```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* This code is a translation of Norvig's (1992) LISP implementation,
though it deviates from it in several respects. In theory you might be
able to track down this code and turn much of it in to satisfy the
project requirements, but consistent with the honor code (and ethics
generally), you would declare this use and would receive a MUCH
reduced grade.

Douglas H. Fisher
*/

/*     How to play the Game of Othello

   Othello is a game between two players on an 8x8 square grid
   (64 positions). The two players, represented as black (b)
   and white (w), initially occupy the center positions as
   indicated below.

   1 2 3 4 5 6 7 8
1  . . . . . . . . .
2  . . . . . . . . .
3  . . . . . . . . .
4  . . . w b . . .
5  . . . b w . . .
6  . . . . . . . . .
7  . . . . . . . . .
8  . . . . . . . . .

  Black moves first, and places a 'b' somewhere on an *empty* location
  (indicated by a '.'). There are constraints on where the b may be
  placed though (other than 'empty'). Notably, the b must be placed
  so that it brackets or sandwiches a contiguous sequence of
  opponent pieces. So, a b can be placed at coordinates (3,4), (4,3),
(5,6),
  or (6,5). When a b is placed that causes a sequence of ws to be
bracketed,
  the ws that are bracketed are flipped to b (i.e., from the
opponent's
  color to the current player's color). So, placing a 'b' at (5,6)
  leads to the following board:

   1 2 3 4 5 6 7 8
1  . . . . . . . . .
2  . . . . . . . . .
3  . . . . . . . . .
4  . . . w b . . .
5  . . . b b b . .
```

```
6  . . . . . . . .
7  . . . . . . . .
8  . . . . . . . .
```

White now moves. Any move must bracket at least one b piece, and after doing so, the bs in the bracketed sequence are switched to ws. For the board above, w could be placed on (6,4), (4,6), or (6,6) -- note that one can bracket pieces along a diagnoal. If w is placed on (6,6), the resulting board is:

```
   1 2 3 4 5 6 7 8
1  . . . . . . . .
2  . . . . . . . .
3  . . . . . . . .
4  . . . w b . . .
5  . . . b w b . .
6  . . . . . w . .
7  . . . . . . . .
8  . . . . . . . .
```

Play continues:

```
   1 2 3 4 5 6 7 8      b to (4,3)
1  . . . . . . . .
2  . . . . . . . .
3  . . . . . . . .
4  . . b b b . . .
5  . . . b w b . .
6  . . . . . w . .
7  . . . . . . . .
8  . . . . . . . .
```

```
   1 2 3 4 5 6 7 8      w to (5,3)
1  . . . . . . . .
2  . . . . . . . .
3  . . . . . . . .
4  . . b b b . . .
5  . . w w w b . .
6  . . . . . w . .
7  . . . . . . . .
8  . . . . . . . .
```

```
   1 2 3 4 5 6 7 8      b to (6,5). Notice that this brackets
1  . . . . . . . .      a white piece along a vertical AND diagonal
2  . . . . . . . .      path, causing both sequences to be switched
3  . . . . . . . .      to b
4  . . b b b . . .
```

```
5  . . w b b b . .
6  . . . . b w . .
7  . . . . . . . .
8  . . . . . . . .
```

```
   1 2 3 4 5 6 7 8
1  . . . . . . . .
2  . . . . . . . .
3  . . w . . . . .
4  . . w w b . . .
5  . . w b w b . .
6  . . . . b w . .
7  . . . . . . . .
8  . . . . . . . .
```

w to (3,3). Again, this brackets b along two sequences. In general, all bracketed sequences (there may by many) are swaped to the current players color.

Play continues until neither player can move. Very often, this happens when all sqaures are filled, but both players may be unable to bracket a sequence of opponents' pieces before all squares are filled, and so the game stops before all squares are filled. The player with the most squares filled at game's end, wins.

If one player cannot move (because they cannot bracket an opposing sequence), but the other player can move, then the former player must pass until such time that they can move.

Note that when running the program, where one of the players is human, a board will be printed as follows:

```
    1 2 3 4 5 6 7 8
10  . . . . . . . .
20  . . . . . . . .
30  . . w . . . . .
40  . . w w b . . .
50  . . w b w b . .
60  . . . . b w . .
70  . . . . . . . .
80  . . . . . . . .
```

Specify a move by the sum of the appropriate row and column. So, for example, 42 would be a legal move for black. If you specify an illegal move (e.g., 23 in this example), you will be prompted for another move.

```
/*****************************************************************
******************** GLOBAL VARS and CONSTANTS *****************
*****************************************************************/

/* global variables and constants are given in all CAPS */

const int ALLDIRECTIONS[8]={-11, -10, -9, -1, 1, 9, 10, 11};
const int BOARDSIZE=100;
```

```
/* Each array/board position can have one of 4 values */

const int EMPTY=0;
const int BLACK=1;
const int WHITE=2;
const int OUTER=3;          /* the value of a square on the perimeter
*/

const int WIN=2000;         /* a WIN and LOSS for player are outside
*/
const int LOSS= -2000;      /* the range of any board evaluation
function */


/* the global variable, BOARDS, is used to count the number of BOARDS
   examined during minmax search (and is printed during a roundrobin
   tournament). See the end of this file for sample results of
   roundrobin tournaments.
