (4) If this element should be increased, and its value is negative, then the corresponding weight will be added by 2.

5.4. APPROXIMATE LEARNING TIMES

The time that the program needs for learning is much more than the move selection time. The learning time depends on the scoring value of the teacher's move. If at the beginning of a learning cycle, the teacher's move scores very low compared with all legal next moves, then the learning time will be very long, sometimes more than one minute. On the other hand, if the teacher's move scores scoring value is high among all legal next moves, then the learning process takes only a few seconds. Normally, after teaching the program a few games, the learning time decreases dramatically for each move, because the program's moves and the teacher's moves are much closer. One thing that should be mentioned here is that when designing terms in the strategy polynomials, these terms must not contradict each other, otherwise the learning process will become an infinite loop.

CHAPTER 6

PROGRAM PERFORMANCE

6.1. GENERAL DESCRIPTON

The performance of the MSLP program will be described in this chapter. After using 4 different strategy game examples (including RUNNING GAME, HOLDING GAME, ATTACKING GAME and a MIXING STRATEGY GAME) from a backgammon instructional book to teach the program, the program wins 90% of its games against human competitors. Althcugh it never had chance to compete with master level players, the players it did play admitted that the program played a very strongly game. Unless the dice numbers it rolls are very bad, it should have very high chance of winning.

At the beginning of the first game, all weights in the coefficients file are set to 50. After using the learning algorithm that was described above, all weights are adjusted up or down depending on the importance of their corresponding terms. After playing 4 games that were selected from a backgammon instructional book, the five strategies polynomials' weights were as follows:

(1) RUNNING GAME WEIGHTS

50 50 50 26 50 50 50 14 2 50 10 89 2 5 50 2 50 50 50 12
12 2 14 20 2 30

(2) HOLDING GAME WEIGHTS

50 2 250 207 1 50 827 2 3 1223 1250 1 123 3 1100 2 122 50
5 1 123 79 2508 1884 87 1110

(3) ATTACKING GAME WEIGHTS

52 53 310 188 219 45 778 3 6 3 7 8 870 3 663 5 3 47 6 5
663 5 987 73 98 5

(4) BEAR IN WEIGHTS

92 86 41 29 41 54 8 50 50

(5) BEAR OFF WEIGHTS

96 5 5 2 44 50 2 5 50 50 338 50 60

From the weights that were described above, we can see that the three middle game strategies' weights are very different from one another, although most of their terms are the same. When the program using the above weights plays human competitors, the results shows that the program knows when and how to make reasonable moves.

In this chapter, I will use two game examples to demonstrate the program's performance. In both examples, X will stand for the program's checker, and O will stand for opponent's checker. On the board, each X's checkers will be represted by positive numbers while the

opponent's will be represted by negative numbers.

6.2. FIRST EXAMPLE

The first game example is illustrated in Appendix B: GAME EXAMPLE ONE. It is a simple race game, and only took 37 rolls, compared to GAME EXAMPLE TWO's 80 rolls. It is a short game. Basically, in GAME EXAMPLE ONE the program used a RUNNING GAME strategy, running it's checkers around the board as quickly and safely as possible.

The program got off to a good start, running a back checker out and safely landing on board position 12 on the first roll. After roll 5, X had built a strong inner board (occuppied position 19, 20 and 22), and is 22 pip counts ahead in the game. Now the program made up his mind to use a RUNNING GAME strategy for future move selections. On roll 7, the program ran his second back checker out from the opponent's inner board, but unfortunately this checker was immediately hit by the opponent on his next roll. X lost 9 pip counts, but he still was ahead in the game, so it continued to used the RUNNING GAME STRATEGY.

Roll 11 for X
X to play {6,4}

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

After roll 10, the opponent had built a strong 5 points prime from position 4 to 8. The single X checker left on position 3 is really in danger now, and X's only chance to run this checker out of the opponent's 5 points prime is that whenever he rolls a dice number 6, use this dice number 6 to move his back checker out of the opponent's prime immediately. On roll 11, X rolls a {6,4}, the program correctly used the dice number 6 to move his back checker from position 3 to position 9. Morecver, because the single X checker on position 9 still was pointed by the opponent's checker on the board position 11, the program wisely use dice number 4 to keep moving this checker from position 9 to 13. Now he has not only run both his checkers out, he also has made all his blots safe and has only two indirect shots left at 8 and 11. Right now the RUNNING GAME strategy is working very well.

Roll 13 for X
X to play {5,3}

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

After roll 12, X's two blots on position 13 and 16 present an immediate danger. Since the opponent has closed 4 points in his inner board, if any of X's blots are hit by the opponent, he will be in danger of being closed out. He must, therefore, save both of his blots on his next roll. On roll 13, X rolls {5,3} and smartly moves 12-17 13-16. With a single number, 3, X safeties both his blots and avoids the danger of been hit. He also uses the dice number 5 to move one extra checker from position 12 to 17, instead of wasting this dice number in his inner board.

