

Final Project Design: Othello

Bryce Lewis, Emily Williams, Jia Li

May 2, 2014

1 Description

Othello is a two-person game played with 64 tokens on an eight-by-eight grid. Every individual token has two sides, X and O, each of which represents a player. The goal of the game is to have more tokens of your symbol than your opponent's symbol on the grid by the end of the game.

To begin, the central four squares of the board will be filled with four tokens: X on the the upper left and bottom right corners, and O on the upper right and bottom left corners.

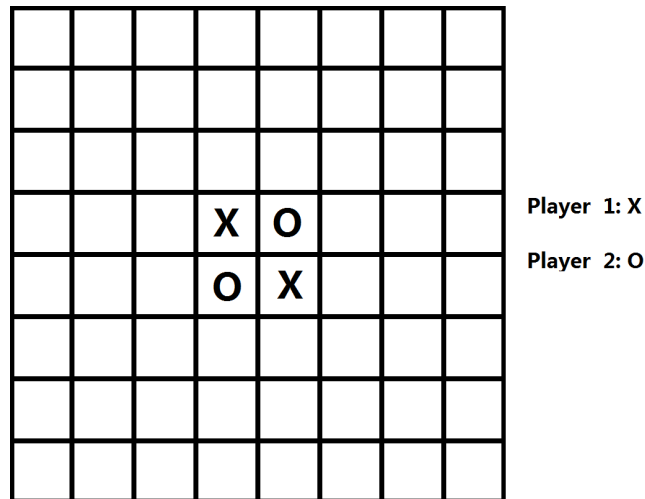


Figure 1: Game Board

Possible moves that a player may make will be marked on the board by an 'M' at the start of their turn. To make a move, a player must place a token

of their symbol on an M, placed adjacent to a token of the opponent's symbol (horizontally, vertically, or diagonally) in such a way that one or more of the opponent's tokens are sandwiched between the new token and a "base" token (any of the current player's tokens already on the board). There can be no empty spaces in the squares between the new and base tokens. All of the tokens between the new and base tokens are then flipped to the current player's symbol, and the turn is over. If a player cannot place a token in a way that meets these conditions, their turn is passed and the next player may move. If neither player can make a move, no matter how many tokens are on the board, the game is over. The winner is the one with the most tokens of their symbol on the board when the game ends.

2 Function Design

The game will be implemented using C according to the structure described in the dependency graph appended to the end of this document. This section gives the description about each file and its functions.

2.1 Variables and Constants

The game requires the following global variables to keep track of the state of the board, update scores for each player, and determine the computer's move in single-player mode.

```
#define N 8

//2D array (the representation of the game board)
char board[N][N];

//2D array (computer's move calculation board for choosing the
    best M)
int computeCount[N][N];

//player score
int p1score;
int p2score;

//row and col to specify the coordinates
```

```

int row;
int col;

//indicator of when the game should end
int pass;

//indicator of who is playing the game currently (their symbol)
int turn;

//indicator of whether bonus mode is on or not
int bonusmode;

//gives the coordinates of bonus_x and bonus_y
int bonus_x;
int bonus_y;

//enumerations of commonly used symbols
typedef enum {X = 1, O = 2, M = 3} token;
typedef enum {COUNTONLY, FLIPONLY} mode;
typedef enum {HUMAN=1, COMPUTER=2} playmode;
typedef enum {BONUSON=1, BONUSOFF=0} bonus;

```

2.2 main.c

The main.c file has several helper functions for the main function. These helper functions make the main function look much cleaner.

2.2.1 int main()

Main starts the game by calling the initializer. It then gets the mode from the user and uses function calls to run the players' turns.

```

int main(){
    //prep for the game
    srand(time(NULL));
    system("clear");

    //create the board
    printf("\n\nWelcome to Othello!\n\n");

```

```

        initialize();

        //get the mode from the user
        int whichmode= getMode();
        puts("in main");

        //and run the game!
        if(whichmode==1)chooseMode(HUMAN,COMPUTER);
        else if(whichmode==2) chooseMode(HUMAN,HUMAN);
        else if(whichmode==3) chooseMode(COMPUTER,COMPUTER);

        //error handling
        else{
            puts("GetMode functions return errors. Abort!");
            exit(0);
        }
        return 0;
}

```

2.2.2 Other functions in main.c

The following functions run the human and computer turns individually. This division helps to break up the main file, making it both more efficient and easier to read.

```

//executes all steps and functions calls for human move
//(the user decides where the token is placed)
void human(int turn);

//executes all steps and functions calls for computer move
//(the computer decides where the token is placed)
void computer(int turn);

//chooses mode based on user input and runs both players'
//turns until game is complete
void chooseMode(int mode1, int mode2);

```

2.3 init.c

This file initializes the 2-dimensional array in which all tokens on the board are stored.

```
//initializes 2d array (8x8) with four starting tokens
void initialize(void);
```

2.4 io.c

This file is in charge of all game-user interaction. These functions are called in main.c.

```
//prints the board and simultaneously keeps score
void print();

//gets user entry for column and row
//checks input and allows user to reenter if input illegal
void getEntry();

//returns a number from 1-3, indicating which mode is chosen
//among the three at the first prompt
int getMode();
```