*/

long int BOARDS;

/* STRATEGIES is an array of strategy names (for printing) and
   pointers to the functions that implement each strategy. It
   is the only structure/statement/function that must be
   modified when a new strategy is added (other than adding
   the actual function(s) that implements a new strategy.
*/

int human (int, int *);
int randomstrategy(int, int *);
int maxdiffstrategy1(int, int *);
int maxdiffstrategy3(int, int *);
int maxdiffstrategy5(int, int *);
int maxweighteddiffstrategy1(int, int *);
int maxweighteddiffstrategy3(int, int *);
int maxweighteddiffstrategy5(int, int *);

void * STRATEGIES[9][4] = {
   {"human", "human", human},
   {"random", "random", randomstrategy},
   {"diff1", "maxdiff, 1-move minmax lookahead", maxdiffstrategy1},
   {"diff3", "maxdiff, 3-move minmax lookahead", maxdiffstrategy3},
   {"diff5", "maxdiff, 5-move minmax lookahead", maxdiffstrategy5},
   {"wdiff1", "maxweigteddiff, 1-move minmax lookahead",
             maxweighteddiffstrategy1},
   {"wdiff3", "maxweigteddiff, 3-move minmax lookahead",
             maxweighteddiffstrategy3},
   {"wdiff5", "maxweigteddiff, 5-move minmax lookahead",
```

```
             maxweighteddiffstrategy5},
   {NULL, NULL, NULL}
 };


typedef int (* fpc) (int, int *);


/
***********************************************************************
*/
/************************* Auxiliary Functions
*************************/
/
***********************************************************************
*/

/* the possible square values are integers (0-3), but we will
   actually want to print the board as a grid of symbols, . instead of
0,
   b instead of 1, w instead of 2, and ? instead of 3 (though we
   might only print out this latter case for purposes of debugging).
*/

char nameof (int piece) {
  static char piecenames[5] = ".bw?";
  return(piecenames[piece]);
}


/* if the current player is BLACK (1), then the opponent is WHITE (2),
   and vice versa
*/

int opponent (int player) {
  switch (player) {
  case 1: return 2;
  case 2: return 1;
  default: printf("illegal player\n"); return 0;
  }
}


/* The copyboard function mallocs space for a board, then copies
   the values of a given board to the newly malloced board.
*/

int * copyboard (int * board) {
  int i, * newboard;
  newboard = (int *)malloc(BOARDSIZE * sizeof(int));
```

```
  for (i=0; i<BOARDSIZE; i++) newboard[i] = board[i];
  return newboard;
}


/* the initial board has values of 3 (OUTER) on the perimeter,
   and EMPTY (0) everywhere else, except the center locations
   which will have two BLACK (1) and two WHITE (2) values.
*/

int * initialboard (void) {
  int i, * board;
  board = (int *)malloc(BOARDSIZE * sizeof(int));
  for (i = 0; i<=9; i++) board[i]=OUTER;
  for (i = 10; i<=89; i++) {
     if (i%10 >= 1 && i%10 <= 8) board[i]=EMPTY; else board[i]=OUTER;
  }
  for (i = 90; i<=99; i++) board[i]=OUTER;
  board[44]=WHITE; board[45]=BLACK; board[54]=BLACK; board[55]=WHITE;
  return board;
}


/* count the number of squares occupied by a given player (1 or 2,
   or alternatively BLACK or WHITE)
*/

int count (int player, int * board) {
  int i, cnt;
  cnt=0;
  for (i=1; i<=88; i++)
    if (board[i] == player) cnt++;
  return cnt;
}



/* The printboard routine does not print "OUTER" values
   of the array, since these are not squares of the board.
   Rather it examines all other locations and prints the
   symbolic representation (.,b,w) of each board square.
   So the initial board would be printed as follows:

   1 2 3 4 5 6 7 8 [b=2 w=2]
10 . . . . . . . .
20 . . . . . . . .
30 . . . . . . . .
40 . . . w b . . .
50 . . . b w . . .
60 . . . . . . . .
```

```
70 . . . . . . . . .
80 . . . . . . . . .

Notice that if you add row, r, and column, c, numbers, you
get the array location that corresponds to (r,c). So
square (50,6) corresponds to 50+6=56.

*/

void printboard (int * board) {
  int row, col;
  printf("    1 2 3 4 5 6 7 8 [%c=%d %c=%d]\n",
      nameof(BLACK), count(BLACK, board), nameof(WHITE), count(WHITE,
board));
  for (row=1; row<=8; row++) {
    printf("%d  ", 10*row);
    for (col=1; col<=8; col++)
      printf("%c ", nameof(board[col + (10 * row)]));
    printf("\n");
  }
}



/
**********************************************************************
*
********************* Routines for insuring legal play
*****************
**********************************************************************
*/

/*
The code that follows enforces the rules of legal play for Othello.
