1986

A learning program for the game of 'Go-Moku'

Dong-Ming Wang

Rochester Institute of Technology
School of Computer Science and Technology

A Learning Program For the Game of 'Go-Moku'

By
Dong-Ming Wang

A thesis, submitted to
The Faculty of The School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
Oct 25, 1986

Approved by:

_____
Professor John A. Biles

_____
Professor Warren R. Carithers

_____
Professor Lawrence Coon

Title of Thesis: *A Learning Program for the Game of Go-Moku*

I, *Thomas Dong-Ming Wang*, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

162 Smithfield Court, Basking Ridge, NJ 07920

Thomas. D. WANG

Date: 5-28-87

ACKNOWLEDGEMENT

I wish to acknowledge the aid of Professors Warren Carithers and Lawrence Coon in helping me complete this thesis. I also wish to acknowledge the aid of Professor John Biles. He was particularly instrumental in the organization and defense of the thesis. My special thanks to the parents of Paul Huebner for allowing me to use their copy of his thesis as reference. And to my parents and my wife, I owe the greatest debt of all.

TABLE OF CONTENTS

1. Introduction and Background
    1.1 Overview
    1.2 History of Computer Games
    1.3 Common Features of Games
    1.4 The Game of Go-Moku
    1.5 Outline of the Paper

2. Evaluation Strategies
    2.1 Overview
    2.2 Strategies

3. The Program
    3.1 Overview
    3.2 Program Organization
    3.3 Primary Data Structure
    3.4 Program Execution
    3.5 More Data Structure

4. Learning
    4.1 Overview
    4.2 Fast Learning Mode
    4.3 Post Game Learning Mode
    4.4 Mode 3 & Mode 5
    4.5 Mode 4

5. Results
    5.2 Overview
    5.3 Effectiveness of Strategies
    5.4 Post Game Learning
    5.5 Fast Learning
    5.6 Computer vs. Computer Game

6. Future Enhancement
    6.1 Overview
    6.2 Strategies Enhancements
    6.3 Look-Ahead
    6.4 Multiple Functions

Bibliography
APPENDIX 1
APPENDIX 2
APPENDIX 3
APPENDIX 4
APPENDIX 5

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Games have been a part of human culture (both Eastern and Western) for centuries. They have served as intellectual, educational, and recreational endeavors for people from all classes of society. Because game playing has become such a pervasive component of human life, it is only natural for it to be an area of interest within Artificial Intelligence, which tries to understand and develop computer implementations of basic mechanisms of intelligence. [NILS 80]

In the 1960's games were recognized as a controlled medium for conflict and aggression between two opponents. Two of the oldest intellectual games, *Chess* and *Go*, were outgrowths of the training of soldiers in military tactics. *Chess*, a familiar game in Western society, is concerned with techniques of King protection and capture. *Go*, a familiar game in Eastern society, is concerned with techniques of territory protection, acquisition, and capture. Although similar in some respects, *Chess* and *Go* differ as much as the societies from which they originated. They are both extremely difficult to play well, and people have spent entire lifetimes studying their play.

An important area of Artificial Intelligence is computer game playing. By programming computers to play games it is possible to study decision making (how to decide which move to make) and learning (how play can be improved). 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 similarities 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 two approaches of computer game playing by programming a computer to play the game of *Go-Moku*.

1.2 HISTORY OF COMPUTER GAMES

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 [SLAG 71] includes several chapters describing problems that arise from trying to program different types of games. Samuel [SAMU 60] concentrated for the most part on the games of chess and checkers.

The earliest modern computer games date from the 1950's. 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.

Previous work on *Go-Moku* has been done by Weizenbaum (1962), Koniver (1963), Elcock and Murray (1966), and Murray and Elcock (1967). The number of *Go-Moku* playing programs has increased since the fall of 1975 due to the creation of the North American Computer *Go-Moku* Tournament [HEUB 77]. Previous work in *Go-Moku* programming has been done by Shannon (1950), Newell, Shaw, & Simon (1958), Kotok (1962), Greenblatt (1967), Berliner (1970), Levy (1970), Gillogly (1972), Eobrist & Carson (1973) and Mittman (1973), Good and Crook (1974), Huebner (1977), Arlazarow and Futer (1979), Wilkin (1979), Perry (1980).

1.3 COMMON FEATURES OF GAMES

As suggested by Slagle (1971), heuristic programming is the approach most commonly incorporated into game playing programs. Slagle defines an heuristic as a rule of thumb, strategy, method, or trick used to improve the efficiency of a system that tries to discover the solutions to complex problems. In game playing programs the problem is to select a 'best' move at each turn.

Slagle also suggested are two basic methods for selecting a move at a given point in a game from the hundreds or thousands available: static evaluation functions and tree-searching procedures [SLAG 71]. A static evaluation function assigns a value to each possible move without generating any successive moves. A search procedure consists of a procedure for generating successive moves, a static evaluation function, and a procedure used for matching the results of successive moves [HEUB 77].

Another common feature in game-playing programs is a look-ahead algorithm. By using this algorithm, the program does not stop evaluating more when it finds a best move on basis of its current situation. For every possible move by the program, it also evaluates each possible responding move by the opponent. The look-ahead algorithm can search three or four pairs of moves, or perhaps until it reaches a certain pre-determined position in the game.

For a game like *Go-Moku* a look-ahead algorithm can be useful, but it can be very expensive. In a look-ahead algorithm, an N-ary game tree where N is the number of possible moves for each turn needs to be built to represent possible moves to be searched. There are well over 300 possible moves to search for in every turn, the cost of this look-ahead algorithm is the main reason it was not used in this thesis. A possible enhancement to include look-ahead will be discussed in the last chapter.

In this thesis we compare two learning algorithms. First is the traditional post-game learning algorithm, in which learning is performed after every round of the game is finished. The second is the fast learning algorithm, in which learning is performed after every move. The

basic technique for doing both involves a polynomial evaluation function. The variables used in this polynomial function represent individual strategies that which affect move making in accordance with a certain viewpoint. Through the learning process, the importance of each variable is adjusted, thus improving future move making. These variables are called *terms*. For each of these terms, numeric *weights* are numeric values that represent the terms' importance in affecting certain moves.

1.4 THE GAME OF GO-MOKU

Go-Moku, or *five-in-a-row*, is a two-person game played on a 19 by 19 grid. Figure 1.1 shows a *Go-Moku* game-board. The pieces used to play are called stones and are played at the intersections of rows and columns. The stones are of two colors, white and black, and each player has 100 stones. The player choosing the black stones always begins the game. The players take turns placing one stone of their color at a time on the game board. Stones can be placed at any location where a stone has not been played in the current game, and once a stone is played it may not be moved.

The first player to form an unbroken straight line of exactly five adjacent matching stones, in either a horizontal, vertical or diagonal direction, is the winner.

Winning strategies are known for situations from which a win can be forced by placing one or two more stones if the opponent doesn't make his (her) next move to prevent this.

As stated previously, the goal in *Go-Moku* is the creation of a row of five matching adjacent stones either horizontally, vertically, or diagonally. A player must play offensively enough to work towards the creation of a winning strategy while at the same time play defensively enough to keep the opponent from building a winning strategy.

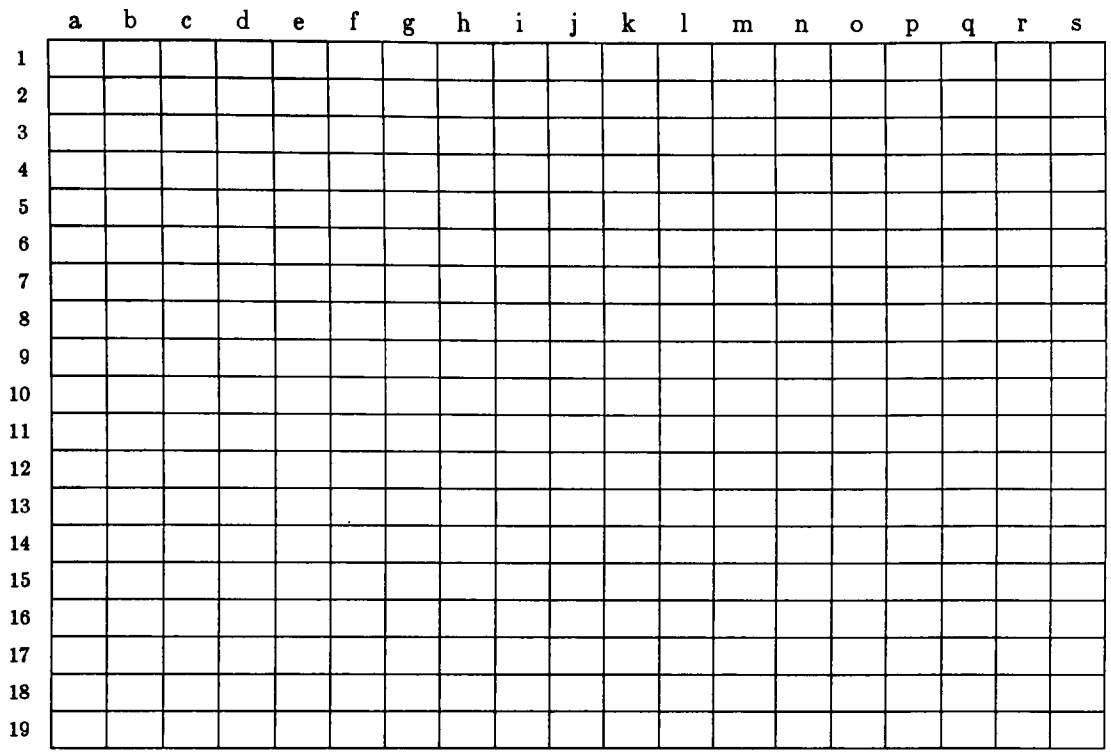


Figure 1.1 a Go-Moku game-board

One question that constantly confronts a *Go-Moku* player is when to play offensively and when to play defensively. If a player plays offensively for too long the opponent may have formed strategies that cannot be blocked. On the other hand, playing defensively for too long may prevent the opponent from winning, but may also mean that a player has not created strategies that could lead to a win.

Koniver (1963) described strategies that a player may form to dictate predictable responses by the opponent and in so doing control the game. These strategies are called *threatening-five, open-four, open-three, crossing open-three,* and strategies that contain the intersection of an *open-three* and a *threatening-five.* We call all of the above mentioned *potential five strategies.*

A *threatening-five* (or *closed-four*) strategy is created when a player has four stones along a line with neither end blocked. Figure 1.2 shows a *threatening-five* strategy (from now on, we use O to stand for stone of white, and use X to stand for black). Suppose the player using white stones just made a move at [10, j]. Black has to block the right end of row 10 where four consecutive stones have already been played by his (her) opponent, for if the opponent's next move is at [10, m] the game is lost.

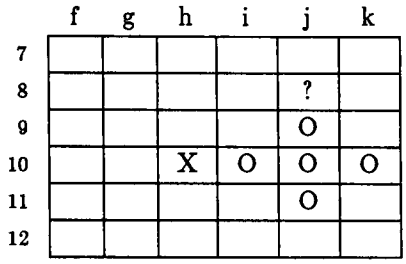


Figure 1.2

On the other hand, if black chooses location[10, m], a special case of a *threatening-five* strategy will be formed white making his (her) next move at [8, j]. This is because column j has three consecutive white stones already played with both ends opened so that a white stone placed on either end will make four consecutive stones with both ends opened, a *threatening five*. Black's next move can only block one end of the consecutive line of four stones and white can easily win the game by placing his (her) next stone on the other end. The strategy of playing four consecutive stones so that both ends open are called an *open-four*. With this strategy, a player is guaranteed a victory unless the opponent can win on the next move. In our example, this means black has to already have either *threatening_five* or an *open_four* so he can win the game on the next move.

The game board situation shown in figure 1.2 has three consecutive white stones played with both ends open on column j, this is called an *open-three*. With this strategy a player, though not guaranteed a win, a response from his (her) opponent at one end, unless the

opponent can create a *threatening-five* strategy, since the opponent will lose the game in two more moves.

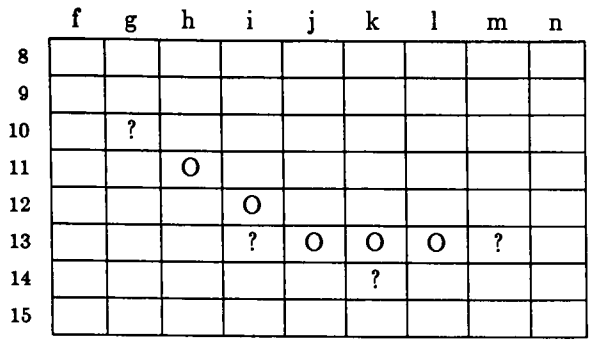


Figure 1.3a

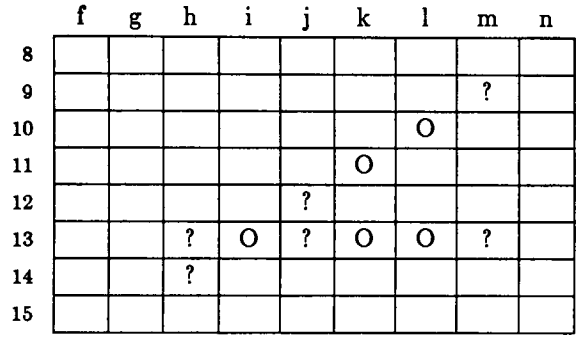


Figure 1.3b

Two *open-three* that cross each other (intersect each other) create a strategy called a crossing *open-three* or *intersecting-three*. This usually assures a win. As the examples in Figure 1.3a and 1.3b, white can win the game by expanding either one of the two crossing *open-three*. Assuming it is black's move. Whichever *open-three* he choose to block, white can choose the other to expand an *open-three* into an *open-four* and guarantee a win.

All of the above strategies are known to be means of winning a game and must be guarded against. A good player should know how to decide between offense and defense by judging whether any of these *potential-five* strategies must be met. In other words, he (she) should play a good offensive game, in which he (she) will attack the opponent as much as possible, but at the same time must know when a defensive move must be made to prevent an irrevocable losing situation from happening.

The program described in this thesis acts as one of the players; it plays against a human opponent. They communicate with each other through a CRT; the human player enters his moves via the keyboard and the program updates the game board on the screen. 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.

1.5 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 and how the game board is maintained. The third chapter describes the decision-making algorithm that the program uses to select the best move. Chapter four deals with the program's learning algorithm, which uses two different approaches in its learning process based on past successes and failures. Chapter five discusses the results of the program's learning. The last chapter gives conclusions, suggested improvements and future enhancements.

CHAPTER 2

EVALUATION STRATEGIES

2.1 OVERVIEW

This chapter deals with how the program evaluates each position and decides which move is the best one to make.

When it is the program's turn to make a move, it calls a function, *PGM_MOV*, to choose a best move. A best move, from the program's point of view, is the move with the highest value calculated from the polynomial evaluation function.

The polynomial evaluation functions is a linear polynomial which has the following form:

$$\text{value} = \sum_{i=0}^{n} t_i w_i$$

Each t_i is a term that acts as a simple, well defined game strategy. Each term is implemented as a function that returns a normalized value between zero and 25. The value of a term reflects the quality of the move, as perceived by that single strategy. For example, assume that a strategy indicates that it is better to place a move 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 form the center, the term's value will be proportionally less than 25.

Each t_i is a term that acts as a simple, well defined game strategy. Each term returns a normalized value between zero and 25. The value of a term reflects the quality of the move, as perceived by that single strategy. For example, assume that a strategy indicates that it is better to place a move 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 form the center, the term's value will be proportionally less than 25.

2.2 STRATEGIES

The following list describes the 17 individual game strategies used in the *Go-Moku* move evaluation polynomial:

1. *Four_directional_evaluation*

This evaluation involves looking up a value in the table given in figure 2.1. The table has an entry for each position on the board. The values in the table represent the number of ways that a stone in that position can participate in a 'win'.

Examples:

A stone in the exact center of the board can be the first, second, third, fourth, or fifth stone in a win, and there are 4 possible directions for that win (horizontal, vertical, and two diagonals). Since the nearer to the center position of the game board the better chance is to win a game, so every one square nearer to the center position get one more point incremented. This center position, therefore would be assigned the value 25.

Contrast this to a stone in the upper left-hand corner of the board, that it can only be the first stone in a win on one of three possible directions (horizontal, vertical, and one diagonal). This corner position, therefore, would be assigned the value 3.

3	4	5	6	8	8	8	8	8	8	8	8	8	8	8	6	5	4	3
4	6	7	9	11	11	11	11	11	11	11	11	11	11	11	9	7	6	4
5	7	10	12	14	14	14	14	14	14	14	14	14	14	14	12	10	7	5
6	9	12	15	17	17	17	17	17	17	17	17	17	17	17	15	12	9	6
8	11	14	17	20	20	20	20	20	20	20	20	20	20	20	17	14	11	8
8	11	14	17	20	21	21	21	21	21	21	21	21	21	20	17	14	11	8
8	11	14	17	20	21	22	22	22	22	22	22	22	21	20	17	14	11	8
8	11	14	17	20	21	22	23	23	23	23	23	22	21	20	17	14	11	8
8	11	14	17	20	21	22	23	24	24	24	23	22	21	20	17	14	11	8
8	11	14	17	20	21	22	23	24	25	24	23	22	21	20	17	14	11	8
8	11	14	17	20	21	22	23	24	24	24	23	22	21	20	17	14	11	8
8	11	14	17	20	21	22	23	23	23	23	23	22	21	20	17	14	11	8
8	11	14	17	20	21	22	22	22	22	22	22	22	21	20	17	14	11	8
8	11	14	17	20	21	21	21	21	21	21	21	21	21	20	17	14	11	8
8	11	14	17	20	20	20	20	20	20	20	20	20	20	20	17	14	11	8
6	9	12	15	17	17	17	17	17	17	17	17	17	17	17	15	12	9	6
5	7	10	12	14	14	14	14	14	14	14	14	14	14	14	12	10	7	5
4	6	7	9	11	11	11	11	11	11	11	11	11	11	11	9	7	6	4
3	4	5	6	8	8	8	8	8	8	8	8	8	8	8	6	5	4	3

Figure 2.1

2 *Neighbor_relation_evaluation*

The value for each position is calculated by looking in 4 directions from that position (which will be called X). The directions are assigned term values based on two criteria:

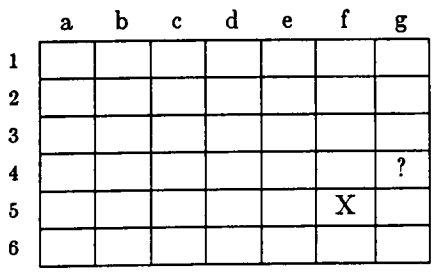


Figure 2.2a

1) *Block*: A block exists if there are no empty positions between X and the next opponent's stone or the edge of the board. Both sides of X in the given direction are tested for a block. (see example of a block at [4, e] in figure 2.2a and at [1, c] in figure 2.2b)

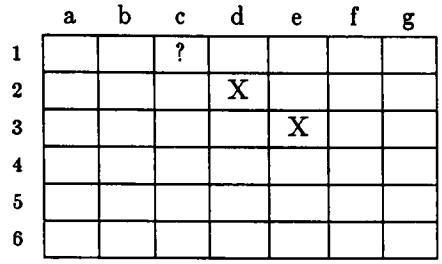


Figure 2.2b

If X is blocked on either side it is "closed", if not it is "open".

