COMPSCI 170 - Numeric Artificial Intelligence
Programming Project
Playing the Game of Othello

Due date: March 21

1 Description

In this programming project you will design and implement an Othello playing program. If you are unfamiliar with the rules of Othello there is a short guide in this document. You can also easily find some java othello clients by searching the web or trying linux games like *lagno* or *reversi*, which are included in many linux distributions. Playing some games against these will help you improve your skills and intuitions for the game.

We make two major changes to the common version of Othello: first, we use a 10×10 board rather than the traditional 8×8 board; second, we will block off some squares into which moves will be disallowed. A maximum time limit for the entire game will be placed on your program. If your program runs out of time during the game, the game is an automatic win for the other player. There won't be enough time for your program to search the entire search tree for the best move, so the program you create should search for the move it estimates is best to play by using alpha-beta search with an evaluation function.

The preferable implementation language is *C*, but other languages can be used as well as long as you can establish socket communication with the game server that we will provide.

The project is designed to be quite open-ended and you are encouraged to have some fun exploring different methods of improving your player. We want to encourage some competition, but not make things so competitive that we take the fun out of it. You will get 90% credit for producing a working program and demonstrating the the things we ask you to demonstrate later in this document.

After everybody has turned in their programs, we will have a tournament. The remaining 10% of your grade will be determined by both your score in the tournament and your creativity in going beyond the bare minimum requirements.

Note that the easiest way to improve your player is to develop a clever heuristic evaluation function. This year we have made this task a bit more challenging by deciding that we will *not* tell you the exact structure of the board used for the competition in advance. The board will have some number of blocked areas (between 4 and 10), so your design of your heuristic function must be flexible enough to apply to different board configurations.

2 Othello

The Othello game-board consists of an 8×8 grid with an initial setup of two red discs and two white discs centered on the grid. In our version of the game the board will be set to 10×10, although you should follow good programming practices and design code that works for *any* size of the board. The objective of the game is to have the majority of discs on the board be of your color at the end of the game. Some terminology:

- A move consists of *outflanking* your opponent's disc(s), then *flipping* the outflanked disc(s) to your color.
- To *outflank* means to place a disc on the board, adjacent to one of your opponent's discs, so that your opponent's line (or lines) of disc(s) is bordered at each end by a disc of your color.
- A *line* is defined as one or more discs in a continuous straight line: a row, a column or a diagonal.

Now, the official rules:

- White always moves first.
- If on your turn you cannot outflank and flip at least one opposing disc, your move is forfeited and your opponent moves again. However, if a move is available to you, you may not forfeit your turn.
- A disc may outflank any number of discs in one or more lines in any number of directions at the same time – horizontally, vertically or diagonally.
- Disc(s) may only be outflanked as a direct result of a move and must fall in the direct line of the disc placed down.
- All discs outflanked in any one move must be flipped, even if it is to the player's advantage not to flip them all.
- When it is no longer possible for either player to move, the game is over. Discs are counted and the player with the majority of his or her color discs on the board is the winner.

3 Tasks to Do

Here is list of tasks that you are expected to accomplish for this assignment:

- Review this document and the associated code. The code is in `http://www.cs.duke.edu/education/courses/spring06/cps170/othello/`.
- Make sure you can access and use the software supplied to you. Go over the sample client program and make sure you understand it. Write the rest of your code as a part of the client.
- Implement minimax search. For this part you should use the evaluation function that simply looks at the difference between the number of your pieces and your opponent's pieces, i.e.,

  $$V(\text{state}) = (\#\text{my pieces}) - (\#\text{opponent's pieces})$$

  (The components for this are all but given to you already.) You will also have to consider what sort of data structures you will need in order to create a game tree. Remember that in Othello it may be possible that one of the sides has no valid move. Be sure that your algorithm can cope with this situation.
- Make the search more efficient by implementing alpha-beta pruning. Make sure that you can turn alpha-beta pruning on and off as you wish.
- Using the code fragments we have provided, demonstrate that alpha-beta pruning a) produces the same values as minimax search while b) producing expanding significantly fewer nodes.
- Construct an improved evaluation function and show that it at least beats the simple one described above.