*/

/* a "valid" move must be a non-perimeter square */

int validp (int move) {
  if ((move >= 11) && (move <= 88) && (move%10 >= 1) && (move%10 <=
8))
     return 1;
  else return 0;
}


/* findbracketingpiece is called from wouldflip (below).
   findbracketingpiece starts from a *square* that is occupied
   by a *player*'s opponent, moves in a direction, *dir*, past
   all opponent pieces, until a square occupied by the *player*
```

```
   is found. If a square occupied by *player* is not found before
   hitting an EMPTY or OUTER square, then 0 is returned (i.e., no
   bracketing piece found). For example, if the current board is

         1 2 3 4 5 6 7 8
     10  . . . . . . . .
     20  . . . . . . . .
     30  . . . . . . . .
     40  . . b b b . . .
     50  . . w w w b . .
     60  . . . . . w . .
     70  . . . . . . . .
     80  . . . . . . . .

   then findbracketingpiece(66, BLACK, board, -11) will return 44
        findbracketingpiece(53, BLACK, board, 1) will return 56
        findbracketingpiece(55, BLACK, board, -9) will return 0

*/


int findbracketingpiece(int square, int player, int * board, int dir)
{
  while (board[square] == opponent(player)) square = square + dir;
  if (board[square] == player) return square;
  else return 0;
}

/* wouldflip is called by legalmove (below). Give a *move* (square)
   to which a player is considering moving, wouldflip returns
   the square that brackets opponent pieces (along with the sqaure
   under consideration, or 0 if no such bracketing square exists
*/

int wouldflip (int move, int player, int * board, int dir) {
  int c;
  c = move + dir;
  if (board[c] == opponent(player))
     return findbracketingpiece(c+dir, player, board, dir);
  else return 0;
}


/* A "legal" move is a valid move, but it is also a move/location
  that will bracket a sequence of the opposing player's pieces,
  thus flipping at least one opponent piece. legalp considers a
  move/square and seaches in all directions for at least one
bracketing
  square. The function returns 1 if at least one bracketing square
  is found, and 0 otherwise.
```

```
*/

int legalp (int move, int player, int * board) {
  int i;
  if (!validp(move)) return 0;
  if (board[move]==EMPTY) {
    i=0;
    while (i<=7 && !wouldflip(move, player, board, ALLDIRECTIONS[i]))
i++;
    if (i==8) return 0; else return 1;
  }
  else return 0;
}


/* makeflips is called by makemove. Once a player has decided
   on a move, all (if any) opponent pieces that are bracketed
   along a given direction, dir, are flipped.
*/

void makeflips (int move, int player, int * board, int dir) {
  int bracketer, c;
  bracketer = wouldflip(move, player, board, dir);
  if (bracketer) {
     c = move + dir;
     do {
         board[c] = player;
         c = c + dir;
        } while (c != bracketer);
  }
}

/* makemove actually places a players symbol (BLACK or WHITE)
in the location indicated by move, and flips all opponent
squares that are now bracketed (along all directions)
*/

void makemove (int move, int player, int * board) {
  int i;
  board[move] = player;
  for (i=0; i<=7; i++) makeflips(move, player, board,
ALLDIRECTIONS[i]);
}



/* anylegalmove returns 1 if a player has at least one legal
   move and 0 otherwise. It is called by nexttoplay (below)
*/
```

```
int anylegalmove (int player, int * board) {
  int move;
  move = 11;
  while (move <= 88 && !legalp(move, player, board)) move++;
  if (move <= 88) return 1; else return 0;
}


/* choose the player with the next move. Typically this
will be the player other than previousplayer, unless
the former has no legal move available, in which case
previousplayer remains the current player. If no players
have a legalmove available, then nexttoplay returns 0.
*/

int nexttoplay (int * board, int previousplayer, int printflag) {
  int opp;
  opp = opponent(previousplayer);
  if (anylegalmove(opp, board)) return opp;
  if (anylegalmove(previousplayer, board)) {
    if (printflag) printf("%c has no moves and must pass.\n",
nameof(opp));
    return previousplayer;
  }
  return 0;
}


/* if a machine player, then legalmoves will be called to store
   all the current legal moves in the moves array, which is then
returned.
   moves[0] gives the number of legal moves of player given board.
   The legal moves (i.e., integers representing board locations)
   are stored in moves[1] through moves[moves[0]].
*/

int * legalmoves (int player, int * board) {
  int move, i, * moves;
  moves = (int *)malloc(65 * sizeof(int));
  moves[0] = 0;
  i = 0;
  for (move=11; move<=88; move++)
    if (legalp(move, player, board)) {
      i++;
      moves[i]=move;
    }
  moves[0]=i;
  return moves;
}
```

```
/*************************************************************
********************* Strategies for playing *******************
/*************************************************************/


/* if a human player, then get the next move from standard input */

int human (int player, int * board) {
  int move;
  printf("%c to move:", nameof(player)); scanf("%d", &move);
  return move;
}


/* a random machine strategy chooses randomly among the
   legal available moves.
