Assignment 4—Game Search

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). The first player to be unable to capture any of their opponent's pieces loses.

The full rules can be read *here* or *here*. Here's my (rather terse) version, though:

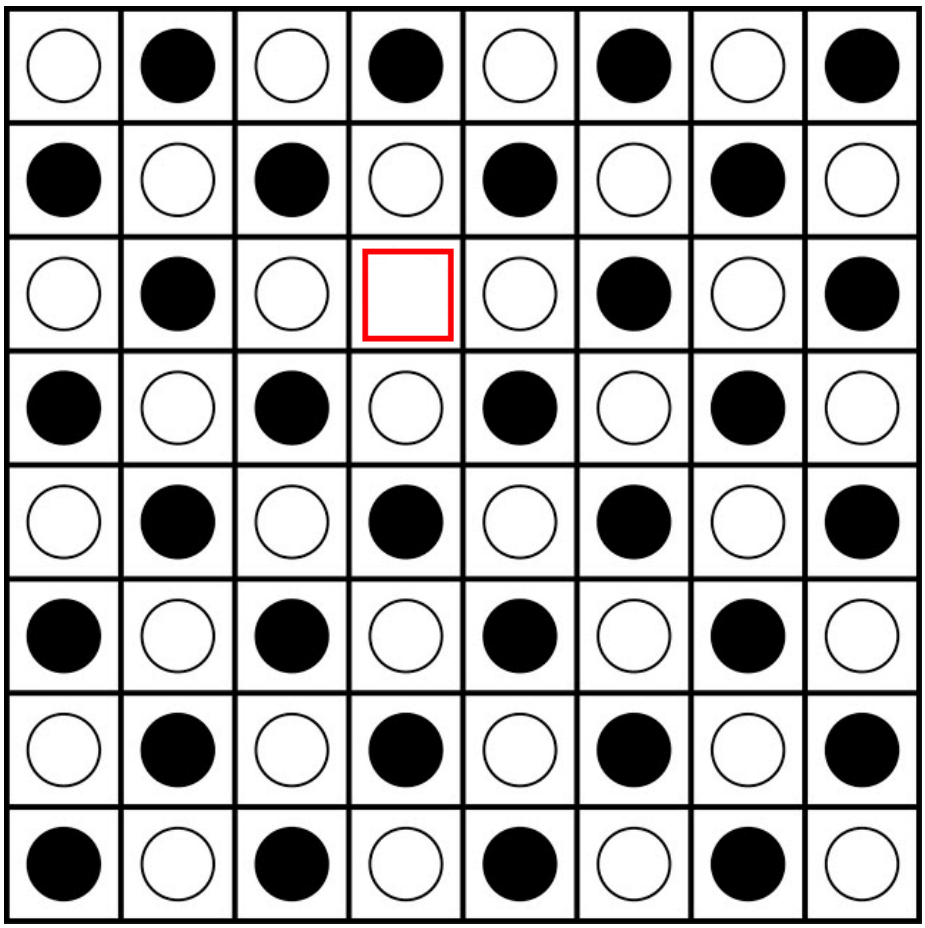

Figure 1: Konane Board

1. Black typically starts. They take one of their pieces off of the board.

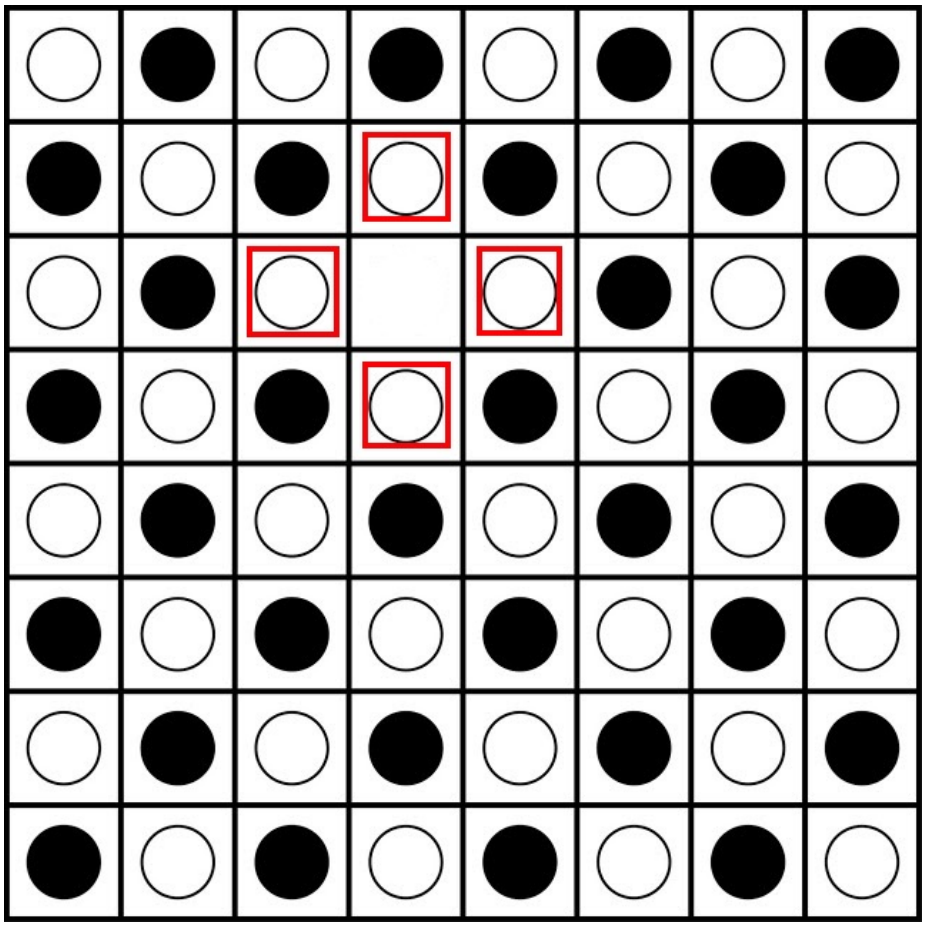

