1983

Computer game playing: the game of Twixt

James R. Huber

Rochester Institute of Technology
School of Computer Science and Technology

Computer Game Playing:
The Game of Twixt

A Thesis submitted in partial fulfillment of a
Master of Science in Computer Science Degree Program

By: James R. Huber

Approved by:

John A. Biles: Advisor

Margaret M. Reek

Warren R. Carithers

Date: 3/10/1983

Computer Game Playing:

The Game of Twixt

I, James R. Huber, hereby grant permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

_____
Signature

10 MARCH 83
Date

TABLE OF CONTENTS

1 Introduction 1
1.1 Overview 1
1.2 Artificial Intelligence Literature 2
1.3 History of Computer Game Playing 2
1.4 Similarities in Game Playing Programs 4
1.5 The Game of Twixt 5
1.6 Outline of Paper 7
2 Basic Program 8
2.1 Overview 8
2.2 Program Structure 9
2.3 User Interface 10
2.4 Data Structures 11
2.4.1 Game Board 11
2.4.2 List of Moves 17
3 Move Evaluation 18
3.1 Overview 18
3.2 Move Evaluation Function 19
3.3 Game Strategies 20
4 Learning 26
4.1 Overview 26
4.2 Scoring of Moves 26
4.3 Learning Modes 29
4.3.1 Absolute Mode 29
4.3.2 Relative Mode 31
4.3.3 Percentage Mode 32
4.4 Remembering 33
5 Results 34
5.1 Overview 34
5.2 First Efforts 34
5.3 Full Implementation 37
5.4 Effectiveness of Strategies 41
5.5 Conclusions 45
6 Future Enhancements 47
6.1 Overview 47
6.2 Better Strategies 47
6.3 Multiple Evaluation Functions 48
6.4 Defense vs. Offense 51
6.5 Look-Ahead 52
Appendix A 56
References 63

LIST OF FIGURES

2.1 Program Structure 9
2.2 Possible Links 12
2.3 Area Covered by BARRIER 13
2.4 Values of BARRIER's Sides 14
2.5 BARRIER Blocking Links 15
3.1 Beam 23
3.2 Tilt 23
3.3 Coign 24
3.4 Mesh 24
4.1 Adjustments to Weights 27
4.2 Absolute Mode Groups 29
5.1 Weights After Round Four 41
5.2 Weights After Round Seven 43

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Artificial Intelligence is a field of study in which researchers try to program computers to do certain things which, if they were done by human beings, would be said to exhibit intelligence; in other words, they try to make machines act intelligently. There are generally three reasons to study machine intelligence: to help us better understand the processes of human intelligence, to be able to replace humans in menial or monotonous tasks which nevertheless require some degree of intelligence, and to be able to develop machine intelligence to such an extent that it will be able to solve problems which are currently beyond the grasp of human intelligence.

An important area of Artificial Intelligence is computer game playing. By programming computers to play games, it is possible to study decision making (how the computer decides which move to make) and learning (how the computer can improve its play). Game playing is especially useful because it offers a controlled environment which is much simpler to deal with than the relative chaos of the real world. Yet at the same time, games have simi-

larities to real problems that eventually should enable researchers to apply what is learned through game playing to many other areas. The purpose of this thesis is to investigate several aspects of computer game playing by programming a computer to play the game of Twixt.

1.2 ARTIFICIAL INTELLIGENCE LITERATURE

Good general introductions to Artificial Intelligence can be found in Winston [WINS 77] and Nilsson [NILS 80]. These include discussions of many different areas of Artificial Intelligence, such as pattern recognition, game playing, generalized learning, etc. Slagle discusses the problems of writing programs in Artificial Intelligence [SLAG 71]. He includes several chapters describing programs for specific games and discusses the various problems that arise from trying to program different types of games. Samuel offers a historical survey of early attempts at computer game playing [SAMU 60]. He concentrates for the most part on the games of chess and checkers.

1.3 HISTORY OF COMPUTER GAME PLAYING

The earliest modern computer games date from the 1950s. The classic game playing program was the checker player developed by Samuel [SAMU 63]. His program played checkers against human opponents and learned to improve

its play through practice. Samuel tested two different learning algorithms, one of which was very similar to the algorithm used in this thesis. Ultimately, Samuel's program was able to play a nearly expert game of checkers.

Weizenbaum has written a program to play a game called five-in-a-row [WEIZ 62a]. His move evaluation function (by which he chooses which move to make) is virtually identical to the Twixt function described later in this paper. The most important difference between my program and the five-in-a-row program is that Weizenbaum's does not learn.

A more recent example of machine intelligence is a backgammon program called BKG 9.8, which was developed by Berliner [BERL 80b]. BKG 9.8 achieved a measure of fame in July of 1979 by defeating the world backgammon champion and winning $5,000. This was the first time a computer program had ever beaten the world champion at any board game. Although backgammon relies to a great extent on luck, this is by no means an insignificant achievement. The BKG 9.8 and Twixt move evaluation functions are similar in their basic elements, but Berliner has included some significant modifications in his, which make it much more effective. These modifications will be discussed in chapter six.

1.4 SIMILARITIES IN GAME PLAYING PROGRAMS

Most game-playing programs are made up of essentially similar components which are tailored to the specific needs of the game. For example, many such programs contain move evaluation functions in the form of either linear or non-linear polynomials. These functions show the relative values of different moves and enable the program to perform the best move in any given turn. The quality of the move evaluation function is directly related to the quality of the program; the moves made by a program can only be as good as its move evaluation function.

Another common feature in game-playing programs is a look-ahead algorithm. Using look-ahead, the program does not stop after evaluating the moves it can make. For every possible move by the program, it also evaluates each possible responding move by its opponent. Depending on the implementation of the algorithm, the program can, for example, look ahead three or four pairs of moves, or perhaps until it reaches a certain pre-determined position in the game.

A look-ahead algorithm corresponds to searching down an N-ary game tree, where N is the number of possible moves for each turn. Although look-ahead can be very useful, for large values of N it can also be very expensive.

In the game of Twixt, there are well over 500 possible moves every turn. Because of the prohibitive costs of the look-ahead algorithm, the Twixt program does not use it. The possibility of using modified and less expensive versions of look-ahead will be discussed in a later chapter.

Many programs are able to improve their play by taking into account their past performance. Learning usually takes place after each game is completed. For example, if the program wins the game, it "remembers" the strategies it used so that it can use them to win again. If the program loses, it knows which strategies are probably bad and it would later avoid using these if possible.

The program discussed in this thesis does not learn after each game, but rather after every move. While it is learning, its opponent acts as its teacher. After every move the teacher rates the move and the program adjusts its move evaluation criteria accordingly. Potentially, this should allow the program to learn faster than programs which use the post-game learning algorithm, but the final results should be quite similar.

1.5 THE GAME OF TWIXT

Twixt was invented by Alexander Randolph, a Czechoslovakian game enthusiast who has several successful games on the U.S. market [GARD 79]. It was marketed and

copyrighted in 1962 by the Minnesota Mining and Manufacturing Company and is still widely available to the public.

Twixt is a two-player confrontational game. It is played on a square peg-board made up of 24 rows and 24 columns of holes. The top and bottom rows on the board serve as one player's home rows; the right- and left-most columns are his opponent's home rows. Playing alternately, each player may place a single peg in any unoccupied hole on the board, except in his opponent's home rows. By placing pegs a certain distance apart, he can also place a link to connect the two pegs. These links can be placed only on pegs which are two rows and one column apart or one row and two columns apart (a knight's move in chess). The object of the game is for a player to build a continuous chain of linked pegs from one of his home rows to the other. The chain that one player builds between his home rows also acts as a barrier to prevent his opponent from completing his chain, so the game requires a good mixture of offensive and defensive positioning.

The program described in this thesis acts as one of the players; it plays against a human opponent. They communicate through a CRT; the human player enters his moves via the keyboard and the program prints its moves on the

screen. Since the program does not display the current state of the game board, the human player must keep track of the game on his own board. After every move the program updates its internal version of the board and checks to see if either player has won. When one player wins, processing terminates.

1.6 OUTLINE OF PAPER

The remainder of this paper is organized as follows: Chapter two describes the basic program. It describes the structure of the program, how the game board is maintained and the background functions performed. (An example of a background function is the storing of a list of all the plays made during the game.) The third chapter describes the decision-making algorithm that the program uses to select the best possible move. The next chapter deals with the program's learning algorithm. It describes how the program alters its decision-making process by taking into account its past successes and failures. Chapter five discusses the results of the program's learning and some of the conclusions reached. The last chapter lists several improvements which could be made in the program and discusses some of their consequences.

CHAPTER 2

BASIC PROGRAM

2.1 OVERVIEW

The program was written in the C programming language [KERN 78] and is run on a PDP-11/70 computer. Its code is stored in six source files, each of which acts as a conceptual unit. The files were kept separate to facilitate editing and re-compiling during the debugging stages. The program is made up of 52 separate functions and, not counting documentation, contains slightly more than 2000 lines of code.

The actual program is made up of three parts. The basic program deals with maintaining the game-board, verifying and executing moves entered by the human player, testing to see if anyone has won and other such bookkeeping functions. The second part includes the code which generates the program's moves. Part three contains the learning algorithm. This thesis is concerned with the last two sections. The basic program serves only as a necessary foundation for the move generation and learning functions.