*/

int randomstrategy(int player, int * board) {
  int r, * moves;
  moves = legalmoves(player, board);
  r = moves[(rand() % moves[0]) + 1];
  free(moves);
  return(r);
}

/*
int randomstrategy(int player, int * board) {
  static int i=0;
  int r, * moves;
  moves = legalmoves(player, board);
  r = moves[(i % moves[0]) + 1];
  i++;
  free(moves);
  return(r);
}
*/


/* diffeval and weighteddiffeval are alternate utility functions
   for evaluation the quality (from the perspective of player)
   of terminal boards in a minmax search.
*/

int diffeval (int player, int * board) { /* utility is measured */
  int i, ocnt, pcnt, opp;               /* by the difference in */
  pcnt=0; ocnt = 0;                     /* number of pieces */
  opp = opponent(player);
  for (i=1; i<=88; i++) {
```

```
    if (board[i]==player) pcnt++;
    if (board[i]==opp) ocnt++;
  }
  return (pcnt-ocnt);
}


/* some machine strategies will regard some squares as more important
   than others. The goodness of a square/move is given by a weight.
Every
   perimeter square will have importance 0. Corner squares are the
   "best" squares to obtain. Why? Negative weights indicate that a
   square should be avoided. Why are weights of certain squares
adjacent to
   corners and certain edges negative? The weights array gives the
   importance of each square/move.
*/

int weighteddiffeval (int player, int * board) {
  int i, ocnt, pcnt, opp;
  const int weights[100]={0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
                          0,120,-20, 20,  5,  5, 20,-20,120,  0,
                          0,-20,-40, -5, -5, -5, -5,-40,-20,  0,
                          0, 20, -5, 15,  3,  3, 15, -5, 20,  0,
                          0,  5, -5,  3,  3,  3,  3, -5,  5,  0,
                          0,  5, -5,  3,  3,  3,  3, -5,  5,  0,
                          0, 20, -5, 15,  3,  3, 15, -5, 20,  0,
                          0,-20,-40, -5, -5, -5, -5,-40,-20,  0,
                          0,120,-20, 20,  5,  5, 20,-20,120,  0,
                          0,  0,  0,  0,  0,  0,  0,  0,  0,  0};
  pcnt=0; ocnt=0;
  opp = opponent(player);
  for (i=1; i<=88; i++) {
     if (board[i]==player) pcnt=pcnt+weights[i];
     if (board[i]==opp) ocnt=ocnt+weights[i];
  }
  return (pcnt - ocnt);
}



/* minmax is called to do a "ply" move lookahead, using
evalfn (i.e., the utility function) to evaluate (terminal) boards at
the end of lookahead. Minmax starts by finding and simulating each
legal
move by player. The move that leads to the best (maximum backed-up
score)
resulting board, is the move (i.e., an integer representing a
board location) that is returned by minmax.
The score of each board (resulting from each move)
```

```
is determined by the function diffeval if no player can
move from the resulting board (i.e., game over), by function
maxchoice if only player can move from the resulting board,
or by function minchoice if the opponent can move from the
resulting board.

Importantly, minmax assumes that ply >= 1.

You are to modify minmax so that it exploits alphabeta pruning,
and so that it randomly selects amongst the best moves for player.

*/


int minmax (int player, int * board, int ply, int (* evalfn) (int, int
*)) {
  int i, max, ntm, newscore, bestmove, * moves, * newboard;
  int maxchoice (int, int *, int, int (*) (int, int *));
  int minchoice (int, int *, int, int (*) (int, int *));
  moves = legalmoves(player, board); /* get all legal moves for player
*/
  max = LOSS - 1;  /* any legal move will exceed this score */
  for (i=1; i <= moves[0]; i++) {
    newboard = copyboard(board); BOARDS = BOARDS + 1;
    makemove(moves[i], player, newboard);
    ntm = nexttoplay(newboard, player, 0);
    if (ntm == 0) {  /* game over, so determine winner */
         newscore = diffeval(player, newboard);
         if (newscore > 0) newscore = WIN; /* a win for player */
         if (newscore < 0) newscore = LOSS; /* a win for opp */
    }
    if (ntm == player)   /* opponent cannot move */
       newscore = maxchoice(player, newboard, ply-1, evalfn);
    if (ntm == opponent(player))
       newscore = minchoice(player, newboard, ply-1, evalfn);
    if (newscore > max) {
        max = newscore;
        bestmove = moves[i];  /* a better move found */
    }
    free(newboard);
  }
  free(moves);
  return(bestmove);
}



/* If ply = 0, then maxchoice should return diffeval(player, board),
else the legal moves that can be made by player from board should
be simulated. maxchoice should return the MAXIMUM board score
```

```
from among the possibilities. The backed-up score of each
board (resulting from each player move) is determined
by function maxchoice if only player can move from the resulting
board,
by function minchoice if the opponent can move from the resulting
board,
is WIN if a win for player, a LOSS if a win for opponent, and
a 0 if a draw.

If two or more boards tie for the maximum backed score, then return
the move that appears first (lowest location) in the moves
array leading to a maximum-score board.

