Dept. of Electrical Eng. & Computer Sci.
York University

Assignment 4

Total marks: 85.

Out: November 27
Due: December 11 at 18:00

Your report for this assignment should be the result of your own individual work. Take care to avoid plagiarism ("copying"). You may discuss the problems with other students, but do not take written notes during these discussions, and do not share your written solutions.

In this assignment, you are supplied with some starter code and have to design a program that plays a slightly modified version of Othello against a human user.

1 Modified Othello

Othello is a boardgame that is played with black and white stones placed on a chessboard. The players (black and white) take turns placing stones on the board. Occasionally, one of the players might have nowhere to place their colored stone. In this case their only valid move is to play a "pass" where they do not place any stones. The next player then takes their turn. A state where neither player can place a stone is a terminal state. In the original Othello, the winner of a terminal state is the player who has **more** stones on the board. However, *in the modified version that you will implement, the winner of a terminal state is the player who has **less** stones on the board*. A tie is declared in a terminal state if the number of white and black stones are equal.

The game begins with four stones placed in a square in the middle of the grid, two white stones and two black stones (Figure 1). Player 1 (black) makes the first move.

To understand rules of the game, it is useful to think of the 8 directions, N (north), NE (north-east), E (east), SE (south-east), S (south), SW (south-west), W (west) and NW (north-west). Viewing the top of the board as being North, these directions specify 8 lines moving away from any position on the board.

At each player's turn, the player may place a stone (of her color) on any square s of the board such that:

- Along at least one of the 8 direction from the square s we have a sequence of one or more opponent stones followed by the player's stone (with no empty squares in between). Note that we can start looking for legal places to put our stone by considering only those squares that are adjacent (in one of the directions) to an opponent's stone.

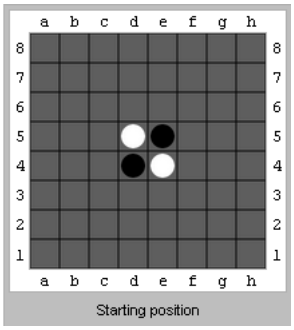

Figure 1: Initial State

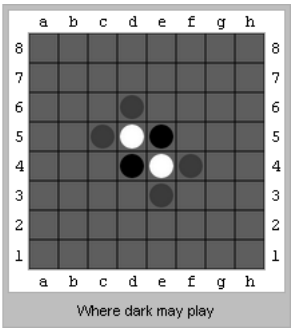

Figure 2: Possible moves of player 1 (black)

For example, from the initial state black can play in any of positions indicated by light-gray pieces in Figure 2.

After placing their stone, the board is updated as follows:

- Looking along all 8 directions, any sequence of opponent stones that are now bracketed by the player's newly placed stone and some previously placed player's stone (again with no empty squares in between) are now flipped in color to become player stones. Note that placing a single stone could cause many opponent stones to flip in different directions.

For example, if black decided to put a piece in the topmost location (6-d on the example figures), one white piece gets turned over, so that the board is transformed to the state as shown in Figure 3.

Now white (player 2) plays. All of white's possibilities at this time are show as gray stones in Figure 4.

If white moves to 4-c this will reverses one black piece as shown in Figure 5.

Othello is also commonly called **reversi**, and to get a better feel for the game you can play it at a number of on-line sites. For example, `http://www.web-games-online.com/reversi/`

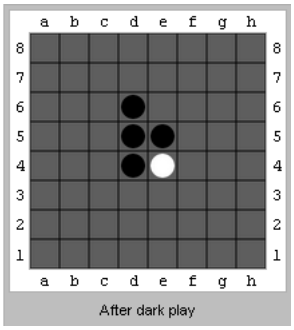

Figure 3: State after the move of player 1 (black).

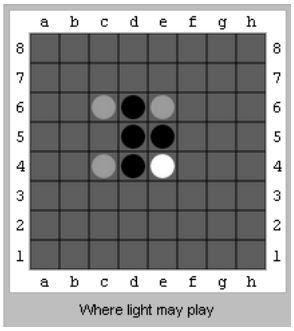

Figure 4: Possible moves of player 2 (white)

shows also the allowed moves when it is your turn. Remember however that in this assignment the winner is the player with the **fewer** stones on the board at the end.

2 The Assignment

You will be provided with the following PROLOG code (available for download from the course web page (follow the Assignment 4 link) :

- An implementation of an interactive depth-first minimax game tree search routine in the file `play.pl`. This file will not work on its own as it needs the definitions of several game-specific predicates. You will not change this file, but please read carefully the code there to see what predicates must be implemented and how they are used in the tree search. To invoke the interactive shell you need to type the query `play`. Assuming all required predicates have already been defined, the interactive game playing shell will prompt the human player to input moves. The player can enter a move (for this game a position pair like "[1,3]"), which will be

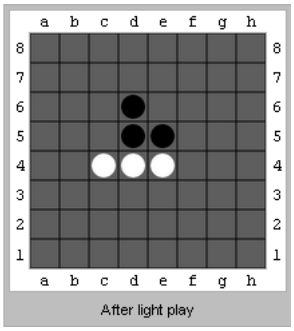

Figure 5: State after the move of player 2 (white).

then checked for validity (using a predicate you have to write). To play a "pass" move simply enter 'n'. Your `validmove` predicate should check the proposed move allowing a pass only if no other move is legal. Note that `read(Proposed)` is used to read the user's move—this will bind the variable `Proposed` to anything the user enters; you have to check that they have entered a valid move in the right syntax (i.e. a pair of numbers enclosed in brackets, or the character 'n'). When it is the computer's turn the engine will invoke a mini-max search for the best move. This search is done to a bounded depth, and you can set the depth bound. You should set a bound that yields reasonable performance.

