HW2—Game Engine

Introduction: Konane

Also known as Hawaiian Checkers, Konane is a strategy game played between two players. Players alternate taking turns, capturing their opponent's pieces by jumping their own pieces over them (if you're familiar with checkers, there is a strong structural analogy to be made here, except the jumping is not diagonal but orthogonal, and while multiple jumps are allowed in a turn, all jumps have to occur in the same direction). The first player to be unable to capture any of their opponent's pieces loses.

The full rules can be read *here* or *here*, and *here's* a nice video explaining the rules simply as well.

Here's a (rather terse) version of the rules, though:

1. Black typically starts. They take one of their pieces off the board. Now, the piece shown below as taken off is actually not the one that's taken off. If we imagine a *(row, column)* coordinate system for the pieces such that the top-left white piece is in position (1, 1), and the top-right black piece is in position (1, 8), then the very first move of the game sees a black piece taken off from the two right in the middle of the board—so, (4, 5) and (5, 4)—or from such a pair in any of the corners of the board. Please make sure your implementation honors that rule.
2. White then takes one of their pieces off the board from a space *orthogonally* adjacent to the piece that black removed.
3. Each player then alternately moves their pieces in capturing moves. A capturing move has a stone move in an orthogonal direction, hopping over an opponent's piece. Multiple captures may be made in a turn, as long as the stone moves in the same direction and captures at least one piece.
4. The first player to be unable to capture a piece loses. :(

Play the game

In this homework, you'll be implementing Minimax and Alpha-Beta Pruning for an agent playing Konane. Wait, isn't Alpha-Beta Pruning a variant of Minimax? Yes, it is. **It's just that here we're using Minimax and Alpha-Beta Pruning to respectively refer to Minimax WITHOUT Alpha-Beta Pruning and Minimax WITH Alpha-Beta Pruning.**

But first, you should get *practically* familiar with how the game is played, not just be familiar with the rules of the game. To do this, play the game with the provided code. You've been distributed a codebase which includes an interface for playing the game in a variety of modes. **Notably, you don't need to actually *make* the game of Konane—just to make an agent that plays it.**

