Turn-Based AI with Othello: Adding AI

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

The time has come to add AI to Othello and see how well your code can do!

Here's how this works. Your code **is not running the whole game**. There is moderator code that runs the game, and runs two different strategies to get the moves for the two players. Each move occurs when the moderator code runs one player's strategy, sending it the current board and token. After a certain number of seconds, the moderator code will **kill the process** running the strategy for that player. It will then take the most recently decided move and use that to play.

There are descriptions on the course website for how to get information from / send moves to the server, and for how to do so with my grading scripts. You'll want to read up on both of them at some point, but for now, regardless of the particular input/output protocol, here's what your code needs to do:

1) Accept a board state when your code is run and make a single move. (Again: get the current board. Make *one move*, and report the index of the move. Your code won't play a whole game from the beginning.)
2) To choose the best move you can in the time given, run an iterative deepening minimax from there. Ie, it will look one move in the future and determine the best move from that information. Then, start over, look two moves into the future, and determine the best move from that information. Etc, etc.
3) When each iterative deepening call reaches its max depth, you'll need to return a score for that board state. As the code won't be able to search all the way to the end, you will need to find a way to score mid-game states based on how good they seem to look. More advice about this is on the next page!
4) Continue the iterative deepening process indefinitely! Your code doesn't need to stop running. It just needs to keep printing/setting the best answer it has been able to find so far, then running the next search, until the moderator code kills the process.
5) Then, on the next turn, your code will be run once again and receive a new board. Each turn is a **separate run of your code**, so no data can be kept between runs.

For BLUE credit:

- You must beat a strategy that does not look into the future, and just tries to take corners, not take spaces next to corners, and otherwise get as many tokens as possible.

For RED credit:

- You must implement alpha-beta pruning.
- You must beat two out of three particular strategies on the Othello server in a game with at least a 2 second time limit, take screenshots, and submit them to me.
- You must also submit your code to me, and it must take 75% of the tokens against a random player across 10 games with a 2-second time limit.

For BLACK credit, you have two options:

- You must find and implement at least two additional strategies that I have not explained, and demonstrate they have improved the performance of your code in some way.
- OR, you must take 90% of the tokens against a random player across 10 games with a 2-second time limit.

Let's think about Othello strategy a little, shall we?

Ok, before you take your minimax / alpha-beta pruning knowledge and start coding, let's talk some basics of Othello strategy.

In the files you downloaded for this assignment, you should have five strategies, all of which you'll need to utterly destroy, but which have something to teach you first. Make sure all these files are in the same directory:

- `run_othello_game_import.py` (a script to run an Othello game between two strategies)
- `oth_random_import.py` (a strategy that plays randomly)
- `oth_greedy_predictable.py` (a deterministic greedy strategy)
- `oth_greedy_unpredictable.py` (a slightly unpredictable greedy strategy)
- `oth_corners_predictable.py` (a deterministic halfway-clever strategy)
- `oth_corners_unpredictable.py` (a slightly unpredictable halfway-clever strategy)
- `othello_imports.py` (the file you wrote and THOROUGHLY tested for part 1)

Take a moment to look around. Note that all of these files import possible_moves and make_move from YOUR othello_imports file. This is so I can give you lots of helpful strategies to test against without giving that content away!

Note also these strategies are all set up according to the same criteria as you need to follow in order to submit to my server, though none of them use minimax or alpha/beta pruning. They take command line arguments representing a board and a token and they print out a choice of move. (More on how you'll need to implement this is coming later.)

Now: open up the game running script. At the top, you should see that it is ready to pit two strategies against each other – the random strategy and the slightly unpredictable greedy strategy. Take a look through this file as well; it has some cool Python code to open up a subprocess to run each strategy on each turn, capture the output, and do a bunch of error checking. You might find it interesting!

Then, before you run the code, make a prediction. We have two strategies here:

- One strategy plays at absolute random.
- One strategy makes the move that gives it the most total tokens after that turn (ie, always trying to maximize how many pieces on the board belong to it.)

Which one of these do you think is going to win?

Run a few games, examine the output, and see if you're right. You should see random win many, if not most, games! This should tell you that being greedy is NOT a good Othello strategy. Why do you think this would be?

And: what **is** a good strategy?