- Try some improvements to your search control that go beyond basic alpha-beta pruning.
- Turn your Othello player in by the due date. Your program should be able to play any number of games without crashing.
- You should also include a *brief* writeup explaining any clever extensions you have devised.
- Your writeup and your code (with any special compilation instructions) should be emailed to the TA.

4 A Few Suggestions

Minimax Search and Alpha-Beta Pruning have been covered in the class and the textbook contains all the information and pseudocode you need for this assignment. Here are a few comments and suggestions:

- Each game has a time limit for each player. The current version of the server is set for a 2 minute limit. Your player must finish within this time. The time limit is, unfortunately, measured in real time by the server. (There's not a great workaround for this if we want to stick with the heterogeneous client/server architecture.) We will be conducting the tournament on an unloaded 3Ghz P4, which should be at least as fast as most machines around the department.
- The textbook mentions that when we perform alpha-beta pruning we would ideally like to expand the best successors first. In general, the order in which you expand nodes can have a tremendous impact the effectiveness of your search. Think about methods for detecting when one node is a more promising candidate for expansion than another.
- You might also think about Quiescence search, which is used to selectively extend the search tree. It doesn't have a direct analog in Othello, since pieces are exchanged on every move. However, a generalization of this concept *might* be useful.
- If you do some research, you will find a number of clever AI approaches to Othello players. You are fee to incorporate these ideas into your player, but you must explain the method you have used and give proper citations in your writeup.

5 Implementation Issues

We provide some of the software for you in C. That includes the infrastructure specific to Othello, the communication services, and the server that manages the games. A graphical user interface in Tcl/Tk is also available. The GUI doesn't handle timing correctly and assumes a single starting configuration. We hope to address these limitations shortly. The communication protocol to both interfaces is *exactly* the same, so if your program runs with the ASCII interface it will be able to run with the graphical one as well.

The Othello infrastructure provides the main data structures and mechanisms for the game. The code includes also some useful procedures (e.g., checking the validity of a move, counting node expansions, and printing out nodes visited) for you to use. Of course, you can always implement your own procedures.

You are not allowed to change the communication protocol between the server and the client, the code of the GUI, and the code of the ASCII server. This is to make sure that your program will be fully compatible with any changes we choose to make to these modules and with the other players as well. You should also avoid modifying (or circumventing) the code that prints out or counts node expansions.

5.1 Othello Infrastructure

The Othello infrastructure is supplied in the file `board.h`. Mainly it includes a data structure that represents a board configuration and some procedures that can be used to manipulate it (e.g., performing a move).

As a convention we refer to the first player as WHITE and to the second player as RED. In fact, these are constants and their values are set to 0 and 1 respectively.

5.1.1 The Board Data Structure

The board data structure is `BoardPosition`, which is defined as:

```
typedef struct {
    SquareState squares[BOARD_SIZE][BOARD_SIZE]; /* access squares[row][col] */
    int pieces[2]; /* 0=White, 1=Red */
} BoardPosition;
```

The field squares is a matrix of size `BOARD_SIZE` x `BOARD_SIZE` which represents the board. `BOARD_SIZE` is currently set to 10, but the part of the code that plays the game should work even if we change the size (using the simple evaluation function that counts pieces).

To access the square (row,col) you can simply use `board.squares[row][col]`. Note that according to the C conventions both `row` and `col` range between `0` and `BOARD_SIZE-1` where (0,0) is the upper left corner of the board. Each square can have one of four values:

- `0=WHITE` – Player 1 has a piece on this square.
- `1=RED` – Player 2 has a piece on this square.
- `2=EMPTY` – The square is empty.
- `3=ILLEGAL` – This square is not a legal position (e.g. the four corners).

The other field in the structure is the array `pieces`, which indicates the number of pieces each side has. This is an integer array of size 2. Note that according to our conventions `board.pieces[WHITE]` would give the number of pieces that the first player has on the board, and `board.pieces[RED]` the number of pieces that the second player has on the board.