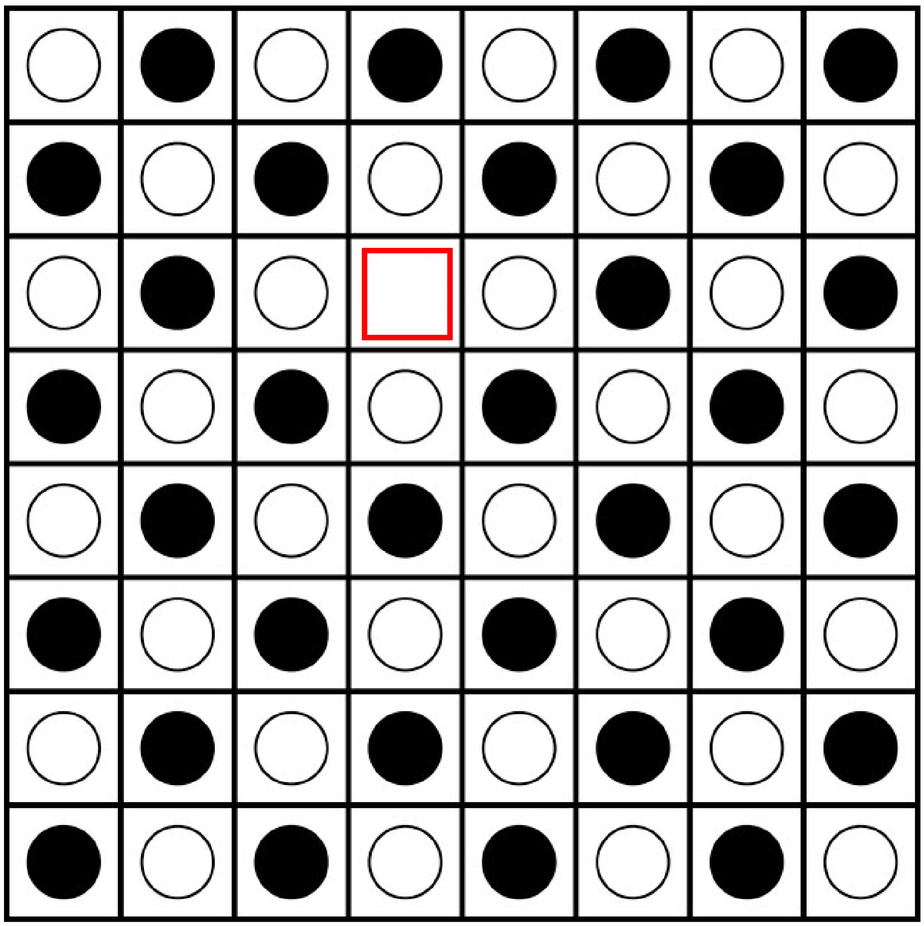

Figure 1: Konane Board

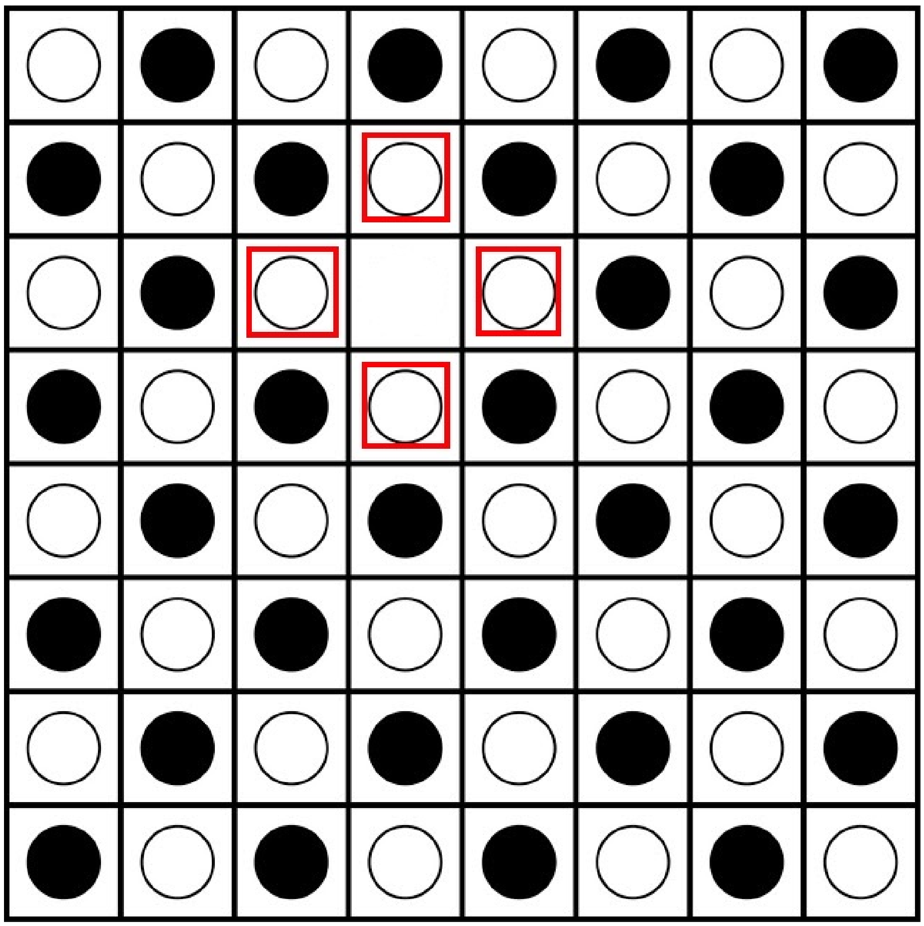

Figure 2: Konane Board

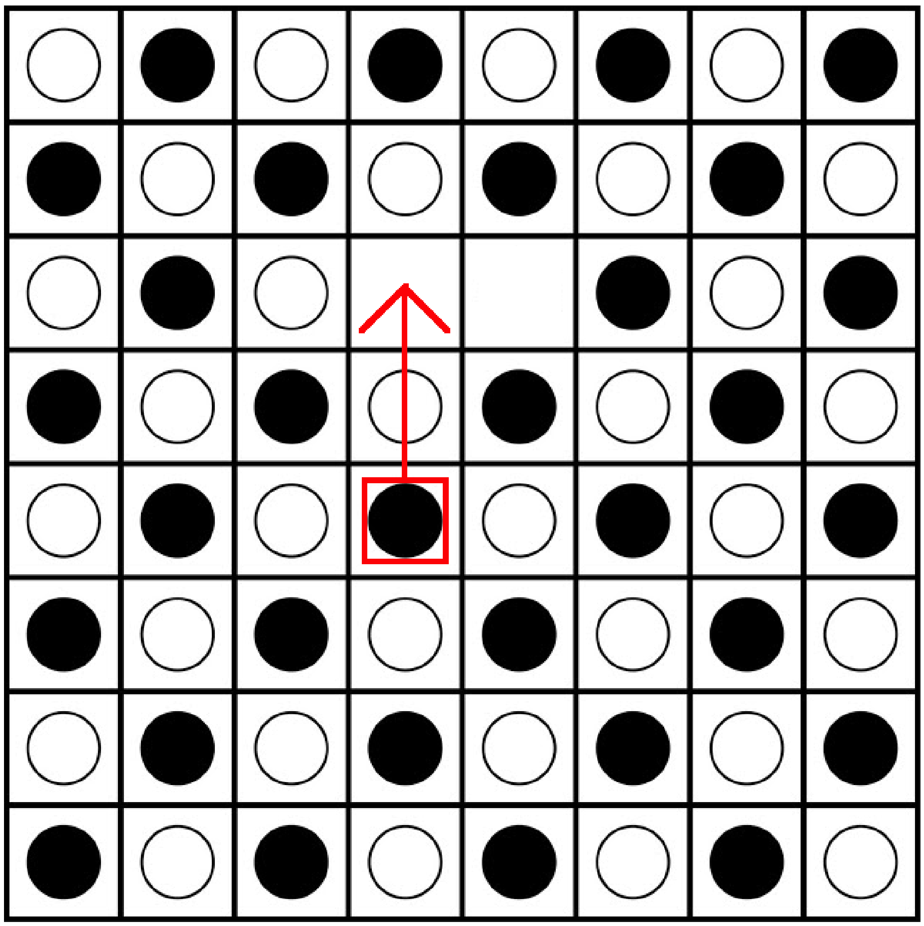

Figure 3: Konane Board

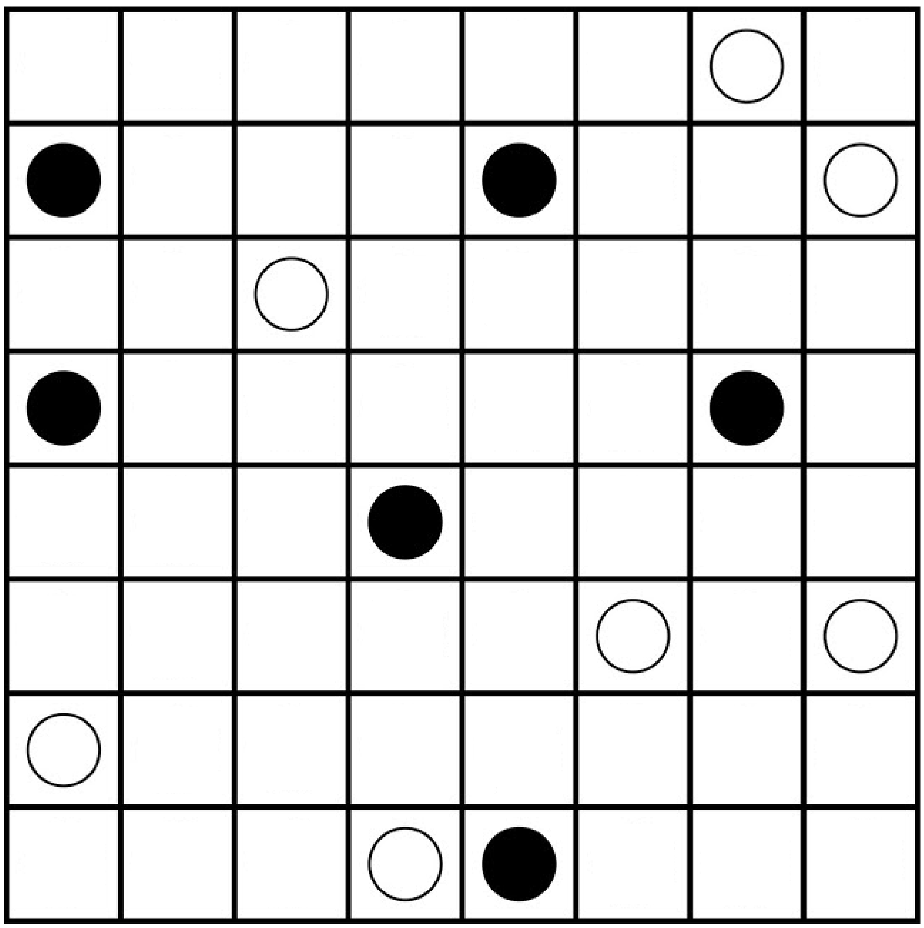

Figure 4: Konane Board

Playing the game interactively may only work on Linux and Mac. However, this is not required for completing the assignment and is only provided to help you get familiar with the game (and for your entertainment).

Run the following from your terminal:

```
python main.py $P1 $P2
```

By default, `main.py` will setup a human player versus a random player on a board that is 10x10. During **Human** mode, move the cursor with the ARROW keys and select the tile with SPACE. When it is a computer's turn, advance the game with the SPACE key. To see the game board in your terminal, you need a minimum terminal size of (rows + 2) x (columns + 2) to see the whole board. To exit the game, kill the process in your terminal (e.g., with CTRL-c).