- Some starter code for your Othello implementation is in the file `othello.pl`. You are given a prespecified state representation of the game as a list of lists. The 6x6 board is treated as a two dimensional array indexed by a pair of numbers $[X, Y]$ where these numbers are in the range 0–5. The file also contains a number of utility routines that allow you to set and get indexed squares on the board.

  You have to define various predicates to interface with the game tree search routine. This involves writing code to generate moves in the game, testing whether or not positions are terminal, evaluating the heuristic merit of positions in the game, etc. Full documentation on the predicates needed by the game tree search routine is provided at the beginning of the file `play.pl`. Please do not change `play.pl`, all your implementation must be done in `othello.pl`.

- There is an example implementation of an interactive tic-tac-toe game in the file `ttt.pl` where player 1 (MAX) is a human and player 2 (Min) is the computer. This sample game illustrates how to implement the routines required by the game tree search. To run the game, simply load file `ttt.pl` and enter the query `play`. You will be prompted to choose your first move (i.e. a number between 1 to 9 followed by a period). Then, computer will choose a move, and it's your turn again, and so on.

The assignment is broken into 3 main parts: (1) implementing a program to play Othello on a

6x6 board, (2) designing a heuristic function, and an optional part (3) which involves adding alpha-beta pruning to the game tree search routine. These 3 main parts are described in more detail below. Please note that your implementation should contain sufficient comments and not be contorted or overly complex. *Bad implementation style may cause deductions of up to 10%.*

2.1 Part I: Othello (Out of 75)

Implement the Othello game by adding your code to the supplied starter file `othello.pl`. In order to accomplish this you have to implement several predicates (feel free to define you own helper predicates for more complex predicates like `nextState`):

1. `initialize(InitialState,InitialPlyr)`
2. `winner(State,Plyr)`
3. `tie(State)`
4. `terminal(State)`
5. `moves(Plyr,State,MvList)`
6. `nextState(Plyr,Move,State,NewState,NextPlyr)`
7. `validmove(Plyr,State,Proposed)`
8. `h(State,Val)`
9. `lowerBound(B)`
10. `upperBound(B)`

Most of these predicates are based on the given state representation. Utilize the given utilities (e.g. get and set a value at a position) to determine the possible next moves: you must implement the predicate `moves(Plyr,State,MvList)` so that it returns a list `MvList` of all legal moves `Plyr` can make in the given state `State`. *The list of moves returned by this predicate should be sorted by position in order left to right, top to bottom.* E.g., if a move into positions $[1,1]$, $[0,0]$, $[2,7]$,$[0,2]$, $[1,5]$ are all possible, then you should return this list of moves in the order $[0,0]$, $[0,2]$, $[1,1]$, $[1,5]$, $[2,7]$.

Similarly, you must implement the predicate `nextState(Plyr,Move,State,NewState, NextPlyr)` that changes the current board `State` by playing `Move`. (Remember that applying a move can cause changes along several different directions.) You can use the given helper predicate `showState` to debug `nextState`. In your implementation account for the fact that the game can end with a tie and implement the `tie` and `winner` predicates. Note that the player with less stones at the end is the winner.

Note that it is necessary to accommodate null moves (since there are positions where one player cannot move) both in the user input and during the minimax search. A simple way of accomplishing this is to have `moves` return the list `[n]` in this case, and when `nextState` is passed an `n` move it can return an unchanged state as the new state, and the other player as the new player.

The predicate `h(State,Val)` requires that you design a heuristic function for the game. See Part II before doing so.

What to hand in:

1. Physical Copy A listing of your code (all relevant predicate listings). Be sure to document your predicate definitions well.

2. Physical Copy Download and print file `testboards.pdf`. Write down your name and student number on the top of the page. Download file `testboards.pl` and perform the tests requested in the `testboards.pdf` for the MiniMax algorithm. Fill out the top table

3. Electronic Copy Submit your `othello.pl` (using the `submit` command). Make sure you fill out the identification portion at the beginning of the file. Do not include any of the code in `play.pl`).

2.2 Part II: State Evaluation Function (Out of 10)

As mentioned above, `play.pl` requires implementing the heuristic function $h(S, V)$. If you decide NOT to do part II, to get credit for part I, you need to define a very simple heuristic instead: your $h(S, V)$ can return $V = 0$ for any non-terminal state S, and if S is a terminal state, h must return a positive value (say 100) for a win state, a negative value (say -100) for losing state, and 0 for a tie state. Clearly, this h provides no guidance in the depth-bounded search.