5.1.2 Some Useful Procedures

The file `board.h` declares some useful procedures, that you may want to use. The procedures are documented in the file, so here we include only a brief description.

- `int NumMove (BoardPosition *board, Side side)`
  This procedure returns the number of legal moves for `side`. `Side` is a type that can take the values `WHITE` and `RED`.
- `bool IsLegalMove (BoardPosition *board, int row, int col,Side side)`
  This procedure checks the legality of a move. It returns `TRUE` iff (`row`,`col`) is a legal move for `side`.
- `BoardPosition *DoMove (BoardPosition *board, int row, int col, Side side, bool allocate_new`
  This procedure performs a move. It either changes the position on the given board or allocates a new board and performs the move there (this is determined by the variable `allocate_new_board`).

- `BoardPosition *AllocateNewBoard (void)`
  This procedures allocates a new board, initialized to the initial position. For those of you familiar with C++, note that this is equivalent to the C++ constructor.

- `void InitializeBoard (BoardPosition *board)`
  Change `board` to the initial position. It is assumed that `board` was previously allocated.

- `void CopyIntoBoard (BoardPosition *in_board, BoardPosition *out board)`
  This procedure copies the board position in `out_board` into `in_board`. `out_board` is not changed. It is assumed that `in_board` was previously allocated.

- `void PrettyPrint (BoardPosition *board)`
  Prints the board as an ASCII board. This is the printing procedure used by the ASCII server.

- `void PrettyPrintWide (BoardPosition *board)`
  Same as `PrettyPrint` but with a wider layout for better viewing.

- `ReadBoardFromFile (BoardPosition *board, char *file)`
  This procedure initializes a board position according to a position written in a file. The file format is the same format that the `PrettyPrint` (or `PrettyPrintWide`) procedure outputs, so you can cut and paste a position into a file and test it further. The procedure that reads the file only cares about the characters '`.`', '`R`' and '`W`' so it will work even if you do not include the boarders and numbers printed by `PrettyPrint`. Some samples of opening position files can be found in the subdirectory `BoardPositions/`.

5.2 Communication

In order to let two programs compete against each other we have implemented a simple server-client architecture. The server communicates with two clients, representing the two players, that may reside on the same or on a difference machine. At each time the server will send to one of the clients the board position and the amount of time left. The client needs to decide on its move, and after that it needs to send the message to the server. All the communication procedures are implemented for you. **Do not change these procedures.** There is also a simple main program for the client (`client.c`) which uses either random moves or asks the user for moves. You can use this code to create your player.

5.2.1 The ASCII Server

In order to run the ASCII server you need to issue the command:
`server <games> [port] [time] [init-position]`.

- `games` is the number of games you want the server to accommodate (these games are played sequentially). This is the only required argument.

- `port` is the port the server will use to listen for the clients (the default for this argument is 5555).

- `time` is the time allocated for each player per game in seconds. (Ignore the default value. In tournament play you will have only 3 minutes, i.e, 180.0 seconds.)

- `init-position` is a name for a file that contains an opening position you want the server to use. The default is for no file, in which case the server uses the standard opening position. You will find this option useful when you test your alpha-beta pruning on some test-cases. The file format is the one used by the function `ReadBoardFromFile`.

(*Note*: If you want to specify `time`, the third argument, you need to specify the second argument, `port`. Similarly, if you want to specify `init-position`, the fourth argument, you need to specify the second argument, `port`, and the third argument, `time`.)

The server will wait until two clients join it. Once this happens it will manage the game. At the end of the game it will either exit or coordinate another game between the same two clients (depending on `games`). The server is designed to close all the communication ports even if the clients crash. However, if you exit the server using `Ctrl-C` when one of the clients is logged in, the communication will not shut down itself. The next time you will try to run the server, you may get the message:
`bind: Address already in use`.

In general, **you should avoid stopping the server in this manner**. If you want to stop a run in the middle, you should kill one of the clients — the server will kill the other client, close the ports and exit. In case the problem occurs, you have two ways to recover. The first is to simply wait a few minutes until the operating system discovers the problem and closes the ports. The second is to use a different port. You can use any number between 5500 and 5599 as an alternative port.

