Turn-Based AI with Othello: Modeling

Eckel, TJHSST AI1, Fall 2022

Background & Explanation

The culminating assignment in gaming AI will be a tournament that pits our various Othello artificial intelligences against each other. This will be awesome. But of course, before we get there, we need to effectively model and play Othello! The first part of this assignment is just to get all the internal modeling working. This is harder than you think; a lot of this assignment (and the future Othello assignments) will be based around giving you tools to test and debug your code, which will be more difficult than usual.

But before you code, let's make sure you know the game.

A good primer on the Othello rules is here: https://www.wikihow.com/Play-Othello

I'd also recommend googling an Othello game you can play online and playing a couple of rounds against an AI.

Some important notes:

- The board is always 8x8.
- Black always moves first.
- All eight directions are available for flipping tiles – up, down, left, right, and all four diagonals.
- If a tile traps enemy tiles in two or more different directions at once, they are all captured.
- If someone can't make a valid move, play passes back to the other player. If neither player can move, the game is over.
- A player cannot intentionally pass if a legal move is available. A player must make a move.

Assignment Overview

By the end of the Othello assignment, you'll be submitting your code twice - once to my grading scripts and once to an Othello server that TJ runs where you can play your strategy against any others. You'll want to use the same strategy for both, but there will be differences in how the two files need to be formatted. Along the way, you'll be downloading many different helpful tools / programs to give you support in debugging and diagnosing your problems. These tools will also need to import some functions from you, to avoid giving away secrets.

As a result, you'll end up needing three files:

- A file that MUST be named `othello_imports.py`, which many of my utilities will import functions from, containing code to model the game.
- Two different versions of a self-contained strategy file that does **not** import from the above file, but **does** contain all the same modeling code as well as the AI strategy you write. Specifically:
    - One version (for my grading scripts) that takes command line arguments, computes one turn, and prints the chosen move.
    - One version (for the server) that uses a multi-threading object to take in information and report the chosen turn.

You don't have to internalize all this right now; I'll provide clear instructions for everything throughout the assignment. I introduce these ideas at the beginning just to give a heads up; you should know there's a lot more pickiness in this assignment than usual. You're writing three separate files to highly picayune specifications, not just one. So be ready to do some patient double checking / referring back to instructions carefully.

Board / Player Representation Conventions

When you're submitting your work to either my grader or to the tournament server, the following conventions will apply. You do NOT have to store your board internally in this way (see later in this document for some alternate suggestions). You DO have to read in input and send out output according to these conventions, so if you do something else, you'll need to write code to convert back and forth.

On the server, the board will be conceptualized as a length-64 string – an 8x8 grid. Empty squares are ".", Black is "x" and White is "o". This 8x8 grid is stored as length-64 string in row major order.

The initial starting configuration looks like **THIS** in the middle:

```
ox
xo
```

This means that, as a 64-character string, the starting board is **always** this:

```
"...........................ox......xo..........................."
```

On the server, locations will be referenced by integer index on this string. You will communicate the move you select to the server using this integer index. Again, it is not required that you track indices this way internally, but that'll be the goal for output.

First: Make `othello_imports.py`

In order to make sure that your model is correct, and then use your model to help test your strategy later on with a few utilities you'll download from me, you need to make functions that perform the two essential modeling tasks here – figuring out what possible moves are available on the board, and making a particular move in a particular place. In order for my later utilities to work, your file name and both function names below **must be named exactly this way**.

1. Write a function with precisely the definition `possible_moves(board, token)`. The two arguments – a board and a token – are as above, a 64-character string representing the board and a single character that is either `"x"` or `"o"`. This function needs to return a list of all possible squares that can be played into by that token. This is harder than you think, and you should plan on spending a lot of time consulting other students and/or figuring out how this will work on paper. If you'd like to figure this out yourself, rock on. If you'd like some hints, **all the way at the end of this assignment** I have a couple.

2. Write a function with precisely the definition `make_move(board, token, index)`. This function takes three arguments – a board and a token as above and an integer representing a valid move position – and returns a board, a 64-character string, where the token has been played to the indicated position and all attendant token flips have been performed. This function does not need to account for invalid input; it will only be called with valid parameters.

You will need to test both of these functions thoroughly! Thankfully, I have provided you many tools to do just that.

How to Check Your Modeling Code

Three of the files provided with this assignment are important here. Place your `othello_imports.py` in the same folder as `check_othello_imports.py`, `boards.txt`, and `answers.txt`. Then, simply open and run the file `check_othello_imports.py`. (If you're using VSCode, make sure the terminal is also on the same directory.)

If your functions don't meet the spec above, you'll get Python errors. Once you fix that, your code will be tested on over 600 boards. On any board, if you produce an incorrect list of moves, or perform an incorrect move, you'll get information on what went wrong. Once your code produces no errors at all, you should also look at the timing information at the end. Your code should process 600 boards incredibly quickly, in the blink of an eye. If it takes longer than half a second, you'll likely have problems making a good strategy later.

Specification

Submit a single Python script to the link on the course website. My grader will rename your file so that I can import from it, so there aren't any special instructions for the file name. Just follow the usual conventions.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Both functions in your code work correctly. (Do not submit until all 600+ tests are perfect.)
- Your runtime is insanely fast (if you're under half a second, you should be fine)

Some Advice for Implementing Code to Find & Make Othello Moves

Ok, so here's the problem.

The problem is that when you start thinking about this you probably start thinking about looking at every index on the board, and then looking in all eight directions from there, and you're thinking that probably means you need a totally separate for loop for each direction.

And you might be right! Because the for loops that go to the sides need to make sure they don't leave one side and come back on the other side, and that's annoying to try and code generally.

So: I give you two pieces of advice; do with them what you will.

1) I know this sounds like a huge pain, but when you get the 64-character board, convert it to a 100-character board where you've added a 1-character border on all sides. Like this:

   ```
   x x o o . x o x
   . . . . . o o x
   . . . . . . o x
   o . . o o x o o
   o . . o o x . o
   x . x x x . . o
   x o o x . . . o
   . . . . . x o x
   ```

   …becomes this:

   ```
   ? ? ? ? ? ? ? ? ? ?
   ? x x o o . x o x ?
   ? . . . . . o o x ?
   ? . . . . . . o x ?
   ? o . . o o x o o ?
   ? o . . o o x . o ?
   ? x . x x x . . o ?
   ? x o o x . . . o ?
   ? . . . . . x o x ?
   ? ? ? ? ? ? ? ? ? ?
   ```

   Why is this useful? Well, now, if you look in any direction and keep moving, you'll hit a ? before you go off the board. So you don't have to check differently for horizontal and vertical movement.

   If it still isn't hitting you why this is a good idea, I offer another hint.

2) Hint the second: `directions = [-11, -10, -9, -1, 1, 9, 10, 11]`

   Are you picking up what I'm laying down?

In past years, I actually required students to store the board this way, with the question marks border. The Othello server has a different standard, though. As a result, I've written this assignment to stick to the standard, and that means that following my two pieces of advice necessitates extra steps – you have to convert the board as it comes in, and you have to convert each index back to where it would be in the 64-character string version once you find it. In my experience, figuring out these two extra steps + one for loop is easier than writing and debugging 8 separate for loops. You also *can* implement general code on all 8 directions without adding the border, but that also turns out to be hard.

Your mileage may vary! Do your thing; solve it however feels right to you. But hopefully this at least gave you a new idea or two.