Roll 15 for X
X to play {4,4}

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

Roll 15 is a critical moment for X. Although X is ahead in the game, the opponent still occupies the 18 point, and he has closed 5 points in his inner board. If X left any blot in the next few rolls, and this blot

were hit by the opponent, X probably will lose a gammon (he will lose twice the stake). So now X's highest priority is to try to move his outer board checkers to his inner board as safely as possible. He rolls a double 4, moves two checkers from position 12 to position 20, this leaves 5 checkers outside his inner board, but they are very close to his inner board enabling him to move his outer board checkers to his inner board more easily. Any other moves will either force him to leave a blot in his outer board or waste a few pip counts in his inner board, which would slow down X's bear off procedure.

Roll 17 for X
X to play {5,4}

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

On this roll 17, the program correctly moves 16-21 16-20, moving two checkers from X's outer board to his inner board without leaving any blot.

After roll 18, the opponent brings his two rearmost checkers closer to home, which makes future contact impossible. For the rest of this game , the only consideration will be how to move as fast as possible and

bear off as efficiently as possible. On each roll, X tries to bear off as many checkers as possible. If he cannot bear off any more checkers, then he uses the dice number(s) to fill up his inner board gap(s). After roll 37, X had born off his last checker and the opponent still had 3 checkers in his inner board. X then, wins a normal game.

6.3. SECOND EXAMPLE

The second game example is illustrated in Appendix B: GAME EXAMPLE TWO. Thwas game is played by the program against one of my thesis committee members, Mr. Warren Carithers. He is an experienced Backgammon player and knows a lot strategies of Backgammon. This game example took 80 rolls - much longer than the first game example. In this game, the program uses all five strategies at different times, so it is a good example for illustrating the capabilities of the program.

At the beginning of this game, the program uses the RUNNING GAME strategy. After roll 7, the board situation was as follows:

Roll 8 for X
X to play {4,5}

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

The opponent had run one of his back checkers out from X's inner board and is 9 pips ahead. The program figures out that the RUNNING GAME strategy is not useful any more, so it selects another strategy - the HOLDING GAME strategy. Now X has dice number 4 and 5, he decides to make the move, 12-16 12-17. Readers may ask why the program give up the 12 point and left a blot there? The reason is that: X's main objective now is to build a strong inner board prime to try to block the opponent's checker that was on the position 24. He therefore, brings as many checkers as possible close to his inner board. The blot on position 16 should be considered as a builder to his inner board point, and the blot on position 12 was forced to to leave, but it only can be hit by the dice number 1. Compare to all other options, this blot was the safest one.

Roll 10 for X
X to play {2,1}

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

For roll 10, X moves 12-14 19-20. 12-14 saves the blot on position 12, 19-20, diversifies the builders. Now positions 16, 17, 19, 20 all have one builder (the two checkers that occupy a point do not count as builders unless one wants to give up this point, otherwise these two checkers cannot move any one of them).

Roll 12 for X
X to play {5,1}

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

On roll 12, all the efforts that X has made on rolls 8 and 10 are rewarded. Because of the builders on position 16 and 20, he can use the dice numbers 5 and 1 to hit the opponent's blot on position 21 and occupy that point. X now has formed a very strong 5 point prime, a strong HOLDING GAME position.

Roll 14 for X
X to play {3,3}

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

In roll 14 X grabs the chance to make the move, 1-4 4-7 7-10 14-17, which not only runs one of his back checkers out but also hit the opponent's blot on position 10. X now is ahead in the game, and if he can bring all his checkers safely to his inner board and make a 6 points prime, he will have a great chance to win a gammon. After a few unlucky rolls, though, X's blots were hit by the opponent, and were not able to enter the board. From roll 20 to roll 46, the only thing X can do is try to enter his BAR checkers on the board. The opponent starts to bear off his checkers on roll 43, while X still has 3 checkers in the opponent's inner board.

Roll 48 for X
X to play {3,2}

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

At roll 48, X is 106 pip counts behind his opponent. In other words, unless he can do something

special, he will lose the game. The program now starts using the ATTACKING GAME strategy, hoping to hit one of the opponent's blots and close it out. On roll 47, the opponent was forced to leave a blot on position 3. X is lucky enough to roll a 3 on roll 48, so he immediately hits this blot, trying to build a strong inner board prime.

Roll 50 for X
X to play {4,4}

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

This is a critical position for X. If he lets the opponent's blot on position 22 gets away, it is almost certain he will lose the game. The program correctly makes the move. 17-21 17-21 18-22 18-22. He wisely gives up board position 18 to hit the opponent's blot. Now that he occupies 5 of his 6 inner board points, it will very difficult for the opponent's BAR checker to enter the board. After roll 54, X completely closes out his inner board, and although the opponent has only one checker on the BAR, he is not able to make any move. Now it is X's turn to approach victory.

Roll 64 for X
X to play {3,6}

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

By roll 64 X has moved all his checkers to his inner board, so the program starts using the BEAR OFF strategy. Because the opponent still has one checker in his BAR point, the highest priority for X now is to leave no blots in his inner board. Most players on this roll will make the move, 19-off 22-off, leaving the board situation as follows:

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

This position is very danger for X, because he has an isolated checker on position 19. On his next roll, if he rolls a 6, he will be forced to leave a blot on the board, and if this blot is hit by the opponent, he will lose the game. The program smartly make a much safer move, 19-22 19-off. This bears off only one checker, but no matter what dice number he rolls later, all his checkers will be safe.

