CS348

Summer 2001

MP 1: GoMoku Game

Due: Thursday, June 28

Name: Joseph S Kim

DESCRIPTION

This machine problem consists of developing and implementing a Min-Max search algorithm in Java for the game of GoMoku(also known as Pente or Five Stones). GoMoku, also known as "five in a row", is a board game similar to tic-tac-toe but on a larger board (13x13).

IMPLEMENTATION

GoMoku is implemented in Java as a web applet in an object-oriented manner. Java enables a smooth, object-oriented development, yet with a tradeoff in performance. The main task is to write a class called **MinMax**, which contains several important member functions, including evalBoard() and minimaxValue().

- **evalBoard()**

  The outcome of the program is mostly determined by the evaluation function, which estimates how close a state is to the goal. The evaluation function assigns points to each state.
  My heuristic function for evaluating the states of the board is the one provided on the web. For each of our stones, count the number of our stones in the 8 neighbors, and subtract the number of our opponent's stones in the 8 neighbors. In my observation from playing with **DeepOrange**, he doesn't really have smart move. Since My design decision in heuristic function is somewhat simple and it doesn't really evaluate the value of each stone.
  For **evalBoard()**, I basically used loops for tracking down two-dimensional array, 3x3 and 13x13 board. Depending on what 'forwhom' is, it returns the value of board evaluation. I commented on my design decisions in the following code.

```
// Function: evalBoard
// Arguments: forwhom -- BLACK or WHITE
// Returns: board evaluation
// Description:
int evalBoard(int forwhom)
{
     int NumWhite = 0,  //number of white stone in the 8 neighbors
         NumBlack = 0,  //number of black stone in the 8 neighbors
         Eval = 0;      //return value of board evaluation

     for (int row = 0; row < 13; row++)
```

```
        for(int col = 0; col < 13; col++)  //track down whole board
            {
                int tempR = row - 1;   //temporary values not to
                 int tempC = col - 1;   //change value of row, col
                int tempRR = row + 1;
                int tempCC = col + 1;

                // For each of our stones, count the number of our
                // stones in the neighbors,
                for(int rr = tempR; rr <= tempRR; rr++)
                    for(int cc = tempC; rr <= tempCC; cc++)
                        {
                            if (workingboard[rr][cc] == 1) //white
                                NumWhite++;
                            else if (workingboard[rr][cc] == 2)//black
                                NumBlack++;
                            else
                                return 0;//empty spot or out of board.
                           }//out for loop that evaluates 9 blocks out of
                            // whole 13x13 board

    // subtract the number of our opponent's stones in the 8 neighbors.
                if (forwhom == 2)     //black stone
                    Eval = NumBlack - NumWhite - 1;
                else if (forwhom == 1)    //white stone
                    Eval = NumWhite - NumBlack - 1;

            }//out of for loop
    return Eval;
}
```

- **minimaxValue()**

This function is for the MiniMax search algorithm using my heuristic function above. This algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. Consequently, a player tries to get to the goal state first by maximizing his points and/or minimizing the opponent's scores.
In the first if case, I applied the utility function(evalBoard(toplay)) to each terminal state to get its value(terminal test). And then, in the else case it returns the operator that corresponding to the best possible move, that is, the move that lead to the outcome with the best utility, under the assumption that the player plays to minimize utility. This function **minimaxValue()** goes trough the whole tree, all the way to the leaves to determine the back-up value of a state by the recursive call.

```
// Function: minimaxValue
// Arguments: toplay -- BLACK or WHITE
//            depth -- number of levels left to explore
// Returns: a minimax value for the state
// Description: The main part of the Minimax algorithm.
int minimaxValue(int toplay, int depth)
{
    int min = -10000; //arbitrary number to compare with eval value.
    int max = 10000;
    depth--;          //Since the depth was initially 3.
                      //It decreases by one every recursive call.
    if (depth == 0)   //This is the case for the last node of tree
        {
            return evalBoard(toplay);
```

```
            }

        else // When we are in the middle of tree.
            {

                for (int row = 0; row < 13; row++)
                    for ( int col = 0; col < 13; col++){

                        if ((nextToPiece(row,col))&&(workingboard[row][col]==
EMPTY)) //there is adjacent stone and current row and col is empty.
                        {
                            if (toplay == BLACK){
                                int evalmax = minimaxValue(BLACK,depth);
                                if (max > evalmax)
                                    max = evalmax; //highest minimax value

                            }

                            if (toplay == WHITE){
                                int evalmin = minimaxValue(WHITE,depth);
                                if (min < evalmin)
                                    min = evalmin; //lowest minimax value

                            }
                        }//out of if clause
                    }//out of for loops
            }//out of else cases

        if (toplay == BLACK)
            return max;          //return the highest minimax value
        else                     // if toplay is WHITE
            return min;          //return the lowest minimax value

    }
```

OBSERVATION

By playing with **DeepOrange**, I realized that the MiniMax search algorithm using my simple heuristic function is not efficient enough. It takes a lot of time to search the tree and evaluate the value. Also, the move of **DeepOrange** is level of novice. I need to make more efficient evaluation function. Then, the move of **DeepOrange** will be improved.