*/

int maxchoice (int player, int * board, int ply,
               int (* evalfn) (int, int *)) {
  int i, max, ntm, newscore, * moves, * newboard;
  int minchoice (int, int *, int, int (*) (int, int *));
  if (ply == 0) return((* evalfn) (player, board));
  moves = legalmoves(player, board);
  max = LOSS - 1;
  for (i=1; i <= moves[0]; i++) {
    newboard = copyboard(board); BOARDS = BOARDS + 1;
    makemove(moves[i], player, newboard);
    ntm = nexttoplay(newboard, player, 0);
    if (ntm == 0) {
         newscore = diffeval(player, newboard);
         if (newscore > 0) newscore = WIN;
         if (newscore < 0) newscore = LOSS;
    }
    if (ntm == player)
       newscore = maxchoice(player, newboard, ply-1, evalfn);
    if (ntm == opponent(player))
       newscore = minchoice(player, newboard, ply-1, evalfn);
    if (newscore > max) max = newscore;
    free(newboard);
  }
  free(moves);
  return(max);
}


/* If ply = 0, then minchoice should return the diffeval(player,
board),
else the legal moves that can be made by player's opponent from board
should
be simulated. minchoice should return the MINIMUM backed up board
score
from among the possibilities. The backed up score of each board
```

```
(resulting from each opponent move) is determined by function
maxchoice
if player can move from the resulting board,
by function minchoice if only the opponent can move
from the resulting board, is WIN if a win for player,
a LOSS if a win for opponent, and a 0 if a draw.

If two or more BOARDS tie for the minimum score, then return
the move that appears first (lowest location) in the moves
array leading to a minimum-score board.

Advanced: DO NOT worry about this,
but note that minchoice and maxchoice could be combined
easily into a single function, finding the board
with minimum score, s, is equivalent to finding the board
with maximum -1 * s. One would have to add an additional
parameter to decide whether to multiply by a -1 factor or
not in computing a board score in this combined function.
With a bit more work, one could combine all three
functions, minmax, maxchoice, and minchoice, into a single
function.

*/

int minchoice (int player, int * board, int ply,
               int (* evalfn) (int, int *)) {
  int i, min, ntm, newscore, * moves, * newboard;
  if (ply == 0) return((* evalfn) (player, board));
  moves = legalmoves(opponent(player), board);
  min = WIN+1;
  for (i=1; i <= moves[0]; i++) {
    newboard = copyboard(board); BOARDS = BOARDS + 1;
    makemove(moves[i], opponent(player), newboard);
    ntm = nexttoplay(newboard, opponent(player), 0);
    if (ntm == 0) {
         newscore = diffeval(player, newboard);
         if (newscore > 0) newscore = WIN;
         if (newscore < 0) newscore = LOSS;
    }
    if (ntm == player)
       newscore = maxchoice(player, newboard, ply-1, evalfn);
    if (ntm == opponent(player))
       newscore = minchoice(player, newboard, ply-1, evalfn);
    if (newscore < min) min = newscore;
    free(newboard);
  }
  free(moves);
  return(min);
}
```

```
/* the following strategies use minmax search, in contrast to
randomstrategy
*/

int maxdiffstrategy1(int player, int * board) { /* 1 ply lookahead */
  return(minmax(player, board, 1, diffeval));   /* diffeval as utility
fn */
}

int maxdiffstrategy3(int player, int * board) { /* 3 ply lookahead */
  return(minmax(player, board, 3, diffeval));
}

int maxdiffstrategy5(int player, int * board) { /* 5 ply lookahead */
  return(minmax(player, board, 5, diffeval));
}


/* use weigteddiffstrategy as utility function */

int maxweighteddiffstrategy1(int player, int * board) {
   return(minmax(player, board, 1, weighteddiffeval));
}

int maxweighteddiffstrategy3(int player, int * board) {
   return(minmax(player, board, 3, weighteddiffeval));
}

int maxweighteddiffstrategy5(int player, int * board) {
   return(minmax(player, board, 5, weighteddiffeval));
}

/*******************************************************************
*************************** Coordinating matches ******************
******************************************************************/


/* get the next move for player using strategy */

void getmove (int (* strategy) (int, int *), int player, int * board,
              int printflag) {
  int move;
  if (printflag) printboard(board);
  move = (* strategy)(player, board);
  if (legalp(move, player, board)) {
     if (printflag) printf("%c moves to %d\n", nameof(player), move);
     makemove(move, player, board);
  }
  else {
```

```
      printf("Illegal move %d\n", move);
      getmove(strategy, player, board, printflag);
   }
}


/* the Othello function coordinates a game between two players,
represented by the strategy of each player. The function can
coordinate play between two humans, two machine players, or one of
each.
*/

void othello (int (* blstrategy) (int, int *),
              int (* whstrategy) (int, int *), int printflag) {
  int * board;
  int player;
  board = initialboard();
  player = BLACK;
  do {
    if (player == BLACK) getmove(blstrategy, BLACK, board, printflag);
    else getmove(whstrategy, WHITE, board, printflag);
    player = nexttoplay(board, player, printflag);
  }
  while (player != 0);
  if (printflag) {
     printf("The game is over. Final result:\n");
     printboard(board);
  }
}


/* The following two functions are used for computer player
tournaments. Since any pair of non-random strategies will
yield exactly the same result from game to game, randomboard
is used to introduce some uncertainty/randomness to a game
so that different games between the same strategies will
yield different results. In particular, randomboard starts
with an initialboard, then uses the randomstrategy to make
the first 10 moves (5 for each player).