2) *No. of stones in a row* (if a stone were placed at X, how many consecutive stones would there be): Each direction is classified as open or closed and has a numeric value based on the number of consecutive stones (see example in figure 2.3, either a move on location [4, d] or [4, f] will give a value 3).

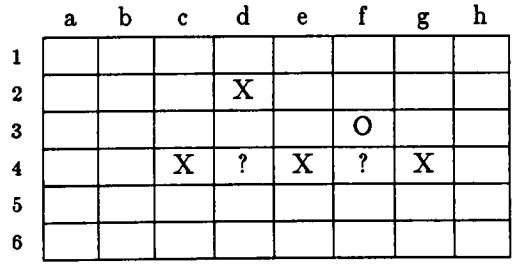


Figure 2.3

These are weighted by the following scheme:

open = number of stones * 2

closed = number of stones

The weighted totals for the 4 directions are added and this value is assigned to X.

3. *Block_opponent_evaluation*

This evaluation is similar to the above, except that it involves the opponent's position, that is, instead of calculating the weighted value for the program, it calculates the opponent's weight value which can be blocked if a move is made on this particular position. The term's value will be doubled the summation of the weighted values of the opponent stones in all four directions from that particular position's point of view. The [4, h] move shown in figure 2.4 blocks one opponent's stone in each of the north-east, east, and south directions and it blocks 2 opponent's stones in the west direction. So the value is ten (2 * 3 + 2 * 2).

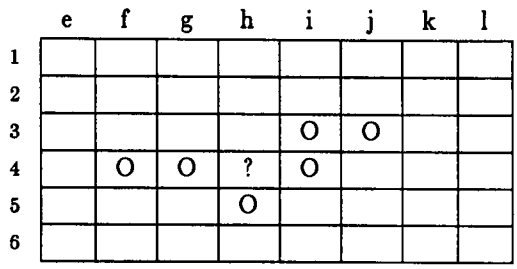


Figure 2.4

4. *Mattered_count_evaluation*

This term is evaluated by counting the number of directions its still possible to form five stones in a line from an unplayed position then by multiplying by 3. During a game some locations on the board might have neighboring opponent stones placed so that location can not be part of a row of five stones. Figure 2.5 shows that in either a horizontal or vertical direction from location [5, k], it is not possible to line up 5 stones.

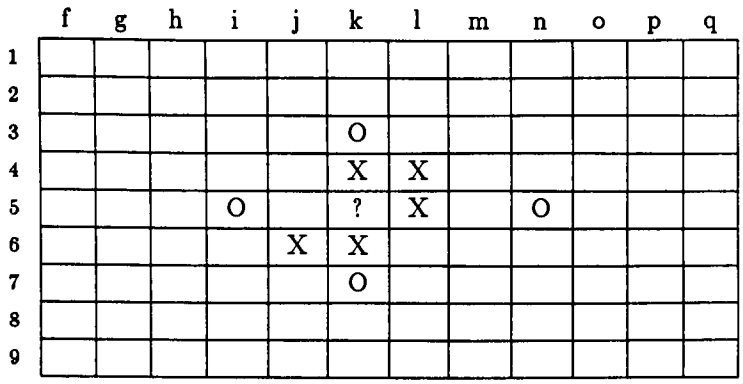


Figure 2.5

5. *Not_being_blocked_evaluation*

This term is evaluated a move by counting the number of directions that are not blocked and multiplying by 3. Potentially, a move should be made on a position which has more possibilities of developing five stones in a line is better than one with fewer. A move made on location [5, j] will get a value of 1, since there is only one direction not being blocked, the east-south direction.

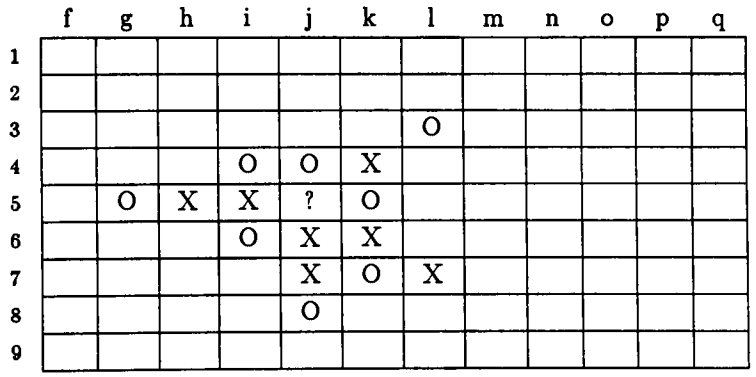


Figure 2.6

6. *Offend_win_move_evaluation*

This term is evaluated by checking to see if there is any move which can make five stones in a line when that move is played. The term value is 1000 if it is made and 12 if it is not yet made.

7. *Defend_win_move_evaluation*

This term applies to the opponent's stones instead of the computer's. An opponent's move

which can make an immediate five (a win) has a term value of 1000 if it is made and 12 if it is not yet made.

8. *Offend_open_3_evaluation*

This term and the following terms involve strategies which demands a response from the opponent except when the opponent has formed such or an immediate strategy (a strategy takes only one more move to win) by which he can demand a response from his opponent (program) as well. This is the case when three stones either consecutively or with only one unoccupied position among them line up in any one of the four directions (an *open 3*). This term values is 25 if it is made, or 12 if it is not.

9. *Defend_open_3_evaluation*

This term is similar to the above one, except it applies to the opponent's stones instead of the program's. The value indicates the importance of making a move to prevent the opponent from developing a threatening open-three strategy. This term value is 25 if it is made, or 12 if it is not.

10. *Offend_closed_4_evaluation*

This term evaluates the weighted values for a strategy of four stones (consecutive or or with an unoccupied position among them) lined up in any of the four directions with one end blocked. This strategy demands an immediate response from the opponent or a win will be made on the next move. This term value is 25 if it is made, otherwise 12.

11. *Defend_closed_4_evaluation*

This term is similar to the above, except it applies to the opponent's *threatening close-four* strategy in stead of the program's. The value indicates the importance of defending against this threatening strategy. The term value for this is 25 if made, otherwise 12.

12. *Offend_cross_3_evaluation*

These two open three strategies could be 1) two consecutive *open-three* intersecting with each other or 2) one consecutive open three intersects with a consecutive open-four with one stone missing (in which there are three stones cross 4 positions in one line). This values are 1000 if made, since it demands a response from its opponent if he hasn't formed any immediate winning strategy, and 12 if it is not made. (see example in figure 2.7, a move on location [5, i] makes 2 *open-threes* on two diagonal directions).

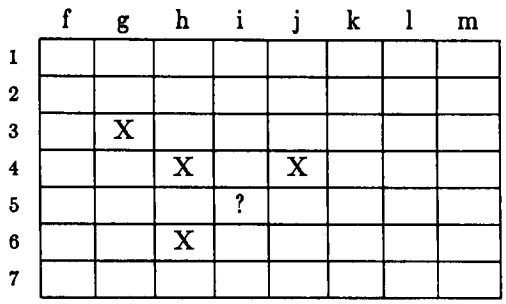


Figure 2.7

13. *Defend_cross_3_evaluation*

This term is similar to the above but applies to opponent's stones. Its term value is 1000 if it is made, 12 if it is not made.

14. *Offend_setup_evaluation*

This term demands a response from the opponent with no chance that the opponent can save the game from a lose. The term value is 1000 if it is made and 12 if it is not made. It could be either an open-three or a consecutive *open-four* with one stone missing in combination with either a *closed-four* or a *closed-five* with one stone missing. A move made on location [5, i] showing in figure 2.8 is an example of this.

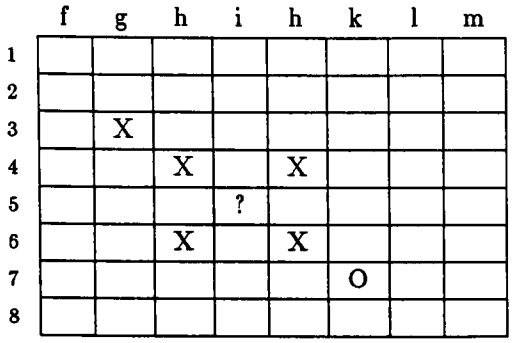


Figure 2.8

15. *Defend_setup_evaluation*

This term is similar to the above but applies to the opponent's stones. The term value is 1000 when it is made, 12 when it is not made.

16. *Offend_open_4_evaluation*

This strategies will guarantee a win if the opponent can't win in the next move. This occurs when a move can make 4 consecutive stones in a line with both ends opened (not being blocked). This term's value is 1000 when it is made, 12 when it is not made.

17. *Defend_open_4_evaluation* This term is similar to the above but applies to the opponent stones. The value is 1000 when it is made, 12 when it is not made.

CHAPTER 3

THE PROGRAM

3.1 OVERVIEW

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 bookkeeping functions. The second part includes the code that generates the program's moves and updates the game board. 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. (see Appendix A for a control flow chart)

This program was implemented in the C language under the Unix® operating system. The primary advantage of using C to implement this project is that C is a structured programming language which provides the ability to manipulate intricate data structures. In addition, there are several software tools available under Unix which simplify software design, and are designed to be used with C.

3.2 PROGRAM ORGANIZATION

The basic program initializes all global variables, sets up the game board on the screen, and determines whether a win has occurred after each player's moves. The algorithm of the program is shown in Figure 3.1.

® Registered Trademark of AT&T Bell Laboratories

Before each move, the program checks to see if anyone has won. If so, processing terminates; otherwise, depending on whose turn it is, the program either generates its move or reads the next move from the player. A series of checks is performed to verify that the human player's 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.

Each move requires updating the game board. This includes marking the move on the board using the current player's stone, generating coordinates for that move, and generating the neighborhood relation of the current move.

The neighborhood relation means locations not being played, but whose future importance is affected by the current move. The current player, as far as offensive approach is concerned, might make a move to generate more threatening strategies to the opponent. The importance of those newly generated threatening location should be gained. On the other hand, a move might take the defensive approach. A move may mean to block the opponent's strategies from developing, so the original opponent's threatening location(s) should be re-evaluated.

A valid move is made by playing a game piece at a location on the game board which is unoccupied. Players alternate make moves until one of the players wins.

```
While game is not over
    repeat
        get next move
        check validity of move
    until move is legal
    execute move
    update game board
Endwhile
```

Figure 3.1

3.3 PRIMARY DATA STRUCTURES

In order to make the program to 'see' what human players can see about the situation of current game board, several data structures are used to store information regarding current game board situation. Three two-dimensional arrays are used for this purpose.

One 19 by 19 matrix is used to represent the game board. Each matrix entry holds information indicating the status of the corresponding game board location (occupied or unoccupied). This matrix is used to update the display, to find empty locations for move making, and to determine the validity of a move.

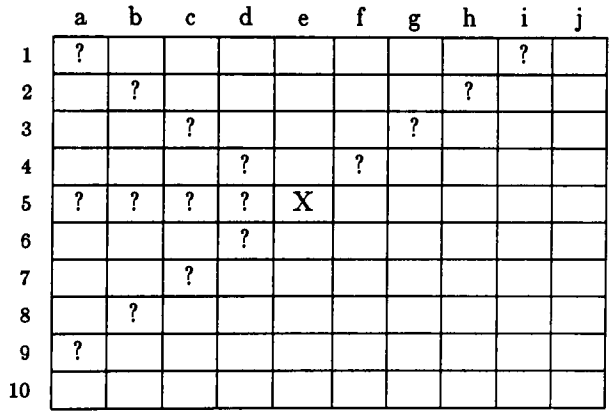


Figure 3.2a

Two matrices are used for game board maintenance, each one serving as an information keeper for one of the players. Each matrix holds information such as whether a location has been blocked within five spaces on both sides in some direction (which presents future development of five consecutive stones in that direction). In the examples shown in figure 3.2a and figure 3.2b, all locations marked with '?' are examples of this situation.

	e	f	g	h	i	j	k	l	m	n
1										
2										
3										
4										X
5									?	
6								?		
7							?			
8						?				
9	X	?	?	?	X					
10				?	?					
11			X		?					
12					?					
13					?					
14					X					

Figure 3.2b

The example shown in figure 3.2a has position [5, e] already played by black. All positions marked with '?' are not possible to form five consecutive stones in the direction pointing to [5, e]. These marked locations because of its one end already blocked by the opponent and can only be formed less than five stones in that direction, so no win is possible.

Figure 3.2b shows a similar example in which white has no way of forming five stones in the indicating direction for the shown range is less than five and already blocked on both ends.

This information is called *mattered count*, and is the number of directions which can still be developed into five stones consecutive. A location with no directions regarded as *mattered* has a *matter count* of zero.

The number of consecutive stones already played adjacent to each empty location in all directions is also kept. This information is used for calculating the new number of consecutive played stones for all affected locations once a new move is made. An example of this is shown in Figure 3.3. Supposed it is O's turn to make a move. A move at location [9, k] will change the information for location [7, m] from one "*adjacent unblocked in that direction*" into "*three*

consecutive stones to the south-west, blocked at the end".

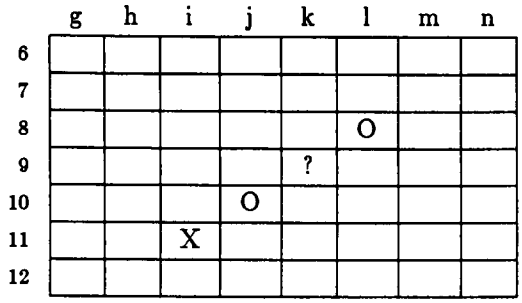


Figure 3.3

This example also shows that information such as whether the other end of a series of consecutive stones next to an empty location is blocked or not must be kept in the matrix.

All of this mentioned information is used to determine what strategies to generate once a move is made on a empty location. In addition, flags are used to represent potential strategies (such as *open_4*, *open_3* and *close_4*) to be attempted for every empty location in every direction. When a move is made to that location, these flags are used to determine what terms are affected by that move.

In order to traverse and update term values at the end of each game in the post-game learning mode, a linked list is used to keep the sequence of moves made during the game. Theoretically, the length of a game can't be predicted, assuming an unlimited number of stones for each players and no game board boundary; In order to store the sequence of moves, a linked list structure was chosen because of its flexibility.

3.4 PROGRAM EXECUTION

When it is the program's turn to make a move, a function *Find_move* is called to execute a search for the best move to make. This function looks at every possible move on the board. It assigns a value to each of these positions with higher values indicating better moves. The coordinates (X and Y) of the move with the highest value are then passed to a function which executes the move.

The Go-Moku program is made up of 7 conceptual operations. They are *INIT_SCREEN, INIT_GLOB, GENERATE_MOV, VISUAL_MOV, WIN_CHECK, STORE_MOV, UPDATE_MOV.*

INIT_SCREEN initializes the screen and prints out the rules for playing the game. Four execution 5 modes can be chosen: Modes one through three require for the program to play against a human, with the human player always playing first; mode four and five causes the program to play against itself.

VISUAL_MOV takes care of updating the game board on the screen. It displays messages about whose turn it is, the coordinates last move, the current cursor position, and whether there has been detected an illegal or a duplicate move. It also controls cursor movement and makes sure the cursor doesn't move out of the legal range for the game board.

INIT_GLOB declares and defines all global variables. These contain all information needed in deciding initial term values for every empty location. Two files held weight values for different strategies, one file is used for the fast learning approach, and the other for the post-game learning approach. Both files are created deriving the first run of the program with all 17 weights initialized to 100. During subsequent *learning* runs of the program all weights are adjusted from the values in the appropriate file, and; after each run, the appropriate file is updated. Destruction of these files causes the learning process to begin again from the initial weight values.

GENERATE_MOV generates the next move for the program to make. It calculates the value of all 17 terms for every unoccupied location on the board, and then uses these values to calculate sum of weights for each location by applying them to the polynomial function. After the calculation, the location that has the highest value is the move to be generated.

WIN_CHECK simply checks to see if either player won. If the program is running in a post-game learning mode, WIN-CHECK will be responsible for traversing moves made in the current round and updating each weight value.

After every move, *UPDATE_MOVE* places a stone on the board and updates the information kept for every affected empty location.

STORE_MOV maintains a linked list which is used to store, in sequence, terms values for every move played during the game. These terms can be checked to verify agreement with the outcome of the game.

3.5 MORE DATA STRUCTURES

The most difficult part of making the program see what a human player can see about the current board situation is reflected in the data structure for doing game board recognition. The approach to game board update used in this program is to concentrate on those empty positions that relate to either player's current move. These relative moves are within five positions from the move in all eight directions.

