CS 4260/5260: Introduction to AI
Programming Assignment 2 [100 points]: (Multi-Agent Search)
Due: Wednesday, October 24, 2018, 11:59 pm Central Time on BrightSpace

General Instructions:
If anything is ambiguous or unclear.
1. Discuss possible interpretations with other students, your TA, and instructor
2. Send e-mail to your TA first, and to your instructor if an issue is not resolved to your satisfaction.
3. Make use of web sources.

Remember that this course's honor code allows you to get help from others, but you must disclose the specific help you received in comments at the top of your submission file. Please refer to the Honor code for more.
Write legibly, be sure to staple all your answer sheets together, and write your name, and the honor pledge on the top of the first answer sheet.
Start early, and avoid last minute stress!

Introduction

In this project, you will write code for a pair of agents who play Othello against each other. The functions that return the next move have been written for you. Your job is to write the helper functions for minimax and minimax with alpha beta pruning. You will build general multi-agent strategy algorithms and apply them to different initial board layouts.

The code for this project is stored in files inside the main directory folder. The file you will be editing is called **strategies.rkt** Example initial boards are stored in text files with .lay extensions inside the layouts folder.

When you have finished the assignment, upload your **strategies.rkt** file to the submissions folder on BrightSpace. Do not upload anything else.

To test your code, you can visualize it running against other strategies in a single game, play against a group of strategies in a round robin, or run targeted tests using the functions in the tester.rkt file.

Submitting:
Once you have finished upload **ONLY strategies.rkt** to BrightSpace.

How to Run Your Code

The code for this project consists of several Racket files, some of which you will need to read and understand in order to complete the assignment, and most of which you can ignore. You can download all the code and supporting files as a zip archive.

Your code should return two values: the value of the best move according to the player making it, and the best move.

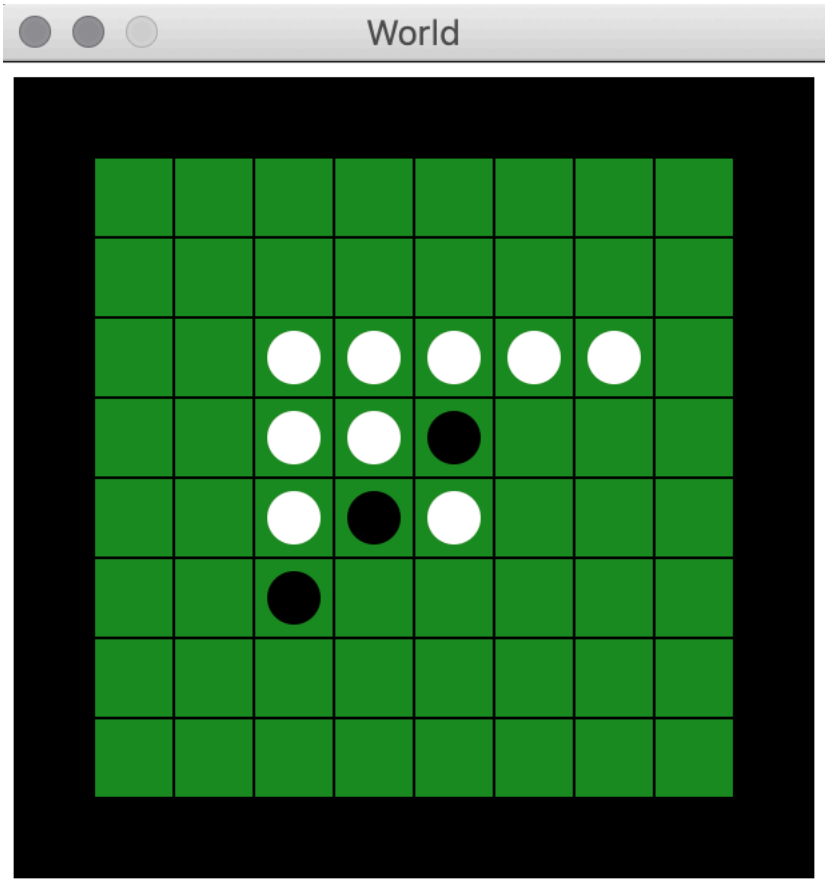
To visualize your code against another strategy run:

```
racket main.rkt -l <path to layout in layouts dir> -w <white player's strategy> -b <black player's strategy> -t <length of time in seconds you want each move to take>
```

For example:

```
racket main.rkt -l layouts/basic.lay -w maximize-difference -b maximize-weight -t .5
```

You should see a visualization like this:



Available implemented strategies include:

```
random-strategy
maximize-difference
maximize-weight
```

Once you have implemented minimax:

```
minimax-searcher-1-weighted-squares
```

```
minimax-searcher-1-count-difference
minimax-searcher-2-weighted-squares
minimax-searcher-2-count-difference
minimax-searcher-3-weighted-squares
minimax-searcher-3-count-difference
minimax-searcher-4-weighted-squares
minimax-searcher-4-count-difference
minimax-searcher-5-weighted-squares
minimax-searcher-5-count-difference
```

Once you have implemented alpha-beta:

```
alpha-beta-searcher-1-weighted-squares
alpha-beta-searcher-1-count-difference
alpha-beta-searcher-2-weighted-squares
alpha-beta-searcher-2-count-difference
alpha-beta-searcher-3-weighted-squares
alpha-beta-searcher-3-count-difference
alpha-beta-searcher-4-weighted-squares
alpha-beta-searcher-4-count-difference
alpha-beta-searcher-5-weighted-squares
alpha-beta-searcher-5-count-difference
```

You can see a complete list of the available layouts in the layouts folder and build your own layout files if you'd like.