If you decide to do part II, you have to implement a smarter heuristic function as described below. In either case, note that the traces of your program required in part I, are based on the heuristic that you implement.

Below, we give you some ideas of good heuristics for the original Othello game. You have to adjust these for our modified version of the game where the player with the fewest stones on the board wins at the end.

2.2.1 Heuristic Functions for the Original Othello Game

Since at the end, the player with more stones wins the game, you might think that the evaluation function $h(s) = V1 - V2$ (where $V1$ and $V2$ are the number of stones for player 1 and 2, respectively), is ideal. This is only true if we expand all nodes in the search tree to reach the terminal nodes (which is practically impossible). For a non-terminal state, having more stones has no meaning (it could even be worse as seen in Figure 6) since many flips might occur in future moves.

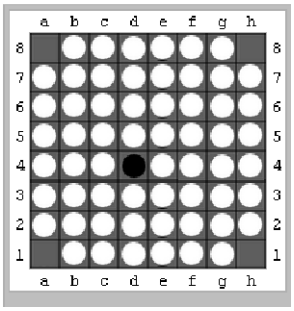

Figure 6: Maximum stones is not a good strategy: white has a lot more stones, while black has only 1. It is black's turn. So, she puts a stone in a8, white has to pass, then black puts a stone in h1, white passes, black plays h8, while passes, and finally black plays a1. Black wins: 40 black stone versus 24 white stones!

Instead, we focus more on the *stable stones* on the board, i.e. those stones that cannot be flipped anymore. Corner positions, once played, remain immune to flipping for the rest of the game (because there can never be an opposite color stone behind them to create a flip): thus a player can use a piece in a corner of the board to anchor groups of pieces (starting with the adjacent edges) permanently. So capturing a corner often proves an effective strategy when the opportunity arises. More generally, a piece is stable when, along all four axes (horizontal, vertical, and each diagonal), it is either on a boundary of the game board, or in a filled row, or next to a stable piece of the same color. The more stable stones you have (and the less stable stone your opponent has) the better. So, you may count the number of stable stones for both players and use them to obtain a good measure to evaluate states.

Another idea is *mobility*. An opponent playing with reasonable strategy will not so easily relinquish the corner or any other good moves for you to play. So to achieve these good moves, you must force your opponent to play moves which make available those good moves. The best way to achieve this involves reducing the number of moves available to your opponent. If you consistently restrict the number of legal moves your opponent can make, then sooner or later they will have to make an undesirable move. An ideal position involves having all your pieces in the center surrounded by your opponent's pieces. In such situations you can dictate what moves your opponent can make.

Note that the above ideas are with respect to the original Othello (taken from Wikipedia) and may require some adjustment to be applicable to our modified version of the game. It is not required, but if you want, you can go beyond these ideas. You can do your own research to find a wide range of other good heuristics (for example, a good place to start is `http://www.radagast.se/othello/Help/strategy.html`). We may give up to 5 bonus points for implementing an advanced heuristic function.

What to hand in:

1. Physical Copy, at most 1 page An English *description and justification* of the heuristic you

implemented. You are welcome to do a little bit research of your own to come up with a better evaluation function. *Make sure to cite all references you used (if any) for this question.*

2. Physical Copy A listing of the code implementing your heuristic function.

3. Electronic Copy Your implementation of the predicate h must be in the `othello.pl`. So, no extra submission is required. You just submit `othello.pl` as requested in Part I.

2.3 Optional Part III: Alpha-Beta Pruning (Worth Up To 5 Bonus Points)

The `play` predicate is based on depth-bounded, depth-first, minimax evaluation; but it does no pruning. This part asks you to replace the predicate `mmeval(Plyr,State,Value,Move,Depth, StatesSearched)` with a new predicate `abmmeval`, that evaluates states using minimax with alpha-beta pruning. The arguments to this predicate can be of your own choosing.

Place your alpha-beta implementation in a file called `abplay.pl`. This file should contain all of the functionality of `play.pl` except that `abmmeval` replaces `mmeval`. To do this, first copy `play.pl` into a new file called `abplay.pl`, and then make necessary changes there.

What to hand in:

1. Physical Copy A listing of your alpha-beta implementation. Be sure to document your code.

2. Physical Copy Repeat the tests you did in part I, but now based on your alpha-beta search engine. Complete the table at the bottom of file `testboards.pdf` accordingly. Also, on page 2 of `testboards.pdf`, write one or two paragraphs discussing your results on the two tables.

3. Electronic Copy Submit the file `abplay.pl` electronically.

To hand in your report for this assignment, put all your files in a directory `a4answers`, submit it electronically, *and* submit a printout in the 3401 drop box in LAS by the deadline. To submit electronically, use the following Prism lab command:

```
submit 3401 a4 a4answers
```

Your Prolog code should work correctly on Prism.