Programming Assignment 2: Othello

Due: Friday 11/07/2025 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. We will not be using Sepia for this assignment: I have developed a game engine for us to use. In this assignment we will be writing agents to play a Othello (also called Reversi).

Othello is a really simple game that is played on an 8×8 board. There are two players; one player placing black pieces on the board and another placing white pieces on the board. Players alternate turns starting with the person controlling the black pieces. A player is only allowed to place pieces on the board in specific squares when it is their turn. A square on the board is considered a "legal" move for player A iff:

1. By placing their piece in that square, a contiguous line of opponent pieces would be "sandwiched" (the formal term for this is called "outflanked") between two pieces controlled by player A along a row.
2. By placing their piece in that square, a contiguous line of opponent pieces would be "sandwiched" (the formal term for this is called "outflanked") between two pieces controlled by player A along a column.
3. By placing their piece in that square, a contiguous line of opponent pieces would be "sandwiched" (the formal term for this is called "outflanked") between two pieces controlled by player A along a diagonal.

The important part of this logic is that the by placing a piece in that square, player A is creating a sandwiching event (e.g. those contiguous line of opponent pieces were **not** sandwiched before player A placed their piece in that square). If player A chooses to place their piece in that square, they "capture" all of the sandwiched pieces who now flip from being pieces controlled by the opponent and now become pieces controlled by player A (e.g. all black pieces flip to white and vice versa). Then the opponent gets to make a move, and the game progresses.

Note that it is possible that there are no legal moves available to a player at a given time. When it is a player's turn to make a move and there are no legal moves to make, that player's turn is skipped. Formally the game is over when neither player has any legal moves remaining.

The difference between Othello and Reversi (yes there is a difference) is the initialization of the board. The game of Othello follows the same rules as Reversi, however the game always begins with the middle-four squares of the board populated as shown in Figure 1:

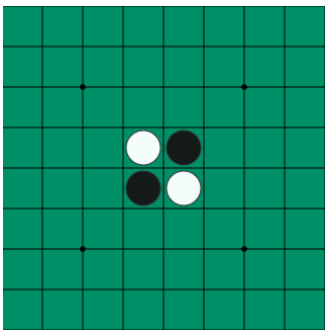

Figure 1: The initial state of Othello (this is not a rendering of our Othello).

Reversi on the other hand comes in many flavors: all of which have different initializations. We will be using Othello instead of Reversi for this reason.

1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/pa2/lib/othello-X.X.X.jar` to `cs440/lib/othello-X.X.X.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/pa2/lib/argparse4j-0.9.0.jar` to `cs440/lib/argparse4j-0.9.0.jar`.
  This is a jarfile that `othello-X.X.X.jar` depends on. It provides similar functionality to Python's `argparse` module. The documentation for `argparse4j` can be found here.
- Copy `Downloads/pa2/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/pa2/othello.srcs` to `cs440/othello.srcs`.
  This file contains the paths to the `.java` files we are working with in this assignment. Just like in the past, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/pa2/doc/pas` to `cs440/doc/pas`. This is the documentation generated from `othello-X.X.X.jar` and will be extremely useful in this assignment. After copying, if you double-click on `cs440/doc/pas/othello/index.html`, the documentation should open in your browser.

2. Test run

If your setup is correct, you should be able to compile and execute the following code. A window should appear:

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @othello.srcs
java -cp "./lib/*:." edu.bu.pas.othello.Main

# Windows. Run from the cs440 directory.
javac -cp "./lib/*;." @othello.srcs
java -cp "./lib/*;." edu.bu.pas.othello.Main
```

NOTE: The commands above will **not** run your code. There are several command line arguments you can provide, and one of them is to specifiy which agents control which color tiles. To specify agents that control the black tiles and the white tiles you would need to do something like this:

