INTRODUCTION

Welcome to the wonderful world of AI. This handout is designed to outline a basic artificial intelligence algorithm that you will implement for your Othello computer player. It would be very wise to review the recursion lecture before going through this handout.

The algorithm you will be implementing is known as a MiniMax search. MiniMax is one in a family of so-called "adversarial searches", which means that its goal is to find a move that gives the player the highest possible score or the opponent the lowest possible score. It does this by looking several moves ahead and checking every possible combination of actions. For instance, a computer of "intelligence 3" would look three moves in advance and choose the move leading to the best outcome. The computer has the ability to examine hundreds and hundreds of moves in an instant and pick one. A human would have trouble remembering and comparing more than three or four moves.

LEVELS OF INTELLIGENCE

Minimax uses a scoring function that takes in a board position and outputs a score that reflects how good that position is for the current player.

A computer with level 1 intelligence maximizes its possible score after one move. It does this by looking at all of the possible moves that can be made and choosing the best one. It does not consider its opponent's responses to any of these moves.

A computer with level 2 intelligence minimizes the maximum score that the opponent can achieve during the opponent's turn. It looks at each next move it can make, and the responses that its opponent can have to each one of these moves. It finds the current player's best move such that after the current player makes this move, the opponent's best move is as bad as possible for the opponent.

A computer with level 3 intelligence maximizes its own score two moves ahead. It does so by considering all the moves it can currently make, considering all the ways the opponent can respond, then considering all the ways to respond to the opponent's moves.
THE MINIMAX ALGORITHM

For Othello, which requires three intelligence levels, the algorithm needs to recur at most three times in order to evaluate the current player's best move (at most, three moves away). The number of turns ahead that the computer player must check will depend on the computer player's intelligence level: level 1 looks one move ahead, level 2 looks two moves ahead, and level 3 looks three moves ahead. The base case for this recursive algorithm is the move on the final level of recursion, i.e. the move that is most “into the future”.

For a level 3 AI, move two’s board evaluation is dependent upon move three, and so, move three is considered the base case. However, in order to get to the base case, MiniMax must make each move on a “dummy” board. If the player is using intelligence 3, then it will keep making moves on “dummy” boards until it reaches a potential third move.

In the base case, while looping through the “dummy” boards for each possible move, MiniMax uses the evaluateBoard method (discussed below) to find the move that results in the best board state. The value of the best move is returned up to the previous layer of recursion. The value received by that previous level of recursion is negated because it represents the score of the board from the other player's point of view.

Each layer of recursion looks for the player's most advantageous move by asking the other player for its best move. This is why the best move value is negated-- by negating player A's best move, it becomes player B's worst move (for example, a value of 10 [player A's best move] becomes -10 and a value of 4 [a not-so-great move] becomes -4... -4 is a better move than -10, so player B's best move is the one that caused player A to return a move value of -4). At each level of recursion we reverse the player perspective from which we view the board.

The evaluateBoard method is used in the base case to evaluate the advantage or disadvantage of the current player by returning an integer value that represents the “goodness” of the board’s state for the player after he/she makes a move. This calculation is based on all of the player’s pieces on the board that has just been played on (i.e., the dummy board). The method sums up all of the board weights for the spaces that the current player controls, and subtracts from that sum the sum of the weights of spaces controlled by the opponent. A square has a high weight in the evaluateBoard method if it is a better move for the current player (for example, a corner piece has a very high value). The number returned by the evaluateBoard method is positive if the current player has an advantage (i.e. is winning) or is negative if the current player has a disadvantage on the game board being evaluated.

If after the following example this does not make sense please please please come talk to a TA!
EXAMPLE

Here is an example of the MiniMax algorithm using level 3 intelligence.

First, MiniMax determines all possible move 1s. Then for each move 1, it creates a “dummy” board in order to simulate the potential move. On each move 1 “dummy” board, it makes every possible move 2 on a new “dummy” board. This repeats for move 3. In the diagram below, each circle represents a “dummy” board and a potential move that MiniMax is considering.

Now that MiniMax has all the potential third moves, it has reached its base case of recursion. Using the evaluateBoard method, it finds the best possible move values for each move 3. The diagram below shows some possible values.

Now that the best values for move 3 have been found, the highest values will be negated and then returned to the previous layer of recursion (i.e., the second layer). In this layer, the algorithm needs to consider the moves from the opponent’s point of view.
Now we play the game from the opponent’s perspective. From these move 2 values, the “current player” (that is, the opponent) chooses the highest value. This value will be then negated and returned up to the previous layer of recursion. In this layer, again, the algorithm needs to consider the moves from the opponent's point of view, so the “current player” is now the opponent of our opponent -- i.e., the original current player.

