Computer Science 384 March 5, 2014
St. George Campus University of Toronto

Homework Assignment #2
Game Tree Search
Due: Mon March 24, 2014
Electronic submission due by 23:59.

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked by email or in person.*

Late Policy: 15% per day after the use of 2 grace days.

Total Marks: This assignment represents **13%** of the course grade. Part B is a fun **optional bonus question**. If you complete it sufficiently well, you will qualify for 1 extra grace day, which can be applied to a past, present, or future assignment.
Fun Competition: Finally we will be running a competition from the heuristics written for Part B. Your performance in the competition will **not** influence your mark on this assignment.

Handing in this Assignment
What to hand in electronically: You must submit your code electronically. Download `othello.pl` and `play.pl` from the course web page. For Part A of this assignment, **write your name and student number** in `othello.pl`, and fill in the definitions of predicates in their designated space (feel free to define helper predicates as necessary). DO NOT implement anything in `play.pl` and do not submit it. For optional Part B you must additionally download, modify, and submit the file `heuristic_c9jdoe.pl` where `c9jdoe` is to be replaced by **your cdf userid** in the file name and inside in the definition of predicates. We would also like a brief description of your heuristic in `hdescription_c9jdoe.txt`, again with `c9jdoe` replaced by your cdf userid in the file name. Part C is mandatory. Submission instructions will follow.

To submit your file electronically, use the CDF secure Web site: `https://www.cdf.utoronto.ca/students`
Warning: marks will be deducted for an incorrect submission.

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact predicate names and argument(s) (including the order of arguments) specified,
- use the exact file names specified in the questions,
- include all your code for Parts A & B in `othello.pl` and do not load any file of yours from within this file (other than the already given `play.pl`).
- not display anything but the predicate output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 2 Clarification page, linked from the CSC384 A2 web page. You are responsible for monitoring the A2 Clarification page.

Questions: Questions concerning the assignment should be directed by email to the Assignment 2 TA: Brent Mombourquette. He can be reached by email at `bgmomb -at - cs.toronto.edu`. Please place "384" and "A2" in your email subject header.

In Assignment 2 you are supplied with some starter code to build a program that plays the game Othello against a human opponent.

1 Othello

Othello is a 2-player board game that is played with distinct pieces that are typically black on one side and white on the other side, each side belonging to one player. Our version of the game is played on a 6x6 chess board, but the typical game is played on an 8x8 board. Players (black and white) take turns placing their pieces on the board. Placement is dictated by the rules of the game, and can result in the *flipping* of coloured pieces from white to black or black to white.

Objective: The player's goal is to have a majority of their coloured pieces showing at the end of the game.

Game Ending: Occasionally a player will have nowhere to place their coloured piece. In this case their only valid move is to play a "pass" and place no piece on the board. The next player then takes their turn. A state where neither player can place a piece is a terminal state. The winner of a terminal state is the player who has **more** of their colour of pieces exposed on the board. A tie is declared in a terminal state if the number of white and black pieces is equal.

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

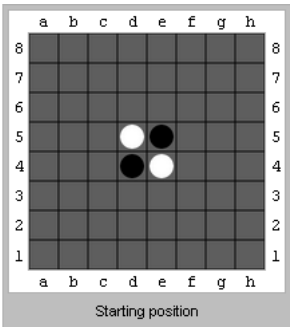

Figure 1: Initial State

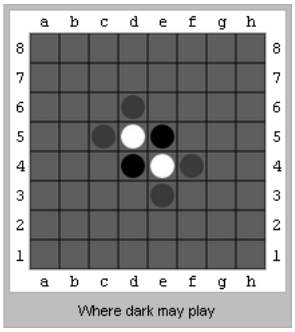

Figure 2: Possible moves of black player are shown in grey

At each player's turn, the player may place a piece of their own colour on an unoccupied square, if it "brackets" one or more opponent pieces in a straight line along at least one axis (vertical, horizontal, or diagnonal). For example, from the initial state black can achieve this bracketing by placing a black piece in any of the positions indicated by grey pieces in Figure 2. Each of these potential placements would create a Black-White-Black sequence, thus

"bracketing" the White piece. Once the piece is placed, all opponent pieces that are bracketed, along any axis, are flipped to become the same colour as the current player's. Returning to our example, if black places a piece in Position 6-d in Figure 1, the white piece in position 5-d will become bracketed and consequently will be flipped to black, resulting in the board depicted in Figure 3.

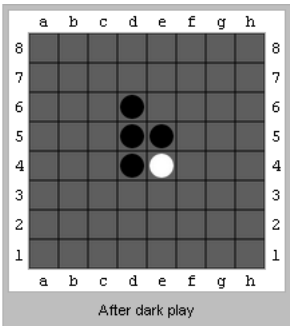

Figure 3: State after the move of black player.