Figure 2: Konane Board

2. White then takes one of their pieces off of the board from a space *orthogonally* adjacent to the piece that black removed.

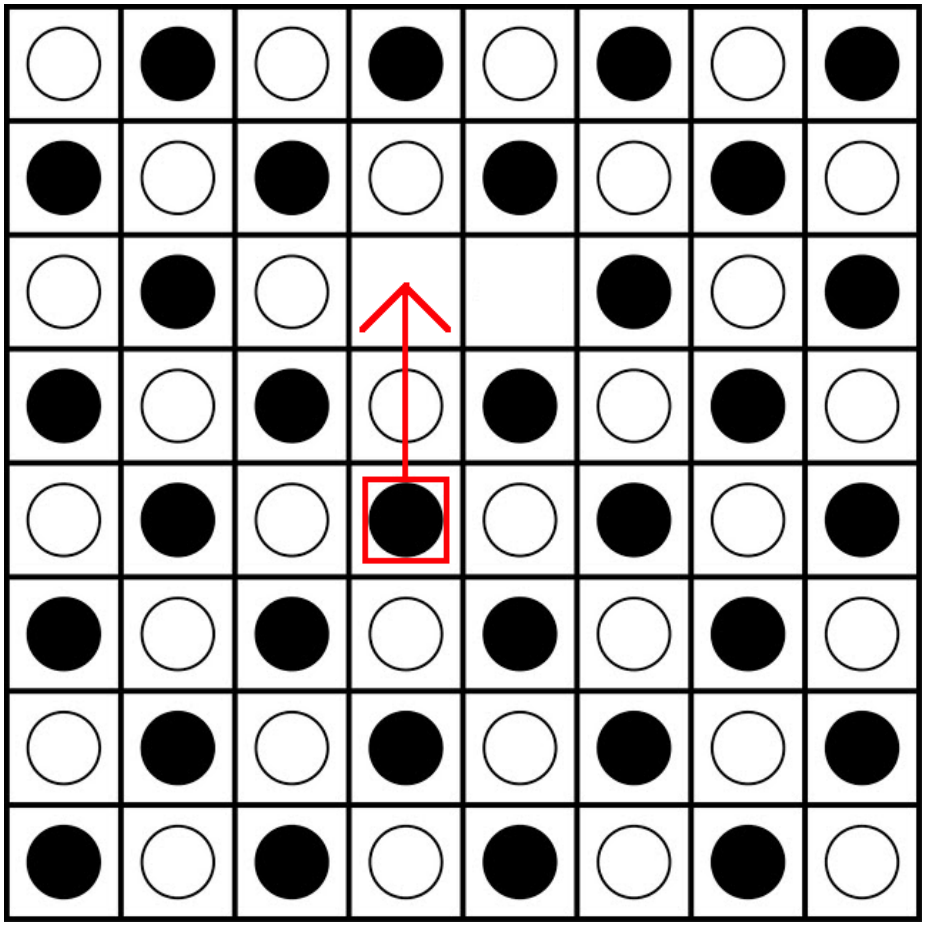

Figure 3: Konane Board

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 assignment, you'll be implementing minimax and alpha-beta pruning for an agent playing one such game—that of konane. But first, you should