2.5 makeMove.c

MakeMove is where most of the complex algorithms are found. This section controls the placement of tokens, houses the artificial intelligence algorithms, and determines where players are allowed (and not allowed) to move.

```
//random number generator with range [min,max]
//must call srand(time(0)) in main
int randGen( int min, int max);

//given t == X, return 0 (and vice versa)
int getOppositeSymbol(int t);
```

```

//looks at the board and stores the number of token each M can
    flip (per M)
//additionally updates variable count (number of M's on the board
    currently)
//and max (the max number of tokens that the "most capable" M can
    flip)
void computerMoveHelper(int t, int nt, int *max, int *count);

//determines where to move given the current player's symbol t
//using a greedy artificial intelligence algorithm
void computerMove(int t);

//crawls in all 8 directions
//depending on the mode, it will either flip or count flips
//returns number of flips in a certain direction
int flipIt(int t, int x, int y, int mode);

//helper function for printing
char getTokenName(int t);

//cleans up all the M's
void clear();

//moveExist helper that return 1 if the direction is invalid
int meCrawler(int t, int step, int m, int n, int *signal);

//returns 1 if there are moves available for the current player(X
    or 0)
//returns 0 otherwise
int moveExist (int t);

```

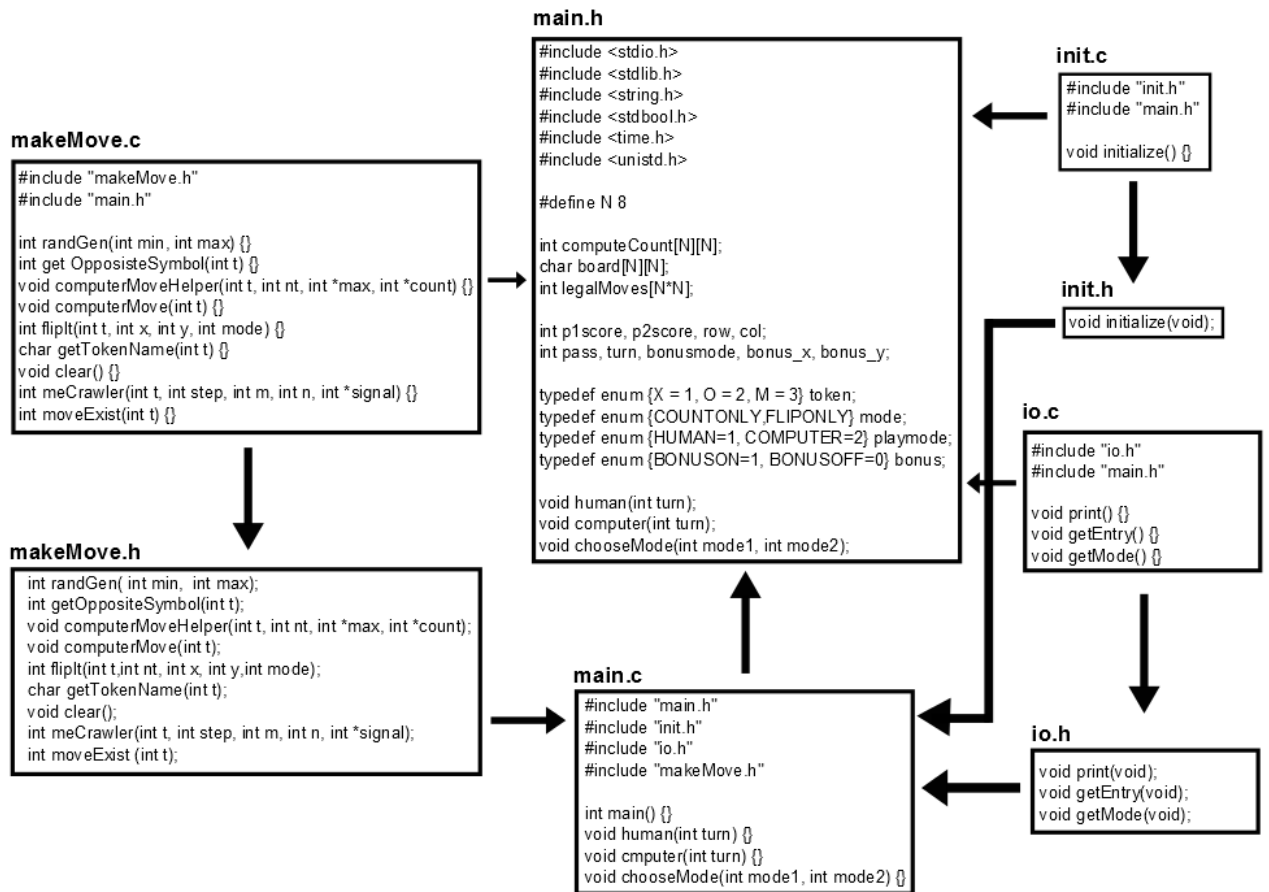


Figure 2: Dependency Graph