Artificial Intelligence Lab B - Adversarial Games

This Lab is worth 60 points. Please follow the Notes for All Labs carefully. The lab should be done in **Python 3**.
This is a pair-optional assignment. If you work as a pair, you and your partner should work together for all portions of the assignment, unless otherwise indicated. You should not split it up and each do half. This lab is a significant amount of work, and will overlap with reading assignments and short discussion assignments. If this is your first upper-level CS class, this assignment is likely to be an adjustment in terms of expectations. Plan accordingly.

I strongly recommend carefully reading the entire lab before you begin.

Your **deliverables** are

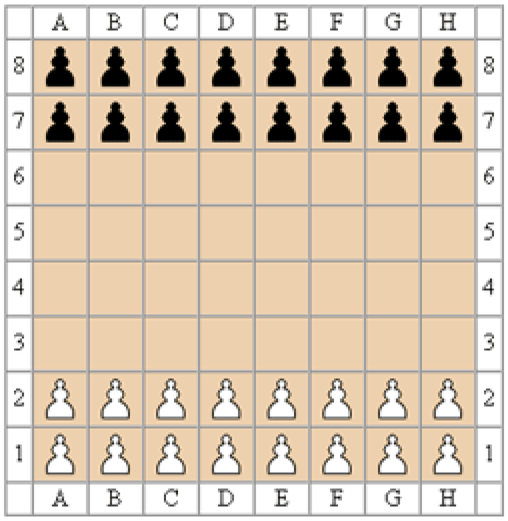
- One or more code files
- A README
- A .pdf report file.
- If you wish to write a script that tests your code, you can also submit that. Indicate how to use it in the README.

This lab has **three** deadlines. Part 1 is due **February 17th.** Part 2 is due **February 23rd**; your second submission should also include your work for Part 1, and you are free to revise your work on Part 1 while working on Part 2. You are strongly encouraged to begin exploring Part 2 before Part 1 is due; it is substantially more complex. Finally, you will submit a finished version on **March 2nd**, which can include revisions to your first parts based on class discussions. This is what will be graded. Note that there is a break in the middle of this period, so you should plan around this. Please let me know if you have any questions

The reason for these intermediate deadlines is to allow class discussion to address tricky spots in the lab. The expectation is that, after these deadlines, you will have explored the problem set in enough detail for class discussions of the lab to be useful to you.

The problems in this lab all involve a competitive game called Breakthrough. You will be designing strategies for an AI agent that plays the game. Make sure that you or your group understand the rules of Breakthrough before you begin. In particular, note that the pieces do **not** behave exactly like pawns in Chess.

Throughout this lab, we will refer to Breakthrough boards using the notation (7x8,2). In order, those numbers are the number of rows, the number of columns, and the number of rows of pieces each player starts with. For example, this board is an (8x8,2) board.



You should read Chapter 5 before beginning the lab. Several portions of the lab rely directly on it.

Part 1) (~20 Points)

A) Play a round or two of this game (on paper, with coins or scraps of paper, or on a whiteboard) with your partner. There's nothing to submit for this part; this is just to get a feel for the game.

B) Devise a representation scheme for your environment. This will need to keep track of where all of the pieces are. There are several reasonable ways to go about this. You may wish to think about your transition function or other parts of the lab before settling on a choice here. Briefly describe your representation scheme in your **report**.

C) Write **(in code)** a function called *display_state* that takes a state as an argument and prints to the screen a pictorial representation of the state. This does not need to be fancy at all. For example, this is an acceptable representation of the starting state of the game above:

```
XXXXXXXX
XXXXXXXX
........
........
........
........
00000000
00000000
```

D) Create **(in code)** a function called *initial_state* that takes three arguments - a number of rows, a number of columns, and a number of rows of pieces. It should return a state where the board is the starting configuration for that board setup.

E) Create **(in code)** a transition function. Think about how you might want to model the actions in this game. If your transition function is an actual function, it might take an argument representing which player is taking the action. Remember that your transition function should not modify the state, but create a new one.

F) Implement **(in code)** a function that takes a board state and reports if it is a game-ending state. (That is, implement a *terminal test*.) A game-ending state is one where a player has reached the last row with one of their pieces or a state where all of a player's pieces have been eliminated.

G) Create **(in code)** a move-generator that, given a player and a board state, generates all possible actions that player could take.

H) Take the **Moodle quiz** called "Lab B Moodle Quiz." This should be done by group members individually, not as a group.

Part 2) (~40 Points)

A)

In this part, you will implement minimax search. The AI will control both players, using minimax search for each. You will have to experiment with what is the maximum amount of lookahead you are able to do with minimax search while still having the games play out in a reasonable amount of time. (This may vary with the size of the board, but 3 steps ahead is reasonable; you can probably do more if you're on a small enough board.)