For each of the eight directions, up to two empty positions must be checked. This is because we are interested in any possible move which affects current strategies after it is played, and in empty locations which are affected by that possible move. Information about the number of stones played next to each empty location can be used to decide what locations are

important here. The following example shows those empty locations of interest (* indicates the proposed move).

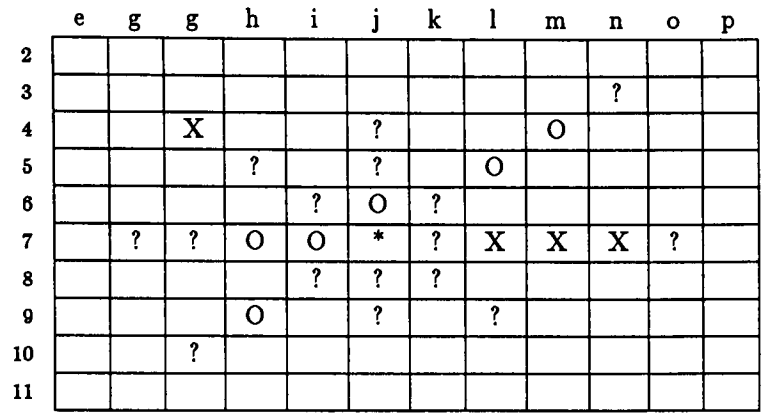


Figure 3.4

This game board update technique eliminates much unnecessary checking of empty locations. Updating of the information used in this method must be done for both offensive and defensive terms after each player's moves.

For each position to be checked, the program examines two pointers pointing to its inside and outside neighbors. Information stored in those neighbors (such as blocked flag, current stones played and mattered count) can be used to decide whether an old term is still set and whether new terms should be set for this checking position. After done with these appropriate term flag update, program sums and assigns the total terms values for each unplayed position.

Flags are used for each term without setting the terms value directly, because there are cross strategies (such as *cross_3_evaluation, setup_evaluation*) which may become important and there is no way of finding these terms until all eight directions for an empty position are checked.

CHAPTER 4

LEARNING

4.1 OVERVIEW

As stated in a earlier chapter, the move evaluation polynomial is made up of 17 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 initialized to 100 to show that no strategy is better or worse than any other strategy.

In the fast learning approach, a human player acts as a teacher while the program is learning. After a move made by the program, the teacher enters a score from -2 to 2 to indicate the quality of the move from very bad to very good. 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. If the value of the term agrees with the teacher's score (i.e. 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.

The fast learning algorithm determines which strategies are good and which are bad, and then raises or lowers their weights after each move made by program.

The traditional post game learning approach, which uses the same 17 terms to do its evaluations on each move, learns at the end of the game instead of learning after every move.

4.2 FAST LEARNING MODE

In this mode a human player acts as the program's teacher while it is learning. After each move made by the program, the human player will be asked to enter a score ranging from -2 to 2. By doing this the program then can use this rating score to adjust its weight values. The program compares this score to each term value used in the polynomial when the program evaluated this move to determine how good or how bad the move was. Some of these terms have high values, indicating that the move was good, and some have low values, indicating that the move was bad.

When one term value agrees with the teacher's score (i.e. 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. For example, assume a move scored by the teacher is very good (a score of 2) and term X gave that move a group value of 2. This means that the evaluations were consistent. Term X, then, should be worth more relative to other term values, and this term should be rewarded.

On the other hand, if term Y evaluated the same move as very bad, then the inconsistency between the program and the teacher should result in a punishment to term Y.

This approach was done by applying the following method. The value of each term can be categorized into 5 groups, as shown in Figure 4.1.

Term Value	Group Number
0 ~ 4	-2
5 ~ 9	-1
10 ~ 14	0
15 ~ 19	1
20 ~ 1000	2

Figure 4.1 Grouping Table

In either the reward or punishment case, the weight value corresponding to that particular term will be increased or decreased. The amount of an increase is determined by the following algorithm.

A zero group value means neutral; a positive group value means a good move; and negative group value means a bad move. For every term the adjusted score is the product of the group value and the score entered by the teacher. The maximum positive adjusted score, then can be 4, and the maximum negative adjusted score can be -4.

When a group value indicates zero, it would not be affected by teacher's score, whatever is entered, since zero multiplied by any number still is zero. Our purpose in this case is to make weight values of terms showing that are irrelevant to certain game board situations remain unchanged. A move with a score of 2 (maximum positive score) entered by the teacher which has a consistent term group value of 2, will be increased by 4. If a term evaluates this move with a -2 group score will be decreased by -4. This adjustment is to reward terms of making good judgement of moves and to punish terms of making bad judgement. After many of such kinds of adjustments are made through game playing, the weight value of each individual term reflects its importance relative to the others in the move making decision.

4.3 POST GAME LEARNING MODE

In this mode the evaluation algorithm will be the same one used by the fast learning algorithm. The difference is that the human player does not act as a teacher for the program. Instead of learning after each move, program learns at the end of a game. The outcome of a game becomes the standard used in deciding whether all the moves in a game are good or bad For a winning game, the program learns by increasing weight values for all terms that were in favor of any move made during the game, while decreasing weight values for all terms disagreed with any move made during the game.

If the game was lost, all terms in favor of any move made during the game will have their corresponding weight values decreased, terms that disagreed with the moves would have their weights increased. The amount to be decreased or increased is the same as for the fast learning algorithm mentioned before.

Drawback of this learning approach is that some moves in a game that was won may not be really good, or even could be bad moves, but they will all be judged "good" as long as the game was won. But the same token, good moves in a losing game, will suffer negative adjustments only because the game was lost.

Generally speaking, this approach takes more time to learn than the fast learning approach. However, the final results of both modes of learning should be of the same.

4.4 MODE 3 AND MODE 5

Mode 3 and mode 5 are designed for the computer to play with a human player in order to see how well computer learning has been performed. Mode 3 does not allow any learning. In this mode all weight values used to do move evaluation are values left over from fast learning mode or all weight values are 100 when no fast learning mode was played before.

Mode 5 also does not perform learning. Weight value used in this mode are left from last post learning game. If no post learning game being played before all weight value of 100 are used.

4.5 MODE 4

Mode 4 is a mode in which program will play against itself. There is just one (and still the same one used for mode 1 and mode 2) polynomial function being used for both non-human

players. This mode is like mode 2 for using post game learning approach. Both computer players use the final weight values left from last post game learning mode.

The adjustments for the weight value are made for the second player, which is the one using black stones.

CHAPTER 5

RESULTS

5.1 OVERVIEW

In this chapter we deal with the performance of the program for the game of *Go-Moku*. We examine the learning abilities of the program along with how fast and how much improvement the program can make throughout each game played.

The performance of the program's learning relies very much on how good a human player it plays with in a game. If the program encounters a player who knows how to play the game quite well (in post game learning mode) or who knew how to be a good teacher (in fast learning mode), then the program will learn rapidly, otherwise it will learn relatively slowly.

5.2 EFFECTIVENESS OF STRATEGIES

In this section we discuss how the importance of each term is changed in an overall scope. Figure 5.1 shows the sum of the 17 terms for every unplayed location after the very first move is made. The human player will always play first, and normally he(she) will choose the center position of the game board to start with because of the potential future development from that position. Here we used H to stand for a stone played by the human player, and C is used to stand for computer's stones.

45	46	47	48	56	56	56	56	56	56	56	56	56	56	56	48	47	46	45
46	48	49	57	59	59	59	59	59	59	59	59	59	59	59	57	49	48	46
47	49	58	60	62	62	62	62	62	62	62	62	62	62	62	60	58	49	47
48	57	60	63	65	65	65	65	65	65	65	65	65	65	65	63	60	57	48
56	59	62	65	68	68	68	68	68	68	68	68	68	68	68	65	62	59	56
56	59	62	65	68	69	69	69	69	69	69	69	69	69	68	65	62	59	56
56	59	62	65	68	69	70	70	70	70	70	70	70	69	68	65	62	59	56
56	59	62	65	68	69	70	71	71	71	71	71	70	69	68	65	62	59	56
56	59	62	65	68	69	70	71	73	73	73	71	70	69	68	65	62	59	56
56	59	62	65	68	69	70	71	73	H	73	71	70	69	68	65	62	59	56
56	59	62	65	68	69	70	71	73	73	73	71	70	69	68	65	62	59	56
56	59	62	65	68	69	70	71	71	71	71	71	70	69	68	65	62	59	56
56	59	62	65	68	69	70	70	70	70	70	70	70	69	68	65	62	59	56
56	59	62	65	68	69	69	69	69	69	69	69	69	69	68	65	62	59	56
56	59	62	65	68	68	68	68	68	68	68	68	68	68	68	65	62	59	56
48	57	60	63	65	65	65	65	65	65	65	65	65	65	65	63	60	57	48
47	49	58	60	62	62	62	62	62	62	62	62	62	62	62	60	58	49	47
46	48	49	57	59	59	59	59	59	59	59	59	59	59	59	57	49	48	46
45	46	47	48	56	56	56	56	56	56	56	56	56	56	56	48	47	46	45

Figure 5.1

From this figure we can easily tell how dominant term 1, the *4_directional_evaluation* is, for it advocates that the nearer a move is to the center position of the game-board, the better the move. Since there has been only one stone played, no other term but term 3 (which is *block_opponent_evaluation*) increased most of its term value. Actually term four, *not_being_blocked_evaluation*, has some negative effect on the sum after this first move, whereas this effect is compensated for by term 2, *block_opponent_evaluation*, because any of the program's moves made on positions right next to the center stone can block the opponent's stone in one direction.

To explain this further, term four (*not_being_blocked_evaluation*) proclaims that a following move of the program should stay away from where its opponent just played, thus from an offensive point of view, the program can get a better chance of forming 5 stones in a row. Yet term 2 would advocate making a move as close to the opponent's move as possible to make

the opponent's situation less threatening.

Before the program's first move, term 2, *neighbor_relation_evaluation*, has a negative weight value on every unplayed location because no move has its neighbor position already played. Opposite to this, term 4, *matter_count_evaluation*, decides that most empty positions, except positions near to the corners of the game board, are great moves to make because there are possible developments into 5 consecutive stones in all 8 directions.

So far, all terms mentioned have a decisive influence on moves made at the beginning of a game. Terms 5 through 16 remain neutral (term value 12) on every empty position for their irrelevancy at this moment (or during the early part of a game).

Not until the opponent of the learning program develops any defensive strategy among term 5 through term 16 will the program decide a move according to that strategy. Because of this nature of the terms during the early learning games, the program tends to play sparsely on the game board.

As the game proceeds, the importance of term 1 (*neighbor_relation_evaluation*), term 2 (*block_opponent_evaluation*) and most offensive strategies among terms 5 through term 16 increase their weights more than the rest of terms, and the program's played stones turn out to be more intensively on the game board. This is because the relative importance of these terms increases among played stones, and making moves in this manner provides more chances of developing threatening strategies or being forced by the opponent to make defensive moves, such as a move to block the opponent's strategy.

5.3 POST GAME LEARNING

The post game learning approach is really much simpler than the fast learning approach;

however this doesn't mean its learning is faster than the other. The teacher (or here it really means the human player, for no third party is involved in a game) who knows how to win a game quite well, should first try to play the game as 'conservatively' as possible in order to emphasize the importance of terms designed to be met during the game. In other words, the human player should not play the best game that he (she) can play at the beginning. Thus the computer can reward some terms when it wins a game. Appendix B, game 2 shows threatening strategies such as terms 7, 8, 9, 10, 13, 14 and 16 gain their weight values pretty much because of the number of their occurrences in a win.

Before all 17 weight values reach what expert judgement can really be based on, a human player should try not to win a game, except when the program plays really absurd moves that can only be improved by causing punishment to those moves in a losing game. The learning process will be slowed down for a case like this, because any good moves will be offset in a losing game.

Human player can adjust terms values from program's previous learned if he (she) wins. The occurrence of any terms depends very much on what moves the human player made, thus affecting the computer's following move.

Our experiments on the post game learning mode show satisfactory results after playing with a teacher for several rounds. Most weight values were adjusted drastically.

Also from the values of weights, which are adjusted after each post game learning mode, we can see that term 5 (*offend_win_move_evaluation*) and term 6 (*defend_win_move_evaluation*) tend to gain their importance slowly, because both are met just once in each round of a game (see Appendix B game 1 and game 2, either term 5 or term 6 only appears once in each game). Similar situations happened to other terms (see table 1, final results among all modes) such as term 11 (*offend_cross_3_evaluation*) and term 12 (*defend_cross_3_evaluation*) which happened fewer times than term 7 through term 10 for their degree of threatening and difficulty of

creating (in terms of more stones and more chances of developing from existing orders of played stones).

As we said before, *Go-Moku* is very player dependent, particularly in the learning approach we used here. A human player can pretty much control the game development when the program plays a bad game. This is because the program is not intelligent enough to threaten a human player at the beginning of its learning process, and the human player can prolong a win to give the program more chances of developing various terms.

An intelligent human player should try to lead the program to make offensive terms met as many times as possible. This is done by staying away from possible defensive moves, allowing the program to occasionally have threatening terms met. By doing so, the number of occurrences of each term be accumulated and delayed until the end of the game to update weight values.

In order to stimulate the program to recognize the importance of defensive terms, the human player has to emphasize them by winning a game when the occurrence of defensive terms are ignored by the program. Overall, the importance of each term relies very much on its occurrence in a game.

5.4 FAST LEARNING

In the fast learning mode there are not too many difference compared to post game learning other than the human player has more control abilities on the importance of immediately occurring terms than in post game learning mode.

Since in this mode the human player will be asked to enter a rating representing the quality of a move presumed by the teacher each time after a move made by the computer in

order to adjust the goodness of that move, the computer has a chance to gain better quality terms right after every move.

Comparing learning results (see appendix B and C) between the two learning modes, we found that in the fast learning mode the computer learned much faster than in post game learning.

Because of the shortened the period of adjustment, a stopped point where learning can be performed, the negative learning effect is minimized. In the post game learning approach, when a game is won, all terms in favor of moves played in that game are rewarded equally by assuming every move made during the game was equally good.

Because the post game learning adjusts weight values based solely on the result of the game, its fairness is the main factor to slowing down its learning pace. In the fast learning mode a good teacher will know to reward agreed terms only those good moves made during the game and punish only those disagreed for that move.

Learning in fast learning mode might not be more successful than in post game learning mode for a human player not being a good teacher. The truth is that in this mode, the computer senses more adjustments being made before because knowledge accumulated from new weight values is done right after each move, which affects later moves decision making comparatively more.

5.5 COMPUTER VS. COMPUTER GAME

In a computer versus computer game, both computer moves are generated using the same polynomial evaluation function, in which same all 17 terms are used to decide the quality of each move. Since there is no human player involved in this mode, no intermediate interruption,

such as rate scoring used in fast learning, is used.

Our experimental trial of running the game in this mode showed that, because of term 0, term 3 and term 4 are pretty decisive in first 10 moves, both computer players making moves to stay away from their opponent. Until both players ran out of space avoiding moves right next to each other, term 1 (*neighbor_relation_evaluation*) and term 2 (*block_opponent_evaluation*) became more decisive in those following moves. (see appendix D)

The emphasis of term 1 and term 2 cause terms 5 through 16 have better chances of being met. Even so, neither computer player seems to win the game by simply developing from occasional occurred threatening strategies met among term 5 through term 16. This is because the best move searching is found by both player by using same algorithm. Besides, there are no human negligent factors to overlook any occurred strategies.

The first game took about 40 moves to decide the first player's win. In this game, defensive terms decided most of the second computer player's (our learning computer) moves. This is not solid enough to play a good game. All terms in favor of offensive moves were lowered pretty much (see appendix D-1).

The second game took 141 moves to decide, and the first player again has the winner (Appendix D-1, game 1) This time the weight values changed quite a bit, and the relative importance of terms differed from the first game. Up to the 5th game second computer was still a loser and lost in no more than 20 moves because of the lengthy second game lowered the weight values for good terms too much (Appendix D-1, game 1-5).

Finally the computer won the sixth game (Appendix D, game 6). Up to the eighth game, second computer remained the winner. Game nine was a fatal game in the whole learning process. In this game it took 120 moves for the first computer to be the winner again. Overall, lengthy games are destructive rather than constructive because under the post game learning

approach, no matter what terms are used the importance of all will be lowered greatly only because a lose happened.

Ultimately, in the ninth game and those thereafter (Appendix D, game 9), the weight values stablized and the program's learning reached a plateau. After this point they all played the same game because about half of the 17 terms were already diminished to our pre-defined minimum weight value and there was no chance for the computer to bring about a radical change in this situation with no win in the learning computer side.

Table 1. final weight value of all modes

term	mode 1 (fast)	mode 2 (post)	mode 4 (post, computer vs. computer)
0	464	532	2
1	1924	1720	368
2	320	276	224
3	312	346	2
4	146	182	2
5	340	476	108
6	272	326	4
7	282	266	4
8	164	192	4
9	284	302	4
10	176	210	4
11	178	164	100
12	138	148	56
13	192	188	100
14	136	124	64
15	314	324	104
16	282	252	16

CHAPTER 6

FUTURE ENHANCEMENTS

6.1 OVERVIEW

This chapter deals with improvements that can be made to our current version of program.

6.2 STRATEGIES ENHANCEMENTS

Beyond the 17 experimental strategies which we used in a polynomial function, there should be more strategies and further more purified strategies can still be derived from our current version of *Go-Moku* program. These enhancements are worth developing for added accuracy and removing ambiguities. Specifically, more and better strategies could be implemented since our current version does learn, but still does not play an expert game.

Terms 5 through 16 can all be broken down into more detailed or simpler strategies, so resulting in more accurately and have less chance of offsetting the weight value of good strategies. Our current learning scheme needs to make more distinctions in the case of certain existing strategies, in order to improve on what program can now learned, otherwise learning can only be performed in a fairly restricted manner.

For example in term 7 (*offend_3*) we regard 3 consecutive stones in a line with both ends opened (supposed this is case A) as the same as 2 consecutive stones followed by one space and one stone with its both ends not being blocked (supposed this is case B). In a losing game, no matter whether the moves are of case A or of case B, when using the post game learning approach both get punishment instead of reward.