Try running one of the corners strategies against random or against greedy. You should see it do much better! If you examine that code, you'll see that the corners strategies add in two other important checks:

1) If a corner is available, take it
2) If a space next to a corner is available, try to avoid it if possible

Play some games of Othello and think about why those might be good ideas. Then, read on for some even more subtle advice.

The corners strategies are pretty good, but your AI will need to beat them handily.

Advice for a good AI

The minimax/negamax process would already be perfect if we could see all the way to the end, like in Dual Coins, but we can't. So, the quality of your AI here is determined by two things:

1) How far into the future it is able to look
2) How good it is at determining whether any given mid-game state is favorable or unfavorable

For #1, much of the answer is an efficient implementation. You should have tested in part 1 and ensured that your possible moves and move making functions are quite fast. As you read this section, think about making sure your scoring function is fast too. The rest of the answer to looking far into the future is alpha-beta pruning. You might not need to implement this to get BLUE credit, though I do recommend it, but either way **you are required to implement alpha-beta pruning for RED credit** (mostly because you'll have a hard time accomplishing the other RED goals without it!)

For #2, the answer is a little more complicated. Here are the general principles:

- As a recap, the sentence "whoever has the most tokens is probably winning right now" is **absolutely not true**, especially not early in the game. Your explorations with my provided greedy strategies have taught you that!
- Instead, it turns out that early on in the game the most important factor is **mobility**. A good AI for Othello will restrict its opponent's options, so on its own turns it has more available moves than its opponent. A simple measure of (# of available moves for black) – (# of available moves for white), perhaps multiplied by a coefficient, makes a good early game AI. Black strictly prefers more positive values; white, more negative.
- At a certain point, the game will near the corners. **Corners are very important locations because they can't be recaptured**. So, your AI will probably want to add some number to the score for each corner captured by black and subtract for each corner captured by white. (Again, black prefers more positive values, white more negative.)
  - Similarly, the three squares adjacent to each corner are bad locations to move, because they allow the opponent to potentially take a corner; perhaps add that to your mid-game score (more positive if WHITE takes corner-adjacent squares, more negative if BLACK does.)
  - Feel free to extend this logic to other squares that seem like good captures (edges?)
- Near the end of the game, it becomes plausible for your A/B pruning to reach the end of the game. If your search reaches the end of the game, it can definitively score whether that board is victory or defeat! It is **extremely important** that the numerical score representing victory or defeat should be **much** larger in either direction – positive or negative – than any possible score from your mid-game guessing. You never want to be in a situation where, numerically, your code thinks that any estimated score of a mid-game board looks like a better option than a guaranteed victory.
  - For the assignment requiring you to take 75% of the tokens against random, you might also want to modify the victory number by the total amount of tokens taken at the end of the game (ie, victory for black is 1000000 + how many total black tokens – how many total white tokens, something like that) so that your code chooses the more overwhelming victories when given the chance.

You need to decide how to numerically weight all of these ideas – mobility, value of certain spaces, etc – against each other. Feel free to experiment and submit to the Othello server as many times as you like!

With that scoring function written, your code will then look as far into the future as it can, score the boards at that depth and run minimax (with alpha-beta pruning) on those scores, and choose what seems like the best move now accordingly. Then, try again at the next deeper depth value, then the next, then the next, until it runs out of time and the process is killed.

How should you work on this assignment?

Ok: you have a lot of thoughts about Othello strategies now. It's time to begin working.

This is a little weird, though, because submitting to my grader and to the Othello website use **different specifications**, though it is possible to put both in the same file. See the course website for details; this sometimes changes from year to year, so I haven't made it a permanent part of this document.

In my experience, many students figure out an Othello strategy, try some tests, submit to the server, and it works great! But: many students have a strategy that doesn't work very well, and it can be hard to narrow down the problems. So: let's take a look at some resources I've given to help you with that.

How to test that your code is fast enough

Independent of how good your scoring strategy is, you want to make sure it's fast enough first. Then if it doesn't perform very well, you'll know you need to tweak your scoring strategy itself.

For comparison purposes, I've timed my own strategy at a variety of depths, with AB pruning enabled and with AB pruning disabled. Please note that, as always, I do not expect you to be as fast as me! But: you probably shouldn't be too far off either. If you're taking more than 5 or 10 times as long to do this stuff as I am, you have a problem.