*/

int * randomboard (void) {
  int player, oldplayer, i, * board;
  board = initialboard();
  player = BLACK;
  i=1;
  do {
    if (player == BLACK) getmove(randomstrategy, BLACK, board, 0);
    else getmove(randomstrategy, WHITE, board, 0);
    oldplayer = player;
```

```
    player = nexttoplay(board, player, 0);
    if (oldplayer == player) {
       free(board);
       return(randomboard());
    }
    i++;
  }
  while (player != 0 && i<=8);
  if (player==0) {
     free(board);
     return(randomboard());
  }
  else return(board);
}

/* Roundrobin pits each known machine strategy found in STRATEGIES
against every other strategy for 10 games, 5 as BLACK and 5 as WHITE.
each game begins with a randomboard. See end of this file for
example outputs of roundrobin.
*/

void roundrobin (void) {
  int i, j, k, cntdiff, player, iwins, jwins, * game1board, *
game2board;
  long int iboards, jboards;

  i=1;
  while (STRATEGIES[i+1][0] != NULL) { /* pit one strategy */
    j = i + 1;
    while (STRATEGIES[j][0] != NULL) { /* against another */
      iwins = 0; jwins = 0; iboards = 0; jboards = 0;
      for (k=1; k<=5; k++) { /* play a strategy as BLACK then as WHITE
*/
        game1board = randomboard();   /* do this 5 times */
        game2board = copyboard(game1board);
        player = BLACK;
        do {
          if (player == BLACK) {
             BOARDS = 0;
             getmove((fpc)STRATEGIES[i][2], BLACK, game1board, 0);
             iboards = iboards + BOARDS;
           }
          else {
             BOARDS = 0;
             getmove((fpc)STRATEGIES[j][2], WHITE, game1board, 0);
             jboards = jboards + BOARDS;
           }
          player = nexttoplay(game1board, player, 0);
        }
        while (player != 0);
```

```
        cntdiff = diffeval(BLACK, game1board);  /* determine winner */
        if (cntdiff>0) iwins++;
        if (cntdiff<0) jwins++;
        free(game1board);

        player = BLACK;
        do {
          if (player == BLACK) {
             BOARDS = 0;
             getmove((fpc)STRATEGIES[j][2], BLACK, game2board, 0);
             jboards = jboards + BOARDS;
           }
          else {
             BOARDS = 0;
             getmove((fpc)STRATEGIES[i][2], WHITE, game2board, 0);
             iboards = iboards + BOARDS;
          }
          player = nexttoplay(game2board, player, 0);
        }
        while (player != 0);
        cntdiff = diffeval(WHITE, game2board);  /* determine winner */
        if (cntdiff>0) iwins++;
        if (cntdiff<0) jwins++;
        free(game2board);
      }
      printf("%s wins=%d boards=%ld || %s wins=%d boards=%ld\n",
              STRATEGIES[i][0], iwins, iboards,
              STRATEGIES[j][0], jwins, jboards);

      j++;
    }
    i++;
  }

}


/* playgame interfaces with user to setup a game. Playgame displays
the player strategy options to a a user, and calls othello to play a
game.
*/

void playgame (void) {
 int i, p1, p2, printflag;
 int (* strfn1)(int, int *);  int (* strfn2)(int, int *);
 char * strnme;

  /* get strategy for player 1 (black) from user */
  i=0;
```

```
  printf("Player 1: ");
  while (STRATEGIES[i][0] != NULL) {
     strnme=STRATEGIES[i][1]; printf("%d (%s)\n", i, strnme);
     printf("          ");
     i++;
  }
  printf(": ");
  scanf("%d", &p1);

 /* get strategy for player 2 (white) from user */
  i=0;
  printf("Player 2: ");
  while (STRATEGIES[i][0] != NULL) {
     strnme=STRATEGIES[i][1]; printf("%d (%s)\n", i, strnme);
     printf("          ");
     i++;
  }
  printf(": ");
  scanf("%d", &p2);

  strfn1 = STRATEGIES[p1][2]; strfn2 = STRATEGIES[p2][2];
  if (strfn1 == human || strfn2 == human) printflag = 1;
  else {
    printf("       \n");
    printf("Neither player is human. Do you want to print each board
(1) or not (0): ");
    scanf("%d", &printflag);
  }

  othello(strfn1, strfn2, printflag);
}



/
**********************************************************************
**
******************************** MAIN
***********************************
**********************************************************************
**/

int main (void) {
  do {
      playgame();
      fflush(stdin);
      printf("Do you want to play another game (y or n)? ");
    } while (getchar() == 'y');

/*  roundrobin(); /* You will uncomment and run roundrobin to test
```

```
                     your alphabeta pruning modifications of minmax,
                     maxchoice and minchoice. */
}


/* Become familiar with Othello by running the code as written
and selecting the following single game options

1) maxdiffstrategy, 1-move lookahead (3) for BLACK against
   maxweighteddiffstrategy, 1-move lookahead (6) for WHITE

    1 2 3 4 5 6 7 8 [b=13 w=0]
10  . . . . . . . .
