EECS 543 Fall 2013

Programming Assignment 3:
Adaptive Opponent Modeling in OTHELLO

Due October 16, 2013 Midnight

YOU MAY WORK IN GROUPS OF UP TO 2 PERSONS ON THIS ASSIGNMENT

Getting a computer to play a game against a human (or another computer) has been of recurring interest to AI programmers, computer scientists, and people more generally. In this assignment, you will have a chance to explore the basic game-playing paradigms of employing a finite-horizon minimax lookahead search with a static evaluation function. In particular, you will augment the book's implementation of the game of OTHELLO with improved abilities to use, and even adapt, a model of the opponent.

OTHELLO (Chapter 18 in the Norvig text) is a strategy game (like chess or checkers) that does not involve an element of chance (so, it is unlike backgammon or poker). Because of the number of possible ways the game can evolve, however, it is generally infeasible to select a move based on looking ahead to all possible concluding states of the game. Instead, the look ahead is generally to a limited depth, and from there the board position is evaluated and scored in terms of how desirable it is. Obviously, the quality of a player will be dependent on the scoring function. But, furthermore, it will also depend on whether the look-ahead process reasonably anticipates the opponent's likely decisions.

In general, if a player uses his or her best knowledge, then the player can model the best action to take assuming that the opponent makes the best response, down to whatever ply is desired. The player thus assumes the opponent possesses the same (or equally good) knowledge, and thus the player plays very conservatively by anticipating the opponent will do the worst thing possible (from the player's perspective). If, however, the player knows that the opponent is a weak player, and what the opponent's weaknesses are, then the player might be able to take actions that exploit the weaknesses. This, however, would require that the player model the sequence in the game not as a series of "smartest" responses to others' moves, but rather as the player's smart responses to the opponent's less-smart moves, and the opponent's less-smart responses to the player's smart moves.

This assignment is intended to explore this issue. Part of what you should learn includes basic game-tree searching paradigms and heuristic evaluation techniques. But on top of that is some initial exposure to issues of using models of other players well, and even recursive multiagent reasoning (about what a player thinks the opponent thinks…).

What to turn in: Turn in your modified code file, and a separate document with your answers to the questions. You should paste lisp output (and code if needed) into this other file to show your code's results and to illustrate your points.

Task 1 (5 points)

Understand OTHELLO (Chapter 18 – especially up to and including 18.4). Because this assignment will be comparing performance between players, explain why in your code you should replace the `final-value` function with the below. (Hint: Read the rest of this assignment before answering this!)

```
(defun final-value (player board)
  "Is this a win, loss, or draw for player?"
  (count-difference player board))
```

Task 2 (15 points)

As a warm up, consider the following. Norvig provides code for associating weights with the benefits (or costs) of having a disk on a particular square. He gives a specific array of `*weights*` that arguably performs well.

a. What would happen if each of the squares that could be occupied was given exactly the same weight? Create an `*equal-weights*` array to represent this case, giving weights of 1 to all positions. Compare how using weighted-squares with these new weights performs using the minimax-searcher with a depth of 3, compared to minimax to depth 3 using count-differences. How would the results change if a different constant value was assigned to all positions? Don't forget to consider non-positive constants!

b. Do the same thing as in part a, except create a `*dumb-weights*` version of the `*weights*` array by inverting the sign of each entry in `*weights*`. Again, how does this impact the performance (using minimax to depth 3, compared to count-differences)?

Task 3 (30 points)

The existing code doesn't let us (easily) play a game where the players use different weights (different players wouldn't necessarily use the same strategy!) because it has a single special-variable pointing to the `*weights*`. We now want to remedy this.

First, define a new special variable `*player-weights*` that points to a nested association-list data structure as follows. We want each agent name to be associated with an association list, where each inner association list associates an agent name with the weights ascribed to that agent's model. So for example (but you should treat the special variable appropriately!), the following indicates that black believes black is using the *smart-weights* and white believes white is using the *dumb-weights*:

```
(setf *player-weights*
   (acons black
       (acons black *smart-weights* NIL)
       (acons white (acons white *dumb-weights* NIL) NIL)))
```

Now, revise `weighted-squares` and any other relevant code to make use of this new information so that the right weights will be used when computing the board value for each player. That is, a player will use its own weights when doing minimax and will assume that its opponent is also using those same weights.