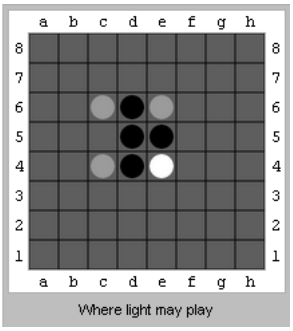

Figure 4: Possible moves of white player

Now it's white's turn to play. All of white's possibilities at this time are shown as grey pieces in Figure 4.
If white places a piece on 4-c it will cause the black piece in 4-d to be bracketed resulting in the 4-d piece being flipped to white as shown in Figure 5. To summarize, a legal move for a player is one that results in at least one of its opponents pieces being flipped. If a player cannot make a legal move, they "pass" and do not place a piece on the board. Players alternate turns until neither player can take a move, at which point the game ends. The player with the most pieces on the board that display the player's colour wins the game.

Othello is Pressman's marketing name for an old game called Reversi. You can get a better feel for the game by playing it at a number of on-line sites. For example, `http://gameknot.com/pg/reversi.htm` shows the allowed moves when it is your turn. In the case where the online description of the games differs from what is specified here, you must follow the description in the assignment.

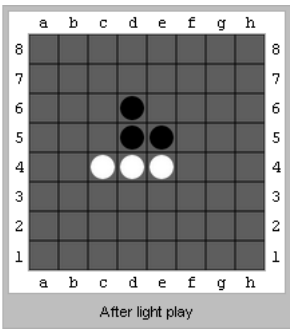

Figure 5: State after the move of white player.

2 The Assignment

You will be provided with the following Prolog code, available for download from the Assignment 2 web page:

- 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 the code carefully to see what predicates need to 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 it's just a position pair like "[1,3]", i.e. 2nd row and 4th column because indices start from 0), which will then be 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. (Observe that the predicate `read(Proposed)` is used to read the user's move—this will bind the variable `Proposed` to anything the user enters, so 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 minimax 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.

- An implementation of an interactive game tree search with alpha-beta pruning in the file `abplay.pl`. This is very similar to `play.pl` except it does the alpha-beta pruning. You will NOT change this file.

- 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 board is treated as a two dimensional 6x6 array indexed by a pair of numbers $[X,Y]$ where X is the row position and Y is the column position, each 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`.

- A sample 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 game tree search. To run the game, simply load the file `ttt.pl` and the ask 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, the computer will choose a move, and it's your turn again, and so on.

This assignment is broken into 3 subparts: (A) implementing Othello on a 6x6 board, (B) designing a heuristic function and (C) testing your code with given test boards, comparing the simple minimax and alpha-beta pruning. Each part is described in detail below. Please note that your implementation should contain sufficient comments and not be contorted or overly complex. **Poor implementation style may cause deductions up to 10%**.

Part A [75 Marks]: Othello

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 your 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 must be sorted by position from top row to bottom row and within a row from left column to right column.** E.g., if moves into positions $[1,1]$, $[0,0]$, $[2,7]$,$[0,2]$, $[1,5]$ are all possible, then you must return this list of moves in the following 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.

The predicate `h(State,Val)` requires that you design a heuristic function for the game. See Part B before doing so. If you decide NOT to do Part B, to get credit for Part A, you still need to define a very simple heuristic: Your $h(S,V)$ returns $V = 0$ for any non-terminal state S. If S is a terminal state, h must return a positive value (say 100) for a win state, a negative value (say -100) for lose state, and 0 for a tie state. You can see that this h provides no guidance in the depth-bounded search.

What to hand in for Part A:

1. Electronic Submission Submit your `othello.pl` (using submit). Make sure you put your name in the comments of that file. (You must not include any of the code in play.pl nor should you submit it.) *Be sure to document your predicate definitions.*

Part B [Optional Bonus Question - 1 Grace Day + Competition Fun]: State Evaluation Function

This question is optional. If you complete it sufficiently well, you will earn one extra grace day for past, present, or future use. Also, all implemented heuristics will be entered into the 2nd Annual Danny Ferreira Memorial Competition. The winner(s) of this competition will be awarded a prize. The competition and prize are unrelated to the mark you receive on this assignment.

As mentioned above, `play.pl` requires implementing the heuristic function $h(S,V)$. If you decide to do Part B, you have to implement a smarter heuristic function as described below. In order to receive the Bonus of "1 Grace Day" you must perform the experiments in Part C with the heuristic you defined here.

Heuristic Functions for Othello Game

Since at the end, the player with more pieces wins the game, you might think that the evaluation function $h(s) = V1 - V2$ (where $V1$ and $V2$ are the number of pieces 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 pieces has no meaning (it could even be worse as seen in figure 6) as many flips might occur in future moves. Instead, we focus more on the *stable pieces* on the board, i.e. those pieces that cannot be flipped

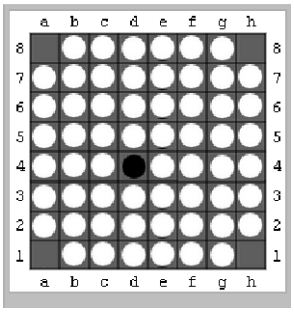

Figure 6: Maximum pieces is not a good strategy: white has a lot more pieces, while black has only 1. It is black's turn. So, she puts a piece in position a1, white has to pass, then black puts a piece in h8, white passes, black plays h1, white passes, and finally black plays a8. Black wins: 40 black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

anymore. Corner positions, once played, remain immune to flipping for the rest of the game (because there is no adjacent opposite colour to cause 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 colour. The more stable pieces you have (and the less stable piece your opponent has) the better. So, you may count the number of stable pieces for both players and use them as a good measure to evaluate states.