2.2 PROGRAM STRUCTURE

The basic program was coded first; only after it was completely tested were the other two parts included. The structure of the program, shown in Figure 2.1, is very simple. Before each move, the program checks to see if anyone has won. If there is a winner, the program indicates who the winner is and processing terminates. If the game is not over, the program reads the next move from the player whose turn it is. A series of checks is performed to verify that the move is valid. If the move is illegal, the player must continue entering moves until he enters a valid one. When the program has determined that it has received a legal move, the move is executed. This simply involves updating the game board to reflect the new state of the game. This cycle continues until one of the players wins or concedes.

```
While game is not over
     Get next move
     Check validity of move
     If move is illegal
        then return to Get next move
     Execute move
Endwhile
```

PROGRAM STRUCTURE

Figure 2.1.

2.3 USER INTERFACE

Before each game starts, the program communicates with the user through a series of questions. The user can, if he wishes, review the rules for Twixt or the rules for entering moves. The user must also indicate which color he would like to play and which of the three learning modes is to be used.

During the game the program prints either "your move..." and waits for the human player's move, or "my move..." followed by its move. When it is the human player's turn, he must enter either a legal move or a command. A legal move has three parts. The second two parts are optional, but if both are included, they must be in the proper order. The first part is the board location where the player wants to place his peg. This is indicated by entering a lower-case letter from a to x for the row and a number from 1 to 24 for the column (e.g. m8). This may be followed by the endpoints of any single link which is to be removed from the board. These endpoints must be within brackets (e.g. m8[c12d14]). The third part indicates the endpoints in parentheses of any links which are to be placed. Up to ten links can be placed in a single move, each of which must be entered within its own set of parentheses. If the peg placed during the move is one of the endpoints, it need not be specified again in the

parentheses. For example, m8(n10)(o9)(f5g7) means to place a peg at location m8, place two links going from m8 to n10 and o9 and place a link from f5 to g7.

Instead of a move, the player could enter one of three commands in the form of a single upper-case letter. A 'C' is entered to concede the game. Entering an 'L' will cause a list of all the moves made so far to be printed on the screen. Finally, an 'R' can be entered to review the rules for entering moves. If one of the last two commands is entered, it is, of course, still that player's turn, so he must then enter a regular move.

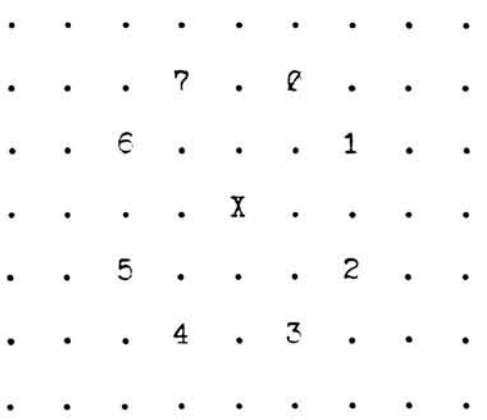
2.4 DATA STRUCTURES

2.4.1 GAME BOARD

The most important data structure is the game-board. Since the actual game-board forms a square matrix, the obvious choice for a data structure was a two-dimensional array. However, the program is only concerned with board locations that already contain pegs. The game-board would actually be a very sparse matrix that could be stored in much less space using a linked list representation. After comparing the relative benefits of a two-dimensional array and a linked list structure, it was ultimately decided that the simplicity of the two-dimensional array more than compensated for the extra space it used.

Each element of the game-board array is a structure made up of three elements: PEG, LINK, and BARRIER. These elements reflect the presence or absence of pegs and links at each location on the board. They are all initialized to zero, indicating that there have been no pegs or links placed yet. During the game, PEG has a value of 1 for every peg-hole with a red peg in it; for those holes containing black pegs, PEG has a value of -1. Player RED is initialized to be 1 and BLACK is initialized as -1.

The variable LINK shows the directions of any links placed on a peg. If any location's PEG has a value of zero, then its LINK must also equal zero since a link

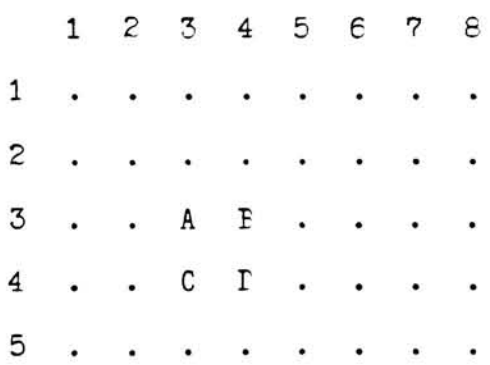


A peg at location X can link to the eight numbered locations.

POSSIBLE LINKS

Figure 2.2.

cannot be placed unless its supporting peg is already present. LINK is an 8-bit character variable, with one bit for each of the eight possible directions any peg's links can point. The possible links are numbered from zero to seven moving clockwise from the upper-right as shown in Figure 2.2. If a particular link is present, then the bit in the variable LINK corresponding to that link is turned on. The rightmost bit is bit zero. For example, if the peg at location X in Figure 2.2 holds links extending to locations 1 and 3, then LINK for location X has a value of 10, since bits 1 and 3 are on (i.e. '00001010'b = 10).

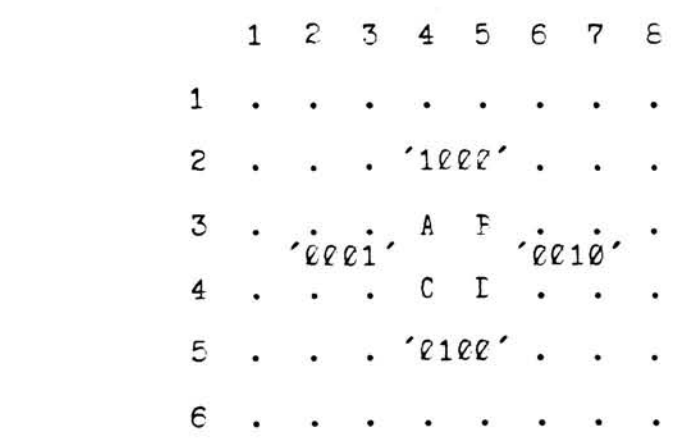


BARRIER at location 3,3 refers to the area bounded by holes A, B, D and C.

AREA COVERED BY BARRIER

Figure 2.3.

LINK deals with the chain a player is building to connect his home rows. BARRIER deals with the same chain but views it as a barrier a player is building to block his opponent. Unlike PEG and LINK, BARRIER refers to an area rather than a peg-hole. The variable BARRIER, at any particular peg-hole on the board, refers to the square area to the lower-right of that peg-hole. The sides of this square extend only to the next peg-holes to the right and below the original hole. (See Figure 2.3.) The value of BARRIER represents the links that are passing through this area. Any link passing through a BARRIER's area is thought of as covering one triangular half of the square.

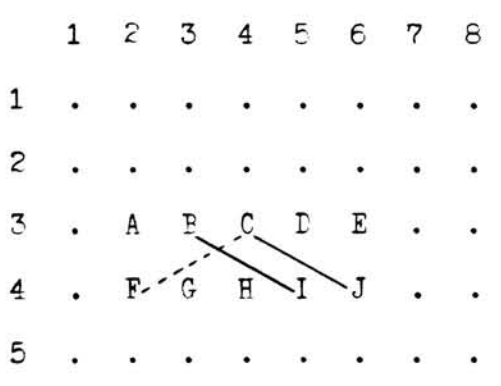


Each side is represented by a number
that has one of its four low-order bits on.

VALUES OF BARRIER'S SIDES

Figure 2.4.

There are four possible triangles covered (i.e. upper-right, upper-left, lower-right and lower-left). Each of the four sides of the square is identified by a binary number which has one of its four low-order bits turned on. (See Figure 2.4.) For example, 'up' (side A-B in Figure 2.4) has a value of 8 ('1000'b) and 'right' (side B-D) has a value of 2 ('0010'b). To represent a barrier (link) covering the upper-right triangle of the area, BARRIER equals the bitwise OR of 'up' and 'right' - that is, 10('1010'b). Likewise, a link covering the lower-left triangle of the square would have a value equal to the bitwise OR of 'left' ('0001'b) and 'down' ('0100'b) - lower-left has a value of 5 ('0101'b).



With link B-I in place,
link C-J is legal and
link C-F is illegal.

BARRIER BLOCKING LINKS

Figure 2.5.

In Figure 2.5, if a link is placed connecting holes B and I, then it covers the upper-right of square B-C-H-G and the lower-left of square C-D-I-H. In this case it would still be possible to place a link from hole C to hole J. Part of this new link would cross square C-D-I-H for which BARRIER already has a value of 5 ('0101'b), meaning a link is crossing it in the lower-left triangle. The new link (C to J) crosses the square in the upper-right. Its value is 10 ('1010'b). Since these bits in BARRIER are still off, this shows that the new link is legal. If this link is placed, its value is ORed with the current value of BARRIER resulting in a new value of 15 ('1111'b). All the bits are now on, which shows that the entire square area is covered. It would be impossible to place another link which passed anywhere through this BARRIER's area.

With the link from B to I in Figure 2.5 already in place, assume that an attempt is made to place a link from C to F. In this case BARRIER in area B-C-H-G already has a value of 10 ('1010'b). The new link would have a value of 9 ('1001'b) in area F-C-H-G. The high order bit in BARRIER is already on, which signifies that the upper portion of the square is already covered. The new link is also trying to pass through this part of the square. Since the upper side is already covered or the 'up' bit is already on, the new link from C to F is clearly illegal

and this move would be rejected.

During each move, the program checks the values of these variables on the board in order to verify the validity of the move. If any illegality is detected (e.g. trying to place a red link on a black peg) an appropriate error message is printed and the player is asked to enter a different move. After every move, these variables are updated at the appropriate locations of the board to reflect the new state of the board.