Evaluate the impact of these changes by assessing what happens when a player with smart weights plays an opponent with dumb weights. How about when a player with smart weights plays an opponent with equal weights, or a player with equal weights plays a player with negative weights? Can you draw any conclusions about how well a player can do when competing against an opponent possessing inferior knowledge? Does it take advantage of its superiority well? Does this depend on how deeply it searches?

Task 4 (40 points)

Now let's explore what happens if a player knows that it is playing against an opponent who has inferior knowledge, and knows what its opponent's inferior knowledge is.

Define a new function called `multimodel-weighted-squares`, based on `weighted-squares`, that will compute two values for a board position: the static evaluation value that the current player ascribes to it, and the value that the player believes the opponent will ascribe to it, making appropriate use of the `*player-weights*` special variable.

Further, define a new function `multimodel-minimax-searcher`, based on `minimax-searcher`, that can work with your `multimodel-weighted-squares` function. The idea is as follows. Let's say that it is black's turn. Black will want to take an action that will lead to the best board position (as judged by black's evaluation function) that it can get to. When it considers what white will do, however, it will think that white will chose an action that looks best for white as judged by what black believes is white's evaluation function! So not only will the players alternate between "minimizing" and "maximizing", but also between which evaluation value to use for the states.

When you have coded up your solution, evaluate the implications of having players use models of their opponents for various depths of search. Do things like:

a. Evaluate whether a player that has good weights, and knows that its opponent has not-so-good weights, can do better against the opponent than if it had assumed the opponent had good weights.

b. Evaluate whether an opponent with worse weights who also had a (correct) model of the other player's weights does better than without such a model.

c. Evaluate what would happen if a player with good weights mistakenly models its opponent as being less smart than its opponent really is.

Task 5 (10 points)

Modeling an opponent as having a different static evaluation function can potentially work out well if that model is (close enough to) correct, but it could be wrong. More generally, a good player adapts his or her model of his or her opponent based on the

behavior of the opponent. That is, if its opponent's plays are smarter (or less smart) than expected, then it upgrades (or downgrades) its model of the opponent's abilities.

a. A possible first step in adapting a model of the opponent is to determine whether the current model is not working well. For this task, your goal is to modify you multimodel functions (and perhaps a few other places) as necessary to do so. Your multimodel-minimax-searcher should not only return the best move for the multimodeling player, but also keep track of the expected next move of the opponent as predicted by the minimax search. When it is this player's turn again, it should compare the actual move of the opponent to what it had predicted, and (for now) flag any discrepancies by printing this out. (Hint: Retrofitting the code to pass around the right information might be more than you want to do. It is okay to define and use appropriate special variables to support the desired functionality.) Here is a snippet from my code running:

```
...
             (*PLAYER-WEIGHTS*
                    (ACONS BLACK
                           (ACONS BLACK *EQUAL-WEIGHTS*
                                  (ACONS WHITE *NEGATIVE-WEIGHTS* NIL))
                           (ACONS WHITE (ACONS WHITE *EQUAL-WEIGHTS* NIL) NIL)))

               (OTHELLO (MULTIMODEL-MINIMAX-SEARCHER 3 #'MULTIMODEL-WEIGHTED-SQUARES)
                        (MINIMAX-SEARCHER 3 #'WEIGHTED-SQUARES) NIL))
Prediction was wrong. Predicted was E3. Actual was C5
Prediction was wrong. Predicted was E3. Actual was D2
Prediction was wrong. Predicted was F4. Actual was D6
and so on...
```

As per the example above, when the multimodeling player's model of the opponent is not right, we would expect lots of wrong predictions. To explore these ideas further, do the following with your code for this:

a. Look at a few cases where the predictions were wrong, and explain why. (Hint: Might be easiest to do for early states in the game when the number of move choices is small.)

b. Look at cases where the multimodeler's model of the opponent is correct. Are its predictions always correct? Explain why or why not.

c. Describe how you would use information about wrong (and possibly right) predictions to adapt the multimodeler's model of the opponent. You do **not** have to code up your solution, but you should **summarize** what parts of the code you'd modify and how you would modify them, as well as any other changes you'd want to make (e.g., new special variables, etc.)

OPTIONAL Task 6 (10 bonus points)

Implement and evaluate your solution to part c of Task 5. Can it can help the situation from the snippet in Task 5 by shifting the multimodeler to a better model of its opponent? Does it sometimes adapt incorrectly (shifting from one wrong model to another, or away from a right model)?