Another idea is *mobility*. An opponent playing with reasonable strategy will not 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 these good moves available. 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.

You can also do your own research to find a wide range of other good heuristics (for example, here is a good start: `http://www.radagast.se/othello/Help/strategy.html`).

The Competition: How to package your code so we can run it in the competition

If you decide to participate in the competition (Please do, it will be fun!), you should submit a file named `heuristic_<id>.pl`, where `<id>` should be your cdf userid. For example, a student John Doe with cdf userid `c9jdoe` would submit the file named `heuristic_c9jdoe.pl`.

This file should contain a predicate named `<id>_h`/2 (John would name it `c9jdoe_h`). This predicate can rely on the functions you are required to implement (eg. tie, winner, nextState etc). Any helper predicates should also be included in the heuristic file. To avoid name clashes, prefix any helper predicates with your id. Note that any calls made to them from this file should use the new predicate names.

You should also duplicate the functions lowerBound/1 and upperBound/1 in this file, again, prefixing them with your cdf userid as noted in the example below.

The following is an example submission by John. He clearly did not understand the game and didn't use comments properly, but he got the naming convention right!

```
%-------------------------
% Surname: Doe
% First Name: John
% Student Number: 123456789

% Helper predicates
c9jdoe_max(C1, C2, C) :- C1 >= C2, !, C=C1.
c9jdoe_max(C1, C2, C2) :- C1 < C2.

c9jdoe_count([], 0).
c9jdoe_count([E|L], C) :- length(E, C1),
     c9jdoe_count(L, C2), c9jdoe_max(C1, C2, C).

% The actual heuristic
c9jdoe_h(State, Val) :- terminal(State), !, Val=42.
c9jdoe_h(State, Val) :- c9jdoe_count(State, Val2), Val is Val2-4.

% The bounds
c9jdoe_lowerBound(-3).
c9jdoe_upperBound(300).
```

Note: In the competition, the search will expand as many nodes as it is able to within a given timeout. So, there is a tradeoff between the quality of the heuristic and how fast it is to compute. A better heuristic would provide better guidance to the search, but a fast one would allow more nodes to be expanded.

What to hand in for Part B: To get credit for Part B, you must do the followings: .

1. Electronic Submission Create 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**. Please submit this in a file called `hdescription_c9jdoe.txt` where c9jdoe is replaced with your cdf userid.

2. Electronic Submission Your implementation of predicate *h* must be in the `othello.pl` which you will have submitted in Part A. If you are participating in the competition, you also submit `heuristic_c9jdoe.pl` appropriately renamed to replace `c9jdoe` with your cdf userid, and with the content as described above.

Part C [10 Marks]: Testing Boards & Comparing MiniMax and $\alpha - \beta$ Pruning

In this part, you will test your code by running it with the given test boards. You will also compare simple MiniMax and Alpha-Beta Pruning. We have provided you with the implementation of Alpha-Beta Pruning in `abplay.pl`. Download it from the the assignment web page.

Students who did not elect to try the bonus question should run their tests on the code from Part A. Students who did try Part B should test their code on Part B as well, if they wish to receive the bonus grace day. (It's fun to see how well your heuristic is performing (for us and you)!)

1. **Testing your code with MiniMax**: Trace your code on test boards 1 to 3 (provided in `testboards.pl`) using the MiniMax algorithm with a depth bound of 5. I.e.,

   ```
   testBoard1(St), mmeval(2,St,Val,BestMv,5,SeF)
   ```

   This will bind SeF to the number of states searched, and BestMv to the computed move for board1. Repeat this for testBoard2 and testBoard3. Submit your results following the submission instructions.

2. **Testing your code with alpha-beta pruning**: Trace your code on test boards 1 to 3 (provided in `testboards.pl`) using the alpha-beta algorithm with a depth bound of 5. To do so, you must load the file `abplay.pl` instead. Make sure NOT to load `play.pl`!). Then execute the following:

   ```
   testBoard1(St), lowerBound(Alpha), upperBound(Beta),
         abeval(2,St,Val,BestMv,5,SeF,Alpha,Beta)
   ```

   This will bind SeF to the number of states searched, and BestMv to the computed move for board1. Repeat this for testBoard2 and testBoard3. Submit your results following the submission instructions

3. In one or two paragraphs, compare the results displayed in your two tables. Submit this description following the submission instructions.

Submission instructions to follow shortly.

Good Luck and Have Fun!