Othello

Othello [https://youtu.be/xDnYEOsjZnM] is a 2-player adversarial game in which a player places pieces into a grid and tries to align them in order to win the game:

The object is to have the majority of your colored disks face up on the board at the end of the game. Every disk is white on one side and black on the other.

Setup: Initially, we place four disks, two with white sides and two with black sides up in the center of the board. Each player has 30 disks. One player plays as white and the other as black. The player playing black goes first, and then turns alternate.

Play: On your turn, you must place one disk on any empty space on the board and outflank your opponent. Outflank means to have your disks on either side of a continuous straight line of your opponent's disks. This line can be any number of disks long and can be horizontal, vertical, or diagonal. When you place a disk so that you have outflanked your opponent's disks, you then flip the outflanked disks over to your color. You are allowed to outflank multiple lines in a single turn.

If you cannot outflank your opponent on your turn, you are not allowed to play, and you must skip your turn. *If you can play, then you must play*. Disks may only be outflanked due to a move and must be in a direct line of the placed disk. An outflank may not skip over your own colored disk to outflank more disks. Only the disks within the immediate outflank are captured.

All outflanked disks must be flipped. You may not choose only to flip some of them. If you flip over the wrong disk and your opponent hasn't moved yet, you may fix the error. If your opponent has already moved, then you may not fix the error. Once a disk is placed on a square, it may never be removed, nor may it be moved to another square. If you run out of disks to place, and you still have a legal move, then your opponent must give you some of theirs to use. When it is no longer possible for either player to move, the game is over. Disks are counted, and the player with the majority of their color showing is the winner.

You can also play multiple rounds and track points. The round winner receives points equal to the difference between the number of their opponent's disks, subtracted from the number of their disks. Once players reach a predetermined point total, the game ends, and the player with the most points is the winner.

The Assignment: In this assignment, we will use the minimax algorithm to make an AI agent that plays the game of Othello. You are given the code to represent the board, handle the placement of pieces, and determine whether the game is over. Your task will be to extend this code into a full-fledged agent that can play and win the game. The sample code provided is written in Python, and the code for this assignment should run on tux.cs.drexel.edu with a shell script, as we have done for previous assignments.

Implementation Setup

The various parts of this assignment will require a shell script that can pass an argument to your code—thus allowing you to use the **python3** command on tux while allowing us to be able to run your code with a consistent interface. Again, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

The shell script is the same code as you used in the previous assignments, including the ability to accept two arguments in the general format:

```
sh run.sh <argument>
```

As before, this scheme will allow you to test your code thoroughly and allow us to test your code using a variety of arguments.

A sample shell script **run.sh** has been provided with this assignment. You should not need to modify this script file.

Game Implementation

Also provided with this assignment is the Python game implementation of the game. You can test it by running the command:

```
sh run.sh human human
```

which should run a game where both players are humans and generate moves from the input command line.

You can assume for this assignment that the two players are represented by 'X' and 'O'. Most importantly, the Othello implementation is structured as follows:

- "OthelloMove": this class contains a "move" (which player made the move and the coordinates of the move)
- "State": this is the core class, which implements most of the functionality of the game. The functions you should be aware of for implementing minimax are:
    - "generateMoves": this function returns the list of moves for the next player to move.
    - "applyMoveCloning": this function creates a new game state that has the result of applying move 'move'.
- "Player": this is an abstract class defining an agent that player Othello. Your agent should be implemented as a class that extends this one.
- "game": this class uses all the above classes to play a game of Othello.

If you look into the agent.py file, you will find an implementation of an agent that plays Othello by the input that comes from the user.

Part 1: Random (3 pts)

For the first part of the assignment, implement an agent that plays Othello by choosing moves at random. You can test it by running the command:

```
sh run.sh random random
```

Part 2: Minimax (5 pts)

In this part, we are asking you to implement an agent that plays Othello using the standard minimax algorithm, as we studied in class. The agent should be able to play both as the first and second players.

For the evaluation function, just use the "score" function that is provided to you in the State class (make sure that your bot can play both as the first or second player). Your agent's constructor should accept the depth up to which we want to search.

To make sure your agent works, make it play against the random agent. Your agent should defeat it easily! We should be able to test it by running the command:

```
sh run.sh minimax random <depth>
```

Part 3: Minimax + Alpha-Beta Pruning (2 pts)

Implement a third agent, which uses alpha-beta search. Again, your agent's constructor should accept the depth up to which we want to search and be able to play both as the first and second players. Compare the times that your two agents take to search up to different depths and *report it in the PDF document*.

We should be able to test it by running the command:

```
sh run.sh alphabeta random <depth>
```

Part 4: Extra Credit (up to 2 pts)

Implement another agent in a file called "<drexel_userid>.py", where <drexel_userid> is your lowercase Drexel user ID (e.g., awm32.py). Name the agent class the same thing (e.g., class awm32:). **Also, include all other classes that are needed to run the agent in the same file**. This agent, instead of receiving the depth at which to perform a search, receives a certain amount of time (in milliseconds) that it can use to search. Make sure that your bot returns a solution within this time (you can assume that you will have at least 100 milliseconds).

We should be able to test it by running the command:

```
sh run.sh extra random <time>
```

Hint: a good way to do this is by making your agent first search at depth 1. Then, if there is still time, search at depth 2. If there is still time, go for depth 3, etc. Also, make sure that you have code that cancels the search if enough time has passed.

You will receive 1 extra point by implementing this agent; furthermore, we will also collect all the agents that comply with the instructions and play a tournament. Results will be announced in class in the case. For this part, you can make changes in the original functions such as "score". **The top 5 will receive up to 1 additional point**.

Important Note: In all the above algorithms, your agent should be able to play as the second player. That will be specifically important if you're doing the extra credit. For example, we should be able to run the below command with Player 'X' winning most of the time, depending on the depth:

```
sh run.sh random alphabeta <depth>
```

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit:

- Your Python code for this assignment.
- Your **run.sh** shell script that can be run as noted in the examples.
- A PDF document with written documentation containing a few paragraphs
    - explaining your program,
    - analyzing the time complexity of algorithms and comparison (one or two paragraphs is enough),
    - results showing testing of your routines.
    - and if you did anything extra as well as anything that is not working.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—do not email your assignment to a TA or instructor! If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment is due. If you have any doubts, complete your work early so that someone can help you.