Case A moves can produce a very good strategy and their importance should never be decreased because of its similarities with others like case B that are not good. The same analysis applies to other terms.

Also new strategies no covered by any of our existing strategies should be derived (if there are any). However, whether learning can be improved by adding new strategies can only be established through experimentation.

Discoveries like these can be gained through experimentation. Better strategies scheme is possible to be uncovered when taking more factors into consideration.

6.3 LOOK-AHEAD

In order to decide the best move to make, our current method evaluation is limited to a weight value being assigned to every empty position. We multiply the term value on each unplayed position with the corresponding weight value, and used the sum of these products to decide best move.

Our current polynomial evaluation function misses lost a lot of good moves. This is because some moves develop to become good, and can only be shown after several moves played. Our current version of *Go-Moku* program lacks the abilities to determine if the next few moves can lead to a win or that the opponent's next few moves can force a lose.

Mini-max searching is a typical look-ahead algorithm. Slagle [SLAG 71] has a good discussion of the details of mini-max. This algorithm searches the game tree and determines the branch that will offer the opponent minimum opportunities while giving maximum benefits to the player searching for a move. However, 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. With a look-ahead algorithm, more moves are taken into account and the program should be able to improve its play considerably.

The major problem which arises when using a look-ahead algorithm is that the amount of time needed to evaluate moves could be extremely large [HUBE 83]. Since *Go-Moku* is played on a board consisting of 361 positions the time needed to do tree searching for each of the unplayed positions is enormous.

What we can do is narrow down the scope of the look-ahead. In other words, instead of doing tree searching for every positions, we can choose just those moves that are more likely to develop into useful or unbeatable strategies.

For instance, the current algorithm chooses a highest value assigned directly in our current game board. A revised version with look-ahead algorithm can choose 10 or 20 candidates, and for each of these create its own separated tree containing the next possible 10 to 20 candidates. We can even go down more levels from there, but this could become so complicated that it takes too much time to be practical.

Such a look-ahead strategy is based on the premise that moves which seem ordinary on the surface might lead to good and unforeseen situations. Such moves may be overlooked by the current program. While the time needed for a conventional look-ahead is prohibited, using fewer candidates chosen by means of the evaluation function could still prove useful. Since the win-or-lose in a *Go-Moku* game is so dependent upon one or two current moves and not so dependent on the length of the game, one or two levels of searching could be good enough to assure the best move is made.

By applying one or two levels of look-ahead, we can have a more thorough determination of potentially good moves. Whether adding this look-ahead will uncover enough good moves to make a difference can only be proven by experimenting with a revised *Go-Moku* program.

6.4 MULTIPLE FUNCTIONS

Usually *Go-Moku* game players tend to start by placing their moves near to the center of the board and stay closely to that central block area. One interesting thing that may happen is that when both players play with comparative equal skills and all intended strategies are detected and stopped by the opponent, one may move away from that fighting area.

Berliner [BERL 80] addresses this by making two classes of game situations, each class with 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.

This raises another possible extension that can be implemented. During the beginning of a game, since it is too early for some threatening strategies to be implemented, so we may want to have a special function to emphasizes other strategies more important to the opening game. As the game develops or when a certain situation occurs in the game, we may then use other functions to do move evaluation.

One problem that arises is when to decide which function to use. A game managing routine containing standards for deciding this should be installed to activate special function calls and then return to the normal function for future control flow.

BIBLIOGRAPHY

J. E. Hayes, D. Michie and Y. H. Pao (1982). Machine Intelligence, Halsted Press, New York, 1982.

Patrick Henry Winston (1984). Artificial Intelligence, second edition, Addison-Wesley Publishing Co.

Chanf-Chung Yang (1983). An Artificial Intelligence Approach To The Game of Backgammon, R. I. T. Master Thesis, Jan. 1983

James R. Huber (1983). Computer Game Playing, The Game of Twixt, R. I. T. Master Thesis, March 1983 Lasker, E. (1960). Go and Go-Moku, Dover Publications, New York, 1960.

Weizenbaum, J. (1962). "How to Make a Computer Appear Intelligent; Five-in-a-row offers no guarantee", Datamation, 1962.

P. F. Huebner (1977). A Methodology for Deriving a Weighted Stratic Evaluation Function for Go-Moku, University of Kansas, M.S. thesis.

Brian W. Kernighan and Dennis M. Ritchie (1978). The C Programming Language, Bell Telephone Laboratories, Incorporated.

Koniver, Deena (1962). "Computer Heuristics for Five-In-A-Row", Master's Thesis, Department of Mathematics, MIT, Cambridge, Massachusets, June 1963.

Nilsson, Nils J. (1980). Principles of Artificial Intelligence, Tioga Publishing, Palo Alto CA

Murray, A. M. and E. W. Elcock (1967). "Automatic Description and Recognition of Board Patterns in Go-Moku", Machine Intelligence, Vol. 2, E Dale and D. Michie, eds., American Elsevier Publishing Company, Inc., Edinburgh, 1967.

John Dixon (1969). "Experiments with some programs that search game trees", Journal ACM, Vol 16, April 1969.

F. Carlson (1973). "An Advice-taking Chess Computer", Scientific American, June 1973.

Chang, Chin-Liang, and Richard Char-Tung Lee (1973). Symbolic Logic & Mechanical Theorem Proving, Academic Press, New York, 1973

Berliner, Hans, "Computer Backgammon", Scientific American, June, 1980

Slagle, James R., Ph.D. (1971). Artificial Intellilgence: The Heuristic Programming Approach, McGraw Hill, New York.

Winston, P. H. (1977). Artificial Intelligence, Addison Wesley, Reading MA.

Fast Learning Mode -- Game 1

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5							O												
6																			
7					X			O	X	O	O	O	X	X					
8						O	O	X	X	X	X	O	X						
9						X	O	O	O	X	X	X	X	O					
10							O	O	X	O	X	O	O						
11							X		O	X									
12									O	X									
13																			
14																			
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 1

Weight No.	Weight Value
0	134
1	95
2	78
3	174
4	97
5	104
6	100
7	114
8	118
9	110
10	118
11	106
12	102
13	102
14	104
15	104
16	110

Fast Learning Mode -- Game 2

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1											X	X							
2										X	O								
3										O	O								
4									O	X	O	O	O						
5								O	X	X	O	X	O			O			
6							O		O	X	X	X	X	O	X		X		
7								O	O	X	O	X	X	X	X	O			
8						X	O	X	X	O		O	X		O				
9						O	X	O	O	O	X	X	X	O					
10					X		O	X	X	O	O	X	O						
11									O	X	O	X							
12												X							
13												O							
14																			
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 2

Weight No.	Weight Value
0	150
1	118
2	86
3	156
4	112
5	104
6	104
7	130
8	122
9	124
10	140
11	100
12	100
13	112
14	118
15	100
16	116

Fast Learning Mode -- Game 3

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5																			
6																			
7									X										
8									X	X	X	O							
9											X	X							
10							X	O	O	O	O	X							
11											O	O	X	O					
12											X	X	O	O					
13											O								
14																			
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 3

Weight No.	Weight Value
0	242
1	65
2	56
3	225
4	182
5	116
6	114
7	144
8	152
9	122
10	118
11	108
12	114
13	102
14	104
15	110
16	124

Game 1 in Post Game Learning Mode

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4						X			X										
5					X	O	O	O											
6			O		O	O	O	X			X								
7				X	O	O	X	O	X	O	X	O							
8				X	X	O	X	X	X	X	O	X							
9					O	X	X	X	O	O	O	X	X						
10						O		O	X	O	O	X	O	O					
11							O		O	O	X	O							
12						X				X									
13									X										
14																			
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 1

Weight No.	Weight Value
0	160
1	70
2	80
3	150
4	134
5	104
6	100
7	116
8	124
9	104
10	120
11	104
12	104
13	100
14	104
15	104
16	104

Game 2 in Post Game learning mode

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4								O		O									
5						O	O		X										
6						X	X	X	O	X									
7							X	X	O		X								
8						X	O		X	O	X	X							
9					O		O		O	X	O	X	O						
10				X	O	O	O	O	X	O	O	O		X					
11						X	O	X	O	O	X	O	X						
12				O	X	X	X	X	O	X		X							
13					O	O		X			O								
14																			
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 2

Weight No.	Weight Value
0	224
1	40
2	60
3	206
4	168
5	108
6	104
7	128
8	136
9	104
10	132
11	108
12	112
13	100
14	104
15	108
16	120

Game 3 in Post Game Learning Mode

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2							X												
3								O											
4									O										
5				X			O	O		O	X		X						
6		O			O		X	X	X	O	O	O							
7			X	X	O	O	O		X	X	O	X							
8				X		X	O	X	X	O	O	X							
9			X	O		X	X	O	O		O								
10				O	X	X		X	O	O	X								
11					O	X			X	X	X	O							
12						O	X			O	O								
13					X	O	O	O	O	X									
14								X											
15																			
16																			
17																			
18																			
19																			

Weight Value Table after Game 3

Weight No.	Weight Value
0	60
1	122
2	120
3	60
4	72
5	100
6	92
7	96
8	80
9	100
10	100
11	100
12	96
13	100
14	100
15	100
16	92

Game 1

Weight No.	Weight Value
0	48
1	160
2	156
3	36
4	38
5	100
6	96
7	96
8	88
9	100
10	96
11	100
12	96
13	100
14	100
15	100
16	92

Game 2

Weight No.	Weight Value
0	2
1	206
2	162
3	2
4	2
5	100
6	48
7	80
8	52
9	36
10	36
11	100
12	96
13	100
14	96
15	100
16	68

Game 3

Weight No.	Weight Value
0	2
1	234
2	186
3	2
4	2
5	100
6	40
7	64
8	24
9	28
10	32
11	100
12	88
13	100
14	92
15	100
16	64

Game 4

Weight No.	Weight Value
0	2
1	256
2	204
3	2
4	2
5	100
6	40
7	60
8	12
9	28
10	32
11	100
12	84
13	100
14	92
15	100
16	48

Game 5

Weight No.	Weight Value
0	2
1	278
2	218
3	2
4	2
5	100
6	32
7	56
8	4
9	28
10	32
11	100
12	76
13	100
14	92
15	100
16	44

Game 6

Weight No.	Weight Value
0	66
1	300
2	230
3	60
4	34
5	104
6	20
7	64
8	12
9	32
10	32
11	100
12	64
13	100
14	92
15	104
16	56

Game 7

Weight No.	Weight Value
0	190
1	250
2	210
3	182
4	108
5	108
6	60
7	80
8	32
9	44
10	44
11	100
12	64
13	100
14	100
15	104
16	80

Game 8

Weight No.	Weight Value
0	4
1	296
2	230
3	2
4	2
5	108
6	28
7	32
8	4
9	4
10	4
11	100
12	60
13	100
14	88
15	104
16	48

Game 9

Weight No.	Weight Value
0	2
1	368
2	224
3	2
4	2
5	108
6	4
7	4
8	4
9	4
10	4
11	100
12	56
13	100
14	64
15	104
16	16

Game 1

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3						X						O							
4										O									
5							X						O						
6					X											X			
7			X							X	X								
8							O	X	X	O									
9				O				O	O	O	X	O							
10									O	O	X						O		
11								X		O					O				
12										X	O								
13					X														
14			X													X			
15						O							X						
16									O										
17							O							X					
18																			
19																			

Game 2

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5							X	O		O									
6					X	O	O	X	X										
7					X	X	O	X	O	X									
8					O	X	X	X	X	O	X								
9						X	O	O	O	O	X	O							
10					O	X	O	X	O	O	O	X							
11		O		X	X	O	O	X	X	X	O	X	X	X	X	O			X
12		O		O	X	O	X	O	X	O	O	X	O	O	X				
13	X	O	O	O	O	X	X	O	X	O	O	O	X	X	O				
14	O	O	X	O	O	X	O	O	O	X	X	O	O	X	X				
15		O	X	O	X	X	X	O	X	X	O	O	X	X	O				
16		X	O	X	O	X	X	X	X	O	X	X	O	X					
17		X	O	O	O	O	X	X	X	X	O			O					
18					X	O	O	O	X	O									
19								X	O	O									

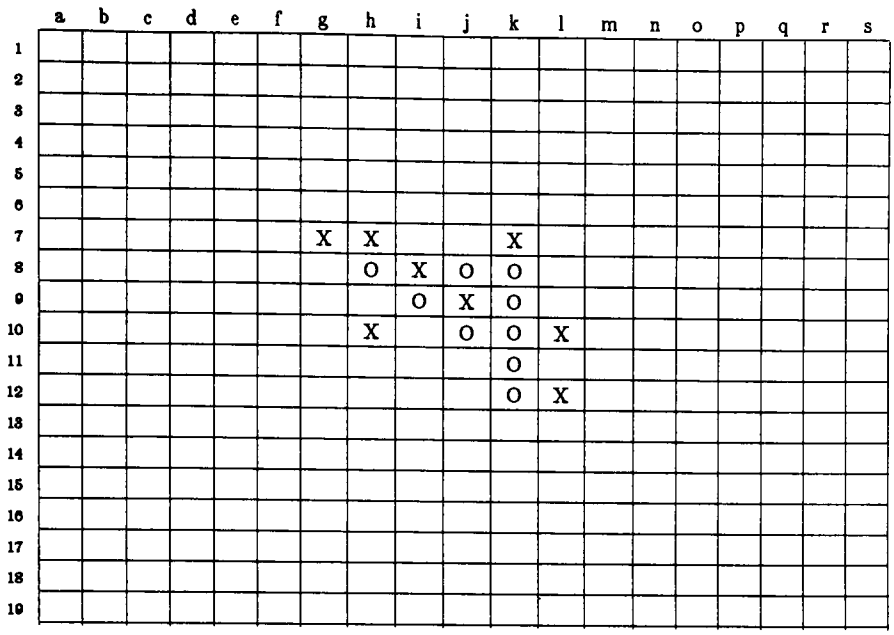
Game 3

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5																			
6																			
7							X	O	O	O		X							
8						O	X	X	X	O		X							
9						O	O	X	O	X	X	O							
10						X	X	X	O	O	O	O	X						
11						O	O	X	X	X	X	O							
12								O	X			O							
13												O							
14																			
15																			
16																			
17																			
18																			
19																			

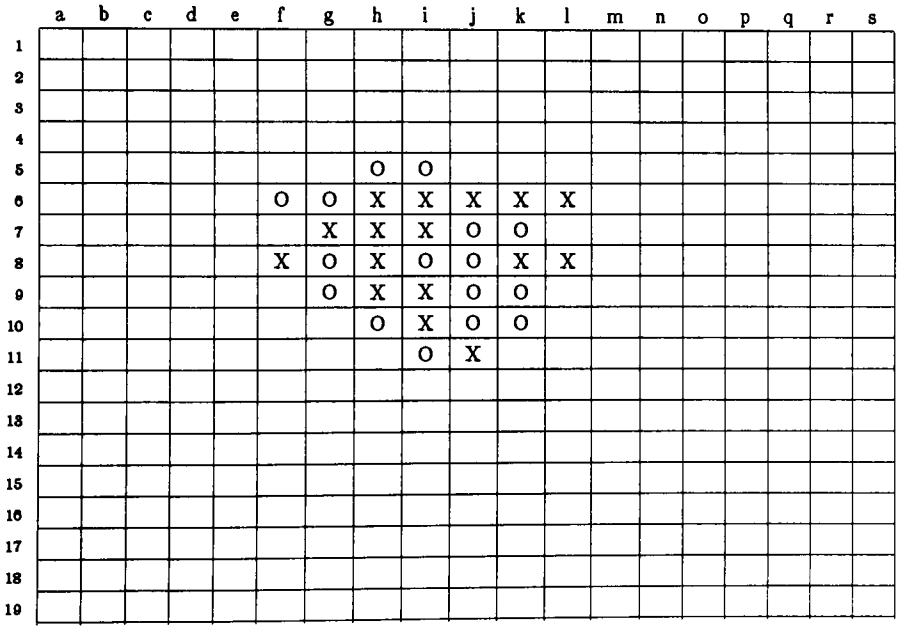
Game 4

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5																			
6																			
7								O	O	O	X								
8								X	X	O	X	X							
9								X	O	X	O								
10							X	X	O	O	X	X							
11							X	O	O	O	O	O							
12																			
13																			
14																			
15																			
16																			
17																			
18																			
19																			

Game 5


a b c d e f g h i j k l m n o p q r s
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
X X
O X O O
O X O
X O O X
O
O X

Game 6


a b c d e f g h i j k l m n o p q r s
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
O O
O O X X X X X
X X X O O
X O X O O X X
O X X O O
O X O O
O X

Game 7

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2										X		X							
3										O	O								
4									X	O	X								
5								X	O	O	X								
6					X		X	O	O	O	O	X							
7						O	X	O	O	X	X	O	X						
8						X	O	O	X	X	X	X	O	X					
9						X	O	O	O	X	X	O	X	X	X				
10						X	O	X	O	O	X	O	O	O					
11						O	O	X	O	X	O	O							
12						O	X	X	X	O		X							
13								O											
14																			
15																			
16																			
17																			
18																			
19																			