20  . . . . . . . .
30  . b b b b b . .
40  . . . b b b . .
50  . . . b b b . .
60  . . . . . b . .
70  . . . . . . b .
80  . . . . . . . .


2) maxdiffstrategy, 1-move lookahead (3) for BLACK against
   maxweighteddiffstrategy, 5-move lookahead (8) for WHITE

    1 2 3 4 5 6 7 8 [b=27 w=37]
10  w w w w w w w w
20  b b w w w w w w
30  b b b w w b b b
40  b b b w b w b b
50  b b w b w w w w
60  b w b b w w b w
70  b w w w b b b b
80  b w w w w w w w


What is interesting about this last game is that BLACK leads
by a large number of squares until near the very end of the game.
Remember that in this latter game, white is doing 5 ply lookahead
and is generating something on the order of 2.5 million boards
per game (look at the diff/1 vs. wdiff/5 line below under
roundrobin results). Thus, expect a delay (a tolerable delay)
each time that white moves in this latter game.

If you run roundrobin with the current code (minmax with no pruning),
then be prepared to wait quite a while. The following will eventually
materialize (which I have indented for a bit better readability):

random wins=1 boards=0        || diff/1 wins=9 boards=1824
random wins=2 boards=0        || diff/3 wins=8 boards=174938
```

```
random wins=0 boards=0        || diff/5 wins=10 boards=14970716
random wins=3 boards=0        || wdiff/1 wins=7 boards=2455
random wins=1 boards=0        || wdiff/3 wins=9 boards=293989
random wins=0 boards=0        || wdiff/5 wins=10 boards=33035881
diff/1 wins=0 boards=1799     || diff/3 wins=10 boards=155657
diff/1 wins=0 boards=1763     || diff/5 wins=10 boards=16361232
diff/1 wins=4 boards=2134     || wdiff/1 wins=6 boards=2397
diff/1 wins=2 boards=2038     || wdiff/3 wins=7 boards=268585
diff/1 wins=0 boards=1736     || wdiff/5 wins=10 boards=24682590
diff/3 wins=3 boards=200450   || diff/5 wins=6 boards=19828063
diff/3 wins=3 boards=202781   || wdiff/1 wins=7 boards=2301
diff/3 wins=1 boards=208165   || wdiff/3 wins=9 boards=242318
diff/3 wins=1 boards=174285   || wdiff/5 wins=9 boards=26257282
diff/5 wins=2 boards=18595500 || wdiff/1 wins=8 boards=2383
diff/5 wins=4 boards=19102429 || wdiff/3 wins=5 boards=202807
diff/5 wins=0 boards=16349918 || wdiff/5 wins=10 boards=22520544
wdiff/1 wins=0 boards=2311    || wdiff/3 wins=10 boards=292058
wdiff/1 wins=1 boards=1944    || wdiff/5 wins=9 boards=29422051
wdiff/3 wins=1 boards=234330  || wdiff/5 wins=8 boards=28452315
```

Each line above shows the results of 10 matches against two strategies.
For example, the last line shows the results of the strategy using 3-ply lookahead with weighteddiffeval (wdiff/3) as the utility function
against 5-ply lookahead with the same utility function (wdiff/5).
Over 10 games, wdiff/5 wins 8, wdiff/3 wins 1, and there is one tie.
Over 10 games, wdiff/5 (in this last case) generates 28,452,315 boards
(or an average of 2,845,232 boards generated per game)!

If and when you were to implement an incomplete (local) alphabeta pruning running, the results will be
something like:

```
random wins=1 boards=0      || diff1 wins=9 boards=1824
random wins=2 boards=0      || diff3 wins=8 boards=44887
random wins=0 boards=0      || diff5 wins=10 boards=944422
random wins=3 boards=0      || wdiff1 wins=7 boards=2455
random wins=1 boards=0      || wdiff3 wins=9 boards=97418
random wins=0 boards=0      || wdiff5 wins=10 boards=2839502
diff1 wins=0 boards=1799    || diff3 wins=10 boards=44741
diff1 wins=0 boards=1763    || diff5 wins=10 boards=1038063
diff1 wins=4 boards=2134    || wdiff1 wins=6 boards=2397
diff1 wins=2 boards=2038    || wdiff3 wins=7 boards=90605
diff1 wins=0 boards=1736    || wdiff5 wins=10 boards=2643711
diff3 wins=3 boards=59175   || diff5 wins=6 boards=1160722
diff3 wins=3 boards=51624   || wdiff1 wins=7 boards=2301
diff3 wins=1 boards=58869   || wdiff3 wins=9 boards=84970
diff3 wins=1 boards=48521   || wdiff5 wins=9 boards=2947041
```

```
diff5 wins=2 boards=1142881 || wdiff1 wins=8 boards=2383
diff5 wins=4 boards=1063738 || wdiff3 wins=5 boards=76715
diff5 wins=0 boards=1124236 || wdiff5 wins=10 boards=2683518
wdiff1 wins=0 boards=2311   || wdiff3 wins=10 boards=89526
wdiff1 wins=1 boards=1944   || wdiff5 wins=9 boards=2635355
wdiff3 wins=1 boards=80708  || wdiff5 wins=8 boards=2844060
```

Notice that the win/loss results do not change at all (minmax with alphabeta pruning chooses exactly the same moves as minmax without alphabeta pruning), but the number of generated boards is vastly reduced. For example, when wdiff5 is pitted against wdiff3, wdiff5 with pruning generates 2,844,060 boards over 10 games or an average of 284,406 boards per game, as opposed to about 10 times this without pruning.