You can change the game settings by passing in values to `python main.py`. You need to pass in *exactly* two arguments. Valid arguments are as follows:

- H (Human)—manually select the tile to move and to where you will move it. Legal moves will be executed.
- D (Deterministic)—the agent will select the first move that it finds (the leftmost option in the tree) during its traversal.
- R (Random)—the agent will pick a random move.
- M (Minimax)—the agent will pick a move using the Minimax algorithm. You will be prompted for a maximum search depth.
- A (Alpha-Beta pruning)—the agent will pick a move using A-B pruning. You will be prompted for a maximum search depth.

Passing in an invalid number or type of arguments will result in the system defaulting to a human vs. a random player.

Your task

Now that you know how the game is played, it is time to make your own intelligent players of the game. You will do this my implementing one player that use Minimax and another player that uses Alpha-Beta Pruning.

For this homework, make sure that you are running Python 3.6 - 3.7. These versions ensure that the legal move ordering is the same as what is expected by the tests. Programming is hard. :(

Part 1: Minimax

Minimax is an algorithm for determing the best move in an adverserial game. It seeks to minimize the maximum loss posed by the opponent's strategy. Minimax is typically employed in competitive, discrete-, and finite-space games with abstracted time and perfect information.

You will complete the implementation of `MinimaxPlayer` in `player.py`. In your implementation, you need to be aware of 2 things: the maximum depth and the evaluation function. The maximum depth is provided to the constructor of the `MinimaxPlayer` and defines the maximum number of plies that the player will simulate when choosing a move. The evaluation function defines a score for a terminal node in the search. Use the function `h1` defined in the parent class `Player` as your evaluation function.