Roll 66 for X
X to play {2,5}

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

On roll 66, some readers might make the move. 20-off 20-22. It looks safe, but this move will leave the board position as follows:

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

This move bears off one checker, and doesn't leave any blot on X's inner board. On X's next roll, however, if he rolls a {6,1} or {5,1}, he will be forced to left a blot on position 19 or 20. The program detects this potential danger and selects the safest move 19-21 19-24. Although he didn't bear any checkers off the board, he makes sure that there will be no accidents happening to him.

On roll 68, the opponent rolls {6,3}. His BAR checker is forced to leave, and the rest of the game is just a simple race. The program abandons the defensive

BEAR OFF strategy, starting to bear off as many checkers as possible on every roll. On roll 80, X bears off his last checker and the opponent still has 4 checkers on the board. The program wins again.

CHAPTER 7

CONCLUSIONS AND FUTURE EXTENSIONS

7.1. CONCLUSIONS

With our MSLP program, we have tried to implement some of the features of the latest game playing program. We also have shown that a program can have the ability to acquire its own decision making process given the judgment criteria. Finally, we have shown that machines can learn just as human beings.

From the performance of our MSLP backgammon program, we find out that machines not only can do some things that people can do, but also that their performance may be better than that of most people. Our dream is that in the near future, machines can solve those problems that currently cannot be solved by human intelligence. However, as with all things on this side of paradise, a lot of difficult things remain to be done.

In the MSLP program certain extensions and enhancements could greatly increase its performance. Our hope is that at least some of the following enhancements will be implemented at a future time.

7.2. WHEN TO OFFER OR ACCEPT DOUBLE

Doubling is one of the most difficult techniques of backgammon; correct doubling decision alone will give a player an enormous advantage over his opponent. Doubling comes down to deciding whether your position is too good to double, not good enough or just right. The crucial issue is how the position will have changed by the next time you have a chance to double. There are 441 possible situations that must be considered, and it will take a lot machine time to evaluate all these situations.

Doubling in backgammon depends very strongly on timing. If one offers the double too early (your board situation is not strong enough for the double), a few unlucky rolls will force you to lose twice the stake. On the other hand, if you offer the double too late (your board situation is too strong for the double), then your opponent may have the chance to concede the game, and only lose a normal game instead of losing a gammon or a backgammon. This is the reason that when to offer a double is very difficult to decide even for expert backgammon players.

The MSLP program doesn't have the ability to decide when to offer a double or when to accept or refuse a double, however, I considered building two doubling

methods into it.

The first method is this: For each of the five different strategy polynomials, set up an upper bound and a lower bound. Whenever one move is selected by the current strategy polynomial, the program will compare this move's scoring value with the current strategy's higher bound and lower bound. If the program currently owns the double cube, and this scoring value is greater than the higher bound, then the program will offer a double. If the opponent currently owns the double cube, and he offers a double, the program will compare its last move's scoring value with the current strategy's lower bound. If this move's scoring value is lower than the lower bound, then the program will refuse the opponent's double offer and concede the game. If this move's scoring value is higher than the lower bound, then the program will accept the double offer.

The second method of doubling is this: Set up another evaluation polynomial for the double strategy, just like the other five strategy polynomials. This polynomial also has a few terms, each one returing an integer value after evaluating the current board situation. The double polynomial will add all the terms' values together. This polynomial also will have five pairs of higher and lower bounds for each strategy. The

program would compare the double polynomial's current value with the higher and lower bounds to decide when to offer double or when to accept or refuse the opponent's double offer. In both methods, to offering or accepting double most of the time will be done only once, because it is very seldom for a player have the chance to offer or accept double more than once during any game. Whenever the program decides to offer or accept a double, then each pair of those higher or lower bounds' values will be increased, in order to make sure the second double offer or acceptance would not be made too easily.

7.3. MORE STRATEGIES AND MORE TERMS

In this MSLP program, there are five strategies including RUNNING GAME, HOLDING GAME, ATTACKING GAME, BEAR IN and BEAR OFF strategies, and there are also 45 terms that are used in these five strategys' polynomials. However, it is never enough. Whenever new strategies are developed, or new terms found that are very important to the evaluation functions, they also should be considered. As was mentioned in Chapter 3, the MSLP program is easily extensible. Whenever new strategies or new terms are found, they easily can be fit into this program without changing or rewriting very much. The more backgammon knowledge built into this program, the better its performance will be. The MSLP program was

designed for almost unlimited extensions of strategies and terms.

7.4. COMBINE LOOK AHEAD METHOD INTO THIS PROGRAM

A typical chess program investigates a great many legal next moves. The possible continuations usually are implemented in the form of a tree structure. One branch of the tree is followed until the program encounters a reason for terminating the search. The program then applies an evaluation function to the terminal position to arrive at a quantitative value that expresses which player is in a better position and by how much. Such a method is called a look-ahead approach. In chess programs the branches are usually followed to a prearranged maximum depth. The average branching factor for chess is about 35. A fast chess program can search to a depth of six plies in about three minutes which is the time limit in most tourment games. In backgammon, however, there are 21 possible rolls on each move and from zero to over one hundred ways of playing each roll (average are 20 ways of playing each roll). The branching factor, then is more than 400 on each roll, so that an exhaustive search would take a very long time for even a single move. This is the main reason that I didn't use look-ahead in the MSIP program. A modified look-ahead, however, could make a significant

improvement in the program's perfcrmance. The modified lock-ahead method could be as follows: during every turn of the program's move selection, the program selects the five best moves from all the legal next moves, and then perfcrms a search on these branches only. The search could go to a depth of 6 or so plies in the game tree. For each of the 6 plies, the program would select only the best five moves, so the total number of the evaluated moves for any given board position would be 105. This number of moves is solvable by the current system.