2.4.2 LIST OF MOVES

The second important data structure stores the moves made during the game. A long one-dimensional character array holds the concatenation of all the moves. For example, if a move is represented in seven characters, it will fill locations zero to six. The other player's move might fill locations seven to twelve and then the first player's next move would start at location thirteen. A pointer points into this array to the next free space (i.e. where the next move will start). Each player has an array of pointers that indicate the first position in the long array of every move that player has made so far. These arrays are used to store and subsequently print a list of all the moves.

CHAPTER 3

MOVE EVALUATION

3.1 OVERVIEW

This chapter deals with the methods the program uses to evaluate moves and to decide which one to play.

When it is the program's turn to make a move, control passes to the function MAKEMOVE. The actions performed depend on which of the following four cases is found: (1) it is the first move of the game; (2) it is the second move of the game; (3) the array holding the concatenation of all the moves or the array of pointers indicating the computer's moves is about to overflow; (4) none of the above. In the first case, the program calls a function which simply places a peg at location m8. This is an excellent place to start play, and for the sake of simplicity the program always moves here when it plays first. The second case means that the program is playing second and that its opponent has already moved. A function is called which causes a peg to be placed approximately two-thirds of the distance from the opponent's peg to the opponent's most distant home row. The program is positioning itself to block its opponent. In the third case, the program catches a run-time error before it happens; it

determines that an array is full and that if appropriate action is not taken an attempt will be made to index the array beyond its upper limit. Therefore, the program concedes the game at this point. This should happen only if the game is extremely long. The fourth case is, of course, the one that occurs most of the time, and the program's actions for this case are discussed in the rest of this chapter.

For the fourth case, MAKEMOVE calls the function FINDMOVE, which executes a search for the best move to make. The function looks at every possible move on the board - in other words, it looks at the consequences of placing a peg in any and every unoccupied hole, except those in its opponent's home rows. It assigns a value to each of these moves; the higher the value, the better the move. The coordinates of the move with the highest value are returned to MAKEMOVE, where the actual execution of the move takes place.

3.2 MOVE EVALUATION FUNCTION

The value of each move is calculated by means of a linear polynomial which has the following form:

$$\text{value} = \sum_{i=1}^{n} t_i w_i .$$

Each t_i is a term which acts as a simple, well-defined game strategy. Each term is implemented as a function

which returns a normalized value between zero and 25. The value of a term is proportional to the quality of the move, as perceived by that single strategy. For example, assume a strategy indicates that it is better to place pegs near the center of the board. For central moves this term will have a value close to 25; for moves which are a greater distance from the center, the term's value will be proportionally less than 25.

Each term or strategy has a corresponding weight (w_i), which indicates the importance of this strategy relative to all the others. The values of these weights are determined by the learning algorithm and will be discussed in the next chapter. The value of each term (how much this move conforms to this strategy) is multiplied by its associated weight (how important this strategy is in playing the game). The resulting products are added together to indicate the overall quality of the move.

3.3 GAME STRATEGIES

The following list describes the seventeen individual game strategies used in the Twixt move evaluation polynomial:

1. RAND This term assigns a random value to each move. It is used in the learning algorithm and will be discussed in the next chapter.

2. RAND This is identical to the previous term.

3. CENTHORZ This strategy gives high values for moves near the horizontal center of the board (from each player's perspective). Red scores high for moves along rows L and M; black scores high for moves along columns 12 and 13. As the moves approach the player's home rows, the value of this term diminishes.

4. CENTVERT High values are assigned for moves along the vertical center. Moves closer to the opponent's home rows receive lower values.

5. MAKEBRIDGE The value of this term increases with the number of bridges placed during this move.

6. SPAN This term's value is equal to the number of rows spanned by the chain to which this move is connected. If the move is a single isolated peg (not linked to any chain), then this term's value is zero. This term is the only one that can have a non-normalized value. If the chain spans 23 rows, it means that this move will result in a win, since a chain covering 23 rows must connect the two home rows. If this is the case, the term receives a value of 1000, which is intended to outweigh the values of all the other terms to ensure that this (winning)

move will be made.

7. COUNT The value of this term is equal to the number of links that are on the chain to which this move is connected. It has a maximum value of 25.

8. ADVANCE This term calculates the number of rows that the move increases the total span of any chain to which the move links. The term's value is proportional to this number of rows.

9. DEFBLOCK This strategy looks at the opponent's most important chain (as determined by its size and whether the opponent added a link to it in his previous move). It calculates the best position to set up a defensive block in front of this chain. The closer the move is to this blocking position, the higher its value.

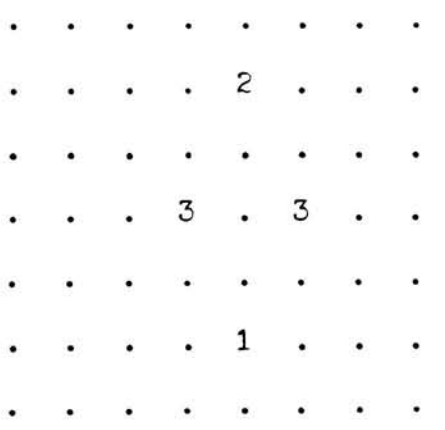
10. TOUCH The value of this term is high when the move is close to the opponent's last move. The farther away the move is, the lower the term's value.

11. DENY This term calculates how many links the opponent could have placed before this move but now cannot place as a result of this move. Its value is equal to either three times this number or 25, whichever is less.

12. BLOCK This strategy determines which areas of the board are already blocked by the opponent and are, therefore, not good areas into which to move. The closer the move is to one of these areas, the lower the value of the term.

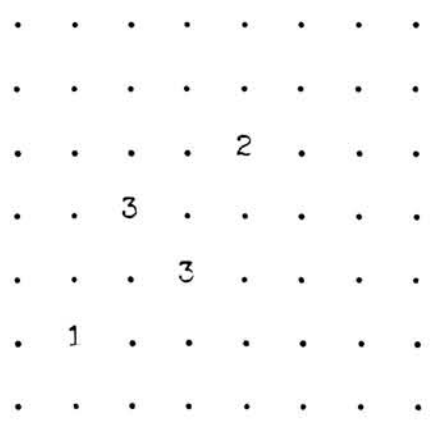
13. EDGEHORZ The value of this term is high when the move is near one of the player's home rows. As the move approaches the center of the board, the term's value decreases. This term is the exact opposite of the CENTHORZ term.

14. BEAM This term and the next three terms involve strategies called set-ups. During a set-up, two pegs are placed a certain distance apart. This makes it



BEAM

Figure 3.1

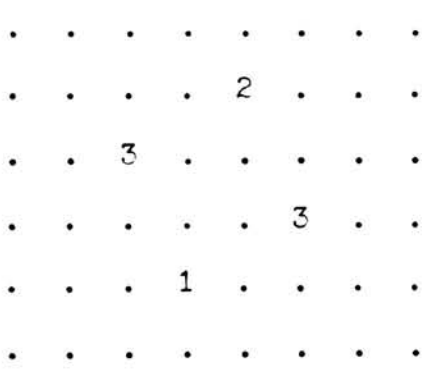


TILT

Figure 3.2

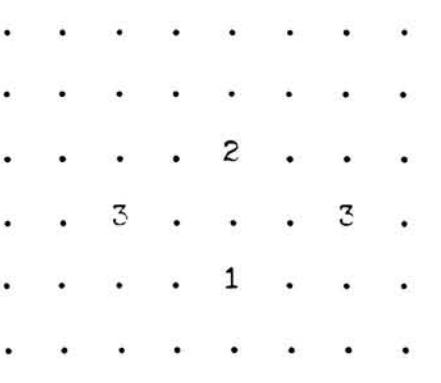
possible to place a third peg in either of two different holes in order to connect the first two pegs with a double-link. If a set-up is made, these terms have a value of nineteen. If not, they have a value of nine. See Figure 3.1 for the position and order of placement of pegs for a beam set-up. The first peg is placed at the location labeled 1 and the second at location 2. The third peg then can be placed in either of the locations labeled 3, to make it possible to link all three pegs.

15. TILT This is another type of set-up. See Figure 3.2 for the peg placement. This term's values are also nineteen if the set-up is made and nine if it is not made.



COIGN

Figure 3.3



MESH

Figure 3.4

16. COIGN The coign set-up is illustrated in Figure 3.3. It has the same values as BEAM and TILT.

17. MESH The mesh set-up is also illustrated in Figure 3.4. Its values are the same as the other set-ups.

CHAPTER 4

LEARNING

4.1 OVERVIEW

This chapter describes the learning algorithm. It discusses how the program uses its past performance to modify its move evaluation process described in the previous chapter.

The move evaluation polynomial is made up of seventeen pairs (t_i, w_i). The value of each t_i represents the quality of that move according to that single strategy. The w_i values show the quality of that strategy, relative to the others, in playing the game. At the beginning of the learning process all the weights (w_i) in the polynomial are set to 100. That they are all equal shows that no strategy is better or worse than any other strategy. The learning algorithm determines which strategies are good and which are bad and then raises or lowers their weights accordingly.

4.2 SCORING OF MOVES

While the program is learning, the human player acts as its teacher. After each move made by the program, the teacher enters a score from one to five (to indicate the

range from very bad to very good moves). This score is compared to the term values that were used in the polynomial when the program evaluated this move and determined that it was the best one. Some of these terms probably indicate (with high values) that the move was good and some probably indicate (with low values) that the move was bad. If the value of the term agrees with the teacher's score (e.g. if they both indicate that the move was good) then the weight associated with that term is increased. If the term value and the score disagree, the weight is decreased. See Figure 4.1. For example, assume the teacher decides a move is very good. (He gives it a score of five.) If term A gave that move a high value, it

		TERM: good	TERM: bad
TEACHER	good	up	down
	bad	down	up

This table shows the direction of the strategy's weight (increased or decreased) when the value of the term is compared to the teacher's score.