Please leave the `selectInitialX` and `selectInitialO` methods alone; all of the editing that you need to do takes place in `getMove`. As always, feel free to add any methods/classes you feel that you need, provided that you change only `player.py`.

Part 2: Alpha-Beta Pruning

You may notice that Minimax starts to get terribly slow when you set your maximum search depth to values above, say, 4. This makes perfect sense when you think about the fact that the total number of nodes in your game tree is the branching factor to the power of the search depth. For comparatively "bushy" games (e.g., *chess*, *Go*, etc.) the branching factor is prohibitively large, which is why agents that play these games use cleverer algorithms to choose what move to take next.

One such cleverer algorithm (although still not clever enough to do well at games like *Go*) is a modification of Minimax known as *Alpha-Beta Pruning*. They are, at their core, the *same algorithm*. The distinction is that A-B Pruning *ignores* subtrees that are provably worse than any that it has considered so far. This drastically reduces the runtime of the algorithm.* Since A-B Pruning is a variant of Minimax, you aren't really writing a new algorithm; rather, you're taking your implementation of Minimax and making it a little smarter.

* Strictly speaking, it doesn't change the upper bound on the algorithm's runtime, since in the worst-case one must still search the entire tree. In practice, however, the performance difference is very noticeable.

As with Minimax, your task is to complete the implementation of `AlphaBetaPlayer`. You will need to again consider the maximum depth and the evaluation function.

Testing Your Work

You can manually test your work by playing against your agent yourself, or by having the agents play against each other. We've also included a few tests for kicking the tires on your implementations of Minimax and Alpha-Beta Pruning. You can find those tests in `test.py`, and you can run them with:

```
python test.py
```

In designing your own tests, consider different board sizes (always square), depths for searching, and time to execute. The timeouts provided in `test.py` should be generous, so see if you can do much better. It is worth noting that the tests can take upwards of five minutes to complete, so don't freak out. :)

To help with debugging and generating new tests, each play of the game automatically records all moves to `game.log`. The log can be helpful for comparing the sequence of moves in your implementation and in the implementation of others. For example, we provide a `game1.log` that is the correct sequence of moves for `test1`.

The logs can also be used to create specific test scenarios. The logs can be used to play the game. For examples, `test1s` is identical to `test1` except it is completely a scripted replay using `game1.log`. You can truncate existing log files to automatically replay a part of a game, perhaps advancing the game to where you have a possible bug. Alternatively, you can create your own log files manually, creating completely new scenarios to test.

Notes

On the codebase:

- `player.py`—this is the file you'll be editing. Note that `MinimaxPlayer` and `AlphaBetaPlayer` are both diked out and replaced with a determinstic player instead.
- `main.py`—to play the game (in Human mode) or to watch your agents duke it out, run `python main.py`. Use the arrow keys and the spacebar to select your actions.
- `test.py`—run tests with `python test.py`.
- `game_manager.py`—holds the board representation and handles turn-taking.
- `game_rules.py`—code determining available moves, their legality, etc.
- You can change the type of player, the board size, etc. in `main.py`

On A-B Pruning:

- It's worth noting that Alpha-Beta Pruning produces answers that look more or less the same as vanilla Minimax (they should be identical, given that your search pattern hasn't changed), but Alpha-Beta will run substantially faster. The grading rig will use timeouts in its tests, so ordinary Minimax won't be fast enough to get you full credit for this part of the homework.
- To see the difference between Minimax and Alpha-Beta, just run the game at progressively deeper search depths. You won't see much of a difference at a depth of 2, but the difference between the two at depth 5 is extreme.

On additional fun:

- Try out a better evaluation function. Define an `h2`, and see how it does. Can it do better than the `h1` evaluation function? Note that we will use `h1` for grading, so be sure to have your Minimax and Alpha-Beta players set up to use `h1` in your final submission.

- Can you beat Alpha-Beta? Use `main.py` to play against the computer and see if you can win.