Such look-ahead method certainly will cut down the time needed for the conventional brute force searching algorithm, but it also has a trade-off in that it will over-look many good moves. The modified look-ahead method will slow down the program's move selection procedure dramatically (right now it only takes a few seconds to select a move), but the look ahead method will make the program behavior much more like a master backgammon player - always looking ahead more moves than the other players.

7.5. GENERATING STRATEGIES AND TERMS BY PROGRAM ITSELF

CONCLUSIONS AND FUTURE EXTENSIONS

So far computer game programs' consideration factors are built in by the programmers who wrote the program. If we can figure out a learning method so that the program can generates strategies and terms by itself while it is playing against some competitors, then the capability of the program will become greatly enhanced. It would be able to find out all the possible ways for improvements and enhancements, and the games it played, the more stronger it would become, I wonder, if such a day comes, how many human board game champions could retain their positions.

More pages could be written, and more extensions could be done on the MSLP backgammon program to enhance its performance. However, it comes down to time and resources. We hope that all the approaches of Artificial Intelligence that had discussed in this thesis will grow and thrive. We also hope that the decision making and learning techniques used in the MSIP backgammon program some day can be applied to the real world (like plant control, robots, ...). We believe that machine intelligence will become the most valuable resource for the future.

The five different strategies uses a total of forty-five terms. Some terms are used only in one strategy, while others are used in more than one.

APPENDIX A : 5 STRATEGIES AND THEIR TERMS

The RUNNING GAME strategy includes 25 terms. There are "POS20", "POS18", "POS5", "POS21", "POS4", "POS19", "POS22", "POS12", "TOTALPT", "CONTINPT", "OVERLOAD", "BEENHIT", "BACKMEN", "HITOPPONT", "OBLOTOUT", "XBLOTOUT", "INNERBLOT", "OINEAR", "DEADCHECKER", "BUILDERS", "DUPLICATION", "DIVERSE", "XBLOCKED", "OBLOCKED", "MIDDLEMEN". This strategy will be selected when X is far ahead in the game. Its goal is to move X's checkers around the board as quickly and safely as possible. Each of the 25 terms' evaluation functions will give a scoring value to the new board that assumes a move has been made on it. These 25 terms include: "POS20", "POS18", "POS5", "POS21", "POS4", "POS19", "POS22", "POS12", which are some important positions on the board. Whoever occupies (has two or more checkers on) these positions first will have advantages in his future board development. Some of these positions have defensive purposes and some, offensive. The positional terms will return a positive value if the position is occupied by X and a zero otherwise.

"TOTALPT" counts the number of points that X had occupied, and returns that number. "CONTINPT" will count from board position 16 to 24 contiguous how many contiguous points. For example, if X occupies board position 17, 18, 19, 20, then he occupies 4 contiguous

points. If X also occupied the 22 point, X still would have 4 contiguous points because the 22 point is not continous with the other points. The more contiguous points X owned, the more difficult for opponent's to pass by.

"OVERLOAD", checks the board to see how many useless checkers X has (more than 3 checkers on one position). This term will return a negative value. "BACKMEN", will count how many of X's checkers still remain between positions 0 and 7. These back checkers are far away from X's inner board and they need high dice numbers to move them around the board. During the movement, they are very easily hit by the opponent, so this term will return a negative value.

"BEENHIT" figures the probabilities that X's blots will be hit by the opponent. This term can look two moves ahead. It will check that after X makes a move, how many X's blots will left on the board, and the probabilities of all blots that will be hit by the opponent's direct shot (within 6 pips in front of X's blot) and indirect shot (more than 6 pips in front of X's blot). It also counts how much ground will he lose. For example, if the board position 20 has an X checker, after having been hit, it must return to X's BAR point - position 0, and X must rolls 20 more pip counts to let

this checker back to position 20 after been hit, X lose 20 pips. This term can be expressed as:

$$\text{term value} = -\sum_{i=1}^{n} Pi * Ii$$

where n is X's blot numbers. Pi is the probability that a given blot will be hit by the opponent. Ii is the pip counts lost when a blot is hit by the opponent. "HITOPPONENT" is similar to the "BEENHIT" term. It can look 3 moves ahead and calculate the probabilities that different positions will hit the opponent's blot(s). It then finds the best position in front of the opponent's blot(s). If the opponent's next move cannot cover his blot(s), X's checkers will have the highest probability of hitting his opponent's blot(s). BEENHIT and HITOPPONENT enable the program to look a few moves ahead without doing the exhaustive search done by many game programs. This saves considerable memory and time.