5.2.2 The Client

The client is designed to connect to the server and take the part of one player. The server decides how many games the client will play in a session; the client must be able to play more than one game (this is very easy). In order to run the client issue the command:
`client <white|red> [server-port] [server-host] [nodelimit]`.

- `white|red` is the color of the player. This is the only required argument.
- `server-port` is the port the server is listening on (the default for this argument is `5555`).
- `server-host` is the name of host machine the server is running on, for example `tetra.cs.duke.edu` or `arch30` (the default for this argument is `localhost` - same machine as the client).
- `nodelimit` is the maximum number of node expansions allowed for each move (the default for this argument will be `50000`).

(*Note*: If you want to specify some argument you have to specify all the previous arguments first.)

The easiest way to understand how the client should look is to look at its code. This piece of code is based on the sample client (`client.c`) which is given to you.

```
int main (int argc, char *argv[])
{
  MsgFromServer msg;
  int row, col, socket;
  Side side;
  unsigned short port;
  char host[100];

  side = GetSide(argc, argv); /* Decide whether the player is WHITE or RED */
  port = GetPort(argc, argv);
  GetHostt(argc, argv, host);
  printf("server port:%d\n", port);

  StartSession (&socket, side, port);
  do {
    StartGame (socket, &msg);
    while (msg.status != GAME_ENDED && msg.status != ABORT) {
```

```
      if (msg.status == CAN_NOT_MOVE) {
        /* I have no move to play; send (-1,-1); get my opponent's move */
        SendAndGetMove (socket -1, -1, &msg); }
      else {
        GetMove (&msg.board, side, &row, &col); /* Decide on move */
        /* Send my move, get opponent's move and new board configuration*/
        SendAndGetMove (socket, row, col, &msg);
      }
    }
  } while (msg.status != ABORT);

  EndSession(socket);
  return 0;
}
```

The first procedure you need to call is `StartSession`. The parameter `side` is either `WHITE` or `RED`. The first parameter (`socket`) is a part of the communication protocol; it specifies the socket in which the communication will work. You do not need to worry about the details, but it may be useful for the training runs to actually open two sockets and play for both players. This way you can keep everything within one process.

We are now ready to start the game. We call the procedure `StartGame`. This procedure waits until the other player is ready and it is your turn to play. When this happens, the server will send the first board position message. This message is returned in the last parameter. The structure for this message is as follows:

```
typedef struct {
  Status status;
  double time_left; /* In seconds */
  int row, col; /* Opponent's last move; (-1,-1) for no move */
  BoardPosition board;
} MsgFromServer;
```

Let us go over the fields in this message one by one. `status` contains one of four values:

- `0. GIVE MOVE` — It is your turn to play and you must come up with a legal move.
- `1. CAN NOT MOVE` — It is your turn but you have no legal moves. The server sends you the position even if you do not have any possible moves to let you know about your opponent's move, in case you need the information.
- `2. GAME ENDED` — The game has ended.
- `3. ABORT` — The set of games has ended. Either all the games were played, or the other player has crashed.

`time_left` is the time left for your side in seconds (for the entire game). `row` and `col` describe the move of your opponent (in case your opponent did not move you will get the values `(-1,-1)`). Finally, `board` is the current board configuration.

After receiving the information from the server you must decide on your move. This is done in the procedure `GetMove` which is the actual part you need to implement (using the game tree, alpha-beta pruning, and so on). Right now the procedure simply chooses a random move.

In order to send your move to the server you need to use the procedure `SendAndGetMove`. This procedure takes as parameters the move which is sent as `row` and `col`. Recall that rows and columns are numbered from `0` to `BOARD_SIZE-1`. The structure for this message is as follows:

```
typedef struct {
    int row, col; /* The player's move; (-1,-1) for no move */
} MsgToServer;
```

In case you have no legal move you must send -1,-1 for `row` and `col`, otherwise it will count as a technical loss. Make sure that the moves you are sending to the server are valid. An invalid move will count as a technical loss. The `SendAndGetMove` procedure will wait until the server sends the move of the other player. Once the move is received the procedure ends, returning the opponent's move, the board position and the time left using the `MsgFromServer` structure.