ADJUSTMENTS TO WEIGHTS

Figure 4.1.

correctly predicted the quality of the move, so it should be worth more in relation to the other terms. Its weight is adjusted upward. If term P gave the move a low value, it was incorrect in its evaluation. The next time moves are evaluated it should not carry as much weight. Therefore, its weight is lowered. Through this process the good terms become more important and the bad ones less important; the program learns which strategies are best and thereby improves its play.

To test the success of the learning algorithm, the first two strategies in the polynomial (both RANDs - random number generators) are used as control terms. The adjustments to their weights should be a good indication of the program's ability to learn. Surely, assigning a random number to a move to indicate its quality is not a good strategy for Twixt or any other game. Since every term's weight starts at 100, the program originally considers these strategies to be as valid or as useful as all the others. By observing the weights of the control terms, the teacher can see whether or how quickly the program learns that they are bad strategies. In other words, it is expected that the weights of these terms should approach zero; the degree to which this occurs should be proportional to the quality of the learning algorithm.

4.3 LEARNING MODES

There are three modes of learning: absolute, relative and percentage. The modes have different ways of interpreting the term values to show the quality of the moves. For example, assume a term evaluates to 21. In the absolute mode this would always be considered a very good score. Using the relative mode, this value is considered relative to the values of the other strategies, so that if most of the terms have values above 21, this would be considered a bad score.

4.3.1 ABSOLUTE MODE

The absolute mode places each strategy into one of five groups, depending on the value of the term. Figure 4.2 shows the mapping of term values to groups. The group numbers indicate the quality of the move, from a range of

Term Value	Group Number
0-4	1
5-9	2
10-14	3
15-19	4
20-25	5

ABSOLUTE MODE GROUPS

Figure 4.2.

very bad (group 1) to very good (group 5). This interpretation is identical to the one used for the score entered by the teacher. As was explained earlier, if the group number and the teacher's score are both five, for example, the weight associated with this term will rise.

The exact amount of the increase is determined by the following algorithm. Both the teacher's score and the term value are reduced by three. Instead of being in a range of one to five, they now fall between -2 and 2, which better illustrates the quality of the move. Zero is neutral; positive numbers mean good moves and negatives, bad moves. For every term the adjusted score is multiplied by the adjusted group number and the product is added to that term's weight. For example, if both sides show that the move was very good, their values will be twos and the product will be four. The weight will increase by four. On the other hand, if one has a value of two (very good move) and the other has a value of negative two (very bad move), the product will be negative four. The associated weight is decreased by four. Since the term value did not agree with the teacher's score, the weight should go down. (See again Figure 4.1.)

To carry this further, suppose that the teacher's score is negative one and the term group number is two. The product will be negative two and the weight will

decrease by two. In this case, the weight goes down because the two evaluations disagreed, but it doesn't fall as much as before because the disagreement was not as large as before. Notice that if the teacher's adjusted score is zero, none of the weights will change. This is as it should be; the teacher decides the move was neither good nor bad, so none of the weights should be rewarded nor penalized.

4.3.2 RELATIVE MODE

Unlike the absolute mode, the relative learning mode deals with each term value only as it relates to other term values. This mode takes all the terms and puts them in order of increasing value. Number one has the lowest value; the last number has the highest value. These numbers are analogous to the group numbers of the absolute mode. But here, there may be as many as seventeen groups, if each term or strategy has a distinct value and forms its own group. Again, the numbers are lowered so that they are equally distributed about zero. (E.g., if there are seventeen distinct groups (each containing one term) they will cover a range from -8 to 8.) Just as in the absolute mode, the numbers are multiplied by the teacher's adjusted score and the product is added to the term's weight. In this case, though, the weight could increase or decrease by as much as 16 in a single move (with a

score of two and a relative group number of eight).

Potentially, this means that the relative mode could learn much faster than the absolute mode. The speed of the learning is probably not very important. The importance of the relative mode is that it takes into account the notion that strategies and their weights are significant only in relation to other strategies and weights in the polynomial. Theoretically, in the absolute mode, every strategy could be considered very good. The purpose of learning is not so much to determine which are good, but rather which are better than the others. The relative mode, then, should accomplish this by viewing each strategy relative to the entire group.

4.3.3 PERCENTAGE MODE

The percentage mode calculates the total sum of the values of all the terms in the polynomial. For each term it assigns a number that represents what percentage this term is of the sum of the term values. These percentages usually fall between zero and ten. As with the other modes, the percentage values are distributed around zero. These adjusted percentage values are multiplied by the teacher's adjusted score and the product is added to the term's weight.

4.4 REMEMBERING

At the end of every game, the set of seventeen weights almost certainly contains different values from those at the beginning. These new weights represent what the program has learned during the game. Depending upon which learning mode was used, this set of new weights is written to one of three external files. The next time this learning mode plays a game, it reads the weights from this file. In this way, each learning mode "remembers" everything it has learned in previous games.

CHAPTER 5

RESULTS

5.1 OVERVIEW

This chapter deals with the results of the learning algorithm. It examines the program's ability to learn, as observed through a series of games. The chapter ends by outlining the conclusions drawn from the learning observations.

It can be difficult to see how well the program is learning because the results of the games are very dependent upon how well the program's opponent plays. If the program wins one game, its opponent might play more carefully the next time. When the program loses the next game, it doesn't mean that the program is playing worse. It might have actually improved its playing ability, but since its opponent is trying harder the final outcome looks worse.

5.2 FIRST EFFORTS

The learning algorithm was first included in the program and tested when the move evaluation polynomial contained only eight terms. In this early stage of teaching, the program simply was supposed to learn to build a chain

across the board in the most efficient way. The teacher-opponent placed his pegs along the sides of the board in order to stay out of the program's way. (In other words, without reacting at all to its opponent's moves, and using only its first eight strategies, the program was taught how to traverse the board.) This early learning will be discussed first, since its results are easier to see than the results from games played at later times.

The program simply placed pegs and links on the board until its chain extended over the entire distance between its home rows (i.e. until it produced a winning chain). As always, after every move by the program, the teacher gave it a score between one and five. In the first game it played, it took 65 moves to build a winning chain. (Remember that this was not really a game, but rather just an attempt to traverse the board.) The resulting game board is illustrated as Game One in Appendix A. The opponent's moves are not shown. Looking at the game board, it's obvious that the program made many unnecessary moves. Since the most direct chain can win in only thirteen moves, the first attempt placed 52 unnecessary pegs.

In the second game it played, the program improved dramatically. Here it needed only 23 moves to cross the board. (Appendix A - Game Two.) After learning for only one game, the program was able to construct a winning

chain in one-third the number of moves it needed previously. The third game produced results similar to the second. Then in the fourth game, the program crossed the board in thirteen moves, the smallest possible number. This game is shown in Appendix A as Game Three. These initial learning results were certainly very encouraging. By playing only four games the program had learned to produce optimum results. But notice that while it played four games, the program made more than one hundred moves. Since the teacher was rating it after every move, the program adjusted its strategies' weights over one hundred times. Obviously, as the results show, this was plenty of time to separate the good strategies from the bad ones.

This should be compared to the more usual game-playing method of learning only after each game rather than after every move. Learning after every move is almost certainly faster. It is unlikely that the program would have been able to produce optimum results after only three games if it had been learning or adjusting its move evaluation function only once every game. Changing the evaluation function after every move also seems to allow the adjustment to be more precise. A single situation is taken into account. After a complete game, it would be more difficult to determine which strategies performed well and which did not.

It is interesting to note the weights of the random number generator strategies during this learning session. As was already stated, all the weights start at 100. After the first game (i.e. 65 moves) the first random number generator's weight had been reduced to two; the second one's weight was ten. Both of the weights ultimately reached zero. This showed that within only four games the program had determined that the random strategies were worthless.

5.3 FULL IMPLEMENTATION

After the learning algorithm was tested using the first eight strategies, more strategies had to be developed. In general, strategies were conceived by playing games of Twixt and observing what types of moves were needed at certain points during the game or, more importantly, why players made certain moves in particular situations. Every strategy that was thought of was implemented. As has already been stated, there were ultimately seventeen strategies.

After all the strategies were included in the program, the seventeen weights were reset to 100 and the main learning process was started again. Several rounds of games were played; in every round one game was played using each of the three learning modes. Each learning mode adjusted and kept a record of its own weights and

used them in subsequent games which it played. Looking at the games themselves, it is somewhat difficult to observe the program's progress. It is worthwhile, though, to look at some of the games and then to observe the differences in the values of the weights among the three learning modes.

Round four (after each learning mode had played three games) was probably the most important one. The final state of the game board and the list of plays for the absolute mode's fourth game are shown in Appendix A - Game Four. The program is playing black. The teacher-opponent started the game by placing his pegs near the center of the board. The program countered, as it should have, by building a chain in rows Q through T and columns 7 through 15. This is a very effective block against its opponent's central moves. Realizing that he was blocked, the program's opponent started a chain centered around p20. This, of course, blocked the program's chain. At this point the program should have abandoned its chain, as its opponent had, and started to build a new chain which would block its opponent and also perhaps subsequently connect with any existing chain segments to form a winning chain. But to start a new chain, the program must place a single, isolated peg without a link. Since the program was so highly rewarded for placing links, once it started a chain it would not abandon it. In other words, the program had

to learn that it is not always appropriate to connect new pegs to existing chains.