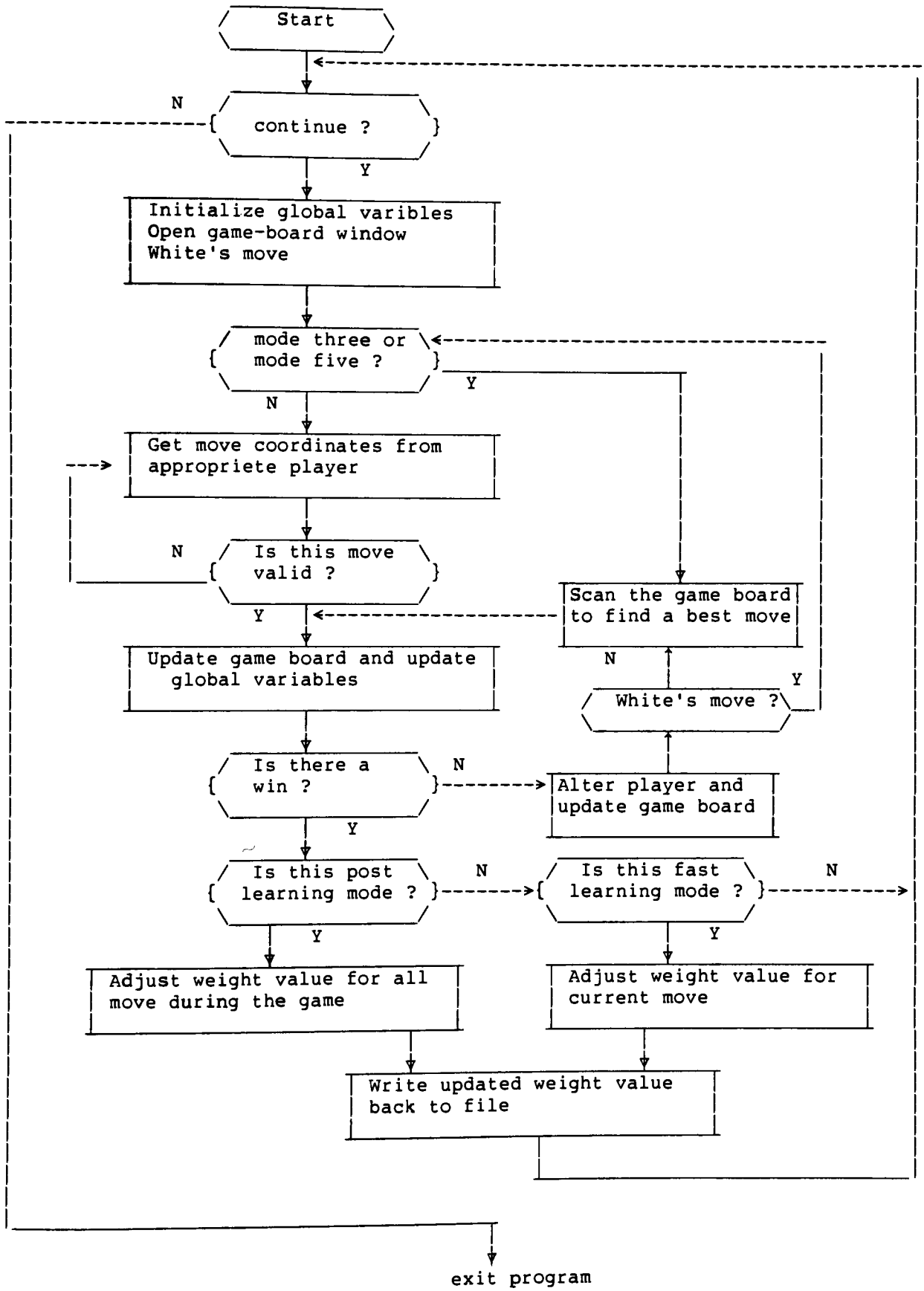
Game 8

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1						O													
2						X	O	O											
3						X	O	X	X		O	O							
4						X	X	O	X	X	X	O	O	O		X			
5					X	X	O	O	O	X	X	O	X	O					
6					O	O	X	O	O	O	X	X	X	X	O	O			
7							X	O	O	O	X	O	O	O	X	O			
8						O	X	X	X	X	O	X	X	X	X	O	X		
9					X	X	X	O	O	O	X	X	X	X	O	X	X		
10					X	O	O	X	O	O	O	X	O	O	X				
11					O	O	X	X	X	O	O	X	X	O					
12					X	X	O	X	O	O	X	O		O					
13					X	O	O	O	O	X			X						
14				O			X	O	X										
15							X	O	O										
16																			
17																			
18																			
19																			

Game 9

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
1																			
2																			
3																			
4																			
5																			
6																			
7								X	X	O	X								
8								X	O	X	O								
9								X	X	O	O	X							
10								O	X	O	O								
11										O	O	X							
12											O								
13																			
14																			
15																			
16																			
17																			
18																			
19																			