"OBLOTOUT" will count that after X made a move, how many of the opponent's blots had been hit by X. Of course, X would try to hit as many the opponent's blots as possible. This term will return a positive value, if X has a prime (owned 5 or 6 continue points) between position 16 and 24, then hitting the opponent's blot(s)

will return a very high positive value. "XBLOTOUT" will count that after X made a move, how many X's blots will be covered or entered some save area of the board. Less blots means fewer chance will X be hit by the opponent. "INNERBLOT" will count how many of the opponent's blot(s) in his inner board (board position 1 to 6). If the opponent has blot(s) in his inner board, X is willing to left some blot(s) between position 13 to 18. The reason is, if X's blot(s) were hit by opponent, on X's next roll, X's BAR checker(s) will have chance to hit the opponent's blot(s) in his inner board (this is called return shot), and the opponent will lost more ground than X. So the opponent will more hesitantly to hit X's blot(s), and X has more chance to use his outer board blots to build some important points in his inner board. This term consider some factors involve psychology. "OINBAR", if the opponent's BAR point has more than one of his checkers, then the opponent must enter his BAR checkers before he can move any checker around the board, so X's blot(s) (except between position 19 and 24) will be safe before the opponent enters all his BAR checkers. In such situation, X should try to set up some builders (even blots) in his outer board, hoping before the opponent enters all his BAR checkers, X could occupy some important points in his inner board. "DEADCHECKER", during the early stage of the game, any checker on board

position 23 or 24 is completely useless, and they are out of the game. One important principle in backgammon is instead of viewing checkers as liabilities to be gotten around the board, it is more useful to consider them as positive assets to used constructively. So the more checkers in position 23 or 24, the more negative value this term will return to the polynomial evaluation function. "BUILDERS", is a term that consider taking some risks but will good for the future board development. In order to occupy some important points of offensive or defensive purposes, X should not wait passively hoping for a lucky numbers. Instead, X must use his checkers actively as building blocks, positioning them to bear on points he wish to make so that the greatest number of dice rolls will be useful. (by bearing on a point, we mean positioning a checker so that it is 6 pips or less away from the point you wish to make, so that one number on one dice can bring the builder to the point). Those important points include board position 21, 20, 18, 5, 4, or the hole(s) in X's prime. "DUPLICATION", the idea of this term is to restrict the number of good rolls the opponent has. For instance, duplicating the numbers your opponent need to hit. The principle is that if X must leave more than one checkers exposed, try to leave them exposed to the same number, thus minimizing the number of ways to be hit. In other words, let the dice numbers

that your opponent needs as few as possible. Some other dice numbers that the opponent needs include: The dice numbers that the opponent needs to enter his BAR checker(s). The dice numbers that the opponent needs to move his trapped checker(s) close to X's 4 or 5 points' prime. The dice numbers that the opponent needs to cover his blot(s). The dice numbers that the opponent needs to complete his prime. The dice numbers that the opponent needs to enter his BAR checkers. "DIVERSE", diversification and duplication are complementary techniques. The opponent wants to duplicate the favorite numbers that X may rolls, and X want to diversify his good numbers, or avoid having them duplicated. X tries to create positions where as many different numbers as possible will be advantageous for X. Those dice numbers that X needs include: The dice numbers that X needs to hit the opponent's blot(s). The dice numbers that X needs to escape from the opponent's broken prime. The dice numbers that X needs to cover his blot(s). the dice numbers that X needs to close to opponent's 4 or 5 points' prime. The dice number that X needs to enter his BAR checkers(s). "XPLOCKED", will check whether the opponent had formeds a prime between board position 1 and 6 ? if the opponent had formed a prime, then all X's checker's that were trapped behind his prime will be very difficult to escape. The more X's checkers were

trapped, the higher negative value this term will return. "OBLOCKED", on the contrarily, will check whether X forms a prime tetween board position 18 to 24 ? if X has a prime, then the more opponent's checker(s) trapped behind this prime, the higher positive value this term will return. "MIDDLEMEN", on the later stage of the game, X's middle point (the board position 12) will be no longer a important landing point for X's back checkers. Those checkers on this point should move as fast and safety as possible to X's inner board points, else they will be very easy been hit by the opponent when they are forced to leave the middle point.

The other two middle game (before X starts bear in and bear off his checkers) strategies are called HOLDING GAME (also called PRIMING GAME) and ATTACKING GAME. The chief objective of HOLDING GAME isfor X to hold a point or points in the opponent's inner or outer board in order to prevent him from safely coming home, and to try to tc trap some of the opponent's checkers behind X's prime.

The chief objective of ATTACKING GAME is: X try to hit and attempt to close out his opponent, usually hitting in X's inner board. This strategy will be used when X is hopelessly behind in the game (at least 3 rolls behind his opponent). The only hope that X has to

win is to try to hit as many of his opponent's blots as he can, even though he will leave some blots, and to try to build X's inner points in order to establish an inner board prime to hopefully close out the opponent's BAR checker(s). Those terms which were used in HOLDING GAME and ATTACKING GAME are the same. Both of them have 26 terms, but the weights of the terms are very different. There are "POS20", "POS18", "POS5", "POS21", "POS4", "POS19", "POS22", "POS12", "TOTAIPT", "CONTINPT", "OVER-LOAD", "CLOSEOUT", "BEENHIT", "HITOPPONT", "OBLOTOUT", "XBLOTOUT", "INNERBLOT", "OINBAR", "DEADCHECKER", "BUILDERS", "DUPLICATION", "DIVERSE", "OBLOCKED", "XBLOCKED", "MIDDLEMEN".