Up to this point (round four) the program's only solution to a blocking situation was to try to go around the opponent's chain. That's exactly what it tried to do here. The program built its chain in the direction of w18, but it was unable to get around the opposing chain before the edge of the board was reached. The program's next move was at r5 with a link to s7. Clearly it didn't know what to do here. Its chain now extended in a single path from r5 to v20. One end (the critical end) of its chain was blocked. Not knowing any other way to get around the block, it chose a move which extended the other end of its chain. Even though it was obvious that the block had to be circumvented, the best move which the program could find was one that extended the wrong end of its chain.

Its opponent placed a peg at l18, so his originally defensive blocking chain now extended from l18 to x20 and was dangerously offensive. The program's next move was probably the most important single move in any game of any round of the learning process: It placed a peg (without a link) at location f16. It had abandoned its semi-worthless chain and had started to build a block in front of the opposing chain. It had finally learned that there

are times when all its previous work must be abandoned; sometimes pegs must be placed, for defensive purposes, in what might seem like the middle of nowhere. More importantly, it had learned that placing links just for the sake of placing links is usually not a good strategy. A link must have a definite purpose or goal, such as extending a chain at an appropriate time or going around an opponent's chain. The program ultimately lost the game; but in its losing effort, it increased its ability tremendously, simply by having learned to place the blocking peg at f16.

Later in round four, the relative and percentage modes played their games. Both of them found themselves in the same situation of having their chains blocked and, rather surprisingly, both were able to choose to abandon their chains and place a blocking peg in front of their opponent's chains. Obviously, all three modes were learning at about the same rate.

The game played by the percentage mode in round eight is illustrated (game board and moves) in Appendix A - Game Five. Again the program is playing black. Here the program has three separate chains. It has learned well not to put too much emphasis on placing links, and it has also done a fairly good job of blocking its opponent. In rows R through W on both sides of the board it was able to stop

red's chain by "pushing" it into the side. Ultimately red was able to build a chain on rows P and Q which passed through the gap between two of the program's chains. By extending this chain to row X, red won the game.

5.4 EFFECTIVENESS OF STRATEGIES

If the strategies' weights from each learning mode are compared after the learning modes have all played the same number of games, the differences in their learning patterns can be seen. Figure 5.1 shows all the weights after each mode had played four games. Looking at the first two terms (RANDs), it can be seen that the relative mode has learned best that these strategies are very bad. The absolute mode has also decreased their weights, but not as much. The percentage mode has actually increased the weight of one of the random strategies. It still seems to consider that strategy to be fairly good. The three modes are unanimous in their judgements of terms eight (ADVANCE) and twelve (BLOCK). These two strategies increase the total distance spanned by the player's chain and try to avoid areas of the board which are blocked or controlled by the opponent. They are both good strategies and each mode has given them very high weights.

Probably the most interesting weight is the zero that the percentage mode has assigned to term five (MAKE-BRIDGE). Strategy five simply says that it's better to

STRATEGY	LEARNING MODE		
	absolute	relative	percentage
1	54	0	107
2	44	18	42
3	91	102	126
4	95	109	87
5	47	80	0
6	86	10	0
7	96	20	24
8	149	273	250
9	107	157	217
10	114	241	121
11	140	171	201
12	150	237	286
13	64	53	81
14	135	126	181
15	135	121	191
16	141	119	181
17	131	121	175

WEIGHTS AFTER ROUND FOUR

Figure 5.1.

place a link than it is not to place one. The weight of zero signifies that the percentage mode has determined that placing a link is useless. It has already been seen that the program should avoid putting too much emphasis on placing links, but it seems that the percentage mode has gone to an extreme and is placing too little emphasis on links. (Chains cannot be built and games cannot be won unless links are placed on the board.) Actually, this is not as bad as it seems. Several other terms (most notably term eight) indirectly reward moves which include links.

The percentage mode simply shows that links which increase the total span of the chain (term eight) are good, but placing a link just for the sake of placing a link is not worthwhile.

Figure 5.2 shows the weights after each mode has played seven games. By this time both the absolute and relative modes have learned very well that random strategies are bad. Percentage mode is making progress in the right direction, but it still has a way to go. All three

STRATEGY	LEARNING MODE		
	absolute	relative	percentage
1	2	8	30
2	4	11	78
3	62	31	122
4	53	30	138
5	4	6	6
6	48	0	44
7	63	0	48
8	191	329	279
9	118	192	237
10	110	172	150
11	161	278	273
12	166	281	333
13	62	52	67
14	182	186	210
15	178	181	210
16	186	175	204
17	172	181	204

WEIGHTS AFTER ROUND SEVEN

Figure 5.2.

modes have continued to increase the importance of terms eight and twelve. By this time also, notice that all three modes seem to have come to the conclusion that links for the sake of links are almost useless. None of them assigns to term five a weight above six.

Before the learning process began (when all the weights were set at 100), the program was run to see how many moves it took to build a chain across the board. As in the initial testing of the learning algorithm, described at the beginning of this chapter, the program's opponent placed his pegs along the sides of the board to stay out of the program's path. With all seventeen terms in the polynomial and without having learned anything yet, the program built a winning chain in 31 moves. After round six the test was run again using the polynomial's new weights. Each mode built a chain across the board to show how well it had learned. The absolute mode needed 16 moves to complete the chain. That's close to optimum (13 moves) and a good improvement over the 31 moves needed before learning began. The relative mode was even better; it built a winning chain in 14 moves. Unfortunately, the percentage mode was not able to construct a complete chain until it had made 45 moves. After it had been taught for six games, it took longer to cross the board than it had when it started.

5.5 CONCLUSIONS

The results of teaching the program show that the relative learning mode learned better than the other two. It can build a winning chain in almost optimum time and the weights it has assigned to its strategies have values which seem to reflect their actual importance quite well. The absolute mode has also learned well. It appears to be following the relative mode's learning path but is not progressing quite as fast. The percentage mode has had trouble learning certain things. In general the percentage mode plays a more interesting game than the other two. (It certainly makes a higher percentage of very creative moves; unfortunately, many of them are also very bad moves.)

It still doesn't seem that the program is good enough to beat a human opponent. Perhaps it could beat a beginner, or maybe a very bad player. This lack of ability is for the most part due to the quality of the game strategies. Implementing some of the suggestions in the next chapter would be a good way to improve the program's play.

The learning algorithm, especially the relative mode, used the strategies it had at its disposal very well. The main purpose of the algorithm was to learn which strategies were good and which were bad or, in other words, to

learn the importance of each strategy relative to the others. The results show that the program attained a very acceptable degree of success in its learning attempts.

CHAPTER 6

FUTURE ENHANCEMENTS

6.1 OVERVIEW

This final chapter discusses some improvements which could be made in the program.

6.2 BETTER STRATEGIES

The most basic and straight-forward improvement would be simply an extension of what already exists. Specifically, more and better strategies could be implemented. Using the strategies which it has now, the program can learn the relative importance of each strategy and adjust its weight accordingly, but ultimately, the weights stabilize and the program's learning reaches a plateau. There are some things the program will never learn through further teaching. For example, if the program is building a chain which is approaching a barrier set up by its opponent, it is smart enough to try to go around the barrier. But it has no way of anticipating a future barrier that the opponent could easily construct. Consider the set-ups described as strategies 14 to 17 and illustrated in Figures 3.1 through 3.4 of chapter three. If the program sees the gap between the two set-up pegs, it will build its chain in that direction and try to pass through.

But on his next move, its opponent can insert the third peg of the set-up to close the hole to effectively block the program's chain. With its current strategies, the program will never learn to avoid this trap. The problem could be taken care of by including an appropriate new strategy.

6.3 MULTIPLE EVALUATION FUNCTIONS

Another problem is that at different times during the game of Twixt, vastly different methods of play should be used. Strategies which are valid and important near the beginning of the game may be useless and even detrimental if used during the middle game. For example, at the beginning of the game wide open spaces should be covered as quickly as possible. A strategy like ADVANCE (term eight), which increases the total span of a chain, should have a high weight. Later in the game, however, large chain segments are usually already constructed. The important strategies would then involve intricate maneuvering around obstacles and through tight spaces. It's often the case at this point that chains must be built in the opposite direction of their ultimate goals so they can combine with other chain segments to find winning paths across the board. If a strategy like ADVANCE has a high weight, it could easily force the evaluation polynomial to choose a move that is not appropriate for this

part of the game. Unfortunately, the program's move evaluation polynomial is static. The same evaluation criteria are used both at the beginning of the game and in the middle game.

The solution to this problem could be to have more than one move evaluation polynomial. At appropriate times the program would switch polynomials. The difficulty with this solution is in deciding when a switch is appropriate. The game actually goes through phases of playing in open and then enclosed spaces. If two polynomials were used, move evaluation would have to switch from one polynomial to the other several times during the game. To implement this multiple polynomial method, a board evaluation function would have to be developed. This function would examine the game board and the players' relative positions to determine where the current critical areas are located. Using this information, it would assign one of the polynomials to be used to choose the next move.

The two polynomials used could be made up of the exact same seventeen terms that the program's single polynomial now has. In others words, their strategies would be identical. The differences in the two polynomials would be in the relative weights of these strategies. For example, the term ADVANCE referred to earlier would probably have a very high weight in the beginning-game polyno-

mial. The middle-game polynomial would still include this strategy, but its weight would be very low.

Berliner's backgammon program [BERL 80b] addresses the problem of having only one evaluation polynomial to take care of many types of moves. Since Berliner's program does not learn, the weights in his evaluation polynomial are constant. In backgammon there are two very different types of endgames. One, which is basically defensive, is called a blockading game; the other one, which is very offensive, is the running game. Berliner originally decided to make two classes of game situations. Each class would have its own evaluation function. By determining which type of situation the game was in, the appropriate evaluation function would be chosen to select a move. Some of the strategies might be in both functions, but their corresponding weights would be very different.