Appendix 5: control flow chart of the Go-Moku program



The Program

```
#include <stdio.h>

#define TOP_LINE 0
#define BOT_LINE 21
 define MESG_LINE 15
 define ERR_LINE 22
 define BLACK 'X'
 define WHITE 'O'
 define EMPTY '.'
 define FAST_MODE 0
 define POST_MODE 1

 /pedef struct _dir {
            int mattered; /* for the directon, matter or not */
            int stones;   /* pre-set value because of built pattern matched */
            int blocked;  /* whether this is a closed subgoal or not */
            int open_3;
            int close_4;
            int open_4;
        };

 /pedef struct _board {
            int sum;
            int term[17];       /* 0 :   4_directional_evaluation
                                   1 :   neighbor_relation_evaluation
                                   2 :   block_opponent_evaluation
                                   3 :   mattered_count_evaluation
                                   4 :   not_being_blocked_evaluation
                                   5 :   offend_win_move_evaluation
                                   6 :   defend_win_move_evaluation
                                   7 :   offend_open_3_evaluation
                                   8 :   defend_open_3_evaluation
                                   9 :   offend_close_4_evaluation
                                  10 :   defend_close_4_evaluation
                                  11 :   offend_cross_3_evaluation
                                  12 :   defend_cross_3_evaluation
                                  13 :   offend_setup_evaluation
                                  14 :   defend_setup_evaluation
                                  15 :   offend_open_4_evaluation
                                  16 :   defend_open_4_evaluation
                                */
            struct _dir dir[8];
        };

typedef struct _step {          /* game step sequence record */
            int term[17];
            struct _step *next;
        };
```

```
/*
 * Course      : ICSS-B95 MS Thesis
 * Advisor     : John A. Biles
 *               Warren R. Carithers
 *               Lawrence Coon
 * Programmer  : Dong-Ming Thomas Wang
 * Topic       : Game of Go-Moku
 */

 include <curses.h>
 include "extern.h"

 ain()

    change_to_cr();                          /* start curses mode */
    while (bye() != 0) (                     /* when not quit */
        init_glob();                         /* init global variable */
        init_screen();                       /* draw initial game-board */
        play();                              /* start the game */
    )
    change_to_nocr();
 inal_table();


 change_to_cr()                             /* curses mode */
 (
     initscr();
     crmode();
 )

 change_to_nocr()                           /* no curses mode */
 (
     clear();
     refresh();
     nocrmode();
     resetty();
     endwin();
 )

 bye()
 {
     clear();
     instruction();
     str_get(buf);
     if ((buf[0] == 'n') || (buf[0] == 'N'))  (
         return (0);
     ) /* if */
     else (
          clear();
          options();
          str_get(buf);
          move(20,1);
          clrtoeol();
          switch (buf[0]) (
              case '1':  mode = 1; mode_1(); start(); break;
              case '2':  mode = 2; mode_2(); start(); break;
```

```
            case '3':  mode = 3; mode_3(); start(); break;
            case '4':  mode = 4; mode_4(); start(); break;
            case '5':  mode = 5; mode_5(); start(); break;
            default ;  mode = 3; mode_3(); start(); break;
        } /* switch */
        return (1);
    }


inal_table()

    int i, j;

    printf("                          ");
    for (i = 2; i <=18; i++, i++)
       printf("%2d  ",i);
    printf("\n");
    printf("                        ");
    for (i = 1; i <=19; i++, i++)
       printf("%2d  ",i);
    printf("\n\n");
    for (i=1 ; i<=19; i++)
    {
        printf("                    %2d   ,i);
        for (j = 1; j <= 19 ; j++)
            printf("%c  ,board[i][j]);
        printf("\n");
     }
}
```

```
#include <curses.h>
#include <fcntl.h>
#include "common.h"

toa(num,tmp)                                /* change integer to char */
nt num;
har *tmp;

    tmp[1] = '0' + num%10;
    tmp[0] = '0' + (num/10)%10;


r()                             /* carriage return */

    char buf[80];

    mvaddstr(ERR_LINE-1,0,'_____________________________________________");
    mvaddstr(ERR_LINE-1,39,"_____________________________________________");
    mvaddstr(ERR_LINE,0,"NOTE:  hit return to continue  ");
    refresh();
    getstr(buf);
    move(ERR_LINE,7);
    clrtoeol();
    refresh();
}

error(number)
int number;
(
    change_to_nocr();
    switch(number) (
      case 0: perror("\nerror encountered when CREATE fast file");
              break;
      case 1: perror("\nerror encountered when CREATE post file");
              break;
      case 2: perror("\nerror with one of the DEFAULT case");     \
              break;
      case 3: perror("\nerror when try to READ file 'fast'");
              break;
      case 4: perror("\nerror when try to READ file 'post'");
              break;
      case 5: perror("\nerror when try to WRITE file 'fast'");
              break;
      case 6: perror("\nerror when try to WRITE file 'post'");
              break;
      case 7: perror("\nerror with final terms value for the current move ");
      default: break;
    )
    exit(0);
)

str_get(num)
char *num;
(
    int i, j;
    char num1[81];
```

```
    getstr(num);
    i = 0;
    j = 0;
    while (num[i] != NULL)
        if (num[i++] == '\010') {
            if (j != 0)
                --j;
        }
        else
            num1[j++] = num[i-1];
    num1[j++] = NULL;
    strcpy(num,num1);


lear_mesg_line()

    move(ERR_LINE,6);
    clrtoeol();
    refresh();


 eep()
{
        putchar('\007');
}

add_step()
{
    int i;
    struct _step *current;

    current = (struct _step *) malloc(sizeof(struct _step));
    current->next = head;
    for (i=0; i <= 16; i++)
        current->term[i] = mat[0][ROW][COL].term[i];
    head = current;
}
```

```
#include <curses.h>
#include <fcntl.h>
#include "common.h"

nit_glob()

    init_variables();
    init_value_for_blank();
    create_init_base();          /* create and initialize pattern base file */
    init_2_board();              /* initialize offensive matrix */
    init_step_lk();


nit_variables()

    ROW = 10;
    COL = 10;


nit_value_for_blank()

    int i, j;

    for (i=1; i <= 19; i++)
       for (j=1; j <= 19; j++)
          board[i][j] = '.';
}

init_2_board()
{
    int i, j;

    for (i=1; i <= 19; i++)
       for (j=1; j <= 19; j++)  {
          init_terms_sum(i,j);
          init_term0(i,j);
          init_dir(i,j);
          init_off_def(i,j);
       } /* for */
}

init_term0(i,j)
int i, j;
{
    switch (i) {
        case 1:
        case 19: switch (j) {
                      case 1:
                      case 19: mat[0][i][j].term[0] = 3;
                               mat[1][i][j].term[0] = 3;
                               mat[0][i][j].term[3] = 18;
                               mat[1][i][j].term[3] = 18;
                               break;
                      case 2:
                      case 18: mat[0][i][j].term[0] = 4;
                               mat[1][i][j].term[0] = 4;
```

```
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 3:
                case 17: mat[0][i][j].term[0] = 5;
                        mat[1][i][j].term[0] = 5;
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 4:
                case 16: mat[0][i][j].term[0] = 6;
                        mat[1][i][j].term[0] = 6;
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                case 10:
                case 11:
                case 12:
                case 13:
                case 14:
                case 15: mat[0][i][j].term[0] = 8;
                        mat[1][i][j].term[0] = 8; break;
                default: break;
            }; break;
        case 2:
        case 18: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 4;
                        mat[1][i][j].term[0] = 4;
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 2:
                case 18: mat[0][i][j].term[0] = 6;
                        mat[1][i][j].term[0] = 6;
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 3:
                case 17: mat[0][i][j].term[0] = 7;
                        mat[1][i][j].term[0] = 7;
                        mat[0][i][j].term[3] = 18;
                        mat[1][i][j].term[3] = 18;
                        break;
                case 4:
                case 16: mat[0][i][j].term[0] = 9;
                        mat[1][i][j].term[0] = 9; break;
                case 5:
                case 6:
                case 7:
                case 8:
```

```
                case 9:
                case 10:
                case 11:
                case 12:
                case 13:
                case 14:
                case 15: mat[0][i][j].term[0] = 11;
                         mat[1][i][j].term[0] = 11; break;
                default: break;
            }; break;
        case 3:
        case 17: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 5;
                         mat[1][i][j].term[0] = 5;
                         mat[0][i][j].term[3] = 18;
                         mat[1][i][j].term[3] = 18;
                         break;
                case 2:
                case 18: mat[0][i][j].term[0] = 7;
                         mat[1][i][j].term[0] = 7;
                         mat[0][i][j].term[3] = 18;
                         mat[1][i][j].term[3] = 18;
                         break;
                case 3:
                case 17: mat[0][i][j].term[0] = 10;
                         mat[1][i][j].term[0] = 10; break;
                case 4:
                case 18: mat[0][i][j].term[0] = 12;
                         mat[1][i][j].term[0] = 12; break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                case 10:
                case 11:
                case 12:
                case 13:
                case 14:
                case 15: mat[0][i][j].term[0] = 14;
                         mat[1][i][j].term[0] = 14; break;
                default: break;
            }; break;
        case 4:
        case 16: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 6;
                         mat[1][i][j].term[0] = 6;
                         mat[0][i][j].term[3] = 18;
                         mat[1][i][j].term[3] = 18;
                         break;
                case 2:
                case 18: mat[0][i][j].term[0] = 9;
                         mat[1][i][j].term[0] = 9; break;
                case 3:
```

```
                case 17: mat[0][i][j].term[0] = 12;
                         mat[1][i][j].term[0] = 12; break;
                case 4:
                case 16: mat[0][i][j].term[0] = 15;
                         mat[1][i][j].term[0] = 15; break;
                case 5:
                case 6:
                case 7:
                case 6:
                case 9:
                case 10:
                case 11:
                case 12:
                case 13:
                case 14:
                case 15: mat[0][i][j].term[0] = 17;
                         mat[1][i][j].term[0] = 17; break;
                default: break;
            }; break;
        case 5:
        case 15: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 8;
                         mat[1][i][j].term[0] = 8; break;
                case 2:
                case 18: mat[0][i][j].term[0] = 11;
                         mat[1][i][j].term[0] = 11; break;
                case 3:
                case 17: mat[0][i][j].term[0] = 14;
                         mat[1][i][j].term[0] = 14; break;
                case 4:
                case 16: mat[0][i][j].term[0] = 17;
                         mat[1][i][j].term[0] = 17; break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                case 10:
                case 11:
                case 12:
                case 13:
                case 14:
                case 15: mat[0][i][j].term[0] = 20;
                         mat[1][i][j].term[0] = 20; break;
                default: break;
            }; break;
        case 6:
        case 14: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 8;
                         mat[1][i][j].term[0] = 8; break;
                case 2:
                case 18: mat[0][i][j].term[0] = 11;
                         mat[1][i][j].term[0] = 11; break;
                case 3:
```

```
                    case 17: mat[0][i][j].term[0] = 14;
                             mat[1][i][j].term[0] = 14; break;
                    case 4:
                    case 16: mat[0][i][j].term[0] = 17;
                             mat[1][i][j].term[0] = 17; break;
                    case 5:
                    case 15: mat[0][i][j].term[0] = 20;
                             mat[1][i][j].term[0] = 20; break;
                    case 6:
                    case 7:
                    case 8:
                    case 9:
                    case 10:
                    case 11:
                    case 12:
                    case 13:
                    case 14: mat[0][i][j].term[0] = 21;
                             mat[1][i][j].term[0] = 21; break;
                    default: break;
                }; break;
        case 7:
        case 13: switch (j) {
                    case 1:
                    case 19: mat[0][i][j].term[0] = 8;
                             mat[1][i][j].term[0] = 8; break;
                    case 2:
                    case 18: mat[0][i][j].term[0] = 11;
                             mat[1][i][j].term[0] = 11; break;
                    case 3:
                    case 17: mat[0][i][j].term[0] = 14;
                             mat[1][i][j].term[0] = 14; break;
                    case 4:
                    case 16: mat[0][i][j].term[0] = 17;
                             mat[1][i][j].term[0] = 17; break;
                    case 5:
                    case 15: mat[0][i][j].term[0] = 20;
                             mat[1][i][j].term[0] = 20; break;
                    case 6:
                    case 14: mat[0][i][j].term[0] = 21;
                             mat[1][i][j].term[0] = 21; break;
                    case 7:
                    case 8:
                    case 9:
                    case 10:
                    case 11:
                    case 12:
                    case 13: mat[0][i][j].term[0] = 22;
                             mat[1][i][j].term[0] = 22; break;
                    default: break;
                }; break;
        case 8:
        case 12: switch (j) {
                    case 1:
                    case 19: mat[0][i][j].term[0] = 8;
                             mat[1][i][j].term[0] = 8; break;
                    case 2:
```

```
                case 18: mat[0][i][j].term[0] = 11;
                         mat[1][i][j].term[0] = 11; break;
                case 3:
                case 17: mat[0][i][j].term[0] = 14;
                         mat[1][i][j].term[0] = 14; break;
                case 4:
                case 16: mat[0][i][j].term[0] = 17;
                         mat[1][i][j].term[0] = 17; break;
                case 5:
                case 15: mat[0][i][j].term[0] = 20;
                         mat[1][i][j].term[0] = 20; break;
                case 6:
                case 14: mat[0][i][j].term[0] = 21;
                         mat[1][i][j].term[0] = 21; break;
                case 7:
                case 13: mat[0][i][j].term[0] = 22;
                         mat[1][i][j].term[0] = 22; break;
                case 8:
                case 9:
                case 10:
                case 11:
                case 12: mat[0][i][j].term[0] = 23;
                         mat[1][i][j].term[0] = 23; break;
                default: break;
            }; break;
        case 9:
        case 11: switch (j) {
                case 1:
                case 19: mat[0][i][j].term[0] = 8;
                         mat[1][i][j].term[0] = 8; break;
                case 2:
                case 18: mat[0][i][j].term[0] = 11;
                         mat[1][i][j].term[0] = 11; break;
                case 3:
                case 17: mat[0][i][j].term[0] = 14;
                         mat[1][i][j].term[0] = 14; break;
                case 4:
                case 16: mat[0][i][j].term[0] = 17;
                         mat[1][i][j].term[0] = 17; break;
                case 5:
                case 15: mat[0][i][j].term[0] = 20;
                         mat[1][i][j].term[0] = 20; break;
                case 6:
                case 14: mat[0][i][j].term[0] = 21;
                         mat[1][i][j].term[0] = 21; break;
                case 7:
                case 13: mat[0][i][j].term[0] = 22;
                         mat[1][i][j].term[0] = 22; break;
                case 8:
                case 12: mat[0][i][j].term[0] = 23;
                         mat[1][i][j].term[0] = 23; break;
                case 9:
                case 10:
                case 11: mat[0][i][j].term[0] = 24;
                         mat[1][i][j].term[0] = 24; break;
                default: break;
```

```
                }; break;
        case 10: switch (j) {
                    case 1:
                    case 19: mat[0][i][j].term[0] = 8;
                             mat[1][i][j].term[0] = 8; break;
                    case 2:
                    case 18: mat[0][i][j].term[0] = 11;
                             mat[1][i][j].term[0] = 11; break;
                    case 3:
                    case 17: mat[0][i][j].term[0] = 14;
                             mat[1][i][j].term[0] = 14; break;
                    case 4:
                    case 16: mat[0][i][j].term[0] = 17;
                             mat[1][i][j].term[0] = 17; break;
                    case 5:
                    case 15: mat[0][i][j].term[0] = 20;
                             mat[1][i][j].term[0] = 20; break;
                    case 6:
                    case 14: mat[0][i][j].term[0] = 21;
                             mat[1][i][j].term[0] = 21; break;
                    case 7:
                    case 13: mat[0][i][j].term[0] = 22;
                             mat[1][i][j].term[0] = 22; break;
                    case 8:
                    case 12: mat[0][i][j].term[0] = 23;
                             mat[1][i][j].term[0] = 23; break;
                    case 9:
                    case 11: mat[0][i][j].term[0] = 24;
                             mat[1][i][j].term[0] = 24; break;
                    case 10: mat[0][i][j].term[0] = 25;
                             mat[1][i][j].term[0] = 25; break;
                    default: break;
                 }; break;
        default: break;
    } /* switch */
}

init_terms_sum(i,j)
int i, j;
{
    int k;

    mat[0][i][j].sum = 0;
    mat[1][i][j].sum = 0;
    for (k=0; k <= 16; k++) {
        switch (k) {
           case 0: break;
           case 1:
           case 2: mat[0][i][j].term[k] = 0;
                   mat[1][i][j].term[k] = 0;
                   break;
           case 3:
           case 4: mat[0][i][j].term[k] = 24;
                   mat[1][i][j].term[k] = 24;
                   break;
           default : mat[0][i][j].term[k] = 12;
```

```
                    mat[1][i][j].term[k] = 12;
                    break;
        } /* switch */
        mat[0][i][j].sum += mat[0][i][j].term[k];
        mat[1][i][j].sum += mat[1][i][j].term[k];
    } /* for */


nit_dir(i,j)
nt i, j;

    int k;

    for (k=0; k <=7; k++) {
       mat[1][i][j].dir[k].mattered = 1;
       mat[1][i][j].dir[k].stones = 0;
       mat[1][i][j].dir[k].blocked = 0;
       mat[0][i][j].dir[k].mattered = 1;
       mat[0][i][j].dir[k].stones = 0;
       mat[0][i][j].dir[k].blocked = 0;
    } /* for k */
    if ((i==1) || (i==19) || (j==1) || (j==19))
        set_blocked(i,j);
}

init_off_def(i,j)
int i, j;
{
    int k;

    for (k=0; k <=7; k++) {
        mat[0][i][j].dir[k].open_3 = 0;
        mat[1][i][j].dir[k].open_3 = 0;
        mat[0][i][j].dir[k].close_4 = 0;
        mat[1][i][j].dir[k].close_4 = 0;
        mat[0][i][j].dir[k].open_4 = 0;
        mat[1][i][j].dir[k].open_4 = 0;
    }
}

set_blocked(i,j,k)
int i,j,k;
{
    switch (i) {
       case 1: switch (j) {
                  case 1: mat[0][i][j].dir[0].blocked = 1;
                          mat[1][i][j].dir[0].blocked = 1;
                          mat[0][i][j].dir[1].blocked = 1;
                          mat[1][i][j].dir[1].blocked = 1;
                          mat[0][i][j].dir[5].blocked = 1;
                          mat[1][i][j].dir[5].blocked = 1;
                          mat[0][i][j].dir[6].blocked = 1;
                          mat[1][i][j].dir[6].blocked = 1;
                          mat[0][i][j].dir[7].blocked = 1;
                          mat[1][i][j].dir[7].blocked = 1;
                          break;
```

```
                case 19: mat[0][i][j].dir[0].blocked = 1;
                         mat[1][i][j].dir[0].blocked = 1;
                         mat[0][i][j].dir[1].blocked = 1;
                         mat[1][i][j].dir[1].blocked = 1;
                         mat[0][i][j].dir[2].blocked = 1;
                         mat[1][i][j].dir[2].blocked = 1;
                         mat[0][i][j].dir[3].blocked = 1;
                         mat[1][i][j].dir[3].blocked = 1;
                         mat[0][i][j].dir[7].blocked = 1;
                         mat[1][i][j].dir[7].blocked = 1;
                         break;
                default: mat[0][i][j].dir[0].blocked = 1;
                         mat[1][i][j].dir[0].blocked = 1;
                         mat[0][i][j].dir[1].blocked = 1;
                         mat[1][i][j].dir[1].blocked = 1;
                         mat[0][i][j].dir[7].blocked = 1;
                         mat[1][i][j].dir[7].blocked = 1;
                         break;
              } break; /* case 1 switch */
      case 19: switch (j) {
                 case 1: mat[0][i][j].dir[3].blocked = 1;
                         mat[1][i][j].dir[3].blocked = 1;
                         mat[0][i][j].dir[4].blocked = 1;
                         mat[1][i][j].dir[4].blocked = 1;
                         mat[0][i][j].dir[5].blocked = 1;