The boards_timing.txt file provides 8 test cases – two boards near the very start of a game, two early-game boards, two mid-game boards, and two late-game boards. Put this code at the bottom of your strategy file (replacing any submission code for either my scripts or the Othello server) and you can get timing data of your own:

```python
results = []
with open("boards_timing.txt") as f:
    for line in f:
        board, token = line.strip().split()
        temp_list = [board, token]
        print(temp_list)
        for count in range(1, 7):
            print("depth", count)
            start = time.perf_counter()
            find_next_move(board, token, count)
            end = time.perf_counter()
            temp_list.append(str(end - start))
        print(temp_list)
        print()
        results.append(temp_list)

with open("boards_timing_my_results.csv", "w") as g:
    for l in results:
        g.write(", ".join(l) + "\n")
```

This creates a CSV file you can open in Excel or Google Sheets and compare to my numbers.

Each of these charts matches one of the tests in `boards_timing.txt`, in corresponding order. My numbers, in seconds, are as follows. The % ignored uses time to estimate how much of the search tree A/B pruning discards.

start-1

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0004434	0.00211	0.009319	0.0530329	0.2933354	1.9327429	
AB	0.0004555	0.0021675	0.0064556	0.0263963	0.091967	0.3598766	1.0872936
% ignored	-2.73%	-2.73%	30.73%	50.23%	68.65%	81.38%	

start-2

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.000694	0.0031339	0.0177515	0.1001697	0.6553353	4.5028121	
AB	0.0006917	0.0031418	0.0115409	0.0412386	0.1485247	0.6003061	1.8781256
% ignored	0.33%	-0.25%	34.99%	58.83%	77.34%	86.67%	

early-1

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0009099	0.0103703	0.0961815	1.0828144	11.7197591	141.462748	
AB	0.000911	0.0103214	0.0596227	0.6093728	2.5293785	22.6222452	82.4602108
% ignored	-0.12%	0.47%	38.01%	43.72%	78.42%	84.01%	

early-2

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0010445	0.0083778	0.078608	0.8045738	8.542104	96.4425228	
AB	0.0010477	0.0083694	0.0557421	0.3362472	2.675824	13.3344526	98.3158231
% ignored	-0.31%	0.10%	29.09%	58.21%	68.67%	86.17%	

mid-1

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0010989	0.0121244	0.1487649	1.5638876	18.853828	198.649807	
AB	0.0011022	0.0121308	0.0923255	0.5916564	4.2325951	20.8848249	116.701077
% ignored	-0.30%	-0.05%	37.94%	62.17%	77.55%	89.49%	

mid-2

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0009289	0.0109014	0.1138584	1.3648756	13.9805387	163.865347	
AB	0.0009283	0.0109799	0.0395025	0.3257061	1.1943708	11.4147787	27.5765438
% ignored	0.06%	-0.72%	65.31%	76.14%	91.46%	93.03%	

late-1

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0002649	0.0017592	0.008716	0.0365714	0.1218396	0.329625	
AB	0.0002647	0.0017607	0.0052682	0.0159604	0.0385692	0.0866919	0.1859729
% ignored	0.08%	-0.09%	39.56%	56.36%	68.34%	73.70%	

late-2

	depth 1	depth 2	depth 3	depth 4	depth 5	depth 6	depth 7
no-AB	0.0002922	0.0015518	0.0070562	0.0232767	0.0645191	0.1238868	
AB	0.0002974	0.0015826	0.0033954	0.0129064	0.0315794	0.0604555	0.0549608
% ignored	-1.78%	-1.98%	51.88%	44.55%	51.05%	51.20%	

There are a couple ways to use those numbers.

1) Check your raw numbers vs my raw numbers. Your depth 4 time on mid-game board 1 doesn't need to be 1.5 seconds, but it shouldn't be 30.
2) Check your AB implementation by comparing YOUR AB numbers to YOUR non-AB numbers. See if you get similar percentages ignored.

If your code is SLOW here, but it was FAST on detecting and making moves, your scoring process is probably the culprit and you'll need to speed it up.