To observe a set of strategies played in a round robin, use the s flag to name each strategy (if there is at least 1 s flag, the code will run a round robin):

```
racket main.rkt -l <path to layout in layouts dir> -s <strategy
1> -s <strategy 2> -s <strategy 3>
```

For example:

```
racket main.rkt -l layouts/basic.lay -s maximize-difference -s
maximize-weight -s minimax-searcher-2-weighted-squares -s
minimax-searcher-2-count-difference -s alpha-beta-searcher-2-
weighted-squares -s alpha-beta-searcher-2-count-difference
```

You should see an output like this:

```
(alpha-beta-searcher-2-weighted-squares 10)
(minimax-searcher-2-weighted-squares 10)
(alpha-beta-searcher-2-count-difference 5)
(minimax-searcher-2-count-difference 5)
(maximize-weight 4)
(maximize-difference 2)
```

Once you believe your code is working, run tester.rkt either from the command line or in DrRacket. This code will take a long time, but it should give you specific feedback about any remaining errors in your code.

Problems

For this project, you will implement minimax and minimax with alpha beta pruning.

Both functions that you implement will return the value of the best next move and the recommended next move.

If the search depth is 0 or there are no legal moves available, the function should return false (#f in racket) for the move.

Minimax Without Pruning

This part is easier than minimax with pruning, so please start with it.

You will be implementing a modified version of the pseudo code shown in chapter 11 section 3 of the book:

```
1:procedure Minimax_alpha_beta(n, α, β)
2:     Inputs
3:         n a node in a game tree
4:         α, β real numbers  best score so far
5:     Output
6:         A pair of a value for node n, path that gives this value
7:     best := None  False
8:     if n is a leaf node then
9:         return evaluate (n) , None  False
10:    else if n is a MAX node then
11:        for each child c of n do
12:            score, path := MinimaxAlphaBeta (c, α, β)
13:            if score ≥ β then
14:                return score, None
15:            else if score > α then
16:                α := -score
17:                best := c : path
18:        return α, best
19:    else
20:        for each child c of n do
21:            score, path := MinimaxAlphaBeta (c, α, β)
22:            if score ≤ α then
23:                return score, None
24:            else if score < β then
25:                β := score
26:                best := c : path
27:        return β, best
```

The major differences (in addition to not using alpha-beta pruning) are that:

1. Your function will take four arguments
    a. player: The player whose turn it is
    b. board: The current board layout
    c. depth: If depth equals 0, then you are at a leaf node, otherwise decrease the depth for the next iteration.
    d. evaluation function: You will use evaluation function to evaluate leaf nodes by call (eval-fn player board).
2. Your code will treat **both** players as **maximizing agents**. When you receive the expected score from an opponent, take the negative of that score, as shown in the assignment of alpha above

3. Your code will return a **single action**, not a path (since the path may be changed if the opponent doesn't play as expected). In the pseudocode above, this corresponds to only returning the action c, not the whole path.

   The list of legal moves for the player is returned by the function (legal-moves player board)
4. You must handle the case when no legal moves exist.
    a. If neither player has legal moves left, return a value of (final-value player board) and #f for the move.
    b. If the opponent can make a move, the current player must pass their turn by playing #f. In this case, the returned values should be the negation of the opponent's minimax score and #f for the move.

Minimax with Alpha-Beta Pruning

The code for alpha-beta is similar to minimax. It takes two additional arguments: achievable and cutoff. Achievable is the value of the maximum path found so far and cutoff is the minimum option found for the opponent so far.

If achievable is less than or equal to cutoff, the search should stop since the current branch is not one that a pair of optimal players would go down.

To exit cutoff branches, you will probably want to use a for loop with a break statement. You can check whether your code is cutting off enough branches by comparing the time for minimax to the time for alpha-beta. Alpha-beta should always run faster when the depth is 2 or more. You can test if you are getting enough of a speedup using the (alpha-beta-faster?) function in tester.rkt.

```
1:procedure Minimax_alpha_beta(n, α, β)
2:     Inputs
3:         n a node in a game tree
4:         α, β real numbers
5:     Output
6:         A pair of a value for node n, path that gives this value
7:     best := None                                   False
8:     if n is a leaf node then
9:         return evaluate (n) , None                 False
10:     else if n is a MAX node then
11:         for each child c of n do
12:             score, path := MinimaxAlphaBeta (c, α, β)   -β, -α
13:             if score ≥ β then
14:                 return score, None                 c
15:             else if score > α then
16:                 α := -score
17:                 best := c : path
18:         return α, best
19:     else
20:         for each child c of n do
21:             score, path := MinimaxAlphaBeta (c, α, β)
22:             if score ≤ α then
23:                 return score, None
24:             else if score < β then
25:                 β := score
26:                 best := c : path
27:         return β, best
```