```
java -cp "./lib/*:." edu.bu.pas.othello.Main -b <BLACK_AGENT> -w <WHITE_AGENT>
```

Your agent's classpath is `src.pas.othello.agents.OthelloAgent`. You can alternatively use `--blackAgent` and `--whiteAgent` instead of `-b` and `-w` if you wish. Remember that if you want to see the other command line arguments available you need to add a `-h` or `--help` to the end of your command!

When you run our Othello with rendering on (remember to use `-s` for silent execution), I have chosen to render two extra pieces of information. Squares colored yellow are the set of legal squares that player can choose from. A green dot will be placed at the center of the most recently placed piece by that player. Since tile colors swap, it is possible for there to be two black tiles with green centers: this just means that the most recent tile placed by the white agent was flipped! Depending on how helpful you find the visualization, I may push more changes in the future (for instance the visualization does **not** show whose turn it currently is: you have to infer that information from the board).

Task 3: The `Node` Type (30 points)

You will be solving this game with deterministic adversarial search, which is a form of tree search. Step one is to implement the tree. I have provided an abstract class `edu.bu.pas.othello.traversal.Node` which you will extend. I have provided a class called `OthelloNode` as a public nested class within `OthelloAgent` inside `src/pas/othello/agents/OthelloAgent.java`. The abstract methods you need to override are:

1. `public double getTermialUtility()` This method is where you will calculate the utility value of a `Node` when it is terminal. I would recommend making this a symmetric interval $[-c, +c]$ where a utility of 0 is when you tie (the terminal node is $+c$ when you win in the best possible way and $-c$ if you lose in the worst possible way). Remember that whatever decisions you make here need to be respected when you make heuristics (e.g. heuristic values cannot go outside this interval and should also play by the same paradigm).

2. `public List<Node> getChildren()` This method is where you will calculate the children of a `Node`. If this `Node` is terminal then there are no children by definition, but things are harder when the `Node` is nonterminal. Note that the set of legal moves is already precalculated for you (called the "frontier" of coordinates), so you don't need to worry about calculating what moves are available. Instead, you will need to worry about how to deal with the scenario when the player who is supposed to move has **no** available moves to them. What should their children be?

I highly encourage you to write your own tests for this. You are welcome to write your own unit tests where you instantiate a `Game` object, then set the board to whatever configuration you want, then create a `OthelloNode` and check that it functions the way you expect.

Task 4: `Heuristics.java` (30 points)

Just like in lab, you will need to implement heuristics because our tree search cannot expand the entire tree. The file `src/pas/othello/heuristics/Heuristics.java` is where all of your heuristic functionality should go. There is one method already there and that is the entry point of your heuristics. When it is time for me to test your code, I will be using that entry point, so be sure that:

1. You don't change the signature of this static method.

2. Any heuristic functionality you want to include needs to be callable from that entry point.

If your agent uses other heuristic functions thats ok! They just need to be called as a part of this entry point, so if your agent calls them directly, then my testing code **will not** and you will probably fail some tests.

Task 5: `MoveOrderer.java` and `OthelloAgent.java` (40 points)

Just like in lab, you will likely need to choose an order that your tree search enumerates the children of `Node`s. The file `src/pas/othello/ordering/MoveOrderer.java` is where any move ordering logic should go. Note that you also need to respect the entry point (which is already there) just like you do for heuristics. There are tree search flavors that do not use move ordering and you are welcome to pick them. If you do and choose not to implement move ordering, it is ok to leave this file unmodified. Be careful that your algorithm runs fast enough though!

The meat of this assignment is your `src/pas/othello/agents/OthelloAgent.java`. There are a few methods that you need to implement in order for your agent to work (and be testable by my code). The methods you need to implement are as follows:

1. `public Node makeRootNode(final GameView game)`. As mentioned previously, this agent contains a class called `OthelloNode` inside it. You are welcome to do whatever you want to this class as long as it obeys the api from `edu.bu.pas.othello.traversal.Node`. In order to get the root node of a game tree, your code should call (and I will call) the `makeRootNode(final GameView game)` method. This method's purpose is to create the root node of a game tree. I will call this method when testing your code to instantiate `OthelloNode` objects (and then call `getChildren()` on them in the future.

2. `public Node treeSearch(Node n)`. The purpose of this method is for you to implement whatever tree search algorithm you are picking for your agent. This could be minimax, alpha-beta pruning, quiescent search, etc. or some combination of them all. The important part is that this method returns the correct `Node` object just like in lab. You are allowed to add whatever fields/methods you need in order to implement your search algorithm but `treeSearch(Node n)` is (at least) the entry point into your tree search algorithm. I will call this algorithm to test your implementation: I will check that the move contained within the returned `Node` is legal, and I will also check the utility value of the returned `Node`.

Returning `null` Moves and Illegal Moves

Be warned. If you choose to return an `OthelloNode` with a `null` move from your `treeSearch` method (and therefore return `null` from your `chooseCoordinateToPlaceTile` method), the game will interpret that `null` in one of two ways:

1. If the agent has no legal moves available, the game will accept the `null` move since the agent cannot make a move that turn. Nothing will happen.

2. If the agent **has** at least one legal move available, then the game will count this `null` move as a strike against the agent. If an agent accumulates three strikes, that agent auto-loses the game.

Also note that if your agent submits an illegal move to the game, then this will also count as a strike against that agent. To summarize: the only time `null` is an acceptable move is if there are no legal moves available, otherwise only one of the currently-legal moves must be chosen. This will be especially applicable when you are creating children of `OthelloNode`s.

Task 5: Extra Credit (50 points)

In order to earn the full extra credit for this assignment, you will have to demonstrate (on the autograder) that your agent can win against our agents. We will release EASY, MEDIUM, HARD, and (maybe) INSANE agents. Both Collin and I are working on opponents for you to test yourselves against. Here is how you will earn extra credit for this assignment:

1. When competing against the EASY agent on the autograder, each game will take a maximum of 5 min meaning that each agent will have at most 4687ms per move (e.g. 4.6s per move) to think. You will face off against the EASY agent 6 times on the autograder, and you must win at least 5 of those games to earn 5 bonus points.

2. When competing against the MEDIUM agent on the autograder, each game will take a maximum of 5 min meaning that each agent will have at most 4687ms per move (e.g. 4.6s per move) to think. You will face off against the MEDIUM agent 6 times on the autograder, and you must win at least 4 of those games to earn 10 bonus points. If we do not release the INSANE agent this will be worth 20 bonus points.

3. When competing against the HARD agent on the autograder, each game will take a maximum of 6 min meaning that each agent will have at most 5625ms per move (e.g. 5.6s per move) to think. You will face off against the HARD agent 6 times on the autograder, and you must win at least 2 of those games to earn 15 bonus points. If we do not release the INSANE agent this will be worth 25 bonus points instead of 15.

4. When competing against the INSANE agent on the autograder, each game will take a maximum of 6 min meaning that each agent will have at most 5625ms per move (e.g. 5.6s per move) to think. You will face off against the INSANE agent 6 times on the autograder, and you must win at least 1 of those games to earn 20 bonus points. We may not release the INSANE agent though, it depends on some other factors.

Task 6: Submitting Your Assignment

Please drag and drop all of the files your modified on gradescope. There will be multiple autograders, so be sure to submit to them all!

Task 7: Tournament Eligibility

In order for your submission to be eligible in the tournament, your submission must satisfy all of the following requirements:

- Your submission must be on time.
- You do not get an extension for this assignment.
- Your agent compiles on the autograder.
- Your agent can beat the MEDIUM agent more than 7 out of 10 trials in timed (10min) games. I will run these off-camera.