If your speed is fine, but your code still isn't winning the games it should, then…

Some common-sense testing for your scoring method

Your scoring method should be **symmetric**. That is, if you replace every "o" with an "x" and vice versa, the scoring method should give a **precisely negated score**. In other words, the amount to which a board is good for "o" should be exactly the amount to which a board is bad for "x".

Also make sure you answer these questions:

1) Early in the game, does a higher score mean more mobility?
2) Does possessing a corner give a big bump to the score?
3) Does possessing a corner-adjacent space hurt the score? What if you possess that corner, is there still a penalty then? How should that work?
4) At the end of the game, is a guaranteed victory unquestionably more points than any intermediate state?
5) At the end of the game, is a bigger wipeout valued more highly than a narrow win?

If this doesn't help, the problem might be with efficiency.

Specification for BLUE credit for "Othello AI: Level 1"

All you need to do to get BLUE credit is to reliably, consistently, verifiably beat both corners strategies that came with the assignment. I'll run your code and give it a 2-second time limit against the unpredictable corners strategy 6 times, three times as "x" and three times as "o". If you win all six games, you get credit.

Submit a single Python script to the link on the course webpage.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code meets the specification on the course website for submitting to **my grading script**.
- Your code performs as expected, detailed above.

Specification for RED credit for “Othello AI: Level 2”

You’ll need to do three things here.

1) Implement Alpha/Beta pruning in your code, and put a comment that says "ALPHA/BETA PRUNING HERE" so I can search for it and verify that you’ve done so. (As a hint, if you run the timing script and you don’t see a huge improvement with Alpha/Beta pruning, you probably aren’t right yet.)
2) There are three canonical strategies on the Othello server. Two, pw1 and pw2, were written by a former teacher, Dr. Patrick White. mceckel_canonical was written by me, and is the same strategy I’ve referenced a few times in this assignment. mceckel_canonical beats both pw1 and pw2, but that doesn’t mean that you’ll find it harder to beat; Othello strategies are not necessarily transitive. Your goal is to beat **any two of these strategies**. You only have to beat them once and you can start as either white or black. You must **disable time hoarding** and use a time limit of **between 2 and 5 seconds**. With each victory, take a screenshot that includes the bottom part of the screen showing that you’ve done those two things, and that the username is yours.
3) You’ll need to submit your script to me, and it will need to take more than 75% of the total tokens against random across 8 games, 4 each as “x” and “o”. I’ll give you a 2 second time limit. For this, practice several games against random on your own computer using the provided random strategy and a 2 second time limit. When you’re satisfied this will work out, and you’ve got the two above screenshots, you’re ready to submit.

Submit **three files** – a Python script and two screenshots in .jpg, .bmp, or .png format – to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code meets the specification on the course website for submitting to **my grading script**.
- Your code performs as expected, detailed above.
- Your screenshots clearly show a victory, your username, time hoarding disabled, and a time limit between 2 and 5 seconds.

Specification for BLACK credit for “Othello AI: Level 3”

You have **two options** here (unlike RED credit, you only need to do **one** of these things, not both.)

1) When you submit for RED credit, if your code takes clears 90% against random instead of just clearing 75%, you’ll automatically get BLACK credit as well. This does not require a separate submission.
2) Alternately, you may add more advanced strategies to your script. Implement **two or more of the following strategies**, and show improvement vs some other strategy, which could be one of the three canonical strategies given in RED or another student’s strategy. I won’t be helping you find additional strategies, but there are an insane number of Othello strategy papers and resources online. Some ideas to explore include:
    a. Negascout
    b. An opening book that pre-stores decisions for the first few moves of the game
    c. (Possibly in conjunction with b to save time early on) using the Time Hoarding feature on the Othello tournament site
    d. Some measure of edge / piece stability
    e. Bitboards
    f. Students in a previous year recommended “transposition tables” and “MTD(f)” – I don’t know what either of these things are, but I trust their judgment
    g. There’s also a technique called “quiescent search”, where you preference your search on some states over others if you suspect that a big move is just about to happen but hasn't actually been searched yet. This requires some sophisticated reasoning and developing your own quiescent search heuristic, but students who go deep on this have reported that it can outperform negascout; worth a try if you really want a stretch.

If you’re going for #1, you don’t need to submit again. For #2, submit your code **and** a document briefly explaining what you did and what improvement you achieved, following the usual naming guidelines.