                         mat[1][i][j].dir[5].blocked = 1;
                         mat[0][i][j].dir[6].blocked = 1;
                         mat[1][i][j].dir[6].blocked = 1;
                         mat[0][i][j].dir[7].blocked = 1;
                         mat[1][i][j].dir[7].blocked = 1;
                         break;
                case 19: mat[0][i][j].dir[1].blocked = 1;
                         mat[1][i][j].dir[1].blocked = 1;
                         mat[0][i][j].dir[2].blocked = 1;
                         mat[1][i][j].dir[2].blocked = 1;
                         mat[0][i][j].dir[3].blocked = 1;
                         mat[1][i][j].dir[3].blocked = 1;
                         mat[0][i][j].dir[4].blocked = 1;
                         mat[1][i][j].dir[4].blocked = 1;
                         mat[0][i][j].dir[5].blocked = 1;
                         mat[1][i][j].dir[5].blocked = 1;
                         break;
                default: mat[0][i][j].dir[3].blocked = 1;
                         mat[1][i][j].dir[3].blocked = 1;
                         mat[0][i][j].dir[4].blocked = 1;
                         mat[1][i][j].dir[4].blocked = 1;
                         mat[0][i][j].dir[5].blocked = 1;
                         mat[1][i][j].dir[5].blocked = 1;
                         break;
               } break; /* case 19 switch */
      default: switch (j) {
                  case 1: mat[0][i][j].dir[5].blocked = 1;
                          mat[1][i][j].dir[5].blocked = 1;
                          mat[0][i][j].dir[6].blocked = 1;
                          mat[1][i][j].dir[6].blocked = 1;
                          mat[0][i][j].dir[7].blocked = 1;
```

```
                        mat[1][i][j].dir[7].blocked = 1;
                        break;
                case 19: mat[0][i][j].dir[1].blocked = 1;
                         mat[1][i][j].dir[1].blocked = 1;
                         mat[0][i][j].dir[2].blocked = 1;
                         mat[1][i][j].dir[2].blocked = 1;
                         mat[0][i][j].dir[3].blocked = 1;
                         mat[1][i][j].dir[3].blocked = 1;
                         break;
                default: printf("\nsomething wrong in set_blocked");
                         break;
              }  break;  /* default switch */
   } /* switch */


reate_init_base()

   int i, j, n;

   strcpy(fast,"/dd4/systst/wang/rit/pgm/fast");
   f_fd = open(fast,O_CREAT|O_RDWR,0600);
   if (f_fd < 0)   error(0);
   strcpy(post,"/dd4/systst/wang/rit/pgm/post");
   p_fd = open(post,O_CREAT|O_RDWR,0600);
   if (p_fd < 0)   error(1);

   lseek(f_fd,0,0);
   lseek(p_fd,0,0);
   switch (mode) {
       case 5:
       case 4:
       case 2: n = read(p_fd,&weight[0],sizeof(int));  break;
       case 1:
       default: n = read(f_fd,&weight[0],sizeof(int)); break;
   }
   if (n == 0) {
        lseek(f_fd,0,0);
        lseek(p_fd,0,0);
        for (i=0; i <= 16; i++) {
            weight[i] = 100;
            err = write(f_fd,&weight[i],sizeof(int));  /* 17 terms in fast */
            if (err == 0)  error(5);
            err = write(p_fd,&weight[i],sizeof(int));  /* 17 terms in post */
            if (err == 0)  error(6);
        }
   }
   else for (i=1; i <= 16; i++) {
            switch (mode) {
                case 5:
                case 4:
                case 2: n = read(p_fd,&weight[i],sizeof(int)); break;
                case 1:
                default: n = read(f_fd,&weight[i],sizeof(int)); break;
            }
            if (n < 0) error (1);
        }
```

```
}

init_step_lk()
{
    head = (struct _step *) malloc(sizeof(struct _step));
    head = NULL;
```

```
#include <curses.h>
#include "common.h"

init_screen()                                   /* print initial chess board */

    clear();
    first_row();
    num_then_blank();
    menu();
    refresh();
    cr();


 irst_row()

    int i;
    char s[4];

    for (i=1; i <= 19; i++) (             /* print out the first row */
      itoa(i,s);
      if (i % 2 == 1)
          mvaddstr(1,i*2+2,s);
      else
          mvaddstr(0,i*2+2,s);
    }
}

num_then_blank()
{
    int i, j;
    char tmp[2];

    for (i=1; i <= 19; i++) (           /* print out the rest row of the table */
      itoa(i,tmp);
      mvaddstr(i+1,1,tmp);
      for (j=1; j <=19; j++)
         mvaddch(i+1,j*2+3,'.');
    }
}

menu()                                          /* print instructions on screen */
{
    mvaddstr(0,50,"menu");
    mvaddstr(1,50,"----");
    mvaddstr(2,50,"1.  O  rep YOUR stone");
    mvaddstr(3,50,"2.  X  rep COMP stone");
    mvaddstr(4,50,"3.  Use:");
    mvaddstr(5,50,     h  move cursor left");
    mvaddstr(6,50,     j  move cursor down");
    mvaddstr(7,50,     k  move cursor up");
    mvaddstr(8,50,     l  move cursor right");
    mvaddstr(9,50,"4.  hit <CR> to place a move");
}
```

```
#include <curses.h>
#include "common.h"

instruction()

    mvaddstr(3,14,"***** Welcome to the game of Go-Moku *****");
    mvaddstr(6,20,"instructions and rules:");
    mvaddstr(7,20,"~~~~~~~~~~~~~~~~~~~~~");
    mvaddstr(8,20,"1.  How to move the cursor around:");
    mvaddstr(9,26,"h  move to left");
    mvaddstr(10,26,"j  move right");
    mvaddstr(11,26,"k  move up");
    mvaddstr(12,26,"l  move right");
    mvaddstr(14,20,"2.  How to place a move:");
    mvaddstr(15,26,"Hit the carriage return key <CR>");
    mvaddstr(19,0,"_____________________________________________");
    mvaddstr(19,39,"_____________________________________");
    mvaddstr(20,5,"Are you ready to play the game of Go-Moku ? (y/n) ");
    refresh();


 ptions()

    mvaddstr(3,14,"There are 4 playing modes you can choose :");
    mvaddstr(5,20,"Mode 1 : fast learning mode");
    mvaddstr(6,20,"Mode 2 : post game learning mode");
    mvaddstr(7,20,"Mode 3 : comp vs. human, no learning, from fast learning");
    mvaddstr(8,20,"Mode 4 : computer vs. computer, post game learning");
    mvaddstr(9,20,"Mode 5 : comp vs. human, no learning, from post learning");
    mvaddstr(19,0,"_____________________________________________");
    mvaddstr(19,39,"_____________________________________");
    mvaddstr(20,1,"Now choose a mode No. (1, 2, 3, 4 or 5) : ");
    refresh();
}

mode_5()
{
  mvaddstr(12,14,"Good!!  So you choose Mode 5");
  mvaddstr(14,20,"In this mode computer will play with itself,");
  mvaddstr(15,20,"This is learning left from POST learning approach.");
}

mode_4()
{
  mvaddstr(12,14,"Good!!  So you choose Mode 4");
  mvaddstr(14,20,"In this mode computer will play with itself,");
  mvaddstr(15,20,"This is learning left from FAST learning approach.");
}

mode_1()
{
    mvaddstr(11,14,"Good!!  So you choose Mode 1");
    mvaddstr(12,20,"This is the 'fast learning' mode.");
    mvaddstr(13,20,"In this you will act as a teacher for the program.");
    mvaddstr(14,18,"* For your move :");
    mvaddstr(15,20,     just follow the instruction list above");
```

```
    mvaddstr(16,18,"* For computer's move :");
    mvaddstr(17,20,"    You have to enter a 'rate' (ranging from -2 to 2");
    mvaddstr(18,20,'    for every move made by the program");
}

ode_2()

    mvaddstr(12,14,"Good!!  So you choose Mode 2");
    mvaddstr(14,20,"This is the 'post game learning' mode.");
    mvaddstr(15,18,"* When it is your turn to make a move :");
    mvaddstr(16,20,     just follow the menu which will be shown");


ode_3()

 mvaddstr(12,14,"Good!!  So you choose Mode 3 (or by default)");
 mvaddstr(14,20,"This is a regular game mode. (no learning will be performed");
 mvaddstr(15,18,"* When it is your turn to make a move :");
 mvaddstr(16,20,"just follow the menu which will be shown on the next window");


tart()

    move(20,1);
    clrtoeol();
    mvaddstr(20,1,"Now, hit return to continue !! ");
    refresh();
    getstr(buf);
}
```

```
#include <curses.h>
#include "common.h"

human_mov()

    visual_mov(WHITE);


gm_mov(color)
har color;

    int r, c;
    int player;

    switch (color) {
       case BLACK: sum_terms(0); player = 0; break;
       case WHITE: sum_terms(1); player = 1; break;
       default: break;
    }
    choose_highest(player,&r,&c);
    if (color == BLACK)  make_move(r,c,BLACK);
    else make_move(r,c,WHITE);
    if (mode == 1)
        upd_weight(r,c);
    if ((mode == 2) || ((mode == 4) && (color == BLACK)))
        add_step();
}

sum_terms(player)
int player;
{
    int i, j, k;

    for (i=1; i <= 19; i++)
        for (j=1; j <= 19; j++)
            if (board[i][j] == EMPTY) {
                mat[player][i][j].sum = 0;
                for (k=0; k <= 16; k++)
                    mat[player][i][j].sum += mat[player][i][j].term[k] *
                                             weight[k];
            }
}

choose_highest(player,r,c)
int player,*r,*c;
{
    int i,j;
    int tmp = 0;

    for (i=1; i <= 19; i++)
        for (j=1; j <= 19; j++)
            if ((board[i][j] == EMPTY) && (mat[player][i][j].sum > tmp)) {
                tmp = mat[player][i][j].sum;
                *r = i;
                *c = j;
            }
```

```
}

upd_weight(r,c)
int r, c;

    int x;
    int offset = 0;

    for (x=0; x <= 16; x++) {
       switch (mat[0][r][c].term[x]) {
          case 0:
          case 1:
          case 2:
          case 3:
          case 4: offset = rate * (-2); break;
          case 5:
          case 6:
          case 7:
          case 8:
          case 9: offset = rate * (-1); break;
          case 10:
          case 11:
          case 12:
          case 13:
          case 14: offset = rate * 0; break;
          case 15:
          case 16:
          case 17:
          case 18:
          case 19: offset = rate * 1; break;
          default: if (mat[0][r][c].term[x] < 0) error(7);
                   offset = rate * 2; break;
       } /* switch */
       if ((weight[x] + offset) > 0)
            weight[x] += offset;
    } /* for */
}
```

```
#include <curses.h>
#include "common.h"

 lay()

    int end = 0;

    while (1) {
        if (mode == 4) pgm_mov(WHITE);          /* generate computer's move */
        else human_mov();                    /* generate human-player's move */
        if (win_chk(WHITE) == 1)
            break;
        else {
             upd_relation(WHITE);
             set_mattered(WHITE);
        } /* else */
        pgm_mov(BLACK);                          /* generate computer's move */
        if (win_chk(BLACK) == 1)
             break;
        else {
             upd_relation(BLACK);
             set_mattered(BLACK);
        } /* else */
    } /* while */
    close (f_fd);
    close (p_fd);
} /* play */
```

```
#include <curses.h>
#include "common.h"

pd_relation(co)
har  co;

    nn_mat(co);
    ne_mat(co);
    ee_mat(co);
    es_mat(co);
    ss_mat(co);
    sw_mat(co);
    ww_mat(co);
    wn_mat(co);


olor_player(co)
har co;

     if (co == BLACK)  return(0);
     else return(1);


nn_mat(co)                                          /* north direction */
char  co;
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW+1;
    oppoy = COL;
    if (mat[player][oppox][oppoy].dir[4].blocked == 1)
        mat[player][ROW][COL].dir[4].blocked = 1;
    if (mat[player][ROW-1][COL].dir[0].blocked == 1)
        mat[player][ROW][COL].dir[0].blocked = 1;
    j = COL;
    for (i = ROW-1; i >= ROW-5; i--)
        if (outside(i) == 1)  return;
        else upd_stones(co,i,j,0,oppox,oppoy,&flip,&first);
}

ne_mat(co)                                          /* north-east direction */
char  co;
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW+1;
    oppoy = COL-1;
    if (mat[player][oppox][oppoy].dir[5].blocked == 1)
        mat[player][ROW][COL].dir[5].blocked = 1;
    if (mat[player][ROW-1][COL+1].dir[1].blocked == 1)
```

```
        mat[player][ROW][COL].dir[1].blocked = 1;
    for (i = ROW-1; i >= ROW-5; i--)
       if (outside(i) == 1)  return;
       else for (j = COL+1; j <= COL+5; j++)
               if ((ROW-i) == (j-COL))
                   upd_stones(co,i,j,1,oppox,oppoy,&flip,&first);


_mat(co)                                            /* east direction */
ar  co;

    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW;
    oppoy = COL-1;
    if (mat[player][oppox][oppoy].dir[6].blocked == 1)
        mat[player][ROW][COL].dir[6].blocked = 1;
    if (mat[player][ROW][COL+1].dir[2].blocked == 1)
        mat[player][ROW][COL].dir[2].blocked = 1;
    i = ROW;
    for (j = COL+1; j <= COL+5; j++)
        if (outside(j) == 1)  return;
        else upd_stones(co,i,j,2,oppox,oppoy,&flip,&first);
}

es_mat(co)                                          /* east-south direction */
char  co;
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW-1;
    oppoy = COL-1;
    if (mat[player][oppox][oppoy].dir[7].blocked == 1)
        mat[player][ROW][COL].dir[7].blocked = 1;
    if (mat[player][ROW+1][COL+1].dir[3].blocked == 1)
        mat[player][ROW][COL].dir[3].blocked = 1;
    for (i = ROW+1; i <= ROW+5; i++)
       if (outside(i) == 1)  return;
       else for (j = COL+1; j <= COL+5; j++)
               if ((i-ROW) == (j-COL))
                  upd_stones(co,i,j,3,oppox,oppoy,&flip,&first);
}

ss_mat(co)                                          /* south direction */
char  co;
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;
```

```
    player = color_player(co);
    oppox = ROW-1;
    oppoy = COL;
    if (mat[player][oppox][oppoy].dir[0].blocked == 1)
        mat[player][ROW][COL].dir[0].blocked = 1;
    if (mat[player][ROW+1][COL].dir[4].blocked == 1)
        mat[player][ROW][COL].dir[4].blocked = 1;
    j = COL;
    for (i = ROW+1; i <= ROW+5; i++)
        if (outside(i) == 1)  return;
        else upd_stones(co,i,j,4,oppox,oppoy,&flip,&first);


 _mat(co)                                          /* south-west direction */
 ar  co;

    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW-1;
    oppoy = COL+1;
    if (mat[player][oppox][oppoy].dir[1].blocked == 1)
        mat[player][ROW][COL].dir[1].blocked = 1;
    if (mat[player][ROW+1][COL-1].dir[5].blocked == 1)
        mat[player][ROW][COL].dir[5].blocked = 1;
    for (i = ROW+1; i <= ROW+5; i++)
       if (outside(i) == 1)  return;
       else for (j = COL-1; j >= COL-5; j--)
                if ((i-ROW) == (COL-j))
                    upd_stones(co,i,j,5,oppox,oppoy,&flip,&first);
}

ww_mat(co)                                           /* west direction */
char  co;
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW;
    oppoy = COL+1;
    if (mat[player][oppox][oppoy].dir[2].blocked == 1)
        mat[player][ROW][COL].dir[2].blocked = 1;
    if (mat[player][ROW][COL-1].dir[6].blocked == 1)
        mat[player][ROW][COL].dir[6].blocked = 1;
    i = ROW;
    for (j = COL-1; j >= COL-5; j--)
        if (outside(j) == 1)  return:
        else upd_stones(co,i,j,6,oppox,oppoy,&flip,&first);
}

wn_mat(co)                                           /* west-north direction */
char  co;
```

```
{
    int i, j, oppox, oppoy;
    int player;
    int flip = 0, first = 1;

    player = color_player(co);
    oppox = ROW+1;
    oppoy = COL+1;
    if (mat[player][oppox][oppoy].dir[3].blocked == 1)
        mat[player][ROW][COL].dir[3].blocked = 1;
    if (mat[player][ROW-1][COL-1].dir[7].blocked == 1)
        mat[player][ROW][COL].dir[7].blocked = 1;
    for (i = ROW-1; i >= ROW-5; i--)
       if (outside(i) == 1)  return;
       else for (j = COL-1; j >= COL-5; j--)
               if ((i-ROW) == (j-COL))
                   upd_stones(co,i,j,7,oppox,oppoy,&flip,&first);
}

upd_stones(co,i,j,k,oppox,oppoy,flip,first)
char co;
int i,j,k,oppox,oppoy;
int *flip, *first;
{
    int blocked;
    int increment;
    int oppo_dir;

    oppo_dir = (k + 4) % 8;
    switch (co) {
      case WHITE: blocked = (board[oppox][oppoy] == BLACK) ? 1 : 0;
                  if (blocked == 0)
                      if (mat[1][oppox][oppoy].dir[oppo_dir].blocked==1)
                          blocked = 1;
                  increment = mat[1][ROW][COL].dir[oppo_dir].stones + 1;
                  switch (board[i][j]) {
                    case BLACK: *flip = 1;
                                mat[0][i][j].dir[oppo_dir].blocked = 1;
                                break;
                    case EMPTY: if ((*flip != 1) && (*first))
                                    dir_w_mat(i,j,oppo_dir,blocked,increment);
                                if (*first) {
                                    mat[0][i][j].dir[oppo_dir].blocked = 1;
                                    mat[0][i][j].term[4] -= 3;
                                    term_2(0,i,j);
                                    *first = 0;
                                }
                                term_1_5_6(1,i,j);
                                clr_terms(i,j);
                                term_7_9_11_13_15(i,j,k);
                                term_8_10_12_14_16(i,j,k);
                                break;
                    case WHITE: if ((*flip != 1) && (blocked))
                                    mat[1][i][j].dir[oppo_dir].blocked = 1;
                                break;
                    default: break;
```

```
                } /* case 1 switch */
                break;
      case BLACK: blocked = (board[oppox][oppoy] == WHITE) ? 1 : 0;
                if (blocked == 0)
                    if (mat[0][oppox][oppoy].dir[oppo_dir].blocked==1)
                        blocked = 1;
                increment = mat[0][ROW][COL].dir[oppo_dir].stones + 1;
                switch (board[i][j]) {
                  case WHITE: *flip = 1;
                              mat[1][i][j].dir[oppo_dir].blocked = 1;
                              break;
                  case EMPTY: if ((*flip != 1) && (*first))
                                  dir_b_mat(i,j,oppo_dir,blocked,increment);
                              if (*first) {
                                  mat[1][i][j].dir[oppo_dir].blocked = 1;
                                  mat[1][i][j].term[4] -= 3;
                                  term_2(1,i,j);
                                  *first = 0;
                              }
                              term_1_5_6(0,i,j);
                              clr_terms(i,j);
                              term_7_9_11_13_15(i,j,k);
                              term_8_10_12_14_16(i,j,k);
                              break;
                  case BLACK: if ((*flip != 1) && (blocked))
                                  mat[0][i][j].dir[oppo_dir].blocked = 1;
                              break;
                  default: break;
                } /* case 0 switch */
                break;
    } /* switch */
}

clr_terms(i,j)
int i,j;
{
    int k;

    for (k=7; k <= 16; k++) {
        mat[0][i][j].term[k] = 10;
        mat[1][i][j].term[k] = 10;
    }
}

dir_b_mat(i,j,k,blocked,increment)
int i,j,k,blocked,increment;
{
    mat[0][i][j].dir[k].stones += increment;
    if (blocked)
        mat[0][i][j].dir[k].blocked = 1;
}

dir_w_mat(i,j,k,blocked,increment)
int i,j,k,blocked,increment;
{
    mat[1][i][j].dir[k].stones += increment;
```

```
    if (blocked)
        mat[1][i][j].dir[k].blocked = 1;
}

utside(rc)
nt rc;

    if ((rc < 1) || (rc > 19))
        return (1);
    else return (0);


et_mattered(co)
har co;

    nn_set_mat(co);
    ne_set_mat(co);
    ee_set_mat(co);
    es_set_mat(co);
    ss_set_mat(co);
    sw_set_mat(co);
    ww_set_mat(co);
    wn_set_mat(co);
}

nn_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    j = COL;
    for (i = ROW-5; i <= ROW-1; i++) {
       if (flag == 0) flag_set(i,j,&flag,co);
       if ((flag==1)&&(board[i][j] != co))
          if ((outside(i) != 1) && (outside(j) != 1))
              doesnt_matter(co,i,j,0);
    }
}

ne_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    for (i = ROW-5; i <= ROW-1; i++)
       for (j = COL+5; j >= COL+1; j--)
          if ((ROW-i) == (j-COL))  {
             if (flag == 0) flag_set(i,j,&flag,co);
             if ((flag==1)&&(board[i][j] != co))
                if ((outside(i) != 1) && (outside(j) != 1))
                    doesnt_matter(co,i,j,1);
          }
}
```

```
ae_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    i = ROW;
    for (j = COL+5; j >= COL+1; j--) {
       if (flag == 0) flag_set(i,j,&flag,co);
       if ((flag==1)&&(board[i][j] != co))
          if ((outside(i) != 1) && (outside(j) != 1))
              doesnt_matter(co,i,j,2);
    }


 s_set_mat(co)
 ıar co;

    int i, j;
    int flag = 0;

    for (i = ROW+5; i >= ROW+1; i--)
       for (j = COL+5; j >= COL+1; j--)
          if ((i-ROW) == (j-COL))  {
             if (flag == 0) flag_set(i,j,&flag,co);
             if ((flag==1)&&(board[i][j] != co))
                if ((outside(i) != 1) && (outside(j) != 1))
                    doesnt_matter(co,i,j,3);
          }
}

ss_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    j = COL;
    for (i = ROW+5; i >= ROW+1; i--) {
       if (flag == 0) flag_set(i,j,&flag,co);
       if ((flag==1)&&(board[i][j] != co))
          if ((outside(i) != 1) && (outside(j) != 1))
              doesnt_matter(co,i,j,4);
    }
}

sw_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    for (i = ROW+5; i >= ROW+1; i--)
       for (j = COL-5; j <= COL-1; j++)
          if ((i-ROW) == (COL-j))  {
             if (flag == 0) flag_set(i,j,&flag,co);
```

```
            if ((flag==1)&&(board[i][j] != co))
               if ((outside(i) != 1) && (outside(j) != 1))
                   doesnt_matter(co,i,j,5);
         }


  _set_mat(co)
  ar co;

    int i, j;
    int flag = 0;

    i = ROW;
    for (j = COL-5; j <= COL-1; j++)  {
       if (flag == 0) flag_set(i,j,&flag,co);
       if ((flag==1)&&(board[i][j] != co))
          if ((outside(i) != 1) && (outside(j) != 1))
              doesnt_matter(co,i,j,6);
    }


 :)_set_mat(co)
char co;
{
    int i, j;
    int flag = 0;

    for (i = ROW-5; i <= ROW-1; i++)
       for (j = COL-5; j <= COL-1; j++)
          if ((i-ROW) == (j-COL))  {
             if (flag == 0) flag_set(i,j,&flag,co);
             if ((flag==1)&&(board[i][j] != co))
                if ((outside(i) != 1) && (outside(j) != 1))
                    doesnt_matter(co,i,j,7);
          }
}

doesnt_matter(co,i,j,k)
char co;
int i,j,k;
{
    switch (co) {
      case WHITE: if ((board[i][j] == EMPTY) &&
                      (mat[0][i][j].dir[k].mattered ==1)) {
                          mat[0][i][j].dir[k].mattered = 0;