![Diagram of game configuration]

Now, we can see that the best value for move 1 is 8. Note that even though 10 is the best move 3, the opponent (move 2) will not make that choice because -2 is a better move. Here, the best choice for move 1 is not the best choice for move 3, nor is it the best move for the opponent-- rather, it is the best move 3 that the current player can make assuming that the opponent makes its best move.

![Diagram of game configuration]
evaluateBOARD

The current player needs to be able to assign some strategic value to a move when it hits the base case. Several factors go into deciding what kind of value a given move has. On a first look, we could say the value of a move is the number of pieces of one color minus the number of pieces of another color on the resulting board. But this is a fairly naive approach because we know by playing the game a few times that certain squares on the board give players a better strategic advantage. To decide the value of a particular spot on the board, we can assign an integer value to each of the 64 squares on the board. This way corner squares, strategically the best spots on the board, will have very high values. Spaces adjacent to corners will have low values because they will usually allow the opponent an opportunity to take the corners. A board evaluation will represent a player's advantage after a given move.

To compute a board value, we say that every Othello piece has a value. A white evaluateBoard would be the sum of all the values of the white pieces minus the sum of all the values of the black pieces. If a evaluateBoard is negative, then the opponent has the advantage.

BOARD WEIGHTS

We are providing a class for you which you can use to get predefined board weights, or relative values, for your squares. It has a few simple methods, simply send it an x and y coordinate and it will return a board weight to you. If you have extra time and want to experiment with different board weight configurations, feel free to replace this class with one of your own which returns different board weights.

To do this, write a class that extends cs015.fnl.OthelloSupport.BoardWeights -- just like the DefaultWeights class extends off this class.

Name:
public cs015.fnl.OthelloSupport.DefaultWeights

Purpose:
This class models a set of weight values for a 10x10 Othello board (note that 10x10 is really 8x8 with padding).

Methods:
public int getWeight(int x, int y)
Returns the weight (relative value) of a given location on a 10x10 Othello board.
MINIMAX, THE NITTY GRITTY

This pseudocode is VERY helpful. Make sure you fully understand this pseudocode. Ask a TA if any part does not make sense. You’ll be asked to give a more detailed version of this pseudocode for your design checks. Make sure you think critically about what important pieces are missing from this pseudocode, and how you can improve it further.

Some high level pseudocode:

```java
public <return type> getBestMove ( <parameters> ) {
    if the game is over
        If the current player won the game, the return value should be very high. If it lost the game, the return value should be very low.
        else if no moves for the current Player
            This means that the current player has to skip its turn. getBestMove should be recursively called, but from the other player's point of view.
        else
            for every legal move for the current player
                Make a test board.
                Make the move on the test board.
                If you're at the base case, the value of this move is just the evaluateBoard for the current player.
                Otherwise, the value of the move is the negation of the value obtained from recursive calls to getBestMove.
                Keep track of the best value along with the move that is associated with that value.
}
```
STORING AND RESTORING THE BOARD

One last thing to think about... How does the computer player test out different moves without destroying the current board? One possible solution is to use something called a copy constructor. A copy constructor is just a regular constructor for a class that takes in a reference to a class of its own type. It is constructed as a copy of the object passed into it. Since we don't want the computer player to alter the real board as it's thinking, the computer player can use a copy constructor to create a new board that is a copy of the previous board. This way, when a computer wants to make an imaginary move, it can do so on a test board without making the moves on the one and only visual board. See the help session slides for more information about how copy constructors work.

SWAPPING AI STRATEGIES

Minimax is only one AI algorithm used in game play. What if you wanted an easy way to swap in and out other AI algorithms in your Othello? You could use polymorphism, of course! Creating a separate Minimax class that implements some interface, say, Strategy, and having your computer player’s move method call methods on a given Strategy would give you the flexibility to change the AI that your computer player uses to make moves.

Note, it is NOT required for this assignment to have your Minimax class implement the Strategy interface, since you will only be writing one AI strategy (unless you have time to do some extra credit). It will also not be extra credit if your Minimax implements Strategy. However, we encourage you to think about how a Strategy interface would look, particularly what method(s) it would declare, and how it would make your life easier if you decided to enhance your Othello with many different AI strategies after graduating from CS15.

This is only one design, and not the only design. If you think you have a better way, think about it more and talk to a TA. Then if you think it will work, do it! (This goes for other parts of the Othello design as well.)