After further testing Berliner noticed an anomaly. When the game situation was near the border of two classes (in other words, it wasn't clearly in one class or the other) the two evaluation functions should have produced results that also were very close. But since the strategies' weights were so different, the results of the polynomials were very dissimilar. Berliner wanted the class boundary to have a smooth transition, instead of

being such an abrupt and total change from one class to the other. He ultimately constructed one evaluation polynomial which had variable weights. The weights were made to change very slowly as the game went from one phase to another. This eliminated the abrupt changes, but still made it possible to play equally well in different game situations. Berliner called the approach SNAC, for "smoothness, nonlinearity and application coefficients." After SNAC was introduced, the program's playing ability improved rather dramatically. According to Berliner, the SNAC approach is directly responsible for defeating the world backgammon champion.

6.4 DEFENSE VS. OFFENSE

It also could be useful to take into account the need at different times for an offensive versus a defensive move. For each different move in the game, the relative importance of offense and defense can fluctuate quite a bit. The strategies in the move evaluation polynomial can be divided into these two categories and a board evaluation function which examines the critical areas could again be used to determine which type of move should be made. If the function determines that a defensive move should be made, then the weights for the defensive strategies would be increased. Certainly, it would not usually be the case that a purely offensive or defensive move

is called for. The board evaluation function could return a ratio of the current relative importance of the two types of moves. The weights in the move evaluation polynomial could then be adjusted either up or down proportional to this ratio.

6.5 LOOK-AHEAD

As it is set up now, the Twixt program only evaluates its next move. It could probably greatly improve its move selection if it were able to look ahead a few moves. As was mentioned in chapter one, a look-ahead algorithm is a very common component of game-playing programs. A look-ahead algorithm would be extremely useful for a game like Twixt. Many times a move which seems to be very good can lead directly to unavoidable and disastrous results. If the program is able to check the consequences of its actions by looking ahead a few moves, it should be able to improve its play considerably.

A typical look-ahead algorithm is called mini-max searching. Slagle [SLAG 71] gives a good discussion of the details of mini-max. This algorithm searches the game tree and follows the branch that will offer the opponent minimum opportunities while giving maximum benefits to the player searching for a move. Searching for a good move is often not enough. If the opponent can respond with a better move, the benefits of the original move will be

lost. A look-ahead algorithm must select a move which not only benefits the current player, but one which also leaves his opponent with a poor selection of moves.

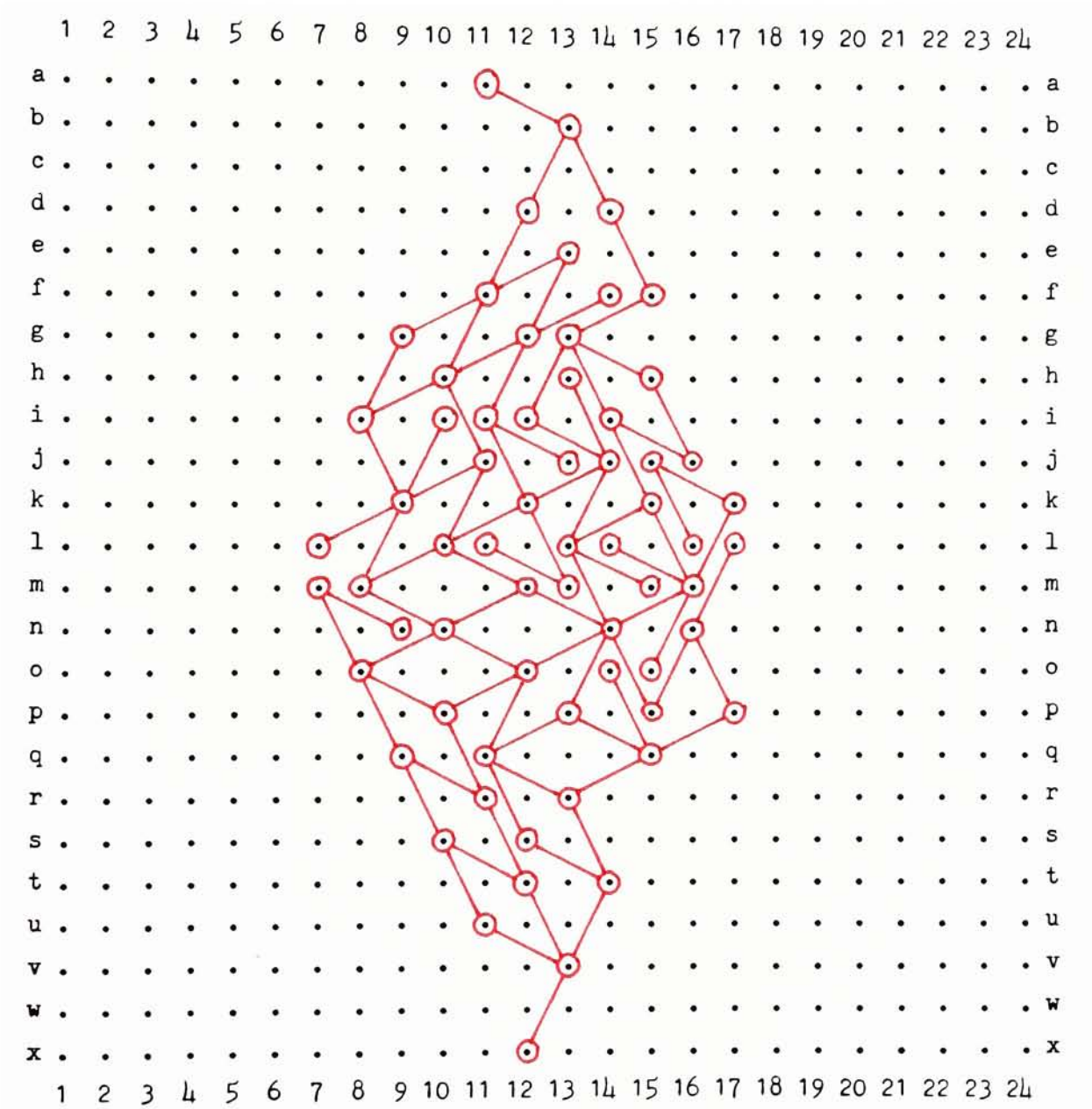
There is one problem which arises when using a look-ahead algorithm; the amount of time needed to evaluate moves would be extremely large. Now the program evaluates moves for every unoccupied hole on the board. (There are 572 holes.) If the program looked ahead only to its opponent's next move, it would have to evaluate every possible move by its opponent for every possible move it could make. In other words, the program would have to make a little less than 572 times 572 move evaluations (omiting impossible moves - in opponents' home rows or where pegs have already been placed). To take the fullest advantage of a look-ahead algorithm, it would probably be reasonable to expect to have to look ahead three or four sets of machine and opponent moves. It would probably be necessary to look ahead at least this many moves to effectively show any significant consequences of the program's moves. Looking ahead, for example, three pairs of moves would result in approximately 572^6 move evaluations. Since it can now take over thirty seconds per move while making only 572 evaluations, trying to evaluate 572^6 moves certainly would consume too much time and space.

Berliner's game [BERL 80b] also does not use a look-ahead algorithm. For each move in backgammon there are over 400 possibilities. This would result in searching a tree which has a branching factor of 400 on every level. Weizenbaum considers a look-ahead function to be the single greatest improvement he could make in his five-in-a-row program [WEIZ 66b]. Five-in-a-row's branching factor is only about 30, which is a considerable improvement over 400 or 572, but Weizenbaum still is, and should be, concerned with finding methods to prune the search tree.