In order to run minimax search, each agent will need a *utility function*. This is a function that takes a board state (and possibly other information) and returns a value indicating how desirable that board state is. For information on one way to implement a tree, see the end of this assignment.

Here is a utility function you can use:

Evasive: e(s) = number_of_own_pieces_remaining + random()

The "+ random()" is a simple way to break ties, as it is a value between 0 and 1. (You will need to import random.) If you are using +random() as a way to break ties, make sure that your heuristic function produces integer results otherwise.

You should design a function *play_game(heuristic_white, heuristic_black, board_state)*. The first argument is the heuristic that white will use. The second argument is the heuristic that black will use. The last is a board state that they will start from.

Have two AI play against each other, each using **Evasive** as their utility heuristic. If you run on a small board, like (5x5,1), the game will end in few enough moves that you should be able to follow along with an entire game by displaying the board after each move. In your **report**, describe briefly (in qualitative terms) how the Evasive AI seems to play. Is "evasive" a good name for it? Include in your report a few final board states, plus a list of how many pieces were captured by each player and the total number of moves taken before victory.

Note that if MAX is using a utility function, it assumes that the other agent is trying to minimize the value of that utility function. An agent doesn't know what utility function the other agent is using. **Update: This section was updated 2/17/20 to be clearer about the agents' assumptions about what the other player values.**

B)

Here is a second utility function you can use:

Conqueror: **(0 - number_of_opponent_pieces_remaining) + random()**

Have two AI play against each other, one using **Evasive** and one using **Conqueror**. Remember that a utility heuristic reflects the agent's beliefs about what constitutes a better board position; in other words, the Evasive AI assumes that its opponent is trying to minimize the value of Evasive, and the Conqueror AI assumes that its opponent is trying to minimize the value of Conqueror. This means that they will sometimes *not* correctly predict what their opponent will do. In your **report**, describe the behavior of the two AI. Is it noticeably different? Try running a few times on different board sizes, including (8x8,2). Which AI usually wins? Include in your report a few final board states, plus a list of how many pieces were captured by each player and the total number of moves taken before victory.

C) If you observe a few games involving Conqueror and Evasive, you can see that they're not particularly strong players. They don't even care about winning the game or about preventing a loss! A simple way to improve them would be to give them a strong preference for board states where they've won and a strong aversion to states where they've lost. However, even aside from that, it's easy to design a utility function that defeats them soundly.

Design at least two other utility functions. (If you design more, you can mention them in the report, but highlight two that you're particularly proud of.) Your functions must be good enough to at least reliably beat both Evasive and Conqueror on an (8x8,2) board. (It should beat both at least 80% of the time.) Your functions must have cool, descriptive nicknames, like mine. If your games take too long to run on an (8x8,2) board, report the results on the largest board you can where the games take less than ten minutes to run. (You can still report results from a larger board if you're able to leave it running for longer, but this is not required.)

In your **report**, describe your functions. Include final board states for matches between your functions and the ones I provided, and ones where your two functions face off against each other. Also include a list of how many pieces were captured by each player and the total number of moves taken before victory. Which function is the champion? Is it consistent across different board sizes? Do you think your champion function has any weaknesses?

In your **readme**, include instructions for how to run various matches using your two functions and the ones I provided.

This lab has been adapted from Svetlana Lazenbik's work.

Implementing a Tree

You can implement a basic tree using a set of connected Node data structures. This is not necessarily the most space- or time-efficient implementation, but it is relatively straightforward. You are not in any way required to use this implementation.

```
class Node(object):
    def __init__(self):
        self.parent = None
        self.child = []
        self.data = None
```

Each node keeps a pointer to its parent and a list of pointers to its children.

For this assignment, you may want each node to keep track of things like the action taken to reach that node, the state of the board after that action has been taken, and an agent's evaluation of the quality of that world, based on its utility function. Your Node objects might thus look like this, when initialized:

```
class Node(object):
    def __init__(self):
        self.parent = None
        self.child = []
        self.action = None
        self.state = None
        self.utility = None
```

And like this after being populated:

```
class Node(object):
    def __init__(self):
        self.parent = <Some Node>
        self.child = [<Some Node>, <Some Node>, ... <Some Node>]
        self.action = (“white”, (4,5), (5,5))
        self.state = (((2,4), (3,4), (5,5)), ((6,4), (7,4), (7,7)))
        self.utility = 13.44321
```

With the exact structure of those values matching your decisions about how to represent actions and states. (These are not necessarily good choices.) You may also not want to store all of this information, or may find it useful to store other information, such as the depth.