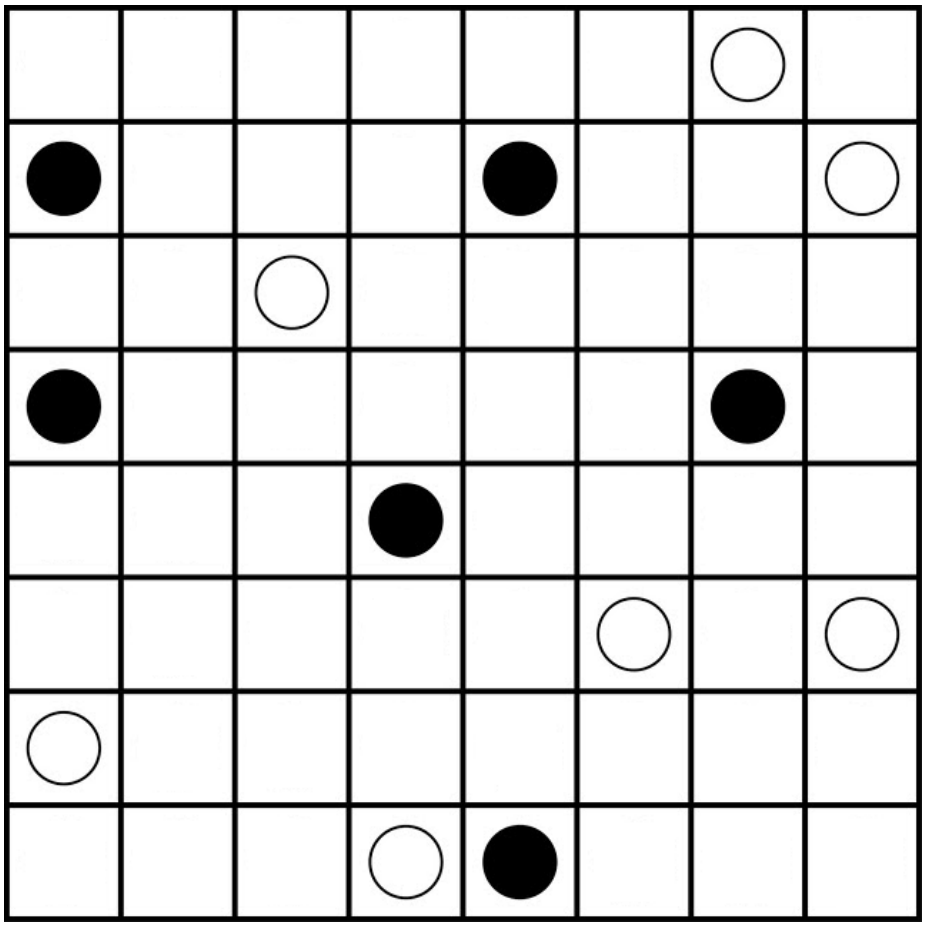

Figure 4: Konane Board

get familiar with how the game is played. 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.

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 max. search depth.
- A (Alpha-beta pruning)—the agent will pick a move using AB pruning. You will be prompted for a max. 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 assignment, make sure that you are running Python 3.6>. We don't know *why* it matters, but the test results vary based on the version. Programming is hard. :(

Part 1: Minimax

Minimax is an algorithm for determing the best move in an adverserial game. Its objective is to find the move that *maximizes* the gain for the player while *minimizing* their loss. Since "maximin" sounds kind of dumb, we get "minimax."

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 max depth and the evaluation function. The max 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 AB pruning *ignores* subtrees that are provably worse than any that it has considered so far. This drastically reduces the runtime of the algorithm.* Since AB pruning is a variant on 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 max 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. :)

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 AB pruning:

- It's worth noting that alpha-beta produces answers which 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 assignment.
- 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 AlphaBeta players setup to use `h1` in your final submission.
- Can you beat AlphaBeta? Use `main.py` to play against the computer and see if you can win.