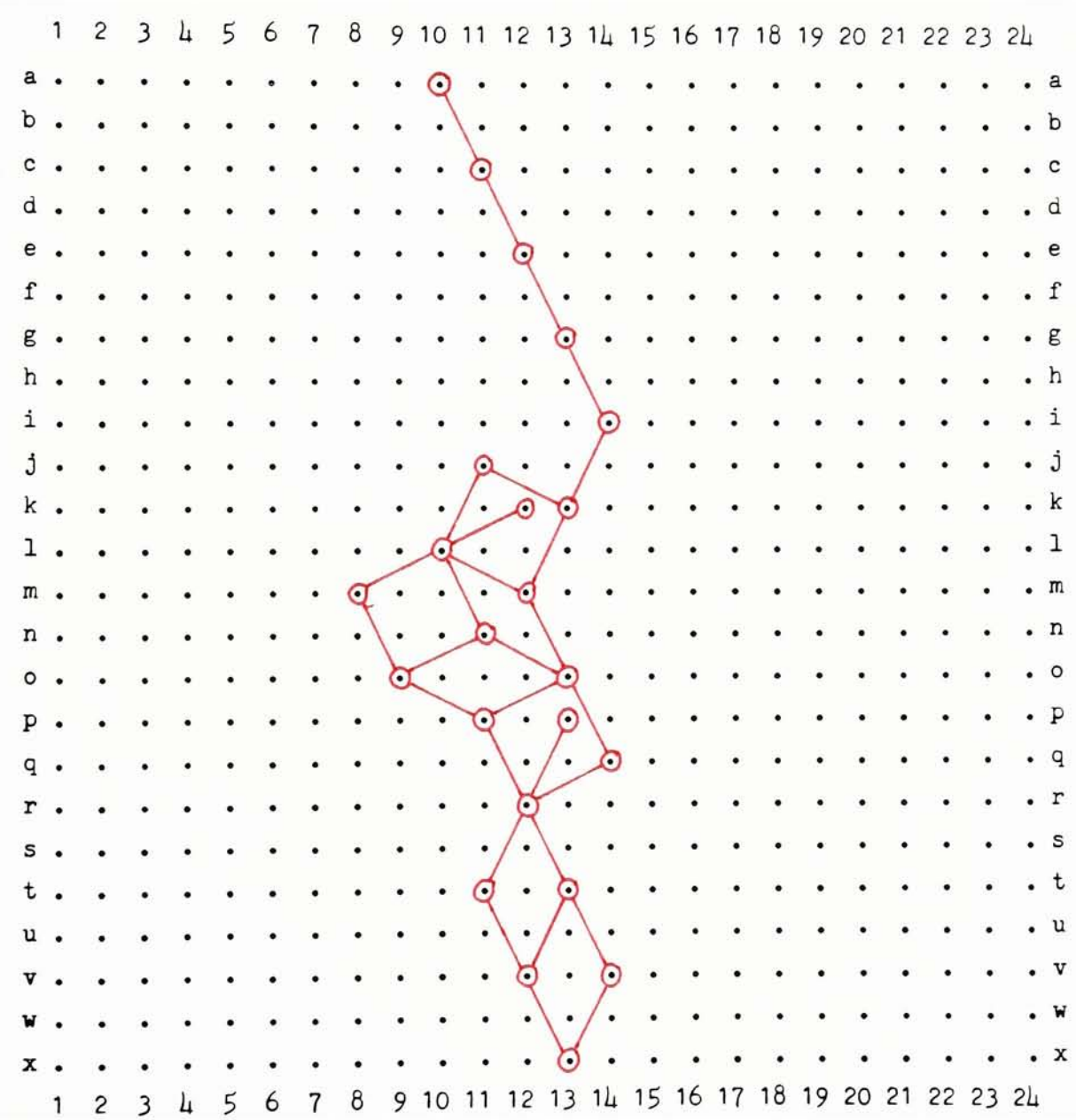
One method which could be adapted to Twixt would be to select the several best moves on each level of the look-ahead tree, and then perform a search on these branches only. This would certainly cut the time needed compared to the conventional look-ahead algorithm, and would probably also improve play. However, it would almost certainly overlook many good moves. The look-ahead strategy is based on the premise that moves which seem ordinary on the surface might lead to wonderful and unforeseen developments. By eliminating these moves from the search tree, the program may be overlooking some excellent moves. Nonetheless, if the time needed for a conventional look-ahead is prohibitive, this abbreviated version could prove useful. Bernstein uses a method similar to this in his chess playing program [BERN 58]. He searches through seven moves at each level.

Another look-ahead modification, which works well with the mini-max searching mentioned above, is called alpha-beta pruning. This is also described by Slagle [SLAG 71]. Alpha-beta pruning avoids searching through the entire game tree. Searching stops along any particular branch as soon as it is determined that, regardless of what lies at a deeper level, the branch will not produce a move as good as one which has already been found.

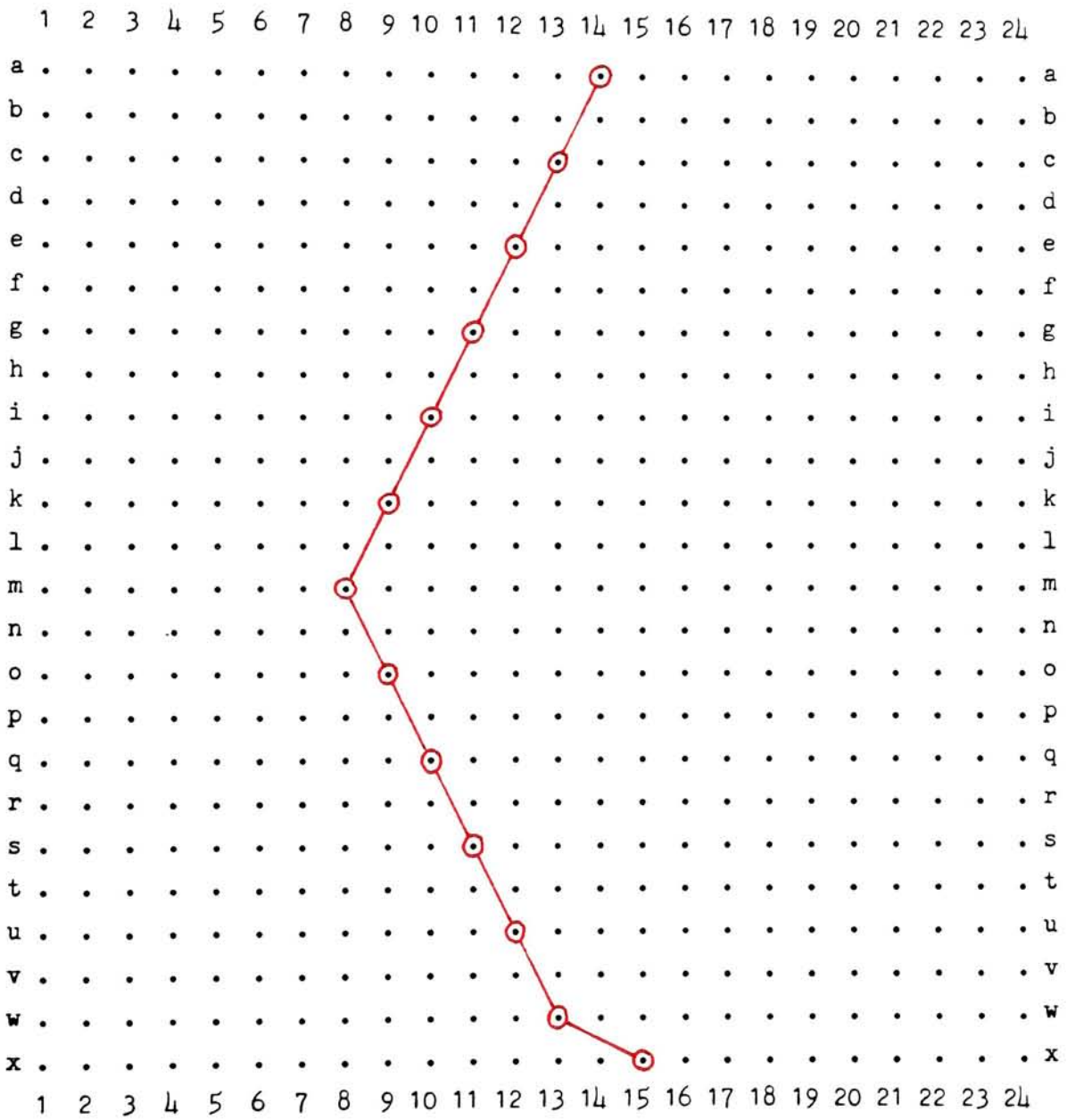
Game 1


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
a b c d e f g h i j k l m n o p q r s t u v w x

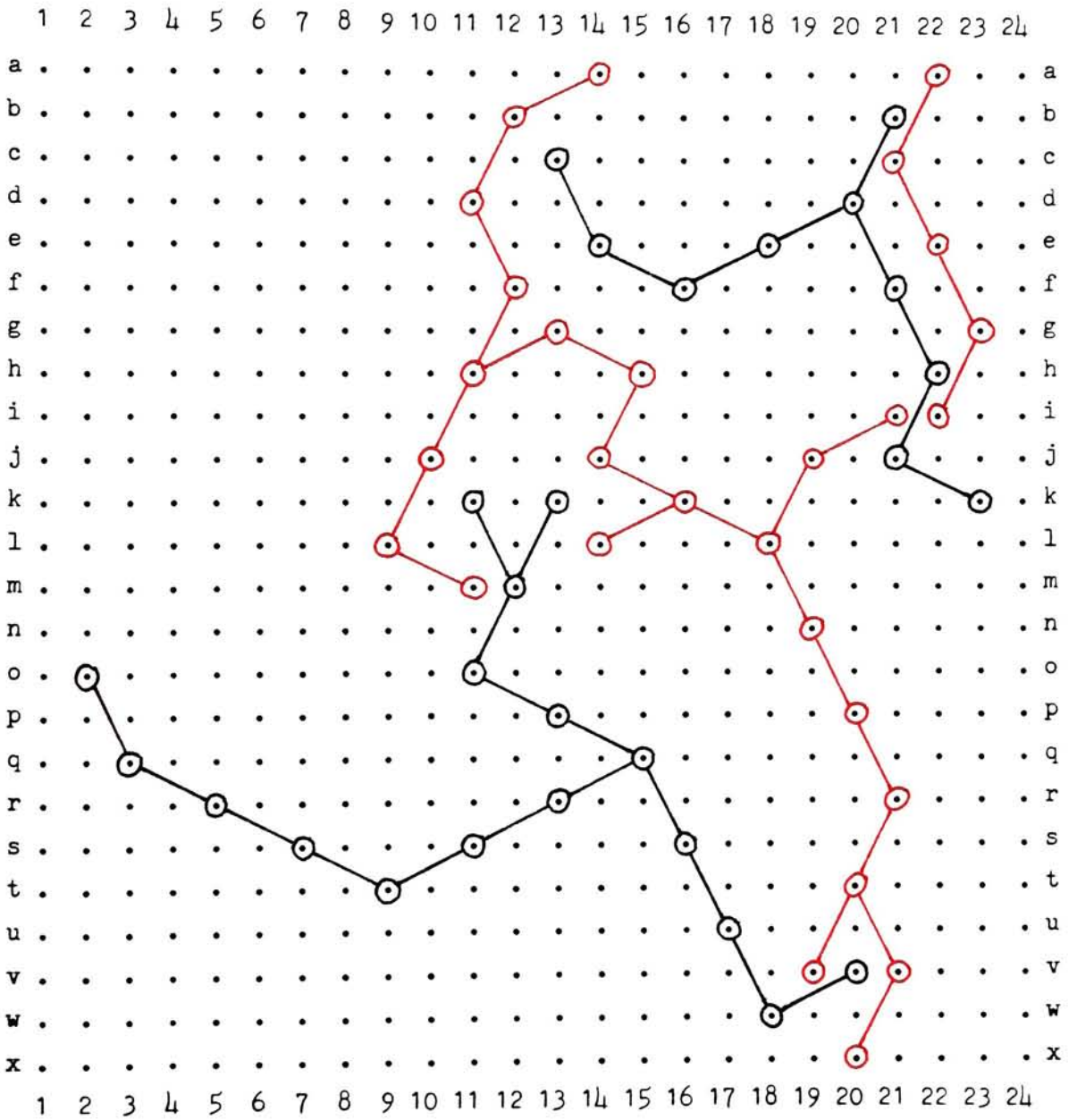
Game 2


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
a b c d e f g h i j k l m n o p q r s t u v w x