Most of their terms are the same as those terms which were used in RUNNING GAME. These terms include "POS20", "POS18", "POS5", "POS21", "POS4", "POS19", "POS22", "POS12", "TOTAIPT", "CONTINPT", "OVERLOAD", "BEENHIT", "HITOPPONENT", "OBIOTOUT", "XBIOTOUT", "INNERBLOT", "OINBAR", "DEADCHECKER", "BUILDERS", "DUPLICATION", "DIVERSE", "XBIOCKED", "OBLOCKED", "MID-DLEMEN", etc. There are two other terms also used in HOLDING GAME and ATTACKING GAME strategies. One is "CIOSEOUT", it is a term to set up builders on X's outer board in order to establish a prime in X's inner board. It will find out the unoccupied position(s) in X's inner

board, and how many X's builders within 6 pips range in front of these positions. The more builders that X have, the higher positive value this term will return. "HITMEN", in RUNNING GAME, those X's checkers that in the opponent's inner board are considered as very bad factors to X. Because when they move toward X's inner board, they are very easy been hit by the opponent. But in HOLDING GAME and ATTACKING GAME, it is not such case. These checkers become an asset for X, they are like guns thta point to the opponent's head when the opponent moves his checkers toward his inner board, so these X's back checkers should tries to stay at their position as long as possible, and in the right time to hit the opponent's blot(s).

There are two other strategies used in the end game (when X starts to bear in and bear off his checkers) - These are the BEAR IN and BEAR OFF strategies. BEAR IN strategy will be used when most of X's checkers are in his inner board with only a few left outside his inner board. This strategy deals with how to bear X's checkers into his inner board with or without opposition. It is also necessary to distinguish between wanting to save a gammon or win a race while bearing in. There are 9 terms were used in BEAR IN polynomial, they include: "MOVEIN-SIDE", "CROSSOVER", "INNERBOARD", "OUTERDIV", "LASTTWO",

"OBLCTOUT", "INNERPTS", "XINBICT", "CIEARRIGHTMOST". The "MOVEINSIDE", is consider if X has the danger of been gammoned, then X should not waste any pips in his inner board, but move them all in the outer board. In any single rolls, the more individual outer board movement, the more higher positive value this term will return. "CROSSOVER", this term considers to maximize the number of cross-overs (checkers moves from one quadrant to another). Each cross-over should be made as efficiently as possible, in other words, move 1 or 2 pips into the next quadrant, not 6. "INNERBOARD", if X have the danger of been gammoned, X shculd try to bring all of his checkers to the 19 point in order to use his pip numbers as efficient as possible. If it is a simple race, X should evenly spread his checkers between position 19 and 22. "CUTERDIV", this term considers that trying to diverse X's outer board checkers, in order to bear in different positions in X's inner board that will enable X to bear off his checkers more easily. "LASTTWO", when X has only one roll left to bear in all of his checkers, it is not necessary correct to play every pip outside. using dice number to fill up X's inner board gaps will be more important. "OBIOTOUT" is the same term that used in the RUNNING GAME strategy. "INNERPTS", will count how many points that X had occupied in his inner board. The more points that X had

occupied, the higher positive value this term will return. "XINBLOT", will counting how many X's blot(s) are in his inner board, if there is possible contact between these blot(s) and the opponent's checkers, this term will return a very high negative value. "CLEAR-RIGHTMOST", if there is possible contact between X and the opponent's checkers, and X must break one of his point, then clear the rightmost point is the best choice.

The last strategy used in the program is the BEAR OFF strategy. There are 13 terms used in BEAR OFF strategy polynomial. They are 'TAKEOFF", "FILLGAP", "SPREADOUT", "OFFBEENHIT", "SPAREMEN", "DANGERMEN", "CLEARRIGHTMOST", "INNERGAP", "FORCEGO", "OBLOTOUT", "INNERPTS", "EVENMEN", 'XINBLOT". Four of them already have been used in the BEAR IN strategy, "CLEARRIGHT-MOST", "OBLOTOUT", "INNERPTS", "XINBLOT". The other nine terms include: "TAKEOFF", this term will counting the total checkers numbers that X had born off in one roll. Of course, the more checkers that X had born off, the more chance X will win. "FILLGAP", if there is no possible contact between X and the opponent's checkers, and X cannot use all his dice numbers to bear off his checkers, then fills in his inner board gap(s) will also consider a good move. "SPREADOUT", this term will consider