                          mat[0][i][j].dir[(k+4)%8].mattered = 0;
                          term_3(0,i,j);
                  } /* case white if */
                  break;
      case BLACK: if ((board[i][j] == 2) &&
                      (mat[1][i][j].dir[k].mattered ==1)) {
                          mat[1][i][j].dir[k].mattered = 0;
                          mat[1][i][j].dir[(k+4)%8].mattered = 0;
                          term_3(1,i,j);
                  } /* case black if */
                  break;
```

```
      default: error(2); break;
    }
}

ag_set(i,j,flag,co)
it i,j,*flag;
iar co;

    if (*flag == 0)
        if ((outside(i) == 1) || (outside(j) == 1))
           *flag = 1;
    if (*flag == 0)
       switch (co) {
         case WHITE: if (board[i][j] == WHITE)  *flag = 1;  break;
         case BLACK: if (board[i][j] == BLACK)  *flag = 1;  break;
         default: break;
       }   /* switch */
```

```
#include <curses.h>
#include "common.h"

erm_1_5_6(player,i,j)  /* neighbor_relation_evaluation, win_move_evaluation */
nt player,i,j;

    int k;

    mat[player][i][j].term[1] = 0;
    for (k=0; k <= 3; k++)
         if ((mat[player][i][j].dir[k].blocked != 1) &&
             (mat[player][i][j].dir[(k+4)%8].blocked != 1))
                 mat[player][i][j].term[1] += sub_sum(i,j,k,1,player);
         else mat[player][i][j].term[1] += sub_sum(i,j,k,0,player);


erm_2(player,i,j)                       /* block_opponent_evaluation */
nt player;
nt i,j;

    int k;

    mat[player][i][j].term[2] = 0;
    for (k=0; k <= 3; k++)
       if ((mat[flip(player)][i][j].dir[k].blocked != 1) &&
           (mat[flip(player)][i][j].dir[(k+4)%8].blocked != 1))
               mat[player][i][j].term[2] += sub_sum(i,j,k,1,flip(player));
       else mat[player][i][j].term[2] += sub_sum(i,j,k,0,flip(player));
}

flip(k)
int k;
{
    if (k == 0) return (1);
    if (k == 1) return (0);
}

sub_sum(i,j,k,blk,player)
int i, j, k, blk;
int  player;
{
    int subsum;

    if (player == 0) {
        subsum = mat[0][i][j].dir[k].stones
                 + mat[0][i][j].dir[(k+4)%8].stones + 1;
        if (subsum == 5) {
            mat[0][i][j].term[5] = 1000;
            mat[1][i][j].term[6] = 1000;
        }
    } /* if */
    else {
         subsum = mat[1][i][j].dir[k].stones
                  + mat[1][i][j].dir[(k+4)%8].stones + 1;
         if (subsum == 5) {
              mat[0][i][j].term[6] = 1000;
```

```
                mat[1][i][j].term[5] = 1000;
            }
        }
        if (blk == 1)  return (subsum);
        else return (subsum*2);


erm_3(player,i,j)                       /* mattered_count_evaluation */
nt player,i,j;

    mat[player][i][j].term[3] -= 6;


lc_in_out(player, in_r, in_c, out_r, out_c, r, c, t)
nt player, *in_r, *in_c, *out_r, *out_c, r, c, t;

    switch (t) {
        case 0: *in_c = c;
                *in_r = r + mat[player][r][c].dir[4].stones + 1;
                *out_c = c;
                *out_r = r - mat[player][r][c].dir[0].stones - 1;
                break;
        case 1: *in_c = c - mat[player][r][c].dir[5].stones - 1;
                *in_r = r + mat[player][r][c].dir[5].stones + 1;
                *out_c = c + mat[player][r][c].dir[1].stones + 1;
                *out_r = r - mat[player][r][c].dir[1].stones - 1;
                break;
        case 2: *in_c = c - mat[player][r][c].dir[6].stones - 1;
                *in_r = r;
                *out_c = c + mat[player][r][c].dir[2].stones + 1;
                *out_r = r;
                break;
        case 3: *in_c = c - mat[player][r][c].dir[7].stones - 1;
                *in_r = r - mat[player][r][c].dir[7].stones - 1;
                *out_c = c + mat[player][r][c].dir[3].stones + 1;
                *out_r = r + mat[player][r][c].dir[3].stones + 1;
                break;
        case 4: *in_c = c;
                *in_r = r - mat[player][r][c].dir[0].stones - 1;;
                *out_c = c;
                *out_r = r + mat[player][r][c].dir[4].stones + 1;
                break;
        case 5: *in_c = c + mat[player][r][c].dir[1].stones + 1;
                *in_r = r - mat[player][r][c].dir[1].stones - 1;
                *out_c = c - mat[player][r][c].dir[5].stones - 1;
                *out_r = r + mat[player][r][c].dir[5].stones + 1;
                break;
        case 6: *in_c = c + mat[player][r][c].dir[2].stones + 1;
                *in_r = r;
                *out_c = c - mat[player][r][c].dir[6].stones - 1;
                *out_r = r;
                break;
        case 7: *in_c = c + mat[player][r][c].dir[3].stones + 1;
                *in_r = r + mat[player][r][c].dir[3].stones + 1;
                *out_c = c - mat[player][r][c].dir[7].stones - 1;
                *out_r = r - mat[player][r][c].dir[7].stones - 1;
```

```
                break;
        default: break;
    } /* switch */


    term_7    offend_open_3_evaluation */
    term_9    offend_close_4_evaluation */
    term_11   offend_cross_3_evaluation */
    term_13   offend_setup_evaluation */
    term_15   offend_open_4_evaluation */

rm_7_9_11_13_15(r,c,t)
nt r,c,t;

    int in_r, in_c, out_r, out_c;

    calc_in_out(0,&in_r,&in_c,&out_r,&out_c,r,c,t);
    clr_flag(0,r,c,t);
    current(0,r,c,t,out_r,out_c,in_r,in_c);
    calc_term_offend(r,c);
    if (board[out_r][out_c] == EMPTY) {
        r = out_r;
        c = out_c;
        calc_in_out(0,&in_r,&in_c,&out_r,&out_c,r,c,t);
        clr_flag(0,r,c,t);
        current(0,r,c,t,out_r,out_c,in_r,in_c);
        calc_term_offend(r,c);
    }
)

/*  term_8    defend_open_3_evaluation */
/*  term_10   defend_close_4_evaluation */
/*  term_12   defend_cross_3_evaluation */
/*  term_14   defend_setup_evaluation */
/*  term_16   defend_open_4_evaluation */

term_8_10_12_14_16(r,c,t)
int r,c,t;
{
    int in_r, in_c, out_r, out_c;

    calc_in_out(1,&in_r,&in_c,&out_r,&out_c,r,c,t);
    clr_flag(1,r,c,t);
    current(1,r,c,t,out_r,out_c,in_r,in_c);
    calc_term_defend(r,c);
    if (board[out_r][out_c] == EMPTY) {
        r = out_r;
        c = out_c;
        calc_in_out(1,&in_r,&in_c,&out_r,&out_c,r,c,t);
        clr_flag(1,r,c,t);
        current(1,r,c,t,out_r,out_c,in_r,in_c);
        calc_term_defend(r,c);
    }
)

current(player,r,c,t,out_r,out_c,in_r,in_c)
```

```
int player,r,c,t,out_r,out_c,in_r,in_c;
{
    int subsum;
    int in_blk, out_blk;

    subsum = mat[player][r][c].dir[t].stones +
             mat[player][r][c].dir[(t+4)%8].stones + 1;
    in_blk = mat[player][r][c].dir[(t+4)%8].blocked;
    out_blk = mat[player][r][c].dir[t].blocked;
    switch (subsum) {
        case 4: if ((in_blk == 0) && (out_blk == 0))
                    switch (player) {
                       case 0: set_open4_flag(0,r,c,t);
                               break;
                       case 1: set_open4_flag(1,r,c,t);
                               break;
                       default:break;
                    }
                else set_close_flag(player,r,c,t);
                break;
        case 3: if ((in_blk == 0) && (out_blk == 0))
                    set_open3_flag(player,r,c,t);
                if ((board[in_r][in_c] == EMPTY) && (in_blk == 0) &&
                    (mat[player][in_r][in_c].dir[(t+4)%8].stones == 1))
                     set_close_flag(player,r,c,t);
                if ((board[out_r][out_c] == EMPTY) && (out_blk == 0) &&
                    (mat[player][out_r][out_c].dir[t].stones == 1))
                     set_close_flag(player,r,c,t);
                break;
         case 2: if ((in_blk == 0) && (out_blk == 0)) {
                    if (board[in_r][in_c] == EMPTY)          /* check inward */
                       if (mat[player][in_r][in_c].dir[(t+4)%8].blocked == 0)
                          switch (mat[player][in_r][in_c].dir[(t+4)%8].stones) {
                             case 1: set_open3_flag(player,r,c,t); break;
                             case 2: set_close_flag(player,r,c,t); break;
                             default: break;
                          } /* switch */
                       else if (mat[player][in_r][in_c].dir[(t+4)%8].stones == 2)
                             set_close_flag(player,r,c,t);
                    if (board[out_r][out_c] == EMPTY)        /* check outward */
                        if (mat[player][out_r][out_c].dir[t].blocked == 0)
                           switch (mat[player][out_r][out_c].dir[t].stones) {
                              case 1: set_open3_flag(player,r,c,t); break;
                              case 2: set_close_flag(player,r,c,t); break;
                              default: break;
                           } /* switch */
                    else if (mat[player][out_r][out_c].dir[t].stones == 2)
                             set_close_flag(player,r,c,t);
                } /* if */
                else {
                     if (board[in_r][in_c] == EMPTY)
                        if ((in_blk == 0) &&
                            (mat[player][in_r][in_c].dir[(t+4)%8].stones == 2))
                             set_close_flag(player,r,c,t);
                     if (board[out_r][out_c] == EMPTY)
                        if ((out_blk == 0) &&
```

```
                        (mat[player][out_r][out_c].dir[t].stones == 2))
                         set_close_flag(player,r,c,t);
                } /* else */
                break;
        case 1: if ((in_blk == 0) && (out_blk == 0)) {
                   if (board[in_r][in_c] == EMPTY)           /* inward check */
                      if (mat[player][in_r][in_c].dir[(t+4)%8].blocked == 0)
                         switch (mat[player][in_r][in_c].dir[(t+4)%8].stones) {
                            case 2: set_open3_flag(player,r,c,t); break;
                            case 3: set_close_flag(player,r,c,t); break;
                            default: break;
                         } /* switch */
                      else if (mat[player][in_r][in_c].dir[(t+4)%8].stones == 3)
                               set_close_flag(player,r,c,t);
                   if (board[out_r][out_c] == EMPTY)         /* outward check */
                      if (mat[player][out_r][out_c].dir[t].blocked == 0)
                         switch (mat[player][out_r][out_c].dir[t].stones) {
                           case 2: set_open3_flag(player,r,c,t); break;
                           case 3: set_close_flag(player,r,c,t); break;
                           default: break;
                         } /* switch */
                       else if (mat[player][out_r][out_c].dir[t].stones == 3)
                               set_close_flag(player,r,c,t);
                } /* if */
                else {
                      if (board[in_r][in_c] == EMPTY)        /* outward check */
                         if ((in_blk == 0) &&
                             (mat[player][in_r][in_c].dir[(t+4)%8].stones == 3))
                              set_close_flag(player,r,c,t);
                      if (board[out_r][out_c] == EMPTY)      /* outward check */
                         if ((out_blk == 0) &&
                             (mat[player][out_r][out_c].dir[t].stones == 3))
                              set_close_flag(player,r,c,t);
                } /* else */
                break;
        default: break;
    }
}

clr_flag(player,r,c,t)
int player, r, c, t;
{
    mat[player][r][c].dir[t].open_4 = 0;
    mat[player][r][c].dir[(t+4)%8].open_4 = 0;
    mat[player][r][c].dir[t].open_3 = 0;
    mat[player][r][c].dir[(t+4)%8].open_3 = 0;
    mat[player][r][c].dir[t].close_4 = 0;
    mat[player][r][c].dir[(t+4)%8].close_4 = 0;
}

set_close_flag(player,r,c,t)
int player,r,c,t;
{
    mat[player][r][c].dir[t].close_4 = 1;
    mat[player][r][c].dir[(t+4)%8].close_4 = 1;
}
```

```
set_open3_flag(player,r,c,t)
int player,r,c,t;
{
    mat[player][r][c].dir[t].open_3 = 1;
    mat[player][r][c].dir[(t+4)%8].open_3 = 1;


 et_open4_flag(player,r,c,t)
 nt player,r,c,t;

    mat[player][r][c].dir[t].open_4 = 1;
    mat[player][r][c].dir[(t+4)%8].open_4 = 1;


 alc_term_offend(r,c)
 nt r,c;

    int k;
    int count = 0;

    for (k=0; k <= 3; k++) {
        if (mat[0][r][c].dir[k].open_4 == 1) {
            mat[0][r][c].term[15] = 1000;
            mat[1][r][c].term[16] = 1000;
        }
        if (mat[0][r][c].dir[k].open_3 == 1) {
             mat[0][r][c].term[7] = 25;
             mat[1][r][c].term[8] = 25;
             count += 1;
             if (find_match_4(0,r,c,k) == 1) {
                 mat[0][r][c].term[13] = 1000;
                 mat[1][r][c].term[14] = 1000;
             }
        }
        if (mat[0][r][c].dir[k].close_4 == 1) {
             mat[0][r][c].term[9] = 25;
             mat[1][r][c].term[10] = 25;
             if (find_match_4(0,r,c,k) == 1) {
                 mat[0][r][c].term[13] = 1000;
                 mat[1][r][c].term[14] = 1000;
             }
        }
    }
    if (count >= 2) {
        mat[0][r][c].term[11] = 1000;
        mat[1][r][c].term[12] = 1000;
    }
}

find_match_4(player, r, c, k)
int player, r, c, k;
{
    int x;

    for (x=1; x <=3; x++)
```

```
        if ((x != k) && (mat[player][r][c].dir[x].close_4 == 1))
            return (1);
    return (0);


alc_term_defend(r,c)
nt r,c;

    int k;
    int count = 0;

    for (k=0; k <= 3; k++) {
        if (mat[1][r][c].dir[k].open_4 == 1) {
            mat[0][r][c].term[16] = 1000;
            mat[1][r][c].term[15] = 1000;
        }
        if (mat[1][r][c].dir[k].open_3 == 1) {
             mat[0][r][c].term[8] = 25;
             mat[1][r][c].term[7] = 25;
             count += 1;
             if (find_match_4(1,r,c,k) == 1) {
                 mat[0][r][c].term[14] = 1000;
                 mat[1][r][c].term[13] = 1000;
             }
        }
        if (mat[1][r][c].dir[k].close_4 == 1) {
             mat[0][r][c].term[10] = 25;
             mat[1][r][c].term[9] = 25;
             if (find_match_4(1,r,c,k) == 1) {
                 mat[0][r][c].term[14] = 1000;
                 mat[1][r][c].term[13] = 1000;
             }
        }
    }
    if (count >= 2) {
        mat[0][r][c].term[12] = 1000;
        mat[1][r][c].term[11] = 1000;
    }
}
```

```
#include <curses.h>
#include "common.h"

visual_mov(color)
char color;

    char c;
    int row, col;
    int made;

    clr_prv_mov_mesg();
    row = ROW;
    col = COL;
    whose_mov(color);
    move(row+1,col*2+3);
    refresh();
    made = 0;
    do {
       c = getch();
       clear_mesg_line();
       mvaddch(row+1,col*2+3,board[row][col]);
       switch (c) {
         case '\010':
         case 'h': col -= 1; break;
         case 'j': row += 1; break;
         case 'k': row -= 1; break;
         case 'l': col += 1; break;
         case '\n': made = test_bad(row,col);
                    break;
         default : break;
       }
       if (over_mov(&row,&col) == 1)
           outside_mov(row,col);
       else move(row+1,col*2+3);
       refresh();
    }
    while (made == 0);

    make_move(row,col,color);
}

make_move(row,col,color)
int row, col;
char color;
{
    board[row][col] = color;
    ROW = row;
    COL = col;
    whose_mov(color);
    up_screen(row,col,color);
    if (mode == 4) restart_mesg();
}

up_screen(r, c, co)
int r,c;
char co;
```

```
    mvaddch(r+1,c*2+3,co);
    recover_rest_of_line(r,c);
    if (co == WHITE)
        mov_mesg(r,c,12);
    else {
         mov_mesg(r,c,15);
         if (mode == 1)
            rating_mesg(r,c);
    }
    refresh();
    if ((mode == 1) && (co == BLACK)) {
        gets(buf);
        rate = atoi(buf);
    }
    move(r+1,c*2+3);
    if (co == WHITE) clr_stars();
    refresh();


recover_rest_of_line(r,c)
int r,c;
{

    while (++c <= 19)
        mvaddch(r+1,c*2+3,board[r][c]);
}

mov_mesg(r,c,x)
int r, c, x;
(
    char s[3];

    mvaddstr(x,66,"row    ");
    clrs(s);
    itoa(r,s);
    mvaddstr(x,74,s);
    mvaddstr(x+1,66,"column ");
    clrs(s);
    itoa(c,s);
    mvaddstr(x+1,74,s);
}

whose_mov(co)
char co;
(
    if (co == WHITE) (
       mvaddstr(12,50,"YOUR MOVE ...");
       mvaddstr(13,50,"(WHITE)");
    }
    else (
         mvaddstr(15,50,"MY MOVE ...");
         mvaddstr(16,50,"(BLACK)");
    }
}
```

```
iting_mesg(r,c)
nt r, c;

    mvaddstr(15,46,'    ");
    mvaddstr(18,46,"--> Rate of this move is...");
    move(18,74);


lrs(s)
har s[];

    int i;

    for (i=0; i <= 2; i++)
        s[i] = ' ';


est_bad(r,c)
nt r,c;
(
    if (occupied(r,c) == 1) (
        bad_mov(r,c);
        return (0);
    }
    else
         return (1);
}

over_mov(r,c)
int *r, *c;
(
    if ((*r >= 1) && (*r <= 19) && (*c >= 1) && (*c <= 19))
        return (0);
    else (
         if (*r < 1)             /* row underflow */
            *r += 1;
         else if (*r > 19)               /* row overflow */
                 *r -= 1;
              else if (*c < 1)           /* column underflow */
                      *c += 1;
                   else if (*c > 19)          /* column overflow */
                           *c -= 1;
                        else ;
         return (1);
         } /* else */
}

outside_mov(r,c)
int r, c;
(
    recover_rest_of_line(r,c);
    mvaddstr(ERR_LINE,6,"you have reached the boundary");
    move(r+1,c*2+3);
}

occupied(r,c)
```

```
int r, c;

    if ((board[r][c] == 'X') || (board[r][c] == 'O'))
       return (1);
    else return (0);


ad_mov(r,c)
nt r, c;

    recover_rest_of_line(r,c);
    mvaddstr(ERR_LINE,6,"this move was already played");
    move(r+1,c*2+3);


lr_prv_mov_mesg()

    restart_mesg();
    move(18,50);
    clrtoeol();
}

restart_mesg()
{
    mvaddstr(15,46,      ");
    mvaddstr(18,46,      ");
    mvaddstr(12,46,"--> ");
    move(12,66);
    clrtoeol();
    move(13,66);
    clrtoeol();
}

clr_stars()
{
    mvaddstr(12,46,      ");
    mvaddstr(15,46,"--> ");
}
```

```
include <curses.h>
include "common.h"

in_chk(color)
har color;

    if (five_check(color) == 1) (
        end_round(color);
        if (mode == 1)
            store_weight();
        return (1);
    } /* if */
    else  return (0);


ive_check(co)
har co;

    if (co == WHITE)  (                                          /* check white winre */
       if ((mat[1][ROW][COL].dir[0].stones+mat[1][ROW][COL].dir[4].stones==4)
         ||(mat[1][ROW][COL].dir[1].stones+mat[1][ROW][COL].dir[5].stones==4)
         ||(mat[1][ROW][COL].dir[2].stones+mat[1][ROW][COL].dir[6].stones==4)
         ||(mat[1][ROW][COL].dir[3].stones+mat[1][ROW][COL].dir[7].stones==4))
          return (1);
       else return (0);
    }  /* if */
    else (                                                       /* check black win */
       if ((mat[0][ROW][COL].dir[0].stones+mat[0][ROW][COL].dir[4].stones==4)
         ||(mat[0][ROW][COL].dir[1].stones+mat[0][ROW][COL].dir[5].stones==4)
         ||(mat[0][ROW][COL].dir[2].stones+mat[0][ROW][COL].dir[6].stones==4)
         ||(mat[0][ROW][COL].dir[3].stones+mat[0][ROW][COL].dir[7].stones==4))
          return (1);
       else return (0);
    } /* else */
}

end_round(co)
char co;
(
    beep();
    clr_mesg_space();
    switch (co) (
      case WHITE: mvaddstr(12,50,"CONGRATULATIONS !!!!!");
                  mvaddstr(14,50,"You win");
                  mvaddstr(16,50,"Believe me");
                  mvaddstr(17,50,"This won't last long.");
                  if ((mode == 2) || (mode == 4))
                       retrieve_lose_terms();
                  break;
      case BLACK: mvaddstr(12,50,"Terribly SORRY !!!!!");
                  mvaddstr(14,50,"I beat you!");
                  mvaddstr(16,50,"You play like a greenhorn!");
                  mvaddstr(17,50,"Get yourself more training...");
                  if ((mode == 2) || (mode == 4))
                       retrieve_win_terms();
```

```
                    break;
      default: error(2); break;
    }
    round_mesg();


)und_mesg()

    mvaddstr(ERR_LINE,7,"Hit return to end this round !!!!!!!!!! ");
    refresh();
    getstr(buf);


 r_mesg_space()

    int i;

    for (i= 12; i <= 20; i++) {
        move(i,46);
        clrtoeol();
    }
}

retrieve_win_terms()
{
    struct _step *current;
    int k;

    current = head;
    while (current != NULL) {
        for (k=0; k <= 16; k++)
            switch (current->term[k]) {
               case 0:
               case 1:
               case 2:
               case 3:
               case 4: if (weight[k] > 4)
                           weight[k] -= 4;
                       break;
               case 5:
               case 6:
               case 7:
               case 8:
               case 9: if (weight[k] > 2)
                           weight[k] -= 2;
                       break;
               case 10:
               case 11:
               case 12:
               case 13:
               case 14: break;
               case 15:
               case 16:
               case 17:
               case 18:
               case 19: weight[k] += 2; break;
```

```
                default: weight[k] += 4; break;
            } /* switch */
        current = current->next;
    } /* while */
    write_back_to_post();
}

write_back_to_post()
{
    int i;

    lseek(p_fd,0,0);
    for (i=0; i <= 16; i++) {
        err = write(p_fd,&weight[i],sizeof(int));
        if (err == 0)  error(6);
    }
}

retrieve_lose_terms()
{
    struct _step *current;
    int k;

    current = head;
    while (current != NULL) {
        for (k=0; k <= 16; k++)
            switch (current->term[k]) {
               case 0:
               case 1:
               case 2:
               case 3:
               case 4: weight[k] += 4; break;
               case 5:
               case 6:
               case 7:
               case 8:
               case 9: weight[k] += 2; break;
               case 10:
               case 11:
               case 12:
               case 13:
               case 14: break;
               case 15:
               case 16:
               case 17:
               case 18:
               case 19: if (weight[k] > 2)
                            weight[k] -= 2;
                        break;
               default: if (weight[k] > 4)
                            weight[k] -= 4;
                        break;
            } /* switch */
        current = current->next;
    } /* while */
    write_back_to_post();
```

```
}

store_weight()
{
    int i;

    lseek(f_fd,0,0);
    for (i=0; i <= 16; i++) {
        err = write(f_fd,&weight[i],4);
        if (err == 0)  error(5);
    }
}
```