Game 3


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
a b c d e f g h i j k l m n o p q r s t u v w x

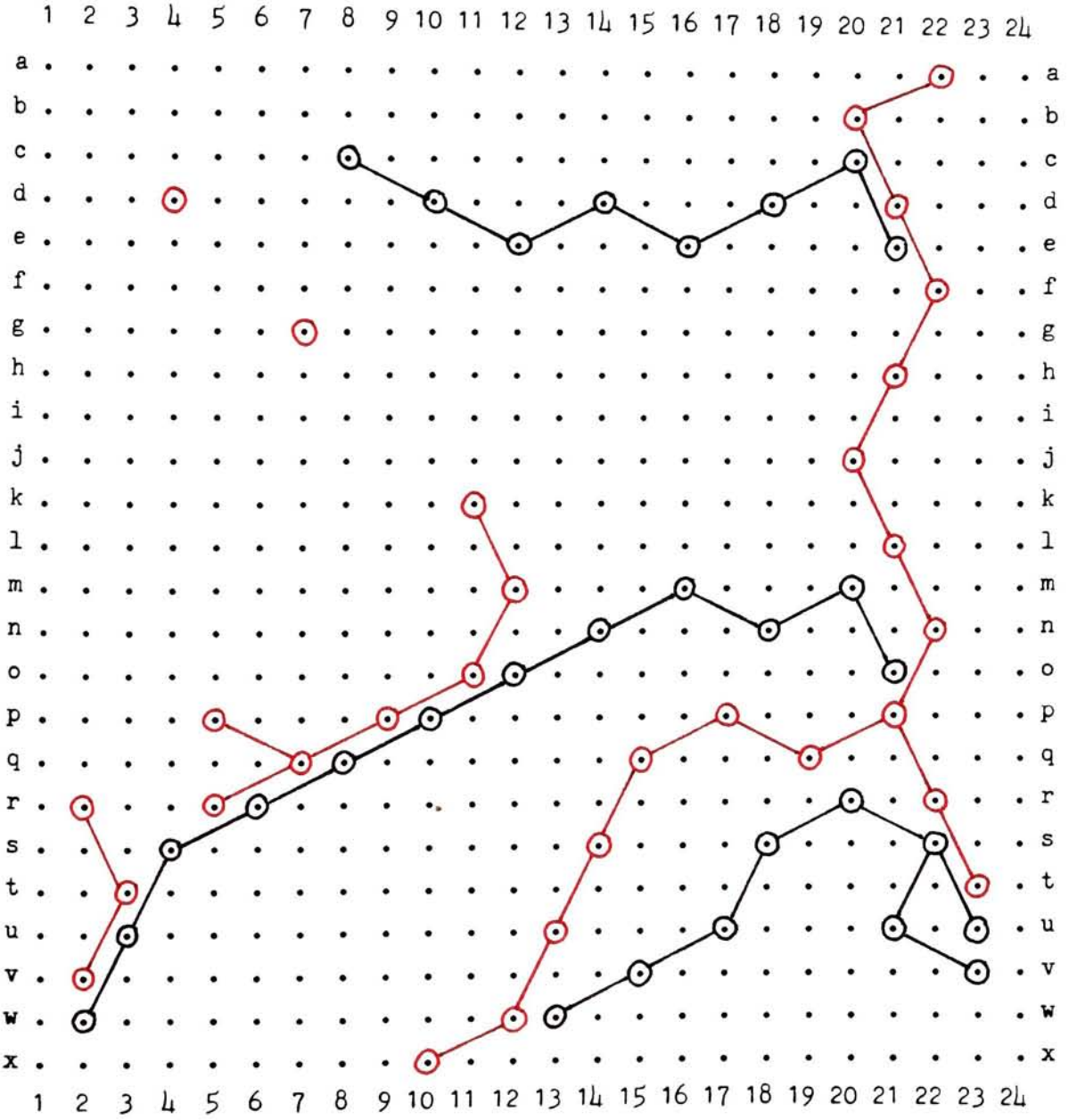
Game 4


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
a b c d e f g h i j k l m n o p q r s t u v w x

Game 4

	-RED-	-BLACK-
1	l9	t9
2	j10(l9)	s11(t9)
3	h11(j10)	r13(s11)
4	r21	q15(r13)
5	p20(r21)	s7(t9)
6	n19(p20)	s16(q15)
7	t20(r21)	u17(s16)
8	v19(t20)	w18(u17)
9	v21(t20)	v20(w18)
10	x20(v21)	r5(s7)
11	l18(n19)	f16
12	f12(h11)	e14(f16)
13	j19(l18)	e18(f16)
14	e22	d20(e18)
15	c21(e22)	b21(d20)
16	a22(c21)	c13(e14)
17	d11(f12)	q3(r5)
18	b12(d11)	o2(q3)
19	a14(b12)	p13(q15)
20	i21(j19)	f21(d20)
21	g23(e22)	h22(f21)
22	i22(g23)	j21(h22)
23	m11(l9)	o11(p13)
24	k16(l18)	k23(j21)
25	l14(k16)	m12(o11)
26	j14(k16)	k13(m12)
27	g13(h11)	k11(m12)
28	h15(g13)(j14)	
	WIN	

Game 5


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
a b c d e f g h i j k l m n o p q r s t u v w x

Game 5

	-RED-	-BLACK-
1	m12	e12
2	k11(m12)	n14
3	g7	o12(n14)
4	o11(m12)	p10(o12)
5	p9(o11)	q8(p10)
6	q7(p9)	r6(q8)
7	r2	d10(e12)
8	p5(q7)	c8(d10)
9	d4	d14(e12)
10	r5(q7)	s4(r6)
11	t3(r2)	u3(s4)
12	v2(t3)	w2(u3)
13	n22	m16(n14)
14	l21(n22)	e16(d14)
15	h21	d18(e16)
16	d21	c20(d18)
17	b20(d21)	e21(c20)
18	f22(d21)(h21)	n18(m16)
19	p21(n22)	m20(n18)
20	j20(h21)(l21)	r20
21	r22(p21)	s22(r20)
22	t23(r22)	u23(s22)
23	q19(p21)	s18(r20)
24	q15	u17(s18)
25	s14(q15)	v15(u17)
26	u13(s14)	w13(v15)
27	w12(u13)	u21(s22)
28	x10(w12)	v23(u21)
29	a22(b20)	o21(m20)
30	p17(q15)(q19)	
	WIN	

REFERENCES

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

[BERL 80a] Berliner, Hans, "Backgammon Program Beats World Champ," ACM Special Interest Group on Artificial Intelligence, SIGART Newsletter, No. 69, Jan. 1980, p. 6.

[BERL 80b] Berliner, Hans, "Computer Backgammon," Scientific American, June, 1980.

[BERN 58] Bernstein, Alex and Roberts, Michael de V., "Computer v. Chess-Player," Scientific American, Vol. 198, No. 6, June 1958, p. 96.

[GARD 79] Gardner, Martin, Mathematical Circus, Knopf, New York, 1979.

[HOFS 80] Hofstadter, Douglas R., Goedel, Escher, Bach: an Eternal Golden Braid, Random House, New York, 1980.

[JONE 80] Jones, A. J., B.Sc., Ph.D., Game Theory: Mathematical Models of Conflict, Ellis Horword, Chichester, West Susex, 1980.

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

[NILS 80] Nilsson, Nils J., Principles of Artificial Intelligence, Tioga Publishing, Palo Alto CA, 1980.

[SAMU 60] Samuel, A. L., "Programing Computers to Play Games," in Alt, Franz L. (ed.), Advances in Computers 1, Academic Press, New York, 1960.

[SAMU 63] Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers," IBM Journal of Research and Development, Vol. 3, No. 3, July, 1959, pp. 210-219. Reprinted in Feigenbaum E. and Feldman J. (eds.), Computers and

Thought, McGraw Hill, New York, 1963.

[SLAG 71] Slagle, James R., Ph.D., Artificial Intelligence: The Heuristic Programing Approach, McGraw Hill, New York, 1971.

[WEIZ 62a] Weizenbaum, Joseph, "How to Make a Computer Appear Intelligent," Datamation, Vol. 8, No. 2, Feb. 1962, p. 24.

[WEIZ 62b] Weizenbaum, Joseph, "Gamesmanship," Datamation, Vol. 8, No. 4, Apr. 1962, p. 10.

[WINS 77] Winston, P. H., Artificial Intelligence, Addison Wesley, Reading MA, 1977.