to distribute X's inner board checkers. If there is no possible contact, X should try to distribute his inner board checkers evenly on his inner board, in order that any dice number he rolls can bear some of his checkers off. "OFFBEENHIT", when the opponent still have some checker(s) in X's inner board, then X's inner board blot(s) are very dangerous, especially when opponent has a prime in his inner board. In such case, this term will return a very high negative value. "SPAREMEN", this term will try to avoid having too many X's checkers landing on any one point that from board position 19 to 22. If any one of these positions has 4 or more X's checkers, or from position 23 to 24, any one of these positions has more than 2 X's checkers, then this term will return a negative value. "DANGERMEN", if there is possible contact between X and the opponent, and the rightmost X's inrer board point has an odd number of X's checker(s), then the checkers on this point will be considered very dangerous. Because in the next few rolls, X would have to leave a blot on this point. But one situation can save this disaster, that is if X's second rightmost or third rightmost inner board point also has an odd number of checker(s) on it, then the rightmost single isolated or spare checker will be saved. "INNERGAP", if there is possible contact, then if X have some gap(s) in his inner board will be very dangerous between X's inner

board checkers and his opponent. Because X will be forced to leave some blot(s) in the next few rolls. The large a gap is, the more danger it will be, the lower a gap (near 24 point) is, the more danger it will be. "FORCEGO", if the opponent have only one checker left deeply in X's inner board (on position 23 or 24), and X is ahead in the game, X should try to hit this blot and on the opponent's next roll, if he rolls a big number, he will be forced to move this checker to some positions not so deep in X's inner board. X will be much safer to bear off the rest of his checkers. "EVENMEN", when X needs less than 25 pip numbers to bear off his remaining checkers, X should try to keep even checker number on the board. By this way, X can save an extra move to bear off a single last checker off the board and waste half roll of the last roll.

7.6. APPROXIMATE DECISION MAKING TIME

The times needed for the program to select a move will depending on the number of all the next legal moves. Normally, rolling doubles will need more time than rolling two different dice numbers and small dice numbers will need more time than large dice numbers. From my experience, decision making never took more than 5 seconds and usually only about 1 or 2 seconds. Compare this with human competitors' speed which is usually

about 20 seconds. We must admit, then, that the machine's thinking speed is considerably faster than the human's thinking speed.

APPENDIX P : GAME EXAMPLE ONE

Opponent rolls 5, X rolls 6, X moves first.

ROLL	TERM	DICE NUMBERS	MOVE
1	X	{6,5}	1-7 7-12
2	O	{6,1}	13-7 8-7
3	X	{3,1}	17-20 19-20
4	O	{5,4}	13-8 13-9
5	X	{5,5}	12-17 12-17 17-22 17-22
6	O	{5,2}	9-4 6-4
7	X	{6,2}	1-3 3-9
8	O	{4,2}	13-9 13-11
9	X	{4,3}	0-3 12-16
10	O	{4,1}	9-5 6-5
11	X	{6,4}	3-9 9-13
12	O	{6,6}	24-18 24-18 8-2 8-2
13	X	{5,3}	12-17 13-16
14	O	{5,4}	8-3 7-3
15	X	{4,4}	12-16 12-16 16-20 16-20
16	O	{3,1}	11-8 7-6
17	X	{5,4}	16-21 16-20
18	O	{6,4}	18-12 18-14
19	X	{5,4}	17-22 17-21
20	O	{4,3}	14-10 8-5
21	X	{4,1}	17-21 22-23
22	O	{5,3}	10-5 12-9
23	X	{5,4}	20-off 21-off
24	O	{6,2}	9-3 2-off
25	X	{5,5}	20-off 20-off 20-off 20-off
26	O	{6,5}	6-off 5-off
27	X	{6,1}	19-off 19-20
28	O	{3,3}	3-off 3-off 3-off 6-3
29	X	{4,3}	21-off 22-off
30	O	{5,4}	5-off 4-off
31	X	{6,3}	19-off 22-off
32	O	{5,2}	5-off 2-off
33	X	{6,2}	19-off 23-off
34	O	{5,4}	5-off 4-off
35	X	{6,1}	20-off 21-22
36	O	{5,4}	6-1 6-2
37	X	{3,2}	22-off

PROGRAM WINS THE GAME

APPENDIX A: GAME EXAMPLE TWO

Opponent rolls 2, X rolls 1, opponent moves first.

roll	turn	dice numbers	move
1	O	{2,1}	8-6 6-5
2	X	{2,3}	12-14 12-15
3	O	{6,5}	24-18 18-13
4	X	{6,1}	14-20 19-20
5	O	{6,1}	13-7 6-5
6	X	{1,1}	15-16 16-17 17-18 17-18
7	O	{4,3}	7-3 6-3
8	X	{4,5}	12-16 12-17
9	O	{3,3}	13-10 13-10 13-10 6-3
10	X	{2,1}	12-14 19-20
11	O	{3,4}	24-21 10-6
12	X	{5,1}	16-21 20-21
13	O	{2,2}	25-23 13-11 13-11 10-8
14	X	{3,3}	1-4 4-7 7-10 14-17
15	O	{5,6}	opponent concedes this move
16	X	{4,1}	19-23 23-24
17	O	{6,3}	25-22
18	X	{1,2}	21-22 22-24
19	O	{3,4}	25-22 25-21
20	X	{3,6}	X concedes this move
21	O	{3,4}	8-5 5-1
22	X	{3,5}	X concedes this move
23	O	{1,6}	8-7 7-1
24	X	{1,5}	X concedes this move
25	O	{5,4}	21-16 16-12
26	X	{2,5}	0-2
27	O	{3,6}	11-8 11-5
28	X	{6,6}	X concedes this move
29	O	{1,1}	5-4 4-3 3-2 3-2
30	X	{1,4}	0-4
31	O	{3,3}	12-9 9-6 8-5 8-5
32	X	{2,5}	X concedes this move
33	O	{1,3}	5-4 4-1
34	X	{4,6}	0-4
35	O	{1,6}	22-16 16-15
36	X	{3,3}	X concedes this move
37	O	{1,3}	5-4 4-1
38	X	{1,6}	X concedes this move
39	O	{4,1}	15-11 11-10
40	X	{4,1}	0-4