When the game ends the server will start another one, until all the games are played. At this point the server will send an `ABORT` message. You need to call `EndSession` to make sure the communication ends in the appropriate way.

5.2.3 The ASCII Socket Interface

This section describes the format of the ASCII strings that are exchanged between the client and the server over the socket connection during a game. The C programmers need not worry about the details of this section, but if you plan to use any other implementation language you will have to confront to these conventions.

There are 3 types of messages between the client and the server.

- **The first message sent by the client to the server** to declare its color. This is sent only once at the very beginning and is never repeated. It consists of a single ASCII character, "`0`" or "`1`", which stands for white or red respectively.

- **The message from the client to the server** that contains the player's move. This is sent from the client to the server everytime it is the client's turn to move. It should basically consist of a string of 9 ASCII characters with the format "`RRR␣CCC␣␣`". The substring `RRR` holds the row-coordinate of the move and `CCC` holds the column-coordinate. Notice the space characters right after `RRR` and `CCC`. The last character can be a space or a "`\0`" termination character. For example, the move `(3,7)` becomes "`␣␣3␣␣␣7␣␣`" or "`003␣007␣␣`", while the move `(-1,-1)` becomes "`␣-1␣␣-1␣␣`".

- **The message from the server to the client** that contains the status of the game, the remaining time, the opponent's move, and the board configuration. This consist of a long ASCII character string of 228 characters total. The general format is the following:

  "`S␣TTTTT.TT␣RRR␣CCC␣WWW␣RRR␣B␣B␣B␣B␣B␣B␣...␣B␣B␣B␣B␣B␣␣`"

  The first character is the status of the game (one of `0`, `1`, `2`, or `3` as described in the previous section). `TTTTT.TT` is the time remaining represented in fixed-point format with five digits and two decimals. `RRR` and `CCC` encode the opponent's move with the same format as the player's move above. `WWW` and `RRR` are the numbers of white and red pieces on the board. Finally, the long sequence "`B B B B B B ... B B B B B`" encodes the board configuration. Each `B` is `0`, `1`, `2`, or `3` as described in the previous section. Squares are traversed row after row beginning with the upper left corner (similar to `PrettyPrint`). Notice the space characters right after each entity; the very last two characters are both spaces or a space and a "`\0`" termination character.

  As an example, assume that there are 250.66 seconds remaining, it's the red player's turn, the opponent's move was `(0,2)`, and that the state of the board is as follows:

```
  0 1 2 3 4 5 6 7 8 9
 +--------------------+
```

```
0| # . W R . . . . . # |0
1| . . . W . . . . . . |1
2| . W R W W . . . . . |2
3| . R . . R W . . . . |3
4| . . . . R W . . . . |4
5| . . . W R W . . . . |5
6| . . . . . . . . . . |6
7| . . . . . . . . . . |7
8| . . . . . . . . . . |8
9| # . . . . . . . . # |9
 +--------------------+
   0 1 2 3 4 5 6 7 8 9
```

The server will send the following string to the red client:

```
0␣00250.66␣000␣002␣009␣006␣
3␣2␣0␣1␣2␣2␣2␣2␣2␣3␣
2␣2␣2␣0␣2␣2␣2␣2␣2␣2␣
2␣0␣1␣0␣0␣2␣2␣2␣2␣2␣
2␣1␣2␣2␣1␣0␣2␣2␣2␣2␣
2␣2␣2␣2␣1␣0␣2␣2␣2␣2␣
2␣2␣2␣0␣1␣0␣2␣2␣2␣2␣
2␣2␣2␣2␣2␣2␣2␣2␣2␣2␣
2␣2␣2␣2␣2␣2␣2␣2␣2␣2␣
2␣2␣2␣2␣2␣2␣2␣2␣2␣2␣
3␣2␣2␣2␣2␣2␣2␣2␣2␣3␣␣
```

without the `<newline>`s of course.

