CS 345: Konane Two Person Minimax Game
8 March 2021 (Revised)

Your job

You will write a Python3 method that plays one side of a Konane game. Your method is inside of a python module.

You have a starter module called `player.py` which has code illustrating two variants: a) pick a random move, or b) do a single max-layer of minmax.

Copy `player.py` to a file *with your own initials* for turning in. For example, Michael Seth Glass would copy the player module to `msg.py` and that is where he would do his programming and that is what he would turn in.

You should turn in *only* the player-module with your code in it. It should run with the unmodified versions of the other python files.

How to run the game

The main program is `konanemain.py`, which you run from command-line. When you run this program, you specify one or two modules on the command line. There is also module called `you.py` which is for the human player. If you specify only one module the other is the human by default.

Here are examples of running the main program. (Assume `$` is your command line prompt.)

```
$ python3 konanemain.py player.py      #  player.py O vs. human X
$ python3 konanemain.py msg.py msg.py  #  msg.py O vs. msg.py X
$ python3 konanemain.py you.py msg.py  #  human O vs. msg.py X
$ python3 konanemain.py msg.py player.py  #  msg vs. player
```

Note: You can also run the program from the Idle Python IDE. Idle by default in the the Windows Python3 download.

- Open `konanemain.py` in the Idle editor
- Pick Run Customized from the menu at top. (Or shift-F5)
- In the box that pops up put the arguments, the names of one or two player modules. They are separated by spaces if you have two of them.
- Click OK, and Idle will start a Python shell window with your running program.

What to implement

You will need to implement minimax with alpha-beta cutoffs, and a scoring function.

The code for the mechanics of playing the game and building the game trees has been constructed for you, most of it is in `konaneutils.py` which you import.

The skeleton `player.py` contains:

- A class called `Konane`. The main program will create an object (one instance) of this class for each player. The instance variables of the Konane class – the variables your playing program will use – are described below.

  (Don't worry that each player has a class with the same name `Konane`, Python can sort them out because they are in different modules.)

- A method `move()`. It returns the best move. You will put your minimax, alpha-beta algorithm in `move` (or you will put it in a method which is called from `move`).

  Return a four-tuple of numbers from 0 to 7 representing a single move: `from_row`, `from_col`, `to_row`, `to_col`. Note in the sample code that you can return the `moved` instance variable from one node of the game tree.

  The example `move()` method has two variants to pick a random move or run the min-max for a single max layer.

- a method `simple_score(board)` which returns a score for the current board from your point of view. It simply counts possible moves for yourself and the opposing player. You may want to write your own scoring function.

The instance variables (inside your player's object) are as follows. You can add more if you need.

- `self.board` A *reference* to the board. YOU DO NOT MODIFY THIS. The actual board is in the main program. The utility routines make *copies* of this board when they play-ahead in the game tree.
- `self.who` This is your player, a single string `'x'` or `'o'`
- `self.other` The other player.
- `self.human` A boolean. Your player is False
- `self.maxdepth` How deep can your tree search go (a number).

Note that you do *not* modify the `board` variable. You have a pointer to the master game board here, modifying it would be the equivalent of turning over the real board and spilling the pieces all over.

The `konaneutils.py` file contains utility functions that you will find useful. The example `move()` and `simple_score()` methods use these, so you can see how they work.

Note that these utilities return `Node` objects, and lists of `Node` objects. Each one of these contains one node of a game tree. The Node object contains:

- `b` a pointer to a copy of the hypothetical future board
- `mover` The player which most recently moved.
- `moved` A four-tuple containing the move in question.

So when it comes time to return your final answer from `move()` method, you need to extract and return the `moved` variable from the node of the game tree that you selected as the best one.

The following subroutines need the current game board passed to them to return information:

- `gameDone` tells whether the game is over for a particular player
- `genmoves` generates all possible moves
- `make_succ` makes successor nodes
- `moveable` says whether a particular piece is movable

The following subroutines are generic, not specific to the current state of play:

- `each_players_places` the squares on the board that belong to each player
- `jumppath` from a proposed x,y starting position to an x,y ending position, return the squares that are landed on and the squares that are jumped over.
- `dests_from` from an x,y starting position, all possible jump destinations

0.1 The game of Konane

You can easily find descriptions of Konane. It is popular in Hawaii, because it comes from Hawaiian culture. Children learn it in school. In this lab, it is played as follows. An 8×8 board is populated with **x** and **o** symbols in a checkerboard pattern. The rows of the board are designated 0–7 and the columns are designated **a**–**f**. Two adjacent symbols are removed before the game starts. The driver program removes the same two starting symbols at the start of every game (the players do not pick these). The initial board looks like this:

```
  a b c d e f g h
0 x o x o x o x o
1 o x o x o x o x
2 x o x o x o x o
3 o x o     x o x
4 x o x o x o x o
5 o x o x o x o x
6 x o x o x o x o
7 o x o x o x o x
```

The **x** player (the human, in our driver) moves first. A single move jumps over one or more opponent squares either vertically or horizontally into an empty square. Jumped-over pieces are removed. One move does not jump both horizontally and vertically. A move is specified by its beginning and ending coordinates. In the above starting board, the **x** player can move **3b**-**3d**, **1d**-**3d**, or **5d**-**3d**.

Thus in the following hypothetical situation, the **x** player can move (among other choices) **3f**-**3d** or **3f**-**3b**, but not **3f**-**1d**.

```
  a b c d e f g h
0 x o x o x o x o
1 o x o   o x o x
2 x o x o x o x o
3 o   o   o x o x
4 x o x o x o x o
5 o x o x o x o x
6 x o x o x o x o
7 o x o x o x o x
```

This board is represented internally in Python code as a list-of-lists. The fundamental item is a single-character string `'x'` or `'o'` or `' '` (blank). Thus the initial board looks like this in Python:

```
  ---------row 0-----------      ----------row 3-----------
[ ['x', 'o', ..., 'x', 'o'], ... [...'o', ' ', ' ', 'x'...], ...]
    0a   0b        0g   0h            3c   3d   3e   3f
```

More on how to do it (with repetition)

If you haven't seen Python object-oriented programming before, know that the variable `self` is similar to `this` in Java. In Python you *must* use `self` to refer to a class variable.

Your `move()` program uses the minimax algorithm (with cutoffs) to determine the next move. It is up to you to determine how many "plies" (moves ahead) deep you want to play. The `self.maxdepth` variable in the constructor contains this depth.

Usually you pick a fixed number, based on how slowly your program runs. Your program could also dynamically adjust the number. For example, you might try progressive deepening, keeping track of how long the previous search took. Or you might adjust the number of plies based on the number of possible moves from the root position, where a higher branching factor means a shallower tree. Keep the machine's moves under 1 minute—preferably under 15 seconds—when playing on the cslab machines.

All the various techniques for improving minimax that we discussed in class are available to you. For example you can re-order the children of a node according to estimated score in order to increase the likelihood of cutoffs. You might be able to identify killer moves (although I think this is much less true of Konane than of Chess).