APPENDIX A: GAME EXAMPLE TWO

41	O	{5,2}	10-5 5-3
42	X	{3,6}	X concedes this move
43	O	{5,2}	5-off 5-3
44	X	{1,3}	X concedes this move
45	O	{2,1}	3-1 1-off
46	X	{2,4}	0-4
47	O	{2,2}	2-0 2-0 3-1 3-1
48	X	{3,2}	0-3 3-5
49	O	{3,5}	25-22 6-1
50	X	{4,4}	17-21 17-21 18-22 18-22
51	O	{2,4}	25-23 6-2
52	X	{2,6}	5-7 17-23
53	O	{5,6}	opponent concedes this move
54	X	{6,6}	4-10 4-10 10-16 17-23
55	O		opponent concedes this move
56	X	{1,2}	16-17 17-19
57	O		opponent concedes this move
58	X	{1,5}	7-8 8-13
59	O		opponent concedes this move
60	X	{4,5}	10-14 14-19
61	O		opponent concedes this move
62	X	{6,3}	13-19 19-22
63	O		opponent concedes this move
64	X	{6,3}	19-22 19-off
65	O		opponent concedes this move
66	X	{2,5}	19-21 19-24
67	O	{3,4}	opponent concedes this move
68	X	{1,5}	20-21 20-off
69	O	{6,3}	25-19 19-16
70	X	{1,4}	24-off 21-off
71	O	{1,5}	16-15 15-10
72	X	{3,4}	22-off 21-off
73	O	{5,6}	10-5 6-off
74	X	{3,4}	22-off 21-off
75	O	{6,4}	6-off 5-1
76	X	{5,5}	21-off 22-off 22-off 23-off
77	O	{3,5}	2-off 1-off
78	X	{2,5}	23-off 24-off
79	O	{5,6}	1-off 1-off
80	X	{3,4}	24-off

Program wins the game

BIBLIOGRAPHY

[NE 80] Banerji, Ranan B.,
Artificial Intelligence: a Theoretical Approach.
North Holland, New York N.Y. 1980.

[RR 81] Barr, Avron & Feigenbaum, Edward A.,
The Handbook of Artificial Intelligence,
Volume 1, William Kaufmann, Inc., Los Altos CA, 1981.

[LI 78] Bellman, Richard,
An Introduction to Artificial Intelligence: Can Computers Think ?,
Boyd & Fraser Publishing Company, San Francisco CA, 1978.

[RI 80] Berliner, Hans, "Computer Backgammon," Scientific American, June, 1980

[E 82] Cohen, Paul R. & Feigenbaum, Edward A.,
The Handbook of Artificial Intelligence,
Volume 3, William Kaufmann, Inc., Los Altos CA, 1982.

[GG81] Feigenbaum, Edward A. & Feldman, Julian,
Computer and Thought,
Robert E. Krieger Publishing Co., Malabar FL, 1981.

[NT 75] Hunt, Earl B.,
Artificial Intelligence,
Academic Press. Inc., New York N.Y., 1975.

[RN 78] Kernighan, Brian W. and Ritchie, Dennis M.,
The C Programming Language,
Prentice Hall,
Englewood Cliffs N.J., 1978.

[IX 79] Klix, Friedhart,
Human and Artificial Intelligence,
North Holland Publishing Co.,
Amsterdam N.Y., 1979.

[GR 76] Magriel, Paul,
Backgammon,
Times Book Co., New York N.Y., 1976.

[LSS80] Nilsson, Nils J.
Principle of Artificial Intelligence
Tioga Publishing Co.,
Palo Alto CA, 1980.

[OL 69] Obolensky, Prince Alexis and James, Ted,
Backgammon: The Action Game
Macmillan Publishing Co.,
New York N.Y., 1969.

[RC 78] Ritchie, Dennis M., "The C programming Language",
The Bell System Technical Journal,
Volume 57, No 6, Part 2.

[SAMU 60] Samuel, A. L., "Programming Computer to Play Games",
In Alt, Franz L. (ed.), Advances in Computers 1, Academic Press, New Y
N.Y. 1960.

[WILK 81] Wilkins, David, "Using Patterns and Plans in Chess", SRI International.
Reprinted in Webber, Bonnie Lynn and Nilsson, Nils J. (eds.),
Readings in Artificial Intelligence,
Tioga Publishing Co.,
Palo Alto CA, 1981.

[WINS 77] Winston, Patrick Henry,
Artificial Intelligence,
Addison Wesley Co. Menlo Park CA, 1977.