Using these conventions it should be fairly easily to decode the server's messages and code the player's messages. For additional details, you may want to look at the C code that implements the communication functions (`protocol.c`, `client-comm.c`, and `server.c`).

Good luck!

5.2.4 The Graphical User Interface

In addition to the ASCII server, we have provided a graphical user interface that can help you test your program and run experiments. The GUI is written in Tcl/Tk and is contained in the `GUI` subdirectory; you must have Tcl/Tk installed on your system in order to use it. The interpreter for Tcl/Tk is called `wish`, so you would start the server by typing `wish othello.tcl`. On many Unix or linux systems, you can run the GUI by just typing `othello.tcl` at the command line. On Windows, you will have to run the `wish` program, with `othello.tcl` as an argument. You may also have to change the `SOURCE_DIR` directory in `othello.tcl`, before you run the server.

The GUI is fairly self-explanatory. The method of control for each player can be chosen from four possibilities: *Manual, Random, Greedy*, and *Program*. In *Manual* control mode, you click on the square on which you want to place a piece. The *Random* controller chooses its move randomly, while the *Greedy* controller chooses the move that turns over the most pieces, without looking ahead (ties are broken at random). You can use the random and greedy controllers to test how well your program plays; it should beat them both handily, and the margin of victory is an indication of how good your program is.

You can also control a player from a client program. While the GUI server is running, it listens to port 5555 for client programs. Client programs can connect to the GUI server the same way they connect to the ASCII server. If you run your client program with the appropriate `port` and `host` parameters, you will see a message on the log display of the GUI that the client has been registered as either Player 1 or Player 2 depending on the color parameter you chose. Now, you can click on the *Program* option to pass control to your program.

You may change the controller of a player at any time during the game. For example, you may want to enter a position manually, and then ask your client to play from that position. Or, you may want to play greedily for a few steps and then let your program continue, or play manually against your program, etc.

There are three modes of play. You may play a *single* game, play *continuously*, or set up the server to play a *fixed number* of games. You may find the repeat and continuous play modes useful for running experiments.

In the last two modes, a new game will start as soon as the previous one is finished (unless the specified number of games has been played in the repeat mode). The GUI will keep track of the total score of the two players. The "RESTART" button restarts the game that is currently being played, but it retains the total wins and loses for each player. The "RESET" button restarts the complete set of games and resets the total wins and loses.

Some important **warnings** in using the GUI:

- Do not click on the *Program* control mode unless you have connected your client to the server first.
- Do NOT kill (`CTRL-C`) clients connected to the GUI, even when you are in a different control mode. The GUI will crash. If you want to try some other client(s) you will have to quit the GUI first, then restart it, and connect the new client(s).
- If you really want to replace a connected client, then you can run the new client (with the appropriate parameters) without "touching" the old one (let it run until you quit). The new client will take over control of the game.

5.3 Measuring Time in C

Note: This section may be outdated. Please consult the README for the code.

Since each player has a time limit, you will probably need to measure time in your program. You may use any method you want, but here we will present a simple way to get a reliable measure of the

total CPU time used by your process. First you need to include the file `time.h`. Now you can use the procedure `clock()` that returns a value of type `clock_t` (which is actually `long int`). This is the number of clock ticks used by your process since the first time `clock()` was called, which we do in the initialization of the client. In Unix, one clock tick is a micro-second; a safer way of converting clock ticks to seconds is to divide by the constant `CLOCKS_PER_SEC`. You should make sure that this total number of seconds never exceeds the time limit! For more information on this function, you can read the Unix man page on `clock`.

If you want to perform a loop for no more than `t` seconds, then (assuming each iteration does not take more than `epsilon` seconds) you can write:

```
clock_t start;
start = clock();
while (((clock() - start)/CLOCKS_PER_SEC) / 1e9 < (t - epsilon)) {
  /* Do something in loop */
}
```

If `epsilon` is too big for effective measurements, you can put some test points inside the loop, based on the time left.