Results immediately above are from incomplete alpha-beta pruning that we discussed in class. If you are careful to select the full alpha beta procedure (also discussed in class) that is in the text, here is a list of results that you might expect. This version communicates the best and worst move scores (alpha and beta) across multiple levels of the minmax search. In contrast, the incomplete version of alphabeta only communicated the alpha and beta values (best or worst scores discovered so far) to the immediate children of a node. Note that the results below indicate a further decrease in number of boards expanded (e.g., an average of 201,482 boards per game, down from 284,406, for wdiff5 when pitted against wdiff3)

Full alpha beta (no randomization of moves)

```
random wins=1 boards=0       || diff1 wins=9 boards=1824
random wins=2 boards=0       || diff3 wins=8 boards=44806
random wins=0 boards=0       || diff5 wins=10 boards=693398
random wins=3 boards=0       || wdiff1 wins=7 boards=2455
random wins=1 boards=0       || wdiff3 wins=9 boards=97374
random wins=0 boards=0       || wdiff5 wins=10 boards=1933766
diff1 wins=0 boards=1799     || diff3 wins=10 boards=44674
diff1 wins=0 boards=1763     || diff5 wins=10 boards=740534
diff1 wins=4 boards=2134     || wdiff1 wins=6 boards=2397
diff1 wins=2 boards=2038     || wdiff3 wins=7 boards=90545
diff1 wins=0 boards=1736     || wdiff5 wins=10 boards=1871014
diff3 wins=3 boards=59128    || diff5 wins=6 boards=833302
diff3 wins=3 boards=51592    || wdiff1 wins=7 boards=2301
diff3 wins=1 boards=58855    || wdiff3 wins=9 boards=84937
diff3 wins=1 boards=48473    || wdiff5 wins=9 boards=2072212
diff5 wins=2 boards=815805   || wdiff1 wins=8 boards=2383
diff5 wins=4 boards=751811   || wdiff3 wins=5 boards=76700
```

```
diff5 wins=0 boards=811652   || wdiff5 wins=10 boards=1910016
wdiff1 wins=0 boards=2311    || wdiff3 wins=10 boards=89485
wdiff1 wins=1 boards=1944    || wdiff5 wins=9 boards=1823471
wdiff3 wins=1 boards=80707   || wdiff5 wins=8 boards=2014816


When you randomly select amongst the best moves in the minmax
function,
you can expect different results than the above. In general, this
may represent a modest degradation in performance (in terms of
Win/Loss for what we regard as the better player) as well
as number of boards expanded. This degradation may be an artifact
of the randomization process and the fact that we are only playing
10 games per pairs of players in the roundrobin tournament, but there
may also be a more systematic bias at work here, which we will
discuss.

Full alpha beta (with random selection among the best moves)

random wins=6 boards=0       || diff1 wins=3 boards=1769
random wins=3 boards=0       || diff3 wins=7 boards=49151
random wins=0 boards=0       || diff5 wins=10 boards=732213
random wins=3 boards=0       || wdiff1 wins=7 boards=2444
random wins=1 boards=0       || wdiff3 wins=9 boards=96911
random wins=0 boards=0       || wdiff5 wins=10 boards=1735804
diff1 wins=2 boards=2224     || diff3 wins=8 boards=63567
diff1 wins=3 boards=2091     || diff5 wins=6 boards=731774
diff1 wins=2 boards=2169     || wdiff1 wins=8 boards=2624
diff1 wins=0 boards=1920     || wdiff3 wins=10 boards=85564
diff1 wins=1 boards=1742     || wdiff5 wins=9 boards=2108210
diff3 wins=4 boards=74811    || diff5 wins=6 boards=1094075
diff3 wins=4 boards=56395    || wdiff1 wins=6 boards=2503
diff3 wins=1 boards=63171    || wdiff3 wins=8 boards=97417
diff3 wins=0 boards=53204    || wdiff5 wins=10 boards=2039680
diff5 wins=3 boards=1104430  || wdiff1 wins=7 boards=2566
diff5 wins=4 boards=1382107  || wdiff3 wins=3 boards=100405
diff5 wins=1 boards=831579   || wdiff5 wins=9 boards=1734347
wdiff1 wins=2 boards=2401    || wdiff3 wins=7 boards=82616
wdiff1 wins=4 boards=2239    || wdiff5 wins=6 boards=2173090
wdiff3 wins=1 boards=80897   || wdiff5 wins=9 boards=2114608


*/
```